10 Generative Models in Healthcare

So far, we have learned about basic neural networks, namely DNN, RNN, CNN, and GNN. These networks are designed to extract feature information from various data types like structured EHR data, clinical notes, medical images, and physiological signals. They predict targets of interest such as disease diagnosis or medication recommendation. These models are supervised, and one needs sufficient annotated data (i.e., labeled data) for training. The output of these models is usually simple such as binary labels of disease presence/absence or numerical values like length of hospital stay.

In this chapter, we will learn about **generative models**, a new type of neural network that generates realistic data from scratch. These models can produce new images, videos, and sequences. The output of generative models is more complex (i.e., a high dimensional data sample) compared to supervised models (i.e., a scalar). After model training, generative models can create realistic new data based on random input or predefined input labels. The training process usually doesn't require annotated labeled data but utilizes cleverly constructed supervision from the data alone.

Generative models have the potential to revolutionize healthcare in various ways, such as personalized medicine [116], medical imaging [117], drug discovery [118], clinical trials [119, 120], rare diseases [121], genomics [122], electronic health records (EHR) [123], EEG [124] and ECG [125] analysis. For example, medical datasets are often sensitive due to privacy and legal constraints and cannot be made public. With generative models, it is possible to publish synthetic but realistic data that have similar distributions as private data, while still meeting privacy and legal compliance [123]. The newly generated data can also serve to augment the original datasets, thereby enriching the volume and diversity of the original datasets.

In recent years, significant advancements have been made in generative models in the fields of vision and natural language processing. OpenAI released a powerful question-answering dialog system called GPT-3.5[1] in late 2022, followed by GPT-4 in 2023. These large language models (LLMs) have revolutionized research in these fields, as well as many industries. By querying the LLM interface, one can get decent and human-like text or code answers. Since then, many open-sourced LLM pre-trained models have been released, such as Llama 2[2] from Meta, which can be fine-tuned for various downstream language or vision-language tasks. In addition, diffusion models

[1] https://platform.openai.com/docs/models, GPT stands for Generated Pretrained Transformer
[2] https://llama.meta.com/llama2

have impressed many people in the field of image generation. These models translate prompt text into images with various styles, such as Stable Diffusion 2.1[3] from Stability and Midjourney[4]. The generated images or videos are difficult to distinguish from real ones. Recently, OpenAI's popular Sora[5] can even generate 60-second videos from text prompts without post-hoc editing.

Now, let's learn about generative models. This chapter will introduce three generative models: Generative Adversarial Networks (GAN), developed by Goodfellow et al. [126, 127], Variational Auto-Encoder (VAE), proposed by Kingma et al. [128], and recently popular Diffusion Models [129, 130]. Towards the end of the chapter, we will demonstrate how to use PyHealth to implement GAN, VAE, and latent diffusion models (LDM) to generate synthetic chest X-ray images.

10.1 Variational Auto-encoder (VAE)

We introduce Variational Auto-encoder (VAE) in this section. As the background, let us start with the auto-encoder (AE) models proposed by Hinton and Salakhutdinov [131] in 2006. AE models use the *bottleneck layer* between encoder and decoder to learn compressed low-dimensional features for complex high-dimensional data. Note that AE models are not generative models but a nonlinear generalization of principal component analysis (PCA).

Motivated from AE, variational auto-encoder (VAE), proposed by Kingma et al [128] in 2013, follows a similar AE architecture which further imposes distribution constraints for the *bottleneck layer*. Different from AE and other AE variants, VAE model can generate new data that are similar to the original training data.

10.1.1 Simple auto-encoder (AE)

Auto-encoder (AE) is an unsupervised low-dimensional feature learning model. Like principle component analysis (PCA) or matrix factorization (MF), AE learns to compress complex data into a low-dimensional representation with non-linear functions (while PCA and MF use linear transformation). For example, given thousands of chest X-ray images, the AE model could learn to obtain meaningful image vector representations without label annotations. It mainly optimizes the reconstruction loss.

Encoder-decoder architecture

In Figure 10.1, the AE model consists of two main components: the *encoder* and the *decoder*. The encoder receives the input data (in this example, a chest X-ray image) and produces a low-dimensional embedding (also known as the bottleneck layer). The decoder takes this low-dimensional embedding and attempts to reconstruct the original

[3] https://huggingface.co/spaces/stabilityai/stable-diffusion
[4] https://docs.midjourney.com/
[5] https://openai.com/sora

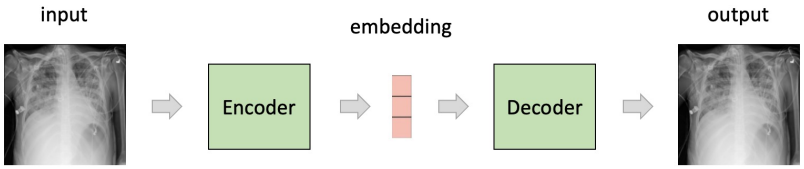

Figure 10.1 Illustration of an auto-encoder (AE) model

image. The entire model is trained using either the mean square error (MSE) or KL-divergence between the original and reconstructed images. Figure 10.2 provides an illustration of the process.

