# 18 Normalizing Flows

*Chapter 17*

We have seen how generative adversarial networks (GANs) extend the framework of linear latent-variable models by using deep neural networks to represent highly flexible and learnable nonlinear transformations from the latent space to the data space. However, the likelihood function is generally either intractable, because the network function cannot be inverted, or may not even be defined if the latent space has a lower dimensionality than the data space. In GANs, a second, discriminative network was therefore introduced to facilitate adversarial training.

*Section 16.4.4*

Here we discuss the second of our four approaches to training nonlinear latent variable models that involves restricting the form of the neural network model such that the likelihood function can be evaluated without approximation while still ensuring that sampling from the trained model is straightforward. Suppose we define a distribution $p_{\mathbf{z}}(\mathbf{z})$, sometimes also called a *base distribution*, over a latent variable $\mathbf{z}$ along with a nonlinear function $\mathbf{x} = \mathbf{f}(\mathbf{z}, \mathbf{w})$, given by a deep neural network, that


C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_18

transforms the latent space into the data space. Assuming $p_{\mathbf{z}}(\mathbf{z})$ is a simple distribution such as a Gaussian, sampling from such a model is easy as each latent sample $\mathbf{z}^\star \sim p_{\mathbf{z}}(\mathbf{z})$ is simply passed through the neural network to generate a corresponding data sample $\mathbf{x}^\star = \mathbf{f}(\mathbf{z}^\star, \mathbf{w})$.

To calculate the likelihood function for this model, we need the data-space distribution, which depends on the *inverse* of the neural network function. We write this as $\mathbf{z} = \mathbf{g}(\mathbf{x}, \mathbf{w})$, and it satisfies $\mathbf{z} = \mathbf{g}(\mathbf{f}(\mathbf{z}, \mathbf{w}), \mathbf{w})$. This requires that, for every value of $\mathbf{w}$, the functions $\mathbf{f}(\mathbf{z}, \mathbf{w})$ and $\mathbf{g}(\mathbf{x}, \mathbf{w})$ are invertible, also called *bijective*, so that each value of $\mathbf{x}$ corresponds to a unique value of $\mathbf{z}$ and vice versa. We can then use the change of variables formula to calculate the data density:

*Section 2.4*

$$p_{\mathbf{x}}(\mathbf{x}|\mathbf{w}) = p_{\mathbf{z}}(\mathbf{g}(\mathbf{x}, \mathbf{w}))\,|\det \mathbf{J}(\mathbf{x})| \tag{18.1}$$

where $\mathbf{J}(\mathbf{x})$ is the Jacobian matrix of partial derivatives whose elements are given by

$$J_{ij}(\mathbf{x}) = \frac{\partial g_i(\mathbf{x}, \mathbf{w})}{\partial x_j} \tag{18.2}$$

and $|\cdot|$ denotes the modulus or absolute value. We will continue to refer to $\mathbf{z}$ as a 'latent' variable even though the deterministic mapping means that any given data value $\mathbf{x}$ corresponds to a unique value of $\mathbf{z}$ whose value is therefore no longer uncertain.

The mapping function $\mathbf{f}(\mathbf{z}, \mathbf{w})$ will be defined in terms of a special form of neural network, whose structure we will discuss shortly. One consequence of requiring an invertible mapping is that the dimensionality of the latent space must be the same as that of the data space, which can lead to large models for high-dimensional data such as images. Also, in general, the cost of evaluating the determinant of a $D \times D$ matrix is $\mathcal{O}(D^3)$, so we will seek to impose some further restrictions on the model in order that evaluation of the Jacobian matrix determinant is more efficient.

If we consider a training set $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ of independent data points, the log likelihood function is given from (18.1) by

$$\ln p(\mathcal{D}|\mathbf{w}) = \sum_{n=1}^{N} \ln p_{\mathbf{x}}(\mathbf{x}_n|\mathbf{w}) \tag{18.3}$$

$$= \sum_{n=1}^{N} \Big\{ \ln p_{\mathbf{z}}(\mathbf{g}(\mathbf{x}_n, \mathbf{w})) + \ln |\det \mathbf{J}(\mathbf{x}_n)| \Big\} \tag{18.4}$$

and our goal is to use the likelihood function to train the neural network. To be able to model a wide range of distributions, we want the transformation function $\mathbf{x} = \mathbf{f}(\mathbf{z}, \mathbf{w})$ to be highly flexible, and so we use a deep neural network architecture. We can ensure that the overall function is invertible if we make each layer of the network invertible. To see this, consider three successive transformations, each corresponding to one layer, of the form:

*Exercise 18.2*

$$\mathbf{x} = \mathbf{f}^A(\mathbf{f}^B(\mathbf{f}^C(\mathbf{z}))). \tag{18.5}$$

