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 kernel size K, the hidden units in each output channel are computed as a weighted sum over all C_i channels and K kernel positions 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

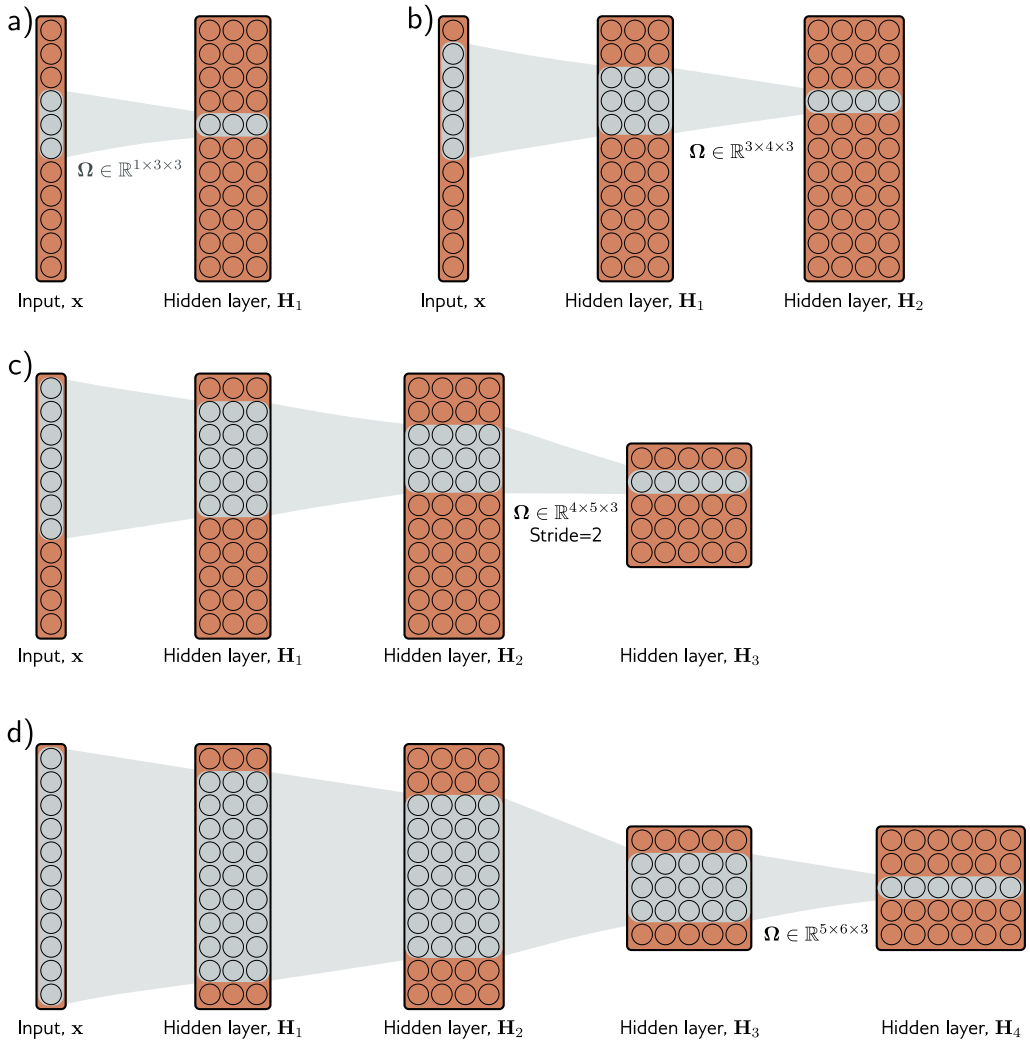

Figure 10.6 Receptive fields for network with kernel width of three. a) An input with eleven dimensions feeds into a hidden layer with three channels and convolution kernel of size three. The pre-activations of the three highlighted hidden units in the first hidden layer $\mathbf{H}_1$ are different 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.

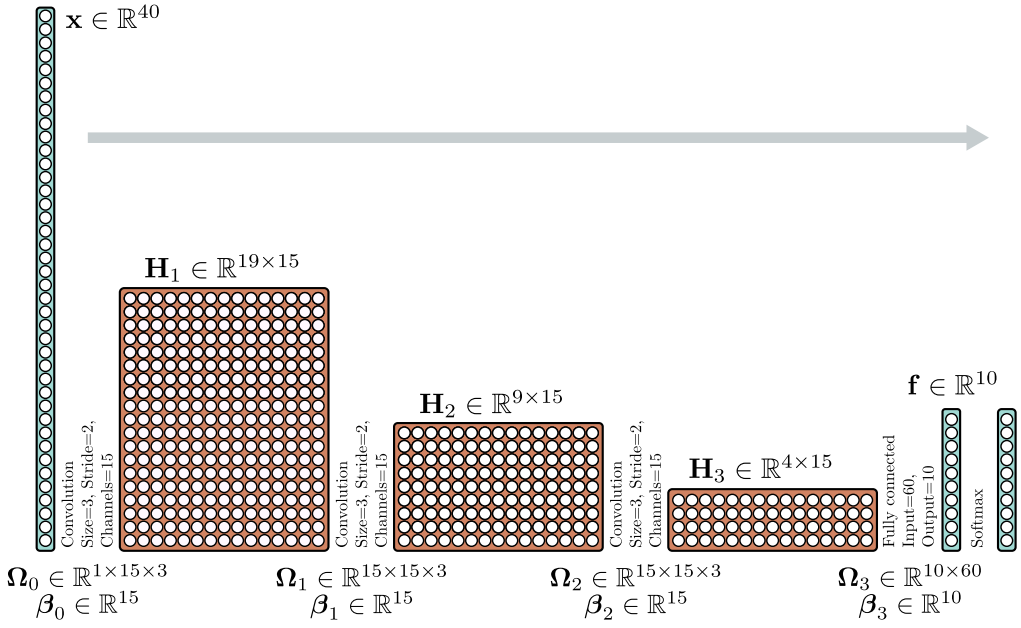

Figure 10.7 Convolutional network for classifying MNIST-1D data (see figure 8.1). The MNIST-1D input has dimension $D_i = 40$. The first convolutional layer has fifteen channels, kernel size three, stride two, and only retains "valid" positions to make a representation with nineteen positions and fifteen 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.

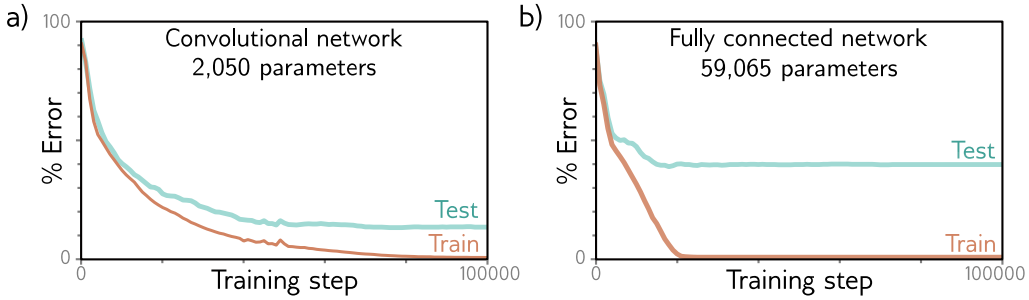

Figure 10.8 MNIST-1D results. a) The convolutional network from figure 10.7 eventually fits the training data perfectly and has ∼17% test error. b) A fully connected network with the same number of hidden layers and the number of hidden units in each learns the training data faster but fails to generalize 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 $\boldsymbol{\Omega} \in \mathbb{R}^{3\times 3}$ 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

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.

Notebook 10.3 2D convolution

Problem 10.14

Appendix B.3 Tensors

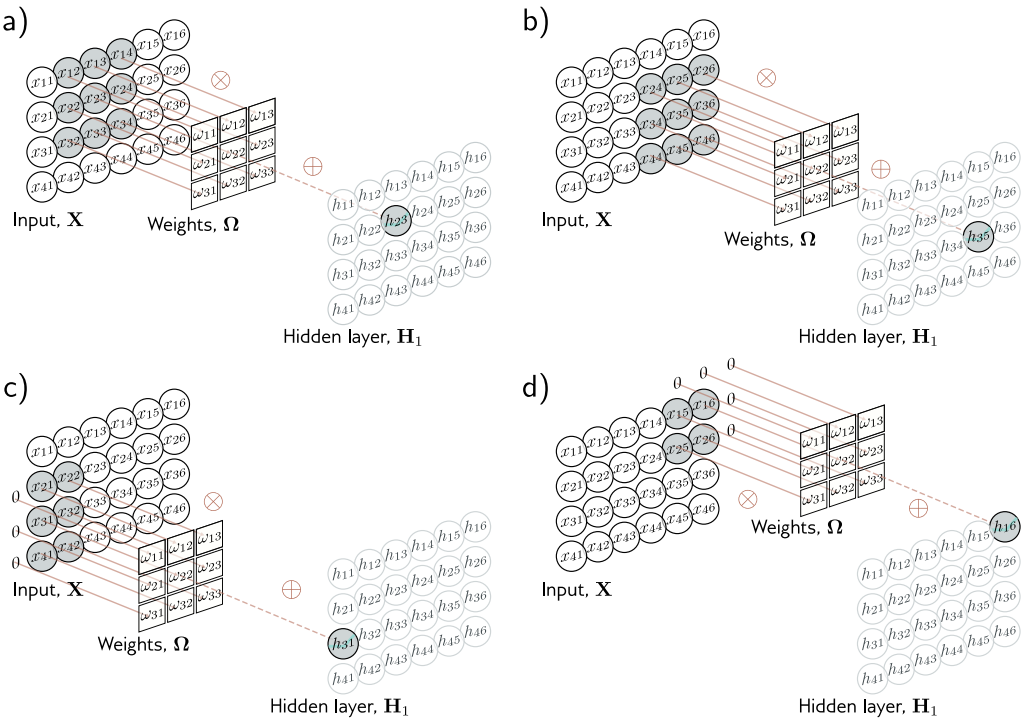

Figure 10.9 2D convolutional layer. Each output h_{ij} computes a weighted sum of the 3×3 nearest inputs, adds a bias, and passes the result through an activation function. a) Here, the output h_{23} (shaded output) is a weighted sum of the nine positions from x_{12} to x_{34} (shaded inputs). b) Different 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

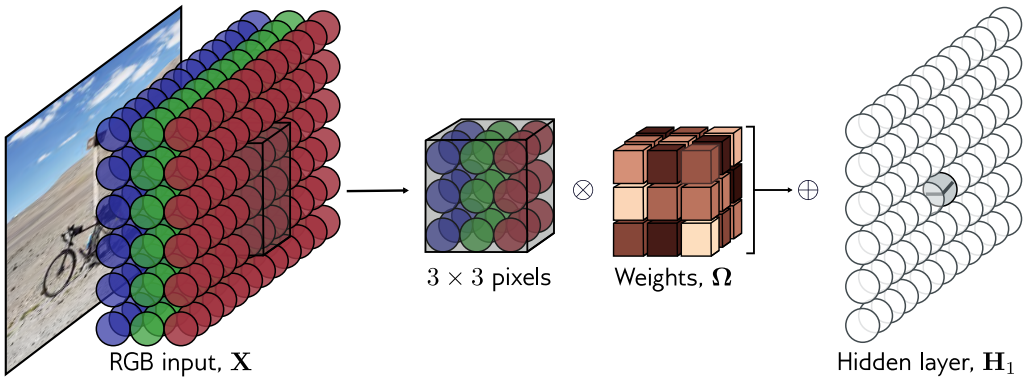

Figure 10.10 2D convolution applied to an image. The image is treated as a 2D input with three channels corresponding to the red, green, and blue components. With a 3×3 kernel, each pre-activation in the first hidden layer is computed by pointwise multiplying the 3×3×3 kernel weights with the 3×3 RGB 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

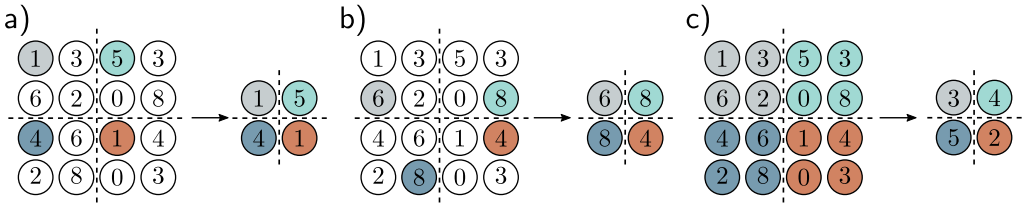

Figure 10.11 Methods for scaling down representation size (downsampling). a) Sub-sampling. The original 4×4 representation (left) is reduced to size 2×2 (right) by retaining every other input. Colors on the left indicate which inputs contribute to the outputs on the right. This is effectively what happens 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.

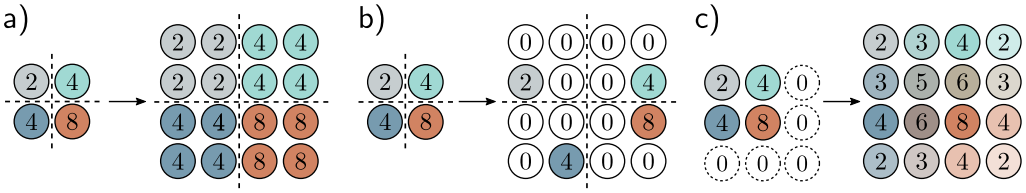

Figure 10.12 Methods for scaling up representation size (upsampling). a) The simplest way to double the size of a 2D layer is to duplicate each input four times. b) In networks where we have previously used a max pooling operation (figure 10.11b), we can redistribute the values to the same positions 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.

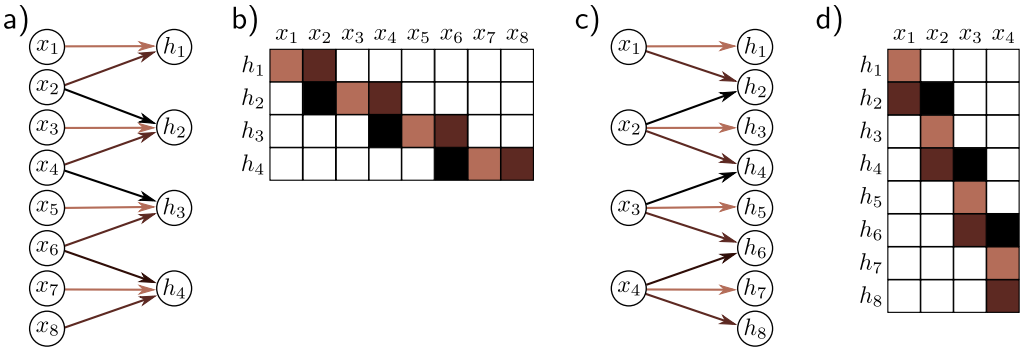

Figure 10.13 Transposed convolution in 1D. a) Downsampling with kernel size three, stride two, and zero padding. Each output is a weighted sum of three inputs (arrows indicate weights). b) This can be expressed by a weight matrix (same color indicates shared weight). c) In transposed convolution, each 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 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

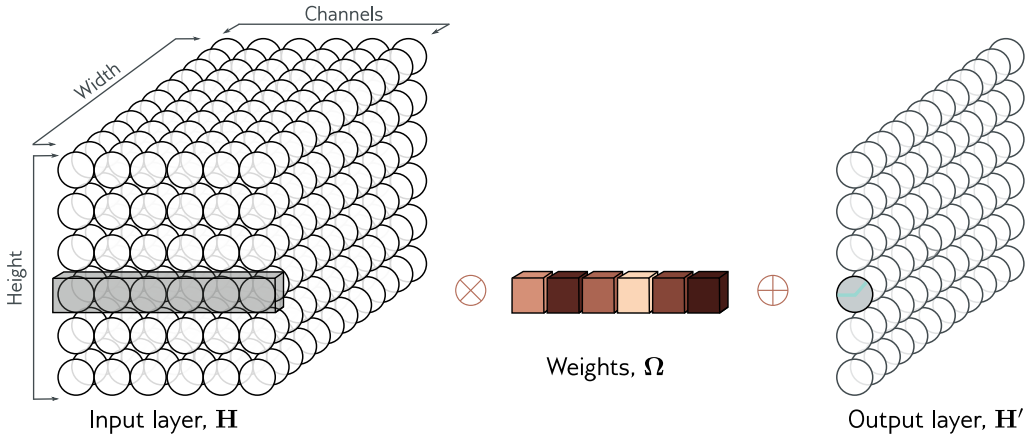

Figure 10.14 1×1 convolution. To change the number of channels without spatial pooling, we apply a 1×1 kernel. Each output channel is computed by taking a weighted sum of all of the channels at the same position, adding a bias, and passing through an activation function. Multiple output channels are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

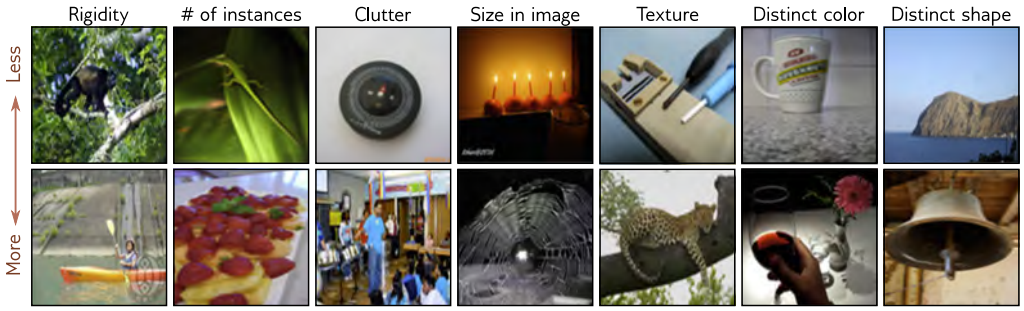

Figure 10.15 Example ImageNet classification images. The model aims to assign an input image to one of 1000 classes. This task is challenging because the images vary widely along different attributes (columns). These include rigidity (monkey < canoe), number of instances in image (lizard < strawberry), 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).

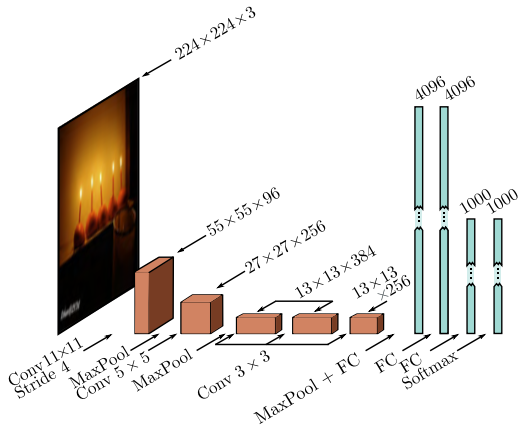

Figure 10.16 AlexNet (Krizhevsky et al., 2012). The network maps a 224×224 color image to a 1000-dimensional vector representing class probabilities. The network first convolves with 11×11 kernels and stride 4 to create 96 channels. It decreases the resolution again using a max pool operation and applies 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.

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.

Problems 10.16–10.17

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.

Notebook 10.5 Convolution for MNIST

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.

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.

Problem 10.18

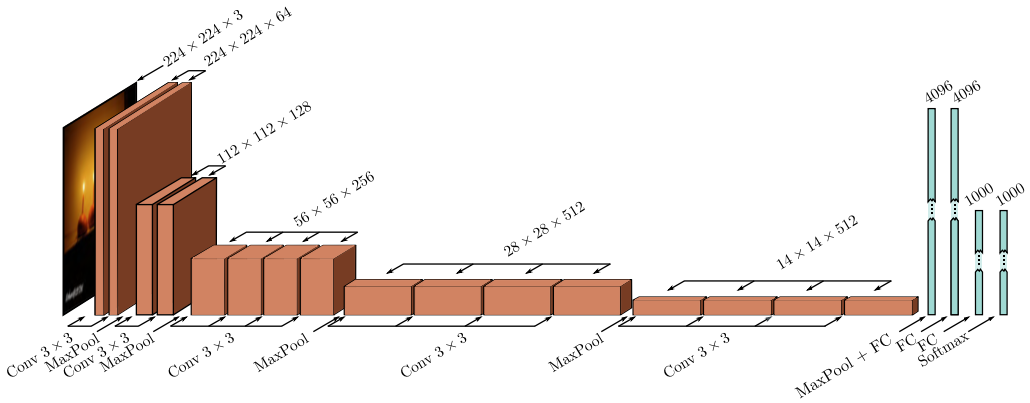

Figure 10.17 VGG network (Simonyan & Zisserman, 2014) depicted at the same scale as AlexNet (see figure 10.16). This network consists of a series of convolutional layers and max pooling operations, in which the spatial scale of the representation gradually decreases, but the number of channels gradually 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.

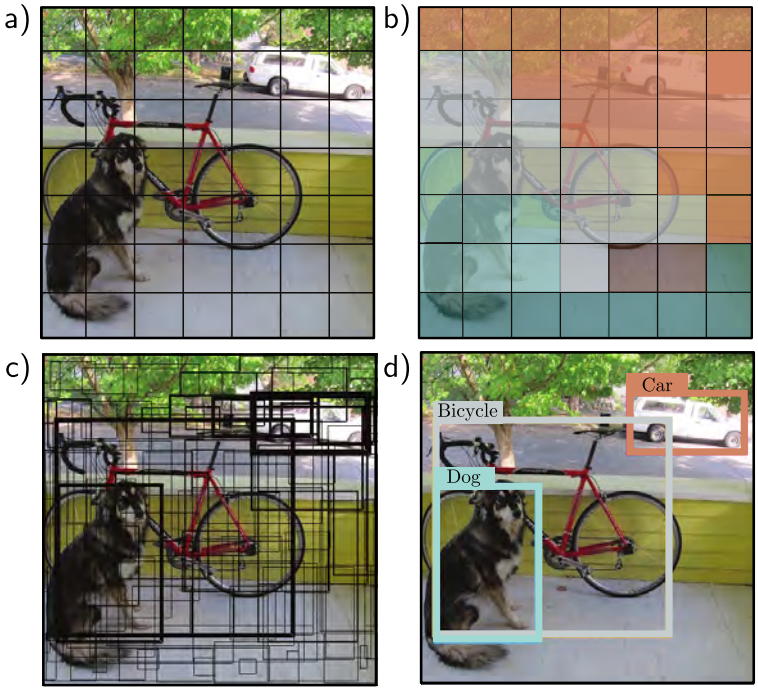

Figure 10.18 YOLO object detection. a) The input image is reshaped to 448×448 and divided into a regular 7×7 grid. b) The system predicts the most likely class at each grid cell. c) It also predicts two bounding boxes per cell, and a confidence value (represented by thickness of line). d) During inference, 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

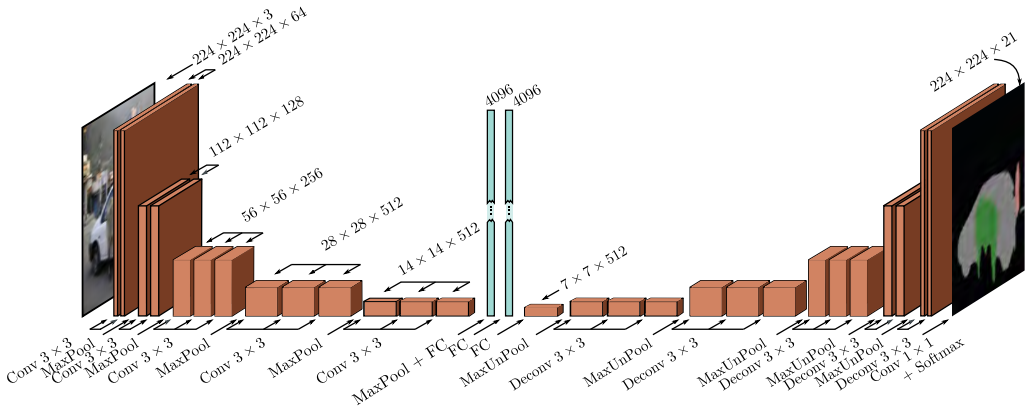

Figure 10.19 Semantic segmentation network of Noh et al. (2015). The input is a 224×224 image, which is passed through a version of the VGG network and eventually transformed into a representation of size 4096 using a fully connected layer. This contains information about the entire image. This is then 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

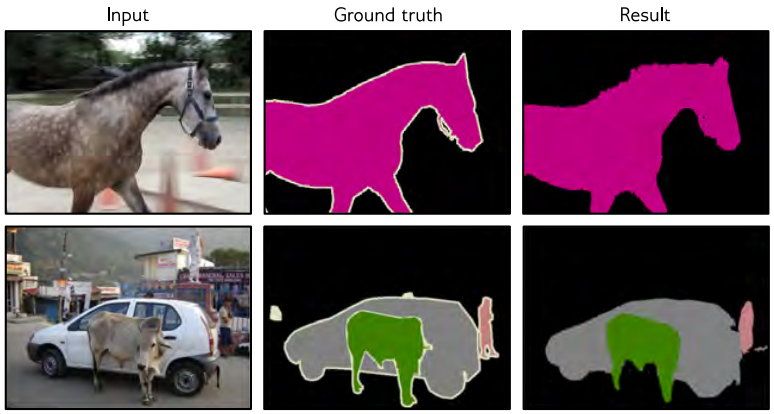

Figure 10.20 Semantic segmentation results. The final result is created from the 21 probability maps by greedily selecting the best class and using a heuristic method to find a sensible binary map based on the probabilities and their spatial proximity. If there is enough evidence, subsequent classes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

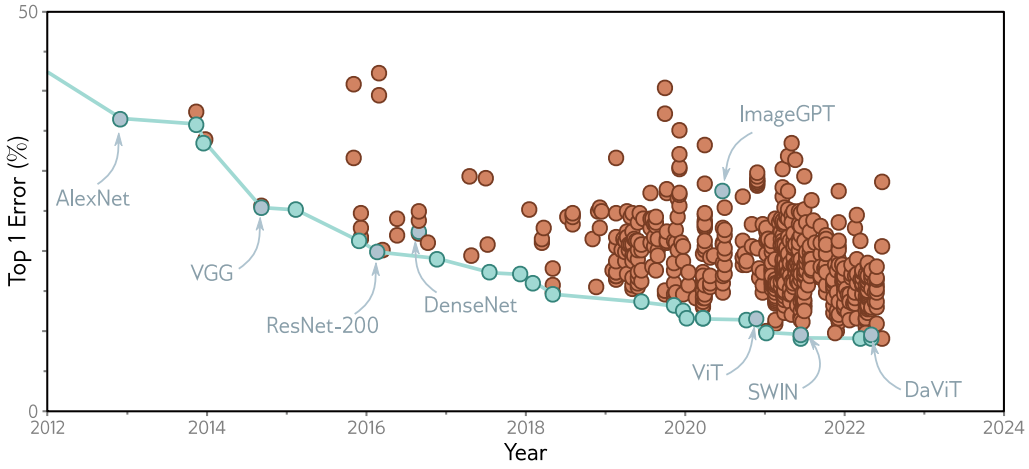

Figure 10.21 ImageNet performance. Each circle represents a different published model. Blue circles represent models that were state-of-the-art. Models discussed in this book are also highlighted. The AlexNet and VGG networks were remarkable for their time but are now far from state of the art. ResNet-200 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 (Xiao et al., 2018a). 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. You may assume that the input is padded with zeros at positions x_{-2}, x_{-1} and x_0.

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 6×12 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 12×6 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 and stride two is equivalent to composing the matrices for 1D convolution with kernel size 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.

Chapter 11

Residual networks

The previous chapter described how image classification performance improved as the depth of convolutional networks was extended from eight layers (AlexNet) to eighteen layers (VGG). This led to experimentation with even deeper networks. However, performance decreased again when many more layers were added.

This chapter introduces *residual blocks*. Here, each network layer computes an additive change to the current representation instead of transforming it directly. This allows deeper networks to be trained but causes an exponential increase in the activation magnitudes at initialization. Residual blocks employ *batch normalization* to compensate for this, which re-centers and rescales the activations at each layer.

Residual blocks with batch normalization allow much deeper networks to be trained, and these networks improve performance across a variety of tasks. Architectures that combine residual blocks to tackle image classification, medical image segmentation, and human pose estimation are described.

11.1 Sequential processing

Every network we have seen so far processes the data sequentially; each layer receives the previous layer's output and passes the result to the next (figure 11.1). For example, a three-layer network is defined by:

$$\begin{aligned}\mathbf{h}_1 &= \mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1] \\ \mathbf{h}_2 &= \mathbf{f}_2[\mathbf{h}_1, \boldsymbol{\phi}_2] \\ \mathbf{h}_3 &= \mathbf{f}_3[\mathbf{h}_2, \boldsymbol{\phi}_3] \\ \mathbf{y} &= \mathbf{f}_4[\mathbf{h}_3, \boldsymbol{\phi}_4],\end{aligned} \tag{11.1}$$

where $\mathbf{h}_1$, $\mathbf{h}_2$, and $\mathbf{h}_3$ denote the intermediate hidden layers, $\mathbf{x}$ is the network input, $\mathbf{y}$ is the output, and the functions $\mathbf{f}_k[\bullet, \boldsymbol{\phi}_k]$ perform the processing.

In a standard neural network, each layer consists of a linear transformation followed by an activation function, and the parameters $\boldsymbol{\phi}_k$ comprise the weights and biases of the

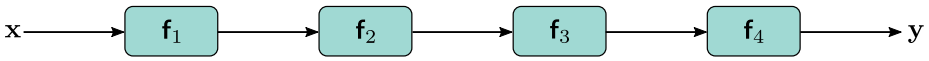

Figure 11.1 Sequential processing. Standard neural networks pass the output of each layer directly into the next layer.

linear transformation. In a convolutional network, each layer consists of a set of convolutions followed by an activation function, and the parameters comprise the convolutional kernels and biases.

Since the processing is sequential, we can equivalently think of this network as a series of nested functions:

$$\mathbf{y} = \mathbf{f}_4\Big[\mathbf{f}_3\Big[\mathbf{f}_2\big[\mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1], \boldsymbol{\phi}_2\big], \boldsymbol{\phi}_3\Big], \boldsymbol{\phi}_4\Big]. \tag{11.2}$$

11.1.1 Limitations of sequential processing

In principle, we can add as many layers as we want, and in the previous chapter, we saw that adding more layers to a convolutional network does improve performance; the VGG network (figure 10.17), which has eighteen layers, outperforms AlexNet (figure 10.16), which has eight layers. However, image classification performance decreases again as further layers are added (figure 11.2). This is surprising since models generally perform better as more capacity is added (figure 8.10). Indeed, the decrease is present for both the training set and the test set, which implies that the problem is training deeper networks rather than the inability of deeper networks to generalize.

This phenomenon is not completely understood. One conjecture is that at initialization, the loss gradients change unpredictably when we modify parameters in early network layers. With appropriate initialization of the weights (see section 7.5), the gradient of the loss with respect to these parameters will be reasonable (i.e., no exploding or vanishing gradients). However, the derivative assumes an infinitesimal change in the parameter, whereas optimization algorithms use a finite step size. Any reasonable choice of step size may move to a place with a completely different and unrelated gradient; the loss surface looks like an enormous range of tiny mountains rather than a single smooth structure that is easy to descend. Consequently, the algorithm doesn't make progress in the way that it does when the loss function gradient changes more slowly.

Notebook 11.1 Shattered gradients

This conjecture is supported by empirical observations of gradients in networks with a single input and output. For a shallow network, the gradient of the output with respect to the input changes slowly as we change the input (figure 11.3a). However, for a deep network, a tiny change in the input results in a completely different gradient (figure 11.3b). This is captured by the autocorrelation function of the gradient (figure 11.3c). Nearby gradients are correlated for shallow networks, but this correlation quickly drops to zero for deep networks. This is termed the *shattered gradients* phenomenon.

Appendix B.2.1 Autocorrelation function

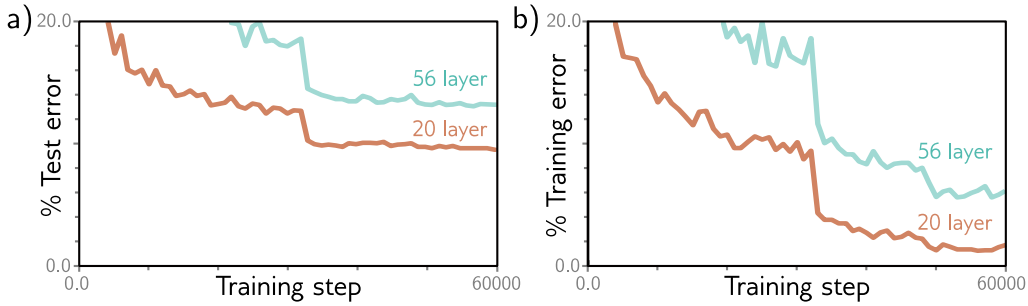

Figure 11.2 Decrease in performance when adding more convolutional layers. a) A 20-layer convolutional network outperforms a 56-layer neural network for image classification on the test set of the CIFAR-10 dataset (Krizhevsky & Hinton, 2009). b) This is also true for the training set, which suggests that the problem relates to training the original network rather than a failure to generalize to new data. Adapted from He et al. (2016a).

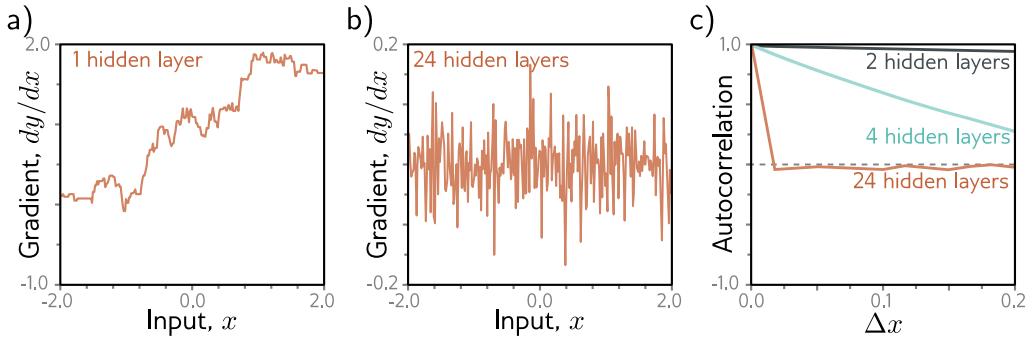

Figure 11.3 Shattered gradients. a) Consider a shallow network with 200 hidden units and Glorot initialization (He initialization without the factor of two) for both the weights and biases. The gradient $\partial y/\partial x$ of the scalar network output y with respect to the scalar input x changes relatively slowly as we change the input x. b) For a deep network with 24 layers and 200 hidden units per layer, this gradient changes very quickly and unpredictably. c) The autocorrelation function of the gradient shows that nearby gradients become unrelated (have autocorrelation close to zero) for deep networks. This *shattered gradients* phenomenon may explain why it is hard to train deep networks. Gradient descent algorithms rely on the loss surface being relatively smooth, so the gradients should be related before and after each update step. Adapted from Balduzzi et al. (2017).

Shattered gradients presumably arise because changes in early network layers modify the output in an increasingly complex way as the network becomes deeper. The derivative of the output $\mathbf{y}$ with respect to the first layer $\mathbf{f}_1$ of the network in equation 11.1 is:

Appendix B.5 Matrix calculus

$$\frac{\partial \mathbf{y}}{\partial \mathbf{f}_1} = \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1}\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_3}. \tag{11.3}$$

When we change the parameters that determine $\mathbf{f}_1$, *all* of the derivatives in this sequence can change since layers $\mathbf{f}_2, \mathbf{f}_3$, and $\mathbf{f}_4$ are themselves computed from $\mathbf{f}_1$. Consequently, the updated gradient at each training example may be completely different, and the loss function becomes badly behaved.[1]

11.2 Residual connections and residual blocks

Residual or *skip connections* are branches in the computational path, whereby the input to each network layer $\mathbf{f}[\bullet]$ is added back to the output (figure 11.4a). By analogy to equation 11.1, the residual network is defined as:

$$\begin{aligned}
\mathbf{h}_1 &= \mathbf{x} + \mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1] \\
\mathbf{h}_2 &= \mathbf{h}_1 + \mathbf{f}_2[\mathbf{h}_1, \boldsymbol{\phi}_2] \\
\mathbf{h}_3 &= \mathbf{h}_2 + \mathbf{f}_3[\mathbf{h}_2, \boldsymbol{\phi}_3] \\
\mathbf{y} &= \mathbf{h}_3 + \mathbf{f}_4[\mathbf{h}_3, \boldsymbol{\phi}_4],
\end{aligned} \tag{11.4}$$

where the first term on the right-hand side of each line is the residual connection. Each function $\mathbf{f}_k$ learns an additive change to the current representation. It follows that their outputs must be the same size as their inputs. Each additive combination of the input and the processed output is known as a *residual block* or *residual layer*.

Once more, we can write this as a single function by substituting in the expressions for the intermediate quantities $\mathbf{h}_k$:

Problem 11.1

$$\begin{aligned}
\mathbf{y} = \mathbf{x} &+ \mathbf{f}_1[\mathbf{x}] \\
&+ \mathbf{f}_2\big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\big] \\
&+ \mathbf{f}_3\Big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\big]\Big] \\
&+ \mathbf{f}_4\Big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\big] + \mathbf{f}_3\Big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\big]\Big]\Big],
\end{aligned} \tag{11.5}$$

where we have omitted the parameters $\boldsymbol{\phi}_\bullet$ for clarity. We can think of this equation as "unraveling" the network (figure 11.4b). We see that the final network output is a sum of the input and four smaller networks, corresponding to each line of the equation; one

[1] In equations 11.3 and 11.6, we overload notation to define $\mathbf{f}_k$ as the output of the function $\mathbf{f}_k[\bullet]$.

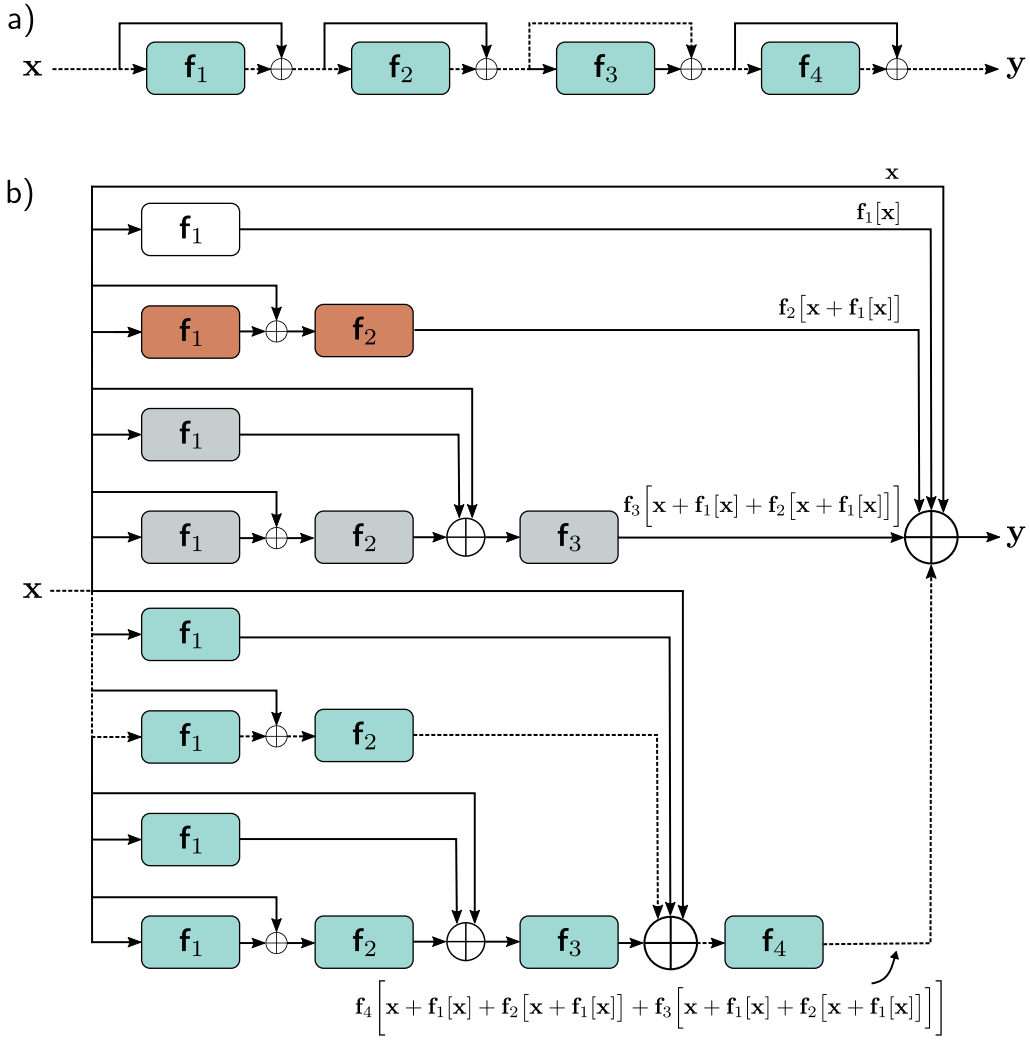

Figure 11.4 Residual connections. a) The output of each function $\mathbf{f}_k[\mathbf{x}, \boldsymbol{\phi}_k]$ is added back to its input, which is passed via a parallel computational path called a residual or skip connection. Hence, the function computes an additive change to the representation. b) Upon expanding (unraveling) the network equations, we find that the output is the sum of the input plus four smaller networks (depicted in white, orange, gray, and cyan, respectively, and corresponding to terms in equation 11.5); we can think of this as an ensemble of networks. Moreover, the output from the cyan network is itself a transformation $\mathbf{f}_4[\bullet, \boldsymbol{\phi}_4]$ of another ensemble, and so on. Alternatively, we can consider the network as a combination of 16 different paths through the computational graph. One example is the dashed path from input $\mathbf{x}$ to output $\mathbf{y}$, which is the same in panels (a) and (b).

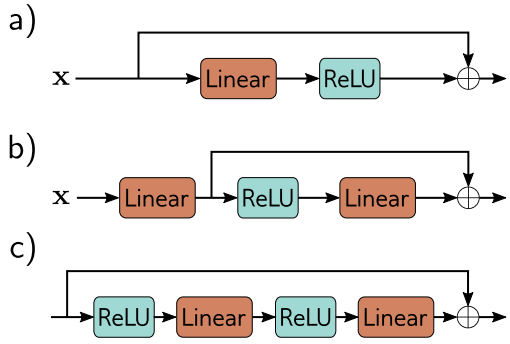

Figure 11.5 Order of operations in residual blocks. a) The usual order of linear transformation or convolution followed by a ReLU nonlinearity means that each residual block can only add non-negative quantities. b) With the reverse order, both positive and negative quantities can be added. However, we must add a linear transformation at the start of the network in case the input is all negative. c) In practice, it's common for a residual block to contain several network layers.

interpretation is that residual connections turn the original network into an ensemble of these smaller networks whose outputs are summed to compute the result.

A complementary way of thinking about this residual network is that it creates sixteen paths of different lengths from input to output. For example, the first function $\mathbf{f}_1[\mathbf{x}]$ occurs in eight of these sixteen paths, including as a direct additive term (i.e., a path length of one), and the analogous derivative to equation 11.3 is:

Problem 11.2

Problem 11.3

$$\frac{\partial \mathbf{y}}{\partial \mathbf{f}_1} = \mathbf{I} + \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1} + \left(\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1}\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_2}\right) + \left(\frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1}\frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_2} + \frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_1}\frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_3} + \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1}\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_3}\right), \quad (11.6)$$

where there is one term for each of the eight paths. The identity term on the right-hand side shows that changes in the parameters $\boldsymbol{\phi}_1$ in the first layer $\mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1]$ contribute directly to changes in the network output $\mathbf{y}$. They also contribute indirectly through the other chains of derivatives of varying lengths. In general, gradients through shorter paths will be better behaved. Since both the identity term and various short chains of derivatives will contribute to the derivative for each layer, networks with residual links suffer less from shattered gradients.

Notebook 11.2 Residual networks

11.2.1 Order of operations in residual blocks

Until now, we have implied that the additive functions $\mathbf{f}[\mathbf{x}]$ could be any valid network layer (e.g., fully connected or convolutional). This is technically true, but the order of operations in these functions is important. They must contain a nonlinear activation function like a ReLU, or the entire network will be linear. However, in a typical network layer (figure 11.5a), the ReLU function is at the end, so the output is non-negative. If we adopt this convention, then each residual block can only increase the input values.

Hence, it is typical to change the order of operations so that the activation function is applied first, followed by the linear transformation (figure 11.5b). Sometimes there may be several layers of processing within the residual block (figure 11.5c), but these usually terminate with a linear transformation. Finally, we note that when we start these blocks with a ReLU operation, they will do nothing if the initial network input is negative since the ReLU will clip the entire signal to zero. Hence, it's typical to start the network with a linear transformation rather than a residual block, as in figure 11.5b.

11.2.2 Deeper networks with residual connections

Adding residual connections roughly doubles the depth of a network that can be practically trained before performance degrades. However, we would like to increase the depth further. To understand why residual connections do not allow us to increase the depth arbitrarily, we must consider how the variance of the activations changes during the forward pass and how the gradient magnitudes change during the backward pass.

11.3 Exploding gradients in residual networks

In section 7.5, we saw that initializing the network parameters is critical. Without careful initialization, the magnitudes of the intermediate values during the forward pass of backpropagation can increase or decrease exponentially. Similarly, the gradients during the backward pass can explode or vanish as we move backward through the network.

Hence, we initialize the network parameters so that the expected variance of the activations (in the forward pass) and gradients (in the backward pass) remains the same between layers. He initialization (section 7.5) achieves this for ReLU activations by initializing the biases $\boldsymbol{\beta}$ to zero and choosing normally distributed weights $\boldsymbol{\Omega}$ with mean zero and variance $2/D_h$ where D_h is the number of hidden units in the previous layer.

Now consider a residual network. We do not have to worry about the intermediate values or gradients vanishing with network depth since there exists a path whereby each layer directly contributes to the network output (equation 11.5 and figure 11.4b). However, even if we use He initialization within the residual block, the values in the forward pass increase exponentially as we move through the network.

To see why, consider that we add the result of the processing in the residual block back to the input. Each branch has some (uncorrelated) variability. Hence, the overall variance increases when we recombine them. With ReLU activations and He initialization, the expected variance is unchanged by the processing in each block. Consequently, when we recombine with the input, the variance doubles (figure 11.6a), growing exponentially with the number of residual blocks. This limits the possible network depth before floating point precision is exceeded in the forward pass. A similar argument applies to the gradients in the backward pass of the backpropagation algorithm.

Problem 11.4

Hence, residual networks still suffer from unstable forward propagation and exploding gradients even with He initialization. One approach that would stabilize the forward and backward passes would be to use He initialization and then multiply the combined output of each residual block by $1/\sqrt{2}$ to compensate for the doubling (figure 11.6b). However, it is more usual to use *batch normalization*.

11.4 Batch normalization

Batch normalization or *BatchNorm* shifts and rescales each activation h so that its mean and variance across the batch $\mathcal{B}$ become values that are learned during training. First, the empirical mean m_h and standard deviation s_h are computed:

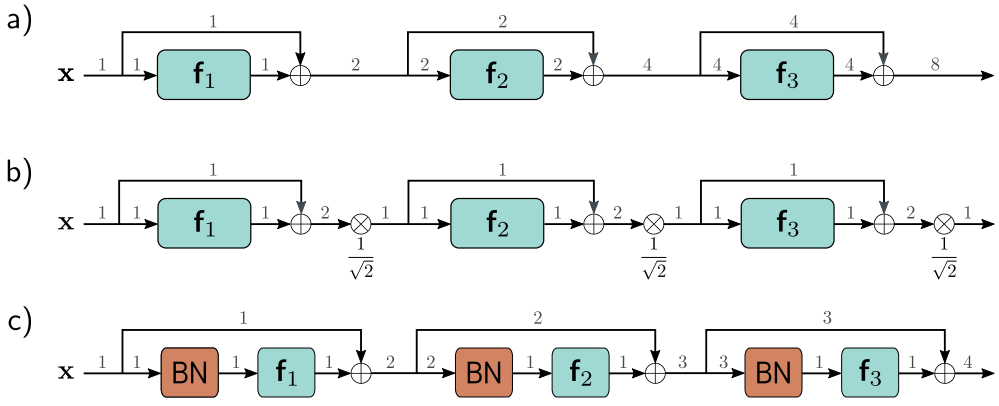

Figure 11.6 Variance in residual networks. a) He initialization ensures that the expected variance remains unchanged after a linear plus ReLU layer $\mathbf{f}_k$. Unfortunately, in residual networks, the input of each block is added back to the output, so the variance doubles at each layer (gray numbers indicate variance) and grows exponentially. b) One approach would be to rescale the signal by $1/\sqrt{2}$ between each residual block. c) A second method uses batch normalization (BN) as the first step in the residual block and initializes the associated offset δ to zero and scale γ to one. This transforms the input to each layer to have unit variance, and with He initialization, the output variance will also be one. Now the variance increases linearly with the number of residual blocks. A side-effect is that, at initialization, later network layers are dominated by the residual connection and are hence close to computing the identity.

$$\begin{aligned} m_h &= \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} h_i \\ s_h &= \sqrt{\frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}(h_i - m_h)^2}, \end{aligned} \tag{11.7}$$

where all quantities are scalars. Then we use these statistics to standardize the batch activations to have mean zero and unit variance:

Appendix C.2.4 Standardization

$$h_i \leftarrow \frac{h_i - m_h}{s_h + \epsilon} \qquad \forall i \in \mathcal{B}, \tag{11.8}$$

where ϵ is a small number that prevents division by zero if h_i is the same for every member of the batch and $s_h = 0$.

Finally, the normalized variable is scaled by γ and shifted by δ:

$$h_i \leftarrow \gamma h_i + \delta \qquad \forall i \in \mathcal{B}. \tag{11.9}$$

Problem 11.5

After this operation, the activations have mean δ and standard deviation γ across all members of the batch. Both of these quantities are learned during training.

Problem 11.6

Batch normalization is applied independently to each hidden unit. In a standard neural network with K layers, each containing D hidden units, there would be KD learned offsets δ and KD learned scales γ. In a convolutional network, the normalizing statistics are computed over both the batch and the spatial position. If there were K layers, each containing C channels, there would be KC offsets and KC scales. At test time, we do not have a batch from which we can gather statistics. To resolve this, the statistics m_h and s_h are calculated across the whole training dataset (rather than just a batch) and frozen in the final network.

Notebook 11.3 BatchNorm

11.4.1 Costs and benefits of batch normalization

Batch normalization makes the network invariant to rescaling the weights and biases that contribute to each activation; if these are doubled, then the activations also double, the estimated standard deviation s_h doubles, and the normalization in equation 11.8 compensates for these changes. This happens separately for each hidden unit. Consequently, there will be a large family of weights and biases that all produce the same effect. Batch normalization also adds two parameters, γ and δ, at every hidden unit, which makes the model somewhat larger. Hence, it both creates redundancy in the weight parameters and adds extra parameters to compensate for that redundancy. This is obviously inefficient, but batch normalization also provides several benefits.

Stable forward propagation: If we initialize the offsets δ to zero and the scales γ to one, then each output activation will have unit variance. In a regular network, this ensures the variance is stable during forward propagation at initialization. In a residual network, the variance must still increase as we add a new source of variation to the input at each layer. However, it will increase linearly with each residual block; the k^{th} layer adds one unit of variance to the existing variance of k (figure 11.6c).

At initialization, this has the side-effect that later layers make a smaller change to the overall variation than earlier ones. The network is effectively less deep at the start of training since later layers are close to computing the identity. As training proceeds, the network can increase the scales γ in later layers and can control its own effective depth.

Higher learning rates: Empirical studies and theory both show that batch normalization makes the loss surface and its gradient change more smoothly (i.e., reduces shattered gradients). This means we can use higher learning rates as the surface is more predictable. We saw in section 9.2 that higher learning rates improve test performance.

Regularization: We also saw in chapter 9 that adding noise to the training process can improve generalization. Batch normalization injects noise because the normalization depends on the batch statistics. The activations for a given training example are normalized by an amount that depends on the other members of the batch and will be slightly different at each training iteration.

11.5 Common residual architectures

Residual connections are now a standard part of deep learning pipelines. This section reviews some well-known architectures that incorporate them.

11.5.1 ResNet

Residual blocks were first used in convolutional networks for image classification. The resulting networks are known as residual networks, or *ResNets* for short. In ResNets, each residual block contains a batch normalization operation, a ReLU activation function, and a convolutional layer. This is followed by the same sequence again before being added back to the input (figure 11.7a). Trial and error have shown that this order of operations works well for image classification. Problem 11.7

For very deep networks, the number of parameters may become undesirably large. *Bottleneck residual blocks* make more efficient use of parameters using three convolutions. The first has a 1×1 kernel and reduces the number of channels. The second is a regular 3×3 kernel, and the third is another 1×1 kernel to increase the number of channels back to the original amount (figure 11.7b). In this way, we can integrate information over a 3×3 pixel area using fewer parameters. Problem 11.8

The ResNet-200 model (figure 11.8) contains 200 layers and was used for image classification on the ImageNet database (figure 10.15). The architecture resembles AlexNet and VGG but uses bottleneck residual blocks instead of vanilla convolutional layers. As with AlexNet and VGG, these are periodically interspersed with decreases in spatial resolution and simultaneous increases in the number of channels. Here, the resolution is decreased by downsampling using convolutions with stride two. The number of channels is increased either by appending zeros to the representation or by using an extra 1×1 convolution. At the start of the network is a 7×7 convolutional layer, followed by a downsampling operation. At the end, a fully connected layer maps the block to a vector of length 1000. This is passed through a softmax layer to generate class probabilities.

The ResNet-200 model achieved a remarkable 4.8% error rate for the correct class being in the top five and 20.1% for identifying the correct class correctly. This compared favorably with AlexNet (16.4%, 38.1%) and VGG (6.8%, 23.7%) and was one of the first networks to exceed human performance (5.1% for being in the top five guesses). However, this model was conceived in 2016 and is far from state-of-the-art. At the time of writing, the best-performing model on this task has a 9.0% error for identifying the class correctly (see figure 10.21). This and all the other current top-performing models for image classification are now based on transformers (see chapter 12).

11.5.2 DenseNet

Residual blocks receive the output from the previous layer, modify it by passing it through some network layers, and add it back to the original input. An alternative is to concatenate the modified and original signals. This increases the representation size

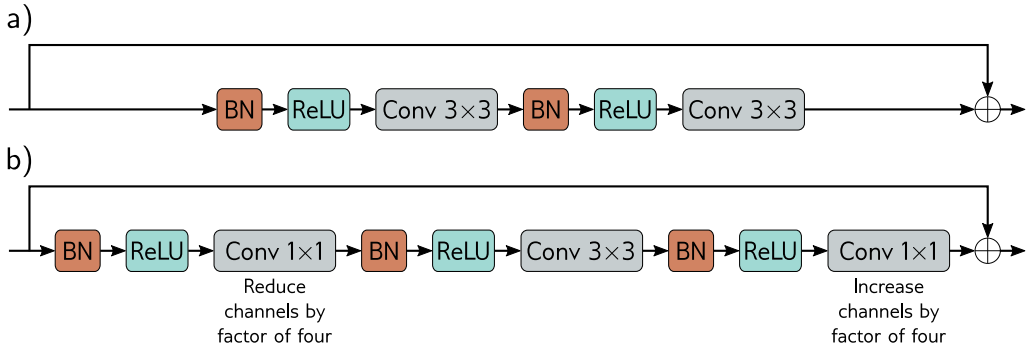

Figure 11.7 ResNet blocks. a) A standard block in the ResNet architecture contains a batch normalization operation, followed by an activation function, and a 3×3 convolutional layer. Then, this sequence is repeated. b). A bottleneck ResNet block still integrates information over a 3×3 region but uses fewer parameters. It contains three convolutions. The first 1×1 convolution reduces the number of channels. The second 3×3 convolution is applied to the smaller representation. A final 1×1 convolution increases the number of channels again so that it can be added back to the input.

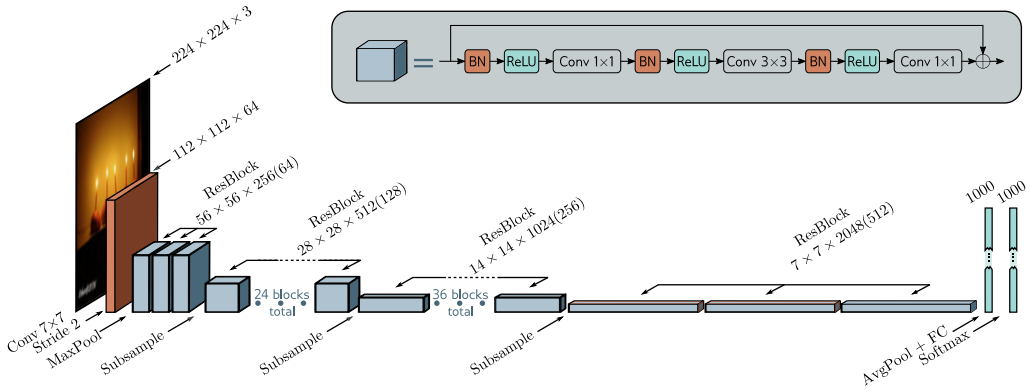

Figure 11.8 ResNet-200 model. A standard 7×7 convolutional layer with stride two is applied, followed by a MaxPool operation. A series of bottleneck residual blocks follow (number in brackets is channels after first 1×1 convolution), with periodic downsampling and accompanying increases in the number of channels. The network concludes with average pooling across all spatial positions and a fully connected layer that maps to pre-softmax activations.

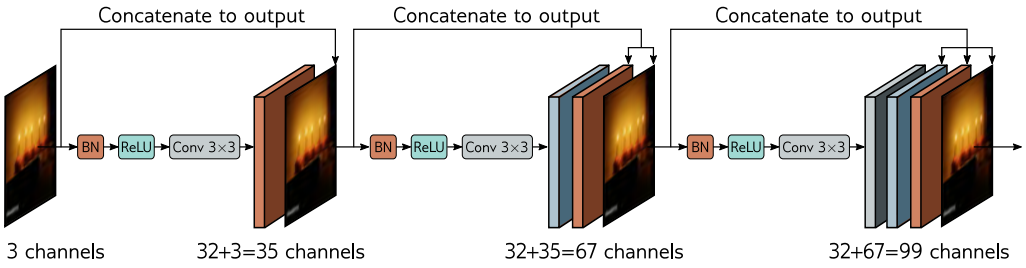

Figure 11.9 DenseNet. This architecture uses residual connections to concatenate the outputs of earlier layers to later ones. Here, the three-channel input image is processed to form a 32-channel representation. The input image is concatenated to this to give a total of 35 channels. This combined representation is processed to create another 32-channel representation, and both earlier representations are concatenated to this to create a total of 67 channels and so on.

(in terms of channels for a convolutional network), but an optional subsequent linear transformation can map back to the original size (a 1×1 convolution for a convolutional network). This allows the model to add the representations together, take a weighted sum, or combine them in a more complex way.

The DenseNet architecture uses concatenation so that the input to a layer comprises the concatenated outputs from *all* previous layers (figure 11.9). These are processed to create a new representation that is itself concatenated with the previous representation and passed to the next layer. This concatenation means there is a direct contribution from earlier layers to the output, so the loss surface behaves reasonably.

In practice, this can only be sustained for a few layers because the number of channels (and hence the number of parameters required to process them) becomes increasingly large. This problem can be alleviated by applying a 1×1 convolution to reduce the number of channels before the next 3×3 convolution is applied. In a convolutional network, the input is periodically downsampled. Concatenation across the downsampling makes no sense since the representations have different sizes. Consequently, the chain of concatenation is broken at this point, and a smaller representation starts a new chain. In addition, another bottleneck 1×1 convolution can be applied when the downsampling occurs to control the representation size further.

This network performs competitively with ResNet models on image classification (see figure 10.21); indeed, it can perform better for a comparable parameter count. This is presumably because it can reuse processing from earlier layers more flexibly.

11.5.3 U-Nets and hourglass networks

Section 10.5.3 described a semantic segmentation network that had an encoder-decoder or hourglass structure. The encoder repeatedly downsamples the image until the receptive fields are large and information is integrated from across the image. Then the decoder

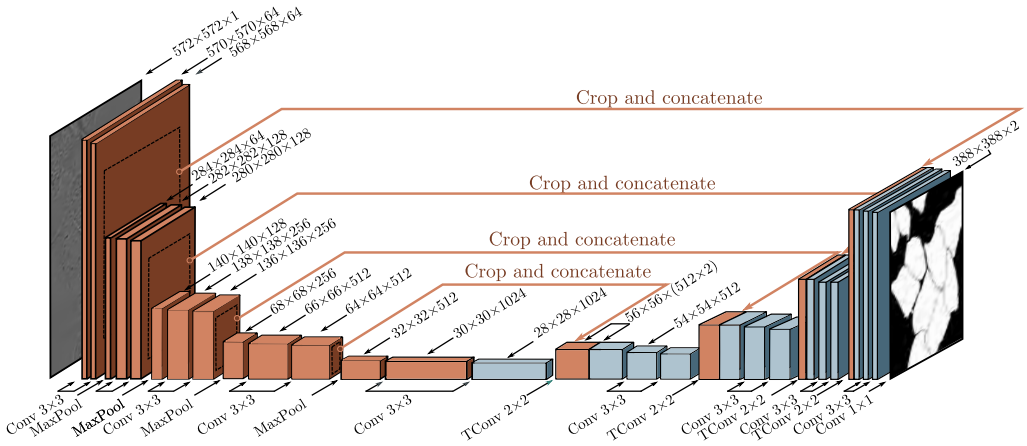

Figure 11.10 U-Net for segmenting HeLa cells. The U-Net has an encoder-decoder structure, in which the representation is downsampled (orange blocks) and then re-upsampled (blue blocks). The encoder uses regular convolutions, and the decoder uses transposed convolutions. Residual connections append the last representation at each scale in the encoder to the first representation at the same scale in the decoder (orange arrows). The original U-Net used "valid" convolutions, so the size decreased slightly with each layer, even without downsampling. Hence, the representations from the encoder were cropped (dashed squares) before appending to the decoder. Adapted from Ronneberger et al. (2015).

upsamples it back to the size of the original image. The final output is a probability over possible object classes at each pixel. One drawback of this architecture is that the low-resolution representation in the middle of the network must "remember" the high-resolution details to make the final result accurate. This is unnecessary if residual connections transfer the representations from the encoder to their partner in the decoder.

The *U-Net* (figure 11.10) is an encoder-decoder architecture where the earlier representations are concatenated to the later ones. The original implementation used "valid" convolutions, so the spatial size decreases by two pixels each time a 3×3 convolutional layer is applied. This means that the upsampled version is smaller than its counterpart in the encoder, which must be cropped before concatenation. Subsequent implementations have used zero padding, where this cropping is unnecessary. Note that the U-Net is completely convolutional, so after training, it can be run on an image of *any size*.

Problem 11.9

The U-Net was intended for segmenting medical images (figure 11.11) but has found many other uses in computer graphics and vision. *Hourglass networks* are similar but apply further convolutional layers in the skip connections and add the result back to the decoder rather than concatenating it. A series of these models form a *stacked hourglass network* that alternates between considering the image at local and global levels. Such networks are used for pose estimation (figure 11.12). The system is trained to predict one "heatmap" for each joint, and the estimated position is the maximum of each heatmap.

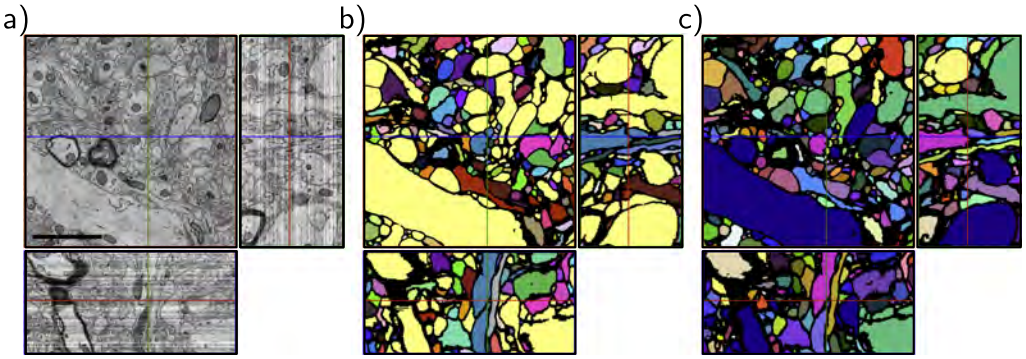

Figure 11.11 Segmentation using U-Net in 3D. a) Three slices through a 3D volume of mouse cortex taken by scanning electron microscope. b) A single U-Net is used to classify voxels as being inside or outside neurites. Connected regions are identified with different colors. c) For a better result, an ensemble of five U-Nets is trained, and a voxel is only classified as belonging to the cell if all five networks agree. Adapted from Falk et al. (2019).