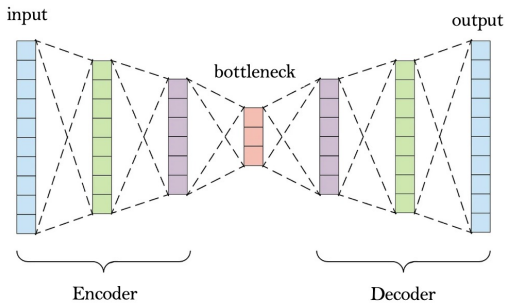

Figure 10.2 Illustration of auto-encoder (AE) architecture

Let's define the following notations: $\mathbf{X}$ represents the original input data, Encoder$(\cdot)$ is the encoder model, $\mathbf{h}$ is the bottleneck layer, Decoder$(\cdot)$ is the decoder model, and $\tilde{\mathbf{X}}$ is the reconstructed image. The encoder model can be any type of model, such as a convolutional neural network for images. It takes the original data as input and generates an embedding $\mathbf{h}$, such as a 256-dimensional vector. In other applications, like modeling electronic health record (EHR) data, the encoder can be an RNN model that models a sequence of patient visits. Formally, the encoder can be represented as:

$$\mathbf{h} = \text{Encoder}(\mathbf{X}). \tag{10.1}$$

The decoder model takes the bottleneck embedding $\mathbf{h}$ as input and generates a reconstructed data sample. In this example for X-ray data, the decoder uses transposed convolutions to perform the opposite operation of the encoder, such as `torch.nn.ConvTranspose2d`[6]. Starting from the bottleneck embedding $\mathbf{h}$, it gradually upscales the representation back to the original input size. Transposed convolutional layers help in this process by increasing the spatial resolution of their input feature maps, effectively "reversing" the effect of convolutional layers used in the encoder.

[6] https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html

$$\tilde{\mathbf{X}} = \text{Decoder}(\mathbf{h}). \tag{10.2}$$

Note that, in AE models, the encoder or the decoder can have multiple layers. The reconstruction is the MSE loss.

$$\mathcal{L} = \|\mathbf{X} - \tilde{\mathbf{X}}\|_F^2. \tag{10.3}$$

```python
# Assume x is a chest X-ray image

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

        # two-layer DNN as encoder
        self.encoder1 = nn.Linear(28*28, 128)
        self.encoder2 = nn.Linear(128, 32)

        # two-layer DNN as decoder
        self.decoder1 = nn.Linear(32, 128)
        self.decoder2 = nn.Linear(128, 28*28)

    def encoder(self, x):
        h = self.encoder2(torch.relu(self.encoder1(x)))
        return h

    def decoder(self, h):
        x_hat = self.decoder2(torch.relu(self.decoder1(h)))
        return x_hat

    def forward(self, x):
        bottleneck = self.encoder(x)
        x_hat = self.decoder(torch.relu(bottleneck))
        # the input, reconstructed images, and the bottleneck
        return x, x_hat, bottleneck
```

In the code, both the encoder and the decoder use a two-layer deep neural network with ReLU as the activation.

Downstream Applications: Afterward, the encoder model can be used as pre-trained encoders in some downstream tasks, such as building a classification layer on top of the encoder for supervised learning tasks.

10.1.2 Variants of AE models

Simple AE models can easily overfit the training data, especially on small data sets. Several follow-up works are proposed to address the overfitting issue in AE. We briefly introduce some of those AE variants.

Denoising autoencoder

Vincent et al. proposed the de-noising auto-encoder (DAE) in 2008 [132]. The DAE model works by adding random noise to the input data and then requiring the auto-

encoder model to reconstruct the original data samples. This approach is based on the observation that humans are able to recognize objects or scenes even when they are partially occluded or corrupted [133]. The DAE architecture is similar to that of the AE architecture, with the only difference being that the input data is the noise-corrupted sample and the model still aims to reconstruct the original data. Some common noises include Gaussian noise and salt-and-pepper noise (i.e., setting a small percentage of pixels to minimum or maximum values). Check out Figure 10.3 for a visual representation of the DAE architecture. In healthcare, DAEs have been used to model patient records to create a latent patient representation [134].

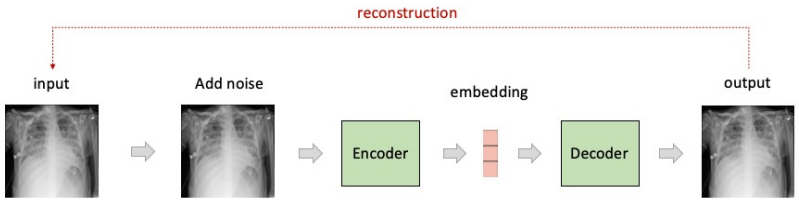

Figure 10.3 Illustration of de-noising auto-encoder (DAE) architecture

Sparse autoencoder

Adding "sparse" constraint on the hidden activation could also mitigate the AE overfitting problem and improve model robustness. Makhzani et al. [135] proposed the sparse auto-encoder (SAE), which forces the model to only keep k largest hidden units at the same time and set all other hidden units to zero. Mathematically, the SAE optimizes the following objective function:

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{X}^{(i)}, \hat{\mathbf{X}}^{(i)}) + \Omega(\mathbf{h}^{(i)})$$

where L is the reconstruction loss, Ω is a sparsity penalty term, and $\mathrm{h}^{(i)}$ is the hidden representation of the i-th sample. The sparsity penalty Ω encourages the hidden units to be sparse, i.e., only a small number of units are active for each input sample.

