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 $\mathbf{z}$'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 $\mathbf{x}$'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:

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

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

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

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

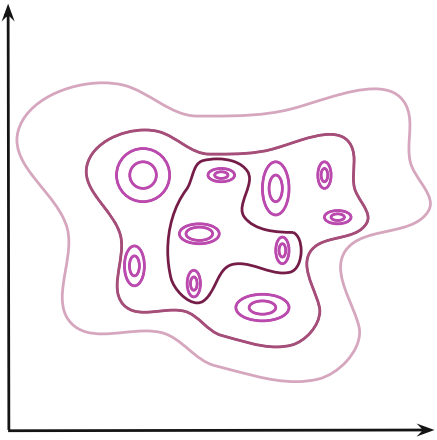


Fig. 4.5 An example of the aggregated posterior. Individual points are encoded as Gaussians in the 2D latent space (magenta), and the mixture of variational posteriors (the aggregated posterior) is presented by contours

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

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

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

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

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: $\sigma_i^2 \to +\infty$. 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.

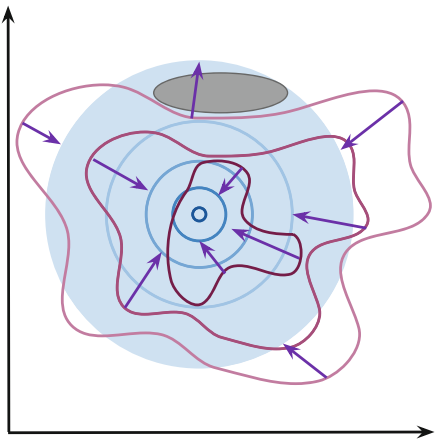


Fig. 4.6 An example of the effect of the cross-entropy optimization with a non-learnable prior. The aggregated posterior (purple contours) tries to match the non-learnable prior (in blue). The purple arrows indicate the change of the aggregated posterior. An example of a hole is presented as a dark gray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

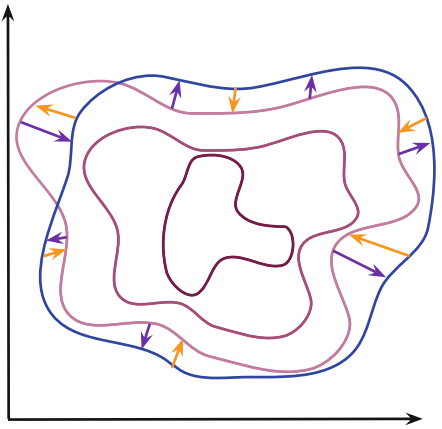


Fig. 4.7 An example of the effect of the cross-entropy optimization with a learnable prior. The aggregated posterior (purple contours) tries to match the learnable prior (blue contours). Notice that the aggregated posterior is modified to fit the prior (purple arrows), but also the prior is updated to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

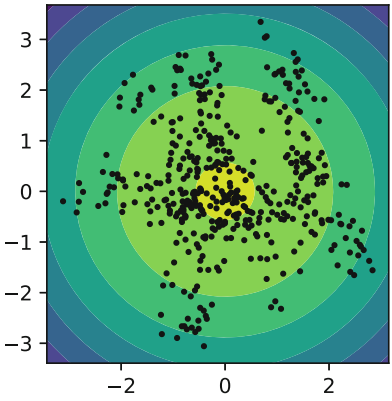


Fig. 4.8 An example of the standard Gaussian prior (contours) and the samples from the aggregated posterior (black dots)

The code for the standard Gaussian prior is presented below:

```
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:

```
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
```

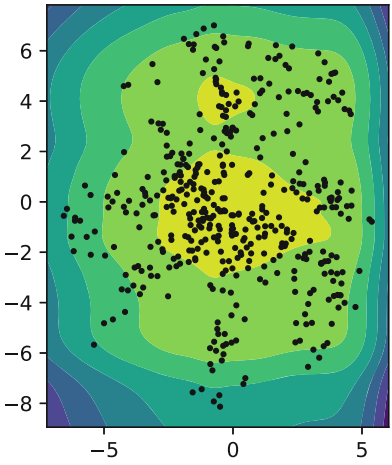


Fig. 4.9 An example of the MoG prior (contours) and the samples from the aggregated posterior (black dots)

```
        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

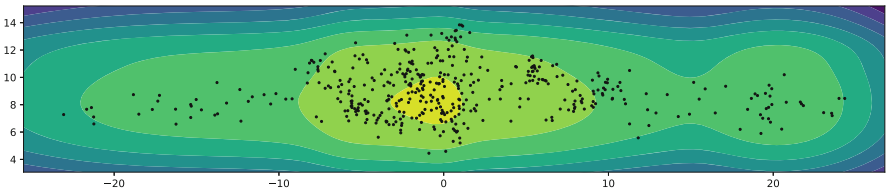


Fig. 4.10 An example of the VampPrior (contours) and the samples from the aggregated posterior (black dots)

