Chapter 3
Flow-Based Models

3.1 Flows for Continuous Random Variables

3.1.1 Introduction

So far, we have discussed a class of deep generative models that model the distribution $p(\mathbf{x})$ directly in an autoregressive manner. The main advantage of ARMs is that they can learn long-range statistics and, in a consequence, powerful density estimators. However, their drawback is that they are parameterized in an autoregressive manner, hence, sampling is rather a slow process. Moreover, they lack a latent representation, therefore, it is not obvious how to manipulate their internal data representation that makes it less appealing for tasks like compression or metric learning. In this chapter, we present a different approach to direct modeling of $p(\mathbf{x})$. However, before we start our considerations, we will discuss a simple example.

Example 3.1 Let us take a random variable $z \in \mathbb{R}$ with $\pi(z) = \mathcal{N}(z|0, 1)$. Now, we consider a new random variable after applying some linear transformation to z, namely $x = 0.75z + 1$. Now the question is the following:

What is the distribution of x, $p(x)$?

We can guess the solution by using properties of Gaussians, or dig in our memory about the **change of variables formula** to calculate this distribution, that is:

$$p(x) = \pi\left(z = f^{-1}(x)\right)\left|\frac{\partial f^{-1}(x)}{\partial x}\right|, \tag{3.1}$$

where f is an invertible function (a bijection). What does it mean? It means that the function maps one point to another, distinctive point, and we can always invert the function to obtain the original point.

J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-030-93158-2_3

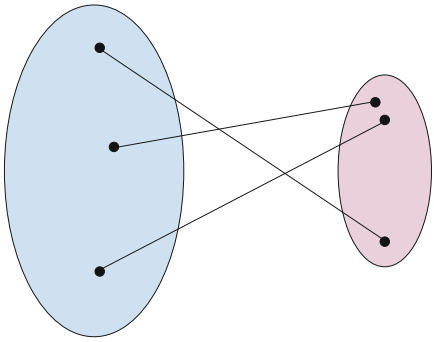


Fig. 3.1 An example of a bijection where for each point in the blue set there is precisely one corresponding point in the purple set (and vice versa)

In Fig. 3.1, we have an example of a bijection. Notice that volumes of the domains do not need to be the same! Keep it in mind and think about it in the context of $\left|\frac{\partial f^{-1}(x)}{\partial x}\right|$.

Coming back to our example, we have

$$f(z) = 0.75z + 1, \tag{3.2}$$

and the inverse of f is

$$f^{-1}(x) = \frac{x-1}{0.75}. \tag{3.3}$$

Then, the derivative of the **change of volume** is

$$\left|\frac{\partial f^{-1}(x)}{\partial x}\right| = \frac{4}{3}. \tag{3.4}$$

Putting all information so far together yields

$$p(x) = \pi\left(z = \frac{x-1}{0.75}\right)\frac{4}{3} = \frac{1}{\sqrt{2\pi\ 0.75^2}}\exp\left\{-(x-1)^2/0.75^2\right\}. \tag{3.5}$$

We immediately realize that we end up with the Gaussian distribution again:

$$p(x) = \mathcal{N}(x|1, 0.75). \tag{3.6}$$

Moreover, we see that the part $\left|\frac{\partial f^{-1}(x)}{\partial x}\right|$ is responsible to **normalize** the distribution $\pi(z)$ after applying the transformation f. In other words, $\left|\frac{\partial f^{-1}(x)}{\partial x}\right|$ counteracts a possible *change of volume* caused by f. ■

First of all, this example indicates that we can calculate a new distribution of a continuous random variable by applying a known bijective transformation f to a

random variable with a known distribution, $z \sim p(z)$. The same holds for multiple variables $\mathbf{x}, \mathbf{z} \in \mathbb{R}^D$:

$$p(\mathbf{x}) = p\left(\mathbf{z} = f^{-1}(\mathbf{x})\right) \left| \frac{\partial f^{-1}(\mathbf{x})}{\partial \mathbf{x}} \right|, \tag{3.7}$$

where:

$$\left| \frac{\partial f^{-1}(\mathbf{x})}{\partial \mathbf{x}} \right| = \left| \det \mathbf{J}_{f^{-1}}(\mathbf{x}) \right| \tag{3.8}$$

is the Jacobian matrix $\mathbf{J}_{f^{-1}}$ that is defined as follows:

$$\mathbf{J}_{f^{-1}} = \begin{bmatrix} \frac{\partial f_1^{-1}}{\partial x_1} & \cdots & \frac{\partial f_1^{-1}}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_D^{-1}}{\partial x_1} & \cdots & \frac{\partial f_D^{-1}}{\partial x_D} \end{bmatrix}. \tag{3.9}$$

Moreover, we can also use the **inverse function theorem** that yields

$$\left| \mathbf{J}_{f^{-1}}(\mathbf{x}) \right| = \left| \mathbf{J}_f(\mathbf{x}) \right|^{-1}. \tag{3.10}$$

Since f is invertible, we can use the inverse function theorem to rewrite (3.7) as follows:

$$p(\mathbf{x}) = p\left(\mathbf{z} = f^{-1}(\mathbf{x})\right) \left| \mathbf{J}_f(\mathbf{x}) \right|^{-1}. \tag{3.11}$$

To get some insight into the role of the Jacobian-determinant, take a look at Fig. 3.2. Here, there are three cases of invertible transformations that play around with a uniform distribution defined over a square.

In the case on top, the transformation turns a square into a rhombus without changing its volume. As a result, the Jacobian-determinant of this transformation is 1. Such transformations are called **volume-preserving**. Notice that the resulting distribution is still uniform and since there is no change of volume, it is defined over the same volume as the original one, thus, the color is the same.

In the middle, the transformation shrinks the volume, therefore, the resulting uniform distribution is "denser" (a darker color in Fig. 3.2). Additionally, the Jacobian-determinant is smaller than 1.

In the last situation, the transformation enlarges the volume, hence, the uniform distribution is defined over a larger area (a lighter color in Fig. 3.2). Since the volume is larger, the Jacobian-determinant is larger than 1.

Notice that shifting operator is volume-preserving. To see that imagine adding an arbitrary value (e.g., 5) to all points of the square. Does it change the volume? Not at all! Thus, the Jacobian-determinant equals 1.

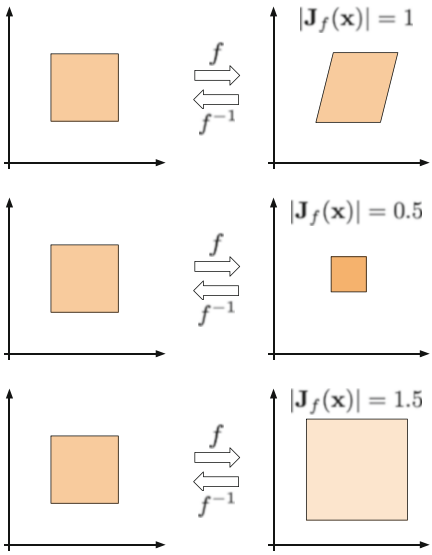


Fig. 3.2 Three examples of invertible transformations: (*top*) a volume-preserving bijection, (*middle*) a bijection that shrinks the original area, (*bottom*) a bijection that enlarges the original area

3.1.2 Change of Variables for Deep Generative Modeling

A natural question is whether we can utilize the idea of the change of variables to model a complex and high-dimensional distribution over images, audio, or other data sources. Let us consider a hierarchical model, or, equivalently, a sequence of invertible transformations, $f_k : \mathbb{R}^D \rightarrow \mathbb{R}^D$. We start with a known distribution $\pi(\mathbf{z}_0) = \mathcal{N}(\mathbf{z}_0|0, \mathbf{I})$. Then, we can sequentially apply the invertible transformations to obtain a flexible distribution [1, 2]:

$$p(\mathbf{x}) = \pi\left(\mathbf{z}_0 = f^{-1}(\mathbf{x})\right) \prod_{i=1}^{K} \left| \det \frac{\partial f_i\left(\mathbf{z}_{i-1}\right)}{\partial \mathbf{z}_{i-1}} \right|^{-1}, \tag{3.12}$$

or by using the notation of a Jacobian for the i-th transformation:

$$p(\mathbf{x}) = \pi\left(\mathbf{z}_0 = f^{-1}(\mathbf{x})\right) \prod_{i=1}^{K} \left|\mathbf{J}_{f_i}(\mathbf{z}_{i-1})\right|^{-1}. \tag{3.13}$$

An example of transforming a unimodal base distribution like Gaussian into a multimodal distribution through invertible transformations is presented in Fig. 3.3. In principle, we should be able to get almost any arbitrarily complex distribution and revert to a *simple* one.