Then the inverse function is given by

$$\mathbf{z} = \mathbf{g}^C(\mathbf{g}^B(\mathbf{g}^A(\mathbf{x}))) \tag{18.6}$$

where $\mathbf{g}^A, \mathbf{g}^B$, and $\mathbf{g}^C$ are the inverse functions of $\mathbf{f}^A, \mathbf{f}^B$, and $\mathbf{f}^C$, respectively. Moreover, the determinant of the Jacobian for such a layered structure is also easy to evaluate in terms of the Jacobian determinants for each of the individual layers by making use of the chain rule of calculus:

$$J_{ij} = \frac{\partial z_i}{\partial x_j} = \sum_k \sum_l \frac{\partial g_i^C}{\partial g_k^B} \frac{\partial g_k^B}{\partial g_l^A} \frac{\partial g_l^A}{\partial x_j}. \tag{18.7}$$

*Appendix A*

We recognize the right-hand side as the product of three matrices, and the determinant of a product is the product of the determinants. Therefore, the log determinant of the overall Jacobian will be the sum of the log determinants corresponding to each layer.

This approach to modelling a flexible distribution is called a *normalizing flow* because the transformation of a probability distribution through a sequence of mappings is somewhat analogous to the flow of a fluid. Also, the effect of the inverse mapping is to transform the complex data distribution into a normalized form, typically a Gaussian or normal distribution. Normalizing flows have been reviewed by Kobyzev, Prince, and Brubaker (2019) and Papamakarios *et al.* (2019). Here we discuss the core concepts from the two main classes of normalizing flows used in practice: *coupling flows* and *autoregressive flows*. We also look at the use of neural differential equations to define invertible mappings, leading to *continuous flows*.

## 18.1. Coupling Flows

Our goal is to design a single invertible function layer, so that we can compose many of them together to define a highly flexible class of invertible functions. Consider first a linear transformation of the form

$$\mathbf{x} = a\mathbf{z} + \mathbf{b}. \tag{18.8}$$

This is easy to invert, giving

$$\mathbf{z} = \frac{1}{a}(\mathbf{x} - \mathbf{b}). \tag{18.9}$$

*Exercise 3.6*

However, linear transformations are closed under composition, meaning that a sequence of linear transformations is equivalent to a single overall linear transformation. Moreover, a linear transformation of a Gaussian distribution is again Gaussian. So even if we have many such 'layers' of linear transformation, we will only ever have a Gaussian distribution. The question is whether we can retain the invertability of a linear transformation while allowing additional flexibility so that the resulting distribution can be non-Gaussian.

One solution to this problem is given by a form of normalizing flow model called *real NVP* (Dinh, Krueger, and Bengio, 2014; Dinh, Sohl-Dickstein, and Bengio, 2016), which is short for 'real-valued non-volume-preserving'. The idea is to partition the latent-variable vector $\mathbf{z}$ into two parts $\mathbf{z} = (\mathbf{z}_A, \mathbf{z}_B)$, so that if $\mathbf{z}$ has dimension $D$ and $\mathbf{z}_A$ has dimension $d$, then $\mathbf{z}_B$ has dimension $D - d$. We similarly

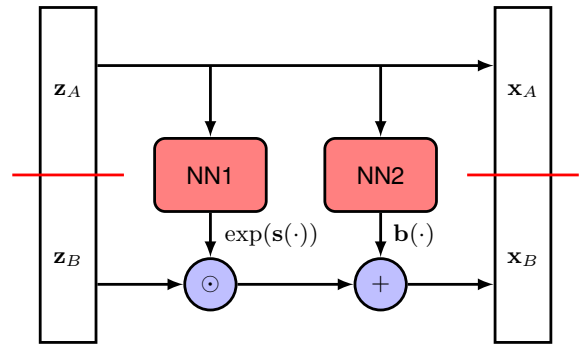


**Figure 18.1** A single layer of the real NVP normalizing flow model. Here the network NN1 computes the function $\exp(\mathbf{s}(\mathbf{z}_A, \mathbf{w}))$ and the network NN2 computes the function $\mathbf{b}(\mathbf{z}_A, \mathbf{w})$. The output vector is then defined by (18.10) and (18.11).

