生成模型之VQ-VAE


小小将
华中科技大学 工学硕士

目录
收起

> “**What I cannot create, I do not understand.**” -- Richard Feynman

---

上一篇文章**生成模型之PixelCNN**介绍了基于自回归的生成模型，这篇文章将介绍DeepMind（和PixelCNN同一作）于2017年提出的一种基于**离散隐变量（Discrete Latent variables）的生成模型：VQ-VAE。VQ-VAE相比VAE有两个重要的区别：首先VQ-VAE采用离散隐变量**，而不是像VAE那样采用连续的隐变量；然后VQ-VAE需要单独**训练一个基于自回归的模型如PixelCNN来学习先验（prior）**，而不是像VAE那样采用一个固定的先验（标准正态分布）。此外，VQ-VAE还是一个强大的**无监督表征学习模型**，它学习的离散编码具有很强的表征能力，最近比较火的文本转图像模型DALL-E也是基于VQ-VAE的，而且最近的一些基于masked image modeling的无监督学习方法如BEiT也用VQ-VAE得到的离散编码作为训练目标。这篇文章将讲解VQ-VAE的原理以及具体的代码实现。

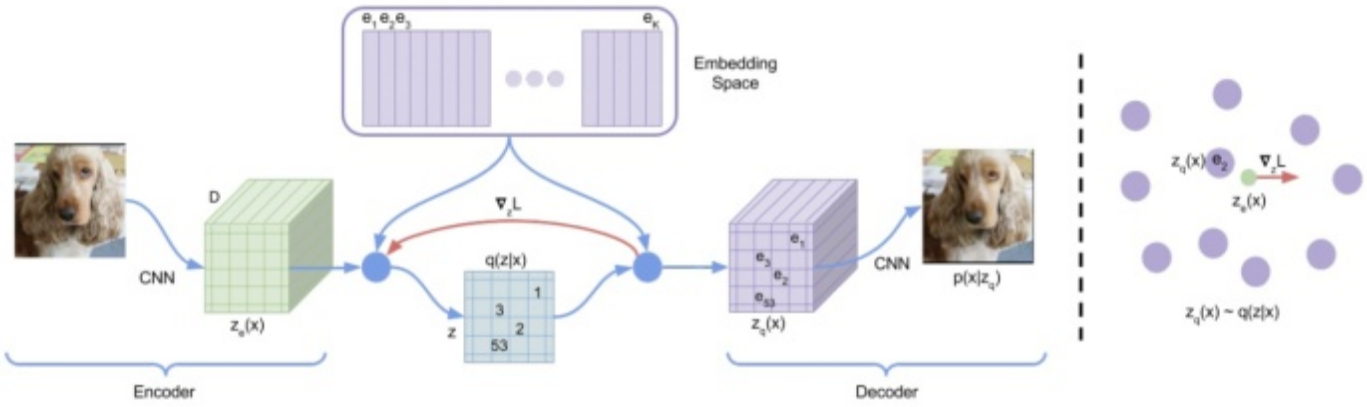


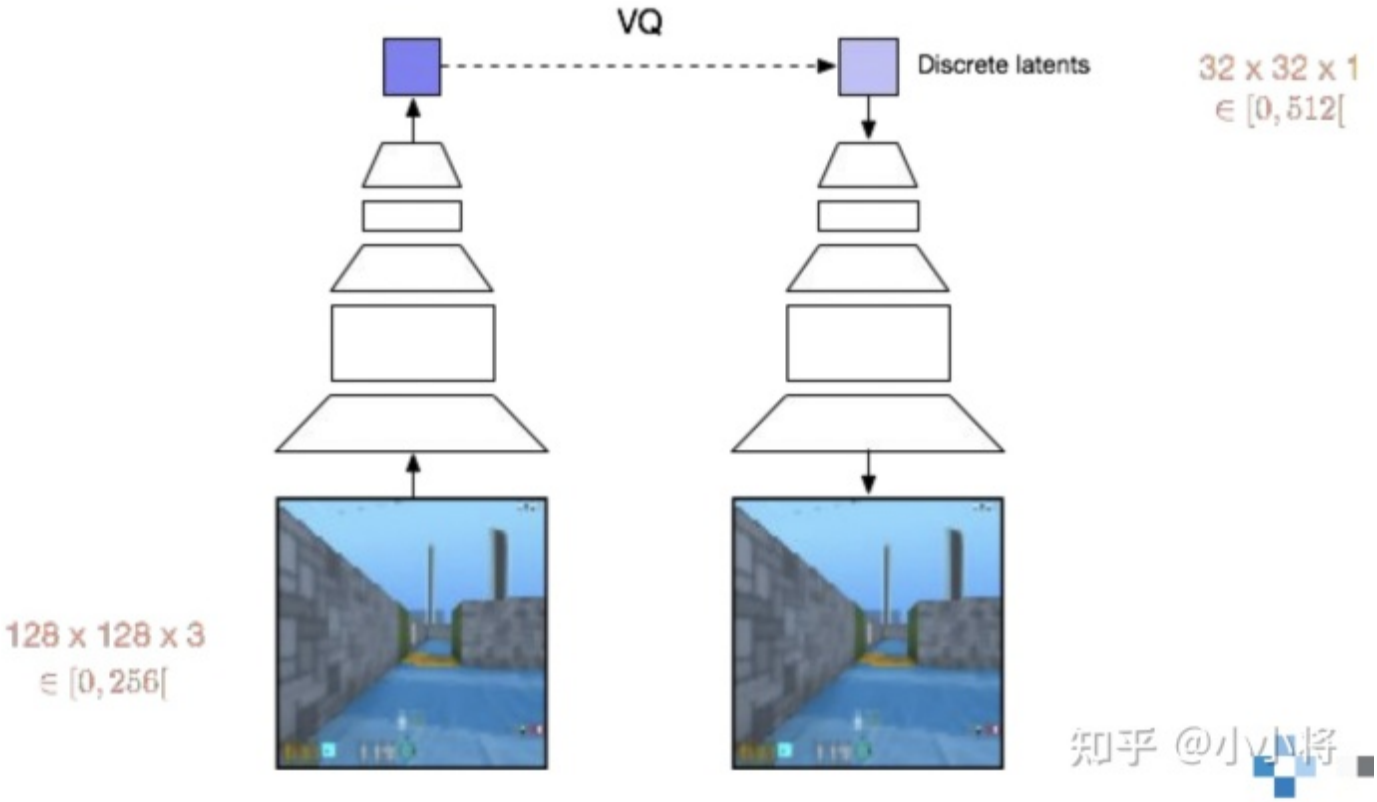
Figure 1: Left: A figure describing the VQ-VAE. Right: Visualisation of the embedding space. The output of the encoder $z(x)$ is mapped to the nearest point e_2. The gradient $\nabla_z L$ (in red) will push the encoder to change its output, which could alter the configuration in the next forward pass.

