Chapter 16

Normalizing flows

Chapters 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, it isn't straightforward to assess the probability that a new example belongs to the same dataset.

In this chapter, we describe *normalizing flows*, which 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 data 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, then

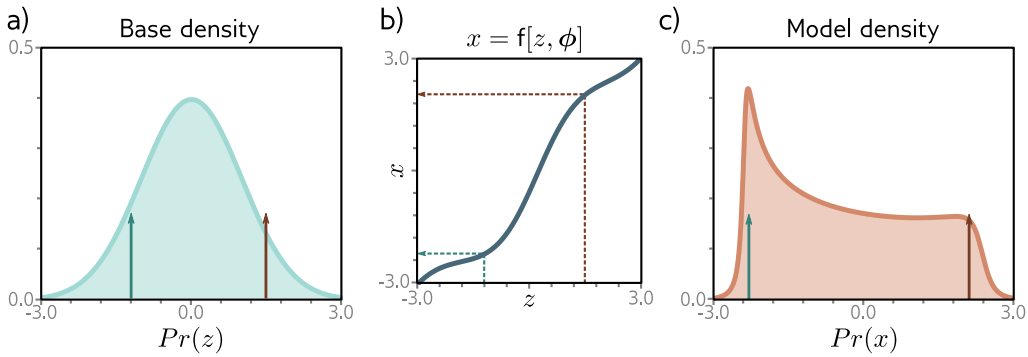

Figure 16.1 Transforming probability distributions. a) The base density is a standard normal defined on a latent variable z. b) This variable is transformed by a function $x = \mathrm{f}[z, \boldsymbol{\phi}]$ to a new variable x, which c) has a new distribution. To sample from this model, we draw values 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).

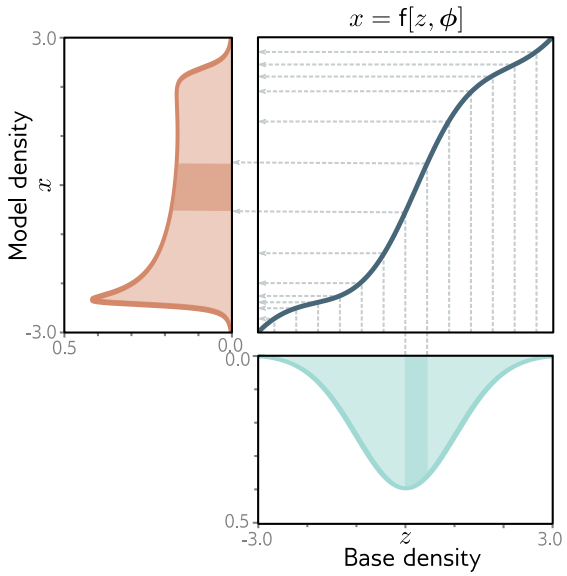

Figure 16.2 Transforming distributions. The base density (cyan, bottom) passes through a function (gray curve, top right) to create the model density (orange, left). Consider dividing the base density into equal intervals (gray vertical lines). The probability mass between adjacent lines must remain the same after transformation. The cyan-shaded region passes through a part of the function where the gradient is larger than one and 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.

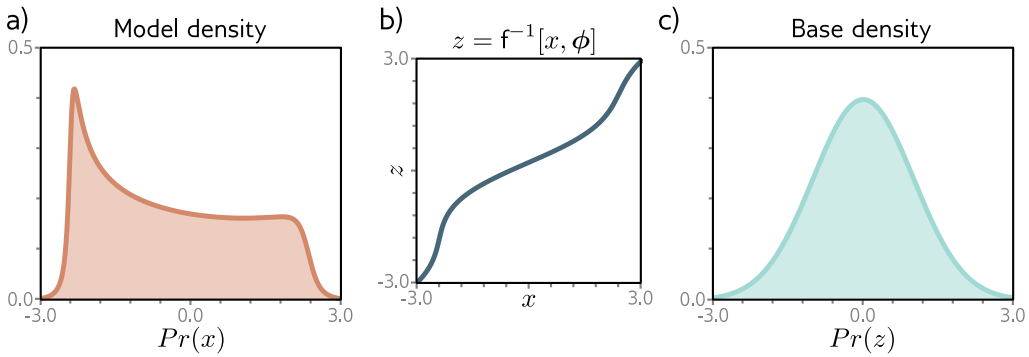

Figure 16.3 Inverse mapping (normalizing direction). If the function is invertible then it's possible to transform the model density back to the original base density. The probability of a point x under the model density depends in part on the probability of the equivalent point z under the base density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

it stretches the function. If a small change to the input causes a smaller change in the output, then 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.

