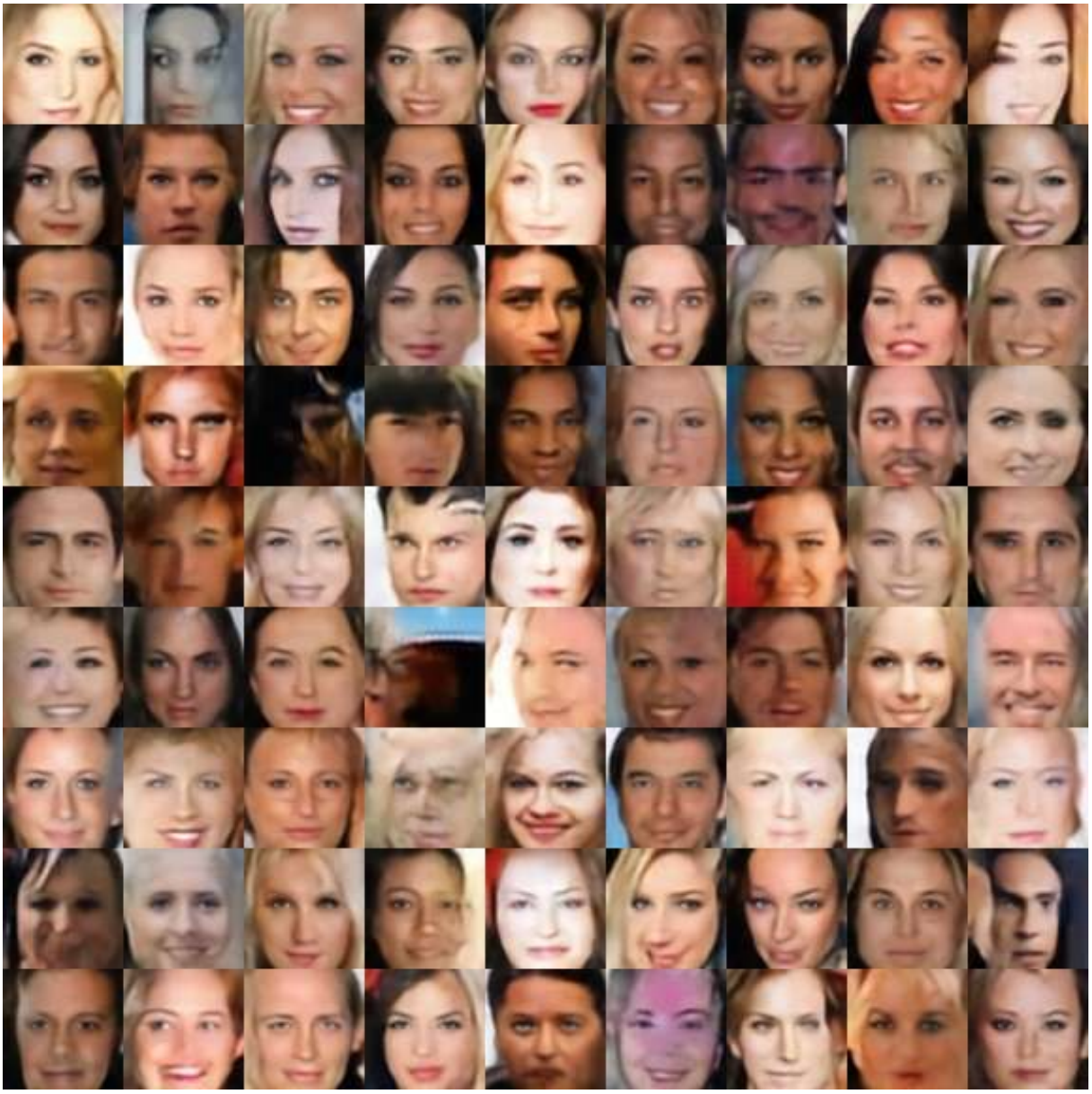
VQVAE PyTorch 实现教程

目录



前段时间我写了一篇VQVAE的解读，现在再补充一篇VQVAE的PyTorch实现教程。在这个项目中，我们会实现VQVAE论文，在MNIST和CelebAHQ两个数据集上完成图像生成。具体来说，我们会先实现并训练一个图像压缩网络VQVAE，它能把真实图像编码成压缩图像，或者把压缩图像解码回真实图像。之后，我们会训练一个生成压缩图像的生成网络PixelCNN。

代码仓库：https://github.com/SingleZombie/DL-Demos/tree/master/dldemos/VQVAE

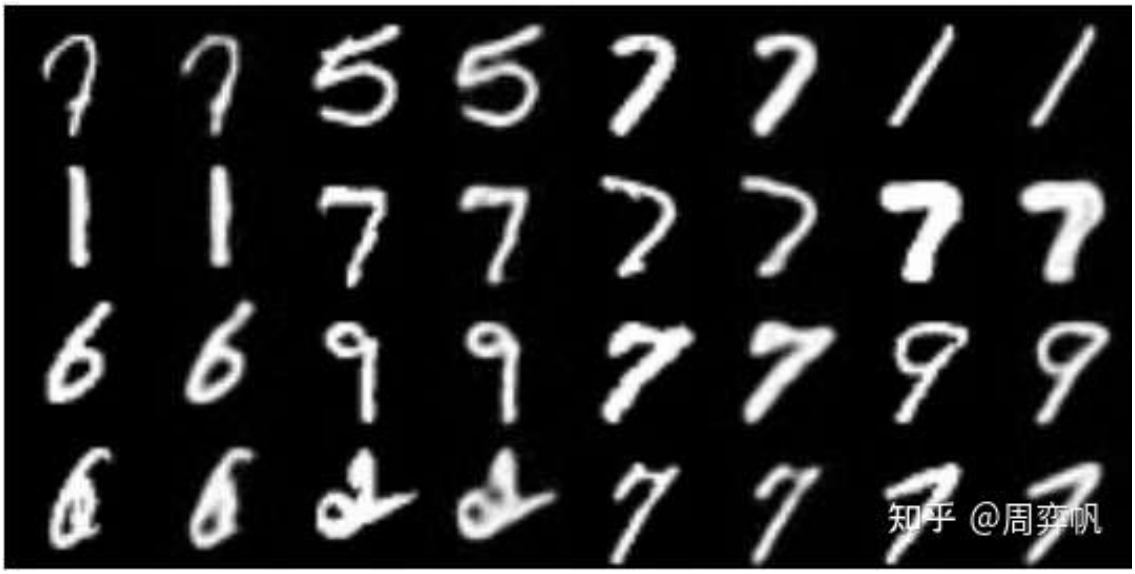
项目运行示例

如果你只是想快速地把项目运行起来，可以只阅读本节。

在本地安装好项目后，运行`python dldemos/VQVAE/dataset.py`来下载MNIST数据集。之后运行`python dldemos/VQVAE/main.py`，这个脚本会完成以下四个任务：

1. 训练VQVAE
2. 用VQVAE重建数据集里的随机数据
3. 训练PixelCNN
4. 用PixelCNN+VQVAE随机生成图片

第二步得到的重建结果大致如下（每对图片中左图是原图，右图是重建结果）：



第四步得到的随机生成结果大致如下：

如果你要使用CelebAHQ数据集，请照着下一节的指示把CelebAHQ下载到指定目录，再执行`python dldemos/VQVAE/main.py -c 4`。