partition the output vector $\mathbf{x} = (\mathbf{x}_A, \mathbf{x}_B)$ where $\mathbf{x}_A$ has dimension $d$ and $\mathbf{x}_B$ has dimension $D - d$. For the first part of the output vector, we simply copy the input:

$$\mathbf{x}_A = \mathbf{z}_A. \tag{18.10}$$

The second part of the vector undergoes a linear transformation, but now the coefficients in the linear transformation are given by nonlinear functions of $\mathbf{z}_A$:

$$\mathbf{x}_B = \exp(\mathbf{s}(\mathbf{z}_A, \mathbf{w})) \odot \mathbf{z}_B + \mathbf{b}(\mathbf{z}_A, \mathbf{w}) \tag{18.11}$$

where $\mathbf{s}(\mathbf{z}_A, \mathbf{w})$ and $\mathbf{b}(\mathbf{z}_A, \mathbf{w})$ are the real-valued outputs of neural networks, and the exponential ensures that the multiplicative term is non-negative. Here $\odot$ denotes the *Hadamard product* involving an element-wise multiplication of the two vectors. Similarly, the exponential in (18.11) is taken element-wise. Note that we have shown the same vector $\mathbf{w}$ in both network functions. In practice, these may be implemented as separate networks with their own parameters, or as one network with two sets of outputs.

Due to the use of neural network functions, the value of $\mathbf{x}_B$ can be a very flexible function of $\mathbf{x}_A$. Nevertheless, the overall transformation is easily invertible: given a value for $\mathbf{x} = (\mathbf{x}_A, \mathbf{x}_B)$ we first compute

$$\mathbf{z}_A = \mathbf{x}_A, \tag{18.12}$$

then we evaluate $\mathbf{s}(\mathbf{z}_A, \mathbf{w})$ and $\mathbf{b}(\mathbf{z}_A, \mathbf{w})$, and finally we compute $\mathbf{z}_B$ using

$$\mathbf{z}_B = \exp(-\mathbf{s}(\mathbf{z}_A, \mathbf{w})) \odot (\mathbf{x}_B - \mathbf{b}(\mathbf{z}_A, \mathbf{w})). \tag{18.13}$$

The overall transformation is illustrated in Figure 18.1. Note that there is no requirement for the individual neural network functions $\mathbf{s}(\mathbf{z}_A, \mathbf{w})$ and $\mathbf{b}(\mathbf{z}_A, \mathbf{w})$ to be invertible.

Now consider the evaluation of the Jacobian defined by (18.2) and its determinant. We can divide the Jacobian matrix into blocks, corresponding to the partitioning of $\mathbf{z}$ and $\mathbf{x}$, giving

$$\mathbf{J} = \begin{bmatrix} \mathbf{I}_d & \mathbf{0} \\ \dfrac{\partial \mathbf{z}_B}{\partial \mathbf{x}_A} & \text{diag}(\exp(-\mathbf{s})) \end{bmatrix}. \tag{18.14}$$

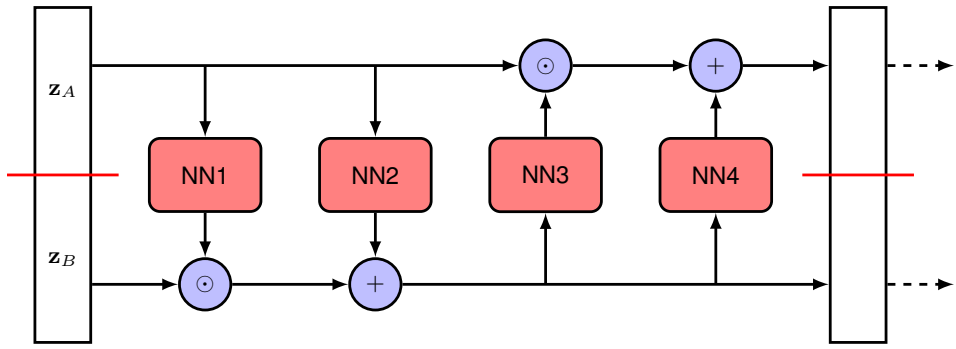


**Figure 18.2** By composing two layers of the form shown in Figure 18.1, we obtain a more flexible, but still invertible, nonlinear layer. Each sub-layer is invertible and has an easily evaluated Jacobian, and hence the overall double layer has the same properties.

The top left block corresponds to the derivatives of $\mathbf{z}_A$ with respect to $\mathbf{x}_A$ and hence from (18.12) is given by the $d \times d$ identity matrix. The top right block corresponds to the derivatives of $\mathbf{z}_A$ with respect to $\mathbf{x}_B$ and these terms vanish, again from (18.12). The bottom left block corresponds to the derivatives of $\mathbf{z}_B$ with respect to $\mathbf{x}_A$. From (18.13), these are complicated expressions involving the neural network functions. Finally, the bottom right block corresponds to the derivatives of $\mathbf{z}_B$ with respect to $\mathbf{x}_B$, which from (18.13) are given by a diagonal matrix whose diagonal elements are given by the exponentials of the negative elements of $\mathbf{s}(\mathbf{z}_A, \mathbf{w})$. We therefore see that the Jacobian matrix (18.14) is a lower triangular matrix, meaning that all elements above the leading diagonal are zero. For such a matrix, the determinant is just the product of the elements along the leading diagonal, and therefore it does not depend on the complicated expressions in the lower left block. Consequently, the determinant of the Jacobian is simply given by the product of the elements of $\exp(-\mathbf{s}(\mathbf{z}_A, \mathbf{w}))$.

