Chapter 4
Latent Variable Models

4.1 Introduction

In the previous chapters, we discussed two approaches to learning $p(\mathbf{x})$: autoregressive models (ARMs) in Chap. 2 and flow-based models (or flows for short) in Chap. 3. Both ARMs and flows model the likelihood function directly, that is, either by factorizing the distribution and parameterizing conditional distributions $p(x_d|\mathbf{x}_{<d})$ as in ARMs or by utilizing invertible transformations (neural networks) for the change of variables formula as in flows. Now, we will discuss a third approach that introduces **latent variables**.

Let us briefly discuss the following scenario. We have a collection of images with horses. We want to learn $p(\mathbf{x})$ for, e.g., generating new images. Before we do that, we can ask ourselves how we should generate a horse, or, in other words, if we were such a generative model, how we would do that. Maybe we would first sketch the general silhouette of a horse, its size and shape, then add hooves, fill in details of a head, color it, etc. In the end, we may consider the background. In general, we can say that there are some *factors* in data (e.g., a silhouette, a color, a background) that are crucial for generating an object (here, a horse). Once we decide about these factors, we can generate them by adding details. I do not want to delve into a philosophical/cognitive discourse, but I hope that we all agree that when we paint something, this is more-or-less our procedure of generating a painting.

We use mathematics now to express this *generative process*. Namely, we have our high-dimensional objects of interest, $\mathbf{x} \in \mathcal{X}^D$ (e.g., for images, $\mathcal{X} \in \{0, 1, \ldots, 255\}$), and **low-dimensional latent variables**, $\mathbf{z} \in \mathcal{Z}^M$ (e.g., $\mathcal{Z} = \mathbb{R}$), that we can call hidden factors in data. In mathematical words, we can refer to $\mathcal{Z}^M$ as a low-dimensional *manifold*. Then, the generative process could be expressed as follows:

1. $\mathbf{z} \sim p(\mathbf{z})$ (Fig. 4.1, in red)
2. $\mathbf{x} \sim p(\mathbf{x}|\mathbf{z})$ (Fig. 4.1, in blue)

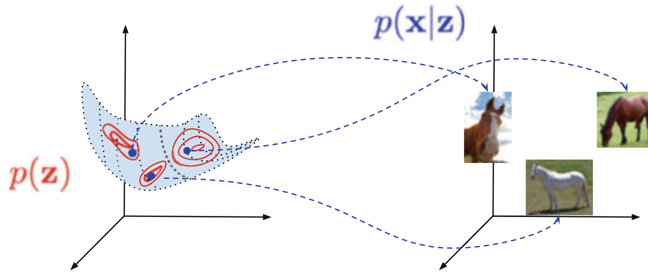

Fig. 4.1 A diagram presenting a latent variable model and a generative process. Notice the low-dimensional manifold (here 2D) embedded in the high-dimensional space (here 3D)

In plain words, we first sample $\mathbf{z}$ (e.g., we imagine the size, the shape, and the color of a horse) and then create an image with all necessary details, i.e., we sample $\mathbf{x}$ from the conditional distribution $p(\mathbf{x}|\mathbf{z})$. One can ask whether we need probabilities here but try to create *precisely the same* image at least two times. Due to various external factors, it is almost impossible to create two identical images. That is why probability theory is so beautiful and allows us to describe reality!

The idea behind **latent variable models** is that we introduce the latent variables $\mathbf{z}$ and the joint distribution is factorized as follows: $p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x}|\mathbf{z})p(\mathbf{z})$. This naturally expressed the generative process described above. However, for training, we have access only to $\mathbf{x}$. Therefore, according to probabilistic inference, we should *sum out* (or *marginalize out*) the unknown, namely, $\mathbf{z}$. As a result, the (marginal) likelihood function is the following:

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})\, \mathrm{d}\mathbf{z}. \tag{4.1}$$

A natural question now is how to calculate this integral. In general, it is a difficult task. There are two possible directions. First, the integral **is** tractable. We will briefly discuss it before we jump into the second approach that utilizes a specific **approximate inference**, namely, **variational inference**.

4.2 Probabilistic Principal Component Analysis

Let us discuss the following situation:

- We consider continuous random variables only, i.e., $\mathbf{z} \in \mathbb{R}^M$ and $\mathbf{x} \in \mathbb{R}^D$.
- The distribution of $\mathbf{z}$ is the standard Gaussian, i.e., $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, \mathbf{I})$.
- The dependency between $\mathbf{z}$ and $\mathbf{x}$ is linear and we assume a Gaussian additive noise:

$$\mathbf{x} = \mathbf{W}\mathbf{z} + \mathbf{b} + \varepsilon, \tag{4.2}$$

where $\varepsilon \sim \mathcal{N}(\varepsilon|0, \sigma^2\mathbf{I})$. The property of the Gaussian distribution yields [1]

$$p(\mathbf{x}|\mathbf{z}) = \mathcal{N}\left(\mathbf{x}|\mathbf{W}\mathbf{z} + \mathbf{b}, \sigma^2\mathbf{I}\right). \tag{4.3}$$

This model is known as the *probabilistic Principal Component Analysis* (pPCA) [2].

Next, we can take advantage of properties of a linear combination of two vectors of normally distributed random variables to calculate the integral explicitly [1]:

$$\begin{aligned} p(\mathbf{x}) &= \int p(\mathbf{x}|\mathbf{z})\ p(\mathbf{z})\ \mathrm{d}\mathbf{z} && (4.4)\\ &= \int \mathcal{N}\left(\mathbf{x}|\mathbf{W}\mathbf{z} + \mathbf{b}, \sigma^2\mathbf{I}\right) \mathcal{N}\left(\mathbf{z}|0, \mathbf{I}\right)\ \mathrm{d}\mathbf{z} && (4.5)\\ &= \mathcal{N}\left(\mathbf{x}|\mathbf{b}, \mathbf{W}\mathbf{W}^\top + \sigma^2\mathbf{I}\right). && (4.6) \end{aligned}$$

Now, we are able to calculate the logarithm of the (marginal) likelihood function $\ln p(\mathbf{x})$! We refer to [1, 2] for more details on learning the parameters in the pPCA model. Moreover, what is interesting about the pPCA is that, due to the properties of Gaussians, we can also calculate the *true* posterior over $\mathbf{z}$ analytically:

$$p(\mathbf{z}|\mathbf{x}) = \mathcal{N}\left(\mathbf{M}^{-1}\mathbf{W}^\top(\mathbf{x} - \mu), \sigma^{-2}\mathbf{M}\right), \tag{4.7}$$

where $\mathbf{M} = \mathbf{W}^\top\mathbf{W} + \sigma^2\mathbf{I}$. Once we find $\mathbf{W}$ that maximize the log-likelihood function, and the dimensionality of the matrix $\mathbf{W}$ is computationally tractable, we can calculate $p(\mathbf{z}|\mathbf{x})$. This is a big thing! Why? Because for a given observation $\mathbf{x}$, we can calculate the distribution over the *latent factors*!

In my opinion, the probabilistic PCA is an extremely important latent variable model for two reasons. First, we can calculate everything *by hand* and, thus, it is a great exercise to develop an intuition about the latent variable models. Second, it is a linear model and, therefore, a curious reader should feel tingling in his or her head already and ask himself or herself the following questions: What would happen if we take non-linear dependencies? And what would happen if we use other distributions than Gaussians? In both cases, the answer is the same: We would not be able to calculate the integral exactly, and some sort of approximation would be necessary. Anyhow, pPCA is a model that everyone interested in latent variable models should study in depth to create an intuition about probabilistic modeling.

4.3 Variational Auto-Encoders: Variational Inference for Non-linear Latent Variable Models

4.3.1 The Model and the Objective

Let us take a look at the integral one more time and think of a general case where we cannot calculate it analytically. The simplest approach would be to use the Monte Carlo approximation:

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})\ p(\mathbf{z})\ \mathrm{d}\mathbf{z} \tag{4.8}$$

$$= \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[p(\mathbf{x}|\mathbf{z})\right] \tag{4.9}$$

$$\approx \frac{1}{K}\sum_{k} p(\mathbf{x}|\mathbf{z}_k), \tag{4.10}$$

where, in the last line, we use samples from the prior over latents, $\mathbf{z}_k \sim p(\mathbf{z})$. Such an approach is relatively easy and since our computational power grows so fast, we can sample a lot of points in reasonably short time. However, as we know from statistics, if $\mathbf{z}$ is multidimensional, and M is relatively large, we get into a trap of the *curse of dimensionality*, and to cover the space properly, the number of samples grows exponentially with respect to M. If we take too few samples, then the approximation is simply very poor.

We can use more advanced Monte Carlo techniques [3]; however, they still suffer from issues associated with the curse of dimensionality. An alternative approach is the application of *variational inference* [4]. Let us consider a family of variational distributions parameterized by ϕ, $\{q_\phi(\mathbf{z})\}_\phi$. For instance, we can consider Gaussians with means and variances, $\phi = \{\mu, \sigma^2\}$. We know the form of these distributions, and we assume that they assign non-zero probability mass to all $\mathbf{z} \in \mathcal{Z}^M$. Then, the logarithm of the marginal distribution could be approximated as follows:

$$\ln p(\mathbf{x}) = \ln \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})\ \mathrm{d}\mathbf{z} \tag{4.11}$$

$$= \ln \int \frac{q_\phi(\mathbf{z})}{q_\phi(\mathbf{z})} p(\mathbf{x}|\mathbf{z})p(\mathbf{z})\ \mathrm{d}\mathbf{z} \tag{4.12}$$

$$= \ln \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z})}\left[\frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{q_\phi(\mathbf{z})}\right] \tag{4.13}$$

$$\geq \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z})} \ln\left[\frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{q_\phi(\mathbf{z})}\right] \tag{4.14}$$

$$= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z})}\left[\ln p(\mathbf{x}|\mathbf{z}) + \ln p(\mathbf{z}) - \ln q_\phi(\mathbf{z})\right] \tag{4.15}$$

$$= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z})}\left[\ln p(\mathbf{x}|\mathbf{z})\right] - \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z})}\left[\ln q_\phi(\mathbf{z}) - \ln p(\mathbf{z})\right]. \tag{4.16}$$

In the fourth line we used *Jensen's inequality*.

If we consider an *amortized variational posterior*, namely, $q_{\phi}(\mathbf{z}|\mathbf{x})$ instead of $q_{\phi}(\mathbf{z})$ for each $\mathbf{x}$, then we get

$$\ln p(\mathbf{x}) \geq \mathbb{E}_{\mathbf{z}\sim q_{\phi}(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x}|\mathbf{z})\right] - \mathbb{E}_{\mathbf{z}\sim q_{\phi}(\mathbf{z}|\mathbf{x})}\left[\ln q_{\phi}(\mathbf{z}|\mathbf{x}) - \ln p(\mathbf{z})\right]. \tag{4.17}$$

Amortization could be extremely useful because we train a single model (e.g., a neural network with some weights), and it returns parameters of a distribution for given input. From now on, we will assume that we use amortized variational posteriors; however, please remember that we do not need to do that! Please take a look at [5] where a semi-amortized variational inference is considered.

As a result, we obtain an auto-encoder-like model, with a *stochastic encoder*, $q_{\phi}(\mathbf{z}|\mathbf{x})$, and a *stochastic decoder*, $p(\mathbf{x}|\mathbf{z})$. We use *stochastic* to highlight that the encoder and the decoder are probability distributions and to stress out a difference to a deterministic auto-encoder. This model, with the amortized variational posterior, is called a **Variational Auto-Encoder** [6, 7]. The lower bound of the log-likelihood function is called the Evidence Lower BOund (**ELBO**).

The first part of the ELBO, $\mathbb{E}_{\mathbf{z}\sim q_{\phi}(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x}|\mathbf{z})\right]$, is referred to as the (negative) *reconstruction error*, because $\mathbf{x}$ is encoded to $\mathbf{z}$ and then decoded back. The second part of the ELBO, $\mathbb{E}_{\mathbf{z}\sim q_{\phi}(\mathbf{z}|\mathbf{x})}\left[\ln q_{\phi}(\mathbf{z}|\mathbf{x}) - \ln p(\mathbf{z})\right]$, could be seen as a *regularizer* and it coincides with the Kullback–Leibler (KL) divergence. Please keep in mind that for more complex models (e.g., hierarchical models), the regularizer(s) may not be interpreted as the KL term. Therefore, we prefer to use the term *the regularizer* because it is more general.

4.3.2 A Different Perspective on the ELBO

For completeness, we provide also a different derivation of the ELBO that will help us to understand why the lower bound might be tricky sometimes:

$$\begin{aligned}
\ln p(\mathbf{x}) &= \mathbb{E}_{\mathbf{z}\sim q_{\phi}(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x})\right] && (4.18)\\
&= \mathbb{E}_{\mathbf{z}\sim q_{\phi}(\mathbf{z}|\mathbf{x})}\left[\ln \frac{p(\mathbf{z}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{z}|\mathbf{x})}\right] && (4.19)\\
&= \mathbb{E}_{\mathbf{z}\sim q_{\phi}(\mathbf{z}|\mathbf{x})}\left[\ln \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{p(\mathbf{z}|\mathbf{x})}\right] && (4.20)\\
&= \mathbb{E}_{\mathbf{z}\sim q_{\phi}(\mathbf{z}|\mathbf{x})}\left[\ln \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{p(\mathbf{z}|\mathbf{x})}\frac{q_{\phi}(\mathbf{z}|\mathbf{x})}{q_{\phi}(\mathbf{z}|\mathbf{x})}\right] && (4.21)\\
&= \mathbb{E}_{\mathbf{z}\sim q_{\phi}(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x}|\mathbf{z})\frac{p(\mathbf{z})}{q_{\phi}(\mathbf{z}|\mathbf{x})}\frac{q_{\phi}(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{x})}\right] && (4.22)
\end{aligned}$$

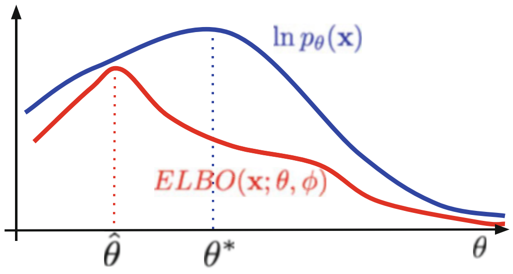

Fig. 4.2 The ELBO is a lower bound on the log-likelihood. As a result, $\hat{\theta}$ maximizing the ELBO does not necessarily coincide with θ^* that maximizes $\ln p(\mathbf{x})$. The looser the ELBO is, the more this can bias maximum likelihood estimates of the model parameters

$$= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x}|\mathbf{z}) - \ln \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p(\mathbf{z})} + \ln \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{x})}\right] \tag{4.23}$$

$$= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x}|\mathbf{z})\right] - KL\left[q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z})\right] + KL\left[q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{x})\right]. \tag{4.24}$$

Please note that in the derivation above we use the sum and the product rules together with multiplying by $1 = \frac{q_\phi(\mathbf{z}|\mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})}$, nothing else, no dirty tricks here! Please try to replicate this by yourself, step by step. If you understand this derivation well, it would greatly help you to see where potential problems of the VAEs (and the latent variable models in general) lie.

Once you analyzed this derivation, let us take a closer look at it:

$$\ln p(\mathbf{x}) = \underbrace{\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x}|\mathbf{z})\right] - KL\left[q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z})\right]}_{ELBO} + \underbrace{KL\left[q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{x})\right]}_{\geq 0}. \tag{4.25}$$

The last component, $KL\left[q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{x})\right]$, measures the difference between the variational posterior and the *real* posterior, but we do not know what the real posterior is! However, we can skip this part since the Kullback–Leibler divergence is always equal to or greater than 0 (from its definition) and, thus, we are left with the ELBO. We can think of $KL\left[q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{x})\right]$ as a gap between the ELBO and the true log-likelihood.

Beautiful! But ok, why this is so important? Well, if we take $q_\phi(\mathbf{z}|\mathbf{x})$ that is a bad approximation of $p(\mathbf{z}|\mathbf{x})$, then the KL term will be larger, and even if the ELBO is optimized well, the gap between the ELBO and the true log-likelihood could be huge! In plain words, if we take too simplistic posterior, we can end up with a bad VAE anyway. What is "bad" in this context? Let us take a look at Fig. 4.2. If the ELBO is a loose lower bound of the log-likelihood, then the optimal solution of the ELBO could be completely different than the solution of the log-likelihood. We will comment on how to deal with that later on and, for now, it is enough to be aware of that issue.

4.3.3 Components of VAEs

Let us wrap up what we know right now. First of all, we consider a class of amortized variational posteriors $\{q_{\phi}(\mathbf{z}|\mathbf{x})\}_{\phi}$ that approximate the true posterior $p(\mathbf{z}|\mathbf{x})$. We can see them as **stochastic encoders**. Second, the conditional likelihood $p(\mathbf{x}|\mathbf{z})$ could be seen as a **stochastic decoder**. Third, the last component, $p(\mathbf{z})$, is the **marginal distribution**, also referred to as a **prior**. Lastly, the objective is the ELBO, a lower bound to the log-likelihood function:

$$\ln p(\mathbf{x}) \geq \mathbb{E}_{\mathbf{z}\sim q_{\phi}(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x}|\mathbf{z})\right] - \mathbb{E}_{\mathbf{z}\sim q_{\phi}(\mathbf{z}|\mathbf{x})}\left[\ln q_{\phi}(\mathbf{z}|\mathbf{x}) - \ln p(\mathbf{z})\right]. \tag{4.26}$$

There are two questions left to get the full picture of the VAEs:

1. How to parameterize the distributions?
2. How to calculate the expected values? After all, these integrals have not disappeared!

4.3.3.1 Parameterization of Distributions

As you can probably guess by now, we use neural networks to parameterize the encoders and the decoders. But before we use the neural networks, we should know *what* distributions we use! Fortunately, in the VAE framework we are almost free to choose any distributions! However, we must remember that they should make sense for a considered problem. So far, we have explained everything through images, so let us continue that. If $\mathbf{x} \in \{0, 1, \ldots, 255\}^{D}$, then we *cannot* use a Normal distribution, because its support is totally different than the support of discrete-valued images. A possible distribution we can use is the *categorical distribution*, that is:

$$p_{\theta}(\mathbf{x}|\mathbf{z}) = \text{Categorical}\left(\mathbf{x}|\theta(\mathbf{z})\right), \tag{4.27}$$

where the probabilities are given by a neural network NN, namely, $\theta(\mathbf{z}) = \text{softmax}\left(\text{NN}(\mathbf{z})\right)$. The neural network NN could be an MLP, a convolutional neural network, RNNs, etc.

The choice of a distribution for the latent variables depends on how we want to express the latent factors in data. For convenience, typically $\mathbf{z}$ is taken as a vector of continuous random variables, $\mathbf{z} \in \mathbb{R}^{M}$. Then, we can use Gaussians for both the variational posterior and the prior:

$$q_{\phi}(\mathbf{z}|\mathbf{x}) = \mathcal{N}\left(\mathbf{z}|\mu_{\phi}(\mathbf{x}), \text{diag}\left[\sigma_{\phi}^{2}(\mathbf{x})\right]\right) \tag{4.28}$$

$$p(\mathbf{z}) = \mathcal{N}\left(\mathbf{z}|0, \mathbf{I}\right), \tag{4.29}$$

where $\mu_\phi(\mathbf{x})$ and $\sigma^2_\phi(\mathbf{x})$ are outputs of a neural network, similarly to the case of the decoder. In practice, we can have a shared neural network $\mathrm{NN}(\mathbf{x})$ that outputs $2M$ values that are further split into M values for the mean μ and M values for the variance σ^2. For convenience, we consider a diagonal covariance matrix. We could use flexible posteriors (see Sect. 4.4.2). Moreover, here we take the standard Gaussian prior. We will comment on that later (see Sect. 4.4.1).