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:

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

where we have 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 base density $Pr(\mathbf{z})$, where $\mathbf{z} \in \mathbb{R}^D$ and $\mathbf{f}[\mathbf{z}, \boldsymbol{\phi}]$ 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 terms $\partial \text{f}_i[\mathbf{z}, \boldsymbol{\phi}]/\partial z_j$ 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 probability of the latent variable under the base density.

Appendix C.5.5 Determinant

Appendix C.6.1 Jacobian

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:

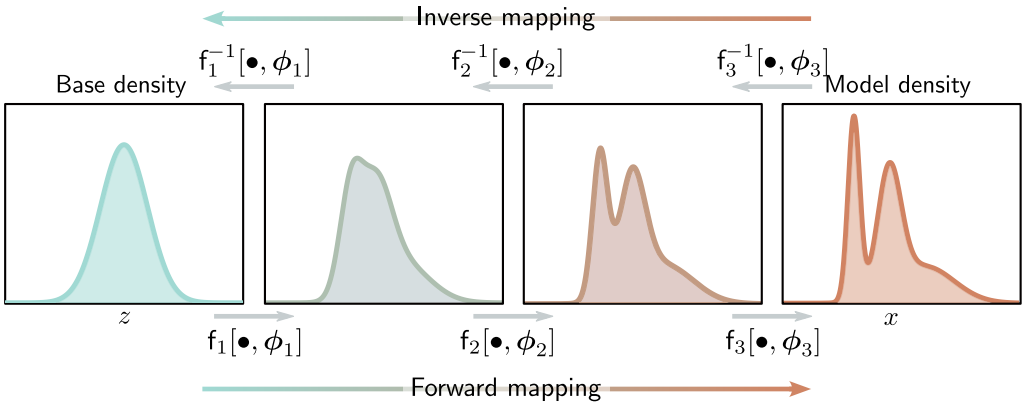

Figure 16.4 Forward and inverse mappings for a deep neural network. The base density (left) is gradually transformed by the network layers $f_1[\bullet, \phi_1], f_2[\bullet, \phi_2], \ldots$ to create the model density. Each layer is invertible and we can equivalently think of the inverse of the layers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

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

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. 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:

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

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, a 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, then 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 C.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 then 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 C.7 Big O notation

If the matrix $\boldsymbol{\Omega}$ takes a special form, then inversion and computation of the determinant becomes 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 C.5.7 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 diagonal 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:

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

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 and 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

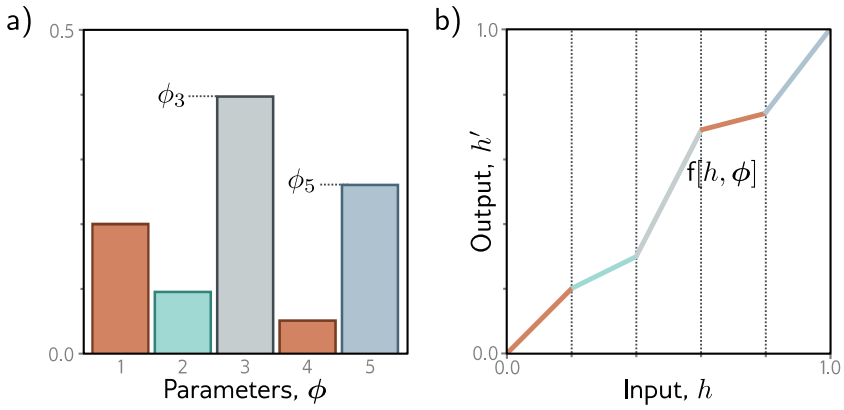

Figure 16.5 Piecewise-linear mapping. An invertible piecewise-linear mapping $h' = \text{f}[h, \boldsymbol{\phi}]$ can be created by dividing the input domain $h \in [0, 1]$ into K equally sized regions (here $K = 5$). Each region has a slope with parameter $\boldsymbol{\phi}_k$. a) If these parameters 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:

