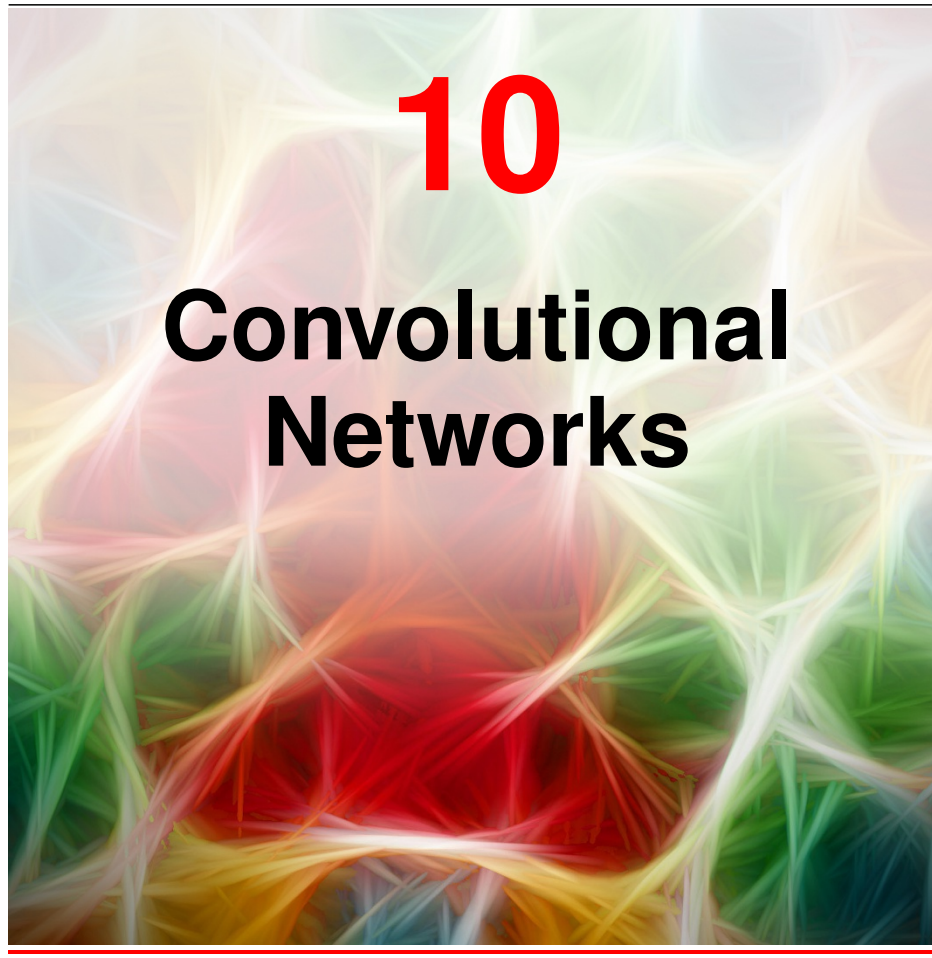


10 Convolutional Networks

The simplest machine learning models assume that the observed data values are unstructured, meaning that the elements of the data vectors $\mathbf{x} = (x_1, \ldots, x_D)$ are treated as if we do not know anything in advance about how the individual elements might relate to each other. If we were to make a random permutation of the ordering of these variables and apply this fixed permutation consistently on all training and test data, there would be no difference in the performance for the models considered so far.

Many applications of machine learning, however, involve *structured data* in which there are additional relationships between input variables. For example, the words in natural language form a *sequence*, and if we were to model language as a generative autoregressive process then we would expect each word to depend more strongly on the immediately preceding words and less so on words much earlier in the sequence. Likewise, the pixels of an image have a well-defined spatial relation-

Chapter 12

C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_10

ship to each other in which the input variables are arranged in a two-dimensional grid, and nearby pixels have highly correlated values.

We have already seen that our knowledge of the structure of specific data modalities can be utilized through the addition of a regularization term to the error function in the training objective, through data augmentation, or through modifications to the model architecture. These approaches can help guide the model to respect certain properties such as invariance and equivariance with respect to transformations of the input data. In this chapter we will take a look at an architectural approach called a *convolutional neural network* (CNN), which we will see can be viewed as a sparsely connected multilayer network with parameter sharing, and designed to encode invariances and equivariances specific to image data.

Section 9.1

Section 9.1.4

10.1. Computer Vision

The automatic analysis and interpretation of image data form the focus of the field of computer vision and represent a major application area for machine learning (Szeliski, 2022). Historically, computer vision was based largely on 3-dimensional projective geometry. Hand-crafted features were constructed and used as input to simple learning algorithms (Hartley and Zisserman, 2004). However, it was one of the first fields to be transformed by the deep learning revolution, predominantly thanks to the CNN architecture. Although the architecture was originally developed in the context of image analysis, it has also been applied in other domains such as the analysis of sequential data. Recently alternative architectures based on transformers have become competitive with convolutional networks in some applications.

Chapter 12

There are many applications for machine learning in computer vision, of which some of the most commonly encountered are the following:

Figure 1.1

1. **Classification** of images, for example classifying an image of a skin lesion as benign or malignant. This is sometimes called 'image recognition'.

Figure 10.19

2. **Detection** of objects in an image and determining their locations within the image, for example detecting pedestrians from camera data collected by an autonomous vehicle.

Figure 10.26

3. **Segmentation** of images, in which each pixel is classified individually thereby dividing the image into regions sharing a common label. For example, a natural scene might be segmented into sky, grass, trees, and buildings, whereas a medical scan image could be segmented into cancerous tissue and normal tissue.

Figure 12.27

4. **Caption generation** in which a textual description is generated automatically from an image.

Figure 1.3

5. **Synthesis** of new images, for example generating images of human faces. Images can also be synthesized based on a text input describing the desired image content.

Figure 20.9

6. **Inpainting** in which a region of an image is replaced with synthesized pixels that are consistent with the rest of the image. This is used, for example, to remove unwanted objects during image editing.

Figure 10.32

7. **Style transfer** in which an input image in one style, for example a photograph, is transformed into a corresponding image in a different style, for example an oil painting.

Figure 20.8

8. **Super-resolution** in which the resolution of an image is improved by increasing the number of pixels and synthesizing associated high-frequency information.

9. **Depth prediction** in which one or more views are used to predict the distance of the scene from the camera at each pixel in a target image.

10. **Scene reconstruction** in which one or more two-dimensional images of a scene are used to reconstruct a three-dimensional representation.

10.1.1 Image data

An image comprises a rectangular array of pixels, in which each pixel has either a grey-scale intensity or more commonly a triplet of red, green, and blue *channels* each with its own intensity value. These intensities are non-negative numbers that also have some maximum value corresponding to the limits of the camera or other hardware device used to capture the image. For the most part, we will view the intensities as continuous variables, but in practice they are represented with finite precision, for example as 8-bit numbers represented as integers in the range $0, \ldots, 255$. Some images, such as the magnetic resonance imaging (MRI) scans used in medical diagnosis, comprise three-dimensional grids of *voxels*. Similarly, videos comprise a sequence of two-dimensional images and therefore can also be viewed as three-dimensional structures in which successive frames are stacked through time.

Now consider the challenge of applying neural networks to image data to address some of the applications highlighted above. Images generally have a high dimensionality, with typical cameras capturing images comprising tens of megapixels. Treating the image data as unstructured may therefore require a model with a vast number of parameters that would be infeasible to train. More significantly, such an approach fails to take account of the highly structured nature of image data, in which the relative positions of different pixels play a crucial role. We can see this because if we take the pixels of an image and randomly permute them, then the result no longer looks like a natural image. Similarly, if we generate a synthetic image by drawing random values for the pixel intensities independently for each pixel, there

Figure 6.8

is essentially zero chance of generating something that looks like a natural image. Local correlations are important, and in a natural image there is a much higher probability that two nearby pixels will have similar colours and intensities compared to two pixels that are far apart. This represents powerful prior knowledge and can be used to encode strong inductive biases into a neural network, leading to models with far fewer parameters and with much better generalization accuracy.

10.2. Convolutional Filters

One motivation for the introduction of convolutional networks is that for image data, which is the modality for which CNNs were designed, a standard fully connected architecture would require vast numbers of parameters due to the high-dimensional nature of images. To see this, consider a colour image with $10^3 \times 10^3$ pixels, each with three values corresponding to red, green, and blue intensities. If the first hidden layer of the network has, say, 1,000 hidden units, then we already have 3×10^9 weights in the first layer. Furthermore, such a network would have to learn any invariances and equivariances by example, which would require huge data sets. By designing an architecture that incorporates our inductive bias about the structure of images, we can reduce the data set requirements dramatically and also improve generalization with respect to symmetries in the image space.

To exploit the two-dimensional structure of image data to create inductive biases, we can use four interrelated concepts: hierarchy, locality, equivariance, and invariance. Consider the task of detecting faces in images. There is a natural hierarchical structure because one image may contain several faces, and each face includes elements such as eyes, and each eye has structure such as an iris, which itself has structure such as edges. At the lowest level of the hierarchy, a node in a neural network could detect the presence of a feature such as an edge using information that is *local* to a small region of an image, and therefore it would only need to see a small subset of the image pixels. More complex structures further up the hierarchy can be detected by composing multiple features found at previous levels. A key point, however, is that although we want to build the general concept of hierarchy into the model, we want the details of the hierarchy, including the type of features extracted at each level, to be learned from data, not hand-coded. Hierarchical models fit naturally within the deep learning framework, which already allows very complex concepts to be extracted from raw data through a succession of, possibly very many, 'layers' of processing, in which the whole system is trained end-to-end.

10.2.1 Feature detectors

For simplicity we will initially restrict our attention to grey-scale images (i.e., ones having a single channel). Consider a single unit in the first layer of a neural network that takes as input just the pixel values from a small rectangular region, or patch, from the image, as illustrated in Figure 10.1(a). This patch is referred to as the *receptive field* of that unit, and it captures the notion of locality. We would like weight values associated with this unit to learn to detect some useful low-level feature. The output of this unit is given by the usual functional form comprising a weighted linear combination of the input values, which is subsequently transformed using a nonlinear activation function:

$$z = \mathrm{ReLU}(\mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0) \tag{10.1}$$

where $\mathbf{x}$ is a vector of pixel values for the receptive field, and we have assumed a ReLU activation function. Because there is one weight associated with each input

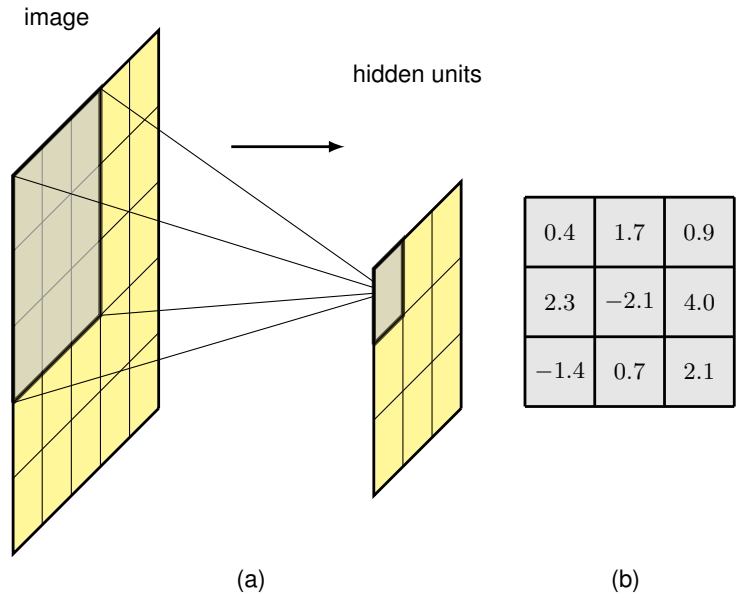


Figure 10.1 (a) Illustration of a receptive field, showing a unit in a hidden layer of a network that receives input from pixels in a 3×3 patch of the image. Pixels in this patch form the receptive field for this unit. (b) The weight values associated with this hidden unit can be visualized as a small 3×3 matrix, known as a kernel. There is also an additional bias parameter that is not shown here.

pixel, the weights themselves form a small two-dimensional grid known as a *filter*, sometimes also called a *kernel*, which itself can be visualized as an image. This is illustrated in Figure 10.1(b).