Let $\pi(\mathbf{z}_0)$ be $\mathcal{N}(\mathbf{z}_0|0, \mathbf{I})$. Then, the logarithm of $p(\mathbf{x})$ is the following:

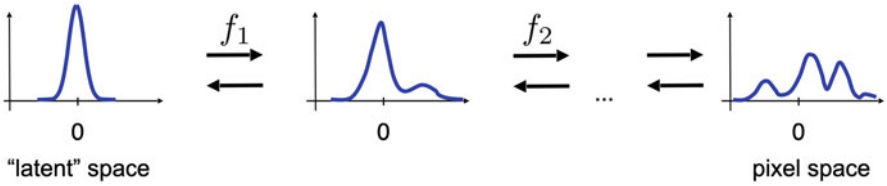


Fig. 3.3 An example of transforming a unimodal distribution (the latent space) to a multimodal distribution (the data space, e.g., the pixel space) through a series of invertible transformations f_i

$$\ln p(\mathbf{x}) = \ln \mathcal{N}\left(\mathbf{z}_0 = f^{-1}(\mathbf{x})|0, \mathbf{I}\right) - \sum_{i=1}^{K} \ln \left|\mathbf{J}_{f_i}(\mathbf{z}_{i-1})\right| . \tag{3.14}$$

Interestingly, we see that the first part, namely $\ln \mathcal{N}\left(\mathbf{z}_0 = f^{-1}(\mathbf{x})|0, \mathbf{I}\right)$, corresponds to the *Mean Square Error* loss function between 0 and $f^{-1}(\mathbf{x})$ plus a constant. The second part, $\sum_{i=1}^{K} \ln \left|\mathbf{J}_{f_i}(\mathbf{z}_{i-1})\right|$, as in our example, ensures that the distribution is properly normalized. However, since it penalizes the change of volume (take a look again at the example above!), we can think of it as a kind of a *regularizer* for the invertible transformations $\{f_i\}$.

Once we have laid down the foundations of the change of variables for expressing density functions, now we must face two questions:

- How to model the invertible transformations?
- What is the difficulty here?

The answer to the first question could be neural networks because they are flexible and easy-to-train. However, we cannot take **any** neural network because of two reasons. First, the transformation must be **invertible**, thus, we must pick an **invertible neural network**. Second, even if a neural network is invertible, we face a problem of calculating the second part of (3.14), i.e., $\sum_{i=1}^{K} \ln \left|\mathbf{J}_{f_i}(\mathbf{z}_{i-1})\right|$, that is non-trivial and computationally intractable for an arbitrary sequence of invertible transformations. As a result, we seek for such neural networks that are both invertible and the logarithm of a Jacobian-determinant is (relatively) easy to calculate. The resulting model that consists of invertible transformations (neural networks) with tractable Jacobian-determinants is referred to as *normalizing flows* or *flow-based models*.

There are various possible invertible neural networks with tractable Jacobian-determinants, e.g., Planar Normalizing Flows [1], Sylvester Normalizing Flows [3], Residual Flows [4, 5], Invertible DenseNets [6]. However, here we focus on a very important class of models: **RealNVP**, *Real-valued Non-Volume Preserving* flows [7] that serve as a starting point for many other flow-based generative models (e.g., GLOW [8]).

3.1.3 Building Blocks of RealNVP

3.1.3.1 Coupling Layers

The main component of RealNVP is a *coupling layer*. The idea behind this transformation is the following. Let us consider an input to the layer that is divided into two parts: $\mathbf{x} = [\mathbf{x}_a, \mathbf{x}_b]$. The division into two parts could be done by dividing the vector $\mathbf{x}$ into $\mathbf{x}_{1:d}$ and $\mathbf{x}_{d+1:D}$ or according to a more sophisticated manner, e.g., a *checkerboard pattern* [7]. Then, the transformation is defined as follows:

$$\mathbf{y}_a = \mathbf{x}_a \tag{3.15}$$

$$\mathbf{y}_b = \exp\left(s\left(\mathbf{x}_a\right)\right) \odot \mathbf{x}_b + t\left(\mathbf{x}_a\right), \tag{3.16}$$

where $s(\cdot)$ and $t(\cdot)$ are **arbitrary neural networks** called *scaling* and *transition*, respectively.

This transformation is invertible by design, namely:

$$\mathbf{x}_b = \left(\mathbf{y}_b - t\left(\mathbf{y}_a\right)\right) \odot \exp\left(-s(\mathbf{y}_a)\right) \tag{3.17}$$

$$\mathbf{x}_a = \mathbf{y}_a. \tag{3.18}$$

Importantly, the logarithm of the Jacobian-determinant is easy to calculate, because:

$$\mathbf{J} = \begin{bmatrix} \mathbf{I}_{d\times d} & \mathbf{0}_{d\times(D-d)} \\ \frac{\partial \mathbf{y}_b}{\partial \mathbf{x}_a} & \operatorname{diag}\left(\exp\left(s\left(\mathbf{x}_a\right)\right)\right) \end{bmatrix} \tag{3.19}$$

that yields

$$\det(\mathbf{J}) = \prod_{j=1}^{D-d} \exp\left(s\left(\mathbf{x}_a\right)\right)_j = \exp\left(\sum_{j=1}^{D-d} s\left(\mathbf{x}_a\right)_j\right). \tag{3.20}$$

Eventually, coupling layers seem to be flexible and powerful transformations with tractable Jacobian-determinants! However, we process only half of the input, therefore, we must think of an appropriate additional transformation a coupling layer could be combined with.

3.1.3.2 Permutation Layers

A simple yet effective transformation that could be combined with a coupling layer is a **permutation layer**. Since permutation is *volume-preserving*, i.e., its Jacobian-

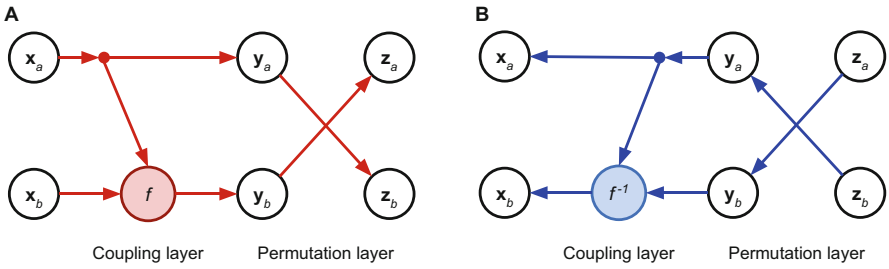


Fig. 3.4 A combination of a coupling layer and a permutation layer that transforms $[\mathbf{x}_a, \mathbf{x}_b]$ to $[\mathbf{z}_a, \mathbf{z}_b]$. (**a**) A forward pass through the block. (**b**) An inverse pass through the block

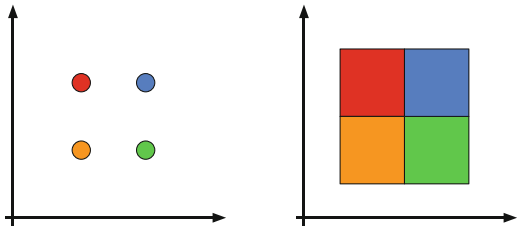


Fig. 3.5 A schematic representation of the uniform dequantization for two binary random variables: (*left*) the probability mass is assigned to points, (*right*) after the uniform dequantization, the probability mass is assigned to square areas. Colors correspond to probability values

determinant is equal to 1, we can apply it each time after the coupling layer. For instance, we can reverse the order of variables.

An example of an invertible block, i.e., a combination of a coupling layer with a permutation layer is schematically presented in Fig. 3.4.

3.1.3.3 Dequantization

As discussed so far, flow-based models assume that $\mathbf{x}$ is a vector of real-valued random variables. However, in practice, many objects are discrete. For instance, images are typically represented as integers taking values in $\{0, 1, \ldots, 255\}^D$. In [9], it has been outlined that adding a uniform noise, $\mathbf{u} \in [-0.5, 0.5]^D$, to original data, $\mathbf{y} \in \{0, 1, \ldots, 255\}^D$, allows applying density estimation to $\mathbf{x} = \mathbf{y} + \mathbf{u}$. This procedure is known as *uniform dequantization*. Recently, there were different schema of dequantization proposed, you can read more on that in [10].