VQ-VAE

一个VAE模型包括三个部分：后验分布（Posterior）$q(z|x)$，先验分布（Prior）$p(z)$，以及似然（Likelihood）$p(x|z)$。其中后验分布用encoder网络来学习，似然用decoder网络来学习，而先验分布采用参数固定的标准正态分布。在VAE学习过程中，后验分布往往假定是一个对角方差的多元正态分布，而隐变量z是一个连续的随机变量。

VQ-VAE与VAE的最主要的区别是VQ-VAE采用离散隐变量，如下图所示，对于encoder的输出通过向量量化（vector quantisation，VQ）的方法来离散化。至于为啥采用离散编码，论文的**slides**给出了三个主要原因：

- 许多重要的事物都是离散的，如语言；
- 更容易对先验建模，VQ-VAE采用PixelCNN来学习先验，离散编码只需要简单地采用softmax多分类；
- 连续的表征往往被encoder/decoder内在地离散化；



要想实现向量量化VQ，首先要定义一个**离散隐变量空间**$e \in R^{K \times D}$，又称为**embedding空间**（类比NLP中的word embedding），其中K为embedding空间大小，即embedding向量的个数，而D为每个embedding向量e_i的维度。因而，这里总共有K个embedding向量$e_i \in R^D, i \in 1, 2, \ldots, K$，我们要将encoder的输出$z_e(x)$量化为其中的一个embedding向量，这里采用**最近邻的查找方法**来量化：

$$z_q(x) = \text{Quantize}(z_e(x)) = e_k \text{ where } k = \arg\min_i \|z_e(x) - \mathbf{e}_i\|_2$$

实际上就是计算$z_e(x)$和每个embedding向量e_i的欧式距离，然后选择距离最近的embedding向量e_k作为量化值$z_q(x)$。这里得到的量化值$z_q(x)$将作为decoder的输入，所以VQ-VAE的整个过程变成：

$$x \to z_e(x) = \text{Encoder}(x) \to z_q(x) = \text{Quantize}(z_q(x)) \to x' = \text{Decoder}(z_q(x))$$

原来的VAE的后验分布$q(z|x)$是多元高斯分布，但对于VQ-VAE，经过VQ之后，后验分布$q(z|x)$可以看成是一个**多类分布（categorical distribution）**，而且其概率分布为one-hot类型：

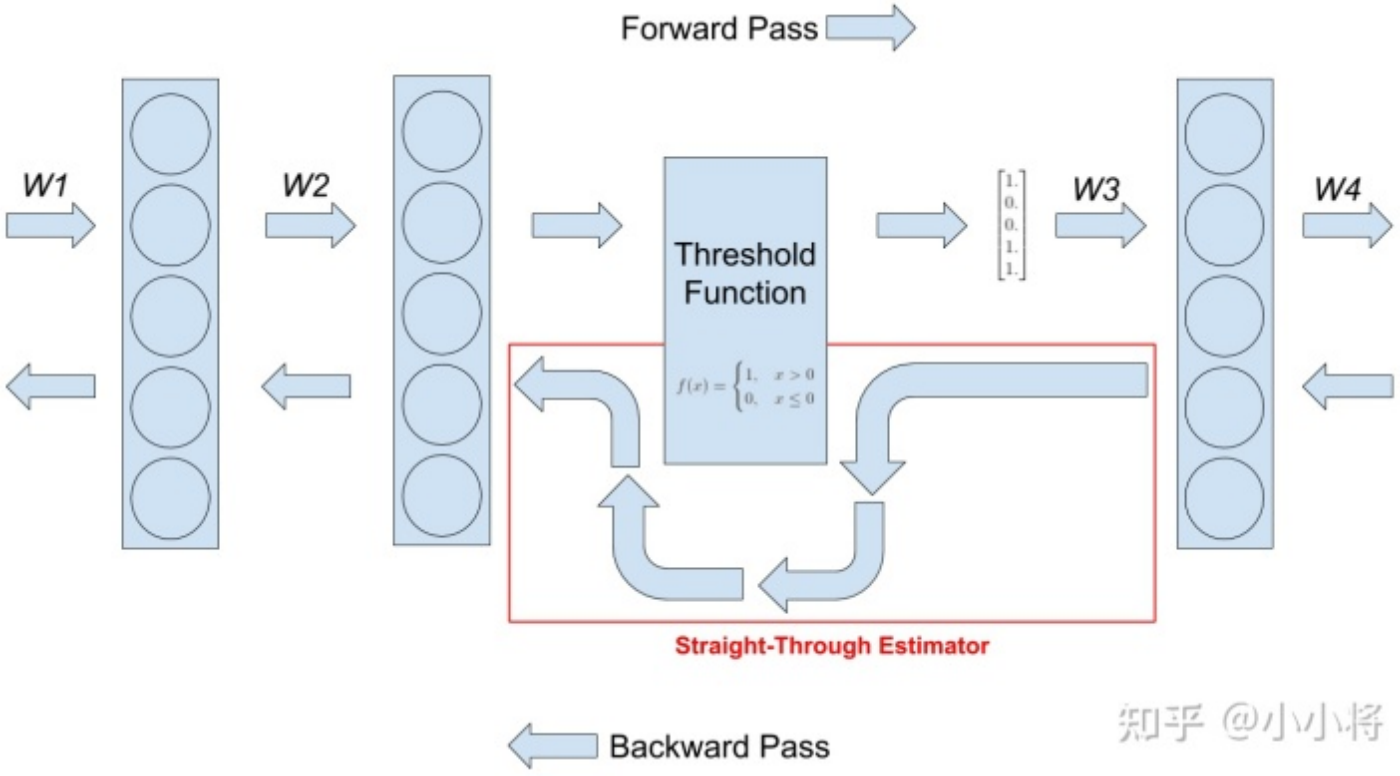
$$q(z=k|x)=\begin{cases}1 & \text{for } k=\arg\min_i \|z_e(x)-\mathbf{e}_i\|_2 \\ 0 & \text{otherwise}\end{cases}$$

此时，后验分布$q(z|x)$其实变成了**一个确定的分布**，因为确定k的过程没有任何随机因素。如果我们定义先验分布$p(z)$为一个均匀的多类分布的话（每个类别的概率均为$1/K$），此时就可以直接简单计算出后验分布$q(z|x)$和先验分布$p(z)$的KL散度：

$$\begin{aligned}\mathrm{KL}(q(z|x)||p(z)) &= \sum q(z|x)\log\frac{q(z|x)}{p(z)} \\ &= 1\cdot\log\frac{1}{1/K}+(K-1)\cdot 0\cdot\log\frac{0}{1/K} \\ &= \log K\end{aligned}$$

