Chapter 7

Flow Models

In the last chapter, we explored how Deep Latent Variable Models (DLVMs) model complex probability distributions as the integral of the product of two simpler distributions. This approach, while powerful, introduces several complexities, such as the intractability of the marginal probability of the observations. To address this, we often resort to approximations like the Evidence Lower Bound (ELBO).

In this chapter, we will examine a simpler yet effective approach for modeling complex distributions: flow models. Flow models are also latent variable models but circumvent many of the challenges associated with DLVMs described in the previous chapter. They allow us to compute marginals (i.e., $p(\mathbf{x})$) exactly, sample efficiently, and use them for representation learning without the need for complex likelihood approximations. Flow models fall into the category of explicit models with a tractable density, as illustrated in Figure 7.1.

7.1 Introduction

Latent Variable Models (LVMs) are characterized by a probability distribution over the latent variable $\mathbf{z}$, denoted by $p(\mathbf{z})$, and a conditional probability distribution over the observation $\mathbf{x}$ given the latent variable $\mathbf{z}$, denoted by $p_\theta(\mathbf{x}|\mathbf{z})$. In Deep Latent Variable Models, the parameters of this conditional probability are determined by a Deep Neural Network called the decoder. The decoder takes $\mathbf{z}$ as input and outputs the parameter of the probability distribution over $\mathbf{x}$. For example, in the case of the Variational Autoencoder (VAE), these parameters are the mean vector and covariance

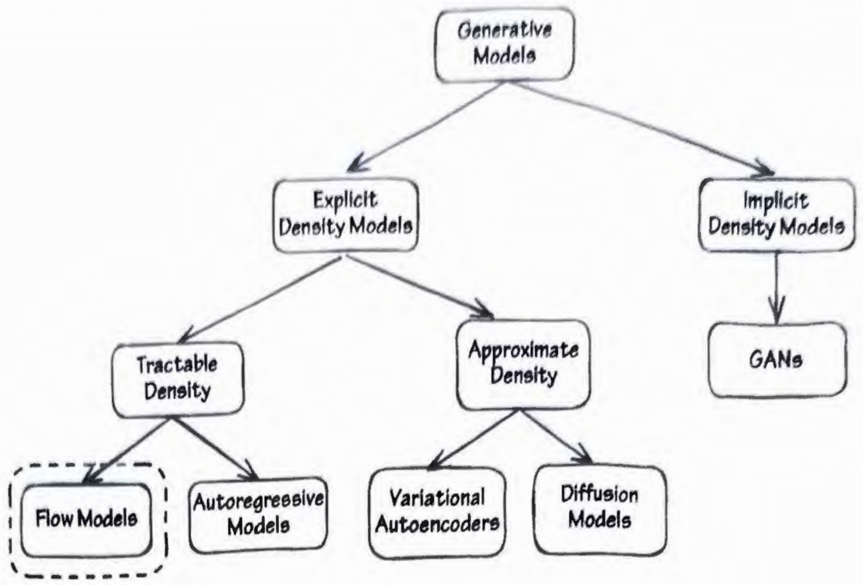


Figure 7.1 Model taxonomy: flow models.

matrix of a Gaussian distribution over **x**. In this setting, the value of **z** influences the probability density over the possible states of **x**, but it does not deterministically specify its state. Thus, the mapping from **z** to **x** provided by the decoder is probabilistic.

In flow models, we replace the probabilistic decoder $p_\theta(\mathbf{x}|\mathbf{z})$, with a deterministic mapping from the latent variable **z** to the observation **x**, denoted by $\mathbf{x} = f(\mathbf{z})$. In the terminology of flow models, $p_\mathbf{z}(\mathbf{z})$ is called the base measure or base distribution, and f is referred as the flow.

In these models, computing the marginal or density of **x** involves solving a *derived distribution* problem, where the base measure $p_\mathbf{z}(\mathbf{z})$ is given, and we aim to find the probability distribution of **x** as a function of the base measure. The distribution of **x** is given by the change of variables formula, which allows us to express the probability density $p(\mathbf{x})$ as a function of the known $p(\mathbf{z})$.

$$p_\mathbf{x}(\mathbf{x}) = p_\mathbf{z}(\mathbf{z}) \left| \det\left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}}\right) \right|^{-1} \tag{7.1}$$

with $\mathbf{z} = f^{-1}(\mathbf{x})$, and $\mathbf{J} = \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}}$ representing the Jacobian matrix.

In flow models, the base measure is typically a simple distribution, such as a normal distribution or a uniform distribution. When the base measure is a normal distribution, the flow model is known as a *normalizing flow*. The goal is to design the flow f to be a highly complex function capable of mapping a simple base measure to a multimodal, complex data distribution.

Compared to deep latent variable models discussed in the previous chapter, normalizing flows allow exact computation of marginals without the need for approximations or lower bounds. Sampling in flow models is also simpler, and can be performed as follows:

$$\begin{aligned} \mathbf{z} &\sim p(\mathbf{z}) \\ \mathbf{x} &= f(\mathbf{z}) \end{aligned} \tag{7.2}$$

Thus, we sample $\mathbf{z}$ from the base measure and apply the flow transformation to obtain the observation $\mathbf{x}$.

The change of variables formula imposes certain constraints on the family of flows f that we can use. Specifically, for this formula to work, f must be invertible and differentiable. Consequently, the dimensionality of the observations $\mathbf{x}$ must be equal to the dimensionality of the latent variable $\mathbf{z}$.

To meet these requirements, we need to design special neural network architectures that are invertible and differentiable.This often involves creating innovative and complex deep models that satisfy to these constraints.

7.2 Model Training

We train flow models using maximum likelihood estimation. The log-likelihood is given by:

$$\begin{aligned}\mathcal{L} &= \sum_n \log p_{\mathbf{x}}(\mathbf{x}_n) \\ &= \sum_n \log\left[p_{\mathbf{z}}(\mathbf{z}_n)\left|\det\left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}}\right)\right|^{-1}\right] \\ &= \sum_n \log p_{\mathbf{z}}(\mathbf{z}_n) - \log\left|\det\left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}}\right)\right| \end{aligned} \tag{7.3}$$

where $\mathbf{z}_n = f^{-1}(\mathbf{x}_n)$.The primary challenge lies in designing neural network architectures to model a flow f that is invertible and has a Jacobian determinant that can be efficiently computed.This is crucial for ensuring that the log-likelihood can be computed quickly for potentially millions of data points in our dataset. In the following sections, we will detail and elaborate on some architectures that meet these constraints.

