Chapter 10

Convolutional networks

Chapters 2–9 introduced the supervised learning pipeline for deep neural networks. However, these chapters only considered fully connected networks with a single path from input to output. Chapters 10–13 introduce more specialized network components with sparser connections, shared weights, and parallel processing paths. This chapter describes *convolutional layers*, which are mainly used for processing image data.

Images have three properties that suggest the need for specialized model architecture. First, they are high-dimensional. A typical image for a classification task contains 224×224 RGB values (i.e., 150,528 input dimensions). Hidden layers in fully connected networks are generally larger than the input size, so even for a shallow network, the number of weights would exceed $150,528^2$, or 22 billion. This poses obvious practical problems in terms of the required training data, memory, and computation.

Second, nearby image pixels are statistically related. However, fully connected networks have no notion of "nearby" and treat the relationship between every input equally. If the pixels of the training and test images were randomly permuted in the same way, the network could still be trained with no practical difference. Third, the interpretation of an image is stable under geometric transformations. An image of a tree is still an image of a tree if we shift it leftwards by a few pixels. However, this shift changes every input to the network. Hence, a fully connected model must learn the patterns of pixels that signify a tree separately at every position, which is clearly inefficient.

Convolutional layers process each local image region independently, using parameters shared across the whole image. They use fewer parameters than fully connected layers, exploit the spatial relationships between nearby pixels, and don't have to re-learn the interpretation of the pixels at every position. A network predominantly consisting of convolutional layers is known as a *convolutional neural network* or *CNN*.

10.1 Invariance and equivariance

We argued above that some properties of images (e.g., tree texture) are stable under transformations. In this section, we make this idea more mathematically precise. A

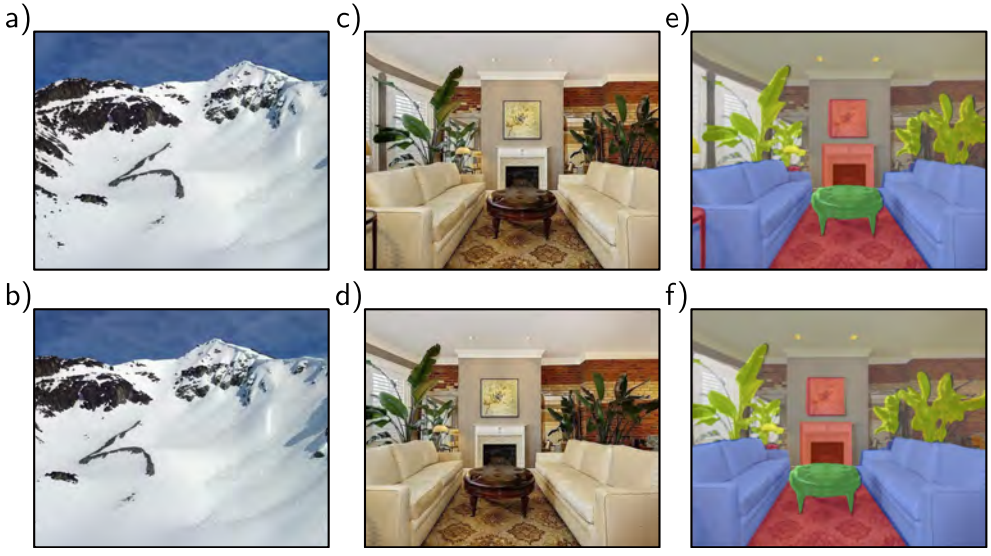

Figure 10.1 Invariance and equivariance for translation. a–b) In image classification, the goal is to categorize both images as "mountain" regardless of the horizontal shift that has occurred. In other words, we require the network prediction to be invariant to translation. c,e) The goal of semantic segmentation is to associate a label with each pixel. d,f) When the input image is translated, we want the output (colored overlay) to translate in the same way. In other words, we require the output to be equivariant with respect to translation. Panels c–f) adapted from Bousselham et al. (2021).

function $\mathrm{f}[\mathbf{x}]$ of an image $\mathbf{x}$ is *invariant* to a transformation $\mathbf{t}[\mathbf{x}]$ if:

$$\mathbf{f}\big[\mathbf{t}[\mathbf{x}]\big] = \mathbf{f}[\mathbf{x}]. \tag{10.1}$$

In other words, the output of the function $\mathrm{f}[\mathbf{x}]$ is the same regardless of the transformation $\mathbf{t}[\mathbf{x}]$. Networks for image classification should be invariant to geometric transformations of the image (figure 10.1a–b). The network $\mathbf{f}[\mathbf{x}]$ should identify an image as containing the same object, even if it has been translated, rotated, flipped, or warped.

A function $\mathrm{f}[\mathbf{x}]$ of an image $\mathbf{x}$ is *equivariant* or *covariant* to a transformation $\mathbf{t}[\mathbf{x}]$ if:

$$\mathbf{f}\big[\mathbf{t}[\mathbf{x}]\big] = \mathbf{t}\big[\mathbf{f}[\mathbf{x}]\big]. \tag{10.2}$$

In other words, $\mathbf{f}[\mathbf{x}]$ is equivariant to the transformation $\mathbf{t}[\mathbf{x}]$ if its output changes in the same way under the transformation as the input. Networks for per-pixel image segmentation should be equivariant to transformations (figure 10.1c–f); if the image is translated, rotated, or flipped, the network $\mathbf{f}[\mathbf{x}]$ should return a segmentation that has been transformed in the same way.

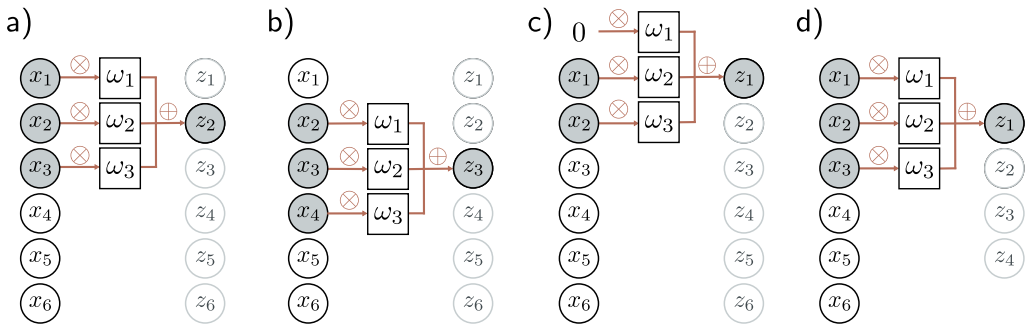

Figure 10.2 1D convolution with kernel size three. Each output z_i is a weighted sum of the nearest three inputs x_{i-1}, x_i, and x_{i+1}, where the weights are $\boldsymbol{\omega} = [\omega_1, \omega_2, \omega_3]$. a) Output z_2 is computed as $z_2 = \omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3$. b) Output z_3 is computed as $z_3 = \omega_1 x_2 + \omega_2 x_3 + \omega_3 x_4$. c) At position z_1, the kernel extends beyond the first input x_1. This can be handled by zero-padding, in which we assume values outside the input are zero. The final output is treated similarly. d) Alternatively, we could only compute outputs where the kernel fits within the input range ("valid" convolution); now, the output will be smaller than the input.

10.2 Convolutional networks for 1D inputs

Convolutional networks consist of a series of convolutional layers, each of which is equivariant to translation. They also typically include pooling mechanisms that induce partial invariance to translation. For clarity of exposition, we first consider convolutional networks for 1D data, which are easier to visualize. In section 10.3, we progress to 2D convolution, which can be applied to image data.

10.2.1 1D convolution operation

Convolutional layers are network layers based on the *convolution* operation. In 1D, a convolution transforms an input vector $\mathbf{x}$ into an output vector $\mathbf{z}$ so that each output z_i is a weighted sum of nearby inputs. The same weights are used at every position and are collectively called the *convolution kernel* or *filter*. The size of the region over which inputs are combined is termed the *kernel size*. For a kernel size of three, we have:

$$z_i = \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1}, \tag{10.3}$$

where $\boldsymbol{\omega} = [\omega_1, \omega_2, \omega_3]^T$ is the kernel (figure 10.2).[1] Notice that the convolution operation is equivariant with respect to translation. If we translate the input x, then the corresponding output z is translated in the same way.

Problem 10.1

[1] Strictly speaking, this is a cross-correlation and not a convolution, in which the weights would be flipped relative to the input (so we would switch x_{i-1} with x_{i+1}). Regardless, this (incorrect) definition is the usual convention in machine learning.

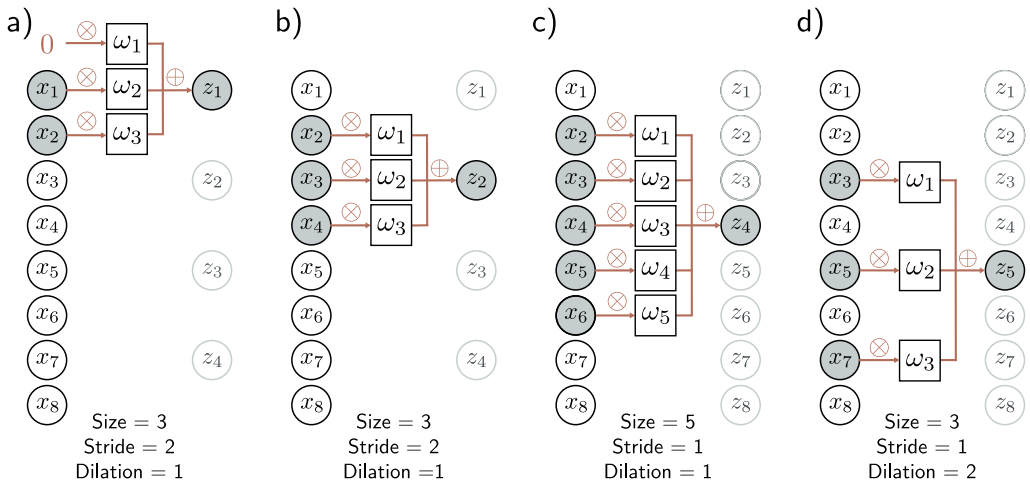

Figure 10.3 Stride, kernel size, and dilation. a) With a stride of two, we evaluate the kernel at every other position, so the first output z_1 is computed from a weighted sum centered at x_1, and b) the second output z_2 is computed from a weighted sum centered at x_3 and so on. c) The kernel size can also be changed. With a kernel size of five, we take a weighted sum of the nearest five inputs. d) In dilated or atrous convolution (from the French "à trous" – with holes), we intersperse zeros in the weight vector to allow us to combine information over a large area using fewer weights.