数据集准备

MNIST数据集可以用PyTorch的API自动下载。我们可以用下面的代码下载MNIST数据集并查看数据的格式。从输出中可知，MNIST的图片形状为`[1, 28, 28]`，颜色取值范围为`[0, 1]`。

```
def download_mnist():
    mnist =
torchvision.datasets.MNIST(root='./data/mnist',
download=True)
    print('length of MNIST', len(mnist))
    id = 4
    img, label = mnist[id]
    print(img)
    print(label)

    # On computer with monitor
    # img.show()

    img.save('work_dirs/tmp_mnist.jpg')
    tensor = transforms.ToTensor()(img)
```

```
    print(tensor.shape)
    print(tensor.max())
    print(tensor.min())
```

我们可以用下面的代码把它封成简单的`Dataset`。

```
class MNISTImageDataset(Dataset):

    def __init__(self, img_shape=(28, 28)):
        super().__init__()
        self.img_shape = img_shape
        self.mnist =
torchvision.datasets.MNIST(root='./data/mnist')

    def __len__(self):
        return len(self.mnist)

    def __getitem__(self, index: int):
        img = self.mnist[index][0]
        pipeline = transforms.Compose(
            [transforms.Resize(self.img_shape),
             transforms.ToTensor()])
        return pipeline(img)
```

接下来准备CelebAHQ。CelebAHQ数据集原本的图像大小是1024x1024，但我们这个项目用不到这么大的图片。我在kaggle上找到了一个256x256的CelebAHQ (https://www.kaggle.com/datasets/badasstechie/celebahq-resized-256x256)，所有文件加起来只有300MB左右，很适合我们项目。请在该页面下载压缩包，并把压缩包解压到项目的`data/celebA/celeba_hq_256`目录下。

下载完数据后，我们可以写一个简单的从目录中读取图片的`Dataset`类。和MNIST的预处理流程不同，我这里给CelebAHQ的图片加了一个中心裁剪的操作，一来可以让人脸占比更大，便于模型学习，二来可以让该类兼容CelebA数据集（CelebA数据集的图片不是正方形，需要裁剪）。这个操作是可选的。

```
class CelebADataset(Dataset):

    def __init__(self, root, img_shape=(64, 64)):
        super().__init__()
        self.root = root
        self.img_shape = img_shape
        self.filenames = sorted(os.listdir(root))

    def __len__(self) -> int:
        return len(self.filenames)
```

```
    def __getitem__(self, index: int):
        path = os.path.join(self.root,
self.filenames[index])
        img = Image.open(path)
        pipeline = transforms.Compose([
            transforms.CenterCrop(168),
            transforms.Resize(self.img_shape),
            transforms.ToTensor()
        ])
        return pipeline(img)
```

有了数据集类后，我们可以用它们生成`Dataloader`。

```
CELEBA_DIR = 'data/celebA/img_align_celeba'
CELEBA_HQ_DIR = 'data/celebA/celeba_hq_256'
def get_dataloader(type,
                   batch_size,
                   img_shape=None,
                   dist_train=False,
                   num_workers=4,
                   **kwargs):
    if type == 'CelebA':
        if img_shape is not None:
            kwargs['img_shape'] = img_shape
        dataset = CelebADataset(CELEBA_DIR,
**kwargs)
```

```
    elif type == 'CelebAHQ':
        if img_shape is not None:
            kwargs['img_shape'] = img_shape
        dataset = CelebADataset(CELEBA_HQ_DIR,
**kwargs)
    elif type == 'MNIST':
        if img_shape is not None:
            dataset = MNISTImageDataset(img_shape)
        else:
            dataset = MNISTImageDataset()
    if dist_train:
        sampler = DistributedSampler(dataset)
        dataloader = DataLoader(dataset,

batch_size=batch_size,
                                sampler=sampler,

num_workers=num_workers)
        return dataloader, sampler
    else:
        dataloader = DataLoader(dataset,

batch_size=batch_size,
                                shuffle=True,

num_workers=num_workers)
```

```
        return dataloader
```

我们可以利用`Dataloader`来查看CelebAHQ数据集的内容及数据格式。

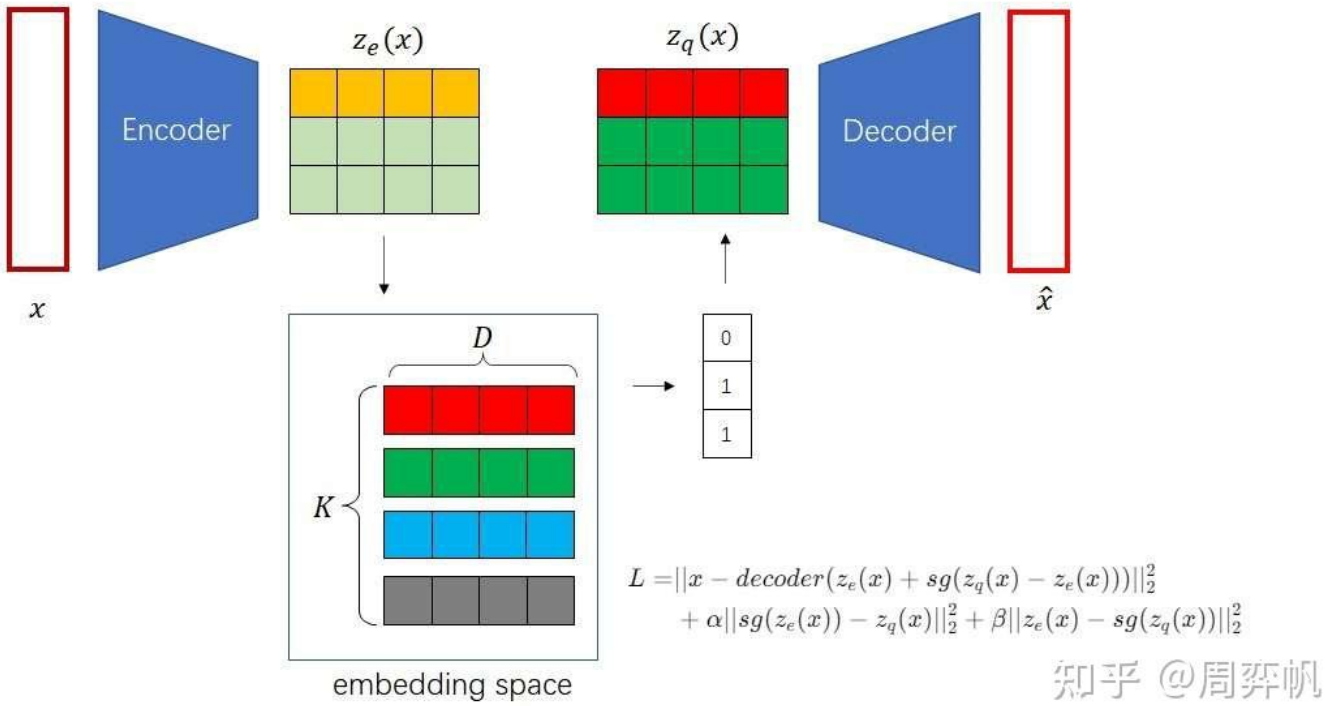
```
if os.path.exists(CELEBA_HQ_DIR):
    dataloader = get_dataloader('CelebAHQ', 16)
    img = next(iter(dataloader))
    print(img.shape)
    N = img.shape[0]
    img = einops.rearrange(img,
                           '(n1 n2) c h w -> c
(n1 h) (n2 w)',
                           n1=int(N**0.5))
    print(img.shape)
    print(img.max())
    print(img.min())
    img = transforms.ToPILImage()(img)
    img.save('work_dirs/tmp_celebahq.jpg')
```