7.3 Linear Flows

Let's revisit the linear factor model from the last chapter, where the prior over $\mathbf{z}$ and the conditional model of $\mathbf{x}|\mathbf{z}$ are given by:

$$\begin{aligned} p_{\mathbf{z}}(\mathbf{z}) &= \mathcal{N}(\mathbf{z};\mathbf{0},\mathbf{I}) \\ p_{\mathbf{x}|\mathbf{z}}(\mathbf{x}|\mathbf{z}) &= \mathcal{N}(\mathbf{x};\boldsymbol{\mu}+\Lambda\mathbf{z},\boldsymbol{\Sigma}_0) \end{aligned} \tag{7.4}$$

To convert this linear factor model into a flow model, a simple approach is to replace the probabilistic mapping or probabilistic decoder from $\mathbf{z}$ to $\mathbf{x}$, defined by the conditional distribution, with a deterministic mapping or flow, represented by f. If we assume the same base measure $p_{\mathbf{z}}(\mathbf{z})$ (prior) and the flow f as $f(\mathbf{z}) = \boldsymbol{\mu} + \Lambda\mathbf{z}$, we obtain what is known as *Linear Normalizing Flows*, see Kobyzev et al. (2021).This can be interpreted as a scale and location transformation of the variable $\mathbf{z}$. In this setting, Λ must be a $D \times D$ invertible matrix in order for the flow f to be invertible.

Sampling under this model is straightforward. We sample $\mathbf{z}$ from the base measure $\mathbf{z} \sim p_{\mathbf{z}}$, and compute $\mathbf{x}$ using the flow and the realized value of $\mathbf{z}$: $\mathbf{x} = f(\mathbf{z})$.

As an exercise, let's compute the marginal distribution over $\mathbf{x}$ using the change of variables formula. To apply this formula, we need to know the inverse flow, which is given by $\mathbf{z} = f^{-1}(\mathbf{x}) = \Lambda^{-1}(\mathbf{x} - \boldsymbol{\mu})$, and the Jacobian, which is equal to Λ. Then, the density over $\mathbf{x}$ can be computed as follows:

$$\begin{aligned} p_{\mathbf{x}}(\mathbf{x}) &= p_{\mathbf{z}}(\mathbf{z}) \left| \det\left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}}\right)\right|^{-1} \\ &= \mathcal{N}(\mathbf{z}; 0, \mathbf{I}) \left| \det\left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}}\right)\right|^{-1} \\ &= (2\pi)^{-D/2} \exp\left(-\tfrac{1}{2}\mathbf{z}^T\mathbf{z}\right) |\det \Lambda|^{-1} \\ &= (2\pi)^{-D/2} \left(\det \Lambda\Lambda^T\right)^{-1/2} \exp\left(-\tfrac{1}{2}\left(\Lambda^{-1}(\mathbf{x} - \boldsymbol{\mu})\right)^T\left(\Lambda^{-1}(\mathbf{x} - \boldsymbol{\mu})\right)\right) \\ &= (2\pi)^{-D/2} \left(\det \Lambda\Lambda^T\right)^{-1/2} \exp\left(-\tfrac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T\left(\Lambda\Lambda^T\right)^{-1}(\mathbf{x} - \boldsymbol{\mu})\right) \\ &= \mathcal{N}\left(\mathbf{x}; \boldsymbol{\mu}, \Lambda\Lambda^T\right) \end{aligned}$$

where we used the fact that $|\det\Lambda|^{-1} = \left(\det \Lambda\Lambda^T\right)^{-1/2}$. Thus, $\mathbf{x}$ follows a Gaussian distribution with mean vector $\boldsymbol{\mu}$ and covariance matrix $\Lambda\Lambda^T$. An easier way to compute the marginal distribution over $\mathbf{x}$ is to recognize that a linear transformation of a Gaussian distribution results in another Gaussian distribution. Hence, the probability distribution over $\mathbf{x}$ can be derived by computing its mean vector and covariance matrix:

$$\begin{aligned} \mathbb{E}[\mathbf{x}] &= \mathbb{E}[\boldsymbol{\mu} + \Lambda\mathbf{z}] = \boldsymbol{\mu} \\ \mathrm{Cov}[\mathbf{x}] &= \Lambda\boldsymbol{\Sigma}_z\Lambda^T = \Lambda\mathbf{I}\Lambda^T = \Lambda\Lambda^T \end{aligned}$$

Therefore,

$$p_{\mathbf{x}}(\mathbf{x}) = \mathcal{N}\left(\mathbf{x}; \boldsymbol{\mu}, \Lambda\Lambda^T\right)$$

However, linear flows are limited in their expressiveness, as they only modify the mean and covariance matrix of the base distribution. Even if we stack multiple linear transformations, they remain closed under composition. This means that the composition of linear transformations is equivalent to a single linear transformation, which limits the expressiveness of this model. This is insufficient for modeling very complex data distributions. In general, for base measures that are members of the exponential family, a linear transformation remains within the exponential family, potentially eliminating the need for the change of variables formula for computing the probability density over the observations. However, in the case of nonlinear flows, this formula is essential.

Additionally, the determinant of the Jacobian $\det\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}}$ equal to $\det\Lambda$. If Λ is a full matrix, computing this determinant requires $\mathcal{O}(D^3)$ operations, which can be prohibitively expensive for high-dimensional datasets.

Table 7.1 Computational complexity of the inverse and determinant of the Jacobian based on matrix structure. Here, D denotes the dimension of the latent variable, and c represents the block size for Block Diagonal matrices or the number of channels for 1×1 convolutions. Adapted from Brubaker and Köthe (2021).