Suppose that $\mathbf{w}$ and w_0 in (10.1) are fixed and we ask for which value of the input image patch $\mathbf{x}$ will this hidden unit give the largest output response. To answer this we need to constrain $\mathbf{x}$ in some way, so let us suppose that its norm $\|\mathbf{x}\|^2$ is fixed. Then the solution for $\mathbf{x}$ that maximizes $\mathbf{w}^{\mathrm{T}}\mathbf{x}$, and hence maximizes the response of the hidden unit, is of the form $\mathbf{x} = \alpha\mathbf{w}$ for some coefficient α. This says that the maximum output response from this hidden unit occurs when it detects a patch of image that, up to an overall scaling, looks like the kernel image. Note that the ReLU generates a non-zero output only when $\mathbf{w}^{\mathrm{T}}\mathbf{x}$ exceeds a threshold of $-w_0$, and therefore the unit acts as a feature detector that signals when it finds a sufficiently good match to its kernel.

Exercise 10.1

10.2.2 Translation equivariance

Next note that if a small patch in a face image represents, for example, an eye at that location, then the same set of pixel values in a different part of the image must represent an eye at the new location. Our neural network needs to be able to generalize what it has learned in one location to all possible locations in the image, without needing to see examples in the training set of the corresponding feature at every possible location. To achieve this, we can simply replicate the *same* hidden-unit weight values at multiple locations across the image, as illustrated for a one-dimensional input space in Figure 10.2.

The units of the hidden layer form a *feature map* in which all the units share the same weights. Consequently if a local patch of an image produces a particular

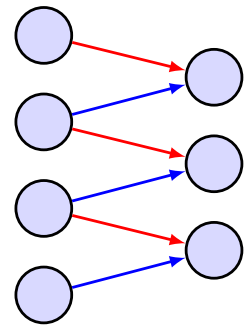


Figure 10.2 Illustration of convolution for a one-dimensional array of input values and a kernel of width 2. The connections are sparse and are shared by the hidden units, as shown by the red and blue arrows in which links with the same colour have the same weight values. This network therefore has six connections but only two independent learnable parameters.

response in the unit connected to that patch, then the same set of pixel values at a different location will produce the same response in the corresponding translated location in the feature map. This is an example of *equivariance*. We see that the connections in this network are *sparse* in that most connections are absent. Also, the values of the weights are *shared* by all the hidden units, as indicated by the colours of the connections. This transformation is an example of a *convolution*.

Section 9.1.3

We can extend the idea of convolution to two-dimensional images as follows (Dumoulin and Visin, 2016). For an image $\mathbf{I}$ with pixel intensities $I(j,k)$, and a filter $\mathbf{K}$ with pixel values $K(l,m)$, the feature map $\mathbf{C}$ has activation values given by

$$C(j,k) = \sum_l \sum_m I(j+l, k+m) K(l,m) \tag{10.2}$$

where we have omitted the nonlinear activation function for clarity. This again is an example of a *convolution* and is sometimes expressed as $\mathbf{C} = \mathbf{I} * \mathbf{K}$. Note that strictly speaking (10.2) is called a *cross-correlation*, which differs slightly from the conventional mathematical definition of convolution, but here we will follow common practice in the machine learning literature and refer to (10.2) as a convolution. The relationship (10.2) is illustrated in Figure 10.3 for a 3×3 image and a 2×2 filter. Importantly, when using batch normalization in a convolutional network, the same value of mean and variance must be used at every spatial location within a feature map when normalizing the states of the units to ensure that the statistics of the feature map are independent of location.

Exercise 10.4

Section 7.4.2

As an example of the application of convolution, we consider the problem of detecting edges in images using a fixed, hand-crafted convolutional filter. Intuitively, we can think of a vertical edge as occurring when there is a significant local change in the intensity between pixels as we move horizontally across the image. We can measure this by convolving the image with a 3×3 filter of the form

$$\begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \tag{10.3}$$

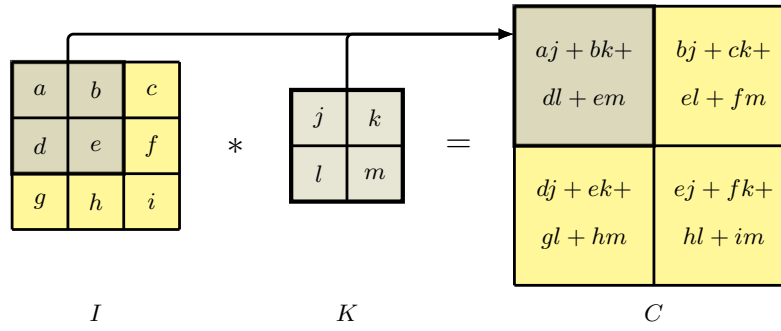


Figure 10.3 Example of a 3×3 image convolved with a 2×2 filter to give a resulting 2×2 feature map.

Similarly we can detect horizontal edges by convolving with the transpose of this filter:

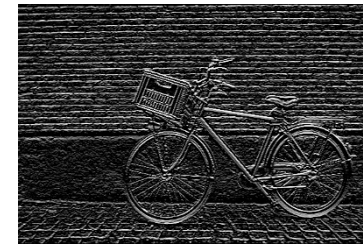
$$\begin{array}{|c|c|c|}\hline -1 & -1 & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 1 & 1 \\ \hline \end{array} \tag{10.4}$$

Figure 10.4 shows the results of applying these two convolutional filters to a sample image. Note that in Figure 10.4(b) if a vertical edge corresponds to an increase in pixel intensity, the corresponding point on the feature map is positive (indicated by a light colour), whereas if the vertical edge corresponds to a decrease in pixel intensity, the corresponding point on the feature map is negative (indicated by a dark colour), with analogous properties for Figure 10.4(c) for horizontal edges.

Comparing this convolutional structure with a standard fully connected network, we see several advantages: (i) the connections are sparse, leading to far fewer weights even with large images, (ii) the weight values are shared, greatly reducing the number of independent parameters and consequently reducing the required size

(a)

(b)



(c)

Figure 10.4 Illustration of edge detection using convolutional filters showing (a) the original image, (b) the result of convolving with the filter (10.3) that detects vertical edges, and (c) the result of convolving with the filter (10.4) that detects horizontal edges.

of the training set needed to learn those parameters, and (iii) the same network can be applied to images of different sizes without the need for retraining. We will return to this final point later, but for the moment, simply note that changing the size of the input image simply changes the size of the feature map but does not change the number of weights, or the number of independent learnable parameters, in the model. One final observation regarding convolutional networks is that, by exploiting the massive parallelism of graphics processing units (GPUs) to achieve high computational throughput, convolutions can be implemented very efficiently.

Section 10.4.3

10.2.3 Padding

We see from Figure 10.3 that the convolution map is smaller than the original image. If the image has dimensionality $J \times K$ pixels and we convolve with a kernel of dimensionality $M \times M$ (filters are typically chosen to be square) the resulting feature map has dimensionality $(J - M + 1) \times (K - M + 1)$. In some cases we want the feature map to have the same dimensions as the original image. This can be achieved by *padding* the original image with additional pixels around the outside, as illustrated in Figure 10.5. If we pad with P pixels then the output map has dimensionality $(J+2P-M+1)\times(K+2P-M+1)$. If there is no padding, so that $P = 0$, this is called a *valid* convolution. When the value of P is chosen such that the output array has the same size as the input, corresponding to $P = (M - 1)/2$, this is called a *same* convolution, because the image and the feature map have the same dimensions. In computer vision, filters generally use odd values of M, so that the padding can be symmetric on all sides of the image and that there is a well-defined central pixel associated with the location of the filter. Finally, we have to choose a suitable value for the intensities associated with the padding pixels. A typical choice is to set the padding values to zero, after first subtracting the mean from each image so that zero represents the average value of the pixel intensity. Padding can also be applied to feature maps for processing by subsequent convolutional layers.

Exercise 10.6

10.2.4 Strided convolutions

In typical image processing applications, the images can have very large numbers of pixels, and since the kernels are often relatively small, so that $M \ll J, K$, the convolutional feature map will be of a similar size to the original image and will be the same size if *same* padding is used. Sometimes we wish to use feature maps that are significantly smaller than the original image to provide flexibility in the design of convolutional network architectures. One way to achieve this is to use *strided convolutions* in which, instead of stepping the filter over the image one pixel at a time, it is moved in larger steps of size S, called the *stride*. If we use the same stride horizontally and vertically, then the number of elements in the feature map will be

Exercise 10.7

$$\left\lfloor \frac{J + 2P - M}{S} - 1 \right\rfloor \times \left\lfloor \frac{K + 2P - M}{S} - 1 \right\rfloor \tag{10.5}$$

where $\lfloor x \rfloor$ denotes the 'floor' of x, i.e., the largest integer that is less than or equal to x. For large images and small filter sizes, the image map will be roughly a factor of $1/S$ smaller than the original image.

Figure 10.5 Illustration of a 4×4 image that has been padded with additional pixels to create a 6×6 image.

0	0	0	0	0	0
0	X_{11}	X_{12}	X_{13}	X_{14}	0
0	X_{21}	X_{22}	X_{23}	X_{24}	0
0	X_{31}	X_{32}	X_{33}	X_{34}	0
0	X_{41}	X_{42}	X_{43}	X_{44}	0
0	0	0	0	0	0

10.2.5 Multi-dimensional convolutions

So far we have considered convolutions over a single grey-scale image. For a colour image there will be three channels corresponding to the red, green, and blue colours. We can easily extend convolutions to cover multiple channels by extending the dimensionality of the filter. An image with $J \times K$ pixels and C channels will be described by a *tensor* of dimensionality $J \times K \times C$. We can introduce a filter described by a tensor of dimensionality $M \times M \times C$ comprising a separate $M \times M$ filter for each of the C channels. Assuming no padding and a stride of 1, this again gives a feature map of size $(J-M+1) \times (K-M+1)$, as is illustrated in Figure 10.6.

Section 6.3.7

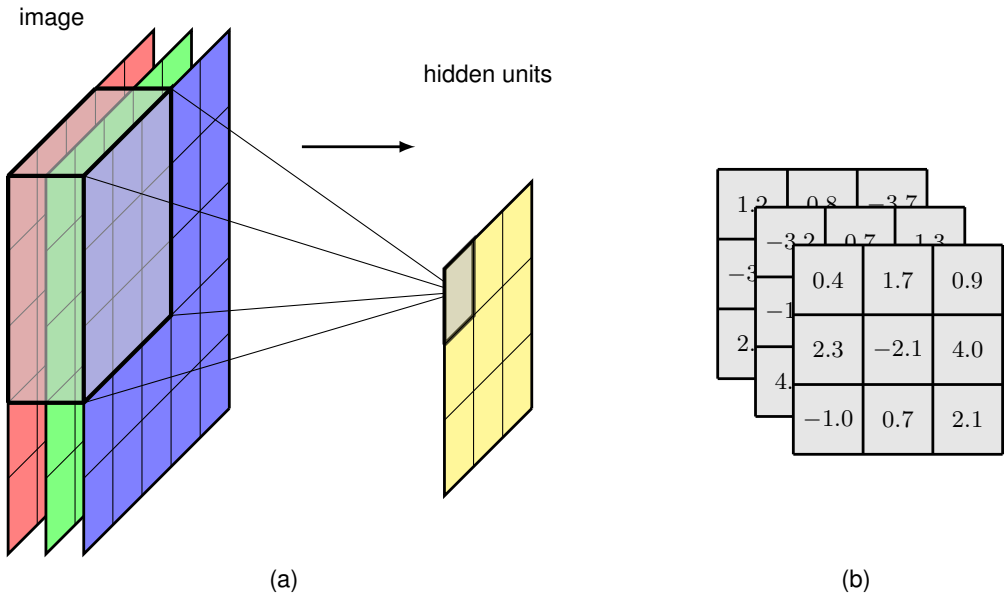


Figure 10.6 (a) Illustration of a multi-dimensional filter that takes input from across the R, G, and B channels. (b) The kernel here has 27 weights (plus a bias parameter not shown) and can be visualized as a $3 \times 3 \times 3$ tensor.

