Generative Latent Flow

Zhenya Liu

Overview

1. Background
2. Generative Latent Flow
3. Experiment

AE-Based Model

Consider an AE based generative model that can generate samples from the data space $\mathcal{X}$ by the latent space $\mathcal{Z}$. The marginal distribution over $\mathcal{Z}$, denoted by $\tilde{p}(z)$, is unknown and depends on the Encoder E. The model also has a predefined prior distribution $p(z)$, and we can generate new images by sampling from the prior. Thus, in order to generate high quality samples, AE based models need to have

- a good decoder G that can output realistic images given latent variables sampled from $\tilde{p}(z)$
- a good match between $\tilde{p}(z)$ and $p(z)$

AE-Based Model

The first criterion is ensured by minimizing the reconstruction loss of the autoencoders. The second one can be achieved by either modifying the encoder so that $\tilde{p}(z)$ is close to $p(z)$, or conversely modifying $p(z)$ to be close to $\tilde{p}(z)$.

VAE adopts the first approach implicitly by using the approximated posterior $q(z|x)$. Training on ELBO results in minimizing $KL[q(z)\|p(z)]$,
where $q(z) = \mathbb{E}_{x \sim p_{\text{data}}}[q(z \mid x)]$ (aggregated approximate posterior) is used to approximate $\tilde{p}(z)$.

VAEs with flow posterior have been shown to improve neither the matching of $q(z)$ and $p(z)$.

AE-Based Model

One problem of training $q(z)$ to approximate $p(z)$ is that regularing $q(z)$ involves one trade-off on reconstruction loss. However, we do not have such limitation when we try to learn the prior $p(z)$ to approximate $q(z)$.

ALso, normalizing flows can provide great flexibility to construct $p(z)$, and above facts motivate us to introduce the **Generative latent flow** (GLF) model.

Architecture

GLF is an autoencoder model embeded with normalizing flows on the prior distribution.

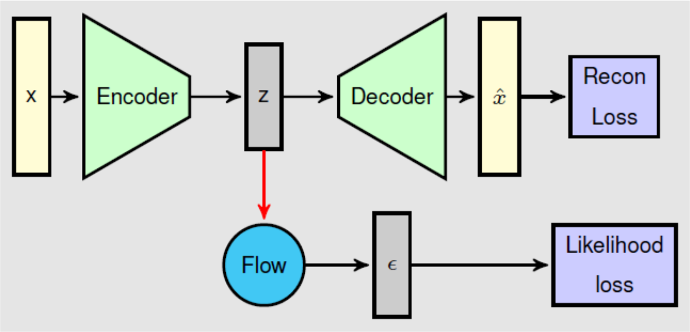


Figure: Archetecture of GLF

Generative Latent Flow

Denote $\mathcal{Z}$ be the unknown target distribution and ϵ be the standard normal distribution. Given a bijection $f_\theta : \mathcal{Z} \to \mathcal{E}$,

$$-\log\left(p_\theta(z)\right) = \mathcal{L}_{\text{NLL}}\left(f_\theta(z)\right) = -\left(\log p_\varepsilon\left(f_\theta(z)\right) + \log\left|\det\left(\frac{\partial f_\theta(z)}{\partial z}\right)\right|\right)$$

Affine Coupling Layer

Similar to **RealNVP**, we use affine coupling layers with alternating pattern to construct the normalizing flow f_θ.

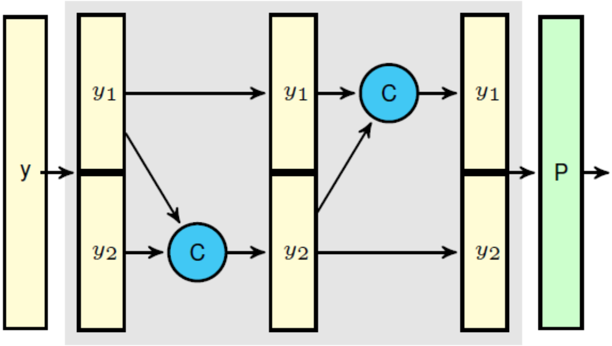


Figure: Affine Coupling Layers

Generative Latent Flow

The loss function fo GLF is composed of the reconstruction loss and NLL loss. Given encoder E_η, decoder G_ϕ, and flow $f_\theta : \mathcal{Z} \to \mathcal{E}$,

$$\mathcal{L}(\eta, \phi, \theta) = \frac{1}{N} \sum_{i=1}^{N} \left(\beta \mathcal{L}_{\text{recon}} \left(\mathrm{x}_i, G_\phi \left(E_\eta \left(\mathrm{x}_i\right)\right)\right) + \mathcal{L}_{\text{NLL}} \left(f_\theta \left(\mathrm{sg}\left[E_\eta \left(\mathrm{x}_i\right)\right]\right)\right)\right)$$

$\mathrm{sg}[\cdot]$ is the stop gradient operation, and β is a hyper-parameter that controls the relative weight of two losses.