An example for two binary random variables and the uniform dequantization is depicted in Fig. 3.5. After adding $\mathbf{u} \in [-0.5, 0.5]^2$ to each discrete value, we obtain a continuous space and now probabilities originally associated with volumeless points are "spread" across small square regions.

3.1.4 Flows in Action!

Let us turn math into a code! We will first discuss the log-likelihood function (i.e., the learning objective) and how mathematical formulas correspond to the code. First, it is extremely important to know what is our learning objective, i.e., the log-likelihood function. In the example, we use coupling layers as described earlier, together with permutation layers. Then, we can plug the logarithm of the Jacobian-determinant for the coupling layers (for the permutation layers it is equal to 1, so $\ln(1) = 0$) in Eq. (3.14) that yields

$$\ln p(\mathbf{x}) = \ln \mathcal{N}\left(\mathbf{z}_0 = f^{-1}(\mathbf{x})|0, \mathbf{I}\right) - \sum_{i=1}^{K} \left(\sum_{j=1}^{D-d} s_k \left(\mathbf{x}_a^k\right)_j \right), \tag{3.21}$$

where s_k is the scale network in the k-th coupling layer, and $\mathbf{x}_a^k$ denotes the input to the k-th coupling layer. Notice that exp in the log-Jacobian-determinant is cancelled by applying the logarithm.

Let us think again about the learning objective from the implementation perspective. First, we definitely need to obtain $\mathbf{z}$ by calculating $f^{-1}(\mathbf{x})$, and then we can calculate $\ln \mathcal{N}\left(\mathbf{z}_0 = f^{-1}(\mathbf{x})|0, \mathbf{I}\right)$. That is actually easy, and we get

$$\ln \mathcal{N}\left(\mathbf{z}_0 = f^{-1}(\mathbf{x})|0, \mathbf{I}\right) = -const - \frac{1}{2}\|f^{-1}(\mathbf{x})\|^2, \tag{3.22}$$

where $const = \frac{D}{2}\ln(2\pi)$ is the normalizing constant of the standard Gaussian, and $\frac{1}{2}\|f^{-1}(\mathbf{x})\|^2 = MSE(0, f^{-1}(\mathbf{x}))$.

Alright, now we should look into the second part of the objective, i.e., the log-Jacobian-determinants. As we can see, we have a sum over transformations, and for each coupling layer, we consider only the outputs of the scale nets. Hence, the only thing we must remember during implementing the coupling layers is to return not only output but also the outcome of the scale layer too.

3.1.5 Code

Now, we have all components to implement our own RealNVP! Below, there is a code with a lot of comments that should help to understand every single line of it. The full code (with auxiliary functions) that you can play with is available here: https://github.com/jmtomczak/intro_dgm.

```
1 class RealNVP(nn.Module):
2     def __init__(self, nets, nett, num_flows, prior, D=2,
      dequantization=True):
3         super(RealNVP, self).__init__()
```

```
4
5           # Well, it's always good to brag about yourself.
6           print('RealNVP by JT.')
7
8           # We need to dequantize discrete data. This attribute is
    used during training to dequantize integer data.
9           self.dequantization = dequantization
10
11          # An object of a prior (here: torch.distribution of
    multivariate normal distribution)
12          self.prior = prior
13          # A module list for translation networks
14          self.t = torch.nn.ModuleList([nett() for _ in range(
    num_flows)])
15          # A module list for scale networks
16          self.s = torch.nn.ModuleList([nets() for _ in range(
    num_flows)])
17          # The number of transformations, in our equations it is
    denoted by K.
18          self.num_flows = num_flows
19
20          # The dimensionality of the input. It is used for
    sampling.
21          self.D = D
22
23      # This is the coupling layer, the core of the RealNVP model.
24      def coupling(self, x, index, forward=True):
25          # x: input, either images (for the first transformation)
    or outputs from the previous transformation
26          # index: it determines the index of the transformation
27          # forward: whether it is a pass from x to y (forward=True
    ), or from y to x (forward=False)
28
29          # We chunk the input into two parts: x_a, x_b
30          (xa, xb) = torch.chunk(x, 2, 1)
31
32          # We calculate s(xa), but without exp!
33          s = self.s[index](xa)
34          # We calculate t(xa)
35          t = self.t[index](xa)
36
37          # Calculate either the forward pass (x -> z) or the
    inverse pass (z -> x)
38          # Note that we use the exp here!
39          if forward:
40              #yb = f^{-1}(x)
41              yb = (xb - t) * torch.exp(-s)
42          else:
43              #xb = f(y)
44              yb = torch.exp(s) * xb + t
45
46          # We return the output y = [ya, yb], but also s for
    calculating the log-Jacobian-determinant
47          return torch.cat((xa, yb), 1), s
```

```
48
49      # An implementation of the permutation layer
50      def permute(self, x):
51          # Simply flip the order.
52          return x.flip(1)
53
54      def f(self, x):
55          # This is a function that calculates the full forward
        pass through the coupling+permutation layers.
56          # We initialize the log-Jacobian-det
57          log_det_J, z = x.new_zeros(x.shape[0]), x
58          # We iterate through all layers
59          for i in range(self.num_flows):
60              # First, do coupling layer,
61              z, s = self.coupling(z, i, forward=True)
62              # then permute.
63              z = self.permute(z)
64              # To calculate the log-Jacobian-determinant of the
        sequence of transformations we sum over all of them.
65              # As a result, we can simply accumulate individual
        log-Jacobian determinants.
66              log_det_J = log_det_J - s.sum(dim=1)
67          # We return both z and the log-Jacobian-determinant,
        because we need z to feed in to the logarithm of the Norma;
68          return z, log_det_J
69
70      def f_inv(self, z):
71          # The inverse path: from z to x.
72          # We apply all transformations in the reversed order.
73          x = z
74          for i in reversed(range(self.num_flows)):
75              x = self.permute(x)
76              x, _ = self.coupling(x, i, forward=False)
77          # Since we use this function for sampling, we don't need
        to return anything else than x.
78          return x
79
80      def forward(self, x, reduction='avg'):
81          # This function is essential for PyTorch.
82          # First, we calculate the forward part: from x to z, and
        also we need the log-Jacobian-determinant.
83          z, log_det_J = self.f(x)
84          # We can use either sum or average as the output.
85          # Either way, we calculate the learning objective: self.
        prior.log_prob(z) + log_det_J.
86          # NOTE: Mind the minus sign! We need it, because, by
        default, we consider the minimization problem,
87          # but normally we look for the maximum likelihood
        estimate. Therefore, we use:
88          # max F(x) <=> min -F(x)
89          if reduction == 'sum':
90              return -(self.prior.log_prob(z) + log_det_J).sum()
91          else:
92              return -(self.prior.log_prob(z) + log_det_J).mean()
```

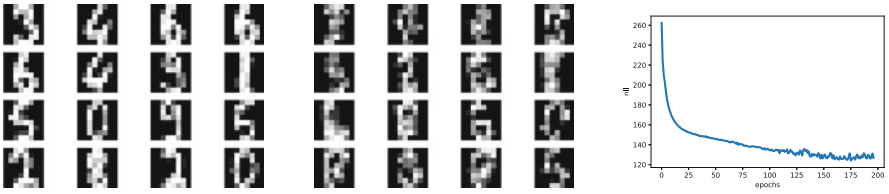
```
93
94      def sample(self, batchSize):
95          # First, we sample from the prior, z ~ p(z) = Normal(z
        |0,1)
96          z = self.prior.sample((batchSize, self.D))
97          z = z[:, 0, :]
98          # Second, we go from z to x.
99          x = self.f_inv(z)
100         return x.view(-1, self.D)
```

Listing 3.1 An example of an implementation of RealNVP

```
# The number of flows
num_flows = 8

# Neural networks for a single transformation (a single flow).
nets = lambda: nn.Sequential(nn.Linear(D//2, M), nn.LeakyReLU(),
                             nn.Linear(M, M), nn.LeakyReLU(),
                             nn.Linear(M, D//2), nn.Tanh())

nett = lambda: nn.Sequential(nn.Linear(D//2, M), nn.LeakyReLU(),
                             nn.Linear(M, M), nn.LeakyReLU(),
                             nn.Linear(M, D//2))

# For the prior, we can use the built-in PyTorch distribution.
prior = torch.distributions.MultivariateNormal(torch.zeros(D),
    torch.eye(D))

# Init of the RealNVP. Please note that we need to dequantize the
     data (i.e., uniform dequantization).
model = RealNVP(nets, nett, num_flows, prior, D=D, dequantization
    =True)
```

Listing 3.2 An example of networks

Et voila! Now we are ready to run the full code. After training our RealNVP, we should obtain results resembling those in Fig. 3.6.