从输出中可知，CelebAHQ的颜色取值范围同样是`[0, 1]`。经我们的预处理流水线得到的图片如下。

实现并训练 VQVAE

要用VQVAE做图像生成，其实要训练两个模型：一个是用于压缩图像的VQVAE，另一个是生成压缩图像的PixelCNN。这两个模型是可以分开训练的。我们先来实现并训练VQVAE。

VQVAE的架构非常简单：一个编码器，一个解码器，外加中间一个嵌入层。损失函数为图像的重建误差与编码器输出与其对应嵌入之间的误差。



VQVAE的编码器和解码器的结构也很简单，仅由普通的上/下采样层和残差块组成。具体来说，编码器先是有两个3x3卷积+2倍下采样卷积的模块，再有两个残差块(ReLU, 3x3卷积, ReLU, 1x1卷积)；解码器则反过来，先有两个残差块，再有两个3x3卷积+2倍上采样反卷积的模块。为了让代码看起来更清楚一点，我们不用过度封装，仅实现一个残差块模块，再用残差块和PyTorch自带模块拼成VQVAE。

先实现残差块。注意，由于模型比较简单，残差块内部和VQVAE其他地方都可以不使用BatchNorm。

```
class ResidualBlock(nn.Module):

    def __init__(self, dim):
```

```
        super().__init__()
        self.relu = nn.ReLU()
        self.conv1 = nn.Conv2d(dim, dim, 3, 1, 1)
        self.conv2 = nn.Conv2d(dim, dim, 1)

    def forward(self, x):
        tmp = self.relu(x)
        tmp = self.conv1(tmp)
        tmp = self.relu(tmp)
        tmp = self.conv2(tmp)
        return x + tmp
```

有了残差块类后，我们可以直接实现VQVAE类。我们先在初始化函数里把模块按顺序搭好。编码器和解码器的结构按前文的描述搭起来即可。嵌入空间(codebook)其实就是个普通的嵌入层。此处我仿照他人代码给嵌入层显式初始化参数，但实测下来和默认的初始化参数方式差别不大。

```
class VQVAE(nn.Module):

    def __init__(self, input_dim, dim,
n_embedding):
        super().__init__()
        self.encoder =
nn.Sequential(nn.Conv2d(input_dim, dim, 4, 2, 1),
```

```
                                        nn.ReLU(),
nn.Conv2d(dim, dim, 4, 2, 1),
                                        nn.ReLU(),
nn.Conv2d(dim, dim, 3, 1, 1),

ResidualBlock(dim), ResidualBlock(dim))
        self.vq_embedding =
nn.Embedding(n_embedding, dim)

self.vq_embedding.weight.data.uniform_(-1.0 /
n_embedding,
                                               1.0
/ n_embedding)
        self.decoder = nn.Sequential(
            nn.Conv2d(dim, dim, 3, 1, 1),
            ResidualBlock(dim),
ResidualBlock(dim),
            nn.ConvTranspose2d(dim, dim, 4, 2, 1),
nn.ReLU(),
            nn.ConvTranspose2d(dim, input_dim, 4,
2, 1))
        self.n_downsample = 2
```

之后，我们来实现模型的前向传播。这里的逻辑就略显复杂了。整体来看，这个函数完成了编码、取最近邻、解码这三步。其中，取最近邻的部分最为复杂。

```
def forward(self, x):
    # encode
    ze = self.encoder(x)

    # ze: [N, C, H, W]
    # embedding [K, C]
    embedding = self.vq_embedding.weight.data
    N, C, H, W = ze.shape
    K, _ = embedding.shape
    embedding_broadcast = embedding.reshape(1, K,
C, 1, 1)
    ze_broadcast = ze.reshape(N, 1, C, H, W)
    distance = torch.sum((embedding_broadcast -
ze_broadcast)**2, 2)
    nearest_neighbor = torch.argmin(distance, 1)
    # make C to the second dim
    zq =
self.vq_embedding(nearest_neighbor).permute(0, 3,
1, 2)
    # stop gradient
    decoder_input = ze + (zq - ze).detach()

    # decode
    x_hat = self.decoder(decoder_input)
    return x_hat, ze, zq
```

我们来详细看一看取最近邻的实现。取最近邻时，我们要用到两块数据：编码器输出`ze`与嵌入矩阵`embedding`。`ze`可以看成一个形状为`[N, H, W]`的数组，数组存储了长度为`C`的向量。而嵌入矩阵里有`K`个长度为`C`的向量。

```
# ze: [N, C, H, W]
# embedding [K, C]
embedding = self.vq_embedding.weight.data
N, C, H, W = ze.shape
K, _ = embedding.shape
```

为了求`N*H*W`个向量在嵌入矩阵里的最近邻，我们要先算这每个向量与嵌入矩阵里`K`个向量的距离。在算距离前，我们要把`embedding`和`ze`的形状变换一下，保证`(embedding_broadcast - ze_broadcast)**2`的形状为`[N, K, C, H, W]`。我们对这个临时结果的第2号维度（`C`所在维度）求和，得到形状为`[N, K, H, W]`的`distance`。它的含义是，对于`N*H*W`个向量，每个向量到嵌入空间里`K`个向量的距离分别是多少。

```
embedding_broadcast = embedding.reshape(1, K, C, 1, 1)
ze_broadcast = ze.reshape(N, 1, C, H, W)
distance = torch.sum((embedding_broadcast - ze_broadcast)**2, 2)
```

有了距离张量后，我们再对其1号维度（`K`所在维度）求最近邻所在下标。

```
nearest_neighbor = torch.argmin(distance, 1)
```

有了下标后，我们可以用`self.vq_embedding(nearest_neighbor)`从嵌入空间取出最近邻了。别忘了，`nearest_neighbor`的形状是`[N, H, W]`，`self.vq_embedding(nearest_neighbor)`的形状会是`[N, H, W, C]`。我们还要把`C`维度转置一下。

```
# make C to the second dim
zq =
self.vq_embedding(nearest_neighbor).permute(0, 3,
1, 2)
```

最后，我们用论文里提到的停止梯度算子，把`zq`变形一下。这样，算误差的时候用的是`zq`，算梯度时`ze`会接收解码器传来的梯度。

```
# stop gradient
decoder_input = ze + (zq - ze).detach()
```

求最近邻的部分就到此结束了。最后再补充一句，前向传播函数不仅返回了重建结果`x_hat`，还返回了`ze`, `zq`。这是因为我们待会要在训练时根据`ze`, `zq`求损失函数。