10.2.2 Padding

Equation 10.3 shows that each output is computed by taking a weighted sum of the previous, current, and subsequent positions in the input. This begs the question of how to deal with the first output (where there is no previous input) and the final output (where there is no subsequent input).

There are two common approaches. The first is to pad the edges of the inputs with new values and proceed as usual. *Zero-padding* assumes the input is zero outside its valid range (figure 10.2c). Other possibilities include treating the input as circular or reflecting it at the boundaries. The second approach is to discard the output positions where the kernel exceeds the range of input positions. These *valid convolutions* have the advantage of introducing no extra information at the edges of the input. However, they have the disadvantage that the representation decreases in size.

10.2.3 Stride, kernel size, and dilation

In the example above, each output was a sum of the nearest three inputs. However, this is just one of a larger family of convolution operations, the members of which are

distinguished by their *stride*, *kernel size*, and *dilation rate*. When we evaluate the output at every position, we term this a *stride* of one. However, it is also possible to shift the kernel by a stride greater than one. If we have a stride of two, we create roughly half the number of outputs (figure 10.3a–b).

The *kernel size* can be increased to integrate over a larger area (figure 10.3c). However, it typically remains an odd number so that it can be centered around the current position. Increasing the kernel size has the disadvantage of requiring more weights. This leads to the idea of *dilated* or *atrous* convolutions, in which the kernel values are interspersed with zeros. For example, we can turn a kernel of size five into a dilated kernel of size three by setting the second and fourth elements to zero. We still integrate information from a larger input region but only require three weights to do this (figure 10.3d). The number of zeros we intersperse between the weights determines the *dilation rate*.

Problems 10.2–10.4

10.2.4 Convolutional layers

A convolutional layer computes its output by convolving the input, adding a bias β, and passing each result through an activation function $\mathrm{a}[\bullet]$. With kernel size three, stride one, and dilation rate one, the i^{th} hidden unit h_i would be computed as:

$$\begin{aligned} h_i &= \mathrm{a}\left[\beta + \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1}\right] \\ &= \mathrm{a}\left[\beta + \sum_{j=1}^{3} \omega_j x_{i+j-2}\right], \end{aligned} \tag{10.4}$$

where the bias β and kernel weights $\omega_1, \omega_2, \omega_3$ are trainable parameters, and (with zero-padding) we treat the input x as zero when it is out of the valid range. This is a special case of a fully connected layer that computes the i^{th} hidden unit as:

$$h_i = \mathrm{a}\left[\beta_i + \sum_{j=1}^{D} \omega_{ij} x_j\right]. \tag{10.5}$$

If there are D inputs $x_\bullet$ and D hidden units $h_\bullet$, this fully connected layer would have D^2 weights $\omega_{\bullet\bullet}$ and D biases $\beta_\bullet$. The convolutional layer only uses three weights and one bias. A fully connected layer can reproduce this exactly if most weights are set to zero and others are constrained to be identical (figure 10.4).

Problem 10.5

10.2.5 Channels

If we only apply a single convolution, information will likely be lost; we are averaging nearby inputs, and the ReLU activation function clips results that are less than zero. Hence, it is usual to compute several convolutions in parallel. Each convolution produces a new set of hidden variables, termed a *feature map* or *channel*.

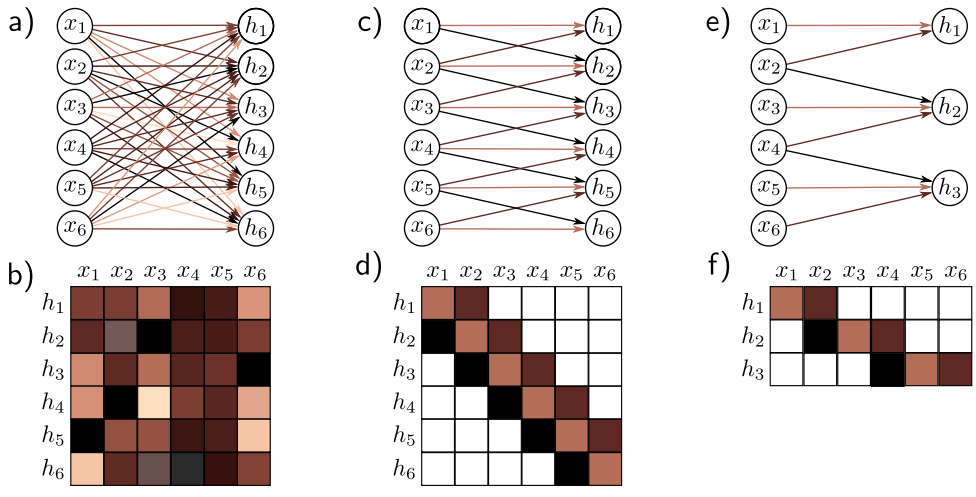

Figure 10.4 Fully connected vs. convolutional layers. a) A fully connected layer has a weight connecting each input x to each hidden unit h (colored arrows) and a bias for each hidden unit (not shown). b) Hence, the associated weight matrix $\mathbf{\Omega}$ contains 36 weights relating the six inputs to the six hidden units. c) A convolutional layer with kernel size three computes each hidden unit as the same weighted sum of the three neighboring inputs (arrows) plus a bias (not shown). d) The weight matrix is a special case of the fully connected matrix where many weights are zero and others are repeated (same colors indicate same value, white indicates zero weight). e) A convolutional layer with kernel size three and stride two computes a weighted sum at every other position. f) This is also a special case of a fully connected network with a different sparse weight structure.

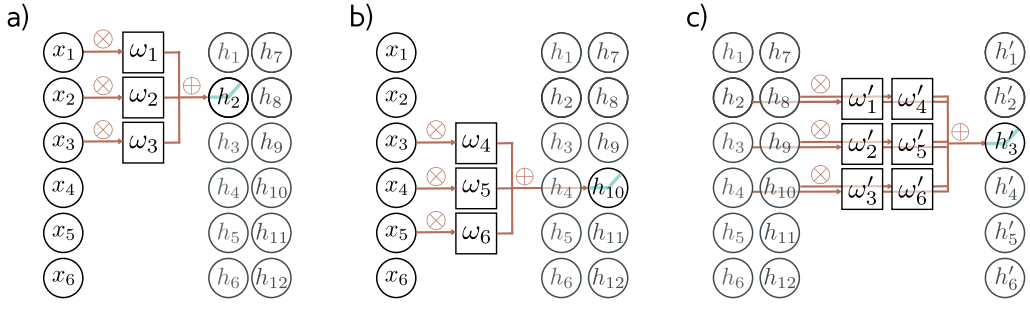

Figure 10.5 Channels. Typically, multiple convolutions are applied to the input $\mathbf{x}$ and stored in channels. a) A convolution is applied to create hidden units h_1 to h_6, which form the first channel. b) A second convolution operation is applied to create hidden units h_7 to h_{12}, which form the second channel. The channels are stored in a 2D array $\mathbf{H}_1$ that contains all the hidden units in the first hidden layer. c) If we add a further convolutional layer, there are now two channels at each input position. Here, the 1D convolution defines a weighted sum over both input channels at the three closest positions to create each new output channel.

Figure 10.5a–b illustrates this with two convolution kernels of size three and with zero-padding. The first kernel computes a weighted sum of the nearest three pixels, adds a bias, and passes the results through the activation function to produce hidden units h_1 to h_6. These comprise the first channel. The second kernel computes a different weighted sum of the nearest three pixels, adds a different bias, and passes the results through the activation function to create hidden units h_7 to h_{12}. These comprise the second channel.

In general, the input and the hidden layers all have multiple channels (figure 10.5c). If the incoming layer has C_i channels and we select a kernel size K per channel, the hidden units in each output channel are computed as a weighted sum over all C_i channels and K kernel entries using a weight matrix $\boldsymbol{\Omega} \in \mathbb{R}^{C_i \times K}$ and one bias. Hence, if there are C_o channels in the next layer, then we need $\boldsymbol{\Omega} \in \mathbb{R}^{C_i \times C_o \times K}$ weights and $\boldsymbol{\beta} \in \mathbb{R}^{C_o}$ biases.

Problems 10.6–10.8

Notebook 10.1
1D convolution

10.2.6 Convolutional networks and receptive fields

Chapter 4 described deep networks, which consisted of a sequence of fully connected layers. Similarly, convolutional networks comprise a sequence of convolutional layers. The *receptive field* of a hidden unit in the network is the region of the original input that feeds into it. Consider a convolutional network where each convolutional layer has kernel size three. The hidden units in the first layer take a weighted sum of the three closest inputs, so have receptive fields of size three. The units in the second layer take a weighted sum of the three closest positions in the first layer, which are themselves weighted sums of three inputs. Hence, the hidden units in the second layer have a receptive field of size five. In this way, the receptive field of units in successive layers increases, and information from across the input is gradually integrated (figure 10.6).