A B C

Fig. 3.6 An example of outcomes after the training: (**a**) Randomly selected real images. (**b**) Unconditional generations from the RealNVP. (**c**) The validation curve during training

3.1.6 Is It All? Really?

Yes and no. Yes in the sense it is the minimalistic example of an implementation of the RealNVP. No, because there are many improvements over the instance of the RealNVP presented here, namely:

- *Factoring-out* [7]: During the forward pass (from $\mathbf{x}$ to $\mathbf{z}$), we can split the variables and proceed with processing only a subset of them. This could help to parameterize the base distribution by using the outputs of intermediate layers. In other words, we can obtain an autoregressive base distribution.
- *Rezero trick* [11]: Introducing additional parameters to the coupling layer, e.g., $\mathbf{y}_b = \exp(\alpha s(\mathbf{x}_a)) \odot \mathbf{x}_b + \beta t(\mathbf{x}_a)$ and α, β are initialized with 0's. This helps to ensure that the transformations act as identity maps in the beginning. It is shown in [12] that this trick helps to learn better transformations by maintaining information about the input through all layers in the beginning of the training process.
- *Masking* or *Checkerboard pattern* [7]: We can use a checkerboard pattern instead of dividing an input into two parts like $[\mathbf{x}_{1:D/2}, \mathbf{x}_{D/2+1:D}]$. This encourages learning local statistics better.
- *Squeezing* [7]: We can also play around with "squeezing" some dimensions. For instance, an image that consists of C channels, width W, and height H could be turned into $4C$ channels, width $W/2$, and height $H/2$.
- *Learnable base distributions*: instead of using a standard Gaussian base distribution, we can consider another model for that, e.g., an autoregressive model.
- *Invertible 1x1 convolution* [8]: A fixed permutation could be replaced with a (learned) invertible 1x1 convolution as in the GLOW model [8].
- *Variational dequantization* [13]: We can also pick a different dequantization scheme, e.g., variational dequantization. This allows to obtain much better scores. However, it is not for free because it leads to a lower bound to the log-likelihood function.

Moreover, there are many new fascinating research directions! I will name them here and point to papers where you can find more details:

- *Data compression with flows* [14]: Flow-based models are perfect candidates for compression since they allow to calculate the exact likelihood. Ho et al. [14] proposed a scheme that allows to use flows in the bit-back-like compression scheme.
- *Conditional flows* [15–17]: Here, we present the unconditional RealNVP. However, we can use a flow-based model for conditional distributions. For instance, we can use the conditioning as an input to the scale network and the translation network.
- *Variational inference with flows* [1, 3, 18–21]: Conditional flow-based models could be used to form a flexible family of variational posteriors. Then, the lower bound to the log-likelihood function could be tighter. We will come back to that in Chap. 4, Sect. 4.4.2.

- *Integer discrete flows* [12, 22, 23]: Another interesting direction is a version of the RealNVP for integer-valued data. We will explain this idea in Sect. 3.2.
- *Flows on manifolds* [24]: Typically, flow-based models are considered in the Euclidean space. However, they could be considered in non-Euclidean spaces, resulting in new properties of (partially) invertible transformations.
- *Flows for ABC* [25]: Approximate Bayesian Computation (ABC) assumes that the posterior over quantities of interest is intractable. One possible approach to mitigate this issue is to approximate it using flow-based models, e.g., masked autoregressive flows [26], as presented in [25].

Many other interesting information on flow-based models could be found in a fantastic review by Papamakarios et al. [27].

3.1.7 ResNet Flows and DenseNet Flows

ResNet Flows [4, 5]
In the previous sections, we have discussed flow-based models with a pre-designed architectures (i.e., blocks consisting of coupling layers and permutation layers) that allow easy calculation of the Jacobian-determinant. However, we can take a different approach and think of how we can approximate the Jacobian-determinant for an almost arbitrary architecture. And, additionally, what kind of requirements we must impose to make the architecture invertible.

In [4], the authors consider widely used residual neural networks (ResNets) and construct an invertible ResNet layer which is only constrained in Lipschitz continuity. A ResNet is defined as: $F(x) = x + g(x)$, where g is modeled by a (convolutional) neural network and F represents a ResNet layer which is in general not invertible. However, g is constructed in such a way that it satisfies the Lipschitz constant being strictly lower than 1, $\text{Lip(g)} < 1$, by using spectral normalization of [28, 29]:

$$\text{Lip}(g) < 1, \quad \text{if} \quad ||W_i||_2 < 1, \tag{3.23}$$

where $|| \cdot ||_2$ is the ℓ_2 matrix norm. Then $\text{Lip}(g) = \text{K} < 1$ and $\text{Lip}(F) < 1 + \text{K}$. Only in this specific case the Banach fixed-point theorem holds and ResNet layer F has a unique inverse. As a result, the inverse can be approximated by fixed-point iterations [4].

To estimate the log-determinant is, especially for high-dimensional spaces, computationally intractable due to expensive computations. Since ResNet blocks have a constrained Lipschitz constant, the logarithm of the Jacobian-determinant is cheaper to compute, tractable, and approximated with guaranteed convergence [4]:

$$\ln p(x) = \ln p(f(x)) + \text{tr}\left(\sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} [J_g(x)]^k\right), \tag{3.24}$$

where $J_g(x)$ is the Jacobian of g at x that satisfies $||J_g|| < 1$. The Skilling-Hutchinson trace estimator [30, 31] is used to compute the trace at a lower cost than to fully compute the trace of the Jacobian. Residual Flows [5] use an improved method to estimate the power series at an even lower cost with an unbiased estimator based on "Russian roulette" of [32]. Intuitively, the method estimates the infinite sum of the power series by evaluating a finite amount of terms. In return, this leads to less computation of terms compared to invertible residual networks. To avoid derivative saturation, which occurs when the second derivative is zero in large regions, the LipSwish activation is proposed [4].

DenseNet Flows [6]

Since it is possible to formulate a flow for a ResNet architecture, a natural question is whether it could be accomplished for densely connected networks (DensNets) [33]. In [6], it was shown that indeed it is possible!

The main component of DenseNet flows is a DenseBlock that is defined as a function $F : \mathbb{R}^d \rightarrow \mathbb{R}^d$ with $F(x) = x + g(x)$, where g consists of dense layers $\{h_i\}_{i=1}^n$. Note that an important modification to make the model invertible is to output $x + g(x)$, whereas a standard DenseBlock would only output $g(x)$. The function g is expressed as follows:

$$g(x) = h_{n+1} \circ h_n \circ \cdots \circ h_1(x), \tag{3.25}$$

where h_{n+1} represents a 1×1 convolution to match the output size of $\mathbb{R}^d$. A layer h_i consists of two parts concatenated to each other. The upper part is a copy of the input signal. The lower part consists of the transformed input, where the transformation is a multiplication of (convolutional) weights W_i with the input signal, followed by a non-linearity ϕ having $\text{Lip}(\phi) \leq 1$, such as ReLU, ELU, LipSwish, or tanh. As an example, a dense layer h_2 can be composed as follows:

$$h_1(x) = \begin{bmatrix} x \\ \phi(W_1 x) \end{bmatrix}, \; h_2(h_1(x)) = \begin{bmatrix} h_1(x) \\ \phi(W_2 h_1(x)) \end{bmatrix}. \tag{3.26}$$