11.6 Why do nets with residual connections perform so well?

Residual networks allow much deeper networks to be trained; it's possible to extend the ResNet architecture to 1000 layers and still train effectively. The improvement in image classification performance was initially attributed to the additional network depth, but two pieces of evidence contradict this viewpoint.

First, shallower, wider residual networks sometimes outperform deeper, narrower ones with a comparable parameter count. In other words, better performance can sometimes be achieved with a network with fewer layers but more channels per layer. Second, there is evidence that the gradients during training do not propagate effectively through very long paths in the unraveled network (figure 11.4b). In effect, a very deep network may act more like a combination of shallower networks.

The current view is that residual connections add some value of their own, as well as allowing deeper networks to be trained. This perspective is supported by the fact that the loss surfaces of residual networks around a minimum tend to be smoother and more predictable than those for the same network when the skip connections are removed (figure 11.13). This may make it easier to learn a good solution that generalizes well.

11.7 Summary

Increasing network depth indefinitely causes both training and test performance for image classification to decrease. This may be because the gradient of the loss with respect to

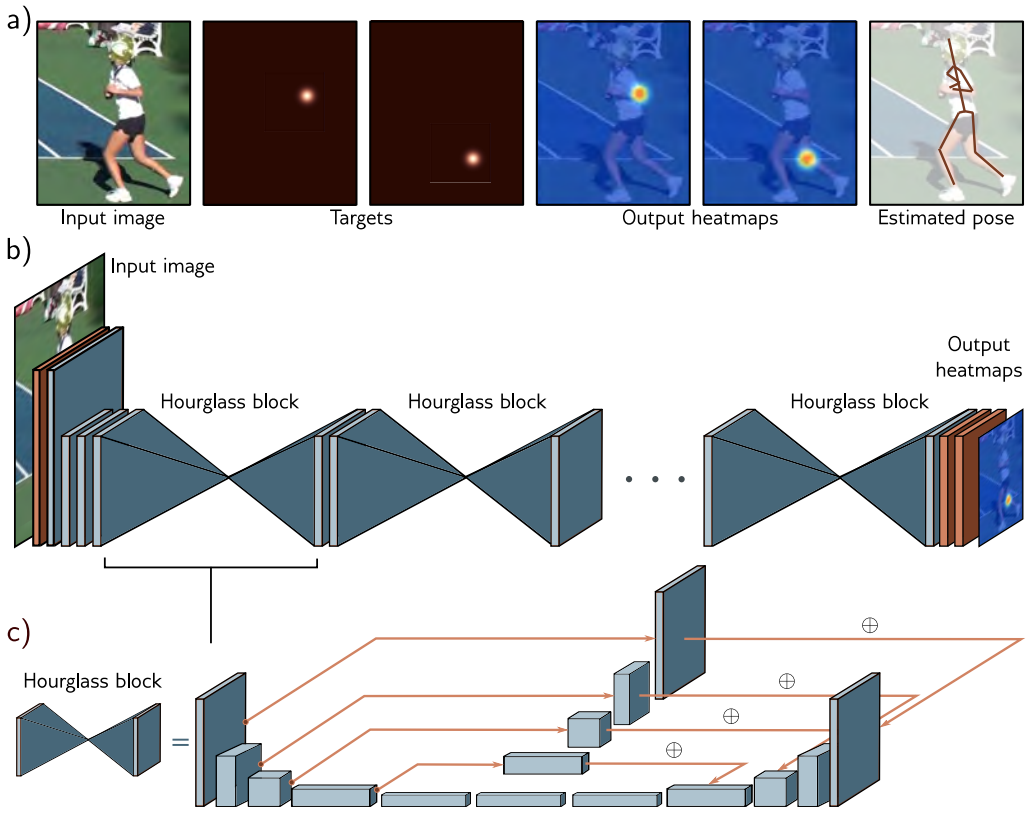

Figure 11.12 Stacked hourglass networks for pose estimation. a) The network input is an image containing a person, and the output is a set of heatmaps, with one heatmap for each joint. This is formulated as a regression problem where the targets are heatmap images with small, highlighted regions at the ground-truth joint positions. The peak of the estimated heatmap is used to establish each final joint position. b) The architecture consists of initial convolutional and residual layers followed by a series of hourglass blocks. c) Each hourglass block consists of an encoder-decoder network similar to the U-Net except that the convolutions use zero padding, some further processing is done in the residual links, and these links add this processed representation rather than concatenate it. Each blue cuboid is itself a bottleneck residual block (figure 11.7b). Adapted from Newell et al. (2016).

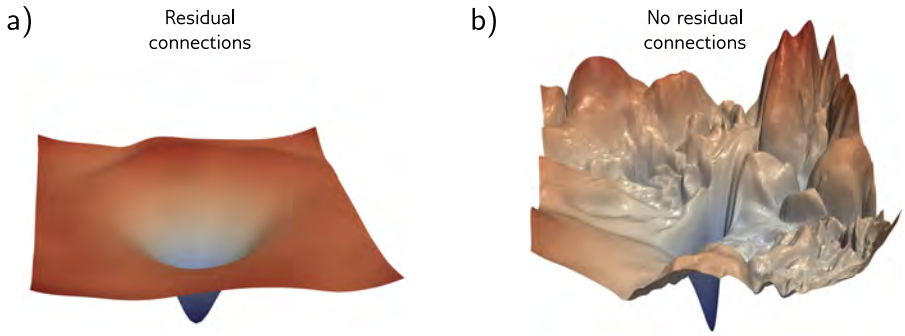

Figure 11.13 Visualizing neural network loss surfaces. Each plot shows the loss surface in two random directions in parameter space around the minimum found by SGD for an image classification task on the CIFAR-10 dataset. These directions are normalized to facilitate side-by-side comparison. a) Residual net with 56 layers. b) Results from the same network without skip connections. The surface is smoother with the skip connections. This facilitates learning and makes the final network performance more robust to minor errors in the parameters, so it will likely generalize better. Adapted from Li et al. (2018b).

parameters early in the network changes quickly and unpredictably relative to the update step size. Residual connections add the processed representation back to their own input. Now each layer contributes directly to the output as well as indirectly, so propagating gradients through many layers is not mandatory, and the loss surface is smoother.

Residual networks don't suffer from vanishing gradients but introduce an exponential increase in the variance of the activations during forward propagation and corresponding problems with exploding gradients. This is usually handled by adding batch normalization, which compensates for the empirical mean and variance of the batch and then shifts and rescales using learned parameters. If these parameters are initialized judiciously, very deep networks can be trained. There is evidence that both residual links and batch normalization make the loss surface smoother, which permits larger learning rates. Moreover, the variability in the batch statistics adds a source of regularization.

Residual blocks have been incorporated into convolutional networks. They allow deeper networks to be trained with commensurate increases in image classification performance. Variations of residual networks include the DenseNet architecture, which concatenates outputs of all prior layers to feed into the current layer, and U-Nets, which incorporate residual connections into encoder-decoder models.

Notes

Residual connections: Residual connections were introduced by He et al. (2016a), who built a network with 152 layers, which was eight times larger than VGG (figure 10.17), and achieved state-of-the-art performance on the ImageNet classification task. Each residual block consisted

of a convolutional layer followed by batch normalization, a ReLU activation, a second convolutional layer, and second batch normalization. A second ReLU function was applied after this block was added back to the main representation. This architecture was termed *ResNet v1*. He et al. (2016b) investigated different variations of residual architectures, in which either (i) processing could also be applied along the skip connection or (ii) after the two branches had recombined. They concluded neither was necessary, leading to the architecture in figure 11.7, which is sometimes termed a *pre-activation residual block* and is the backbone of *ResNet v2*. They trained a network with 200 layers that improved further on the ImageNet classification task (see figure 11.8). Since this time, new methods for regularization, optimization, and data augmentation have been developed, and Wightman et al. (2021) exploit these to present a more modern training pipeline for the ResNet architecture.

Why residual connections help: Residual networks certainly allow deeper networks to be trained. Presumably, this is related to reducing shattered gradients (Balduzzi et al., 2017) at the start of training and the smoother loss surface near the minima as depicted in figure 11.13 (Li et al., 2018b). Residual connections alone (i.e., without batch normalization) increase the trainable depth of a network by roughly a factor of two (Sankararaman et al., 2020). With batch normalization, very deep networks can be trained, but it is unclear that depth is critical for performance. Zagoruyko & Komodakis (2016) showed that wide residual networks with only 16 layers outperformed all residual networks of the time for image classification. Orhan & Pitkow (2017) propose a different explanation for why residual connections improve learning in terms of eliminating singularities (places on the loss surface where the Hessian is degenerate).

Related architectures: Residual connections are a special case of *highway networks* (Srivastava et al., 2015) which also split the computation into two branches and additively recombine. Highway networks use a gating function that weights the inputs to the two branches in a way that depends on the data itself, whereas residual networks send the data down both branches in a straightforward manner. Xie et al. (2017) introduced the ResNeXt architecture, which places a residual connection around multiple parallel convolutional branches.

Residual networks as ensembles: Veit et al. (2016) characterized residual networks as ensembles of shorter networks and depicted the "unraveled network" interpretation (figure 11.4b). They provide evidence that this interpretation is valid by showing that deleting layers in a trained network (and hence a subset of paths) only has a modest effect on performance. Conversely, removing a layer in a purely sequential network like VGG is catastrophic. They also looked at the gradient magnitudes along paths of different lengths and showed that the gradient vanishes in longer paths. In a residual network consisting of 54 blocks, almost all of the gradient updates during training were from paths of length 5 to 17 blocks long, even though these only constitute 0.45% of the total paths. It seems that adding more blocks effectively adds more parallel shorter paths rather than creating a network that is truly deeper.

Regularization for residual networks: L2 regularization of the weights has a fundamentally different effect in vanilla networks and residual networks without BatchNorm. In the former, it encourages the output of the layer to be a constant function determined by the biases. In the latter, it encourages the residual block to compute the identity plus a constant determined by the biases.

Several regularization methods have been developed that are targeted specifically at residual architectures. ResDrop (Yamada et al., 2016), stochastic depth (Huang et al., 2016), and RandomDrop (Yamada et al., 2019) all regularize residual networks by randomly dropping residual blocks during the training process. In the latter case, the propensity for dropping a block is determined by a Bernoulli variable, whose parameter is linearly decreased during training. At test time, the residual blocks are added back in with their expected probability. These methods are effectively versions of dropout, in which all the hidden units in a block are simultaneously

dropped in concert. In the multiple paths view of residual networks (figure 11.4b), they simply remove some of the paths at each training step. Wu et al. (2018b) developed BlockDrop, which analyzes an existing network and decides which residual blocks to use at runtime with the goal of improving the efficiency of inference.

Other regularization methods have been developed for networks with multiple paths inside the residual block. Shake-shake (Gastaldi, 2017a,b) randomly re-weights the paths during the forward and backward passes. In the forward pass, this can be viewed as synthesizing random data, and in the backward pass, as injecting another form of noise into the training method. ShakeDrop (Yamada et al., 2019) draws a Bernoulli variable that decides whether each block will be subject to Shake-Shake or behave like a standard residual unit on this training step.

Batch normalization: Batch normalization was introduced by Ioffe & Szegedy (2015) outside of the context of residual networks. They showed empirically that it allowed higher learning rates, increased convergence speed, and made sigmoid activation functions more practical (since the distribution of outputs is controlled, so examples are less likely to fall in the saturated extremes of the sigmoid). Balduzzi et al. (2017) investigated the activation of hidden units in later layers of deep networks with ReLU functions at initialization. They showed that many such hidden units were always active or always inactive regardless of the input but that BatchNorm reduced this tendency.

Although batch normalization helps stabilize the forward propagation of signals through a network, Yang et al. (2019) showed that it causes gradient explosion in ReLU networks without skip connections, with each layer increasing the magnitude of the gradients by $\sqrt{\pi/(\pi-1)} \approx 1.21$. This argument is summarized by Luther (2020). Since a residual network can be seen as a combination of paths of different lengths (figure 11.4), this effect must also be present in residual networks. Presumably, however, the benefit of removing the 2^K increases in magnitude in the forward pass of a network with K layers outweighs the harm done by increasing the gradients by 1.21^K in the backward pass, so overall BatchNorm makes training more stable.

Variations of batch normalization: Several variants of BatchNorm have been proposed (figure 11.14). BatchNorm normalizes each channel separately based on statistics gathered across the batch. *Ghost batch normalization* or *GhostNorm* (Hoffer et al., 2017) uses only part of the batch to compute the normalization statistics, which makes them noisier and increases the amount of regularization when the batch size is very large (figure 11.14b).

When the batch size is very small or the fluctuations within a batch are very large (as is often the case in natural language processing), the statistics in BatchNorm may become unreliable. Ioffe (2017) proposed *batch renormalization*, which keeps a running average of the batch statistics and modifies the normalization of any batch to ensure that it is more representative. Another problem is that batch normalization is unsuitable for use in recurrent neural networks (networks for processing sequences, in which the previous output is fed back as an additional input as we move through the sequence (see figure 12.19). Here, the statistics must be stored at each step in the sequence, and it's unclear what to do if a test sequence is longer than the training sequences. A third problem is that batch normalization needs access to the whole batch. However, this may not be easily available when training is distributed across several machines.

Layer normalization or *LayerNorm* (Ba et al., 2016) avoids using batch statistics by normalizing each data example separately, using statistics gathered across the channels and spatial position (figure 11.14c). However, there is still a separate learned scale γ and offset δ per channel. *Group normalization* or *GroupNorm* (Wu & He, 2018) is similar to LayerNorm but divides the channels into groups and computes the statistics for each group separately across the within-group channels and the spatial positions (figure 11.14d). Again, there are still separate scale and offset parameters per channel. *Instance normalization* or *InstanceNorm* (Ulyanov et al., 2016) takes this to the extreme where the number of groups is the same as the number of channels, so each channel is normalized separately (figure 11.14e), using statistics gathered across spatial

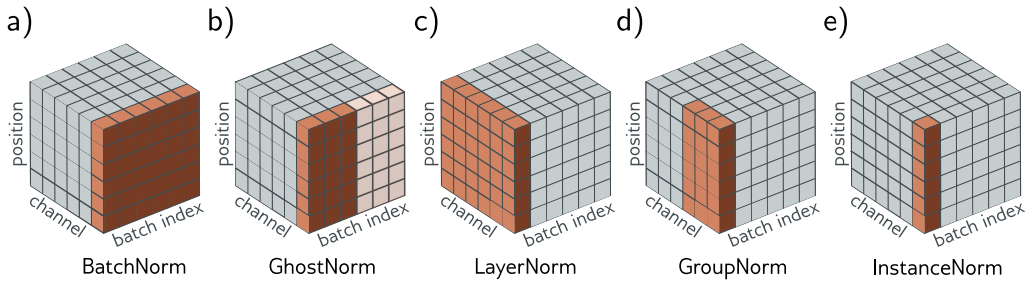

Figure 11.14 Normalization schemes. BatchNorm modifies each channel separately but adjusts each batch member in the same way based on statistics gathered across the batch and spatial position. Ghost BatchNorm computes these statistics from only part of the batch to make them more variable. LayerNorm computes statistics for each batch member separately, based on statistics gathered across the channels and spatial position. It retains a separate learned scaling factor for each channel. GroupNorm normalizes within each group of channels and also retains a separate scale and offset parameter for each channel. InstanceNorm normalizes within each channel separately, computing the statistics only across spatial position. Adapted from Wu & He (2018).

position alone. Salimans & Kingma (2016) investigated normalizing the network weights rather than the activations, but this has been less empirically successful. Teye et al. (2018) introduced *Monte Carlo batch normalization*, which can provide meaningful estimates of uncertainty in the predictions of neural networks. A recent comparison of the properties of different normalization schemes can be found in Lubana et al. (2021).

Why BatchNorm helps: BatchNorm helps control the initial gradients in a residual network (figure 11.6c). However, the mechanism by which BatchNorm improves performance is not well understood. The stated goal of Ioffe & Szegedy (2015) was to reduce problems caused by *internal covariate shift*, which is the change in the distribution of inputs to a layer caused by updating preceding layers during the backpropagation update. However, Santurkar et al. (2018) provided evidence against this view by artificially inducing covariate shift and showing that networks with and without BatchNorm performed equally well.

Motivated by this, they searched for another explanation for why BatchNorm should improve performance. They showed empirically for the VGG network that adding batch normalization decreases the variation in both the loss and its gradient as we move in the gradient direction. In other words, the loss surface is both smoother and changes more slowly, which is why larger learning rates are possible. They also provide theoretical proofs for both these phenomena and show that for any parameter initialization, the distance to the nearest optimum is less for networks with batch normalization. Bjorck et al. (2018) also argue that BatchNorm improves the properties of the loss landscape and allows larger learning rates.

Other explanations of why BatchNorm improves performance include decreasing the importance of tuning the learning rate (Ioffe & Szegedy, 2015; Arora et al., 2018). Indeed Li & Arora (2019) show that using an exponentially increasing learning rate schedule is possible with batch normalization. Ultimately, this is because batch normalization makes the network invariant to the scales of the weight matrices (see Huszár, 2019, for an intuitive visualization).

Hoffer et al. (2017) identified that BatchNorm has a regularizing effect due to statistical fluc-

tuations from the random composition of the batch. They proposed using a *ghost batch size*, in which the mean and standard deviation statistics are computed from a subset of the batch. Large batches can now be used without losing the regularizing effect of the extra noise in smaller batch sizes. Luo et al. (2018) investigate the regularization effects of batch normalization.

Alternatives to batch normalization: Although BatchNorm is widely used, it is not strictly necessary to train deep residual nets; there are other ways of making the loss surface tractable. Balduzzi et al. (2017) proposed the rescaling by $\sqrt{1/2}$ in figure 11.6b; they argued that it prevents gradient explosion but does not resolve the problem of shattered gradients.

Other work has investigated rescaling the function's output in the residual block before adding it back to the input. For example, De & Smith (2020) introduce SkipInit, in which a learnable scalar multiplier is placed at the end of each residual branch. This helps if this multiplier is initialized to less than $\sqrt{1/K}$, where K is the number of residual blocks. In practice, they suggest initializing this to zero. Similarly, Hayou et al. (2021) introduce Stable ResNet, which rescales the output of the function in the k^{th} residual block (before addition to the main branch) by a constant λ_k. They prove that in the limit of infinite width, the expected gradient norm of the weights in the first layer is lower bounded by the sum of squares of the scalings λ_k. They investigate setting these to a constant $\sqrt{1/K}$, where K is the number of residual blocks and show that it is possible to train networks with up to 1000 blocks.

Zhang et al. (2019a) introduce *FixUp*, in which every layer is initialized using He normalization, but the last linear/convolutional layer of every residual block is set to zero. Now the initial forward pass is stable (since each residual block contributes nothing), and the gradients do not explode in the backward pass (for the same reason). They also rescale the branches so that the magnitude of the total expected change in the parameters is constant regardless of the number of residual blocks. These methods allow training of deep residual networks but don't usually achieve the same test performance as when using BatchNorm. This is probably because they do not benefit from the regularization induced by the noisy batch statistics. De & Smith (2020) modify their method to induce regularization via dropout, which helps close this gap.

DenseNet and U-Net: DenseNet was first introduced by Huang et al. (2017b), U-Net was developed by Ronneberger et al. (2015), and stacked hourglass networks by Newell et al. (2016). Of these architectures, U-Net has been the most extensively adapted. Çiçek et al. (2016) introduced 3D U-Net, and Milletari et al. (2016) introduced V-Net, both of which extend U-Net to process 3D data. Zhou et al. (2018) combine the ideas of DenseNet and U-Net in an architecture that downsamples and re-upsamples the image but also repeatedly uses intermediate representations. U-Nets are commonly used in medical image segmentation (see Siddique et al., 2021, for a review). However, they have been applied to other areas, including depth estimation (Garg et al., 2016), semantic segmentation (Iglovikov & Shvets, 2018), inpainting (Zeng et al., 2019), pansharpening (Yao et al., 2018), and image-to-image translation (Isola et al., 2017). U-Nets are also a key component in diffusion models (chapter 18).

Problems

Problem 11.1 Derive equation 11.5 from the network definition in equation 11.4.

Problem 11.2 Unraveling the four-block network in figure 11.4a produces one path of length zero, four paths of length one, six paths of length two, four paths of length three, and one path of length four. How many paths of each length would there be if with (i) three residual blocks and (ii) five residual blocks? Deduce the rule for K residual blocks.

Problem 11.3 Show that the derivative of the network in equation 11.5 with respect to the first layer $\mathbf{f}_1[\mathbf{x}]$ is given by equation 11.6.

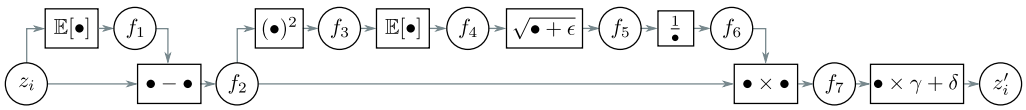

Figure 11.15 Computational graph for batch normalization (see problem 11.5).

Problem 11.4* Explain why the values in the two branches of the residual blocks in figure 11.6a are uncorrelated. Show that the variance of the sum of uncorrelated variables is the sum of their individual variances.

Problem 11.5* The forward pass for batch normalization given a batch of scalar values $\{z_i\}_{i=1}^I$ consists of the following operations (figure 11.15):

$$\begin{aligned} f_1 &= \mathbb{E}[z_i] & f_5 &= \sqrt{f_4+\epsilon} \\ f_{2i} &= z_i - f_1 & f_6 &= 1/f_5 \\ f_{3i} &= f_{2i}^2 & f_{7i} &= f_{2i} \times f_6 \\ f_4 &= \mathbb{E}[f_{3i}] & z_i' &= f_{7i} \times \gamma + \delta, \end{aligned} \tag{11.10}$$

where $\mathbb{E}[z_i] = \frac{1}{I}\sum_i z_i$. Write Python code to implement the forward pass. Now derive the algorithm for the backward pass. Work backward through the computational graph computing the derivatives to generate a set of operations that computes $\partial z_i'/\partial z_i$ for every element in the batch. Write Python code to implement the backward pass.

Problem 11.6 Consider a fully connected neural network with one input, one output, and ten hidden layers, each of which contains twenty hidden units. How many parameters does this network have? How many parameters will it have if we place a batch normalization operation between each linear transformation and ReLU?

Problem 11.7* Consider applying an L2 regularization penalty to the weights in the convolutional layers in figure 11.7a, but not to the scaling parameters of the subsequent BatchNorm layers. What do you expect will happen as training proceeds?

Problem 11.8 Consider a convolutional residual block that contains a batch normalization operation, followed by a ReLU activation function, and then a 3×3 convolutional layer. If the input and output both have 512 channels, how many parameters are needed to define this block? Now consider a bottleneck residual block that contains three batch normalization/ReLU/convolution sequences. The first uses a 1×1 convolution to reduce the number of channels from 512 to 128. The second uses a 3×3 convolution with the same number of input and output channels. The third uses a 1×1 convolution to increase the number of channels from 128 to 512 (see figure 11.7b). How many parameters are needed to define this block?

Problem 11.9 The U-Net is completely convolutional and can be run with any sized image after training. Why do we not train with a collection of arbitrarily-sized images?

Chapter 12

Transformers

Chapter 10 introduced convolutional networks, which are specialized for processing data that lie on a regular grid. They are particularly suited to processing images, which have a very large number of input variables, precluding the use of fully connected networks. Each layer of a convolutional network employs parameter sharing so that local image patches are processed similarly at every position in the image.

This chapter introduces transformers. These were initially targeted at natural language processing (NLP) problems, where the network input is a series of high-dimensional embeddings representing words or word fragments. Language datasets share some of the characteristics of image data. The number of input variables can be very large, and the statistics are similar at every position; it's not sensible to re-learn the meaning of the word dog at every possible position in a body of text. However, language datasets have the complication that text sequences vary in length, and unlike images, there is no easy way to resize them.

12.1 Processing text data

To motivate the transformer, consider the following passage:

The restaurant refused to serve me a ham sandwich because it only cooks vegetarian food. In the end, they just gave me two slices of bread. Their ambiance was just as good as the food and service.

The goal is to design a network to process this text into a representation suitable for downstream tasks. For example, it might be used to classify the review as positive or negative or to answer questions such as "Does the restaurant serve steak?".

We can make three immediate observations. First, the encoded input can be surprisingly large. In this case, each of the 37 words might be represented by an embedding vector of length 1024, so the encoded input would be of length $37 \times 1024 = 37888$ even for this small passage. A more realistically sized body of text might have hundreds or even thousands of words, so fully connected neural networks are impractical.

Second, one of the defining characteristics of NLP problems is that each input (one or more sentences) is of a different length; hence, it's not even obvious how to apply a fully connected network. These observations suggest that the network should share parameters across words at different input positions, similarly to how convolutional networks share parameters across different image positions.

Third, language is ambiguous; it is unclear from the syntax alone that the pronoun it refers to the restaurant and not to the ham sandwich. To understand the text, the word it should somehow be connected to the word restaurant. In the parlance of transformers, the former word should pay *attention* to the latter. This implies that there must be connections between the words and that the strength of these connections will depend on the words themselves. Moreover, these connections need to extend across large text spans. For example, the word their in the last sentence also refers to the restaurant.

12.2 Dot-product self-attention

The previous section argued that a model for processing text will (i) use parameter sharing to cope with long input passages of differing lengths and (ii) contain connections between word representations that depend on the words themselves. The transformer acquires both properties by using *dot-product self-attention*.

A standard neural network layer $\mathbf{f}[\mathbf{x}]$, takes a $D \times 1$ input $\mathbf{x}$ and applies a linear transformation followed by an activation function like a ReLU, so:

$$\mathbf{f}[\mathbf{x}] = \mathbf{ReLU}[\boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{x}], \tag{12.1}$$

where $\boldsymbol{\beta}$ contains the biases, and $\boldsymbol{\Omega}$ contains the weights.

A self-attention block $\mathbf{sa}[\bullet]$ takes N inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N$, each of dimension $D \times 1$, and returns N output vectors of the same size. In the context of NLP, each input represents a word or word fragment. First, a set of *values* are computed for each input:

$$\mathbf{v}_m = \boldsymbol{\beta}_v + \boldsymbol{\Omega}_v \mathbf{x}_m, \tag{12.2}$$

where $\boldsymbol{\beta}_v \in \mathbb{R}^{D\times 1}$ and $\boldsymbol{\Omega}_v \in \mathbb{R}^{D\times D}$ represent biases and weights, respectively.

Then the n^{th} output $\mathbf{sa}_n[\mathbf{x}_1, \ldots, \mathbf{x}_N]$ is a weighted sum of all the values $\mathbf{v}_1, \ldots, \mathbf{v}_N$:

$$\mathbf{sa}_n[\mathbf{x}_1, \ldots, \mathbf{x}_N] = \sum_{m=1}^{N} a[\mathbf{x}_m, \mathbf{x}_n]\mathbf{v}_m. \tag{12.3}$$

The scalar weight $a[\mathbf{x}_m, \mathbf{x}_n]$ is the *attention* that the n^{th} output pays to input $\mathbf{x}_m$. The N weights $a[\bullet, \mathbf{x}_n]$ are non-negative and sum to one. Hence, self-attention can be thought of as *routing* the values in different proportions to create each output (figure 12.1).

The following sections examine dot-product self-attention in more detail. First, we consider the computation of the values and their subsequent weighting (equation 12.3). Then we describe how to compute the attention weights $a[\mathbf{x}_m, \mathbf{x}_n]$ themselves.

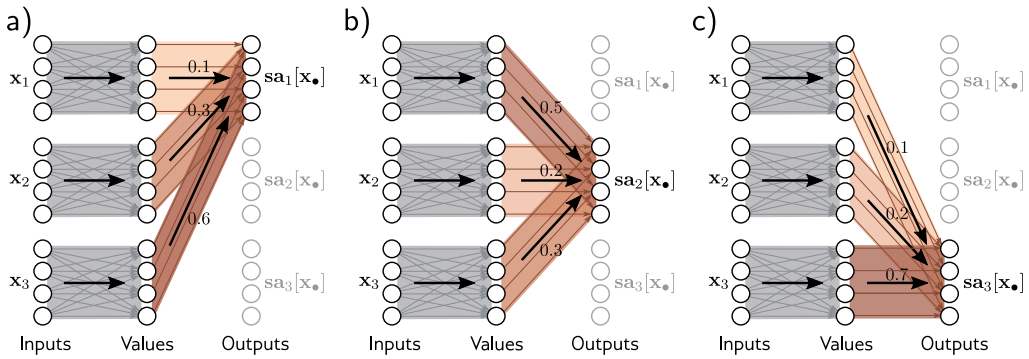

Figure 12.1 Self-attention as routing. The self-attention mechanism takes N inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathbb{R}^D$ (here $N = 3$ and $D = 4$) and processes each separately to compute N value vectors. The n^{th} output $\mathbf{sa}_n[\mathbf{x}_1, \ldots \mathbf{x}_N]$ (written as $\mathbf{sa}_n[\mathbf{x}_\bullet]$ for short) is then computed as a weighted sum of the N value vectors, where the weights are positive and sum to one. a) Output $\mathbf{sa}_1[\mathbf{x}_\bullet]$ is computed as $a[\mathbf{x}_1, \mathbf{x}_1] = 0.1$ times the first value vector, $a[\mathbf{x}_2, \mathbf{x}_1] = 0.3$ times the second value vector, and $a[\mathbf{x}_3, \mathbf{x}_1] = 0.6$ times the third value vector. b) Output $\mathbf{sa}_2[\mathbf{x}_\bullet]$ is computed in the same way, but this time with weights of 0.5, 0.2, and 0.3. c) The weighting for output $\mathbf{sa}_3[\mathbf{x}_\bullet]$ is different again. Each output can hence be thought of as a different routing of the N values.

12.2.1 Computing and weighting values

Equation 12.2 shows that the same weights $\boldsymbol{\Omega}_v \in \mathbb{R}^{D \times D}$ and biases $\boldsymbol{\beta}_v \in \mathbb{R}^D$ are applied to each input $\mathbf{x}_\bullet \in \mathbb{R}^D$. This computation scales linearly with the sequence length N, so it requires fewer parameters than a fully connected network relating all DN inputs to all DN outputs. The value computation can be viewed as a sparse matrix operation with shared parameters (figure 12.2b).

The attention weights $a[\mathbf{x}_m, \mathbf{x}_n]$ combine the values from different inputs. They are also sparse since there is only one weight for each ordered pair of inputs $(\mathbf{x}_m, \mathbf{x}_n)$, regardless of the size of these inputs (figure 12.2c). It follows that the number of attention weights has a quadratic dependence on the sequence length N, but is independent of the length D of each input.

Problem 12.1

12.2.2 Computing attention weights

In the previous section, we saw that the outputs result from two chained linear transformations; the value vectors $\boldsymbol{\beta}_v + \boldsymbol{\Omega}_v \mathbf{x}_m$ are computed independently for each input $\mathbf{x}_m$, and these vectors are combined linearly by the attention weights $a[\mathbf{x}_m, \mathbf{x}_n]$. However, the overall self-attention computation is *nonlinear*. As we'll see shortly, the attention weights are themselves nonlinear functions of the input. This is an example of a *hypernetwork*, where one network branch computes the weights of another.

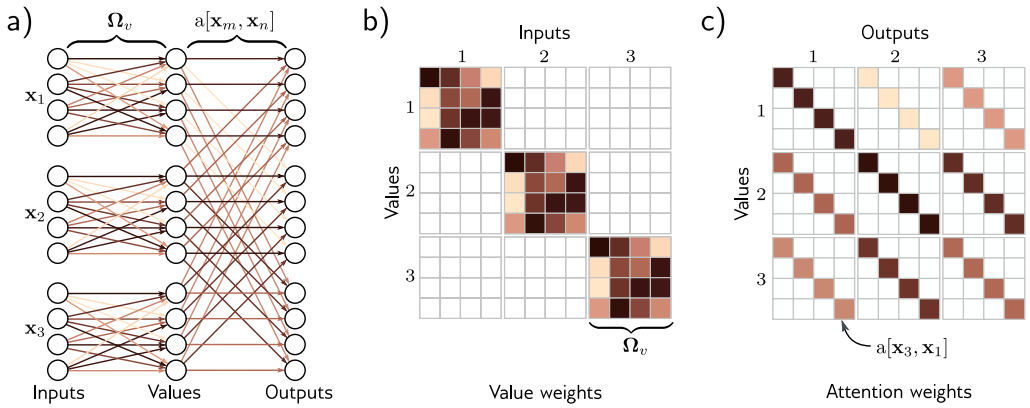

Figure 12.2 Self-attention for $N = 3$ inputs $\mathbf{x}_n$, each with dimension $D = 4$. a) Each input $\mathbf{x}_n$ is operated on independently by the same weights $\boldsymbol{\Omega}_v$ (same color equals same weight) and biases $\boldsymbol{\beta}_v$ (not shown) to form the values $\boldsymbol{\beta}_v + \boldsymbol{\Omega}_v\mathbf{x}_n$. Each output is a linear combination of the values, with a shared attention weight $a[\mathbf{x}_m, \mathbf{x}_n]$ defining the contribution of the m^{th} value to the n^{th} output. b) Matrix showing block sparsity of linear transformation $\boldsymbol{\Omega}_v$ between inputs and values. c) Matrix showing sparsity of attention weights relating values and outputs.

To compute the attention, we apply two more linear transformations to the inputs:

$$\begin{aligned}\mathbf{q}_n &= \boldsymbol{\beta}_q + \boldsymbol{\Omega}_q\mathbf{x}_n \\ \mathbf{k}_m &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{x}_m,\end{aligned} \tag{12.4}$$

Appendix B.3.4 Dot product

where $\{\mathbf{q}_n\}$ and $\{\mathbf{k}_m\}$ are termed *queries* and *keys*, respectively. Then we compute dot products between the queries and keys and pass the results through a softmax function:

$$\begin{aligned}a[\mathbf{x}_m, \mathbf{x}_n] &= \text{softmax}_m\left[\mathbf{k}_\bullet^T\mathbf{q}_n\right] \\ &= \frac{\exp\left[\mathbf{k}_m^T\mathbf{q}_n\right]}{\sum_{m'=1}^{N}\exp\left[\mathbf{k}_{m'}^T\mathbf{q}_n\right]},\end{aligned} \tag{12.5}$$

so for each $\mathbf{x}_n$, they are positive and sum to one (figure 12.3). For obvious reasons, this is known as *dot-product self-attention*.

The names "queries" and "keys" were inherited from the field of information retrieval and have the following interpretation: the dot product operation returns a measure of similarity between its inputs, so the weights $a[\mathbf{x}_\bullet, \mathbf{x}_n]$ depend on the relative similarities between the n^{th} query and all of the keys. The softmax function means that the key vectors "compete" with one another to contribute to the final result. The queries and keys must have the same dimensions. However, these can differ from the dimension of

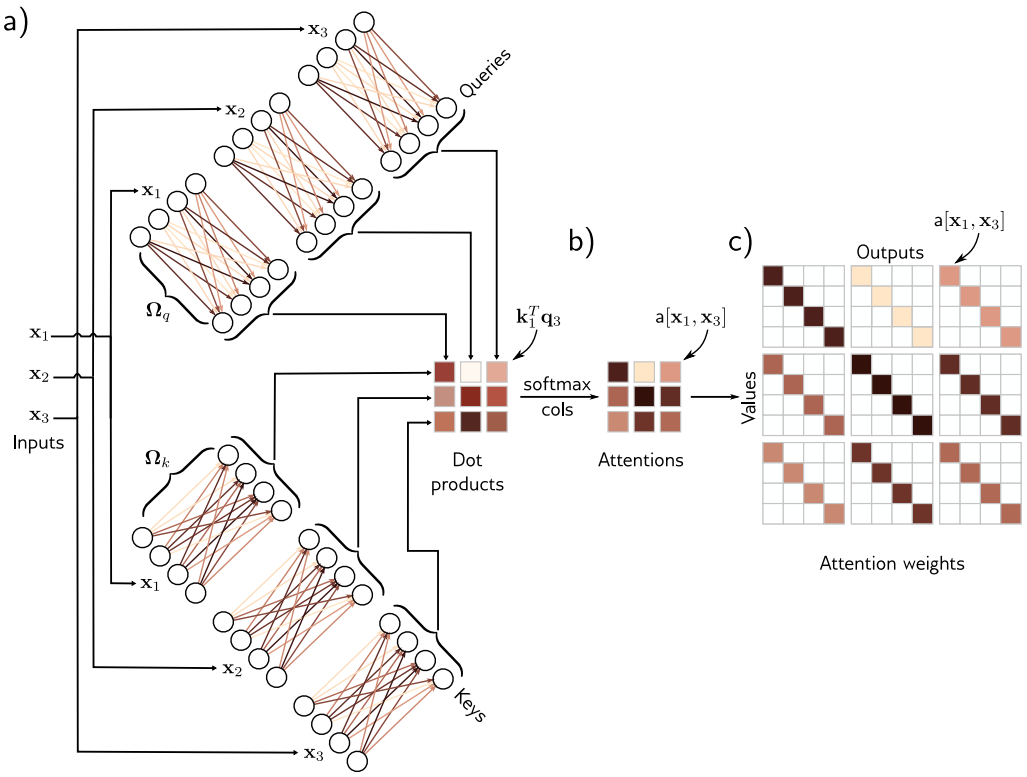

Figure 12.3 Computing attention weights. a) Query vectors $\mathbf{q}_n = \boldsymbol{\beta}_q + \boldsymbol{\Omega}_q\mathbf{x}_n$ and key vectors $\mathbf{k}_n = \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{x}_n$ are computed for each input $\mathbf{x}_n$. b) The dot products between each query and the three keys are passed through a softmax function to form non-negative attentions that sum to one. c) These route the value vectors (figure 12.1) via the sparse matrix from figure 12.2c.

the values, which is usually the same size as the input, so the representation doesn't change size. Problem 12.2

12.2.3 Self-attention summary

The n^{th} output is a weighted sum of the same linear transformation $\mathbf{v}_\bullet = \boldsymbol{\beta}_v + \boldsymbol{\Omega}_v\mathbf{x}_\bullet$ applied to all of the inputs, where these attention weights are positive and sum to one. The weights depend on a measure of similarity between input $\mathbf{x}_n$ and the other inputs. There is no activation function, but the mechanism is nonlinear due to the dot-product and a softmax operation used to compute the attention weights.

Note that this mechanism fulfills the initial requirements. First, there is a single shared set of parameters $\boldsymbol{\phi} = \{\boldsymbol{\beta}_v, \boldsymbol{\Omega}_v, \boldsymbol{\beta}_q, \boldsymbol{\Omega}_q, \boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}$. This is independent of the

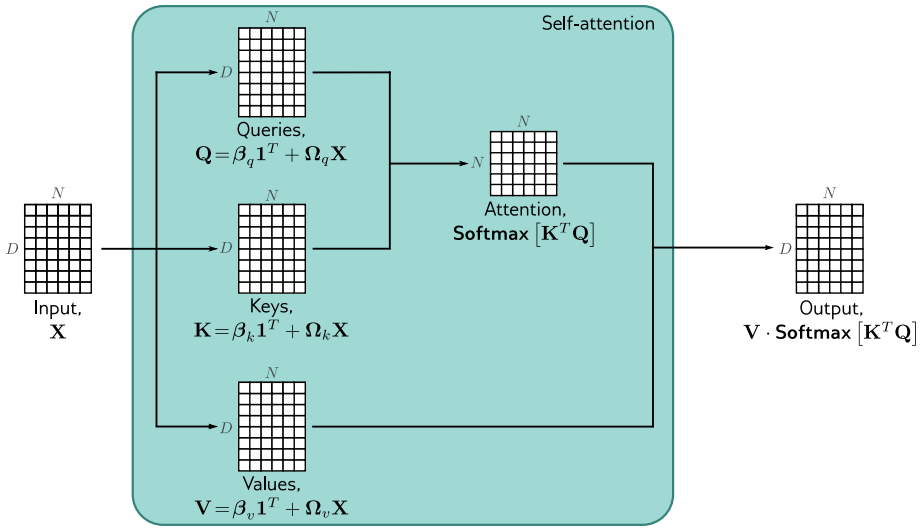

Figure 12.4 Self-attention in matrix form. Self-attention can be implemented efficiently if we store the N input vectors $\mathbf{x}_n$ in the columns of the $D\times N$ matrix $\mathbf{X}$. The input $\mathbf{X}$ is operated on separately by the query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$, and value matrix $\mathbf{V}$. The dot products are then computed using matrix multiplication, and a softmax operation is applied independently to each column of the resulting matrix to calculate the attentions. Finally, the values are post-multiplied by the attentions to create an output of the same size as the input.

number of inputs N, so the network can be applied to different sequence lengths. Second, there are connections between the inputs (words), and the strength of these connections depends on the inputs themselves via the attention weights.

12.2.4 Matrix form

The above computation can be written in a compact form if the N inputs $\mathbf{x}_n$ form the columns of the $D \times N$ matrix $\mathbf{X}$. The values, queries, and keys can be computed as:

$$\begin{aligned}\mathbf{V}[\mathbf{X}] &= \boldsymbol{\beta}_v\mathbf{1}^\mathbf{T}+\boldsymbol{\Omega}_v\mathbf{X}\\ \mathbf{Q}[\mathbf{X}] &= \boldsymbol{\beta}_q\mathbf{1}^\mathbf{T}+\boldsymbol{\Omega}_q\mathbf{X}\\ \mathbf{K}[\mathbf{X}] &= \boldsymbol{\beta}_k\mathbf{1}^\mathbf{T}+\boldsymbol{\Omega}_k\mathbf{X},\end{aligned} \tag{12.6}$$

where $\mathbf{1}$ is an $N \times 1$ vector containing ones. The self-attention computation is then:

$$\mathbf{Sa}[\mathbf{X}] = \mathbf{V}[\mathbf{X}]\cdot\mathbf{Softmax}\Big[\mathbf{K}[\mathbf{X}]^T\mathbf{Q}[\mathbf{X}]\Big], \tag{12.7}$$

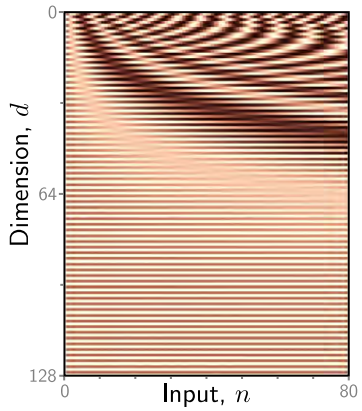

Figure 12.5 Positional encodings. The self-attention architecture is equivariant to permutations of the inputs. To ensure that inputs at different positions are treated differently, a positional encoding matrix $\mathbf{\Pi}$ can be added to the data matrix. Each column is different, so the positions can be distinguished. Here, the position encodings use a predefined procedural sinusoidal pattern (which can be extended to larger values of N if necessary). However, in other cases, they are learned.

where the function $\textbf{Softmax}[\bullet]$ takes a matrix and performs the softmax operation independently on each of its columns (figure 12.4). In this formulation, we have explicitly included the dependence of the values, queries, and keys on the input $\mathbf{X}$ to emphasize that self-attention computes a kind of triple product based on the inputs. However, from now on, we will drop this dependence and just write:

Notebook 12.1 Self-attention

$$\textbf{Sa}[\mathbf{X}] = \mathbf{V} \cdot \textbf{Softmax}[\mathbf{K}^T\mathbf{Q}]. \tag{12.8}$$

12.3 Extensions to dot-product self-attention

In the previous section, we described self-attention. Here, we introduce three extensions that are almost always used in practice.

12.3.1 Positional encoding

Observant readers will have noticed that the self-attention mechanism discards important information: the computation is the same regardless of the order of the inputs $\mathbf{x}_n$. More precisely, it is equivariant with respect to input permutations. However, order *is* important when the inputs correspond to the words in a sentence. The sentence The woman ate the raccoon has a different meaning than The raccoon ate the woman. There are two main approaches to incorporating position information.

Problem 12.3

Absolute positional encodings: A matrix $\mathbf{\Pi}$ is added to the input $\mathbf{X}$ that encodes positional information (figure 12.5). Each column of $\mathbf{\Pi}$ is unique and hence contains information about the absolute position in the input sequence. This matrix can be chosen by hand or learned. It may be added to the network inputs or at every network layer. Sometimes it is added to $\mathbf{X}$ in the computation of the queries and keys but not to the values.

Relative positional encodings: The input to a self-attention mechanism may be an entire sentence, many sentences, or just a fragment of a sentence, and the absolute position of a word is much less important than the relative position between two inputs. Of course, this can be recovered if the system knows the absolute position of both, but relative positional encodings encode this information directly. Each element of the attention matrix corresponds to a particular offset between key position a and query position b. Relative positional encodings learn a parameter $\pi_{a,b}$ for each offset and use this to modify the attention matrix by adding these values, multiplying by them, or using them to alter the attention matrix in some other way.

12.3.2 Scaled dot-product self-attention

The dot products in the attention computation can have large magnitudes and move the arguments to the softmax function into a region where the largest value completely dominates. Small changes to the inputs to the softmax function now have little effect on the output (i.e., the gradients are very small), making the model difficult to train. To prevent this, the dot products are scaled by the square root of the dimension D_q of the queries and keys (i.e., the number of rows in $\boldsymbol{\Omega}_q$ and $\boldsymbol{\Omega}_k$, which must be the same):

Problem 12.4

$$\mathbf{Sa}[\mathbf{X}] = \mathbf{V} \cdot \mathbf{Softmax}\left[\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{D_q}}\right]. \tag{12.9}$$

This is known as *scaled dot-product self-attention.*

12.3.3 Multiple heads

Multiple self-attention mechanisms are usually applied in parallel, and this is known as *multi-head self-attention.* Now H different sets of values, keys, and queries are computed:

$$\begin{aligned}\mathbf{V}_h &= \boldsymbol{\beta}_{vh}\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_{vh}\mathbf{X} \\ \mathbf{Q}_h &= \boldsymbol{\beta}_{qh}\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_{qh}\mathbf{X} \\ \mathbf{K}_h &= \boldsymbol{\beta}_{kh}\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_{kh}\mathbf{X}.\end{aligned} \tag{12.10}$$

The h^{th} self-attention mechanism or *head* can be written as:

$$\mathbf{Sa}_h[\mathbf{X}] = \mathbf{V}_h \cdot \mathbf{Softmax}\left[\frac{\mathbf{K}_h^T\mathbf{Q}_h}{\sqrt{D_q}}\right], \tag{12.11}$$

where we have different parameters $\{\boldsymbol{\beta}_{vh}, \boldsymbol{\Omega}_{vh}\}$, $\{\boldsymbol{\beta}_{qh}, \boldsymbol{\Omega}_{qh}\}$, and $\{\boldsymbol{\beta}_{kh}, \boldsymbol{\Omega}_{kh}\}$ for each head. Typically, if the dimension of the inputs $\mathbf{x}_m$ is D and there are H heads, the values, queries, and keys will all be of size D/H, as this allows for an efficient implementation. The outputs of these self-attention mechanisms are vertically concatenated, and another linear transform $\boldsymbol{\Omega}_c$ is applied to combine them (figure 12.6):

Problem 12.5

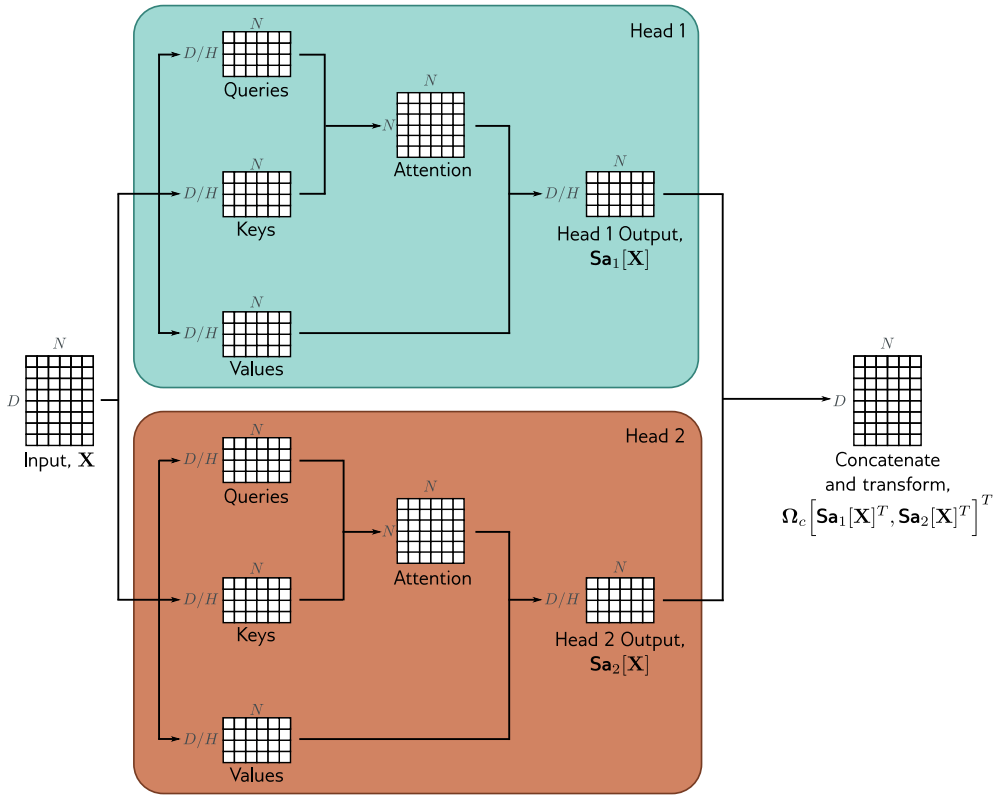

Figure 12.6 Multi-head self-attention. Self-attention occurs in parallel across multiple "heads." Each has its own queries, keys, and values. Here two heads are depicted, in the cyan and orange boxes, respectively. The outputs are vertically concatenated, and another linear transformation $\boldsymbol{\Omega}_c$ is used to recombine them.

$$\mathbf{MhSa}[\mathbf{X}] = \boldsymbol{\Omega}_c\Big[\mathbf{Sa}_1[\mathbf{X}]^T, \mathbf{Sa}_2[\mathbf{X}]^T, \ldots, \mathbf{Sa}_H[\mathbf{X}]^T\Big]^T. \tag{12.12}$$

Notebook 12.2 Multi-head self-attention

Multiple heads seem to be necessary to make self-attention work well. It has been speculated that they make the self-attention network more robust to bad initializations.

12.4 Transformer layers

Self-attention is just one part of a larger *transformer* layer. This consists of a multi-head self-attention unit (which allows the word representations to interact with each other)

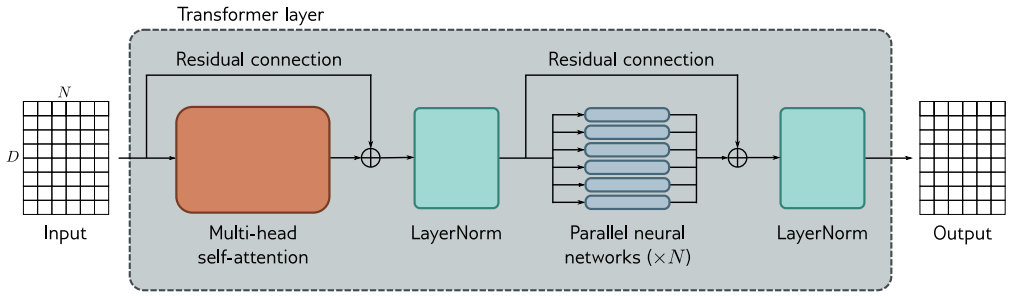

Figure 12.7 Transformer layer. The input consists of a $D \times N$ matrix containing the D-dimensional word embeddings for each of the N input tokens. The output is a matrix of the same size. The transformer layer consists of a series of operations. First, there is a multi-head attention block, allowing the word embeddings to interact with one another. This forms the processing of a residual block, so the inputs are added back to the output. Second, a LayerNorm operation is applied. Third, there is a second residual layer where the same fully connected neural network is applied separately to each of the N word representations (columns). Finally, LayerNorm is applied again.

followed by a fully connected network $\mathbf{mlp}[\mathbf{x}_\bullet]$ (that operates separately on each word). Both units are residual networks (i.e., their output is added back to the original input). In addition, it is typical to add a LayerNorm operation after both the self-attention and fully connected networks. This is similar to BatchNorm but uses statistics across the tokens within a single input sequence to perform the normalization (section 11.4 and figure 11.14). The complete layer can be described by the following series of operations (figure 12.7):

$$\begin{aligned}
\mathbf{X} &\leftarrow \mathbf{X} + \mathbf{MhSa}[\mathbf{X}] \\
\mathbf{X} &\leftarrow \mathbf{LayerNorm}[\mathbf{X}] \\
\mathbf{x}_n &\leftarrow \mathbf{x}_n + \mathbf{mlp}[\mathbf{x}_n] \qquad \forall\, n \in \{1, \ldots, N\} \\
\mathbf{X} &\leftarrow \mathbf{LayerNorm}[\mathbf{X}], \qquad (12.13)
\end{aligned}$$

where the column vectors $\mathbf{x}_n$ are separately taken from the full data matrix $\mathbf{X}$. In a real network, the data passes through a series of these transformer layers.

12.5 Transformers for natural language processing

The previous section described the transformer layer. This section describes how it is used in natural language processing (NLP) tasks. A typical NLP pipeline starts with a *tokenizer* that splits the text into words or word fragments. Then each of these tokens

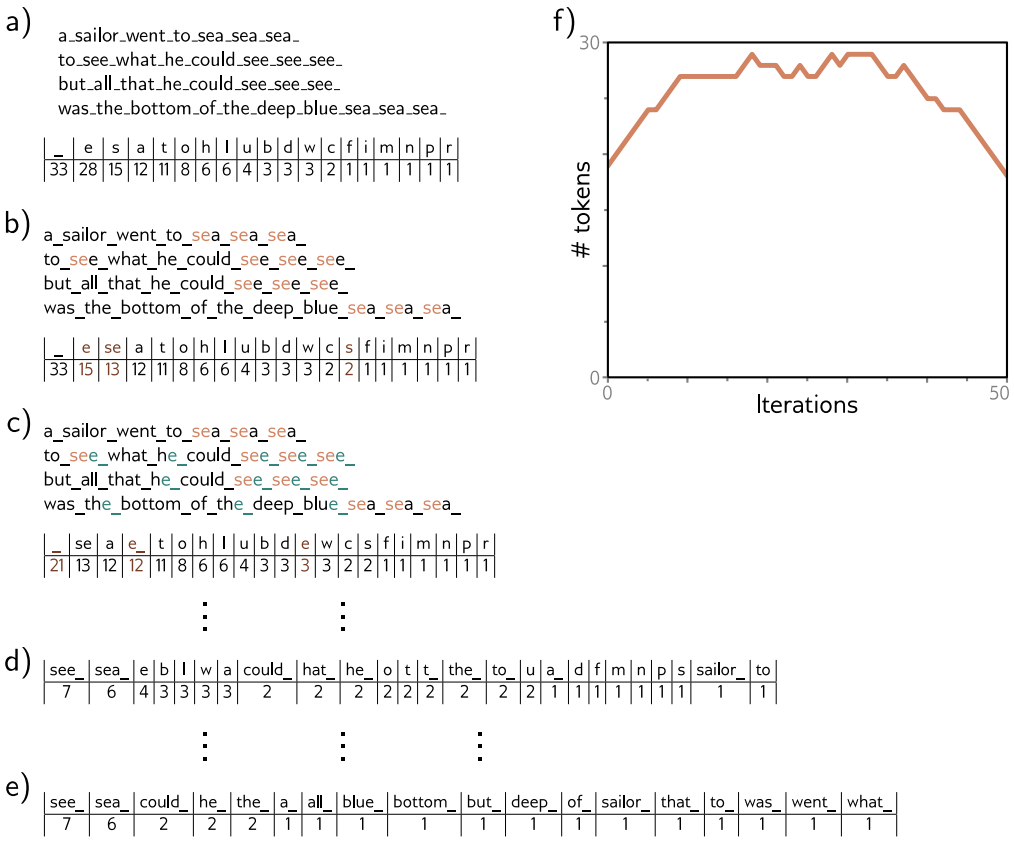

Figure 12.8 Sub-word tokenization. a) A passage of text from a nursery rhyme. The tokens are initially just the characters and whitespace (represented by an underscore), and their frequencies are displayed in the table. b) At each iteration, the sub-word tokenizer looks for the most commonly occurring adjacent pair of tokens (in this case, se) and merges them. This creates a new token and decreases the counts for the original tokens s and e. c) At the second iteration, the algorithm merges e and the whitespace character_. Note that the last character of the first token to be merged cannot be whitespace, which prevents merging across words. d) After 22 iterations, the tokens consist of a mix of letters, word fragments, and commonly occurring words. e) If we continue this process indefinitely, the tokens eventually represent the full words. f) Over time, the number of tokens increases as we add word fragments to the letters and then decreases again as we merge these fragments. In a real situation, there would be a very large number of words, and the algorithm would terminate when the vocabulary size (number of tokens) reached a predetermined value. Punctuation and capital letters would also be treated as separate input characters.

is mapped to a learned embedding. These embeddings are passed through a series of transformer layers. We now consider each of these stages in turn.

12.5.1 Tokenization

A text processing pipeline begins with a *tokenizer*. This splits the text into smaller constituent units (tokens) from a *vocabulary* of possible tokens. In the discussion above, we have implied that these tokens represent words, but there are several difficulties.

- Inevitably, some words (e.g., names) will not be in the vocabulary.
- It's unclear how to handle punctuation, but this is important. If a sentence ends in a question mark, we must encode this information.
- The vocabulary would need different tokens for versions of the same word with different suffixes (e.g., walk, walks, walked, walking), and there is no way to clarify that these variations are related.

One approach would be to use letters and punctuation marks as the vocabulary, but this would mean splitting text into very small parts and requiring the subsequent network to re-learn the relations between them.

Notebook 12.3 Tokenization

In practice, a compromise between letters and full words is used, and the final vocabulary includes both common words and word fragments from which larger and less frequent words can be composed. The vocabulary is computed using a *sub-word tokenizer* such as *byte pair encoding* (figure 12.8) that greedily merges commonly occurring sub-strings based on their frequency.

12.5.2 Embeddings

Each token in the vocabulary $\mathcal{V}$ is mapped to a unique *word embedding*, and the embeddings for the whole vocabulary are stored in a matrix $\boldsymbol{\Omega}_e \in \mathbb{R}^{D\times|\mathcal{V}|}$. To accomplish this, the N input tokens are first encoded in the matrix $\mathbf{T} \in \mathbb{R}^{|\mathcal{V}|\times N}$, where the n^{th} column corresponds to the n^{th} token and is a $|\mathcal{V}| \times 1$ *one-hot vector* (i.e., a vector where every entry is zero except for the entry corresponding to the token, which is set to one). The input embeddings are computed as $\mathbf{X} = \boldsymbol{\Omega}_e\mathbf{T}$, and $\boldsymbol{\Omega}_e$ is learned like any other network parameter (figure 12.9). A typical embedding size D is 1024, and a typical total vocabulary size $|\mathcal{V}|$ is 30,000, so even before the main network, there are many parameters in $\boldsymbol{\Omega}_e$ to learn.

12.5.3 Transformer model

Finally, the embedding matrix $\mathbf{X}$ representing the text is passed through a series of K transformer layers, called a *transformer model*. There are three types of transformer models. An *encoder* transforms the text embeddings into a representation that can support a variety of tasks. A *decoder* predicts the next token to continue the input

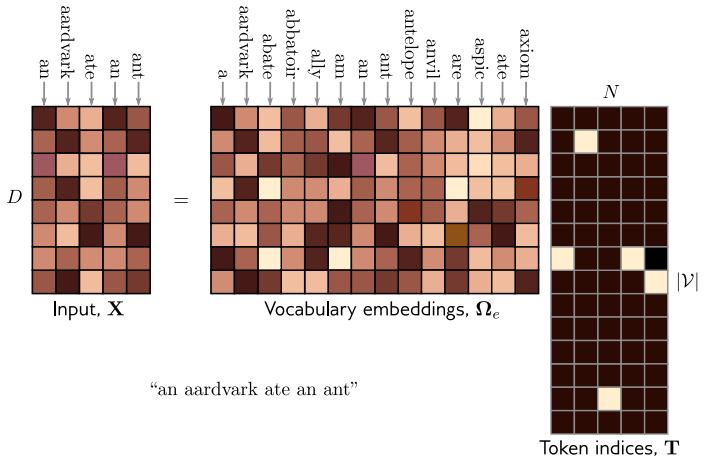

Figure 12.9 The input embedding matrix $\mathbf{X} \in \mathbb{R}^{D \times N}$ contains N embeddings of length D and is created by multiplying a matrix $\boldsymbol{\Omega}_e$ containing the embeddings for the entire vocabulary with a matrix containing one-hot vectors in its columns that correspond to the word or sub-word indices. The vocabulary matrix $\boldsymbol{\Omega}_e$ is considered a parameter of the model and is learned along with the other parameters. Note that the two embeddings for the word an in $\mathbf{X}$ are the same.

text. *Encoder-decoders* are used in *sequence-to-sequence tasks*, where one text string is converted into another (e.g., machine translation). These variations are described in sections 12.6–12.8, respectively.

12.6 Encoder model example: BERT

BERT is an encoder model that uses a vocabulary of 30,000 tokens. Input tokens are converted to 1024-dimensional word embeddings and passed through 24 transformer layers. Each contains a self-attention mechanism with 16 heads. The queries, keys, and values for each head are of dimension 64 (i.e., the matrices $\boldsymbol{\Omega}_{vh}, \boldsymbol{\Omega}_{qh}, \boldsymbol{\Omega}_{kh}$ are 1024×64). The dimension of the single hidden layer in the fully connected networks is 4096. The total number of parameters is $\sim$ 340 million. When BERT was introduced, this was considered large, but it is now much smaller than state-of-the-art models.

Encoder models like BERT exploit *transfer learning* (section 9.3.6). During *pre-training*, the parameters of the transformer architecture are learned using *self-supervision* from a large corpus of text. The goal here is for the model to learn general information about the statistics of language. In the *fine-tuning stage*, the resulting network is adapted to solve a particular task using a smaller body of supervised training data.

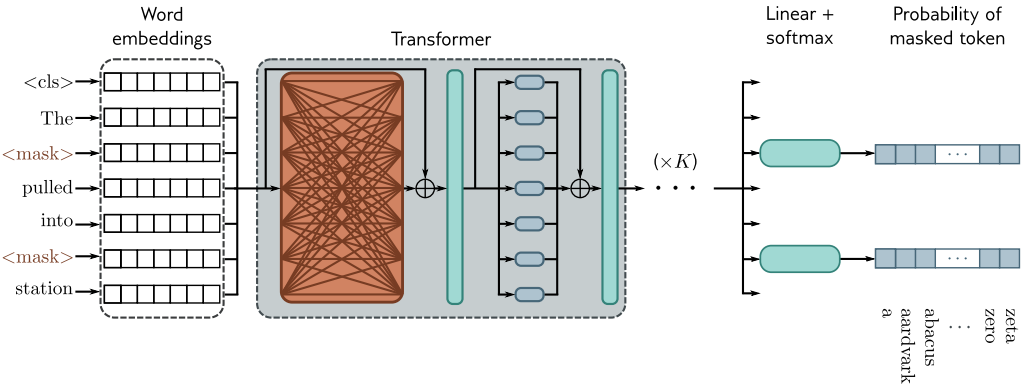

Figure 12.10 Pre-training for BERT-like encoder. The input tokens (and a special <cls> token denoting the start of the sequence) are converted to word embeddings. Here, these are represented as rows rather than columns, so the box labeled "word embeddings" is $\mathbf{X}^T$. These embeddings are passed through a series of transformer layers (orange connections indicate that every token attends to every other token in these layers) to create a set of output embeddings. A small fraction of the input tokens are randomly replaced with a generic <mask> token. In pre-training, the goal is to predict the missing word from the associated output embedding. As such, the output embeddings are passed through a softmax function, and the multiclass classification loss (section 5.24) is used. This task has the advantage that it uses both the left and right context to predict the missing word but has the disadvantage that it does not make efficient use of data; here, seven tokens need to be processed to add two terms to the loss function.

12.6.1 Pre-training

Problem 12.6

In the pre-training stage, the network is trained using self-supervision. This allows the use of enormous amounts of data without the need for manual labels. For BERT, the self-supervision task consists of predicting missing words from sentences from a large internet corpus (figure 12.10).[1] During training, the maximum input length is 512 tokens, and the batch size is 256. The system is trained for a million steps, corresponding to roughly 50 epochs of the 3.3-billion word corpus.