*Appendix A*

A clear limitation of this approach is that the value of $\mathbf{z}_A$ is unchanged by the transformation. This is easily resolved by adding another layer in which the roles of $\mathbf{z}_A$ and $\mathbf{z}_B$ are reversed, as illustrated in Figure 18.2. This double-layer structure can then be repeated multiple times to facilitate a very flexible class of generative models.

The overall training procedure involves creating mini-batches of data points, in which the contribution of each data point to the log likelihood function is obtained from (18.4). For a latent distribution of the form $\mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I})$, the log density is simply $-\|\mathbf{z}\|^2/2$ up to an additive constant. The inverse transformation $\mathbf{z} = \mathbf{g}(\mathbf{x})$ is calculated using a sequence of inverse transformations of the form (18.13). Similarly, the log of the Jacobian determinant is given by a sum of log determinants for each layer where each term is itself a sum of terms of the form $-s_i(\mathbf{x}, \mathbf{w})$. Gradients of the log likelihood can be evaluated using automatic differentiation, and the network parameters updated by stochastic gradient descent.

The real NVP model belongs to a broad class of normalizing flows called *coupling flows*, in which the linear transformation (18.11) is replaced by a more general

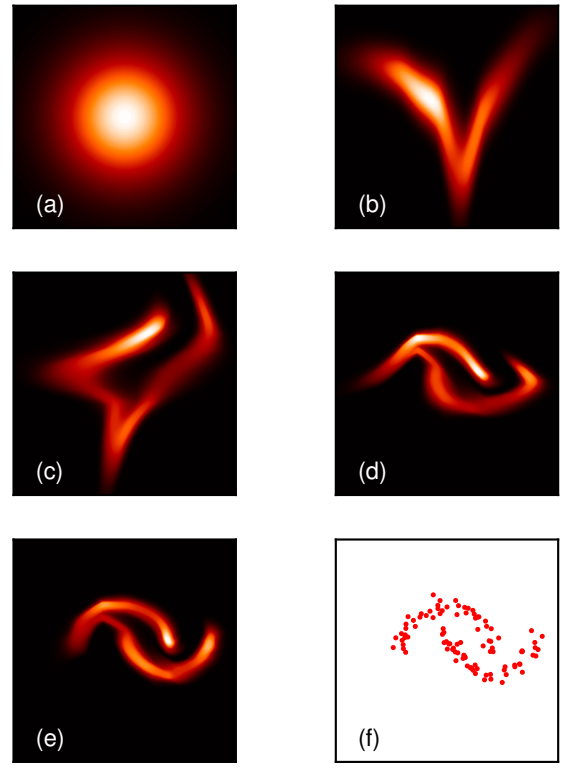


**Figure 18.3** Illustration of the real NVP normalizing flow model applied to the two-moons data set showing (a) the Gaussian base distribution, (b) the distribution after a transformation of the vertical axis only, (c) the distribution after a subsequent transformation of the horizontal axis, (d) the distribution after a second transformation of the vertical axis, (e) the distribution after a second transformation of the horizontal axis, and (f) the data set on which the model was trained.

form:

$$\mathbf{x}_B = \mathbf{h}(\mathbf{z}_B, \mathbf{g}(\mathbf{z}_A, \mathbf{w})) \tag{18.15}$$

where $\mathbf{h}(\mathbf{z}_B, \mathbf{g})$ is a function of $\mathbf{z}_B$ that is efficiently invertible for any given value of $\mathbf{g}$ and is called the *coupling function*. The function $\mathbf{g}(\mathbf{z}_A, \mathbf{w})$ is called a *conditioner* and is typically represented by a neural network.

We can illustrate the real NVP normalizing flow using a simple data set, sometimes known as 'two moons', as shown in Figure 18.3. Here a two-dimensional Gaussian distribution is transformed into a more complex distribution by using two successive layers each of which consists of alternate transformations on each of the two dimensions.