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

where the parameters $\phi_1, \phi_2, \ldots \phi_K$ are positive and sum to one, and $b = \lfloor Kh_d \rfloor$ is the index of the bin that contains h_d. 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_d 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 that the function is monotonic and hence invertible.

Problem 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, but, 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 $\mathbf{h} = [\mathbf{h}_1, \mathbf{h}_2]^T$ and define the flow $\mathbf{f}[\mathbf{h}, \boldsymbol{\phi}]$ as:

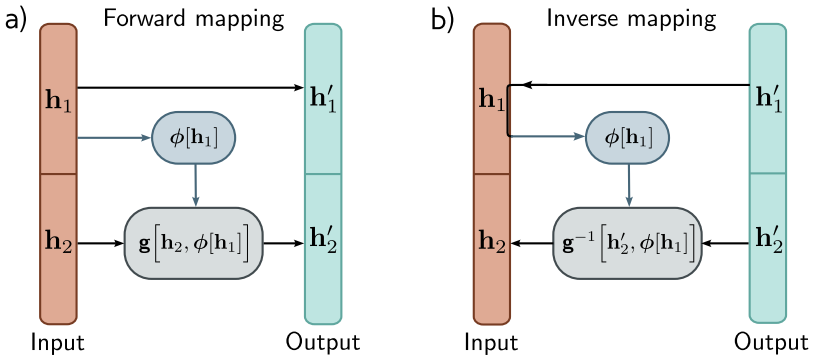

Figure 16.6 Coupling flows. a) The input (orange vector) is divided into $\mathbf{h}_1$ and $\mathbf{h}_2$. The first part $\mathbf{h}'_1$ of the output (cyan vector) is a copy of $\mathbf{h}_1$. The output $\mathbf{h}'_2$ is created by applying an invertible transformation $\mathbf{g}[\bullet, \boldsymbol{\phi}]$ 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, then 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 C.5.8 Permutation matrix

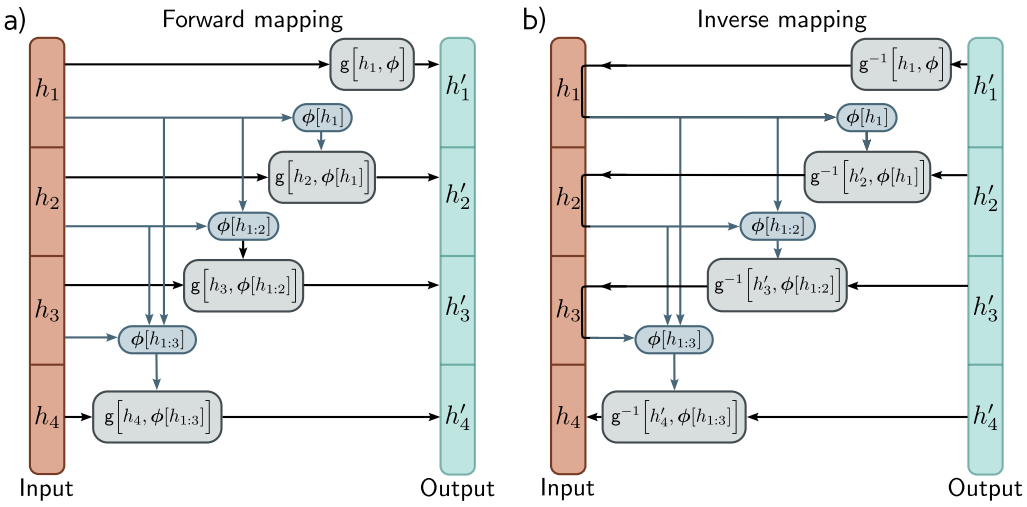

Figure 16.7 Autoregressive flows. The input $\mathbf{h}$ (orange column) and output $\mathbf{h}'$ (cyan column) are split into their constituent dimensions (here four dimensions). a) Output h_1' is an invertible transformation of input h_1. Output h_2' is an invertible function of input h_2 where the 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_2, 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 neural networks. If the transformer and conditioner are sufficiently flexible, then 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.

