《机器学习编程实践》课程——DAY 4

课程内容

- **VAE 代码实现**
- **GAN 代码实现**

一、 VAE 的实现

1 基本结构

VAE 由**编码器 (Encoder)** 和**解码器 (Decoder)** 组成：

- **编码器** $q_\phi(z|x)$：将输入 x 映射到潜在空间 z，具体实现：
  - 输入层：$x \in \mathbb{R}^n$（如图像展平向量）
  - 隐藏层：两层 MLP，ReLU 激活
  - 输出层：并行输出 $\mu \in \mathbb{R}^d$ 和 $\log \sigma^2 \in \mathbb{R}^d$
- **潜在空间**
- **解码器** $p_\theta(x|z)$：从潜在变量 z 重构数据，具体实现：
  - 输入层：采样 $z = \mu + \sigma \odot \epsilon$, $\epsilon \sim \mathcal{N}(0, I)$
  - 隐藏层：两层 MLP，ReLU 激活
  - 输出层：Sigmoid 激活（针对像素值归一化）

2 重参数化技巧

通过噪声 $\epsilon \sim \mathcal{N}(0, I)$ 生成潜在变量：

$$z = \mu + \sigma \odot \epsilon \tag{1}$$

使得梯度可以通过随机节点反向传播。

3 代码实现

3.1 整体架构

```
class VanillaVAE(BaseVAE):
  def __init__(self,
       in_channels: int,  # 输入图像的通道数
       latent_dim: int,   # 潜在空间的维度
       hidden_dims: List = None, # 编码器和解码器中各层的通道数列表
       **kwargs) -> None:

    super(VanillaVAE, self).__init__()
    self.latent_dim = latent_dim

    modules = []
    if hidden_dims is None:
      hidden_dims = [32, 64, 128, 256, 512] #
          如果没有提供潜在空间维度，则使用默认维度

    # 构建编码器的卷积层，每层包括卷积层、BatchNorm层、激活函数
    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())
      )
      in_channels = h_dim

    self.encoder = nn.Sequential(*modules) # 将所有编码器层组合成一个序列

    # 定义两个全连接层，分别输出潜在分布的均值(mu)和方差的对数(log_var)
    self.fc_mu = nn.Linear(hidden_dims[-1]*4, latent_dim)
    self.fc_var = nn.Linear(hidden_dims[-1]*4, latent_dim)

    # 构建解码器
    modules = []

    # 解码器的输入层，将潜在变量映射回特征空间
    self.decoder_input = nn.Linear(latent_dim, hidden_dims[-1] * 4)

    # 反转 hidden_dims 的顺序，用于解码器
```

```
    hidden_dims.reverse()

    # 构建编码器的转置卷积层，每层包括转置卷积层、BatchNorm层、激活函数
    for i in range(len(hidden_dims) - 1):
        modules.append(
            nn.Sequential(
                nn.ConvTranspose2d(hidden_dims[i],
                                   hidden_dims[i + 1],
                                   kernel_size=3,
                                   stride = 2,
                                   padding=1,
                                   output_padding=1),
            nn.BatchNorm2d(hidden_dims[i + 1]),
            nn.LeakyReLU())
        )

    # 将所有解码器层组合成一个序列
    self.decoder = nn.Sequential(*modules)

    # 最终输出层，将特征映射回图像空间，并使用 Tanh 激活函数将输出限制在 [-1,
        1] 范围内
    self.final_layer = nn.Sequential(
                nn.ConvTranspose2d(hidden_dims[-1],
                                   hidden_dims[-1],
                                   kernel_size=3,
                                   stride=2,
                                   padding=1,
                                   output_padding=1),
                nn.BatchNorm2d(hidden_dims[-1]),
                nn.LeakyReLU(),
                nn.Conv2d(hidden_dims[-1], out_channels= 3,
                kernel_size= 3, padding= 1),
                nn.Tanh())

# 将输入图像编码为潜在空间的均值和方差对数
def encode(self, input: Tensor) -> List[Tensor]:
    # 中间省略
    return [mu, log_var]

# 将潜在变量解码回图像空间
```

```
    def decode(self, z: Tensor) -> Tensor:
        # 中间省略
        return result

    # 实现重参数化技巧，使得梯度可以通过随机采样过程反向传播
    def reparameterize(self, mu: Tensor, logvar: Tensor) -> Tensor:
        #中间省略
        return eps * std + mu

    # 完整的前向传播过程，返回解码结果、原始输入、均值和方差对数
    def forward(self, input: Tensor, **kwargs) -> List[Tensor]:
        # 中间省略
        return [self.decode(z), input, mu, log_var]

    # 计算VAE的损失函数
    def loss_function(self, *args, **kwargs) -> dict:
        # 中间省略
        return {'loss': loss, 'Reconstruction_Loss':recons_loss.detach(),
            'KLD':-kld_loss.detach()}

    # 从潜在空间采样并生成新图像
    def sample(self, num_samples:int, current_device: int, **kwargs) ->
        Tensor:
        # 中间省略
        return samples

    # 给定输入图像，返回其重建结果
    def generate(self, x: Tensor, **kwargs) -> Tensor:
        return self.forward(x)[0]
```

3.2 encode 方法

```
    def encode(self, input: Tensor) -> List[Tensor]:
        result = self.encoder(input)
        result = torch.flatten(result, start_dim=1)

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

```
        return [mu, log_var]
```

3.3 decode 方法

```
    def decode(self, z: Tensor) -> Tensor:
        result = self.decoder_input(z)
        result = result.view(-1, 512, 2, 2)
        result = self.decoder(result)
        result = self.final_layer(result)
        return result
```

3.4 重参数方法

```
    def reparameterize(self, mu: Tensor, logvar: Tensor) -> Tensor:
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return eps * std + mu
```

3.5 forward 方法