Stop Gradient Operation

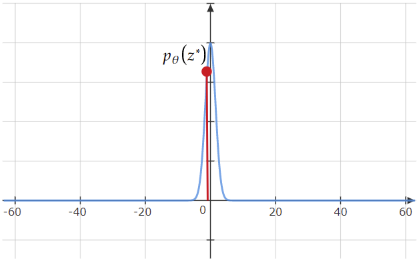
Note that we have $\mathrm{sg}[\cdot]$ term in the loss.

If we let gradients of the NLL loss back propagate into the latent variables, it can lead to degenerate of z into $\mathbf{0}$. That is, the encoder G_η will send all x to a small neighbor of 0.

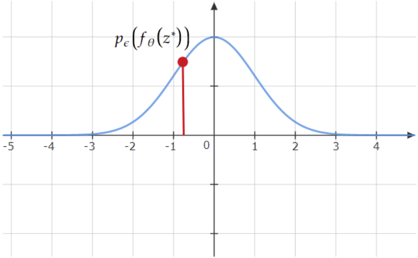
This is because f_θ has to transform the z to unit Gaussian noise, so the smaller the scale of the z's, the more negative the log-determinant of the Jacobian becomes.

We have no control over the scale of z's. This limitation is simliar to the problem of mode collapse in GANs.

Stop Gradient Operation



(a) p_θ



(b) p_ϵ

$$-\log\left(p_\theta(z^*)\right) = -\left(\log p_\varepsilon\left(f_\theta(z^*)\right) + \log\left|\det\left(\frac{\partial f_\theta(z^*)}{\partial z^*}\right)\right|\right)$$

Stop Gradient Operation

While the latent variables cannot become exactly 0 because of the presence of reconstruction loss in the objective, the extremely small scale of z may cause numerical issues that cause severe fluctuations.

We call our original model with stopped gradients GLF and without stopped gradients regularized GLF.

VAEs with Flow Prior

The idea of GLF is closely related to VAEs with normalizing flow priors.

$$\mathrm{ELBO}(\eta, \phi) = \mathbb{E}_{p_{\text{data}}\ (\mathbf{x})} \mathbb{E}_{q_\eta(\mathbf{z}|\mathbf{x})} \left[\log p_{\phi,\beta}(\mathbf{x} \mid \mathbf{z}) + \log p(\mathbf{z}) - \log q_\eta(\mathbf{z} \mid \mathbf{x})\right]$$

By introducing the flow f_θ, $p_\theta(\mathbf{z}) = p_\varepsilon\left(f_\theta(\mathbf{z})\right)\left|\det\left(\frac{\partial f_\theta(\mathbf{z})}{\partial \mathbf{z}}\right)\right|$, ELBO becomes

$$\mathbb{E}_{p_{\text{data}}\ (\mathbf{x})} \mathbb{E}_{q_\eta(\mathbf{z}|\mathbf{x})} \left[\log p_{\phi,\beta}(\mathbf{x} \mid \mathbf{z}) + \log p_\varepsilon\left(f_\theta(\mathbf{z})\right) + \log\left|\det\left(\frac{\partial f_\theta(\mathbf{z})}{\partial \mathbf{z}}\right)\right| - \log q_\eta(\mathbf{z} \mid \mathbf{x})\right]$$

VAEs with Flow Prior

When the expectation over $q_\eta(z \mid x)$ is estimated by sampling, this ELBO is the negative of GLF's objective (without stopping gradients) plus the entropy loss on the encoder $q_\eta(z \mid x)$.

Gaussian VAEs with flow prior does not suffer from the degeneracy of regularized GLF because of the presence of the entropy term. It is the sum of the log variances of the latent variables, and thus it encourages the encoder to output large posterior variance, preventing latent variables from collapsing to 0.