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 formuation 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:

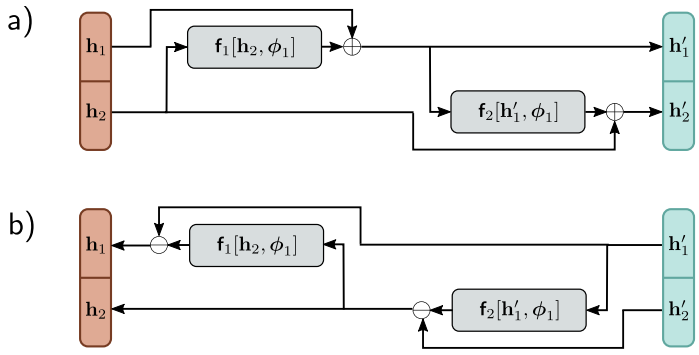

Figure 16.8 Residual flows. a) An invertible function is computed by splitting the input into $\mathbf{h}_1$ and $\mathbf{h}_2$ and creating two residual layers. In the first, $\mathbf{h}_2$ is processed and $\mathbf{h}_1$ is added. In the second, the result is processed and $\mathbf{h}_2$ is added.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

$$\begin{aligned}\mathbf{h}'_1 &= \mathbf{h}_1 + \mathbf{f}_1[\mathbf{h}_2, \phi_1] \\ \mathbf{h}'_2 &= \mathbf{h}_2 + \mathbf{f}_2[\mathbf{h}'_1, \phi_2],\end{aligned} \tag{16.18}$$

where $\mathbf{f}_1[\bullet, \phi_1]$ and $\mathbf{f}_2[\bullet, \phi_2]$ 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:

$$\begin{aligned}\mathbf{h}_2 &= \mathbf{f}_2[\mathbf{h}'_1, \phi_1] - \mathbf{h}'_2 \\ \mathbf{h}_1 &= \mathbf{f}_1[\mathbf{h}_2, \phi_2] - \mathbf{h}'_1.\end{aligned} \tag{16.19}$$

As for coupling flows, the division into blocks restricts the family of transformations that can be represented, and 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 $\mathbf{f}_1[\bullet, \phi_1]$ and $\mathbf{f}_2[\bullet, \phi_2]$, 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, there is no need to store the activations at each layer in the forward pass.

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:

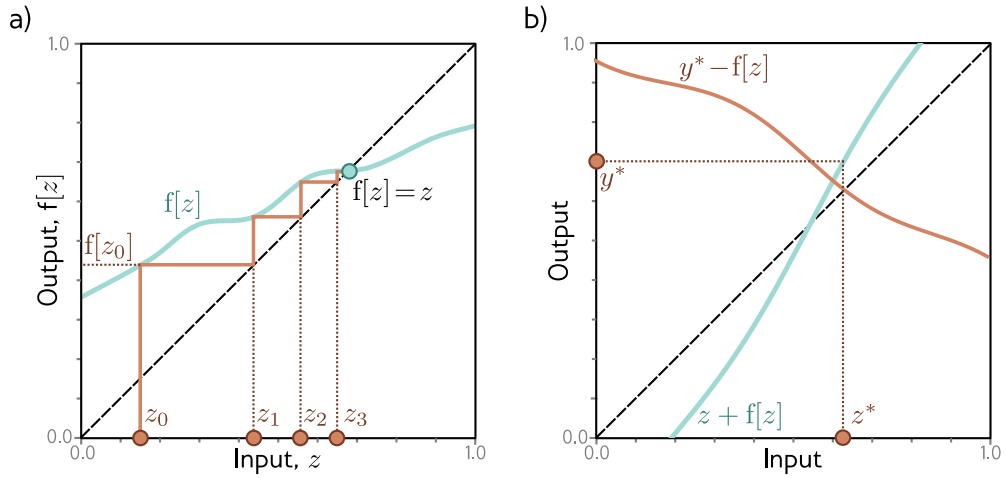

Figure 16.9 Contraction mappings. If a function has an absolute slope of less than one everywhere, then iterating the function converges to a fixed point $\mathrm{f}[z] = z$. a) Starting at z_0, we evaluate $z_1 = \mathrm{f}[z_0]$. We then pass z_1 back into the function and iterate. Eventually, the 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, and so the current position must get closer to the fixed point.

Problem 16.11

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 y_0 and iterating $z = y^* - \mathrm{f}[z]$. 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 C.2 Lipschitz constant

Appendix C.5.4 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 by 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.

This series is truncated and the constituent terms can then be estimated 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 $Pr(\boldsymbol{\epsilon})$:

Appendix C.5.5 Trace

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

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

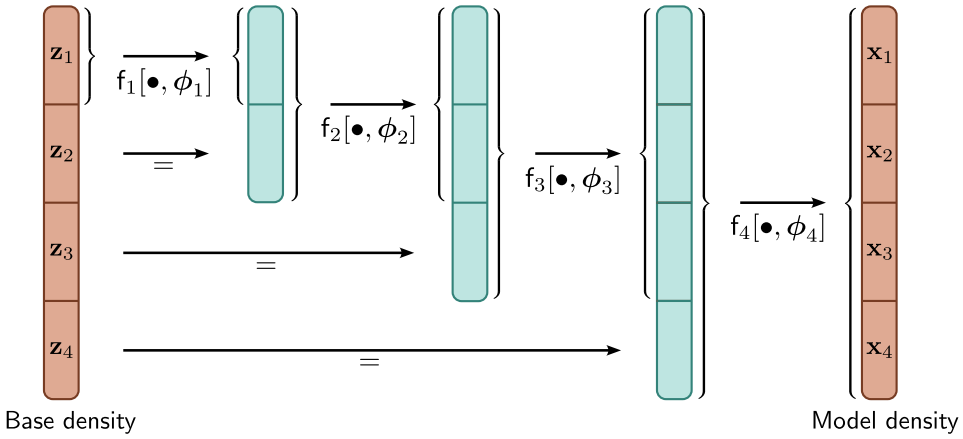

Figure 16.10 Multiscale flows. The latent space $\mathbf{z}$ must be the same size as the model density in normalizing flows. However, it can be partitioned into several components, and these can be gradually introduced at different layers. This makes both density estimation and sampling faster.

some point we have to introduce all of these variables, but it is inefficient to pass them through the entire network. The 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 is 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

In this section, we consider three applications of normalizing flows. First, we consider modeling probability densities. Second, we consider the GLOW model, which can be used to synthesize images. Finally, we discuss how normalizing flows can be used 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

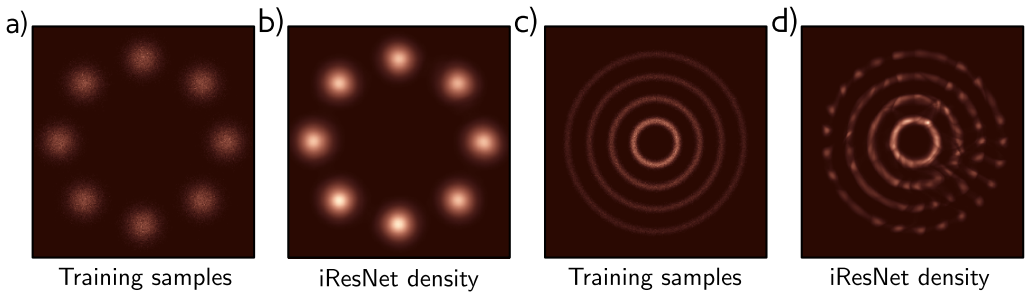

Figure 16.11 Modeling densities. a) Toy 2D data samples. b) Modeled density using iResNet. c-d) Second example. Adapted from Behrmann et al. (2019)

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 do not 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 turning each 2×2 patch into a single position with four times as many channels. This is a multi-scale flow in which a portion of the channels are periodically removed and directly become part of the latent vector $\mathbf{z}$. Since images are discrete distributions (due to the quantization of RGB values), noise is added to the inputs to prevent the likelihood increasing without bound during learning. 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

[2]The lower bound for diffusion models can actually be better than the exact computation in normalizing flows, but computation is much slower (see chapter 18).

