

《模式识别与机器学习 A》实验报告

实验题目： 基于生成模型的图像生成

班级： 2203601

学号： 2022113416

姓名： 刘子康

实验报告内容

一、实验目的

- 掌握生成式神经网络的基本原理与结构，掌握搭建和训练生成式神经网络的方法；
- 采用任意一种课程中介绍过的或者其它生成式神经网络模型（如变分自编码器等）用于解决图像生成问题，并加深对生成模型的认识；
- 理解不同激活函数、dropout 比例、数据量和超参数对模型性能的影响。

二、实验内容

- 参照变分自编码器模型，基于现有框架 Pytorch 构建一个生成式神经网络，实现数据样本的分类和预测。
- 选择 MNIST 数据集进行训练和测试，该数据集为手写体数字图像标准数据集，包含 60000 个训练样本和 10000 个测试样本，每个样本为单通道 28*28 像素灰度图像。
- 尝试选择不同激活函数，使用 dropout 等技巧，分析实验结果和可能原因。
- 使用不同数据量，不同超参数（如学习率和批次大小），比较实验效果，并给出截图和分析。

三、实验环境

- 操作系统：Windows 11
- 编程语言：Python 3.10
- 第三方库：PyTorch 2.4.0+cu118, torchvision0.19.0, Numpy 1.23.4, Matplotlib 3.8.2
- IDE：Pycharm 2022 社区版

四、实验过程、结果及分析

（包括代码截图、运行结果截图及必要的理论支撑等）

4.1 实验原理

生成式神经网络可以分为两个主要步骤：

- 训练：假设 x 表示样本，模型从训练集中学习得到的样本数据分布为 $p_{model}(x)$ ，以尽可能接近训练数据中的真实分布 $p_{data}(x)$ ；
- 生成：从 $p_{model}(x)$ 中通过采样生成新的样本。

自编码器是一类在半监督学习和非监督学习中使用的神经网络，其功能是通过将输入信息作为学习目标，对输入信息进行表征学习。自编码器通常分为两个步骤：

- 编码器(encoder) ϕ : $x \rightarrow z$ 。即将输入样本 x 转换为压缩的隐含特征表示 z ；
- 解码器(decoder) ψ : $z \rightarrow x'$ 。即将隐含的特征表示 z 重构为样本 x' ， x' 的维度和 x 的维度相同，以使 x' 尽量与原始输入样本 x 相同。

变分自编码器（Variational auto-encoder, VAE）继承了传统自动编码器的架构，并使用它来学习数据生成分布，这允许我们从潜在空间中随机抽取样本，然后使用解码器网络对这些随机样本进行解码，以生成具有与训练网络的特征类似的特征的独特图像。在传统自编码器中，编码器的输出是一个固定的潜在向量，而 VAE 引入概率分布的思想，使得潜在空间的表示不再是一个确定的点，而是一个概率分布。

变分自编码器采用一种“重采样”的技巧，即利用标准正态分布来完成采样过程： $z = \mu + \varepsilon \times \sigma$ ，从而得到解码器的输入向量 z 。该公式在反向传播时可以对均值、方差求偏

导，这里的 ε 是从标准正态分布 $N(0, 1)$ 中采样得到的随机数向量。

4.2 实验过程

4.2.1 数据集预处理

如图 4-1，使用 Numpy 库的 `random.choice` 随机选择样本索引，使用 `torch.utils.data.dataset.Subset` 创建子数据集，以便设置不同的训练集数据量。

```
47 # 随机选择样本
   单元测试 | 注释生成 | 代码解释 | 缺陷检测
48 def create_subset(data, sample_size):
49     indices = np.random.choice(len(data), sample_size, replace=False) # 随机选择样本索引
50     return Subset(data, indices)
```