The DenseNet flows [6] rely on the same techniques for approximating the Jacobian-determinant as in the ResNet flows. The main difference between the DenseNet flows and the ResNet flows lies in normalizing weights so that the Lipschitz constant of the transformation is smaller than 1 and, thus, the transformation is invertible. Formally, to satisfy $\text{Lip}(g) < 1$, we need to enforce $\text{Lip}(h_i) < 1$ for all n layers, since $\text{Lip}(g) \leq \text{Lip}(h_{n+1}) \cdot \ldots \cdot \text{Lip}(h_1)$. Therefore, we first need to determine the Lipschitz constant for a dense layer h_i. We know that a function f is K-Lipschitz if for all points v and w the following holds:

$$d_Y(f(v), f(w)) \leq \mathrm{K} d_X(v, w), \tag{3.27}$$

where we assume that the distance metrics $d_X = d_Y = d$ are chosen to be the ℓ_2-norm. Further, let two functions f_1 and f_2 be concatenated in h:

$$h_v = \begin{bmatrix} f_1(v) \\ f_2(v) \end{bmatrix}, \quad h_w = \begin{bmatrix} f_1(w) \\ f_2(w) \end{bmatrix}, \tag{3.28}$$

where function f_1 is the upper part and f_2 is the lower part. We can now find an analytical form to express a limit on K for the dense layer in the form of Eq. (3.27):

$$\begin{aligned} d(h_v, h_w)^2 &= d(f_1(v), f_1(w))^2 + d(f_2(v), f_2(w))^2, \\ d(h_v, h_w)^2 &\leq \left(\mathrm{K}_1^2 + \mathrm{K}_2^2\right) d(v, w)^2, \end{aligned} \tag{3.29}$$

where we know that the Lipschitz constant of h consist of two parts, namely $\mathrm{Lip}(f_1) = \mathrm{K}_1$ and $\mathrm{Lip}(f_2) = \mathrm{K}_2$. Therefore, the Lipschitz constant of layer h can be expressed as

$$\mathrm{Lip}(h) = \sqrt{\left(\mathrm{K}_1^2 + \mathrm{K}_2^2\right)}. \tag{3.30}$$

With spectral normalization of Eq. (3.23), we know that we can enforce (convolutional) weights W_i to be at most 1-Lipschitz. Hence, for all n dense layers we apply the spectral normalization on the lower part which locally enforces $\mathrm{Lip}(f_2) = \mathrm{K}_2 < 1$. Further, since we enforce each layer h_i to be at most 1-Lipschitz and we start with h_1, where $f_1(x) = x$, we know that $\mathrm{Lip}(f_1) = 1$. Therefore, the Lipschitz constant of an entire layer can be at most $\mathrm{Lip}(h) = \sqrt{1^2 + 1^2} = \sqrt{2}$, thus dividing by this limit enforces each layer to be at most 1-Lipschitz. To read more about DenseNet flows and further improvements, please see the original paper [6].

3.2 Flows for Discrete Random Variables

3.2.1 Introduction

While discussing flow-based models in the previous section, we presented them as *density estimators*, namely models that represent stochastic dependencies among continuous random variables. We introduced the *change of variables* formula that helps to express a random variable by transforming it using invertible maps (bijections) f to a random variable with a known probability density function. Formally, it is defined as follows:

$$p(\mathbf{x}) = p\left(\mathbf{v} = f^{-1}(\mathbf{x})\right) \left|\mathbf{J}_f(x)\right|^{-1}, \tag{3.31}$$

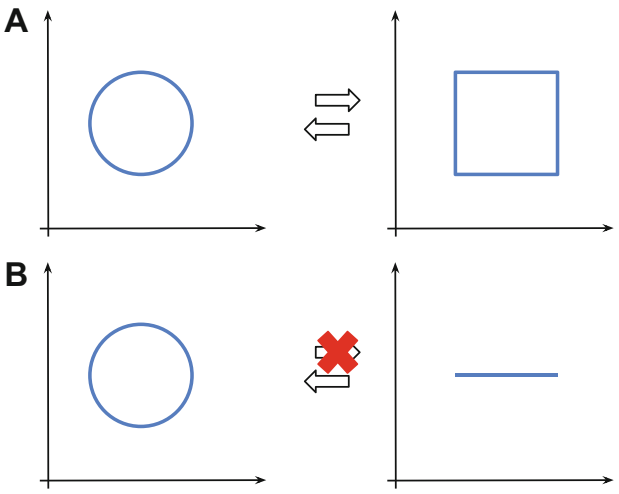


Fig. 3.7 Examples of: (**a**) homeomorphic spaces, and (**b**) non-homeomorphic spaces. The red cross indicates it is impossible to invert the transformation

where $\mathbf{J}_f(x)$ is the Jacobian of f at $\mathbf{x}$.

However, there are potential issues with such an approach. First of all, in many problems (e.g., image processing) the considered random variables (objects) are discrete. For instance, images typically take values in $\{0, 1, \ldots, 255\} \subset \mathbb{Z}$. In order to apply flows, we must apply *dequantization* [10] that results in a lower bound to the original probability distribution.

A continuous space possesses various potential pitfalls. One of them is that if we a transformation is a bijection (as in flows), not all continuous deformations are possible. It is tightly connected with *topology* and, more precisely, homeomorphisms, i.e., a continuous function between topological spaces that has a continuous inverse function, and diffeomorphisms, i.e., invertible functions that map one differentiable manifold to another such that both the function and its inverse are smooth. It is not crucial to know topology, but a curious reader may take a detour and read on that, it is definitely a fascinating field and I wish to know more about it! Anyway, let us consider three examples.

Imagine we want to transform a square into a circle (Fig. 3.7a). It is possible to find a homeomorphism (i.e., a bijection) that turns the square into the circle and back. Imagine you have a hammer and an iron square. If you start hitting the square infinitely many times, you can get an iron circle. Then, you can do it "backward" to get the square back. I know, it is unrealistic but hey, we are talking about math here!

However, if we consider a line segment and a circle (Fig. 3.7b), the situation is a bit more complicated. It is possible to transform the line segment into a circle, but not the other way around. Why? Because while transforming the circle to the line segment, it is unclear which point of the circle corresponds to the beginning (or the end) of the line segment. That is why we cannot invert the transformation!

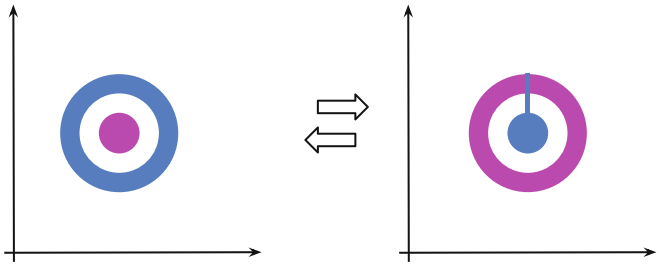


Fig. 3.8 An example of "replacing" a ring (in blue) with a ball (in magenta)

Another example that I really like, and which is closer to the potential issues of continuous flows, is transforming a ring into a ball as in Fig. 3.8. The goal is to replace the blue ring with the magenta ball. In order to make the transformation bijective, while transforming the blue ring in place of the magenta ball, we must ensure that the new magenta "ring" is in fact "broken" so that the new blue "ball" can get inside! Again, why? If the magenta ring is not broken, then we cannot say how the blue ball got inside that destroys bijectivity! In the language of topology, it is impossible because the two spaces are non-homeomorphic.

Alright, but how this affects the flow-based models? I hope that some of you asked this question, or maybe even imagined possible cases where this might hinder learning flows. In general, I would say it is fine, and we should not look for faults where there is none or almost none. However, if you work with flows that require dequantization, then you can spot cases like the one in Fig. 3.9. In this simple example, we have two discrete random variables that after uniform dequantization have two regions with equal probability mass, and the remaining two regions with zero probability mass [10]. After training a flow-based model, we have a density estimator that assigns non-zero probability mass where the true distribution has zero density! Moreover, the transformation in the flow must be a bijection, therefore, there is a continuity between the two squares (see Fig. 3.9, right). Where did we see that? Yes, in Fig. 3.8! We must know how to invert the transformation, thus, there must be a "trace" of how the probability mass moves between the regions.

Again, we can ask ourselves if it is bad. Well, I would say not really, but if we think of a case with more random variables, and there is always some little error here and there, this causes a *probability mass leakage* that could result in a far-from-perfect model. And, overall, the model could err in proper probability assignment.