Predicting missing words forces the transformer network to understand some syntax. For example, it might learn that the adjective red is often found before nouns like house or car but never before a verb like shout. It also allows the model to learn superficial *common sense* about the world. For example, after training, the model will assign a higher probability to the missing word train in the sentence The <mask> pulled into the station than it would to the word peanut. However, the degree of "understanding" this type of model can ever have is limited.

[1] BERT also uses a secondary task that predicts whether two sentences were originally adjacent in the text or not, but this only marginally improves performance.

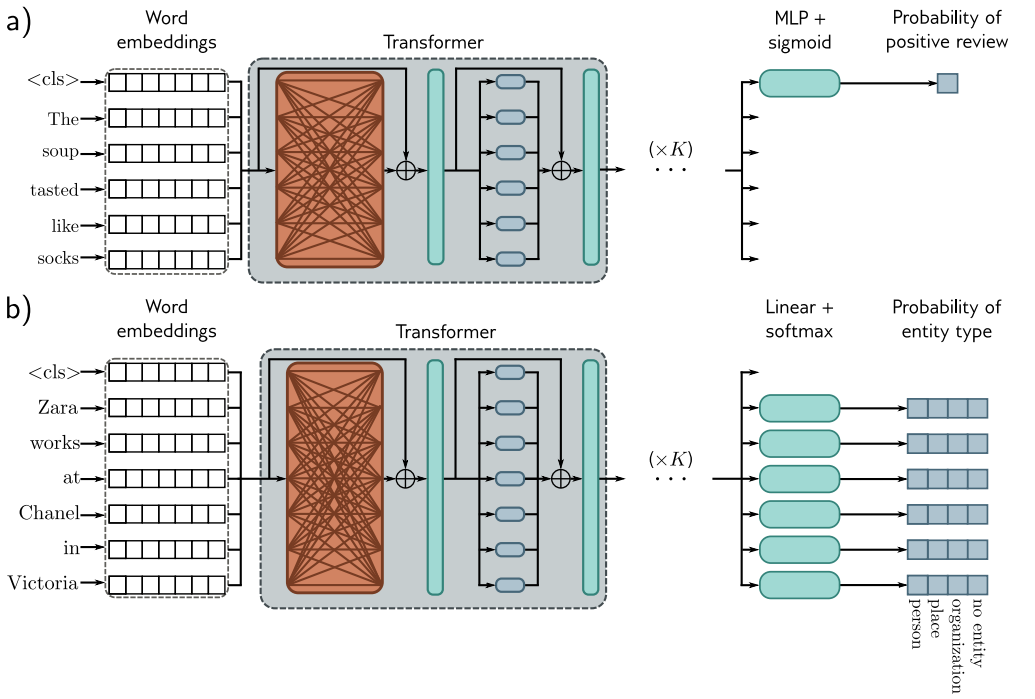

Figure 12.11 After pre-training, the encoder is fine-tuned using manually labeled data to solve a particular task. Usually, a linear transformation or a multi-layer perceptron (MLP) is appended to the encoder to produce whatever output is required. a) Example text classification task. In this sentiment classification task, the <cls> token embedding is used to predict the probability that the review is positive. b) Example word classification task. In this named entity recognition problem, the embedding for each word is used to predict whether the word corresponds to a person, place, or organization, or is not an entity.

12.6.2 Fine-tuning

In the fine-tuning stage, the model parameters are adjusted to specialize the network to a particular task. An extra layer is appended onto the transformer network to convert the output vectors to the desired output format. Examples include:

Text classification: In BERT, a special token known as the classification or <cls> token is placed at the start of each string during pre-training. For text classification tasks like *sentiment analysis* (in which the passage is labeled as having a positive or negative emotional tone), the vector associated with the <cls> token is mapped to a single number and passed through a logistic sigmoid (figure 12.11a). This contributes to a standard binary cross-entropy loss (section 5.4).

Word classification: The goal of *named entity recognition* is to classify each word as an entity type (e.g., person, place, organization, or no-entity). To this end, each input embedding $\mathbf{x}_n$ is mapped to an $E \times 1$ vector where the E entries correspond to the E entity types. This is passed through a softmax function to create probabilities for each class, which contribute to a multiclass cross-entropy loss (figure 12.11b).

Text span prediction: In the SQuAD 1.1 question answering task, the question and a passage from Wikipedia containing the answer are concatenated and tokenized. BERT is then used to predict the text span in the passage that contains the answer. Each token maps to two numbers indicating how likely it is that the text span begins and ends at this location. The resulting two sets of numbers are put through two softmax functions. The likelihood of any text span being the answer can be derived by combining the probability of starting and ending at the appropriate places.

12.7 Decoder model example: GPT3

This section presents a high-level description of GPT3, an example of a decoder model. The basic architecture is extremely similar to the encoder model and comprises a series of transformer layers that operate on learned word embeddings. However, the goal is different. The encoder aimed to build a representation of the text that could be fine-tuned to solve a variety of more specific NLP tasks. Conversely, the decoder has one purpose: to generate the next token in a sequence. It can generate a coherent text passage by feeding the extended sequence back into the model.

12.7.1 Language modeling

GPT3 constructs an autoregressive language model. This is easiest to understand with a concrete example. Consider the sentence It takes great courage to let yourself appear weak. For simplicity, let's assume that the tokens are the full words. The probability of the full sentence is:

$$\begin{aligned} Pr(\text{It takes great courage to let yourself appear weak}) \quad = \\ Pr(\text{It}) \times Pr(\text{takes}|\text{It}) \times Pr(\text{great}|\text{It takes}) \times Pr(\text{courage}|\text{It takes great}) \times \\ Pr(\text{to}|\text{It takes great courage}) \times Pr(\text{let}|\text{It takes great courage to}) \times \\ Pr(\text{yourself}|\text{It takes great courage to let}) \times \\ Pr(\text{appear}|\text{It takes great courage to let yourself}) \times \\ Pr(\text{weak}|\text{It takes great courage to let yourself appear}). \end{aligned} \tag{12.14}$$

More formally, an autoregressive model factors the joint probability $Pr(t_1, t_2, \ldots, t_N)$ of the N observed tokens into an autoregressive sequence:

$$Pr(t_1, t_2, \ldots, t_N) = Pr(t_1) \prod_{n=2}^{N} Pr(t_n | t_1, \ldots, t_{n-1}). \tag{12.15}$$

The autoregressive formulation demonstrates the connection between maximizing the log probability of the tokens in the loss function and the next token prediction task.

12.7.2 Masked self-attention

To train a decoder, we maximize the log probability of the input text under the autoregressive model. Ideally, we would pass in the whole sentence and compute all the log probabilities and gradients simultaneously. However, this poses a problem; if we pass in the full sentence, the term computing $\log\left[Pr(\text{great}|\text{It takes})\right]$ has access to both the answer great and the right context courage to let yourself appear weak. Hence, the system can cheat rather than learn to predict the following words and will not train properly.

Fortunately, the tokens only interact in the self-attention layers in a transformer network. Hence, the problem can be resolved by ensuring that the attention to the answer and the right context is zero. This can be achieved by setting the corresponding dot products in the self-attention computation (equation 12.5) to negative infinity before they are passed through the $\textbf{softmax}[\bullet]$ function. This is known as *masked self-attention*. The effect is to make the weight of all the upward-angled arrows in figure 12.1 zero.

The entire decoder network operates as follows. The input text is tokenized, and the tokens are converted to embeddings. The embeddings are passed into the transformer network, but now the transformer layers use masked self-attention so that they can only attend to the current and previous tokens. Each of the output embeddings can be thought of as representing a partial sentence, and for each, the goal is to predict the next token in the sequence. Consequently, after the transformer layers, a linear layer maps each word embedding to the size of the vocabulary, followed by a $\textbf{softmax}[\bullet]$ function that converts these values to probabilities. During training, we aim to maximize the sum of the log probabilities of the next token in the ground truth sequence at every position using a standard multiclass cross-entropy loss (figure 12.12).

12.7.3 Generating text from a decoder

The autoregressive language model is the first example of a *generative model* discussed in this book. Since it defines a probability model over text sequences, it can be used to sample new examples of plausible text. To generate from the model, we start with an input sequence of text (which might be just a special <start> token indicating the beginning of the sequence) and feed this into the network, which then outputs the probabilities over possible subsequent tokens. We can then either pick the most likely token or sample from this probability distribution. The new extended sequence can be fed back into the decoder network that outputs the probability distribution over the next token. By repeating this process, we can generate large bodies of text. The computation can be made quite efficient as prior embeddings do not depend on subsequent ones due to

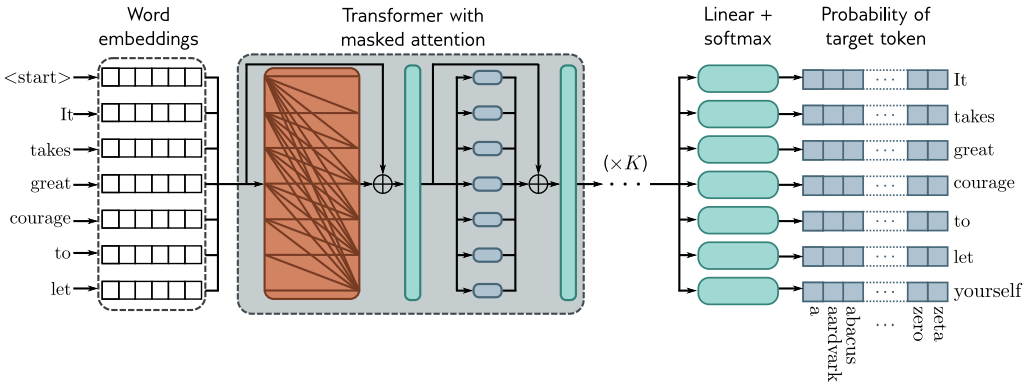

Figure 12.12 Training GPT3-type decoder network. The tokens are mapped to word embeddings with a special <start> token at the beginning of the sequence. The embeddings are passed through a series of transformer layers that use masked self-attention. Here, each position in the sentence can only attend to its own embedding and those of tokens earlier in the sequence (orange connections). The goal at each position is to maximize the probability of the following ground truth token in the sequence. In other words, at position one, we want to maximize the probability of the token It; at position two, we want to maximize the probability of the token takes; and so on. Masked self-attention ensures the system cannot cheat by looking at subsequent inputs. The autoregressive task has the advantage of making efficient use of the data since every word contributes a term to the loss function. However, it only exploits the left context of each word.

Problem 12.7

the masked self-attention. Hence, much of the earlier computation can be recycled as we generate subsequent tokens.

Notebook 12.4 Decoding strategies

In practice, many strategies can make the output text more coherent. For example, *beam search* keeps track of multiple possible sentence completions to find the overall most likely (which is not necessarily found by greedily choosing the most likely next word at each step). *Top-k sampling* randomly draws the next word from only the top-K most likely possibilities to prevent the system from accidentally choosing from the long tail of low-probability tokens and leading to an unnecessary linguistic dead end.

12.7.4 GPT3 and few-shot learning

Large language models like GPT3 apply these ideas on a massive scale. In GPT3, the sequence lengths are 2048 tokens long, and the total batch size is 3.2 million tokens. There are 96 transformer layers (some of which implement a sparse version of attention), each processing a word embedding of size 12288. There are 96 heads in the self-attention layers, and the value, query, and key dimension is 128. It is trained with 300 billion tokens and contains 175 billion parameters.

Here's an example of completing text from the GPT3 model where the text provided to the model is in orange, and the generated text is in cyan:

Understanding Deep Learning is a new textbook from MIT Press by Simon Prince that's designed to offer an accessible, broad introduction to the field. Deep learning is a branch of machine learning that is concerned with algorithms that learn from data that is unstructured or unlabeled. The book is divided into four sections:

1. Introduction to deep learning
2. Deep learning architecture
3. Deep learning algorithms
4. Applications of deep learning

The first section offers an introduction to deep learning, including its history and origins. The second section covers deep learning architecture, discussing various types of neural networks and their applications. The third section dives into deep learning algorithms, including supervised and unsupervised learning, reinforcement learning, and more. The fourth section applies deep learning to various domains, such as computer vision, natural language processing, and robotics.

This continuation is plausible, albeit only partially accurate.

One surprising property of learning models on this scale is that they can perform many tasks without fine-tuning. If we provide several examples of correct question/answer pairs and then another question, they often answer the final question correctly by completing the sequence. One example is correcting English grammar:

Poor English input: I eated the purple berries.
Good English output: I ate the purple berries.
Poor English input: Thank you for picking me as your designer. I'd appreciate it.
Good English output: Thank you for choosing me as your designer. I appreciate it.
Poor English input: The mentioned changes have done. or I did the alteration that you requested. or I changed things you wanted and did the modifications.
Good English output: The requested changes have been made. or I made the alteration that you requested. or I changed things you wanted and made the modifications.
Poor English input: I'd be more than happy to work with you in another project.
Good English output: I'd be more than happy to work with you on another project.

(result from Brown et al., 2020)

Here, the text containing the paired examples in orange was provided as context for GPT3, and the system then generated the correct answer in cyan. This phenomenon extends to many situations, including generating code snippets based on natural language descriptions, arithmetic, translating between languages, and answering questions about text passages. Consequently, it is argued that enormous language models are *few-shot learners*; they can learn to do novel tasks based on just a few examples. However, performance is erratic in practice, and the extent to which it is extrapolating from learned examples rather than merely interpolating or copying verbatim is unclear.

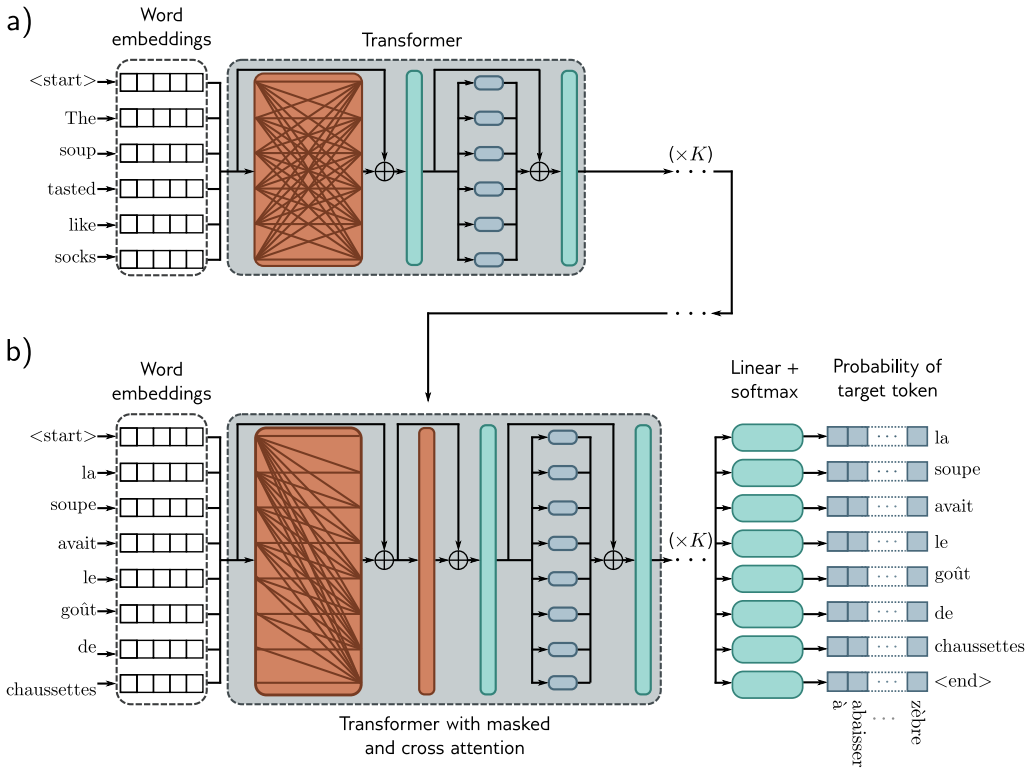

Figure 12.13 Encoder-decoder architecture. Two sentences are passed to the system with the goal of translating the first into the second. a) The first sentence is passed through a standard encoder. b) The second sentence is passed through a decoder that uses masked self-attention but also attends to the output embeddings of the encoder using cross-attention (orange rectangle). The loss function is the same as for the decoder model; we want to maximize the probability of the next word in the output sequence.

12.8 Encoder-decoder model example: machine translation

Translation between languages is an example of a *sequence-to-sequence* task. This requires an encoder (to compute a good representation of the source sentence) and a decoder (to generate the sentence in the target language). This task can be tackled using an *encoder-decoder* model.

Consider translating from English to French. The encoder receives the sentence in English and processes it through a series of transformer layers to create an output representation for each token. During training, the decoder receives the ground truth translation in French and passes it through a series of transformer layers that use masked self-attention and predict the following word at each position. However, the decoder layers also attend to the output of the encoder. Consequently, each French output word is

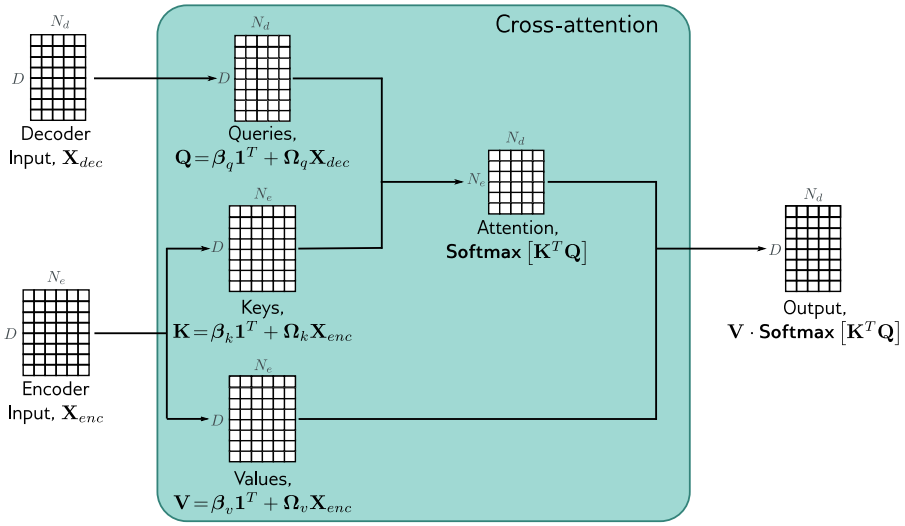

Figure 12.14 Cross-attention. The flow of computation is the same as in standard self-attention. However, the queries are calculated from the decoder embeddings $\mathbf{X}_{dec}$, and the keys and values from the encoder embeddings $\mathbf{X}_{enc}$. In the context of translation, the encoder contains information about the source language, and the decoder contains information about the target language statistics.

conditioned on the previous output words *and* the source English sentence (figure 12.13).

This is achieved by modifying the transformer layers in the decoder. Originally, these consisted of a masked self-attention layer followed by a neural network applied individually to each embedding (figure 12.12). A new self-attention layer is added between these two components, in which the decoder embeddings attend to the encoder embeddings. This uses a version of self-attention known as *encoder-decoder attention* or *cross-attention*, where the queries are computed from the decoder embeddings and the keys and values from the encoder embeddings (figure 12.14).

12.9 Transformers for long sequences

Since each token in a transformer encoder model interacts with every other token, the computational complexity scales quadratically with the length of the sequence. For a decoder model, each token only interacts with previous tokens, so there are roughly half the number of interactions, but the complexity still scales quadratically. These relationships can be visualized as interaction matrices (figure 12.15a–b).

This quadratic increase in the amount of computation ultimately limits the length of sequences that can be used. Many methods have been developed to extend the trans-

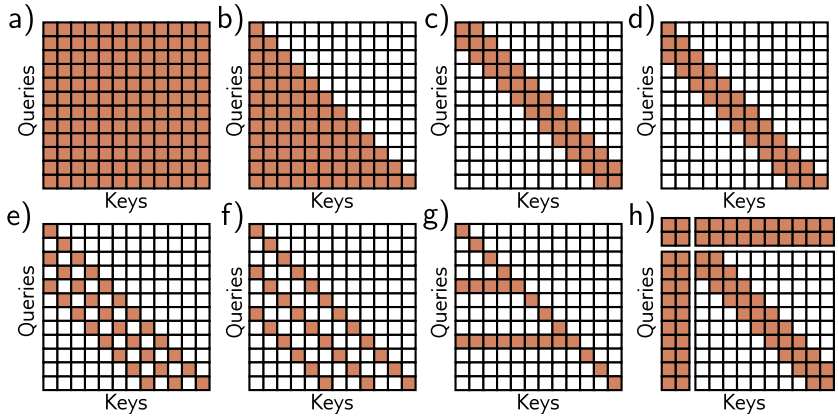

Figure 12.15 Interaction matrices for self-attention. a) In an encoder, every token interacts with every other token, and computation expands quadratically with the number of tokens. b) In a decoder, each token only interacts with the previous tokens, but complexity is still quadratic. c) Complexity can be reduced by using a convolutional structure (encoder case). d) Convolutional structure for decoder case. e–f) Convolutional structure with dilation rate of two and three (decoder case). g) Another strategy is to allow selected tokens to interact with all the other tokens (encoder case) or all the previous tokens (decoder case pictured). h) Alternatively, global tokens can be introduced (left two columns and top two rows). These interact with all of the tokens as well as with each other.

former to cope with longer sequences. One approach is to prune the self-attention interactions or, equivalently, to sparsify the interaction matrix (figures 12.15c-h). For example, this can be restricted to a convolutional structure so that each token only interacts with a few neighboring tokens. Across multiple layers, tokens still interact at larger distances as the receptive field expands. As for convolution in images, the kernel can vary in size and dilation rate.

A pure convolutional approach requires many layers to integrate information over large distances. One way to speed up this process is to allow select tokens (perhaps at the start of every sentence) to attend to all other tokens (encoder model) or all previous tokens (decoder model). A similar idea is to have a small number of global tokens that connect to all the other tokens and themselves. Like the <cls> token, these do not represent any word but serve to provide long-distance connections.

12.10 Transformers for images

Transformers were initially developed for text data. Their enormous success in this area led to experimentation on images. This was not obviously a promising idea for two

reasons. First, there are many more pixels in an image than words in a sentence, so the quadratic complexity of self-attention poses a practical bottleneck. Second, convolutional nets have a good inductive bias because each layer is equivariant to spatial translation, and they take into account the 2D structure of the image. However, this must be learned in a transformer network.

Regardless of these apparent disadvantages, transformer networks for images have now eclipsed the performance of convolutional networks for image classification and other tasks. This is partly because of the enormous scale at which they can be constructed and the large amounts of data that can be used to pre-train the networks. This section describes transformer models for images.

12.10.1 ImageGPT

ImageGPT is a transformer decoder; it builds an autoregressive model of image pixels that ingests a partial image and predicts the subsequent pixel value. The quadratic complexity of the transformer network means that the largest model (which contained 6.8 billion parameters) could still only operate on 64×64 images. Moreover, to make this tractable, the original 24-bit RGB color space had to be quantized into a nine-bit color space, so the system ingests (and predicts) one of 512 possible tokens at each position.

Images are naturally 2D objects, but ImageGPT simply learns a different positional encoding at each pixel. Hence it must learn that each pixel has a close relationship with its preceding neighbors and also with nearby pixels in the row above. Figure 12.16 shows example generation results.

The internal representation of this decoder was used as a basis for image classification. The final pixel embeddings are averaged, and a linear layer maps these to activations which are passed through a softmax layer to predict class probabilities. The system is pre-trained on a large corpus of web images and then fine-tuned on the ImageNet database resized to 48×48 pixels using a loss function that contains both a cross-entropy term for image classification and a generative loss term for predicting the pixels. Despite using a large amount of external training data, the system achieved only a 27.4% top-1 error rate on ImageNet (figure 10.15). This was less than convolutional architectures of the time (see figure 10.21) but is still impressive given the small input image size; unsurprisingly, it fails to classify images where the target object is small or thin.

12.10.2 Vision Transformer (ViT)

The *Vision Transformer* tackled the problem of image resolution by dividing the image into 16×16 patches (figure 12.17). Each patch is mapped to a lower dimension via a learned linear transformation, and these representations are fed into the transformer network. Once again, standard 1D positional encodings are learned.

Problem 12.8

This is an encoder model with a <cls> token (see figures 12.10–12.11). However, unlike BERT, it uses *supervised* pre-training on a large database of 303 million labeled images from 18,000 classes. The <cls> token is mapped via a final network layer to create activations that are fed into a softmax function to generate class probabilities. After pre-training, the system is applied to the final classification task by replacing this

Figure 12.16 ImageGPT. a) Images generated from the autoregressive ImageGPT model. The top-left pixel is drawn from the estimated empirical distribution at this position. Subsequent pixels are generated in turn, conditioned on the previous ones, working along the rows until the bottom-right of the image is reached. For each pixel, the transformer decoder generates a conditional distribution as in equation 12.15, and a sample is drawn. The extended sequence is then fed back into the network to generate the next pixel, and so on. b) Image completion. In each case, the lower half of the image is removed (top row), and ImageGPT completes the remaining part pixel by pixel (three different completions shown). Adapted from https://openai.com/blog/image-gpt/.

final layer with one that maps to the desired number of classes and is fine-tuned.

For the ImageNet benchmark, this system achieved an 11.45% top-1 error rate. However, it did not perform as well as the best contemporary convolutional networks without supervised pre-training. The strong inductive bias of convolutional networks can only be superseded by employing extremely large amounts of training data.

12.10.3 Multi-scale vision transformers

The Vision Transformer differs from convolutional architectures in that it operates on a single scale. Several transformer models that process the image at multiple scales have been proposed. Similarly to convolutional networks, these generally start with high-resolution patches and few channels and gradually decrease the resolution while simultaneously increasing the number of channels.

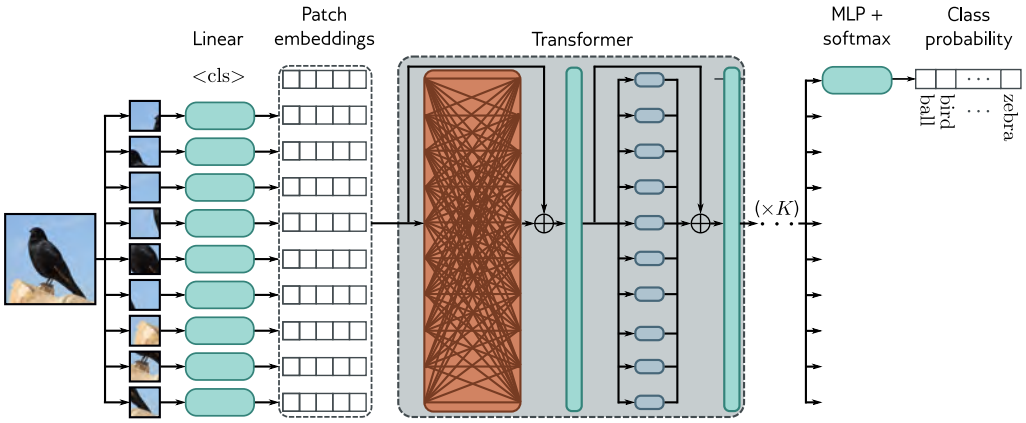

Figure 12.17 Vision transformer. The Vision Transformer (ViT) breaks the image into a grid of patches (16×16 in the original implementation). Each of these is projected via a learned linear transformation to become a patch embedding. These patch embeddings are fed into a transformer encoder network, and the <cls> token is used to predict the class probabilities.

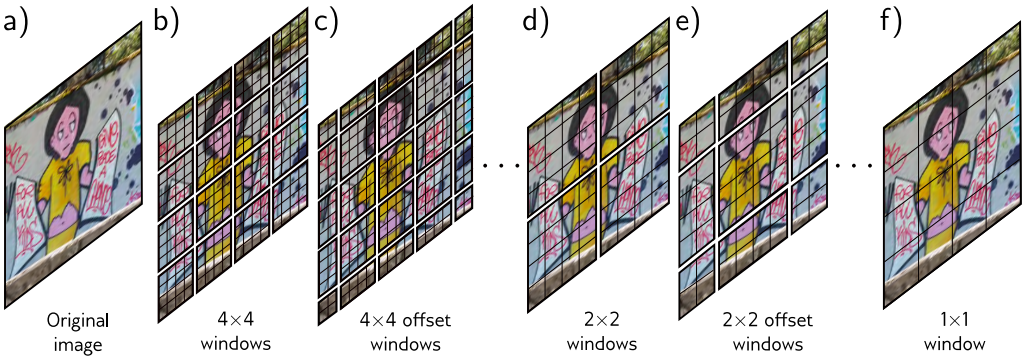

Figure 12.18 Shifted window (SWin) transformer (Liu et al., 2021c). a) Original image. b) The SWin transformer breaks the image into a grid of windows and each of these windows into a sub-grid of patches. The transformer network applies self-attention to the patches within each window independently. c) Each alternate layer shifts the windows so that the subsets of patches that interact with one another change, and information can propagate across the whole image. d) After several layers, the 2×2 blocks of patch representations are concatenated to increase the effective patch (and window) size. e) Alternate layers use shifted windows at this new lower resolution. f) Eventually, the resolution is such that there is just a single window, and the patches span the entire image.

A representative example of a multi-scale transformer is the *shifted-window* or *SWin* transformer. This is an encoder transformer that divides the image into patches and groups these patches into a grid of windows within which self-attention is applied independently (figure 12.18). These windows are shifted in adjacent transformers, so the effective receptive field at a given patch can expand beyond the window border.

The scale is reduced periodically by concatenating features from non-overlapping 2×2 patches and applying a linear transformation that maps these concatenated features to twice the original number of channels. This architecture does not have a <cls> token but instead averages the output features at the last layer. These are then mapped via a linear layer to the desired number of classes and passed through a softmax function to output class probabilities. At the time of writing, the most sophisticated version of this architecture achieves a 9.89% top-1 error rate on the ImageNet database.

A related idea is periodically to integrate information from across the whole image. *Dual attention vision transformers* (DaViT) alternate two types of transformers. In the first, image patches attend to one another, and the self-attention computation uses all the channels. In the second, the channels attend to one another, and the self-attention computation uses all the image patches. This architecture reaches a 9.60% top-1 error rate on ImageNet and is close to the state-of-the-art at the time of writing.

Problem 12.9

12.11 Summary

This chapter introduced self-attention and the transformer architecture. Encoder, decoder, and encoder-decoder models were then described. The transformer operates on sets of high-dimensional embeddings. It has a low computational complexity per layer, and much of the computation can be performed in parallel using the matrix form. Since every input embedding interacts with every other, it can describe long-range dependencies in text. Ultimately, the computation scales quadratically with the sequence length; one approach to reducing the complexity is sparsifying the interaction matrix.

The training of transformers with very large unlabeled datasets is the first example of *unsupervised learning* (learning without labels) in this book. Encoders learn a representation that can be used for other tasks by predicting missing tokens. Decoders build an autoregressive model over the inputs and are the first example of a *generative model* in this book. The generative decoders can be used to create new data examples.

Chapter 13 considers networks for processing graph data. These have connections with transformers in that the nodes of the graph attend to one another in each network layer. Chapters 14–18 return to unsupervised learning and generative models.

Notes

Natural language processing: Transformers were developed for natural language processing (NLP) tasks. This is an enormous area that deals with text analysis, categorization, generation, and manipulation. Example tasks include part of speech tagging, translation, text classification, entity recognition (people, places, companies, etc.), text summarization, question answering,

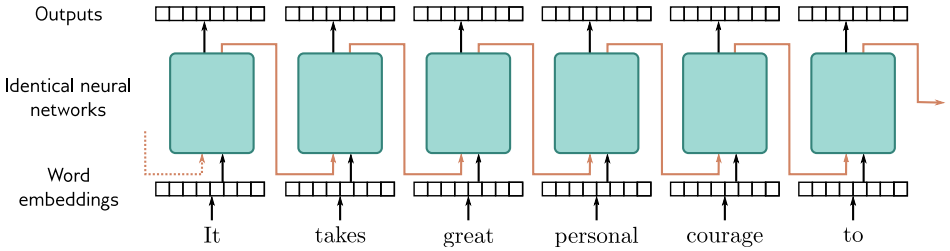

Figure 12.19 Recurrent neural networks (RNNs). The word embeddings are passed sequentially through a series of identical neural networks. Each network has two outputs; one is the output embedding, and the other (orange arrows) feeds back into the next neural network, along with the next word embedding. Each output embedding contains information about the word itself and its context in the preceding sentence fragment. In principle, the final output contains information about the entire sentence and could be used to support classification tasks similarly to the <cls> token in a transformer encoder model. However, RNNs sometimes gradually "forget" about tokens that are further back in time.

word sense disambiguation, and document clustering. NLP was initially tackled by rule-based methods that exploited the structure and statistics of grammar. See Manning & Schutze (1999) and Jurafsky & Martin (2000) for early approaches.

Recurrent neural networks: Before the introduction of transformers, many state-of-the-art NLP applications used *recurrent neural networks*, or *RNNs* for short (figure 12.19). The term "recurrent" was introduced by Rumelhart et al. (1985), but the main idea dates to at least Minsky & Papert (1969). RNNs ingest a sequence of inputs (words in NLP) one at a time. At each step, the network receives both the new input and a hidden representation computed from the previous time step (the recurrent connection). The final output contains information about the whole input. This representation can then support NLP tasks like classification or translation. They have also been used in a decoding context in which generated tokens are fed back into the model to form the next input to the sequence. For example, the PixelRNN (Van den Oord et al., 2016c) used RNNs to build an autoregressive model of images.

From RNNs to transformers: One of the problems with RNNs is that they can forget information that is further back in the sequence. More sophisticated versions of this architecture, such as *long short-term memory networks* or *LSTMs* (Hochreiter & Schmidhuber, 1997b) and *gated recurrent units* or *GRUs* (Cho et al., 2014; Chung et al., 2014) partially addressed this problem. However, in machine translation, the idea emerged that all of the intermediate representations in the RNN could be exploited to produce the output sentence. Moreover, certain output words should *attend* more to certain input words according to their relation (Bahdanau et al., 2015). This ultimately led to dispensing with the recurrent structure and replacing it with the encoder-decoder transformer (Vaswani et al., 2017). Here input tokens attend to one another (self-attention), output tokens attend to those earlier in the sequence (masked self-attention), and output tokens also attend to the input tokens (cross-attention). A formal algorithmic description of the transformer can be found in Phuong & Hutter (2022), and a survey of work can be found in Lin et al. (2022). The literature should be approached with caution, as many enhancements to transformers do not make meaningful performance improvements when carefully assessed in controlled experiments (Narang et al., 2021).

Applications: Models based on self-attention and/or the transformer architecture have been applied to text sequences (Vaswani et al., 2017), image patches (Dosovitskiy et al., 2021), protein sequences (Rives et al., 2021), graphs (Veličković et al., 2019), database schema (Xu et al., 2021b), speech (Wang et al., 2020c), mathematical integration when formulated as a translation problem (Lample & Charton, 2020), and time series (Wu et al., 2020b). However, their most celebrated successes have been in building language models and, more recently, as a replacement for convolutional networks in computer vision.

Large language models: Vaswani et al. (2017) targeted translation tasks, but transformers are now more usually used to build either pure encoder or pure decoder models, the most famous of which are BERT (Devlin et al., 2019) and GPT2/GPT3 (Radford et al., 2019; Brown et al., 2020), respectively. These models are usually tested against benchmarks like GLUE (Wang et al., 2019b), which includes the SQuAD question-answering task (Rajpurkar et al., 2016) described in section 12.6.2, SuperGLUE (Wang et al., 2019a) and BIG-bench (Srivastava et al., 2022), which combine many NLP tasks to create an aggregate score for measuring language ability. Decoder models are generally not fine-tuned for these tasks but can perform well anyway when given a few examples of questions and answers and asked to complete the text from the next question. This is referred to as *few-shot learning* (Brown et al., 2020).

Since GPT3, many decoder language models have been released with steady improvement in few-shot results. These include GLaM (Du et al., 2022), Gopher (Rae et al., 2021), Chinchilla (Hoffmann et al., 2023), Megatron-Turing NLG (Smith et al., 2022), and LaMDa (Thoppilan et al., 2022). Most of the performance improvement is attributable to increased model size, using sparsely activated modules, and exploiting larger datasets. At the time of writing, the most recent model is PaLM (Chowdhery et al., 2022), which has 540 billion parameters and was trained on 780 billion tokens across 6144 processors. Interestingly, since text is highly compressible, this model has more than enough capacity to memorize the entire training dataset. This is true for many language models. Many bold statements have been made about how large language models exceed human performance. This is probably true for some tasks, but such statements should be treated with caution (see Ribeiro et al.,2021; McCoy et al., 2019; Bowman & Dahl, 2021; and Dehghani et al., 2021).

These models have considerable world knowledge. For example, in section 12.7.4, the model knows key facts about deep learning, including that it is a type of machine learning with associated algorithms and applications. Indeed, one such model has been mistakenly identified as being sentient (Clark, 2022). However, there are persuasive arguments that the degree of "understanding" this type of model can ever have is limited (Bender & Koller, 2020).

Tokenizers: Schuster & Nakajima (2012) and Sennrich et al. (2015) introduced *WordPiece* and *byte pair encoding* (*BPE*), respectively. Both methods greedily merge pairs of tokens based on their frequency of adjacency (figure 12.8), with the main difference being how the initial tokens are chosen. For example, in BPE, the initial tokens are characters or punctuation with a special token to denote whitespace. The merges cannot occur over the whitespace. As the algorithm proceeds, new tokens are formed by combining characters recursively so that sub-word and word tokens emerge. The unigram language model (Kudo, 2018) generates several possible candidate merges and chooses the best one based on the likelihood in a language model. Provilkov et al. (2020) develop BPE dropout, which generates the candidates more efficiently by introducing randomness into the process of counting frequencies. Versions of both byte pair encoding and the unigram language model are included in the SentencePiece library (Kudo & Richardson, 2018), which works directly on Unicode characters and can work with any language. He et al. (2020) introduce a method that treats the sub-word segmentation as a latent variable that should be marginalized out for learning and inference.

Decoding algorithms: Transformer decoder models take a body of text and return a probability over the next token. This is then added to the preceding text, and the model is run

again. The process of choosing tokens from these probability distributions is known as *decoding*. Näive ways to do this would be to either (i) greedily choose the most likely token or (ii) choose a token randomly according to the distribution. However, neither of these methods works well in practice. In the former case, the results may be very generic, and the latter case may lead to degraded quality outputs (Holtzman et al., 2020). This is partly because, during training, the model was only exposed to sequences of ground truth tokens (known as *teacher forcing*) but sees its own output when deployed.

It is not computationally feasible to try every combination of tokens in the output sequence, but it is possible to maintain a fixed number of parallel hypotheses and choose the most likely overall sequence. This is known as *beam search*. Beam search tends to produce many similar hypotheses and has been modified to investigate more diverse sequences (Vijayakumar et al., 2016; Kulikov et al., 2018). One possible problem with random sampling is that there is a very long tail of unlikely following words that collectively have a significant probability. This has led to the development of *top-K sampling*, in which tokens are sampled from only the K most likely hypotheses (Fan et al., 2018). Top-K sampling still sometimes allows unreasonable token choices when there are only a few high-probability choices. To resolve this problem, Holtzman et al. (2020) proposed *nucleus sampling*, in which tokens are sampled from a fixed proportion of the total probability mass. El Asri & Prince (2020) discuss decoding algorithms in more depth.

Types of attention: Scaled dot-product attention (Vaswani et al., 2017) is just one of a family of attention mechanisms that includes additive attention (Bahdanau et al., 2015), multiplicative attention (Luong et al., 2015), key-value attention (Daniluk et al., 2017), and memory-compressed attention (Liu et al., 2019c). Zhai et al. (2021) constructed "attention-free" transformers, in which the tokens interact in a way that does not have quadratic complexity. Multi-head attention was also introduced by Vaswani et al. (2017). Interestingly, it appears that most of the heads can be pruned after training without critically affecting the performance (Voita et al., 2019); it has been suggested that their role is to guard against bad initializations. Hu et al. (2018b) propose squeeze-and-excitation networks, attention-like mechanisms that re-weight the channels in a convolutional layer based on globally computed features.

Relationship of self-attention to other models: The self-attention computation has close connections to other models. First, it is an example of a hypernetwork (Ha et al., 2017) in that it uses one part of the network to choose the weights of another part: the attention matrix forms the weights of a sparse network layer that maps the values to the outputs (figure 12.3). The *synthesizer* (Tay et al., 2021) simplifies this idea by simply using a neural network to create each row of the attention matrix from the corresponding input. Even though the input tokens no longer interact with each other to create the attention weights, this works surprisingly well. Wu et al. (2019) present a similar system that produces an attention matrix with a convolutional structure so the tokens attend to their neighbors. The gated multi-layer perceptron (Wu et al., 2019) computes a matrix that pointwise multiplies the values and hence modifies them without mixing them. Transformers are also closely related to *fast weight memory systems*, which were the intellectual forerunners of hypernetworks (Schlag et al., 2021).

Self-attention can also be thought of as a routing mechanism (figure 12.1), and from this viewpoint, there is a connection to capsule networks (Sabour et al., 2017). These capture hierarchical relations in images; lower network levels might detect facial parts (noses, mouths), which are then combined (routed) in higher-level capsules that represent a face. However, capsule networks use *routing by agreement*. In self-attention, the inputs compete with each other for how much they contribute to a given output (via the softmax operation). In capsule networks, the outputs of the layer compete with each other for inputs from earlier layers. Once we consider self-attention as a routing network, we can question whether making this routing dynamic (i.e., dependent on the data) is necessary. The random synthesizer (Tay et al., 2021) removed the dependence of the attention matrix on the inputs entirely and either used predetermined random values or learned values. This performed surprisingly well across a variety of tasks.

Multi-head self-attention also has close connections to graph neural networks (see chapter 13), convolution (Cordonnier et al., 2020), recurrent neural networks (Choromanski et al., 2020), and memory retrieval in Hopfield networks (Ramsauer et al., 2021). For more information on the relationships between transformers and other models, consult Prince (2021a).

Positional encoding: The original transformer paper (Vaswani et al., 2017) experimented with predefining the positional encoding matrix $\mathbf{\Pi}$, and learning the positional encoding $\mathbf{\Pi}$. It might seem odd to *add* the positional encodings to the $D \times N$ data matrix $\mathbf{X}$ rather than concatenate them. However, the data dimension D is usually greater than the number of tokens N, so the positional encoding lies in a subspace. The word embeddings in $\mathbf{X}$ are learned, so the system can theoretically keep the two components in orthogonal subspaces and retrieve the positional encodings as required. The predefined embeddings chosen by Vaswani et al. (2017) were a family of sinusoidal components with two attractive properties: (i) the relative position of two embeddings is easy to recover using a linear operation and (ii) their dot product generally decreased as the distance between positions increased (see Prince, 2021a, for more details). Many systems, such as GPT3 and BERT, learn positional encodings. Wang et al. (2020a) examined the cosine similarities of the positional encodings in these models and showed that they generally decline with relative distance, although they also have a periodic component.

Much subsequent work has modified just the attention matrix so that in the scaled dot-product self-attention equation:

$$\mathbf{Sa}[\mathbf{X}] = \mathbf{V} \cdot \mathbf{Softmax}\left[\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{D_q}}\right], \tag{12.16}$$

only the queries and keys contain position information:

$$\begin{aligned}\mathbf{V} &= \boldsymbol{\beta}_v\mathbf{1}^\mathbf{T} + \mathbf{\Omega}_v\mathbf{X} \\ \mathbf{Q} &= \boldsymbol{\beta}_q\mathbf{1}^\mathbf{T} + \mathbf{\Omega}_q(\mathbf{X}+\mathbf{\Pi}) \\ \mathbf{K} &= \boldsymbol{\beta}_k\mathbf{1}^\mathbf{T} + \mathbf{\Omega}_k(\mathbf{X}+\mathbf{\Pi}).\end{aligned} \tag{12.17}$$

This has led to the idea of multiplying out the quadratic component in the numerator of equation 12.16 and retaining only some of the terms. For example, Ke et al. (2021) decouple or *untie* the content and position information by retaining only the content-content and position-position terms and using different projection matrices $\mathbf{\Omega}_\bullet$ for each.

Another modification is to inject information directly about the relative position. This is more important than absolute position since a batch of text can start at an arbitrary place in a document. Shaw et al. (2018), Raffel et al. (2020), and Huang et al. (2020b) all developed systems where a single term was learned for each relative position offset, and the attention matrix was modified in various ways using these *relative positional encodings*. Wei et al. (2019) investigated relative positional encodings based on predefined sinusoidal embeddings rather than learned values. DeBERTa (He et al., 2021) combines these ideas; they retain only a subset of terms from the quadratic expansion, apply different projection matrices to them, and use relative positional encodings. Other work has explored sinusoidal embeddings that encode absolute and relative position information in more complex ways (Su et al., 2021).

Wang et al. (2020a) compare the performance of transformers in BERT with different positional encodings. They found that relative positional encodings perform better than absolute positional encodings, but there was little difference between using sinusoidal and learned embeddings. A survey of positional encodings can be found in Dufter et al. (2021).

Extending transformers to longer sequences: The complexity of the self-attention mechanism increases quadratically with the sequence length. Some tasks like summarization or

question answering may require long inputs, so this quadratic dependence limits performance. Three lines of work have attempted to address this problem. The first decreases the size of the attention matrix, the second makes the attention sparse, and the third modifies the attention mechanism to make it more efficient.

To decrease the size of the attention matrix, Liu et al. (2018b) introduced *memory-compressed attention*. This applies strided convolution to the keys and values, which reduces the number of positions in a very similar way to downsampling in a convolutional network. Attention is now applied between weighted combinations of neighboring positions, where the weights are learned. Along similar lines, Wang et al. (2020b) observed that the quantities in the attention mechanism are often low rank in practice and developed the *LinFormer*, which projects the keys and values onto a smaller subspace before computing the attention matrix.

To make attention sparse, Liu et al. (2018b) proposed *local attention*, in which neighboring blocks of tokens only attend to one another. This creates a block diagonal interaction matrix (see figure 12.15). Information cannot pass from block to block, so such layers are typically alternated with full attention. Along the same lines, GPT3 (Brown et al., 2020) uses a convolutional interaction matrix and alternates this with full attention. Child et al. (2019) and Beltagy et al. (2020) experimented with various interaction matrices, including convolutional structures with different dilation rates but allowing some queries to interact with every other key. Ainslie et al. (2020) introduced the *extended transformer construction* (figure 12.15h), which uses a set of global embeddings that interact with every other token. This can only be done in the encoder version, or these implicitly allow the system to "look ahead." When combined with relative position encoding, this scheme requires special encodings for mapping to, from, and between these global embeddings. *BigBird* (Ainslie et al., 2020) combined global embeddings and a convolutional structure with a random sampling of possible connections. Other work has investigated learning the sparsity pattern of the attention matrix (Roy et al., 2021; Kitaev et al., 2020; Tay et al., 2020).

Finally, it has been noted that the terms in the numerator and denominator of the softmax operation that computes attention have the form $\exp[\mathbf{k}^T\mathbf{q}]$. This can be treated as a kernel function and, as such, can be expressed as the dot product $\mathbf{g}[\mathbf{k}]^T\mathbf{g}[\mathbf{q}]$ where $\mathbf{g}[\bullet]$ is a nonlinear transformation. This formulation decouples the queries and keys, making the attention computation more efficient. Unfortunately, to replicate the form of the exponential terms, the transformation $\mathrm{g}[\bullet]$ must map the inputs to the infinite space. The linear transformer (Katharopoulos et al., 2020) recognizes this and replaces the exponential term with a different similarity measure. The *Performer* (Choromanski et al., 2020) approximates this infinite mapping with a finite-dimensional one. More details about extending transformers to longer sequences can be found in Tay et al. (2023) and Prince (2021a).

Problem 12.10

Training transformers: Training transformers is challenging and requires both learning rate warm-up (Goyal et al., 2018) and Adam (Kingma & Ba, 2015). Indeed Xiong et al. (2020a) and Huang et al. (2020a) show that the gradients vanish, and the Adam updates decrease in magnitude without learning rate warm-up. Several interacting factors cause this problem. Residual connections cause the exploding gradients (figure 11.6), but normalization layers prevent this. Vaswani et al. (2017) used LayerNorm rather than BatchNorm because NLP statistics are highly variable between batches, although subsequent work has modified BatchNorm for transformers (Shen et al., 2020a). The positioning of the LayerNorm outside of the residual block causes gradients to shrink as they pass back through the network (Xiong et al., 2020a). In addition, the relative weight of the residual connections and main self-attention mechanism varies as we move through the network upon initialization (see figure 11.6c). There is the additional complication that the gradients for the query and key parameters are smaller than for the value parameters (Liu et al., 2020), which necessitates the use of Adam. These factors interact in a complex way, making training unstable and necessitating learning rate warm-up.

There have been various attempts to stabilize training, including (i) a variation of FixUp called *TFixup* (Huang et al., 2020a) that allows the LayerNorm components to be removed, (ii) chang-

ing the position of the LayerNorm components in the network (Liu et al., 2020), and (iii) re-weighting the two paths in the residual branches (Liu et al., 2020; Bachlechner et al., 2021). Xu et al. (2021b) introduced an initialization scheme called *DTFixup* that allows transformers to be trained with smaller datasets. A detailed discussion can be found in Prince (2021b).

Applications in vision: ImageGPT (Chen et al., 2020a) and the Vision Transformer (Dosovitskiy et al., 2021) were both early transformer architectures applied to images. Transformers have been used for image classification (Dosovitskiy et al., 2021; Touvron et al., 2021), object detection (Carion et al., 2020; Zhu et al., 2020b; Fang et al., 2021), semantic segmentation (Ye et al., 2019; Xie et al., 2021; Gu et al., 2022), super-resolution (Yang et al., 2020a), action recognition (Sun et al., 2019; Girdhar et al., 2019), image generation (Chen et al., 2021b; Nash et al., 2021), visual question answering (Su et al., 2019b; Tan & Bansal, 2019), inpainting (Wan et al., 2021; Zheng et al., 2021; Zhao et al., 2020b; Li et al., 2022), colorization (Kumar et al., 2021), and many other vision tasks (Khan et al., 2022; Liu et al., 2023b).

Transformers and convolutional networks: Transformers have been combined with convolutional neural networks for many tasks, including image classification (Wu et al., 2020a), object detection (Hu et al., 2018a; Carion et al., 2020), video processing (Wang et al., 2018c; Sun et al., 2019), unsupervised object discovery (Locatello et al., 2020) and various text/vision tasks (Chen et al., 2020d; Lu et al., 2019; Li et al., 2019). Transformers can outperform convolutional networks for vision tasks but usually require large quantities of data to achieve superior performance. Often, they are pre-trained on enormous datasets like JRT (Sun et al., 2017) and LAION (Schuhmann et al., 2021). The transformer doesn't have the inductive bias of convolutional networks, but by using huge amounts of data, it can surmount this disadvantage.

From pixels to video: Non-local networks (Wang et al., 2018c) were an early application of self-attention to image data. Transformers were initially applied to pixels in local neighborhoods (Parmar et al., 2018; Hu et al., 2019; Parmar et al., 2019; Zhao et al., 2020a). ImageGPT (Chen et al., 2020a) scaled this to model all pixels in a small image. The Vision Transformer (ViT) (Dosovitskiy et al., 2021) used non-overlapping patches to analyze bigger images.

Since then, many multi-scale systems have been developed, including the SWin transformer (Liu et al., 2021c), SWinV2 (Liu et al., 2022), multi-scale transformers (MViT) (Fan et al., 2021), and pyramid vision transformers (Wang et al., 2021). The Crossformer (Wang et al., 2022b) models interactions between spatial scales. Ali et al. (2021) introduced cross-covariance image transformers, in which the channels rather than spatial positions attend to one another, hence making the size of the attention matrix indifferent to the image size. The dual attention vision transformer (DaViT) was developed by Ding et al. (2022) and alternates between local spatial attention within sub-windows and spatially global attention between channels. Chu et al. (2021) similarly alternate between local attention within sub-windows and global attention by subsampling the spatial domain. Dong et al. (2022) adapt the ideas of figure 12.15, in which the interactions between elements are sparsified to the 2D image domain.

Transformers were subsequently adapted to video processing (Arnab et al., 2021; Bertasius et al., 2021; Liu et al., 2021c; Neimark et al., 2021; Patrick et al., 2021). A survey of transformers applied to video can be found in Selva et al. (2022).

Combining images and text: CLIP (Radford et al., 2021) learns a joint encoder for images and their captions using a contrastive pre-training task. The system ingests N images and their captions and produces a matrix of compatibility between images and captions. The loss function encourages the correct pairs to have a high score and the incorrect pairs to have a low score. Ramesh et al. (2021) and Ramesh et al. (2022) train a diffusion decoder to invert the CLIP image encoder for text-conditional image generation (see chapter 18).

Problems

Problem 12.1 Consider a self-attention mechanism that processes N inputs of length D to produce N outputs of the same size. How many weights and biases are used to compute the queries, keys, and values? How many attention weights $\mathrm{a}[\bullet, \bullet]$ will there be? How many weights and biases would there be in a fully connected shallow network relating all DN inputs to all DN outputs?

Problem 12.2 Why might we want to ensure that the input to the self-attention mechanism is the same size as the output?

Problem 12.3* Show that the self-attention mechanism (equation 12.8) is equivariant to a permutation $\mathbf{XP}$ of the data $\mathbf{X}$, where $\mathbf{P}$ is a permutation matrix. In other words, show that:

Appendix B.4.4 Permutation matrix

$$\mathbf{Sa}[\mathbf{XP}] = \mathbf{Sa}[\mathbf{X}]\mathbf{P}. \tag{12.18}$$

Problem 12.4 Consider the softmax operation:

$$y_i = \mathrm{softmax}_i[\mathbf{z}] = \frac{\exp[z_i]}{\sum_{j=1}^{5}\exp[z_j]}, \tag{12.19}$$

in the case where there are five inputs with values: $z_1 = -3$, $z_2 = 1$, $z_3 = 100$, $z_4 = 5$, $z_5 = -1$. Compute the 25 derivatives, $\partial y_i/\partial z_j$ for all $i, j \in \{1, 2, 3, 4, 5\}$. What do you conclude?

Problem 12.5 Why is implementation more efficient if the values, queries, and keys in each of the H heads each have dimension D/H where D is the original dimension of the data?

Problem 12.6 BERT was pre-trained using two tasks. The first task requires the system to predict missing (masked) words. The second task requires the system to classify pairs of sentences as being adjacent or not in the original text. Identify whether each of these tasks is generative or contrastive (see section 9.3.6). Why do you think they used two tasks? Propose two novel contrastive tasks that could be used to pre-train a language model.

Problem 12.7 Consider adding a new token to a precomputed masked self-attention mechanism with N tokens. Describe the *extra* computation that must be done to incorporate this new token.

Problem 12.8 Computation in vision transformers expands quadratically with the number of patches. Devise two methods to reduce the computation using the principles from figure 12.15.

Problem 12.9 Consider representing an image with a grid of 16×16 patches, each represented by a patch embedding of length 512. Compare the amount of computation required in the DaViT transformer to perform attention (i) between the patches, using all of the channels, and (ii) between the channels, using all of the patches.

Problem 12.10* Attention weights are usually computed as:

$$a[\mathbf{x}_m, \mathbf{x}_n] = \mathrm{softmax}_m\left[\mathbf{k}_\bullet^T\mathbf{q}_n\right] = \frac{\exp\left[\mathbf{k}_m^T\mathbf{q}_n\right]}{\sum_{m'=1}^{N}\exp\left[\mathbf{k}_{m'}^T\mathbf{q}_n\right]}. \tag{12.20}$$

Consider replacing $\exp\left[\mathbf{k}_m^T\mathbf{q}_n\right]$ with the dot product $\mathbf{g}[\mathbf{k}_m]^T\mathbf{g}[\mathbf{q}_n]$ where $\mathbf{g}[\bullet]$ is a nonlinear transformation. Show how this makes the computation of the attention weights more efficient.

Chapter 13

Graph neural networks

Chapter 10 described convolutional networks, which specialize in processing regular arrays of data (e.g., images). Chapter 12 described transformers, which specialize in processing sequences of variable length (e.g., text). This chapter describes *graph neural networks*. As the name suggests, these are neural architectures that process graphs (i.e., sets of nodes connected by edges).

There are three novel challenges associated with processing graphs. First, their topology is variable, and it is hard to design networks that are both sufficiently expressive and can cope with this variation. Second, graphs may be enormous; a graph representing connections between users of a social network might have a billion nodes. Third, there may only be a single monolithic graph available, so the usual protocol of training with many data examples and testing with new data is not always appropriate.

This chapter starts by presenting real-world examples of graphs. It then describes how to encode these graphs and how to formulate supervised learning problems for graphs. The algorithmic requirements for processing graphs are discussed, and these lead naturally to *graph convolutional networks*, a particular type of graph neural network.

13.1 What is a graph?

A graph is a very general structure and consists of a set of *nodes* or *vertices*, where pairs of nodes are connected by *edges* or *links*. Graphs are typically sparse; only a small subset of the possible edges are present.

Some objects in the real world naturally take the form of graphs. For example, road networks can be considered graphs where the nodes are physical locations, and the edges represent roads between them (figure 13.1a). Chemical molecules are small graphs where the nodes represent atoms, and the edges represent chemical bonds (figure 13.1b). Electrical circuits are graphs where the nodes represent components and junctions, and the edges are electrical connections (figure 13.1c).

Furthermore, many datasets can also be represented by graphs, even if this is not their obvious surface form. For example:

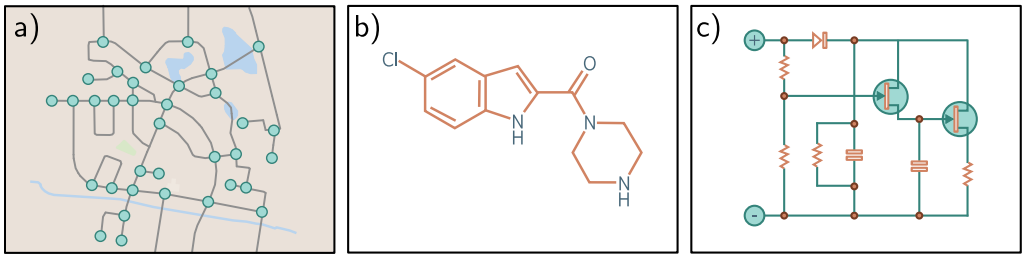

Figure 13.1 Real-world graphs. Some objects, such as a) road networks, b) molecules, and c) electrical circuits, are naturally structured as graphs.

- Social networks are graphs where nodes are people, and the edges represent friendships between them.
- The scientific literature can be viewed as a graph where the nodes are papers, and the edges represent citations.
- Wikipedia can be considered a graph where the nodes are articles, and the edges represent hyperlinks between articles.
- Computer programs can be represented as graphs where the nodes are syntax tokens (variables at different points in the program flow), and the edges represent computations involving these variables.
- Geometric point clouds can be represented as graphs. Here, each point is a node with edges connecting to other nearby points.
- Protein interactions in a cell can be expressed as graphs, where the nodes are the proteins, and there is an edge between two proteins if they interact.

In addition, a set (an unordered list) can be treated as a graph in which every member is a node and connects to every other. An image can be treated as a graph with regular topology, in which each pixel is a node with edges to the adjacent pixels.

13.1.1 Types of graphs

Graphs can be categorized in various ways. The social network in figure 13.2a contains *undirected edges*; each pair of individuals with a connection between them have mutually agreed to be friends, so there is no sense that the relationship is directional. In contrast, the citation network in figure 13.2b contains *directed edges*. Each paper cites other papers, and this relationship is inherently one-way.

Figure 13.2c depicts a *knowledge graph* that encodes a set of facts about objects by defining relations between them. Technically, this is a *directed heterogeneous multigraph*. It is heterogeneous because the nodes can represent different types of entities (e.g., people, countries, companies). It is a multigraph because there can be multiple edges of different types between any two nodes.

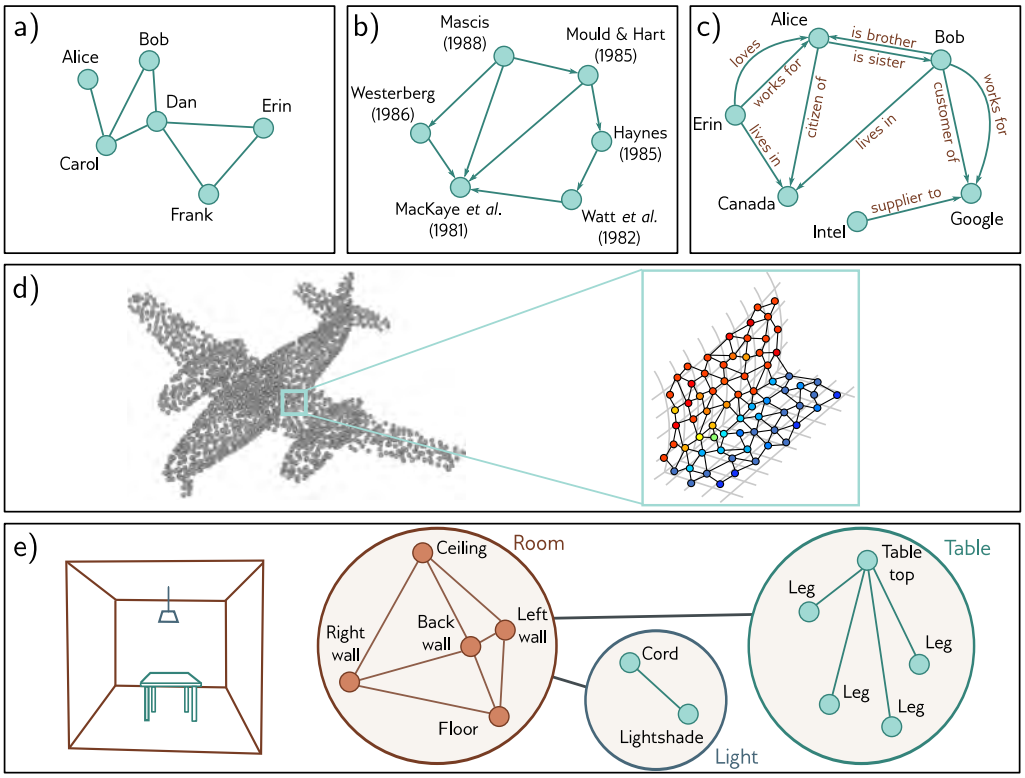

Figure 13.2 Types of graphs. a) A social network is an undirected graph; the connections between people are symmetric. b) A citation network is a directed graph; one publication cites another, so the relationship is asymmetric. c) A knowledge graph is a directed heterogeneous multigraph. The nodes are heterogeneous in that they represent different object types (people, places, companies) and multiple edges may represent different relations between each node. d) A point set can be converted to a graph by forming edges between nearby points. Each node has an associated position in 3D space, and this is termed a geometric graph (adapted from Hu et al., 2022). e) The scene on the left can be represented by a hierarchical graph. The topology of the room, table, and light are all represented by graphs. These graphs form nodes in a larger graph representing object adjacency (adapted from Fernández-Madrigal & González, 2002).

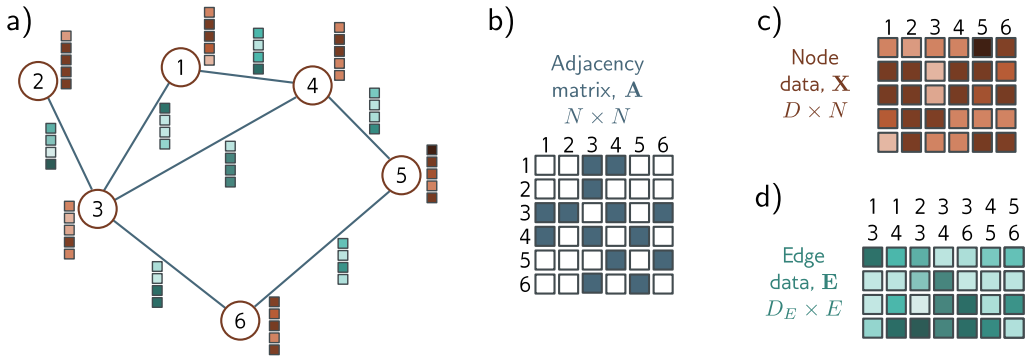

Figure 13.3 Graph representation. a) Example graph with six nodes and seven edges. Each node has an associated embedding of length five (brown vectors). Each edge has an associated embedding of length four (blue vectors). This graph can be represented by three matrices. b) The adjacency matrix is a binary matrix where element (m, n) is set to one if node m connects to node n. c) The node data matrix $\mathbf{X}$ contains the concatenated node embeddings. d) The edge data matrix $\mathbf{E}$ contains the edge embeddings.

The point set representing the airplane in figure 13.2d can be converted into a graph by connecting each point to its K nearest neighbors. The result is a *geometric graph* where each point is associated with a position in 3D space. Figure 13.2e represents a *hierarchical graph*. The table, light, and room are each described by graphs representing the adjacency of their respective components. These three graphs are themselves nodes in another graph that represents the topology of the objects in a larger model.