准备好了模型类后，假设我们已经用某些超参数初始化好了模型`model`，我们可以用下面的代码训练VQVAE。

```
def train_vqvae(model: VQVAE,
                img_shape=None,
                device='cuda',

ckpt_path='dldemos/VQVAE/model.pth',
                batch_size=64,
                dataset_type='MNIST',
                lr=1e-3,
                n_epochs=100,
                l_w_embedding=1,
                l_w_commitment=0.25):
    print('batch size:', batch_size)
    dataloader = get_dataloader(dataset_type,
                                batch_size,

img_shape=img_shape,
                                use_lmdb=USE_LMDB)
    model.to(device)
    model.train()
```

```
    optimizer =
torch.optim.Adam(model.parameters(), lr)
    mse_loss = nn.MSELoss()
    tic = time.time()
    for e in range(n_epochs):
        total_loss = 0

        for x in dataloader:
            current_batch_size = x.shape[0]
            x = x.to(device)

            x_hat, ze, zq = model(x)
            l_reconstruct = mse_loss(x, x_hat)
            l_embedding = mse_loss(ze.detach(),
zq)
            l_commitment = mse_loss(ze,
zq.detach())
            loss = l_reconstruct + \
                l_w_embedding * l_embedding +
l_w_commitment * l_commitment
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            total_loss += loss.item() *
current_batch_size
        total_loss /= len(dataloader.dataset)
```

```
        toc = time.time()
        torch.save(model.state_dict(), ckpt_path)
        print(f'epoch {e} loss: {total_loss}
elapsed {(toc - tic):.2f}s')
    print('Done')
```

先看一下训练函数的参数。其他参数都没什么特别的，只有误差权重`l_w_embedding=1,l_w_commitment=0.25`需要讨论一下。误差函数有三项，但论文只给了第三项的权重（0.25），默认第二项的权重为1。我在实现时把第二项的权重`l_w_embedding`也加上了。

```
def train_vqvae(model: VQVAE,
                img_shape=None,
                device='cuda',

ckpt_path='dldemos/VQVAE/model.pth',
                batch_size=64,
                dataset_type='MNIST',
                lr=1e-3,
                n_epochs=100,
                l_w_embedding=1,
                l_w_commitment=0.25):
```

再来把函数体过一遍。一开始，我们可以用传来的参数把`dataloader`初始化一下。

```
print('batch size:', batch_size)
dataloader = get_dataloader(dataset_type,
                            batch_size,
                            img_shape=img_shape,
                            use_lmdb=USE_LMDB)
```

再把模型的状态调好，并准备好优化器和算均方误差的函数。

```
model.to(device)
model.train()
optimizer = torch.optim.Adam(model.parameters(),
lr)
mse_loss = nn.MSELoss()
```

准备好变量后，进入训练循环。训练的过程比较常规，唯一要注意的就是误差计算部分。由于我们把复杂的逻辑都放在了模型类中，这里我们可以直接先用`model(x)`得到重建图像`x_hat`和算误差的`ze, zq`，再根据论文里的公式算3个均方误差，最后求一个加权和，代码比较简明。

```
for e in range(n_epochs):
    for x in dataloader:
        current_batch_size = x.shape[0]
        x = x.to(device)

        x_hat, ze, zq = model(x)
```

```
        l_reconstruct = mse_loss(x, x_hat)
        l_embedding = mse_loss(ze.detach(), zq)
        l_commitment = mse_loss(ze, zq.detach())
        loss = l_reconstruct + \
            l_w_embedding * l_embedding +
l_w_commitment * l_commitment
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

训练完毕后，我们可以用下面的代码来测试VQVAE的重建效果。所谓重建，就是模拟训练的过程，随机取一些图片，先编码后解码，看解码出来的图片和原图片是否一致。为了获取重建后的图片，我们只需要直接执行前向传播函数`model(x)`即可。

```
def reconstruct(model, x, device,
dataset_type='MNIST'):
    model.to(device)
    model.eval()
    with torch.no_grad():
        x_hat, _, _ = model(x)
    n = x.shape[0]
    n1 = int(n**0.5)
    x_cat = torch.concat((x, x_hat), 3)
    x_cat = einops.rearrange(x_cat, '(n1 n2) c h w
-> (n1 h) (n2 w) c', n1=n1)
```

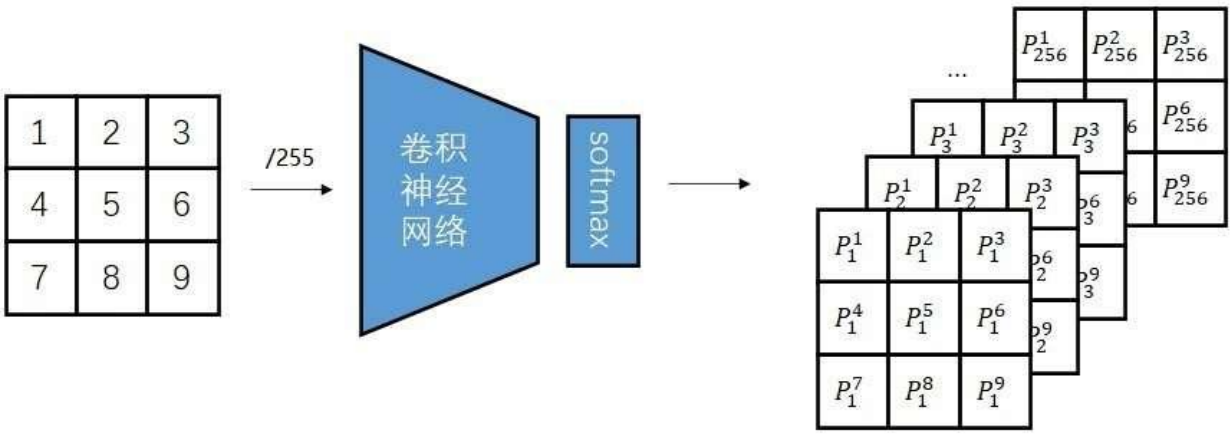
```
    x_cat = (x_cat.clip(0, 1) *
255).cpu().numpy().astype(np.uint8)
    if dataset_type == 'CelebA' or dataset_type ==
'CelebAHQ':
        x_cat = cv2.cvtColor(x_cat,
cv2.COLOR_RGB2BGR)

cv2.imwrite(f'work_dirs/vqvae_reconstruct_{dataset
_type}.jpg', x_cat)