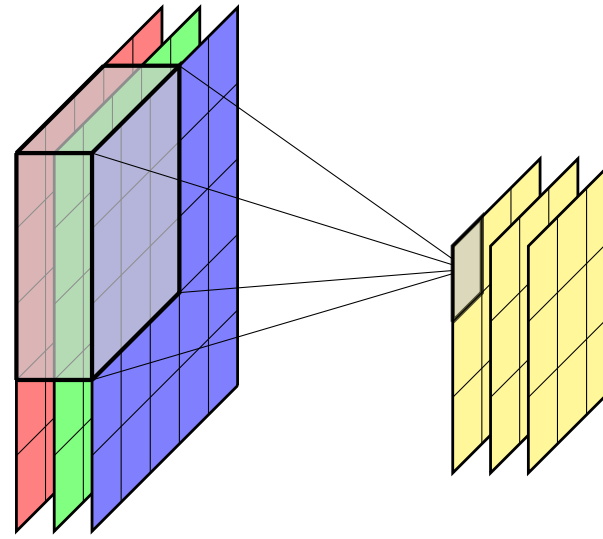


Figure 10.7 The multi-dimensional convolutional filter layer shown in Figure 10.6 can be extended to include multiple independent filter channels.

We now make a further important extension to convolutions. Up to now we have created a single feature map in which all the points in the feature map share the same set of learnable parameters. For a filter of dimensionality $M \times M \times C$, this will have M^2C weight parameters, irrespective of the size of the image. In addition there will be a bias parameter associated with this unit. Such a filter is analogous to a single hidden node in a fully connected network, and it can learn to detect only one kind of feature and is therefore very limited. To build more flexible models, we simply include multiple such filters, in which each filter has its own independent set of parameters giving rise to its own independent feature map, as illustrated in Figure 10.7. We will again refer to these separate feature maps as *channels*. The filter tensor now has dimensionality $M \times M \times C \times C_{\text{OUT}}$ where C is the number of input channels and C_{OUT} is the number of output channels. Each output channel will have its own associated bias parameter, so the total number of parameters will be $(M^2C + 1)C_{\text{OUT}}$.

A useful concept in designing convolutional networks is the 1×1 *convolution* (Lin, Chen, and Yan, 2013), which is simply a convolutional layer in which the filter size is a single pixel. The filters have C weights, one for each input channel, plus a bias. One application for 1×1 convolutions is simply to change the number of channels (typically to reduce the number of channels) without changing the size of the feature maps, by setting the number of output channels to be different to the number of input channels. It is therefore complementary to strided convolutions or pooling in that it reduces the number of channels rather than the dimensionality of the channels.

10.2.6 Pooling

A convolutional layer encodes translation *equivariance*, whereby if a small patch of pixels, representing the receptive field of a hidden unit, is moved to a different

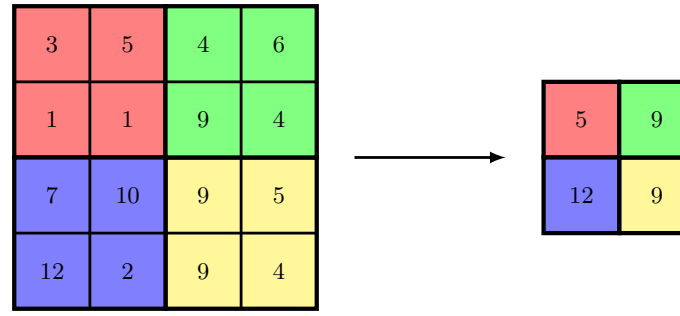


Figure 10.8 Illustration of max-pooling in which blocks of 2×2 pixels in a feature map are combined using the 'max' operator to generate a new feature map of smaller dimensionality.

location in the image, the associated outputs of the feature map will move to the corresponding location in the feature map. This is valuable for applications such as finding the location of an object within an image. For other applications, such as classifying an image, we want the output to be *invariant* to translations of the input. In all cases, however, we want the network to be able to learn hierarchical structure in which complex features at a particular level are built up from simpler features at the previous level. In many cases the spatial relationship between those simpler features will be important. For example, it is the relative positions of the eyes, nose, and mouth that help determine the presence of a face and not just the presence of these features in arbitrary locations within the image. However, small changes in the relative locations do not affect the classification, and we want to be invariant to such small translations of individual features. This can be achieved using *pooling* applied to the output of the convolutional layer.

Pooling has similarities to using a convolutional layer in that an array of units is arranged in a grid, with each unit taking input from a receptive field in the previous feature map layer. Again, there is a choice of filter size and of stride length. The difference, however, is that the output of a pooling unit is a simple, fixed function of its inputs, and so there are no learnable parameters in pooling. A common example of a pooling function is *max-pooling* (Zhou and Chellappa, 1988) in which each unit simply outputs the $\max$ function applied to the input values. This is illustrated with a simple example in Figure 10.8. Here the stride length is equal to the filter width, and so there is no overlap of the receptive fields.

As well as building in some local translation invariance, pooling can also be used to reduce the dimensionality of the representation by down-sampling the feature map. Note that using strides greater than 1 in a convolutional layer also has the effect of down-sampling the feature maps.

We can interpret the activation of a unit in a feature map as a measure of the strength of detection of a corresponding feature, so that the max-pooling preserves information on whether the feature is present and with what strength but discards some positional information. There are many other choices of pooling function, for example *average pooling* in which the pooling function computes the average of the values from the corresponding receptive field in the feature map. These all introduce some degree of local translation invariance.

Pooling is usually applied to each channel of a feature map independently. For

example, if we have a feature map with 8 channels, each of dimensionality 64×64, and we apply max-pooling with a receptive field of size 2×2 and a stride of 2, the output of the pooling operation will be a tensor of dimensionality $32 \times 32 \times 8$.

We can also apply pooling across multiple channels of a feature map, which gives the network the potential to *learn* other invariances beyond simple translation invariance. For example, if several channels in a convolutional layer learn to detect the same feature but at different orientations, then max-pooling across those feature maps will be approximately invariant to rotations.

Pooling also allows a convolutional network to process images of varying sizes. Ultimately, the output, and generally some of the intermediate layers, of a convolutional network must have a fixed size. Variable-sized inputs can be accommodated by varying the stride length of the pooling according to the size of the image such that the number of pooled outputs remains constant.

10.2.7 Multilayer convolutions

The convolutional network structure described so far is analogous to a single layer in a standard fully connected neural network. To allow the network to discover and represent hierarchical structure in the data, we now extend the architecture by considering multiple layers of the kind described above. Each convolutional layer is described by a filter tensor of dimensionality $M \times M \times C_{\text{IN}} \times C_{\text{OUT}}$ in which the number of independent weight and bias parameters is $(M^2 C_{\text{IN}} + 1)C_{\text{OUT}}$. Each such convolutional layer can optionally be followed by a pooling layer. We can now apply multiple such layers of convolution and pooling in succession, in which the C_{OUT} output channels of a particular layer, analogous to the RGB channels of the input image, form the input channels of the next layer. Note that the number of channels in a feature map is sometimes called the 'depth' of the feature map, but we prefer to reserve the term depth to mean the number of layers in a multilayer network.

A key property that we built into the convolutional framework is that of locality, in which a given unit in a feature map takes information only from a small patch, the *receptive field*, in the previous layer. When we construct a deep neural network in which each layer is convolutional then the effective receptive field of a unit in later layers in the network becomes much larger than those in earlier layers, as seen in Figure 10.9.

In many applications, the output units of the network need to make predictions about the image as a whole, for example in a classification task, and so they need to combine information from across the whole of the input image. This is typically achieved by introducing one or two standard fully connected layers as the final stages of the network, in which each unit is connected to every unit in the previous layer. The number of parameters in such an architecture can be manageable because the final convolutional layer generally has much lower dimensionality than the input layer due to the intermediate pooling layers. Nevertheless, the final fully connected layers may contain the majority of the independent degrees of freedom in the network even if the number of (shared) connections in the network is larger in the convolutional layers.

A complete CNN therefore comprises multiple layers of convolutions inter-

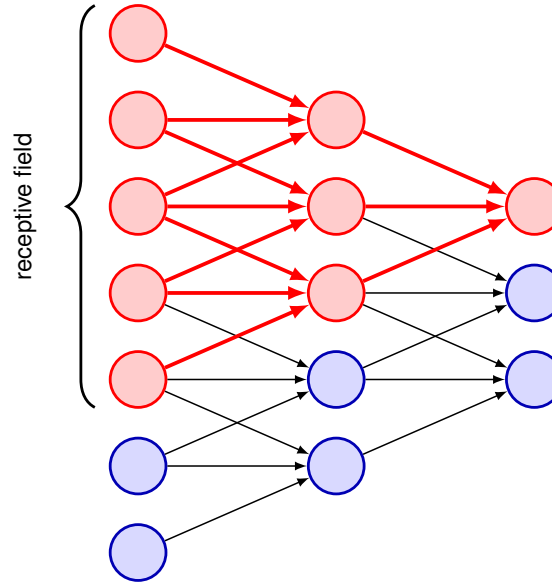


Figure 10.9 Illustration of how the effective receptive field grows with depth in a multilayer convolutional network. Here we see that the red unit at the top of the output layer takes inputs from a receptive field in the middle layer of units, each of which has a receptive field in the first layer of units. Thus, the activation of the red unit in the output layer depends on the outputs of 3 units in the middle layer and 5 units in the input layer.

spersed with pooling operations, and often with conventional fully connected layers in the final stages of the network. There are many choices to be made in designing such an architecture including the number of layers, the number of channels in each layer, the filter sizes, the stride widths, and multiple other such hyperparameters. A wide variety of different architectures have been explored, although in practice it is difficult to make a systematic comparison of hyperparameter values using hold-out data due to the high computational cost of training each candidate configuration.

10.2.8 Example network architectures

Convolutional networks were the first deep neural networks (i.e., ones with more than two learnable layers of parameters) to be successfully deployed in applications. An early example was *LeNet*, which was used to classify low-resolution monochrome images of handwritten digits (LeCun *et al.*, 1989; LeCun *et al.*, 1998). The development of more powerful convolutional networks was accelerated through the introduction of a large-scale benchmark data set called *ImageNet* (Deng *et al.*, 2009) comprising some 14 million natural images each of which has been hand labelled into one of nearly 22,000 categories. This was a much larger data set than had been used previously, and the advances in the field driven by ImageNet served to emphasize the importance of large-scale data, alongside well-designed models having appropriate inductive biases, in building successful deep learning solutions.

A subset of images comprising 1,000 non-overlapping categories formed the basis for the annual *ImageNet Large Scale Visual Recognition Challenge*. Again, this was a much larger number of categories than the typically few dozen classes previously considered. Having so many categories made the problem much more challenging because, if the classes were distributed uniformly, random guessing would

have an error rate of 99.9%. The data set has just over 1.28 million training images, 50,000 validation images, and 100,000 test images. The classifiers are designed to produce a ranked list of predicted output classes on test images, and results are reported in terms of top-1 and top-5 error rates, meaning an image is deemed to be correctly classified if the true class appears at the top of the list or if it is in one of the five highest-ranked class predictions. Early results with this data set achieved a top-5 error rate of around 25.5%. An important advance was made by the *AlexNet* convolutional network architecture (Krizhevsky, Sutskever, and Hinton, 2012), which won the 2012 competition and reduced the top-5 error rate to a new record of 15.3%. Key aspects of this model were the use of the ReLU activation function, the application of GPUs to train the network, and the use of dropout regularization. Subsequent years saw further advances, leading to error rates of around 3%, which is somewhat better than human-level performance for the same data, which is around 5% (Dodge and Karam, 2017). This can be attributed to the difficulty humans have in distinguishing subtly different classes (for example multiple varieties of mushrooms).