3.2.2 *Flows in $\mathbb{R}$ or Maybe Rather in $\mathbb{Z}$?*

Before we consider any specific cases and discuss discrete flows, first we need to answer whether there is a change of variables formula for discrete random variables. The answer, fortunately, is yes! Let us consider $\mathbf{x} \in \mathcal{X}^D$ where $\mathcal{X}$ is a discrete space,

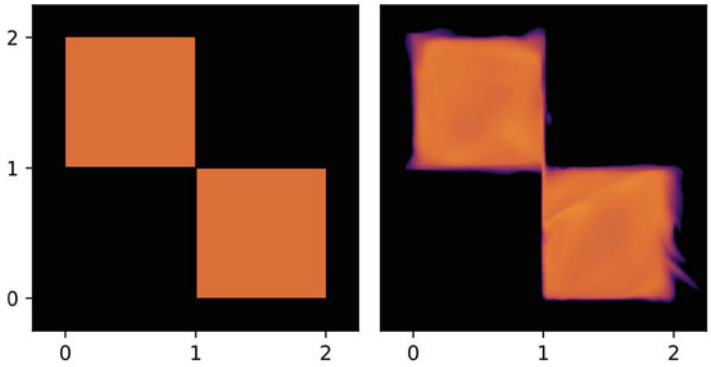


Fig. 3.9 An example of uniformly dequantized discrete random variables (*left*) and a flow-based model (*right*). Notice that in these examples, the true distribution assigns equal probability mass to the two regions in orange, and zero probability mass to the remaining two regions (in black). However, the flow-based model assigns probability mass outside the original non-zero probability regions

e.g., $\mathcal{X} = \{0, 1\}$ or $\mathcal{X} = \mathbb{Z}$. Then the change of variables takes the following form:

$$p(\mathbf{x}) = \pi \left(\mathbf{z}_0 = f^{-1}(\mathbf{x}) \right), \tag{3.32}$$

where f is an invertible transformation and $\pi(\cdot)$ is a base distribution. Immediately we can spot a "missing" Jacobian-determinant. This is correct! Why? Because now we live in the discrete world where the probability mass is assigned to points that are "shapeless" and the bijection cannot change the volume. Thus, the Jacobian-determinant is always equal to 1! That seems to be good news, isn't it? We can take any bijective transformations and we do not need to bother about the Jacobian. That is obviously true, however, we need to remember that the output of the transformation must be still discrete, i.e., $z \in \mathcal{X}^D$. As a result, we cannot use any arbitrary invertible neural network. We will discuss it in a minute, however, before we do that, it is worth discussing the expressivity of discrete flows.

Let us assume that we have an invertible transformation $f : \mathcal{X}^D \to \mathcal{X}^D$. Moreover, we have $\mathcal{X} = \{0, 1\}$. As noted by Papamakarios et al. [27], a discrete flow can only permute probability masses. Since there is no Jacobian (or, rather, the Jacobian-determinant is equal to 1), there is no chance to decrease or increase the probability for specific values. We depict it in Fig. 3.10. You can easily imagine the situation as the space is the Rubik's cube and your hands are the flows. If you record your moves, you can always play the video backward, thus, it is invertible. However, you can only shuffle the colors around! As a result, we do not gain anything by

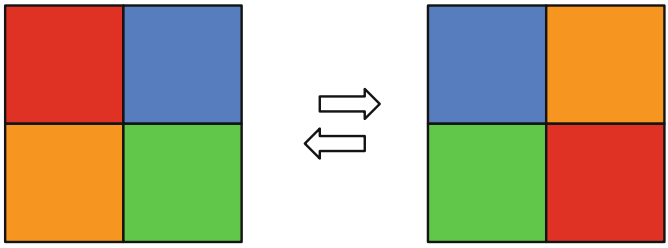


Fig. 3.10 An example of a discrete flow for two binary random variables. Colors represent various probabilities (i.e., the sum of all squares is 1)

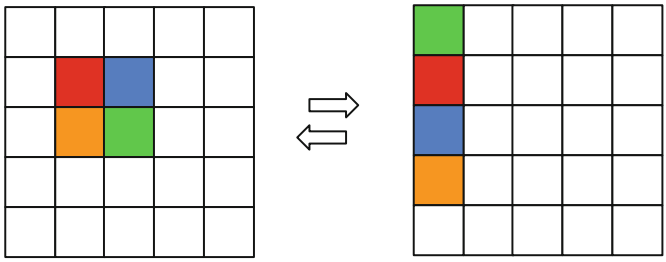


Fig. 3.11 An example of a discrete flow for two binary random variables but in the extended space. Colors represent various probabilities (i.e., the sum of all squares is 1)

applying the discrete flow, and learning the discrete flow is equivalent to learning the base distribution π.[1] So we are back to square one.

However, as pointed out by van den Berg et al. [12], the situation looks differently if we consider an extended space (or infinite space like $\mathbb{Z}$). The discrete flow can still only shuffle the probabilities, but now it can re-organize them in such a way that the probabilities can be factorized! In other words, it can help the base distribution to be a product of marginals, $\pi(\mathbf{z}) = \prod_{d=1}^{D} \pi_d(z_d|\theta_d)$, and the dependencies among variables are now encoded in the invertible transformations. An example of this case is presented in Fig. 3.11. We refer to [12] for a more thorough discussion with an appropriate lemma.

This is amazing information! It means that building a flow-based model in the discrete space makes sense. Now we can think of how to build an invertible neural network in discrete spaces and we have it!

[1] Well, this is not entirely true, we can still learn some correlations but it is definitely highly limited.

3.2.3 *Integer Discrete Flows*

We know now that it makes sense to work with discrete flows and that they are flexible as long as we use extended spaces or infinite spaces like $\mathbb{Z}$. However, the question is how to formulate an invertible transformation (or rather: an invertible neural network) that will output discrete values.

Hoogeboom et al. [22] proposed to focus on integers since they can be seen as discretized continuous values. As such, we consider coupling layers [7] and modify them accordingly. Let us remind ourselves the definition of bipartite coupling layers for $\mathbf{x} \in \mathbb{R}^D$:

$$\mathbf{y}_a = \mathbf{x}_a \tag{3.33}$$

$$\mathbf{y}_b = \exp\left(s\left(\mathbf{x}_a\right)\right) \odot \mathbf{x}_b + t\left(\mathbf{x}_a\right), \tag{3.34}$$

where $s(\cdot)$ and $t(\cdot)$ are arbitrary neural networks called *scaling* and *transition*, respectively.

Considering integer-valued variables, $\mathbf{x} \in \mathbb{Z}^D$, requires modifying this transformation. First, using scaling might be troublesome because multiplying by integers is still possible, but when we invert the transformation, we divide by integers, and dividing an integer by an integer does not necessarily result in an integer. Therefore, we must remove scaling just in case. Second, we use an arbitrary neural network for the transition. However, this network must return integers! Hoogeboom et al. [22] utilize a relatively simple trick, namely they say that we can round the output of $t(\cdot)$ to the closest integer. As a result, we add (in the forward) or subtract (in the inverse) integers from integers that is perfectly fine. (the outcome is still integer-valued.) Eventually, we get the following bipartite coupling layer:

$$\mathbf{y}_a = \mathbf{x}_a \tag{3.35}$$

$$\mathbf{y}_b = \mathbf{x}_b + \lfloor t\left(\mathbf{x}_a\right) \rceil, \tag{3.36}$$

where $\lfloor \cdot \rceil$ is the rounding operator. An inquisitive reader could ask at this point whether the rounding operator still allows using the backpropagation algorithm. In other words, whether the rounding operator is differentiable. The answer is **no**, but [22] showed that using the straight-through estimator (STE) of a gradient is sufficient. As a side note, the STE in this case uses the rounding in the forward pass of the network, $\lfloor t\left(\mathbf{x}_a\right) \rceil$, but it utilizes $t\left(\mathbf{x}_a\right)$ in the backward pass (to calculate gradients). van den Berg et al. [12] further indicated that indeed the STE works well and the bias does not hinder training too much. The implementation of the rounding operator using the STE is presented below.

```
# We need to turn torch.round (i.e., the rounding operator) into
    a differentiable function. For this purpose, we use the
    rounding in the forward pass, but the original input for the
    backward pass. This is nothing else than the STE.
class RoundStraightThrough(torch.autograd.Function):
```

```
    def __init__(self):
        super().__init__()

    @staticmethod
    def forward(ctx, input):
        rounded = torch.round(input, out=None)
        return rounded

    @staticmethod
    def backward(ctx, grad_output):
        grad_input = grad_output.clone()
        return grad_input
```

Listing 3.3 An implementation of the rounding operator using the STE

In [23] it has been shown how to generalize invertible transformations like bipartite coupling layers, among others, namely ($\mathcal{X}_{i:j}$ denotes a subset of $\mathcal{X}$ corresponding to variables from the i-th dimension to the j-th dimension, $\mathbf{x}_{i:j}$, we assume that $\mathcal{X}_{1:0} = \emptyset$ and $\mathcal{X}_{n+1:n} = \emptyset$):

Proposition 3.1 ([23]) *Let us take* $\mathbf{x}, \mathbf{y} \in \mathcal{X}$. *If binary transformations* $\circ$ *and* $\triangleright$ *have inverses* $\bullet$ *and* $\blacktriangleleft$, *respectively, and* $g_2, \ldots, g_D$ *and* $f_1, \ldots, f_D$ *are arbitrary functions, where* $g_i : \mathcal{X}_{1:i-1} \to \mathcal{X}_i$, $f_i : \mathcal{X}_{1:i-1} \times \mathcal{X}_{i+1:n} \to \mathcal{X}_i$, *then the following transformation from* $\mathbf{x}$ *to* $\mathbf{y}$*:*

$$
\begin{aligned}
y_1 &= x_1 \circ f_1(\emptyset, \mathbf{x}_{2:D}) \\
y_2 &= (g_2(y_1) \triangleright x_2) \circ f_2(y_1, \mathbf{x}_{3:D}) \\
&\ldots \\
y_d &= (g_d(\mathbf{y}_{1:d-1}) \triangleright x_d) \circ f_d(\mathbf{y}_{1:d-1}, \mathbf{x}_{d+1:D}) \\
&\ldots \\
y_D &= (g_D(\mathbf{y}_{1:D-1}) \triangleright x_D) \circ f_D(\mathbf{y}_{1:D-1}, \emptyset)
\end{aligned}
$$

is invertible.