10.1.3 Variational Auto-encoder (VAE)

The autoencoder (AE) architecture, while powerful for learning compact representations of complex data, cannot generate new samples from the learned latent distribution. This limitation is addressed by the variational autoencoder (VAE), introduced by [128] in 2014, which builds upon the AE structure but introduces randomness and additional distribution constraints on the middle bottleneck layer.

Specifically, VAE enforces the bottleneck layer to follow a predefined standard distribution, such as the *multi-variate Gaussian distribution*. By doing so, VAE enables

the generation of new samples by sampling noise from the Gaussian distribution and using the decoder to map these noise to the data space.

Assumption in bottleneck layer distribution

For a d-dimensional bottleneck layer, usually a multi-variate Gaussian distribution needs d parameters for the mean vector μ and $d \times d$ parameters for the co-variance matrix Σ. In VAE, the authors assume each dimension in the multi-variate Gaussian distribution be independent, which leads to a diagonal co-variance matrix, thereby reducing its number of parameters from $d \times d$ to d diagonal elements. This makes the model more efficient and easier to train.

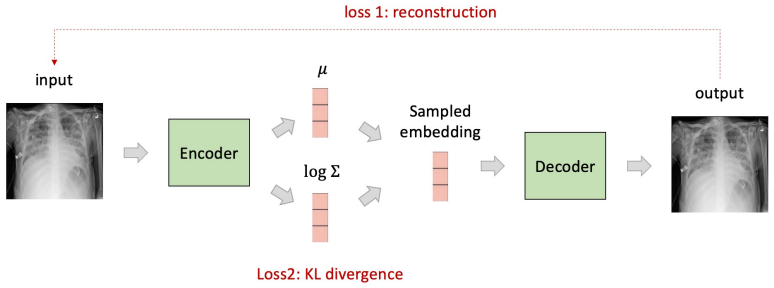

Figure 10.4 Illustration of variational auto-encoder (VAE) architecture. It has two objective functions: one is the MSE reconstruction loss between the original figures and the reconstructed figures, and another one is the KL-divergence between the standard normal distribution and the parameterized normal distribution.

An illustration of VAE is shown in Figure 10.4. We elaborate its forward propagation process below and explain the loss function later.

- **Step 1**: Given the high-dimensional input $\mathbf{X}$, the encoder of VAE first maps it to a mean vector $\mu \in \mathbb{R}^d$ and a co-variance diagonal vector $\log \Sigma \in \mathbb{R}^d$. Different from AE that directly map $\mathbf{X}$ to the bottleneck layer, VAE essentially builds a parameterized multi-variate normal distribution at this step. Here, $\log \Sigma$ is a vector that contains the log-transformed diagonal elements from Σ (since diagonals of Σ is non-negative, we use log transformation).

$$\mu = \text{Encoder}_{\mu}(\mathbf{X}), \tag{10.4}$$

$$\log \Sigma = \text{Encoder}_{\Sigma}(\mathbf{X}). \tag{10.5}$$

- **Step 2**: Obtaining μ and $\log \Sigma$, we first apply a exp() on $\log \Sigma$ to restore the co-variance as Σ. Next, we want to get a sample from this parameterized distribution (μ, Σ). However, directly sampling from there is in-differentiable and can break the loss back-propagation. The authors introduce a **re-parameterization trick** that samples a d-dimensional vector ϵ, where each element is independently sampled from a standard normal distribution. Then, re-scaling and translating ϵ

would generate a bottleneck embedding $\mathbf{h} \in \mathbb{R}^d$ that follows the parameterized distribution (μ, Σ), and importantly $\mathbf{h}$ is differentiable.

$$\epsilon_i \sim \mathcal{N}(0,1), \quad i \in [0,1,...,d-1], \tag{10.6}$$

$$\mathbf{h} = \mu + \epsilon \odot \exp(\log \Sigma)^{\frac{1}{2}} = \mu + \epsilon \odot \Sigma^{\frac{1}{2}}. \tag{10.7}$$

Here, $\odot$ is the vector element-wise product.

- **Step 3**: Afterwards, the model uses a decoder to reconstruct the sample $\mathbf{X}$.

$$\tilde{\mathbf{X}} = \text{Decoder}(\mathbf{h}). \tag{10.8}$$

Objective functions

Now, we explain the loss function below, which contains two parts: one is the reconstruction loss, and another part is the distribution gap between (μ, Σ) and standard multi-variate Gaussian (where dimensions are all independent).

Optimizing the reconstruction loss. Similar to the AE model, reconstruction loss is defined by the MSE loss.

$$\mathcal{L}_1 = \|\mathbf{X} - \tilde{\mathbf{X}}\|_F^2. \tag{10.9}$$

Optimize KL-divergence loss. The new constraints on VAE is that on the middle bottleneck layer $\mathbf{h}$, we want the parameterized distribution follows standard Gaussian and use Kullback–Leibler (KL) divergence [7] to measure the distribution gap.