Section 9.6.1

As an example of a typical convolutional network architecture, we look in detail at the VGG-16 model (Simonyan and Zisserman, 2014), where VGG stands for the Visual Geometry Group, who developed the model, and 16 refers to the number of learnable layers in the model. VGG-16 has some simple design principles leading to a relatively uniform architecture, shown in Figure 10.10, that minimizes the number of hyperparameter choices that need to be made. It takes an input image having 224×224 pixels and three colour channels, followed by sets of convolutional layers interspersed with down-sampling. All convolutional layers have filters of size 3×3 with a stride of 1, same padding, and a ReLU activation function, whereas the max-pooling operations all use stride 2 and filter size 2×2 thereby down-sampling the number of units by a factor of 4. The first learnable layer is a convolutional layer in which each unit takes input from a $3 \times 3 \times 3$ 'cube' from the stack of input channels and so has 28 parameters including the bias. These parameters are shared across all units in the feature map for that channel. There are 64 such feature channels in the first layer, giving an output tensor of size $224 \times 224 \times 64$. The second layer is also convolutional and again has 64 channels. This is followed by max-pooling giving feature maps of size 112×112. Layers 3 and 4 are again convolutional, of dimensionality 112×112, and each was chosen to have 128 channels. This increase in the number of channels offsets to some extent the down-sampling in the max-pooling layer to ensure that the number of variables in the representation at each layer does not decrease too rapidly through the network. Again, this is followed by a max-pooling operation to give a feature map size of 56×56. Next come three more convolutional layers each with 256 channels, thereby again doubling the number of channels in association with the down-sampling. This is followed by another max-pooling to give feature maps of size 28×28 followed by three more convolutional layers each having 512 channels, followed by another max-pooling, which down-samples to feature maps of size 14×14. This is followed by three more convolutional layers, although the number of feature maps in these layers is kept at 512, followed by another max-pooling, which brings the size of the feature maps down to 7×7. Finally there are three more layers that are *fully connected* meaning that they are

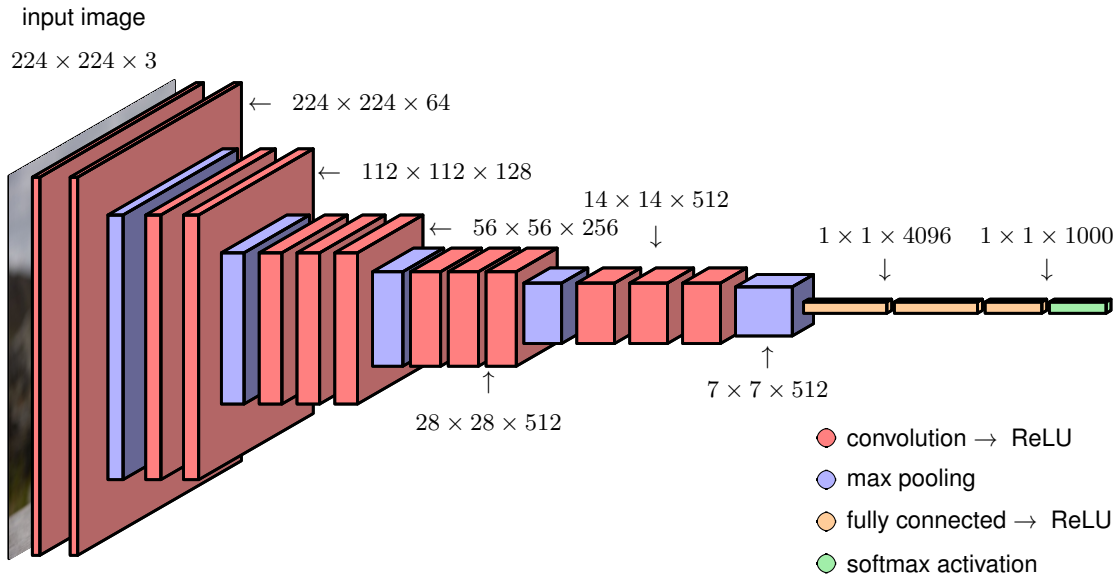


Figure 10.10 The architecture of a typical convolutional network, in this case a model called VGG-16.

standard neural network layers with full connectivity and no sharing of parameters. The final max-pooling layer has 512 channels each of size 7×7 giving 25,088 units in total. The first fully connected layer has 4,096 units, each of which is connected to each of the max-pooling units. This is followed by a second fully connected layer again with 4,096 units, and finally there is a third fully connected layer with 1,000 units so that the network can be applied to a classification problem involving 1,000 classes. All the learnable layers in the network have nonlinear ReLU activation functions, except for the output layer, which has a softmax activation function. In total there are roughly 138 million independently learnable parameters in VGG-16, the majority of which (nearly 103 million) are in the first fully connected layer, whereas most of the connections are in the first convolutional layer.

Exercise 10.8

Earlier CNNs typically had fewer convolutional layers, as they had larger receptive fields. For example, Alexnet (Krizhevsky, Sutskever, and Hinton, 2012) has 11×11 receptive fields with a stride of 4. We saw in Figure 10.9 that larger receptive fields can also be achieved implicitly by using multiple layers each having smaller receptive fields. The advantage of the latter approach is that it requires significantly fewer parameters, effectively imposing an inductive bias on the larger filters as they must be composed of convolutional sub-filters. Although this is a highly complex architecture, only the network function itself needs to be coded explicitly since the derivatives of the cost function can be evaluated using automatic differentiation and the cost function optimized using stochastic gradient descent.

Section 8.2

10.3. Visualizing Trained CNNs

We turn now to an exploration of the features learned by modern deep CNNs, and we will see some remarkable similarities to the properties of the mammalian visual cortex.

10.3.1 Visual cortex

Historically, much of the motivation for CNNs came from pioneering research in neuroscience, which gave insights into the nature of visual processing in mammals including humans. Electrical signals from the retina are transformed through a series of processing layers in the visual cortex, which is at the back of the brain, where the neurons are organized into two-dimensional sheets each of which forms a map of the two-dimensional visual field. In their pioneering work, Hubel and Wiesel (1959) measured the electrical responses of individual neurons in the visual cortex of cats while presenting visual stimuli to the cats' eyes. They discovered that some neurons, called 'simple cells', have a strong response to visual inputs with a simple edge oriented at a particular angle and located at a particular position within the visual field, whereas other stimuli generated relatively little response in those neurons. More detailed studies showed that the response of these simple cells can be modelled using *Gabor filters*, which are two-dimensional functions defined by

$$G(x, y) = A \exp\left(-\alpha\widetilde{x}^2 - \beta\widetilde{y}^2\right) \sin\left(\omega\widetilde{x} + \phi\right) \tag{10.6}$$

where

$$\widetilde{x} = (x - x_0)\cos(\theta) + (y - y_0)\sin(\theta) \tag{10.7}$$
$$\widetilde{y} = -(x - x_0)\sin(\theta) + (y - y_0)\cos(\theta). \tag{10.8}$$

Equations (10.7) and (10.8) represent a rotation of the coordinate system through an angle θ and therefore the $\sin(\cdot)$ term in (10.6) represents a sinusoidal spatial oscillation oriented in a direction defined by the polar angle θ, with frequency ω and phase angle ϕ . The exponential factor in (10.6) creates a decay envelope that localizes the filter in the neighbourhood of position (x_0, y_0) and with decay rates governed by α and β. Example Gabor filters are shown in Figure 10.11.

Hubel and Wiesel also discovered the presence of 'complex cells', which respond to more complex stimuli and which seem to be derived by combining and processing the output of simple cells. These responses exhibit some degree of invariance to small changes in the input such as shifts in location, analogous to the pooling units in a convolutional deep network. Deeper levels of the mammalian visual processing system have even more specific responses and even greater invariance to transformations of the visual input. Such cells have been termed 'grandmother cells' because such a cell could notionally respond if, and only if, the visual input corresponds to a person's grandmother, irrespective of location, scale, lighting, or other transformations of the scene. This work directly inspired an early form of deep neural network called the *neocognitron* (Fukushima, 1980), which was the forerunner of

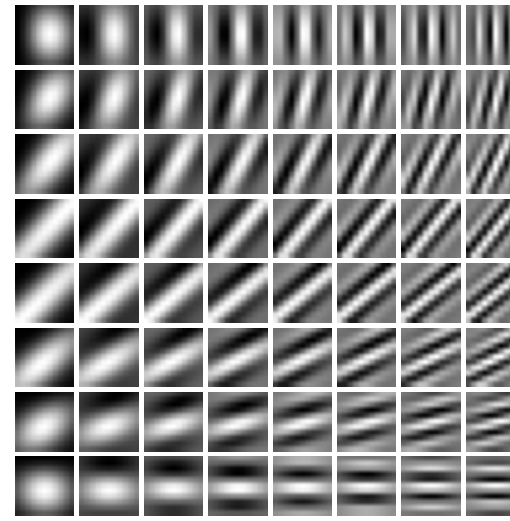


Figure 10.11 Examples of Gabor filters defined by (10.6). The orientation angle θ varies from 0 in the top row to $\pi/2$ in the bottom row, whereas the frequency varies from $\omega = 1$ in the left column to $\omega = 10$ in the right column.

convolutional neural networks. The neocognitron had multiple layers of processing comprising local receptive fields with shared weights followed by local averaging or max-pooling to confer positional invariance. However, it lacked an end-to-end training procedure since it predated the development of backpropagation, relying instead on greedy layer-wise learning through an unsupervised clustering algorithm.

10.3.2 Visualizing trained filters

Suppose we have a trained deep CNN and we wish to explore what the hidden units have learned to detect. For the filters in the first convolutional layer this is relatively straightforward, as they correspond to small patches in the original input image space, and so we can visualize the network weights associated with these filters directly as small images. The first convolutional layer computes inner products between the filters and the corresponding image patches, and so the unit will have a large activation when the inner product has a large magnitude.

Figure 10.12 shows some example filters from the first layer of a CNN trained on the ImageNet data set. We see a remarkable similarity between these filters and the Gabor filters of Figure 10.11. However, this does not imply that a convolutional neural network is a good model of how the brain works, because very similar results can be obtained from a wide variety of statistical methods (Hyvärinen, Hurri, and Hoyer, 2009). This is because these characteristic filters are a general property of the statistics of natural images and therefore prove useful for image understanding in both natural and artificial systems.

Although we can visualize the filters in the first layer directly, the subsequent layers in the network are harder to interpret because their inputs are not patches of images but groups of filter responses. One approach, analogous to that used by Hubel and Wiesel, is to present a large number of image patches to the network and

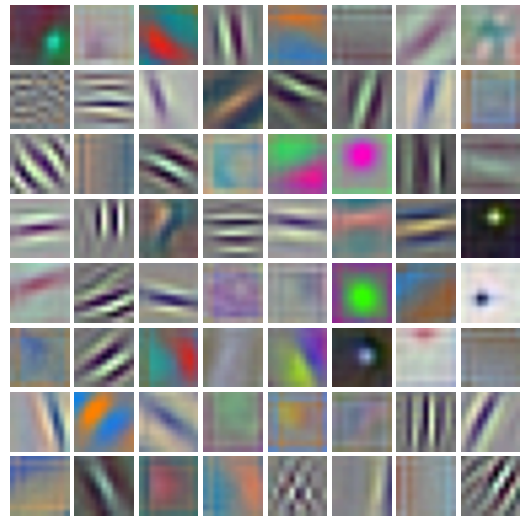


Figure 10.12 Examples of learned filters from the first layer of AlexNet. Note the remarkable similarity of many of the learned filters to the Gabor filters in Figure 10.11, which correspond to features detected by living neurons in the visual cortex of mammals.

see which produce the highest activation value in any particular hidden unit. Figure 10.13 shows examples obtained using a network with five convolutional layers, followed by two fully connected layers, trained on 1.3 million ImageNet data points spanning 1,000 classes. We see a natural hierarchical structure, with the first layer responding to edges, the second layer responding to textures and simple shapes, layer 3 showing components of objects (such as wheels), and layer 5 showing entire objects.

