```python

        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:

```python
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
```

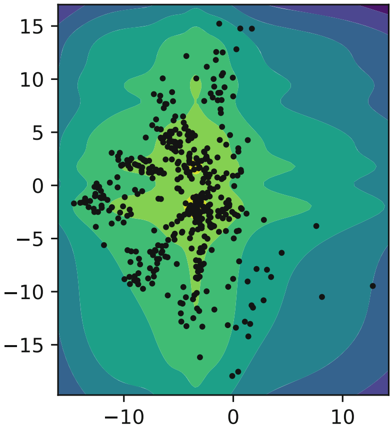

Fig. 4.11 An example of the GTM-based prior (contours) and the samples from the aggregated posterior (black dots)

```
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.

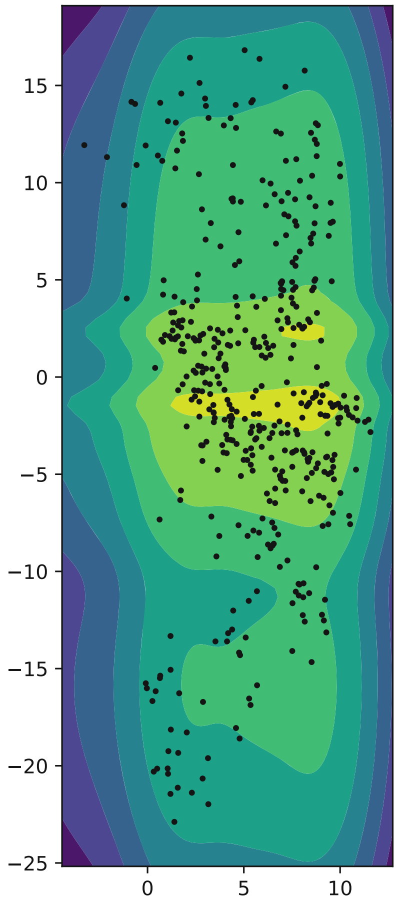

Fig. 4.12 An example of the GTM-VampPrior (contours) and the samples from the aggregated posterior (black dots)

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)
```