## 18.2. Autoregressive Flows

A related formulation of normalizing flows can be motivated by noting that the joint distribution over a set of variables can always be written as the product of conditional distributions, one for each variable. We first choose an ordering of the variables in

*Section 11.1*

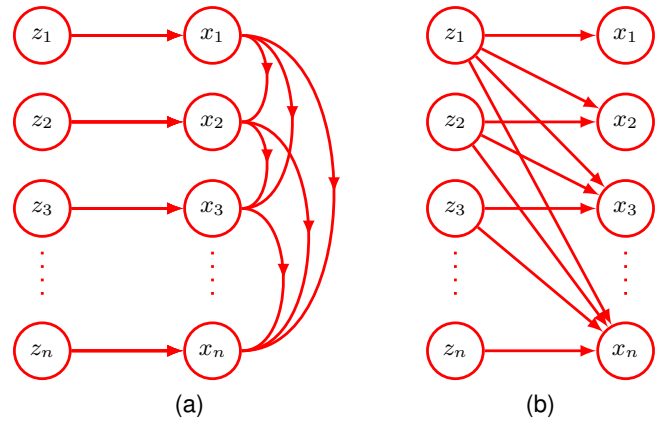


**Figure 18.4** Illustration of two alternative structures for autoregressive normalizing flows. The *masked autoregressive flow* shown in (a) allows efficient evaluation of the likelihood function, whereas the alternative *inverse autoregressive flow* shown in (b) allows for efficient sampling.

the vector $\mathbf{x}$, from which we can write, without loss of generality,

$$p(x_1, \ldots, x_D) = \prod_{i=1}^{D} p(x_i | \mathbf{x}_{1:i-1}) \tag{18.16}$$

where $\mathbf{x}_{1:i-1}$ denotes $x_1, \ldots, x_{i-1}$. This factorization can be used to construct a class of normalizing flow called a *masked autoregressive flow*, or MAF (Papamakarios, Pavlakou, and Murray, 2017), given by

$$x_i = h(z_i, \mathbf{g}_i(\mathbf{x}_{1:i-1}, \mathbf{w}_i)) \tag{18.17}$$

which is illustrated in Figure 18.4(a). Here $h(z_i, \cdot)$ is the *coupling function*, which is chosen to be easily invertible with respect to $z_i$, and $\mathbf{g}_i$ is the *conditioner*, which is typically represented by a deep neural network. The term *masked* refers to the use of a single neural network to implement a set of equations of the form (18.17) along with a binary mask (Germain *et al.*, 2015) that force a subset of the network weights to be zero to implement the autoregressive constraint (18.16).

In this case the reverse calculations needed to evaluate the likelihood function are given by

$$z_i = h^{-1}(x_i, \mathbf{g}_i(\mathbf{x}_{1:i-1}, \mathbf{w}_i)) \tag{18.18}$$

and hence can be performed efficiently on modern hardware since the individual functions in (18.18) needed to evaluate $z_1, \ldots, z_D$ can be evaluated in parallel. The Jacobian matrix corresponding to the set of transformations (18.18) has elements $\partial z_i / \partial x_j$, which form an upper-triangular matrix whose determinant is given by the product of the diagonal elements and can therefore also be evaluated efficiently. However, sampling from this model must be done by evaluating (18.17), which is intrinsically sequential and therefore slow because the values of $x_1, \ldots, x_{i-1}$ must be evaluated before $x_i$ can be computed.

*Exercise 18.4*

To avoid this inefficient sampling, we can instead define an *inverse autoregressive flows*, or IAF (Kingma *et al.*, 2016), given by

$$x_i = h(z_i, \widetilde{\mathbf{g}}_i(\mathbf{z}_{1:i-1}, \mathbf{w}_i)) \tag{18.19}$$

as illustrated in Figure 18.4(b). Sampling is now efficient since, for a given choice of $\mathbf{z}$, the evaluation of the elements $x_1, \ldots, x_D$ using (18.19) can be performed in parallel. However, the inverse function, which is needed to evaluate the likelihood, requires a series of calculations of the form

$$z_i = h^{-1}(x_i, \widetilde{\mathbf{g}}_i(\mathbf{z}_{1:i-1}, \mathbf{w}_i)), \tag{18.20}$$

which are intrinsically sequential and therefore slow. Whether a masked autoregressive flow or an inverse autoregressive flow is preferred will depend on the specific application.

We see that coupling flows and autoregressive flows are closely related. Although autoregressive flows introduce considerable flexibility, this comes with a computational cost that grows linearly in the dimensionality $D$ of the data space due to the need for sequential ancestral sampling. Coupling flows can be viewed as a special case of autoregressive flows in which some of this generality is sacrificed for efficiency by dividing the variables into two groups instead of $D$ groups.

## 18.3. Continuous Flows

The final approach to normalizing flows that we consider in this chapter will make use of deep neural networks defined in terms of an ordinary differential equation, or ODE. This can be thought of as a deep network with an infinite number of layers. We first introduce the concept of a neural ODE then we see how this can be applied to the formulation of a normalizing flow model.