4.3.3.2 Reparameterization Trick

So far, we played around with the log-likelihood and we ended up with the ELBO. However, there is still a problem with calculating the expected value, because it contains an integral! Therefore, the question is how we can calculate it and why it is better than the MC-approximation of the log-likelihood without the variational posterior. In fact, we will use the MC-approximation, but now, instead of sampling from the prior $p(\mathbf{z})$, we will sample from the variational posterior $q_\phi(\mathbf{z}|\mathbf{x})$. Is it better? Yes, because the variational posterior assigns typically more probability mass to a smaller region than the prior. If you play around with your code of a VAE and examine the variance, you will probably notice that the variational posteriors are almost deterministic (whether it is good or bad is rather an open question). As a result, we should get a better approximation! However, there is still an issue with the variance of the approximation. Simply speaking, if we sample $\mathbf{z}$ from $q_\phi(\mathbf{z}|\mathbf{x})$, plug them into the ELBO, and calculate gradients with respect to the parameters of a neural network ϕ, the variance of the gradient may still be pretty large! A possible solution to that, first noticed by statisticians (e.g., see [8]), is the approach of **reparameterizing** the distribution. The idea is to realize that we can express a random variable as a composition of primitive transformations (e.g., arithmetic operations, logarithm, etc.) of an independent random variable with a simple distribution. For instance, if we consider a Gaussian random variable z with a mean μ and a variance σ^2, and an independent random variable $\epsilon \sim \mathcal{N}(\epsilon|0, 1)$, then the following holds (see Fig. 4.3):

$$z = \mu + \sigma \cdot \epsilon. \tag{4.30}$$

Now, if we start sampling ϵ from the standard Gaussian and apply the above transformation, then we get a sample from $\mathcal{N}(z|\mu, \sigma)$!

If you do not remember this fact from statistics, or you simply do not believe me, write a simple code for that and play around with it. In fact, this idea could be applied to many more distributions [9].

The **reparameterization trick** could be used in the encoder $q_\phi(\mathbf{z}|\mathbf{x})$. As observed by Kingma and Welling [6] and Rezende et al. [7], we can drastically reduce the variance of the gradient by using this reparameterization of the Gaussian distribution. Why? Because the randomness comes from the independent source $p(\epsilon)$, and we calculate gradient with respect to a deterministic function (i.e., a neural

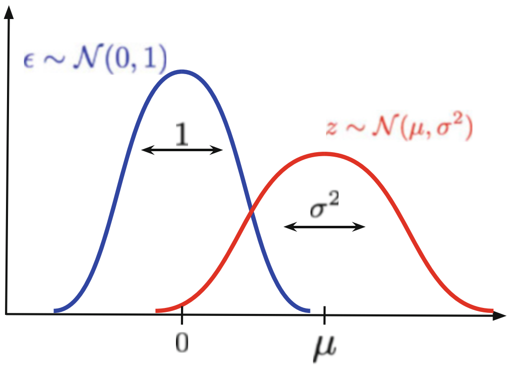

Fig. 4.3 An example of reparameterizing a Gaussian distribution: We scale ϵ distributed according to the standard Gaussian by σ and shift it by μ

network), not random objects. Even better, since we learn the VAE using stochastic gradient descent, it is enough to sample $\mathbf{z}$ only once during training!

4.3.4 *VAE in Action!*

We went through a lot of theory and discussions, and you might think it is impossible to implement a VAE. However, it is actually simpler than it might look. Let us sum up what we know so far and focus on very specific distributions and neural networks.

First of all, we will use the following distributions:

- $q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}\left(\mathbf{z}|\mu_\phi(\mathbf{x}), \sigma^2_\phi(\mathbf{x})\right)$;
- $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, \mathbf{I})$;
- $p_\theta(\mathbf{x}|\mathbf{z}) = \text{Categorical}(\mathbf{x}|\theta(\mathbf{z}))$.

We assume that $x_d \in \mathcal{X} = \{0, 1, \ldots, L-1\}$.

Next, we will use the following networks:

- The *encoder network*:

$$\mathbf{x} \in \mathcal{X}^D \rightarrow \text{Linear}(D, 256) \rightarrow \text{LeakyReLU} \rightarrow$$
$$\text{Linear}(256, 2 \cdot M) \rightarrow \text{split} \rightarrow \mu \in \mathbb{R}^M,\ \log \sigma^2 \in \mathbb{R}^M.$$

  Notice that the last layer outputs $2M$ values because we must have M values for the mean and M values for the (log-)variance. Moreover, a variance must be positive; therefore, instead, we consider the logarithm of the variance because it can take real values then. As a result, we do not need to bother about variances being always positive. An alternative is to apply a non-linearity like softplus.
- The *decoder network*:

$$\mathbf{z} \in \mathbb{R}^M \rightarrow \text{Linear}(M, 256) \rightarrow \text{LeakyReLU} \rightarrow$$
$$\text{Linear}(256, D \cdot L) \rightarrow \text{reshape} \rightarrow \text{softmax} \rightarrow \theta \in [0, 1]^{D\times L}.$$

Since we use the categorical distribution for $\mathbf{x}$, the outputs of the decoder network are probabilities. Thus, the last layer must output $D \cdot L$ values, where D is the number of pixels and L is the number of possible values of a pixel. Then, we must reshape the output to a tensor of the following shape: (B, D, L), where B is the batch size. Afterward, we can apply the softmax activation function to obtain probabilities.

Finally, for a given dataset $\mathcal{D} = \{\mathbf{x}_n\}_{n=1}^{N}$, the training objective is the ELBO where we use a single sample from the variational posterior $\mathbf{z}_{\phi,n} = \mu_\phi(\mathbf{x}_n) + \sigma_\phi(\mathbf{x}_n) \odot \epsilon$. We must remember that in almost any available package we minimize by default, so we must take the negative sign, namely:

$$-ELBO(\mathcal{D}; \theta, \phi) = \sum_{n=1}^{N} - \left\{ \ln \text{Categorical}\left(\mathbf{x}_n | \theta\left(\mathbf{z}_{\phi,n}\right)\right) + \left[\ln \mathcal{N}\left(\mathbf{z}_{\phi,n} | \mu_\phi(\mathbf{x}_n), \sigma_\phi^2(\mathbf{x}_n)\right) + \ln \mathcal{N}\left(\mathbf{z}_{\phi,n} | 0, \mathbf{I}\right) \right] \right\}. \tag{4.31}$$

So as you can see, the whole math boils down to a relatively simple learning procedure:

1. Take $\mathbf{x}_n$ and apply the encoder network to get $\mu_\phi(\mathbf{x}_n)$ and $\ln \sigma_\phi^2(\mathbf{x}_n)$.
2. Calculate $\mathbf{z}_{\phi,n}$ by applying the reparameterization trick, $\mathbf{z}_{\phi,n} = \mu_\phi(\mathbf{x}_n) + \sigma_\phi(\mathbf{x}_n) \odot \epsilon$, where $\epsilon \sim \mathcal{N}(0, \mathbf{I})$.
3. Apply the decoder network to $\mathbf{z}_{\phi,n}$ to get the probabilities $\theta(\mathbf{z}_{\phi,n})$.
4. Calculate the ELBO by plugging in $\mathbf{x}_n$, $\mathbf{z}_{\phi,n}$, $\mu_\phi(\mathbf{x}_n)$, and $\ln \sigma_\phi^2(\mathbf{x}_n)$.

4.3.5 Code

Now, all components are ready to be turned into a code! For the full implementation, please take a look at https://github.com/jmtomczak/intro_dgm. Here, we focus only on the code for the VAE model. We provide details in the comments. We divide the code into four classes: Encoder, Decoder, Prior, and VAE. It might look like overkill, but it may help you to think of the VAE as a composition of three parts and better comprehend the whole approach.

```python
class Encoder(nn.Module):
    def __init__(self, encoder_net):
        super(Encoder, self).__init__()

        # The init of the encoder network.
        self.encoder = encoder_net

    # The reparameterization trick for Gaussians.
    @staticmethod
    def reparameterization(mu, log_var):
        # The formula is the following:
        # z = mu + std * epsilon
        # epsilon ~ Normal(0,1)

        # First, we need to get std from log-variance.
        std = torch.exp(0.5*log_var)

        # Second, we sample epsilon from Normal(0,1).
        eps = torch.randn_like(std)

        # The final output
        return mu + std * eps

    # This function implements the output of the encoder network (i.e., parameters of a Gaussian).
    def encode(self, x):
        # First, we calculate the output of the encoder network of size 2M.
        h_e = self.encoder(x)
        # Second, we must divide the output to the mean and the log-variance.
        mu_e, log_var_e = torch.chunk(h_e, 2, dim=1)
        return mu_e, log_var_e

    # Sampling procedure.
    def sample(self, x=None, mu_e=None, log_var_e=None):
        #If we don't provide a mean and a log-variance, we must first calculate it:
        if (mu_e is None) and (log_var_e is None):
            mu_e, log_var_e = self.encode(x)
        # Or the final sample
        else:
        # Otherwise, we can simply apply the reparameterization trick!
            if (mu_e is None) or (log_var_e is None):
                raise ValueError('mu and log-var can't be None!')
        z = self.reparameterization(mu_e, log_var_e)
        return z

    # This function calculates the log-probability that is later used for calculating the ELBO.
    def log_prob(self, x=None, mu_e=None, log_var_e=None, z=None):
        # If we provide x alone, then we can calculate a corresponding sample:
        if x is not None:
            mu_e, log_var_e = self.encode(x)
            z = self.sample(mu_e=mu_e, log_var_e=log_var_e)
        else:
        # Otherwise, we should provide mu, log-var and z!
            if (mu_e is None) or (log_var_e is None) or (z is None):
                raise ValueError('mu, log-var, z can't be None')

        return log_normal_diag(z, mu_e, log_var_e)

    # PyTorch forward pass: it is either log-probability (by
    default) or sampling.
    def forward(self, x, type='log_prob'):
        assert type in ['encode', 'log_prob'], 'Type could be
    either encode or log_prob'
        if type == 'log_prob':
            return self.log_prob(x)
        else:
            return self.sample(x)
```

Listing 4.1 An encoder class

```
class Decoder(nn.Module):
    def __init__(self, decoder_net, distribution='categorical',
    num_vals=None):
        super(Decoder, self).__init__()

        # The decoder network.
        self.decoder = decoder_net
        # The distribution used for the decoder (it is
    categorical by default, as discussed above).
        self.distribution = distribution
        # The number of possible values. This is important for
    the categorical distribution.
        self.num_vals=num_vals

    # This function calculates parameters of the likelihood
    function p(x|z)
    def decode(self, z):
        # First, we apply the decoder network.
        h_d = self.decoder(z)

        # In this example, we use only the categorical
    distribution...
        if self.distribution == 'categorical':
            # We save the shapes: batch size
            b = h_d.shape[0]
            # and the dimensionality of x.
            d = h_d.shape[1]//self.num_vals
            # Then we reshape to (Batch size, Dimensionality,
    Number of Values).
            h_d = h_d.view(b, d, self.num_vals)
            # To get probabilities, we apply softmax.
            mu_d = torch.softmax(h_d, 2)
            return [mu_d]
        # ... however, we also present the Bernoulli distribution
    . We are nice, aren't we?
        elif self.distribution == 'bernoulli':
            # In the Bernoulli case, we have x_d \in {0,1}.
    Therefore, it is enough to output a single probability,
            # because p(x_d=1|z) = \theta and p(x_d=0|z) = 1 - \
    theta
```

```
            mu_d = torch.sigmoid(h_d)
            return [mu_d]

        else:
            raise ValueError('Only: `categorical`, `bernoulli`')

    # This function implements sampling from the decoder.
    def sample(self, z):
        outs = self.decode(z)

        if self.distribution == 'categorical':
            # We take the output of the decoder
            mu_d = outs[0]
            # and save shapes (we will need that for reshaping).
            b = mu_d.shape[0]
            m = mu_d.shape[1]
            # Here we use reshaping
            mu_d = mu_d.view(mu_d.shape[0], -1, self.num_vals)
            p = mu_d.view(-1, self.num_vals)
            # Eventually, we sample from the categorical (the built-in PyTorch function).
            x_new = torch.multinomial(p, num_samples=1).view(b,m)

        elif self.distribution == 'bernoulli':
            # In the case of Bernoulli, we don't need any reshaping
            mu_d = outs[0]
            # and we can use the built-in PyTorch sampler!
            x_new = torch.bernoulli(mu_d)

        else:
            raise ValueError('Only: `categorical`, `bernoulli`')

        return x_new

    # This function calculates the conditional log-likelihood function.
    def log_prob(self, x, z):
        outs = self.decode(z)

        if self.distribution == 'categorical':
            mu_d = outs[0]
            log_p = log_categorical(x, mu_d, num_classes=self.num_vals, reduction='sum', dim=-1).sum(-1)

        elif self.distribution == 'bernoulli':
            mu_d = outs[0]
            log_p = log_bernoulli(x, mu_d, reduction='sum', dim=-1)

        else:
            raise ValueError('Only: `categorical`, `bernoulli`')

        return log_p

    # The forward pass is either a log-prob or a sample.
    def forward(self, z, x=None, type='log_prob'):
        assert type in ['decoder', 'log_prob'], 'Type could be either decode or log_prob'
        if type == 'log_prob':
            return self.log_prob(x, z)
        else:
            return self.sample(x)
```

Listing 4.2 A decoder class

```
# The current implementation of the prior is very simple, namely, it is a standard Gaussian.
# We could have used a built-in PyTorch distribution. However, we didn't do that for two reasons:
# (i) It is important to think of the prior as a crucial component in VAEs.
# (ii) We can implement a learnable prior (e.g., a flow-based prior, VampPrior, a mixture of distributions).
class Prior(nn.Module):
    def __init__(self, L):
        super(Prior, self).__init__()
        self.L = L

    def sample(self, batch_size):
        z = torch.randn((batch_size, self.L))
        return z

    def log_prob(self, z):
        return log_standard_normal(z)
```

Listing 4.3 A prior class

```
class VAE(nn.Module):
    def __init__(self, encoder_net, decoder_net, num_vals=256, L=16, likelihood_type='categorical'):
        super(VAE, self).__init__()

        print('VAE by JT.')

        self.encoder = Encoder(encoder_net=encoder_net)
        self.decoder = Decoder(distribution=likelihood_type, decoder_net=decoder_net, num_vals=num_vals)
        self.prior = Prior(L=L)

        self.num_vals = num_vals

        self.likelihood_type = likelihood_type

    def forward(self, x, reduction='avg'):
        # encoder
        mu_e, log_var_e = self.encoder.encode(x)
        z = self.encoder.sample(mu_e=mu_e, log_var_e=log_var_e)
```

```python

        # ELBO
        RE = self.decoder.log_prob(x, z)
        KL = (self.prior.log_prob(z) - self.encoder.log_prob(mu_e=mu_e, log_var_e=log_var_e, z=z)).sum(-1)

        if reduction == 'sum':
            return -(RE + KL).sum()
        else:
            return -(RE + KL).mean()

    def sample(self, batch_size=64):
        z = self.prior.sample(batch_size=batch_size)
        return self.decoder.sample(z)
```

Listing 4.4 A VAE class

```python
# Examples of neural networks used for parameterizing the encoder and the decoder.

# Remember that the encoder outputs 2 times more values because we need L means and L log-variances for a Gaussian.
encoder = nn.Sequential(nn.Linear(D, M), nn.LeakyReLU(),
                        nn.Linear(M, M), nn.LeakyReLU(),
                        nn.Linear(M, 2 * L))

# Here we must remember that if we use the categorical distribution, we must output num_vals per each pixel.
decoder = nn.Sequential(nn.Linear(L, M), nn.LeakyReLU(),
                        nn.Linear(M, M), nn.LeakyReLU(),
                        nn.Linear(M, num_vals * D))
```

Listing 4.5 Examples of networks

Perfect! Now we are ready to run the full code and after training our VAE, we should obtain results similar to those like in Fig. 4.4.

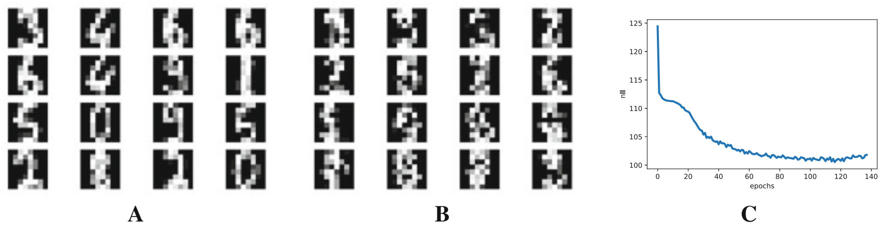

Fig. 4.4 An example of outcomes after the training: (**a**) Randomly selected real images. (**b**) Unconditional generations from the VAE. (**c**) The validation curve during training

4.3.6 Typical Issues with VAEs

VAEs constitute a very powerful class of models, mainly due to their flexibility. Unlike flow-based models, they do not require the invertibility of neural networks and, thus, we can use any arbitrary architecture for encoders and decoders. In contrast to ARMs, they learn a low-dimensional data representation and we can control the bottleneck (i.e., the dimensionality of the latent space). However, they also suffer from several issues. Except the ones mentioned before (i.e., a necessity of an efficient integral estimation, a gap between the ELBO and the log-likelihood function for too simplistic variational posteriors), the potential problems are the following:

- Let us take a look at the ELBO and the regularization term. For a non-trainable prior like the standard Gaussian, the regularization term will be minimized if $\forall_{\mathbf{x}} q_{\phi}(\mathbf{z}|\mathbf{x}) = p(\mathbf{z})$. This may happen if the decoder is so powerful that it treats $\mathbf{z}$ as a noise, e.g., a decoder is expressed by an ARM [10]. This issue is known as the *posterior collapse* [11].
- Another issue is associated with a mismatch between the aggregated posterior, $q_{\phi}(\mathbf{z}) = \frac{1}{N}\sum_{n} q_{\phi}(\mathbf{z}|\mathbf{x}_n)$, and the prior $p(\mathbf{z})$. Imagine that we have the standard Gaussian prior and the aggregated posterior (i.e., an average of variational posteriors over all training data). As a result, there are regions where there prior assigns high probability but the aggregated posterior assigns low probability, or other way around. Then, sampling from these holes provides unrealistic latent values and the decoder produces images of very low quality. This problem is referred to as the *hole problem* [12].
- The last problem we want to discuss is more general and, in fact, it affects all deep generative models. As it was noticed in [13], the deep generative models (including VAEs) fail to properly detect out-of-distribution examples. Out-of-distribution datapoints are examples that follow a totally different distribution than the one a model was trained on. For instance, let us assume that our model is trained on MNIST, then Fashion MNIST examples are out-of-distribution. Thus, an intuition tells that a properly trained deep generative model should assign high probability to in-distribution examples and low probability to out-of-distribution points. Unfortunately, as shown in [13], this is not the case. The *out-of-distribution problem* remains one of the main unsolved problems in deep generative modeling [14].

4.3.7 There Is More!

There are a plethora of papers that extend VAEs and apply them to many problems. Below, we will list out selected papers and only touch upon the vast literature on the topic!

Estimation of the Log-Likelihood Using Importance Weighting As we indicated multiple times, the ELBO is the lower bound to the log-likelihood and it rather should not be used as a good estimate of the log-likelihood. In [7, 15], an *importance weighting* procedure is advocated to better approximate the log-likelihood, namely:

$$\ln p(\mathbf{x}) \approx \ln \frac{1}{K} \sum_{k=1}^{K} \frac{p(\mathbf{x}|\mathbf{z}_k)}{q_{\phi}(\mathbf{z}_k|\mathbf{x})}, \tag{4.32}$$

where $\mathbf{z}_k \sim q_{\phi}(\mathbf{z}_k|\mathbf{x})$. Notice that the logarithm is **outside** the expected value. As shown in [15], using importance weighting with sufficiently large K gives a good estimate of the log-likelihood. In practice, K is taken to be 512 or more if the computational budget allows.

Enhancing VAEs: Better Encoders After introducing the idea of VAEs, many papers focused on proposing a flexible family of variational posteriors. The most prominent direction is based on utilizing conditional flow-based models [16–21]. We discuss this topic more in Sect. 4.4.2.

Enhancing VAEs: Better Decoders VAEs allow using any neural network to parameterize the decoder. Therefore, we can use fully connected networks, fully convolutional networks, ResNets, or ARMs. For instance, in [22], a PixelCNN-based decoder was used utilized in a VAE.

Enhancing VAEs: Better Priors As mentioned before, this could be a serious issue if there is a big mismatch between the aggregated posterior and the prior. There are many papers that try to alleviate this issue by using a multimodal prior mimicking the aggregated posterior (known as the VampPrior) [23], or a flow-based prior (e.g., [24, 25]), an ARM-based prior [26], or using an idea of resampling [27]. We present various priors in Sect. 4.4.1.

Extending VAEs Here, we present the unsupervised version of VAEs. However, there is no restriction to that and we can introduce labels or other variables. In [28] a semi-supervised VAE was proposed. This idea was further extended to the concept of fair representations [29, 30]. In [30], the authors proposed a specific latent representation that allows domain generalization in VAEs. In [31] variational inference and the reparameterization trick were used for Bayesian Neural Nets. [31] is not necessarily introducing a VAE, but a VAE-like way of dealing with Bayesian neural nets.