此时KL散度就是一个常量，那么VQ-VAE的训练损失就剩下了一项重建误差$\log p(x|z)$。实际上VQ-VAE的训练过程就没有用到先验分布，所以后面我们需要单独训练一个先验模型来生成数据，这是VQ-VAE和VAE的第二个区别。VQ-VAE分成两个阶段来得到生成模型$p(z)p(x|z)$，可以避免VAE训练过程中容易出现的“posterior collapse”。

VQ-VAE还存在一个问题，那就是**由于argmin操作不可导，所以重建误差的梯度就无法传导到encoder**。论文采用**straight-through estimator**来解决这个问题，所谓**straight-through estimator**其实就是一种用来估计一些不可导函数梯度的方法，如下图所示，threshold function是不可导的，此时我们在计算梯度时，直接忽略它而采用上游得到的梯度，这个行为就认为threshold function是一个identity function一样。



VQ-VAE的decoder的输入是$z_q(x)$，这里我们直接用重建误差相对于$z_q(x)$的梯度来作为encoder的输出$z_e(x)$的梯度，由于$z_q(x)$和$z_e(x)$的维度一样，所以这样做不需要任何特殊处理。虽然通过**straight-through estimator**方法，重建误差的梯度可以传导到encoder，但是embedding向量e_i就接收不到重建误差带的梯度了，这也意味着embedding空间无法参与学习。为了让embedding空间参与训练，论文采用了一种简单的字典学习方法，即计算encoder的输出$z_e(x)$和对应的量化得到的embedding向量e_k的L2误差：

$$\|\text{sg}[z_e(x)] - e_k\|_2^2$$

这里的**sg指的是stop gradient操作**，这意味着这个L2损失只会更新embedding空间，而不会传导到encoder。这里，我们也可以采用另外一种方式：**指数移动平均（exponential moving averages，EMA）**来更新embedding空间。假定$\{z_{i,1}, z_{i,2}, \dots, z_{i,n_i}\}$为一系列和embedding向量$e_i$对应的encoder的输出，此时L2损失为：

$$\sum_j^{n_i} \|z_{i,j} - e_i\|_2^2$$

此时embedding向量e_i的最优值有解析解，即对所有的元素求平均值：$e_i^* = \frac{1}{n_i}\sum_j^{n_i} z_{i,j}$ 然而，训练过程中无法直接这样更新，因为训练是基于mini-batch的，并不是训练数据的全部。类比BatchNorm，我们可以采用EMA来更新embedding：

$$N_i^{(t)} = \gamma N_i^{(t-1)} + (1-\gamma)n_i^{(t)} \quad m_i^{(t)} = \gamma m_i^{(t-1)} + (1-\gamma)\sum_{j=1}^{n_i^{(t)}} z_{i,j}^{(t)} \quad e_i^{(t)} = m_i^{(t)}/N_i^{(t)}$$

这里共需要维护两套EMA参数，一是每个embedding向量e_i的对应的$\{z_{i,1}, z_{i,2}, \dots, z_{i,n_i}\}$元素数量$N_i$，二是$\{z_{i,1}, z_{i,2}, \dots, z_{i,n_i}\}$的求和值$m_i$。每次forward时，我们根据当前mini-batch得到$\{z_{i,1}, z_{i,2}, \dots, z_{i,n_i}\}$，然后执行EMA，而用$m_i$除以$N_i$即可得到当前的embedding向量。**采用EMA这种更新方式往往比直接采用L2损失收敛速度更快**，论文采用的decay值λ为0.99。

除此之外，论文还额外增加一个训练loss：**commitment loss**，这个主要是约束encoder的输出和embedding空间保持一致，以避免encoder的输出变动较大（从一个embedding向量转向另外一个）。**commitment loss**也比较简单，直接计算encoder的输出$z_e(x)$和对应的量化得到的embedding向量e_k的L2误差：

$$\|z_e(x) - \text{sg}[e_k]\|_2^2$$

注意这里的sg是作用在embedding向量e_k上，这意味着这个约束只会影响encoder。

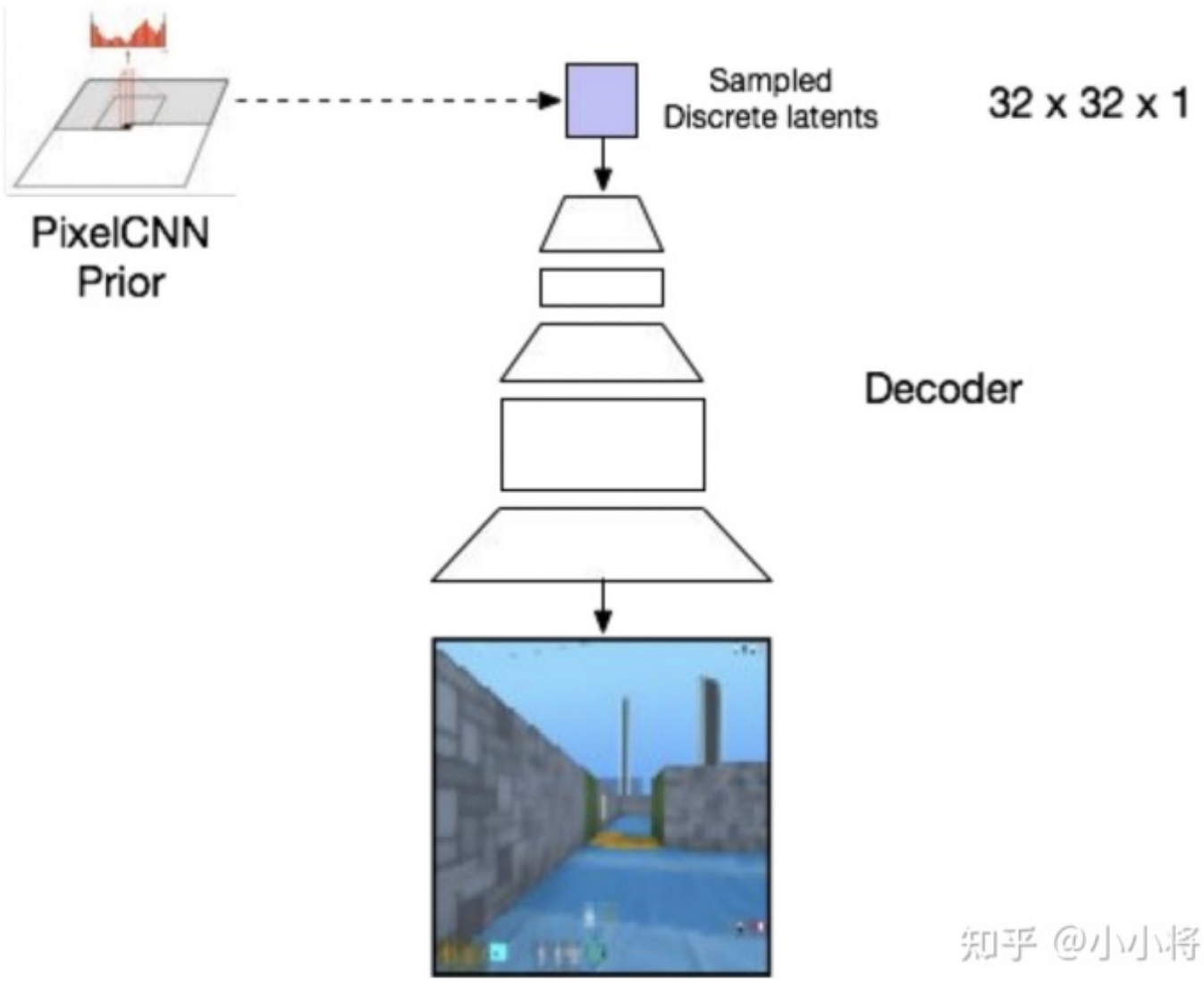
综上，VQ-VAE共包含三个部分的训练loss：**reconstruction loss，VQ loss，commitment loss**。