vqvae = ...
dataloader = get_dataloader(...)
img = next(iter(dataloader)).to(device)
reconstruct(vqvae, img, device,
cfg['dataset_type'])
```

训练压缩图像生成模型 PixelCNN

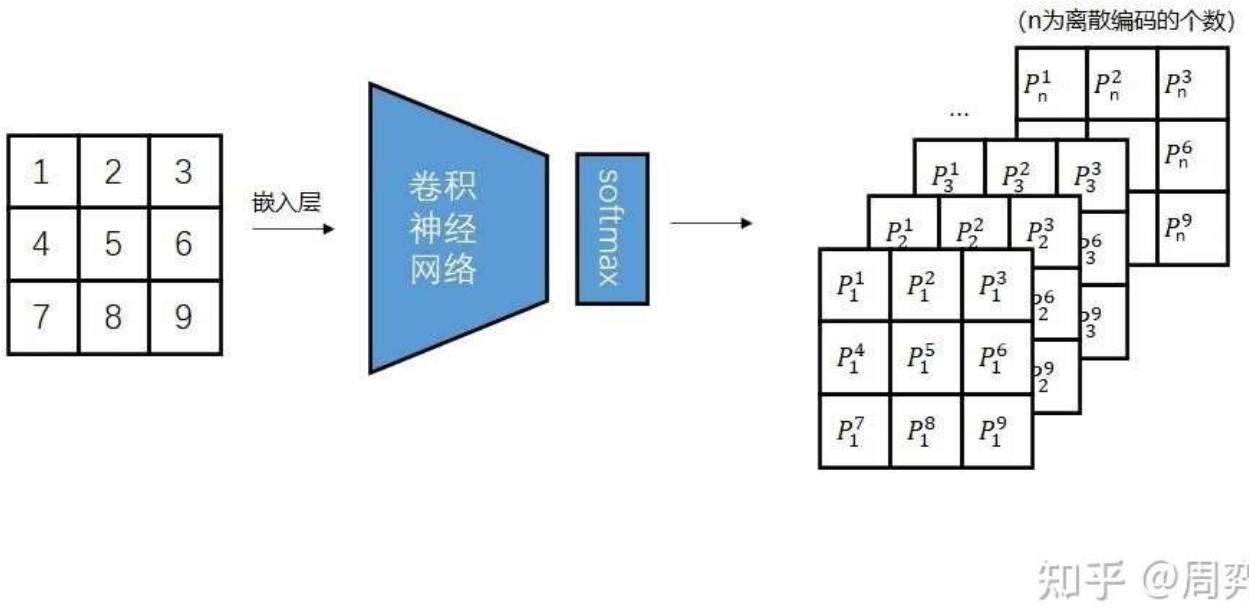
有了一个VQVAE后，我们要用另一个模型对VQVAE的离散空间采样，也就是训练一个能生成压缩图片的模型。我们可以按照VQVAE论文的方法，使用PixelCNN来生成压缩图片。

PixelCNN 的原理及实现方法就不在这里过多介绍了。详情可以参见我之前的PixelCNN解读文章。简单来说，PixelCNN给每个像素从左到右，从上到下地编了一个序号，让每个像素仅由之前所有像素决定。采样时，PixelCNN按序号从左上到右下逐个生成图像的每一个像素；训练时，PixelCNN使用了某种掩码机制，使得每个像素只能看到编号更小的像素，并行地输出每一个像素的生成结果。

PixelCNN具体的训练示意图如下。模型的输入是一幅图片，每个像素的取值是0_{255}；模型给图片的每个像素输出了一个概率分布，即表示此处颜色取0，取1，……，取255的概率。由于神经网络假设数据的输入符合标准正态分布，我们要在数据输入前把整型的颜色转换成0^1之间的浮点数。最简单的转换方法是除以255。



以上是训练PixelCNN生成普通图片的过程。而在训练PixelCNN生成压缩图片时，上述过程需要修改。压缩图片的取值是离散编码。离散编码和颜色值不同，它不是连续的。你可以说颜色1和颜色0、2相近，但不能说离散编码1和离散编码0、2相近。因此，为了让PixelCNN建模离散编码，需要把原来的除以255操作换成一个嵌入层，使得网络能够读取离散编码。



反映在代码中，假设我们已经有了一个普通的PixelCNN模型`GatedPixelCNN`，我们需要在整个模型的最前面套一个嵌入层，嵌入层的嵌入个数等于离散编码的个数(`color_level`)，嵌入长度等于模型的特征长度(`p`)。由于嵌入层会直接输出一个长度为`p`的向量，我们还需要把第一个模块的输入通道数改成`p`。

```
from dldemos.pixelcnn.model import GatedPixelCNN,
GatedBlock
```

```
import torch.nn as nn


class PixelCNNWithEmbedding(GatedPixelCNN):

    def __init__(self, n_blocks, p, linear_dim,
bn=True, color_level=256):
        super().__init__(n_blocks, p, linear_dim,
bn, color_level)
        self.embedding = nn.Embedding(color_level,
p)
        self.block1 = GatedBlock('A', p, p, bn)

    def forward(self, x):
        x = self.embedding(x)
        x = x.permute(0, 3, 1, 2).contiguous()
        return super().forward(x)
```

有了一个能处理离散编码的PixelCNN后，我们可以用下面的代码来训练PixelCNN。

```
def train_generative_model(vqvae: VQVAE,
                           model,
                           img_shape=None,
                           device='cuda',
```

```
ckpt_path='dldemos/VQVAE/gen_model.pth',
                            dataset_type='MNIST',
                            batch_size=64,
                            n_epochs=50):
    print('batch size:', batch_size)
    dataloader = get_dataloader(dataset_type,
                                batch_size,

img_shape=img_shape,
                                use_lmdb=USE_LMDB)
    vqvae.to(device)
    vqvae.eval()
    model.to(device)
    model.train()
    optimizer =
torch.optim.Adam(model.parameters(), 1e-3)
    loss_fn = nn.CrossEntropyLoss()
    tic = time.time()
    for e in range(n_epochs):
        total_loss = 0
        for x in dataloader:
            current_batch_size = x.shape[0]
            with torch.no_grad():
                x = x.to(device)
                x = vqvae.encode(x)
```

```
            predict_x = model(x)
            loss = loss_fn(predict_x, x)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            total_loss += loss.item() *
current_batch_size
        total_loss /= len(dataloader.dataset)
        toc = time.time()
        torch.save(model.state_dict(), ckpt_path)
        print(f'epoch {e} loss: {total_loss}
elapsed {(toc - tic):.2f}s')
    print('Done')
gen_model =
PixelCNNWithEmbedding(cfg['pixelcnn_n_blocks'],

cfg['pixelcnn_dim'],

cfg['pixelcnn_linear_dim'], True,

cfg['n_embedding'])
vqvae.load_state_dict(torch.load(cfg['vqvae_path']
))
train_generative_model(vqvae,
                       gen_model,
```

```
                            img_shape=(img_shape[1],
img_shape[2]),
                            device=device,

ckpt_path=cfg['gen_model_path'],

dataset_type=cfg['dataset_type'],

batch_size=cfg['batch_size_2'],

n_epochs=cfg['n_epochs_2'])
```

训练部分的核心代码如下：

```
loss_fn = nn.CrossEntropyLoss()
for x in dataloader:
    current_batch_size = x.shape[0]
    with torch.no_grad():
        x = x.to(device)
        x = vqvae.encode(x)

    predict_x = model(x)
    loss = loss_fn(predict_x, x)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

这段代码的意思是说，从训练集里随机取图片`x`，再将图片压缩成离散编码`x = vqvae.encode(x)`。这时，`x`既是PixelCNN的输入，也是PixelCNN的拟合目标。把它输入进PixelCNN，PixelCNN会输出每个像素的概率分布。用交叉熵损失函数约束输出结果即可。

训练完毕后，我们可以用下面的函数来完成整套图像生成流水线。

```
def sample_imgs(vqvae: VQVAE,
                gen_model,
                img_shape,
                n_sample=81,
                device='cuda',
                dataset_type='MNIST'):
    vqvae = vqvae.to(device)
    vqvae.eval()
    gen_model = gen_model.to(device)
    gen_model.eval()

    C, H, W = img_shape
    H, W = vqvae.get_latent_HW((C, H, W))
    input_shape = (n_sample, H, W)
    x =
torch.zeros(input_shape).to(device).to(torch.long)
```

```
    with torch.no_grad():
        for i in range(H):
            for j in range(W):
                output = gen_model(x)
                prob_dist = F.softmax(output[:, :,
i, j], -1)
                pixel =
torch.multinomial(prob_dist, 1)
                x[:, i, j] = pixel[:, 0]

    imgs = vqvae.decode(x)

    imgs = imgs * 255
    imgs = imgs.clip(0, 255)
    imgs = einops.rearrange(imgs,
                            '(n1 n2) c h w -> (n1
h) (n2 w) c',
                            n1=int(n_sample**0.5))

    imgs =
imgs.detach().cpu().numpy().astype(np.uint8)
    if dataset_type == 'CelebA' or dataset_type ==
'CelebAHQ':
        imgs = cv2.cvtColor(imgs,
cv2.COLOR_RGB2BGR)
```

```
cv2.imwrite(f'work_dirs/vqvae_sample_{dataset_type}.jpg', imgs)
```