VAEs for Non-image Data So far, we explain everything on images. However, there is no restriction on that! In [11] a VAE was proposed to deal with sequential data (e.g., text). The encoder and the decoder were parameterized by LSTMs. An interesting application of the VAE framework was also presented in [32] where VAEs were used for the molecular graph generation. In [26] the authors proposed a VAE-like model for video compression.

Different Latent Spaces Typically, the Euclidean latent space is considered. However, the VAE framework allows us to think of other spaces. For instance, in [33, 34] a hyperspherical latent space was used, and in [35] the hyperbolic latent space was utilized. More details about hyperspherical VAEs could be found in Sect. 4.4.2.3.

Discrete Latent Spaces We discuss the VAE framework with continuous latent variables. However, an interesting question is how to deal with discrete latent variables. The problem here is that we cannot use the reparameterization trick anymore. There are two potential solutions to that. First, a relaxation to the discrete variables could be used like the *Gumbel-Softmax trick* [36, 37]. Second, a method for a gradient estimation could be used [38].

The Posterior Collapse There were many ideas proposed to deal with the posterior collapse. For instance, [39] proposes to update variational posteriors more often than the decoder. In [40] a new architecture of the decoder is proposed by introducing *skip connections* to allow a better flow of information (thus, the gradients) in the decoder.

Various Perspectives on the Objective The core of the VAE is the ELBO. However, we can consider different objectives. For instance, [41] proposes an upper bound to the log-likelihood that is based on the chi-square divergence (CUBO). In [10] an information-theoretic perspective on the ELBO is presented. [42] introduced the β-VAE where the regularization term is weighted by a fudge factor β. The objective does not correspond to the lower bound of the log-likelihood though.

Deterministic Regularized Auto-Encoders We can take a look at the VAE and the objective, as mentioned before, and think of it as a regularized version of an auto-encoder with a stochastic encoder and a stochastic decoder. [43] "peeled off" VAEs from all stochasticity and indicated similarities between deterministic regularized auto-encoders and VAEs and highlighted potential issues with VAEs. Moreover, they brilliantly pointed out that even with a deterministic encoder, due to the stochasticity of the empirical distribution, we can fit a model to the aggregated posterior. As a result, the deterministic (regularized) auto-encoder could be turned into a generative model by sampling from a model of the aggregated posterior, $p_{\lambda}(\mathbf{z})$, and then, deterministically, mapping $\mathbf{z}$ to the space of observable $\mathbf{x}$. In my opinion, this direction should be further explored and an important question is whether we indeed need any stochasticity at all.

Hierarchical VAEs Very recently, there are many VAEs with a deep, hierarchical structure of latent variables that achieved remarkable results! The most important ones are definitely BIVA [44], NVAE [45], and very deep VAEs [46]. Another interesting perspective on a deep, hierarchical VAE was presented in [25] where, additionally, a series of deterministic functions was used. We delve into that topic in Sect. 4.5.

Adversarial Auto-Encoders Another interesting perspective on VAEs is presented in [47]. Since learning the aggregated posterior as the prior is an important component mentioned in some papers (e.g., [23, 48]), a different approach would be to train the prior with an adversarial loss. Further, [47] present various ideas how auto-encoders could benefit from adversarial learning.

4.4 Improving Variational Auto-Encoders

4.4.1 Priors

Insights from Rewriting the ELBO

One of the crucial components of VAEs is the marginal distribution over **z**'s. Now, we will take a closer look at this distribution, also called the *prior*. Before we start thinking about improving it, we inspect the ELBO one more time. We can write ELBO as follows:

$$\mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[\ln p(\mathbf{x})] \geq \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p_\theta(\mathbf{x}|\mathbf{z}) + \ln p_\lambda(\mathbf{z}) - \ln q_\phi(\mathbf{z}|\mathbf{x})\right]\right], \tag{4.33}$$

where we explicitly highlight the summation over training data, namely, the expected value with respect to **x**'s from the empirical distribution $p_{data}(\mathbf{x}) = \frac{1}{N}\sum_{n=1}^{N}\delta(\mathbf{x}-\mathbf{x}_n)$, and $\delta(\cdot)$ is the Dirac delta.

The ELBO consists of two parts, namely, the reconstruction error:

$$RE \triangleq \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p_\theta(\mathbf{x}|\mathbf{z})\right]\right], \tag{4.34}$$

and the regularization term between the encoder and the prior:

$$\Omega \triangleq \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p_\lambda(\mathbf{z}) - \ln q_\phi(\mathbf{z}|\mathbf{x})\right]\right]. \tag{4.35}$$

Further, let us play a little bit with the regularization term Ω:

$$\Omega = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p_\lambda(\mathbf{z}) - \ln q_\phi(\mathbf{z}|\mathbf{x})\right]\right] \tag{4.36}$$

$$= \int p_{data}(\mathbf{x}) \int q_\phi(\mathbf{z}|\mathbf{x})\left[\ln p_\lambda(\mathbf{z}) - \ln q_\phi(\mathbf{z}|\mathbf{x})\right] \mathrm{d}\mathbf{z}\, \mathrm{d}\mathbf{x} \tag{4.37}$$

$$= \iint p_{data}(\mathbf{x}) q_\phi(\mathbf{z}|\mathbf{x})\left[\ln p_\lambda(\mathbf{z}) - \ln q_\phi(\mathbf{z}|\mathbf{x})\right] \mathrm{d}\mathbf{z}\, \mathrm{d}\mathbf{x} \tag{4.38}$$

$$= \iint \frac{1}{N}\sum_{n}\delta(\mathbf{x}-\mathbf{x}_n) q_\phi(\mathbf{z}|\mathbf{x})\left[\ln p_\lambda(\mathbf{z}) - \ln q_\phi(\mathbf{z}|\mathbf{x})\right] \mathrm{d}\mathbf{z}\, \mathrm{d}\mathbf{x} \tag{4.39}$$

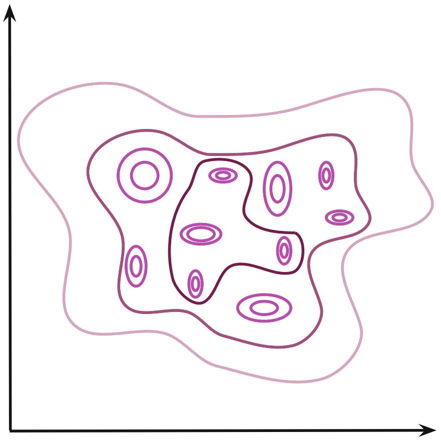
Fig. 4.5 An example of the aggregated posterior. Individual points are encoded as Gaussians in the 2D latent space (magenta), and the mixture of variational posteriors (the aggregated posterior) is presented by contours

$$= \int \frac{1}{N}\sum_{n=1}^{N} q_{\phi}(\mathbf{z}|\mathbf{x}_n)\left[\ln p_{\lambda}(\mathbf{z}) - \ln q_{\phi}(\mathbf{z}|\mathbf{x}_n)\right]\mathrm{d}\mathbf{z} \tag{4.40}$$

$$= \int \frac{1}{N}\sum_{n=1}^{N} q_{\phi}(\mathbf{z}|\mathbf{x}_n)\ln p_{\lambda}(\mathbf{z})\mathrm{d}\mathbf{z} - \int \frac{1}{N}\sum_{n=1}^{N} q_{\phi}(\mathbf{z}|\mathbf{x}_n)\ln q_{\phi}(\mathbf{z}|\mathbf{x}_n)\mathrm{d}\mathbf{z} \tag{4.41}$$

$$= \int q_{\phi}(\mathbf{z})\ln p_{\lambda}(\mathbf{z})\mathrm{d}\mathbf{z} - \int \sum_{n=1}^{N}\frac{1}{N} q_{\phi}(\mathbf{z}|\mathbf{x}_n)\ln q_{\phi}(\mathbf{z}|\mathbf{x}_n)\mathrm{d}\mathbf{z} \tag{4.42}$$

$$= -\mathbb{CE}\left[q_{\phi}(\mathbf{z})||p_{\lambda}(\mathbf{z})\right] + \mathbb{H}\left[q_{\phi}(\mathbf{z}|\mathbf{x})\right], \tag{4.43}$$

where we use the property of the Dirac delta: $\int \delta(a-a')f(a)\mathrm{d}a = f(a')$, and we use the notion of the **aggregated posterior** [47, 48] defined as follows:

$$q(\mathbf{z}) = \frac{1}{N}\sum_{n=1}^{N} q_{\phi}\left(\mathbf{z}|\mathbf{x}_n\right). \tag{4.44}$$

An example of the aggregated posterior is schematically depicted in Fig. 4.5.

Eventually, we obtain two terms:

(i) The first one, $\mathbb{CE}\left[q_{\phi}(\mathbf{z})||p_{\lambda}(\mathbf{z})\right]$, is the cross-entropy between the aggregated posterior and the prior.
(ii) The second term, $\mathbb{H}\left[q_{\phi}(\mathbf{z}|\mathbf{x})\right]$, is the conditional entropy of $q_{\phi}(\mathbf{z}|\mathbf{x})$ with the empirical distribution $p_{data}(\mathbf{x})$.

I highly recommend doing this derivation step-by-step, as it helps a lot in understanding what is going on here. Interestingly, there is another possibility to rewrite Ω using three terms, with the total correlation [49]. We will not use it here, so it is left as a "homework."

Anyway, one may ask why is it useful to rewrite the ELBO? The answer is rather straightforward: We can analyze it from a different perspective! In this section, we will focus on the **prior**, an important component in the generative part that is very often neglected. Many Bayesianists are stating that a prior should not be learned. But VAEs are not Bayesian models, please remember that! Besides, who says we cannot learn the prior? As we will see shortly, a non-learnable prior could be pretty annoying, especially for the generation process.

What Does ELBO Tell Us About the Prior?

Alright, we see that Ω consists of the cross-entropy and the entropy. Let us start with the entropy since it is easier to be analyzed. While optimizing, we want to maximize the ELBO and, hence, we maximize the entropy:

$$\mathbb{H}\left[q_{\phi}(\mathbf{z}|\mathbf{x})\right] = -\int \sum_{n=1}^{N} \frac{1}{N} q_{\phi}(\mathbf{z}|\mathbf{x}_n) \ln q_{\phi}(\mathbf{z}|\mathbf{x}_n) \mathrm{d}\mathbf{z}. \tag{4.45}$$

Before we make any conclusions, we should remember that we consider Gaussian encoders, $q_{\phi}(\mathbf{z}|\mathbf{x}) = \mathcal{N}\left(\mathbf{z}|\mu(\mathbf{x}), \sigma^2(\mathbf{x})\right)$. The entropy of a Gaussian distribution with a diagonal covariance matrix is equal to $\frac{1}{2}\sum_i \ln\left(2e\pi\sigma_i^2\right)$. Then, the question is when this quantity is maximized? The answer is: $\sigma_i^2 \rightarrow +\infty$. In other words, the entropy terms tries to stretch the encoders as much as possible by enlarging their variances! Of course, this does not happen in practice because we use the encoder together with the decoder in the RE term and the decoder tries to make the encoder as peaky as possible (i.e., ideally one $\mathbf{x}$ for one $\mathbf{z}$, like in the non-stochastic auto-encoder).

The second term in Ω is the cross-entropy:

$$\mathbb{CE}\left[q_{\phi}(\mathbf{z})||p_{\lambda}(\mathbf{z})\right] = -\int q_{\phi}(\mathbf{z}) \ln p_{\lambda}(\mathbf{z}) \mathrm{d}\mathbf{z}. \tag{4.46}$$

The cross-entropy term influences the VAE in a different manner. First, we can ask the question how to interpret the cross-entropy between $q_{\phi}(\mathbf{z})$ and $p_{\lambda}(\mathbf{z})$. In general, the cross-entropy tells us the average number of bits (or rather nats because we use the natural logarithm) needed to identify an event drawn from $q_{\phi}(\mathbf{z}$ if a coding scheme used for it is $p_{\lambda}(\mathbf{z})$. Notice that in Ω we have the negative cross-entropy. Since we maximize the ELBO, it means that we aim for minimizing $\mathbb{CE}\left[q_{\phi}(\mathbf{z})||p_{\lambda}(\mathbf{z})\right]$. This makes sense because we would like $q_{\phi}(\mathbf{z})$ to match $p_{\lambda}(\mathbf{z})$. And we have accidentally touched upon the most important issue here: What do we really want here? The cross-entropy forces the aggregated posterior to **match** the prior! That is the reason why we have this term here. If you think about it, it is a beautiful result that gives another connection between VAEs and the information theory.

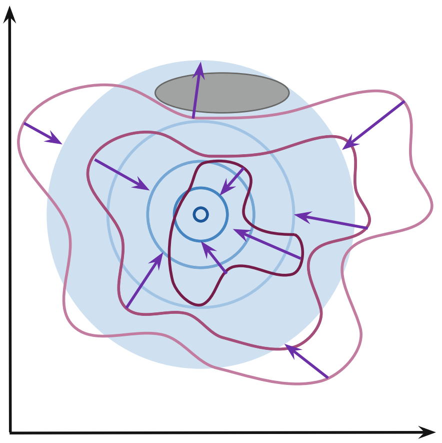
Fig. 4.6 An example of the effect of the cross-entropy optimization with a non-learnable prior. The aggregated posterior (purple contours) tries to match the non-learnable prior (in blue). The purple arrows indicate the change of the aggregated posterior. An example of a hole is presented as a dark gray ellipse

Alright, so we see what the cross-entropy does, but there are two possibilities here. First, the prior is fixed (**non-learnable**), e.g., the standard Gaussian prior. Then, optimizing the cross-entropy *pushes* the aggregated posterior to match the prior. It is schematically depicted in Fig. 4.6. The prior acts like an anchor and the *amoeba* of the aggregated posterior moves so that to fit the prior. In practice, this optimization process is troublesome because the decoder forces the encoder to be peaked and, in the end, it is almost impossible to match a fixed-shaped prior. As a result, we obtain **holes**, namely, regions in the latent space where the aggregated posterior assigns low probability while the prior assigns (relatively) high probability (see a dark gray ellipse in Fig. 4.6). This issue is especially apparent in generations because sampling from the prior, from the hole, may result in a sample that is of an extremely low quality. You can read more about it in [12].

On the other hand, if we consider a learnable **prior**, the situation looks different. The optimization allows to change the aggregated posterior **and** the prior. As the consequence, both distributions try to match each other (see Fig. 4.7). The problem of holes is then less apparent, especially if the prior is flexible enough. However, we can face other optimization issues when the prior and the aggregated posteriors chase each other. In practice, the learnable prior seems to be a better option, but it is still an open question whether training all components at once is the best approach. Moreover, the learnable prior does not impose any specific constraint on the latent representation, e.g., sparsity. This could be another problem that would result in undesirable problems (e.g., non-smooth encoders).

Eventually, we can ask the fundamental question: What is the *best* prior then?! The answer is already known and is hidden in the cross-entropy term: It is the aggregated posterior. If we take $p_{\lambda}(\mathbf{z}) = \sum_{n=1}^{N} \frac{1}{N} q_{\phi}(\mathbf{z}|\mathbf{x}_n)$, then, theoretically, the cross-entropy equals the entropy of $q_{\phi}(\mathbf{z})$ and the regularization term Ω is smallest. However, in practice, this is infeasible because:

- We cannot sum over tens of thousands of points and backpropagate through them.

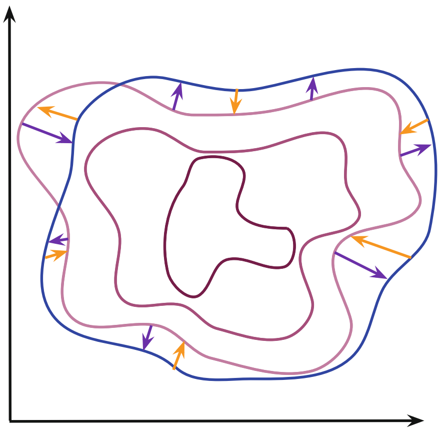

Fig. 4.7 An example of the effect of the cross-entropy optimization with a learnable prior. The aggregated posterior (purple contours) tries to match the learnable prior (blue contours). Notice that the aggregated posterior is modified to fit the prior (purple arrows), but also the prior is updated to cover the aggregated posterior (orange arrows)

- This result is fine from the theoretical point of view; however, the optimization process is stochastic and could cause additional errors.
- As mentioned earlier, choosing the aggregated posterior as the prior does not constrain the latent representation in any obvious manner and, thus, the encoder could behave unpredictably.
- The aggregated posterior may work well if the get $N \to +\infty$ points, because then we can get any distribution; however, this is not the case in practice and it contradicts also the first bullet.

As a result, we can keep this theoretical solution in mind and formulate **approximations** to it that are computationally tractable. In the next sections, we will discuss a few of them.

4.4.1.1 Standard Gaussian

The vanilla implementation of the VAE assumes a standard Gaussian marginal (prior) over $\mathbf{z}$, $p_\lambda(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, \mathrm{I})$. This prior is simple, non-trainable (i.e., no extra parameters to learn), and easy to implement. In other words, it is amazing! However, as discussed above, the standard normal distribution could lead to very poor hidden representations with holes resulting from the mismatch between the aggregated posterior and the prior.

To strengthen our discussion, we trained a small VAE with the standard Gaussian prior and a two-dimensional latent space. In Fig. 4.8, we present samples from the encoder for the test data (black dots) and the contour plot for the standard prior. We can spot holes where the aggregated posterior does not assign any points (i.e., the mismatch between the prior and the aggregated posterior).

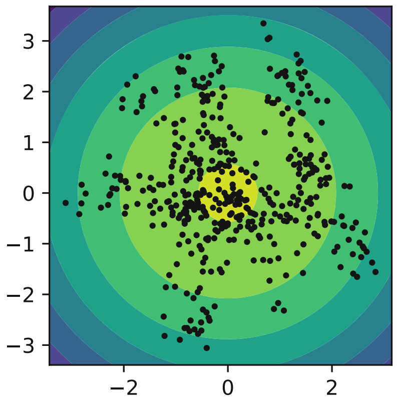

Fig. 4.8 An example of the standard Gaussian prior (contours) and the samples from the aggregated posterior (black dots)

The code for the standard Gaussian prior is presented below:

```python
class StandardPrior(nn.Module):
    def __init__(self, L=2):
        super(StandardPrior, self).__init__()

        self.L = L

        # params weights
        self.means = torch.zeros(1, L)
        self.logvars = torch.zeros(1, L)

    def get_params(self):
        return self.means, self.logvars

    def sample(self, batch_size):
        return torch.randn(batch_size, self.L)

    def log_prob(self, z):
        return log_standard_normal(z)
```

Listing 4.6 A standard Gaussian prior class

4.4.1.2 Mixture of Gaussians

If we take a closer look at the aggregated posterior, we immediately notice that it is a mixture model, and a mixture of Gaussians, to be more precise. Therefore, we can use the Mixture of Gaussians (MoG) prior with K components:

$$p_{\lambda}(\mathbf{z}) = \sum_{k=1}^{K} w_k \mathcal{N}(\mathbf{z}|\mu_k, \sigma_k^2), \tag{4.47}$$

where $\lambda = \left\{\{w_k\}, \{\mu_k\}, \{\sigma_k^2\}\right\}$ are trainable parameters.

Similarly to the standard Gaussian prior, we trained a small VAE with the mixture of Gaussians prior (with $K = 16$) and a two-dimensional latent space. In Fig. 4.9, we present samples from the encoder for the test data (black dots) and the contour plot for the MoG prior. Comparing to the standard Gaussian prior, the MoG prior fits better the aggregated posterior, allowing to *patch* holes.