Matrix Type	Inverse Complexity	Determinant Complexity
Full	$\mathcal{O}(D^3)$	$\mathcal{O}(D^3)$
Diagonal	$\mathcal{O}(D)$	$\mathcal{O}(D)$
Triangular	$\mathcal{O}(D^2)$	$\mathcal{O}(D)$
Block Diagonal	$\mathcal{O}(c^3 D)$	$\mathcal{O}(c^3 D)$
LU Factorized	$\mathcal{O}(D^2)$	$\mathcal{O}(D)$
Special Convolution	$\mathcal{O}(D\log D)$	$\mathcal{O}(D)$
1×1 Convolution	$\mathcal{O}(c^3 + c^2 D)$	$\mathcal{O}(c^3)$

One way to reduce the computational complexity of the Jacobian's determinant and its inverse is to impose some particular structure on the linear transformation, in our case on the matrix Λ, that reduce the computational cost of inverse and determinant calculations. For example, Table 7.1 shows useful structures that can be imposed on the matrix used for the linear transformation, along with the computational complexity for computing its inverse and determinant. More useful structures are discussed in Kobyzev et al. (2021) and Brubaker and Köthe (2021).

7.4 Designing Nonlinear Flows

As we saw in the previous example with a linear flow, its expressiveness is very limited since a composition of linear transformations is equivalent to a single linear transformation. Additionally, we highlighted some of the main requirements that a flow must satisfy for the change of variables formula to work, as well as some key properties for model expressiveness and computational efficiency. We want flows that are:

- Expressive enough to model complex data distributions
- Invertible
- Differentiable
- Computationally efficient to invert (f^{-1})
- Computationally efficient to compute the Jacobian determinant

Flows are also called flow layers or bijections, see Kobyzev et al. (2021).

From deep neural networks, we know that an easy way to improve model expressiveness is through function composition. An important aspect of flow models is that the composition of invertible flows or layers remains invertible. So, if we compose

K different flows, such as $f = f_K \circ f_{K-1} \circ ... \circ f_2 \circ f_1$, and each f_k is invertible and differentiable, then f will also be invertible and differentiable.

Additionally, the determinant of the Jacobian for the composed function has a very convenient form:

$$\begin{aligned} \det \mathbf{J} &= \det \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \\ &= \det \prod_{k=1}^{K} \frac{\partial f_k(\mathbf{z}_k)}{\partial \mathbf{z}_k} \\ &= \prod_{k=1}^{K} \det \frac{\partial f_k(\mathbf{z}_k)}{\partial \mathbf{z}_k} \\ &= \prod_{k=1}^{K} \det \mathbf{J}_k \end{aligned} \tag{7.5}$$

where $\mathbf{J}_k = \frac{\partial f_k(\mathbf{z}_k)}{\partial \mathbf{z}_k}$. Thus, under the composition of flows, the determinant of the Jacobian $\mathbf{J}$ is equal to the product of the determinants of the Jacobians of the individual flows, which provides significant computational advantages. This form simplifies the computation of log-likelihoods to the sum of the log-determinants of each function involved in the composition.

In the following sections, we will describe one main way of designing nonlinear flows that are used in practice–coupling flows. Coupling flows are based on the principle of composition of invertible layers, a principle widely applied to the design of deep invertible neural network flows. Within this class, we will describe two specific cases: Nonlinear Independent Components Estimation (NICE) and Non-volume Preserving Transformation (Real-NVP).

7.5 Coupling Flows

Coupling flows are a widely used approach for constructing nonlinear flows (Dinh et al., 2015). The main idea is to partition the vector **z** into two parts:

$$\mathbf{z} = \begin{bmatrix} \mathbf{z}_{1:D'} \\ \mathbf{z}_{D'+1:D} \end{bmatrix} \tag{7.6}$$

To transform **z** into **x**, we apply the flow f, such that $\mathbf{x} = f(\mathbf{z})$. In the same manner, we can partition **x**:

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_{1:D'} \\ \mathbf{x}_{D'+1:D} \end{bmatrix} \tag{7.7}$$

A flow f is considered a coupling flow if:

$$f(\mathbf{z}) = \begin{bmatrix} \mathbf{x}_{1:D'} \\ \mathbf{x}_{D'+1:D} \end{bmatrix} = \begin{bmatrix} \mathbf{z}_{1:D'} \\ f_1(\mathbf{z}_{D'+1:D}; \boldsymbol{\Theta}(\mathbf{z}_{1:D'})) \end{bmatrix} \tag{7.8}$$

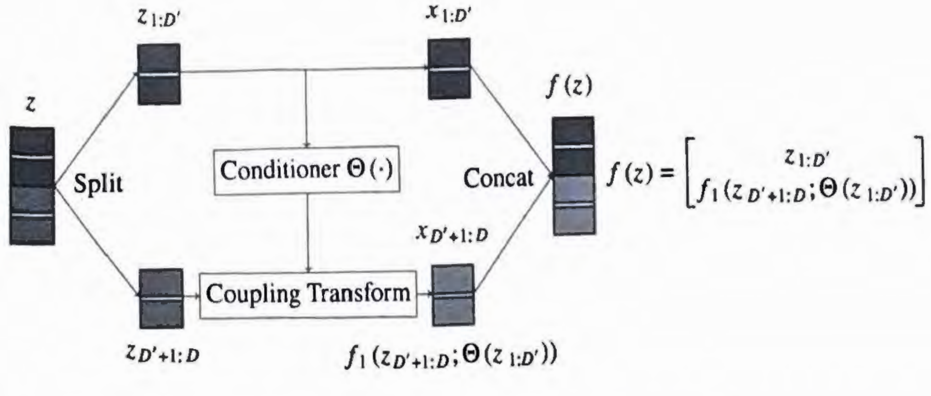
where the first partition of $\mathbf{z}(\mathbf{z}_{1:D'})$ remains unaffected by the flow f, and f_1 is another invertible transformation (another flow) with parameters θ, applied to the second partition of $\mathbf{z}(\mathbf{z}_{D'+1:D})$.