Proof In order to inverse $\mathbf{y}$ to $\mathbf{x}$ we start from the last element to obtain the following:

$$
x_D = g_D(\mathbf{y}_{1:D-1}) \blacktriangleleft (y_D \bullet f_D(\mathbf{y}_{1:D-1}, \emptyset)) \,.
$$

Then, we can proceed with next expressions in the decreasing order (i.e., from $D-1$ to 1) to eventually obtain

$$
\begin{aligned}
x_{D-1} &= g_{D-1}(\mathbf{y}_{1:D-2}) \blacktriangleleft (y_{D-1} \bullet f_{D-1}(\mathbf{y}_{1:D-2}, x_D)) \\
&\ldots
\end{aligned}
$$

$$
\begin{aligned}
x_d &= g_d(\mathbf{y}_{1:d-1}) \blacktriangleleft (y_d \bullet f_d(\mathbf{y}_{1:d-1}, \mathbf{x}_{d+1:D})) \\
&\dots \\
x_2 &= g_2(y_1) \blacktriangleleft (y_2 \bullet f_2(y_1, \mathbf{x}_{3:D})) \\
x_1 &= y_1 \bullet f_1(\emptyset, \mathbf{x}_{2:D}).
\end{aligned}
$$

□

For instance, we can divide $\mathbf{x}$ into four parts, $\mathbf{x} = [\mathbf{x}_a, \mathbf{x}_b, \mathbf{x}_c, \mathbf{x}_d]$, and the following transformation (a quadripartite coupling layer) is invertible [23]:

$$\mathbf{y}_a = \mathbf{x}_a + \lfloor t\,(\mathbf{x}_b, \mathbf{x}_c, \mathbf{x}_d)\rceil \tag{3.37}$$

$$\mathbf{y}_b = \mathbf{x}_b + \lfloor t\,(\mathbf{y}_a, \mathbf{x}_c, \mathbf{x}_d)\rceil \tag{3.38}$$

$$\mathbf{y}_c = \mathbf{x}_c + \lfloor t\,(\mathbf{y}_a, \mathbf{y}_b, \mathbf{x}_d)\rceil \tag{3.39}$$

$$\mathbf{y}_d = \mathbf{x}_d + \lfloor t\,(\mathbf{y}_a, \mathbf{y}_b, \mathbf{y}_c)\rceil. \tag{3.40}$$

This new invertible transformation could be seen as a kind of autoregressive processing since $\mathbf{y}_a$ is used to calculate $\mathbf{y}_b$, then both $\mathbf{y}_a$ and $\mathbf{y}_b$ are used for obtaining $\mathbf{y}_c$ and so on. As a result, we get a more powerful transformation than the bipartite coupling layer.

If we stick to a coupling layer, we need to remember to use a permutation layer to reverse the order of variables. Otherwise, some inputs would be only partially processed. This holds true for any coupling layer, either they are used for continuous flows or integer-valued flows.

The last component we need to think of is the base distribution. Similarly to flow-based models, we can use various tricks to boost the performance of the model. For instance, we can consider squeezing, factoring-out, and a mixture model for the base distribution [22]. However, in this section, we try to keep the model as simple as possible, therefore, we use the product of marginals as the base distribution. For images represented as integers, we use the following:

$$\pi(\mathbf{z}) = \prod_{d=1}^{D} \pi_d(z_d) \tag{3.41}$$

$$= \prod_{d=1}^{D} \mathrm{DL}(z_d|\mu_d, \nu_d), \tag{3.42}$$

where $\pi_d(z_d) = \mathrm{DL}(z_d|\mu_d, \nu_d)$ is the discretized logistic distribution that is defined as a difference of CDFs of the logistic distribution as follows [34]:

$$\pi(z) = \mathrm{sigm}\left((z + 0.5 - \mu)/\nu\right) - \mathrm{sigm}\left((z - 0.5 - \mu)/\nu\right), \tag{3.43}$$

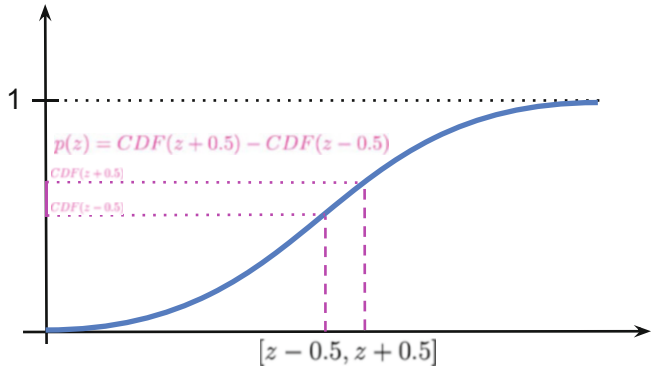


Fig. 3.12 An example of the discretized logistic distribution with $\mu = 0$ and $\nu = 1$. The magenta area on the y-axis corresponds to the probability mass of a bin of size 1

where $\mu \in \mathbb{R}$ and $\nu > 0$ denote the mean and the scale, respectively, sigm$(\cdot)$ is the sigmoid function. Notice that this is equivalent to calculating the probability of z falling into a bin of length 1, therefore, we add 0.5 in the first CDF and subtract 0.5 from the second CDF. An example of the discretized distribution is presented in Fig. 3.12 and the implementation follows. Interestingly, we can use this distribution to replace the Categorical distribution in Chap. 2, as it was done in [18]. We can even use a mixture of discretized logistic distribution to further improve the final performance [22, 35].

```
# This function implements the log of the discretized logistic
    distribution.
def log_integer_probability(x, mean, logscale):
    scale = torch.exp(logscale)

    logp = log_min_exp(
        F.logsigmoid((x + 0.5 - mean) / scale),
        F.logsigmoid((x - 0.5 - mean) / scale))

    return logp
```

Listing 3.4 The logarithm of the discretized logistic distribution [34]

Eventually, our log-likelihood function takes the following form:

$$\ln p(\mathbf{x}) = \sum_{d=1}^{D} \ln \mathrm{DL}(z_d = f^{-1}(\mathbf{x})|\mu_d, \nu_d) \tag{3.44}$$

$$= \sum_{d=1}^{D} \ln \left(\mathrm{sigm}\left((z_d + 0.5 - \mu_d)/\nu_d\right) - \mathrm{sigm}\left((z_d - 0.5 - \mu_d)/\nu_d\right)\right), \tag{3.45}$$

where we make all μ_d and ν_d learnable parameters. Notice that ν_d must be positive (strictly larger than 0), therefore, in the implementation, we will consider the logarithm of the scale because taking exp of the log-scale ensures having strictly positive values.