An example of the code is presented below:

```python
class MoGPrior(nn.Module):
    def __init__(self, L, num_components):
        super(MoGPrior, self).__init__()

        self.L = L
        self.num_components = num_components

        # params
        self.means = nn.Parameter(torch.randn(num_components, 
    self.L)*multiplier)
        self.logvars = nn.Parameter(torch.randn(num_components, 
    self.L))

        # mixing weights
        self.w = nn.Parameter(torch.zeros(num_components, 1, 1))

    def get_params(self):
        return self.means, self.logvars

    def sample(self, batch_size):
        # mu, lof_var
        means, logvars = self.get_params()

        # mixing probabilities
        w = F.softmax(self.w, dim=0)
        w = w.squeeze()

        # pick components
        indexes = torch.multinomial(w, batch_size, replacement=
    True)

        # means and logvars
        eps = torch.randn(batch_size, self.L)
        for i in range(batch_size):
            indx = indexes[i]
            if i == 0:
                z = means[[indx]] + eps[[i]] * torch.exp(logvars
    [[indx]])
            else:
                z = torch.cat((z, means[[indx]] + eps[[i]] * 
    torch.exp(logvars[[indx]])), 0)
        return z

    def log_prob(self, z):
        # mu, lof_var
```

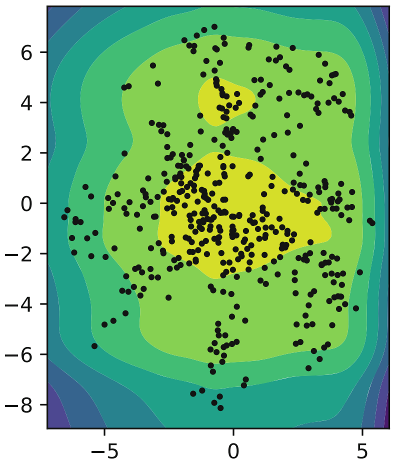

Fig. 4.9 An example of the MoG prior (contours) and the samples from the aggregated posterior (black dots)

```
        means, logvars = self.get_params()

        # mixing probabilities
        w = F.softmax(self.w, dim=0)

        # log-mixture-of-Gaussians
        z = z.unsqueeze(0) # 1 x B x L
        means = means.unsqueeze(1) # K x 1 x L
        logvars = logvars.unsqueeze(1) # K x 1 x L

        log_p = log_normal_diag(z, means, logvars) + torch.log(w)
     # K x B x L
        log_prob = torch.logsumexp(log_p, dim=0, keepdim=False) #
     B x L

        return log_prob
```

Listing 4.7 A Mixture of Gaussians prior class

4.4.1.3 VampPrior: Variational Mixture of Posterior Prior

In [23], it was noticed that we can improve on the MoG prior and approximate the aggregated posterior by introducing *pseudo-inputs*:

$$p_{\lambda}(\mathbf{z}) = \frac{1}{N}\sum_{k=1}^{K} q_{\phi}(\mathbf{z}|\mathbf{u}_k), \tag{4.48}$$

where $\lambda = \left\{\phi, \{\mathbf{u}_k^2\}\right\}$ are trainable parameters and $\mathbf{u}_k \in \mathcal{X}^D$ is a pseudo-input. Notice that ϕ is a part of the trainable parameters. The idea of pseudo-input is to

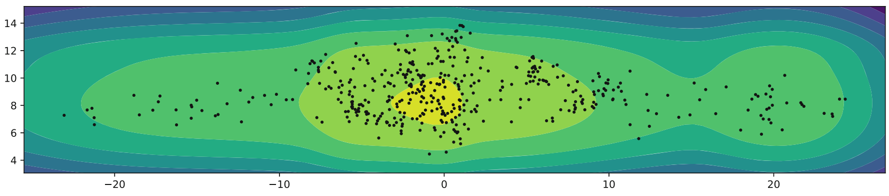

Fig. 4.10 An example of the VampPrior (contours) and the samples from the aggregated posterior (black dots)

randomly initialize objects that mimic observable variables (e.g., images) and learn them by backpropagation.

This approximation to the aggregated posterior is called the **variational mixture of posterior prior**, VampPrior for short. In [23] you can find some interesting properties and further analysis of the VampPrior. The main drawback of the VampPrior lies in initializing the pseudo-inputs; however, it serves as a good proxy to the aggregated posterior that improves the generative quality of the VAE, e.g., [10, 50, 51].

Alemi et al. [10] presented a nice connection of the VampPrior with the information-theoretic perspective on the VAE. They further proposed to introduce learnable probabilities of the components:

$$p_{\lambda}(\mathbf{z}) = \sum_{k=1}^{K} w_k q_{\phi}(\mathbf{z}|\mathbf{u}_k), \tag{4.49}$$

to allow the VampPrior to select more relevant components (i.e., pseudo-inputs).

As in the previous cases, we train a small VAE with the VampPrior (with $K = 16$) and a two-dimensional latent space. In Fig. 4.10, we present samples from the encoder for the test data (black dots) and the contour plot for the VampPrior. Similar to the MoG prior, the VampPrior fits better the aggregated posterior and has fewer holes. In this case, we can see that the VampPrior allows the encoders to spread across the latent space (notice the values).

An example of an implementation of the VampPrior is presented below:

```
class VampPrior(nn.Module):
    def __init__(self, L, D, num_vals, encoder, num_components,
    data=None):
        super(VampPrior, self).__init__()

        self.L = L
        self.D = D
        self.num_vals = num_vals

        self.encoder = encoder

        # pseudo-inputs
        u = torch.rand(num_components, D) * self.num_vals
        self.u = nn.Parameter(u)

        # mixing weights
        self.w = nn.Parameter(torch.zeros(self.u.shape[0], 1, 1)) # K x 1 x 1

    def get_params(self):
        # u->encoder->mu, lof_var
        mean_vampprior, logvar_vampprior = self.encoder.encode(self.u) #(K x L), (K x L)
        return mean_vampprior, logvar_vampprior

    def sample(self, batch_size):
        # u->encoder->mu, lof_var
        mean_vampprior, logvar_vampprior = self.get_params()

        # mixing probabilities
        w = F.softmax(self.w, dim=0) # K x 1 x 1
        w = w.squeeze()

        # pick components
        indexes = torch.multinomial(w, batch_size, replacement=True)

        # means and logvars
        eps = torch.randn(batch_size, self.L)
        for i in range(batch_size):
            indx = indexes[i]
            if i == 0:
                z = mean_vampprior[[indx]] + eps[[i]] * torch.exp(logvar_vampprior[[indx]])
            else:
                z = torch.cat((z, mean_vampprior[[indx]] + eps[[i]] * torch.exp(logvar_vampprior[[indx]])), 0)
        return z

    def log_prob(self, z):
        # u->encoder->mu, lof_var
        mean_vampprior, logvar_vampprior = self.get_params() # (K x L) & (K x L)

        # mixing probabilities
        w = F.softmax(self.w, dim=0) # K x 1 x 1

        # log-mixture-of-Gaussians
        z = z.unsqueeze(0) # 1 x B x L
        mean_vampprior = mean_vampprior.unsqueeze(1) # K x 1 x L
        logvar_vampprior = logvar_vampprior.unsqueeze(1) # K x 1 x L

        log_p = log_normal_diag(z, mean_vampprior, logvar_vampprior) + torch.log(w) # K x B x L
        log_prob = torch.logsumexp(log_p, dim=0, keepdim=False) # B x L
        return log_prob
```

Listing 4.8 A VampPrior class

4.4.1.4 GTM: Generative Topographic Mapping

In fact, we can use any density estimator to model the prior. In [52] a density estimator called **generative topographic mapping** (GTM) was proposed that defines a grid of K points in a low-dimensional space, $\mathbf{u} \in \mathbb{R}^C$, namely:

$$p(\mathbf{u}) = \sum_{k=1}^{K} w_k \delta(\mathbf{u} - \mathbf{u}_k) \tag{4.50}$$

that is further transformed to a higher-dimensional space by a transformation g_γ. The transformation g_γ predicts parameters of a distribution, e.g., the Gaussian distribution and, thus, $g_\gamma : \mathbb{R}^C \to \mathbb{R}^{2\times M}$. Eventually, we can define the distribution as follows:

$$p_\lambda(\mathbf{z}) = \int p(\mathbf{u}) \mathcal{N}\left(\mathbf{z}|\mu_g(\mathbf{u}), \sigma_g^2(\mathbf{u})\right) \mathrm{d}\mathbf{u} \tag{4.51}$$

$$= \sum_{k=1}^{K} w_k \mathcal{N}\left(\mathbf{z}|\mu_g(\mathbf{u}_k), \sigma_g^2(\mathbf{u}_k)\right), \tag{4.52}$$

where $\mu_g(\mathbf{u})$ and σ_g^2 rare outputs of the transformation $g_\gamma(\mathbf{u})$.

For instance, for $C = 2$ and $K = 3$, we can define the following grid: $\mathbf{u} \in \{[-1,-1], [-1,0], [-1,1], [0,-1], [0,1], [0,1], [1,-1], [1,0], [1,-1]\}$. Notice that the grid is fixed and only the transformation (e.g., a neural network) g_γ is trained.

As in the previous cases, we train a small VAE with the GTM-based prior (with $K = 16$, i.e., a 4×4 grid) and a two-dimensional latent space. In Fig. 4.11, we present samples from the encoder for the test data (black dots) and the contour plot for the GTM-based prior. Similar to the MoG prior and the VampPrior, the GTM-based prior learns a pretty flexible distribution.

An example of an implementation of the GTM-based prior is presented below:

```
class GTMPrior(nn.Module):
    def __init__(self, L, gtm_net, num_components, u_min=-1., u_max=1.):
    super(GTMPrior, self).__init__()

    self.L = L

    # 2D manifold
```

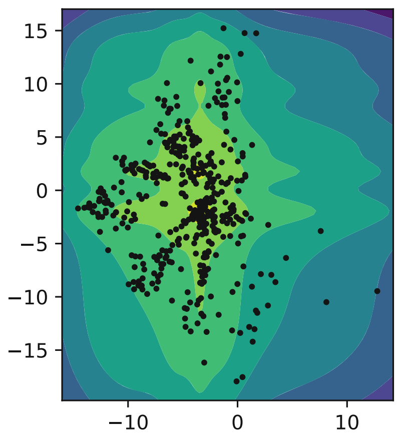

Fig. 4.11 An example of the GTM-based prior (contours) and the samples from the aggregated posterior (black dots)

```
    self.u = torch.zeros(num_components**2, 2) # K**2 x 2
    u1 = torch.linspace(u_min, u_max, steps=num_components)
    u2 = torch.linspace(u_min, u_max, steps=num_components)

    k = 0
    for i in range(num_components):
        for j in range(num_components):
            self.u[k,0] = u1[i]
            self.u[k,1] = u2[j]
            k = k + 1

    # gtm network: u -> z
    self.gtm_net = gtm_net

    # mixing weights
    self.w = nn.Parameter(torch.zeros(num_components**2, 1, 1))

    def get_params(self):
        # u->z
        h_gtm = self.gtm_net(self.u) #K**2 x 2L
        mean_gtm, logvar_gtm = torch.chunk(h_gtm, 2, dim=1) # K
**2 x L and K**2 x L
        return mean_gtm, logvar_gtm

    def sample(self, batch_size):
        # u->z
        mean_gtm, logvar_gtm = self.get_params()

        # mixing probabilities
        w = F.softmax(self.w, dim=0)
        w = w.squeeze()

        # pick components
        indexes = torch.multinomial(w, batch_size, replacement=
True)
```

```

        # means and logvars
        eps = torch.randn(batch_size, self.L)
        for i in range(batch_size):
            indx = indexes[i]
            if i == 0:
                z = mean_gtm[[indx]] + eps[[i]] * torch.exp(
    logvar_gtm[[indx]])
            else:
                z = torch.cat((z, mean_gtm[[indx]] + eps[[i]] *
    torch.exp(logvar_gtm[[indx]])), 0)
        return z

    def log_prob(self, z):
        # u->z
        mean_gtm, logvar_gtm = self.get_params()

        # log-mixture-of-Gaussians
        z = z.unsqueeze(0) # 1 x B x L
        mean_gtm = mean_gtm.unsqueeze(1) # K**2 x 1 x L
        logvar_gtm = logvar_gtm.unsqueeze(1) # K**2 x 1 x L

        w = F.softmax(self.w, dim=0)

        log_p = log_normal_diag(z, mean_gtm, logvar_gtm) + torch.
    log(w) # K**2 x B x L
        log_prob = torch.logsumexp(log_p, dim=0, keepdim=False) #
     B x L

        return log_prob
```

Listing 4.9 A GTM-based prior class

4.4.1.5 GTM-VampPrior

As mentioned earlier, the main issue with the VampPrior is the initialization of the pseudo-inputs. Instead, we can use the idea of the GTM to learn the pseudo-inputs. Combining these two approaches, we get the following prior:

$$p_{\lambda}(\mathbf{z}) = \sum_{k=1}^{K} w_k q_{\phi}\left(\mathbf{z}|g_{\gamma}(\mathbf{u}_k)\right), \tag{4.53}$$

where we first define a grid in a low-dimensional space, $\{\mathbf{u}_k\}$, and then transform them to $\mathcal{X}^D$ using the transformation g_{γ}.

Now, we train a small VAE with the GTM-VampPrior (with $K = 16$, i.e., a 4×4 grid) and a two-dimensional latent space. In Fig. 4.12, we present samples from the encoder for the test data (black dots) and the contour plot for the GTM-VampPrior.

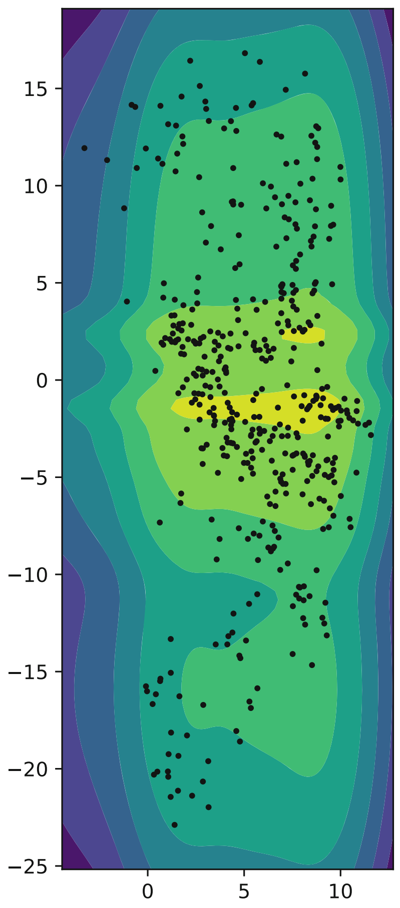

Fig. 4.12 An example of the GTM-VampPrior (contours) and the samples from the aggregated posterior (black dots)

Again, this mixture-based prior allows to wrap the points (the aggregated posterior) and assign the probability to proper regions.

An example of an implementation of the GTM-VampPrior is presented below:

```
class GTMVampPrior(nn.Module):
    def __init__(self, L, D, gtm_net, encoder, num_points, u_min
    =-10., u_max=10., num_vals=255):
    super(GTMVampPrior, self).__init__()

    self.L = L
    self.D = D
    self.num_vals = num_vals

    self.encoder = encoder

    # 2D manifold
    self.u = torch.zeros(num_points**2, 2) # K**2 x 2
    u1 = torch.linspace(u_min, u_max, steps=num_points)
    u2 = torch.linspace(u_min, u_max, steps=num_points)

    k = 0
    for i in range(num_points):
        for j in range(num_points):
            self.u[k,0] = u1[i]
            self.u[k,1] = u2[j]
            k = k + 1

    # gtm network: u -> x
    self.gtm_net = gtm_net

    # mixing weights
    self.w = nn.Parameter(torch.zeros(num_points**2, 1, 1))

    def get_params(self):
        # u->gtm_net->u_x
        h_gtm = self.gtm_net(self.u) #K x D
        h_gtm = h_gtm * self.num_vals
        # u_x->encoder->mu, lof_var
        mean_vampprior, logvar_vampprior = self.encoder.encode(h_gtm) #(K x L), (K x L)
        return mean_vampprior, logvar_vampprior

    def sample(self, batch_size):
    # u->encoder->mu, lof_var
    mean_vampprior, logvar_vampprior = self.get_params()

    # mixing probabilities
    w = F.softmax(self.w, dim=0)
    w = w.squeeze()

        # pick components
        indexes = torch.multinomial(w, batch_size, replacement=True)

        # means and logvars
        eps = torch.randn(batch_size, self.L)
        for i in range(batch_size):
            indx = indexes[i]
            if i == 0:
                z = mean_vampprior[[indx]] + eps[[i]] * torch.exp(logvar_vampprior[[indx]])
            else:
                z = torch.cat((z, mean_vampprior[[indx]] + eps[[i]] * torch.exp(logvar_vampprior[[indx]])), 0)
        return z

    def log_prob(self, z):
        # u->encoder->mu, lof_var
        mean_vampprior, logvar_vampprior = self.get_params()

        # mixing probabilities
        w = F.softmax(self.w, dim=0)

        # log-mixture-of-Gaussians
        z = z.unsqueeze(0) # 1 x B x L
        mean_vampprior = mean_vampprior.unsqueeze(1) # K x 1 x L
        logvar_vampprior = logvar_vampprior.unsqueeze(1) # K x 1
    x L

        log_p = log_normal_diag(z, mean_vampprior,
    logvar_vampprior) + torch.log(w) # K x B x L
        log_prob = torch.logsumexp(log_p, dim=0, keepdim=False) #
     B x L

        return log_prob
```

Listing 4.10 A GTM-VampPrior prior class

4.4.1.6 Flow-Based Prior

The last distribution we want to discuss here is a flow-based prior. Since flow-based models can be used to estimate any distribution, it is almost obvious to use them for approximating the aggregated posterior. Here, we use the implementation of the RealNVP presented before (see Chap. 3 for details).

As in the previous cases, we train a small VAE with the flow-based prior and two-dimensional latent space. In Fig. 4.13, we present samples from the encoder for the test data (black dots) and the contour plot for the flow-based prior. Similar to the previous mixture-based priors, the flow-based prior allows approximating the aggregated posterior very well. This is in line with many papers using flows as the prior in the VAE [24, 25]; however, we must remember that the flexibility of the flow-based prior comes with the cost of an increased number of parameters and potential training issues inherited from the flows.

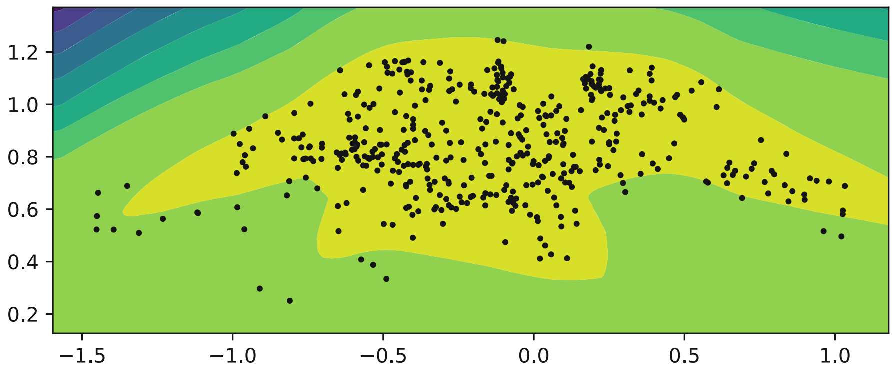

Fig. 4.13 An example of the flow-based prior (contours) and the samples from the aggregated posterior (black dots)

An example of an implementation of the flow-based prior is presented below:

```python
class FlowPrior(nn.Module):
    def __init__(self, nets, nett, num_flows, D=2):
        super(FlowPrior, self).__init__()

        self.D = D

        self.t = torch.nn.ModuleList([nett() for _ in range(num_flows)])
        self.s = torch.nn.ModuleList([nets() for _ in range(num_flows)])
        self.num_flows = num_flows

    def coupling(self, x, index, forward=True):
        (xa, xb) = torch.chunk(x, 2, 1)

        s = self.s[index](xa)
        t = self.t[index](xa)

        if forward:
            #yb = f^{-1}(x)
            yb = (xb - t) * torch.exp(-s)
        else:
            #xb = f(y)
            yb = torch.exp(s) * xb + t

        return torch.cat((xa, yb), 1), s

    def permute(self, x):
        return x.flip(1)

    def f(self, x):
        log_det_J, z = x.new_zeros(x.shape[0]), x
        for i in range(self.num_flows):
            z, s = self.coupling(z, i, forward=True)
            z = self.permute(z)
            log_det_J = log_det_J - s.sum(dim=1)

        return z, log_det_J

    def f_inv(self, z):
        x = z
        for i in reversed(range(self.num_flows)):
            x = self.permute(x)
            x, _ = self.coupling(x, i, forward=False)

        return x

    def sample(self, batch_size):
        z = torch.randn(batch_size, self.D)
        x = self.f_inv(z)
        return x.view(-1, self.D)

    def log_prob(self, x):
        z, log_det_J = self.f(x)
        log_p = (log_standard_normal(z) + log_det_J.unsqueeze(1))
        return log_p
```

Listing 4.11 A flow-based prior class

4.4.1.7 Remarks

In practice, we can use any density estimator to model $p_\lambda(\mathbf{z})$. For instance, we can use an autoregressive model [26] or more advanced approaches like resampled priors [27] or hierarchical priors [51]. Therefore, there are many options! However, there is still an open question **how** to do that and **what** role the prior (the marginal) should play. As I mentioned in the beginning, Bayesianists would say that the marginal should impose some constraints on the latent space or, in other words, our prior knowledge about it. I am a Bayesiast deep down in my heart and this way of thinking is very appealing to me. However, it is still unclear what is a good latent representation. This question is as old as mathematical modeling. I think that it would be interesting to look at optimization techniques, maybe applying a gradient-based method to all parameters/weights at once is not the best solution. Anyhow, I am pretty sure that modeling the prior is more important than many people think and plays a crucial role in VAEs.