Problems 10.9–10.11

10.2.7 Example: MNIST-1D

We now apply a convolutional network to the MNIST-1D data (see figure 8.1). The input $\mathbf{x}$ is a 40D vector, and the output $\mathbf{f}$ is a 10D vector that is passed through a softmax layer to produce class probabilities. We use a network with three hidden layers (figure 10.7). The fifteen channels of the first hidden layer $\mathbf{H}_1$ are each computed using a kernel size of three and a stride of two with "valid" padding, giving nineteen spatial positions. The second hidden layer $\mathbf{H}_2$ is also computed using a kernel size of three, a stride of two, and "valid" padding. The third hidden layer is computed similarly. At this stage, the representation has four spatial positions and fifteen channels. These values are reshaped into a vector of size sixty, which is mapped by a fully connected layer to the ten output activations.

This network was trained for 100,000 steps using SGD without momentum, a learning rate of 0.01, and a batch size of 100 on a dataset of 4,000 examples. We compare this to a fully connected network with the same number of layers and hidden units (i.e., three hidden layers with 285, 135, and 60 hidden units, respectively). The convolutional network has 2,050 parameters, and the fully connected network has 59,065 parameters. By the logic of figure 10.4, the convolutional network is a special case of the fully connected

Problem 10.12

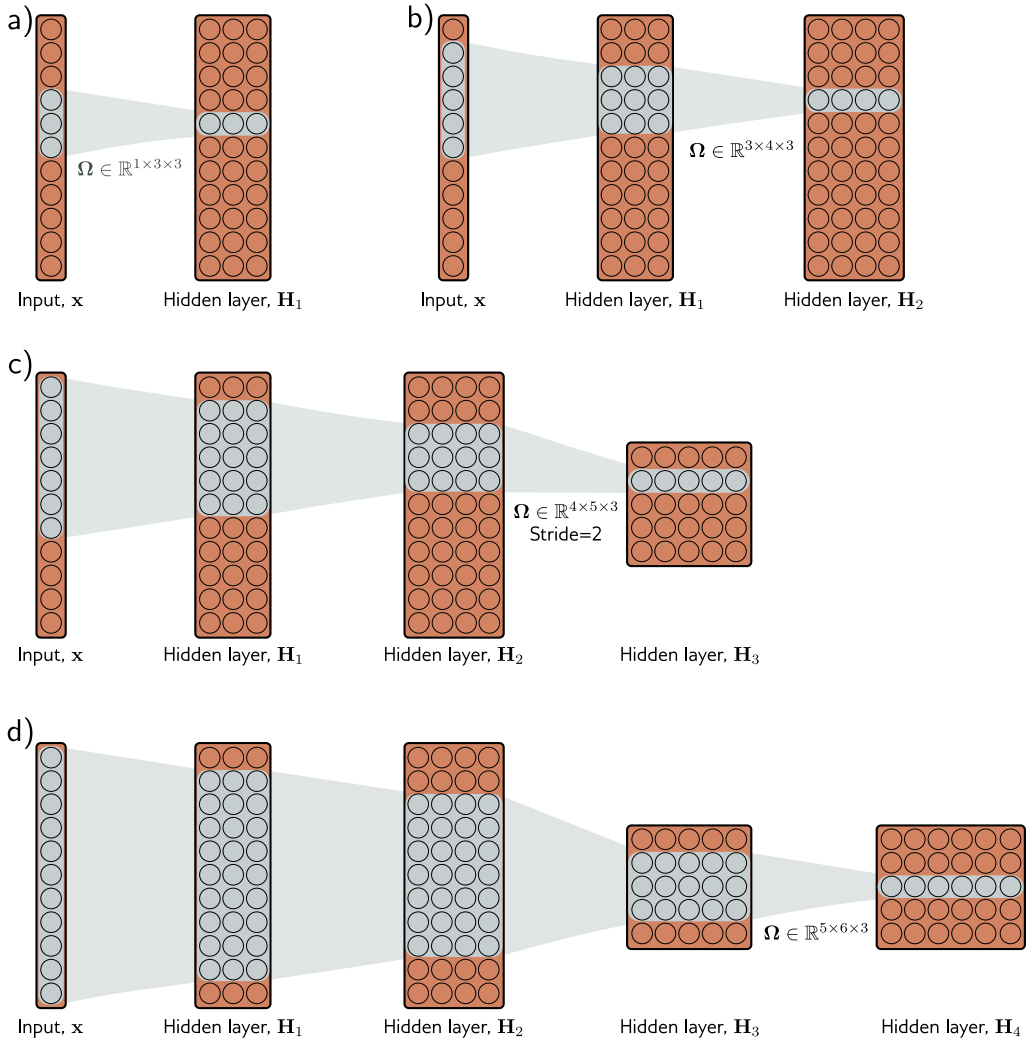

Figure 10.6 Receptive fields for network with kernel width of three. a) An input with eleven dimensions feeds into a hidden layer with three channels and convolution kernel of size three. The pre-activations of the three highlighted hidden units in the first hidden layer $\mathbf{H}_1$ are different weighted sums of the nearest three inputs, so the receptive field in $\mathbf{H}_1$ has size three. b) The pre-activations of the four highlighted hidden units in layer $\mathbf{H}_2$ each take a weighted sum of the three channels in layer $\mathbf{H}_1$ at each of the three nearest positions. Each hidden unit in layer $\mathbf{H}_1$ weights the nearest three input positions. Hence, hidden units in $\mathbf{H}_2$ have a receptive field size of five. c) The hidden units in the third layer (kernel size three, stride two) increases the receptive field size to seven. d) By the time we add a fourth layer, the receptive field of the hidden units at position three have a receptive field that covers the entire input.

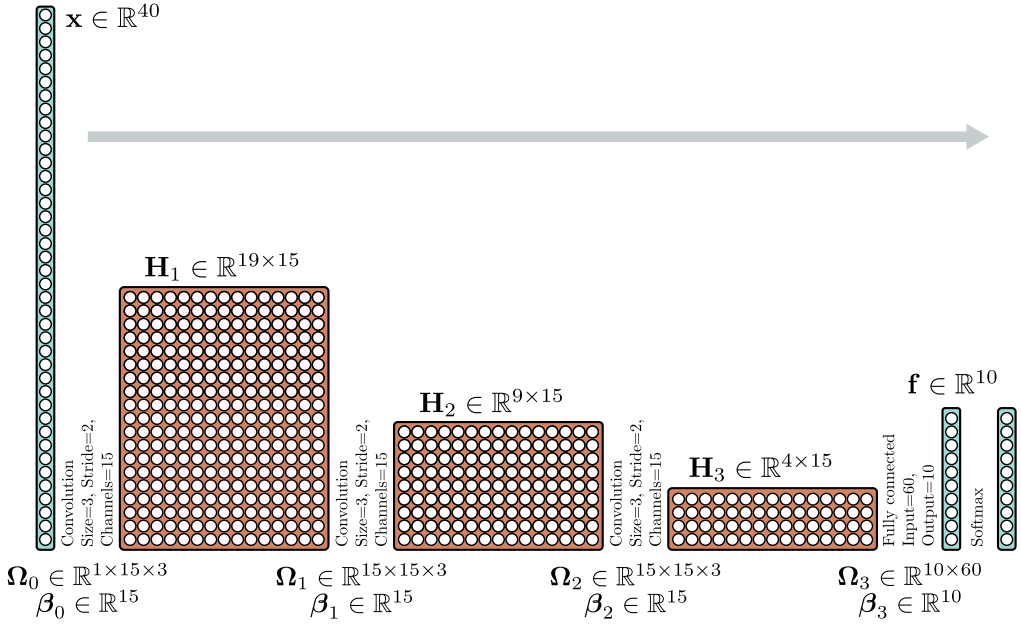

Figure 10.7 Convolutional network for classifying MNIST-1D data (see figure 8.1). The MNIST-1D input has dimension $D_i = 40$. The first convolutional layer has fifteen channels, kernel size three, stride two, and only retains "valid" positions to make a representation with nineteen positions and fifteen channels. The following two convolutional layers have the same settings, gradually reducing the representation size. Finally, a fully connected layer takes all sixty hidden units from the third hidden layer. It outputs ten activations that are subsequently passed through a softmax layer to produce the ten class probabilities.

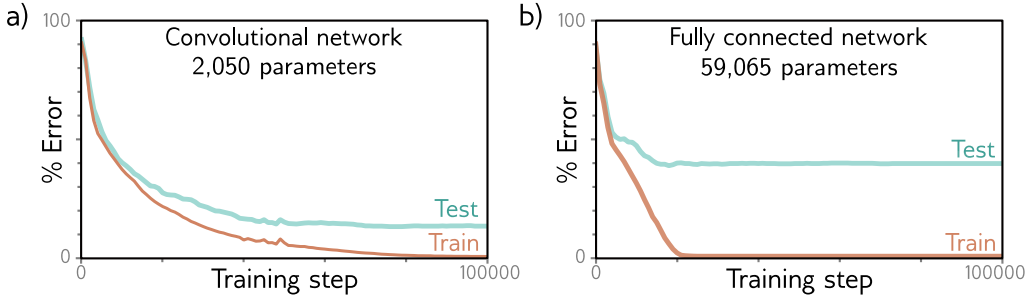

Figure 10.8 MNIST-1D results. a) The convolutional network from figure 10.7 eventually fits the training data perfectly and has ∼17% test error. b) A fully connected network with the same number of hidden layers and the number of hidden units in each learns the training data faster but fails to generalize well with ∼40% test error. The latter model can reproduce the convolutional model but fails to do so. The convolutional structure restricts the possible mappings to those that process every position similarly, and this restriction improves performance.

Notebook 10.2 Convolution for MNIST-1D

one. The latter has enough flexibility to replicate the former exactly. Figure 10.8 shows both models fit the training data perfectly. However, the test error for the convolutional network is much less than for the fully connected network.