Specifically, we denote the standard Gaussian as $p(\mathbf{h})$, which can be written down as

$$\mathcal{N}\left(\begin{bmatrix}0\\0\\\vdots\\0\end{bmatrix}, \begin{bmatrix}1 & 0 & \cdots & 0\\0 & 1 & \cdots & 0\\\vdots & \vdots & \ddots & \vdots\\0 & 0 & \cdots & 1\end{bmatrix}\right). \tag{10.10}$$

Its probability density function (PDF) is

$$p(\mathbf{h}) = \frac{1}{(2\pi)^{\frac{d}{2}}} exp\left(-\frac{\mathbf{h}^\top \mathbf{h}}{2}\right). \tag{10.11}$$

Our parameterized Gaussian $\mathcal{N}(\mu, \Sigma)$ is denoted as $q(\mathbf{h} \mid \mu, \Sigma)$, and here μ and Σ contain parameters. The PDF of this distribution could be written down as

$$q(\mathbf{h} \mid \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{d}{2}} det(\Sigma)^{\frac{1}{2}}} \exp\left(-\frac{(\mathbf{h}-\mu)^\top \Sigma^{-1}(\mathbf{h}-\mu)}{2}\right). \tag{10.12}$$

[7] https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence

The KL-divergence loss can be derived analytically,

$$\int_{\mathbf{h}\sim q} \ln \frac{p(\mathbf{h})}{q(\mathbf{h} \mid \mu, \Sigma)} d\mathbf{h} = \int q(\mathbf{h} \mid \mu, \Sigma) \left(\ln p(\mathbf{h}) - \ln q(\mathbf{h} \mid \mu, \Sigma)\right) d\mathbf{h} \tag{10.13}$$

$$= \underbrace{\int q(\mathbf{h} \mid \mu, \Sigma) \ln p(\mathbf{h}) d\mathbf{h}}_{PartI} - \underbrace{\int q(\mathbf{h} \mid \mu, \Sigma) \ln q(\mathbf{h} \mid \mu, \Sigma) d\mathbf{h}}_{PartII} \tag{10.14}$$

Here, we decompose the integral into two parts. Also, we know that $\mathbf{h} = \mu + \epsilon \odot \Sigma^{\frac{1}{2}}$. For Part I,

$$\int q(\mathbf{h} \mid \mu, \Sigma) \ln p(\mathbf{h}) d\mathbf{h} \tag{10.15}$$

$$= \int \frac{1}{(2\pi)^{\frac{d}{2}} det(\Sigma)^{\frac{1}{2}}} \exp\left(-\frac{(\mathbf{h}-\mu)^\top \Sigma^{-1} (\mathbf{h}-\mu)}{2}\right) \ln \frac{1}{(2\pi)^{\frac{d}{2}}} exp\left(-\frac{\mathbf{h}^\top \mathbf{h}}{2}\right) d\mathbf{h} \tag{10.16}$$

$$= \int \frac{1}{(2\pi)^{\frac{d}{2}} det(\Sigma)^{\frac{1}{2}}} \exp\left(-\frac{(\mathbf{h}-\mu)^\top \Sigma^{-1} (\mathbf{h}-\mu)}{2}\right) \left(-\frac{d}{2}\ln(2\pi) - \frac{\mathbf{h}^\top \mathbf{h}}{2}\right) d\mathbf{h} \tag{10.17}$$

$$= -\frac{d}{2}\ln(2\pi) \int \frac{1}{(2\pi)^{\frac{d}{2}} det(\Sigma)^{\frac{1}{2}}} \exp\left(-\frac{(\mathbf{h}-\mu)^\top \Sigma^{-1} (\mathbf{h}-\mu)}{2}\right) d\mathbf{h} \tag{10.18}$$

$$- \int \frac{\mathbf{h}^\top \mathbf{h}}{2} \frac{1}{(2\pi)^{\frac{d}{2}} det(\Sigma)^{\frac{1}{2}}} \exp\left(-\frac{(\mathbf{h}-\mu)^\top \Sigma^{-1} (\mathbf{h}-\mu)}{2}\right) d\mathbf{h} \tag{10.19}$$

$$= -\frac{d}{2}\ln(2\pi) - (\mu^\top \mu + \|\Sigma\|_F^2). \tag{10.20}$$

For Part II, we have

$$\int q(\mathbf{h} \mid \mu, \Sigma) \ln q(\mathbf{h} \mid \mu, \Sigma) d\mathbf{h} \tag{10.21}$$

$$= \int \frac{1}{(2\pi)^{\frac{d}{2}} det(\Sigma)^{\frac{1}{2}}} \exp\left(-\frac{(\mathbf{h}-\mu)^\top \Sigma^{-1} (\mathbf{h}-\mu)}{2}\right) \ln \frac{1}{(2\pi)^{\frac{d}{2}} det(\Sigma)^{\frac{1}{2}}} \exp\left(-\frac{(\mathbf{h}-\mu)^\top \Sigma^{-1} (\mathbf{h}-\mu)}{2}\right) d\mathbf{h} \tag{10.22}$$

$$= \int \frac{1}{(2\pi)^{\frac{d}{2}} det(\Sigma)^{\frac{1}{2}}} \exp\left(-\frac{(\mathbf{h}-\mu)^\top \Sigma^{-1} (\mathbf{h}-\mu)}{2}\right) \left(-\frac{d}{2}\ln(2\pi) - \frac{1}{2}\ln(det(\Sigma)) - \frac{(\mathbf{h}-\mu)^\top \Sigma^{-1} (\mathbf{h}-\mu)}{2}\right) d\mathbf{h} \tag{10.23}$$

