VAE、GAN 实现

蔚全爱

July 2025

1 VAE

- VAE：隐变量模型，随机采隐变量然后就可以生成新的数据
- 生成模型：潜在变量，先压缩成一个标准正态分布，然后再解压缩成数据分布，然后就可以生成了。
- 与扩散模型的离散版本很像，理解上：多步 VAE

1.1 VAE 理论

- 建模公式：$P_\theta(x) = \int P_0(x \mid z)\, P(z)\, dz$
- 最大似然估计：$\max \log P_\theta(x)$

$$
\begin{aligned}
\log p(x) &= \int q(z|x) \log \left(\frac{p(x,z)}{p(z|x)} \right) dz \\
&= \int q(z|x) \log \left(\frac{p(x,z)}{q(z|x)} \cdot \frac{q(z|x)}{p(z|x)} \right) dz \\
&= \int q(z|x) \log \left(\frac{p(x,z)}{q(z|x)} \right) dz + \mathrm{KL}(q(z|x) \| p(z|x))
\end{aligned}
$$

根据 Jensen 不等式（$\log \mathbb{E}[f(z)] \geq \mathbb{E}[\log f(z)]$）可得：

$$
\log p(x) \geq \int q(z|x) \log \left(\frac{p(x,z)}{q(z|x)} \right) dz = \mathbb{E}_{q(z|x)} \left[\log \frac{p(x,z)}{q(z|x)} \right]
$$

变分下界（ELBO）：

$$
\mathrm{ELBO}(q) = \mathbb{E}_{q(z|x)} \left[\log p(x,z) - \log q(z|x) \right]
$$

$\log p(x)$ 就变成了：

$$
\log p(x) = \mathrm{ELBO}(q) + \mathrm{KL}(q(z|x) \| p(z|x))
$$

目标是最大化 ELBO：

$$
\begin{aligned}
\mathcal{L} &= \int q(z|x) \log \frac{p(x|z)p(z)}{q(z|x)} \, dz \\
&= \int q(z|x) \log \frac{p(z)}{q(z|x)} \, dz + \int q(z|x) \log p(x|z) \, dz \\
&= -\mathrm{KL}(q(z|x) \,\|\, p(z)) + \mathbb{E}_{q(z|x)}[\log p(x|z)]
\end{aligned}
$$

- 一部分是编码器的损失，一部分是解码器的损失
- 希望潜变量分布是简单的：标准正态分布

$$
p(z) : \mathcal{N}(0, 1)
$$

- 变分分布

$$
q_\phi(z|x) : \mathcal{N}(\mu_\phi, \sigma_\phi^2)
$$

- 重参数化技巧：

$$
\begin{aligned}
z &= \mu_\phi + \sigma_\phi \odot \epsilon \\
\epsilon &\sim \mathcal{N}(0, I)
\end{aligned}
$$

- -KL = $1/2 \sum(\mu^2 + \sigma^2 - \log \sigma^2 - 1)$
- 解码器，积分不好计算：蒙特卡洛采样

$$
\mathbb{E}_{q_\phi(z|x)}\left[P_\theta(x|z)\right] \approx \frac{1}{L} \sum_{\ell=1}^{L} \log P_\theta(x \mid z^{(\ell)}), \quad \text{where } z^{(\ell)} \sim q_\phi(z|x)
$$

- 不同的假设：1. $P_\theta(x|z)$ 是高斯分布，方差一般就假设为 1 了，方便计算，导出来平方损失

$$
\frac{1}{L} \sum_{\ell=1}^{L} \left\| x - \hat{x}(z^{(\ell)}) \right\|^2
$$

  实际执行：L=1 因为 batch 本身提供了多个样本

  2. $P_\theta(x|z)$ 是 Bernoulli 分布 (不常用):

$$
P_\theta(x|z) = \text{Bernoulli}(\hat{x}(z)),
$$

  其中 $\hat{x}(z)$ 是一个概率向量，表示每个像素为 1 的概率。对数似然函数为：

$$
\log P_\theta(x|z) = \sum_{i=1}^{D} \left[x_i \log \hat{x}_i(z) + (1 - x_i) \log(1 - \hat{x}_i(z)) \right].
$$

  因此，最大化对数似然等价于最小化 Binary Cross-Entropy 损失：

$$
\mathcal{L}_{\text{recon}} = -\mathbb{E}_{q_\phi(z|x)}\left[\log P_\theta(x|z)\right] = -\sum_{i=1}^{D} \left[x_i \log \hat{x}_i(z) + (1 - x_i) \log(1 - \hat{x}_i(z)) \right].
$$