The parameters of the function f_1, denoted as θ, are defined by an arbitrary function $\boldsymbol{\Theta}$. In practice, $\boldsymbol{\Theta}$ is modelled by a neural network, whose input depend only on the first partition of $\mathbf{z}(\mathbf{z}_{1:D'})$ and not in the second partition $\mathbf{z}(\mathbf{z}_{D'+1:D})$. The function $\boldsymbol{\Theta}$ is referred to as a **conditioner**, which does not need to be invertible. The function f_1 is known as a **coupling function** or **coupling transform**, and must be a monotonic function of its argument, therefore invertible. The function f is called a **coupling flow** or **coupling layer**. As we will see in the following examples, different well-known coupling flows differ in how they define the conditioner and the coupling function. Specifically, we will examine two cases: one utilizing an additive coupling function and another employing an affine coupling function.

This structure has several advantages for building complex nonlinear mappings, which are invertible. By stacking multiple "coupling layers," we can create highly expressive flows while keeping the computation of the Jacobian determinant efficient, as we will see in the next section.

As an illustration of the process, Figure 7.2 demonstrates how the vector **z** is partitioned and then transformed using the flow f as described.

In the following section, inspired by the exposition style of Levine (2021), we will explore two specific implementations of coupling layers: **Nonlinear Independent Components Estimation (NICE)** and **Non-volume Preserving Transformation (Real-NVP)**.



$z_{1:D'}$
$x_{1:D'}$
z
$f(z)$
Split
Conditioner $\Theta(\cdot)$
Concat
$f(z) = \begin{bmatrix} z_{1:D'} \\ f_1(z_{D'+1:D}; \Theta(z_{1:D'})) \end{bmatrix}$
$x_{D'+1:D}$
Coupling Transform
$z_{D'+1:D}$
$f_1(z_{D'+1:D}; \Theta(z_{1:D'}))$

Figure 7.2 Illustration of the operations involved in coupling flows, including partitioning and transformation using a coupling layer. Adapted from Brubaker and Köthe (2021).

7.5.1 NICE: Nonlinear Independent Components Estimation

In this section, we explain the approach to designing invertible layers based on the work by Dinh et al. (2015), known as NICE: Nonlinear Independent Components Estimation.

As mentioned before, the approach involves partitioning the vector $\mathbf{z}$ into two parts, described as $\mathbf{z} = [\mathbf{z}_{1:D'}, \mathbf{z}_{D'+1:D}]^T$. Due to the constraints of the invertible mapping, the dimension of $\mathbf{x}$ must also be equal to D. We decompose the entries of $\mathbf{x}$ also into two components, namely $\mathbf{x} = [\mathbf{x}_{1:D'}, \mathbf{x}_{D'+1:D}]^T$. In NICE, the invertible layer is defined as follows:

$$\begin{aligned} \mathbf{x}_{1:D'} &= \mathbf{z}_{1:D'} \\ \mathbf{x}_{D'+1:D} &= \mathbf{z}_{D'+1:D} + g_\theta(\mathbf{z}_{1:D'}) \end{aligned} \tag{7.9}$$

NICE uses an additive coupling function, which is an invertible transformation with parameters defined by $g_\theta(\mathbf{z}_{1:D'})$. In practice, g_θ is implemented using a neural network, such as an MLP, a CNN, or other architectures tailored to the specific problem. Given the value $\mathbf{x}$, we can recover the vector $\mathbf{z}$ by first recovering its first D' entries: $\mathbf{z}_{1:D'} = \mathbf{x}_{1:D'}$, and then using this result to recover the second part of the entries $\mathbf{z}_{D'+1:D} = \mathbf{x}_{D'+1:D} - g_\theta(\mathbf{z}_{1:D'})$.

The Jacobian of this mapping is given by:

$$\begin{aligned} \mathbf{J} = \frac{\partial f(\mathbf{z})}{\partial z} &= \begin{bmatrix} \frac{\partial(\mathbf{x}_{1:D'})}{\partial \mathbf{z}_{1:D'}} & \frac{\partial(\mathbf{x}_{1:D'})}{\partial \mathbf{z}_{D'+1:D}} \\ \frac{\partial(\mathbf{x}_{D'+1:D})}{\partial \mathbf{z}_{1:D'}} & \frac{\partial(\mathbf{x}_{D'+1:D})}{\partial \mathbf{z}_{(D'+1):D}} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{I}_{D'} & 0 \\ g'_\theta(\mathbf{z}_{1:D'}) & \mathbf{I}_{D-D'} \end{bmatrix} \end{aligned} \tag{7.10}$$

where $\mathbf{I}_{D'}$ is an identity matrix with D' diagonal elements.

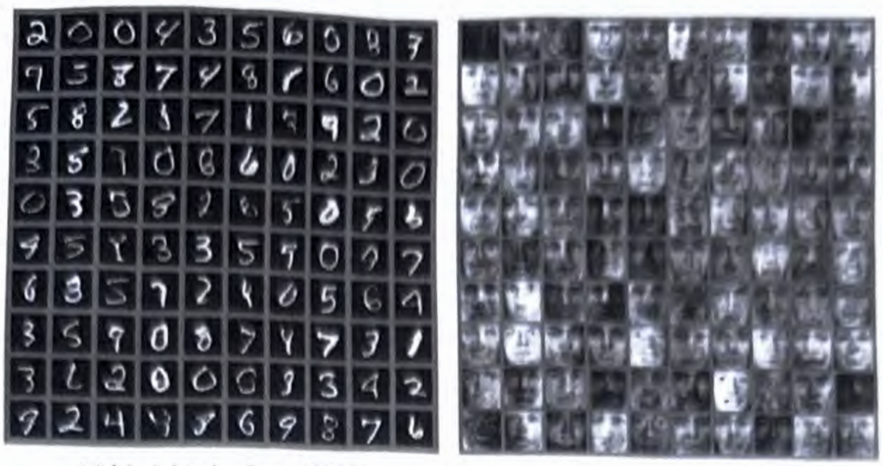
Since we are interested in the determinant of the Jacobian, we do not need to compute the $(2,1)$ entry of $\mathbf{J}$, as the $(1,2)$ of $\mathbf{J}$ is zero, and $g'\theta(\mathbf{z}_{1:D'})$ will not contribute to the result. Thus, the determinant of the Jacobian is $\det \mathbf{J} = 1$.