All types of graphs can be processed using deep learning. However, this chapter focuses on undirected graphs like the social network in figure 13.2a.

13.2 Graph representation

In addition to the graph structure itself, information is typically associated with each node. For example, in a social network, each individual might be characterized by a fixed-length vector representing their interests. Sometimes, the edges also have information attached. For example, in the road network example, each edge might be characterized by its length, number of lanes, frequency of accidents, and speed limit. The information at a node is stored in a *node embedding*, and the information at an edge is stored in an *edge embedding*.

More formally, a graph consists of a set of N nodes connected by a set of E edges. The graph can be encoded by three matrices $\mathbf{A}$, $\mathbf{X}$, and $\mathbf{E}$, representing the graph structure, node embeddings, and edge embeddings, respectively (figure 13.3).

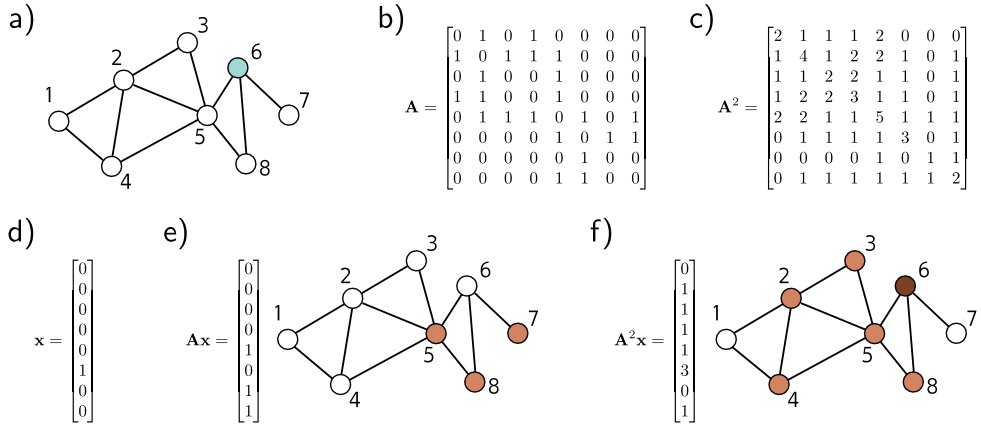

Figure 13.4 Properties of the adjacency matrix. a) Example graph. b) Position (m, n) of the adjacency matrix $\mathbf{A}$ contains the number of walks of length one from node m to node n. c) Position (m, n) of the squared adjacency matrix $\mathbf{A}^2$ contains the number of walks of length two from node n to node m. d) One hot vector representing node six, which was highlighted in panel (a). e) When we pre-multiply this vector by $\mathbf{A}$, the result contains the number of walks of length one from node six to each node; we can reach nodes five, seven, and eight in one move. f) When we pre-multiply this vector by $\mathbf{A}^2$, the resulting vector contains the number of walks of length two from node six to each node; we can reach nodes two, three, four, five, and eight in two moves, and we can return to the original node in three different ways (via nodes five, seven, and eight).

Problems 13.1–13.2

The graph structure is represented by the *adjacency matrix*, $\mathbf{A}$. This is an $N \times N$ matrix where entry (m, n) is set to one if there is an edge between nodes m and n and zero otherwise. For undirected graphs, this matrix is always symmetric. For large sparse graphs, it can be stored as a list of connections (m, n) to save memory.

The n^{th} node has an associated node embedding $\mathbf{x}^{(n)}$ of length D. These embeddings are concatenated and stored in the $D \times N$ node data matrix $\mathbf{X}$. Similarly, the e^{th} edge has an associated edge embedding $\mathbf{e}^{(e)}$ of length D_E. These edge embeddings are collected into the $D_E \times E$ matrix $\mathbf{E}$. For simplicity, we initially consider graphs that only have node embeddings and return to edge embeddings in section 13.9.

13.2.1 Properties of the adjacency matrix

The adjacency matrix can be used to find the neighbors of a node using linear algebra. Consider encoding the n^{th} node as a one-hot column vector (a vector with only one non-zero entry at position n, which is set to one). When we pre-multiply this vector by the adjacency matrix, it extracts the n^{th} column of the adjacency matrix and returns a vector with ones at the positions of the neighbors (i.e., all the places we can reach in a

walk of length one from the n^{th} node). If we repeat this procedure (i.e., pre-multiply by $\mathbf{A}$ again), the resulting vector contains the number of walks of length two from node n to every node (figures 13.4d–f).

Problems 13.3–13.4

In general, if we raise the adjacency matrix to the power of L, the entry at position (m, n) of $\mathbf{A}^L$ contains the number of unique *walks* of length L from node n to node m (figures 13.4a–c). This is not the same as the number of unique paths since it includes routes that visit the same node more than once. Nonetheless, $\mathbf{A}^L$ still contains valuable information about the graph connectivity; a non-zero entry at position (m, n) indicates that the distance from m to n must be less than or equal to L.

Notebook 13.1 Encoding graphs

13.2.2 Permutation of node indices

Node indexing in graphs is arbitrary; permuting the node indices results in a permutation of the columns of the node data matrix $\mathbf{X}$ and a permutation of both the rows and columns of the adjacency matrix $\mathbf{A}$. However, the underlying graph is unchanged (figure 13.5). This is in contrast to images, where permuting the pixels creates a different image, and to text, where permuting the words creates a different sentence.

The operation of exchanging node indices can be expressed mathematically by a *permutation matrix*, $\mathbf{P}$. This is a matrix where exactly one entry in each row and column take the value one, and the remaining values are zero. When position (m, n) of the permutation matrix is set to one, it indicates that node m will become node n after the permutation. To map from one indexing to another, we use the operations:

Problem 13.5

$$\begin{aligned}\mathbf{X}' &= \mathbf{X}\mathbf{P} \\ \mathbf{A}' &= \mathbf{P}^T\mathbf{A}\mathbf{P},\end{aligned} \tag{13.1}$$

where post-multiplying by $\mathbf{P}$ permutes the columns and pre-multiplying by $\mathbf{P}^T$ permutes the rows. It follows that any processing applied to the graph should also be indifferent to these permutations. Otherwise, the result will depend on the choice of node indices.

13.3 Graph neural networks, tasks, and loss functions

A graph neural network is a model that takes the node embeddings $\mathbf{X}$ and the adjacency matrix $\mathbf{A}$ as inputs and passes them through a series of K layers. The node embeddings are updated at each layer to create intermediate "hidden" representations $\mathbf{H}_k$ before finally computing output embeddings $\mathbf{H}_K$.

At the start of this network, each column of the input node embeddings $\mathbf{X}$ just contains information about the node itself. At the end, each column of the model output $\mathbf{H}_K$ includes information about the node and its context within the graph. This is similar to word embeddings passing through a transformer network. These represent words at the start, but represent the word meanings in the context of the sentence at the end.

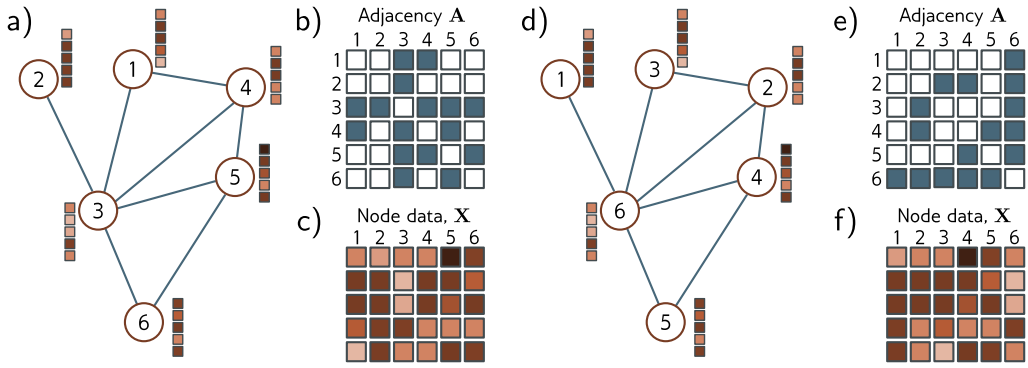

Figure 13.5 Permutation of node indices. a) Example graph, b) associated adjacency matrix and c) node embeddings. d) The same graph where the (arbitrary) order of the indices has been changed. e) The adjacency matrix and f) node matrix are now different. Consequently, any network layer that operates on the graph should be indifferent to the ordering of the nodes.

13.3.1 Tasks and loss functions

We defer discussion of graph neural network models until section 13.4 and first describe the types of problems these networks tackle and their associated loss functions. Supervised graph problems usually fall into one of three categories (figure 13.6).

Graph-level tasks: The network assigns a label or estimates one or more values from the entire graph, exploiting both the structure and node embeddings. For example, we might want to predict the temperature at which a molecule becomes liquid (a regression task) or whether a molecule is poisonous to human beings or not (a classification task).

For graph-level tasks, the output node embeddings are combined (e.g., by averaging), and the resulting vector is mapped via a linear transformation or neural network to a fixed-size vector. For regression, the mismatch between the result and the ground truth values is computed using the least squares loss. For binary classification, the output is passed through a sigmoid function, and the mismatch is calculated using the binary cross-entropy loss. Here, the probability that the graph belongs to class one might be given by:

$$Pr(y=1|\mathbf{X},\mathbf{A}) = \text{sig}\left[\beta_K + \boldsymbol{\omega}_K \mathbf{H}_K \mathbf{1}/N\right], \tag{13.2}$$

where the scalar β_K and $1 \times D$ vector $\boldsymbol{\omega}_K$ are learned parameters. Post-multiplying the output embedding matrix $\mathbf{H}_K$ by the column vector $\mathbf{1}$ that contains ones has the effect of summing together all the embeddings and subsequently dividing by the number of nodes N computes the average. This is known as *mean pooling* (see figure 10.11).

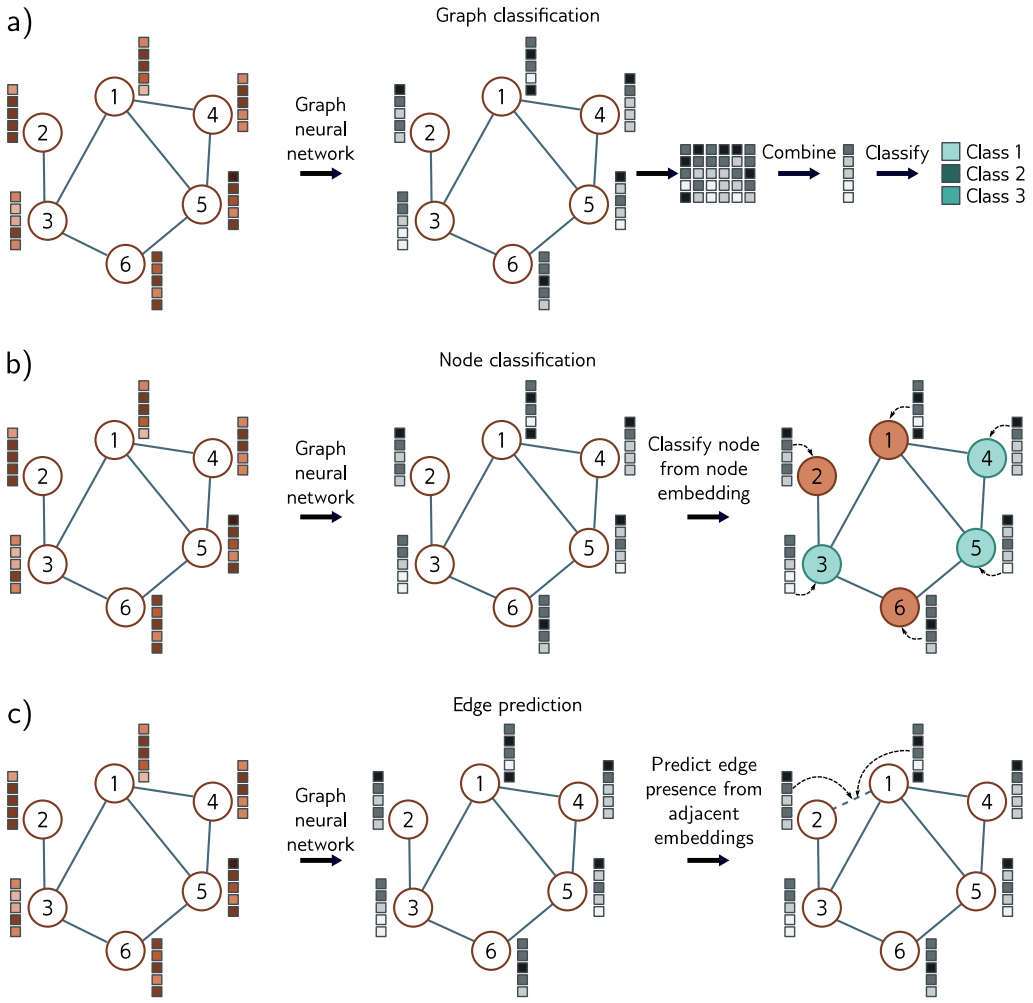

Figure 13.6 Common tasks for graphs. In each case, the input is a graph represented by its adjacency matrix and node embeddings. The graph neural network processes the node embeddings by passing them through a series of layers. The node embeddings at the last layer contain information about both the node and its context in the graph. a) Graph classification. The node embeddings are combined (e.g., by averaging) and then mapped to a fixed-size vector that is passed through a softmax function to produce class probabilities. b) Node classification. Each node embedding is used individually as the basis for classification (cyan and orange colors represent assigned node classes). c) Edge prediction. Node embeddings adjacent to the edge are combined (e.g., by taking the dot product) to compute a single number that is mapped via a sigmoid function to produce a probability that a missing edge should be present.

Node-level tasks: The network assigns a label (classification) or one or more values (regression) to each node of the graph, using both the graph structure and node embeddings. For example, given a graph constructed from a 3D point cloud similar to figure 13.2d, the goal might be to classify the nodes according to whether they belong to the wings or fuselage. Loss functions are defined in the same way as for graph-level tasks, except that now this is done independently at each node n:

$$Pr(y^{(n)} = 1|\mathbf{X}, \mathbf{A}) = \text{sig}\left[\beta_K + \boldsymbol{\omega}_K \mathbf{h}_K^{(n)}\right]. \tag{13.3}$$

Edge prediction tasks: The network predicts whether or not there should be an edge between nodes n and m. For example, in the social network setting, the network might predict whether two people know and like each other and suggest that they connect if that is the case. This is a binary classification task where the two node embeddings must be mapped to a single number representing the probability that the edge is present. One possibility is to take the dot product of the node embeddings and pass the result through a sigmoid function to create the probability:

$$Pr(y^{(mn)} = 1|\mathbf{X}, \mathbf{A}) = \text{sig}\left[\mathbf{h}^{(m)T}\mathbf{h}^{(n)}\right]. \tag{13.4}$$

13.4 Graph convolutional networks

There are many types of graph neural networks, but here we focus on *spatial-based convolutional graph neural networks*, or *GCNs* for short. These models are convolutional in that they update each node by aggregating information from nearby nodes. As such, they induce a *relational inductive bias* (i.e., a bias toward prioritizing information from neighbors). They are spatial-based because they use the original graph structure. This contrasts with *spectral-based methods*, which apply convolutions in the Fourier domain.

Each layer of the GCN is a function $\mathbf{F}[\bullet]$ with parameters $\boldsymbol{\Phi}$ that takes the node embeddings and adjacency matrix and outputs new node embeddings. The network can hence be written as:

$$\begin{aligned}
\mathbf{H}_1 &= \mathbf{F}[\mathbf{X}, \mathbf{A}, \boldsymbol{\phi}_0] \\
\mathbf{H}_2 &= \mathbf{F}[\mathbf{H}_1, \mathbf{A}, \boldsymbol{\phi}_1] \\
\mathbf{H}_3 &= \mathbf{F}[\mathbf{H}_2, \mathbf{A}, \boldsymbol{\phi}_2] \\
\vdots &= \vdots \\
\mathbf{H}_K &= \mathbf{F}[\mathbf{H}_{K-1}, \mathbf{A}, \boldsymbol{\phi}_{K-1}],
\end{aligned} \tag{13.5}$$

where $\mathbf{X}$ is the input, $\mathbf{A}$ is the adjacency matrix, $\mathbf{H}_k$ contains the modified node embeddings at the k^{th} layer, and $\boldsymbol{\phi}_k$ denotes the parameters that map from layer k to layer $k+1$.

13.4.1 Equivariance and invariance

We noted before that the indexing of the nodes in the graph is arbitrary, and any permutation of the node indices does not change the graph. It is hence imperative that any model respects this property. It follows that each layer must be equivariant (see section 10.1) with respect to permutations of the node indices. In other words, if we permute the node indices, the node embeddings at each stage will be permuted in the same way. In mathematical terms, if $\mathbf{P}$ is a permutation matrix, then we must have:

$$\mathbf{H}_{k+1}\mathbf{P} = \mathbf{F}[\mathbf{H}_k\mathbf{P}, \mathbf{P}^T\mathbf{A}\mathbf{P}, \boldsymbol{\phi}_k]. \tag{13.6}$$

For node classification and edge prediction tasks, the output should also be equivariant with respect to permutations of the node indices. However, for graph-level tasks, the final layer aggregates information from across the graph, so the output is invariant to the node order. In fact, the output layer from equation 13.2 achieves this because:

Problem 13.6

$$y = \text{sig}\left[\beta_K + \boldsymbol{\omega}_K\mathbf{H}_K\mathbf{1}/N\right] = \text{sig}\left[\beta_K + \boldsymbol{\omega}_K\mathbf{H}_K\mathbf{P}\mathbf{1}/N\right], \tag{13.7}$$

for any permutation matrix $\mathbf{P}$ (see problem 13.6).

This mirrors the case for images, where segmentation should be equivariant to geometric transformations, and image classification should be invariant (figure 10.1). Here, convolutional and pooling layers partially achieve this with respect to translations, but there is no known way to guarantee these properties exactly for more general transformations. However, for graphs, it is possible to define networks that ensure equivariance or invariance to permutations.

13.4.2 Parameter sharing

Chapter 10 argued applying fully connected networks to images isn't sensible because this requires the network to learn how to recognize an object separately at every image position. Instead, we used convolutional layers that processed every position in the image identically. This reduced the number of parameters and introduced an inductive bias that forced the model to treat every part of the image in the same way.

The same argument can be made about nodes in a graph. We could learn a model with separate parameters associated with each node. However, now the network must independently learn the meaning of the connections in the graph at each position, and training would require many graphs with the same topology. Instead, we build a model that uses the same parameters at every node, reducing the number of parameters and sharing what the network learns at each node across the entire graph.

Recall that a convolution (equation 10.3) updates a variable by taking a weighted sum of information from its neighbors. One way to think of this is that each neighbor sends a message to the variable of interest, which aggregates these messages to form the update. When we considered images, the neighbors were pixels from a fixed-size square region around the current position, so the spatial relationships at each position are the same. However, in a graph, each node may have a different number of neighbors, and there are no consistent relationships; there is no sense that we can weight information

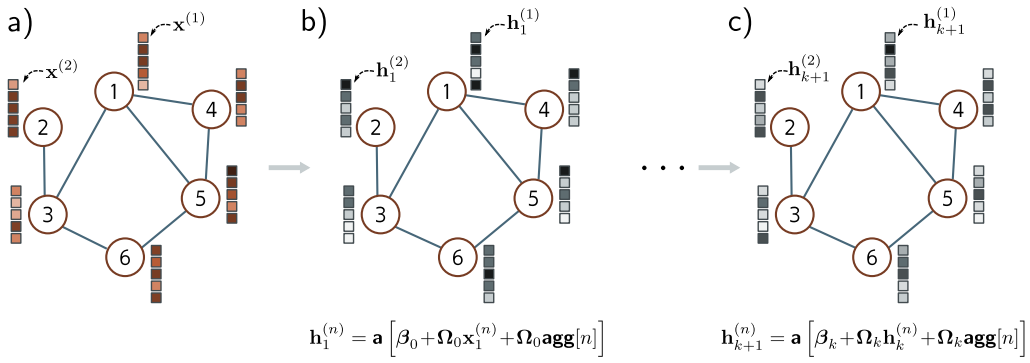

Figure 13.7 Simple Graph CNN layer. a) Input graph consists of structure (embodied in graph adjacency matrix $\mathbf{A}$, not shown) and node embeddings (stored in columns of $\mathbf{X}$). b) Each node in the first hidden layer is updated by (i) aggregating the neighboring nodes to form a single vector, (ii) applying a linear transformation $\boldsymbol{\Omega}_0$ to the aggregated nodes, (iii) applying the same linear transformation $\boldsymbol{\Omega}_0$ to the original node, (iv) adding these together with a bias $\boldsymbol{\beta}_0$, and finally (v) applying a nonlinear activation function $\mathbf{a}[\bullet]$ like a ReLU. c) This process is repeated at subsequent layers (but with different parameters for each layer) until we produce the final embeddings at the end of the network.

from a node that is "above" the node of interest differently to information from a node that is "below" it.

13.4.3 Example GCN layer

These considerations lead to a simple GCN layer (figure 13.7). At each node n in layer k, we aggregate information from neighboring nodes by summing their node embeddings $\mathbf{h}_\bullet$:

$$\mathbf{agg}[n,k] = \sum_{m \in \mathrm{ne}[n]} \mathbf{h}_k^{(m)}, \tag{13.8}$$

where $\mathrm{ne}[n]$ returns the set of indices of the neighbors of node n. Then we apply a linear transformation $\boldsymbol{\Omega}_k$ to the embedding $\mathbf{h}_k^{(n)}$ at the current node and to this aggregated value, add a bias term $\boldsymbol{\beta}_k$, and pass the result through a nonlinear activation function $\mathbf{a}[\bullet]$, which is applied independently to every member of its vector argument:

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \cdot \mathbf{h}_k^{(n)} + \boldsymbol{\Omega}_k \cdot \mathbf{agg}[n,k]\right]. \tag{13.9}$$

We can write this more succinctly by noting that post-multiplication of a matrix by a vector returns a weighted sum of its columns. The n^{th} column of the adjacency matrix $\mathbf{A}$ contains ones at the positions of the neighbors. Hence, if we collect the node

embeddings into the $D \times N$ matrix $\mathbf{H}_k$ and post-multiply by the adjacency matrix $\mathbf{A}$, the n^{th} column of the result is $\mathbf{agg}[n, k]$. The update for the nodes is now:

$$\begin{aligned}\mathbf{H}_{k+1} &= \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k + \boldsymbol{\Omega}_k \mathbf{H}_k \mathbf{A}\right] \\ &= \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k (\mathbf{A} + \mathbf{I})\right], \end{aligned} \tag{13.10}$$

where $\mathbf{1}$ is an $N \times 1$ vector containing ones. Here, the nonlinear activation function $\mathbf{a}[\bullet]$ is applied independently to every member of its matrix argument.

This layer satisfies the design considerations: it is equivariant to permutations of the node indices, can cope with any number of neighbors, exploits the graph structure to provide a relational inductive bias, and shares parameters throughout the graph.

Problem 13.7

13.5 Example: graph classification

We now combine these ideas to describe a network that classifies molecules as toxic or harmless. The network inputs are the adjacency matrix and node embedding matrix $\mathbf{X}$. The adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ derives from the molecular structure. The columns of the node embedding matrix $\mathbf{X} \in \mathbb{R}^{118 \times N}$ are one-hot vectors indicating which of the 118 elements of the periodic table are present. In other words, they are vectors of length 118 where every position is zero except for the position corresponding to the relevant element, which is set to one. The node embeddings can be transformed to an arbitrary size D by the first weight matrix $\boldsymbol{\Omega}_0 \in \mathbb{R}^{D \times 118}$.

Notebook 13.2
Graph classification

The network equations are:

$$\begin{aligned}\mathbf{H}_1 &= \mathbf{a}\left[\boldsymbol{\beta}_0 \mathbf{1}^T + \boldsymbol{\Omega}_0 \mathbf{X}(\mathbf{A} + \mathbf{I})\right] \\ \mathbf{H}_2 &= \mathbf{a}\left[\boldsymbol{\beta}_1 \mathbf{1}^T + \boldsymbol{\Omega}_1 \mathbf{H}_1 (\mathbf{A} + \mathbf{I})\right] \\ \vdots &= \vdots \\ \mathbf{H}_K &= \mathbf{a}\left[\boldsymbol{\beta}_{K-1} \mathbf{1}^T + \boldsymbol{\Omega}_{K-1} \mathbf{H}_{k-1} (\mathbf{A} + \mathbf{I})\right] \\ \mathrm{f}[\mathbf{X}, \mathbf{A}, \boldsymbol{\Phi}] &= \mathrm{sig}\left[\beta_K + \boldsymbol{\omega}_K \mathbf{H}_K \mathbf{1}/N\right], \end{aligned} \tag{13.11}$$

where the network output $\mathrm{f}[\mathbf{X}, \mathbf{A}, \boldsymbol{\Phi}]$ is a single value that determines the probability that the molecule is toxic (see equation 13.2).

13.5.1 Training with batches

Given I training graphs $\{\mathbf{X}_i, \mathbf{A}_i\}$ and their labels y_i, the parameters $\boldsymbol{\Phi} = \{\boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}_{k=0}^K$ can be learned using SGD and the binary cross-entropy loss (equation 5.19). Fully connected networks, convolutional networks, and transformers all exploit the parallelism of modern hardware to process an entire batch of training examples concurrently. To this end, the batch elements are concatenated into a higher-dimensional tensor (section 7.4.2).

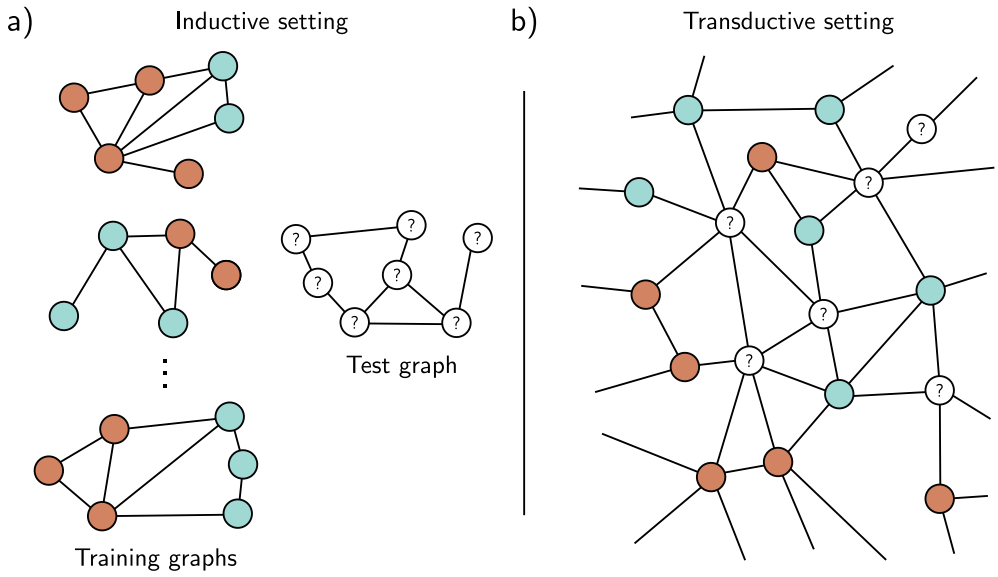

Figure 13.8 Inductive vs. transductive problems. a) Node classification task in the inductive setting. We are given a set of I training graphs, where the node labels (orange and cyan colors) are known. After training, we are given a test graph and must assign labels to each node. b) Node classification in the transductive setting. There is one large graph in which some nodes have labels (orange and cyan colors), and others are unknown. We train the model to predict the known labels correctly and then examine the predictions at the unknown nodes.

However, each graph may have a different number of nodes. Hence, the matrices $\mathbf{X}_i$ and $\mathbf{A}_i$ have different sizes, and there is no way to concatenate them into 3D tensors.

Luckily, a simple trick allows us to process the whole batch in parallel. The graphs in the batch are treated as disjoint components of a single large graph. The network can then be run as a single instance of the network equations. The mean pooling is carried out only over the individual graphs to make a single representation per graph that can be fed into the loss function.

13.6 Inductive vs. transductive models

Until this point, all of the models in this book have been *inductive*: we exploit a training set of labeled data to learn the relation between the inputs and outputs. Then we apply this to new test data. One way to think of this is that we are learning the rule that maps inputs to outputs and then applying it elsewhere.

By contrast, a *transductive* model considers both the labeled and unlabeled data

at the same time. It does not produce a rule but merely a labeling for the unknown outputs. This is sometimes termed *semi-supervised learning*. It has the advantage that it can use patterns in the unlabeled data to help make its decisions. However, it has the disadvantage that the model needs to be retrained when extra unlabeled data are added.

Both problem types are commonly encountered for graphs (figure 13.8). Sometimes, we have many labeled graphs and learn a mapping between the graph and the labels. For example, we might have many molecules, each labeled according to whether it is toxic to humans. We learn the rule that maps the graph to the toxic/non-toxic label and then apply this rule to new molecules. However, sometimes there is a single monolithic graph. In the graph of scientific paper citations, we might have labels indicating the field (physics, biology, etc.) for some nodes and wish to label the remaining nodes. Here, the training and test data are irrevocably connected.

Graph-level tasks only occur in the inductive setting where there are training and test graphs. However, node-level tasks and edge prediction tasks can occur in either setting. In the transductive case, the loss function minimizes the mismatch between the model output and the ground truth where this is known. New predictions are computed by running the forward pass and retrieving the results where the ground truth is unknown.

13.7 Example: node classification

As a second example, consider a binary node classification task in a transductive setting. We start with a commercial-sized graph with millions of nodes. Some nodes have ground truth binary labels, and the goal is to label the remaining unlabeled nodes. The body of the network will be the same as in the previous example (equation 13.11) but with a different final layer that produces an output vector of size $1 \times N$:

$$\mathbf{f}[\mathbf{X}, \mathbf{A}, \boldsymbol{\Phi}] = \mathbf{sig}\left[\beta_K \mathbf{1}^T + \boldsymbol{\omega}_K \mathbf{H}_K\right], \tag{13.12}$$

where the function $\mathbf{sig}[\bullet]$ applies the sigmoid function independently to every element of the row vector input. As usual, we use the binary cross-entropy loss, but now only at nodes where we know the ground truth label y. Note that equation 13.12 is just a vectorized version of the node classification loss from equation 13.3.

Training this network raises two problems. First, it is logistically difficult to train a graph neural network of this size. Consider that we must store the node embeddings at every network layer in the forward pass. This will involve both storing and processing a structure several times the size of the entire graph, and this may not be practical. Second, we have only a single graph, so it's not obvious how to perform stochastic gradient descent. How can we form a batch if there is only a single object?

13.7.1 Choosing batches

One way to form a batch is to choose a random subset of labeled nodes at each training step. Each node depends on its neighbors in the previous layer. These, in turn, depend

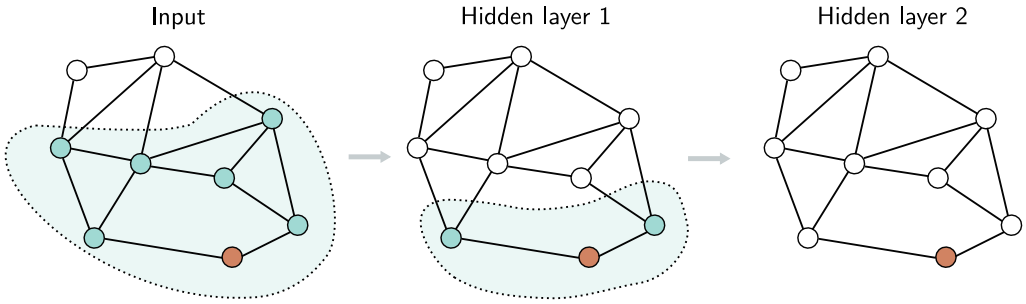

Figure 13.9 Receptive fields in graph neural networks. Consider the orange node in hidden layer two (right). This receives input from the nodes in the 1-hop neighborhood in hidden layer one (shaded region in center). These nodes in hidden layer one receive inputs from their neighbors in turn, and the orange node in layer two receives inputs from all the input nodes in the 2-hop neighborhood (shaded area on left). The region of the graph that contributes to a given node is equivalent to the notion of a receptive field in convolutional neural networks.

on their neighbors in the layer before, so each node has the equivalent of a receptive field (figure 13.9). The size of the receptive field is termed the *k-hop neighborhood*. We can hence perform a gradient descent step using the graph that forms the union of the k-hop neighborhoods of the batch nodes; the remaining inputs do not contribute.

Unfortunately, if there are many layers and the graph is densely connected, every input node may be in the receptive field of every output, and this may not reduce the graph size at all. This is known as the *graph expansion problem*. Two approaches that tackle this problem are *neighborhood sampling* and *graph partitioning*.

Neighborhood sampling: The full graph that feeds into the batch of nodes is sampled, thereby reducing the connections at each network layer (figure 13.10). For example, we might start with the batch nodes and randomly sample a fixed number of their neighbors in the previous layer. Then, we randomly sample a fixed number of *their* neighbors in the layer before, and so on. The graph still increases in size with each layer but in a much more controlled way. This is done anew for each batch, so the contributing neighbors differ even if the same batch is drawn twice. This is also reminiscent of dropout (section 9.3.3) and adds some regularization.

Notebook 13.3 Neighborhood sampling

Graph partitioning: A second approach is to cluster the original graph into disjoint subsets of nodes (i.e., smaller graphs that are not connected to one another) before processing (figure 13.11). There are standard algorithms to choose these subsets to maximize the number of internal links. These smaller graphs can each be treated as batches, or a random subset of them can be combined to form a batch (reinstating any edges between them from the original graph).

Given one of the above methods to form batches, we can now train the network parameters in the same way as for the inductive setting, dividing the labeled nodes into

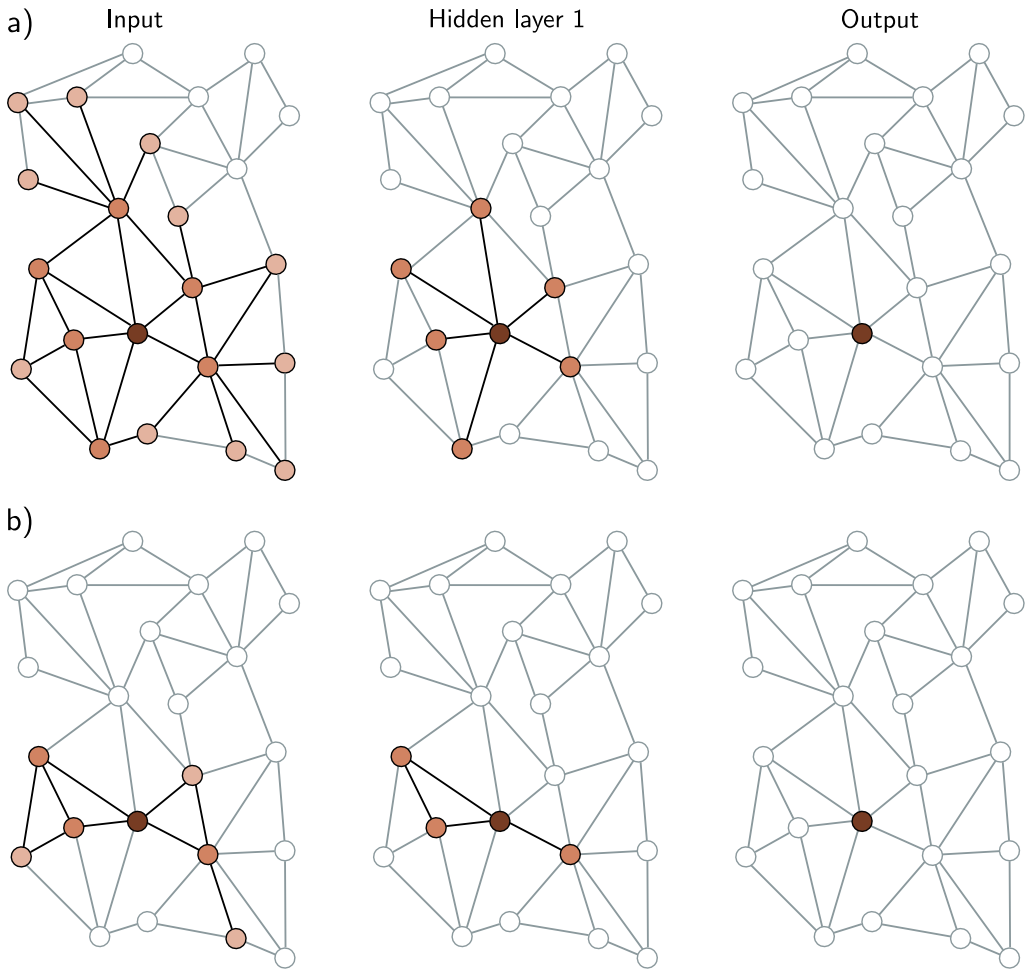

Figure 13.10 Neighborhood sampling. a) One way of forming batches on large graphs is to choose a subset of labeled nodes in the output layer (here, just one node in layer two, right) and then working back to find all of the nodes in the K-hop neighborhood (receptive field). Only this sub-graph is needed to train this batch. Unfortunately, if the graph is densely connected, this may retain a large proportion of the graph. b) One solution is neighborhood sampling. As we work back from the final layer, we select a subset of neighbors (here, three) in the layer before and a subset of the neighbors of these in the layer before that. This restricts the size of the graph for training the batch. In all panels, the brightness represents the distance from the original node.

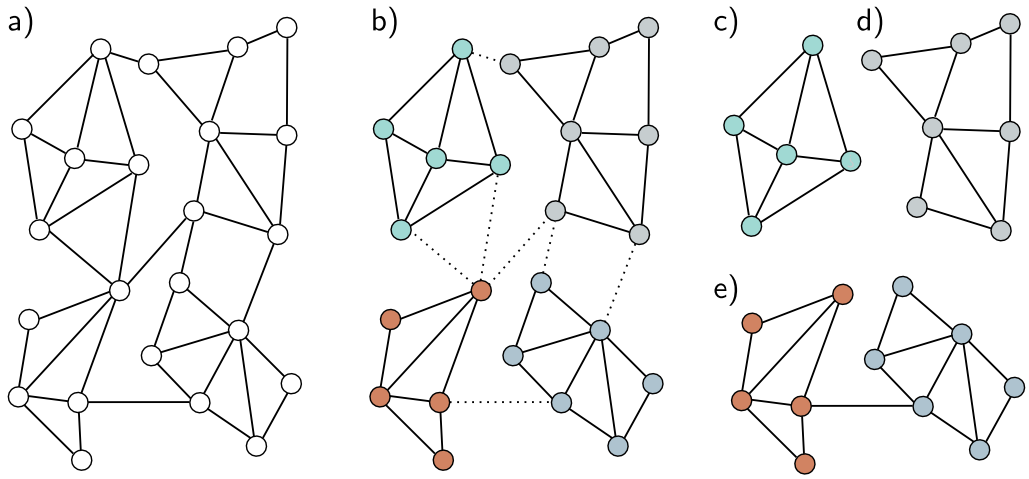

Figure 13.11 Graph partitioning. a) Input graph. b) The input graph is partitioned into smaller subgraphs using a principled method that removes the fewest edges. c-d) We can now use these subgraphs as batches to train in a transductive setting, so here, there are four possible batches. e) Alternatively, we can use combinations of the subgraphs as batches, reinstating the edges between them. If we use pairs of subgraphs, there would be six possible batches here.

train, test, and validation sets as desired; we have effectively converted a transductive problem to an inductive one. To perform inference, we compute predictions for the unknown nodes based on their k-hop neighborhood. Unlike training, this does not require storing the intermediate representations, so it is much more memory efficient.

13.8 Layers for graph convolutional networks

In the previous examples, we combined messages from adjacent nodes by summing them together with the transformed current node. This was accomplished by post-multiplying the node embedding matrix $\mathbf{H}$ by the adjacency matrix plus the identity $\mathbf{A}+\mathbf{I}$. We now consider different approaches to both (i) the combination of the current embedding with the aggregated neighbors and (ii) the aggregation process itself.

13.8.1 Combining current node and aggregated neighbors

In the example GCN layer above, we combined the aggregated neighbors $\mathbf{HA}$ with the current nodes $\mathbf{H}$ by just summing them:

$$\mathbf{H}_{k+1} = \mathbf{a}\Big[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k(\mathbf{A}+\mathbf{I})\Big]. \tag{13.13}$$

In another variation, the current node is multiplied by a factor of $(1+\epsilon_k)$ before contributing to the sum, where ϵ_k is a learned scalar that is different for each layer:

$$\mathbf{H}_{k+1} = \mathbf{a}\Big[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k(\mathbf{A}+(1+\epsilon_k)\mathbf{I})\Big]. \tag{13.14}$$

This is known as *diagonal enhancement*. A related variation applies a different linear transform $\boldsymbol{\Psi}_k$ to the current node:

$$\begin{aligned}
\mathbf{H}_{k+1} &= \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k \mathbf{A} + \boldsymbol{\Psi}_k \mathbf{H}_k\right] \\
&= \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \begin{bmatrix}\boldsymbol{\Omega}_k & \boldsymbol{\Psi}_k\end{bmatrix}\begin{bmatrix}\mathbf{H}_k\mathbf{A} \\ \mathbf{H}_k\end{bmatrix}\right] \\
&= \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}'_k\begin{bmatrix}\mathbf{H}_k\mathbf{A} \\ \mathbf{H}_k\end{bmatrix}\right],
\end{aligned} \tag{13.15}$$

where we have defined $\boldsymbol{\Omega}'_k = \begin{bmatrix}\boldsymbol{\Omega}_k & \boldsymbol{\Psi}_k\end{bmatrix}$ in the third line.

13.8.2 Residual connections

With residual connections, the aggregated representation from the neighbors is transformed and passed through the activation function before summation or concatenation with the current node. For the latter case, the associated network equations are:

$$\mathbf{H}_{k+1} = \begin{bmatrix}\mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k \mathbf{A}\right] \\ \mathbf{H}_k\end{bmatrix}. \tag{13.16}$$

13.8.3 Mean aggregation

The above methods aggregate the neighbors by summing the node embeddings. However, it's possible to combine the embeddings in different ways. Sometimes it's better to take the average of the neighbors rather than the sum; this can be superior if the embedding information is more important and the structural information less so since the magnitude of the neighborhood contributions will not depend on the number of neighbors:

$$\mathbf{agg}[n] = \frac{1}{|\text{ne}[n]|}\sum_{m\in\text{ne}[n]} \mathbf{h}_m, \tag{13.17}$$

where as before, ne$[n]$ denotes a set containing the indices of the neighbors of the n^{th} node. Equation 13.17 can be computed neatly in matrix form by introducing the diagonal $N \times N$ degree matrix $\mathbf{D}$. Each non-zero element of this matrix contains the number of neighbors for the associated node. It follows that each diagonal element in the inverse

Problem 13.8

matrix $\mathbf{D}^{-1}$ contains the denominator that we need to compute the average. The new GCN layer can be written as:

$$\mathbf{H}_{k+1} = \mathbf{a}\Big[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k(\mathbf{A}\mathbf{D}^{-1} + \mathbf{I})\Big]. \tag{13.18}$$

13.8.4 Kipf normalization

There are many variations of graph neural networks based on mean aggregation. Sometimes the current node is included with its neighbors in the mean computation rather than treated separately. In Kipf normalization, the sum of the node representations is normalized as:

Problem 13.9

$$\mathbf{agg}[n] = \sum_{m \in \text{ne}[n]} \frac{\mathbf{h}_m}{\sqrt{|\text{ne}[n]||\text{ne}[m]|}}, \tag{13.19}$$

with the logic that information coming from nodes with a very large number of neighbors should be down-weighted since there are many connections and they provide less unique information. This can also be expressed in matrix form using the degree matrix:

$$\mathbf{H}_{k+1} = \mathbf{a}\Big[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k(\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2} + \mathbf{I})\Big]. \tag{13.20}$$

13.8.5 Max pooling aggregation

An alternative operation that is also invariant to permutation is computing the maximum of a set of objects. The *max pooling* aggregation operator is:

$$\mathbf{agg}[n] = \underset{m \in \text{ne}[n]}{\mathbf{max}}\big[\mathbf{h}_m\big], \tag{13.21}$$

where the operator $\mathbf{max}[\bullet]$ returns the element-wise maximum of the vectors $\mathbf{h}_m$ that are neighbors to the current node n.

13.8.6 Aggregation by attention

The aggregation methods discussed so far either weight the contribution of the neighbors equally or in a way that depends on the graph topology. Conversely, in *graph attention layers*, the weights depend on the data at the nodes. A linear transform is applied to the current node embeddings so that:

$$\mathbf{H}'_k = \boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k. \tag{13.22}$$

Then the similarity s_{mn} of each transformed node embedding $\mathbf{h}'_m$ to the transformed node embedding $\mathbf{h}'_n$ is computed by concatenating the pairs, taking a dot product with a column vector $\boldsymbol{\phi}_k$ of learned parameters, and applying an activation function:

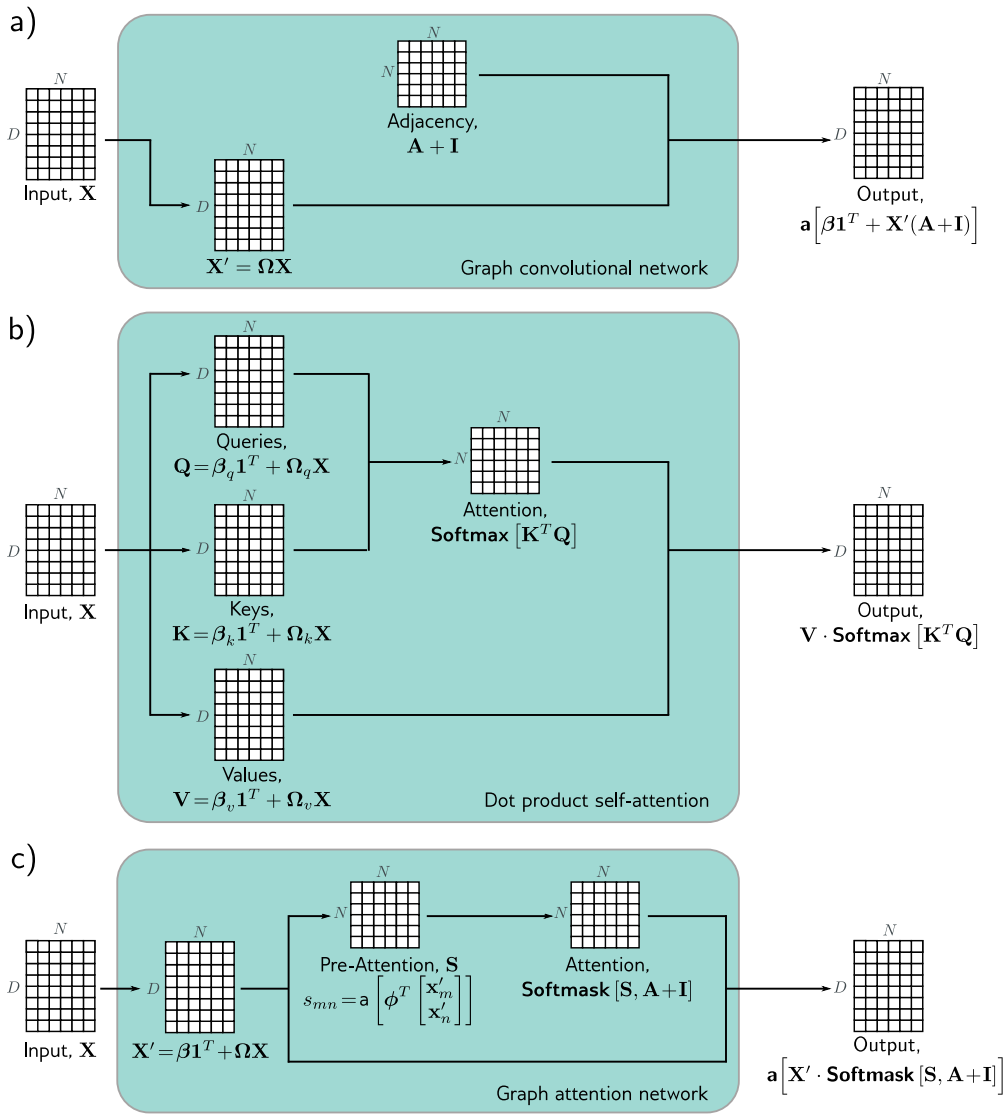

Figure 13.12 Comparison of graph convolutional network, dot product attention, and graph attention network. In each case, the mechanism maps N embeddings of size D stored in a $D \times N$ matrix $\mathbf{X}$ to an output of the same size. a) The graph convolutional network applies a linear transformation $\mathbf{X}' = \boldsymbol{\Omega}\mathbf{X}$ to the data matrix. It then computes a weighted sum of the transformed data, where the weighting is based on the adjacency matrix. A bias $\boldsymbol{\beta}$ is added, and the result is passed through an activation function. b) The outputs of the self-attention mechanism are also weighted sums of the transformed inputs, but this time the weights depend on the data itself via the attention matrix. c) The graph attention network combines both of these mechanisms; the weights are both computed from the data and based on the adjacency matrix.

$$s_{mn} = \mathrm{a}\left[\boldsymbol{\phi}_k^T \begin{bmatrix}\mathbf{h}'_m \\ \mathbf{h}'_n\end{bmatrix}\right]. \tag{13.23}$$

These variables are stored in an $N \times N$ matrix $\mathbf{S}$, where each element represents the similarity of every node to every other. As in dot-product self-attention, the attention weights contributing to each output embedding are normalized to be positive and sum to one using the softmax operation. However, only those values corresponding to the current node and its neighbors should contribute. The attention weights are applied to the transformed embeddings:

$$\mathbf{H}_{k+1} = \mathbf{a}\Big[\mathbf{H}'_k \cdot \mathbf{Softmask}[\mathbf{S}, \mathbf{A}+\mathbf{I}]\Big], \tag{13.24}$$

where $\mathbf{a}[\bullet]$ is a second activation function. The function $\mathbf{Softmask}[\bullet,\bullet]$ computes the attention values by applying softmax operation separately to each column of its first argument $\mathbf{S}$, but only after setting values where the second argument $\mathbf{A}+\mathbf{I}$ is zero to negative infinity, so they do not contribute. This ensures that the attention to non-neighboring nodes is zero.

Notebook 13.4 Graph attention

Problem 13.10

This is very similar to the self-attention computation in transformers (see figure 13.12), except that (i) The keys, queries, and values are all the same, (ii) The measure of similarity is different, and (iii) The attentions are masked so that each node only attends to itself and its neighbors. As in transformers, this system can be extended to use multiple heads that are run in parallel and recombined.

13.9 Edge graphs

Until now, we have focused on processing node embeddings. These evolve as they are passed through the network so that by the end of the network, they represent both the node and its context in the graph. We now consider the case where the information is associated with the edges of the graph.

It is easy to adapt the machinery for node embeddings to process edge embeddings using the *edge graph* (also known as the *adjoint graph* or *line graph*). This is a complementary graph, in which each edge in the original graph becomes a node, and every two edges with a common node in the original graph create an edge in the new graph (figure 13.13). In general, a graph can be recovered from its edge graph, so it's possible to swap between these two representations.

Problems 13.11–13.13

To process edge embeddings, the graph is translated to its edge graph. Then we use exactly the same techniques, aggregating information at each new node from its neighbors and combining this with the current representation. When both node and edge embeddings are present, we can translate back and forth between the two graphs. Now there are four possible updates (nodes update nodes, nodes update edges, edges update nodes, and edges update edges), and these can be alternated as desired, or with minor modifications, nodes can be updated simultaneously from both nodes and edges.

Problem 13.14

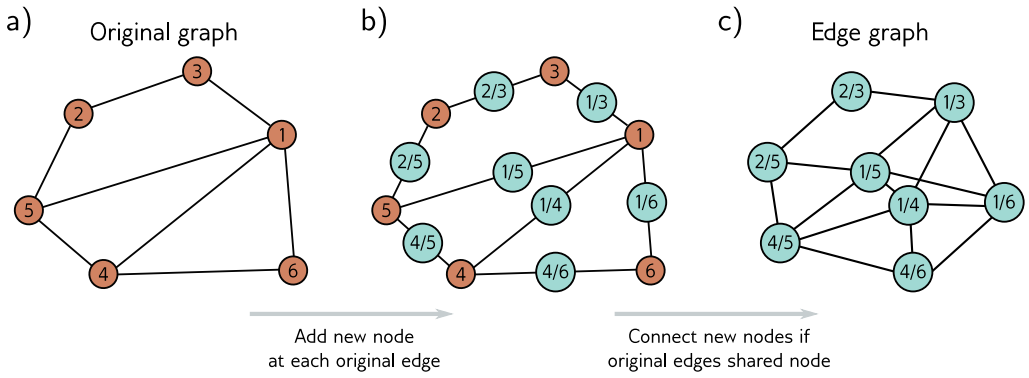

Figure 13.13 Edge graph. a) Graph with six nodes. b) To create the edge graph, we assign one node for each original edge (cyan circles), and c) connect the new nodes if the edges they represent connect to the same node in the original graph.

13.10 Summary

Graphs consist of a set of nodes, where pairs of these nodes are connected by edges. Both nodes and edges can have data attached, and these are referred to as node embeddings and edge embeddings, respectively. Many real-world problems can be framed in terms of graphs, where the goal is to establish a property of the entire graph, properties of each node or edge, or the presence of additional edges in the graph.

Graph neural networks are deep learning models that are applied to graphs. Since the node order in graphs is arbitrary, the layers of graph neural networks must be equivariant to permutations of the node indices. Spatial-based convolutional networks are a family of graph neural networks that aggregate information from the neighbors of a node and then use this to update the node embeddings.

One challenge of processing graphs is that they often occur in the transductive setting, where there is only one partially labeled graph rather than sets of training and test graphs. This graph can be extremely large, which adds further challenges in terms of training and has led to sampling and partitioning algorithms. The edge graph has a node for every edge in the original graph. By converting to this representation, graph neural networks can be used to update the edge embeddings.

Notes

Sanchez-Lengeling et al. (2021) and Daigavane et al. (2021) present good introductory articles on graph processing using neural networks. Recent surveys of research in graph neural networks can be found in articles by Zhou et al. (2020a), Wu et al. (2020c), and Veličković (2023), and the books of Hamilton (2020) and Ma & Tang (2021). GraphEDM (Chami et al., 2020) unifies

many existing graph algorithms into a single framework. In this chapter, we have related graphs to convolutional networks following Bruna et al. (2013), but there are also strong connections with belief propagation (Dai et al., 2016) and graph isomorphism tests (Hamilton et al., 2017a). Zhang et al. (2019c) provide a review focusing specifically on graph convolutional networks. Bronstein et al. (2021) provide a general overview of geometric deep learning, including learning on graphs. Loukas (2020) discusses what types of functions graph neural networks can learn.

Applications: Applications include graph classification (e.g., Zhang et al., 2018b), node classification (e.g., Kipf & Welling, 2017), edge prediction (e.g., Zhang & Chen, 2018), graph clustering (e.g., Tsitsulin et al., 2020), and recommender systems (e.g., Wu et al., 2023). Methods for node classification are reviewed by Xiao et al. (2022a), methods for graph classification by Errica et al. (2019), and methods for edge prediction by Mutlu et al. (2020) and Kumar et al. (2020a).

Graph neural networks: Graph neural networks were introduced by Gori et al. (2005) and Scarselli et al. (2008), who formulated them as a generalization of recursive neural networks. The latter model used the iterative update:

$$\mathbf{h}_n \leftarrow \mathbf{f}\left[\mathbf{x}_n, \mathbf{x}_{m\in \mathrm{ne}[n]}, \mathbf{e}_{e\in \mathrm{nee}[n]}, \mathbf{h}_{m\in \mathrm{ne}[n]}, \boldsymbol{\phi}\right], \tag{13.25}$$

in which each node embedding $\mathbf{h}_n$ is updated from the initial embedding $\mathbf{x}_n$, initial embeddings $\mathbf{x}_{m\in \mathrm{ne}[n]}$ at the adjacent nodes, initial embeddings $\mathbf{e}_{e\in \mathrm{nee}[n]}$ at the adjacent edges, and adjacent node embeddings $\mathbf{h}_{m\in \mathrm{ne}[n]}$. For convergence, the function $\mathbf{f}[\bullet, \bullet, \bullet, \bullet, \boldsymbol{\phi}]$ must be a contraction mapping (see figure 16.9). If we unroll this equation in time for K steps and allow different parameters $\boldsymbol{\phi}_k$ at each time K, then equation 13.25 becomes similar to the graph convolutional network. Subsequent work extended graph neural networks to use gated recurrent units (Li et al., 2016b) and long short-term memory networks (Selsam et al., 2019).

Spectral methods: Bruna et al. (2013) applied the convolution operation in the Fourier domain. The Fourier basis vectors can be found by taking the eigendecomposition of the *graph Laplacian matrix*, $\mathbf{L} = \mathbf{D} - \mathbf{A}$ where $\mathbf{D}$ is the degree matrix and $\mathbf{A}$ is the adjacency matrix. This has disadvantages: the filters are not localized, and the decomposition is prohibitively expensive for large graphs. Henaff et al. (2015) tackled the first problem by forcing the Fourier representation to be smooth (and hence the spatial domain to be localized). Defferrard et al. (2016) introduced ChebNet, which approximates the filters efficiently by using the recursive properties of Chebyshev polynomials. This both provides spatially localized filters and reduces the computation. Kipf & Welling (2017) simplified this further to construct filters that use only a 1-hop neighborhood, resulting in a formulation similar to the spatial methods described in this chapter and providing a bridge between spectral and spatial methods.

Spatial methods: Spectral methods are ultimately based on the Graph Laplacian, so if the graph changes, the model must be retrained. This problem spurred the development of spatial methods. Duvenaud et al. (2015) defined convolutions in the spatial domain, using a different weight matrix to combine the adjacent embeddings for each node degree. This has the disadvantage that it becomes impractical if some nodes have a very large number of connections. Diffusion convolutional neural networks (Atwood & Towsley, 2016) use powers of the normalized adjacency matrix to blend features across different scales, sum these, pointwise multiply by weights, and pass through an activation function to create the node embeddings. Gilmer et al. (2017) introduced *message-passing neural networks*, which defined convolutions on the graph as propagating messages from spatial neighbors. The "aggregate and combine" formulation of *GraphSAGE* (Hamilton et al., 2017a) fits into this framework.

Aggregate and combine: *Graph convolutional networks* (Kipf & Welling, 2017) take a weighted average of the neighbors and current node and then apply a linear mapping and ReLU. *GraphSAGE* (Hamilton et al., 2017a) applies a neural network layer to each neighbor, taking the elementwise maximum to aggregate. Chiang et al. (2019) propose *diagonal enhancement* in which the previous embedding is weighted more than the neighbors. Kipf & Welling (2017) introduced Kipf normalization, which normalizes the sum of the neighboring embeddings based on the degrees of the current node and its neighbors (see equation 13.19).

The *mixture model network* or *MoNet* (Monti et al., 2017) takes this one step further by *learning* a weighting based on the degrees of the current node and the neighbor. They associate a pseudo-coordinate system with each node, where the positions of the neighbors depend on these two quantities. They then learn a continuous function based on a mixture of Gaussians and sample this at the pseudo-coordinates of the neighbors to get the weights. In this way, they can learn the weightings for nodes and neighbors with arbitrary degrees. Pham et al. (2017) use a linear interpolation of the node embedding and neighbors with a different weighted combination for each dimension. The weight of this gating mechanism is generated as a function of the data.

Higher-order convolutional layers: Zhou & Li (2017) used higher-order convolutions by replacing the adjacency matrix $\mathbf{A}$ with $\tilde{\mathbf{A}} = \mathbf{Min}[\mathbf{A}^L + \mathbf{I}, \mathbf{1}]$ where L is the maximum walk-length, $\mathbf{1}$ is a matrix containing only ones, and $\mathbf{Min}[\bullet]$ takes the pointwise minimum of its two matrix arguments; the updates now sum together contributions from any nodes where there is at least one walk of length L. Abu-El-Haija et al. (2019) proposed *MixHop*, which computes node updates from the neighbors (using the adjacency matrix $\mathbf{A}$), the neighbors of the neighbors (using $\mathbf{A}^2$), and so on. They concatenate these updates at each layer. Lee et al. (2018) combined information from nodes beyond the immediate neighbors using geometric *motifs*, which are small local geometric patterns in the graph (e.g., a fully connected clique of five nodes).

Residual connections: Kipf & Welling (2017) proposed a residual connection in which the original embeddings are added to the updated ones. Hamilton et al. (2017b) concatenate the previous embedding to the output of the next layer (see equation 13.16). Rossi et al. (2020) present an inception-style network where the node embedding is concatenated to not only the aggregation of its neighbors but also the aggregation of all neighbors within a walk of two (via computing powers of the adjacency matrix). Xu et al. (2018) introduced *jump knowledge connections* in which the final output at each node consists of the concatenated node embeddings throughout the network. Zhang & Meng (2019) present a general formulation of residual embeddings called *GResNet* and investigate several variations in which the embeddings from the previous layer are added, the input embeddings are added, or versions of these that aggregate information from their neighbors (without further transformation) are added.

Attention in graph neural networks: Veličković et al. (2019) developed the *graph attention network* (figure 13.12c). Their formulation uses multiple heads whose outputs are combined symmetrically. *Gated Attention Networks* (Zhang et al., 2018a) weight the output of the different heads in a way that depends on the data itself. *Graph-BERT* (Zhang et al., 2020) performs node classification using self-attention alone; the graph's structure is captured by adding position embeddings to the data, similarly to how the absolute or relative position of words is captured in the transformer (chapter 12). For example, they add positional information that depends on the number of hops between nodes in the graph.

Permutation invariance: In *DeepSets*, Zaheer et al. (2017) presented a general permutation invariant operator for processing sets. Janossy pooling (Murphy et al., 2018) accepts that many functions are not permutation equivariant and instead uses a permutation-sensitive function and averages the results across many permutations.

Edge graphs: The notation of the *edge graph*, *line graph*, or *adjoint graph* dates to Whitney (1932). The idea of "weaving" layers that update node embeddings from node embeddings, node embeddings from edge embeddings, edge embeddings from edge embeddings, and edge embeddings from node embeddings was proposed by Kearnes et al. (2016). However, here the node-node and edge-edge updates do not involve the neighbors. Monti et al. (2018) introduced the *dual-primal graph CNN*, a modern formulation in a CNN framework that alternates between updates in the original and edge graphs.

Power of graph neural networks: Xu et al. (2019) argue that a neural network should be able to distinguish different graph structures; it is undesirable to map two graphs to the same output if they have the same initial node embeddings but different adjacency matrices. They identified graph structures that could not be distinguished by previous approaches such as GCNs (Kipf & Welling, 2017) and GraphSAGE (Hamilton et al., 2017a). They developed a more powerful architecture with the same discriminative power as the Weisfeiler-Lehman graph isomorphism test (Weisfeiler & Leman, 1968), which is known to discriminate a broad class of graphs. This resulting *graph isomorphism network* was based on the aggregation operation:

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{mlp}\left[(1+\epsilon_k)\,\mathbf{h}_k^{(n)} + \sum_{m\in \mathrm{ne}[n]} \mathbf{h}_k^{(m)}\right]. \tag{13.26}$$

Batches: The original paper on graph convolutional networks (Kipf & Welling, 2017) used full-batch gradient descent. This has memory requirements proportional to the number of nodes, embedding size, and number of layers during training. Since then, three types of methods have been proposed to reduce the memory requirements and create batches for SGD in the transductive setting: node sampling, layer sampling, and sub-graph sampling.

Node sampling methods start by randomly selecting a subset of target nodes and then work back through the network, adding a subset of the nodes in the receptive field at each stage. GraphSAGE (Hamilton et al., 2017a) proposed a fixed number of neighborhood samples as in figure 13.10b. Chen et al. (2018b) introduce a variance reduction technique, but this uses historical activations of nodes and so still has a high memory requirement. *PinSAGE* (Ying et al., 2018a) uses random walks from the target nodes and chooses the K nodes with the highest visit count. This prioritizes ancestors that are more closely connected.

Node sampling still requires increasing numbers of nodes as we pass back through the graph. *Layer sampling methods* address this by directly sampling the receptive field in each layer independently. Examples of layer sampling include FastGCN (Chen et al., 2018a), adaptive sampling (Huang et al., 2018b), and layer-dependent importance sampling (Zou et al., 2019).

Subgraph sampling methods randomly draw subgraphs or divide the original graph into subgraphs. These are then trained as independent data examples. Examples of these approaches include *GraphSAINT* (Zeng et al., 2020), which samples sub-graphs during training using random walks and then runs a full GCN on the subgraph while also correcting for the bias and variance of the minibatch. *Cluster GCN* (Chiang et al., 2019) partitions the graph into clusters (by maximizing the embedding utilization or number of within-batch edges) in a pre-processing stage and randomly selects clusters to form minibatches. To create more randomness, they train random subsets of these clusters plus the edges between them (see figure 13.11).