$$= -\left(\frac{d}{2}\ln(2\pi) + \frac{1}{2}\ln(det(\Sigma))\right) \int \frac{1}{(2\pi)^{\frac{d}{2}} det(\Sigma)^{\frac{1}{2}}} \exp\left(-\frac{(\mathbf{h}-\mu)^\top \Sigma^{-1} (\mathbf{h}-\mu)}{2}\right) d\mathbf{h} \tag{10.24}$$

$$- \int \frac{(\mathbf{h}-\mu)^\top \Sigma^{-1} (\mathbf{h}-\mu)}{2} \frac{1}{(2\pi)^{\frac{d}{2}} det(\Sigma)^{\frac{1}{2}}} \exp\left(-\frac{(\mathbf{h}-\mu)^\top \Sigma^{-1} (\mathbf{h}-\mu)}{2}\right) d\mathbf{h} \tag{10.25}$$

$$= -\frac{d}{2}\ln(2\pi) - \frac{1}{2}\ln(det(\Sigma)) - \frac{d}{2}. \tag{10.26}$$

Thus, the KL divergence is calculated by subtracting Part II from Part I, which

yields,

$$\int_{\mathbf{h}\sim q} \ln \frac{p(\mathbf{h})}{q(\mathbf{h} \mid \mu, \Sigma)} d\mathbf{h} = \frac{d}{2} + \frac{1}{2} \ln det(\Sigma) - \mu^\top \mu - \|\Sigma\|_F^2. \tag{10.27}$$

We further add the reconstruction loss and the KL divergence together to be the final VAE loss function. Now, we will learn the PyTorch implementation of VAE on the MNIST dataset below. At the end of this chapter, we will learn to apply VAE on the patient chest X-ray dataset.

10.1.4 PyTorch Implementation of VAE

Step 1: Loading the MNIST dataset

We first load the training and test data from online MNIST sources. We could see that there are 60k training data and 10k test data samples, and each is a digit image of 28×28 size. An illustration of the first five training images of MNIST is shown in Figure 10.5.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import cv2
import numpy as np

# we will use the first GPU on the machine for training
# user could also change into "cpu" if no GPU is available
device = torch.device('cuda:0')

# for training set, we load the data and label
trainset = torchvision.datasets.MNIST(\
            root='./data', train = True, download = True)
xtrain = trainset.data.numpy()
ytrain = trainset.targets.numpy()

# for test set, we load the data and label
testset = torchvision.datasets.MNIST(
            root='./data', train = False, download = True)
xtest = testset.data.numpy()
ytest = testset.targets.numpy()

print (xtrain.shape, ytrain.shape)
print (xtest.shape, ytest.shape)
"""
(60000, 28, 28) (60000,)
(10000, 28, 28) (10000,)
"""
```

We then wrap them up as the data loaders. To make life easier, we transform the pixel range from $(0, 255)$ to binary $\{0, 1\}$ by setting a threshold 128.

```
# transform pixel value (0, 255) into {0,1}
xtrain = np.where(xtrain > 128, 1, 0)
xtrain = xtrain.astype(np.float32)
```

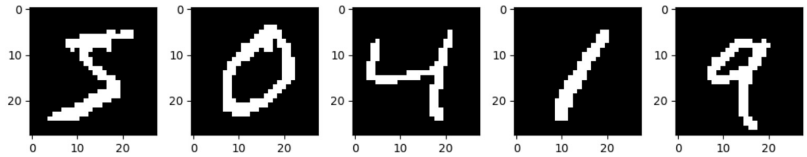

Figure 10.5 Illustration of the first 5 MNIST training samples

```
# transform pixel value (0, 255) into {0,1}
xtest = np.where(xtest > 128, 1, 0)
xtest = xtest.astype(np.float32)

batch_size = 256
trainloader = torch.utils.data.DataLoader(
                        [[xtrain[i], ytrain[i]] for i in range(len(ytrain))], \
                        shuffle=True, batch_size=batch_size)
testloader = torch.utils.data.DataLoader(
                        [[xtest[i], ytest[i]] for i in range(len(ytest))], \
                        shuffle=False, batch_size=batch_size)
```

Step 2: build the VAE model

We then build the VAE model, which has an encoder and an decoder. The encoder is a two-layer neural network with tanh as the activation, and it has two heads, `self.mu` is used for predicting the μ vector, and `self.log_sigma2` is used for predicting the $\ln \Sigma$ vector. The decoder is also a two-layer neural network with tanh as the activation while we finally use Sigmoid to re-scale the output. Since the MNIST dataset has all 0 or 1 as pixel values, we use binary cross entropy loss instead of MSE loss. Here, $d = 8$ is used as the bottleneck dimension.

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

        # DNN as encoder
        self.encoder1 = nn.Linear(28*28, 128)
        self.mu = nn.Linear(128, 8)
        self.log_sigma2 = nn.Linear(128, 8)

        # DNN as decoder
        self.decoder1 = nn.Linear(8, 128)
        self.decoder2 = nn.Linear(128, 28*28)

    def encoder(self, x):
        h = torch.tanh(self.encoder1(x))
        return self.mu(h), torch.sqrt(torch.exp(self.log_sigma2(h)))

    def sampling(self, mu, std): # Reparameterization trick
        eps = torch.randn_like(std)
        return mu + eps * std