```
    def forward(self, input: Tensor, **kwargs) -> List[Tensor]:
        mu, log_var = self.encode(input)
        z = self.reparameterize(mu, log_var)
        return [self.decode(z), input, mu, log_var]
```

3.6 loss_function 方法

```
    # 包括重建损失 (MSE)和KL散度损失 (衡量潜在分布与标准正态分布的差异)
    def loss_function(self, *args, **kwargs) -> dict:
        recons = args[0]
        input = args[1]
        mu = args[2]
        log_var = args[3]

        kld_weight = kwargs['M_N'] # Account for the minibatch samples from
            the dataset
        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), dim = 0)

        loss = recons_loss + kld_weight * kld_loss
        return {'loss': loss, 'Reconstruction_Loss':recons_loss.detach(),
            'KLD':-kld_loss.detach()}
```

3.7 sample 方法

```
    def sample(self, num_samples:int, current_device: int, **kwargs) ->
        Tensor:
        z = torch.randn(num_samples, self.latent_dim)
        z = z.to(current_device)
        samples = self.decode(z)
        return samples
```

3.8 generate 方法

```
    def generate(self, x: Tensor, **kwargs) -> Tensor:
        return self.forward(x)[0]
```

二、 GAN 的实现

1 基本框架

GAN 由**生成器** G 和**判别器** D 组成：

- **生成器** $G(z)$：将噪声 z 映射到数据空间
- **判别器** $D(x)$：判断 x 来自真实数据还是生成器

2 目标函数

二者**极小极大博弈**：

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))] \tag{2}$$

3 训练过程

- 固定 G，更新 D 以最大化区分真实/生成样本
- 固定 D，更新 G 以最小化 $\log(1 - D(G(z)))$

4 代码实现

4.1 生成器

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.label_emb = nn.Embedding(opt.n_classes, opt.n_classes)

        # 定义构建块函数，包含线性层、可选BatchNorm层和LeakyReLU激活
        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

        # 生成器的主要结构
        self.model = nn.Sequential(
            # 输入：噪声(100维) + 类别嵌入(10维) = 110维
            # 4个全连接块，维度逐渐增加
            *block(opt.latent_dim + opt.n_classes, 128, normalize=False),
            *block(128, 256),
            *block(256, 512),
            *block(512, 1024),
            # 使用Tanh激活将输出压缩到[-1,1]范围
            nn.Linear(1024, int(np.prod(img_shape))),
            nn.Tanh()
        )

    # 前向传播
    def forward(self, noise, labels):
        gen_input = torch.cat((self.label_emb(labels), noise), -1) #
            将标签嵌入和噪声拼接
        img = self.model(gen_input) # 通过模型生成图像
```

```
        img = img.view(img.size(0), *img_shape) # 调整形状为(批量大小, 通道,
            高, 宽)
        return img
```

4.2 判别器

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.label_embedding = nn.Embedding(opt.n_classes, opt.n_classes)

        # 判别器的主要结构
        self.model = nn.Sequential(
            # 输入: 展平的图像(32x32=1024) + 类别嵌入(10) = 1034维
            # 3个全连接层, 使用Dropout防止过拟合
            nn.Linear(opt.n_classes + int(np.prod(img_shape)), 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 512),
            nn.Dropout(0.4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 512),
            nn.Dropout(0.4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 1),
        )

    # 前向传播
    def forward(self, img, labels):
        # 拼接图像和标签嵌入
        d_in = torch.cat((img.view(img.size(0), -1),
            self.label_embedding(labels)), -1)
        # 通过模型输出真假判断
        validity = self.model(d_in)
        return validity
```

4.3 训练过程

```
for epoch in range(opt.n_epochs): # 循环遍历训练周期
```

```
for i, (imgs, labels) in enumerate(dataloader): # 循环遍历数据批次

batch_size = imgs.shape[0]

# 定义真实和假标签
valid = Variable(FloatTensor(batch_size, 1).fill_(1.0),
    requires_grad=False)
fake = Variable(FloatTensor(batch_size, 1).fill_(0.0),
    requires_grad=False)

real_imgs = Variable(imgs.type(FloatTensor))
labels = Variable(labels.type(LongTensor))

# 训练生成器
optimizer_G.zero_grad() # 清空梯度

# 生成随机噪声和随机标签
z = Variable(FloatTensor(np.random.normal(0, 1, (batch_size,
    opt.latent_dim))))
gen_labels = Variable(LongTensor(np.random.randint(0, opt.n_classes,
    batch_size)))

# 生成图像
gen_imgs = generator(z, gen_labels)

# Loss measures generator's ability to fool the discriminator
validity = discriminator(gen_imgs, gen_labels)
g_loss = adversarial_loss(validity, valid)

g_loss.backward()
optimizer_G.step()

# 训练判别器
optimizer_D.zero_grad()

# 计算真实图像损失
validity_real = discriminator(real_imgs, labels)
d_real_loss = adversarial_loss(validity_real, valid)

# 计算生成图像损失
```

```
        validity_fake = discriminator(gen_imgs.detach(), gen_labels)
        d_fake_loss = adversarial_loss(validity_fake, fake)

        # 得到总损失
        d_loss = (d_real_loss + d_fake_loss) / 2

        d_loss.backward()
        optimizer_D.step()

        print(
        "[Epoch %d/%d] [Batch %d/%d] [D loss: %f] [G loss: %f]"
        % (epoch, opt.n_epochs, i, len(dataloader), d_loss.item(),
            g_loss.item())
        )

        batches_done = epoch * len(dataloader) + i

        # 定期保存生成的样本图像
        if batches_done % opt.sample_interval == 0:
           sample_image(n_row=10, batches_done=batches_done)
```