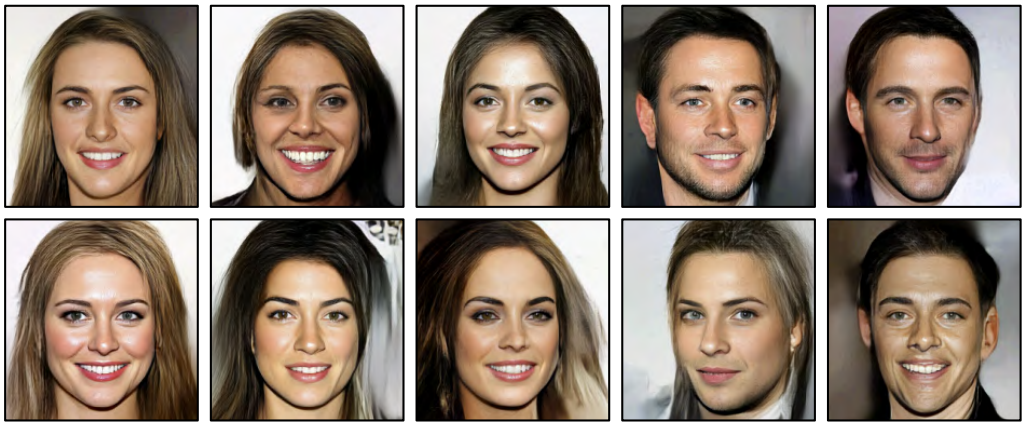

Figure 16.12 Samples from GLOW trained on the CelebA HQ dataset (Karras et al., 2018). The samples are of reasonable quality, although GANs and diffusion models produce superior results. Adapted from Kingma & Dhariwal (2018).

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).

density, rather than from the tails. This is a similar to the truncation trick in GANs (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 $\mathbf{x}_i$ 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):

$$\boldsymbol{\phi} = \underset{\boldsymbol{\phi}}{\text{argmin}} \left[\text{KL} \left[\sum_{i=1}^{I} Pr(\mathbf{x}_i, \boldsymbol{\phi}) \middle|\middle| \frac{1}{I} \sum_{i=1}^{I} q(\mathbf{x}_i) \right] \right]. \tag{16.25}$$

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):

$$\boldsymbol{\phi} = \underset{\boldsymbol{\phi}}{\text{argmin}} \left[\text{KL} \left[\frac{1}{I} \sum_{i=1}^{I} \delta[\mathbf{x} - \mathbf{x}_i] \middle|\middle| Pr(\mathbf{x}_i, \boldsymbol{\phi}) \right] \right]. \tag{16.26}$$

A similar trick is used to exploit normalizing flows to model the posterior in variational auto-encoders (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.