This discrepancy is probably not due to the difference in the number of parameters; we know overparameterization usually improves performance (section 8.4.1). The likely explanation is that the convolutional architecture has a superior inductive bias (i.e., interpolates between the training data better) because we have embodied some prior knowledge in the architecture; we have forced the network to process each position in the input in the same way. We know that the data were created by starting with a template that is (among other operations) randomly translated, so this is sensible.

The fully connected network has to learn what each digit template looks like at every position. In contrast, the convolutional network shares information across positions and hence learns to identify each category more accurately. Another way of thinking about this is that when we train the convolutional network, we search through a smaller family of input/output mappings, all of which are plausible. Alternatively, the convolutional structure can be considered a regularizer that applies an infinite penalty to most of the solutions that a fully connected network can describe.

10.3 Convolutional networks for 2D inputs

The previous section described convolutional networks for processing 1D data. Such networks can be applied to financial time series, audio, and text. However, convolutional networks are more usually applied to 2D image data. The convolutional kernel is now a 2D object. A 3×3 kernel $\mathbf{\Omega} \in \mathbb{R}^{3\times3}$ applied to a 2D input comprising of elements x_{ij} computes a single layer of hidden units h_{ij} as:

$$h_{ij} = \mathrm{a}\left[\beta + \sum_{m=1}^{3}\sum_{n=1}^{3}\omega_{mn}x_{i+m-2,j+n-2}\right], \tag{10.6}$$

where ω_{mn} are the entries of the convolutional kernel. This is simply a weighted sum over a square 3×3 input region. The kernel is translated both horizontally and vertically across the 2D input (figure 10.9) to create an output at each position.

Problem 10.13

Notebook 10.3 2D convolution

Problem 10.14

Appendix B.3 Tensors

Often the input is an RGB image, which is treated as a 2D signal with three channels (figure 10.10). Here, a 3×3 kernel would have 3×3×3 weights and be applied to the three input channels at each of the 3×3 positions to create a 2D output that is the same height and width as the input image (assuming zero-padding). To generate multiple output channels, we repeat this process with different kernel weights and append the results to form a 3D tensor. If the kernel is size $K \times K$, and there are C_i input channels, each output channel is a weighted sum of $C_i \times K \times K$ quantities plus one bias. It follows that to compute C_o output channels, we need $C_i \times C_o \times K \times K$ weights and C_o biases.

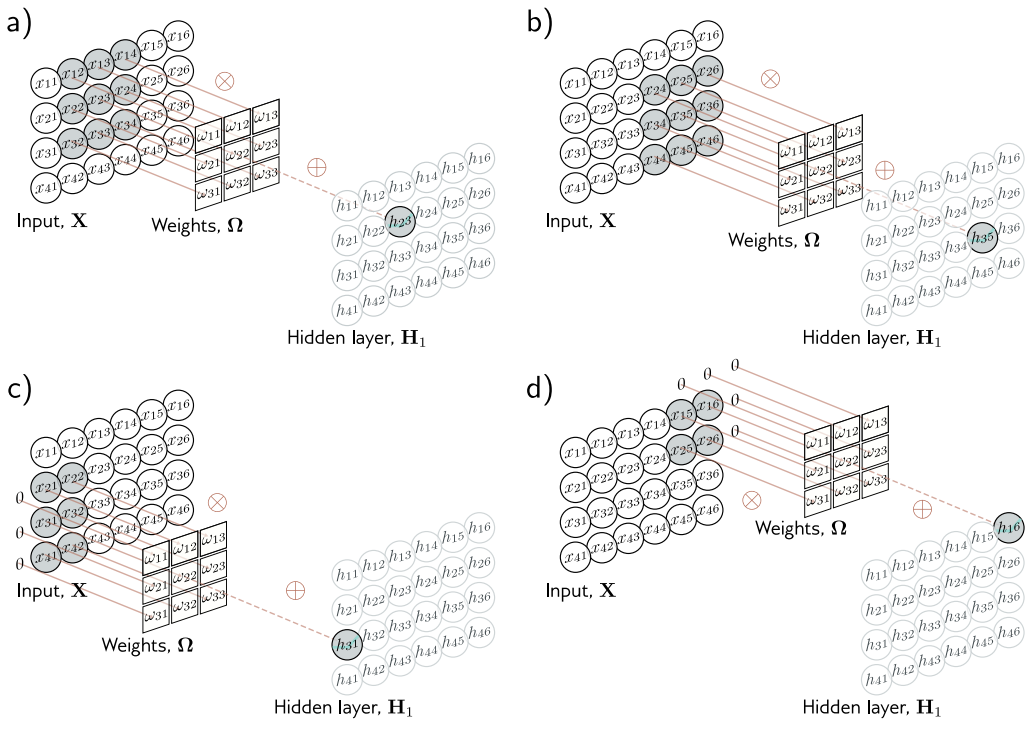

Figure 10.9 2D convolutional layer. Each output h_{ij} computes a weighted sum of the 3×3 nearest inputs, adds a bias, and passes the result through an activation function. a) Here, the output h_{23} (shaded output) is a weighted sum of the nine positions from x_{12} to x_{34} (shaded inputs). b) Different outputs are computed by translating the kernel across the image grid in two dimensions. c–d) With zero-padding, positions beyond the image's edge are considered to be zero.

10.4 Downsampling and upsampling

The network in figure 10.7 increased receptive field size by scaling down the representation at each layer using stride two convolutions. We now consider methods for scaling down or *downsampling* 2D input representations. We also describe methods for scaling them back up (*upsampling*), which is useful when the output is also an image. Finally, we consider methods to change the number of channels between layers. This is helpful when recombining representations from two branches of a network (chapter 11).

10.4.1 Downsampling

There are three main approaches to scaling down a 2D representation. Here, we consider the most common case of scaling down both dimensions by a factor of two. First, we

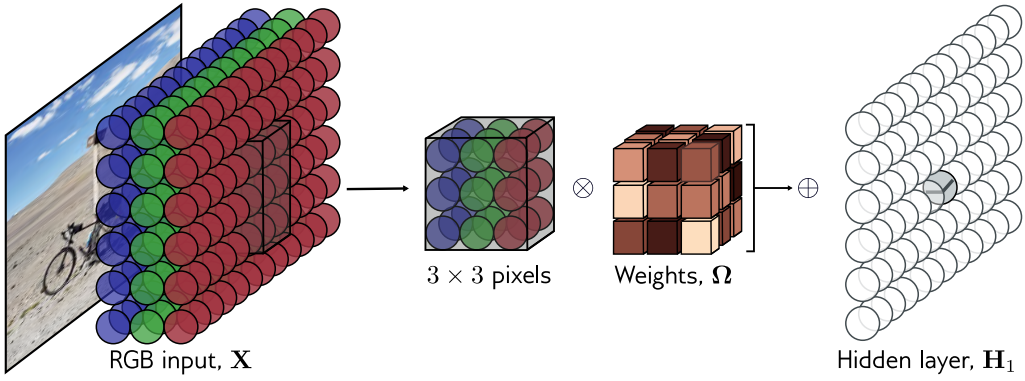

Figure 10.10 2D convolution applied to an image. The image is treated as a 2D input with three channels corresponding to the red, green, and blue components. With a 3×3 kernel, each pre-activation in the first hidden layer is computed by pointwise multiplying the 3×3×3 kernel weights with the 3×3 RGB image patch centered at the same position, summing, and adding the bias. To calculate all the pre-activations in the hidden layer, we "slide" the kernel over the image in both horizontal and vertical directions. The output is a 2D layer of hidden units. To create multiple output channels, we would repeat this process with multiple kernels, resulting in a 3D tensor of hidden units at hidden layer $\mathbf{H}_1$.

Problem 10.15

can sample every other position. When we use a stride of two, we effectively apply this method simultaneously with the convolution operation (figure 10.11a).

Second, *max pooling* retains the maximum of the 2×2 input values (figure 10.11b). This induces some invariance to translation; if the input is shifted by one pixel, many of these maximum values remain the same. Finally, *mean pooling* or *average pooling* averages the inputs. For all approaches, we apply downsampling separately to each channel, so the output has half the width and height but the same number of channels.

10.4.2 Upsampling

The simplest way to scale up a network layer to double the resolution is to duplicate all the channels at each spatial position four times (figure 10.12a). A second method is max unpooling; this is used where we have previously used a max pooling operation for downsampling, and we distribute the values to the positions they originated from (figure 10.12b). A third approach uses bilinear interpolation to fill in the missing values between the points where we have samples. (figure 10.12c).

Notebook 10.4 Downsampling & upsampling

A fourth approach is roughly analogous to downsampling using a stride of two. In that method, there were half as many outputs as inputs, and for kernel size three, each output was a weighted sum of the three closest inputs (figure 10.13a). In *transposed convolution*, this picture is reversed (figure 10.13c). There are twice as many outputs

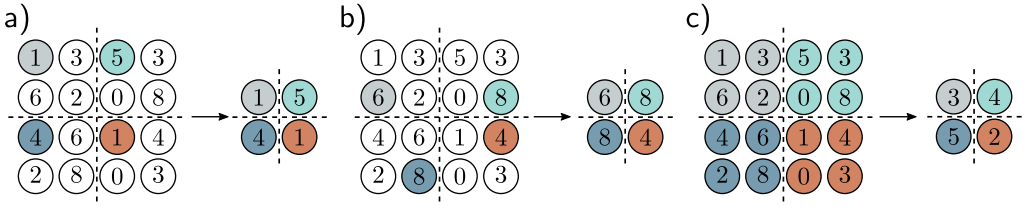

Figure 10.11 Methods for scaling down representation size (downsampling). a) Sub-sampling. The original 4×4 representation (left) is reduced to size 2×2 (right) by retaining every other input. Colors on the left indicate which inputs contribute to the outputs on the right. This is effectively what happens with a kernel of stride two, except that the intermediate values are never computed. b) Max pooling. Each output comprises the maximum value of the corresponding 2×2 block. c) Mean pooling. Each output is the mean of the values in the 2×2 block.