抛掉前后处理，和图像生成有关的代码如下。一开始，我们要随便创建一个空图片`x`，用于储存PixelCNN生成的压缩图片。之后，我们按顺序遍历每个像素，把当前图片输入进PixelCNN，让PixelCNN预测下一个像素的概率分布`prob_dist`。我们再用`torch.multinomial`从概率分布中采样，把采样的结果填回图片。遍历结束后，我们用VQVAE的解码器把压缩图片变成真实图片。

```
C, H, W = img_shape
H, W = vqvae.get_latent_HW((C, H, W))
input_shape = (n_sample, H, W)
x =
torch.zeros(input_shape).to(device).to(torch.long)
with torch.no_grad():
    for i in range(H):
        for j in range(W):
            output = gen_model(x)
            prob_dist = F.softmax(output[:, :, i,
j], -1)
            pixel = torch.multinomial(prob_dist,
1)
            x[:, i, j] = pixel[:, 0]

imgs = vqvae.decode(x)
```

至此，我们已经实现了用VQVAE做图像生成的四个任务：训练VQVAE、重建图像、训练PixelCNN、随机生成图像。完整的`main`函数如下：

```
if __name__ == '__main__':
    os.makedirs('work_dirs', exist_ok=True)

    parser = argparse.ArgumentParser()
```

```
    parser.add_argument('-c', type=int, default=0)
    parser.add_argument('-d', type=int, default=0)
    args = parser.parse_args()
    cfg = get_cfg(args.c)

    device = f'cuda:{args.d}'

    img_shape = cfg['img_shape']

    vqvae = VQVAE(img_shape[0], cfg['dim'],
cfg['n_embedding'])
    gen_model =
PixelCNNWithEmbedding(cfg['pixelcnn_n_blocks'],

cfg['pixelcnn_dim'],

cfg['pixelcnn_linear_dim'], True,

cfg['n_embedding'])
    # 1. Train VQVAE
    train_vqvae(vqvae,
                img_shape=(img_shape[1],
img_shape[2]),
                device=device,
                ckpt_path=cfg['vqvae_path'],
                batch_size=cfg['batch_size'],
```

```
                dataset_type=cfg['dataset_type'],
                lr=cfg['lr'],
                n_epochs=cfg['n_epochs'],

l_w_embedding=cfg['l_w_embedding'],

l_w_commitment=cfg['l_w_commitment'])

    # 2. Test VQVAE by visualizaing reconstruction
result

vqvae.load_state_dict(torch.load(cfg['vqvae_path']
))
    dataloader =
get_dataloader(cfg['dataset_type'],
                                16,
                                img_shape=
(img_shape[1], img_shape[2]))
    img = next(iter(dataloader)).to(device)
    reconstruct(vqvae, img, device,
cfg['dataset_type'])

    # 3. Train Generative model (Gated PixelCNN in
our project)
```

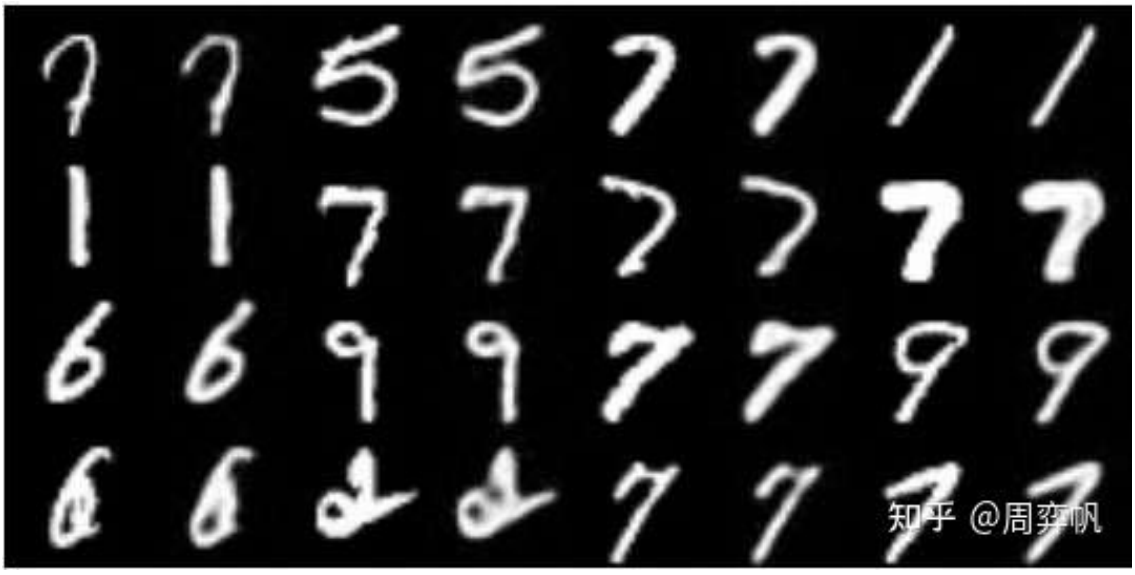
```
vqvae.load_state_dict(torch.load(cfg['vqvae_path']
))

    train_generative_model(vqvae,
                           gen_model,
                           img_shape=
(img_shape[1], img_shape[2]),
                           device=device,

ckpt_path=cfg['gen_model_path'],

dataset_type=cfg['dataset_type'],

batch_size=cfg['batch_size_2'],

n_epochs=cfg['n_epochs_2'])

    # 4. Sample VQVAE

vqvae.load_state_dict(torch.load(cfg['vqvae_path']
))

gen_model.load_state_dict(torch.load(cfg['gen_mode
l_path']))
    sample_imgs(vqvae,
```

```
                    gen_model,
                    cfg['img_shape'],
                    device=device,
                    dataset_type=cfg['dataset_type'])
```

实验

VQVAE有两个超参数：嵌入个数`n_embedding`、特征向量长度`dim`。论文中`n_embedding=512`，`dim=256`。而经我实现发现，用更小的参数量也能达到不错的效果。