The log-likelihood for this model is given by:

$$\begin{aligned} \mathcal{L} &= \sum_n \log p(\mathbf{z}_n) - \log\left|\det\left(\frac{\partial f(\mathbf{z})}{\partial z}\right)\right| \\ &= \sum_n \log p(\mathbf{z}_n) - \log 1 \\ &= \sum_n \log p(\mathbf{z}_n) \end{aligned} \tag{7.11}$$

which is computationally convenient. However, the fact that the determinant of the Jacobian is constant and equal to 1 means that this flow only transforms the base measure but does not rescale it. This limits the expressiveness of the model and makes it difficult to fit very complex data distributions (see Levine [2021]). Therefore, in practical settings, a rescaling layer is often inserted into the model (see Dinh et al. [2015]).

In practice, we stack several of these invertible layers to train generative flow models effectively. As an illustrative example of how these models perform on simple image data,



(a) Model trained on MNIST (b) Model trained on TFD

Figure 7.3 Illustration of unbiased samples generated by the NICE model when trained on the MNIST dataset (subfigure a) and the TFD dataset (subfigure b). *Source:* Dinh, Krueger, and Bengio (2015).

in Figure 7.3, subfigures (a) and (b) show samples from a NICE model that has been trained to model the distribution of the simple MNIST digits dataset and basic images from the TFD dataset. The architecture used in both experiments consists of a stack of four NICE layers/functions. Each NICE layer is implemented as a deep rectified network with linear output units. The authors use the same network architecture for all NICE functions: five hidden layers with 1,000 units each for the MNIST dataset and four hidden layers with 5,000 units each for the TFD dataset. As observed, the generated digit images (subfigure a) look like real digits from the MNIST dataset, and the generated simple face images (subfigure b) resemble faces to some extent. This model, developed in 2015, was the first normalizing flow model capable of generating images of reasonable quality for its time. We will discuss how to extend these models to capture temporal dependencies in Section 7.8, as well as an example in financial applications in Section 7.8.3.

However, when the model is trained on more complex datasets, as shown in Figure 7.4, its performance deteriorates. In subfigure (c), we see samples generated from a model trained on the SVHN dataset, which consists of color images of digits taken from house numbers. The model begins to struggle to generate these kinds of color digits. When trained on CIFAR-10, which is a more complex dataset containing a diverse set of realistic object images, the model generates very low-quality images, showing its limitations in handling more complex data distributions.

In the accompanying notebook for this chapter ([*NICE Notebook*]), we implement and test the NICE coupling flow with custom architectures for generating equity financial data, which can be easily customized for your own applications. Additionally, we include experimental results to demonstrate the performance and practical applications of this approach.

7.5.2 Real-NVP: Non-volume Preserving Transformation

The Non-volume Preserving Transformation (Real-NVP) introduced by Dinh et al. (2017) proposes a new kind of invertible layer that enhances the model's expressiveness.

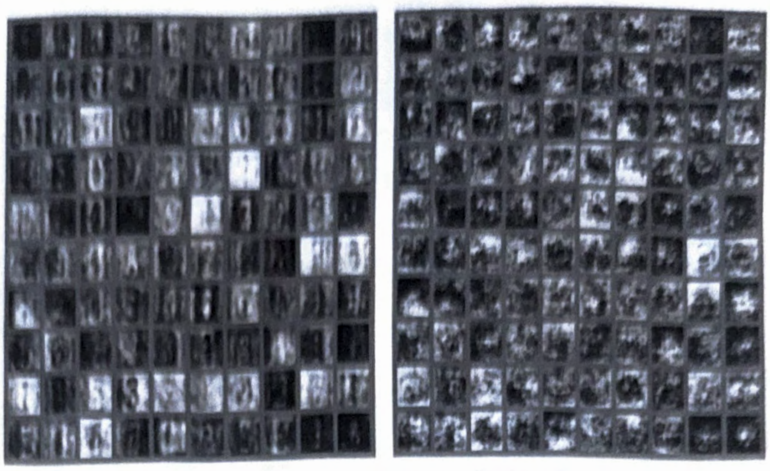


Figure 7.4 Illustration of unbiased samples generated by the NICE model when trained on the SVHN dataset (subfigure c) and on the CIFAR-10 dataset (subfigure d). *Source:* Dinh, Krueger, and Bengio (2015).

In particular, this model's Jacobian determinant is not constant, allowing for greater flexibility. Similar to the NICE model, in Real-NVP, they partition the vectors $\mathbf{z}$ and $\mathbf{x}$ into two parts, namely $\mathbf{z} = [\mathbf{z}_{1:D'}, \mathbf{z}_{D'+1:D}]^T$ and $\mathbf{x} = [\mathbf{x}_{1:D'}, \mathbf{x}_{D'+1:D}]^T$. The proposed invertible layer (coupling function) is defined as follows:

$$\begin{aligned} \mathbf{x}_{1:D'} &= \mathbf{z}_{1:D'} \\ \mathbf{x}_{D'+1:D} &= \mathbf{z}_{D'+1:D} \odot \exp(h_\theta(\mathbf{z}_{1:D'})) + g_\theta(\mathbf{z}_{1:D'}) \end{aligned} \tag{7.12}$$

where $\odot$ stands for the element-wise product, and h_θ and g_θ are neural network layers that can be implemented using architectures such as MLPs, CNNs, or other designs tailored to your specific problem. As defined in Equation 7.12, Real-NVP uses an affine coupling function, which is by definition an invertible transformation with parameters given by $\{h_\theta(\mathbf{z}_{1:D'}), g_\theta(\mathbf{z}_{1:D'})\}$. This can be interpreted as a scale and location transformation of the second partition of $\mathbf{z}(\mathbf{z}_{D'+1:D})$.