We can extend this technique to go beyond simply selecting image patches from the validation set and instead perform a numerical optimization over the input variables to maximize the activation of a particular unit (Zeiler and Fergus, 2013; Simonyan, Vedaldi, and Zisserman, 2013; Yosinski *et al.*, 2015). If we chose the unit to be one of the outputs then we can look for an image that is most representative of the corresponding class label. Because the output units generally have a softmax activation function, it is better to maximise the pre-activation value that feeds into the softmax rather than the class probability directly, as this ensures the optimization depends on only one class. For example, if we seek the image that produces the strongest response to the class 'dog', then if we optimize the softmax output it could drive the image to be, say, less like a cat because of the denominator in the softmax.

Chapter 17

This approach is related to adversarial training. Unconstrained optimization of the output-unit activation, however, leads to individual pixel values being driven to infinity and also creates high-frequency structure that is difficult to interpret, and so some form of regularization is required to find solutions that are closer to natural images. Yosinski *et al.* (2015) used a regularization function comprising the sum of squares of the pixel values along with a procedure that alternates gradient-based updates to the image pixel values with a blurring operation to remove high-frequency structure and a clipping operation that sets to zero those pixel values that make only small contributions to the class label. Example results are shown in Figure 10.14.

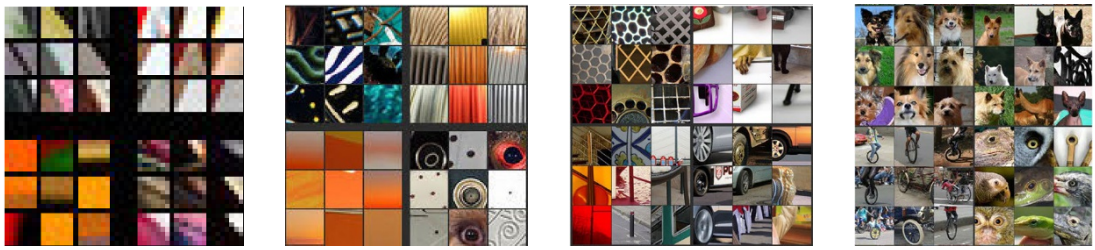


Figure 10.13 Examples of image patches (taken from a validation set) that produce the strongest activation in the hidden units in a network having five convolutional layers trained on ImageNet data. The top nine activations in each feature map are arranged as a 3×3 grid for four randomly chosen channels in each of the corresponding layers. We see a steady progression in complexity with depth, from simple edges in layer 1 to complete objects in layer 5. [From Zeiler and Fergus (2013) with permission.]

10.3.3 Saliency maps

Another way to gain insight into the features used by a convolutional network is through *saliency maps*, which aim to identify those regions of an image that are most significant in determining the class label. This is best done by investigating the final convolutional layer because this still retains spatial localization, which becomes lost in the subsequent fully connected layers, and yet it has the highest level of semantic representation. The Grad-CAM (gradient class activation mapping) method (Selvaraju *et al.*, 2016) first computes, for a given input image, the derivatives of the output-unit pre-activation $a^{(c)}$ for a given class c, before the softmax, with respect to the pre-activations $a_{ij}^{(k)}$ of all the units in the final convolutional layer for channel k. For each channel in that layer, the average of those derivatives is evaluated to give

$$\alpha_k = \frac{1}{M_k} \sum_i \sum_j \frac{\partial a^{(c)}}{\partial a_{ij}^{(k)}} \tag{10.9}$$

where i and j index the rows and columns of channel k, and M_k is the total number of units in that channel. These averages are then used to form a weighted sum of the form:

$$\mathbf{L} = \sum_k \alpha_k \mathbf{A}^{(k)} \tag{10.10}$$

in which $\mathbf{A}^{(k)}$ is a matrix with elements $a_{ij}^{(k)}$. The resulting array has the same dimensionality as the final convolutional layer, for example 14×14 for the VGG network shown in Figure 10.10, and can be superimposed on the original image in the form of a 'heat map' as seen in Figure 10.15.

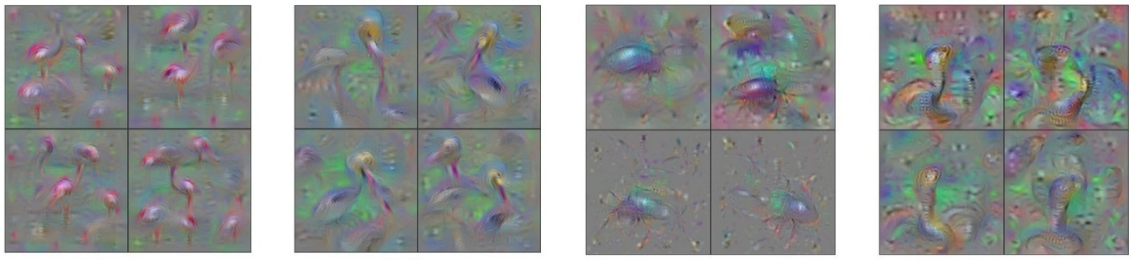


Figure 10.14 Examples of synthetic images generated by maximizing the class probability with respect to the image pixel channel values for a trained convolutional classifier. Four different solutions, obtained with different settings of the regularization parameters, are shown for each of four object classes. [From Yosinski *et al.* (2015) with permission.]

10.3.4 Adversarial attacks

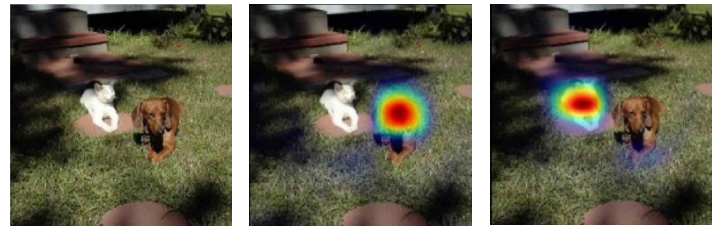
Gradients with respect to changes in the input image pixel values can also be used to create *adversarial attacks* against convolutional networks (Szegedy *et al.*, 2013). These attacks involve making very small modifications to an image, at a level that is imperceptible to a human, which cause the image to be misclassified by the neural network. One simple approach to creating adversarial images is called the *fast gradient sign* method (Goodfellow, Shlens, and Szegedy, 2014). This involves changing each pixel value in an image $\mathbf{x}$ by a fixed amount ϵ with a sign determined by the gradient of an error function $E(\mathbf{x}, t)$ with respect to the pixel values. This gives a modified image defined by

$$\mathbf{x}' = \mathbf{x} + \epsilon \, \text{sign}(\nabla_{\mathbf{x}} E(\mathbf{x}, t)). \tag{10.11}$$

Here t is the true label of $\mathbf{x}$, and the error $E(\mathbf{x}, t)$ could, for example, be the negative log likelihood of $\mathbf{x}$. The required gradient can be computed efficiently using backpropagation. During conventional training of a neural network, the network parameters are adjusted to minimize this error, whereas the modification defined by (10.11) alters the image (while keeping the trained network parameters fixed) so as

Chapter 8

Figure 10.15 Saliency maps for the VGG-16 network with respect to the 'dog' and 'cat' categories. [From Selvaraju *et al.* (2016) with permission.]



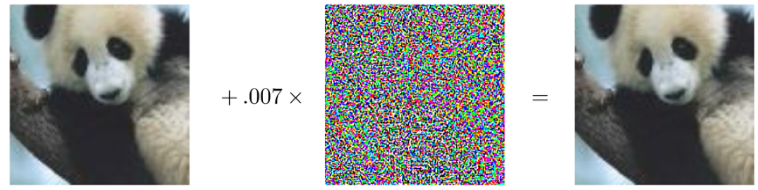


Figure 10.16 Example of an adversarial attack against a trained convolutional network. The image on the left is classified as a panda with confidence 57.7%. The addition of a small level of a random-looking perturbation (that itself is classified as a nematode with confidence 8.2%) results in the image on the right, which is classified as a gibbon with confidence 99.3%. [From Goodfellow, Shlens, and Szegedy (2014) with permission.]

to increase the error. By keeping ϵ small, we ensure that the changes to the image are undetectable to the human eye. Remarkably, this can give images that are misclassified by the network with high confidence, as seen in the example in Figure 10.16.

The ability to fool neural networks in this way raises potential security concerns as it creates opportunities for attacking trained classifiers. It might appear that this issue arises from over-fitting, in which a high-capacity model has adapted precisely to the specific image such that small changes in the input produce large changes in the predicted class probabilities. However, it turns out that an image that has been adapted to give a spurious output for a particular trained network can give similarly spurious outputs when fed to other networks (Goodfellow, Shlens, and Szegedy, 2014). Moreover, a similar adversarial result can be obtained with much less flexible linear models. It is even possible to create physical artefacts such that a regular, uncorrupted image of the artefact will give erroneous predictions when presented to a trained neural network, as seen in Figure 10.17. Although these basic kinds of adversarial attacks can be addressed by simple modifications to the network training process, more sophisticated approaches are harder to defeat. Understanding the implications of these results and mitigating their potential pitfalls remain open areas of research.

Figure 10.17 Two examples of physical stop signs that have been modified. Images of these objects are robustly classified as 45 mph speed-limit signs by CNNs. [From Eykholt *et al.* (2018) with permission.]

10.3.5 Synthetic images

As a final example of image modification that provides additional insights into the operation of a trained convolutional network, we consider a technique called *DeepDream* (Mordvintsev, Olah, and Tyka, 2015). The goal is to generate a synthetic image with exaggerated characteristics. We do this by determining which nodes in a particular hidden layer of the network respond strongly to a particular image and then modifying the image to amplify those responses. For example, if we present an image of some clouds to a network trained on object recognition and a particular node detects a cat-like pattern at a particular region of the image, then we modify the image to be more 'cat like' in that region. To do this, we apply an image to the input of the network and forward propagate through to some particular layer. We then set the backpropagation δ variables for that layer equal to the pre-activations of the nodes and run backpropagation to the input layer to get a gradient vector over the pixels of the image. Finally, we modify the image by taking a small step in the direction of the gradient vector. This procedure can be viewed as a gradient-based method for increasing the function

Section 8.1.2

Exercise 10.10

$$F(\mathbf{I}) = \sum_{i,j,k} a_{ijk}(\mathbf{I})^2 \tag{10.12}$$

where $a_{ijk}(\mathbf{I})$ is the pre-activation of the unit in row i and column j of channel k in the chosen layer when the input image is $\mathbf{I}$, and the sum is over all units and over all channels in that layer. To generate smooth-looking images, some regularization is applied in the form of spatial smoothing and pixel clipping. This process can then be repeated multiple times if stronger enhancements are desired. Examples of the resulting image are shown in Figure 10.18. It is interesting that even though convolutional networks are trained to discriminate between object classes, they seem able to capture at least some of the information needed to generate images from those classes.

This technique can be applied to a photograph, or we can start with inputs consisting of random noise to obtain an image generated entirely from the trained network. Although DeepDream provides some insights into the operation of the trained network, it has primarily been used to generate interesting looking images as a form of artwork.