Wolfe et al. (2021) proposed a distributed training method that both partitions the graph and trains narrower GCNs in parallel by partitioning the feature space at different layers. More information about sampling graphs can be found in Rozemberczki et al. (2020).

Regularization and normalization: Rong et al. (2020) proposed *DropEdge*, which randomly drops edges from the graph during each training iteration by masking the adjacency matrix. This

can be done for the whole neural network or differently in each layer (layer-wise DropEdge). In a sense, this is similar to dropout in that it breaks connections in the flow of data, but it can also be considered an augmentation method since changing the graph is similar to perturbing the data. Schlichtkrull et al. (2018), Teru et al. (2020), and Veličković et al. (2019) also proposed randomly dropping edges from the graph as a form of regularization similar to dropout. Node sampling methods (Hamilton et al., 2017a; Huang et al., 2018b; Chen et al., 2018a) can also be considered regularizers. Hasanzadeh et al. (2020) present a general framework called *DropConnect* that unifies many of the above approaches.

There are also many proposed normalization schemes for graph neural networks, including *PairNorm* (Zhao & Akoglu, 2020), *weight normalization* (Oono & Suzuki, 2019), *differentiable group normalization* (Zhou et al., 2020b), and *GraphNorm* (Cai et al., 2021).

Multi-relational graphs: Schlichtkrull et al. (2018) proposed a variation of graph convolutional networks for multi-relational graphs (i.e., graphs with more than one edge type). Their scheme separately aggregates information from each edge type using different parameters. If there are many edge types, the number of parameters may become large, and to combat this, they propose that each edge type uses a different weighting of a basis set of parameters.

Hierarchical representations and pooling: CNNs for image classification gradually decrease the representation size but increase the number of channels as the network progresses. However, the GCNs for graph classification in this chapter maintain the entire graph until the last layer and then combine all the nodes to compute the final prediction. Ying et al. (2018b) proposed *DiffPool*, which clusters graph nodes to make a graph that gets progressively smaller as the depth increases in a way that is differentiable, and so can be learned. This can be done based on the graph structure alone or adaptively based on the graph structure and the embeddings. Other pooling methods include *SortPool* (Zhang et al., 2018b) and *self-attention graph pooling* (Lee et al., 2019). A comparison of pooling layers for graph neural networks can be found in Grattarola et al. (2022). Gao & Ji (2019) propose an encoder-decoder structure for graphs based on the U-Net (see figure 11.10).

Geometric graphs: The *MoNet* model (Monti et al., 2017) can exploit geometric information because neighboring nodes have well-defined spatial positions. They learn a mixture of Gaussians function and sample from this based on the relative coordinates of the neighbor. In this way, they can weight neighboring nodes based on their relative positions as in standard convolutional neural networks, even though these positions are not constant. The *geodesic CNN* (Masci et al., 2015) and *anisotropic CNN* (Boscaini et al., 2016) both adapt convolution to manifolds (i.e., surfaces) as represented by triangular meshes. They locally approximate the surface as a plane and define a coordinate system on this plane around the current node.

Oversmoothing and suspended animation: Unlike other deep learning models, graph neural networks did not, until recently, benefit significantly from increasing depth. Indeed, the original GCN paper (Kipf & Welling, 2017) and GraphSAGE (Hamilton et al., 2017a) both only use two layers, and Chiang et al. (2019) trained a five-layer Cluster-GCN to get state-of-the-art performance on the PPI dataset. One possible explanation is *over-smoothing* (Li et al., 2018c); at each layer, the network incorporates information from a larger neighborhood, and it may be that this ultimately results in the dissolution of (important) local information. Indeed (Xu et al., 2018) prove that the influence of one node on another is proportional to the probability of reaching that node in a K-step random walk. This approaches the stationary distribution of walks over the graph with increasing K, causing the local neighborhood to be washed out.

Alon & Yahav (2021) proposed another explanation for why performance doesn't improve with network depth. They argue that adding depth allows information to be aggregated from longer paths. However, in practice, the exponential growth in the number of neighbors means there is a bottleneck whereby too much information is "squashed" into the fixed-size node embeddings.

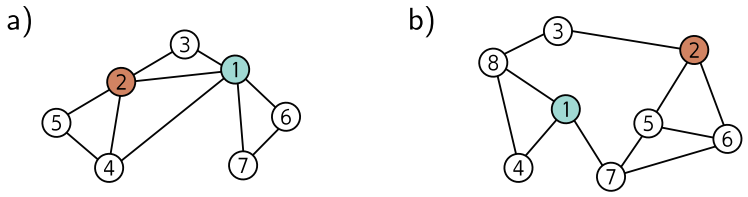

Figure 13.14 Graphs for problems 13.1, 13.3, and 13.8.

Ying et al. (2018a) also note that when the depth of the network exceeds a certain limit, the gradients no longer propagate back, and learning fails for both the training and test data. They term this effect *suspended animation*. This is similar to when many layers are naïvely added to convolutional neural networks (figure 11.2). They propose a family of residual connections that allow deeper networks to be trained. Vanishing gradients (section 7.5) have also been identified as a limitation by Li et al. (2021b).

It has recently become possible to train deeper graph neural networks using various forms of residual connection (Xu et al., 2018; Li et al., 2020a; Gong et al., 2020; Chen et al., 2020b; Xu et al., 2021a). Li et al. (2021a) train a state-of-the-art model with more than 1000 layers using an invertible network to reduce the memory requirements of training (see chapter 16).

Problems

Problem 13.1 Write out the adjacency matrices for the two graphs in figure 13.14.

Problem 13.2* Draw graphs that correspond to the following adjacency matrices:

$$\mathbf{A}_1 = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{A}_2 = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}.$$

Problem 13.3* Consider the two graphs in figure 13.14. How many ways are there to walk from node one to node two in (i) three steps and (ii) seven steps?

Problem 13.4 The diagonal of $\mathbf{A}^2$ in figure 13.4c contains the number of edges that connect to each corresponding node. Explain this phenomenon.

Appendix B.4.4 Permutation matrix

Problem 13.5 What permutation matrix is responsible for the transformation between the graphs in figures 13.5a–c and figure 13.5d–f?

Problem 13.6 Prove that:

$$\text{sig}\left[\beta_K + \boldsymbol{\omega}_K \mathbf{H}_K \mathbf{1}\right] = \text{sig}\left[\beta_K + \boldsymbol{\omega}_K \mathbf{H}_K \mathbf{P} \mathbf{1}\right], \tag{13.27}$$

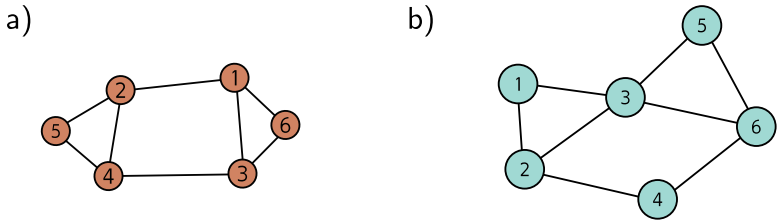

Figure 13.15 Graphs for problems 13.11–13.13.

where $\mathbf{P}$ is an $N \times N$ permutation matrix (a matrix that is all zeros except for exactly one entry in each row and each column, which is one), and $\mathbf{1}$ is an $N \times 1$ vector of ones.

Problem 13.7* Consider the simple GNN layer:

$$\begin{aligned}\mathbf{H}_{k+1} &= \text{GraphLayer}[\mathbf{H}_k, \mathbf{A}] \\ &= \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \begin{bmatrix}\mathbf{H}_k \\ \mathbf{H}_k\mathbf{A}\end{bmatrix}\right], \end{aligned} \tag{13.28}$$

where $\mathbf{H}$ is a $D \times N$ matrix containing the N node embeddings in its columns, $\mathbf{A}$ is the $N \times N$ adjacency matrix, $\boldsymbol{\beta}$ is the bias vector, and $\boldsymbol{\Omega}$ is the weight matrix. Show that this layer is equivariant to permutations of the node order so that:

$$\text{GraphLayer}[\mathbf{H}_k, \mathbf{A}]\mathbf{P} = \text{GraphLayer}[\mathbf{H}_k\mathbf{P}, \mathbf{P}^T\mathbf{A}\mathbf{P}], \tag{13.29}$$

where $\mathbf{P}$ is an $N \times N$ permutation matrix.

Problem 13.8 What is the degree matrix $\mathbf{D}$ for each graph in figure 13.14?

Problem 13.9 The authors of GraphSAGE (Hamilton et al., 2017a) propose a pooling method in which the node embedding is averaged together with its neighbors so that:

$$\mathbf{agg}[n] = \frac{1}{1+|\text{ne}[n]|}\left(\mathbf{h}_n + \sum_{m \in \text{ne}[n]} \mathbf{h}_m\right). \tag{13.30}$$

Show how this operation can be computed simultaneously for all node embeddings in the $D \times N$ embedding matrix $\mathbf{H}$ using linear algebra. You will need to use both the adjacency matrix $\mathbf{A}$ and the degree matrix $\mathbf{D}$.

Problem 13.10* Devise a graph attention mechanism based on dot-product self-attention and draw its mechanism in the style of figure 13.12.

Problem 13.11* Draw the edge graph associated with the graph in figure 13.15a.

Problem 13.12* Draw the node graph corresponding to the edge graph in figure 13.15b.

Problem 13.13 For a general undirected graph, describe how the adjacency matrix of the node graph relates to the adjacency matrix of the corresponding edge graph.

Problem 13.14* Design a layer that updates a node embedding $\mathbf{h}_n$ based on its neighboring node embeddings $\{\mathbf{h}_m\}_{m \in \text{ne}[n]}$ and neighboring edge embeddings $\{\mathbf{e}_m\}_{m \in \text{nee}[n]}$. You should consider the possibility that the edge embeddings are not the same size as the node embeddings.

Chapter 14

Unsupervised learning

Chapters 2–9 walked through the *supervised learning* pipeline. We defined models that mapped observed data $\mathbf{x}$ to output values $\mathbf{y}$ and introduced loss functions that measured the quality of that mapping for a training dataset $\{\mathbf{x}_i, \mathbf{y}_i\}$. Then we discussed how to fit and measure the performance of these models. Chapters 10–13 introduced more sophisticated model architectures incorporating parameter sharing and allowing parallel computational paths.

The defining characteristic of *unsupervised learning models* is that they are learned from a set of observed data $\{\mathbf{x}_i\}$ in the absence of labels. All unsupervised models share this property, but they have diverse goals. They may be used to generate plausible new samples from the dataset or to manipulate, denoise, interpolate between, or compress examples. They can also be used to reveal the internal structure of a dataset (e.g., by dividing it into coherent clusters) or to distinguish whether new examples belong to the same dataset or are outliers.

This chapter introduces a taxonomy of unsupervised learning models and then discusses the desirable properties of models and how to measure their performance. The four subsequent chapters discuss four particular models: generative adversarial networks (GANs), variational autoencoders (VAEs), normalizing flows, and diffusion models.[1]

14.1 Taxonomy of unsupervised learning models

A common strategy in unsupervised learning is to define a mapping between the data examples $\mathbf{x}$ and a set of unseen *latent* variables $\mathbf{z}$. These latent variables capture underlying structure in the dataset and usually have a lower dimension than the original data; in this sense, a latent variable $\mathbf{z}$ can be considered a compressed version of a data example $\mathbf{x}$ that captures its essential qualities (figures 1.9–1.10).

In principle, the mapping between the observed and latent variables can be in either direction. Some models map from the data $\mathbf{x}$ to latent variables $\mathbf{z}$. For example, the

[1] Until this point, almost all of the relevant math has been embedded in the text. However, the following four chapters require a solid knowledge of probability. Appendix C covers the relevant material.

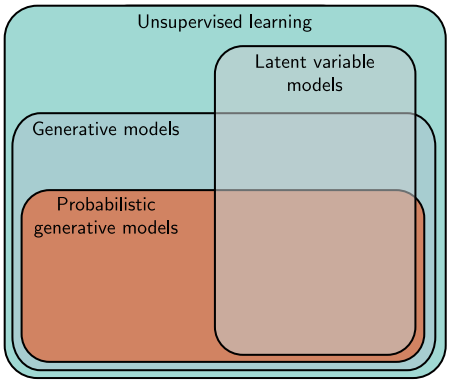

Figure 14.1 Taxonomy of unsupervised learning models. Unsupervised learning refers to any model trained on datasets without labels. Generative models can synthesize (generate) new examples with similar statistics to the training data. A subset of these are probabilistic and define a distribution over the data. We draw samples from this distribution to generate new examples. Latent variable models define a mapping between an underlying explanatory (latent) variable and the data. They may fall into any of the above categories.

famous *k-means* algorithm maps the data $\mathbf{x}$ to a cluster assignment $z \in \{1, 2, \ldots, K\}$. Other models map from the latent variables $\mathbf{z}$ to the data $\mathbf{x}$. Consider defining a distribution $Pr(\mathbf{z})$ over the latent variable $\mathbf{z}$ in these models. New examples can now be *generated* by (i) drawing from this distribution and (ii) mapping the sample to the data space $\mathbf{x}$. Accordingly, these are termed *generative models* (see figure 14.1).

The four models in chapters 15 to 18 are all generative models that use latent variables. *Generative adversarial networks* (chapter 15) learn to generate data examples $\mathbf{x}^*$ from latent variables $\mathbf{z}$, using a loss that encourages the generated samples to be indistinguishable from real examples (figure 14.2a).

Normalizing flows, *variational autoencoders*, and *diffusion models* (chapters 16–18) are *probabilistic generative models*. In addition to generating new examples, they assign a probability $Pr(\mathbf{x}|\boldsymbol{\phi})$ to each data point $\mathbf{x}$. This will depend on the model parameters $\boldsymbol{\phi}$, and in training, we maximize the probability of the observed data $\{\mathbf{x}_i\}$, so the loss is the sum of the negative log-likelihoods (figure 14.2b):

$$L[\boldsymbol{\phi}] = -\sum_{i=1}^{I} \log\Big[Pr(\mathbf{x}_i|\boldsymbol{\phi})\Big]. \tag{14.1}$$

Since probability distributions must sum to one, this implicitly reduces the probability of examples that lie far from the observed data. As well as providing a training criterion, assigning probabilities is useful in its own right; the probability on a test set can be used to compare two models quantitatively, and the probability for an example can be thresholded to determine if it belongs to the same dataset or is an *outlier*.[2]

14.2 What makes a good generative model?

Generative models based on latent variables should have the following properties:

[2]Note that not all probabilistic generative models rely on latent variables. The transformer decoder (section 12.7) was learned without labels, can generate new examples, and can assign a probability to these examples but is based on an autoregressive formulation (equation 12.15).

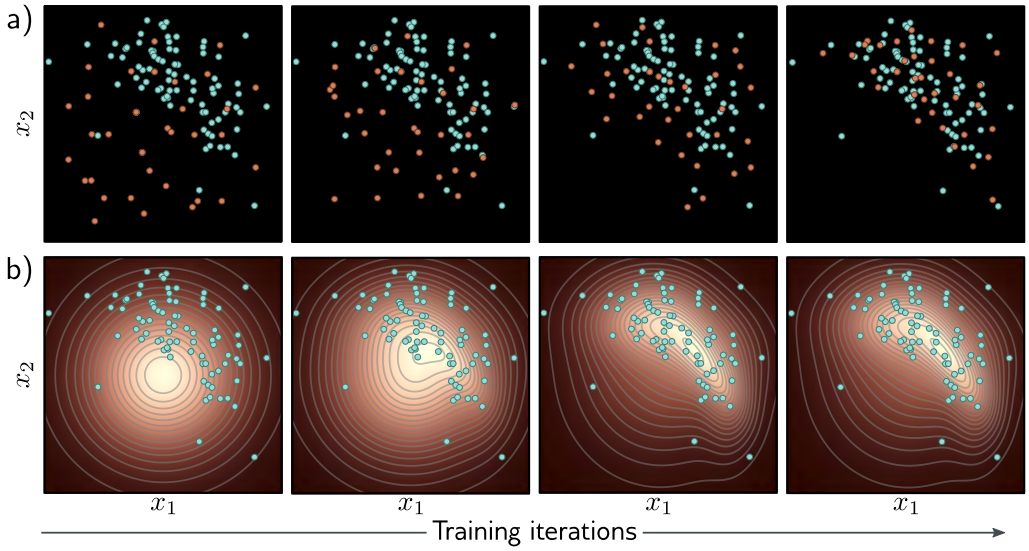

Figure 14.2 Fitting generative models a) Generative adversarial models provide a mechanism for generating samples (orange points). As training proceeds (left to right), the loss function encourages these samples to become progressively less distinguishable from real examples (cyan points). b) Probabilistic models (including variational autoencoders, normalizing flows, and diffusion models) learn a probability distribution over the training data. As training proceeds (left to right), the likelihood of the real examples increases under this distribution, which can be used to draw new samples and assess the probability of new data points.

- **Efficient sampling:** Generating samples from the model should be computationally inexpensive and take advantage of the parallelism of modern hardware.
- **High-quality sampling:** The samples should be indistinguishable from the real data with which the model was trained.
- **Coverage:** Samples should represent the entire training distribution. It is insufficient to generate samples that all look like a subset of the training examples.
- **Well-behaved latent space:** Every latent variable $\mathbf{z}$ corresponds to a plausible data example $\mathbf{x}$. Smooth changes in $\mathbf{z}$ correspond to smooth changes in $\mathbf{x}$.
- **Disentangled latent space:** Manipulating each dimension of $\mathbf{z}$ should correspond to changing an interpretable property of the data. For example, in a model of language, it might change the topic, tense, or verbosity.
- **Efficient likelihood computation:** If the model is probabilistic, we would like to be able to calculate the probability of new examples efficiently and accurately.

This naturally leads to the question of whether the generative models that we consider satisfy these properties. The answer is subjective, but figure 14.3 provides guidance. The precise assignments are disputable, but most practitioners would agree that there is no single model that satisfies all of these characteristics.

Model	Efficient	Sample quality	Coverage	Well-behaved latent space	Disentangled latent space	Efficient likelihood
GANs	✓	✓	✗	✓	?	n/a
VAEs	✓	✗	?	✓	?	✗
Flows	✓	✗	?	✓	?	✓
Diffusion	✗	✓	?	✗	✗	✗

Figure 14.3 Properties of four generative models. Neither generative adversarial networks (GANs), variational autoencoders (VAEs), normalizing flows (Flows), nor diffusion models (diffusion) have the full complement of desirable properties.

14.3 Quantifying performance

The previous section discussed the desirable properties of generative models. We now consider quantitative measures of success for generative models. Much experimentation with generative models has used images due to the widespread availability of that data and the ease of qualitatively judging the samples. Consequently, some of these metrics only apply to images.

Test likelihood: One way to compare probabilistic models is to measure their likelihood for a test dataset. It is ineffective to measure the training data likelihood because a model could assign a very high probability to each training point and very low probabilities in between. This model would have a very high training likelihood but could only reproduce the training data. The test likelihood captures how well the model generalizes from the training data and also the coverage; if the model assigns a high probability to just a subset of the training data, it must assign lower probabilities elsewhere, so a portion of the test examples will have low probability.

Test likelihood is a sensible way to quantify probabilistic models, but unfortunately, it is not relevant for generative adversarial models (which do not assign a probability) and is expensive to estimate for variational autoencoders and diffusion models (although it is possible to compute a lower bound on the log-likelihood). Normalizing flows are the only type of model for which the likelihood can be computed exactly and efficiently.

Inception score: The inception score (IS) is specialized for images and ideally for generative models trained on the ImageNet database. The score is calculated using a pre-trained classification model – usually the "Inception" model, from which the name is derived. It is based on two criteria. First, each generated image $\mathbf{x}^*$ should look like one and only one of the 1000 possible classes y in the ImageNet database. Hence, the probability distribution $Pr(y_i|\mathbf{x}_i^*)$ should be highly peaked at the correct class. Second, the entire set of generated images should be assigned to the classes with equal probability, so $Pr(y)$ should be flat when averaged over all generated examples.

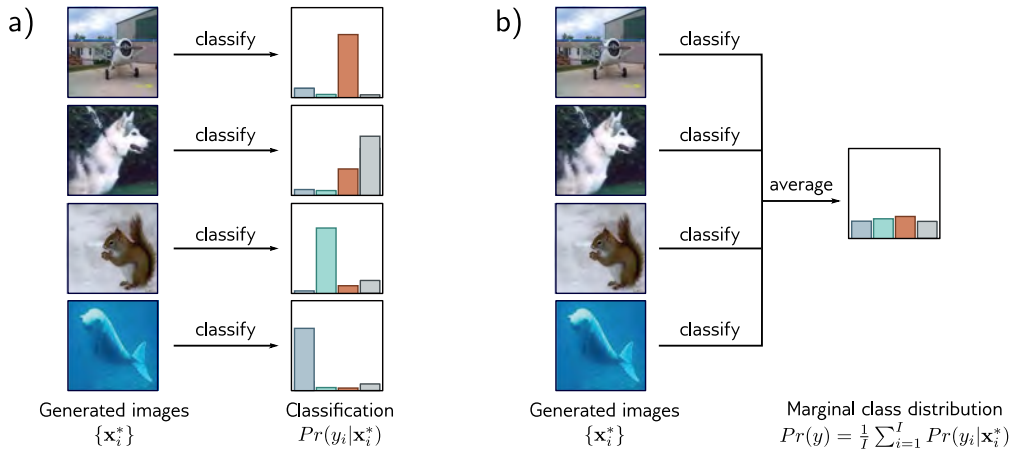

Figure 14.4 Inception score. a) A pretrained network classifies the generated images. If the images are realistic, the resulting class probabilities $Pr(y_i|\mathbf{x}_i^*)$ should be peaked at the correct class. b) If the model generates all classes equally frequently, the marginal (average) class probabilities should be flat. The inception score measures the average distance between the distributions in (a) and the distribution in (b). Images from Deng et al. (2009).

The inception score measures the average distance between these two distributions over the generated set. This distance will be large if one is peaked and the other flat (figure 14.4). More precisely, it returns the exponential of the expected KL-divergence between $Pr(y_i|\mathbf{x}_i^*)$ and $Pr(y)$:

Appendix C.5.1 KL divergence

$$IS = \exp\left[\frac{1}{I}\sum_{i=1}^{I} D_{KL}\Big[Pr(y_i|\mathbf{x}_i^*)||Pr(y)\Big]\right], \tag{14.2}$$

where I is the number of generated examples and:

$$Pr(y) = \frac{1}{I}\sum_{i=1}^{I} Pr(y_i|\mathbf{x}_i^*). \tag{14.3}$$

This metric is only sensible for generative models of the ImageNet database and is sensitive to the particular classification model; retraining this model can give quite different numerical results. Moreover, it does not reward diversity within an object class; it returns a high value if the model only generates one realistic example of each class.

Fréchet inception distance: This measure is also intended for images and computes a symmetric distance between the distributions of generated samples and real examples. This must be approximate since it is hard to characterize either distribution (indeed,

characterizing the distribution of real examples is the job of generative models in the first place). Hence, the Fréchet inception distance approximates both distributions by multivariate Gaussians and (as the name suggests) estimates the distance between them using the Fréchet distance.

Appendix C.5.4 Fréchet distance

However, it does not model the distance with respect to the original data but rather the activations in the deepest layer of the inception classification network. These hidden units are the ones most associated with object classes, so the comparison occurs at a semantic level, ignoring the more fine-grained details of the images. This metric does take account of diversity within classes but relies heavily on the information retained by the features in the inception network; any information discarded by the network does not contribute to the result. Some of this discarded information may still be important to generate realistic samples.

Manifold precision/recall: Fréchet inception distance is sensitive both to the realism of the samples and their diversity but does not distinguish between these factors. To disentangle these qualities, we consider the overlap between the data *manifold* (i.e., the subset of the data space where the real examples lie) and the model manifold (i.e., where the generated samples lie). The *precision* is the fraction of model samples that fall into the data manifold. This measures the proportion of generated samples that are realistic. The *recall* is the fraction of data examples that fall within the model manifold. This measures the proportion of the real data the model can generate (figure 14.5).

To estimate the manifold, we place a hypersphere around each data example, whose radius is the distance to the k^{th} nearest neighbor. The union of these spheres is an approximation of the manifold, and it's easy to determine if a new point lies within it. This manifold is also typically computed in the feature space of a classifier with the advantages and disadvantages that entails.

14.4 Summary

Unsupervised models learn about the structure of a dataset in the absence of labels. A subset of these models is generative and can synthesize new data examples. A further subset is probabilistic in that they can both generate new examples and assign a probability to observed data. The models considered in the following four chapters start with a latent variable $\mathbf{z}$ which has a known distribution. A deep neural network then maps from the latent variable to the observed data space. We considered desirable properties of generative models and introduced metrics that attempt to quantify their performance.

Notes

Popular generative models include generative adversarial networks (Goodfellow et al., 2014), variational autoencoders (Kingma & Welling, 2014), normalizing flows (Rezende & Mohamed,

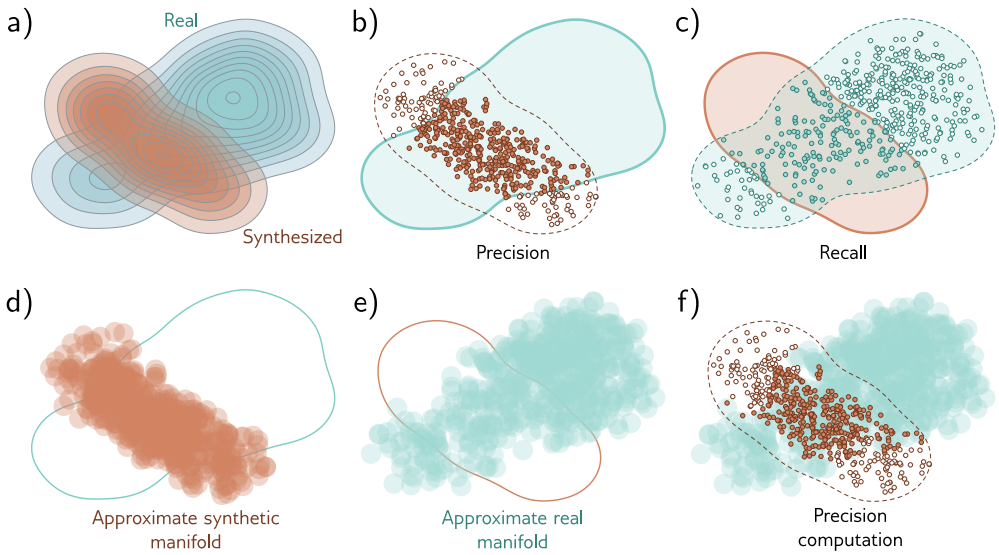

Figure 14.5 Manifold precision/recall. a) True distributions of real examples and samples synthesized by the generative model. b) The overlap can be summarized by the *precision* (the proportion of synthesized samples that overlap with the distribution or *manifold* of real examples), and c) *recall* (the proportion of real examples that overlap with the manifold of the synthesized samples). d) The manifold of synthesized samples can be approximated by taking the union of a set of hyperspheres centered on each sample. Here, these have constant radius, but more commonly, the radius is based on the distance to the k^{th} nearest neighbor. e) The manifold for real examples is approximated similarly. f) The precision can be computed as the proportion of real examples that lie within the approximated manifold of samples. Similarly, the recall is computed as the proportion of samples that lie within the approximated manifold of real examples (not shown). Adapted from Kynkäänniemi et al. (2019).

2015), diffusion models (Sohl-Dickstein et al., 2015; Ho et al., 2020), autoregressive models (Bengio et al., 2000; Van den Oord et al., 2016b), and energy-based models (LeCun et al., 2006). All except energy models are discussed in this book. Bond-Taylor et al. (2022) provide a recent survey of generative models.

Evaluation: Salimans et al. (2016) introduced the inception score, and Heusel et al. (2017) introduced the Fréchet inception distance, both of which are based on the Pool-3 layer of the Inception V3 model (Szegedy et al., 2016). Nash et al. (2021) used earlier layers of the same network that retain more spatial information to ensure that the spatial statistics of images are also replicated. Kynkäänniemi et al. (2019) introduced the manifold precision/recall method. Barratt & Sharma (2018) discuss the inception score in detail and point out its weaknesses. Borji (2022) discusses the pros and cons of different methods for assessing generative models.

Chapter 15

Generative Adversarial Networks

A *generative adversarial network* or *GAN* is an unsupervised model that aims to generate new samples that are indistinguishable from a set of training examples. GANs are just mechanisms to create new samples; they do not build a probability distribution over the modeled data and hence cannot evaluate the probability that a new data point belongs to the same distribution.

In a GAN, the main *generator* network creates samples by mapping random noise to the output data space. If a second *discriminator* network cannot distinguish between the generated samples and the real examples, the samples must be plausible. If this network *can* tell the difference, this provides a training signal that can be fed back to improve the quality of the samples. This idea is simple, but training GANs is difficult: the learning algorithm can be unstable, and although GANs may learn to generate realistic samples, this does not imply that they learn to generate *all* possible samples.

GANs have been applied to many types of data, including audio, 3D models, text, video, and graphs. However, they have found the most success in the image domain, where they can produce samples that are almost indistinguishable from real pictures. Accordingly, the examples in this chapter focus on synthesizing images.

15.1 Discrimination as a signal

We aim to generate new samples $\{\mathbf{x}_j^*\}$ that are drawn from the same distribution as a set of real training data $\{\mathbf{x}_i\}$. A single new sample $\mathbf{x}_j^*$ is generated by (i) choosing a *latent variable* $\mathbf{z}_j$ from a simple base distribution (e.g., a standard normal) and then (ii) passing this data through a network $\mathbf{x}_j^* = \mathbf{g}[\mathbf{z}_j, \boldsymbol{\theta}]$ with parameters $\boldsymbol{\theta}$. This network is known as the *generator*. During the learning process, the goal is to find parameters $\boldsymbol{\theta}$ so that the samples $\{\mathbf{x}_j^*\}$ look "similar" to the real data $\{\mathbf{x}_i\}$ (see figure 14.2a).

Similarity can be defined in many ways, but the GAN uses the principle that the samples should be statistically indistinguishable from the true data. To this end, a second network $\mathrm{f}[\bullet, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$ called the *discriminator* is introduced. This network aims to classify its input as being a real example or a generated sample. If this

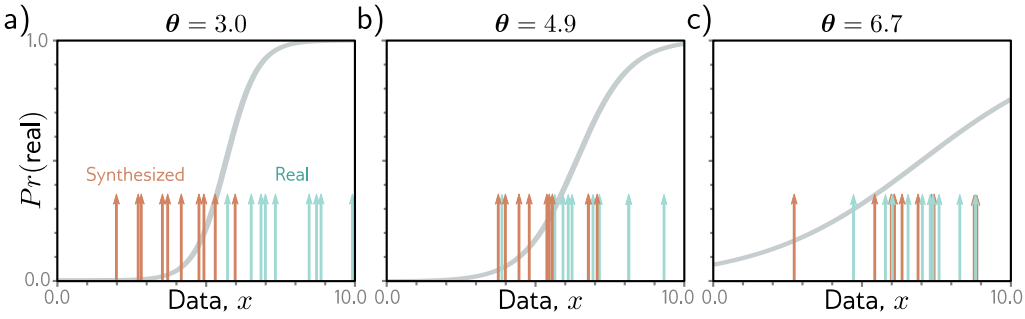

Figure 15.1 GAN mechanism. a) Given a parameterized function (a generator) that synthesizes samples (orange arrows) and a batch of real examples (cyan arrows), we train a discriminator to distinguish the real examples from the generated samples (sigmoid curve indicates the estimated probability that the data point is real). b) The generator is trained by modifying its parameters so that the discriminator becomes less confident the samples were synthetic (in this case, by moving the orange samples to the right). The discriminator is then updated. c) Alternating updates to the generator and discriminator cause the generated samples to become indistinguishable from real examples and the impetus to change the generator (i.e., the slope of the sigmoid function) to diminish.

proves impossible, the generated samples are indistinguishable from the real examples, and we have succeeded. If it is possible, the discriminator provides a signal that can be used to improve the generation process.

Figure 15.1 illustrates this scheme. We start with a training set $\{x_i\}$ of real 1D examples. A different batch of ten of these examples $\{x_i\}_{i=1}^{10}$ is shown in each panel (cyan arrows). To create a batch of samples $\{x_j^*\}$, we use the simple generator:

$$x_j^* = \mathrm{g}[z_j, \theta] = z_j + \theta, \tag{15.1}$$

where latent variables $\{z_j\}$ are drawn from a standard normal distribution, and the parameter θ translates the generated samples along the x-axis (figure 15.1).

At initialization, $\theta = 3.0$, and the generated samples (orange arrows) lie to the left of the real examples (cyan arrows). The discriminator is trained to distinguish the generated samples from the real examples (the sigmoid curve indicates the probability that a data point is real). During training, the generator parameters θ are manipulated to increase the probability that its samples are classified as real. Here, this means increasing θ so that the samples move rightwards where the sigmoid curve is higher.

Notebook 15.1
GAN toy example

We alternate between updating the discriminator and the generator. Figures 15.1b–c show two iterations of this process. It gradually becomes harder to classify the data, so the impetus to change θ becomes weaker (i.e., the sigmoid becomes flatter). At the end of the process, there is no way to distinguish the two sets of data; the discriminator, which now has chance performance, is discarded, and we are left with a generator that makes plausible samples.

15.1.1 GAN loss function

We now define the loss function for training GANs more precisely. The discriminator $\mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]$ takes input $\mathbf{x}$, has parameters $\boldsymbol{\phi}$, and returns a scalar that is higher when it believes the input is a real example. This is a binary classification task, so we adapt the binary cross-entropy loss function (section 5.4), which originally had the form:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_i -(1-y_i)\log\Big[1-\mathrm{sig}[\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}]]\Big] - y_i \log\Big[\mathrm{sig}[\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}]]\Big] \right], \tag{15.2}$$

where $y_i \in \{0,1\}$ is the label, and $\mathrm{sig}[\bullet]$ is the logistic sigmoid function (figure 5.7).

In this case, we assume that the real examples $\mathbf{x}$ have label $y=1$ and the generated samples $\mathbf{x}^*$ have label $y=0$ so that:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_j -\log\Big[1-\mathrm{sig}[\mathrm{f}[\mathbf{x}^*_j,\boldsymbol{\phi}]]\Big] - \sum_i \log\Big[\mathrm{sig}[\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}]]\Big] \right], \tag{15.3}$$

where i and j index the real examples and generated samples, respectively.

Now we substitute the definition for the generator $\mathbf{x}^*_j = \mathbf{g}[\mathbf{z}_j, \boldsymbol{\theta}]$ and note that we must maximize with respect to $\boldsymbol{\theta}$ since we want the generated samples to be misclassified (i.e., have low likelihood of being synthetic or high negative log-likelihood):

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \left[\min_{\boldsymbol{\phi}} \left[\sum_j -\log\Big[1-\mathrm{sig}[\mathrm{f}[\mathbf{g}[\mathbf{z}_j,\boldsymbol{\theta}],\boldsymbol{\phi}]]\Big] - \sum_i \log\Big[\mathrm{sig}[\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}]]\Big] \right] \right]. \tag{15.4}$$

15.1.2 Training GANs

Equation 15.4 is a more complex loss function than we have seen before; the discriminator parameters $\boldsymbol{\phi}$ are manipulated to minimize the loss function, and the generative parameters $\boldsymbol{\theta}$ are manipulated to maximize the loss function. GAN training is characterized as a *minimax game*; the generator tries to find new ways to fool the discriminator, which in turn searches for new ways to distinguish generated samples from real examples. Technically, the solution is a *Nash equilibrium* — the optimization algorithm searches for a position that is simultaneously a minimum of one function and a maximum of the other. If training proceeds as planned, then upon convergence, $\mathbf{g}[\mathbf{z}, \boldsymbol{\theta}]$ will be drawn from the same distribution as the data, and $\mathrm{sig}[\mathbf{f}[\bullet, \boldsymbol{\phi}]]$ will be at chance (i.e., 0.5).

To train the GAN, we can divide equation 15.4 into two loss functions:

$$\begin{aligned} L[\boldsymbol{\phi}] &= \sum_j -\log\Big[1-\mathrm{sig}[\mathrm{f}[\mathbf{g}[\mathbf{z}_j,\boldsymbol{\theta}],\boldsymbol{\phi}]]\Big] - \sum_i \log\Big[\mathrm{sig}[\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}]]\Big] \\ L[\boldsymbol{\theta}] &= \sum_j \log\Big[1-\mathrm{sig}[\mathrm{f}[\mathbf{g}[\mathbf{z}_j,\boldsymbol{\theta}],\boldsymbol{\phi}]]\Big], \end{aligned} \tag{15.5}$$

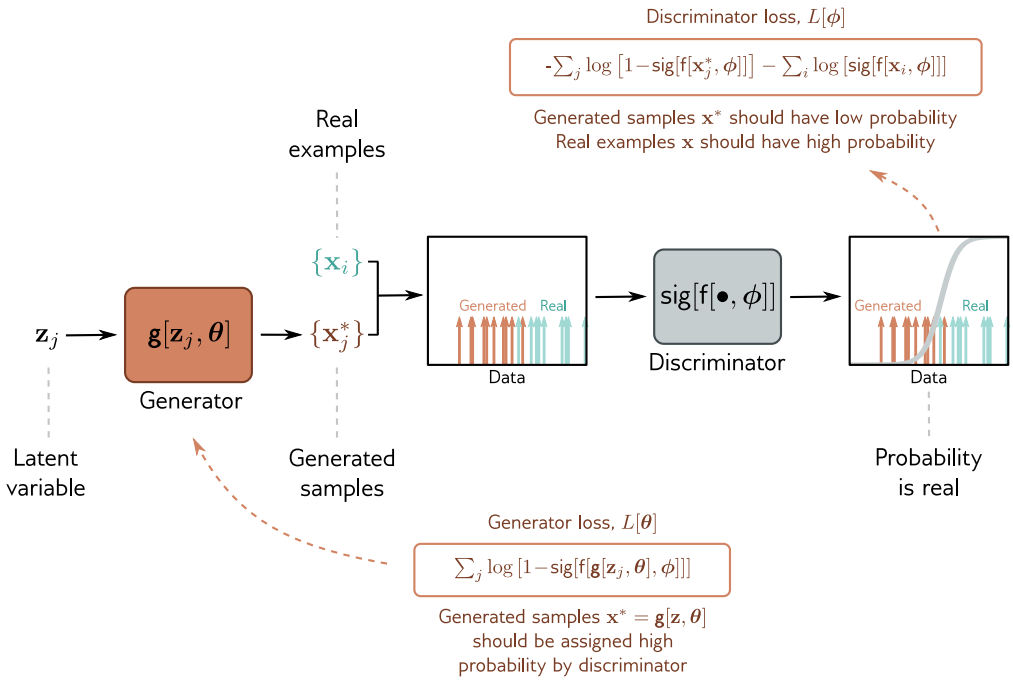

Figure 15.2 GAN loss functions. A latent variable $\mathbf{z}_j$ is drawn from the base distribution and passed through the generator to create a sample $\mathbf{x}^*$. A batch $\{\mathbf{x}_j^*\}$ of samples and a batch of real examples $\{\mathbf{x}_i\}$ are passed to the discriminator, which assigns a probability that each is real. The discriminator parameters $\boldsymbol{\phi}$ are modified to assign high probability to the real examples and low probability to the generated samples. The generator parameters $\boldsymbol{\theta}$ are modified to "fool" the discriminator into assigning the generated samples a high probability.

where we multiplied the second function by minus one to convert to a minimization problem and dropped the second term, which has no dependence on $\boldsymbol{\theta}$. Minimizing the first loss function trains the discriminator. Minimizing the second trains the generator.

At each step, we draw a batch of latent variables $\mathbf{z}_j$ from the base distribution and pass these through the generator to create samples $\mathbf{x}_j^* = \mathbf{g}[\mathbf{z}_j,\boldsymbol{\theta}]$. Then we choose a batch of real training examples $\mathbf{x}_i$. Given the two batches, we can now perform one or more gradient descent steps on each loss function (figure 15.2).

15.1.3 Deep convolutional GAN

The *deep convolutional GAN* or *DCGAN* was an early GAN architecture specialized for generating images (figure 15.3). The input to the generator $\mathbf{g}[\mathbf{z},\boldsymbol{\theta}]$ is a 100D latent

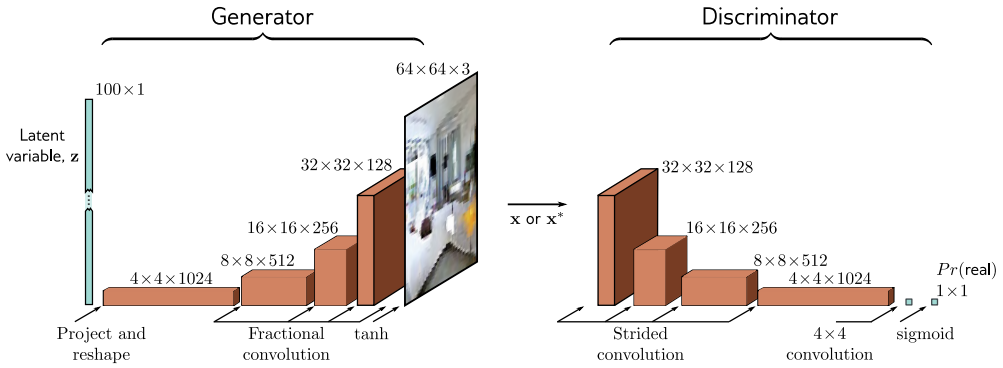

Figure 15.3 DCGAN architecture. In the generator, a 100D latent variable $\mathbf{z}$ is drawn from a uniform distribution and mapped by a linear transformation to a 4×4 representation with 1024 channels. This is then passed through a series of convolutional layers that gradually upsample the representation and decrease the number of channels. At the end is a tanh function that maps the 64×64×3 representation to a fixed range so that it can represent an image. The discriminator consists of a standard convolutional net that classifies the input as either a real example or a generated sample.

variable $\mathbf{z}$ sampled from a uniform distribution. This is then mapped to a 4×4 spatial representation with 1024 channels using a linear transformation. Four convolutional layers follow, each of which uses a fractionally-strided convolution that doubles the resolution (i.e., a convolution with a stride of 0.5). At the final layer, the 64×64×3 signal is passed through a tanh function to generate an image $\mathbf{x}^*$ in the range $[-1, 1]$. The discriminator $\mathrm{f}[\bullet, \boldsymbol{\phi}]$ is a standard convolutional network where the final convolutional layer reduces the size to 1×1 with one channel. This single number is passed through a sigmoid function $\mathrm{sig}[\bullet]$ to create the output probability.

After training, the discriminator is discarded. To create new samples, latent variables $\mathbf{z}$ are drawn from the base distribution and passed through the generator. Example results are shown in figure 15.4.

15.1.4 Difficulty training GANs

Theoretically, the GAN is fairly straightforward. However, GANs are notoriously difficult to train. For example, to get the DCGAN to train reliably, it was necessary to (i) use strided convolutions for upsampling and downsampling; (ii) use BatchNorm in both generator and discriminator except in the last and first layers, respectively; (iii) use the leaky ReLU activation function (figure 3.13) in the discriminator; and (iv) use the Adam optimizer but with a lower momentum coefficient than usual. This is unusual. Most deep learning models are relatively robust to such choices.

Figure 15.4 Synthesized images from the DCGAN model. a) Random samples drawn from DCGAN trained on a faces dataset. b) Random samples using the ImageNet database (see figure 10.15). c) Random samples drawn from the LSUN scene understanding dataset. Adapted from Radford et al. (2015).

Figure 15.5 Mode collapse. Synthesized images from a GAN trained on the LSUN scene understanding dataset using an MLP generator with a similar number of parameters and layers to the DCGAN. The samples are low quality, and many are similar. Adapted from Arjovsky et al. (2017).

A common failure mode is that the generator makes plausible samples, but these only represent a subset of the data (e.g., for faces, it might never generate faces with beards). This is known as *mode dropping*. An extreme version of this phenomenon can occur where the generator entirely or mostly ignores the latent variables $\mathbf{z}$ and collapses all samples to one or a few points; this is known as *mode collapse* (figure 15.5).

15.2 Improving stability

To understand *why* GANs are difficult to train, it's necessary to understand exactly *what* the loss function represents.

15.2.1 Analysis of GAN loss function

If we divide the two sums in the first line of equation 15.5 by the numbers I, J of real and generated samples, then the loss function can be written in terms of expectations:

$$\begin{aligned} L[\boldsymbol{\phi}] &= -\frac{1}{J}\sum_{j=1}^{J}\Big(\log\Big[1-\text{sig}[\text{f}[\mathbf{x}_j^*,\boldsymbol{\phi}]]\Big]\Big) - \frac{1}{I}\sum_{i=1}^{I}\Big(\log\Big[\text{sig}[\text{f}[\mathbf{x}_i,\boldsymbol{\phi}]]\Big]\Big) \\ &\approx -\mathbb{E}_{\mathbf{x}^*}\Big[\log\Big[1-\text{sig}[\text{f}[\mathbf{x}^*,\boldsymbol{\phi}]]\Big]\Big] - \mathbb{E}_{\mathbf{x}}\Big[\log\Big[\text{sig}[\text{f}[\mathbf{x},\boldsymbol{\phi}]]\Big]\Big] \\ &= -\int Pr(\mathbf{x}^*)\log\Big[1-\text{sig}[\text{f}[\mathbf{x}^*,\boldsymbol{\phi}]]\Big]d\mathbf{x}^* - \int Pr(\mathbf{x})\log\Big[\text{sig}[\text{f}[\mathbf{x},\boldsymbol{\phi}]]\Big]d\mathbf{x}, \end{aligned} \tag{15.6}$$

where $Pr(\mathbf{x}^*)$ is the probability distribution over the generated samples, and $Pr(\mathbf{x})$ is the true probability distribution over the real examples.

When $I = J$, the optimal discriminator for an example $\tilde{\mathbf{x}}$ of unknown origin is:

$$Pr(\text{real}|\tilde{\mathbf{x}}) = \text{sig}\big[\text{f}[\tilde{\mathbf{x}},\boldsymbol{\phi}]\big] = \frac{Pr(\tilde{\mathbf{x}}|\text{real})}{Pr(\tilde{\mathbf{x}}|\text{generated}) + Pr(\tilde{\mathbf{x}}|\text{real})} = \frac{Pr(\mathbf{x})}{Pr(\mathbf{x}^*)+Pr(\mathbf{x})}, \tag{15.7}$$

where on the right hand side, we evaluate $\tilde{\mathbf{x}}$ against the generated distribution $Pr(\mathbf{x}^*)$ and the real distribution $Pr(\mathbf{x})$. Substituting into equation 15.6, we get:

$$\begin{aligned} L[\boldsymbol{\phi}] &= -\int Pr(\mathbf{x}^*)\log\Big[1-\text{sig}[\text{f}[\mathbf{x}^*,\boldsymbol{\phi}]]\Big]d\mathbf{x}^* - \int Pr(\mathbf{x})\log\Big[\text{sig}[\text{f}[\mathbf{x},\boldsymbol{\phi}]]\Big]d\mathbf{x} \\ &= -\int Pr(\mathbf{x}^*)\log\left[1-\frac{Pr(\mathbf{x})}{Pr(\mathbf{x}^*)+Pr(\mathbf{x})}\right]d\mathbf{x}^* - \int Pr(\mathbf{x})\log\left[\frac{Pr(\mathbf{x})}{Pr(\mathbf{x}^*)+Pr(\mathbf{x})}\right]d\mathbf{x} \\ &= -\int Pr(\mathbf{x}^*)\log\left[\frac{Pr(\mathbf{x}^*)}{Pr(\mathbf{x}^*)+Pr(\mathbf{x})}\right]d\mathbf{x}^* - \int Pr(\mathbf{x})\log\left[\frac{Pr(\mathbf{x})}{Pr(\mathbf{x}^*)+Pr(\mathbf{x})}\right]d\mathbf{x}. \end{aligned} \tag{15.8}$$

Disregarding additive and multiplicative constants, this is the Jensen-Shannon divergence between the synthesized distribution $Pr(x^*)$ and the true distribution $Pr(x)$:

Problems 15.1–15.2

Appendix C.5.2 Jensen-Shannon divergence

$$\begin{aligned} &D_{JS}\Big[Pr(\mathbf{x}^*)\,||\,Pr(\mathbf{x})\Big] \\ &\quad= \frac{1}{2}D_{KL}\left[Pr(\mathbf{x}^*)\,\middle|\middle|\,\frac{Pr(\mathbf{x}^*)+Pr(\mathbf{x})}{2}\right] + \frac{1}{2}D_{KL}\left[Pr(\mathbf{x})\,\middle|\middle|\,\frac{Pr(\mathbf{x}^*)+Pr(\mathbf{x})}{2}\right] \\ &\quad= \frac{1}{2}\int \underbrace{Pr(\mathbf{x}^*)\log\left[\frac{2Pr(\mathbf{x}^*)}{Pr(\mathbf{x}^*)+Pr(\mathbf{x})}\right]d\mathbf{x}^*}_{\text{quality}} + \underbrace{\frac{1}{2}\int Pr(\mathbf{x})\log\left[\frac{2Pr(\mathbf{x})}{Pr(\mathbf{x}^*)+Pr(\mathbf{x})}\right]d\mathbf{x}}_{\text{coverage}}. \end{aligned} \tag{15.9}$$

where $D_{KL}[\bullet||\bullet]$ is the Kullback-Leibler divergence.

Appendix C.5.1 Kullback-Leibler divergence

The first term indicates the distance will be small if, wherever the sample density $Pr(\mathbf{x}^*)$ is high, the mixture $(Pr(\mathbf{x}^*) + Pr(\mathbf{x}))/2$ has high probability. In other words, it penalizes regions with samples $\mathbf{x}^*$ but no real examples $\mathbf{x}$; it enforces *quality*. The second term says that the distance will be small if, wherever the true density $Pr(\mathbf{x})$

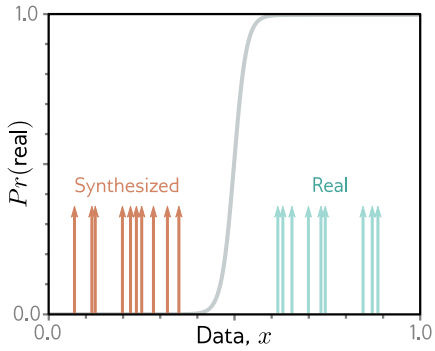

Figure 15.6 Problem with GAN loss function. If the generated samples (orange arrows) are easy to distinguish from the real examples (cyan arrows), then the discriminator (sigmoid) may have a very shallow slope at the positions of the samples; hence, the gradient to update the parameter of the generator may be tiny.

is high, the mixture $(Pr(\mathbf{x}^*) + Pr(\mathbf{x}))/2$ has high probability. In other words, it penalizes regions with real examples but no samples. It enforces *coverage*. Referring to equation 15.6, we see that the second term does not depend on the generator, which consequently doesn't care about coverage; it is happy to generate a subset of possible examples accurately. This is the putative reason for mode dropping.

15.2.2 Vanishing gradients

In the previous section, we saw that when the discriminator is optimal, the loss function minimizes a measure of the distance between the generated and real samples. However, there is a potential problem with using this distance between probability distributions as the criterion for optimizing GANs. If the probability distributions are completely disjoint, this distance is infinite, and any small change to the generator will not decrease the loss. The same phenomenon can be seen when we consider the original formulation; if the discriminator can perfectly separate the generated and real samples, no small change to the generated data will change the classification score (figure 15.6).

Unfortunately, the distributions of generated samples and real examples may *really be* disjoint; the generated samples lie in a subspace that is the size of the latent variable $\mathbf{z}$, and the real examples also lie in a low-dimensional subspace due to the physical processes that created the data (figure 1.9). There may be little or no overlap between these subspaces, and the result is very small or no gradients.

Figure 15.7 provides empirical evidence to support this hypothesis. If the DCGAN generator is frozen and the discriminator is updated repeatedly so that its classification performance improves, the generator gradients decrease. In short, there is a very fine balance between the quality of the discriminator and the generator; if the discriminator becomes too good, the training updates of the generator are attenuated.

15.2.3 Wasserstein distance

The previous sections showed that (i) the GAN loss can be interpreted in terms of distances between probability distributions and that (ii) the gradient of this distance

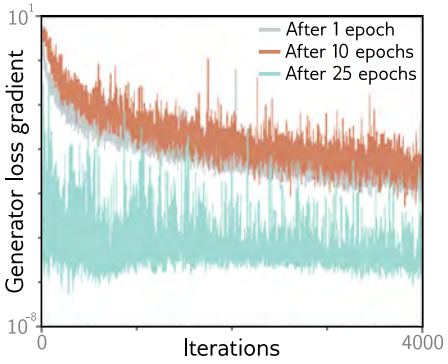

Figure 15.7 Vanishing gradients in the generator of a DCGAN. The generator is frozen after 1, 10, and 25 epochs, and the discriminator is trained further. The gradient of the generator decreases rapidly (note log scale); if the discriminator becomes too accurate, the gradients for the generator vanish. Adapted from Arjovsky & Bottou (2017).

becomes zero when the generated samples are too easy to distinguish from the real examples. The obvious way forward is to choose a distance metric with better properties.

The *Wasserstein* or (for discrete distributions) *earth mover's* distance is the quantity of work required to transport the probability mass from one distribution to create the other. Here, "work" is defined as the mass multiplied by the distance moved. This immediately sounds more promising; the Wasserstein distance is well-defined even when the distributions are disjoint and decreases smoothly as they become closer to one another.

15.2.4 Wasserstein distance for discrete distributions

The Wasserstein distance is easiest to understand for discrete distributions (figure 15.8). Consider distributions $Pr(x=i)$ and $q(x=j)$ defined over K bins. Assume there is a cost C_{ij} associated with moving one unit of mass from bin i in the first distribution to bin j in the second; this cost might be the absolute difference $|i-j|$ between the indices. The amounts that are moved form the *transport plan* and are stored in a matrix $\mathbf{P}$.

The Wasserstein distance is defined as:

$$D_w\Big[Pr(x)||q(x)\Big] = \min_{\mathbf{P}}\left[\sum_{i,j} P_{ij}\cdot|i-j|\right], \tag{15.10}$$

subject to the constraints that:

$$\begin{array}{rcll} \sum_j P_{ij} & = & Pr(x=i) & \text{initial distribution of } Pr(x) \\ \sum_i P_{ij} & = & q(x=j) & \text{initial distribution of } q(x) \\ P_{ij} & \geq & 0 & \text{non-negative masses.} \end{array} \tag{15.11}$$

In other words, the Wasserstein distance is the solution to a constrained minimization problem that maps the mass of one distribution to the other. This is inconvenient as we must solve this minimization problem over the elements P_{ij} every time we want to compute the distance. Fortunately, this is a standard problem that is easily solved for small systems of equations. It is a *linear programming problem* in its *primal form.*

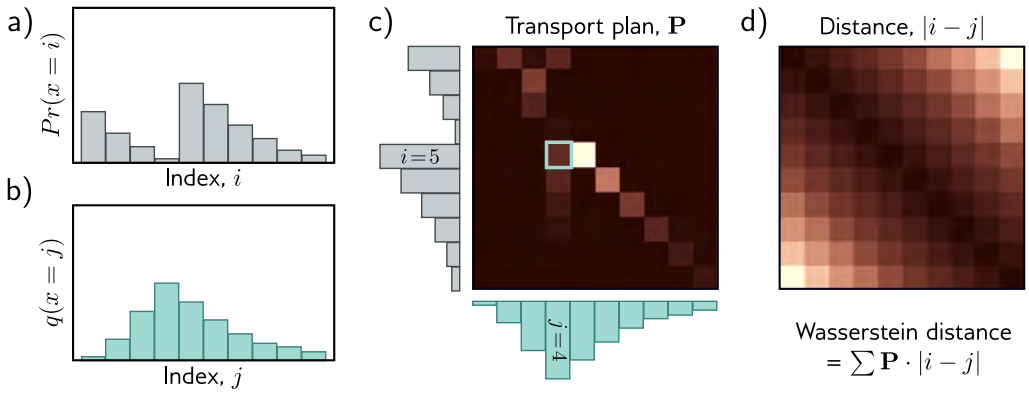

Figure 15.8 Wasserstein or earth mover's distance. a) Consider the discrete distribution $Pr(x=i)$. b) We wish to move the probability mass to create the target distribution $q(x=j)$. c) The transport plan $\mathbf{P}$ identifies how much mass will be moved from i to j. For example, the cyan highlighted square p_{54} indicates how much mass will be moved from $i=5$ to $j=4$. The elements of the transport plan must be non-negative, the sum over j must be $Pr(x=i)$, and the sum over i must be $q(x=j)$. Hence $\mathbf{P}$ is a joint probability distribution. d) The distance matrix between elements i and j. The optimal transport plan $\mathbf{P}$ minimizes the sum of the pointwise product of $\mathbf{P}$ and the distance matrix (termed the Wasserstein distance). Hence, the elements of $\mathbf{P}$ tend to lie close to the diagonal where the distance cost is lowest. Adapted from Hermann (2017).

primal form				**dual form**			
minimize	$\mathbf{c}^T\mathbf{p}$,			maximize	$\mathbf{b}^T\mathbf{f}$,		
such that	$\mathbf{Ap}$	$=$	$\mathbf{b}$	such that	$\mathbf{A}^T\mathbf{f}$	$\leq$	$\mathbf{c}$
and	$\mathbf{p}$	$\geq$	$\mathbf{0}$				

where $\mathbf{p}$ contains the vectorized elements P_{ij} that determine the amount of mass moved, $\mathbf{c}$ contains the distances, $\mathbf{Ap}=\mathbf{b}$ contains the initial distribution constraints, and $\mathbf{p} \geq 0$ ensures the masses moved are non-negative.[1]

Problem 15.3

As for all linear programming problems, there is an equivalent *dual problem* with the same solution. Here, we maximize with respect to a variable $\mathbf{f}$ that is applied to the initial distributions, subject to constraints that depend on the distances $\mathbf{c}$. The solution to this dual problem is:

Notebook 15.2 Wasserstein distance

$$D_w\Big[Pr(x)||q(x)\Big] = \max_{\mathbf{f}}\left[\sum_i Pr(x=i)f_i - \sum_j q(x=j)f_j\right], \tag{15.12}$$

[1]The mathematical background is omitted due to space constraints. Linear programming is a standard problem with well-known algorithms for finding the minimum.

subject to the constraint that:

$$|f_{i+1} - f_i| < 1. \tag{15.13}$$

In other words, we optimize over a new set of variables $\{f_i\}$ where adjacent values cannot change by more than one.

15.2.5 Wasserstein distance for continuous distributions

Translating these results back to the continuous multi-dimensional domain, the equivalent of the primal form (equation 15.10) is:

Problems 15.4–15.5

$$D_w\Big[Pr(\mathbf{x}), q(\mathbf{x})\Big] = \min_{\pi[\bullet,\bullet]} \left[\iint \pi(\mathbf{x}_1, \mathbf{x}_2) \cdot ||\mathbf{x}_1 - \mathbf{x}_2|| d\mathbf{x}_1 d\mathbf{x}_2\right], \tag{15.14}$$

subject to constraints similar to equation 15.11 on the transport plan $\pi(\mathbf{x}_1, \mathbf{x}_2)$ representing the mass moved from position $\mathbf{x}_1$ to $\mathbf{x}_2$. The equivalent of the dual form (equation 15.12) is:

$$D_w\Big[Pr(\mathbf{x}), q(\mathbf{x})\Big] = \max_{\mathrm{f}[\mathbf{x}]} \left[\int Pr(\mathbf{x})\mathrm{f}[\mathbf{x}] d\mathbf{x} - \int q(\mathbf{x})\mathrm{f}[\mathbf{x}] d\mathbf{x}\right], \tag{15.15}$$

subject to the constraint that the Lipschitz constant of the function $\mathrm{f}[\mathbf{x}]$ is less than one (i.e., the absolute gradient of the function is less than one).

Appendix B.1.1 Lipschitz constant

15.2.6 Wasserstein GAN loss function

In the context of neural networks, we maximize over the space of functions $\mathrm{f}[\mathbf{x}]$ by optimizing the parameters $\boldsymbol{\phi}$ in a neural network $\mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]$, and we approximate these integrals using generated samples $\mathbf{x}_i^*$ and real examples $\mathbf{x}_i$:

$$\begin{aligned} L[\boldsymbol{\phi}] &= \sum_j \mathrm{f}[\mathbf{x}_j^*, \boldsymbol{\phi}] - \sum_i \mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}] \\ &= \sum_j \mathrm{f}[\mathbf{g}[\mathbf{z}_j, \boldsymbol{\theta}], \boldsymbol{\phi}] - \sum_i \mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}], \end{aligned} \tag{15.16}$$

where we must constrain the neural network discriminator $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ to have an absolute gradient norm of less than one at every position $\mathbf{x}$:

$$\left|\frac{\partial \mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]}{\partial \mathbf{x}}\right| < 1. \tag{15.17}$$

One way to achieve this is to clip the discriminator weights to a small range (e.g., $[-0.01, 0.01]$). An alternative is the *gradient penalty Wasserstein GAN* or *WGAN-GP*, which adds a regularization term that increases as the gradient norm deviates from unity.

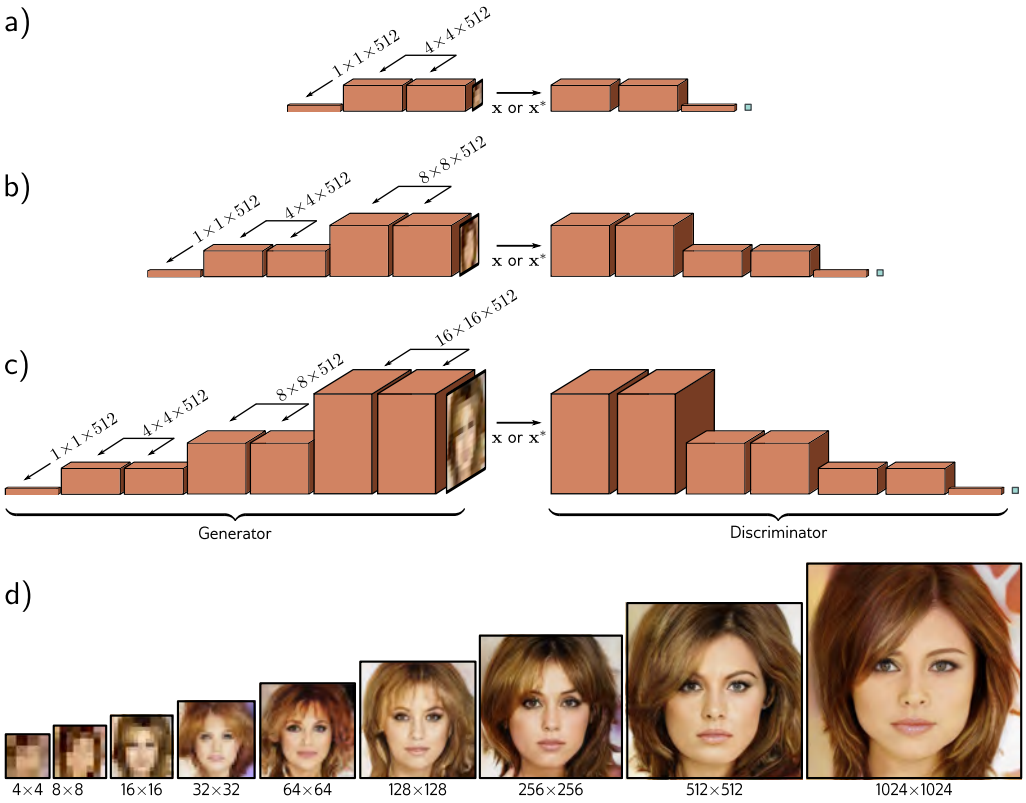

Figure 15.9 Progressive growing. a) The generator is initially trained to create very small (4×4) images, and the discriminator to identify if these images are synthesized or downsampled real images. b) After training at this low-resolution terminates, subsequent layers are added to the generator to generate (8×8) images. Similar layers are added to the discriminator to downsample back again. c) This process continues to create (16×16) images and so on. In this way, a GAN that produces very realistic high-resolution images can be trained. d) Images of increasing resolution generated at different stages from the same latent variable. Adapted from Wolf (2021), using method of Karras et al. (2018).

15.3 Progressive growing, minibatch discrimination, and truncation

The Wasserstein formulation makes GAN training more stable. However, further machinery is needed to generate high-quality images. We now review *progressive growing*, *minibatch discrimination*, and *truncation*, which all improve output quality.