图 4-1 随机选取样本

如图 4-2，使用 Torchvision 库的 `torchvision.datasets.MNIST` 下载 MNIST 数据集，并通过 `torchvision.transform` 模块进行将图像转换为 Tensor 格式、展平为一维张量等预处理，以便符合 VAE 的输入格式。使用 PyTorch 库的 `torch.utils.data.DataLoader` 作为数据集加载器，并设置批次大小。

```
92 transform = transforms.Compose([
93     transforms.ToTensor(),
94     transforms.Lambda(lambda x: x.view(-1))
95 ])
96
97 # 获取MNIST数据集
98 sample_sizes = [10000, 30000, 60000]
99 train_dataset = datasets.MNIST(root="./data", train=True, download=True, transform=transform)
100 test_dataset = datasets.MNIST(root="./data", train=False, download=True, transform=transform)
101
102 # 加载数据集
103 batch_sizes = [64, 128, 256, 512, 1024]
104 train_loader = DataLoader(dataset=create_subset(train_dataset, sample_sizes[2]), batch_size=batch_sizes[2], shuffle=True)
105 test_loader = DataLoader(dataset=test_dataset, batch_size=32, shuffle=False)
```

图 4-2 获取和加载 MNIST 数据集

4.2.2 VAE 模型搭建

定义一个 VAE 类，以 `torch.nn.Module` 作为基类。

(1) 初始化：接收参数为激活函数和 dropout 比例，按照 VAE 模型的网络结构依次定义潜在空间维度、编码器和解码器的各个层。

```
14 class VAE(nn.Module):
   单元测试 | 注释生成 | 代码解释 | 缺陷检测
15     def __init__(self, latent_dim=20, activation=nn.ReLU()):
16         super(VAE, self).__init__()
17         self.latent_dim = latent_dim # 潜在空间维度
18         self.activation = activation
19
20         # 编码器
21         self.fc1 = nn.Linear(28 * 28, 400)
22         self.fc2_mu = nn.Linear(400, latent_dim)
23         self.fc2_logvar = nn.Linear(400, latent_dim)
24
25         # 解码器
26         self.fc3 = nn.Linear(latent_dim, 400)
27         self.fc4 = nn.Linear(400, 28 * 28)
```

图 4-3 VAE 模型初始化

(2) 编码操作：将输入数据通过全连接层 `fc1` 的线性变换和激活函数，映射到隐藏层 `h1` ($28 \times 28 = 784$ 维降至 400 维)，并进一步经全连接层 `fc2` 计算出潜在空间的中间特征表示 `z` 的分布均值和方差（对数形式）。

```

29      # 编码
      单元测试 | 注释生成 | 代码解释 | 缺陷检测
30      def encode(self, x):
31          h1 = self.activation(self.fc1(x))
32          return self.fc2_mu(h1), self.fc2_logvar(h1)

```

图 4-4 编码操作

(3) 重参数化操作：计算标准差 ($std = e^{0.5 \cdot \log(var)} = \sqrt{var}$)，并使用 `torch.randn_like()` 对其添加高斯噪声，返回重参数化后的样本数据分布。

```

34      # 重参数化
      单元测试 | 注释生成 | 代码解释 | 缺陷检测
35      def reparameterize(self, mu, logvar):
36          std = torch.exp(0.5 * logvar) # 标准差
37          eps = torch.randn_like(std)   # 高斯噪声
38          return mu + eps * std

```

图 4-5 重参数化操作

(4) 解码操作：将编码的潜在表示 z 通过全连接层 `fc3` 和激活函数，映射到隐藏层 `h3`，并进一步经全连接层 `fc4` 的线性变换和 Sigmoid 函数，映射回原始输入空间（400 维升至 $28 \times 28 = 784$ 维），生成新样本。

```

40      # 解码
      单元测试 | 注释生成 | 代码解释 | 缺陷检测
41      def decode(self, z):
42          h3 = self.activation(self.fc3(z))
43          return torch.sigmoid(self.fc4(h3))

```

图 4-6 解码操作

(5) 前向传播：按照编码→重采样→解码的顺序，迭代一轮并输出结果。

```

      单元测试 | 注释生成 | 代码解释 | 缺陷检测
45      def forward(self, x):
46          mu, logvar = self.encode(x)
47          z = self.reparameterize(mu, logvar)
48          return self.decode(z), mu, logvar