1.2 VAE 实现

- 编码器：x 到 z，只关注到 μ_ϕ 和 σ_ϕ。
- 解码器：z 到 $\hat{x}$。
- 只需要输出均值和 log 方差，这里是因为网络输出一般是实数，z 的概率分布
- 重参数化技巧：$z = \mu + \sigma \odot \epsilon$，$\epsilon$
- https://github.com/AntixK/PyTorch-VAE/blob/master/models/vanilla_vae.py
- 条件生成模型，实现的时候 Z 分成两项，一个是随机噪声，一个是条件的

```
import torch
from models import BaseVAE
from torch import nn
from torch.nn import functional as F
from .types_ import *


class VanillaVAE(BaseVAE):


    def __init__(self,
                 in_channels: int,
                 latent_dim: int,
                 hidden_dims: List = None,
                 **kwargs) -> None:
        super(VanillaVAE, self).__init__()

        self.latent_dim = latent_dim

        modules = []
        if hidden_dims is None:
            hidden_dims = [32, 64, 128, 256, 512]

        # Build Encoder
        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)
        self.fc_mu = nn.Linear(hidden_dims[-1]*4, latent_dim)
        self.fc_var = nn.Linear(hidden_dims[-1]*4, latent_dim)


        # Build Decoder
        modules = []

        self.decoder_input = nn.Linear(latent_dim,
            hidden_dims[-1] * 4)

        hidden_dims.reverse()

        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())
            )



        self.decoder = nn.Sequential(*modules)

        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())

    def encode(self, input: Tensor) -> List[Tensor]:
        """
        Encodes the input by passing through the encoder
```

```
            network
        and returns the latent codes.
        :param input: (Tensor) Input tensor to encoder [N x C
            x H x W]
        :return: (Tensor) List of latent codes
        """
        result = self.encoder(input)
        result = torch.flatten(result, start_dim=1)

        # Split the result into mu and var components
        # of the latent Gaussian distribution
        mu = self.fc_mu(result)
        log_var = self.fc_var(result)

        return [mu, log_var]

    def decode(self, z: Tensor) -> Tensor:
        """
        Maps the given latent codes
        onto the image space.
        :param z: (Tensor) [B x D]
        :return: (Tensor) [B x C x H x W]
        """
        result = self.decoder_input(z)
        result = result.view(-1, 512, 2, 2)
        result = self.decoder(result)
        result = self.final_layer(result)
        return result

    def reparameterize(self, mu: Tensor, logvar: Tensor) ->
        Tensor:
        """
        Reparameterization trick to sample from N(mu, var) from
        N(0,1).
        :param mu: (Tensor) Mean of the latent Gaussian [B x D]
        :param logvar: (Tensor) Standard deviation of the
            latent Gaussian [B x D]
        :return: (Tensor) [B x D]
        """
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return eps * std + mu

    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]

    def loss_function(self,
                      *args,
```

```
                    **kwargs) -> dict:
        """
        Computes the VAE loss function.
        KL(N(\mu, \sigma), N(0, 1)) = \log \frac{1}{\sigma} +
            \frac{\sigma^2 + \mu^2}{2} - \frac{1}{2}
        :param args:
        :param kwargs:
        :return:
        """
        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()}

    def sample(self,
               num_samples:int,
               current_device: int, **kwargs) -> Tensor:
        """
        Samples from the latent space and return the
            corresponding
        image space map.
        :param num_samples: (Int) Number of samples
        :param current_device: (Int) Device to run the model
        :return: (Tensor)
        """
        z = torch.randn(num_samples,
                        self.latent_dim)

        z = z.to(current_device)

        samples = self.decode(z)
        return samples

    def generate(self, x: Tensor, **kwargs) -> Tensor:
        """
        Given an input image x, returns the reconstructed image
        :param x: (Tensor) [B x C x H x W]
```

```
        :return: (Tensor) [B x C x H x W]
        """

        return self.forward(x)[0]
```

2 GAN

- GAN：生成对抗网络，生成器和判别器对抗训练，交替优化，固定判别器，训练生成器，欺骗判别器，固定生成器，训练判别器，使得 gap 尽量大
- min-max 内层优化与外层优化
- 元学习中内层优化与外层优化是非常相关的，冻结判别器，训练生成器，冻结生成器，训练判别器
- GAN 的目标函数：