Comparison of VAEs with flow prior on different β

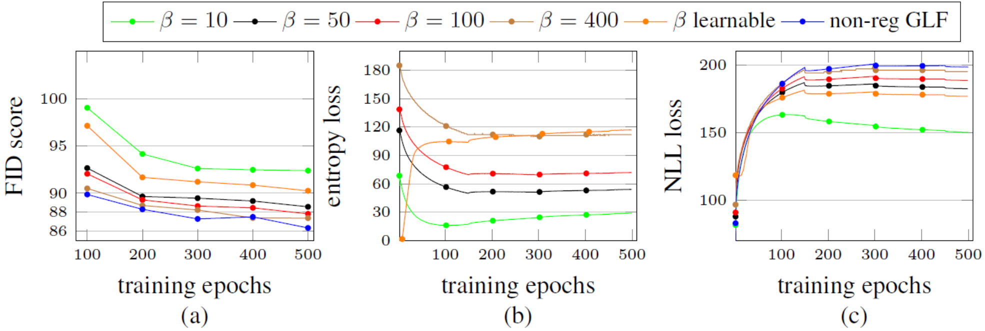


Figure: [1](a) Record of FID scores on CIFAR-10 for VAEs+flow prior with different values of β and GLF. (b) Record of entropy losses for corresponding models. (c) Record of NLL losses for corresponding models

Implementation of GLF

```python
class AffineCoupling(nn.Module):
    def __init__(self, input_dim, hidden_size):
        super().__init__()

        self.net = nn.Sequential(
            nn.Linear(input_dim // 2, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, input_dim)
        )
```

Implementation of GLF

```python
    def forward(self, x):
        z1, z2 = x.chunk(2, 1)
        log_s, t = self.net(z1).chunk(2, 1)
        s = torch.sigmoid(log_s + 2)
        y1 = z1
        y2 = (z2 + t) * s
        y = torch.cat([y1, y2], dim=1)
        logdet = torch.sum(torch.log(s).view(x.shape[0], -1), 1)
        return y, logdet

    def reverse(self, y):
        y1, y2 = y.chunk(2, 1)
        log_s, t = self.net(y1).chunk(2, 1)
        s = torch.sigmoid(log_s + 2)
        z1 = y1
        z2 = y2 / s - t
        x = torch.cat([z1, z2], dim=1)
        return x
```

Implementation of GLF

```
class Permutation(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.input_dim = input_dim
        self.perm = nn.Parameter(torch.randperm(input_dim), requires_grad=False)

    def forward(self, x):
        assert x.shape[1] == self.input_dim
        out = x[:, self.perm]
        return out

    def reverse(self, x):
        assert x.shape[1] == self.input_dim
        out = x[:, torch.argsort(self.perm)]
        return out
```

Implementation of GLF

```python
class FlowNet(nn.Module):
    def __init__(self, input_dim, hidden_size, nblocks=4):
        super().__init__()

        self.nblocks = nblocks

        self.affine_layers = nn.ModuleList([AffineCoupling(input_dim, hidden_size) for _ in range(nblocks)])

        self.perms = nn.ModuleList([Permutation(input_dim) for _ in range(nblocks - 1)])
```

Implementation of GLF

```python
    def forward(self, x):
        out = x
        logdets = 0
        for i in range(self.nblocks - 1):
            out, logdet = self.affine_layers[i](out)
            out = self.perms[i](out)
            logdets += logdets
        out, logdet = self.affine_layers[-1](out)
        logdets = logdets + logdet
        return out, logdets

    def reverse(self, x):
        out = self.affine_layers[-1].reverse(x)
        for i in range(self.nblocks - 1):
            out = self.perms[-1 - i].reverse(out)
            out = self.affine_layers[-2 - i].reverse(out)
        return out
```

Implementation of GLF

```python
class GLF(nn.Module):
    def __init__(self,  input_dim, hidden_dims, latent_dim, n_flows):

        super(GLF, self).__init__()
        self.input_dim = input_dim
        self.hidden_dims = hidden_dims
        self.z_dims = latent_dim
        ## infos about using flows
        self.n_flows = n_flows
        ## we should create the encoder and the decoder
        self.encoder = Encoder(input_dim, hidden_dims, latent_dim)
        self.flows = FlowNet(latent_dim,hidden_dims,n_flows)
        self.decoder = Decoder(latent_dim, hidden_dims, input_dim)
```

Implementation of GLF

```python
    def forward(self, input):
        # we pass the input through the encoder
        z= self.encoder(input)
        # we have to process the z through the flows
        z_nograd = z.detach()

        new_z, logdet = self.flows(z_nograd)
        nll = -(log_standard_gaussian(new_z)+logdet)
        x_reconstruct = self.decoder(z)

        return x_reconstruct,nll

    def sample(self, n_images):
        # in a VAE + normalizing flows, we should start by a random sample from N(0,1)
        # and then we should propagate the z into the flows
        z = torch.randn((n_images, self.z_dims), dtype = torch.float)
        new_z = self.flows.reverse(z)
        samples = self.decoder(new_z)
```