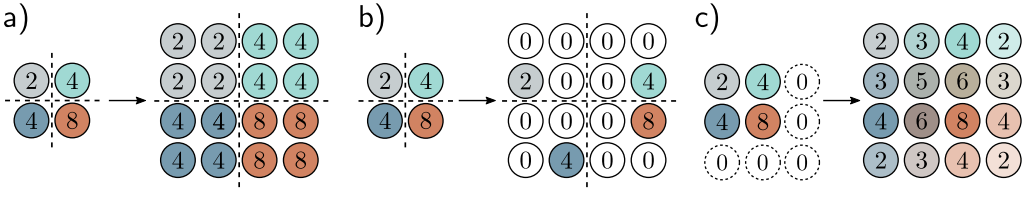

Figure 10.12 Methods for scaling up representation size (upsampling). a) The simplest way to double the size of a 2D layer is to duplicate each input four times. b) In networks where we have previously used a max pooling operation (figure 10.11b), we can redistribute the values to the same positions they originally came from (i.e., where the maxima were). This is known as max unpooling. c) A third option is bilinear interpolation between the input values.

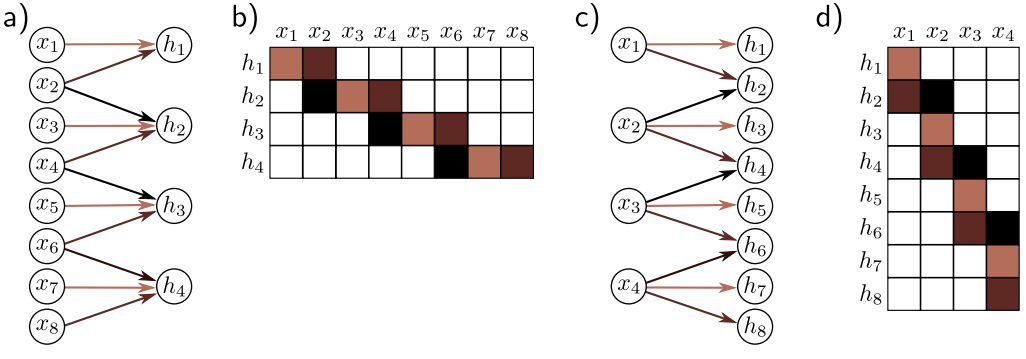

Figure 10.13 Transposed convolution in 1D. a) Downsampling with kernel size three, stride two, and zero-padding. Each output is a weighted sum of three inputs (arrows indicate weights). b) This can be expressed by a weight matrix (same color indicates shared weight). c) In transposed convolution, each input contributes three values to the output layer, which has twice as many outputs as inputs. d) The associated weight matrix is the transpose of that in panel (b).

as inputs, and each input contributes to three of the outputs. When we consider the associated weight matrix of this upsampling mechanism (figure 10.13d), we see that it is the transpose of the matrix for the downsampling mechanism (figure 10.13b).

10.4.3 Changing the number of channels

Sometimes we want to change the number of channels between one hidden layer and the next without further spatial pooling. This is usually so we can combine the representation with another parallel computation (see chapter 11). To accomplish this, we apply a convolution with kernel size one. Each element of the output layer is computed by taking a weighted sum of all the channels at the same position (figure 10.14). We can repeat this multiple times with different weights to generate as many output channels as we need. The associated convolution weights have size $1 \times 1 \times C_i \times C_o$. Hence, this is known as *1×1 convolution*. Combined with a bias and activation function, it is equivalent to running the same fully connected network on the input channels at every position.

10.5 Applications

We conclude by describing three computer vision applications. We describe convolutional networks for image classification where the goal is to assign the image to one of a predetermined set of categories. Then we consider object detection, where the goal is to identify multiple objects in an image and find the bounding box around each. Finally, we describe an early system for semantic segmentation where the goal is to assign a label to each pixel according to which object is present.

10.5.1 Image classification

Much of the pioneering work on deep learning in computer vision focused on image classification using the ImageNet dataset (figure 10.15). This contains 1,281,167 training images, 50,000 validation images, and 100,000 test images, and every image is labeled as belonging to one of 1000 possible categories.

Most methods reshape the input images to a standard size; in a typical system, the input $\mathbf{x}$ to the network is a 224×224 RGB image, and the output is a probability distribution over the 1000 classes. The task is challenging; there are a large number of classes, and they exhibit considerable variation (figure 10.15). In 2011, before deep networks were applied, the state-of-the-art method classified the test images with $\sim 25\%$ errors for the correct class being in the top five suggestions. Five years later, the best deep learning models eclipsed human performance.

In 2012, *AlexNet* was the first convolutional network to perform well on this task. It consists of eight hidden layers with ReLU activation functions, of which the first five are convolutional and the rest fully connected (figure 10.16). The network starts by

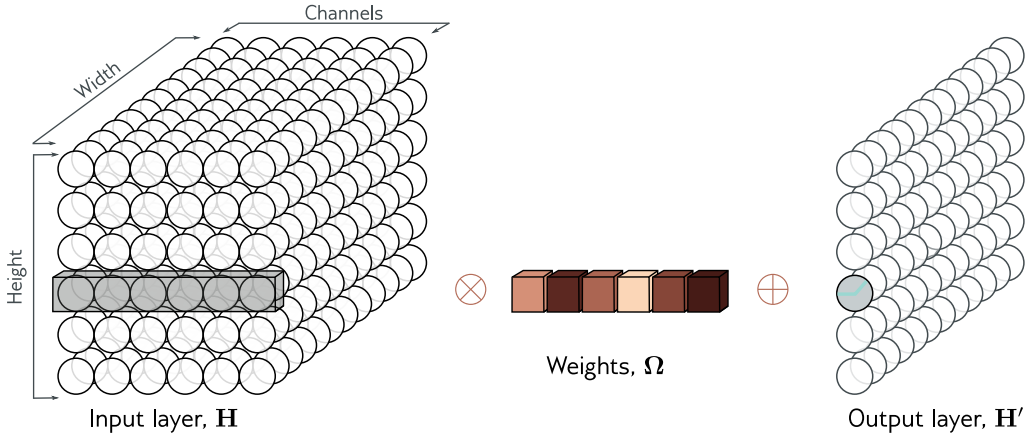

Figure 10.14 1×1 convolution. To change the number of channels without spatial pooling, we apply a 1×1 kernel. Each output channel is computed by taking a weighted sum of all of the channels at the same position, adding a bias, and passing through an activation function. Multiple output channels are created by repeating this operation with different weights and biases.

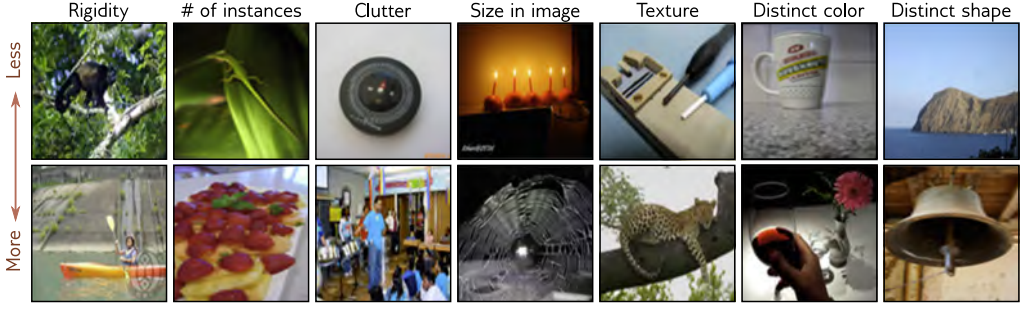

Figure 10.15 Example ImageNet classification images. The model aims to assign an input image to one of 1000 classes. This task is challenging because the images vary widely along different attributes (columns). These include rigidity (monkey < canoe), number of instances in image (lizard < strawberry), clutter (compass < steel drum), size (candle < spiderweb), texture (screwdriver < leopard), distinctiveness of color (mug < red wine), and distinctiveness of shape (headland < bell). Adapted from Russakovsky et al. (2015).

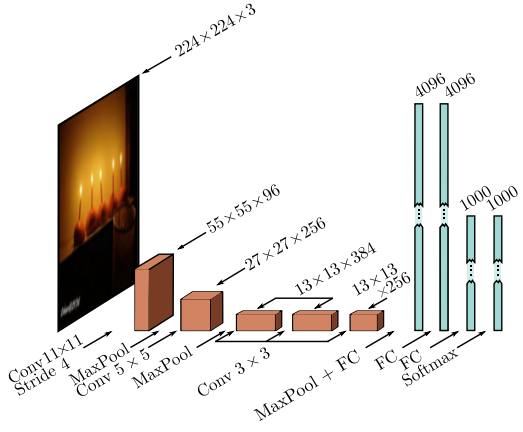

Figure 10.16 AlexNet (Krizhevsky et al., 2012). The network maps a 224×224 color image to a 1000-dimensional vector representing class probabilities. The network first convolves with 11×11 kernels and stride 4 to create 96 channels. It decreases the resolution again using a max pool operation and applies a 5×5 convolutional layer. Another max pooling layer follows, and three 3×3 convolutional layers are applied. After a final max pooling operation, the result is vectorized and passed through three fully connected (FC) layers and finally the softmax layer.