To recover the vector $\mathbf{z}$ from $\mathbf{x}$, we apply a transformation similar to that used in the NICE model:

$$\begin{aligned} \mathbf{z}_{1:D'} &= \mathbf{x}_{1:D'} \\ \mathbf{z}_{D'+1:D} &= (\mathbf{x}_{D'+1:D} - g_\theta(\mathbf{z}_{1:D'}))/\exp(h_\theta(\mathbf{z}_{1:D'})) \end{aligned} \tag{7.13}$$

Given this mapping, we can derive the Jacobian, which is given by:

$$\begin{aligned} \mathrm{J} = \frac{\partial f(\mathbf{z})}{\partial z} &= \begin{bmatrix} \frac{\partial(\mathbf{x}_{1:D'})}{\partial \mathbf{z}_{1:D}} & \frac{\partial(\mathbf{x}_{1:D'})}{\partial \mathbf{z}_{D'+1:D}} \\ \frac{\partial(\mathbf{x}_{D'+1:D})}{\partial \mathbf{z}_{1:D'}} & \frac{\partial(\mathbf{x}_{D'+1:D})}{\partial \mathbf{z}_{(D'+1):D}} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{I} & 0 \\ g'_\theta(\mathbf{z}_{1:D'}) & \mathrm{diag}(\exp(h_\theta(\mathbf{z}_{1:D'}))) \end{bmatrix} \end{aligned} \tag{7.14}$$

The Jacobian is triangular, so its determinant is the product of diagonal entries:

$$\det \mathbf{J} = \prod_{d=D'+1}^{D} \exp(h_\theta(\mathbf{z}_{1:D'})_d)$$
$$= \exp\left(\sum_{d=D'+1}^{D} h_\theta\left(\mathbf{z}_{1:D'}\right)_d\right) \tag{7.15}$$

Here we can see that the determinant of the Jacobian is not constant, allowing for the rescaling of the base measure and enabling greater expressiveness for the model to fit very complex data distributions. This makes the Real-NVP model more expressive compared to the NICE model.

Once again, as an illustrative example of how this performs on simple image data, Figure 7.5 shows samples generated by the RealNVP model, see Dinh et al. (2017). The authors of the paper present results of a RealNVP model trained on various datasets: CIFAR-10, ImageNet (32 × 32), ImageNet (64 × 64), CelebA, and LSUN (bedroom). In each pair of columns, the left column shows examples from the dataset, while the right column displays samples generated by the model trained on that dataset.

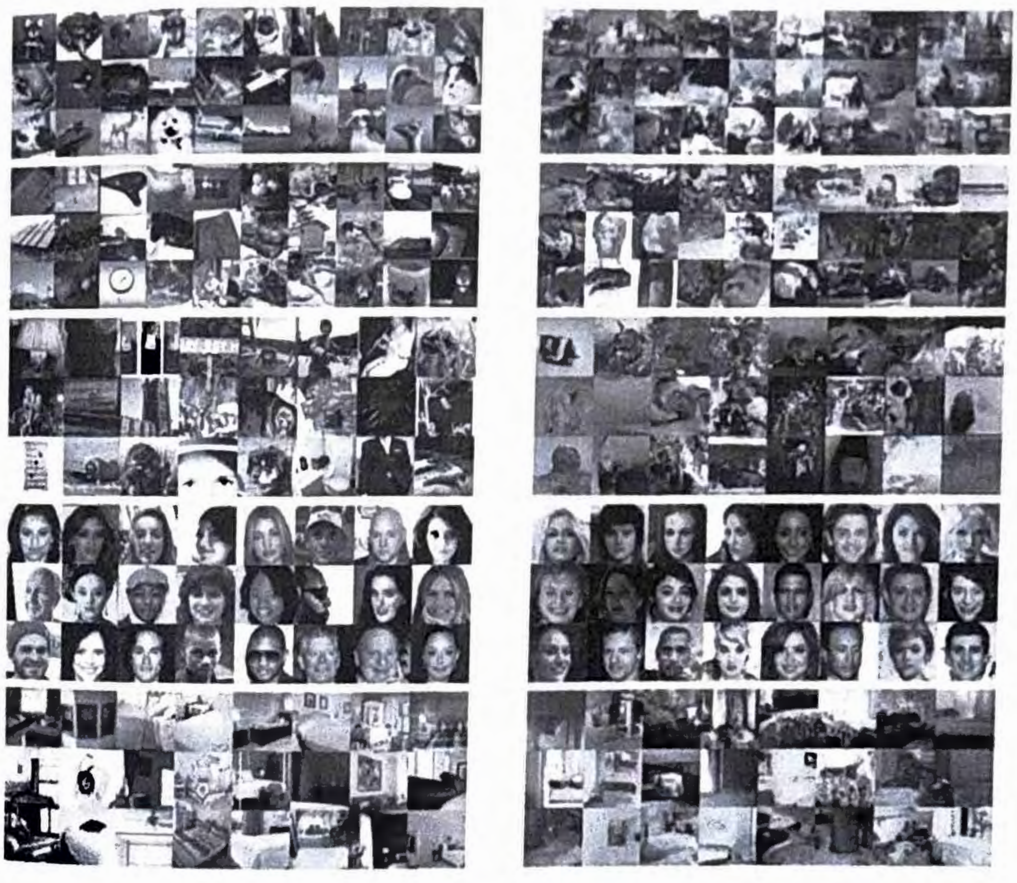


Figure 7.5 Samples generated by the Real-NVP model across four datasets: CIFAR-10, ImageNet (32 × 32), ImageNet (64 × 64), CelebA, and LSUN (bedroom). For each dataset (subfigures), the left column shows real examples, while the right column displays samples generated by the model trained on the respective dataset. *Source:* Dinh, Sohl-Dickstein, and Bengio (2017).

The authors of this work employed a multi-scale architecture, as described in Section 3.6 of Dinh et al. (2017). This architecture incorporates a sequence of coupling-squeezing coupling layers, utilizing deep convolutional residual networks within the coupling transformations. For a more detailed explanation of the architecture used in the original experiments, we encourage readers to refer to the work by Dinh et al. (2017).

As observed, this model can generate much more realistic samples, including images of various objects, bedrooms, natural scenes, and human faces.

Another interesting aspect demonstrated in the RealNVP paper is the model's ability for representation learning. Recall from the beginning of the chapter that one limitation of these models being invertible is that the dimensions of the latent variable **z** and the observed variable **x** must be the same, with **x** being high dimensional. Normally, we prefer the latent dimension to be lower than the data dimension, as in the VAEs studied in Chapter 6, to provide a compressed representation of the input data.

In Figure 7.6, the authors showcase Real-NVP's ability to capture a useful representation of the data in the latent space. This figure shows new images generated by interpolations in the latent space between four examples for different datasets. The datasets used, in clockwise order from the top left, are CelebA, ImageNet (64 × 64), LSUN (tower), and LSUN (bedroom).