10.4. Object Detection

We have motivated the design of CNNs primarily by the image classification problem, in which an entire image is assigned to a single class, for example 'cat' or 'bicycle'. This is reasonable for data sets such as ImageNet where, by design, each image is dominated by a single object. However, there are many other applications for CNNs that are able to exploit the inbuilt inductive biases. More generally, the convolutional layers of a CNN trained on a large image data base for a particular task can learn internal representations that have broad applicability, and therefore a

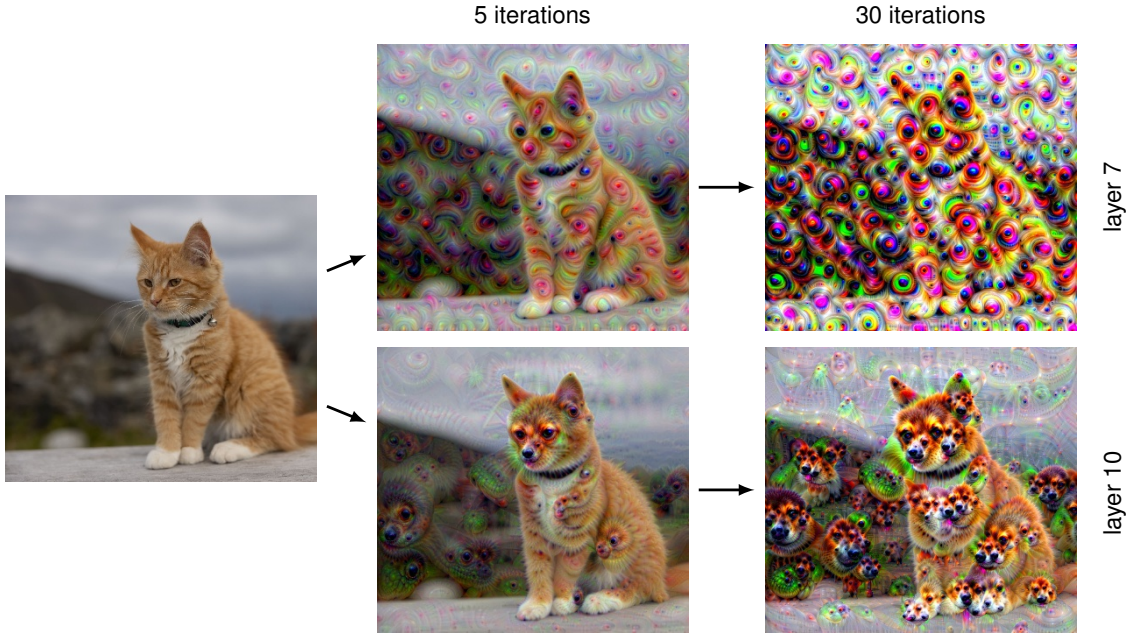


Figure 10.18 Examples of DeepDream applied to an image. The top row shows outputs when the algorithm is applied using the activations from the 7th convolutional layer of the VGG-16 network after five iterations and after 30 iterations. Similarly, the bottom row shows examples using the 10th layer, again after five iterations and after 30 iterations.

Section 6.3.4 CNN can be fine-tuned for a wide range of specific tasks. We have already seen an example of a convolutional network trained on ImageNet data, which through transfer learning was able to achieve human-level performance on skin lesion classi-
Section 1.1.1 fication.

10.4.1 Bounding boxes

Many images have multiple objects belonging to one or more classes, and we may wish to detect the presence and class of each object. Moreover, in many applications of computer vision we also need to determine the locations within the image of any objects that are detected. For example, an autonomous vehicle that uses RGB cameras may need to detect the presence and location of pedestrians and also identify road signs, other vehicles, etc.

Consider the problem of specifying the location of an object in an image. A widely used approach is to define a *bounding box*, which consists of a rectangle that fits closely to the boundary of the object, as illustrated in Figure 10.19. The bounding box can be defined by the coordinates of its centre along with its width and height in the form of a vector $\mathbf{b} = (b_x, b_y, b_{\mathrm{W}}, b_{\mathrm{H}})$. Here the elements of $\mathbf{b}$ can be specified in terms of pixels or as continuous numbers where, by convention, the top left of the image is given coordinates $(0, 0)$ and the bottom right is given coordinates $(1, 1)$.

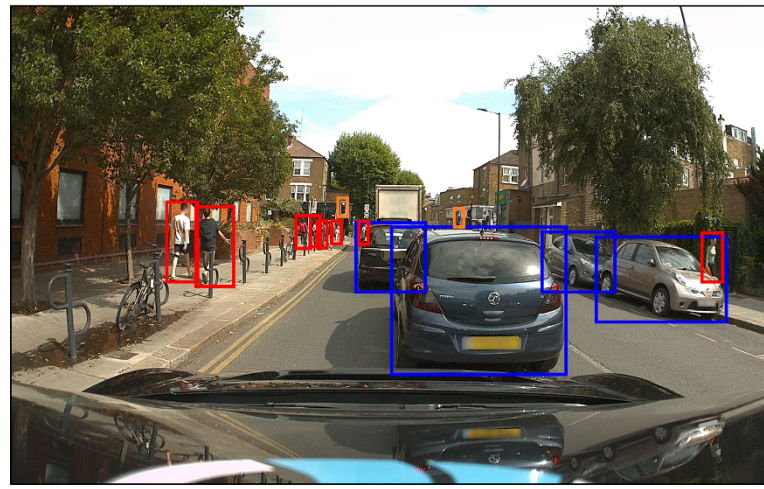


Figure 10.19 An image containing several objects from different classes in which the location of each object is labelled by a close-fitting rectangle known as a bounding box. Here blue boxes correspond to the class 'car', red boxes to the class 'pedestrian', and orange boxes to the class 'traffic light'. [Original image courtesy of Wayve Technologies Ltd.]

When images are assumed to contain one, and only one, object drawn from a predefined set of C classes, a CNN will generally have C output units whose activation functions are defined by the softmax function. An object can be localized by using an additional four outputs, with linear activation functions trained to predict the bounding box coordinates $(b_x, b_y, b_{\mathrm{W}}, b_{\mathrm{H}})$. Since these quantities are continuous, a sum-of-squares error function over the corresponding outputs may be appropriate. This is used for example by Redmon *et al.* (2015), who first divide the image into a 7×7 grid. For each grid cell, they use a convolutional network to output the class and bounding box coordinates of any object associated with that grid cell, based on features taken from the whole image.

Section 5.3

10.4.2 Intersection-over-union

We need a meaningful way to measure the performance of a trained network that can predict bounding boxes. In image classification the output of the network is a probability distribution over class labels, and we can measure performance by looking at the log likelihood for the true class labels on a test set. For object localization, however, we need some way to measure the accuracy of a bounding box relative to some ground truth, where the latter could, for example, be obtained by human labelling. The extent to which the predicted and target boxes overlap can be used as the basis for such a measure, but the area of the overlap will depend on the size of the object within the image. Also a predicted bounding box should be be penalized for the region of the prediction that lies outside the ground truth bounding box. A better metric that addresses both of these issues is called *intersection-over-union*, or IoU, and is simply the ratio of the area of the intersection of the two bounding boxes

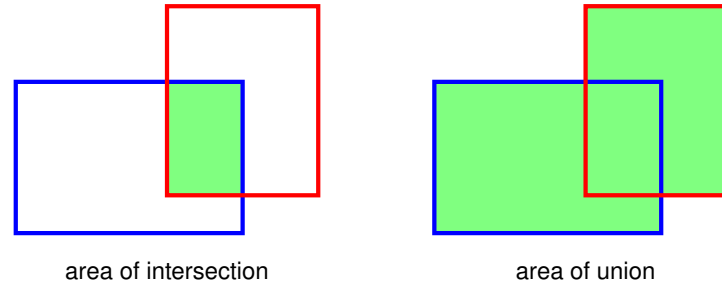


Figure 10.20 Illustration of the intersection-over-union metric for quantifying the accuracy of a bounding box prediction. If the predicted bounding box is shown by the blue rectangle and the ground truth by the red rectangle, then the intersection-over-union is defined as the ratio of the area of the intersection of the boxes, shown in green on the left, divided by the area of their union, shown in green on the right.

to that of their union, as illustrated in Figure 10.20. Note that the IoU measure lies in the range 0 to 1. Predictions can be labelled as correct if the IoU measure exceeds a threshold, which is typically set at 0.5. Note that IoU is not generally used directly as a loss function for training as it is hard to optimize by gradient descent, and so training is typically performed using centred objects, and the IoU score is mainly used an evaluation metric.

10.4.3 Sliding windows

One approach to object detection and object localization starts by creating a training set consisting of tightly cropped examples of the object to be detected, as well as examples of similarly cropped sections of images that do not contain any object (the 'background' class). This data set is used to train a classifier, such as a deep CNN, whose outputs represent the probability of there being an object of each particular class in the input window. The trained model is then used to detect objects in a new image by 'scanning' an input window across the image and, for each location, taking the resulting subset of the image as input to the classifier. This is called a *sliding window*. When an object is detected with high probability, the associated window location then defines the corresponding bounding box.

One obvious drawback of this approach is that it can be computationally very costly due to the large number of potential window positions in the image. Furthermore, the process may have to be repeated using windows of various scales to allow for different sizes of object within the image. A cost saving can be made by moving the input window in strides across the image, both horizontally and vertically, which are larger than one pixel. However, there is a trade-off between precision of location using a small stride and reducing the computational cost by using a larger stride. The computational cost of a sliding window approach may be reasonable for simple classifiers, but for deep neural networks potentially containing millions of parameters, the cost of a naive implementation can be prohibitive.

Fortunately, the convolutional structure of the neural network allows for a dramatic improvement in efficiency (Sermanet *et al.*, 2013). We note that a convolutional layer within such a network itself involves sliding a feature detector, with shared weights, across the input image in strides. Consequently, when a sliding window is used to generate multiple forward passes through a convolutional network

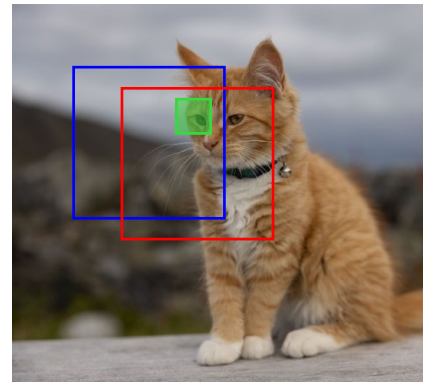


Figure 10.21 Illustration of replicated calculations when a CNN is used to process data from a sliding input window, in which the red and blue boxes show two overlapping locations for the input window. The green box represents one of the locations for the receptive field of a hidden unit in the first convolutional layer, and the evaluation of the corresponding hidden-unit activation is shared across the two window positions.

there is substantial redundancy in the computation, as illustrated in Figure 10.21.

Because the computational structure of sliding windows mirrors that of convolutions, it turns out to be remarkably simple to implement sliding windows efficiently in a convolutional network. Consider the simplified convolutional network in Figure 10.22, which consists of a convolutional layer followed by a max-pooling layer followed by a fully connected layer. For simplicity we have shown only a single channel in each layer, but the extension to multiple channels is straightforward. The input image to the network has size 6×6, the filters in the convolutional layer have size 3×3 with stride 1, and the max-pooling layer has non-overlapping receptive fields of size 2×2 with stride 1. This is followed by a fully connected layer with a single output unit. Note that we can also view this final layer as another convolutional layer with a filter size that is 2×2, so that there is only a single position for the filter and hence a single output.

Now suppose this network is trained on centred images of objects and then applied to a larger image of size 8×8, as shown in Figure 10.23 in which we simply enlarge the network by increasing the size of the convolutional and max-pooling layers. The convolution layer now has size 6×6 and the pooling layer has size

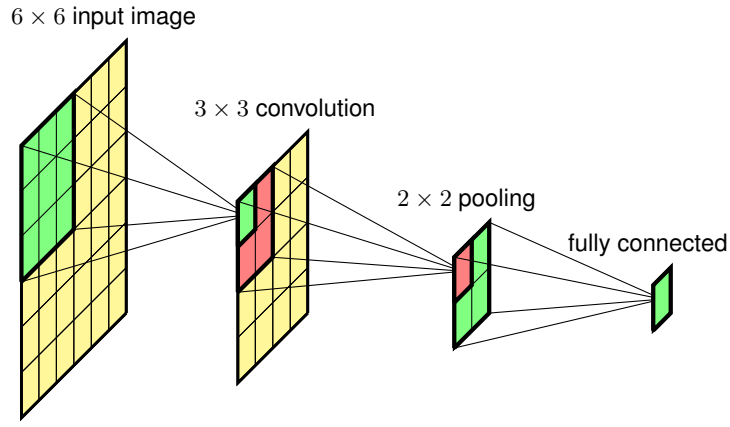


Figure 10.22 Example of a simple convolutional network having a single channel at each layer used to illustrate the concept of a sliding window for detecting objects in images.