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

- conditional GAN：条件生成对抗网络，输入条件信息猫与狗，生成的图像与条件相关

2.1 GAN 实现

```
 import argparse
import os
import numpy as np
import math

import torchvision.transforms as transforms
from torchvision.utils import save_image

from torch.utils.data import DataLoader
from torchvision import datasets
from torch.autograd import Variable

import torch.nn as nn
import torch.nn.functional as F
import torch

os.makedirs("images", exist_ok=True)

parser = argparse.ArgumentParser()
parser.add_argument("--n_epochs", type=int, default=200,
    help="number of epochs of training")
parser.add_argument("--batch_size", type=int, default=64,
    help="size of the batches")
```

```
parser.add_argument("--lr", type=float, default=0.0002,
    help="adam: learning rate")
parser.add_argument("--b1", type=float, default=0.5,
    help="adam: decay of first order momentum of gradient")
parser.add_argument("--b2", type=float, default=0.999,
    help="adam: decay of first order momentum of gradient")
parser.add_argument("--n_cpu", type=int, default=8,
    help="number of cpu threads to use during batch generation")
parser.add_argument("--latent_dim", type=int, default=100,
    help="dimensionality of the latent space")
parser.add_argument("--n_classes", type=int, default=10,
    help="number of classes for dataset")
parser.add_argument("--img_size", type=int, default=32,
    help="size of each image dimension")
parser.add_argument("--channels", type=int, default=1,
    help="number of image channels")
parser.add_argument("--sample_interval", type=int,
    default=400, help="interval between image sampling")
opt = parser.parse_args()
print(opt)

img_shape = (opt.channels, opt.img_size, opt.img_size)

cuda = True if torch.cuda.is_available() else False


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

        self.label_emb = nn.Embedding(opt.n_classes,
            opt.n_classes)

        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 + opt.n_classes, 128,
                normalize=False),
            *block(128, 256),
            *block(256, 512),
            *block(512, 1024),
            nn.Linear(1024, int(np.prod(img_shape))),
            nn.Tanh()
        )
```

```
    def forward(self, noise, labels):
        # Concatenate label embedding and image to produce
            input
        gen_input = torch.cat((self.label_emb(labels), noise),
            -1)
        img = self.model(gen_input)
        img = img.view(img.size(0), *img_shape)
        return img


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

        self.label_embedding = nn.Embedding(opt.n_classes,
            opt.n_classes)

        self.model = nn.Sequential(
            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),
        )

    def forward(self, img, labels):
        # Concatenate label embedding and image to produce
            input
        d_in = torch.cat((img.view(img.size(0), -1),
            self.label_embedding(labels)), -1)
        validity = self.model(d_in)
        return validity


# Loss functions
adversarial_loss = torch.nn.MSELoss()

# Initialize generator and discriminator
generator = Generator()
discriminator = Discriminator()

if cuda:
    generator.cuda()
    discriminator.cuda()
    adversarial_loss.cuda()
```

```
# Configure data loader
os.makedirs("../../data/mnist", exist_ok=True)
dataloader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "../../data/mnist",
        train=True,
        download=True,
        transform=transforms.Compose(
            [transforms.Resize(opt.img_size),
                transforms.ToTensor(),
                transforms.Normalize([0.5], [0.5])]
        ),
    ),
    batch_size=opt.batch_size,
    shuffle=True,
)

# Optimizers
optimizer_G = torch.optim.Adam(generator.parameters(),
    lr=opt.lr, betas=(opt.b1, opt.b2))
optimizer_D = torch.optim.Adam(discriminator.parameters(),
    lr=opt.lr, betas=(opt.b1, opt.b2))

FloatTensor = torch.cuda.FloatTensor if cuda else
    torch.FloatTensor
LongTensor = torch.cuda.LongTensor if cuda else
    torch.LongTensor


def sample_image(n_row, batches_done):
    """Saves a grid of generated digits ranging from 0 to
        n_classes"""
    # Sample noise
    z = Variable(FloatTensor(np.random.normal(0, 1, (n_row **
        2, opt.latent_dim))))
    # Get labels ranging from 0 to n_classes for n rows
    labels = np.array([num for _ in range(n_row) for num in
        range(n_row)])
    labels = Variable(LongTensor(labels))
    gen_imgs = generator(z, labels)
    save_image(gen_imgs.data, "images/%d.png" % batches_done,
        nrow=n_row, normalize=True)


# ----------
#  Training
# ----------

for epoch in range(opt.n_epochs):
```

```python
for i, (imgs, labels) in enumerate(dataloader):

    batch_size = imgs.shape[0]

    # Adversarial ground truths
    valid = Variable(FloatTensor(batch_size,
        1).fill_(1.0), requires_grad=False)
    fake = Variable(FloatTensor(batch_size, 1).fill_(0.0),
        requires_grad=False)

    # Configure input
    real_imgs = Variable(imgs.type(FloatTensor))
    labels = Variable(labels.type(LongTensor))

    # -----------------
    #  Train Generator
    # -----------------

    optimizer_G.zero_grad()

    # Sample noise and labels as generator input
    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)))

    # Generate a batch of images
    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()

    # ---------------------
    #  Train Discriminator
    # ---------------------

    optimizer_D.zero_grad()

    # Loss for real images
    validity_real = discriminator(real_imgs, labels)
    d_real_loss = adversarial_loss(validity_real, valid)

    # Loss for fake images
    validity_fake = discriminator(gen_imgs.detach(),
        gen_labels)
```

```
        d_fake_loss = adversarial_loss(validity_fake, fake)

        # Total discriminator loss
        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
        if batches_done % opt.sample_interval == 0:
            sample_image(n_row=10, batches_done=batches_done)
```