For each dataset, four examples are placed at the corners of each subimage, and the interpolations between them are generated. The results suggest that even if the latent space is high dimensional, the model still learns an interesting structure, where similar images in the data space are also close in the latent space. This indicates that the model is capturing a meaningful and semantic representation that goes beyond simple pixel-space interpolation.

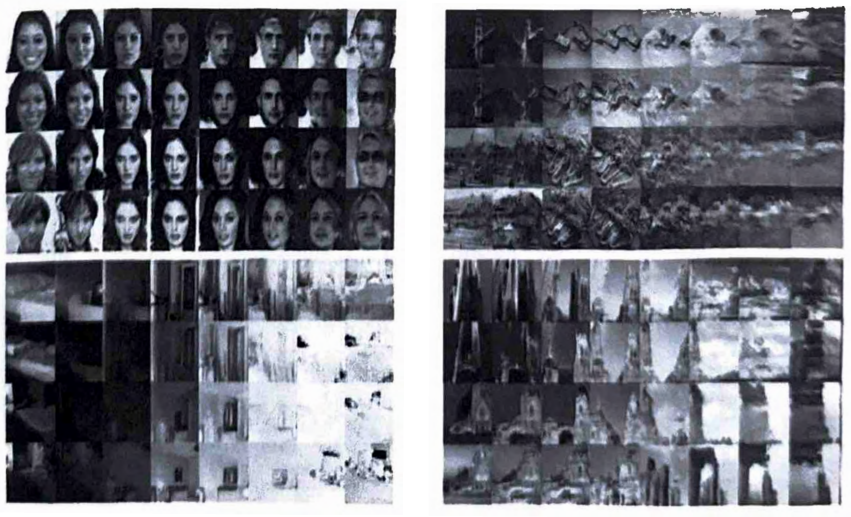


Figure 7.6 Illustration of new images generated through interpolations between four examples from the dataset. The datasets used are, in clockwise order from the top left: CelebA, ImageNet (64 × 64), LSUN (tower), and LSUN (bedroom). *Source:* Dinh, Sohl-Dickstein, and Bengio (2017).

To further test whether the latent space has a consistent semantic interpretation, the authors trained a class-conditional model on the CelebA dataset. They found that the learned representation had a consistent semantic meaning across class labels, as described in Dinh et al. (2017).

In the accompanying notebook for this chapter ([*RealNVP Notebook*]), we implement and test the Real-NVP coupling flow with custom architectures for generating equity financial data, which can be easily customized for your own applications. We will discuss how to extend these models to capture temporal dependencies in Section 7.8, along with an application to financial data in Section 7.8.3.

In the following sections, we will briefly introduce two additional approaches for constructing flows: **Autoregressive Flows** and **Continuous Normalizing Flows**.

7.6 Autoregressive Flows

Another type of flow model is known as **Autoregressive Flows**, which are, in fact, one of the first classes of flows developed (see Papamakarios et al. [2021]). These flows are motivated by the idea that the joint distribution over $\mathbf{x}$ can be factorized into a product of one-dimensional conditional probability distributions using the chain rule, as introduced in Chapter 5 on autoregressive models. By parameterizing each conditional distribution, these factors form the basis of a normalizing flow known as the **Masked Autoregressive Flow (MAF)** (see Papamakarios et al. [2017]; Bishop and Bishop [2023]).

Some of our experiments in Section 7.8 will incorporate MAF flows as building blocks in a model's architecture. Regarding the modeling flexibility of Autoregressive flows, an interesting result, presented in Papamakarios et al. (2021), shows that all autoregressive models for continuous variables can be interpreted as autoregressive flows with a single autoregressive layer. If you want to know more about this result, refer to Papamakarios et al. (2021) for further details.

7.7 Continuous Normalizing Flows

Another type of flow model is known as **Continuous Normalizing Flows**, which take a different approach to define the mapping from $\mathbf{z}$ to $\mathbf{x}$. In the previous sections, we discussed constructing flows by parameterizing a function f and applying a finite sequence of transformations (e.g., K coupling layers in Real-NVP) to transform the base measure $p(\mathbf{z})$ into a more complex distribution $p(\mathbf{x})$.

In contrast, Continuous Normalizing Flows view the mapping from $\mathbf{z}$ to $\mathbf{x}$ as a **continuous process**. Instead of applying a finite number of discrete transformations, the transformation can be thought of, loosely speaking, as applying an "infinite number of infinitesimal transformations." This process is mathematically described by a **Neural Ordinary Differential Equation (Neural ODE)**. In this framework, the flow is defined by an ODE, and during the learning process, the parameters of the ODE are learned from the data, rather than learning the parameters of individual discrete flows, as in the previous cases we studied. Continuous Normalizing Flows can be used to generate the complex continuous stochastic process of financial data (see Deng et al. [2020]) For more details on Continuous Normalizing Flows, we encourage the reader to take a look at Chen et al. (2018) and Papamakarios et al. (2021).

7.8 Modeling Financial Time Series with Flow Models

7.8.1 *Transitioning from Image Data to Time-series Dynamics*

In earlier sections, we introduced two flow models, NICE and Real-NVP. These models create flows that are differentiable, invertible, and allow for quick computation of log-likelihoods. The efficiency in log-likelihood computation is mainly due to the structure imposed in these flows, which simplifies the computation of the determinant of the Jacobian. While so far we had used image data as examples to demonstrate the capacity of these models to approximate complex data distributions (and to highlight the field where these models were originally developed), flows are not restricted to this domain and can be used whenever we need to model complex probability distributions. They can be extended to other areas, such as time-series modeling, unlocking new modeling opportunities.

To transition into time series, we can start by simply giving another interpretation to the data $\mathbf{x}$. Instead of thinking as $\mathbf{x}_n$ as a vector of pixel intensities for the n-th image in our dataset, we can think of $\mathbf{x}_n$ as a vector representing multivariate time series at time n. Each component or "dimension" might correspond to a variable, such as the returns of a given stock at time n, with the vector $\mathbf{x}_n$ providing a snapshot of the cross-sectional returns at that time. For simplicity, we will use t to index time instead of n, as it is more intuitive in the context of time series.