3.2.4 Code

Now, we have all components to implement our own Integer Discrete Flow (IDF)! Below, there is a code with a lot of comments that should help to understand every single line of it.

```
# That's the class of the Integer Discrete Flows (IDFs).
# There are two options implemented:
# Option 1: The bipartite coupling layers as in (Hoogeboom et al., 2019).
# Option 2: A new coupling layer with 4 parts as in (Tomczak, 2021).
# We implement the second option explicitly, without any loop, so that it is very clear how it works.
class IDF(nn.Module):
    def __init__(self, netts, num_flows, D=2):
        super(IDF, self).__init__()

        print('IDF by JT.')

        # Option 1:
        if len(netts) == 1:
            self.t = torch.nn.ModuleList([netts[0]() for _ in range(num_flows)])
            self.idf_git = 1

        # Option 2:
        elif len(netts) == 4:
            self.t_a = torch.nn.ModuleList([netts[0]() for _ in range(num_flows)])
            self.t_b = torch.nn.ModuleList([netts[1]() for _ in range(num_flows)])
            self.t_c = torch.nn.ModuleList([netts[2]() for _ in range(num_flows)])
            self.t_d = torch.nn.ModuleList([netts[3]() for _ in range(num_flows)])
            self.idf_git = 4

        else:
            raise ValueError('You can provide either 1 or 4 translation nets.')

        # The number of flows (i.e., invertible transformations).
        self.num_flows = num_flows

```

```
31          # The rounding operator
32          self.round = RoundStraightThrough.apply
33
34          # Initialization of the parameters of the base
    distribution.
35          # Notice they are parameters, so they are trained
    alongside the weights of neural networks.
36          self.mean = nn.Parameter(torch.zeros(1, D)) #mean
37          self.logscale = nn.Parameter(torch.ones(1, D)) #log-scale
38
39          # The dimensionality of the problem.
40          self.D = D
41
42      # The coupling layer.
43      def coupling(self, x, index, forward=True):
44
45          # Option 1:
46          if self.idf_git == 1:
47              (xa, xb) = torch.chunk(x, 2, 1)
48
49              if forward:
50                  yb = xb + self.round(self.t[index](xa))
51              else:
52                  yb = xb - self.round(self.t[index](xa))
53
54              return torch.cat((xa, yb), 1)
55
56          # Option 2:
57          elif self.idf_git == 4:
58              (xa, xb, xc, xd) = torch.chunk(x, 4, 1)
59
60              if forward:
61                  ya = xa + self.round(self.t_a[index](torch.cat((
    xb, xc, xd), 1)))
62                  yb = xb + self.round(self.t_b[index](torch.cat((
    ya, xc, xd), 1)))
63                  yc = xc + self.round(self.t_c[index](torch.cat((
    ya, yb, xd), 1)))
64                  yd = xd + self.round(self.t_d[index](torch.cat((
    ya, yb, yc), 1)))
65              else:
66                  yd = xd - self.round(self.t_d[index](torch.cat((
    xa, xb, xc), 1)))
67                  yc = xc - self.round(self.t_c[index](torch.cat((
    xa, xb, yd), 1)))
68                  yb = xb - self.round(self.t_b[index](torch.cat((
    xa, yc, yd), 1)))
69                  ya = xa - self.round(self.t_a[index](torch.cat((
    yb, yc, yd), 1)))
70
71              return torch.cat((ya, yb, yc, yd), 1)
72
73      # Similarly to RealNVP, we have also the permute layer.
74      def permute(self, x):
```

```
75          return x.flip(1)
76
77      # The main function of the IDF: forward pass from x to z.
78      def f(self, x):
79          z = x
80          for i in range(self.num_flows):
81              z = self.coupling(z, i, forward=True)
82              z = self.permute(z)
83
84          return z
85
86      # The function for inverting z to x.
87      def f_inv(self, z):
88          x = z
89          for i in reversed(range(self.num_flows)):
90              x = self.permute(x)
91              x = self.coupling(x, i, forward=False)
92
93          return x
94
95      # The PyTorch forward function. It returns the log-
        probability.
96      def forward(self, x, reduction='avg'):
97          z = self.f(x)
98          if reduction == 'sum':
99              return -self.log_prior(z).sum()
100         else:
101             return -self.log_prior(z).mean()
102
103     # The function for sampling:
104     # First we sample from the base distribution.
105     # Second, we invert z.
106     def sample(self, batchSize, intMax=100):
107         # sample z:
108         z = self.prior_sample(batchSize=batchSize, D=self.D,
        intMax=intMax)
109         # x = f^-1(z)
110         x = self.f_inv(z)
111         return x.view(batchSize, 1, self.D)
112
113     # The function for calculating the logarithm of the base
        distribution.
114     def log_prior(self, x):
115         log_p = log_integer_probability(x, self.mean, self.
        logscale)
116         return log_p.sum(1)
117
118     # A function for sampling integers from the base distribution
        .
119     def prior_sample(self, batchSize, D=2):
120         # Sample from logistic
121         y = torch.rand(batchSize, self.D)
122         # Here we use a property of the logistic distribution:
```

```
        # In order to sample from a logistic distribution, first sample y ~ Uniform[0,1].
        # Then, calculate log(y / (1.-y)), scale is with the scale, and add the mean.
        x = torch.exp(self.logscale) * torch.log(y / (1. - y)) + self.mean
        # And then round it to an integer.
        return torch.round(x)
```

Listing 3.5 An example of networks

Below, we provide examples of neural networks that could be used to run the IDFs.

```
# The number of invertible transformations
num_flows = 8

# This variable defines whether we use:
#   Option 1: 1 - the classic coupling layer proposed in (Hogeboom et al., 2019)
#   Option 2: 4 - the general invertible transformation in (Tomczak, 2021) with 4 partitions
idf_git = 1

if idf_git == 1:
    nett = lambda: nn.Sequential(
                        nn.Linear(D//2, M), nn.LeakyReLU(),
                        nn.Linear(M, M), nn.LeakyReLU(),
                        nn.Linear(M, D//2))
    netts = [nett]

elif idf_git == 4:
    nett_a = lambda: nn.Sequential(
                        nn.Linear(3 * (D//4), M), nn.LeakyReLU(),
                        nn.Linear(M, M), nn.LeakyReLU(),
                        nn.Linear(M, D//4))

    nett_b = lambda: nn.Sequential(
                        nn.Linear(3 * (D//4), M), nn.LeakyReLU(),
                        nn.Linear(M, M), nn.LeakyReLU(),
                        nn.Linear(M, D//4))

    nett_c = lambda: nn.Sequential(
                        nn.Linear(3 * (D//4), M), nn.LeakyReLU(),
                        nn.Linear(M, M), nn.LeakyReLU(),
                        nn.Linear(M, D//4))

    nett_d = lambda: nn.Sequential(
                        nn.Linear(3 * (D//4), M), nn.LeakyReLU(),
                        nn.Linear(M, M), nn.LeakyReLU(),
                        nn.Linear(M, D//4))

    netts = [nett_a, nett_b, nett_c, nett_d]

```

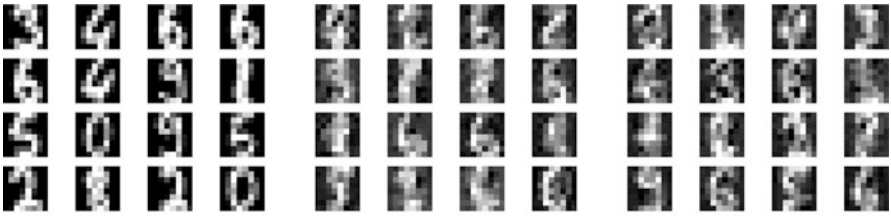


Fig. 3.13 An example of outcomes after the training: (**a**) Randomly selected real images. (**b**) Unconditional generations from the IDF with bipartite coupling layers. (**c**) Unconditional generations from the IDF with quadripartite coupling layers

```
# Init IDF
model = IDF(netts, num_flows, D=D)
# Print the summary (like in Keras)
print(summary(model, torch.zeros(1, 64), show_input=False,
    show_hierarchical=False))
```

Listing 3.6 An example of networks

And we are done, this is all we need to have! After running the code (take a look at: https://github.com/jmtomczak/intro_dgm) and training the IDFs, we should obtain results similar to those in Fig. 3.13.

3.2.5 What's Next?

Similarly to our example of RealNVP, here we present rather a simplified implementation of IDFs. We can use many of the tricks presented in the section on RealNVP (see Sect. 3.1.6). On recent developments on IDFs, please see also [12].

Integer discrete flows have a great potential in data compression. Since IDFs learn the distribution $p(\mathbf{x})$ directly on the integer-valued objects, they are excellent candidates for lossless compression. As presented in [22], they are competitive with other codecs for lossless compression of images.

The paper by van den Berg et al. [12] further shows that the potential bias following from the STE of the gradients is not as dramatic as originally thought [22], and they can learn flexible distributions. This result suggests that IDFs require special attention, especially for real-life applications like data compression.

It seems that the next step would be to think of more powerful transformations for discrete variables, e.g., see [23], and developing powerful architectures. Another interesting direction is utilizing alternative learning algorithms in which gradients could be better estimated [36], or even replaced [37].