```

```
    def decoder(self, z):
        h = torch.tanh(self.decoder1(z))
        return torch.sigmoid(self.decoder2(h))

    def forward(self, x):
        mu, std = self.encoder(x.view(-1, 28*28))
        z = self.sampling(mu, std)
        return self.decoder(z), mu, std

    @staticmethod
    def loss_func(x_hat, x, mu, std):
        # define the reconstruction loss
        ERR = F.binary_cross_entropy(x_hat, x, reduction='sum')
        # define the KL divergence loss
        KLD = -0.5 * torch.sum(1 + torch.log(std**2) - mu**2 - std**2)
        return ERR + KLD, ERR, KLD

model = VAE()
model.to(device) # move model to GPU
```

In the above scripts, the VAE model has an encoder layer `self.encoder1()` with the tanh() activation that maps the input images into hidden embedding **h**. From **h**, the `self.h()` function will generate the mean of Gaussian distribution while the `self.log_sigma2()` will generate $\log \Sigma$ (we will use `exp()` to restore the standard deviation Σ of the Gaussian distribution). Let us look at the `self.forward()` function, after obtaining the mean (μ) and standard deviation (Σ), we use the `self.sampling()` method to obtain a differentiable sample from the Gaussian distribution by $\mu + \epsilon \odot \Sigma^{\frac{1}{2}}$ and decode the it into a reconstructed image.

Step 3: Model Training and Evaluation

We use the standard training pipeline and evaluate the model on test data after each training epoch. We could observe that both the training and test error (i.e., reconstruction error).

Ideally the KL divergence should increase as well but it goes down gradually, which is a common issue in practical implementation of VAE model. This issue could be addressed by carefully tuning the hyperparameters or using learning rate scheduling as discussed in a Microsoft Research (MSR) blog [8]. Later on chest X-ray application, we will show that the latent diffusion model (LDM) could fix the problem and improve the generation quality.

```
optimizer = torch.optim.AdamW(model.parameters(), lr = 0.002)

rec_loss, kl_loss = [], []
for epoch in range(50):

    """ model training """
    model.train()
    cur_rec_loss, cur_kl_loss = [], []
```

[8] https://www.microsoft.com/en-us/research/blog/less-pain-more-gain-a-simple-method-for-vae-training-with-less-of-that-kl-vanishing-agony/

```
    for batch_idx, (data, _) in enumerate(trainloader):
        data = data.to(device)
        # get reconstruction, mean and std vectors
        rec, mu, std = model(data)
        # calculate the loss, reconstruction error, and KL divergence
        loss, err, kl = model.loss_func(rec, data.reshape(-1, 28*28), mu, std)

        # loss backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        cur_rec_loss.append(err.item())
        cur_kl_loss.append(kl.item())

    rec_loss.append(np.mean(cur_rec_loss))
    kl_loss.append(np.mean(cur_kl_loss))

    """ model evaluation """
    with torch.no_grad():
        test_loss = []
        for batch_idx, (data, _) in enumerate(testloader):
            data = data.to(device)
            rec, mu, std = model(data)
            _, loss, _ = model.loss_func(rec, data.reshape(data.shape[0], -1),
                                                    mu, std)
            test_loss.append(loss.item())

    if epoch % 10 == 0:
        print (f"-- epoch {epoch} --, train loss: {np.mean(cur_rec_loss)}, \
            train KL: {np.mean(cur_kl_loss)}, test loss: {np.mean(test_loss)}")

"""
-- epoch 0 --, train loss: 45738.48, train KL: 2987.92, test loss: 33421.22
-- epoch 10 --, train loss: 24595.57, train KL: 4055.006, test loss: 24020.88
-- epoch 20 --, train loss: 23028.66, train KL: 4170.09, test loss: 22486.60
-- epoch 30 --, train loss: 22330.19, train KL: 4242.63, test loss: 21889.96
-- epoch 40 --, train loss: 21925.85, train KL: 4286.16, test loss: 21725.06
"""
```

Step 4: Generating new digit images

After training the model, we can generate synthetic digit images. One could sample 8-dimensional Gaussian noise and inputs them into the VAE decoder for obtaining the synthesized images. Figure 10.6 shows 10 images generated by VAE.

```
plt.figure(figsize=(10, 3))

model.eval()
with torch.no_grad():
    for l in range(10):
        plt.subplot(2, 5, l+1)

        # randomly generate a 8-dim bottleneck vector
        x = np.random.normal(0, 1, 8)
        x = x.astype(np.float32)
```

```
        x = torch.from_numpy(x).to(device).unsqueeze(0)

        # use decoder to generate the digit image
        rec = model.decoder(x).detach().cpu().numpy()
        rec = rec.reshape((1, 28, 28))[0]
        plt.imshow(rec, cmap="gray")

plt.tight_layout()
plt.show()
```

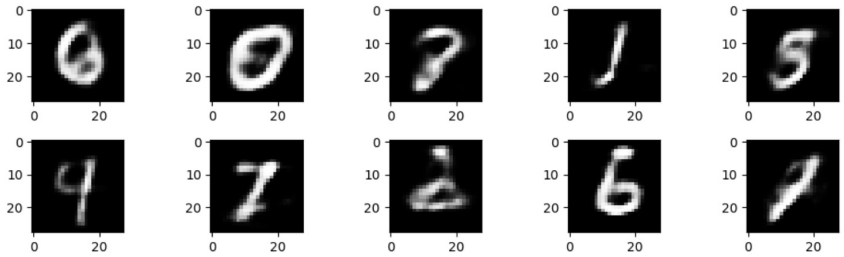

Figure 10.6 Ten digit images generated by VAE

Step 5: Conditional VAE

The VAE model can generate new digit images from random Gaussian vectors, but currently we can't control which digit is generated. To controllably generate class-specific digit images, we could build a ConditionalVAE model by adding the class label information as part of the input. The ConditionalVAE model is similar to the VAE model, and the only difference is in the `self.decoder` part. We additionally learn embedding vectors for each label (0-9), stored in `self.digit`, and this label information is added to the bottleneck layer `z` before feeding it into the decoder model.