```

图 4-6 前向传播

4.2.3 模型训练

如图 4-7，分批次进行训练，首先初始化优化器，然后前向传播一次并保存训练结果，最后计算 loss 并反向传播更新参数。每 50 个批次输出一一次训练进度和 loss。

```

60      def train(model, device, train_loader, optimizer):
61          model.train()
62          train_loss = 0
63          for batch_idx, (data, _) in enumerate(train_loader):
64              data = data.to(device)
65              optimizer.zero_grad() # 梯度初始化
66              recon_batch, mu, logvar = model(data) # 前向传播
67              loss = loss_function(recon_batch, data, mu, logvar) # 计算损失
68              loss.backward() # 反向传播
69              train_loss += loss.item()
70              optimizer.step() # 更新参数
71              if batch_idx % 50 == 0:
72                  print('Train Epoch: {} [{}/{}] ({:.0f}%) \t Loss: {:.6f}'.format(
73                      epoch, batch_idx * len(data), len(train_loader.dataset),
74                      100. * batch_idx / len(train_loader), loss.item() / len(data)))
75
76          train_loss /= len(train_loader.dataset)
77          print(f'====> Epoch: {epoch} Average loss: {train_loss:.4f}')
78          return train_loss

```

图 4-7 模型训练

VAE 的损失函数包括重建误差和 KL 散度。如图 4-8，重建误差用于度量生成数据与

输入数据的相似度（这里使用交叉熵损失函数），KL 散度用于度量 z 的近似后验分布与先验分布的相似度。

```
55 def loss_function(recon_x, x, mu, logvar):
56     BCE = F.binary_cross_entropy(recon_x, x, reduction='sum')
57     KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
58     return BCE + KLD
59
```

图 4-8 损失函数

4.2.4 生成图像并绘制曲线

如图 4-9，每训练 10 个 epoch，从设定好的先验标准正态分布中采样，利用解码器生成 n 个新样本图像并保存。训练全部完成后，绘制训练过程的平均 loss 值变化曲线。

```
80 # 生成图像
81 # 单元测试 | 注释生成 | 代码解释 | 缺陷检测
82 def generate_images(model, device, n_images=16, latent_dim=20, param=None, value=None):
83     model.eval()
84     with torch.no_grad():
85         # 从标准正态分布中采样
86         z = torch.randn(n_images, latent_dim, device=device)
87         sample = model.decode(z).cpu()
88         save_image(sample.view(n_images, 1, 28, 28), f'image_{param}_{value}.png')
```

图 4-9 生成图像

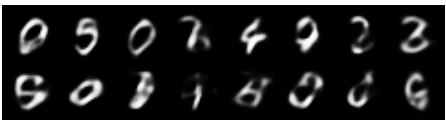
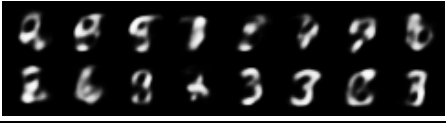
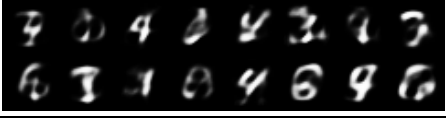
4.3 实验结果及分析

4.3.1 不同激活函数

分别设置激活函数为 ReLU 函数、Sigmoid 函数、Tanh 函数，训练过程 loss 变化如图 4-10 所示，不同激活函数生成图像如表 4-1 所示，根据生成图像清晰和准确程度可以看出最佳激活函数为 ReLU。

Sigmoid 函数优化稳定，但指数运算的计算复杂度较高，且在深层网络中易出现梯度消失的问题，因此适用于较为简单的网络结构；Tanh 函数相比 Sigmoid 函数优点在于均值为 0，不会对梯度产生影响，但仍存在梯度饱和与指数计算的问题；ReLU 函数收敛速度更快，计算简单，且不会出现梯度饱和或消失的问题，但可能导致“神经元坏死”。