In *progressive growing* (figure 15.9), we first train a GAN that synthesizes 4×4 images using an architecture similar to the DCGAN. Then we add subsequent layers to the generator, which upsample the representation and perform further processing to create

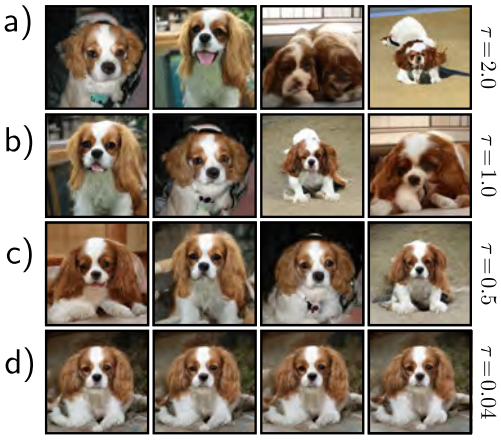

Figure 15.10 Truncation. The quality of GAN samples can be traded off against diversity by rejecting samples from the latent variable $\mathbf{z}$ that fall further than τ standard deviations from the mean. a) If this threshold is large ($\tau = 2.0$), the samples are visually varied but may have defects. b–c) As this threshold is decreased, the average visual quality improves, but the diversity decreases. d) With a very small threshold, the samples look almost identical. By judiciously choosing this threshold, it's possible to increase the average quality of GAN results. Adapted from Brock et al. (2019).

Figure 15.11 Progressive growing. This method generates realistic images of faces when trained on the CELEBA-HQ dataset and more complex, variable objects when trained on LSUN categories. Adapted from Karras et al. (2018).

Figure 15.12 Traversing latent space of progressive GAN trained on LSUN cars. Moving in the latent space produces car images that change smoothly. This usually only works for short trajectories; eventually, the latent variable moves to somewhere that produces unrealistic images. Adapted from Karras et al. (2018).

an 8×8 image. The discriminator also has extra layers added to it so that it can receive the higher-resolution images and classify them as either being generated samples or real examples. In practice, the higher-resolution layers gradually "fade in" over time; initially, the higher-resolution image is an upsampled version of the previous result, passed via a residual connection, and the new layers gradually take over.

Mini-batch discrimination ensures that the samples have sufficient variety and hence helps prevent mode collapse. This can be done by computing feature statistics across the mini-batches of synthesized and real data. These can be summarized and added as a feature map (usually toward the end of the discriminator). This allows the discriminator to send a signal back to the generator, encouraging it to include a similar amount of variation in the synthesized data as in the original dataset.

Another trick to improve generation results is *truncation* (figure 15.10), in which only latent variables $\mathbf{z}$ with high probability (i.e., close to the mean) are chosen during sampling. This reduces the variation in the samples but improves their quality. Careful normalization and regularization schemes also improve sample quality. Using combinations of these methods, GANs can synthesize varied and realistic images (figure 15.11). Moving smoothly through the latent space can also sometimes produce realistic interpolations from one synthesized image to another (figure 15.12).

Problem 15.6

15.4 Conditional generation

GANs produce realistic images but don't specify their attributes: we can't choose the hair color, ethnicity, or age of faces, without training separate GANs for each combination of characteristics. *Conditional generation* models provide us with this control.

15.4.1 Conditional GAN

The *conditional GAN* passes a vector $\mathbf{c}$ of attributes to both the generator and discriminator, which are now written as $\mathbf{g}[\mathbf{z}, \mathbf{c}, \boldsymbol{\theta}]$ and $\mathrm{f}[\mathbf{x}, \mathbf{c}, \boldsymbol{\phi}]$, respectively. The generator aims to transform the latent variable $\mathbf{z}$ into a data sample $\mathbf{x}$ with the correct attribute $\mathbf{c}$. The discriminator's goal is to distinguish between (i) the generated sample with the target attribute or (ii) a real example with the real attribute (figure 15.13a).

For the generator, the attribute $\mathbf{c}$ can be appended to the latent vector $\mathbf{z}$. For the discriminator, it may be appended to the input if the data are 1D. If the data comprise images, the attribute can be linearly transformed to a 2D representation and appended as an extra channel to the discriminator input or to one of its intermediate hidden layers.

15.4.2 Auxiliary classifier GAN

The *auxiliary classifier GAN* or *ACGAN* simplifies conditional generation by requiring that the discriminator correctly predicts the attribute (figure 15.13b). For a discrete

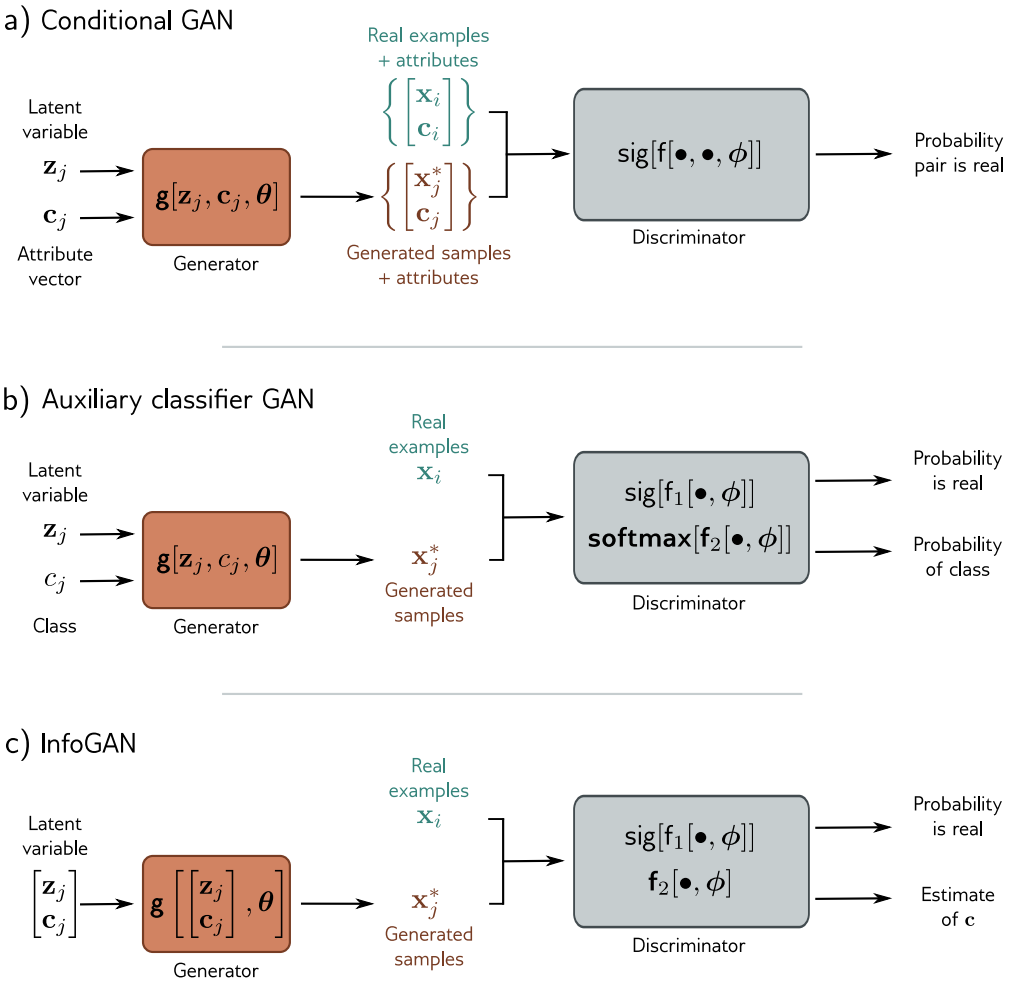

Figure 15.13 Conditional generation. a) The generator of the conditional GAN also receives an attribute vector $\mathbf{c}$ describing some aspect of the image. As usual, the discriminator receives either a real example or a generated sample, but now it also receives the attribute vector; this encourages the samples both to be realistic and compatible with the attribute. b) The generator of the auxiliary classifier GAN (ACGAN) takes a discrete attribute variable. The discriminator must both (i) determine if its input is real or synthetic and (ii) identify the class correctly. c) The InfoGAN splits the latent variable into noise $\mathbf{z}$ and unspecified random attributes $\mathbf{c}$. The discriminator must distinguish if its input is real and also reconstruct these attributes. In practice, this means that the variables $\mathbf{c}$ correspond to salient aspects of the data with real-world interpretations (i.e., the latent space is *disentangled*).

Figure 15.14 Auxiliary classifier GAN. The generator takes a class label as well as the latent vector. The discriminator must both identify if the data point is real *and* predict the class label. This model was trained on ten ImageNet classes. Left to right: generated examples of monarch butterflies, goldfinches, daisies, redshanks, and gray whales. Adapted from Odena et al. (2017).

attribute with C categories, the discriminator takes the real/synthesized image as input and has $C+1$ outputs; the first is passed through a sigmoid function and predicts if the sample is generated or real. The remaining outputs are passed through a softmax function to predict the probability that the data belongs to each of the C classes. Networks trained with this method can synthesize multiple classes from ImageNet (figure 15.14).

15.4.3 InfoGAN

The conditional GAN and ACGAN both generate samples that have predetermined attributes. By contrast, InfoGAN (figure 15.13c) attempts to identify important attributes automatically. The generator takes a vector consisting of random noise variables $\mathbf{z}$ and *random* attribute variables $\mathbf{c}$. The discriminator both predicts whether the image is real or synthesized and estimates the attribute variables.

The insight is that interpretable real-world characteristics should be easiest to predict and hence will be represented in the attribute variables $\mathbf{c}$. The attributes in $\mathbf{c}$ may be discrete (and a binary or multiclass cross-entropy loss would be used) or continuous (and a least squares loss would be used). The discrete variables identify categories in the data, and the continuous ones identify gradual modes of variation (figure 15.15).

15.5 Image translation

Although the adversarial discriminator was first used in the context of the GAN for generating random samples, it can also be used as a prior that favors realism in tasks that translate one data example into another. This is most commonly done with images,

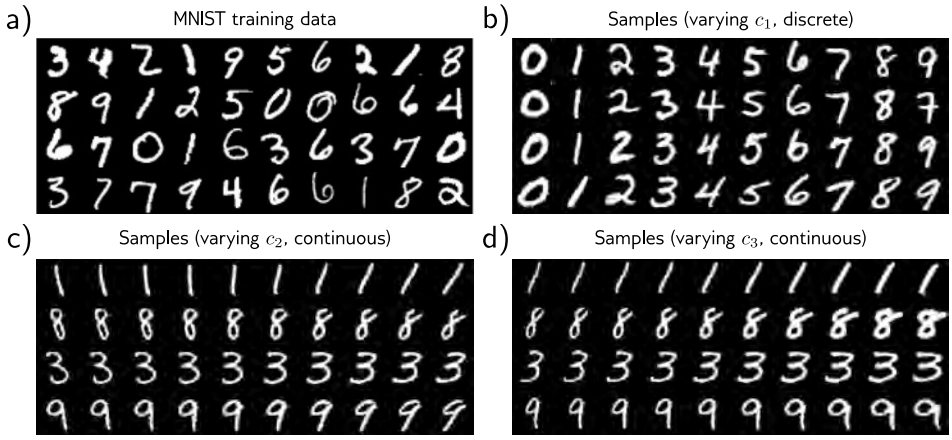

Figure 15.15 InfoGAN for MNIST. a) Training examples from the MNIST database, which consists of 28×28 pixel images of handwritten digits. b) The first attribute c_1 is categorical with 10 categories; each column shows samples generated with one of these categories. The InfoGAN recovers the ten digits. The attribute vectors c_2 and c_3 are continuous. c) Moving from left to right, each column represents a different value of c_2 while keeping the other latent variables constant. This attribute seems to correspond to the orientation of the character. d) The third attribute seems to correspond to the thickness of the stroke. Adapted from Chen et al. (2016b).

where we might want to translate a grayscale image to color, a noisy image to a clean one, a blurry image to a sharp one, or a sketch to a photo-realistic image.

This section discusses three image translation models that use different amounts of manual labeling. The Pix2Pix model uses before/after pairs for training. Models with adversarial losses use before/after pairs for the main model but also exploit unpaired "after" images in the discriminator. The CycleGAN model uses unpaired images.

15.5.1 Pix2Pix

The Pix2Pix model (figure 15.16) is a network $\mathbf{x} = \mathbf{g}[\mathbf{c}, \boldsymbol{\theta}]$ that maps one image $\mathbf{c}$ to a different style image $\mathbf{x}$ using a U-Net (figure 11.10) with parameters $\boldsymbol{\theta}$. A typical use case would be colorization, where the input is grayscale, and the output is color. The output should be similar to the input, and this is encouraged using a *content loss* that penalizes the ℓ_1 norm $||\mathbf{x} - \mathbf{g}[\mathbf{c}, \boldsymbol{\theta}]||_1$ between the input and output.

Appendix B.3.2 ℓ_1 norm

However, the output image should also look like a realistic conversion of the input. This is encouraged by using an adversarial discriminator $\mathbf{f}[\mathbf{c}, \mathbf{x}, \boldsymbol{\phi}]$, which ingests the before and after images $\mathbf{c}$ and $\mathbf{x}$. At each step, the discriminator tries to distinguish between a real before/after pair and a before/synthesized pair. To the extent that these

can be distinguished successfully, a feedback signal is provided to modify the U-Net to make its output more realistic. Since the content loss ensures that the large-scale image structure is correct, the discriminator is mainly needed to ensure that the local texture is plausible. To this end, the *PatchGAN* loss is based on a purely convolutional classifier. At the last layer, each hidden unit indicates whether the region within its receptive field is real or synthesized. These responses are averaged to provide the final output.

One way to think of this model is that it is a conditional GAN where the U-Net is the generator and is conditioned on an image rather than a label. Notice, though, that the U-Net input does not include noise and so is not really a "generator" in the conventional sense. Interestingly, the original authors experimented with adding noise $\mathbf{z}$ to the U-Net in addition to the input image $\mathbf{c}$. However, the network just learned to ignore it.

15.5.2 Adversarial loss

The discriminator of the Pix2Pix model attempted to distinguish whether before/after pairs in an image translation task were plausible. This has the disadvantage that we need ground truth before/after pairs to exploit the discriminator loss. Fortunately, there is a simpler way to exploit the power of adversarial discriminators in the context of supervised learning without the need for additional labeled training data.

An *adversarial loss* adds a penalty if a discriminator can distinguish the output of a supervised network from a real example from its output domain. Accordingly, the supervised model changes its predictions to decrease this penalty. This may be done at the scale of the entire output or at the level of patches, as in the Pix2Pix algorithm. This helps improve the *realism* of complex structured outputs. However, it doesn't necessarily lead to a better solution in terms of the original loss function.

The *super-resolution GAN* or *SRGAN* uses this approach (figure 15.17). The main model consists of a convolutional network with residual connections that ingests a low-resolution image and converts this via upsampling layers to a high-resolution image. The network is trained with three losses. The content loss measures the squared difference between the output and the true high-resolution image. The *VGG loss* or *perceptual loss* passes the synthesized and ground truth outputs through the VGG network and measures the squared difference between their activations. This encourages the image to be semantically similar to the target. Finally, the adversarial loss uses a discriminator that attempts to distinguish whether this is a real high-resolution image or an upsampled one. This encourages the output to be indistinguishable from real examples.

15.5.3 CycleGAN

The adversarial loss assumes that we have labeled before/after images for the main supervised network. The *CycleGAN* addresses the situation where we have two sets of data with distinct styles but *no* matching pairs. An example is converting a photo to the artistic style of Monet. There exist many photos and many Monet paintings, but no correspondence between them. CycleGAN exploits the idea that converting an image in

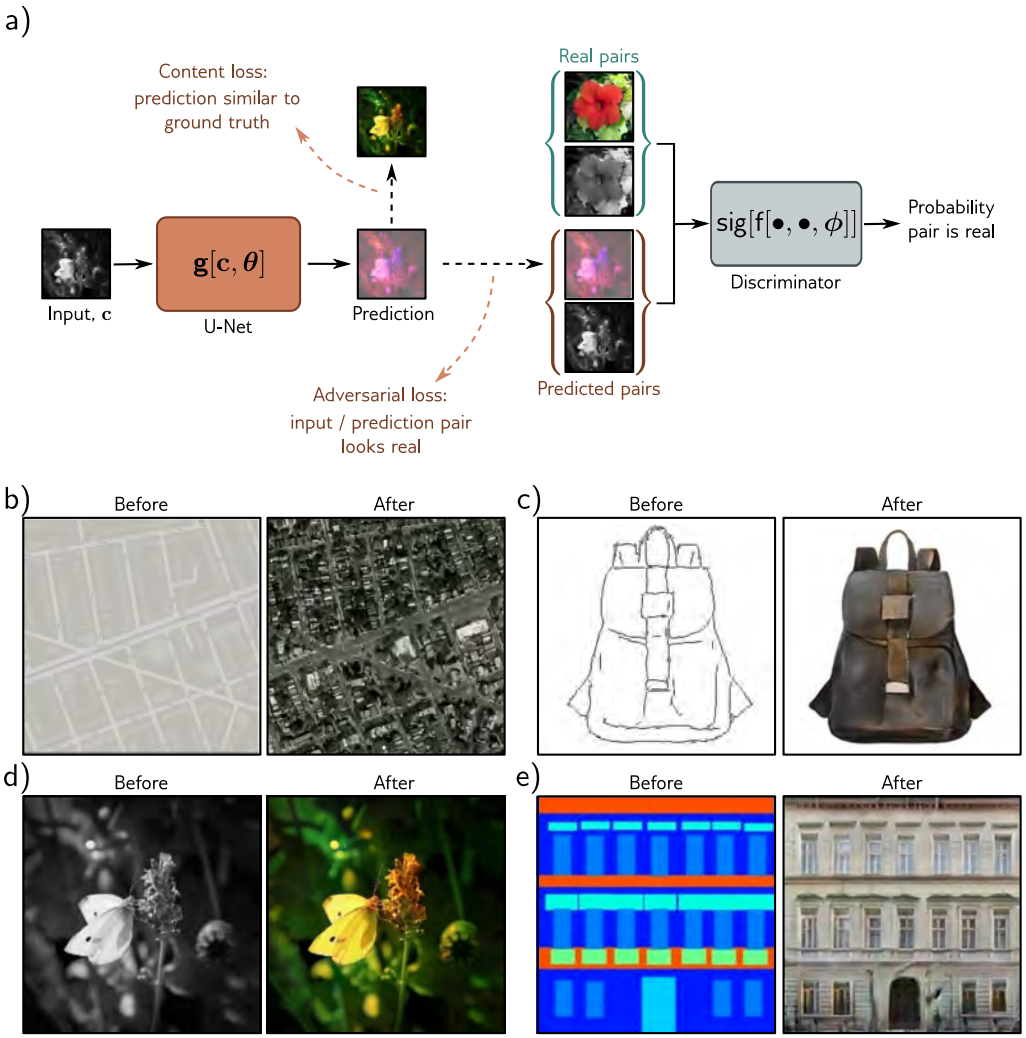

Figure 15.16 Pix2Pix model. a) The model translates an input image to a prediction in a different style using a U-Net (see figure 11.10). In this case, it maps a grayscale image to a plausibly colored version. The U-Net is trained with two losses. First, the content loss encourages the output image to have a similar structure to the input image. Second, the adversarial loss encourages the grayscale/color image pair to be indistinguishable from a real pair in each local region of these images. This framework can be adapted to many tasks, including b) translating maps to satellite imagery, c) converting sketches of bags to photorealistic examples, d) colorization, and e) converting label maps to photorealistic building facades. Adapted from Isola et al. (2017).

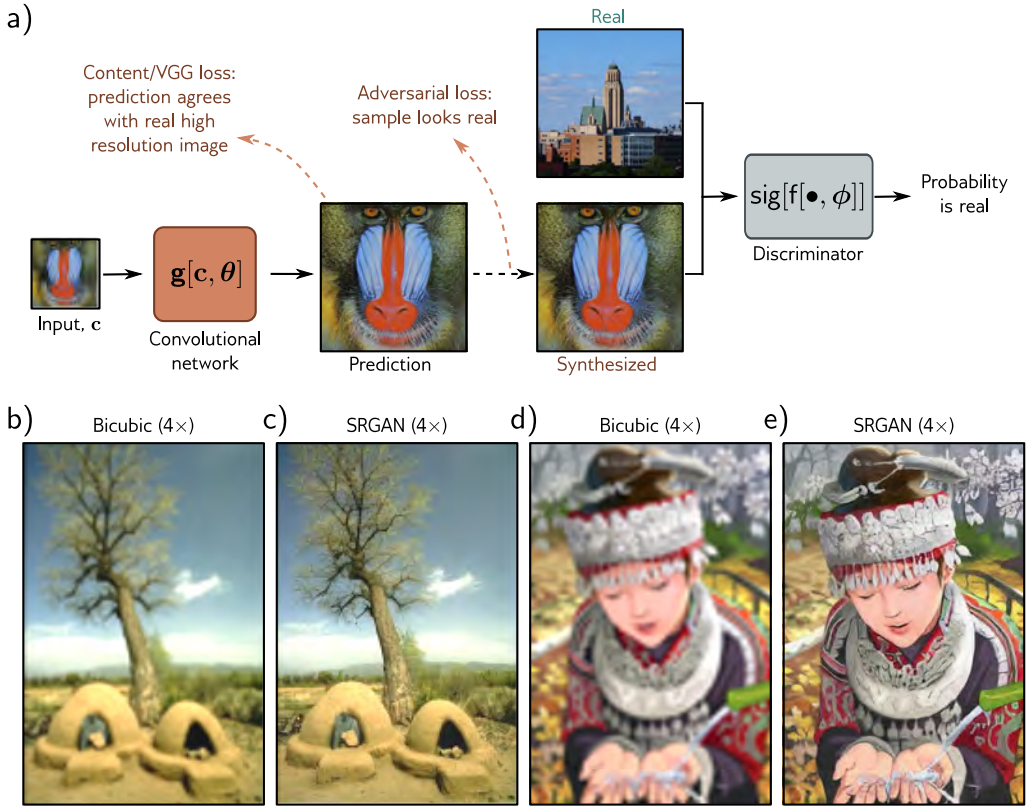

Figure 15.17 Super-resolution generative adversarial network (SRGAN). a) A convolutional network with residual connections is trained to increase the resolution of images by a factor of four. The model has losses that encourage the content to be close to the true high-resolution image. However, it also includes an adversarial loss, which penalizes results that can be distinguished from real high-resolution images. b) Upsampled image using bicubic interpolation. c) Upsampled image using SRGAN. d) Upsampled image using bicubic interpolation. e) Upsampled image using SRGAN. Adapted from Ledig et al. (2017).

one direction (e.g., photo→Monet) and then back again should recover the original.

The CycleGAN loss function is a weighted sum of three losses (figure 15.18). The content loss encourages the before and after images to be similar and is based on the ℓ_1 norm. The adversarial loss uses a discriminator to encourage the output to be indistinguishable from real examples of the target domain. Finally, the *cycle-consistency* loss encourages the mapping to be reversible. Here, two models are trained together. One maps from the first domain to the second, and the other in the opposite direction. The cycle-consistency loss will be low if the translated image can be itself translated successfully back to the image in the original domain. The model combines these three losses to train networks to translate images from one style to another and back again.

15.6 StyleGAN

StyleGAN is a more contemporary GAN that partitions the variation in a dataset into meaningful components, each of which is controlled by a subset of the latent variables. In particular, StyleGAN controls the output image at different scales and separates style from noise. For face images, large-scale changes include face shape and head pose, medium-scale changes include the shape and details of facial features, and fine-scale changes include hair and skin color. The style components represent aspects of the image that are salient to human beings, and the noise aspects represent unimportant variation such as the exact placement of hairs, stubble, freckles, or skin pores.

The GANs that we have seen until now started from a latent variable $\mathbf{z}$ which is drawn from a standard base distribution. This was passed through a series of convolutional layers to produce the output image. However, the latent variable inputs to the generator can (i) be introduced at various points in the architecture and (ii) modify the current representation at these points in different ways. StyleGAN makes these choices judiciously to control scale and to separate style from noise (figure 15.19).

The main generative branch of StyleGAN starts with a learned constant 4×4 representation with 512 channels. This passes through a series of convolutional layers that gradually upsample the representation to generate the image at its final resolution. Two sets of random latent variables representing style and noise are introduced at each scale; the closer that they are to the output, the finer scale details they represent.

The latent variables that represent noise are independently sampled Gaussian vectors $\mathbf{z}_1, \mathbf{z}_2 \ldots$ and are injected additively after each convolution operation in the main generative pipeline. They are the same spatial size as the main representation at the point that they are added but are multiplied by learned per-channel scaling factors $\boldsymbol{\psi}_1, \boldsymbol{\psi}_2 \ldots$ and so contribute in different amounts to each channel. As the resolution of the network increases, this noise contributes at finer scales.

The latent variables that represent style begin as a 1×1×512 noise tensor, which is passed through a seven-layer fully connected network to create an intermediate variable $\mathbf{w}$. This allows the network to decorrelate aspects of style so that each dimension of $\mathbf{w}$ can represent an independent real-world factor such as head pose or hair color. This variable $\mathbf{w}$ is linearly transformed to a 2×1×512 tensor $\mathbf{y}$, which is used to set the per-channel mean and variance of the representation across spatial positions in the

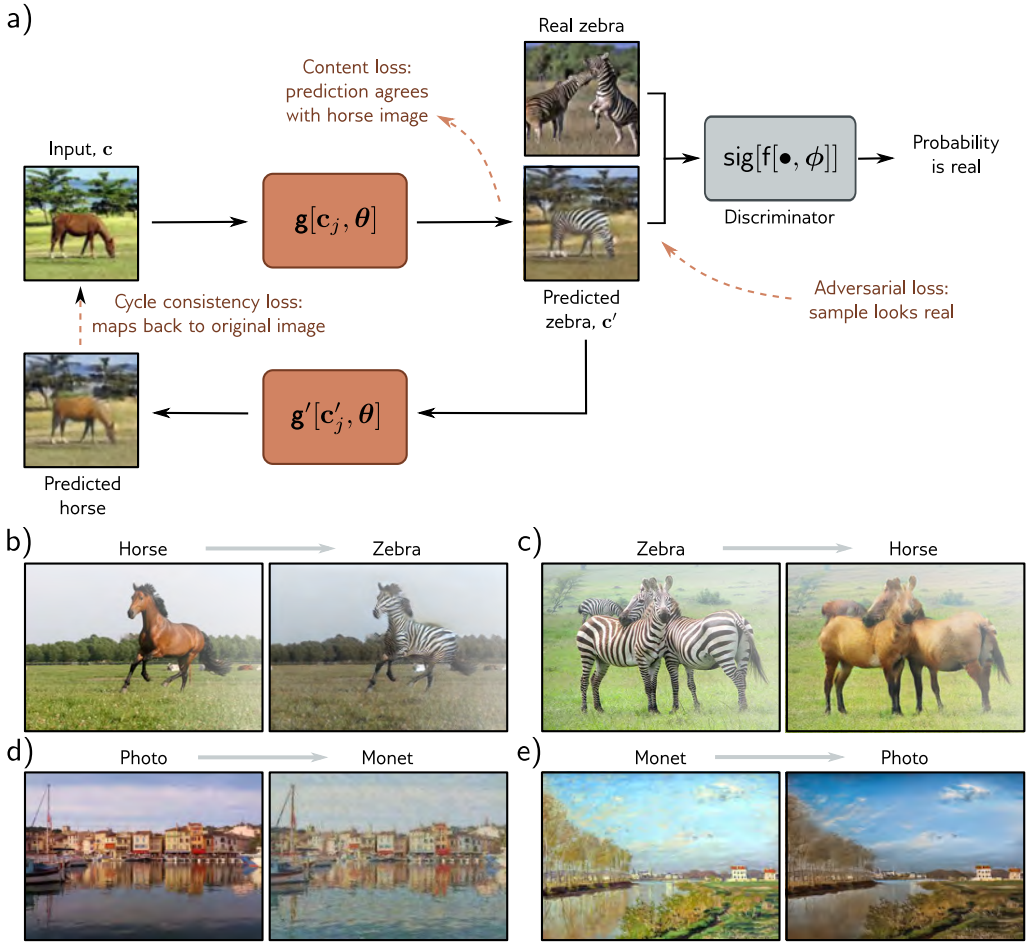

Figure 15.18 CycleGAN. Two models are trained simultaneously. The first $\mathbf{c}' = \mathbf{g}[\mathbf{c}_j, \boldsymbol{\theta}]$ translates from an image $\mathbf{c}$ in the first style (horse) to an image $\mathbf{c}'$ in the second style (zebra). The second model $\mathbf{c} = \mathbf{g}'[\mathbf{c}', \boldsymbol{\theta}]$ learns the opposite mapping. The cycle consistency loss penalizes both models if they cannot successfully convert an image to the other domain and back to the original. In addition, two adversarial losses encourage the translated images to look like realistic examples of the target domain (shown here for zebra only). Two content losses encourage the details and layout of the images before and after each mapping to be similar (i.e., the zebra is in the same position and pose that the horse was and against the same background and vice versa). Adapted from Zhu et al. (2017).

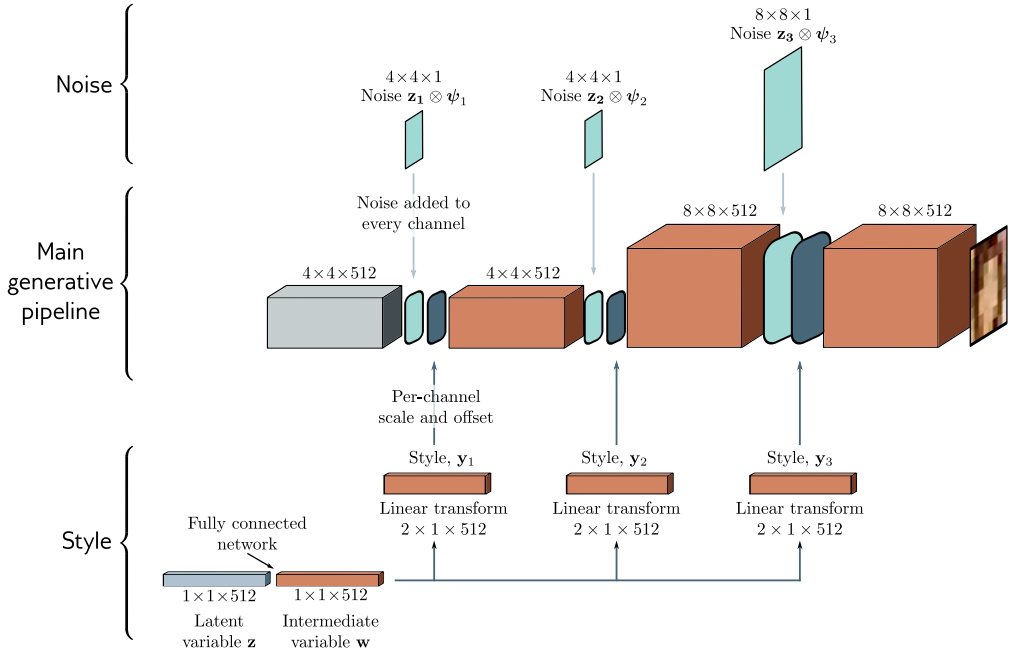

Figure 15.19 StyleGAN. The main pipeline (center row) starts with a constant learned representation (gray box). This is passed through a series of convolutional layers and gradually upsampled to create the output. Noise (top row) is added at different scales by periodically adding Gaussian variables $\mathbf{z}_\bullet$ with per-channel scaling $\boldsymbol{\psi}_\bullet$. The Gaussian style variable $\mathbf{z}$ is passed through a fully connected network to create intermediate variable $\mathbf{w}$ (bottom row). This is used to set the mean and variance of each channel at various points in the pipeline.

main branch after noise addition. This is termed *adaptive instance normalization* (figure 11.14e). A series of vectors $\mathbf{y}_1, \mathbf{y}_2, \ldots$ are injected in this way at several different points in the main branch, so the same style contributes at different scales. Figure 15.20 shows examples of manipulating the style and noise vectors are different scales.

15.7 Summary

GANs learn a generator network that transforms random noise into data that is indistinguishable from a training set. To this end, the generator is trained using a discriminator network that tries to distinguish real examples from generated samples. The generator is then updated so that the data that it creates is identified as being more "real" by the discriminator. The original formulation of this idea has the flaw that the training signal is weak when it's easy to determine if the samples are real or generated. This led to the

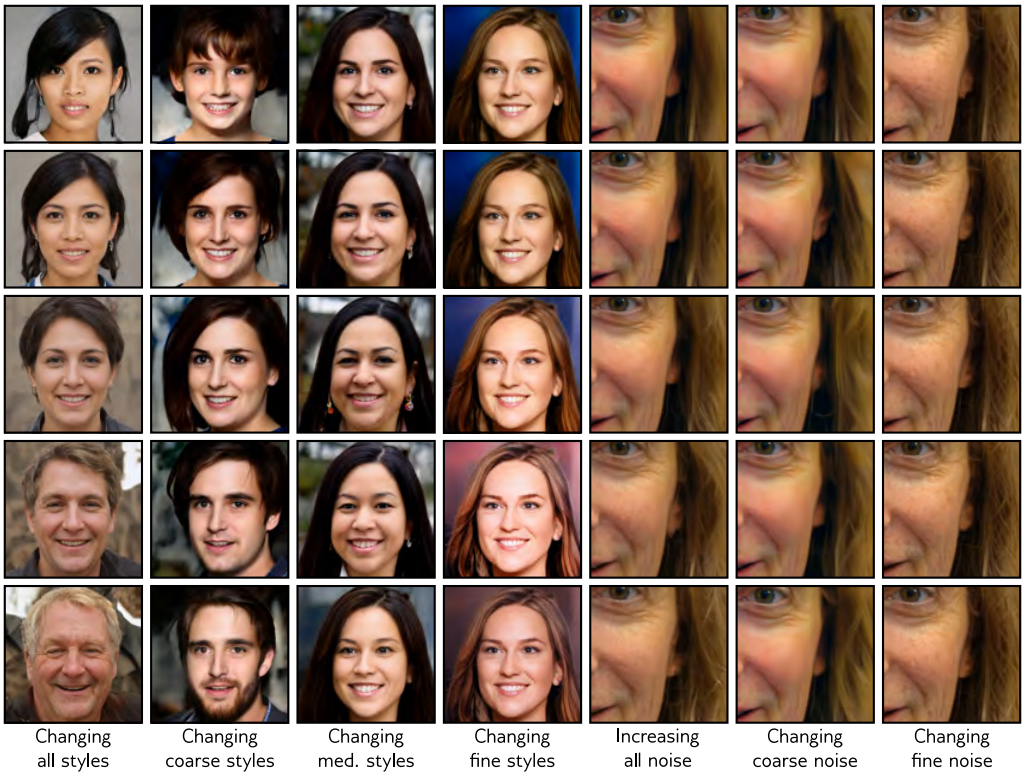

Figure 15.20 StyleGAN results. First four columns show systematic changes in style at various scales. Fifth column shows the effect of increasing noise magnitude. Last two columns show different noise vectors at two different scales.

Wasserstein GAN, which provides a more consistent training signal.

We reviewed convolutional GANs for generating images and a series of tricks that improve the quality of the generated images, including progressive growing, mini-batch discrimination, and truncation. Conditional GAN architectures introduce an auxiliary vector that allows control over the output (e.g., the choice of object class). Image translation tasks retain this conditional information in the form of an image but dispense with the random noise. The GAN discriminator now works as an additional loss term that favors "realistic" looking images. Finally, we described StyleGAN, which injects noise into the generator strategically to control the style and noise at different scales.

Notes

Goodfellow et al. (2014) introduced generative adversarial networks. An early review of progress can be found in Goodfellow (2016). More recent overviews include Creswell et al. (2018) and

Gui et al. (2021). Park et al. (2021) present a review of GAN models that focuses on computer vision applications. Hindupur (2022) maintains a list of named GAN models (numbering 501 at the time of writing) from ABC-GAN (Susmelj et al., 2017) right through to ZipNet-GAN (Zhang et al., 2017b). Odena (2019) lists open problems concerning GANs.

Data: GANs have primarily been developed for image data. Examples include the deep convolutional GAN (Radford et al., 2015), progressive GAN (Karras et al., 2018), and StyleGAN (Karras et al., 2019) models presented in this chapter. For this reason, most GANs are based on convolutional layers, although more recently, GANs that exploit transformers in the generator and discriminator to capture long-range correlations have been developed (e.g., SAGAN, Zhang et al., 2019b). However, GANs have also been used to generate molecular graphs (De Cao & Kipf, 2018), voice data (Saito et al., 2017; Donahue et al., 2018b; Kaneko & Kameoka, 2017; Fang et al., 2018), EEG data (Hartmann et al., 2018), text (Lin et al., 2017a; Fedus et al., 2018), music (Mogren, 2016; Guimaraes et al., 2017; Yu et al., 2017), 3D models (Wu et al., 2016), DNA (Killoran et al., 2017), and video data (Vondrick et al., 2016; Wang et al., 2018a).

GAN loss functions: It was originally claimed that GANs converged to Nash equilibria during training. However, more recent evidence suggests that this isn't always the case (Farnia & Ozdaglar, 2020; Jin et al., 2020; Berard et al., 2019). (Arjovsky et al., 2017; Metz et al., 2017; Qi, 2020) identified that the original GAN loss function was unstable, and this led to different formulations. Mao et al. (2017) introduced the least squares GAN. For some parameter choices, this implicitly minimizes the Pearson χ^2 divergence. Nowozin et al. (2016) argue that the Jensen-Shannon divergence is a special case of a larger family of f-divergences and show that any f-divergence can be used for training GANs. Jolicoeur-Martineau (2019) introduces the relativistic GAN in which the discriminator estimates the probability that a real data example is more realistic than a generated one rather than the absolute probability that it is real. Zhao et al. (2017a) reformulate the GAN into a general energy-based framework in which the discriminator is a function that attributes low energies to real data and higher energies elsewhere. As an example, they use an autoencoder and base the energy on reconstruction error.

Arjovsky & Bottou (2017) analyzed vanishing gradients in GANs, and this led to the Wasserstein GAN (Arjovsky et al., 2017), which is based on earth mover's distance/optimal transport. The Wasserstein formulation requires that the Lipschitz constant of the discriminator is less than one; the original paper proposed to clip the weights in the discriminator, but subsequent work imposed a gradient penalty (Gulrajani et al., 2016) or applied spectral normalization (Miyato et al., 2018) to limit the Lipschitz constant. Other variations of the Wasserstein GAN were introduced by Wu et al. (2018a), Bellemare et al. (2017b), and Adler & Lunz (2018). Hermann (2017) presents an excellent blog post discussing duality and the Wasserstein GAN. For more information about optimal transport, consult the book by Peyré et al. (2019). Lucic et al. (2018) present an empirical comparison of GAN loss functions of the time.

Tricks for training GANs: Many heuristics improve the stability of training GANs and the quality of the final results. Marchesi (2017) first used the truncation trick (figure 15.10) to trade off the variability of GAN outputs relative to their quality. This was also proposed by Pieters & Wiering (2018) and Brock et al. (2019), who added a regularizer that encourages the weight matrices in the generator to be orthogonal. This means that truncating the latent variable has a closer relationship to truncating the output variance and improves sample quality.

Other tricks include only using the gradients from the top K most realistic images (Sinha et al., 2020), label smoothing in the discriminator (Salimans et al., 2016), updating the discriminator using a history of generated images rather than the ones produced by the latest generator to avoid model "oscillation" (Salimans et al., 2016), and adding noise to the discriminator input (Arjovsky & Bottou, 2017). Kurach et al. (2019) present an overview of normalization and regularization in GANs. Chintala et al. (2020) provide further suggestions for training GANs.

Sample diversity: The original GAN paper (Goodfellow et al., 2014) argued that given enough capacity, training samples, and computation time, a GAN can learn to minimize the Jensen-Shannon divergence between the generated samples and the true distribution. However, subsequent work has cast doubt on whether this happens in practice. Arora et al. (2017) suggest that the finite capacity of the discriminator means that the GAN training objective can approach its optimum value even when the variation in the output distribution is limited. Wu et al. (2017) approximated the log-likelihoods of the distributions produced by GANs using annealed importance sampling and found a mismatch between the generated and real distributions. Arora & Zhang (2017) ask human observers to identify GAN samples that are (near-)duplicates and infer the diversity of images from the frequency of these duplicates. They found that for DCGAN, a duplicate occurs with probability >50% with 400 samples; this implies that the support size was $\sim 400,000$, which is smaller than the training set. They also showed that the diversity increased as a function of the discriminator size. Bau et al. (2019) take a different approach and investigate the parts of the data space that GANs *cannot* generate.

Increasing diversity and preventing mode collapse: The extreme case of lack of diversity is *mode collapse*, in which the network repeatedly produces the same image (Salimans et al., 2016). This is a particular problem for conditional GANs, where the latent variable is sometimes completely ignored, and the output depends only on the conditional information. Mao et al. (2019) introduce a regularization term to help prevent mode collapse in conditional GANs, which maximizes the ratio of the distance between generated images with respect to the corresponding latent variables and hence encourages diversity in the outputs. Other work that aims to reduce mode collapse includes VEEGAN (Srivastava et al., 2017), which introduces a reconstruction network that maps the generated image back to the original noise and hence discourages many-to-one mappings from noise to images.

Salimans et al. (2016) suggested computing statistics across the mini-batch and using the discriminator to ensure that these are indistinguishable from the statistics of batches of real images. This is known as *mini-batch discrimination* and is implemented by adding a layer toward the end of the discriminator that learns a tensor for each image that captures the statistics of the batch. This was simplified by Karras et al. (2018), who computed a standard deviation for each feature in each spatial location over the mini-batch. Then they average over spatial locations and features to get a single estimate. This is replicated to get a single feature map, which is appended to a layer near the end of the discriminator network. Lin et al. (2018) pass concatenated (real or generated) samples to the discriminator and provide a theoretical analysis of how presenting multiple samples to the discriminator increases diversity. MAD-GAN (Ghosh et al., 2018) increases the diversity of GAN samples by using multiple generators and requiring the single discriminator to identify which generator created the samples, thus providing a signal to help push the generators to create different samples from one another.

Multiple scales: Wang et al. (2018b) used multiple discriminators at different scales to help ensure that image quality is high in all frequency bands. Other work defined both generators and discriminators at different resolutions (Denton et al., 2015; Zhang et al., 2017d; Huang et al., 2017c). Karras et al. (2018) introduced the progressive growing method (figure 15.9), which is somewhat simpler and faster to train.

StyleGAN: Karras et al. (2019) introduced the StyleGAN framework (section 15.6). In subsequent work (Karras et al., 2020b), they improved the quality of generated images by (i) redesigning the normalization layers in the generator to remove "water droplet" artifacts and (ii) reducing artifacts where fine details do not follow the coarse details by changing the progressive growing framework. Further improvements include developing methods to train GANs with limited data (Karras et al., 2020a) and fixing aliasing artifacts (Karras et al., 2021). A large body of work finds and manipulates the latent variables in the StyleGAN to edit images (e.g., Abdal et al., 2021; Collins et al., 2020; Härkönen et al., 2020; Patashnik et al., 2021; Shen et al., 2020b; Tewari et al., 2020; Wu et al., 2021; Roich et al., 2022).

Conditional GANs: The conditional GAN was developed by Mirza & Osindero (2014), the auxiliary classifier GAN by Odena et al. (2017), and the InfoGAN by Chen et al. (2016b). The discriminators of these models usually append the conditional information to the discriminator input (Mirza & Osindero, 2014; Denton et al., 2015; Saito et al., 2017) or to an intermediate hidden layer in the discriminator (Reed et al., 2016a; Zhang et al., 2017d; Perarnau et al., 2016). However, Miyato & Koyama (2018) experimented with taking the inner product between embedded conditional information with a layer of the discriminator, motivated by the role of the class information in the underlying probabilistic model. Images generated by GANs have variously been conditioned on classes (e.g., Odena et al., 2017), input text (Reed et al., 2016a; Zhang et al., 2017d), attributes (Yan et al., 2016; Donahue et al., 2018a; Xiao et al., 2018b), bounding boxes and keypoints (Reed et al., 2016b), and images (e.g., Isola et al., 2017)).

Image translation: Isola et al. (2017) developed the Pix2Pix algorithm (figure 15.16), and a similar system with higher-resolution results was subsequently developed by Wang et al. (2018b). StarGAN (Choi et al., 2018) performs image-to-image translation across multiple domains using only a single model. The idea of cycle consistency loss was introduced by Zhou et al. (2016b) in DiscoGAN and Zhu et al. (2017) in CycleGAN (figure 15.18).

Adversarial loss: In many image translation tasks, there is no "generator"; such models can be considered supervised learning tasks with an adversarial loss that encourages realism. The super-resolution algorithm of Ledig et al. (2017) is a good example of this (figure 15.17). Esser et al. (2021) used an autoencoder with an adversarial loss. This network takes an image, reduces the representation size to create a "bottleneck," and then reconstructs the image from this reduced data space. In practice, the architecture is similar to encoder-decoder networks (e.g., figure 10.19). After training, the autoencoder reproduces something that is both close to the image and looks highly realistic. They vector quantize (discretize) the bottleneck of the autoencoder and then learn a probability distribution over the discrete variables using a transformer decoder. By sampling from this transformer decoder, they can produce extremely large high-quality images.

Inverting GANs: One way to edit real images is to project them to the latent space, manipulate the latent variable, and then re-project them to image space. This process is known as *resynthesis*. Unfortunately, GANs only map from the latent variable to the observed data, not vice versa. This has led to methods to *invert* GANs (i.e., find the latent variable that corresponds as closely as possible to an observed image). These methods fall into two classes. The first learns a network that maps in the opposite direction (Donahue et al., 2018b; Luo et al., 2017a; Perarnau et al., 2016; Dumoulin et al., 2017; Guan et al., 2020). This is known as an *encoder*. The second approach is to start with some latent variable $\mathbf{z}$ and optimize it until it reconstructs the image as closely as possible (Creswell & Bharath, 2018; Karras et al., 2020b; Abdal et al., 2019; Lipton & Tripathi, 2017). Zhu et al. (2020a) combine both approaches.

There has been particular interest in inversion for StyleGAN because it produces excellent results and can control the image at different scales. Unfortunately, Abdal et al. (2020) showed that it is not possible to invert StyleGAN without artifacts and proposed inverting to an extended style space, and Richardson et al. (2021) trained an encoder that reliably maps to this space. Even after inverting to the extended space, editing images that are out of domain may still not work well. Roich et al. (2022) address this issue by fine-tuning the generator of StyleGAN so that it reconstructs the image exactly and show that the result can be edited well. They also add extra terms that reconstruct nearby points exactly so that the modification is local. This technique is known as *pivotal tuning*. A survey of GAN inversion techniques can be found in Xia et al. (2022).

Editing images with GANs: The iGAN (Zhu et al., 2016) allows users to make interactive edits by scribbling or warping parts of an existing image. The tool then adjusts the output

image to be both realistic and to fit these new constraints. It does this by finding a latent vector that produces an image that is similar to the edited image and obeys the edge map of any added lines. It is typical also to add a mask so that only parts of the image close to the edits are changed. EditGAN (Ling et al., 2021) jointly models images and their semantic segmentation masks and allows edits to that mask.

Problems

Problem 15.1 What will the loss be in equation 15.8 when $q(\mathbf{x}) = Pr(\mathbf{x})$?

Problem 15.2* Write an equation relating the loss L in equation 15.8 to the Jensen-Shannon distance $D_{JS}\left[q(\mathbf{x}) \,||\, Pr(\mathbf{x})\right]$ in equation 15.9.

Problem 15.3 Consider computing the earth mover's distance using linear programming in the primal form. The discrete distributions $Pr(x{=}i)$ and $q(x{=}j)$ are defined on $x = 1, 2, 3, 4$ and:

$$\mathbf{b} = \begin{bmatrix} Pr(x{=}1), Pr(x{=}2), Pr(x{=}3), Pr(x{=}4), q(x{=}1), q(x{=}2), q(x{=}3), q(x{=}4) \end{bmatrix}^T. \quad (15.18)$$

Write out the contents of the $8{\times}16$ matrix $\mathbf{A}$. You may assume that the contents of $\mathbf{P}$ have been vectorized into $\mathbf{p}$ column-first.

Problem 15.4* Calculate (i) the KL divergence, (ii) the reverse KL divergence,(iii) the Jensen-Shannon divergence, and (iv) the Wasserstein distance between the distributions:

$$Pr(z) = \begin{cases} 0 & z < 0 \\ 1 & 0 \leq z \leq 1 \\ 0 & z > 1 \end{cases}, \quad \text{and} \quad Pr(z) = \begin{cases} 0 & z < a \\ 1 & a \leq z \leq a+1 \\ 0 & z > a \end{cases}. \quad (15.19)$$

for the range $a \in [-3, 3]$. To get a formula for the Wasserstein distance for this special case, consider the total "earth" (i.e., probability mass) that must be moved and multiply this by the squared distance it must move.

Problem 15.5 The KL distance and Wasserstein distances between univariate Gaussian distributions are given by:

$$D_{kl} = \log\left[\frac{\sigma_2}{\sigma_1}\right] + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}, \quad (15.20)$$

and

$$D_w = (\mu_1 - \mu_2)^2 + \sigma_1 + \sigma_2 - 2\sqrt{\sigma_1\sigma_2}, \quad (15.21)$$

respectively. Plot these distances as a function of $\mu_1 - \mu_2$ for the case when $\sigma_1 = \sigma_2 = 1$.

Problem 15.6 Consider a latent variable $\mathbf{z}$ with dimension 100. Consider truncating the values of this variable to (i) $\tau = 2.0$, (ii) $\tau = 1.0$, (iii) $\tau = 0.5$, (iv) $\tau = 0.04$ standard deviations. What proportion of the original probability distribution is disregarded in each case?

Chapter 16

Normalizing flows

Chapter 15 introduced generative adversarial networks (GANs). These are generative models that pass a latent variable through a deep network to create a new sample. GANs are trained using the principle that the samples should be indistinguishable from real data. However, they don't define a distribution over data examples. Hence, assessing the probability that a new example belongs to the same dataset isn't straightforward.

In this chapter, we describe *normalizing flows*. These learn a probability model by transforming a simple distribution into a more complicated one using a deep network. Normalizing flows can both sample from this distribution and evaluate the probability of new examples. However, they require specialized architecture: each layer must be *invertible*. In other words, it must be able to transform data in both directions.

16.1 1D example

Normalizing flows are probabilistic generative models: they fit a probability distribution to training data (figure 14.2b). Consider modeling a 1D distribution $Pr(x)$. Normalizing flows start with a simple tractable *base* distribution $Pr(z)$ over a latent variable z and apply a function $x = \mathrm{f}[z, \boldsymbol{\phi}]$, where the parameters $\boldsymbol{\phi}$ are chosen so that $Pr(x)$ has the desired distribution (figure 16.1). Generating a new example x^* is easy; we draw z^* from the base density and pass this through the function so that $x^* = \mathrm{f}[z^*, \boldsymbol{\phi}]$.

16.1.1 Measuring probability

Measuring the probability of a data point x is more challenging. Consider applying a function $\mathrm{f}[z, \boldsymbol{\phi}]$ to random variable z with known density $Pr(z)$. The probability density will decrease in areas that are stretched by the function and increase in areas that are compressed so that the area under the new distribution remains one. The degree to which a function $\mathrm{f}[z, \boldsymbol{\phi}]$ stretches or compresses its input depends on the magnitude of its gradient. If a small change to the input causes a larger change in the output, it

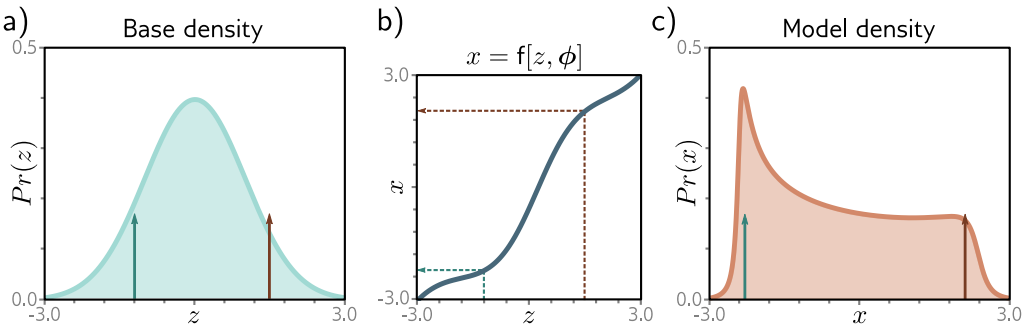

Figure 16.1 Transforming probability distributions. a) The base density is a standard normal defined on a latent variable z. b) This variable is transformed by a function $x = \mathrm{f}[z, \boldsymbol{\phi}]$ to a new variable x, which c) has a new distribution. To sample from this model, we draw values z from the base density (green and brown arrows in panel (a) show two examples). We pass these through the function $\mathrm{f}[z, \boldsymbol{\phi}]$ as shown by dotted arrows in panel (b) to generate the values of x, which are indicated as arrows in panel (c).

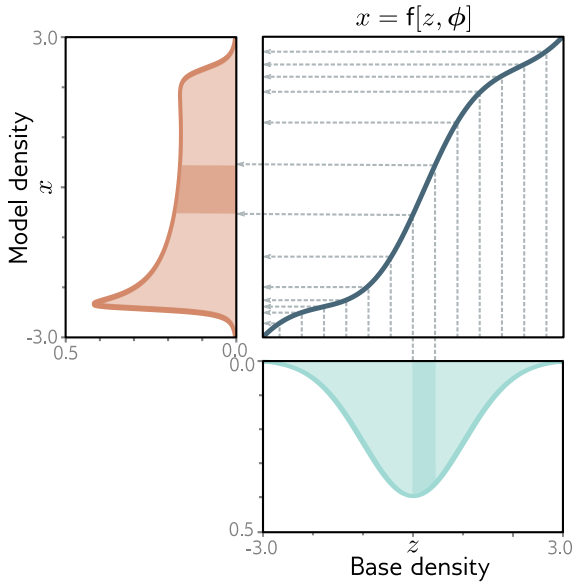

Figure 16.2 Transforming distributions. The base density (cyan, bottom) passes through a function (blue curve, top right) to create the model density (orange, left). Consider dividing the base density into equal intervals (gray vertical lines). The probability mass between adjacent lines must remain the same after transformation. The cyan-shaded region passes through a part of the function where the gradient is larger than one, so this region is stretched. Consequently, the height of the orange-shaded region must be lower so that it retains the same area as the cyan-shaded region. In other places (e.g., $z = -2$), the gradient is less than one, and the model density increases relative to the base density.

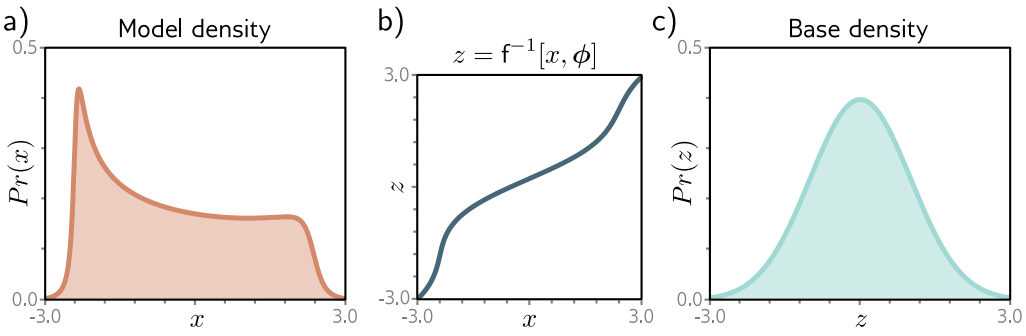

Figure 16.3 Inverse mapping (normalizing direction). If the function is invertible, then it's possible to transform the model density back to the original base density. The probability of a point x under the model density depends partly on the probability of the equivalent point z under the base density (see equation 16.1).

stretches the function. If a small change to the input causes a smaller change in the output, it compresses the function (figure 16.2).

More precisely, the probability of data x under the transformed distribution is:

$$Pr(x|\boldsymbol{\phi}) = \left|\frac{\partial \mathrm{f}[z, \boldsymbol{\phi}]}{\partial z}\right|^{-1} \cdot Pr(z), \tag{16.1}$$

where $z = \mathrm{f}^{-1}[x, \boldsymbol{\phi}]$ is the latent variable that created x. The term $Pr(z)$ is the original probability of this latent variable under the base density. This is moderated according to the magnitude of the derivative of the function. If this is greater than one, then the probability decreases. If it is smaller, the probability increases.

Notebook 16.1
1D normalizing flows

16.1.2 Forward and inverse mappings

To draw samples from the distribution, we need the forward mapping $x = \mathrm{f}[z, \boldsymbol{\phi}]$, but to measure the likelihood, we need to compute the inverse $z = \mathrm{f}^{-1}[x, \boldsymbol{\phi}]$. Hence, we need to choose $\mathrm{f}[z, \boldsymbol{\phi}]$ judiciously so that it is *invertible*.

Problems 16.1–16.2

The forward mapping is sometimes termed the *generative direction*. The base density is usually chosen to be a standard normal distribution. Hence, the inverse mapping is termed the *normalizing direction* since this takes the complex distribution over x and turns it into a normal distribution over z (figure 16.3).

16.1.3 Learning

To learn the distribution, we find parameters $\boldsymbol{\phi}$ that maximize the likelihood of the training data $\{x_i\}_{i=1}^I$ or equivalently minimize the negative log-likelihood:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(x_i|\boldsymbol{\phi})\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_{i=1}^{I} -\log\Big[Pr(x_i|\boldsymbol{\phi})\Big]\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_{i=1}^{I} \log\left[\left|\frac{\partial \mathrm{f}[z_i,\boldsymbol{\phi}]}{\partial z_i}\right|\right] - \log\big[Pr(z_i)\big]\right], \end{aligned} \tag{16.2}$$

where we have assumed that the data are independent and identically distributed in the first line and used the likelihood definition from equation 16.1 in the third line.

16.2 General case

The previous section developed a simple 1D example that modeled a probability distribution $Pr(x)$ by transforming a simpler base density $Pr(z)$. We now extend this to multivariate distributions $Pr(\mathbf{x})$ and $Pr(\mathbf{z})$ and add the complication that the transformation is defined by a deep neural network.

Consider applying a function $\mathbf{x} = \mathbf{f}[\mathbf{z}, \boldsymbol{\phi}]$ to a random variable $\mathbf{z} \in \mathbb{R}^D$ with base density $Pr(\mathbf{z})$, where $\mathbf{f}[\mathbf{z}, \boldsymbol{\phi}]$ is a deep network. The resulting variable $\mathbf{x} \in \mathbb{R}^D$ has a new distribution. A new sample $\mathbf{x}^*$ can be drawn from this distribution by (i) drawing a sample $\mathbf{z}^*$ from the base density and (ii) passing this through the neural network so that $\mathbf{x}^* = \mathbf{f}[\mathbf{z}^*, \boldsymbol{\phi}]$.

By analogy with equation 16.1, the likelihood of a sample under this distribution is:

$$Pr(\mathbf{x}|\boldsymbol{\phi}) = \left|\frac{\partial \mathbf{f}[\mathbf{z}, \boldsymbol{\phi}]}{\partial \mathbf{z}}\right|^{-1} \cdot Pr(\mathbf{z}), \tag{16.3}$$

where $\mathbf{z} = \mathbf{f}^{-1}[\mathbf{x}, \boldsymbol{\phi}]$ is the latent variable $\mathbf{z}$ that created $\mathbf{x}$. The first term is the inverse of the determinant of the $D \times D$ Jacobian matrix $\partial \mathbf{f}[\mathbf{z}, \boldsymbol{\phi}]/\partial \mathbf{z}$, which contains elements $\partial \mathrm{f}_i[\mathbf{z}, \boldsymbol{\phi}]/\partial z_j$ at position (i, j). Just as the absolute derivative measured the change of area at a point on a 1D function when the function was applied, the absolute determinant measures the change in volume at a point in the multivariate function. The second term is the probability of the latent variable under the base density.

Appendix B.3.8 Determinant

Appendix B.5 Jacobian

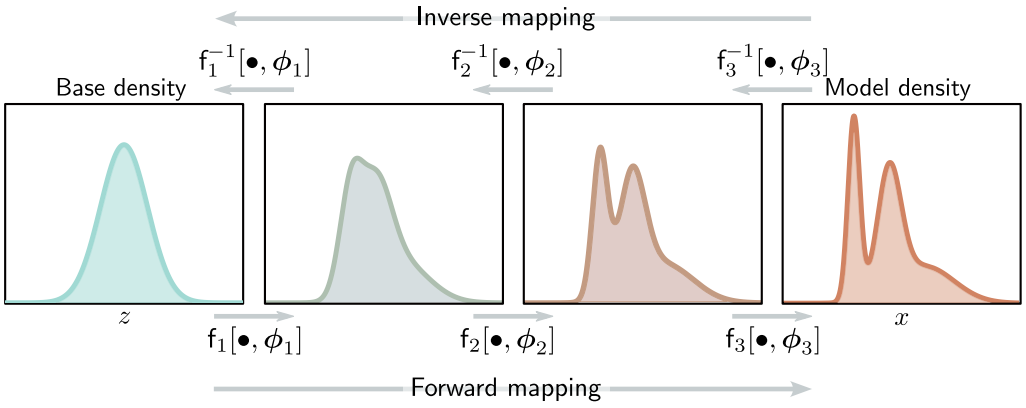

Figure 16.4 Forward and inverse mappings for a deep neural network. The base density (left) is gradually transformed by the network layers $f_1[\bullet, \phi_1], f_2[\bullet, \phi_2], \ldots$ to create the model density. Each layer is invertible, and we can equivalently think of the inverse of the layers as gradually transforming (or "flowing") the model density back to the base density.

16.2.1 Forward mapping with a deep neural network

In practice, the forward mapping $\mathbf{f}[\mathbf{z}, \boldsymbol{\phi}]$ is usually defined by a neural network, consisting of a series of layers $\mathbf{f}_k[\bullet, \boldsymbol{\phi}_k]$ with parameters $\boldsymbol{\phi}_k$, which are composed together as:

$$\mathbf{x} = \mathbf{f}[\mathbf{z}, \boldsymbol{\phi}] = \mathbf{f}_K\Big[\mathbf{f}_{K-1}\Big[\ldots \mathbf{f}_2\big[\mathbf{f}_1[\mathbf{z}, \boldsymbol{\phi}_1], \boldsymbol{\phi}_2\big], \ldots \boldsymbol{\phi}_{K-1}\Big], \boldsymbol{\phi}_K\Big]. \tag{16.4}$$

The inverse mapping (normalizing direction) is defined by the composition of the inverse of each layer $\mathbf{f}_k^{-1}[\bullet, \boldsymbol{\phi}_k]$ applied in the opposite order:

$$\mathbf{z} = \mathbf{f}^{-1}[\mathbf{x}, \boldsymbol{\phi}] = \mathbf{f}_1^{-1}\Big[\mathbf{f}_2^{-1}\Big[\ldots \mathbf{f}_{K-1}^{-1}\big[\mathbf{f}_K^{-1}[\mathbf{x}, \boldsymbol{\phi}_K], \boldsymbol{\phi}_{K-1}\big], \ldots \boldsymbol{\phi}_2\Big], \boldsymbol{\phi}_1\Big]. \tag{16.5}$$

The base density $Pr(\mathbf{z})$ is usually defined as a multivariate standard normal (i.e., with mean zero and identity covariance). Hence, the effect of each subsequent inverse layer is to gradually move or "flow" the data density toward this normal distribution (figure 16.4). This gives rise to the name "normalizing flows."

The Jacobian of the forward mapping can be expressed as:

$$\frac{\partial \mathbf{f}[\mathbf{z}, \boldsymbol{\phi}]}{\partial \mathbf{z}} = \frac{\partial \mathbf{f}_K[\mathbf{f}_{K-1}, \boldsymbol{\phi}_K]}{\partial \mathbf{f}_{K-1}} \cdot \frac{\partial \mathbf{f}_{K-1}[\mathbf{f}_{K-2}, \boldsymbol{\phi}_{K-1}]}{\partial \mathbf{f}_{K-2}} \cdots \frac{\partial \mathbf{f}_2[\mathbf{f}_1, \boldsymbol{\phi}_2]}{\partial \mathbf{f}_1} \cdot \frac{\partial \mathbf{f}_1[\mathbf{z}, \boldsymbol{\phi}_1]}{\partial \mathbf{z}}, \tag{16.6}$$

where we have overloaded the notation to make $\mathbf{f}_k$ the output of the function $\mathbf{f}_k[\bullet, \boldsymbol{\phi}_k]$. The absolute determinant of this Jacobian can be computed by taking the product of the individual absolute determinants:

$$\left|\frac{\partial \mathbf{f}[\mathbf{z},\boldsymbol{\phi}]}{\partial \mathbf{z}}\right| = \left|\frac{\partial \mathbf{f}_K[\mathbf{f}_{K-1},\boldsymbol{\phi}_K]}{\partial \mathbf{f}_{K-1}}\right| \cdot \left|\frac{\partial \mathbf{f}_{K-1}[\mathbf{f}_{K-2},\boldsymbol{\phi}_{K-1}]}{\partial \mathbf{f}_{K-2}}\right| \cdots \left|\frac{\partial \mathbf{f}_2[\mathbf{f}_1,\boldsymbol{\phi}_2]}{\partial \mathbf{f}_1}\right| \cdot \left|\frac{\partial \mathbf{f}_1[\mathbf{z},\boldsymbol{\phi}_1]}{\partial \mathbf{z}}\right|. \tag{16.7}$$

Problem 16.3

The absolute determinant of the Jacobian of the inverse mapping is found by applying the same rule to equation 16.5. It is the reciprocal of the absolute determinant in the forward mapping.