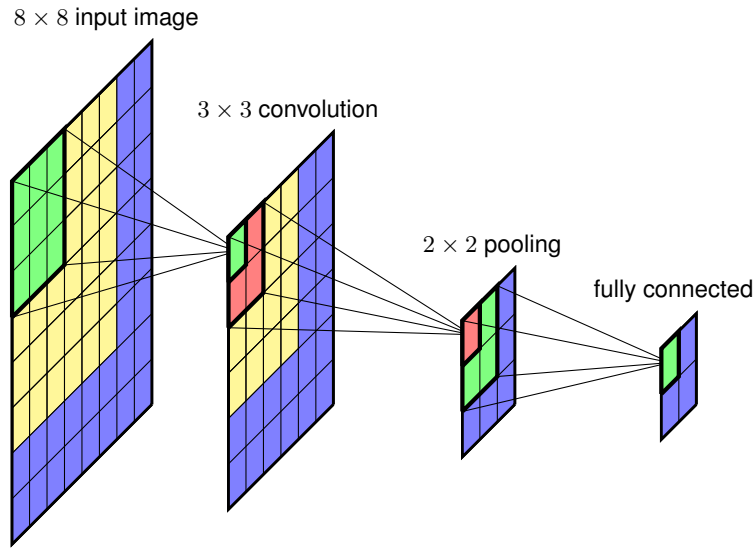


Figure 10.23 Application of the network shown in Figure 10.22 to a larger image in which the additional computation required corresponds to the blue regions.

3×3. There are now four output units each of which has its own softmax function. The weights into this unit are shared across the four units. We see that the calculations needed to process the input corresponding to a window position in the top left corner of the input image are the same as those used to process the original 6×6 inputs used in training. For the remaining window positions, only a small amount of additional computation is needed, as indicated by the blue squares, leading to a significant increase in efficiency compared to a naive repeated application of the full convolutional network. Note that the fully connected layers themselves now have a convolutional structure.

Exercise 10.12

10.4.4 Detection across scales

As well as looking for objects in different positions in the image, we also need to look for objects at different scales and at different aspect ratios. For example, a tight bounding box drawn around a cat will have a different aspect ratio when the cat is sitting upright compared to when it is lying down. Instead of using multiple detectors with different sizes and shapes of input window, it is simpler but equivalent to use a fixed input window and to make multiple copies of the input image each with a different pair of horizontal and vertical scaling factors. The input window is then scanned over each of the image copies to detect objects, and the associated scaling factors are then used to transform the bounding box coordinates back into the original image space, as illustrated in Figure 10.24.

Figure 10.24 Illustration of the detection and localization of objects at multiple scales and aspect ratios using a fixed input window. The original image (a) is replicated multiple times and each copy is scaled in the horizontal and/or vertical directions, as illustrated for a horizontal scaling in (b). A fixed-sized window is then scanned over the scaled images. When an object is detected with high probability, as illustrated by the red box in (b), the corresponding window coordinates can be projected back into the original image space to determine the corresponding bounding box as shown in (c).

10.4.5 Non-max suppression

By scanning a trained convolutional network over an image, it is possible to detect multiple instances of the same class of object within the image as well as instances of objects from other classes. However, this also tends to produce multiple detections of the same object at similar locations, as illustrated in Figure 10.25. This can be addressed using *non-max suppression*, which, for each object class in turn, works as follows. It first runs the sliding window over the whole image and evaluates the probability of an object of that class being present at each location. Next it eliminates all the associated bounding boxes whose probability is below some threshold, say 0.7, giving a result of the kind illustrated in Figure 10.25. The box with the highest probability is considered to be a successful detection, and the corresponding bounding box is recorded as a prediction. Next, any other boxes whose IoU with the successful detection box exceeds some threshold, say 0.5, is discarded. This is intended to eliminate multiple nearby detections of the same object. Then of the remaining boxes, the one with the highest probability is declared to be another successful detection, and the elimination step is repeated. The process continues until all bounding boxes have either been discarded or declared as successful detections.

10.4.6 Fast region CNNs

Another way to speed up object detection and localization is to note that a scanning window approach applies the full power of a deep convolutional network to all areas of the image, even though some areas may be unlikely to contain an object. Instead, we can apply some form of computationally cheaper technique, for example a segmentation algorithm, to identify parts of the image where there is a higher probability of finding an object, and then apply the full network only to these areas, leading to techniques such as *fast region proposals with CNN* or *fast R-CNN*

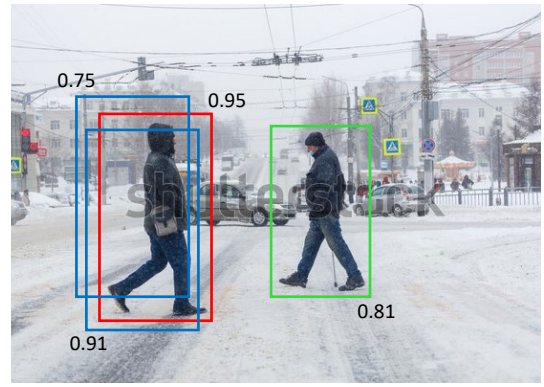


Figure 10.25 Schematic illustration of multiple detections of the same object at nearby locations, along with their associated probabilities. The red bounding box corresponds to the highest overall probability. Non-max suppression eliminates the other overlapping candidate bounding boxes shown in blue, while preserving the detection of another instance of the same object class shown by the bounding box in green.

(Girshick, 2015). It is also possible to use a *region proposal* convolutional network to identify the most promising regions, leading to *faster R-CNN* (Ren *et al.*, 2015), which allows end-to-end training of both the region proposal network and the detection and localization network.

A disadvantage of the sliding window approach is that if we want a very precise localization the objects then we need to consider large numbers of finely spaced window positions, which becomes computationally costly. A more efficient approach is to combine sliding windows with the direct bounding box predictions that we discussed at the start of this section (Sermanet *et al.*, 2013). In this case, the continuous outputs predict the position of the bounding box relative to the window position and therefore provide some fine-tuning to the predicted position.

10.5. Image Segmentation

Section 10.4

In an image classification problem, an entire image is assigned to a single class label. We have seen that more detailed information is provided if multiple objects are detected and their positions recorded using bounding boxes. An even more detailed analysis is obtained with *semantic segmentation* in which every pixel of an image is assigned to one of a predefined set of classes. This means that the output space will have the same dimensionality as the input image and can therefore be conveniently represented as an image with the same number of pixels. Although the input image will generally have three channels for R, G, and B, the output array will have C channels, if there are C classes, representing the probability for each class. If we associate a different (arbitrarily chosen) colour with each class, then the prediction of a segmentation network can be represented as an image in which each pixel is coloured according to the class having the highest probability, as illustrated in Figure 10.26.

10.5.1 Convolutional segmentation

A simple way to approach a semantic segmentation problem would be to construct a convolutional network that takes as input a rectangular section of the image

Figure 10.26 Example of an image and its corresponding semantic segmentation in which each pixel is coloured according to its class. For example, blue pixels correspond to the class 'car', red pixels to the class 'pedestrian', and orange pixels to the class 'traffic light'. [Courtesy of Wayve Technologies Ltd.]

centred on a pixel and that has a single softmax output that classifies that pixel. By applying such a network to each pixel in turn, the entire image can be segmented (this would require edge padding around the image depending on the size of the input window). However, this approach would be extremely inefficient due to redundant calculations caused by overlapping patches. As we have seen, we can remove this inefficiency by grouping together the forward-pass calculations for different input locations into a single network, which results in a model in which the final fully connected layers are also convolutional. We could therefore create a CNN in which each layer has the same dimensionality as the input image, by having stride 1 at each layer with same padding and no pooling. Each output unit has a softmax activation function with weights that are shared across all outputs. Although this could work, such a network would still need many layers, with multiple channels in each layer, to learn the complex internal representations needed to achieve high accuracy, and overall this would be prohibitively costly for images of reasonable resolution.

Figure 10.21

Figure 10.23

10.5.2 Up-sampling

As we have already seen, most convolutional networks use several levels of down-sampling so that as the number of channels increases, the size of the feature maps decreases, keeping the overall size and cost of the network tractable, while allowing the network to extract semantically meaningful high-order features from the image. We can use this concept to create a more efficient architecture for semantic segmentation by taking a standard deep convolutional network and adding additional learnable layers that take the low-dimensional internal representation and transform it back up to the original image resolution (Long, Shelhamer, and Darrell, 2014; Noh, Hong, and Han, 2015; Badrinarayanan, Kendall, and Cipolla, 2015), as illustrated in Figure 10.27.

To do this we need a way to reverse the down-sampling effects of strided convolutions and pooling operations. Consider first the up-sampling analogue of pooling, where the output layer has a larger number of units than the input layer, for example

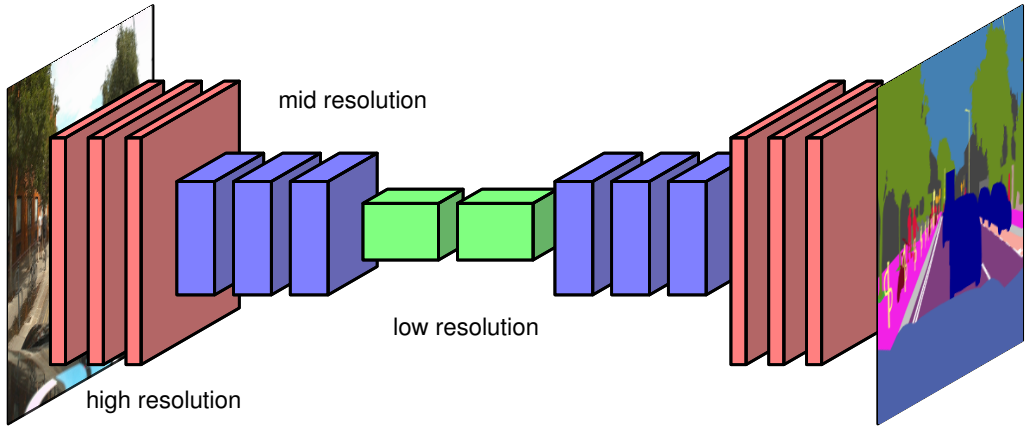


Figure 10.27 Illustration of a convolutional neural network used for semantic image segmentation, showing the reduction in the dimensionality of the feature maps through a series of strided convolutions and/or pooling operations, followed by a series of transpose convolutions and/or unpooling which increase the dimensionality back up to that of the original image.

with each input unit corresponding to a 2×2 block of output units. The question is then what values to use for the outputs. To find an up-sampling analogue of average pooling, we can simply copy over each input value into all the corresponding output units, as shown in Figure 10.28(a). We see that applying average pooling to the output of this operation regenerates the input.

For max-pooling, we can consider the operation shown in Figure 10.28(b) in which each input value is copied into the first unit of the corresponding output block, and the remaining values in each block are set to zero. Again we see that applying a max-pooling operation to the output layer regenerates the input layer. This is sometimes called *max-unpooling*. Assigning the non-zero value to the first element of the output block seems arbitrary, and so a modified approach can be used that also preserves more of the spatial information from the down-sampling layers (Badrinarayanan, Kendall, and Cipolla, 2015). This is done by choosing a network architecture in which each max-pooling down-sampling layer has a corresponding

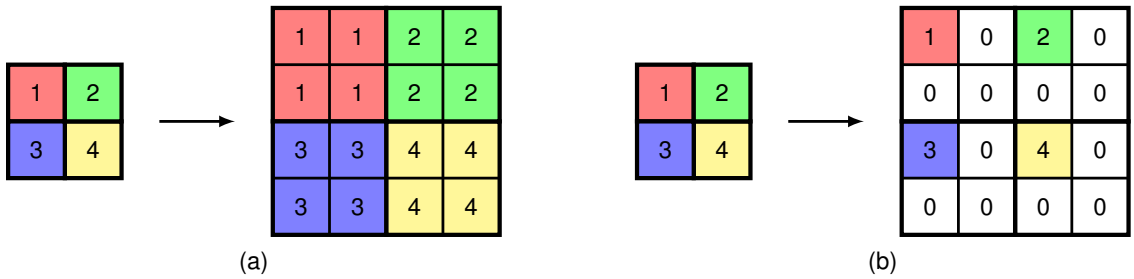


Figure 10.28 Illustration of unpooling operations showing (a) an analogue of average pooling and (b) an analogue of max-pooling.