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
```

```
13          self.u = nn.Parameter(u)
14
15          # mixing weights
16          self.w = nn.Parameter(torch.zeros(self.u.shape[0], 1, 1))
     # K x 1 x 1
17
18      def get_params(self):
19          # u-->encoder-->mu, lof_var
20          mean_vampprior, logvar_vampprior = self.encoder.encode(
    self.u) #(K x L), (K x L)
21          return mean_vampprior, logvar_vampprior
22
23      def sample(self, batch_size):
24          # u-->encoder-->mu, lof_var
25          mean_vampprior, logvar_vampprior = self.get_params()
26
27          # mixing probabilities
28          w = F.softmax(self.w, dim=0) # K x 1 x 1
29          w = w.squeeze()
30
31          # pick components
32          indexes = torch.multinomial(w, batch_size, replacement=
    True)
33
34          # means and logvars
35          eps = torch.randn(batch_size, self.L)
36          for i in range(batch_size):
37              indx = indexes[i]
38              if i == 0:
39                  z = mean_vampprior[[indx]] + eps[[i]] * torch.exp
    (logvar_vampprior[[indx]])
40              else:
41                  z = torch.cat((z, mean_vampprior[[indx]] + eps[[i
    ]] * torch.exp(logvar_vampprior[[indx]])), 0)
42          return z
43
44      def log_prob(self, z):
45          # u-->encoder-->mu, lof_var
46          mean_vampprior, logvar_vampprior = self.get_params() # (K
     x L) & (K x L)
47
48          # mixing probabilities
49          w = F.softmax(self.w, dim=0) # K x 1 x 1
50
51          # log-mixture-of-Gaussians
52          z = z.unsqueeze(0) # 1 x B x L
53          mean_vampprior = mean_vampprior.unsqueeze(1) # K x 1 x L
54          logvar_vampprior = logvar_vampprior.unsqueeze(1) # K x 1
    x L
55
56          log_p = log_normal_diag(z, mean_vampprior,
    logvar_vampprior) + torch.log(w) # K x B x L
57          log_prob = torch.logsumexp(log_p, dim=0, keepdim=False) #
     B x L
```

```
58
59          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, $g_\gamma : \mathbb{R}^C \rightarrow \mathbb{R}^{2\times M}$. 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:

```
1 class GTMPrior(nn.Module):
2     def __init__(self, L, gtm_net, num_components, u_min=-1.,
      u_max=1.):
3     super(GTMPrior, self).__init__()
4
5     self.L = L
6
7     # 2D manifold
```

