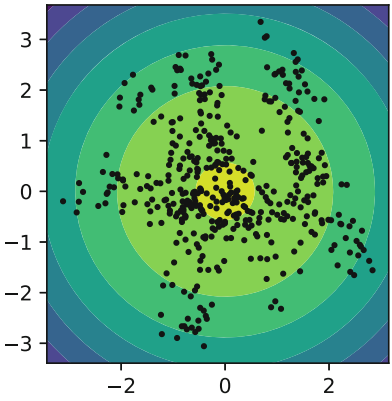

Fig. 4.8 An example of the standard Gaussian prior (contours) and the samples from the aggregated posterior (black dots)

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
```

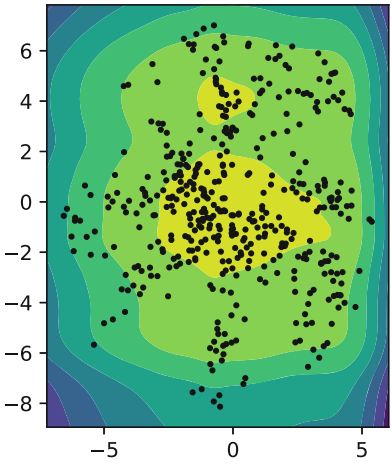

Fig. 4.9 An example of the MoG prior (contours) and the samples from the aggregated posterior (black dots)

```
        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

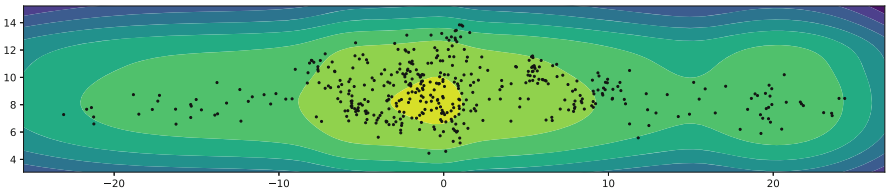

Fig. 4.10 An example of the VampPrior (contours) and the samples from the aggregated posterior (black dots)

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
```