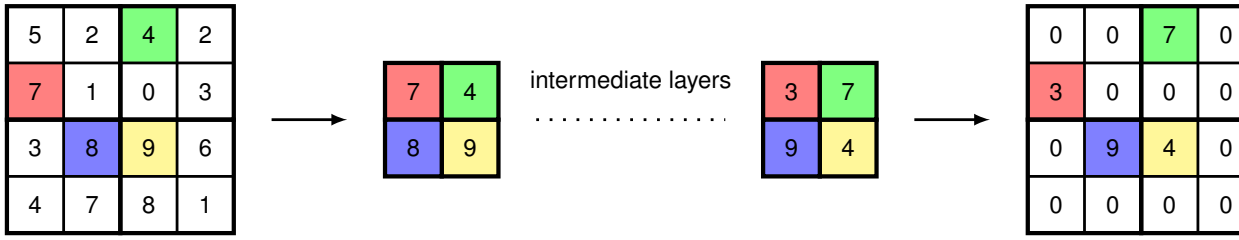


Figure 10.29 Some of the spatial information from a max-pooling layer, shown on the left, can be preserved by noting the location of the maximum value for each 2×2 block in the input array, and then in the corresponding up-sampling layer, placing the non-zero entry at the corresponding location in the output array.

up-sampling layer later in the network. Then during down-sampling, a record is kept of which element in each block had the maximum value, and then in the corresponding up-sampling layer, the non-zero element is chosen to have the same location, as illustrated for 2×2 max-pooling in Figure 10.29.

10.5.3 Fully convolutional networks

The up-sampling methods considered above are fixed functions, much like the average-pooling and max-pooling down-sampling operations. We can also use a *learned* up-sampling that is analogous to strided convolution for down-sampling. In strided convolution, each unit on the output map is connected via shared learnable weights to a small patch on the input map, and as we move one step through the output array, the filter is moved two or more steps across the input array, and hence the output array has lower dimensionality than the input array. For up-sampling, we use a filter that connects one pixel in the input array to a patch in the output array, and then chose the architecture so that as we move one step across the input array, we move two or more steps across the output array (Dumoulin and Visin, 2016). This is illustrated for 3×3 filters, and an output stride of 2, in Figure 10.30. Note that there are output cells for which multiple filter positions overlap, and the corresponding output values can be found either by summing or by averaging the contributions from the individual filter positions.

Exercise 10.13

This up-sampling is called *transpose convolution* because, if the down-sampling convolution is expressed in matrix form, the corresponding up-sampling is given by the transpose matrix. It is also called 'fractionally strided convolution' because the stride of a standard convolution is the ratio of the step size in the output layer to the step size in the input layer. In Figure 10.30, for example, this ratio is $1/2$. Note that this is sometimes also referred to as 'deconvolution', but it is better to avoid this term since deconvolution is widely used in mathematics to mean the inverse of the operation of convolution used in functional analysis, which is a different concept. If we have a network architecture with no pooling layers, so that the down-sampling and up-sampling are done purely using convolutions, then the architecture is known as a *fully convolutional network* (Long, Shelhamer, and Darrell, 2014). It can take an arbitrarily sized image and will output a segmentation map of the same size.

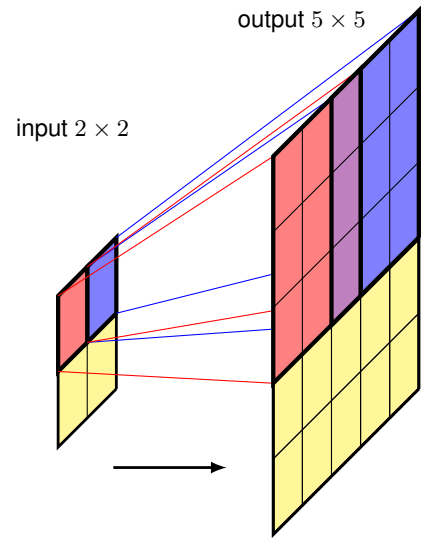


Figure 10.30 Illustration of transpose convolution for a 3×3 filter with an output stride of 2. This can be thought of as the inverse operation to a 3×3 convolution. The red output patch is given by multiplying the kernel by the activation of the red unit in the input layer, and similarly for the blue output patch. The activations of cells for which patches overlap are calculated by summing or averaging the contributions from the individual patches.

10.5.4 The U-net architecture

We have seen that the down-sampling associated with strided convolutions and pooling allows the number of channels to be increased without the size of the network becoming prohibitive. This also has the effect of reducing the spatial resolution and hence discarding positional information as the signals flow through the network. Although this is fine for image classification, the loss of spatial information is a problem for semantic segmentation as we want to classify each pixel. One approach for addressing this is the *U-net* architecture (Ronneberger, Fischer, and Brox, 2015) illustrated in Figure 10.31, where the name comes from the U-shape of the diagram.

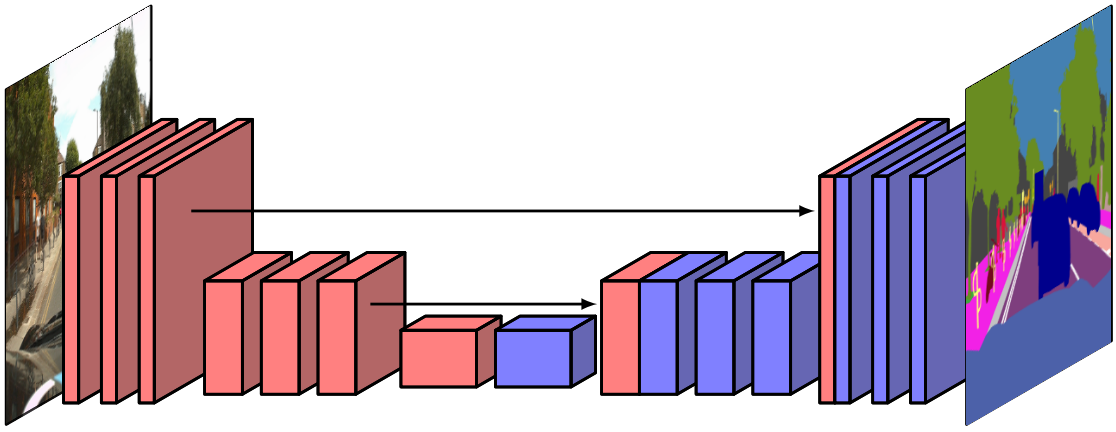


Figure 10.31 The U-net architecture has a symmetrical arrangement of down-sampling and up-sampling layers, and the output from each down-sampling layer is concatenated with the corresponding up-sampling layer.

Figure 10.32 An example of neural style transfer showing a photograph of a canal scene (left) that has been rendered in the style of *The Wreck of a Transport Ship* by J. M. W. Turner (centre) and in the style of *The Starry Night* by Vincent van Gogh (right). In each case the image used to provide the style is shown in the inset. [From Gatys, Ecker, and Bethge (2015) with permission.]

The core concept is that for each down-sampling layer there is a corresponding up-sampling layer, and the final set of channel activations at each down-sampling layer is concatenated with the corresponding first set of channels in the up-sampling layer, thereby giving those layers access to higher-resolution spatial information. Note that 1×1 convolutions may be used in the final layer of a U-net to reduce the number of channels down to the number of classes, which is then followed by a softmax activation function.

10.6. Style Transfer

Section 10.3

As we have seen, early layers in a deep convolutional network learn to detect simple features such as edges and textures whereas later layers learn to detect more complex entities such as objects. We can exploit this property to re-render an image in the style of a different image using a process called *neural style transfer* (Gatys, Ecker, and Bethge, 2015). This is illustrated in Figure 10.32.

Our goal is to generate a synthetic image $\mathbf{G}$ whose 'content' is defined by an image $\mathbf{C}$ and whose 'style' is taken from some other image $\mathbf{S}$. This is achieved by defining an error function $E(\mathbf{G})$ given by the sum of two terms, one of which encourages $\mathbf{G}$ to have a similar content to $\mathbf{C}$ whereas the other encourages $\mathbf{G}$ to have a similar style to $\mathbf{S}$:

$$E(\mathbf{G}) = E_{\text{content}}(\mathbf{G}, \mathbf{C}) + E_{\text{style}}(\mathbf{G}, \mathbf{S}). \tag{10.13}$$

The concepts of content and style are defined implicitly by the functional forms of these two terms. We can then find $\mathbf{G}$ by starting from a randomly initialized image and using gradient descent to minimize $E(\mathbf{G})$.

To define $E_{\text{content}}(\mathbf{G}, \mathbf{C})$, we can pick a particular convolutional layer in the network and measure the activations of the units in that layer when image $\mathbf{G}$ is used as input and also when image $\mathbf{C}$ is used as input. We can then encourage the

corresponding pre-activations to be similar by using a sum-of-squares error function of the form

$$E_{\text{content}}(\mathbf{G}, \mathbf{C}) = \sum_{i,j,k} \{a_{ijk}(\mathbf{G}) - a_{ijk}(\mathbf{C})\}^2 \tag{10.14}$$

where $a_{ijk}(\mathbf{G})$ denotes the pre-activation of the unit at position (i, j) in channel k of that layer when the input image is $\mathbf{G}$, and similarly for $a_{ijk}(\mathbf{C})$. The choice of which layer to use in defining the pre-activations will influence the final result, with earlier layers aiming to match low-level features like edges and later layers matching more complex structures or even entire objects.

In defining $E_{\text{style}}(\mathbf{G}, \mathbf{C})$, the intuition is that style is determined by the co-occurrence of features from different channels within a convolutional layer. For example, if the style image $\mathbf{S}$ is such that vertical edges are generally associated with orange blobs, then we would like the same to be true for the generated image $\mathbf{G}$. However, although $E_{\text{content}}(\mathbf{G}, \mathbf{C})$ tries to match features in $\mathbf{G}$ at the same locations as corresponding features in $\mathbf{C}$, for the style error $E_{\text{style}}(\mathbf{G}, \mathbf{S})$ we want $\mathbf{G}$ to have characteristics that match those of $\mathbf{S}$ but taken from any location, and so we take an average over locations in a feature map. Again, consider a particular convolutional layer. We can measure the extent to which a feature in channel k co-occurs with the corresponding feature in channel k' for input image $\mathbf{G}$ by forming the cross-correlation matrix

$$F_{kk'}(\mathbf{G}) = \sum_{i=1}^{I} \sum_{j=1}^{J} a_{ijk}(\mathbf{G}) a_{ijk'}(\mathbf{G}) \tag{10.15}$$

where I and J are the dimensions of the feature maps in this particular convolutional layer, and the product $a_{ijk}a_{ijk'}$ will be large if both features are activated. If there are K channels in this layer, then $F_{kk'}$ form the elements of a $K \times K$ matrix, called the *style matrix*. We can measure the extent to which the two images $\mathbf{G}$ and $\mathbf{S}$ have the same style by comparing their style matrices using

$$E_{\text{style}}(\mathbf{G}, \mathbf{S}) = \frac{1}{(2IJK)^2} \sum_{k=1}^{K} \sum_{k'=1}^{K} \{F_{kk'}(\mathbf{G}) - F_{kk'}(\mathbf{S})\}^2 . \tag{10.16}$$

Although we could again make use of a single layer, more pleasing results are obtained by using contributions from multiple layers in the form

$$E_{\text{style}}(\mathbf{G}, \mathbf{S}) = \sum_{l} \lambda_l E^{(l)}_{\text{style}}(\mathbf{G}, \mathbf{S}) \tag{10.17}$$

where l denotes the convolutional layer. The coefficients λ_l determine the relative weighting between the different layers and also the weighting relative to the content error term. These weighting coefficients are adjusted empirically using subjective judgement.