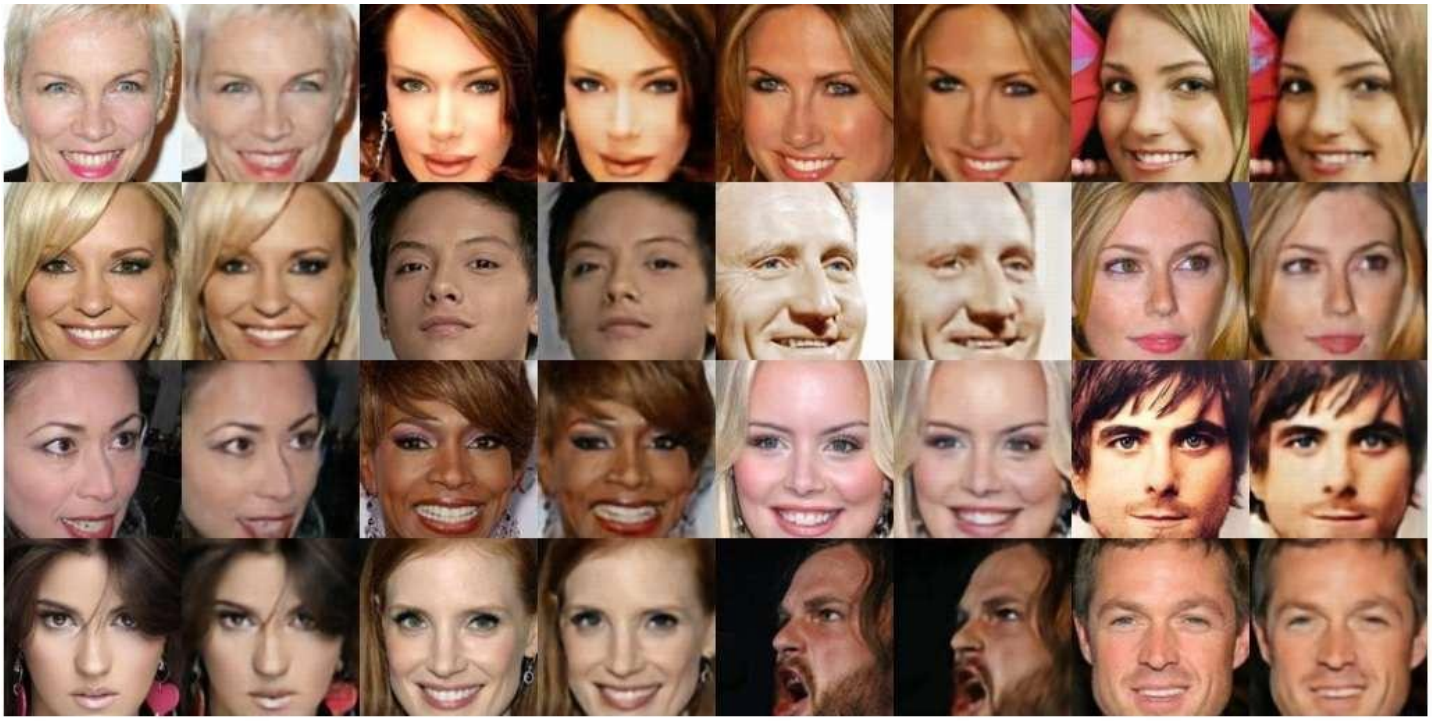
所有实验的配置文件我都放在了该项目目录下`config.py`文件中。对于MNIST数据集，我使用的模型超参数为：`dim=32, n_embedding=32`。VQVAE重建结果如下所示。可以说重建得几乎完美（每对图片左图为原图，右图为重建结果）。



而对于CelebAHQ数据集，我测试了不同输入尺寸下的不同VQVAE，共有4组配置。

1. `shape=(3, 128, 128) dim=128 n_embedding=64`
2. `shape=(3, 128, 128) dim=128 n_embedding=128`
3. `shape=(3, 64, 64) dim=128 n_embedding=64`
4. `shape=(3, 64, 64) dim=128 n_embedding=32`

实验的结果很好预测。对于同尺寸的图片，嵌入数越多重建效果越好。这里我只展示下第一组和第二组的重建结果。



dim 128 n_embedding 64

dim 128 n_embedding 128

可以看出，VQVAE的重建效果还不错。但由于只使用了均方误差，重建图片在细节上还是比较模糊。重建效果还是很重要的，它决定了该方法做图像生成的质量上限。后续有很多工作都试图提升VQVAE的重建效果。

接下来来看一下随机图像生成的实验。PixelCNN主要有模块数`n_blocks`、特征长度`dim`，输出线性层特征长度`linear_dim`这三个超参数。其中模块数一般是固定的，而输出线性层就被用了一次，其特征长度的影响不大。最需要调节的是特征长度`dim`。对于MNIST，我的超参数设置为

- `n_blocks=15 dim=128 linear_dim=32`.

对于CelebAHQ，我的超参数设置为

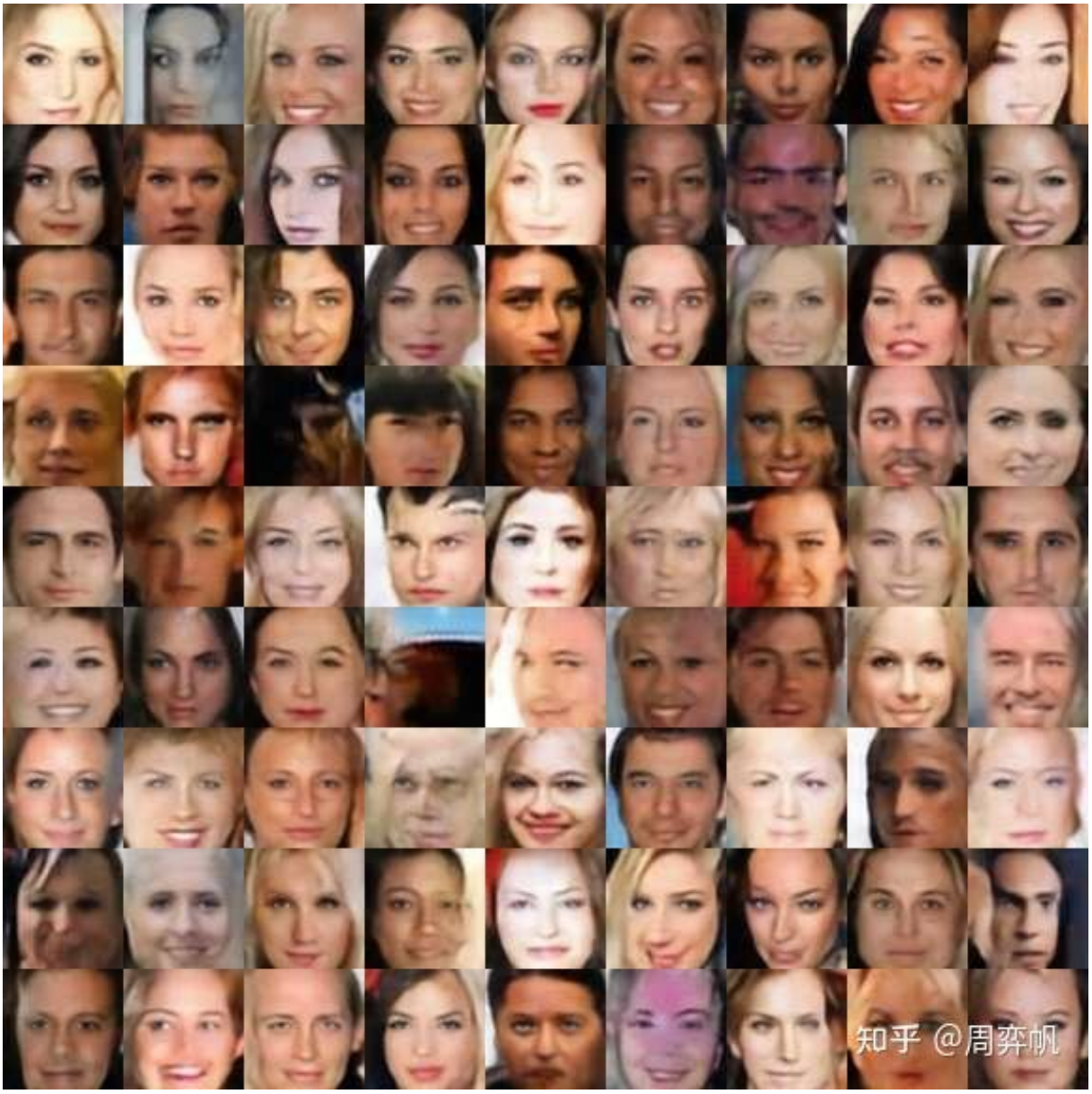
- `n_blocks=15 dim=384 linear_dim=256`.