### 18.3.1 Neural differential equations

We have seen that neural networks are especially useful when they comprise many layers of processing, and so we can ask what happens if we explore the limit of an infinitely large number of layers. Consider a residual network where each layer of processing generates an output given by the input vector with the addition of some parameterized nonlinear function of that input vector:

$$\mathbf{z}^{(t+1)} = \mathbf{z}^{(t)} + \mathbf{f}(\mathbf{z}^{(t)}, \mathbf{w}) \tag{18.21}$$

where $t = 1, \ldots, T$ labels the layers in the network. Note that we have used the same function at each layer, with a shared parameter vector $\mathbf{w}$, because this allows us to consider an arbitrarily large number of such layers while keeping the number of parameters bounded. Imagine that we increase the number of layers while ensuring that the changes introduced at each layer become correspondingly smaller. In the limit, the hidden-unit activation vector becomes a function $\mathbf{z}(t)$ of a continuous variable $t$, and we can express the evolution of this vector through the network as a differential equation:

*Exercise 18.5*

$$\frac{\mathrm{d}\mathbf{z}(\mathbf{t})}{\mathrm{d}t} = \mathbf{f}(\mathbf{z}(t), \mathbf{w}) \tag{18.22}$$

where $t$ is often referred to as 'time'. The formulation in (18.22) is known as a *neural ordinary differential equation* or *neural ODE* (Chen *et al.*, 2018). Here 'ordinary'

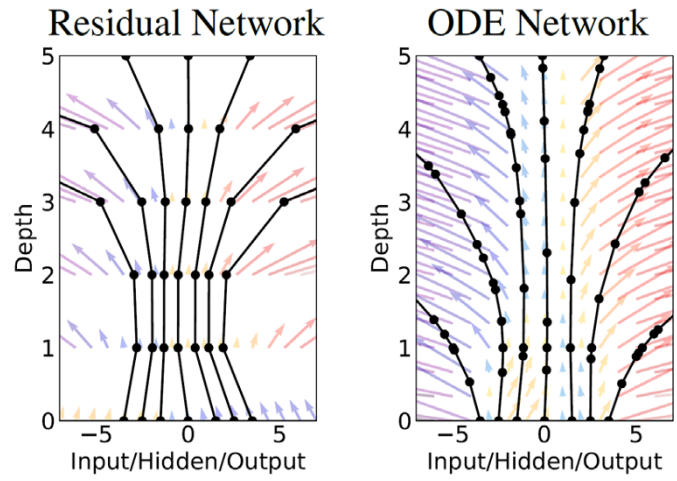


**Figure 18.5** Comparison of a conventional layered network with a neural differential equation. The diagram on the left corresponds to a residual network with five layers and shows trajectories for several starting values of a single scalar input. The diagram on the right shows the result of numerical integration of a continuous neural ODE, again for several starting values of the scalar input, in which we see that the function is not evaluated at uniformly-spaced time intervals, but instead the evaluation points are chosen adaptively by the numerical solver and depend on the choice of input value. [From Chen *et al.* (2018) with permission.]

means that there is a single variable $t$. If we denote the input to the network by the vector $\mathbf{z}(0)$, then the output $\mathbf{z}(T)$ is obtained by integration of the differential equation

$$\mathbf{z}(T) = \int_0^T \mathbf{f}(\mathbf{z}(t), \mathbf{w})\, \mathrm{d}t. \tag{18.23}$$

This integral can be evaluated using standard numerical integration packages. The simplest method for solving differential equations is *Euler's forward integration* method, which corresponds to the expression (18.21). In practice, more powerful numerical integration algorithms can adapt their function evaluation to achieve. In particular, they can adaptively choose values of $t$ that typically are not uniformly spaced. The number of such evaluations replaces the concept of depth in a conventional layered network. A comparison of a standard layered neural network and a neural differential equation are shown in Figure 18.5.

### 18.3.2 Neural ODE backpropagation

We now need to address the challenge of how to train a neural ODE, that is how to determine the value of $\mathbf{w}$ by optimizing a loss function. Let us assume that we are given a data set comprising values of the input vector $\mathbf{z}(0)$ along with an associated output target vector and a loss function $L(\cdot)$ that depends on the output vector $\mathbf{z}(T)$.

*Section 8.2*

One approach would be to use automatic differentiation to differentiate through all of the operations performed by the ODE solver during the forward pass. Although

this is straightforward to do, it is costly from a memory perspective and is not optimal in terms of controlling numerical error. Instead, Chen *et al.* (2018) treat the ODE solver as a black box and use a technique called the *adjoint sensitivity method*, which can be viewed as the continuous analogue of explicit backpropagation. Recall that backpropagation involves, for each data point, three successive phases: first a forward propagation to evaluate the activation vectors at each layer of the network, second the evaluation of the derivatives of the loss with respect to the activations at each layer starting at the output and propagating backwards through the network by exploiting the chain rule of calculus, and third the evaluation of the derivatives with respect to network parameters by forming products of activations from the forward pass and gradients from the backward pass. We will see that there are analogous steps when computing the gradients for a neural ODE.