Implementation of GLF

```
BATCH_SIZE = 64
HIDDEN_LAYERS = 200
Z_DIM = 40
N_FLOWS = 4

for i,data in enumerate(train_loader):
        images = data[0]
        reconstruction,nll = model(images)
        likelihood = F.binary_cross_entropy(reconstruction, images,
    reduction='sum')
        total_loss = torch.sum(likelihood) + torch.sum(nll)
        L = total_loss / len(images)
        L.backward()
        optimizer.step()
        optimizer.zero_grad()
```

Result

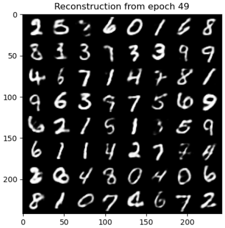
VAE:Epoch: 50, **Elbo**: -101.5485916015625, **recon_error**: 70.64453887939453, **kl**: 30.852636337280273

VAE_Posterior Flow: Epoch: 50, **Elbo**: -102.53454217122396, **recon_error**: 78.20411682128906, **kl**: 24.275081634521484

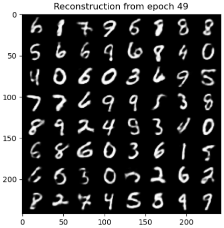
VAE_Prior Flow($\beta = 10$): Epoch: 19, **Elbo**: -78.6134489827474, **recon_error**: 62.59688186645508, **kl**: 16.016551971435547

GLF:Epoch: 15, **Total_loss**: 178.63508473307292, **recon_error**: 61.97129440307617, **NLL**: 116.56587982177734

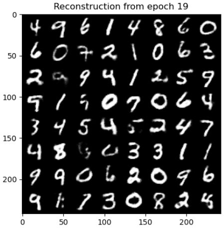
Reconstruction



(a) VAE

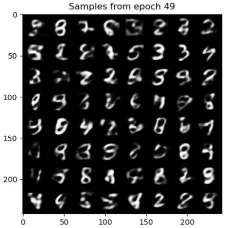


(b) VAE_Posterior

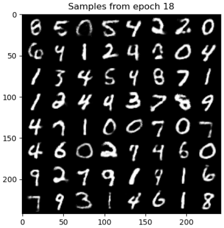


(c) VAE_PriorFlow

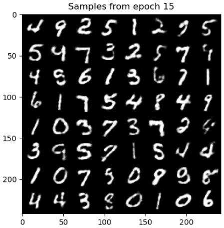
Sampling



(d) VAE_Posterior



(e) VAE_Prior



(f) GLF

Experiment

Now we present some results from one recurrent implementation of GLF by Ivan Fursov et al.

`https://github.com/rakhimovv/GenerativeLatentFlow`

Experiment

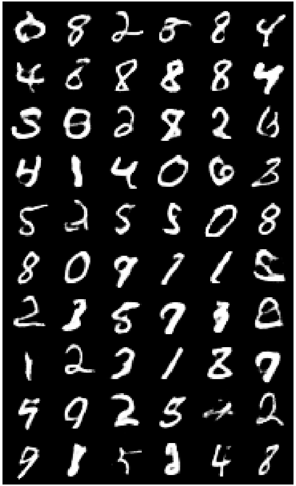


Figure: MNIST Reconstruction after 200 epoch from GLF

We use the GLF model which was trained on CelebA dataset for 40 epochs. The dimensionality of the hidden space is 64.

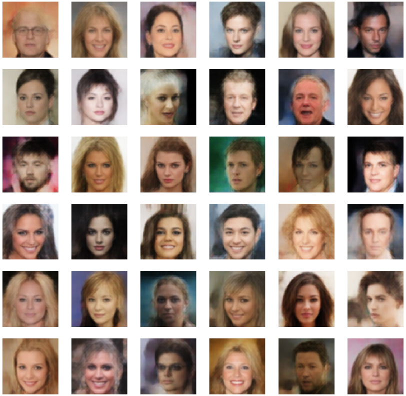


Figure: Random Samples from $\mathcal{N}(0, \mathbf{I})$

Figure: Random Samples from $\mathcal{N}(0, \mathbf{I})$ without processing normalizing flows

Figure: Interpolation of Two Random Samples

References

Xiao, Z., Yan, Q. & Amit, Y. Generative latent flow. *ArXiv Preprint ArXiv:1905.10485*. (2019)