表 4-1 不同激活函数生成图像

激活函数	生成图像
ReLU 函数	
Sigmoid 函数	
Tanh 函数	

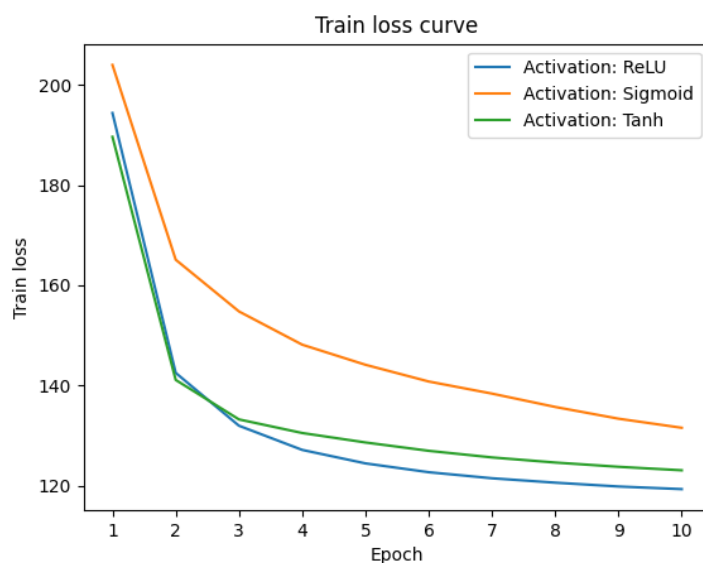


图 4-10 不同激活函数训练过程

4.3.2 不同 dropout 比例

分别设置 dropout 比例为 0、0.1、0.2、0.3、0.4、0.5，训练过程 loss 变化如图 4-11 所示，不同 dropout 比例生成图像如表 4-2 所示，根据生成图像清晰和准确程度可以看到不使用 dropout 技巧时效果最佳。

dropout 技巧主要用于防止模型过拟合，而对应该数据集分类任务，当 dropout 比例逐渐增大时，loss 值逐渐增大。猜测可能是模型并未出现明显过拟合现象，较高的 dropout 比例反而影响了模型的特征学习。

表 4-2 不同 dropout 比例生成图像

dropout 比例	生成图像
0	
0.1	
0.2	
0.3	
0.4	
0.5	

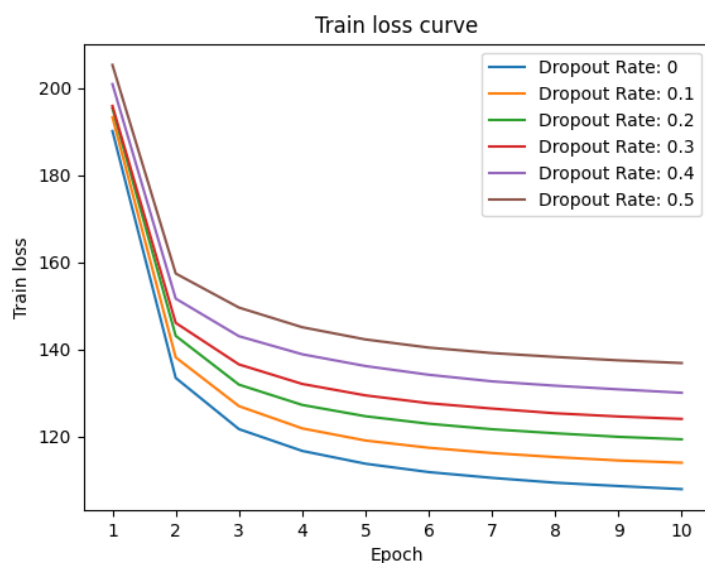


图 4-11 不同 dropout 比例训练过程

4.3.3 不同数据量

分别设置训练集数据量为 10000、30000、60000，训练过程 loss 变化如图 4-12 所示，不同数据量生成图像如表 4-3 所示，根据生成图像清晰和准确程度可以看到最佳数据量为 60000。