4.4.2 Variational Posteriors

In general, variational inference searches for the best posterior approximation within a parametric family of distributions. Hence, recovering the true posterior is possible only if it happens to be in the chosen family. In particular, with widely used variational families such as diagonal covariance Gaussian distributions, the variational approximation is likely to be insufficient. Therefore, designing tractable and more expressive variational families is an important problem in VAEs. Here, we present two families of conditional normalizing flows that can be used for that purpose, namely, Householder flows [20] and Sylvester flows [16]. There are other interesting families and we refer the reader to the original papers, e.g., the generalized Sylvester flows [17] and the Inverse Autoregressive Flows [18].

The general idea about using the normalizing flows to parameterize the variational posteriors is to start with a relatively simple distribution like the Gaussian with the diagonal covariance matrix and then transform it to a complex distribution through a series of invertible transformations [19]. Formally speaking, we start with the latents $\mathbf{z}^{(0)}$ distributed according to $\mathcal{N}(\mathbf{z}^{(0)}|\mu(\mathbf{x}, \sigma^2(\mathbf{x})$ and then after applying a series of invertible transformations $\mathbf{f}^{(t)}$, for $t = 1, \ldots, T$, the last iterate gives a random variable $\mathbf{z}^{(T)}$ that has a more flexible distribution. Once we choose transformations $\mathbf{f}^{(t)}$ for which the Jacobian-determinant can be computed, we aim at optimizing the following objective:

$$\ln p(\mathbf{x}) \geq \mathbb{E}_{q(\mathbf{z}^{(0)}|\mathbf{x})}\Big[\ln p(\mathbf{x}|\mathbf{z}^{(T)}) + \sum_{t=1}^{T} \ln \Big|\det \frac{\partial \mathbf{f}^{(t)}}{\partial \mathbf{z}^{(t-1)}}\Big|\Big] - \mathrm{KL}\big(q(\mathbf{z}^{(0)}|\mathbf{x})||p(\mathbf{z}^{(T)})\big). \tag{4.54}$$

In fact, the normalizing flow can be used to enrich the posterior of the VAE with small or even none modifications in the architecture of the encoder and the decoder.

4.4.2.1 Variational Posteriors with Householder Flows [20]

Motivation

First, we notice that any full-covariance matrix $\mathbf{\Sigma}$ can be represented by the eigenvalue decomposition using eigenvectors and eigenvalues:

$$\mathbf{\Sigma} = \mathbf{U}\mathbf{D}\mathbf{U}^{\top}, \tag{4.55}$$

where $\mathbf{U}$ is an orthogonal matrix with eigenvectors in columns and $\mathbf{D}$ is a diagonal matrix with eigenvalues. In the case of the vanilla VAE, it would be tempting to model the matrix $\mathbf{U}$ to obtain a full-covariance matrix. The procedure would require a linear transformation of a random variable using an orthogonal matrix $\mathbf{U}$. Since the absolute value of the Jacobian-determinant of an orthogonal matrix is 1, for $\mathbf{z}^{(1)} = \mathbf{U}\mathbf{z}^{(0)}$ one gets $\mathbf{z}^{(1)} \sim \mathcal{N}(\mathbf{U}\boldsymbol{\mu}, \mathbf{U}\ \mathrm{diag}(\boldsymbol{\sigma}^2)\ \mathbf{U}^{\top})$. If $\mathrm{diag}(\boldsymbol{\sigma}^2)$ coincides with true $\mathbf{D}$, then it would be possible to resemble the true full-covariance matrix. Hence, the main goal would be to model the orthogonal matrix of eigenvectors.

Generally, the task of modeling an orthogonal matrix in a principled manner is rather non-trivial. However, first we notice that any orthogonal matrix can be represented in the following form [53, 54]:

Theorem 4.1 (The Basis-Kernel Representation of Orthogonal Matrices) *For any $M \times M$ orthogonal matrix $\mathbf{U}$, there exist a full-rank $M \times K$ matrix $\mathbf{Y}$ (the basis) and a nonsingular (triangular) $K \times K$ matrix $\mathbf{S}$ (the kernel), $K \leq M$, such that:*

$$\mathbf{U} = \mathbf{I} - \mathbf{Y}\mathbf{S}\mathbf{Y}^{\top}. \tag{4.56}$$

The value K is called the *degree* of the orthogonal matrix. Further, it can be shown that any orthogonal matrix of degree K can be expressed using the product of Householder transformations [53, 54], namely:

Theorem 4.2 *Any orthogonal matrix with the basis acting on the K-dimensional subspace can be expressed as a product of exactly K Householder matrices:*

$$\mathbf{U} = \mathbf{H}_K \mathbf{H}_{K-1} \cdots \mathbf{H}_1, \tag{4.57}$$

where $\mathbf{H}_k = \mathbf{I} - \mathbf{S}_{kk}\mathbf{Y}_{\cdot k}(\mathbf{Y}_{\cdot k})^{\top}$, for $k = 1, \ldots, K$.

Theoretically, Theorem 4.2 shows that we can model any orthogonal matrix in a principled fashion using K Householder transformations. Moreover, the Householder matrix $\mathbf{H}_k$ is *orthogonal* matrix itself [55]. Therefore, this property and the Theorem 4.2 put the Householder transformation as a perfect candidate for formulating a volume-preserving flow that allows to approximate (or even capture) the true full-covariance matrix.

Householder Flows

The *Householder transformation* is defined as follows. For a given vector $\mathbf{z}^{(t-1)}$, the reflection hyperplane can be defined by a vector (a *Householder vector*) $\mathbf{v}_t \in \mathbb{R}^M$ that is orthogonal to the hyperplane, and the reflection of this point about the hyperplane is [55]

$$\mathbf{z}^{(t)} = \left(\mathbf{I} - 2\frac{\mathbf{v}_t\mathbf{v}_t^\top}{||\mathbf{v}_t||^2}\right)\mathbf{z}^{(t-1)} \tag{4.58}$$

$$= \mathbf{H}_t\mathbf{z}^{(t-1)}, \tag{4.59}$$

where $\mathbf{H}_t = \mathbf{I} - 2\frac{\mathbf{v}_t\mathbf{v}_t^\top}{||\mathbf{v}_t||^2}$ is called the *Householder matrix*.

The most important property of $\mathbf{H}_t$ is that it is an orthogonal matrix and hence the absolute value of the Jacobian-determinant is equal to 1. This fact significantly simplifies the objective (4.54) because $\ln\left|\det\frac{\partial \mathbf{H}_t\mathbf{z}^{(t-1)}}{\partial \mathbf{z}^{(t-1)}}\right| = 0$, for $t = 1, \ldots, T$. Starting from a simple posterior with the diagonal covariance matrix for $\mathbf{z}^{(0)}$, the series of T linear transformations given by (4.58) defines a new type of volume-preserving flow that we refer to as the *Householder flow* (HF). The vectors $\mathbf{v}_t$, $t = 1, \ldots, T$, are produced by the encoder network along with means and variances using a linear layer with the input $\mathbf{v}_{t-1}$, where $\mathbf{v}_0 = \mathbf{h}$ is the last hidden layer of the encoder network. The idea of the Householder flow is schematically presented in Fig. 4.14. Once the encoder returns the first Householder vector, the Householder

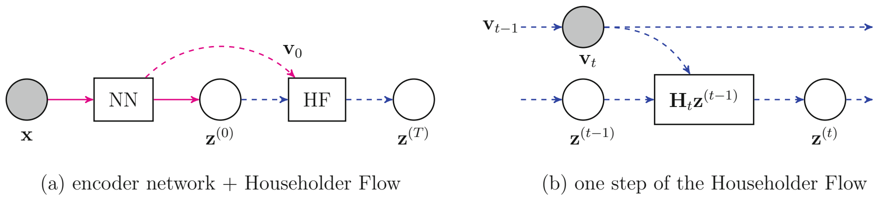

Fig. 4.14 A schematic representation of the encoder network with the Householder flow. (**a**) The general architecture of the VAE+HF: The encoder returns means and variances for the posterior and the first Householder vector that is further used to formulate the Householder flow. (**b**) A single step of the Householder flow that uses linear Householder transformation. In both panels solid lines correspond to the encoder network and the dashed lines are additional quantities required by the HF

flow requires T linear operations to produce a sample from a more flexible posterior with an approximate full-covariance matrix.

4.4.2.2 Variational Posteriors with Sylvester Flows [16]

Motivation

The Householder flows can model only full-covariance Gaussians that is still not necessarily a rich family of distributions. Now, we will look into a generalization of the Householder flows. For this purpose, let us consider the following transformation similar to a single layer MLP with M hidden units and a residual connection:

$$\mathbf{z}^{(t)} = \mathbf{z}^{(t-1)} + \mathbf{A}h(\mathbf{B}\mathbf{z}^{(t-1)} + \mathbf{b}), \tag{4.60}$$

with $\mathbf{A} \in \mathbb{R}^{D\times M}$, $\mathbf{B} \in \mathbb{R}^{M\times D}$, $\mathbf{b} \in \mathbb{R}^{M}$, and $M \leq D$. The Jacobian-determinant of this transformation can be obtained using *Sylvester's determinant identity*, which is a generalization of the matrix determinant lemma.

Theorem 4.3 (Sylvester's Determinant Identity) *For all* $\mathbf{A} \in \mathbb{R}^{D\times M}, \mathbf{B} \in \mathbb{R}^{M\times D}$,

$$\det\left(\mathbf{I}_D + \mathbf{AB}\right) = \det\left(\mathbf{I}_M + \mathbf{BA}\right), \tag{4.61}$$

where $\mathbf{I}_M$ *and* $\mathbf{I}_D$ *are* M*- and* D*-dimensional identity matrices, respectively.*

When $M < D$, the computation of the determinant of a $D \times D$ matrix is thus reduced to the computation of the determinant of an $M \times M$ matrix.

Using Sylvester's determinant identity, the Jacobian-determinant of the transformation in Eq. (4.60) is given by:

$$\det\left(\frac{\partial \mathbf{z}^{(t)}}{\partial \mathbf{z}^{(t-1)}}\right) = \det\Big(\mathbf{I}_M + \operatorname{diag}\Big(h'(\mathbf{B}\mathbf{z}^{(t-1)} + \mathbf{b})\Big)\mathbf{BA}\Big). \tag{4.62}$$

Since Sylvester's determinant identity plays a crucial role in the proposed family of normalizing flows, we will refer to them as *Sylvester normalizing flows*.

Parameterization of **A** and **B**

In general, the transformation in (4.60) will not be invertible. Therefore, we propose the following special case of the above transformation:

$$\mathbf{z}^{(t)} = \mathbf{z}^{(t-1)} + \mathbf{QR}h(\tilde{\mathbf{R}}\mathbf{Q}^T\mathbf{z}^{(t-1)} + \mathbf{b}), \tag{4.63}$$

where $\mathbf{R}$ and $\tilde{\mathbf{R}}$ are upper triangular $M \times M$ matrices, and

$$\mathbf{Q} = (\mathbf{q}_1 \dots \mathbf{q}_M)$$

with the columns $\mathbf{q}_m \in \mathbb{R}^D$ forming an orthonormal set of vectors. By Theorem 4.3, the determinant of the Jacobian $\mathbf{J}$ of this transformation reduces to:

$$\begin{aligned}\det(\mathbf{J}) &= \det\left(\mathbf{I}_M + \operatorname{diag}\left(h'(\tilde{\mathbf{R}}\mathbf{Q}^T\mathbf{z}^{(t-1)} + \mathbf{b})\right)\tilde{\mathbf{R}}\mathbf{Q}^T\mathbf{Q}\mathbf{R}\right)\\ &= \det\left(\mathbf{I}_M + \operatorname{diag}\left(h'(\tilde{\mathbf{R}}\mathbf{Q}^T\mathbf{z}^{(t-1)} + \mathbf{b})\right)\tilde{\mathbf{R}}\mathbf{R}\right), \qquad (4.64)\end{aligned}$$

which can be computed in $O(M)$, since $\tilde{\mathbf{R}}\mathbf{R}$ is also upper triangular. The following theorem gives a sufficient condition for this transformation to be invertible.

Theorem 4.4 *Let* $\mathbf{R}$ *and* $\tilde{\mathbf{R}}$ *be upper triangular matrices. Let* $h : \mathbb{R} \longrightarrow \mathbb{R}$ *be a smooth function with bounded, positive derivative. Then, if the diagonal entries of* $\mathbf{R}$ *and* $\tilde{\mathbf{R}}$ *satisfy* $r_{ii}\tilde{r}_{ii} > -1/\|h'\|_\infty$ *and* $\tilde{\mathbf{R}}$ *is invertible, the transformation given by* (4.63) *is invertible.*

The proof of this theorem could be found in [16].

Preserving Orthogonality of $\mathbf{Q}$

Orthogonality is a convenient property, mathematically, but hard to achieve in practice. In this chapter we consider three different flows based on the theorem above and various ways to preserve the orthogonality of $\mathbf{Q}$. The first two use explicit differentiable constructions of orthogonal matrices, while the third variant assumes a specific fixed permutation matrix as the orthogonal matrix.

Orthogonal Sylvester Flows First, we consider a Sylvester flow using matrices with M orthogonal columns (O-SNF). In this flow we can choose $M < D$ and thus introduce a flexible bottleneck. Similar to [56], we ensure orthogonality of $\mathbf{Q}$ by applying the following differentiable iterative procedure proposed by Björck and Bowie [57] and Kovarik [58]:

$$\mathbf{Q}^{(k+1)} = \mathbf{Q}^{(k)}\left(\mathbf{I} + \frac{1}{2}\left(\mathbf{I} - \mathbf{Q}^{(k)\top}\mathbf{Q}^{(k)}\right)\right), \qquad (4.65)$$

with a sufficient condition for convergence given by $\|\mathbf{Q}^{(0)\top}\mathbf{Q}^{(0)} - \mathbf{I}\|_2 < 1$. Here, the 2-norm of a matrix $\mathbf{X}$ refers to $\|\mathbf{X}\|_2 = \lambda_{\max}(\mathbf{X})$, with $\lambda_{\max}(\mathbf{X})$ representing the largest singular value of $\mathbf{X}$. In our experimental evaluations we ran the iterative procedure until $\|\mathbf{Q}^{(k)\top}\mathbf{Q}^{(k)} - \mathbf{I}\|_F \le \epsilon$, with $\|\mathbf{X}\|_F$ the Frobenius norm, and ϵ a small convergence threshold. We observed that running this procedure up to 30 steps was sufficient to ensure convergence with respect to this threshold. To minimize the computational overhead introduced by orthogonalization, we perform this orthogonalization in parallel for all flows.

Since this orthogonalization procedure is differentiable, it allows for the calculation of gradients with respect to $\mathbf{Q}^{(0)}$ by backpropagation, allowing for any standard optimization scheme such as stochastic gradient descent to be used for updating the flow parameters.

Householder Sylvester Flows Second, we study Householder Sylvester flows (H-SNF) where the orthogonal matrices are constructed by products of Householder reflections. Householder transformations are reflections about hyperplanes. Let $\mathbf{v} \in \mathbb{R}^D$, then the reflection about the hyperplane orthogonal to $\mathbf{v}$ is given by Eq. (4.58).

It is worth noting that performing a single Householder transformation is very cheap to compute, as it only requires D parameters. Chaining together several Householder transformations results in more general orthogonal matrices, and Theorem 4.2 shows that any $M \times M$ orthogonal matrix can be written as the product of $M - 1$ Householder transformations. In our Householder Sylvester flow, the number of Householder transformations H is a hyperparameter that trades off the number of parameters and the generality of the orthogonal transformation. Note that the use of Householder transformations forces us to use $M = D$, since Householder transformations result in square matrices.

Triangular Sylvester Flows Third, we consider a triangular Sylvester flow (T-SNF), in which all orthogonal matrices $\mathbf{Q}$ alternate per transformation between the identity matrix and the permutation matrix corresponding to reversing the order of $\mathbf{z}$. This is equivalent to alternating between lower and upper triangular $\tilde{\mathbf{R}}$ and $\mathbf{R}$ for each flow.

Amortizing Flow Parameters

When using normalizing flows in an amortized inference setting, the parameters of the base distribution as well as the flow parameters can be functions of the datapoint $\mathbf{x}$ [19]. Figure 4.15 (left) shows a diagram of one SNF step and the amortization procedure. The inference network takes datapoints $\mathbf{x}$ as input and provides as an output the mean and variance of $\mathbf{z}^{(0)}$ such that $\mathbf{z}^{(0)} \sim \mathcal{N}(\mathbf{z}|\mu^0, \sigma^0)$. Several SNF transformations are then applied to $\mathbf{z}^{(0)} \rightarrow \mathbf{z}^{(1)} \rightarrow \dots \mathbf{z}^{(T)}$, producing a flexible posterior distribution for $\mathbf{z}^{(T)}$. All of the flow parameters ($\mathbf{R}$, $\tilde{\mathbf{R}}$, and $\mathbf{Q}$ for each transformation) are produced as an output by the inference network and are thus fully amortized.

4.4.2.3 Hyperspherical Latent Space

Motivation

In the VAE framework, choosing Gaussian priors and Gaussian posteriors from the mathematical convenience leads to Euclidean latent space. However, such choice could be limiting for the following reasons:

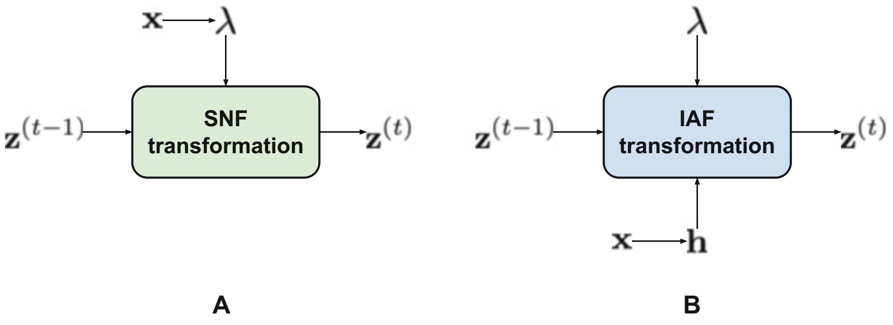

Fig. 4.15 Different amortization strategies for Sylvester normalizing flows and Inverse Autoregressive Flows. (**a**) Our inference network produces amortized flow parameters. This strategy is also employed by planar flows. (**b**) Inverse Autoregressive Flow [18] introduces a measure of $\mathbf{x}$ dependence through a context variable $\mathbf{h}(\mathbf{x})$. This context acts as an additional input for each transformation. The flow parameters themselves are independent of $\mathbf{x}$

- In low dimensions, the standard Gaussian probability presents a concentrated probability mass around the mean, encouraging points to cluster in the center. However, this is particularly problematic when the data is divided into multiple clusters. Then, a better suited prior would be *uniform*. Such a uniform prior, however, is not well-defined on the hyperplane.
- It is a well-known phenomenon that the standard Gaussian distribution in high dimensions tends to resemble a uniform distribution on the surface of a hypersphere, with the vast majority of its mass concentrated on the hyperspherical shell (the so-called soap bubble effect). A natural question is whether it would be better to use a distribution defined on the hypersphere.

A distribution that would allow solving both problems at once is the von Mises–Fisher distribution. It was advocated in [33] to use this distribution in the context of VAEs.

von Mises–Fisher Distribution

The *von Mises–Fisher* (vMF) distribution is often described as the Normal Gaussian distribution on a hypersphere. Analogous to a Gaussian, it is parameterized by $\mu \in \mathbb{R}^m$ indicating the mean direction, and $\kappa \in \mathbb{R}_{\geq 0}$ the concentration around μ. For the special case of $\kappa = 0$, the vMF represents a Uniform distribution. The probability density function of the vMF distribution for a random unit vector $\mathbf{z} \in \mathbb{R}^m$ (or $\mathbf{z} \in \mathcal{S}^{m-1}$) is then defined as

$$q(\mathbf{z}|\boldsymbol{\mu}, \kappa) = C_m(\kappa) \exp\left(\kappa \boldsymbol{\mu}^T \mathbf{z}\right) \tag{4.66}$$

$$C_m(\kappa) = \frac{\kappa^{m/2-1}}{(2\pi)^{m/2} \mathcal{I}_{m/2-1}(\kappa)}, \tag{4.67}$$

where $||\boldsymbol{\mu}||^2 = 1$, $C_m(\kappa)$ is the normalizing constant, and $\mathcal{I}_v$ denotes the modified Bessel function of the first kind at order v.