$$L = \underbrace{\log p(x|z_q(x))}_{\text{reconstruction loss}} + \underbrace{\|\text{sg}[z_e(x)] - e_k\|_2^2}_{\text{VQ loss}} + \underbrace{\beta\|z_e(x) - \text{sg}[e_k]\|_2^2}_{\text{commitment loss}}$$

其中**reconstruction loss**作用在encoder和decoder上，**VQ loss**用来更新embedding空间（也可用EMA方式），而**commitment loss**用来约束encoder，这里的β为权重系数，论文默认设置为0.25。

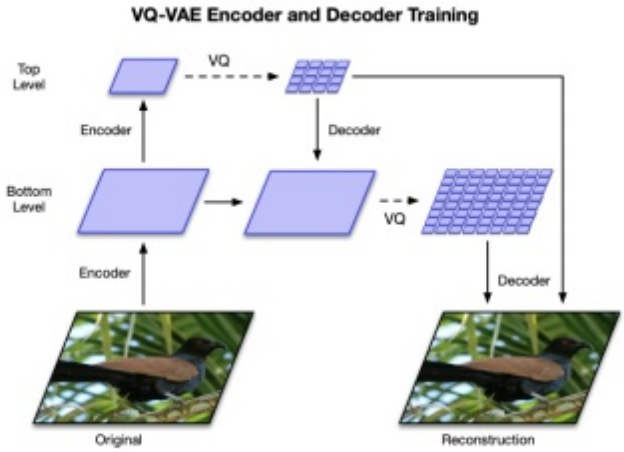
另外，在实际实验中，一张图像会采用N个离散隐变量，这个和encoder得到的特征图大小有关。对于ImageNet数据集，采用32x32大小的中间特征图，所以$N = 32 \times 32$；对于CIFAR10数据集，采用8x8大小的中间特征图，所以$N = 8 \times 8$。对于一张图像，**VQ loss**和**commitment loss**取N个离散编码的loss平均值。从自动编码器的角度来看，VQ-VAE实现了对图像的压缩，即将一张图像压缩成N个离散编码。这里要说明的一点是，VQ-VAE适用于多种模态的数据，除了图像之外，还可以用于语音和视频的生成，这里只讨论图像。

训练好VQ-VAE后，还需要训练一个先验模型来完成数据生成，对于图像来说，可以采用PixelCNN模型，这里我们不再是学习生成原始的pixels，而是学习生成离散编码。首先，我们需要用已经训练好的VQ-VAE模型对训练图像推理，得到每张图像对应的离散编码；然后用一个PixelCNN来对离散编码进行建模，最后的预测层采用基于softmax的多分类，类别数为embedding空间的大小K。那么，生成图像的过程就比较简单了，首先用训练好的PixelCNN模型来采样一个离散编码样本，然后送入VQ-VAE的encoder中，得到生成的图像。整个过程如下图所示：

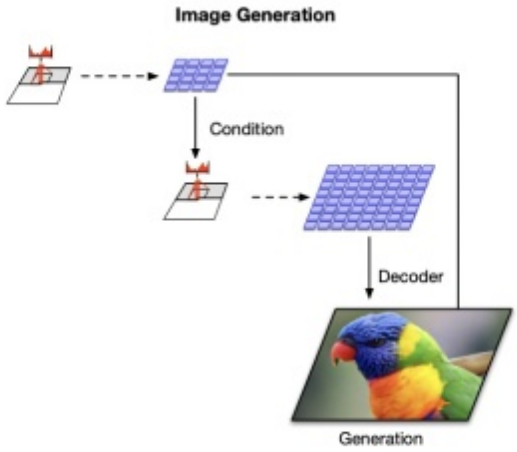


VQ-VAE-2

VQ-VAE-2是DeepMind团队于2019年提出的VQ-VAE的升级版，相比VQ-VAE，**VQ-VAE-2采用多尺度的层级结构**，如下图所示，这里采用了两个尺度的特征来进行量化。**采用多尺度的好处是可以用将图像的局部特征和全局特征来分别建模**，比如这里的Bottom Level的特征用于提取局部信息，而Top Level的特征用于提出全局信息。而且采用层级结构将可以用来生成尺寸较大的图像。



(a) Overview of the architecture of our hierarchical VQ-VAE. The encoders and decoders consist of deep neural networks. The input to the model is a 256×256 image that is compressed to quantized latent maps of size 64×64 and 32×32 for the *bottom* and *top* levels, respectively. The decoder reconstructs the image from the two latent maps.



(b) Multi-stage image generation. The top-level PixelCNN prior is conditioned on the class label, the bottom level PixelCNN is conditioned on the class label as well as the first level code. Thanks to the feed-forward decoder, the mapping between latents to pixels is fast. (The example image with a parrot is generated with this model).