在模型提取特征能力足够的前提下，更多的数据样本可以帮助模型更好地学习数据分布和多样化的特征，减少过拟合的风险，并提高模型的泛化能力。

表 4-3 不同数据量生成图像

数据量	生成图像
10000	
30000	
60000	

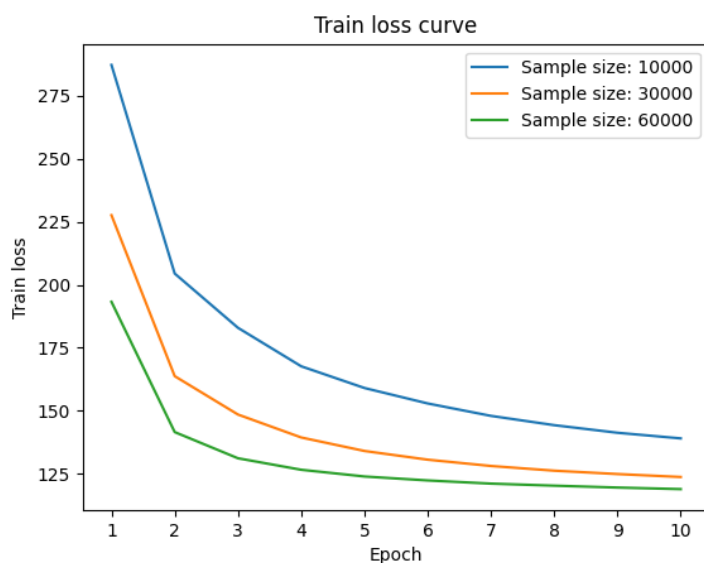


图 4-12 不同数据量训练过程

4.3.4 不同学习率

分别设置学习率为 $1e-5$ 、 $1e-4$ 、 $1e-3$ 、 $1e-2$ 、 $1e-1$ ，训练过程 loss 变化如图 4-13 所示，不同学习率生成图像如表 4-4 所示，根据生成图像清晰和准确程度可以看到最佳学习率为 $1e-2$ 。

当学习率较低时（如 $1e-5$ ），模型收敛速度很慢，不能很好地学习到数据特征，故 loss 值整体较高，且 epoch=0 时尚未收敛；当学习率为 $1e-4 \sim 1e-2$ 时，随 epoch 增加，loss 值先是迅速下降，之后保持平稳，模型收敛；当学习率为 $1e-1$ 时，同样先是迅速下降，之后保持平稳，但初始和最终 loss 值都很大，可能是学习率过大导致参数更新波动较大，模型无法完全收敛。

表 4-4 不同学习率生成图像

学习率	生成图像
$1e-5$	
$1e-4$	
$1e-3$	
$1e-2$	
$1e-1$	

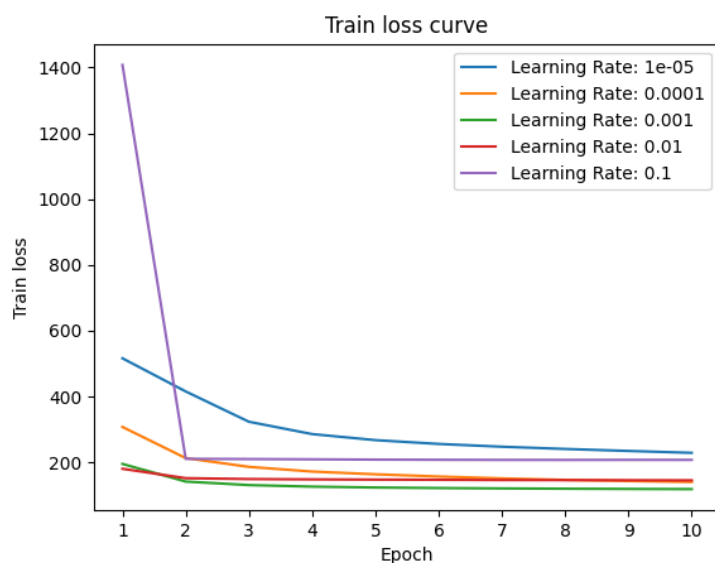


图 4-13 不同学习率训练过程

4.3.5 不同批次大小

分别设置批次大小为 64、128、256、512、1024，训练过程 loss 变化如图 4-14 所示，不同批次大小生成图像如表 4-5 所示，根据生成图像清晰和准确程度可以看到最佳批次大小为 128 或 256。

较小的 Batch Size 可以加快每轮训练的速度，更好地拟合复杂的数据分布，提高模型精度，且使得训练更加随机化，有助于跳出局部极小值，从而提高最终模型的泛化能力，但由于每次更新都是基于少量样本，也存在着梯度波动较大，收敛速度变慢的问题。

较大的 Batch Size 可以充分利用现代 GPU 的强大并行计算能力，加速整体训练过程，且可以获得更稳定的梯度估计，优化过程更加直接地朝向全局极值前进，收敛速度更快，但过大可能会导致陷入局部极小值，影响最终的模型性能。