Interestingly, since we define a distribution over a hypersphere, it is possible to formulate a uniform prior over the hypersphere. Then it turns out that if we take the vMF distribution as the variational posterior, it is possible to calculate the Kullback–Leibler divergence between the vMF distribution and the uniform defined over $\mathcal{S}^{m-1}$ analytically [33]:

$$KL[\text{vMF}(\mu, \kappa)||\text{Unif}(\mathcal{S}^{m-1})] = \kappa + \log C_m(\kappa) - \log \left(\frac{2(\pi^{m/2})}{\Gamma(m/2)}\right)^{-1}. \tag{4.68}$$

To sample from the vMF, one can follow the procedure of [59]. Importantly, the reparameterization cannot be easily formulated for the vMF distribution. Fortunately, [60] allows extending the reparameterization trick to the wide class of distributions that can be simulated using rejection sampling. [33] presents how to formulate the acceptance–rejection sampling reparameterization procedure. Being equipped with the sampling procedure and the reparameterization trick, and having an analytical form of the Kullback–Leibler divergence, we have everything to be able to build a hyperspherical VAE. However, please note the all these procedures are less trivial than the ones for Gaussians. Therefore, a curious reader is referred to [33] for further details.

4.5 Hierarchical Latent Variable Models

4.5.1 Introduction

The main goal of AI is to formulate and implement systems that can interact with an environment, process, store, and transmit information. In other words, we wish an AI system *understands* the world around it by identifying and disentangling hidden factors in the observed low-sensory data [61]. If we think about the problem of building such a system, we can formulate it as learning a probabilistic model, i.e., a joint distribution over observed data, $\mathbf{x}$, and hidden factors, $\mathbf{z}$, namely, $p(\mathbf{x}, \mathbf{z})$. Then learning a *useful representation* is equivalent to finding a posterior distribution over the hidden factors, $p(\mathbf{z}|\mathbf{x})$. However, it is rather unclear what we really mean by *useful* in this context. In a beautiful blog post [62], Ferenc Huszar outlines why learning a latent variable model by maximizing the likelihood function is not necessarily useful from the representation learning perspective. Here, we will use it as a good starting point for a discussion of why applying hierarchical latent variable models could be beneficial.

Let us start by defining the setup. We assume the empirical distribution $p_{data}(\mathbf{x})$ and a latent variable model $p_\theta(\mathbf{x}, \mathbf{z})$. The way we parameterize the latent variable model is not constrained in any manner; however, we assume that the distribution is parameterized using deep neural networks (DNNs). This is important for two reasons:

1. DNNs are non-linear transformations and as such, they are flexible and allow parameterizing a wide range of distributions.
2. We must remember that DNNs **will not** solve all problems for us! In the end, we need to think about the model as a whole, not only about the parameterization. What I mean by that is the distribution we choose and how random variables interact, etc. DNNs are definitely helpful, but there are many potential pitfalls (we will discuss some of them later on) that even the largest and coolest DNN is unable to take care of.

It is worth to remember that the joint distribution could be factorized in two ways, namely:

$$p_\theta(\mathbf{x}, \mathbf{z}) = p_\theta(\mathbf{x}|\mathbf{z}) p_\theta(\mathbf{z}) \tag{4.69}$$

$$= p_\theta(\mathbf{z}|\mathbf{x}) p_\theta(\mathbf{x}). \tag{4.70}$$

Moreover, the training problem of learning θ could be defined as an unconstrained optimization problem with the following training objective:

$$KL[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})] = -\mathbb{H}[p_{data}(\mathbf{x})] + \mathbb{CE}[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})] \tag{4.71}$$

$$= const + \mathbb{CE}[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})], \tag{4.72}$$

where $p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}, \mathbf{z})\, \mathrm{d}\mathbf{z}$, $\mathbb{H}[\cdot]$ denotes the entropy, and $\mathbb{CE}[\cdot||\cdot]$ is the cross-entropy. Notice that the entropy of the empirical distribution is simply a constant since it does not contain θ. The cross-entropy could be further re-written as follows:

$$\mathbb{CE}[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})] = -\int p_{data}(\mathbf{x}) \ln p_\theta(\mathbf{x})\, \mathrm{d}\mathbf{x} \tag{4.73}$$

$$= -\frac{1}{N} \sum_{n=1}^{N} \ln p_\theta(\mathbf{x}_n). \tag{4.74}$$

Eventually, we have obtained the objective function we use all the time, namely, the negative log-likelihood function.

If we think of *usefulness* of a representation (i.e., hidden factors) $\mathbf{z}$, we intuitively think of some kind of information that is shared between $\mathbf{z}$ and $\mathbf{x}$. However, the unconstrained training problem we consider, i.e., the minimization of the negative log-likelihood function, does not necessarily say **anything** about the latent

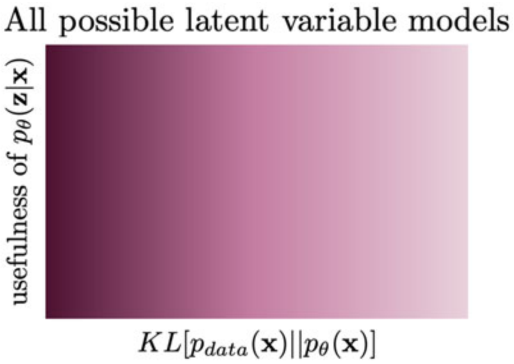
All possible latent variable models
usefulness of $p_\theta(\mathbf{z}|\mathbf{x})$
$KL[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})]$

Fig. 4.16 A schematic diagram representing a dependency between *usefulness* and the objective function for all possible latent variable models. The darker the color, the better the objective function value. Reproduced based on [62]

representation. In the end, we optimize the **marginal** over observable variables because we do not have access to values of latent variables. Even more, typically we do not know what these hidden factors are or should be! As a result, our latent variable model can learn to disregard the latent variables completely. Let us look into this problem in more detail.

A Potential Problem with Latent Variable Models

Following the discussion presented in [62], we can visualize two scenarios that are pretty common in deep generative modeling with latent variable models. Before delving into that, it is beneficial to explain the general picture. We are interested in analyzing a class of latent variable models with respect to *usefulness* of latents and the value of the objective function $KL[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})]$. In Fig. 4.16, we depict a case when all models are possible, namely, a search space where models are evaluated according to the training objective (x-axis) and *usefulness* (y-axis). The ideal model is the one in the top-left corner that maximizes both criteria. However, it is possible to find a model that completely disregards the latents (the bottom-left corner) while maximizing the fit to data. We already can see that there is a potentially huge problem! Running a (numerical) optimization procedure could give infinitely many models that are equally good with respect to $KL[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})]$ but with completely different posteriors over latents! That puts in question the applicability of the latent variable models. However, in practice, we see that learned latent variables are useful (or, in other words, they contain information about observables). So how is it possible?

As pointed out by Huszár [62], the reason for that is the inductive bias of the chosen class of models. By picking a very specific class of DNNs, we implicitly constrain the search space. First, the left-most models in Fig. 4.16 are typically unattainable. However, using some kind of bottlenecks in our class of models potentially leads to a situation that latents must contain some information about observables. As a result, they become *useful*. An example of such a situation is depicted in Fig. 4.17. After running a training algorithm, we can end in one of the two "spikes" where the training objective is the highest and the *usefulness* is non-

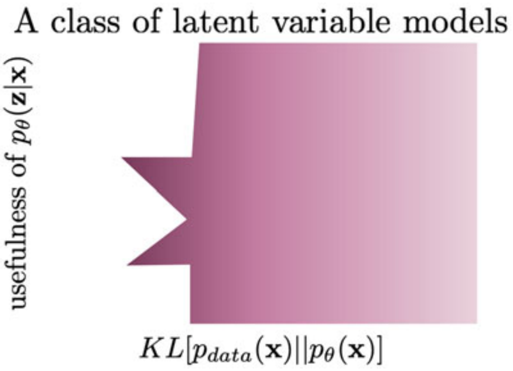
A class of latent variable models
usefulness of $p_\theta(\mathbf{z}|\mathbf{x})$
$KL[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})]$

Fig. 4.17 A schematic diagram representing a dependency between *usefulness* and the objective function for a constrained class of models. The darker the color, the better the objective function value. Reproduced based on [62]

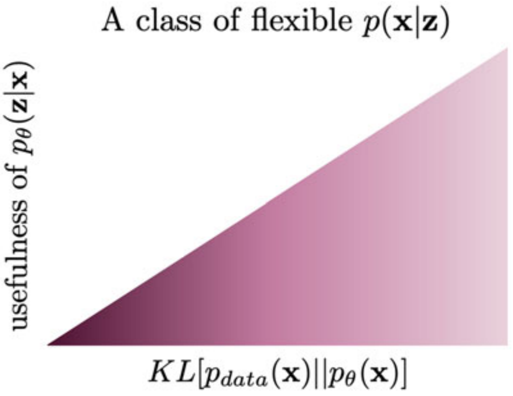
A class of flexible $p(\mathbf{x}|\mathbf{z})$
usefulness of $p_\theta(\mathbf{z}|\mathbf{x})$
$KL[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})]$

Fig. 4.18 A schematic diagram representing a dependency between *usefulness* and the objective function for a class of models with flexible $p(\mathbf{x}|\mathbf{z})$. The darker the color, the better the objective function value. Reproduced based on [62]

zero. Still, we can achieve the same performing models at two different levels of the *usefulness* but at least the information flows from $\mathbf{x}$ to $\mathbf{z}$. Obviously, the considered scenario is purely hypothetical, but it shows that the inductive bias of a model can greatly help to learn representations without being specified by the objective function. Please keep this thought in mind because it will play a crucial role later on!

The next situation is more tricky. Let us assume that we have a constrained class of models; however, the conditional likelihood $p(\mathbf{x}|\mathbf{z})$ is parameterized by a flexible, enormous DNN. A potential danger here is that this model could learn to completely disregard $\mathbf{z}$, treating it as a noise. As a result, $p(\mathbf{x}|\mathbf{z})$ becomes an unconditional distribution that mimics $p_{data}(\mathbf{x})$ almost perfectly. At the first glance, this scenario sounds unrealistic, but it is a well-known phenomenon in the field. For instance, [10] conducted a thorough experiment with variational auto-encoders, and taking a PixelCNN++-based decoder resulted in a VAE that was unable to reconstruct images. Their conclusion was exactly the same, namely, taking a class of models with too flexible $p(\mathbf{x}|\mathbf{z})$ could lead to the model in the bottom-left corner in Fig. 4.18.

How to Define a *Proper* Class of Models?

Alright, you are probably a bit confused about what we have discussed so far. The general picture is rather pessimistic because it seems that picking a proper class of models, i.e., a class of models that allow achieving *useful* latent representations is a non-trivial task. Moreover, the whole story sounds like walking in the dark, trying out various DNN architectures, and hoping that we obtain a meaningful representation.

Fortunately, the problem is not so horrible as it looks at the first glance. Some ideas formulate a constrained optimization problem [12, 63] or add an auxiliary regularizer [64, 65] to (implicitly) define *usefulness* of the latents. Here, we will discuss one of the possible approaches that utilizes hierarchical architectures. However, it is worth remembering that the issue of learning *useful* representations remains an open question and is a vivid research direction.

Hierarchical models have a long history in deep generative modeling and deep learning and were advocated by many prominent researchers, e.g., [66–68]. The main hypothesis is that the concepts describing the world around us could be organized hierarchically. In the light of our discussion, if a latent variable model takes a hierarchical structure, it may introduce an inductive bias, constrain the class of models, and, eventually, force information flow between latents and observables. At least in theory. Shortly, we will see that we must be very careful with formulating stochastic dependencies in the hierarchy. In the next sections, we will focus on latent variable models with variational inference, i.e., hierarchical Variational Auto-Encoders.

A side note: One may be tempted to associate hierarchical modeling with Bayesian hierarchical modeling. These two terms are not necessarily equivalent. Bayesian hierarchical modeling is about treating (hyper)parameters *as random variables and formulating distributions over (hyper)parameters [69]. Here, we do not take advantage of Bayesian modeling and consider a hierarchy among latent variables, not parameters.*

4.5.2 Hierarchical VAEs

4.5.2.1 Two-Level VAEs

Let us start with a VAE with two latent variables: $\mathbf{z}_1$ and $\mathbf{z}_2$. The joint distribution could be factorized as follows:

$$p(\mathbf{x}, \mathbf{z}_1, \mathbf{z}_2) = p(\mathbf{x}|\mathbf{z}_1)p(\mathbf{z}_1|\mathbf{z}_2)p(\mathbf{z}_2). \tag{4.75}$$

This model defines a straightforward generative process: First sample $\mathbf{z}_2$, then sample $\mathbf{z}_1$ given $\mathbf{z}_2$, and eventually sample $\mathbf{x}$ given $\mathbf{z}_1$.

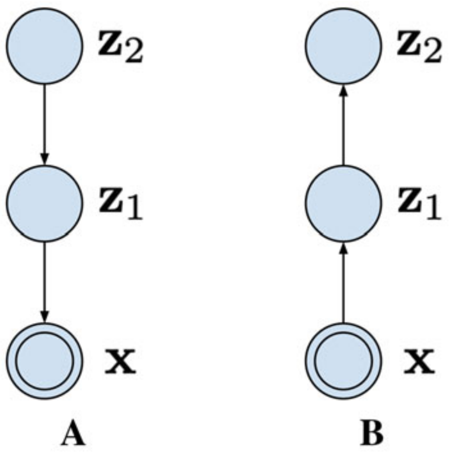

Fig. 4.19 An example of a two-level VAE. (**a**) The generative part. (**b**) The variational part

Since we know already that even for a single latent variable calculating posteriors over latents is intractable (except the linear Gaussian case, it is worth remembering that!), we can utilize the variational inference with a family of variational posteriors $Q(\mathbf{z}_1, \mathbf{z}_2|\mathbf{x})$. Now, the main part is how to define the variational posteriors. A rather natural approach would be to reverse the dependencies and factorize the posterior in the following fashion:

$$Q(\mathbf{z}_1, \mathbf{z}_2|\mathbf{x}) = q(\mathbf{z}_1|\mathbf{x})q(\mathbf{z}_2|\mathbf{z}_1, \mathbf{x}), \tag{4.76}$$

or even we can simplify it as follows (dropping the dependency on $\mathbf{x}$ for the second latent variable):

$$Q(\mathbf{z}_1, \mathbf{z}_2|\mathbf{x}) = q(\mathbf{z}_1|\mathbf{x})q(\mathbf{z}_2|\mathbf{z}_1). \tag{4.77}$$

If we take the continuous latents, we can use the Gaussian distributions:

$$p(\mathbf{z}_1|\mathbf{z}_2) = \mathcal{N}(\mathbf{z}_1|\mu(\mathbf{z}_2), \sigma^2(\mathbf{z}_2)) \tag{4.78}$$

$$p(\mathbf{z}_2) = \mathcal{N}(\mathbf{z}_2|0, 1) \tag{4.79}$$

$$q(\mathbf{z}_1|\mathbf{x}) = \mathcal{N}(\mathbf{z}_1|\mu(\mathbf{x}), \sigma^2(\mathbf{x})) \tag{4.80}$$

$$q(\mathbf{z}_2|\mathbf{z}_1) = \mathcal{N}(\mathbf{z}_2|\mu(\mathbf{z}_1), \sigma^2(\mathbf{z}_1)), \tag{4.81}$$

where $\mu_i(\mathbf{v})$ means that a mean parameter is parameterized by a neural network that takes a random variable $\mathbf{v}$ as input, analogously we parameterize variances (i.e., diagonal covariance matrices). As we can see, this is a straightforward extension of a VAE we discussed before.

The two-level VAE is depicted in Fig. 4.19. Notice how the stochastic dependencies are defined, namely, there is always a dependency on a single random variable.

A Potential Pitfall

Alright, so are we done? Do we have a better class of VAEs? Unfortunately, the answer is **no**. We noticed that this two-level version of a VAE is a rather straightforward extension of a one-level VAE. Thus, our discussion about potential problems with latent variable models holds true. We get even get an extra insight if we look into the ELBO for the two-level VAE (if you do not remember how to derive the ELBO, please go back to the post on VAEs first):

$$ELBO(\mathbf{x}) = \mathbb{E}_{Q(\mathbf{z}_1,\mathbf{z}_2|\mathbf{x})}\Big[\ln p(\mathbf{x}|\mathbf{z}_1) - KL[q(\mathbf{z}_1|\mathbf{x})||p(\mathbf{z}_1|\mathbf{z}_2)] - KL[q(\mathbf{z}_2|\mathbf{z}_1)||p(\mathbf{z}_2)]\Big]. \quad (4.82)$$

To shed some light on the ELBO for the two-level VAE, we notice the following:

1. All conditions $(\mathbf{z}_1, \mathbf{z}_2, \mathbf{x})$ are either samples from $Q(\mathbf{z}_1, \mathbf{z}_2|\mathbf{x})$ or $p_{data}(\mathbf{x})$.
2. We obtain the Kullback–Leibler divergence terms by looking at the variables *per layer*. You are encouraged to derive the ELBO step-by-step, it is a great exercise to get familiar with the variational inference.
3. It is worth remembering that the Kullback–Leibler divergence is always non-negative.

Theoretically, everything should work perfectly fine, but there are a couple of potential problems. First, we initialize all DNNs that parameterize the distributions randomly. As a result, all Gaussians are basically standard Gaussians. Second, if the decoder is powerful and flexible, there is a huge danger that the model will try take advantage of the optimum for the last KL-term, $KL[q(\mathbf{z}_2|\mathbf{z}_1)||p(\mathbf{z}_2)]$], that is, $q(\mathbf{z}_2|\mathbf{z}_1) \approx p(\mathbf{z}_2) \approx \mathcal{N}(0, 1)$. Then, since $q(\mathbf{z}_2|\mathbf{z}_1) \approx \mathcal{N}(0, 1)$, the second layer is not used at all (it is a Gaussian noise) and we get back to the same issues as in the one-level VAE architecture. It turns out that learning the two-level VAE is even more problematic than a VAE with a single latent because even for a relatively simple decoder the second latent variable $\mathbf{z}_2$ is mostly unused [15, 70]. This effect is called the *posterior collapse*.

4.5.2.2 Top-Down VAEs

A take-away from our considerations in the two-level VAE is that adding an extra level does not necessarily provide anything comparing to the one-level VAE. However, so far we have considered only one class of variational posteriors, namely:

$$Q(\mathbf{z}_1, \mathbf{z}_2|\mathbf{x}) = q(\mathbf{z}_1|\mathbf{x})q(\mathbf{z}_2|\mathbf{z}_1). \quad (4.83)$$

A natural question is whether we can do better. You can already guess the answer, but before shouting it out loud, let us think for a second. In the generative part, we have *top-down* dependencies, going from the highest level of abstraction (latents) down to the observable variables. Let us repeat it here again:

$$p(\mathbf{x}, \mathbf{z}_1, \mathbf{z}_2) = p(\mathbf{x}|\mathbf{z}_1)p(\mathbf{z}_1|\mathbf{z}_2)p(\mathbf{z}_2). \tag{4.84}$$

Perhaps, we can mirror such dependencies in the variational posteriors as well. Then we get the following:

$$Q(\mathbf{z}_1, \mathbf{z}_2|\mathbf{x}) = q(\mathbf{z}_1|\mathbf{z}_2, \mathbf{x})q(\mathbf{z}_2|\mathbf{x}). \tag{4.85}$$

Do you see any resemblance? Yes, the variational posteriors have the extra $\mathbf{x}$, but the dependencies are pointing in the same direction. Why this could be beneficial? Because now we could have a shared *top-down* path that would make the variational posteriors and the generative part tightly connected through a shared parameterization. That could be a very useful inductive bias!

This idea was originally proposed in ResNet VAEs [18] and Ladder VAEs [71], and it was further developed in BIVA [44], NVAE [45], and the very deep VAE [46]. These approaches differ in their implementations and parameterizations used (i.e., architectures of DNNs); however, they all could be categorized as instantiations of top-down VAEs. The main idea, as mentioned before, is to share the top-down path between the variational posteriors and the generative distributions and use a *side*, deterministic path going from $\mathbf{x}$ to the last latents. Alright, let us write this idea down.

