机器学习编程课笔记

2025 年 7 月 31 日

1 变分自编码器（VAE）

VAE 原理简介

VAE 通过将输入映射为潜在空间的分布参数（均值和方差），再通过 **重参数化技巧**从中采样潜变量，最后再映射回原始空间。VAE 的目标是最大化对观测数据 x 的边际似然 $p_\theta(x)$，但由于 $p_\theta(x) = \int p_\theta(x|z)p(z)\,dz$ 不可直接计算，因此引入变分推断，通过构造一个可学习的近似分布 $q_\phi(z|x)$ 来优化证据下界（ELBO）：

$$\begin{aligned}
\log p_\theta(x) &= \log \int p_\theta(x|z)p(z)\,dz \\
&= \log \int \frac{q_\phi(z|x)}{q_\phi(z|x)} p_\theta(x|z)p(z)\,dz \\
&= \log \mathbb{E}_{q_\phi(z|x)}\left[\frac{p_\theta(x|z)p(z)}{q_\phi(z|x)}\right] \\
&\geq \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\mathrm{KL}}(q_\phi(z|x) \parallel p(z)) \\
&=: \mathcal{L}_{\mathrm{ELBO}}(\theta, \phi; x)
\end{aligned}$$

其中第一项 $\mathbb{E}_{q(z|x)}[\log p(x|z)]$ 是重建项，鼓励模型在重构输入时尽量接近；第二项是 KL 散度，约束潜在分布靠近标准正态先验。

KL 散度具体形式 假设 $q_\phi(z|x) = \mathcal{N}(\mu, \mathrm{diag}(\sigma^2))$ 且 $p(z) = \mathcal{N}(0, I)$，那么二者之间的 KL 散度具有如下解析表达式：

$$D_{\mathrm{KL}}(\mathcal{N}(\mu, \sigma^2) \| \mathcal{N}(0, 1)) = \frac{1}{2} \sum_{i=1}^{d} \left(\mu_i^2 + \sigma_i^2 - \log \sigma_i^2 - 1\right)$$

为便于数值稳定与反向传播，实际实现中通常以对数方差 $\log \sigma^2$ 作为参数化形式。对应代码中的实现：

$$\sigma = \exp\left(\frac{1}{2} \log \sigma^2\right) \quad \Rightarrow \quad z = \mu + \sigma \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

最终目标函数 整合上述两部分损失后，VAE 的训练目标可表达为：

$$\mathcal{L}(x) = \underbrace{\|x - \hat{x}\|_2^2}_{\text{重建误差}} + \beta \cdot \underbrace{D_{\mathrm{KL}}(q(z|x) \parallel p(z))}_{\text{先验约束}}$$

其中 β 是一个可调节的超参数（在 β-VAE 中尤为重要），用于控制生成质量与潜在空间可解释性之间的权衡。

模型结构解析：VanillaVAE

Encoder 编码器结构

Listing 1: Encoder 构建

```
modules = []
for h_dim in hidden_dims:
    modules.append(
        nn.Sequential(
            nn.Conv2d(in_channels, out_channels=h_dim,
                      kernel_size= 3, stride= 2, padding = 1),
            nn.BatchNorm2d(h_dim),
            nn.LeakyReLU())
    )
self.encoder = nn.Sequential(*modules)
self.fc_mu = nn.Linear(hidden_dims[-1]*4, latent_dim)
self.fc_var = nn.Linear(hidden_dims[-1]*4, latent_dim)
```

编码器由多个卷积层堆叠而成，每层通道数逐渐增加（如 32, 64, 128...），每层包含卷积、批归一化和 LeakyReLU 激活。最终通过两个线性层分别输出均值 μ 和对数方差 $\log \sigma^2$，用于后续的潜变量采样。

Reparameterization Trick

Listing 2: 重参数化采样

```
std = torch.exp(0.5 * logvar)
eps = torch.randn_like(std)
return eps * std + mu
```

由于无法直接对随机变量反向传播，引入重参数化技巧，将采样过程转换为：

$$z = \mu + \sigma \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

从而实现端到端可导的采样。

Decoder 解码器结构

Listing 3: Decoder 构建

```
self.decoder_input = nn.Linear(latent_dim, hidden_dims[-1] * 4)
...
self.final_layer = nn.Sequential(
    nn.ConvTranspose2d(...),
    ...
    nn.Conv2d(hidden_dims[-1], out_channels=3, kernel_size=3, padding=1),
    nn.Tanh())
```

解码器先将潜变量 z 映射回卷积特征空间，再通过多层转置卷积网络逐步上采样至原图大小，最终输出通道为 3，并使用 Tanh 激活保证像素值范围在 $[-1, 1]$ 内。

完整前向传播流程

Listing 4: VAE 前向过程

```
mu, log_var = self.encode(input)
z = self.reparameterize(mu, log_var)
return [self.decode(z), input, mu, log_var]
```

在前向过程中，输入图像首先通过卷积堆叠的编码器提取高维特征：

```
result = self.encoder(input)
result = torch.flatten(result, start_dim=1)
```

假设输入图像的尺寸为 $[N, C, H, W] = [N, 3, 64, 64]$，编码器网络共包含 5 层卷积，每层使用 kernel size $= 3$、stride $= 2$、padding $= 1$，因此每经过一层空间尺寸均减半（向下取整），输出通道依次为 `[32, 64, 128, 256, 512]`。