对于一张大小为256x256的图像，首先通过一系列卷积得到下采样1/4的中间特征，其大小为64x64，这个称为Bottom Level特征，然后进一步下采样1/2得到大小为32x32的特征，称为Top Level特征，这里对其量化得到Top Level特征对应的embeddings，将其和Bottom Level特征融合后得到增强的Bottom Level特征，然后进行量化；最后将Top Level和Bottom Level特征对应的embedding送入decoder来重建图像。整个训练过程如下所示：

Algorithm 1 VQ-VAE training (stage 1)

Require: Functions E_{top} , E_{bottom}, D, $\mathbf{x}$ (batch of training images)

$\mathbf{h}_{top} \leftarrow E_{top}(\mathbf{x})$

▷ quantize with top codebook eq 1

$\mathbf{e}_{top} \leftarrow Quantize(\mathbf{h}_{top})$

$\mathbf{h}_{bottom} \leftarrow E_{bottom}(\mathbf{x}, \mathbf{e}_{top})$

▷ quantize with bottom codebook eq 1

$\mathbf{e}_{bottom} \leftarrow Quantize(\mathbf{h}_{bottom})$

$\hat{\mathbf{x}} \leftarrow D(\mathbf{e}_{top}, \mathbf{e}_{bottom})$

▷ Loss according to eq 2

$\theta \leftarrow Update(\mathcal{L}(\mathbf{x}, \hat{\mathbf{x}}))$

Algorithm 2 Prior training (stage 2)

$\mathbf{T}_{top}, \mathbf{T}_{bottom} \leftarrow \emptyset$ ▷ training set

for $\mathbf{x} \in$ training set **do**

  $\mathbf{e}_{top} \leftarrow Quantize(E_{top}(\mathbf{x}))$

  $\mathbf{e}_{bottom} \leftarrow Quantize(E_{bottom}(\mathbf{x}, \mathbf{e}_{top}))$

  $\mathbf{T}_{top} \leftarrow \mathbf{T}_{top} \cup \mathbf{e}_{top}$

  $\mathbf{T}_{bottom} \leftarrow \mathbf{T}_{bottom} \cup \mathbf{e}_{bottom}$

end for

$p_{top} = \texttt{TrainPixelCNN}(\mathbf{T}_{top})$

$p_{bottom} = \texttt{TrainCondPixelCNN}(\mathbf{T}_{bottom}, \mathbf{T}_{top})$

▷ Sampling procedure

while true **do**

  $\mathbf{e}_{top} \sim p_{top}$

  $\mathbf{e}_{bottom} \sim p_{bottom}(\mathbf{e}_{top})$

  $\mathbf{x} \leftarrow D(\mathbf{e}_{top}, \mathbf{e}_{bottom})$

end while

同样地，VQ-VAE-2也采用PixelCNN来对先验建模，只不过这里也需要训练两个PixelCNN：分别用于生成Top Level和Bottom Level特征对应的离散编码。要注意的是第2个PixelCNN其实是一个建立在Top Level基础上的CondPixelCNN。 这里是以两个尺度的特征为例，其实也可以采用更多尺度的特征，即采用更多的层级。下图为采用3个层级的VQ-VAE-2模型的重建效果，可以看到，逐渐融合多个尺度的特征后，生成图像的细节越来越丰富。

Figure 3: Reconstructions from a hierarchical VQ-VAE with three latent maps (top, middle, bottom). The rightmost image is the original. Each latent map adds extra detail to the reconstruction. These latent maps are approximately 3072x, 768x, 192x times smaller than the original image (respectively).

VQ-VAE的代码实现

这里参考**Keras vq_vae blog**和**官方代码**来用PyTorch实现VQ-VAE，首先我们以MNIST数据集来实现VQ-VAE的标准版本（非EMA）。首先要实现的是VQ-VAE最核心的部分：**向量量化VQ**，这里我们也将训练loss的实现放在了类的forward中，不过区分train和eval模式：如果是train模式，除了返回量化后的特征外，还返回VQ loss+commitment loss；而对于eval模式只返回量化后的特征。

```
class VectorQuantizer(nn.Module):
    """
    VQ-VAE layer: Input any tensor to be quantized.
    Args:
        embedding_dim (int): the dimensionality of the tensors in the
          quantized space. Inputs to the modules must be in this format as well.
        num_embeddings (int): the number of vectors in the quantized space.
        commitment_cost (float): scalar which controls the weighting of the loss terms (se
          equation 4 in the paper - this variable is Beta).
    """
    def __init__(self, embedding_dim, num_embeddings, commitment_cost):
        super().__init__()
        self.embedding_dim = embedding_dim
        self.num_embeddings = num_embeddings
        self.commitment_cost = commitment_cost

        # initialize embeddings
        self.embeddings = nn.Embedding(self.num_embeddings, self.embedding_dim)
```

```
    def forward(self, x):
        # [B, C, H, W] -> [B, H, W, C]
        x = x.permute(0, 2, 3, 1).contiguous()
        # [B, H, W, C] -> [BHW, C]
        flat_x = x.reshape(-1, self.embedding_dim)

        encoding_indices = self.get_code_indices(flat_x)
        quantized = self.quantize(encoding_indices)
        quantized = quantized.view_as(x) # [B, H, W, C]

        if not self.training:
            quantized = quantized.permute(0, 3, 1, 2).contiguous()
            return quantized

        # embedding loss: move the embeddings towards the encoder's output
        q_latent_loss = F.mse_loss(quantized, x.detach())
        # commitment loss
        e_latent_loss = F.mse_loss(x, quantized.detach())
        loss = q_latent_loss + self.commitment_cost * e_latent_loss

        # Straight Through Estimator
        quantized = x + (quantized - x).detach()

        quantized = quantized.permute(0, 3, 1, 2).contiguous()
        return quantized, loss

    def get_code_indices(self, flat_x):
        # compute L2 distance
        distances = (
            torch.sum(flat_x ** 2, dim=1, keepdim=True) +
            torch.sum(self.embeddings.weight ** 2, dim=1) -
            2. * torch.matmul(flat_x, self.embeddings.weight.t())
        ) # [N, M]
        encoding_indices = torch.argmin(distances, dim=1) # [N,]
        return encoding_indices

    def quantize(self, encoding_indices):
        """Returns embedding tensor for a batch of indices."""
        return self.embeddings(encoding_indices)
```

对于encoder和decoder，我们采用对称的结构，其中decoder采用stride=2的反卷积来进行上采用：

```
class Encoder(nn.Module):
    """Encoder of VQ-VAE"""

    def __init__(self, in_dim=3, latent_dim=16):
        super().__init__()
```