表 4-5 不同批次大小生成图像

批次大小	生成图像
64	
128	
256	
512	
1024	

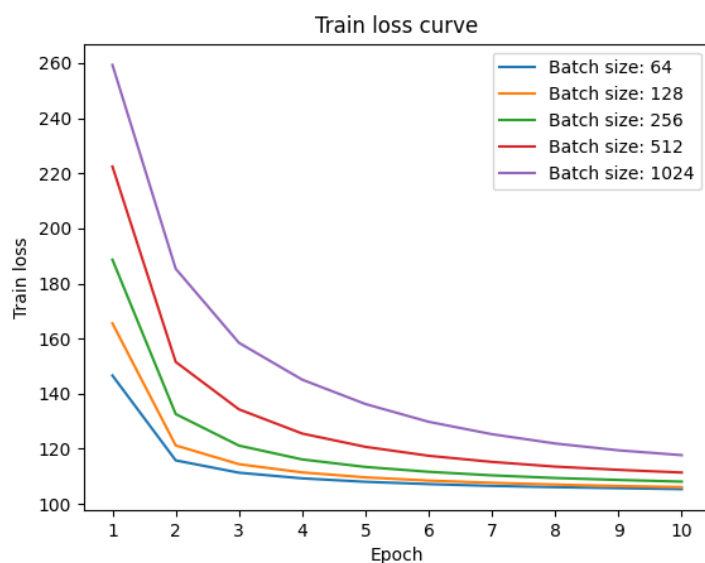


图 4-14 不同批次大小训练过程

五、实验总体结论

此次实验参照 VAE 模型，基于 PyTorch 搭建了一个生成式神经网络，并实现了 MNIST 数据集的训练和图像生成，数据集的训练和测试，达到了较好的性能和效果。并且使用不同激活函数、dropout 比例、数据量和超参数进行训练，分析了其对模型性能的影响。

由实验结果可以得到，当激活函数为 ReLU，数据量为 60000，学习率为 $1e-2$ ，批次大小为 128 或 256 时，模型达到最佳性能。

常见的生成式神经网络模型包括两大类，即变分自编码器（VAE）和生成式对抗网络（GAN）。VAE 通过编码器和解码器的训练过程，学习数据的概率分布，能够进行变分推断，但也存在生成的样本质量较低、训练过程较慢的缺点；而 GAN 通过生成器和判别器的竞争过程进行训练，生成与真实数据相似的样本，且生成的样本质量较高，缺点是训练过程不稳定，容易出现模型崩溃。

六、完整实验代码

```

1. import torch
2. import torch.nn as nn
3. import torch.optim as optim
4. import torch.nn.functional as F
5. from torch.utils.data import DataLoader
6. from torch.utils.data.dataset import Subset
7. from torchvision import datasets, transforms
8. from torchvision.utils import save_image
9. import numpy as np
10. import matplotlib.pyplot as plt
11. import time
12.
13.
14. class VAE(nn.Module):

```

```

15.     def __init__(self, latent_dim=20, activation=nn.ReLU(), p=0.1):
16.         super(VAE, self).__init__()
17.         self.latent_dim = latent_dim    # 潜在空间维度
18.         self.activation = activation
19.         self.dropout = nn.Dropout(p)
20.
21.         # 编码器
22.         self.fc1 = nn.Linear(28 * 28, 400)
23.         self.fc2_mu = nn.Linear(400, latent_dim)
24.         self.fc2_logvar = nn.Linear(400, latent_dim)
25.
26.         # 解码器
27.         self.fc3 = nn.Linear(latent_dim, 400)
28.         self.fc4 = nn.Linear(400, 28 * 28)
29.
30.         # 编码
31.         def encode(self, x):
32.             h1 = self.activation(self.fc1(x))
33.             h1 = self.dropout(h1)
34.             return self.fc2_mu(h1), self.fc2_logvar(h1)
35.
36.         # 重参数化
37.         def reparameterize(self, mu, logvar):
38.             std = torch.exp(0.5 * logvar)    # 标准差
39.             eps = torch.randn_like(std)     # 高斯噪声
40.             return mu + eps * std
41.
42.         # 解码
43.         def decode(self, z):
44.             h3 = self.activation(self.fc3(z))
45.             h3 = self.dropout(h3)
46.             return torch.sigmoid(self.fc4(h3))
47.
48.         def forward(self, x):
49.             mu, logvar = self.encode(x)
50.             z = self.reparameterize(mu, logvar)
51.             return self.decode(z), mu, logvar
52.
53. # 随机选择样本
54. def create_subset(data, sample_size):
55.     indices = np.random.choice(len(data), sample_size, replace=False)    # 随机选择样本索引
56.     return Subset(data, indices)
57.
58. def loss_function(recon_x, x, mu, logvar):