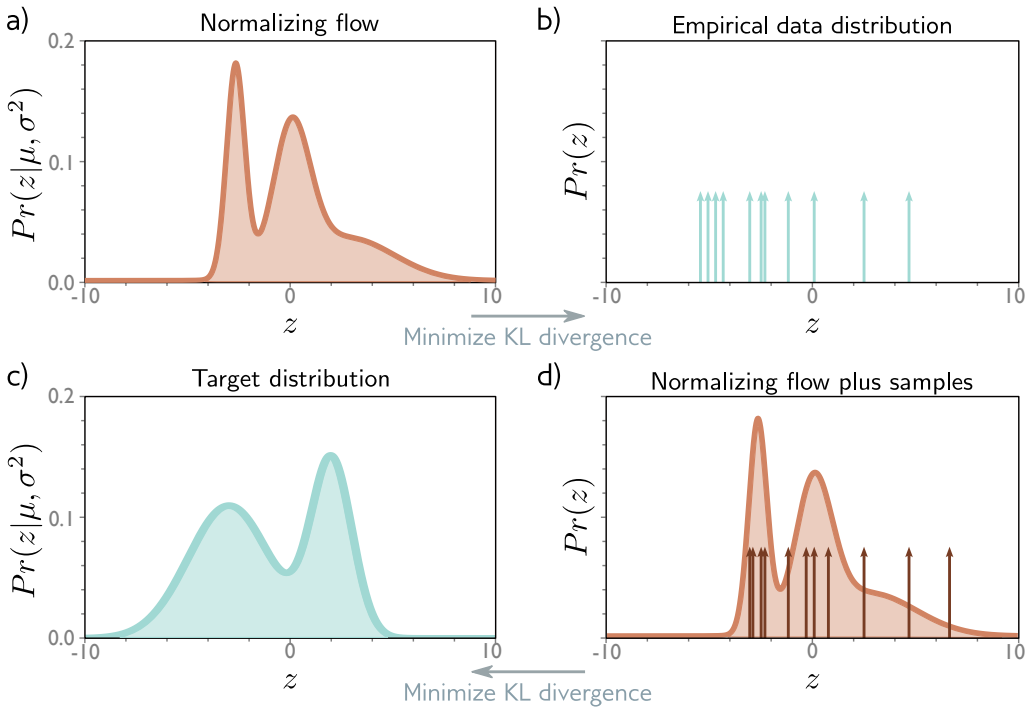

Figure 16.14 Approximating density models. a) Normally we modify the flow model to minimize the KL divergence to b) the empirical data distribution. This is equivalent to maximum likelihood fitting (see section 5.7). c) However, in some cases, we have a target distribution model where we can evaluate the density (but perhaps not draw samples efficiently), and wish to train a normalizing flow to approximate it. d) Here we minimize the reverse KL divergence between samples from the normalizing flows model and the target distribution.

It's also important that the Jacobian can be estimated efficiently to evaluate the likelihood. This is particularly important as the likelihood must be evaluated repeatedly to learn the density. However, invertible layers 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 flows, autoregressive flows, 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 and 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 flow 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 systematically 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 pre-determined 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 but this is effectively a full linear transformation that is 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 auto-regressive 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. Unfortunately, it is hard to train a network with the constraint that the weights are positive and 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 multiscale 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 and 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 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 was less than one. This led to iResNet, in which 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 less 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 equation 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 variance $\boldsymbol{\Sigma}$ that is transformed as $\mathbf{x} = \mathbf{A}\mathbf{z} + \mathbf{b}$. Show that $\mathbb{E}[\mathbf{x}] = \mathbf{A}\boldsymbol{\mu} + \mathbf{b}$ and $\mathrm{Var}[\mathbf{x}] = \mathbf{A}\boldsymbol{\Sigma}\mathbf{A}^T$.

Problem 16.6 Prove that if $\mathbf{x} = \mathbf{f}[\mathbf{z}] = \mathbf{A}\mathbf{z} + \mathbf{b}$ and $Pr(\mathbf{z}) = \mathrm{Norm}_{\mathbf{z}}[\boldsymbol{\mu}, \boldsymbol{\Sigma}]$, then $Pr(\mathbf{x}) = \mathrm{Norm}_{\mathbf{x}}[\mathbf{A}\boldsymbol{\mu} + \mathbf{b}, \mathbf{A}\boldsymbol{\Sigma}\mathbf{A}^T]$ using the relation:

$$Pr(\mathbf{x}) = \mathrm{Pr}(\mathbf{z}) \cdot \left| \frac{\partial \mathbf{f}^{-1}[\mathbf{x}]}{\partial \mathbf{x}} \right|, \tag{16.30}$$

where $\mathbf{z} = \mathbf{f}^{-1}[\mathbf{x}, \boldsymbol{\phi}]$ and the result from problem 16.5.