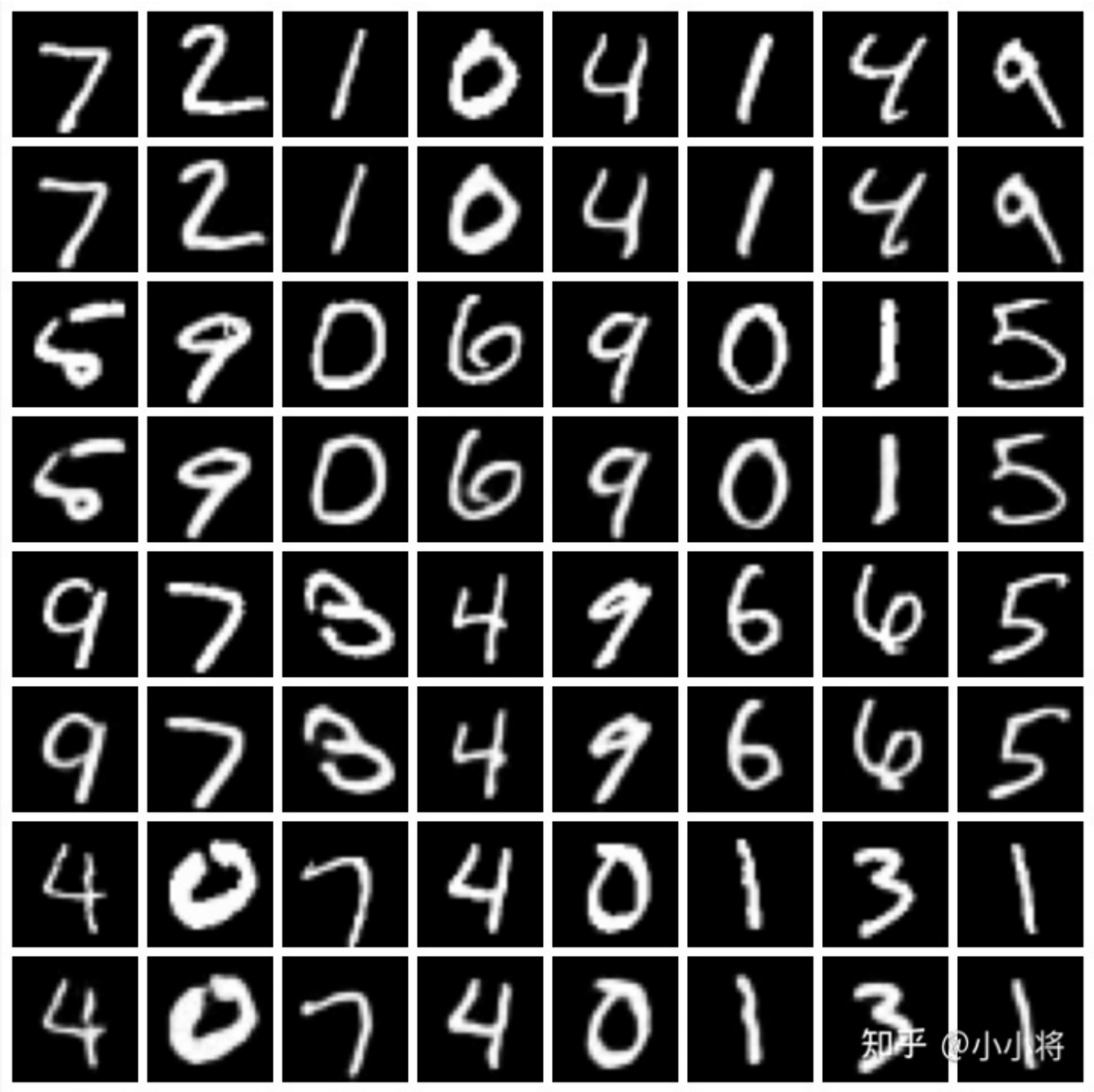
```
        self.in_dim = in_dim
        self.latent_dim = latent_dim

        self.convs = nn.Sequential(
            nn.Conv2d(in_dim, 32, 3, stride=2, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, 64, 3, stride=2, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, latent_dim, 1),
        )

    def forward(self, x):
        return self.convs(x)


class Decoder(nn.Module):
    """Decoder of VQ-VAE"""

    def __init__(self, out_dim=1, latent_dim=16):
        super().__init__()
        self.out_dim = out_dim
        self.latent_dim = latent_dim

        self.convs = nn.Sequential(
            nn.ConvTranspose2d(latent_dim, 64, 3, stride=2, padding=1, output_padding=1),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(64, 32, 3, stride=2, padding=1, output_padding=1),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(32, out_dim, 3, padding=1),
        )

    def forward(self, x):
        return self.convs(x)
```

有了VQ，encoder和decoder，就可以定义VQ-VAE模型了，这里也同样区分train和eval模式。此外，这里采用L2的重建误差，而且我们用训练数据的标准差来归一化这个误差。

```
class VQVAE(nn.Module):
    """VQ-VAE"""

    def __init__(self, in_dim, embedding_dim, num_embeddings, data_variance,
                 commitment_cost=0.25):
        super().__init__()
        self.in_dim = in_dim
        self.embedding_dim = embedding_dim
        self.num_embeddings = num_embeddings
```

```
        self.data_variance = data_variance

        self.encoder = Encoder(in_dim, embedding_dim)
        self.vq_layer = VectorQuantizer(embedding_dim, num_embeddings, commitment_cost)
        self.decoder = Decoder(in_dim, embedding_dim)

    def forward(self, x):
        z = self.encoder(x)
        if not self.training:
            e = self.vq_layer(z)
            x_recon = self.decoder(e)
            return e, x_recon

        e, e_q_loss = self.vq_layer(z)
        x_recon = self.decoder(e)

        recon_loss = F.mse_loss(x_recon, x) / self.data_variance

        return e_q_loss + recon_loss
```