First, we have the top-down path that defines $p(\mathbf{x}|\mathbf{z}_1)$, $p(\mathbf{z}_1|\mathbf{z}_2)$, and $p(\mathbf{z}_2)$. Thus, we need a DNN that outputs μ_1 and σ_1^2 for given $\mathbf{z}_2$, and another DNN that outputs the parameters of $p(\mathbf{x}|\mathbf{z}_1)$ for given $\mathbf{z}_1$. Since $p(\mathbf{z}_2)$ is an unconditional distribution (e.g., the standard Gaussian), we do not need a separate DNN here.

Second, we have a side, deterministic path that gives two deterministic variables: $\mathbf{r}_1 = f_1(\mathbf{x})$ and $\mathbf{r}_2 = f_2(\mathbf{r}_1)$. Both transformations, f_1 and f_2, are DNNs. Then, we can use additional DNNs that return some modifications of the means and the variances, namely, $\Delta\mu_1$, $\Delta\sigma_1^2$, and $\Delta\mu_2$, $\Delta\sigma_2^2$. These modifications could be defined in many ways. Here we follow the way it is done in NVAE [45], namely, the modifications are relative location and scales of the values given in the top-down path. If you do not fully follow this idea, it should be clear once we define the variational posteriors.

Finally, we can define the whole procedure. We define various neural networks by specifying different indices. For sampling, we use the top-down path:

1. $\mathbf{z}_2 \sim \mathcal{N}(0, 1)$
2. $[\mu_1, \sigma_1^2] = NN_1(\mathbf{z}_2)$
3. $\mathbf{z}_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$
4. $\vartheta = NN_x(\mathbf{z}_1)$
5. $\mathbf{x} \sim p_\vartheta(\mathbf{x}|\mathbf{z}_1)$

Now (please focus!) we calculate samples from the variational posteriors as follows:

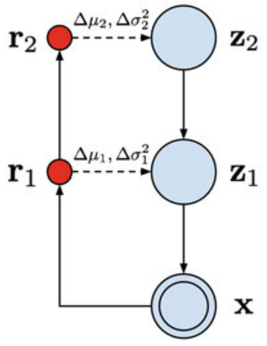

Fig. 4.20 An example of the top-down VAE. Red nodes denote the deterministic path and blue nodes depict random variables

1. (*Bottom-up deterministic path*) $\mathbf{r}_1 = f_1(\mathbf{x})$ and $\mathbf{r}_2 = f_2(\mathbf{r}_1)$
2. $[\Delta\mu_1, \Delta\sigma_1^2] = NN_{\Delta 1}(r_1)$
3. $[\Delta\mu_2, \Delta\sigma_2^2] = NN_{\Delta 2}(r_2)$
4. $\mathbf{z}_2 \sim \mathcal{N}(0 + \Delta\mu_2, 1 \cdot \Delta\sigma_2^2)$
5. $[\mu_1, \sigma_1^2] = NN_1(\mathbf{z}_2)$
6. $\mathbf{z}_1 \sim \mathcal{N}(\mu_1 + \Delta\mu_1, \sigma_1^2 \cdot \Delta\sigma_1^2)$
7. $\vartheta = NN_x(\mathbf{z}_1)$
8. $\mathbf{x} \sim p_\vartheta(\mathbf{x}|\mathbf{z}_1)$

These operations are schematically presented in Fig. 4.20.

Please note that the deterministic bottom-up path modifies parameters of the top-down path. As advocated by Vahdat and Kautz [45], this idea is especially useful because "when the prior moves, the approximate posterior moves accordingly, if not changed." Moreover, as noted in [45], the Kullback–Leibler between two Gaussians simplifies as follows (we remove some additional dependencies for clarity):

$$KL\left(q\left(z_i \mid \boldsymbol{x}\right) \| p\left(z_i\right)\right) = \frac{1}{2}\left(\frac{\Delta\mu_i^2}{\sigma_i^2} + \Delta\sigma_i^2 - \log\Delta\sigma_i^2 - 1\right).$$

Eventually, we implicitly force a close connection between the variational posteriors and the generative part. This inductive bias helps to encode information about the observables in the latents. Moreover, there is no need to use overly flexible decoders since the latents take care of distilling the essence from data. I know, it is still a bit hand-wavy since we do not define the magical *usefulness*, but I hope you get the picture. The top-down VAEs entangle the variational posteriors and the generative path and, as a result, the Kullback–Leibler terms will not collapse (i.e., they will be greater than zero). Empirical studies strongly back up this hypothesis [44–46, 71].

4.5.2.3 Code

Let us delve into an implementation of a top-down VAE. We stick to the two-level VAE to match the description provided above. We will use precisely the same steps as in the procedures used above. For clarity, we will use a single class to the code as similar to the mathematical expressions above as possible. We use the reparameterization trick for sampling. There is one difference between the math and the code, namely, in the code we use $\log \Delta\sigma$ instead of $\Delta\sigma$. Then, we use $\log \sigma + \log \Delta\sigma$ instead of $\sigma \cdot \Delta\sigma$ because $e^{\log a + \log b} = e^{\log a} \cdot e^{\log b} = a \cdot b$.

```python
class HierarchicalVAE(nn.Module):
    def __init__(self, nn_r_1, nn_r_2, nn_delta_1, nn_delta_2,
    nn_z_1, nn_x, num_vals=256, D=64, L=16, likelihood_type='
    categorical'):
        super(HierarchicalVAE, self).__init__()

        print('Hierachical VAE by JT.')

        # bottom-up path
        self.nn_r_1 = nn_r_1
        self.nn_r_2 = nn_r_2

        self.nn_delta_1 = nn_delta_1
        self.nn_delta_2 = nn_delta_2

        # top-down path
        self.nn_z_1 = nn_z_1
        self.nn_x = nn_x


        # other params
        self.D = D # dim of inputs

        self.L = L # dim of the second latent layer

        self.num_vals = num_vals # num of values per pixel

        self.likelihood_type = likelihood_type # the conditional
    likelihood type (categorical/bernoulli)

    # If you don't remember the reparameterization trick, please
    go back to the post on VAEs.
    def reparameterization(self, mu, log_var):
        std = torch.exp(0.5*log_var)
        eps = torch.randn_like(std)
        return mu + std * eps

    def forward(self, x, reduction='avg'):
        #=====
        # First, we need to calculate the bottom-up deterministic
     path.
        # Here we use a small trick to keep the delta of variance
     contained, namely, we apply the hard-tanh non-linearity.

```

```
        # bottom-up
        # step 1
        r_1 = self.nn_r_1(x)
        r_2 = self.nn_r_2(r_1)

        #step 2
        delta_1 = self.nn_delta_1(r_1)
        delta_mu_1, delta_log_var_1 = torch.chunk(delta_1, 2, dim=1)
        delta_log_var_1 = F.hardtanh(delta_log_var_1, -7., 2.)

        # step 3
        delta_2 = self.nn_delta_2(r_2)
        delta_mu_2, delta_log_var_2 = torch.chunk(delta_2, 2, dim=1)
        delta_log_var_2 = F.hardtanh(delta_log_var_2, -7., 2.)

        # Next, we can do the top-down path.

        # top-down
        # step 4
        z_2 = self.reparameterization(delta_mu_2, delta_log_var_2)

        # step 5
        h_1 = self.nn_z_1(z_2)
        mu_1, log_var_1 = torch.chunk(h_1, 2, dim=1)

        # step 6
        z_1 = self.reparameterization(mu_1 + delta_mu_1, log_var_1 + delta_log_var_1)

        # step 7
        h_d = self.nn_x(z_1)

        if self.likelihood_type == 'categorical':
            b = h_d.shape[0]
            d = h_d.shape[1]//self.num_vals
            h_d = h_d.view(b, d, self.num_vals)
            mu_d = torch.softmax(h_d, 2)

        elif self.likelihood_type == 'bernoulli':
            mu_d = torch.sigmoid(h_d)

        #=====ELBO
        # RE
        if self.likelihood_type == 'categorical':
            RE = log_categorical(x, mu_d, num_classes=self.num_vals, reduction='sum', dim=-1).sum(-1)

        elif self.likelihood_type == 'bernoulli':
            RE = log_bernoulli(x, mu_d, reduction='sum', dim=-1)

        # KL
```

```python
        # For the Kullback--Leibler part, we need calculate two divergences:
        # 1) KL[q(z_2|z) || p(z_2)] where p(z_2) = N(0,1)
        # 2) KL[q(z_1|z_2, x) || p(z_1|z_2)]
        # Note: We use the analytical form of the KL between two Gaussians here. If you use a different distribution,
        # please pay attention! You would need to use a different expression here.
        KL_z_2 = 0.5 * (delta_mu_2**2 + torch.exp(delta_log_var_2) - delta_log_var_2 - 1).sum(-1)
        KL_z_1 = 0.5 * (delta_mu_1**2 / torch.exp(log_var_1) + torch.exp(delta_log_var_1) -\
                        delta_log_var_1 - 1).sum(-1)

        KL = KL_z_1 + KL_z_2

        # Final ELBO
        if reduction == 'sum':
            loss = -(RE - KL).sum()
        else:
            loss = -(RE - KL).mean()

        return loss

    # Sampling is the top-down path but without calculating delta mean and delta variance.
    def sample(self, batch_size=64):
        # step 1
        z_2 = torch.randn(batch_size, self.L)
        # step 2
        h_1 = self.nn_z_1(z_2)
        mu_1, log_var_1 = torch.chunk(h_1, 2, dim=1)
        # step 3
        z_1 = self.reparameterization(mu_1, log_var_1)

        # step 4
        h_d = self.nn_x(z_1)

        if self.likelihood_type == 'categorical':
            b = batch_size
            d = h_d.shape[1]//self.num_vals
            h_d = h_d.view(b, d, self.num_vals)
            mu_d = torch.softmax(h_d, 2)
            # step 5
            p = mu_d.view(-1, self.num_vals)
            x_new = torch.multinomial(p, num_samples=1).view(b,d)

        elif self.likelihood_type == 'bernoulli':
            mu_d = torch.sigmoid(h_d)
            # step 5
            x_new = torch.bernoulli(mu_d)
        return x_new
```

Listing 4.12 A top-down VAE class

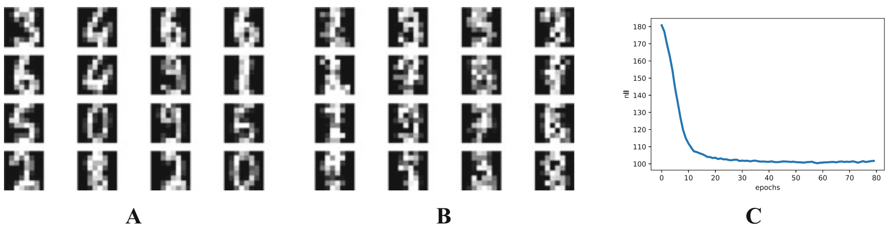

Fig. 4.21 An example of outcomes after the training: (**a**) Randomly selected real images. (**b**) Unconditional generations from the top-down VAE. (**c**) The validation curve during training

That's it! Now we are ready to run the full code (take a look at: https://github.com/jmtomczak/intro_dgm). After training our top-down VAE, we should obtain results like in Fig. 4.21.

4.5.2.4 Further Reading

What we have discussed here is just touching upon the topic. Hierarchical models in probabilistic modeling seem to be important research direction and modeling paradigm. Moreover, the technical details are also crucial for achieving state-of-the-art performance. I strongly suggest reading about NVAE [45], ResNet VAE [18], Ladder VAE [71], BIVA [44], and very deep VAEs [46] and compare various tricks and parameterizations used therein. These models share the same idea, but implementations vary significantly.

The research on hierarchical generative modeling is very up-to-date and develops very quickly. As a result, this is nearly impossible to mention even a fraction of interesting papers. I will mention only a few worth noticing papers:

- Pervez and Gavves [72] provides an insightful analysis about a potential problem with hierarchical VAEs, namely, the KL divergence term is closely related to the harmonics of the parameterizing function. In other words, using DNNs result in high-frequency components of the KL term and, eventually, to the posterior collapse. The authors propose to smooth the VAE by applying Ornstein–Uhlenbeck (OU) Semigroup. I refer to the original paper for details.
- Wu et al. [73] proposes greedy layer-wise learning of a hierarchical VAE. The authors used this idea in the context of video prediction. The main motivation for utilizing greedy layer-wise learning is a limited amount of computational resources. However, the idea of greedy layer-wise training has been extensively utilized in the past [66–68].
- Gatopoulos and Tomczak [25] discusses incorporating pre-defined transformations like downscaling into the model. The idea is to learn a reversed transformation to, e.g., downscaling in a stochastic manner. The resulting VAE has a set of auxiliary variables (e.g., downscaled versions of observables) and a set of latent variables that encode missing information in the auxiliary variables.

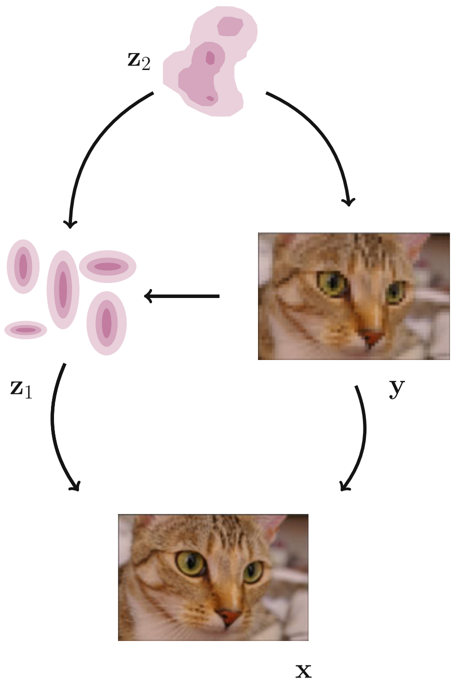

Fig. 4.22 A two-level VAE with an auxiliary set of deterministic variables **y** (e.g., downscaled images)

The hypothesis in such an approach is that learning a distribution over smaller or already processed observable variables is easier and, thus, we can decompose the problem into multiple problems of learning simpler distributions. A diagram for this approach is presented in Fig. 4.22.

The beauty of the latent variable modeling paradigm is that we can play with stochastic relationships among objects and, eventually, formulate a *useful* representation of data. As we will see in the next blog posts, there are other interesting classes of models that take advantage of diffusion models and energy functions.

4.5.3 Diffusion-Based Deep Generative Models

4.5.3.1 Introduction

In Sect. 4.5, we discussed the issue of learning *useful* representations in latent variable models, taking a closer look at hierarchical Variational Auto-Encoders. We hypothesize that we can obtain *useful* data representations by applying a hierarchical latent variable model. Moreover, highlighted a real problem in hierarchical VAEs of the variational posterior collapsing to the prior, resulting in learning meaningless

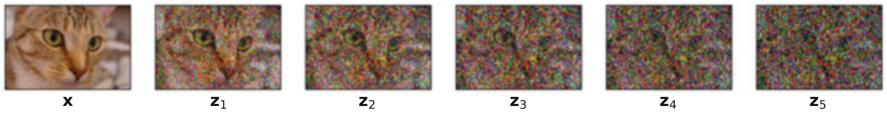

Fig. 4.23 An example of applying a Gaussian diffusion to an image of a cat, $\mathbf{x}$

representation. In other words, it seems that architecture with bottom-up variational posteriors (i.e., stochastic dependencies going from observables to the last latents) and top-down generative distributions seems to be a mediocre inductive bias and is rather troublesome to train. A potential solution is top-down VAEs. However, is there nothing we can do about the *vanilla* structure? As you may imagine, nothing is lost and some approaches take advantage of the bottom-up and the top-down structures. Here, we will look into the *diffusion-based deep generative models* (DDGMs) (a.k.a. *deep diffusion probabilistic models*) [74, 75].

DDGM could be briefly explained as hierarchical VAEs with the bottom-up path (i.e., the variational posteriors) defined by a diffusion process (e.g., a Gaussian diffusion) and the top-down path parameterized by DNNs (a reversed diffusion). Interestingly, the bottom-up path could be **fixed**, namely, it necessarily does not have any learnable parameters. An example of applying a Gaussian diffusion is presented in Fig. 4.23. Since the variational posteriors are fixed, we can think of them as adding Gaussian noise at each layer. Then, the final layer resembles Gaussian noise (see $\mathbf{z}_5$ in Fig. 4.23). If we recall the discussion about a potential issue of the posterior collapse in hierarchical VAEs, this should not be a problem anymore. Why? Because we should get a standard Gaussian distribution in the last layer **by design**. Pretty neat, isn't it?

DDGMs have become extremely popular these days. They are appealing for at least two reasons:

1. They give amazing results for image synthesis [74, 76, 77], audio synthesis [78], and promising results for text synthesis [79, 80] while being relatively simple to implement.
2. They are closely related to stochastic differential equations and, thus, their theoretical properties seem to be especially of great interest [81–83].

There are two potential drawbacks though, namely:

1. DDGMs are unable (for now at least) to learn a representation.
2. Similarly to flow-based models, the dimensionality of input is kept across the whole model (i.e., there is no bottleneck on the way).

4.5.3.2 Model Formulation

Originally, deep diffusion probabilistic models were proposed in [75] and they took inspiration from non-equilibrium statistical physics. The main idea is to iteratively destroy the structure in data through a forward diffusion process and, afterward, to learn a reverse diffusion process to restore the structure in data. In a follow-up paper [74] recent developments in deep learning were used to train a powerful and flexible diffusion-based deep generative model that achieved SOTA results in the task of image synthesis. Here, we will abuse the original notation to make a clear connection between hierarchical latent variable models and DDGMs. As previously, we are interested in finding a distribution over data, $p_\theta(\mathbf{x})$; however, we assume an additional set of latent variables $\mathbf{z}_{1:T} = [\mathbf{z}_1, \ldots, \mathbf{z}_T]$. The marginal likelihood is defined by integrating out all latents:

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}, \mathbf{z}_{1:T})\, \mathrm{d}\mathbf{z}_{1:T}. \tag{4.86}$$

The joint distribution is modeled as a first-order Markov chain with Gaussian transitions, namely:

$$p_\theta(\mathbf{x}, \mathbf{z}_{1:T}) = p_\theta(\mathbf{x}|\mathbf{z}_1) \left(\prod_{i=1}^{T-1} p_\theta(\mathbf{z}_i|\mathbf{z}_{i+1}) \right) p_\theta(\mathbf{z}_T), \tag{4.87}$$

where $\mathbf{x} \in \mathbb{R}^D$ and $\mathbf{z}_i \in \mathbb{R}^D$ for $i = 1, \ldots, T$. Please note that the latents have the same dimensionality as the observables. This is the same situation as in the case of flow-based models. We parameterize all distributions using DNNs.

So far, we have not introduced anything new! This is again a hierarchical latent variable model. As in the case of hierarchical VAEs, we can introduce a family of variational posteriors as follows:

$$Q_\phi(\mathbf{z}_{1:T}|\mathbf{x}) = q_\phi(\mathbf{z}_1|\mathbf{x}) \left(\prod_{i=2}^{T} q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1}) \right). \tag{4.88}$$

The key point is how we define these distributions. Before, we used normal distributions parameterized by DNNs, but now we formulate them as the following Gaussian diffusion process [75]:

$$q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1}) = \mathcal{N}(\mathbf{z}_i|\sqrt{1-\beta_i}\mathbf{z}_{i-1}, \beta_i \mathbf{I}), \tag{4.89}$$

where $\mathbf{z}_0 = \mathbf{x}$. Notice that a single step of the diffusion, $q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1})$, works in a relatively easy way. Namely, it takes the previously generated object $\mathbf{z}_{i-1}$, scales it by $\sqrt{1-\beta_i}$, and then adds noise with variance β_i. To be even more explicit, we can write it using the reparameterization trick:

$$\mathbf{z}_i = \sqrt{1-\beta_i}\mathbf{z}_{i-1} + \sqrt{\beta_i} \odot \epsilon, \tag{4.90}$$

where $\epsilon \sim \mathcal{N}(0, \mathbf{I})$. In principle, β_i could be learned by backpropagation; however, as noted by Sohl-Dickstein et al. [75] and Ho et al.[74], it could be fixed. For instance, [74] suggests to change it linearly from $\beta_1 = 10^{-4}$ to $\beta_T = 0.02$.