*Chapter 8*

To apply backpropagation to neural ODEs, we define a quantity called the *adjoint* given by

$$\mathbf{a}(t) = \frac{\mathrm{d}L}{\mathrm{d}\mathbf{z}(t)}. \tag{18.24}$$

We see that $\mathbf{a}(T)$ corresponds to the usual derivative of the loss with respect to the output vector. The adjoint satisfies its own differential equation given by

*Exercise 18.6*

$$\frac{\mathrm{d}\mathbf{a}(t)}{\mathrm{d}t} = -\mathbf{a}(t)^{\mathrm{T}} \nabla_{\mathbf{z}} f(\mathbf{z}(t), \mathbf{w}), \tag{18.25}$$

which is a continuous version of the chain rule of calculus. This can be solved by integrating backwards starting from $\mathbf{a}(T)$, which again can be done using a black-box ODE solver. In principle, this requires that we have stored the trajectory $\mathbf{z}(t)$ computed during the forward phase, which could be problematic as the inverse solver might wish to evaluate $\mathbf{z}(t)$ at different values of $t$ compared to the forward solver. Instead we simply allow the backwards solver to recompute any required values of $\mathbf{z}(t)$ by integrating (18.22) alongside (18.25) starting with the output value $\mathbf{z}(T)$.

The third step in the backpropagation method is to evaluate derivatives of the loss with respect to network parameters by forming appropriate products of activations and gradients. When a parameter value is shared across multiple connections in a network, the total derivative is formed from the sum of derivatives for each of the connections. For our neural ODE, in which the same parameter vector $\mathbf{w}$ is shared throughout the network, this summation becomes an integration over $t$, which takes the form

*Exercise 9.7*

*Exercise 18.7*

$$\nabla_{\mathbf{w}} L = -\int_0^T \mathbf{a}(t)^{\mathrm{T}} \nabla_{\mathbf{w}} f(\mathbf{z}(t), \mathbf{w})\, \mathrm{d}t. \tag{18.26}$$

The derivatives $\nabla_{\mathbf{z}} f$ in (18.25) and $\nabla_{\mathbf{w}} f$ in (18.26) can be evaluated efficiently using automatic differentiation. Note that the above results can equally be applied to a more general neural network function $\mathbf{f}(\mathbf{z}(t), t, \mathbf{w})$ that has an explicit dependence on $t$ in addition to the implicit dependence through $\mathbf{z}(t)$.

*Section 8.2*

One benefit of neural ODEs trained using the adjoint method, compared to conventional layered networks, is that there is no need to store the intermediate results of the forward propagation, and hence the memory cost is constant. Furthermore,

neural ODEs can naturally handle continuous-time data in which observations occur at arbitrary times. If the error function $L$ depends on values of $\mathbf{z}(t)$ other than the output value, then multiple runs of the reverse-model solver are required, with one run for each consecutive pair of outputs, so that the single solution is broken down into multiple consecutive solutions in order to access the intermediate states (Chen *et al.*, 2018). Note that a high level of accuracy in the solver can be used during training, with a lower accuracy, and hence fewer function evaluations, during inference in applications for which compute resources are limited.

### 18.3.3 Neural ODE flows

We can make use of a neural ordinary differential equation to define an alternative approach to the construction of tractable normalizing flow models. A neural ODE defines a highly flexible transformation from an input vector $\mathbf{z}(0)$ to an output vector $\mathbf{z}(T)$ in terms of a differential equation of the form

$$\frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}t} = \mathbf{f}(\mathbf{z}(t), \mathbf{w}). \tag{18.27}$$

If we define a base distribution over the input vector $p(\mathbf{z}(0))$ then the neural ODE propagates this forward through time to give a distribution $p(\mathbf{z}(t))$ for each value of $t$, leading to a distribution over the output vector $p(\mathbf{z}(T))$. Chen *et al.* (2018) showed that for neural ODEs, the transformation of the density can be evaluated by integrating a differential equation given by

*Exercise 18.8*

$$\frac{\mathrm{d}\ln p(\mathbf{z}(t))}{\mathrm{d}t} = -\mathrm{Tr}\left(\frac{\partial \mathbf{f}}{\partial \mathbf{z}(t)}\right) \tag{18.28}$$

where $\partial\mathbf{f}/\partial\mathbf{z}$ represents the Jacobian matrix with elements $\partial f_i/\partial z_j$. This integration can be performed using standard ODE solvers. Likewise, samples from this density can be obtained by sampling from the base density $p(\mathbf{z}(0))$, which is chosen to be a simple distribution such as a Gaussian, and propagating the values to the output by integrating (18.27) again using the ODE solver. The resulting framework is known as a *continuous normalizing flow* and is illustrated in Figure 18.6. Continuous normalizing flows can be trained using the *adjoint sensitivity method* used for neural ODEs, which can be viewed as the continuous time equivalent of backpropagation.