We train normalizing flows with a dataset $\{\mathbf{x}_i\}$ of I training examples using the negative log-likelihood criterion:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}}\left[\prod_{i=1}^{I} Pr(\mathbf{z}_i)\cdot\left|\frac{\partial \mathbf{f}[\mathbf{z}_i,\boldsymbol{\phi}]}{\partial \mathbf{z}_i}\right|^{-1}\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[\sum_{i=1}^{I}\log\left[\left|\frac{\partial \mathbf{f}[\mathbf{z}_i,\boldsymbol{\phi}]}{\partial \mathbf{z}_i}\right|\right] - \log\big[Pr(\mathbf{z}_i)\big]\right],\end{aligned} \tag{16.8}$$

where $\mathbf{z}_i = \mathbf{f}^{-1}[\mathbf{x}_i,\boldsymbol{\phi}]$, $Pr(\mathbf{z}_i)$ is measured under the base distribution, and the absolute determinant $|\partial \mathbf{f}[\mathbf{z}_i,\boldsymbol{\phi}]/\partial \mathbf{z}_i|$ is given by equation 16.7.

16.2.2 Desiderata for network layers

The theory of normalizing flows is straightforward. However, for this to be practical, we need neural network layers $\mathbf{f}_k$ that have four properties.

1. Collectively, the set of network layers must be sufficiently *expressive* to map a multivariate standard normal distribution to an arbitrary density.
2. The network layers must be *invertible*; each must define a unique one-to-one mapping from any input point to an output point (a *bijection*). If multiple inputs were mapped to the same output, the inverse would be ambiguous.
3. It must be possible to compute the *inverse* of each layer *efficiently*. We need to do this every time we evaluate the likelihood. This happens repeatedly during training, so there must be a closed-form solution or a fast algorithm for the inverse.
4. It also must be possible to evaluate the *determinant* of the Jacobian *efficiently* for either the forward or inverse mapping.

Appendix B.1 Bijection

16.3 Invertible network layers

We now describe different invertible network layers or *flows* for use in these models. We start with linear and elementwise flows. These are easy to invert, and it's possible to compute the determinant of their Jacobians, but neither is sufficiently expressive to describe arbitrary transformations of the base density. However, they form the building blocks of coupling, autoregressive, and residual flows, which are all more expressive.

16.3.1 Linear flows

A linear flow has the form $\mathbf{f}[\mathbf{h}] = \boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{h}$. If the matrix $\boldsymbol{\Omega}$ is invertible, the linear transform is invertible. For $\boldsymbol{\Omega} \in \mathbb{R}^{D\times D}$, the computation of the inverse is $\mathcal{O}[D^3]$. The determinant of the Jacobian is just the determinant of $\boldsymbol{\Omega}$, which can also be computed in $\mathcal{O}[D^3]$. This means that linear flows become expensive as the dimension D increases.

Appendix A Big O notation

If the matrix $\boldsymbol{\Omega}$ takes a special form, then inversion and computation of the determinant can become more efficient, but the transformation becomes less general. For example, diagonal matrices require only $\mathcal{O}[D]$ computation for the inversion and determinant, but the elements of $\mathbf{h}$ don't interact. Orthogonal matrices are also more efficient to invert, and their determinant is fixed, but they do not allow scaling of the individual dimensions. Triangular matrices are more practical; they are invertible using a process known as back-substitution, which is $\mathcal{O}[D^2]$, and the determinant is just the product of the diagonal values.

Appendix B.4 Matrix types

Problem 16.4

One way to make a linear flow that is general, efficient to invert, and for which the Jacobian can be computed efficiently is to parameterize it directly in terms of the LU decomposition. In other words, we use:

$$\boldsymbol{\Omega} = \mathbf{PL}(\mathbf{U} + \mathbf{D}), \tag{16.9}$$

where $\mathbf{P}$ is a predetermined permutation matrix, $\mathbf{L}$ is a lower triangular matrix, $\mathbf{U}$ is an upper triangular matrix with zeros on the diagonal, and $\mathbf{D}$ is a diagonal matrix that supplies those missing diagonal elements. This can be inverted in $\mathcal{O}[D^2]$, and the log determinant is just the sum of the log of the absolute values on the diagonal of $\mathbf{D}$.

Unfortunately, linear flows are not sufficiently expressive. When a linear function $\mathbf{f}[\mathbf{h}] = \boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{h}$ is applied to normally distributed input $\text{Norm}_{\mathbf{h}}[\boldsymbol{\mu}, \boldsymbol{\Sigma}]$, then the result is also normally distributed with mean and variance, $\boldsymbol{\beta} + \boldsymbol{\Omega}\boldsymbol{\mu}$ and $\boldsymbol{\Omega}\boldsymbol{\Sigma}\boldsymbol{\Omega}^T$, respectively. Hence, it is not possible to map a normal distribution to an arbitrary density using linear flows alone.

Problems 16.5–16.6

16.3.2 Elementwise flows

Since linear flows are not sufficiently expressive, we must turn to nonlinear flows. The simplest of these are elementwise flows, which apply a pointwise nonlinear function $\text{f}[\bullet, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$ to each element of the input so that:

$$\mathbf{f}[\mathbf{h}] = \Big[\text{f}[h_1, \boldsymbol{\phi}], \text{f}[h_2, \boldsymbol{\phi}], \ldots \text{f}[h_D, \boldsymbol{\phi}]\Big]^T. \tag{16.10}$$

The Jacobian $\partial\mathbf{f}[\mathbf{h}]/\partial\mathbf{h}$ is diagonal since the d^{th} input to $\mathbf{f}[\mathbf{h}]$ only affects the d^{th} output. Its determinant is the product of the entries on the diagonal, so:

$$\left|\frac{\partial\mathbf{f}[\mathbf{h}]}{\partial\mathbf{h}}\right| = \prod_{d=1}^{D}\left|\frac{\partial\text{f}[h_d]}{\partial h_d}\right|. \tag{16.11}$$

The function $\text{f}[\bullet, \boldsymbol{\phi}]$ could be a fixed invertible nonlinearity like the leaky ReLU (figure 3.13), in which case there are no parameters, or it may be any parameterized

Problem 16.7

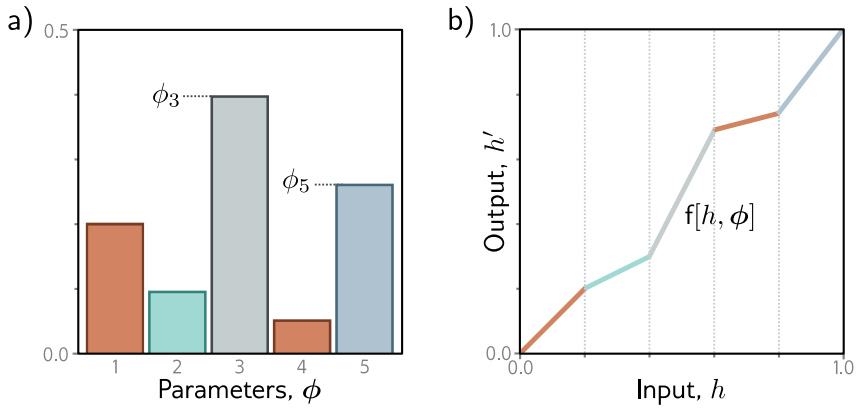

Figure 16.5 Piecewise linear mapping. An invertible piecewise linear mapping $h' = \text{f}[h, \boldsymbol{\phi}]$ can be created by dividing the input domain $h \in [0, 1]$ into K equally sized regions (here $K = 5$). Each region has a slope with parameter, $\boldsymbol{\phi}_k$. a) If these parameters are positive and sum to one, then b) the function will be invertible and map to the output domain $h' \in [0, 1]$.

invertible one-to-one mapping. A simple example is a piecewise linear function with K regions (figure 16.5) which maps $[0, 1]$ to $[0, 1]$ as:

$$\text{f}[h, \boldsymbol{\phi}] = \left(\sum_{k=1}^{b-1} \phi_k\right) + (hK - b)\phi_b, \tag{16.12}$$

where the parameters $\phi_1, \phi_2, \ldots, \phi_K$ are positive and sum to 1, and $b = \lfloor Kh \rfloor$ is the index of the bin that contains h. The first term is the sum of all the preceding bins, and the second term represents the proportion of the way through the current bin that h lies. This function is easy to invert, and its gradient can be calculated almost everywhere. There are many similar schemes for creating smooth functions, often using splines with parameters that ensure the function is monotonic and hence invertible.

Problems 16.8–16.9

Elementwise flows are nonlinear but don't mix input dimensions, so they can't create correlations between variables. When alternated with linear flows (which do mix dimensions), more complex transformations can be modeled. However, in practice, elementwise flows are used as components of more complex layers like *coupling flows*.

16.3.3 Coupling flows

Coupling flows divide the input $\mathbf{h}$ into two parts so that $\mathbf{h} = [\mathbf{h}_1^T, \mathbf{h}_2^T]^T$ and define the flow $\mathbf{f}[\mathbf{h}, \boldsymbol{\phi}]$ as:

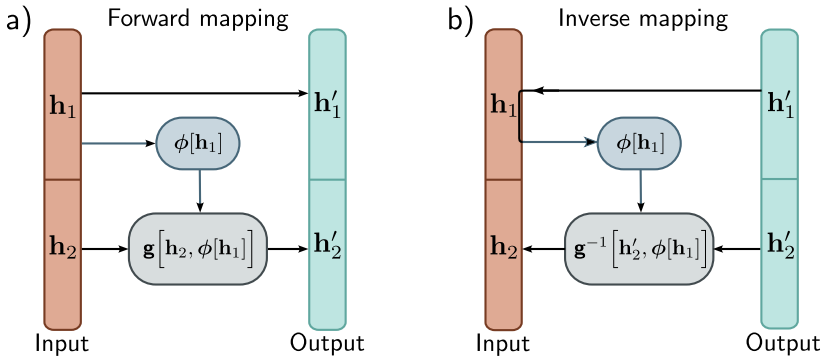

Figure 16.6 Coupling flows. a) The input (orange vector) is divided into $\mathbf{h}_1$ and $\mathbf{h}_2$. The first part $\mathbf{h}'_1$ of the output (cyan vector) is a copy of $\mathbf{h}_1$. The output $\mathbf{h}'_2$ is created by applying an invertible transformation $\mathbf{g}[\bullet, \boldsymbol{\phi}]$ to $\mathbf{h}_2$, where the parameters $\boldsymbol{\phi}$ are themselves a (not necessarily invertible) function of $\mathbf{h}_1$. b) In the inverse mapping, $\mathbf{h}_1 = \mathbf{h}'_1$. This allows us to calculate the parameters $\boldsymbol{\phi}[\mathbf{h}_1]$ and then apply the inverse $\mathbf{g}^{-1}[\mathbf{h}'_2, \boldsymbol{\phi}]$ to retrieve $\mathbf{h}_2$.

$$\begin{aligned}\mathbf{h}'_1 &= \mathbf{h}_1 \\ \mathbf{h}'_2 &= \mathbf{g}\Big[\mathbf{h}_2, \boldsymbol{\phi}[\mathbf{h}_1]\Big].\end{aligned} \tag{16.13}$$

Here $\mathbf{g}[\bullet, \boldsymbol{\phi}]$ is an elementwise flow (or other invertible layer) with parameters $\boldsymbol{\phi}[\mathbf{h}_1]$ that are themselves a nonlinear function of the inputs $\mathbf{h}_1$ (figure 16.6). The function $\boldsymbol{\phi}[\bullet]$ is usually a neural network of some kind and does not have to be invertible. The original variables can be recovered as:

$$\begin{aligned}\mathbf{h}_1 &= \mathbf{h}'_1 \\ \mathbf{h}_2 &= \mathbf{g}^{-1}\Big[\mathbf{h}'_2, \boldsymbol{\phi}[\mathbf{h}_1]\Big].\end{aligned} \tag{16.14}$$

If the function $\mathbf{g}[\bullet, \boldsymbol{\phi}]$ is an elementwise flow, the Jacobian will be diagonal with the identity matrix in the top-left quadrant and the derivatives of the elementwise transformation in the bottom right. Its determinant is the product of these diagonal values.

The inverse and Jacobian can be computed efficiently, but this approach only transforms the second half of the parameters in a way that depends on the first half. To make a more general transformation, the elements of $\mathbf{h}$ are randomly shuffled using permutation matrices between layers, so every variable is ultimately transformed by every other. In practice, these permutation matrices are difficult to learn. Hence, they are initialized randomly and then frozen. For structured data like images, the channels are divided into two halves $\mathbf{h}_1$ and $\mathbf{h}_2$ and permuted between layers using 1×1 convolutions.

Appendix B.4.4 Permutation matrix

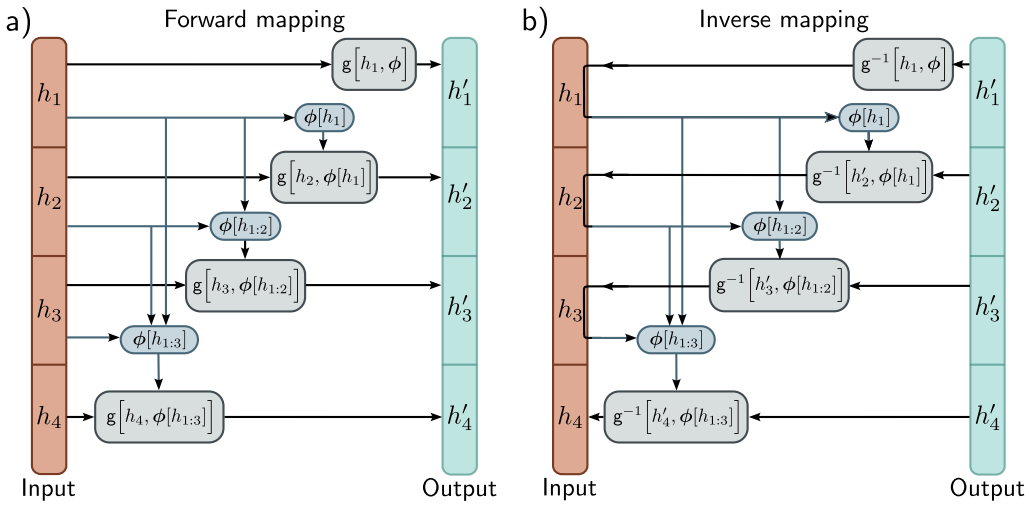

Figure 16.7 Autoregressive flows. The input $\mathbf{h}$ (orange column) and output $\mathbf{h}'$ (cyan column) are split into their constituent dimensions (here four dimensions). a) Output h_1' is an invertible transformation of input h_1. Output h_2' is an invertible function of input h_2 where the parameters depend on h_1. Output h_3' is an invertible function of input h_3 where the parameters depend on previous inputs h_1 and h_2, and so on. None of the outputs depend on one another, so they can be computed in parallel. b) The inverse of the autoregressive flow is computed using a similar method as for coupling flows. However, notice that to compute h_2 we must already know h_1, to compute h_3, we must already know h_1 and h_2, and so on. Consequently, the inverse cannot be computed in parallel.

16.3.4 Autoregressive flows

Autoregressive flows are a generalization of coupling flows that treat each input dimension as a separate "block" (figure 16.7). They compute the d^{th} dimension of the output $\mathbf{h}'$ based on the first $d-1$ dimensions of the input $\mathbf{h}$:

$$h_d' = \mathrm{g}\Big[h_d, \boldsymbol{\phi}[\mathbf{h}_{1:d-1}]\Big]. \tag{16.15}$$

The function $\mathrm{g}[\bullet, \bullet]$ is termed the *transformer*,[1] and the parameters $\boldsymbol{\phi}$, $\boldsymbol{\phi}[h_1]$, $\boldsymbol{\phi}[h_1, h_2], \ldots$ are termed *conditioners*. As for coupling flows, the transformer $\mathrm{g}[\bullet, \boldsymbol{\phi}]$ must be invertible, but the conditioners $\boldsymbol{\phi}[\bullet]$ can take any form and are usually neural networks. If the transformer and conditioner are sufficiently flexible, autoregressive flows are *universal approximators* in that they can represent any probability distribution.

It's possible to compute all of the entries of the output $\mathbf{h}'$ in parallel using a network with appropriate masks so that the parameters $\boldsymbol{\phi}$ at position d only depend on previous

[1] This is nothing to do with the transformer layers discussed in chapter 12.

positions. This is known as a *masked autoregressive flow*. The principle is very similar to masked self-attention (section 12.7.2); connections that relate inputs to previous outputs are pruned.

Inverting the transformation is less efficient. Consider the forward mapping:

$$\begin{aligned} h'_1 &= \mathrm{g}\Big[h_1, \boldsymbol{\phi}\Big] \\ h'_2 &= \mathrm{g}\Big[h_2, \boldsymbol{\phi}[h_1]\Big] \\ h'_3 &= \mathrm{g}\Big[h_3, \boldsymbol{\phi}[h_{1:2}]\Big] \\ h'_4 &= \mathrm{g}\Big[h_4, \boldsymbol{\phi}[h_{1:3}]\Big]. \end{aligned} \tag{16.16}$$

This must be inverted sequentially using a similar principle as for coupling flows:

$$\begin{aligned} h_1 &= \mathrm{g}^{-1}\Big[h'_1, \boldsymbol{\phi}\Big] \\ h_2 &= \mathrm{g}^{-1}\Big[h'_2, \boldsymbol{\phi}[h_1]\Big] \\ h_3 &= \mathrm{g}^{-1}\Big[h'_3, \boldsymbol{\phi}[h_{1:2}]\Big] \\ h_4 &= \mathrm{g}^{-1}\Big[h'_4, \boldsymbol{\phi}[h_{1:3}]\Big]. \end{aligned} \tag{16.17}$$

This can't be done in parallel as the computation for h_d depends on $h_{1:d-1}$ (i.e., the partial results so far). Hence, inversion is time-consuming when the input is large.

Notebook 16.2
Autoregressive flows

16.3.5 Inverse autoregressive flows

Masked autoregressive flows are defined in the normalizing (inverse) direction. This is required to evaluate the likelihood efficiently and hence to learn the model. However, sampling requires the forward direction, in which each variable must be computed sequentially at each layer, which is slow. If we use an autoregressive flow for the forward (generative) transformation, then sampling is efficient, but computing the likelihood (and training) is slow. This is known as an *inverse autoregressive flow*.

A trick that allows fast learning and also fast (but approximate) sampling is to build a masked autoregressive flow to learn the distribution (the teacher) and then use this to train an inverse autoregressive flow from which we can sample efficiently (the student). This requires a different formulation of normalizing flows that learns from another function rather than a set of samples (see section 16.5.3).

16.3.6 Residual flows: iRevNet

Residual flows take their inspiration from residual networks. They divide the input into two parts $\mathbf{h} = [\mathbf{h}_1^T, \mathbf{h}_2^T]^T$ (as for coupling flows) and define the outputs as:

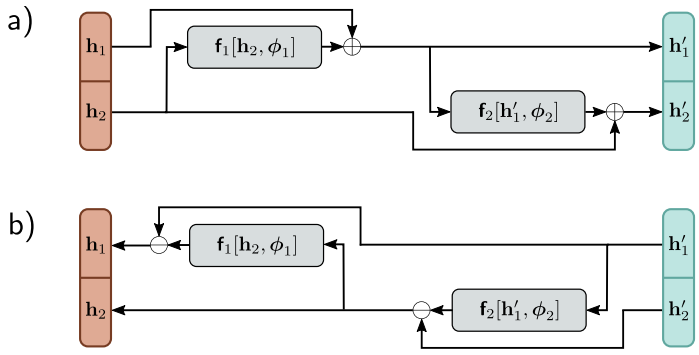

Figure 16.8 Residual flows. a) An invertible function is computed by splitting the input into $\mathbf{h}_1$ and $\mathbf{h}_2$ and creating two residual layers. In the first, $\mathbf{h}_2$ is processed and $\mathbf{h}_1$ is added. In the second, the result is processed, and $\mathbf{h}_2$ is added. b) In the reverse mechanism the functions are computed in the opposite order, and the addition operation becomes subtraction.

$$\begin{aligned}\mathbf{h}'_1 &= \mathbf{h}_1 + \mathbf{f}_1[\mathbf{h}_2, \boldsymbol{\phi}_1] \\ \mathbf{h}'_2 &= \mathbf{h}_2 + \mathbf{f}_2[\mathbf{h}'_1, \boldsymbol{\phi}_2],\end{aligned} \tag{16.18}$$

where $\mathbf{f}_1[\bullet, \boldsymbol{\phi}_1]$ and $\mathbf{f}_2[\bullet, \boldsymbol{\phi}_2]$ are two functions that do not necessarily have to be invertible (figure 16.8). The inverse can be computed by reversing the order of computation:

$$\begin{aligned}\mathbf{h}_2 &= \mathbf{h}'_2 - \mathbf{f}_2[\mathbf{h}'_1, \boldsymbol{\phi}_2] \\ \mathbf{h}_1 &= \mathbf{h}'_1 - \mathbf{f}_1[\mathbf{h}_2, \boldsymbol{\phi}_1].\end{aligned} \tag{16.19}$$

As for coupling flows, the division into blocks restricts the family of transformations that can be represented. Hence, the inputs are permuted between layers so that the variables can mix in arbitrary ways.

This formulation can be inverted easily, but for general functions $\mathbf{f}_1[\bullet, \boldsymbol{\phi}_1]$ and $\mathbf{f}_2[\bullet, \boldsymbol{\phi}_2]$, there is no efficient way to compute the Jacobian. This formulation is sometimes used to save memory when training residual networks; because the network is invertible, storing the activations at each layer in the forward pass is unnecessary.

Problem 16.10

16.3.7 Residual flows and contraction mappings: iResNet

A different approach to exploiting residual networks is to utilize the *Banach fixed point theorem* or *contraction mapping theorem*, which states that every contraction mapping has a fixed point. A contraction mapping f[•] has the property that:

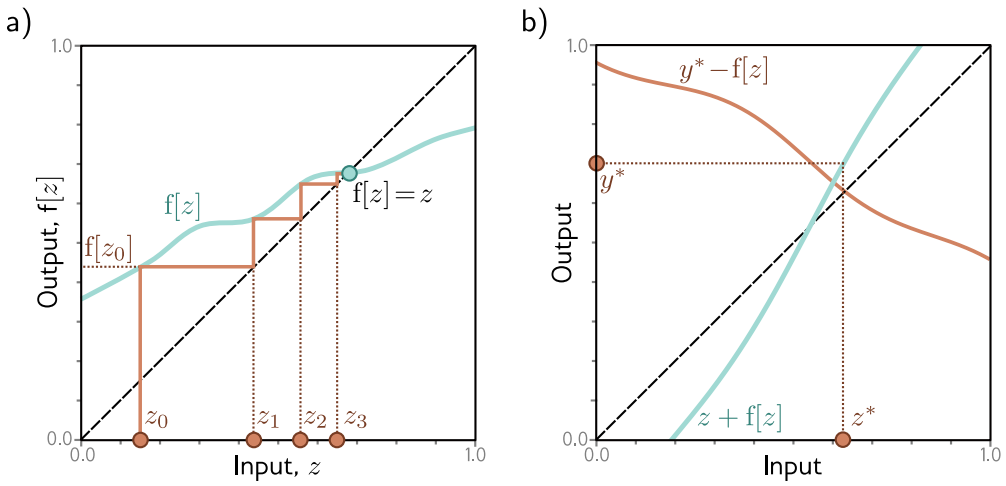

Figure 16.9 Contraction mappings. If a function has an absolute slope of less than one everywhere, iterating the function converges to a fixed point $\mathrm{f}[z] = z$. a) Starting at z_0, we evaluate $z_1 = \mathrm{f}[z_0]$. We then pass z_1 back into the function and iterate. Eventually, the process converges to the point where $\mathrm{f}[z] = z$ (i.e., where the function crosses the dashed diagonal identity line). b) This can be used to invert equations of the form $y = z + \mathrm{f}[z]$ for a value y^* by noticing that the fixed point of $y^* - f[z]$ (where the orange line crosses the dashed identity line) is at the same position as where $y^* = z + \mathrm{f}[z]$.

$$\mathrm{dist}\Big[\mathrm{f}[z'], \mathrm{f}[z]\Big] < \beta \cdot \mathrm{dist}\Big[z', z\Big] \qquad \forall\, z, z', \tag{16.20}$$

where $\mathrm{dist}[\bullet, \bullet]$ is a distance function and $0 < \beta < 1$. When a function with this property is iterated (i.e., the output is repeatedly passed back in as an input), the result converges to a fixed point where $\mathrm{f}[z] = z$ (figure 16.9). To understand this, consider applying the function to both the fixed point and the current position; the fixed point remains static, but the distance between the two must become smaller, so the current position must get closer to the fixed point.

Notebook 16.3 Contraction mappings

This theorem can be exploited to invert an equation of the form:

$$y = z + \mathrm{f}[z] \tag{16.21}$$

if $\mathrm{f}[z]$ is a contraction mapping. In other words, it can be used to find the z^* that maps to a given value, y^*. This can be done by starting with any point z_0 and iterating $z_{k+1} = y^* - \mathrm{f}[z_k]$. This has a fixed point at $z + \mathrm{f}[z] = y^*$ (figure 16.9b).

The same principle can be used to invert residual network layers of the form $\mathbf{h}' = \mathbf{h} + \mathbf{f}[\mathbf{h}, \boldsymbol{\phi}]$ if we ensure that $\mathbf{f}[\mathbf{h}, \boldsymbol{\phi}]$ is a contraction mapping. In practice, this means that the Lipschitz constant must be less than one. Assuming that the slope of the activation functions is not greater than one, this is equivalent to ensuring the largest eigenvalue of

Appendix B.1.1 Lipschitz constant

Appendix B.3.7 Eigenvalues

each weight matrix $\boldsymbol{\Omega}$ must be less than one. A crude way to do this is to ensure that the absolute magnitudes of the weights $\boldsymbol{\Omega}$ are small by clipping them.

The Jacobian determinant cannot be computed easily, but its logarithm can be approximated using a series of tricks.

$$\begin{aligned}\log\left[\left|\mathbf{I}+\frac{\partial \mathbf{f}[\mathbf{h},\boldsymbol{\phi}]}{\partial \mathbf{h}}\right|\right] &= \text{trace}\left[\log\left[\mathbf{I}+\frac{\partial \mathbf{f}[\mathbf{h},\boldsymbol{\phi}]}{\partial \mathbf{h}}\right]\right] \\ &= \sum_{k=1}^{\infty}(-1)^{k-1}\text{trace}\left[\frac{\partial \mathbf{f}[\mathbf{h},\boldsymbol{\phi}]}{\partial \mathbf{h}}\right]^{k}, \end{aligned} \tag{16.22}$$

where we have used the identity $\log[|\mathbf{A}|] = \text{trace}[\log[\mathbf{A}]]$ in the first line and expanded this into a power series in the second line.

Appendix B.3.8 Trace

Even when we truncate this series, it's still computationally expensive to compute the trace of the constituent terms. Hence, we approximate this using *Hutchinson's trace estimator*. Consider a normal random variable $\boldsymbol{\epsilon}$ with mean $\mathbf{0}$ and variance $\mathbf{I}$. The trace of a matrix $\mathbf{A}$ can be estimated as:

$$\begin{aligned}\text{trace}[\mathbf{A}] &= \text{trace}\left[\mathbf{A}\mathbb{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{T}\right]\right] \\ &= \text{trace}\left[\mathbb{E}\left[\mathbf{A}\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{T}\right]\right] \\ &= \mathbb{E}\left[\text{trace}\left[\mathbf{A}\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{T}\right]\right] \\ &= \mathbb{E}\left[\text{trace}\left[\boldsymbol{\epsilon}^{T}\mathbf{A}\boldsymbol{\epsilon}\right]\right] \\ &= \mathbb{E}\left[\boldsymbol{\epsilon}^{T}\mathbf{A}\boldsymbol{\epsilon}\right], \end{aligned} \tag{16.23}$$

where the first line is true because $\mathbb{E}[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{T}] = \mathbf{I}$. The second line derives from the properties of the expectation operator. The third line comes from the linearity of the trace operator. The fourth line is due to the invariance of the trace to cyclic permutation. The final line is true because the argument in the fourth line is now a scalar. We estimate the trace by drawing samples $\boldsymbol{\epsilon}_i$ from $Pr(\boldsymbol{\epsilon})$:

$$\begin{aligned}\text{trace}[\mathbf{A}] &= \mathbb{E}\left[\boldsymbol{\epsilon}^{T}\mathbf{A}\boldsymbol{\epsilon}\right] \\ &\approx \frac{1}{I}\sum_{i=1}^{I}\boldsymbol{\epsilon}_i^{T}\mathbf{A}\boldsymbol{\epsilon}_i. \end{aligned} \tag{16.24}$$

In this way, we can approximate the trace of the powers of the Taylor expansion (equation 16.22) and evaluate the log probability.

16.4 Multi-scale flows

In normalizing flows, the latent space $\mathbf{z}$ must be the same size as the data space $\mathbf{x}$, but we know that natural datasets can often be described by fewer underlying variables. At

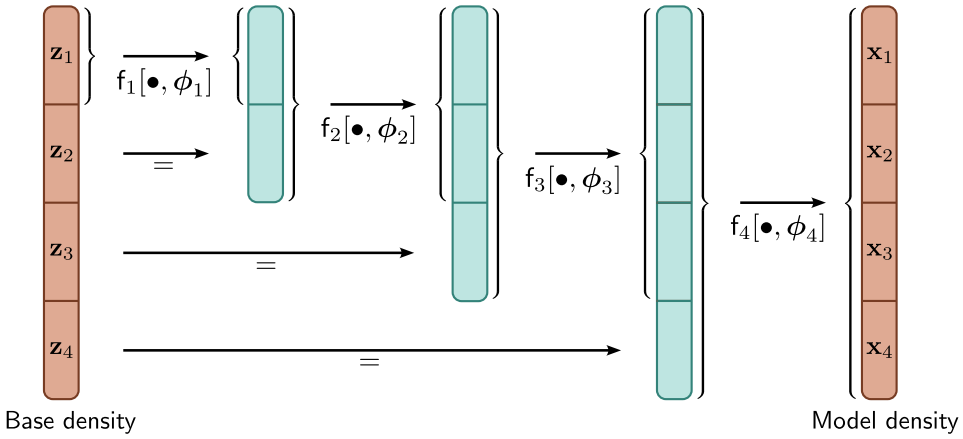

Figure 16.10 Multiscale flows. The latent space $\mathbf{z}$ must be the same size as the model density in normalizing flows. However, it can be partitioned into several components, which can be gradually introduced at different layers. This makes both density estimation and sampling faster. For the inverse process, the black arrows are reversed, and the last part of each block skips the remaining processing. For example, $\mathbf{f}_3^{-1}[\bullet, \phi_3]$ only operates on the first three blocks, and the fourth block becomes $\mathbf{z}_4$ and is assessed against the base density.

some point, we have to introduce all of these variables, but it is inefficient to pass them through the entire network. This leads to the idea of *multi-scale flows* (figure 16.10).

In the generative direction, multi-scale flows partition the latent vector into $\mathbf{z} = [\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_N]$. The first partition $\mathbf{z}_1$ is processed by a series of reversible layers with the same dimension as $\mathbf{z}_1$ until, at some point, $\mathbf{z}_2$ is appended and combined with the first partition. This continues until the network is the same size as the data $\mathbf{x}$. In the normalizing direction, the network starts at the full dimension of $\mathbf{x}$, but when it reaches the point where $\mathbf{z}_n$ was added, this is assessed against the base distribution.

16.5 Applications

We now describe three applications of normalizing flows. First, we consider modeling probability densities. Second, we consider the GLOW model for synthesizing images. Finally, we discuss using normalizing flows to approximate other distributions.

16.5.1 Modeling densities

Of the four generative models discussed in this book, normalizing flows is the only model that can compute the exact log-likelihood of a new sample. Generative adversarial

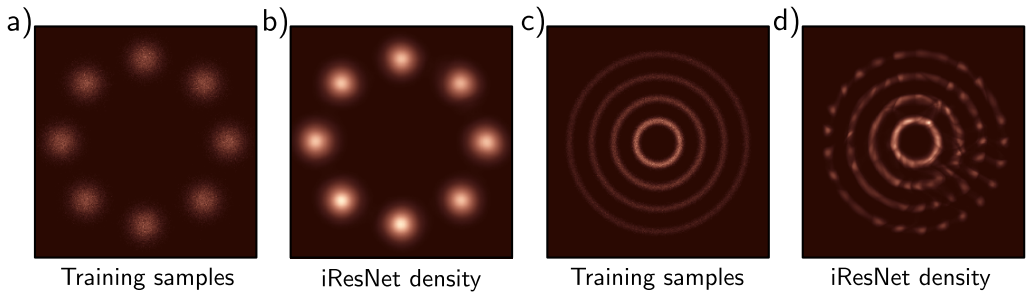

Figure 16.11 Modeling densities. a) Toy 2D data samples. b) Modeled density using iResNet. c–d) Second example. Adapted from Behrmann et al. (2019)

networks are not probabilistic, and both variational autoencoders and diffusion models can only return a lower bound on the likelihood.[2] Figure 16.11 depicts the estimated probability distributions in two toy problems using i-ResNet. One application of density estimation is anomaly detection; the data distribution of a clean dataset is described using a normalizing flow model. New examples with low probability are flagged as outliers. However, caution must be used as there may exist outliers with high probability that don't fall in the typical set (see figure 8.13).

16.5.2 Synthesis

Generative flows, or *GLOW*, is a normalizing flow model that can create high-fidelity images (figure 16.12) and uses many of the ideas from this chapter. It is easiest understood in the normalizing direction. GLOW starts with a $256 \times 256 \times 3$ tensor containing an RGB image. It uses coupling layers, in which the channels are partitioned into two halves. The second half is subject to a different affine transform at each spatial position, where the parameters of the affine transformation are computed by a 2D convolutional neural network run on the other half of the channels. The coupling layers are alternated with 1×1 convolutions, parameterized as LU decompositions which mix the channels.

Periodically, the resolution is halved by combining each 2×2 patch into one position with four times as many channels. GLOW is a multi-scale flow, and some of the channels are periodically removed to become part of the latent vector $\mathbf{z}$. Images are discrete (due to the quantization of RGB values), so noise is added to the inputs to prevent the training likelihood increasing without bound. This is known as *dequantization*.

To sample more realistic images, the GLOW model samples from the base density raised to a positive power. This chooses examples that are closer to the center of the density rather than from the tails. This is similar to the truncation trick in GANs

[2]The lower bound on the likelihood for diffusion models can actually exceed the exact computation in normalizing flows, but data generation is much slower (see chapter 18).

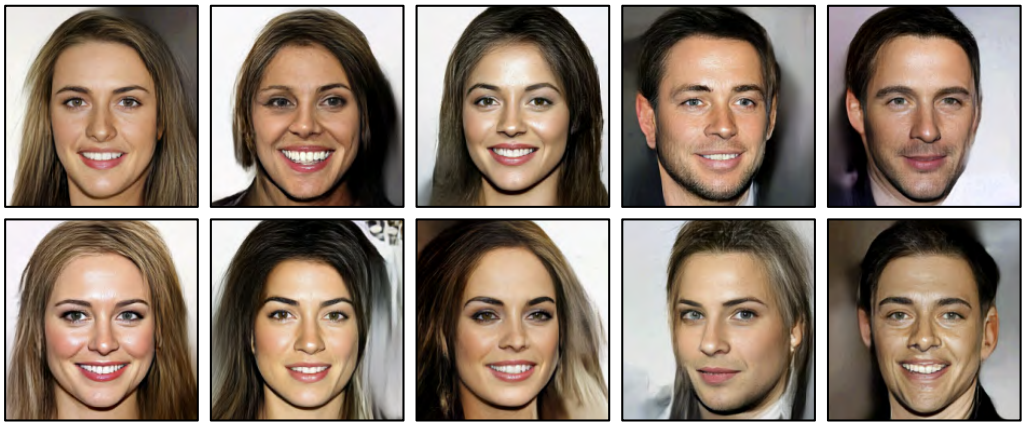

Figure 16.12 Samples from GLOW trained on the CelebA HQ dataset (Karras et al., 2018). The samples are of reasonable quality, although GANs and diffusion models produce superior results. Adapted from Kingma & Dhariwal (2018).

Figure 16.13 Interpolation using GLOW model. The left and right images are real people. The intermediate images were computed by projecting the real images to the latent space, interpolating, and then projecting the interpolated points back to image space. Adapted from Kingma & Dhariwal (2018).

(figure 15.10). Notably, the samples are not as good as those from GANs or diffusion models. It is unknown whether this is due to a fundamental restriction associated with invertible layers or merely because less research effort has been invested in this goal.

Figure 16.13 shows an example of interpolation using GLOW. Two latent vectors are computed by transforming two real images in the normalizing direction. Intermediate points between these latent vectors are computed by linear interpolation, and these are projected back to image space using the network in the generative direction. The result is a set of images that interpolate realistically between the two real ones.

16.5.3 Approximating other density models

Normalizing flows can also learn to generate samples that approximate an existing density which is easy to evaluate but difficult to sample from. In this context, we denote the normalizing flow $Pr(\mathbf{x}|\boldsymbol{\phi})$ as the *student* and the target density $q(\mathbf{x})$ as the *teacher*.

To make progress, we generate samples $\mathbf{x}_i = \mathrm{f}[\mathbf{z}_i, \boldsymbol{\phi}]$ from the student. Since we generated these samples ourselves, we know their corresponding latent variables $\mathbf{z}_i$, and we can calculate their likelihood in the student model without inversion. Thus, we can use a model like a masked-autoregressive flow where inversion is slow. We define a loss function based on the reverse KL divergence that encourages the student and teacher likelihood to be identical and use this to train the student model (figure 16.14):

Problem 16.11

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\mathrm{KL}\left[\frac{1}{I}\sum_{i=1}^{I} \delta\big[\mathbf{x} - \mathrm{f}[\mathbf{z}_i, \boldsymbol{\phi}]\big] \middle\| q(\mathbf{x}) \right] \right]. \tag{16.25}$$

This approach contrasts with the typical use of normalizing flows to build a probability model $Pr(\mathbf{x}_i, \boldsymbol{\phi})$ of data that came from an unknown distribution with samples $\mathbf{x}_i$ using maximum likelihood, which relies on the cross-entropy term from the forward KL divergence (section 5.7):

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\mathrm{KL}\left[\frac{1}{I}\sum_{i=1}^{I} \delta[\mathbf{x} - \mathbf{x}_i] \middle\| Pr(\mathbf{x}_i, \boldsymbol{\phi}) \right] \right]. \tag{16.26}$$

Normalizing flows can model the posterior in VAEs using this trick (see chapter 17).

16.6 Summary

Normalizing flows transform a base distribution (usually a normal distribution) to create a new density. They have the advantage that they can both evaluate the likelihood of samples exactly and generate new samples. However, they have the architectural constraint that each layer must be invertible; we need the forward transformation to generate samples and the backward transformation to evaluate the likelihoods.

It's also important that the Jacobian can be estimated efficiently to evaluate the likelihood; this must be done repeatedly to learn the density. However, invertible layers

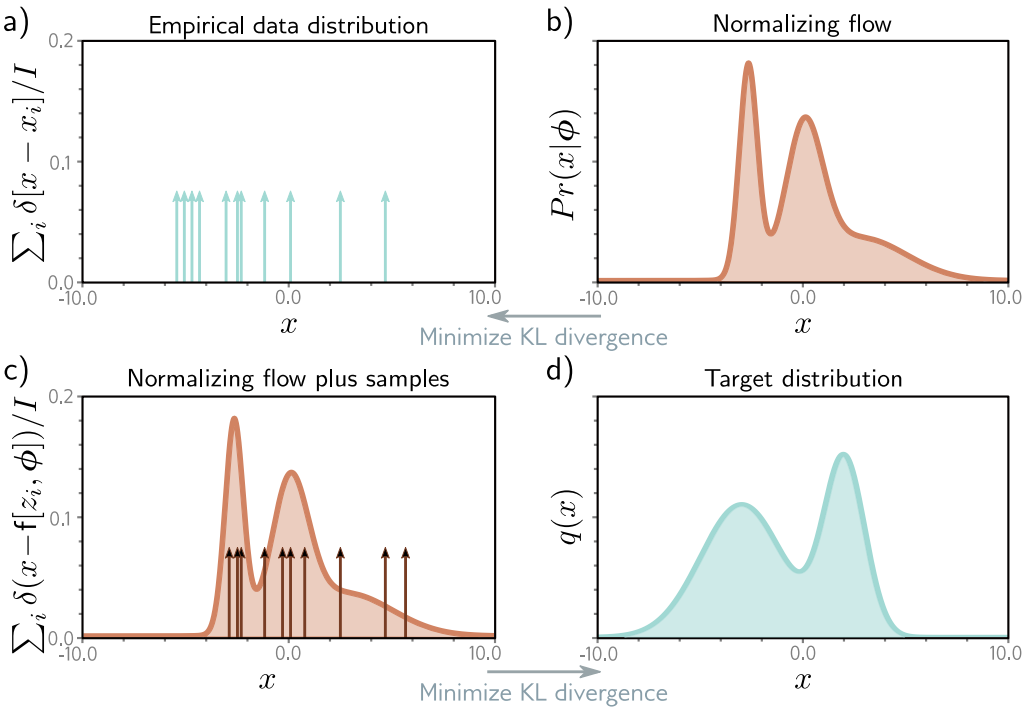

Figure 16.14 Approximating density models. a) Training data. b) Usually, we modify the flow model parameters to minimize the KL divergence from the training data to the flow model. This is equivalent to maximum likelihood fitting (section 5.7). c) Alternatively, we can modify the flow parameters $\boldsymbol{\phi}$ to minimize the KL divergence from the flow samples $x_i = \mathrm{f}[z_i, \boldsymbol{\phi}]$ to d) a target density.

are still useful in their own right even when the Jacobian cannot be estimated efficiently; they reduce the memory requirements of training a K-layer network from $\mathcal{O}[K]$ to $\mathcal{O}[1]$.

This chapter reviewed invertible network layers or flows. We considered linear flows and elementwise flows, which are simple but insufficiently expressive. Then we described more complex flows, such as coupling, autoregressive, and residual flows. Finally, we showed how normalizing flows can be used to estimate likelihoods, generate and interpolate between images, and approximate other distributions.

Notes

Normalizing flows were first introduced by Rezende & Mohamed (2015) but had intellectual antecedents in the work of Tabak & Vanden-Eijnden (2010), Tabak & Turner (2013), and Rippel & Adams (2013). Reviews of normalizing flows can be found in Kobyzev et al. (2020) and Papamakarios et al. (2021). Kobyzev et al. (2020) presented a quantitative comparison of

many normalizing flow approaches. They concluded that the Flow++ model (a coupling flow with a novel elementwise transformation and other innovations) performed best at the time.

Invertible network layers: Invertible layers decrease the memory requirements of the backpropagation algorithm; the activations in the forward pass no longer need to be stored since they can be recomputed in the backward pass. In addition to the regular network layers and residual layers (Gomez et al., 2017; Jacobsen et al., 2018) discussed in this chapter, invertible layers have been developed for graph neural networks (Li et al., 2021a), recurrent neural networks (MacKay et al., 2018), masked convolutions (Song et al., 2019), U-Nets (Brügger et al., 2019; Etmann et al., 2020), and transformers (Mangalam et al., 2022).

Radial and planar flows: The original normalizing flows paper (Rezende & Mohamed, 2015) used planar flows (which contract or expand the distribution along certain dimensions) and radial flows (which expand or contract around a certain point). Inverses for these flows can't be computed easily, but they are useful for approximating distributions where sampling is slow or where the likelihood can only be evaluated up to an unknown scaling factor (figure 16.14).

Applications: Applications include image generation (Ho et al., 2019; Kingma & Dhariwal, 2018), noise modeling (Abdelhamed et al., 2019), video generation (Kumar et al., 2019b), audio generation (Esling et al., 2019; Kim et al., 2018; Prenger et al., 2019), graph generation (Madhawa et al., 2019), image classification (Kim et al., 2021; Mackowiak et al., 2021), image steganography (Lu et al., 2021), super-resolution (Yu et al., 2020; Wolf et al., 2021; Liang et al., 2021), style transfer (An et al., 2021), motion style transfer (Wen et al., 2021), 3D shape modeling (Paschalidou et al., 2021), compression (Zhang et al., 2021b), sRGB to RAW image conversion (Xing et al., 2021), denoising (Liu et al., 2021b), anomaly detection (Yu et al., 2021), image-to-image translation (Ardizzone et al., 2020), synthesizing cell microscopy images under different molecular interventions (Yang et al., 2021), and light transport simulation (Müller et al., 2019b). For applications using image data, noise must be added before learning since the inputs are quantized and hence discrete (see Theis et al., 2016).

Rezende & Mohamed (2015) used normalizing flows to model the posterior in VAEs. Abdal et al. (2021) used normalizing flows to model the distribution of attributes in the latent space of StyleGAN and then used these distributions to change specified attributes in real images. Wolf et al. (2021) use normalizing flows to learn the conditional image of a noisy input image given a clean one and hence simulate noisy data that can be used to train denoising or super-resolution models.

Normalizing flows have also found diverse uses in physics (Kanwar et al., 2020; Köhler et al., 2020; Noé et al., 2019; Wirnsberger et al., 2020; Wong et al., 2020), natural language processing (Tran et al., 2019; Ziegler & Rush, 2019; Zhou et al., 2019; He et al., 2018; Jin et al., 2019), and reinforcement learning (Schroecker et al., 2019; Haarnoja et al., 2018a; Mazoure et al., 2020; Ward et al., 2019; Touati et al., 2020).

Linear flows: Diagonal linear flows can represent normalization transformations like BatchNorm (Dinh et al., 2016) and ActNorm (Kingma & Dhariwal, 2018). Tomczak & Welling (2016) investigated combining triangular matrices and using orthogonal transformations parameterized by the Householder transform. Kingma & Dhariwal (2018) proposed the LU parameterization described in section 16.5.2. Hoogeboom et al. (2019b) proposed using the QR decomposition instead, which does not require predetermined permutation matrices. Convolutions are linear transformations (figure 10.4) that are widely used in deep learning, but their inverse and determinant are not straightforward to compute. Kingma & Dhariwal (2018) used 1×1 convolutions, which is effectively a full linear transformation applied separately at each position. Zheng et al. (2017) introduced ConvFlow, which was restricted to 1D convolutions. Hoogeboom et al. (2019b) provided more general solutions for modeling 2D convolutions either by stacking together masked autoregressive convolutions or by operating in the Fourier domain.

Elementwise flows and coupling functions: Elementwise flows transform each variable independently using the same function (but with different parameters for each variable). The same flows can be used to form the coupling functions in coupling and autoregressive flows, in which case their parameters depend on the preceding variables. To be invertible, these functions must be monotone.

An additive coupling function (Dinh et al., 2015) just adds an offset to the variable. Affine coupling functions scale the variable and add an offset and were used by Dinh et al. (2015), Dinh et al. (2016), Kingma & Dhariwal (2018), Kingma et al. (2016), and Papamakarios et al. (2017). Ziegler & Rush (2019) propose the nonlinear squared flow, which is an invertible ratio of polynomials with five parameters. Continuous mixture CDFs (Ho et al., 2019) apply a monotone transformation based on the cumulative density function (CDF) of a mixture of K logistics, post-composed by an inverse logistic sigmoid, scaled, and offset.

The piecewise linear coupling function (figure 16.5) was developed by Müller et al. (2019b). Since then, systems based on cubic splines (Durkan et al., 2019a) and rational quadratic splines (Durkan et al., 2019b) have been proposed. Huang et al. (2018a) introduced neural autoregressive flows, in which the function is represented by a neural network that produces a monotonic function. A sufficient condition is that the weights are all positive and the activation functions are monotone. It is hard to train a network with the constraint that the weights are positive, so this led to unconstrained monotone neural networks (Wehenkel & Louppe, 2019), which model strictly positive functions and then integrate them numerically to get a monotone function. Jaini et al. (2019) construct positive functions that can be integrated in closed form based on a classic result that all positive single-variable polynomials are the sum of squares of polynomials. Finally, Dinh et al. (2019) investigated piecewise monotonic coupling functions.

Coupling flows: Dinh et al. (2015) introduced coupling flows in which the dimensions were split in half (figure 16.6). Dinh et al. (2016) introduced *RealNVP*, which partitioned the image input by taking alternating pixels or blocks of channels. Das et al. (2019) proposed selecting features for the propagated part based on the magnitude of the derivatives. Dinh et al. (2016) interpreted multi-scale flows (in which dimensions are gradually introduced) as coupling flows in which the parameters $\boldsymbol{\phi}$ have no dependence on the other half of the data. Kruse et al. (2021) introduce a hierarchical formulation of coupling flows in which each partition is recursively divided into two. GLOW (figures 16.12–16.13) was designed by Kingma & Dhariwal (2018) and uses coupling flows, as do NICE (Dinh et al., 2015), RealNVP (Dinh et al., 2016), FloWaveNet (Kim et al., 2018), WaveGlOW (Prenger et al., 2019), and Flow++ (Ho et al., 2019).

Autoregressive flows: Kingma et al. (2016) used autoregressive models for normalizing flows. Germain et al. (2015) developed a general method for masking previous variables. This was exploited by Papamakarios et al. (2017) to compute all of the outputs in the forward direction simultaneously in masked autoregressive flows. Kingma et al. (2016) introduced the inverse autoregressive flow. Parallel WaveNet (Van den Oord et al., 2018) distilled WaveNet (Van den Oord et al., 2016a), which is a different type of generative model for audio, into an inverse autoregressive flow so that sampling would be fast (see figure 16.14c–d).

Residual flows: Residual flows are based on residual networks (He et al., 2016a). RevNets (Gomez et al., 2017) and iRevNets (Jacobsen et al., 2018) divide the input into two sections (figure 16.8), each of which passes through a residual network. These networks are invertible, but the determinant of the Jacobian cannot be computed easily. The residual connection can be interpreted as the discretization of an ordinary differential equation, and this perspective led to different invertible architectures (Chang et al., 2018, 2019a). However, the Jacobian of these networks could still not be computed efficiently. Behrmann et al. (2019) noted that the network can be inverted using fixed point iterations if its Lipschitz constant is less than one. This led to iResNet, in which the log determinant of the Jacobian can be estimated using Hutchinson's trace

estimator (Hutchinson, 1989). Chen et al. (2019) removed the bias induced by the truncation of the power series in equation 16.22 by using the Russian Roulette estimator.

Infinitesimal flows: If residual networks can be viewed as a discretization of an ordinary differential equation (ODE), then the next logical step is to represent the change in the variables directly by an ODE. The neural ODE was explored by Chen et al. (2018e) and exploits standard methods for forward and backward propagation in ODEs. The Jacobian is no longer required to compute the likelihood; this is represented by a different ODE in which the change in log probability is related to the trace of the derivative of the forward propagation. Grathwohl et al. (2019) used the Hutchinson estimator to estimate the trace and simplified this further. Finlay et al. (2020) added regularization terms to the loss function that make training easier, and Dupont et al. (2019) augmented the representation to allow the neural ODE to represent a broader class of diffeomorphisms. Tzen & Raginsky (2019) and Peluchetti & Favaro (2020) replaced the ODEs with stochastic differential equations.

Universality: The universality property refers to the ability of a normalizing flow to model any probability distribution arbitrarily well. Some flows (e.g., planar, elementwise) do not have this property. Autoregressive flows can be shown to have the universality property when the coupling function is a neural monotone network (Huang et al., 2018a), based on monotone polynomials (Jaini et al., 2020) or based on splines (Kobyzev et al., 2020). For dimension D, a series of D coupling flows can form an autoregressive flow. To understand why, note that the partitioning into two parts $\mathbf{h}_1$ and $\mathbf{h}_2$ means that at any given layer $\mathbf{h}_2$ depends only on the previous variables (figure 16.6). Hence, if we increase the size of $\mathbf{h}_1$ by one at every layer, we can reproduce an autoregressive flow, and the result is universal. It is not known whether coupling flows can be universal with fewer than D layers. However, they work well in practice (e.g., GLOW) without the need for this induced autoregressive structure.

Other work: Active areas of research in normalizing flows include the investigation of *discrete flows* (Hoogeboom et al., 2019a; Tran et al., 2019), normalizing flows on non-Euclidean manifolds (Gemici et al., 2016; Wang & Wang, 2019), and *equivariant flows* (Köhler et al., 2020; Rezende et al., 2019) which aim to create densities that are invariant to families of transformations.

Problems

Problem 16.1 Consider transforming a uniform base density defined on $z \in [0, 1]$ using the function $x = \mathrm{f}[z] = z^2$. Find an expression for the transformed distribution $Pr(x)$.

Problem 16.2* Consider transforming a standard normal distribution:

$$Pr(z) = \frac{1}{\sqrt{2\pi}} \exp\left[\frac{-z^2}{2}\right], \tag{16.27}$$

with the function:

$$x = \mathrm{f}[z] = \frac{1}{1 + \exp[-z]}. \tag{16.28}$$

Find an expression for the transformed distribution $Pr(x)$.

Problem 16.3* Write expressions for the Jacobian of the inverse mapping $\mathbf{z} = \mathbf{f}^{-1}[\mathbf{x}, \boldsymbol{\phi}]$ and the absolute determinant of that Jacobian in forms similar to equations 16.6 and 16.7.

Problem 16.4 Compute the inverse and the determinant of the following matrices by hand:

$$\boldsymbol{\Omega}_1 = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & -5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} \qquad \boldsymbol{\Omega}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 4 & 0 & 0 \\ 1 & -1 & 2 & 0 \\ 4 & -2 & -2 & 1 \end{bmatrix}. \tag{16.29}$$

Problem 16.5 Consider a random variable $\mathbf{z}$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$ that is transformed as $\mathbf{x} = \mathbf{A}\mathbf{z} + \mathbf{b}$. Show that the expected value of $\mathbf{x}$ is $\mathbf{A}\boldsymbol{\mu} + \mathbf{b}$ and that the covariance of $\mathbf{x}$ is $\mathbf{A}\boldsymbol{\Sigma}\mathbf{A}^T$.

Problem 16.6* Prove that if $\mathbf{x} = \mathbf{f}[\mathbf{z}] = \mathbf{A}\mathbf{z} + \mathbf{b}$ and $Pr(\mathbf{z}) = \text{Norm}_{\mathbf{z}}[\boldsymbol{\mu}, \boldsymbol{\Sigma}]$, then $Pr(\mathbf{x}) = \text{Norm}_{\mathbf{x}}[\mathbf{A}\boldsymbol{\mu} + \mathbf{b}, \mathbf{A}\boldsymbol{\Sigma}\mathbf{A}^T]$ using the relation:

$$Pr(\mathbf{x}) = \text{Pr}(\mathbf{z}) \cdot \left| \frac{\partial \mathbf{f}[\mathbf{z}]}{\partial \mathbf{z}} \right|^{-1}. \tag{16.30}$$

Problem 16.7 The Leaky ReLU is defined as:

$$\text{LReLU}[z] = \begin{cases} 0.1z & z < 0 \\ z & z \geq 0 \end{cases}. \tag{16.31}$$

Write an expression for the inverse of the leaky ReLU. Write an expression for the inverse absolute determinant of the Jacobian $|\partial \mathbf{f}[\mathbf{z}]/\partial \mathbf{z}|^{-1}$ for an elementwise transformation $\mathbf{x} = \mathbf{f}[z]$ of the multivariate variable $\mathbf{z}$ where:

$$\mathbf{f}[\mathbf{z}] = \Big[\text{LReLU}[z_1], \text{LReLU}[z_2], \ldots, \text{LReLU}[z_D]\Big]^T. \tag{16.32}$$

Problem 16.8 Consider applying the piecewise linear function $\text{f}[h, \boldsymbol{\phi}]$ defined in equation 16.12 for the domain $h' \in [0, 1]$ elementwise to an input $\mathbf{h} = [h_1, h_2, \ldots, h_D]^T$ so that $\mathbf{f}[\mathbf{h}] = [\text{f}[h_1, \boldsymbol{\phi}], \text{f}[h_2, \boldsymbol{\phi}], \ldots, \text{f}[h_D, \boldsymbol{\phi}]]$. What is the Jacobian $\partial \text{f}[\mathbf{h}]/\partial \mathbf{h}$? What is the determinant of the Jacobian?

Problem 16.9* Consider constructing an element-wise flow based on a conical combination of square root functions in equally spaced bins:

$$\mathbf{h}' = \text{f}[h, \boldsymbol{\phi}] = \sqrt{[Kh - b]\,\phi_b} + \sum_{k=1}^{b} \sqrt{\phi_k}, \tag{16.33}$$

where $b = \lfloor Kh \rfloor$ is the bin that h falls into, and the parameters $\boldsymbol{\phi}_k$ are positive, and sum to one. Consider the case where $K = 5$ and $\phi_1 = 0.1, \phi_2 = 0.2, \phi_3 = 0.5, \phi_4 = 0.1, \phi_5 = 0.1$. Draw the function $\text{f}[h, \boldsymbol{\phi}]$. Draw the inverse function $\text{f}^{-1}[h', \boldsymbol{\phi}]$.

Problem 16.10 Draw the structure of the Jacobian (indicating which elements are zero) for the forward mapping of the residual flow in figure 16.8 for the cases where $\mathbf{f}_1[\bullet, \boldsymbol{\phi}_1]$ and $\mathbf{f}_2[\bullet, \boldsymbol{\phi}_2]$ are (i) a fully connected neural network, (ii) an elementwise flow.

Problem 16.11* Write out the expression for the KL divergence in equation 16.25. Why does it not matter if we can only evaluate the probability $q(\mathbf{x})$ up to a scaling factor κ? Does the network have to be invertible to minimize this loss function? Explain your reasoning.

Chapter 17

Variational autoencoders

Generative adversarial networks learn a mechanism for creating samples that are statistically indistinguishable from the training data $\{\mathbf{x}_i\}$. In contrast, like normalizing flows, *variational autoencoders*, or *VAEs*, are *probabilistic generative models*; they aim to learn a distribution $Pr(\mathbf{x})$ over the data (see figure 14.2). After training, it is possible to draw (generate) samples from this distribution. However, the properties of the VAE mean that it is unfortunately *not* possible to evaluate the probability of new examples $\mathbf{x}^*$ exactly.

It is common to talk about the VAE as if it *is* the model of $Pr(\mathbf{x})$, but this is misleading; the VAE is a neural architecture that is designed to help *learn* the model for $Pr(\mathbf{x})$. The final model for $Pr(\mathbf{x})$ contains neither the "variational" nor the "autoencoder" parts and might be better described as a *nonlinear latent variable model*.

This chapter starts by introducing latent variable models in general and then considers the specific case of the nonlinear latent variable model. It will become clear that maximum likelihood learning of this model is not straightforward. Nevertheless, it is possible to define a lower bound on the likelihood, and the VAE architecture approximates this bound using a Monte Carlo (sampling) method. The chapter concludes by presenting several applications of the VAE.

17.1 Latent variable models

Latent variable models take an indirect approach to describing a probability distribution $Pr(\mathbf{x})$ over a multi-dimensional variable $\mathbf{x}$. Instead of directly writing the expression for $Pr(\mathbf{x})$, they model a joint distribution $Pr(\mathbf{x}, \mathbf{z})$ of the data $\mathbf{x}$ and an unobserved *hidden* or *latent variable* $\mathbf{z}$. They then describe the probability of $Pr(\mathbf{x})$ as a marginalization of this joint probability so that:

Appendix C.1.2 Marginalization

$$Pr(\mathbf{x}) = \int Pr(\mathbf{x}, \mathbf{z}) d\mathbf{z}. \tag{17.1}$$

Typically, the joint probability $Pr(\mathbf{x}, \mathbf{z})$ is broken down using the rules of conditional probability into the *likelihood* of the data with respect to the latent variables term $Pr(\mathbf{x}|\mathbf{z})$ and the *prior* $Pr(\mathbf{z})$:

Appendix C.1.3 Conditional probability

$$Pr(\mathbf{x}) = \int Pr(\mathbf{x}|\mathbf{z})Pr(\mathbf{z})d\mathbf{z}. \tag{17.2}$$

This is a rather indirect approach to describing $Pr(\mathbf{x})$, but it is useful because relatively simple expressions for $Pr(\mathbf{x}|\mathbf{z})$ and $Pr(\mathbf{z})$ can define complex distributions $Pr(\mathbf{x})$.

17.1.1 Example: mixture of Gaussians

In a 1D mixture of Gaussians (figure 17.1a), the latent variable z is discrete, and the prior $Pr(z)$ is a categorical distribution (figure 5.9) with one probability λ_n for each possible value of z. The likelihood $Pr(x|z=n)$ of the data x given that the latent variable z takes value n is normally distributed with mean μ_n and variance σ_n^2:

Problem 17.1

$$\begin{aligned} Pr(z=n) &= \lambda_n \\ Pr(x|z=n) &= \text{Norm}_x\left[\mu_n, \sigma_n^2\right]. \end{aligned} \tag{17.3}$$

As in equation 17.2, the likelihood $Pr(x)$ is given by the marginalization over the latent variable z (figure 17.1b). Here, the latent variable is discrete, so we sum over its possible values to marginalize:

$$\begin{aligned} Pr(x) &= \sum_{n=1}^{N} Pr(x, z=n) \\ &= \sum_{n=1}^{N} Pr(x|z=n) \cdot Pr(z=n) \\ &= \sum_{n=1}^{N} \lambda_n \cdot \text{Norm}_x\left[\mu_n, \sigma_n^2\right]. \end{aligned} \tag{17.4}$$

From simple expressions for the likelihood and prior, we describe a complex multi-modal probability distribution.

17.2 Nonlinear latent variable model

In the nonlinear latent variable model, both the data $\mathbf{x}$ and the latent variable $\mathbf{z}$ are continuous and multivariate. The prior $Pr(\mathbf{z})$ is a standard multivariate normal:

Appendix C.3.2 Multivariate normal

$$Pr(\mathbf{z}) = \text{Norm}_{\mathbf{z}}[\mathbf{0}, \mathbf{I}]. \tag{17.5}$$

The likelihood $Pr(\mathbf{x}|\mathbf{z}, \boldsymbol{\phi})$ is also normally distributed; its mean is a nonlinear function $\mathbf{f}[\mathbf{z}, \boldsymbol{\phi}]$ of the latent variable, and its covariance $\sigma^2\mathbf{I}$ is spherical:

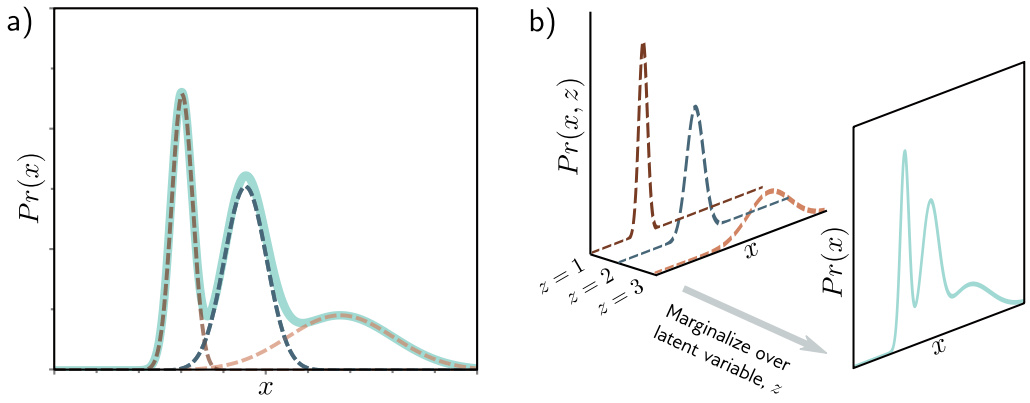

Figure 17.1 Mixture of Gaussians (MoG). a) The MoG describes a complex probability distribution (cyan curve) as a weighted sum of Gaussian components (dashed curves). b) This sum is the marginalization of the joint density $Pr(x, z)$ between the continuous observed data x and a discrete latent variable z.

$$Pr(\mathbf{x}|\mathbf{z}, \boldsymbol{\phi}) = \text{Norm}_{\mathbf{x}}\Big[\mathbf{f}[\mathbf{z}, \boldsymbol{\phi}], \sigma^2\mathbf{I}\Big]. \tag{17.6}$$

The function $\mathbf{f}[\mathbf{z}, \boldsymbol{\phi}]$ is described by a deep network with parameters $\boldsymbol{\phi}$. The latent variable $\mathbf{z}$ is lower dimensional than the data $\mathbf{x}$. The model $\mathbf{f}[\mathbf{z}, \boldsymbol{\phi}]$ describes the important aspects of the data, and the remaining unmodeled aspects are ascribed to the noise $\sigma^2\mathbf{I}$.