下图为训练的VQ-VAE的模型在测试集上的重建效果，其中上面一行为原图，而下面一行对重建的图，可以看到VQ-VAE基本能完美重建MNIST数据集。

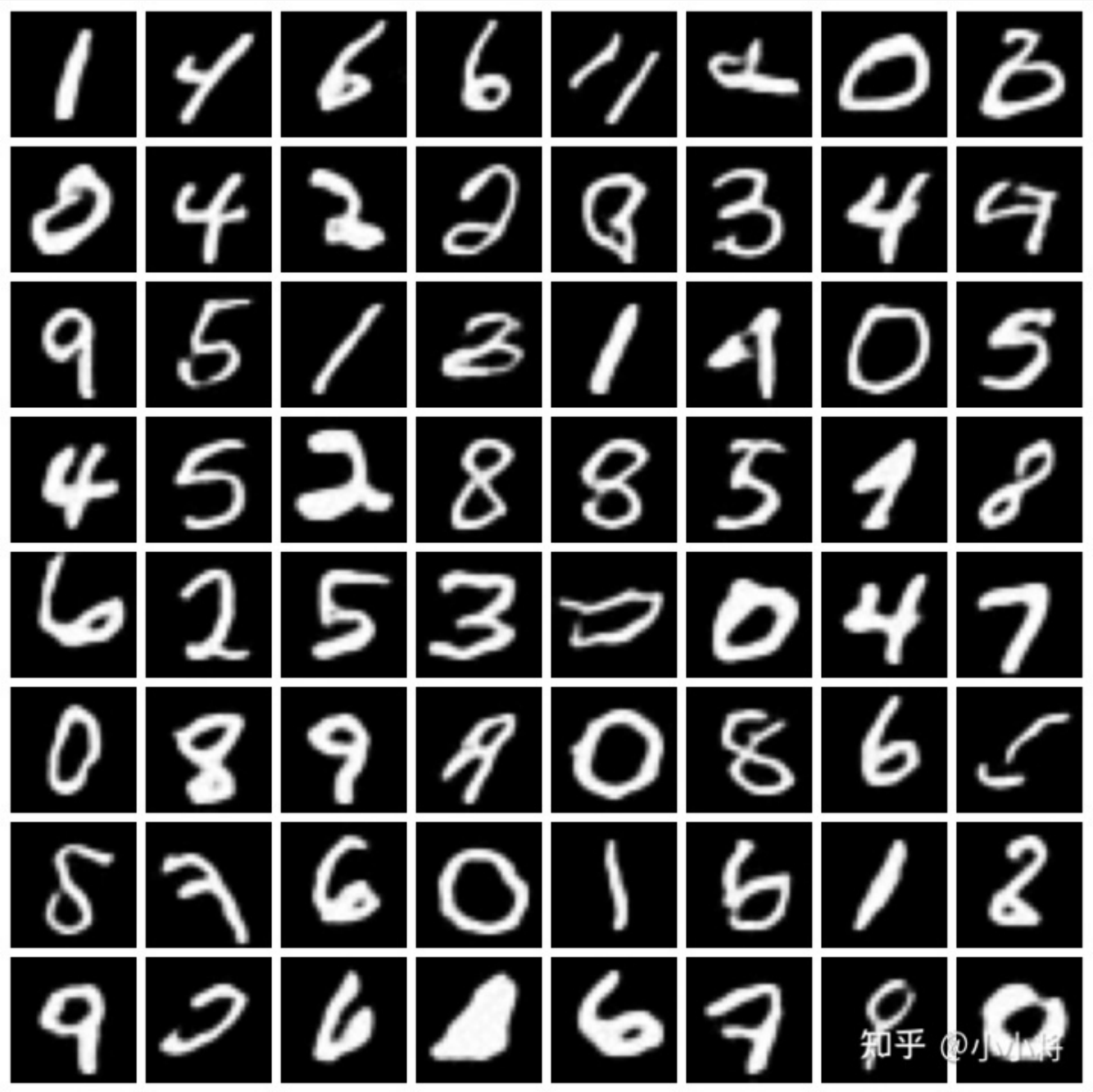


训练还VQ-VAE，我们还需要训练一个PixelCNN来学习先验，首先要用已经训练好的VQ-VAE来提取所有训练数据的离散编码，这将作为PixelCNN的训练样本：

```
# get encode_indices of training images
train_indices = []
for images, labels in train_loader:
    images = images - 0.5 # normalize to [-0.5, 0.5]
    images = images.cuda()
    with torch.inference_mode():
        z = model.encoder(images) # [B, C, H, W]
        b, c, h, w = z.size()
        # [B, C, H, W] -> [B, H, W, C]
        z = z.permute(0, 2, 3, 1).contiguous()
        # [B, H, W, C] -> [BHW, C]
        flat_z = z.reshape(-1, c)
        encoding_indices = model.vq_layer.get_code_indices(flat_z) # [BHW,]
```

```
        encoding_indices = encoding_indices.reshape(b, h, w)
        train_indices.append(encoding_indices.cpu())
```

这里我们采用GatedPixelCNN模型来学习先验，训练好先验模型后，可以先用先验模型随机采样得到离散编码，然后送入VQ-VAE的decoder得到生成的图像，下图为一些生成样例：



最后，我们来实现基于EMA的VQ-VAE版本，首先先实现一个EMA类，注意这里我们采用Adam优化算法采用的移动平均方式，与标准方式相比，还要除以$(1-\beta^t)$，这个可以防止移动平均值偏向初始值：

```
class ExponentialMovingAverage(nn.Module):
    """Maintains an exponential moving average for a value.

      This module keeps track of a hidden exponential moving average that is
      initialized as a vector of zeros which is then normalized to give the average.
      This gives us a moving average which isn't biased towards either zero or the
```

```
      initial value. Reference (https://arxiv.org/pdf/1412.6980.pdf)

      Initially:
          hidden_0 = 0
      Then iteratively:
          hidden_i = hidden_{i-1} - (hidden_{i-1} - value) * (1 - decay)
          average_i = hidden_i / (1 - decay^i)
    """

    def __init__(self, init_value, decay):
        super().__init__()

        self.decay = decay
        self.counter = 0
        self.register_buffer("hidden", torch.zeros_like(init_value))

    def forward(self, value):
        self.counter += 1
        self.hidden.sub_((self.hidden - value) * (1 - self.decay))
        average = self.hidden / (1 - self.decay ** self.counter)
        return average
```