Since we realized that the difference between a DDGM and a hierarchical VAE lies in the definition of the variational posteriors and the dimensionality of the latents, but the whole construction is basically the same, we can predict what is the learning objective. Do you remember? Yes, it is ELBO! We can derive the ELBO as follows:

$$
\begin{aligned}
\ln p_\theta(\mathbf{x}) =& \ln \int Q_\phi(\mathbf{z}_{1:T}|\mathbf{x}) \frac{p_\theta(\mathbf{x}, \mathbf{z}_{1:T})}{Q_\phi(\mathbf{z}_{1:T}|\mathbf{x})} \,\mathrm{d}\mathbf{z}_{1:T} \\
\geq& \mathbb{E}_{Q_\phi(\mathbf{z}_{1:T}|\mathbf{x})} \Bigg[\ln p_\theta(\mathbf{x}|\mathbf{z}_1) + \sum_{i=1}^{T-1} \ln p_\theta(\mathbf{z}_i|\mathbf{z}_{i+1}) + \ln p_\theta(\mathbf{z}_T) + \\
& - \sum_{i=2}^{T} \ln q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1}) - \ln q_\phi(\mathbf{z}_1|\mathbf{x}) \Bigg] \\
=& \mathbb{E}_{Q_\phi(\mathbf{z}_{1:T}|\mathbf{x})} \Bigg[\ln p_\theta(\mathbf{x}|\mathbf{z}_1) + \ln p_\theta(\mathbf{z}_1|\mathbf{z}_2) + \sum_{i=2}^{T-1} \ln p_\theta(\mathbf{z}_i|\mathbf{z}_{i+1}) + \ln p_\theta(\mathbf{z}_T) + \\
& - \sum_{i=2}^{T-1} \ln q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1}) - \ln q_\phi(\mathbf{z}_T|\mathbf{z}_{T-1}) - \ln q_\phi(\mathbf{z}_1|\mathbf{x}) \Bigg] \\
=& \mathbb{E}_{Q_\phi(\mathbf{z}_{1:T}|\mathbf{x})} \Bigg[\ln p_\theta(\mathbf{x}|\mathbf{z}_1) + \sum_{i=2}^{T-1} \big(\ln p_\theta(\mathbf{z}_i|\mathbf{z}_{i+1}) - \ln q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1}) \big) + \\
& + \ln p_\theta(\mathbf{z}_T) - \ln q_\phi(\mathbf{z}_T|\mathbf{z}_{T-1}) + \\
& + \ln p_\theta(\mathbf{z}_1|\mathbf{z}_2) - \ln q_\phi(\mathbf{z}_1|\mathbf{x}) \Bigg] \qquad (4.91) \\
\overset{df}{=}& \mathcal{L}(\mathbf{x}; \theta, \phi).
\end{aligned}
$$

We can rewrite the ELBO in terms of Kullback–Leibler divergences (note that we use the expected value with respect to $Q_\phi(\mathbf{z}_{-i}|\mathbf{x})$ to highlight that a proper variational posterior is used for the definition of the Kullback–Leibler divergence):

$$
\begin{aligned}
\mathcal{L}(\mathbf{x}; \theta, \phi) =& \mathbb{E}_{Q_\phi(\mathbf{z}_{1:T}|\mathbf{x})} [\ln p_\theta(\mathbf{x}|\mathbf{z}_1)] + \\
& - \sum_{i=2}^{T-1} \mathbb{E}_{Q_\phi(\mathbf{z}_{-i}|\mathbf{x})} \big[KL \left[q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1}) || p_\theta(\mathbf{z}_i|\mathbf{z}_{i+1}) \right] \big] + \\
& - \mathbb{E}_{Q_\phi(\mathbf{z}_{-T}|\mathbf{x})} \big[KL \left[q_\phi(\mathbf{z}_T|\mathbf{z}_{T-1}) || p_\theta(\mathbf{z}_T) \right] \big] +
\end{aligned}
$$

$$- \mathbb{E}_{Q_\phi(\mathbf{z}_{-1}|\mathbf{x})} \left[KL \left[q_\phi(\mathbf{z}_1|\mathbf{x}) || p_\theta(\mathbf{z}_1|\mathbf{z}_2) \right] \right]. \tag{4.92}$$

Example 4.1 Let us take $T = 5$. This is not much (e.g., [74] uses $T = 1000$), but it is easier to explain the idea with a very specific model. Moreover, let us use a fixed $\beta_t \equiv \beta$. Then we have the following DDGM:

$$p_\theta(\mathbf{x}, \mathbf{z}_{1:5}) = p_\theta(\mathbf{x}|\mathbf{z}_1) p_\theta(\mathbf{z}_1|\mathbf{z}_2) p_\theta(\mathbf{z}_2|\mathbf{z}_3) p_\theta(\mathbf{z}_3|\mathbf{z}_4) p_\theta(\mathbf{z}_4|\mathbf{z}_5) p_\theta(\mathbf{z}_5), \tag{4.93}$$

and the variational posteriors:

$$Q_\phi(\mathbf{z}_{1:5}|\mathbf{x}) = q_\phi(\mathbf{z}_1|\mathbf{x}) q_\phi(\mathbf{z}_2|\mathbf{z}_1) q_\phi(\mathbf{z}_3|\mathbf{z}_2) q_\phi(\mathbf{z}_4|\mathbf{z}_3) q_\phi(\mathbf{z}_5|\mathbf{z}_4). \tag{4.94}$$

In the considered case, the ELBO takes the following form:

$$\begin{aligned}
\mathcal{L}(\mathbf{x}; \theta, \phi) =& \mathbb{E}_{Q_\phi(\mathbf{z}_{1:5}|\mathbf{x})} \left[\ln p_\theta(\mathbf{x}|\mathbf{z}_1) \right] + \\
& - \sum_{i=2}^{4} \mathbb{E}_{Q_\phi(\mathbf{z}_{-i}|\mathbf{x})} \left[KL \left[q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1}) || p_\theta(\mathbf{z}_i|\mathbf{z}_{i+1}) \right] \right] + \\
& - \mathbb{E}_{Q_\phi(\mathbf{z}_{-i}|\mathbf{x})} \left[KL \left[q_\phi(\mathbf{z}_5|\mathbf{z}_4) || p_\theta(\mathbf{z}_5) \right] \right] + \\
& - \mathbb{E}_{Q_\phi(\mathbf{z}_{-i}|\mathbf{x})} \left[KL \left[q_\phi(\mathbf{z}_1|\mathbf{x}) || p_\theta(\mathbf{z}_1|\mathbf{z}_2) \right] \right],
\end{aligned} \tag{4.95}$$

where

$$p_\theta(\mathbf{z}_5) = \mathcal{N}(\mathbf{z}_5|0, \mathbf{I}). \tag{4.96}$$

The last interesting question is how to model inputs and, eventually, what distribution we should use to model $p(\mathbf{x}|\mathbf{z}_1)$. So far, we used the categorical distribution because pixels were integer-valued. However, for the DDGM, we assume that they are continuous and we will use a simple trick. We normalize our inputs to values between -1 and 1 and apply the Gaussian distribution with the unit variance and the mean being constrained to $[-1, 1]$ using the tanh non-linearity:

$$p(\mathbf{x}|\mathbf{z}_1) = \mathcal{N}(\mathbf{x}|\tanh\left(NN(\mathbf{z}_1)\right), \mathbf{I}), \tag{4.97}$$

where $NN(\mathbf{z}_1)$ is a neural network. As a result, since the variance is one, $\ln p(\mathbf{x}|\mathbf{z}_1) = -MSE\left(\mathbf{x}, \tanh(NN(\mathbf{z}_1))\right) + const$, so it is equivalent to the (negative) *Mean Squared Error*! I know, it is not a perfect way to do, but it is simple and it works. ■

That's it! As you can see, there is no special magic here and we are ready to implement our DDGM. In fact, we can use the code of a hierarchical VAE and modify it accordingly. What is convenient about the DDGM is that the forward diffusion (i.e., the variational posteriors) are fixed and we need to sample from them, and only the reverse diffusion requires applying DDNs. But without any further mumbling, let us dive into the code!

4.5.3.3 Code

At this point, you might think that it is pretty complicated and a lot of math is involved here. However, if you followed our previous discussions on VAEs, it should be rather clear what we need to do here.

```python
class DDGM(nn.Module):
    def __init__(self, p_dnns, decoder_net, beta, T, D):
        super(DDGM, self).__init__()

        print('DDGM by JT.')

        self.p_dnns = p_dnns  # a list of sequentials; a single Sequential defines a DNN to parameterize a distribution p(z_i | z_i+1)

        self.decoder_net = decoder_net # the last DNN for p(x|z1)

        # other params
        self.D = D # the dimensionality of the inputs (necessary for sampling!)

        self.T = T # the number of steps

        self.beta = torch.FloatTensor([beta]) # the fixed variance of diffusion

    # The reparameterization trick for the Gaussian distribution
    @staticmethod
    def reparameterization(mu, log_var):
        std = torch.exp(0.5*log_var)
        eps = torch.randn_like(std)
        return mu + std * eps

    # The reparameterization trick for the Gaussian forward diffusion
    def reparameterization_gaussian_diffusion(self, x, i):
        return torch.sqrt(1. - self.beta) * x + torch.sqrt(self.beta) * torch.randn_like(x)

    def forward(self, x, reduction='avg'):
        # =====
        # Forward Diffusion
        # Please note that we just ‘‘wander’’ around in the space using Gaussian random walk.
        # We save all z's in a list
        zs = [self.reparameterization_gaussian_diffusion(x, 0)]

        for i in range(1, self.T):
            zs.append(self.reparameterization_gaussian_diffusion(zs[-1], i))

        # =====
        # Backward Diffusion
        # We start with the last z and proceed to x.
        # At each step, we calculate means and variances.
        mus = []
        log_vars = []

        for i in range(len(self.p_dnns) - 1, -1, -1):
            h = self.p_dnns[i](zs[i+1])
            mu_i, log_var_i = torch.chunk(h, 2, dim=1)
            mus.append(mu_i)
            log_vars.append(log_var_i)

        # The last step: outputting the means for x.
        # NOTE: We assume the last distribution is Normal(x | tanh(NN(z_1)), 1)!
        mu_x = self.decoder_net(zs[0])

        # =====ELBO
        # RE
        # This is equivalent to - MSE(x, mu_x) + const
        RE = log_standard_normal(x - mu_x).sum(-1)

        # KL: We need to go through all the levels of latents
        KL = (log_normal_diag(zs[-1], torch.sqrt(1. - self.beta) * zs[-1], torch.log(self.beta)) - log_standard_normal(zs[-1])).sum(-1)

        for i in range(len(mus)):
            KL_i = (log_normal_diag(zs[i], torch.sqrt(1. - self.beta) * zs[i], torch.log(self.beta)) - log_normal_diag(zs[i], mus[i], log_vars[i])).sum(-1)

            KL = KL + KL_i

        # Final ELBO
        if reduction == 'sum':
            loss = -(RE - KL).sum()
        else:
            loss = -(RE - KL).mean()

        return loss

    # Sampling is the reverse diffusion with sampling at each step.
    def sample(self, batch_size=64):
        z = torch.randn([batch_size, self.D])
        for i in range(len(self.p_dnns) - 1, -1, -1):
            h = self.p_dnns[i](z)
            mu_i, log_var_i = torch.chunk(h, 2, dim=1)
            z = self.reparameterization(torch.tanh(mu_i),
    log_var_i)

        mu_x = self.decoder_net(z)

        return mu_x

    # For sanity check, we also can sample from the forward
    diffusion.
    # The result should resemble a white noise.
    def sample_diffusion(self, x):
        zs = [self.reparameterization_gaussian_diffusion(x, 0)]

        for i in range(1, self.T):
            zs.append(self.reparameterization_gaussian_diffusion(
    zs[-1], i))

        return zs[-1]
```

Listing 4.13 A DDGM class

That's it! Now we are ready to run the full code (take a look at: https://github.com/jmtomczak/intro_dgm). After training our DDGM, we should obtain results like in Fig. 4.24.

4.5.3.4 Discussion

Extensions

Currently, DDGMs are very popular deep generative models. What we present here is very close to the original formulation of the DDGMs [75]. However, [74] introduced many interesting insights and improvements on the original idea, such as:

- Since the forward diffusion consists of Gaussian distributions and linear transformations of means, it is possible to analytically marginalize out intermediate steps, which yields:

$$q(\mathbf{z}_t|\mathbf{x}) = \mathcal{N}(\mathbf{z}_t|\sqrt{\bar{\alpha}_t}\mathbf{x}, (1 - \bar{\alpha}_t\mathbf{I}), \tag{4.98}$$

  where $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_t$. This is an extremely interesting result because we can sample $\mathbf{z}_t$ without sampling all intermediate steps!
- As a follow-up, we can calculate also the following distribution:

$$q(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{x}) = \mathcal{N}\left(\mathbf{z}_{t-1}|\tilde{\boldsymbol{\mu}}_t\left(\mathbf{z}_t, \mathbf{x}\right), \tilde{\beta}_t\mathbf{I}\right), \tag{4.99}$$

  where:

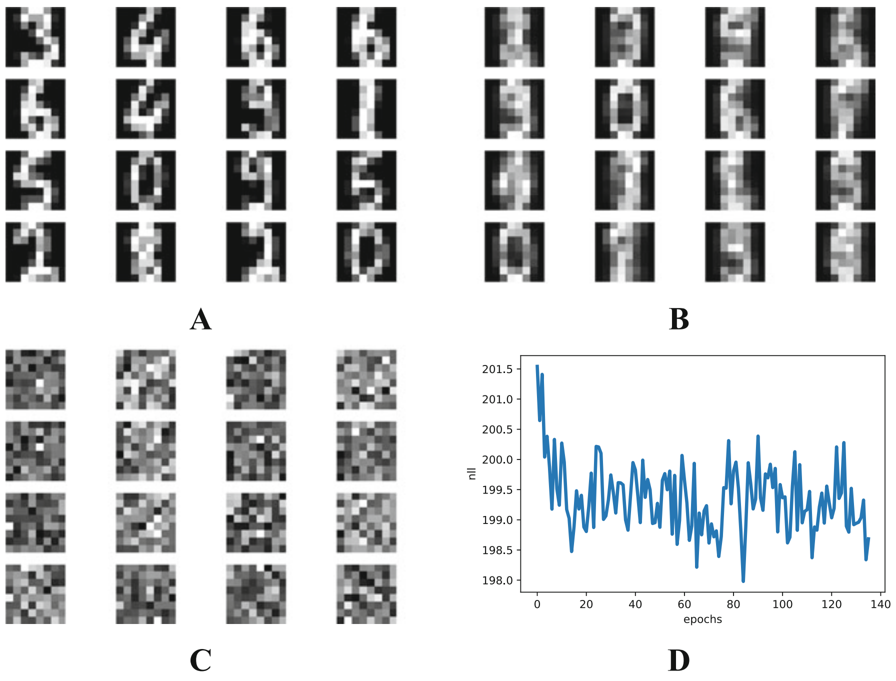

Fig. 4.24 An example of outcomes after the training: (**a**) Randomly selected real images. (**b**) Unconditional generations from the DDGM. (**c**) A visualization of the last stochastic level after applying the forward diffusion. As expected, the resulting images resemble pure noise. (**d**) An example of a validation curve for the ELBO

$$\tilde{\boldsymbol{\mu}}_t\left(\mathbf{z}_t, \mathbf{x}\right) = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\mathbf{x} + \frac{\sqrt{\alpha_t}\left(1-\bar{\alpha}_{t-1}\right)}{1-\bar{\alpha}_t}\mathbf{z}_t \tag{4.100}$$

and

$$\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t. \tag{4.101}$$

Then, we can rewrite the ELBO as follows:

$$\begin{aligned}\mathcal{L}(\mathbf{x};\theta,\phi) = \mathbb{E}_Q\Big[& \underbrace{KL\left[q(\mathbf{z}_T|\mathbf{x})\|p(\mathbf{z}_T)\right]}_{L_T} + \\ & + \sum_{t>1}\underbrace{KL\left[q(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{x})\|p_\theta(\mathbf{z}_{t-1}|\mathbf{z}_t)\right]}_{L_{t-1}} + \\ & \underbrace{-\log p_\theta\left(\mathbf{x}\mid\mathbf{z}_1\right)}_{L_0}\Big].\end{aligned} \tag{4.102}$$

Now, instead of differentiating all components of the objective, we can randomly pick L_t and treat it as the objective. Such an approach has a clear advantage: It does not require keeping all gradients in the memory! Instead, we update only one layer at a time. Since our training is stochastic anyway (remember that we typically use stochastic gradient descent), we can introduce this extra stochasticity during training. And the benefit is enormous because we can train extremely deep models, even with 1000 layers as in [74].

- If you play a little with the code here, you may notice that training the reverse diffusion is pretty problematic. Why? Because by adding extra layers of latents, we add additional KL-terms. In the case of far from perfect models, each KL-term will be strictly greater than 0 and, thus, we will increase the ELBO with each additional step of stochasticity. Therefore, it is so important to be smart about formulating reverse diffusion. [74] again provides very interesting insight! We skip here the full reasoning, but it turns out that to make the model $p_\theta(\mathbf{z}_{t-1} \mid \mathbf{z}_t) = \mathcal{N}\left(\mathbf{z}_{t-1} | \mu_\theta(\mathbf{z}_t), \sigma_t^2 \mathbf{I}\right)$ more powerful, $\mu_\theta(\mathbf{z}_t)$ should be as close as possible to $\tilde{\boldsymbol{\mu}}_t(\mathbf{z}_t, \mathbf{x})$. Following the derivation in [74], we get

$$\mu_\theta(\mathbf{z}_t) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{z}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{z}_t)\right),$$

where $\epsilon_\theta(\mathbf{z}_t)$ is parameterized by a DNN and it aims for estimating the noise from $\mathbf{z}_t$.

- Even further, each L_t could be simplified to:

$$L_{t,\text{simple}} = \mathbb{E}_{t,\mathbf{x}_0,\epsilon}\left[\left\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta\left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t\right)\right\|^2\right].$$

Ho et al. [74] provides empirical results that such an objective could be beneficial for training and the final quality of synthesized images.

There were many follow-ups on [74], we mention only a few of them here:

- *Improving DDGMs*: Nichol and Dhariwal [84] introduces further tricks on improving training stability and performance of DDGMs by learning the covariance matrices in the reverse diffusion, proposing a different noise schedule, among others. Interestingly, the authors of [76] propose to extend the observables (i.e., pixels) and latents with Fourier features as additional channels. The rationale behind this is that the high frequency features allow neural networks to cope better with noise. Moreover, they introduce a new noise scheduling and an application of certain functions to improve the numerical stability of the forward diffusion process.
- *Sampling speed-up*: Kong and Ping [85] and Watson et al. [86] focus on speeding up the sampling process.
- *Superresolution*: Saharia et al. [77] uses DDGMs for the task of superresolution.

- *Connection to score-based generative models*: It turns out that score-based generative models [87] are closely related to DDGMs as indicated by Song and Kingma [88] and Song and Ermon [87]. This perspective gives a neat connection between DDGMs and stochastic differential equations [81, 83].
- *Variational perspective on DDGMs*: There is a nice variational perspective on DDGMs [76, 81] that gives an intuitive interpretation of DDGMs and allows achieving astonishing results on the image synthesis task. It is worth to study [76] further because there are a lot of interesting improvements presented therein.
- *Discrete DDGMs*: So far, DDGMs are mainly focused on continuous spaces. [79, 80] propose DDGMs on discrete spaces.
- *DDGMs for audio*: Kong et al. [78] proposes to use DDGMs for audio synthesis.
- *DDGMs as priors in VAEs*: Vahdat et al. [89] and Wehenkel and Louppe [90] propose to use DDGMs as flexible priors in VAEs.

DDGMs vs. VAEs vs. Flows

In the end, it is worth making a comparison of DDGMs with VAEs and flow-based models. In Table 4.1, we provide a comparison based on rather arbitrary criteria:

- Whether the training procedure is stable or not
- Whether the likelihood could be calculated
- Whether a reconstruction is possible
- Whether a model is invertible
- Whether the latent representation could be lower dimensional than the input space (i.e., a bottleneck in a model)

The three models share a lot of similarities. Overall, training is rather stable even though numerical issues could arise in all models. Hierarchical VAEs could be seen as a generalization of DDGMs. There is an open question of whether it is indeed more beneficial to use fixed variational posteriors by sacrificing the possibility of having a bottleneck. There is also a connection between flows and DDGMs. Both classes of models aim for going from data to noise. Flows do that by applying invertible transformations, while DDGMs accomplish that by a diffusion process. In flows, we know the inverse but we pay the price of calculating the Jacobian-determinant, while DDGMs require flexible parameterizations of the reverse diffusion but there are no extra strings attached. Looking into connections among these models is definitely an interesting research line.

Table 4.1 A comparison among DDGMs, VAEs, and Flows

Model	Training	Likelihood	Reconstruction	Invertible	Bottleneck (latents)
DDGMs	Stable	Approximate	Difficult	No	No
VAEs	Stable	Approximate	Easy	No	Possible
Flows	Stable	Exact	Easy	Yes	No