PixelCNN的训练时间主要由输入图片尺寸和`dim`决定，训练难度主要由VQVAE的嵌入个数（即多分类的类别数）决定。PixelCNN训起来很花时间。如果时间有限，在CelebAHQ上建议只训练最小最简单的第4组配置。我在项目中提供了PixelCNN的并行训练脚本，比如用下面的命令可以用4张卡在1号配置下并行训练。

```
torchrun --nproc_per_node=4
dldemos/VQVAE/dist_train_pixelcnn.py -c 1
```

来看一下实验结果。MNIST上的采样结果还是非常不错的。

CelebAHQ上的结果会差一点。以下是第4组配置(图像边长`64`，嵌入数`32`)的采样结果。大部分图片都还行，起码看得出是一张人脸。但`64x64`的图片本来就分辨率不高，加上VQVAE解码的损耗，放大来看人脸还是比较模糊的。



第1组配置（图像边长`128`，嵌入数`64`）的PixelCNN实在训练得太慢了，我只训了一个半成品模型。由于部分生成结果比较吓人，我只挑了几个还能看得过去的生成结果。可以看出，如果把模型训完的话，边长128的模型肯定比边长64的模型效果更好。

参考资料

网上几乎找不到在CelebAHQ上训练的VQVAE PyTorch项目。我在实现这份代码时，参考了以下项目：

- 官方TensorFlow实现 https://github.com/deepmind/sonnet/blob/v1/sonnet/examples/vqvae_example.ipynb。主要代码都写在一个notebook里。
- 官方实现的PyTorch复现 https://github.com/MishaLaskin/vqvae。

- 苏剑林的TensorFlow实现。用的生成模型不是PixelCNN而是Transformer。https://github.com/bojone/vae/blob/master/vq_vae_keras.py

实验经历分享

别看VQVAE的代码不难，我做这些实验时还是经历了不少波折的。

一开始，我花一天就把代码写完了，并完成了MNIST上的实验。我觉得在MNIST上做实验的难度太低，不过瘾，就准备把数据集换成CelebA再做一些实验。结果这一做就是两个星期。

换成CelebA后，我碰到的第一个问题是VQVAE训练速度太慢。我尝试减半模型参数，训练时间却减小得不明显。我大致猜出是数据读取占用了大量时间，用性能分析工具一查，果然如此。原来我在`DataLoader`中一直只用了一个线程，加上`num_workers=4`就好了。我还把数据集打包成LMDB格式进一步加快数据读取速度。

之后，我又发现VQVAE在CelebA上的重建效果很差。我尝试增加模型参数，没起作用。我又怀疑是64x64的图片质量太低，模型学不到东西，就尝试把输入尺寸改成128x128，并把数据集从CelebA换成CelebAHQ，重建效果依然不行。我调了很多参数，

发现了一些奇怪的现象：在嵌入层前使用和不使用BatchNorm对结果的影响很大，且显式初始化嵌入层会让模型的误差一直居高不下。我实在是找不到问题，就拿代码对着别人的PyTorch实现一行一行比较过去。总算，我发现我在使用嵌入层时是用`vq_embedding.weight.data[x]`（因为前面已经获取了这个矩阵，这样写比较自然），别人是用`vq_embedding(x)`。我的写法会把嵌入层排除在梯度计算外，嵌入层根本得不到优化。我说怎么换了一个嵌入层的初始化方法模型就根本训不动了。改完bug之后，只训了5个epoch，新模型的误差比原来训练数小时的模型要低了。新模型的重建效果非常好。

总算，任务完成了一半，现在只剩PixelCNN要训练了。我先尝试训练输入为128x128，嵌入数64的模型，采样结果很差。为了加快实验速度，我把输入尺寸减小到64x64，再次训练，采样结果还是不行。根据我之前的经验，PixelCNN的训练难度主要取决于类别数。于是，我把嵌入的数量从64改成了32，并大幅增加PixelCNN的参数量，再次训练。过了很久，训练误差终于降到0.08左右。我一测，这次的采样结果还不错。

这样看来，之前的采样效果不好，是输入128x128，嵌入数64的实验太难了。我毕竟只是想做一个demo，在一个小型实验上成功就行了，没必要花时间去做更耗时的实验。按理说，我应该就此收手。但是，我就是咽不下这一口气，就是想在128x128的实验上成功。我再次加大了PixelCNN的参数量，用128x128的配

置，大火慢炖，训练了一天一夜。第二天一早起来，我看到这回的误差也降到了0.08。上次的实验误差降到这个程度时实验已经成功了。我迫不及待地去测试采样效果，却发现采样效果还是稀烂。没办法，我选择投降，开始写这篇文章，准备收工。

写到PixelCNN介绍的那一章节时，我正准备讲解代码。看到PixelCNN训练之前预处理除以`color_level`那一行时，我楞了一下：这行代码是用来做什么的来着？这段代码全是从PixelCNN项目里复制过来的。当时是做普通图片的图像生成，所以要对输入颜色做一个预处理，把整数颜色变成0~1之间的浮点数。但现在是在生成压缩图片，不能这样处理啊！我恍然大悟，知道是在处理离散输入时做错了。应该多加一个嵌入层，把离散值转换成向量。由于VQVAE的重点不在生成模型上，原论文根本没有强调PixelCNN在离散编码上的实现细节。网上几乎所有文章也都没谈这一点。因此，我在实现PixelCNN时，直接不假思索地把原来的代码搬了过来，根本没想过这种地方会出现bug。

把这处bug改完后，我再次开启训练。这下所有模型的采样结果都正常了。误差降到0.5左右就已经有不错的采样结果了，原来我之前把误差降到0.08完全是无用功。太气人了。

这次的实验让我学到了很多东西。首先是PyTorch编程上的一些注意事项：

- 调用`embedding.weight.data[x]`是传不了梯度的。

- 如果读数据时有费时的处理操作（读写硬盘、解码），要在`Dataloader`里设置`num_workers`。

另外，在测试一个模型是否实现成功时有一个重要的准则：

- 不要仅在简单的数据集（如MNIST）上测试。测试成功可能只是暴力拟合的结果。只有在一个难度较大的数据集上测试成功才能说模型没有问题。

在观察模型是否训成功时，还需要注意：

- 训练误差降低不代表模型更优。训练误差的评价方法和模型实际使用方法可能完全不同。不能像我这样偷懒不加测试指标。

除了学到的东西外，我还有一些感想。在别人的项目的基础上修改、照着他人代码复现、完全自己动手从零开始写，对于深度学习项目来说，这三种实现方式的难度是依次递增的。改别人的项目，你可能去配置文件里改一两个数字就行了。而照着他人代码复现，最起码你能把代码改成和他人的代码一模一样，然后再去比较哪一块错了。自己动手写，则是有bug都找不到可以参考的地方了。说深度学习的算法难以调试，难就难在这里。效果不好，你很难说清是训练代码错了、超参数没设置好、训练流程错了，或是测试代码错了。可以出错的地方太多了，通常的代码调试手段难以用在深度学习项目上。

对于想要在深度学习上有所建树的初学者，我建议一定要从零动手复现项目。很多工程经验是难以总结的，只有踩了一遍坑才能知道。除了凭借经验外，还可以掌握一些特定的工程方法来减少bug的出现。比如运行训练之前先拿性能工具分析一遍，看看代码是否有误，是否可以提速；又比如可以训练几步后看所有可学习参数是否被正确修改。