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

        # for each digit, we learn an 8-dim vector, initialized randomly
        # they are learned together with the encoder and decoder model
        self.digit = nn.Embedding(10, 8)

        # DNN as encoder
        self.encoder1 = nn.Linear(28*28, 128)
        self.mu = nn.Linear(128, 8)
        self.log_sigma2 = nn.Linear(128, 8)

        # DNN as decoder
        self.decoder1 = nn.Linear(8, 128)
        self.decoder2 = nn.Linear(128, 28*28)

    def encoder(self, x):
        h = torch.tanh(self.encoder1(x))
```

```
        return self.mu(h), torch.sqrt(torch.exp(self.log_sigma2(h)))

    def sampling(self, mu, std): # Reparameterization trick
        eps = torch.randn_like(std)
        return mu + eps * std

    def decoder(self, z, l):
        # add label embedding to random vector
        z = z + self.digit(l)
        h = torch.tanh(self.decoder1(z))
        return torch.sigmoid(self.decoder2(h))

    def forward(self, x, l):
        mu, std = self.encoder(x.view(-1, 28*28))
        z = self.sampling(mu, std)
        return self.decoder(z, l), mu, std

    @staticmethod
    def loss_func(x_hat, x, mu, std):
        # define the reconstruction loss
        ERR = F.binary_cross_entropy(x_hat, x, reduction='sum')
        # define the KL divergence loss
        KLD = -0.5 * torch.sum(1 + torch.log(std**2) - mu**2 - std**2)
        return ERR + KLD, ERR, KLD

model = ConditionalVAE()
model.to(device) # move model to GPU
```

Figure 10.7 shows an illustration of five synthetic "0" and five synthetic "1" generated using this method.

```
plt.figure(figsize=(10, 3))

model.eval()
with torch.no_grad():
    for idx, l in enumerate([0 for i in range(5)] + [1 for i in range(5)]):
        plt.subplot(2, 5, idx+1)

        # randomly generate a 8-dim bottleneck vector
        x = np.random.normal(0, 1, 8)
        x = x.astype(np.float32)
        x = torch.from_numpy(x).to(device).unsqueeze(0)
        l = torch.LongTensor([l]).to(device).unsqueeze(0)

        # use decoder to generate the digit image
        rec = model.decoder(x, l).detach().cpu().numpy()
        rec = rec.reshape((1, 28, 28))[0]
        plt.imshow(rec, cmap="gray")