In this setup, flows like NICE or Real-NVP can model the cross-sectional relationships (i.e., correlations) among the time series at time t. However, by design, these architectures assume independence between consecutive observations. Recall how sampling works in standard flow models: to generate the observations $\mathbf{x}_t$, we sample $\mathbf{z}_t$ from the base measure, apply the flow to it to produce $\mathbf{x}_t$, and repeat this process for every t. This assumption of independence can limit the performance of these models at certain time scales, where dependencies between observations may be significant.

7.8.2 *Adapting Flows for Time Series*

Conventional flows like NICE and Real-NVP are good at modeling cross-sectional dynamics but impose independence between observations by design. This might not be enough for modeling financial time series at certain time granularities, where past prices/returns, market conditions, and other factors significantly impact future prices/returns.

So, how can we adapt flows to capture the temporal dynamics (dependencies over time) present in the data we want to model? Here are a few ideas:

1. Latent state dynamics: Introduce temporal dependencies into the latent variables, such that $\mathbf{z}_t$ depends on past latent variables $\mathbf{z}_{t-1}, \mathbf{z}_{t-2}, \ldots$, either deterministically or stochastically.

$$\begin{aligned} \mathbf{z}_t &\sim p_{\mathbf{z}_t \mid \mathbf{z}_{t-1}, \mathbf{z}_{t-2}, \ldots} \\ \mathbf{x}_t &= f(\mathbf{z}_t) \end{aligned}$$

   This mirrors the idea behind models like the Kalman filter, or RNN, where latent states depends on the previous latent space.
2. AR-style dependencies: Extend the generative process to depend on past observations, making $\mathbf{x}_t$ a function of both the latent variable $\mathbf{z}_t$ and previous observations $\mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \ldots,$

$$\mathbf{x}_t = f\left(\mathbf{z}_t, \phi(\mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \ldots, \mathbf{x}_{t-p})\right)$$

   where ϕ is a function that can encode the effect of past p observations on $\mathbf{x}_t$. This approach mirrors with autoregressive models.
3. Hybrid approaches: Combine the two methods, allowing $\mathbf{x}_t$ to depend both of past observations and past latent variables

$$\mathbf{x}_t = f\left(\mathbf{z}_t, \phi(\mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \ldots, \mathbf{x}_{t-p}), \psi(\mathbf{z}_{t-1}, \mathbf{z}_{t-2}, \ldots, \mathbf{z}_{t-q})\right)$$

   where ϕ encodes the effect of past p observations and ψ the effect of past q latent variables.
4. Time-varying flows: allow the flow to vary with time f_t.

7.8.3 *Case Study: A Practical Example—Conditioned Normalizing Flows*

To demonstrate these ideas, let's explore practical implementations of flows adapted for modeling time-series dynamic. One notable example is presented in the paper *Multivariate Probabilistic Time Series Forecasting via Conditioned Normalizing Flows* (Rasul et al., 2021). In this work, the authors use Real-NVP flows but adapt them to handle temporal dependencies. They key idea is to condition the flow transformation f on a hidden state $\mathbf{h}_t$, which encodes information about the past:

$$\mathbf{x}_t = f(\mathbf{z}_t, \mathbf{h}_t)$$

So, how do we get $\mathbf{h}_t$? The authors explore two approaches.

1. RNN-based hidden state: the first option is to use an RNN (like an LSTM or GRU) to summarize past observations and covariates into $\mathbf{h}_t$:

$$\mathbf{h}_t = \mathrm{RNN}(\mathrm{concat}(\mathbf{x}_{t-1}, \mathbf{c}_{t-1}), \mathbf{h}_{t-1})$$

Here $\mathbf{c}_{t-1}$ includes any additional covariates (i.e., in financial time series think like additional market indicators or macroeconomic variables, or categorical variables).

2. Transformer-based hidden state: The second option is to use an encoder-decoder transformer architecture. Here, the encoder embeds the past observations and covariates, while the decoder generates the hidden states for the flow.

As we know, both RNNs and Transformers are effective at modeling sequential data, with Transformers being particularly better than RNN for capturing long-range dependencies.

At this point, you might notice that we are only limited by our creativity when designing architectures to capture temporal dynamics with flows. This is another example of how the building blocks we've covered in previous chapters come together—combining flows with powerful sequential models like RNNs or Transformers to achieve the same goal.

Exploring examples like this one can bring to life many ideas for customizing your own solutions. You might consider training the model differently, adding or removing components, or adapting the architecture to better fit your specific needs.

7.8.3.1 Importance of Domain Knowledge in Financial Time Series. In financial time-series modeling, incorporating domain knowledge during the modeling process is critical. This field is well known for its low signal-to-noise ratio and non-stationarity is all over the place, which violates the core assumptions of independence and identical distribution that underlie most machine learning solutions. These challenges make it difficult to achieve good results with generic models.

This means that most of the time we can't simply throw a generic model at the data and expect great outcomes in this domain. Who knows—perhaps in the near future, maybe even a few months after this book is published, this might change. For now, let's focus on customizing our solutions to the specific problem at hand so we can extract valuable insights and, hopefully, some profit.

In the accompanying notebook for this chapter, we explore these ideas using the Exchange Rate dataset, which contains daily exchange rates for the currencies of eight countries (Australia, the United Kingdom, Canada, Switzerland, China, Japan, New Zealand, and Singapore) from 1990 to 2016. We will walk through the main components of the implementation and discuss variations in the architecture and its performance on other datasets.

7.9 Conclusion

In this chapter, we have seen how flow models are a powerful addition to our toolbox for handling high-dimensional data. They combine the fast sampling capabilities of traditional latent variable models with the key advantage of exact likelihood computation. By designing flows with the right structure, we can make computations like the determinant of the log Jacobian very efficient, which is a key element for likelihood computation.

We also explored how to extend flows to capture temporal dependencies. This opens up opportunities to combine flow models with architectures that we know excels at this task, such as autoregressive models based on RNN and Transformers, both of which work great at modeling sequential data.

By mastering flow models, you add a flexible and efficient tool to your modeling toolbox, giving you another great option for modeling high-dimensional multimodal data.