Problem 16.7 The Leaky ReLU is defined as:

$$\mathrm{LReLU}[z] = \begin{cases} 0.1z & z < 0 \\ z & z \geq 0 \end{cases}. \tag{16.31}$$

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:

$$\mathbf{f}[\mathbf{z}] = \Big[\mathrm{LReLU}[z_1, \boldsymbol{\phi}], \mathrm{LReLU}[z_2, \boldsymbol{\phi}], \ldots \mathrm{LReLU}[z_D, \boldsymbol{\phi}]\Big]^T. \tag{16.32}$$

Problem 16.8 Find the inverse $\mathrm{f}^{-1}[h', \boldsymbol{\phi}]$ of the piecewise linear function $\mathrm{f}[h, \boldsymbol{\phi}]$ defined in equation 16.12, for the domain $h' \in [0, 1]$. Consider applying this nonlinear function elementwise to an input $\mathbf{h} = [h_1, h_2, \ldots h_D]^T$ so that $\mathbf{f}[\mathbf{h}] = [\mathrm{f}[h_1, \boldsymbol{\phi}], \mathrm{f}[h_2, \boldsymbol{\phi}], \ldots, \mathrm{f}[h_D, \boldsymbol{\phi}]]$ What is the Jacobian $\partial \mathrm{f}[\mathbf{h}]/\partial \mathbf{h}$? What is the determinant of the Jacobian?

Problem 16.9 Consider constructing an element-wise flow based on a conical combination of sinusoids in equally spaced bins:

$$\mathbf{h}' = \mathrm{f}[h, \boldsymbol{\phi}] = \sin\left[\frac{Kh - b \times \pi}{2}\right] \phi_b + \sum_{k=1}^{b} \phi_k, \tag{16.33}$$

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 $\mathrm{f}[h, \boldsymbol{\phi}]$. Draw the inverse function $\mathrm{f}^{-1}[h', \boldsymbol{\phi}]$.

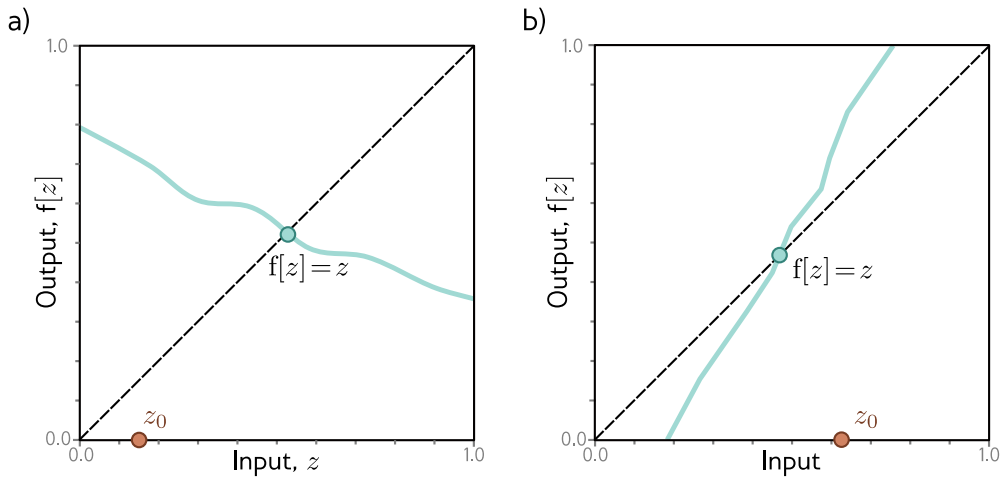

Figure 16.15 Functions for problem 16.11.

Problem 16.10 Draw a picture of the structure of the Jacobian 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 For each of the functions in figure 16.15, describe the behavior if we perform a fixed point iteration $z \leftarrow \text{f}[z]$ starting at point z_0. For each case, indicate whether the iterations will converge to the fixed point $\text{f}[z] = z$.

Problem 16.12 Write out the expression for the KL divergence in equation 16.25. Why does it not matter that we can only evaluate the posterior probability up to a scaling factor κ? Does the network have to be invertible to minimize this loss function? Explain your reasoning.