```

```

59.     BCE = F.binary_cross_entropy(recon_x, x, reduction='sum')
60.     KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
61.     return BCE + KLD
62.
63. def train(model, device, train_loader, optimizer, epoch):
64.     model.train()
65.     train_loss = 0
66.     for batch_idx, (data, _) in enumerate(train_loader):
67.         data = data.to(device)
68.         optimizer.zero_grad() # 梯度初始化
69.         recon_batch, mu, logvar = model(data) # 前向传播
70.         loss = loss_function(recon_batch, data, mu, logvar) # 计算损失
71.         loss.backward() # 反向传播
72.         train_loss += loss.item()
73.         optimizer.step() # 更新参数
74.         if batch_idx % 50 == 0:
75.             print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.4f}'.format(
76.                 epoch, batch_idx * len(data), len(train_loader.dataset),
77.                 100. * batch_idx / len(train_loader), loss.item() / len(data)))
78.
79.     train_loss /= len(train_loader.dataset)
80.     print(f'====> Epoch: {epoch} Average loss: {train_loss:.4f}')
81.     return train_loss
82.
83. # 生成图像
84. def generate_images(model, device, n_images=16, latent_dim=20, param=None, value=None):
85.     model.eval()
86.     with torch.no_grad():
87.         # 从标准正态分布中采样
88.         z = torch.randn(n_images, latent_dim, device=device)
89.         sample = model.decode(z).cpu()
90.         save_image(sample.view(n_images, 1, 28, 28), f'image_{param, value}.png')
91.
92. if __name__ == "__main__":
93.     epoch = 10
94.     Loss = []
95.     transform = transforms.Compose([
96.         transforms.ToTensor(),
97.         transforms.Lambda(lambda x: x.view(-1))
98.     ])
99.
100. # 获取MNIST 数据集
101. sample_sizes = [10000, 30000, 60000]

```

```

102.     train_dataset = datasets.MNIST(root="./data", train=True, download=True, transform=
        transform)
103.
104.     # 加载数据集
105.     batch_sizes = [64, 128, 256, 512, 1024]
106.     train_loader = DataLoader(dataset=create_subset(train_dataset, sample_sizes[2]), ba
        tch_size=batch_sizes[1], shuffle=True)
107.
108.     # 部署GPU, 创建VAE 模型实例
109.     activation = [nn.ReLU(), nn.Sigmoid(), nn.Tanh()]
110.     dropout_p = [0, 0.1, 0.2, 0.3, 0.4, 0.5] # 不同dropout 比例
111.     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
112.     model = VAE(activation=activation[0], p=dropout_p[0]).to(device)
113.
114.     # 创建优化器
115.     lrs = [1e-5, 1e-4, 1e-3, 0.01, 0.1] # 不同学习率
116.     optimizer = optim.Adam(model.parameters(), lr=lrs[3])
117.
118.     # 模型训练
119.     train_losses = []
120.     print(f"Start Time: {time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())}")
121.     for i in range(epoch):
122.         train_loss = train(model, device, train_loader, optimizer, i)
123.         train_losses.append(train_loss)
124.
125.     # 生成图像
126.     generate_images(model, device, n_images=16, param='sample')
127.     print(f"End Time: {time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())}\n")
128.     Loss.append(train_losses)
129.
130.     # 绘制训练损失曲线
131.     x_ticks = np.arange(1, epoch + 1)
132.     plt.plot(x_ticks, Loss[0])
133.     plt.xticks(x_ticks)
134.     plt.xlabel('Epoch')
135.     plt.ylabel('Train loss')
136.     plt.title('Train loss curve')
137.     # plt.legend()
138.     plt.show()

```

七、参考文献

[1] 刘远超. 深度学习基础: 高等教育出版社, 2023.