*Exercise 18.9*
*Section 18.3.1*

Since (18.28) involves the trace of the Jacobian rather than the determinant, which arises in discrete normalizing flows, it might appear to be more computationally efficient. In general, evaluating the determinant of a $D \times D$ matrix requires $\mathcal{O}(D^3)$ operations, whereas evaluating the trace requires $\mathcal{O}(D)$ operations. However, if the determinant is lower diagonal, as in many forms of normalizing flow, then the determinant is the product of the diagonal terms and therefore also involves $\mathcal{O}(D)$ operations. Since evaluating the individual elements of the Jacobian matrix requires a separate forward propagation, which itself requires $\mathcal{O}(D)$ operations, evaluating the trace or the determinant (for a lower triangular matrix) takes $\mathcal{O}(D^2)$ operations overall. However, the cost of evaluating the trace can be reduced to $\mathcal{O}(D)$ by using *Hutchinson's trace estimator* (Grathwohl *et al.*, 2018), which for a matrix

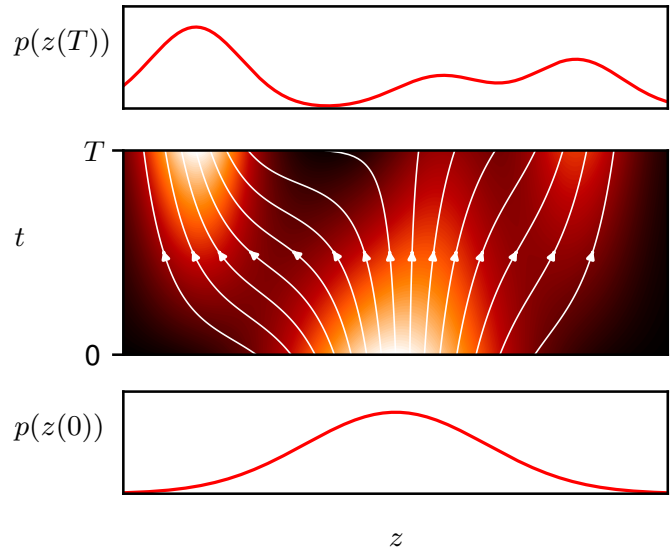


**Figure 18.6** Illustration of a continuous normalizing flow showing a simple Gaussian distribution at $t = 0$ that is continuously transformed into a multimodal distribution at $t = T$. The flow lines show how points along the $z$-axis evolve as a function of $t$. Where the flow lines spread apart the density is reduced, and where they move together the density is increased.

$\mathbf{A}$ takes the form

$$\text{Tr}(\mathbf{A}) = \mathbb{E}_{\boldsymbol{\epsilon}}\left[\boldsymbol{\epsilon}^{\text{T}}\mathbf{A}\boldsymbol{\epsilon}\right] \tag{18.29}$$

where $\boldsymbol{\epsilon}$ is a random vector whose distribution has zero mean and unit covariance, for example, a Gaussian $\mathcal{N}(\mathbf{0}, \mathbf{I})$. For a specific $\boldsymbol{\epsilon}$, the matrix-vector product $\mathbf{A}\boldsymbol{\epsilon}$ can be evaluated efficiently in a single pass using reverse-mode automatic differentiation. We can then approximate the trace using a finite number of samples in the form

$$\text{Tr}(\mathbf{A}) \simeq \frac{1}{M}\sum_{m=1}^{M} \boldsymbol{\epsilon}_m^{\text{T}}\mathbf{A}\boldsymbol{\epsilon}_m. \tag{18.30}$$

In practice we can set $M = 1$ and just use a single sample, which is refreshed for each new data point. Although this is a noisy estimate, this might not be too significant since it forms part of a noisy stochastic gradient descent procedure. Importantly it is unbiased, meaning that the expectation of the estimator is equal to the true value.

*Exercise 18.11*

Significant improvements in training efficiency for continuous normalizing flows can be achieved using a technique called *flow matching* (Lipman *et al.*, 2022). This brings normalizing flows closer to diffusion models and avoids the need for back-propagation through the integrator while significantly reducing memory requirements and enabling faster inference and more stable training.

*Chapter 20*