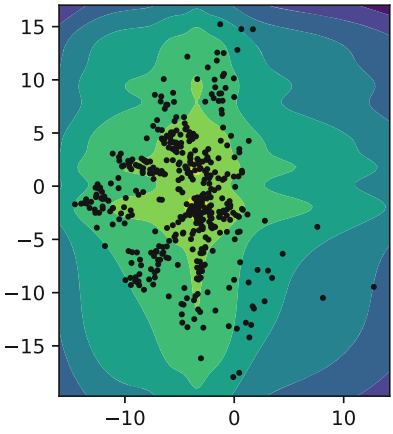


Fig. 4.11 An example of the GTM-based prior (contours) and the samples from the aggregated posterior (black dots)

```
    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)
```

```
41
42          # means and logvars
43          eps = torch.randn(batch_size, self.L)
44          for i in range(batch_size):
45              indx = indexes[i]
46              if i == 0:
47                  z = mean_gtm[[indx]] + eps[[i]] * torch.exp(
    logvar_gtm[[indx]])
48              else:
49                  z = torch.cat((z, mean_gtm[[indx]] + eps[[i]] *
    torch.exp(logvar_gtm[[indx]])), 0)
50          return z
51
52      def log_prob(self, z):
53          # u->z
54          mean_gtm, logvar_gtm = self.get_params()
55
56          # log-mixture-of-Gaussians
57          z = z.unsqueeze(0) # 1 x B x L
58          mean_gtm = mean_gtm.unsqueeze(1) # K**2 x 1 x L
59          logvar_gtm = logvar_gtm.unsqueeze(1) # K**2 x 1 x L
60
61          w = F.softmax(self.w, dim=0)
62
63          log_p = log_normal_diag(z, mean_gtm, logvar_gtm) + torch.
    log(w) # K**2 x B x L
64          log_prob = torch.logsumexp(log_p, dim=0, keepdim=False) #
     B x L
65
66          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.

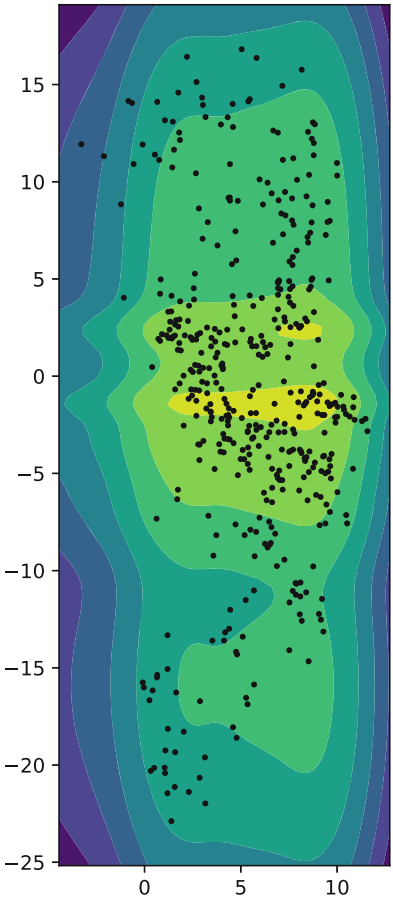


Fig. 4.12 An example of the GTM-VampPrior (contours) and the samples from the aggregated posterior (black dots)

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.

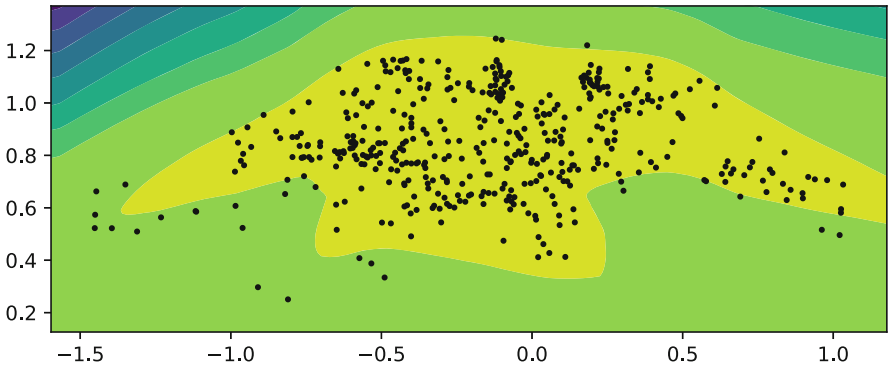


Fig. 4.13 An example of the flow-based prior (contours) and the samples from the aggregated posterior (black dots)

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

```
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

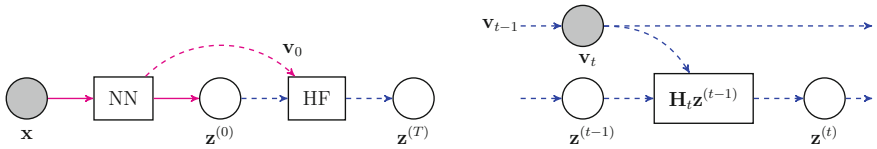
The *Householder transformation* is defined as follows. For a given vector $\mathbf{z}^{(t-1)}$, the reflection hyperplane can be defined by a vector (a *Householder vector*) $\mathbf{v}_t \in \mathbb{R}^M$ that is orthogonal to the hyperplane, and the reflection of this point about the hyperplane is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

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

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

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



(a) encoder network + Householder Flow (b) one step of the Householder Flow

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* $\mathbf{A} \in \mathbb{R}^{D\times M}, \mathbf{B} \in \mathbb{R}^{M\times D}$,

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

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:

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

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 $\mathbf{A}$ and $\mathbf{B}$

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

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

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:

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

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]:

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

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 $\|\mathbf{Q}^{(k)\top}\mathbf{Q}^{(k)} - \mathbf{I}\|_F \leq \epsilon$, 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 $\mathbf{z}^{(0)} \rightarrow \mathbf{z}^{(1)} \rightarrow \ldots \mathbf{z}^{(T)}$, 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:

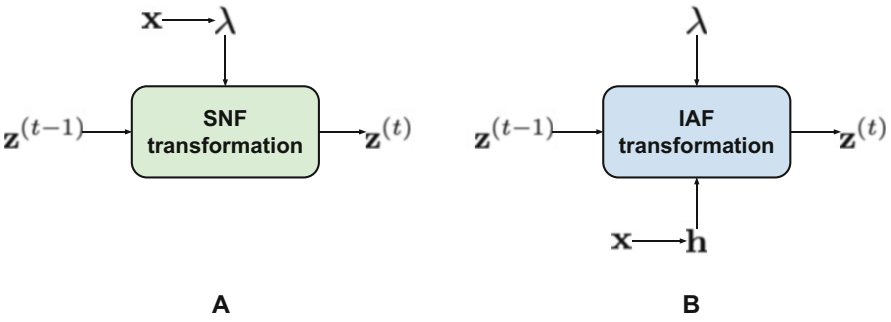


Fig. 4.15 Different amortization strategies for Sylvester normalizing flows and Inverse Autoregressive Flows. (**a**) Our inference network produces amortized flow parameters. This strategy is also employed by planar flows. (**b**) Inverse Autoregressive Flow [18] introduces a measure of **x** dependence 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 **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

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

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

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