Problems 10.16–10.17

downsampling the input using an 11×11 kernel with a stride of four to create 96 channels. It then downsamples again using a max pooling layer before applying a 5×5 kernel to create 256 channels. There are three more convolutional layers with kernel size 3×3, eventually resulting in a 13×13 representation with 256 channels. A final max-pooling layer yields a 6×6 representation with 256 channels which is resized into a vector of length $9,216$ and passed through three fully connected layers containing 4096, 4096, and 1000 hidden units, respectively. The last layer is passed through the softmax function to output a probability distribution over the 1000 classes. The complete network contains $\sim$60 million parameters, most of which are in the fully connected layers.

Notebook 10.5 Convolution for MNIST

The dataset size was augmented by a factor of 2048 using (i) spatial transformations and (ii) modifications of the input intensities. At test time, five different cropped and mirrored versions of the image were run through the network, and their predictions averaged. The system was learned using SGD with a momentum coefficient of 0.9 and a batch size of 128. Dropout was applied in the fully connected layers, and an L2 (weight decay) regularizer was used. This system achieved a 16.4% top-5 error rate and a 38.1% top-1 error rate. At the time, this was an enormous leap forward in performance at a task considered far beyond the capabilities of contemporary methods. This result revealed the potential of deep learning and kick-started the modern era of AI research.

The *VGG network* was also targeted at classification in the ImageNet task and achieved a considerably better performance of 6.8% top-5 error rate and a 23.7% top-1 error rate. This network is similarly composed of a series of interspersed convolutional and max pooling layers, where the spatial size of the representation gradually decreases, but the number of channels increase. These are followed by three fully connected layers (figure 10.17). The VGG network was also trained using data augmentation, weight decay, and dropout.

Problem 10.18

Although there were various minor differences in the training regime, the most important change between AlexNet and VGG was the depth of the network. The latter used 19 hidden layers and 144 million parameters. The networks in figures 10.16 and 10.17 are depicted at the same scale for comparison. There was a general trend for several years for performance on this task to improve as the depth of the networks increased, and this is evidence that depth is important in neural networks.

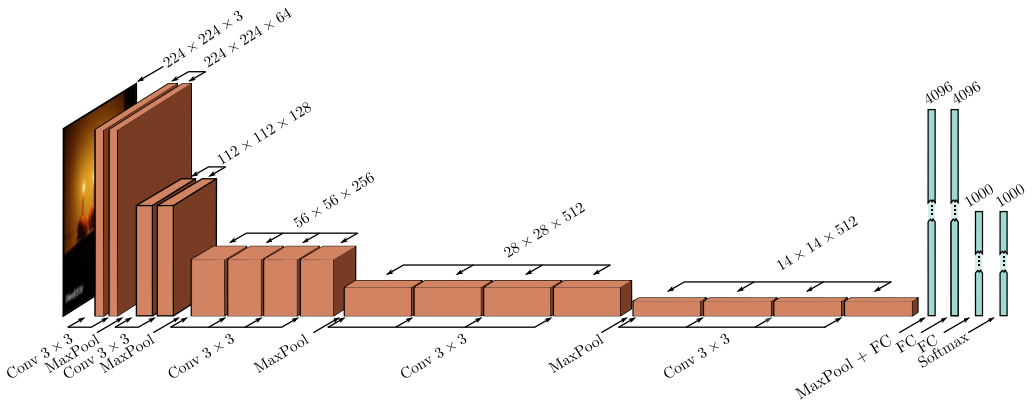

Figure 10.17 VGG network (Simonyan & Zisserman, 2014) depicted at the same scale as AlexNet (see figure 10.16). This network consists of a series of convolutional layers and max pooling operations, in which the spatial scale of the representation gradually decreases, but the number of channels gradually increases. The hidden layer after the last convolutional operation is resized to a 1D vector and three fully connected layers follow. The network outputs 1000 activations corresponding to the class labels that are passed through a softmax function to create class probabilities.

10.5.2 Object detection

In *object detection*, the goal is to identify and localize multiple objects within the image. An early method based on convolutional networks was *You Only Look Once*, or *YOLO* for short. The input to the YOLO network is a 448×448 RGB image. This is passed through 24 convolutional layers that gradually decrease the representation size using max pooling operations while concurrently increasing the number of channels, similarly to the VGG network. The final convolutional layer is of size 7×7 and has 1024 channels. This is reshaped to a vector, and a fully connected layer maps it to 4096 values. One further fully connected layer maps this representation to the output.

The output values encode which class is present at each of a 7×7 grid of locations (figure 10.18a–b). For each location, the output values also encode a fixed number of bounding boxes. Five parameters define each box: the x- and y-positions of the center, the height and width of the box, and the confidence of the prediction (figure 10.18c). The confidence estimates the overlap between the predicted and ground truth bounding boxes. The system is trained using momentum, weight decay, dropout, and data augmentation. Transfer learning is employed; the network is initially trained on the ImageNet classification task and is then fine-tuned for object detection.

After the network is run, a heuristic process is used to remove rectangles with low confidence and to suppress predicted bounding boxes that correspond to the same object so only the most confident one is retained.

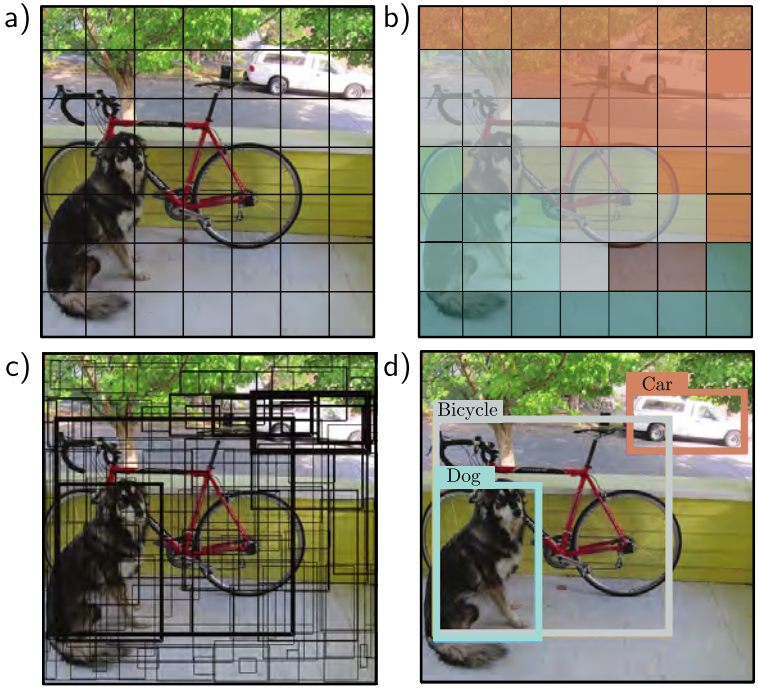

Figure 10.18 YOLO object detection. a) The input image is reshaped to 448×448 and divided into a regular 7×7 grid. b) The system predicts the most likely class at each grid cell. c) It also predicts two bounding boxes per cell, and a confidence value (represented by thickness of line). d) During inference, the most likely bounding boxes are retained, and boxes with lower confidence values that belong to the same object are suppressed. Adapted from Redmon et al. (2016).

10.5.3 Semantic segmentation

The goal of semantic segmentation is to assign a label to each pixel according to the object that it belongs to or no label if that pixel does not correspond to anything in the training database. An early network for semantic segmentation is depicted in figure 10.19. The input is a 224×224 RGB image, and the output is a 224×224×21 array that contains the probability of each of 21 possible classes at each position.

The first part of the network is a smaller version of VGG (figure 10.17) that contains thirteen rather than sixteen convolutional layers and downsizes the representation to size 14×14. There is then one more max pooling operation, followed by two fully connected layers that map to two 1D representations of size 4096. These layers do not represent spatial position but instead, combine information from across the whole image.

Here, the architecture diverges from VGG. Another fully connected layer reconstitutes the representation into 7×7 spatial positions and 512 channels. This is followed

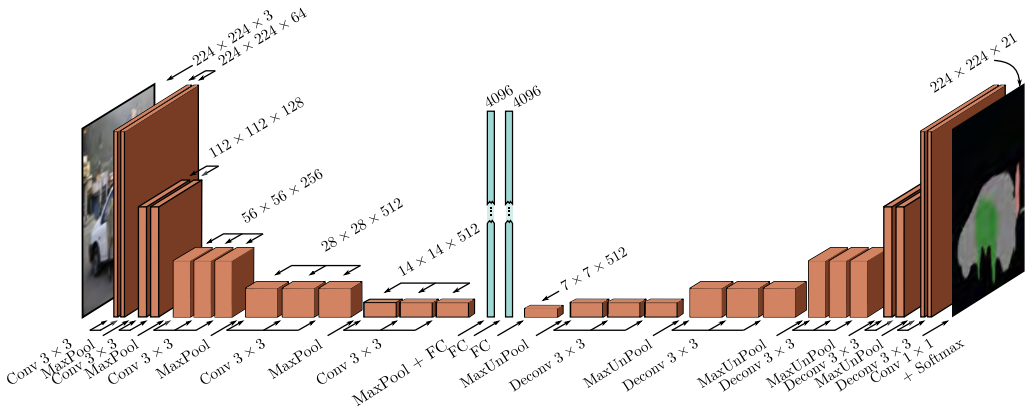

Figure 10.19 Semantic segmentation network of Noh et al. (2015). The input is a 224×224 image, which is passed through a version of the VGG network and eventually transformed into a representation of size 4096 using a fully connected layer. This contains information about the entire image. This is then reformed into a representation of size 7×7 using another fully connected layer, and the image is upsampled and deconvolved (transposed convolutions without upsampling) in a mirror image of the VGG network. The output is a 224×224×21 representation that gives the output probabilities for the 21 classes at each position.

by a series of max unpooling layers (see figure 10.12b) and *deconvolution* layers. These are transposed convolutions (see figure 10.13) but in 2D and without the upsampling. Finally, there is a 1×1 convolution to create 21 channels representing the possible classes and a softmax operation at each spatial position to map the activations to class probabilities. The downsampling side of the network is sometimes referred to as an *encoder*, and the upsampling side as a *decoder*, so networks of this type are sometimes called *encoder-decoder networks* or *hourglass networks* due to their shape.