Notebook 17.1 Latent variable models

The data probability $Pr(\mathbf{x}|\boldsymbol{\phi})$ is found by marginalizing over the latent variable $\mathbf{z}$:

$$\begin{aligned} Pr(\mathbf{x}|\boldsymbol{\phi}) &= \int Pr(\mathbf{x}, \mathbf{z}|\boldsymbol{\phi}) d\mathbf{z} \\ &= \int Pr(\mathbf{x}|\mathbf{z}, \boldsymbol{\phi}) \cdot Pr(\mathbf{z}) d\mathbf{z} \\ &= \int \text{Norm}_{\mathbf{x}}\Big[\mathbf{f}[\mathbf{z}, \boldsymbol{\phi}], \sigma^2\mathbf{I}\Big] \cdot \text{Norm}_{\mathbf{z}}\left[\mathbf{0}, \mathbf{I}\right] d\mathbf{z}. \end{aligned} \tag{17.7}$$

This can be viewed as an infinite weighted sum (i.e., an infinite mixture) of spherical Gaussians with different means, where the weights are $Pr(\mathbf{z})$ and the means are the network outputs $\mathbf{f}[\mathbf{z}, \boldsymbol{\phi}]$ (figure 17.2).

17.2.1 Generation

Appendix C.4.2 Ancestral sampling

A new example $\mathbf{x}^*$ can be generated using ancestral sampling (figure 17.3). We draw $\mathbf{z}^*$ from the prior $Pr(\mathbf{z})$ and pass this through the network $\mathbf{f}[\mathbf{z}^*, \boldsymbol{\phi}]$ to compute the mean of the likelihood $Pr(\mathbf{x}|\mathbf{z}^*, \boldsymbol{\phi})$ (equation 17.6), from which we draw $\mathbf{x}^*$. Both the prior and likelihood are normal distributions, so this is straightforward.

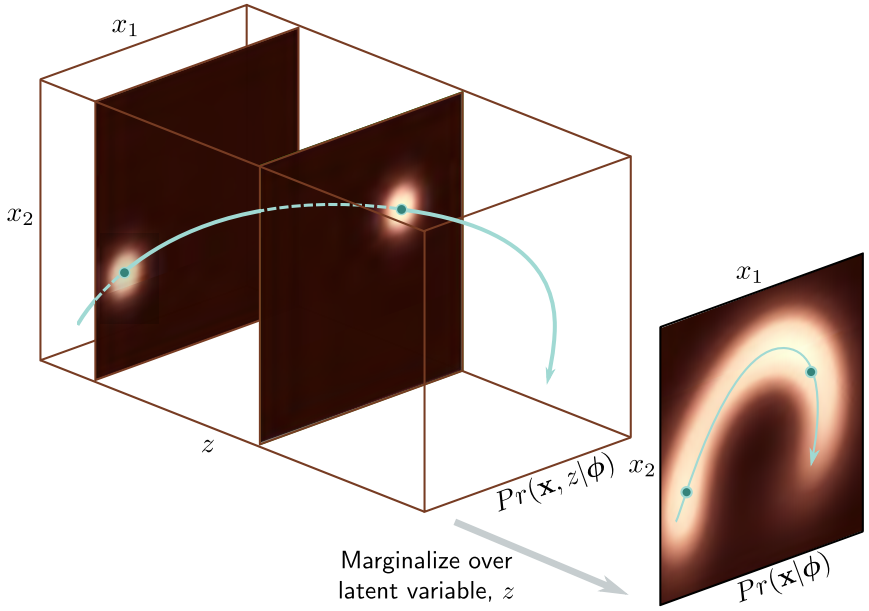

Figure 17.2 Nonlinear latent variable model. A complex 2D density $Pr(\mathbf{x})$ (right) is created as the marginalization of the joint distribution $Pr(\mathbf{x}, z)$ (left) over the latent variable z; to create $Pr(\mathbf{x})$, we integrate the 3D volume over the dimension z. For each z, the distribution over $\mathbf{x}$ is a spherical Gaussian (two slices shown) with a mean $\mathbf{f}[z, \boldsymbol{\phi}]$ that is a nonlinear function of z and depends on parameters $\boldsymbol{\phi}$. The distribution $Pr(\mathbf{x})$ is a weighted sum of these Gaussians.

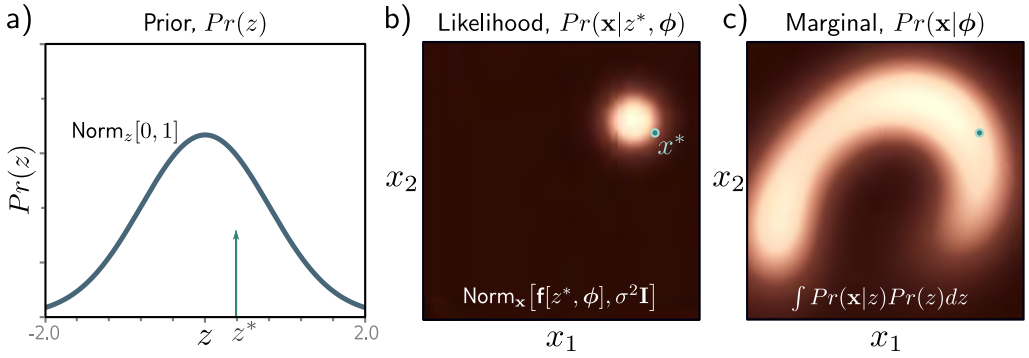

Figure 17.3 Generation from nonlinear latent variable model. a) We draw a sample z^* from the prior probability $Pr(z)$ over the latent variable. b) A sample $\mathbf{x}^*$ is then drawn from $Pr(\mathbf{x}|z^*, \boldsymbol{\phi})$. This is a spherical Gaussian with a mean that is a nonlinear function $\mathbf{f}[\bullet, \boldsymbol{\phi}]$ of z^* and a fixed variance $\sigma^2\mathbf{I}$. c) If we repeat this process many times, we recover the density $Pr(\mathbf{x}|\boldsymbol{\phi})$.

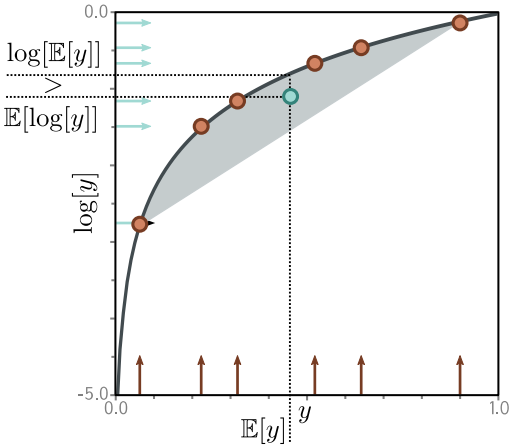

Figure 17.4 Jensen's inequality (discrete case). The logarithm (black curve) is a concave function; you can draw a straight line between any two points on the curve, and this line will always lie underneath it. It follows that any convex combination (weighted sum with positive weights that sum to one) of the six points on the log function must lie in the gray region under the curve. Here, we have weighted the points equally (i.e., taken the mean) to yield the cyan point. Since this point lies below the curve, $\log[\mathbb{E}[y]] > \mathbb{E}[\log[y]]$.

17.3 Training

To train the model, we maximize the log-likelihood over a training dataset $\{\mathbf{x}_i\}_{i=1}^{I}$ with respect to the model parameters. For simplicity, we assume that the variance term σ^2 in the likelihood expression is known and concentrate on learning $\boldsymbol{\phi}$:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\sum_{i=1}^{I} \log\Big[Pr(\mathbf{x}_i|\boldsymbol{\phi}) \Big] \right], \tag{17.8}$$

where:

$$Pr(\mathbf{x}_i|\boldsymbol{\phi}) = \int \text{Norm}_{\mathbf{x}_i}[\mathbf{f}[\mathbf{z},\boldsymbol{\phi}], \sigma^2\mathbf{I}] \cdot \text{Norm}_{\mathbf{z}}[\mathbf{0},\mathbf{I}] d\mathbf{z}. \tag{17.9}$$

Unfortunately, this is intractable. There is no closed-form expression for the integral and no easy way to evaluate it for a particular value of $\mathbf{x}$.

17.3.1 Evidence lower bound (ELBO)

To make progress, we define a *lower bound* on the log-likelihood. This is a function that is always less than or equal to the log-likelihood for a given value of $\boldsymbol{\phi}$ and will also depend on some other parameters $\boldsymbol{\theta}$. Eventually, we will build a network to compute this lower bound and optimize it. To define this lower bound, we need *Jensen's inequality*.

17.3.2 Jensen's inequality

Appendix B.1.2 Concave functions

Jensen's inequality says that a concave function $g[\bullet]$ of the expectation of data y is greater than or equal to the expectation of the function of the data:

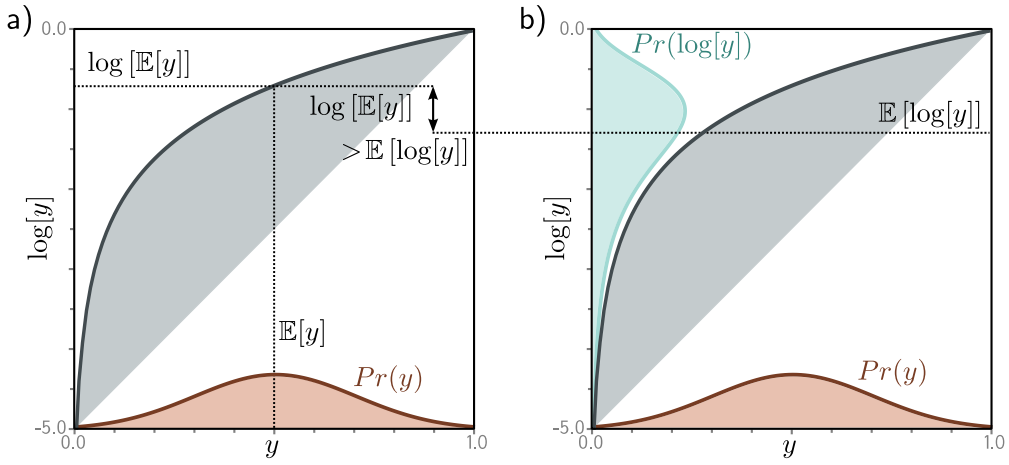

Figure 17.5 Jensen's inequality (continuous case). For a concave function, computing the expectation of a distribution $Pr(y)$ and passing it through the function gives a result greater than or equal to transforming the variable y by the function and then computing the expectation of the new variable. In the case of the logarithm, we have $\log[\mathbb{E}[y]] \geq \mathbb{E}[\log[y]]$. The left-hand side of the figure corresponds to the left-hand side of this inequality and the right-hand side of the figure to the right-hand side. One way of thinking about this is to consider that we are taking a convex combination of the points in the orange distribution defined over $y \in [0, 1]$. By the logic of figure 17.4, this must lie under the curve. Alternatively, we can think about the concave function as compressing the high values of y relative to the low values, so the expected value is lower when we pass y through the function first.

$$g[\mathbb{E}[y]] \geq \mathbb{E}\big[g[y]\big]. \tag{17.10}$$

In this case, the concave function is the logarithm, so we have:

Problems 17.2–17.3

$$\log\big[\mathbb{E}[y]\big] \geq \mathbb{E}\big[\log[y]\big], \tag{17.11}$$

or writing out the expression for the expectation in full, we have:

$$\log\left[\int Pr(y) y dy\right] \geq \int Pr(y)\log[y] dy. \tag{17.12}$$

This is explored in figures 17.4–17.5. In fact, the slightly more general statement is true:

$$\log\left[\int Pr(y) h[y] dy\right] \geq \int Pr(y)\log[h[y]] dy. \tag{17.13}$$

where $h[y]$ is a function of y. This follows because $h[y]$ is another random variable with a new distribution. Since we never specified $Pr(y)$, the relation remains true.

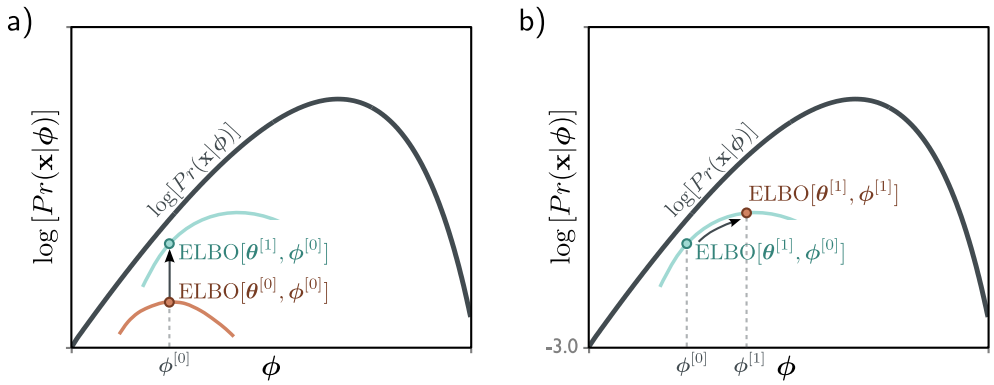

Figure 17.6 Evidence lower bound (ELBO). The goal is to maximize the log-likelihood $\log[Pr(\mathbf{x}|\boldsymbol{\phi})]$ (black curve) with respect to the parameters $\boldsymbol{\phi}$. The ELBO is a function that lies everywhere below the log-likelihood. It is a function of both $\boldsymbol{\phi}$ and a second set of parameters $\boldsymbol{\theta}$. For fixed $\boldsymbol{\theta}$, we get a function of $\boldsymbol{\phi}$ (two colored curves for different values of $\boldsymbol{\theta}$). Consequently, we can increase the log-likelihood by either improving the ELBO with respect to a) the new parameters $\boldsymbol{\theta}$ (moving from colored curve to colored curve) or b) the original parameters $\boldsymbol{\phi}$ (moving along the current colored curve).

17.3.3 Deriving the bound

We now use Jensen's inequality to derive the lower bound for the log-likelihood. We start by multiplying and dividing the log-likelihood by an arbitrary probability distribution $q(\mathbf{z})$ over the latent variables:

$$\begin{aligned}\log[Pr(\mathbf{x}|\boldsymbol{\phi})] &= \log\left[\int Pr(\mathbf{x},\mathbf{z}|\boldsymbol{\phi})d\mathbf{z}\right] \\ &= \log\left[\int q(\mathbf{z})\frac{Pr(\mathbf{x},\mathbf{z}|\boldsymbol{\phi})}{q(\mathbf{z})}d\mathbf{z}\right], \end{aligned} \tag{17.14}$$

We then use Jensen's inequality for the logarithm (equation 17.12) to find a lower bound:

$$\log\left[\int q(\mathbf{z})\frac{Pr(\mathbf{x},\mathbf{z}|\boldsymbol{\phi})}{q(\mathbf{z})}d\mathbf{z}\right] \geq \int q(\mathbf{z})\log\left[\frac{Pr(\mathbf{x},\mathbf{z}|\boldsymbol{\phi})}{q(\mathbf{z})}\right]d\mathbf{z}, \tag{17.15}$$

where the right-hand side is termed the *evidence lower bound* or *ELBO*. It gets this name because $Pr(\mathbf{x}|\boldsymbol{\phi})$ is called the *evidence* in the context of Bayes' rule (equation 17.19).

In practice, the distribution $q(\mathbf{z})$ has parameters $\boldsymbol{\theta}$, so the ELBO can be written as:

$$\text{ELBO}[\boldsymbol{\theta},\boldsymbol{\phi}] = \int q(\mathbf{z}|\boldsymbol{\theta})\log\left[\frac{Pr(\mathbf{x},\mathbf{z}|\boldsymbol{\phi})}{q(\mathbf{z}|\boldsymbol{\theta})}\right]d\mathbf{z}. \tag{17.16}$$

To learn the nonlinear latent variable model, we maximize this quantity as a function of both $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$. The neural architecture that computes this quantity is the VAE.

17.4 ELBO properties

When first encountered, the ELBO is a somewhat mysterious object, so we now provide some intuition about its properties. Consider that the original log-likelihood of the data is a function of the parameters $\boldsymbol{\phi}$ and that we want to find its maximum. For any fixed $\boldsymbol{\theta}$, the ELBO is still a function of the parameters but one that must lie below the original likelihood function. When we change $\boldsymbol{\theta}$, we modify this function, and depending on our choice, the lower bound may move closer or further from the log-likelihood. When we change $\boldsymbol{\phi}$, we move along the lower bound function (figure 17.6).

17.4.1 Tightness of bound

The ELBO is *tight* when, for a fixed value of $\boldsymbol{\phi}$, the ELBO and the likelihood function coincide. To find the distribution $q(\mathbf{z}|\boldsymbol{\theta})$ that makes the bound tight, we factor the numerator of the log term in the ELBO using the definition of conditional probability:

Appendix C.1.3 Conditional probability

$$\begin{aligned}
\text{ELBO}[\boldsymbol{\theta}, \boldsymbol{\phi}] &= \int q(\mathbf{z}|\boldsymbol{\theta}) \log\left[\frac{Pr(\mathbf{x}, \mathbf{z}|\boldsymbol{\phi})}{q(\mathbf{z}|\boldsymbol{\theta})}\right] d\mathbf{z} \\
&= \int q(\mathbf{z}|\boldsymbol{\theta}) \log\left[\frac{Pr(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi}) Pr(\mathbf{x}|\boldsymbol{\phi})}{q(\mathbf{z}|\boldsymbol{\theta})}\right] d\mathbf{z} \\
&= \int q(\mathbf{z}|\boldsymbol{\theta}) \log\big[Pr(\mathbf{x}|\boldsymbol{\phi})\big] d\mathbf{z} + \int q(\mathbf{z}|\boldsymbol{\theta}) \log\left[\frac{Pr(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})}{q(\mathbf{z}|\boldsymbol{\theta})}\right] d\mathbf{z} \\
&= \log\big[Pr(\mathbf{x}|\boldsymbol{\phi})\big] + \int q(\mathbf{z}|\boldsymbol{\theta}) \log\left[\frac{Pr(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})}{q(\mathbf{z}|\boldsymbol{\theta})}\right] d\mathbf{z} \\
&= \log\big[Pr(\mathbf{x}|\boldsymbol{\phi})\big] - \mathrm{D}_{KL}\Big[q(\mathbf{z}|\boldsymbol{\theta})\Big|\Big|Pr(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})\Big]. \qquad (17.17)
\end{aligned}$$

Here, the first integral disappears between lines three and four since $\log[Pr(\mathbf{x}|\boldsymbol{\phi})]$ does not depend on $\mathbf{z}$, and the integral of the probability distribution $q(\mathbf{z}|\boldsymbol{\theta})$ is one. In the last line, we have just used the definition of the Kullback-Leibler (KL) divergence.

Appendix C.5.1 KL divergence

This equation shows that the ELBO is the original log-likelihood minus the KL divergence $\mathrm{D}_{KL}\left[q(\mathbf{z}|\boldsymbol{\theta})||Pr(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})\right]$. The KL divergence measures the "distance" between distributions and can only take non-negative values. It follows the ELBO is a lower bound on $\log[Pr(\mathbf{x}|\boldsymbol{\phi})]$. The KL distance will be zero, and the bound will be *tight* when $q(\mathbf{z}|\boldsymbol{\theta}) = Pr(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})$. This is the posterior distribution over the latent variables $\mathbf{z}$ given observed data $\mathbf{x}$; it indicates which values of the latent variable could have been responsible for the data point (figure 17.7).

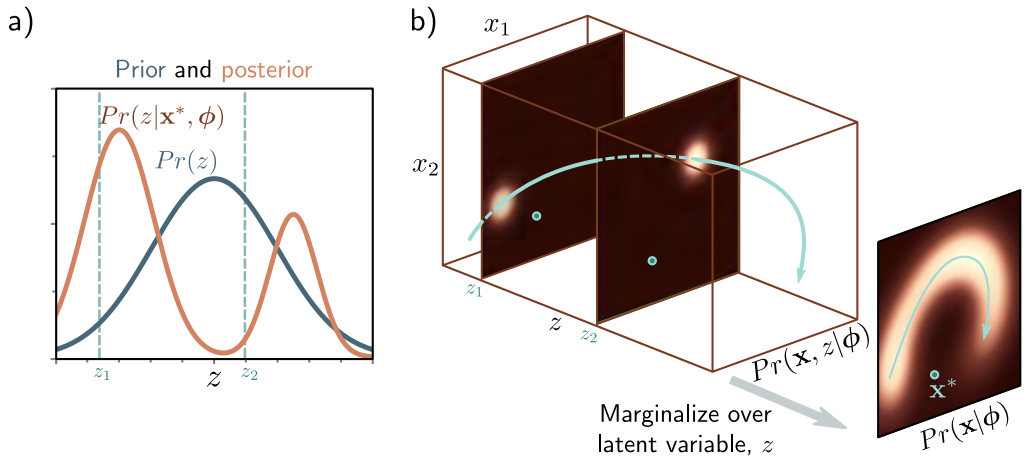

Figure 17.7 Posterior distribution over latent variable. a) The posterior distribution $Pr(z|\mathbf{x}^*,\boldsymbol{\phi})$ is the distribution over the values of the latent variable z that could be responsible for a data point $\mathbf{x}^*$. We calculate this via Bayes' rule $Pr(z|\mathbf{x}^*,\boldsymbol{\phi}) \propto Pr(\mathbf{x}^*|z,\boldsymbol{\phi})Pr(z)$. b) We compute the first term on the right-hand side (the likelihood) by assessing the probability of $\mathbf{x}^*$ against the symmetric Gaussian associated with each value of z. Here, it was more likely to have been created from z_1 than z_2. The second term is the prior probability $Pr(z)$ over the latent variable. Combining these two factors and normalizing so the distribution sums to one gives us the posterior $Pr(z|\mathbf{x}^*,\boldsymbol{\phi})$.

17.4.2 ELBO as reconstruction loss minus KL distance to prior

Equations 17.16 and 17.17 are two different ways to express the ELBO. A third way is to consider the bound as reconstruction error minus the distance to the prior:

$$\begin{aligned}
\text{ELBO}[\boldsymbol{\theta},\boldsymbol{\phi}] &= \int q(\mathbf{z}|\boldsymbol{\theta}) \log\left[\frac{Pr(\mathbf{x},\mathbf{z}|\boldsymbol{\phi})}{q(\mathbf{z}|\boldsymbol{\theta})}\right] d\mathbf{z} \\
&= \int q(\mathbf{z}|\boldsymbol{\theta}) \log\left[\frac{Pr(\mathbf{x}|\mathbf{z},\boldsymbol{\phi})Pr(\mathbf{z})}{q(\mathbf{z}|\boldsymbol{\theta})}\right] d\mathbf{z} \\
&= \int q(\mathbf{z}|\boldsymbol{\theta}) \log\left[Pr(\mathbf{x}|\mathbf{z},\boldsymbol{\phi})\right] d\mathbf{z} + \int q(\mathbf{z}|\boldsymbol{\theta}) \log\left[\frac{Pr(\mathbf{z})}{q(\mathbf{z}|\boldsymbol{\theta})}\right] d\mathbf{z} \\
&= \int q(\mathbf{z}|\boldsymbol{\theta}) \log\Big[Pr(\mathbf{x}|\mathbf{z},\boldsymbol{\phi})\Big] d\mathbf{z} - \mathrm{D}_{KL}\Big[q(\mathbf{z}|\boldsymbol{\theta})\Big|\Big|Pr(\mathbf{z})\Big], \qquad (17.18)
\end{aligned}$$

Problem 17.4

where the joint distribution $Pr(\mathbf{x},\mathbf{z}|\boldsymbol{\phi})$ has been factored into conditional probability $Pr(\mathbf{x}|\mathbf{z},\boldsymbol{\phi})Pr(\mathbf{z})$ between the first and second lines, and the definition of KL divergence is used again in the last line.

In this formulation, the first term measures the average agreement $Pr(\mathbf{x}|\mathbf{z},\boldsymbol{\phi})$ of the latent variable and the data. This is termed the *reconstruction loss*. The second term measures the degree to which the auxiliary distribution $q(\mathbf{z}|\boldsymbol{\theta})$ matches the prior. This formulation is the one that is used in the variational autoencoder.

17.5 Variational approximation

We saw in equation 17.17 that the ELBO is tight when $q(\mathbf{z}|\boldsymbol{\theta})$ is the posterior $Pr(\mathbf{z}|\mathbf{x},\boldsymbol{\phi})$. In principle, we can compute the posterior using Bayes' rule:

$$Pr(\mathbf{z}|\mathbf{x},\boldsymbol{\phi}) = \frac{Pr(\mathbf{x}|\mathbf{z},\boldsymbol{\phi})Pr(\mathbf{z})}{Pr(\mathbf{x}|\boldsymbol{\phi})}, \tag{17.19}$$

but in practice, this is intractable because we can't evaluate the evidence term $Pr(\mathbf{x}|\boldsymbol{\phi})$ in the denominator (see section 17.3).

One solution is to make a variational approximation: we choose a simple parametric form for $q(\mathbf{z}|\boldsymbol{\theta})$ and use this to approximate the true posterior. Here, we choose a multivariate normal distribution with mean $\boldsymbol{\mu}$ and diagonal covariance $\boldsymbol{\Sigma}$. This will not always match the posterior well but will be better for some values of $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ than others. During training, we will find the normal distribution that is "closest" to the true posterior $Pr(\mathbf{z}|\mathbf{x})$ (figure 17.8). This corresponds to minimizing the KL divergence in equation 17.17 and moving the colored curves in figure 17.6 upwards.

Appendix C.3.2 Multivariate normal

Since the optimal choice for $q(\mathbf{z}|\boldsymbol{\theta})$ was the posterior $Pr(\mathbf{z}|\mathbf{x})$, and this depends on the data example $\mathbf{x}$, the variational approximation should do the same, so we choose:

$$q(\mathbf{z}|\mathbf{x},\boldsymbol{\theta}) = \text{Norm}_{\mathbf{z}}\Big[\mathbf{g}_{\boldsymbol{\mu}}[\mathbf{x},\boldsymbol{\theta}], \mathbf{g}_{\boldsymbol{\Sigma}}[\mathbf{x},\boldsymbol{\theta}]\Big], \tag{17.20}$$

where $\mathbf{g}[\mathbf{x},\boldsymbol{\theta}]$ is a second neural network with parameters $\boldsymbol{\theta}$ that predicts the mean $\boldsymbol{\mu}$ and variance $\boldsymbol{\Sigma}$ of the normal variational approximation.

17.6 The variational autoencoder

Finally, we can describe the VAE. We build a network that computes the ELBO:

$$\text{ELBO}[\boldsymbol{\theta},\boldsymbol{\phi}] = \int q(\mathbf{z}|\mathbf{x},\boldsymbol{\theta}) \log\big[Pr(\mathbf{x}|\mathbf{z},\boldsymbol{\phi})\big] d\mathbf{z} - \text{D}_{KL}\Big[q(\mathbf{z}|\mathbf{x},\boldsymbol{\theta})\Big|\Big|Pr(\mathbf{z})\Big], \tag{17.21}$$

where the distribution $q(\mathbf{z}|\mathbf{x},\boldsymbol{\theta})$ is the approximation from equation 17.20.

The first term still involves an intractable integral, but since it is an expectation with respect to $q(\mathbf{z}|\mathbf{x},\boldsymbol{\theta})$, we can approximate it by sampling. For any function $\text{a}[\bullet]$ we have:

Appendix C.2 Expectation

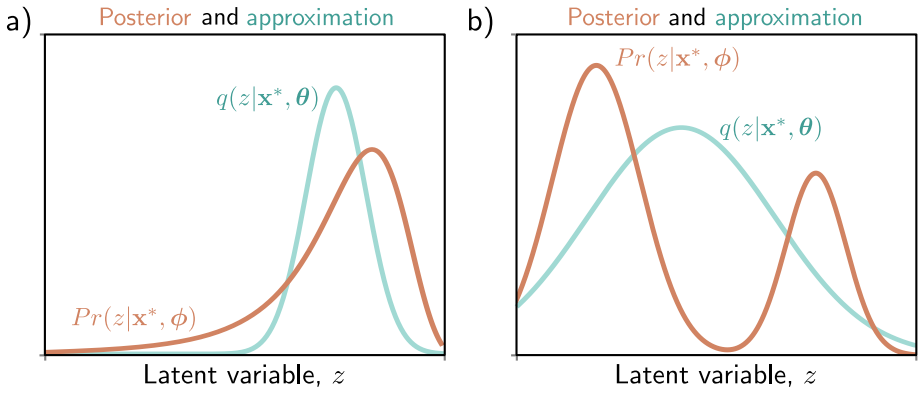

Figure 17.8 Variational approximation. The posterior $Pr(\mathbf{z}|\mathbf{x}^*, \boldsymbol{\phi})$ can't be computed in closed form. The variational approximation chooses a family of distributions $q(\mathbf{z}|\mathbf{x}, \boldsymbol{\theta})$ (here Gaussians) and tries to find the closest member of this family to the true posterior. a) Sometimes, the approximation (cyan curve) is good and lies close to the true posterior (orange curve). b) However, if the posterior is multi-modal (as in figure 17.7), then the Gaussian approximation will be poor.

$$\mathbb{E}_{\mathbf{z}}\big[\mathrm{a}[\mathbf{z}]\big] = \int \mathrm{a}[\mathbf{z}] q(\mathbf{z}|\mathbf{x}, \boldsymbol{\theta}) d\mathbf{z} \approx \frac{1}{N}\sum_{n=1}^{N} \mathrm{a}[\mathbf{z}_n^*], \tag{17.22}$$

where $\mathbf{z}_n^*$ is the n^{th} sample from $q(\mathbf{z}|\mathbf{x}, \boldsymbol{\theta})$. This is known as a *Monte Carlo estimate*. For a very approximate estimate, we can just use a single sample $\mathbf{z}^*$ from $q(\mathbf{z}|\mathbf{x}, \boldsymbol{\theta})$:

$$\mathrm{ELBO}[\boldsymbol{\theta}, \boldsymbol{\phi}] \approx \log\big[Pr(\mathbf{x}|\mathbf{z}^*, \boldsymbol{\phi})\big] - \mathrm{D}_{KL}\Big[q(\mathbf{z}|\mathbf{x}, \boldsymbol{\theta})\Big|\Big|Pr(\mathbf{z})\Big]. \tag{17.23}$$

Appendix C.5.4 KL divergence between normal distributions

The second term is the KL divergence between the variational distribution $q(\mathbf{z}|\mathbf{x}, \boldsymbol{\theta}) = \mathrm{Norm}_{\mathbf{z}}[\boldsymbol{\mu}, \boldsymbol{\Sigma}]$ and the prior $Pr(\mathbf{z}) = \mathrm{Norm}_{\mathbf{z}}[\mathbf{0}, \mathbf{I}]$. The KL divergence between two normal distributions can be calculated in closed form. For the special case where one distribution has parameters $\boldsymbol{\mu}, \boldsymbol{\Sigma}$ and the other is a standard normal, it is given by:

$$\mathrm{D}_{KL}\Big[q(\mathbf{z}|\mathbf{x}, \boldsymbol{\theta})\Big|\Big|Pr(\mathbf{z})\Big] = \frac{1}{2}\Big(\mathrm{Tr}[\boldsymbol{\Sigma}] + \boldsymbol{\mu}^T\boldsymbol{\mu} - D_{\mathbf{z}} - \log\Big[\det[\boldsymbol{\Sigma}]\Big]\Big). \tag{17.24}$$

where $D_{\mathbf{z}}$ is the dimensionality of the latent space.

17.6.1 VAE algorithm

To summarize, we aim to build a model that computes the evidence lower bound for a point $\mathbf{x}$. Then we use an optimization algorithm to maximize this lower bound over the

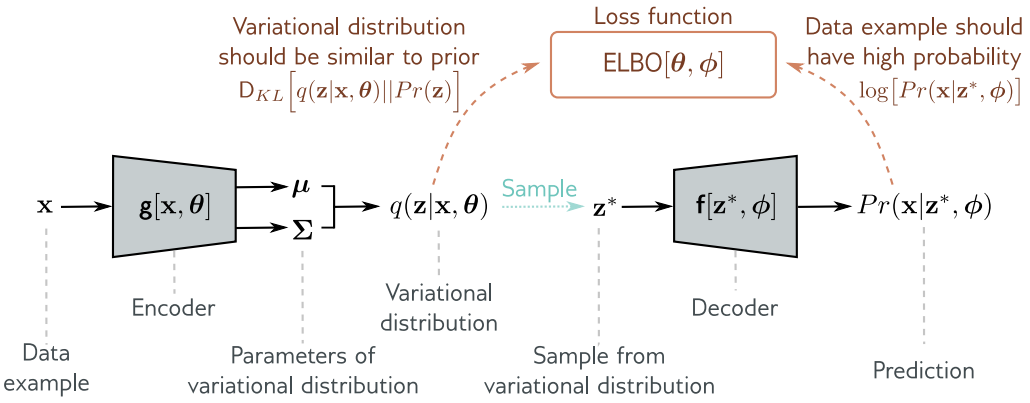

Figure 17.9 Variational autoencoder. The encoder $\mathbf{g}[\mathbf{x},\boldsymbol{\theta}]$ takes a training example $\mathbf{x}$ and predicts the parameters $\boldsymbol{\mu},\boldsymbol{\Sigma}$ of the variational distribution $q(\mathbf{z}|\mathbf{x},\boldsymbol{\theta})$. We sample from this distribution and then use the decoder $\mathbf{f}[\mathbf{z},\boldsymbol{\phi}]$ to predict the data $\mathbf{x}$. The loss function is the negative ELBO, which depends on how accurate this prediction is and how similar the variational distribution $q(\mathbf{z}|\mathbf{x},\boldsymbol{\theta})$ is to the prior $Pr(\mathbf{z})$ (equation 17.21).

dataset and hence improve the log-likelihood. To compute the ELBO we:

- compute the mean $\boldsymbol{\mu}$ and variance $\boldsymbol{\Sigma}$ of the variational posterior distribution $q(\mathbf{z}|\boldsymbol{\theta},\mathbf{x})$ for this data point $\mathbf{x}$ using the network $\mathbf{g}[\mathbf{x},\boldsymbol{\theta}]$,
- draw a sample $\mathbf{z}^*$ from this distribution, and
- compute the ELBO using equation 17.23.

The associated architecture is shown in figure 17.9. It should now be clear why this is called a variational autoencoder. It is variational because it computes a Gaussian approximation to the posterior distribution. It is an autoencoder because it starts with a data point $\mathbf{x}$, computes a lower-dimensional latent vector $\mathbf{z}$ from this, and then uses this vector to recreate the data point $\mathbf{x}$ as closely as possible. In this context, the mapping from the data to the latent variable by the network $\mathbf{g}[\mathbf{x},\boldsymbol{\theta}]$ is called the *encoder*, and the mapping from the latent variable to the data by the network $\mathbf{f}[\mathbf{z},\boldsymbol{\phi}]$ is called the *decoder*.

The VAE computes the ELBO as a function of both $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$. To maximize this bound, we run mini-batches of samples through the network and update these parameters with an optimization algorithm such as SGD or Adam. The gradients of the ELBO with respect to the parameters are computed as usual using automatic differentiation. During this process, we are both moving between the colored curves (changing $\boldsymbol{\theta}$) and along them (changing $\boldsymbol{\phi}$) in figure 17.10. During this process, the parameters $\boldsymbol{\phi}$ change to assign the data a higher likelihood in the nonlinear latent variable model.

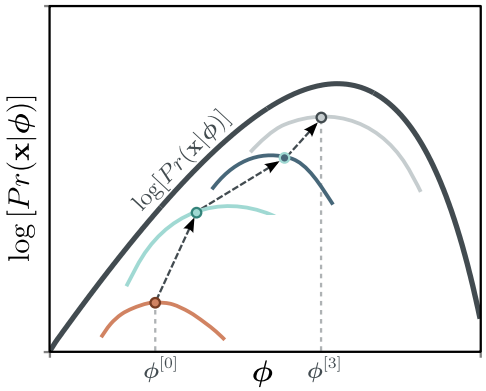

Figure 17.10 The VAE updates both factors that determine the lower bound at each iteration. Both the parameters $\boldsymbol{\phi}$ of the decoder and the parameters $\boldsymbol{\theta}$ of the encoder are manipulated to increase this lower bound.

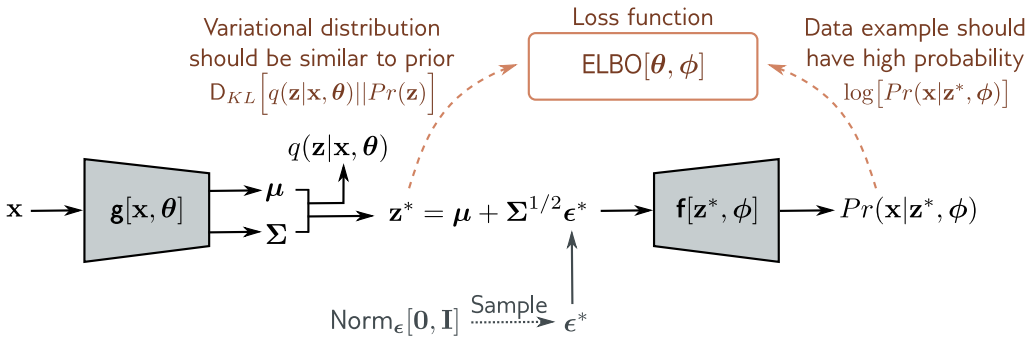

Figure 17.11 Reparameterization trick. With the original architecture (figure 17.9), we cannot easily backpropagate through the sampling step. The reparameterization trick removes the sampling step from the main pipeline; we draw from a standard normal and combine this with the predicted mean and covariance to get a sample from the variational distribution.

17.7 The reparameterization trick

There is one more complication; the network involves a sampling step, and it is difficult to differentiate through this stochastic component. However, differentiating past this step is necessary to update the parameters $\boldsymbol{\theta}$ that precede it in the network.

Problem 17.5

Fortunately, there is a simple solution; we can move the stochastic part into a branch of the network that draws a sample $\boldsymbol{\epsilon}^*$ from $\mathrm{Norm}_{\boldsymbol{\epsilon}}[\mathbf{0},\mathbf{I}]$ and then use the relation:

$$\mathbf{z}^* = \boldsymbol{\mu} + \boldsymbol{\Sigma}^{1/2}\boldsymbol{\epsilon}^*, \tag{17.25}$$

Notebook 17.2 Reparameterization trick

to draw from the intended Gaussian. Now we can compute the derivatives as usual because the backpropagation algorithm does not need to pass down the stochastic branch. This is known as the *reparameterization trick* (figure 17.11).

17.8 Applications

Variational autoencoders have many uses, including denoising, anomaly detection, and compression. This section reviews several applications for image data.

17.8.1 Approximating sample probability

In section 17.3, we argued that it is not possible to evaluate the probability of a sample with the VAE, which describes this probability as:

$$\begin{aligned} Pr(\mathbf{x}) &= \int Pr(\mathbf{x}|\mathbf{z})Pr(\mathbf{z})d\mathbf{z} \\ &= \mathbb{E}_{\mathbf{z}}\Big[Pr(\mathbf{x}|\mathbf{z})\Big] \\ &= \mathbb{E}_{\mathbf{z}}\Big[\text{Norm}_{\mathbf{x}}[\mathbf{f}[\mathbf{z},\boldsymbol{\phi}],\sigma^2\mathbf{I}]\Big]. \end{aligned} \tag{17.26}$$

In principle, we could *approximate* this probability using equation 17.22 by drawing samples from $Pr(\mathbf{z}) = \text{Norm}_{\mathbf{z}}[\mathbf{0},\mathbf{I}]$ and computing:

$$Pr(\mathbf{x}) \approx \frac{1}{N}\sum_{n=1}^{N} Pr(\mathbf{x}|\mathbf{z}_n). \tag{17.27}$$

However, the curse of dimensionality means that almost all values of $\mathbf{z}_n$ that we draw would have a very low probability; we would have to draw an enormous number of samples to get a reliable estimate. A better approach is to use *importance sampling*. Here, we sample $\mathbf{z}$ from an auxiliary distribution $q(\mathbf{z})$, evaluate $Pr(\mathbf{x}|\mathbf{z}_n)$, and rescale the resulting values by the probability $q(\mathbf{z})$ under the new distribution:

$$\begin{aligned} Pr(\mathbf{x}) &= \int Pr(\mathbf{x}|\mathbf{z})Pr(\mathbf{z})d\mathbf{z} \\ &= \int \frac{Pr(\mathbf{x}|\mathbf{z})Pr(\mathbf{z})}{q(\mathbf{z})} q(\mathbf{z})d\mathbf{z} \\ &= \mathbb{E}_{q(\mathbf{z})}\left[\frac{Pr(\mathbf{x}|\mathbf{z})Pr(\mathbf{z})}{q(\mathbf{z})}\right] \\ &\approx \frac{1}{N}\sum_{n=1}^{N}\frac{Pr(\mathbf{x}|\mathbf{z}_n)Pr(\mathbf{z}_n)}{q(\mathbf{z}_n)}, \end{aligned} \tag{17.28}$$

where now we draw the samples from $q(\mathbf{z})$. If $q(\mathbf{z})$ is close to the region of $\mathbf{z}$ where the $Pr(\mathbf{x}|\mathbf{z})$ has high likelihood, then we will focus the sampling on the relevant area of space and estimate $Pr(\mathbf{x})$ much more efficiently.

Notebook 17.3 Importance sampling

The product $Pr(\mathbf{x}|\mathbf{z})Pr(\mathbf{z})$ that we are trying to integrate is proportional to the posterior distribution $Pr(\mathbf{z}|\mathbf{x})$ (by Bayes' rule). Hence, a sensible choice of auxiliary distribution $q(\mathbf{z})$ is the variational posterior $q(\mathbf{z}|\mathbf{x})$ computed by the encoder.

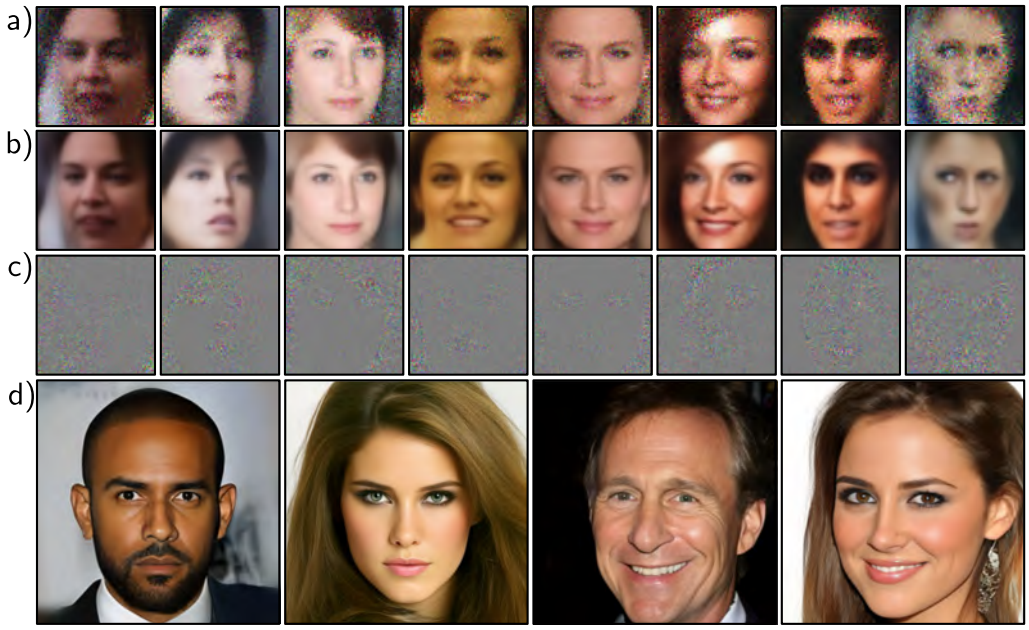

Figure 17.12 Sampling from a standard VAE trained on CELEBA. In each column, a latent variable $\mathbf{z}^*$ is drawn and passed through the model to predict the mean $\mathbf{f}[\mathbf{z}^*, \boldsymbol{\phi}]$ before adding independent Gaussian noise (see figure 17.3). a) A set of samples that are the sum of b) the predicted means and c) spherical Gaussian noise vectors. The images look too smooth before we add the noise and too noisy afterward. This is typical, and usually, the noise-free version is shown since the noise is considered to represent aspects of the image that are not modeled. Adapted from Dorta et al. (2018). d) It is now possible to generate high-quality images from VAEs using hierarchical priors, specialized architecture, and careful regularization. Adapted from Vahdat & Kautz (2020).

In this way, we can approximate the probability of new samples. With sufficient samples, this will provide a better estimate than the lower bound and could be used to evaluate the quality of the model by evaluating the log-likelihood of test data. Alternatively, it could be used as a criterion for determining whether new examples belong to the distribution or are anomalous.

17.8.2 Generation

VAEs build a probabilistic model, and it's easy to sample from this model by drawing from the prior $Pr(\mathbf{z})$ over the latent variable, passing this result through the decoder $\mathbf{f}[\mathbf{z}, \boldsymbol{\phi}]$, and adding noise according to $Pr(\mathbf{x}|\mathbf{f}[\mathbf{z}, \boldsymbol{\phi}])$. Unfortunately, samples from

vanilla VAEs are generally low-quality (figure 17.12a–c). This is partly because of the naïve spherical Gaussian noise model and partly because of the Gaussian models used for the prior and variational posterior. One trick to improve generation quality is to sample from the *aggregated posterior* $q(\mathbf{z}|\boldsymbol{\theta}) = (1/I)\sum_i q(\mathbf{z}|\mathbf{x}_i, \boldsymbol{\theta})$ rather than the prior; this is the average posterior over all samples and is a mixture of Gaussians that is more representative of true distribution in latent space.

Modern VAEs can produce high-quality samples (figure 17.12d), but only by using hierarchical priors and specialized network architecture and regularization techniques. Diffusion models (chapter 18) can be viewed as VAEs with hierarchical priors. These also create very high-quality samples.

17.8.3 Resynthesis

VAEs can also be used to modify real data. A data point $\mathbf{x}$ can be projected into the latent space by either (i) taking the mean of the distribution predicted by the encoder or (ii) by using an optimization procedure to find the latent variable $\mathbf{z}$ that maximizes the posterior probability, which Bayes' rule tells us is proportional to $Pr(\mathbf{x}|\mathbf{z})Pr(\mathbf{z})$.

In figure 17.13, multiple images labeled as "neutral" or "smiling" are projected into latent space. The vector representing this change is estimated by taking the difference in latent space between the means of these two groups. A second vector is estimated to represent "mouth closed" versus "mouth open."

Now the image of interest is projected into the latent space, and then the representation is modified by adding or subtracting these vectors. To generate intermediate images, *spherical linear interpolation* or *Slerp* is used rather than linear interpolation. In 3D, this would be the difference between interpolating along the surface of a sphere versus digging a straight tunnel through its body.

Problem 17.6

The process of encoding (and possibly modifying) input data before decoding again is known as *resynthesis*. This can also be done with GANs and normalizing flows. However, in GANs, there is no encoder, so a separate procedure must be used to find the latent variable that corresponds to the observed data.

17.8.4 Disentanglement

In the resynthesis example above, the directions in space representing interpretable properties had to be estimated using labeled training data. Other work attempts to improve the characteristics of the latent space so that its coordinate directions correspond to real-world properties. When each dimension represents an independent real-world factor, the latent space is described as *disentangled*. For example, when modeling face images, we might hope to uncover head pose or hair color as independent factors.

Methods to encourage disentanglement typically add regularization terms to the loss function based on either (i) the posterior $q(\mathbf{z}|\mathbf{x}, \boldsymbol{\theta})$ over the latent variables $\mathbf{z}$, or (ii) the aggregated posterior $q(\mathbf{z}|\boldsymbol{\theta}) = (1/I)\sum_i q(\mathbf{z}|\mathbf{x}_i, \boldsymbol{\theta})$:

$$L_{\text{new}} = -\text{ELBO}[\boldsymbol{\theta}, \boldsymbol{\phi}] + \lambda_1 \mathbb{E}_{Pr(\mathbf{x})}\Big[\text{r}_1\big[q(\mathbf{z}|\mathbf{x}, \boldsymbol{\theta})\big]\Big] + \lambda_2 \text{r}_2\big[q(\mathbf{z}|\boldsymbol{\theta})\big]. \tag{17.29}$$

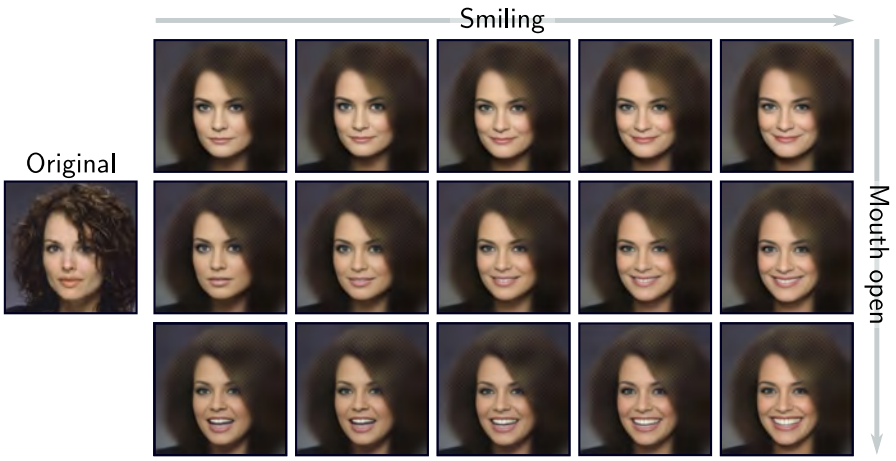

Figure 17.13 Resynthesis. The original image on the left is projected into the latent space using the encoder, and the mean of the predicted Gaussian is chosen to represent the image. The center-left image in the grid is the reconstruction of the input. The other images are reconstructions after manipulating the latent space in directions representing smiling/neutral (horizontal) and mouth open/closed (vertical). Adapted from White (2016).

Here the regularization term $\mathrm{r}_1[\bullet]$ is a function of the posterior and is weighted by λ_1. The term $\mathrm{r}_2[\bullet]$ is a function of the aggregated posterior and is weighted by λ_2.

For example, the *beta VAE* upweights the second term in the ELBO (equation 17.18):

$$\mathrm{ELBO}[\boldsymbol{\theta}, \boldsymbol{\phi}] \approx \log\big[Pr(\mathbf{x}|\mathbf{z}^*, \boldsymbol{\phi})\big] - \beta \cdot \mathrm{D}_{KL}\Big[q(\mathbf{z}|\mathbf{x}, \boldsymbol{\theta})\Big|\Big|Pr(\mathbf{z})\Big], \tag{17.30}$$

where $\beta > 1$ determines how much more the deviation from the prior $Pr(\mathbf{z})$ is weighted relative to the reconstruction error. Since the prior is usually a multivariate normal with a spherical covariance matrix, its dimensions are independent. Hence, up-weighting this term encourages the posterior distributions to be less correlated. Another variant is the total correlation VAE, which adds a term to decrease the total correlation between variables in the latent space (figure 17.14) and maximizes the mutual information between a small subset of the latent variables and the observations.

17.9 Summary

The VAE is an architecture that helps to learn a nonlinear latent variable model over $\mathbf{x}$. This model can generate new examples by sampling from the latent variable, passing the result through a deep network, and then adding independent Gaussian noise.

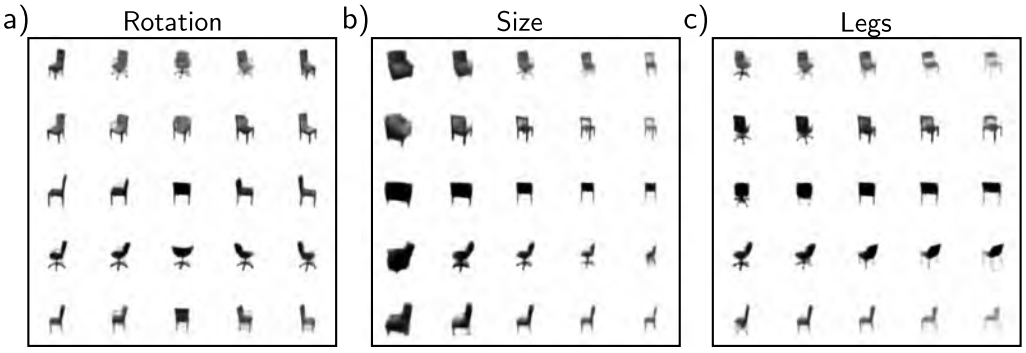

Figure 17.14 Disentanglement in the total correlation VAE. The VAE model is modified so that the loss function encourages the total correlation of the latent variables to be minimized and hence encourages disentanglement. When trained on a dataset of images of chairs, several of the latent dimensions have clear real-world interpretations, including a) rotation, b) overall size, and c) legs (swivel chair versus normal). In each case, the central column depicts samples from the model, and as we move left to right, we are subtracting or adding a coordinate vector in latent space. Adapted from Chen et al. (2018d).

It is not possible to compute the likelihood of a data point in closed form, and this poses problems for training with maximum likelihood. However, we can define a lower bound on the likelihood and maximize this bound. Unfortunately, for the bound to be tight, we need to compute the posterior probability of the latent variable given the observed data, which is also intractable. The solution is to make a variational approximation. This is a simpler distribution (usually a Gaussian) that approximates the posterior and whose parameters are computed by a second encoder network.

To create high-quality samples from the VAE, it seems to be necessary to model the latent space with more sophisticated probability distributions than the Gaussian prior and posterior. One option is to use hierarchical priors (in which one latent variable generates another). The next chapter discusses diffusion models, which produce very high-quality examples and can be viewed as hierarchical VAEs.

Notes

The VAE was originally introduced by Kingma & Welling (2014). A comprehensive introduction to variational autoencoders can be found in Kingma et al. (2019).

Applications: The VAE and variants thereof have been applied to images (Kingma & Welling, 2014; Gregor et al., 2016; Gulrajani et al., 2016; Akuzawa et al., 2018), speech (Hsu et al., 2017b), text (Bowman et al., 2015; Hu et al., 2017; Xu et al., 2020), molecules (Gómez-Bombarelli et al.,

2018; Sultan et al., 2018), graphs (Kipf & Welling, 2016; Simonovsky & Komodakis, 2018), robotics (Hernández et al., 2018; Inoue et al., 2018; Park et al., 2018), reinforcement learning (Heess et al., 2015; Van Hoof et al., 2016), 3D scenes (Eslami et al., 2016, 2018; Rezende Jimenez et al., 2016), and handwriting (Chung et al., 2015).

Applications include resynthesis and interpolation (White, 2016; Bowman et al., 2015), collaborative filtering (Liang et al., 2018), and compression (Gregor et al., 2016). Gómez-Bombarelli et al. (2018) use the VAE to construct a continuous representation of chemical structures that can then be optimized for desirable properties. Ravanbakhsh et al. (2017) simulate astronomical observations for calibrating measurements.

Relation to other models: The autoencoder (Rumelhart et al., 1985; Hinton & Salakhutdinov, 2006) passes data through an encoder to a bottleneck layer and then reconstructs it using a decoder. The bottleneck is similar to latent variables in the VAE, but the motivation differs. Here, the goal is not to learn a probability distribution but to create a low-dimensional representation that captures the essence of the data. Autoencoders also have various applications, including denoising (Vincent et al., 2008) and anomaly detection (Zong et al., 2018).

If the encoder and decoder are linear transformations, the autoencoder is just principal component analysis (PCA). Hence, the nonlinear autoencoder is a generalization of PCA. There are also probabilistic forms of PCA. Probabilistic PCA (Tipping & Bishop, 1999) adds spherical Gaussian noise to the reconstruction to create a probability model, and factor analysis adds diagonal Gaussian noise (see Rubin & Thayer, 1982). If we make the encoder and decoder of these probabilistic variants nonlinear, we return to the variational autoencoder.

Architectural variations: The conditional VAE (Sohn et al., 2015) passes class information c into both the encoder and decoder. The result is that the latent space does not need to encode the class information. For example, when MNIST data are conditioned on the digit label, the latent variables might encode the orientation and width of the digit rather than the digit category itself. Sønderby et al. (2016a) introduced ladder variational autoencoders, which recursively correct the generative distribution with a data-dependent approximate likelihood term.

Modifying likelihood: Other work investigates more sophisticated likelihood models $Pr(\mathbf{x}|\mathbf{z})$. The PixelVAE (Gulrajani et al., 2016) used an autoregressive model over the output variables. Dorta et al. (2018) modeled the covariance of the decoder output as well as the mean. Lamb et al. (2016) improved the quality of reconstruction by adding extra regularization terms that encourage the reconstruction to be similar to the original image in the space of activations of a layer of an image classification model. This model encourages semantic information to be retained and was used to generate the results in figure 17.13. Larsen et al. (2016) use an adversarial loss for reconstruction, which also improves results.

Latent space, prior, and posterior: Many different forms for the variational approximation to the posterior have been investigated, including normalizing flows (Rezende & Mohamed, 2015; Kingma et al., 2016), directed graphical models (Maaløe et al., 2016), undirected models (Vahdat et al., 2020), and recursive models for temporal data (Gregor et al., 2016, 2019).

Other authors have investigated using a discrete latent space (Van Den Oord et al., 2017; Razavi et al., 2019b; Rolfe, 2017; Vahdat et al., 2018a,b) For example, Razavi et al. (2019b) use a vector quantized latent space and model the prior with an autoregressive model (equation 12.15). This is slow to sample from but can describe very complex distributions.

Jiang et al. (2016) use a mixture of Gaussians for the posterior, allowing clustering. This is a hierarchical latent variable model that adds a discrete latent variable to improve the flexibility of the posterior. Other authors (Salimans et al., 2015; Ranganath et al., 2016; Maaløe et al., 2016; Vahdat & Kautz, 2020) have experimented with hierarchical models that use continuous variables. These have a close connection with diffusion models (chapter 18).

Combination with other models: Gulrajani et al. (2016) combined VAEs with an autoregressive model to produce more realistic images. Chung et al. (2015) combine the VAE with recurrent neural networks to model time-varying measurements.

As discussed above, adversarial losses have been used to inform the likelihood term directly. However, other models have combined ideas from generative adversarial networks (GANs) with VAEs in different ways. Makhzani et al. (2015) use an adversarial loss in the latent space; the idea is that the discriminator will ensure that the aggregated posterior distribution $q(\mathbf{z})$ is indistinguishable from the prior distribution $Pr(\mathbf{z})$. Tolstikhin et al. (2018) generalize this to a broader family of distances between the prior and aggregated posterior. Dumoulin et al. (2017) introduced adversarially learned inference which uses an adversarial loss to distinguish two pairs of latent/observed data points. In one case, the latent variable is drawn from the latent posterior distribution and, in the other, from the prior. Other hybrids of VAEs and GANs were proposed by Larsen et al. (2016), Brock et al. (2016), and Hsu et al. (2017a).

Posterior collapse: One potential problem in training is *posterior collapse*, in which the encoder always predicts the prior distribution. This was identified by Bowman et al. (2015) and can be mitigated by gradually increasing the term that encourages the KL distance between the posterior and the prior to be small during training. Several other methods have been proposed to prevent posterior collapse (Razavi et al., 2019a; Lucas et al., 2019b,a), and this is also part of the motivation for using a discrete latent space (Van Den Oord et al., 2017).

Blurry reconstructions: Zhao et al. (2017c) provide evidence that the blurry reconstructions are partly due to Gaussian noise and also because of the sub-optimal posterior distributions induced by the variational approximation. It is perhaps not coincidental that some of the best synthesis results have come from using a discrete latent space modeled by a sophisticated autoregressive model (Razavi et al., 2019b) or from using hierarchical latent spaces (Vahdat & Kautz, 2020; see figure 17.12d). Figure 17.12a-c used a VAE that was trained on the CELEBA database (Liu et al., 2015). Figure 17.12d uses a hierarchical VAE that was trained on the CELEBA HQ dataset (Karras et al., 2018).

Other problems: Chen et al. (2017) noted that when more complex likelihood terms are used, such as the PixelCNN (Van den Oord et al., 2016c), the output can cease to depend on the latent variables at all. They term this the *information preference* problem. This was addressed by Zhao et al. (2017b) in the InfoVAE, which added an extra term that maximized the mutual information between the latent and observed distributions.

Another problem with the VAE is that there can be "holes" in the latent space that do not correspond to any realistic sample. Xu et al. (2020) introduce the constrained posterior VAE, which helps prevent these vacant regions in latent space by adding a regularization term. This allows for better interpolation from real samples.

Disentangling latent representation: Methods to "disentangle" the latent representation include the beta VAE (Higgins et al., 2017) and others (e.g., Kim & Mnih, 2018; Kumar et al.,

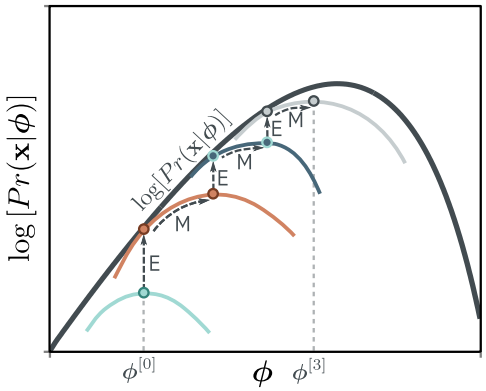

Figure 17.15 Expectation maximization (EM) algorithm. The EM algorithm alternately adjusts the auxiliary parameters $\boldsymbol{\theta}$ (moves between colored curves) and model parameters $\boldsymbol{\phi}$ (moves along colored curves) until the a maximum is reached. These adjustments are known as the E-step and the M-step, respectively. Because the E-Step uses the posterior distribution $Pr(h|\mathbf{x}, \boldsymbol{\phi})$ for $q(h|\mathbf{x}, \boldsymbol{\theta})$, the bound is tight, and the colored curve touches the black likelihood curve after each E-Step.

2018). Chen et al. (2018d) further decomposed the ELBO to show the existence of a term measuring the total correlation between the latent variables (i.e., the distance between the aggregate posterior and the product of its marginals). They use this to motivate the total correlation VAE, which attempts to minimize this quantity. The Factor VAE (Kim & Mnih, 2018) uses a different approach to minimize the total correlation. Mathieu et al. (2019) discuss the factors that are important in disentangling representations.

Reparameterization trick: Consider computing an expectation of some function, where the probability distribution with which the expectation is taken depends on some parameters. The reparameterization trick computes the derivative of this expectation with respect to these parameters. This chapter introduced this as a method to differentiate through the sampling procedure approximating the expectation; there are alternative approaches (see problem 17.5), but the reparameterization trick gives an estimator that (usually) has low variance. This issue is discussed in Rezende et al. (2014), Kingma et al. (2015), and Roeder et al. (2017).

Lower bound and the EM algorithm: VAE training is based on optimizing the evidence lower bound (sometimes also referred to as the ELBO, variational lower bound, or negative variational free energy). Hoffman & Johnson (2016) and Lücke et al. (2020) re-express this lower bound in several ways that elucidate its properties. Other work has aimed to make this bound tighter (Burda et al., 2016; Li & Turner, 2016; Bornschein et al., 2016; Masrani et al., 2019). For example, Burda et al. (2016) use a modified bound based on using multiple importance-weighted samples from the approximate posterior to form the objective function.

The ELBO is tight when the distribution $q(\mathbf{z}|\boldsymbol{\theta})$ matches the posterior $Pr(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})$. This is the basis of the *expectation maximization* (*EM*) algorithm (Dempster et al., 1977). Here, we alternately (i) choose $\boldsymbol{\theta}$ so that $q(\mathbf{z}|\boldsymbol{\theta})$ equals the posterior $Pr(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})$ and (ii) change $\boldsymbol{\phi}$ to maximize the lower bound (figure 17.15). This is viable for models like the mixture of Gaussians, where we can compute the posterior distribution in closed form. Unfortunately, this is not the case for the nonlinear latent variable model, so this method cannot be used.

Problem 17.7

Problems

Problem 17.1 How many parameters are needed to create a 1D mixture of Gaussians with $n = 5$