```
        # 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.

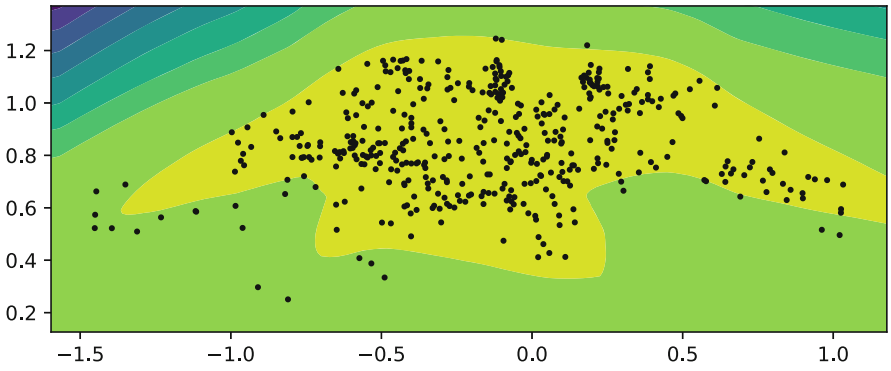

Fig. 4.13 An example of the flow-based prior (contours) and the samples from the aggregated posterior (black dots)

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)

```