The final segmentation is generated using a heuristic method that greedily searches for the class that is most represented and infers its region, taking into account the probabilities but also encouraging connectedness. Then the next most-represented class is added where it dominates at the remaining unlabeled pixels. This continues until there is insufficient evidence to add more (figure 10.20).

10.6 Summary

In convolutional layers, each hidden unit is computed by taking a weighted sum of the nearby inputs, adding a bias, and applying an activation function. The weights and the bias are the same at every spatial position, so there are far fewer parameters than in a fully connected network, and the number of parameters doesn't increase with the input image size. To ensure that information is not lost, this operation is repeated with

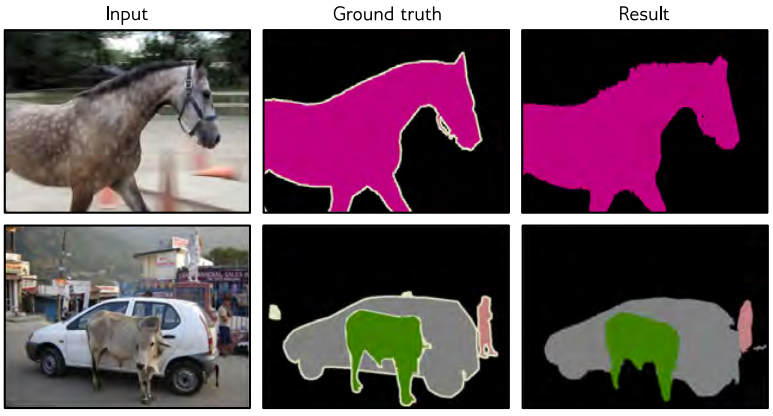

Figure 10.20 Semantic segmentation results. The final result is created from the 21 probability maps by greedily selecting the best class and using a heuristic method to find a sensible binary map based on the probabilities and their spatial proximity. If there is enough evidence, subsequent classes are added, and their segmentation maps are combined. Adapted from Noh et al. (2015).

different weights and biases to create multiple channels at each spatial position.

Typical convolutional networks consist of convolutional layers interspersed with layers that downsample by a factor of two. As the network progresses, the spatial dimensions usually decrease by factors of two, and the number of channels increases by factors of two. At the end of the network, there are typically one or more fully connected layers that integrate information from across the entire input and create the desired output. If the output is an image, a mirrored "decoder" upsamples back to the original size.

The translational equivariance of convolutional layers imposes a useful inductive bias that increases performance for image-based tasks relative to fully connected networks. We described image classification, object detection, and semantic segmentation networks. Image classification performance was shown to improve as the network became deeper. However, subsequent experiments showed that increasing the network depth indefinitely doesn't continue to help; after a certain depth, the system becomes difficult to train. This is the motivation for *residual connections*, which are the topic of the next chapter.

Notes

Dumoulin & Visin (2016) present an overview of the mathematics of convolutions that expands on the brief treatment in this chapter.

Convolutional networks: Early convolutional networks were developed by Fukushima & Miyake (1982), LeCun et al. (1989a), and LeCun et al. (1989b). Initial applications included

handwriting recognition (LeCun et al., 1989a; Martin, 1993), face recognition (Lawrence et al., 1997), phoneme recognition (Waibel et al., 1989), spoken word recognition (Bottou et al., 1990), and signature verification (Bromley et al., 1993). However, convolutional networks were popularized by LeCun et al. (1998), who built a system called LeNet for classifying 28×28 grayscale images of handwritten digits. This is immediately recognizable as a precursor of modern networks; it uses a series of convolutional layers, followed by fully connected layers, sigmoid activations rather than ReLUs, and average pooling rather than max pooling. AlexNet (Krizhevsky et al., 2012) is widely considered the starting point for modern deep convolutional networks.

ImageNet Challenge: Deng et al. (2009) collated the ImageNet database and the associated classification challenge drove progress in deep learning for several years after AlexNet. Notable subsequent winners of this challenge include the *network-in-network* architecture (Lin et al., 2014), which alternated convolutions with fully connected layers that operated independently on all of the channels at each position (i.e., 1×1 convolutions). Zeiler & Fergus (2014) and Simonyan & Zisserman (2014) trained larger and deeper architectures that were fundamentally similar to AlexNet. Szegedy et al. (2017) developed an architecture called *GoogLeNet*, which introduced *inception blocks*. These use several parallel paths with different filter sizes, which are then recombined. This effectively allowed the system to learn the filter size.

The trend was for performance to improve with increasing depth. However, it ultimately became difficult to train deeper networks without modifications; these include residual connections and normalization layers, both of which are described in the next chapter. Progress in the ImageNet challenges is summarized in Russakovsky et al. (2015). A more general survey of image classification using convolutional networks can be found in Rawat & Wang (2017). The improvement of image classification networks over time is visualized in figure 10.21.

Types of convolutional layers: Atrous or dilated convolutions were introduced by Chen et al. (2018c) and Yu & Koltun (2015). Transposed convolutions were introduced by Long et al. (2015). Odena et al. (2016) pointed out that they can lead to checkerboard artifacts and should be used with caution. Lin et al. (2014) is an early example of convolution with 1×1 filters.

Many variants of the standard convolutional layer aim to reduce the number of parameters. These include *depthwise* or *channel-separate convolution* (Howard et al., 2017; Tran et al., 2018), in which a different filter convolves each channel separately to create a new set of channels. For a kernel size of $K \times K$ with C input channels and C output channels, this requires $K \times K \times C$ parameters rather than the $K \times K \times C \times C$ parameters in a regular convolutional layer. A related approach is *grouped convolutions* (Xie et al., 2017), where each convolution kernel is only applied to a subset of the channels with a commensurate reduction in the parameters. In fact, grouped convolutions were used in AlexNet for computational reasons; the whole network could not run on a single GPU, so some channels were processed on one GPU and some on another, with limited interaction points. *Separable convolutions* treat each kernel as an outer product of 1D vectors; they use $C + K + K$ parameters for each of the C channels. *Partial convolutions* (Liu et al., 2018a) are used when inpainting missing pixels and account for the partial masking of the input. *Gated convolutions* learn the mask from the previous layer (Yu et al., 2019; Chang et al., 2019b). Hu et al. (2018b) propose squeeze-and-excitation networks which re-weight the channels using information pooled across all spatial positions.

Downsampling and upsampling: Average pooling dates back to at least LeCun et al. (1989a) and max pooling to Zhou & Chellappa (1988). Scherer et al. (2010) compared these methods and concluded that max pooling was superior. The max unpooling method was introduced by Zeiler et al. (2011) and Zeiler & Fergus (2014). Max pooling can be thought of as applying

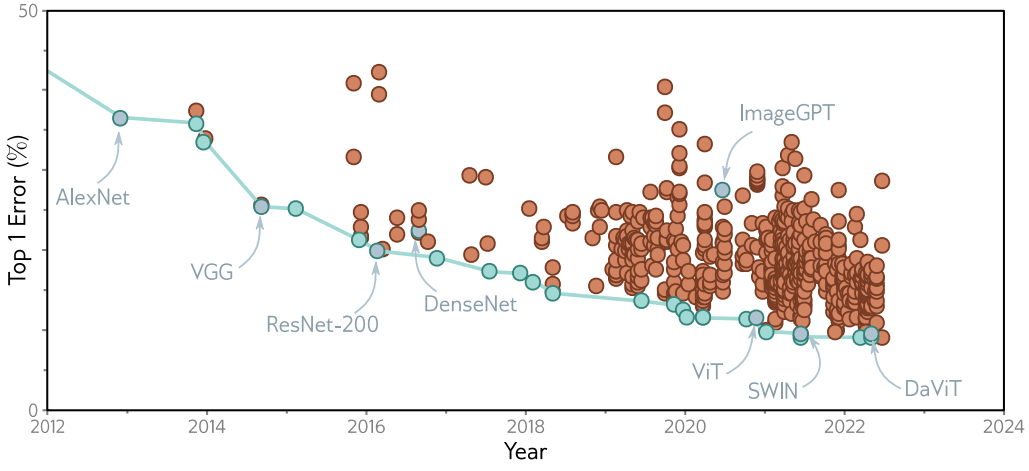

Figure 10.21 ImageNet performance. Each circle represents a different published model. Blue circles represent models that were state-of-the-art. Models discussed in this book are also highlighted. The AlexNet and VGG networks were remarkable for their time but are now far from state of the art. ResNet-200 and DenseNet are discussed in chapter 11. ImageGPT, ViT, SWIN, and DaViT are discussed in chapter 12. Adapted from https://paperswithcode.com/sota/image-classification-on-imagenet.

Appendix B.3.2 Vector norms

an L_∞ norm to the hidden units that are to be pooled. This led to applying other L_k norms (Springenberg et al., 2015; Sainath et al., 2013), although these require more computation and are not widely used. Zhang (2019) introduced *max-blur-pooling*, in which a low-pass filter is applied before downsampling to prevent aliasing, and showed that this improves generalization over translation of the inputs and protects against adversarial attacks (see section 20.4.6).

Shi et al. (2016) introduced *PixelShuffle*, which used convolutional filters with a stride of $1/s$ to scale up 1D signals by a factor of s. Only the weights that lie exactly on positions are used to create the outputs, and the ones that fall between positions are discarded. This can be implemented by multiplying the number of channels in the kernel by a factor of s, where the s^{th} output position is computed from just the s^{th} subset of channels. This can be trivially extended to 2D convolution, which requires s^2 channels.