因此，输出特征图的形状依次为：

$$
\begin{aligned}
&[N, 32, 32, 32] \rightarrow \\
&[N, 64, 16, 16] \rightarrow \\
&[N, 128, 8, 8] \rightarrow \\
&[N, 256, 4, 4] \rightarrow \\
&[N, 512, 2, 2]
\end{aligned}
$$

最终输出的张量为 $[N, 512, 2, 2]$，在展平后（即 `start_dim=1`）变为二维张量 $[N, 2048]$，其中 $2048 = 512 \times 2 \times 2$。

均值与方差输出 展平后通过两组线性层分别生成潜在高斯分布的均值与对数方差向量：

```
mu = self.fc_mu(result)
log_var = self.fc_var(result)
```

这两组全连接层的输出形状均为 $[N, D]$，其中 $D =$ `latent_dim`，为超参数设定的潜在空间维度，常见值如 16、32 或 128 等。

因此，encode 函数最终返回两个张量 $[\mu, \log \sigma^2]$，分别代表潜在变量的均值和对数方差，用于后续采样。整体流程为：输入图像 $\rightarrow$ 编码得到 $\mu, \log \sigma^2$ $\rightarrow$ 重参数化采样 z $\rightarrow$ 解码重建 $\hat{x}$。

损失函数定义

Listing 5: 损失函数

```
recons_loss = F.mse_loss(recons, input)
kld_loss = torch.mean(-0.5 * torch.sum(1 + log_var - mu ** 2 - log_var.exp(), dim=1))
loss = recons_loss + kld_weight * kld_loss
```

重建误差使用 MSE 损失计算像素级差距，KL 散度使用解析形式计算，并乘以批次归一化因子 `kld_weight` 控制平衡。

最终返回：

$$\mathcal{L} = \text{MSE}(x, \hat{x}) + \beta \cdot D_{\text{KL}}(q(z|x) \| p(z))$$

2 生成对抗网络（GAN）

GAN 原理简介

GAN 核心思想是通过博弈论中“生成器 vs 判别器”的对抗机制，实现对数据分布的建模与样本生成。

GAN 包含两个神经网络：

- **生成器（Generator）** $G(z)$：将随机噪声 $z \sim p_z(z)$ 映射为接近真实数据分布的样本。
- **判别器（Discriminator）** $D(x)$：判断输入 x 是否来源于真实数据而非生成器。

GAN 的训练目标可表示为一个极小极大值问题：

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))]$$

训练过程中，D 尽量区分真实图像与伪造图像，而 G 尽量生成能骗过 D 的图像，二者相互博弈，最终达到纳什均衡。

生成器模块：Generator

Listing 6: Generator 结构

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        def block(in_feat, out_feat, normalize=True):
            layers = [nn.Linear(in_feat, out_feat)]
            if normalize:
                layers.append(nn.BatchNorm1d(out_feat, 0.8))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers

        self.model = nn.Sequential(
            *block(opt.latent_dim, 128, normalize=False),
            *block(128, 256),
            *block(256, 512),
            *block(512, 1024),
            nn.Linear(1024, int(np.prod(img_shape))),
            nn.Tanh()
        )
```

生成器从低维高斯噪声向量 $z \in \mathbb{R}^d$ 开始，通过逐层全连接网络不断升维，最终输出大小与图像一致的一维向量（后 reshape 为图像）。中间层采用 `LeakyReLU` 激活，并使用 `BatchNorm` 加速训练，最后一层使用 `Tanh` 映射至 $[-1, 1]$ 区间，对应图像像素的归一化。

```
def forward(self, z):
    img = self.model(z)
    img = img.view(img.size(0), *img_shape)
    return img
```

判别器模块：Discriminator

Listing 7: Discriminator 结构

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(int(np.prod(img_shape)), 512),
            nn.LeakyReLU(0.2, inplace=True),
```

```
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )
```

判别器的输入是图像张量展平后的向量，经过多层全连接层与 LeakyReLU 激活，最终输出一个标量，表示输入图像为真实图像的概率。输出通过 Sigmoid 函数约束在 $[0, 1]$ 区间，便于与真实标签计算交叉熵损失。

```
def forward(self, img):
    img_flat = img.view(img.size(0), -1)
    validity = self.model(img_flat)
    return validity
```

训练过程

在训练过程中，Generator 和 Discriminator 交替优化，各自目标函数如下：

生成器的目标 生成器希望生成的图像能骗过判别器，因此它的损失是希望判别器认为“伪造图”为真实图：

$$\mathcal{L}_G = \mathbb{E}_{z \sim p_z}[\log D(G(z))]$$

```
optimizer_G.zero_grad()
z = Variable(Tensor(np.random.normal(0, 1, (imgs.shape[0], opt.latent_dim))))
gen_imgs = generator(z)
g_loss = adversarial_loss(discriminator(gen_imgs), valid)
g_loss.backward()
optimizer_G.step()
```

判别器的目标 判别器需分别识别真实图像和生成图像，其损失由两部分组成：

$$\mathcal{L}_D = -\mathbb{E}_{x \sim p_{\text{data}}}[\log D(x)] - \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))]$$

```
optimizer_D.zero_grad()
real_loss = adversarial_loss(discriminator(real_imgs), valid)
fake_loss = adversarial_loss(discriminator(gen_imgs.detach()), fake)
d_loss = (real_loss + fake_loss) / 2
d_loss.backward()
optimizer_D.step()
```

在每一个训练 iteration 中，判别器和生成器交替优化，使得 G 学会生成逼真的图像，D 学会分辨真伪，最终达到对抗平衡。