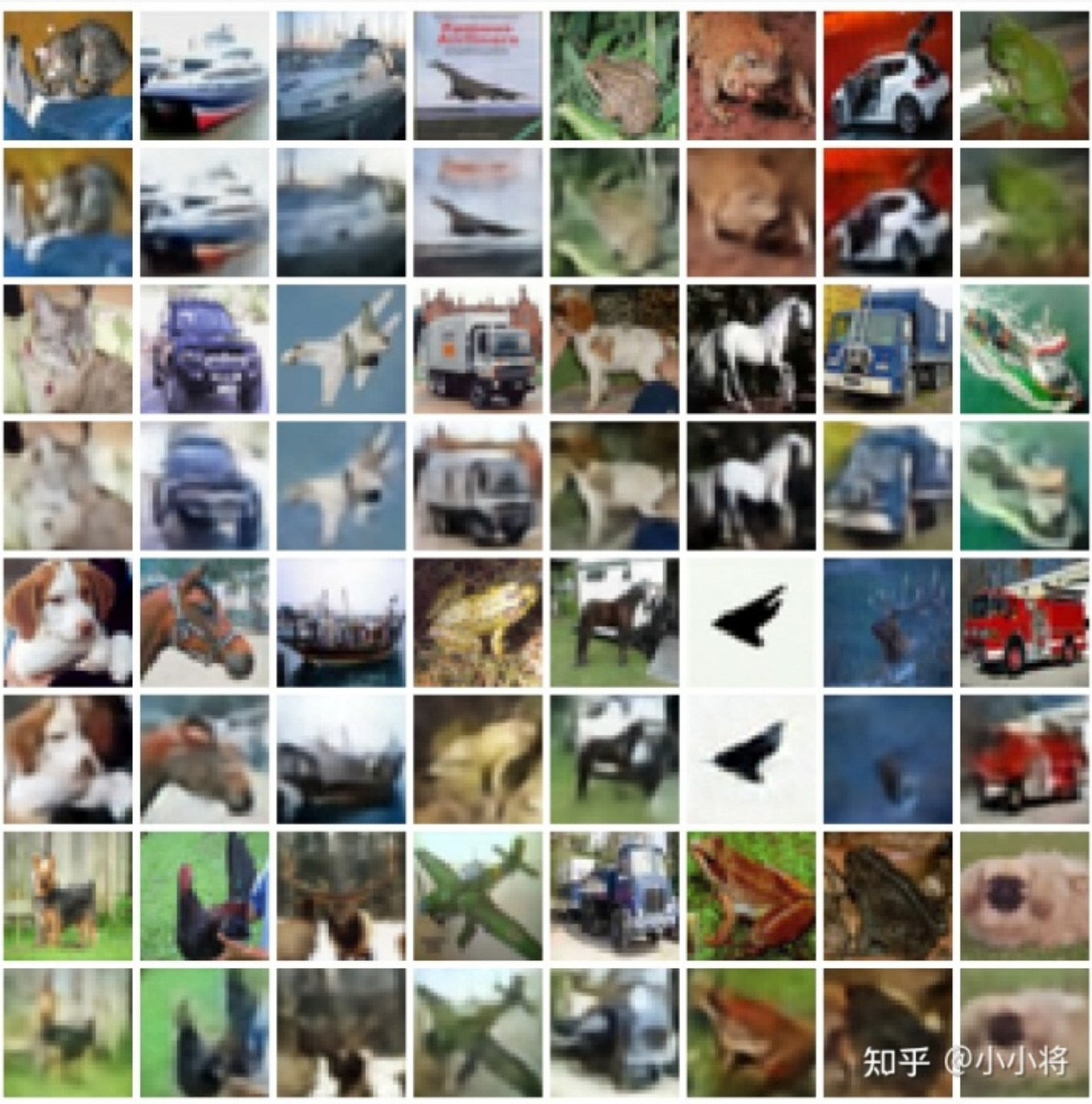
然后来实现基于EMA的VQ，这里维护了两个EMA参数，分别是每个embedding向量对应的encoder输出集合的特征数量以及特征之和，然后我们去掉原来的VQ loss而采用EMA来更新embeddings，注意这个过程要忽略梯度：

```
class VectorQuantizerEMA(nn.Module):
    """
    VQ-VAE layer: Input any tensor to be quantized. Use EMA to update embeddings.
    Args:
        embedding_dim (int): the dimensionality of the tensors in the
          quantized space. Inputs to the modules must be in this format as well.
        num_embeddings (int): the number of vectors in the quantized space.
        commitment_cost (float): scalar which controls the weighting of the loss terms (se
          equation 4 in the paper - this variable is Beta).
        decay (float): decay for the moving averages.
        epsilon (float): small float constant to avoid numerical instability.
    """
    def __init__(self, embedding_dim, num_embeddings, commitment_cost, decay,
                 epsilon=1e-5):
        super().__init__()
        self.embedding_dim = embedding_dim
        self.num_embeddings = num_embeddings
        self.commitment_cost = commitment_cost
        self.epsilon = epsilon

        # initialize embeddings as buffers
```

```
        embeddings = torch.empty(self.num_embeddings, self.embedding_dim)
        nn.init.xavier_uniform_(embeddings)
        self.register_buffer("embeddings", embeddings)
        self.ema_dw = ExponentialMovingAverage(self.embeddings, decay)

        # also maintain ema_cluster_size, which record the size of each embedding
        self.ema_cluster_size = ExponentialMovingAverage(torch.zeros((self.num_embeddings,


    def forward(self, x):
        # [B, C, H, W] -> [B, H, W, C]
        x = x.permute(0, 2, 3, 1).contiguous()
        # [B, H, W, C] -> [BHW, C]
        flat_x = x.reshape(-1, self.embedding_dim)

        encoding_indices = self.get_code_indices(flat_x)
        quantized = self.quantize(encoding_indices)
        quantized = quantized.view_as(x) # [B, H, W, C]

        if not self.training:
            quantized = quantized.permute(0, 3, 1, 2).contiguous()
            return quantized

        # update embeddings with EMA
        with torch.no_grad():
            encodings = F.one_hot(encoding_indices, self.num_embeddings).float()
            updated_ema_cluster_size = self.ema_cluster_size(torch.sum(encodings, dim=0))
            n = torch.sum(updated_ema_cluster_size)
            updated_ema_cluster_size = ((updated_ema_cluster_size + self.epsilon) /
                                        (n + self.num_embeddings * self.epsilon) * n)
            dw = torch.matmul(encodings.t(), flat_x) # sum encoding vectors of each cluste
            updated_ema_dw = self.ema_dw(dw)
            normalised_updated_ema_w = (
              updated_ema_dw / updated_ema_cluster_size.reshape(-1, 1))
            self.embeddings.data = normalised_updated_ema_w

        # commitment loss
        e_latent_loss = F.mse_loss(x, quantized.detach())
        loss = self.commitment_cost * e_latent_loss

        # Straight Through Estimator
        quantized = x + (quantized - x).detach()

        quantized = quantized.permute(0, 3, 1, 2).contiguous()
        return quantized, loss

    def get_code_indices(self, flat_x):
        # compute L2 distance
        distances = (
```

```
            torch.sum(flat_x ** 2, dim=1, keepdim=True) +
            torch.sum(self.embeddings ** 2, dim=1) -
            2. * torch.matmul(flat_x, self.embeddings.t())
        ) # [N, M]
        encoding_indices = torch.argmin(distances, dim=1) # [N,]
        return encoding_indices

    def quantize(self, encoding_indices):
        """Returns embedding tensor for a batch of indices."""
        return F.embedding(encoding_indices, self.embeddings)
```

这里用EMA版本的VQ-VAE在CIFAR10数据集训练之后在测试集上的重建效果：



以上具体的代码已经放在了GitHub上：**github.com/xiaohu2015/n...**。

小结

本文简单地介绍了VQ-VAE的原理以及具体的代码实现，相比VAE，VQ-VAE采用离散编码，这也使得VQ-VAE需要两个阶段来得到生成模型。最近OpenAI提出的文本转图像的生成模型DALL-E更让我们体会到了VQ-VAE的强大之处。

参考

- Neural Discrete Representation Learning
- Generating Diverse High-Fidelity Images with VQ-VAE-2
- github.com/deepmind/son...
- keras.io/examples/gener...
- github.com/AntixK/PyTor...
- avdnoord.github.io/home...

编辑于 2022-07-16 18:24