Convolution in 1D and 3D: Convolutional networks are usually applied to images but have also been applied to 1D data in applications that include speech recognition (Abdel-Hamid et al., 2012), sentence classification (Zhang et al., 2015; Conneau et al., 2017), electrocardiogram classification (Kiranyaz et al., 2015), and bearing fault diagnosis (Eren et al., 2019). A survey of 1D convolutional networks can be found in Kiranyaz et al. (2021). Convolutional networks have also been applied to 3D data, including video (Ji et al., 2012; Saha et al., 2016; Tran et al., 2015) and volumetric measurements (Wu et al., 2015b; Maturana & Scherer, 2015).

Invariance and equivariance: Part of the motivation for convolutional layers is that they are approximately equivariant with respect to translation, and part of the motivation for max

pooling is to induce invariance to small translations. Zhang (2019) considers the degree to which convolutional networks really have these properties and proposes the max-blur-pooling modification that demonstrably improves them. There is considerable interest in making networks equivariant or invariant to other types of transformations, such as reflections, rotations, and scaling. Sifre & Mallat (2013) constructed a system based on wavelets that induced both translational and rotational invariance in image patches and applied this to texture classification. Kanazawa et al. (2014) developed locally scale-invariant convolutional neural networks. Cohen & Welling (2016) exploited group theory to construct *group CNNs*, which are equivariant to larger families of transformations, including reflections and rotations. Esteves et al. (2018) introduced *polar transformer networks*, which are invariant to translations and equivariant to rotation and scale. Worrall et al. (2017) developed *harmonic networks*, the first example of a group CNN that was equivariant to continuous rotations.

Initialization and regularization: Convolutional networks are typically initialized using Xavier initialization (Glorot & Bengio, 2010) or He initialization (He et al., 2015), as described in section 7.5. However, the *ConvolutionOrthogonal* initializer (Xiao et al., 2018a) is specialized for convolutional networks. Networks of up to 10,000 layers can be trained using this initialization without the need for residual connections. Problem 10.19

Dropout is effective for fully connected networks but less so for convolutional layers (Park & Kwak, 2016). This may be because neighboring image pixels are highly correlated, so if a hidden unit drops out, the same information is passed on via adjacent positions. This is the motivation for spatial dropout and cutout. In spatial dropout (Tompson et al., 2015), entire feature maps are discarded instead of individual pixels. This circumvents the problem of neighboring pixels carrying the same information. Similarly, DeVries & Taylor (2017b) propose *cutout*, in which a square patch of each input image is masked at training time. Wu & Gu (2015) modified max pooling for dropout layers using a method that involves sampling from a probability distribution over the constituent elements rather than always taking the maximum.

Adaptive Kernels: The *inception block* (Szegedy et al., 2017) applies convolutional filters of different sizes in parallel and, as such, provides a crude mechanism by which the network can learn the appropriate filter size. Other work has investigated learning the scale of convolutions as part of the training process (e.g., Pintea et al., 2021; Romero et al., 2021) or the stride of downsampling layers (Riad et al., 2022).

In some systems, the kernel size is changed adaptively based on the data. This is sometimes in the context of guided convolution, where one input is used to help guide the computation from another input. For example, an RGB image might be used to help upsample a low-resolution depth map. Jia et al. (2016) directly predicted the filter weights themselves using a different network branch. Xiong et al. (2020b) change the kernel size adaptively. Su et al. (2019a) moderate weights of fixed kernels by a function learned from another modality. Dai et al. (2017) learn offsets of weights so that they do not have to be applied in a regular grid.

Object detection and semantic segmentation: Object detection methods can be divided into *proposal-based* and *proposal-free* schemes. In the former case, processing occurs in two stages. A convolutional network ingests the whole image and proposes regions that might contain objects. These proposal regions are then resized, and a second network analyzes them to establish whether there is an object there and what it is. An early example of this approach was *R-CNN* (Girshick et al., 2014). This was subsequently extended to allow end-to-end training (Girshick, 2015) and to reduce the cost of the region proposals (Ren et al., 2015). Subsequent work on *feature pyramid networks* improved both performance and speed by combining features

across multiple scales Lin et al. (2017b). In contrast, proposal-free schemes perform all the processing in a single pass. YOLO Redmon et al. (2016), which was described in section 10.5.2, is the most celebrated example of a proposal-free scheme. The most recent iteration of this framework at the time of writing is YOLOv7 (Wang et al., 2022a). A recent review of object detection can be found in Zou et al. (2023).

The semantic segmentation network described in section 10.5.3 was developed by Noh et al. (2015). Many subsequent approaches have been variations of U-Net (Ronneberger et al., 2015), which is described in section 11.5.3. Recent surveys of semantic segmentation can be found in Minaee et al. (2021) and Ulku & Akagündüz (2022).

Visualizing Convolutional Networks: The dramatic success of convolutional networks led to a series of efforts to visualize the information they extract from the image (see Qin et al., 2018, for a review). Erhan et al. (2009) visualized the optimal stimulus that activated a hidden unit by starting with an image containing noise and then optimizing the input to make the hidden unit most active using gradient ascent. Zeiler & Fergus (2014) trained a network to reconstruct the input and then set all the hidden units to zero except the one they were interested in; the reconstruction then provides information about what drives the hidden unit. Mahendran & Vedaldi (2015) visualized an entire layer of a network. Their *network inversion* technique aimed to find an image that resulted in the activations at that layer but also incorporates prior knowledge that encourages this image to have similar statistics to natural images.

Finally, Bau et al. (2017) introduced *network dissection*. Here, a series of images with known pixel labels capturing color, texture, and object type are passed through the network, and the correlation of a hidden unit with each property is measured. This method has the advantage that it only uses the forward pass of the network and does not require optimization. These methods did provide some partial insight into how the network processes images. For example, Bau et al. (2017) showed that earlier layers correlate more with texture and color and later layers with the object type. However, it is fair to say that fully understanding the processing of networks containing millions of parameters is currently not possible.

Problems

Problem 10.1* Show that the operation in equation 10.3 is equivariant with respect to translation.

Problem 10.2 Equation 10.3 defines 1D convolution with a kernel size of three, stride of one, and dilation one. Write out the equivalent equation for the 1D convolution with a kernel size of three and a stride of two as pictured in figure 10.3a–b.

Problem 10.3 Write out the equation for the 1D dilated convolution with a kernel size of three and a dilation rate of two, as pictured in figure 10.3d.

Problem 10.4 Write out the equation for a 1D convolution with kernel size of seven, a dilation rate of three, and a stride of three.

Problem 10.5 Draw weight matrices in the style of figure 10.4d for (i) the strided convolution in figure 10.3a–b, (ii) the convolution with kernel size 5 in figure 10.3c, and (iii) the dilated convolution in figure 10.3d.

Problem 10.6* Draw a 12×6 weight matrix in the style of figure 10.4d relating inputs $x_1, \ldots, x_6$ to outputs $h_1, \ldots, h_{12}$ in the multi-channel convolution as depicted in figures 10.5a–b.

Problem 10.7* Draw a 6×12 weight matrix in the style of figure 10.4d relating inputs $h_1, \ldots, h_{12}$ to outputs $h'_1, \ldots, h'_6$ in the multi-channel convolution in figure 10.5c.

Problem 10.8 Consider a 1D convolutional network where the input has three channels. The first hidden layer is computed using a kernel size of three and has four channels. The second hidden layer is computed using a kernel size of five and has ten channels. How many biases and how many weights are needed for each of these two convolutional layers?

Problem 10.9 A network consists of three 1D convolutional layers. At each layer, a zero-padded convolution with kernel size three, stride one, and dilation one is applied. What size is the receptive field of the hidden units in the third layer?

Problem 10.10 A network consists of three 1D convolutional layers. At each layer, a zero-padded convolution with kernel size seven, stride one, and dilation one is applied. What size is the receptive field of hidden units in the third layer?

Problem 10.11 Consider a convolutional network with 1D input $\mathbf{x}$. The first hidden layer $\mathbf{H}_1$ is computed using a convolution with kernel size five, stride two, and a dilation rate of one. The second hidden layer $\mathbf{H}_2$ is computed using a convolution with kernel size three, stride one, and a dilation rate of one. The third hidden layer $\mathbf{H}_3$ is computed using a convolution with kernel size five, stride one, and a dilation rate of two. What are the receptive field sizes at each hidden layer?

Problem 10.12 The 1D convolutional network in figure 10.7 was trained using stochastic gradient descent with a learning rate of 0.01 and a batch size of 100 on a training dataset of 4,000 examples for 100,000 steps. How many epochs was the network trained for?

Problem 10.13 Draw a weight matrix in the style of figure 10.4d that shows the relationship between the 24 inputs and the 24 outputs in figure 10.9.

Problem 10.14 Consider a 2D convolutional layer with kernel size 5×5 that takes 3 input channels and returns 10 output channels. How many convolutional weights are there? How many biases?

Problem 10.15 Draw a weight matrix in the style of figure 10.4d that samples every other variable in a 1D input (i.e., the 1D analog of figure 10.11a). Show that the weight matrix for 1D convolution with kernel size three and stride two is equivalent to composing the matrices for 1D convolution with kernel size three and stride one and this sampling matrix.

Problem 10.16* Consider the AlexNet network (figure 10.16). How many parameters are used in each convolutional and fully connected layer? What is the total number of parameters?

Problem 10.17 What is the receptive field size at each of the first three layers of AlexNet (figure 10.16)?

Problem 10.18 How many weights and biases are there at each convolutional layer and fully connected layer in the VGG architecture (figure 10.17)?

Problem 10.19* Consider two hidden layers of size 224×224 with C_1 and C_2 channels, respectively, connected by a 3×3 convolutional layer. Describe how to initialize the weights using He initialization.