plt.tight_layout()
plt.show()
```

A concrete notebok including the implementation of VAE and ConditionalVAE could be found in this public repository[9].

[9] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap8-GenAI/notebook

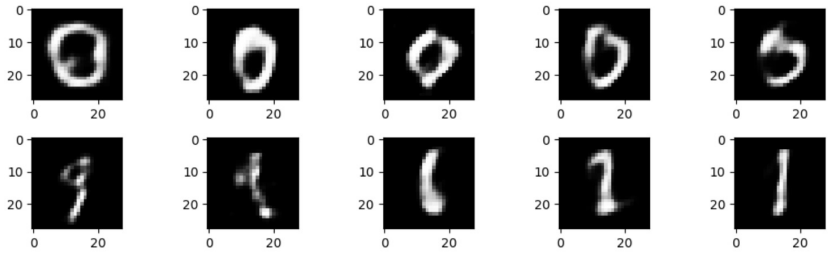

Figure 10.7 Five "0" and Five "1" digit images generated by ConditionalVAE

10.2 Generative Adversarial Network (GAN)

Generative adversarial networks (GAN) were proposed by Goodfellow [126, 127] as a new way of training synthetic data generators. The GAN model is composed of two parts: a generator (G) and a discriminator (D).

The generator (G) starts by taking a random noise from standard distributions (such as multi-variate Gaussian) and converting it into a "fake" data sample. The goal of the GAN model is to ensure that the generated samples look similar to the real samples. To achieve this, the discriminator (D), which is a supervised learning classifier, takes a data sample as input and determines whether it is real or fake. If the discriminator (D) is strong enough and cannot tell the difference between the real and fake samples, then the generator (G) is proven to produce highly realistic images.

Figure 10.8 depicts the GAN training process: the discriminator tries to differentiate between real and fake images while the generator tries to fool the discriminator by creating realistic fake images.

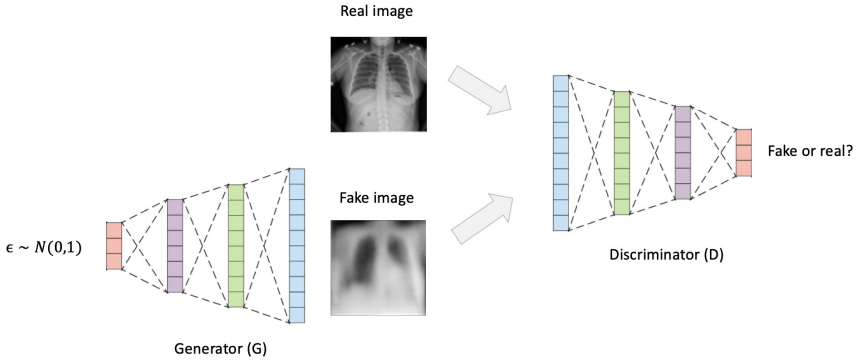

Figure 10.8 Illustration of generative adversarial network (GAN)

10.2.1 GAN training process within generator (G) and discriminator (D)

Formally, we assume the generator is $G_\theta(\cdot)$ with θ as the parameters and the discriminator is $D_\phi(\cdot)$ with ϕ as the parameters. The model works as follows,

- Generate a random sample ϵ from a standard distribution, such as 8-dimensional multi-variate Gaussian distribution with independent dimensions.

$$\epsilon \sim \mathcal{N}(0, \mathbf{I}). \tag{10.28}$$

- The generator (G) will encode the random noise into a fake sample $\mathbf{X}_f$, the same size as the real images.

$$\mathbf{X}_f = G_\theta(\epsilon). \tag{10.29}$$

- Collect a dataset with fake samples and real samples. Fake samples $\mathbf{X}_f$ have label $y = 0$, and real samples $\mathbf{X}_r$ have label $y = 1$. Use the discriminator $D(\cdot)$ to predict whether the data sample is fake or real. The output of discriminator is within $(0, 1)$ range since it passes through the Softmax function.

$$\hat{y} = D_\phi(\mathbf{X}) \in (0, 1). \tag{10.30}$$

- Calculate the binary cross entropy (BCE) loss $\mathcal{L}(y, \hat{y})$. To minimize this BCE loss, essentially, we will maximize $\log \hat{y} = \log D_\phi(\mathbf{X})$ for real samples and maximize $\log(1 - \hat{y}) = \log(1 - D_\phi(\mathbf{X}))$ for fake samples.

$$\mathcal{L}(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}). \tag{10.31}$$

To train a good discriminator D, we maximize the following quantity, including the real image part and the fake image part.

$$\max_\phi \mathbb{E}_{\mathbf{X}_r \sim p_r} \left[\log D_\phi(\mathbf{X}_r)\right] + \mathbb{E}_{\mathbf{X}_f \sim \theta} \left[\log(1 - D_\phi(\mathbf{X}_f))\right], \tag{10.32}$$

where $\mathbf{X}_r \sim p_r$ means the real samples are obtained from true data distribution, and $\mathbf{X}_f \sim \theta$ means that the fake ones are generated from the generator with parameter θ. It can also be written as

$$\max_\phi \mathbb{E}_{\mathbf{X}_r \sim p_r} \left[\log D_\phi(\mathbf{X}_r)\right] + \mathbb{E}_{\epsilon \sim \mathcal{N}(0,1)} \left[\log(1 - D_\phi(G_\theta(\epsilon)))\right]. \tag{10.33}$$

However, **for training a good generator** G, we want the fake image to look as real as possible, which means fixing the discriminator parameters ϕ, we want to minimize the following.

$$\min_\theta \mathbb{E}_{\epsilon \sim \mathcal{N}(0,1)} \left[\log(1 - D_\phi(G_\theta(\epsilon)))\right]. \tag{10.34}$$

Overall, the goal of the GAN training follows a min-max principle. At each iteration, we train the discriminator first (maximizing the following quantity over ϕ by fixing θ) and then train the generator (minimizing the same quantity over θ by fixing ϕ).

$$\min_\theta \max_\phi \mathbb{E}_{\mathbf{X}_r \sim p_r} \left[\log D_\phi(\mathbf{X}_r)\right] + \mathbb{E}_{\epsilon \sim \mathcal{N}(0,1)} \left[\log(1 - D_\phi(G_\theta(\epsilon)))\right]. \tag{10.35}$$

To improve the model stability, researchers usually use another form, which has the same global optimal solutions but enjoys better numerical properties.

$$\min_{\theta}\max_{\phi}\mathbb{E}_{\mathbf{X}_r\sim p_r}\left[\log D_\phi(\mathbf{X}_r)\right]-\mathbb{E}_{\epsilon\sim\mathcal{N}(0,1)}\left[\log(D_\phi(G_\theta(\epsilon)))\right]. \tag{10.36}$$

10.2.2 PyTorch Implementation of GAN

We will learn how to implement a GAN model from scratch with PyTorch. In the code, we use a three-layer neural network for the discriminator and another three-layer neural network for the generator in GAN. The dataset will still be MNIST. We show the application on chest X-ray images at the end of the chapter.

Define the GAN model

We define the GAN model class below. In the model, the discriminator uses a ReLU as the activation function and the generator also uses ReLU as the activation function. The model has three functions: `self.discriminate()` function takes a 28×28 image as input and generates a probability between $(0, 1)$ while 0 indicating "fake" and 1 indicating "real". The `self.generate_fake()` function will sample `n_samples` random noise and use the generator to generate `n_samples` fake images.

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

        # three layer neural network as discriminator
        self.discriminator = nn.Sequential(
            nn.Linear(28*28, 256),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Linear(256, 128),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Linear(128, 1),
            nn.Sigmoid()
        )

        self.generator = nn.Sequential(
            nn.Linear(8, 128),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Linear(128, 256),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Linear(256, 28*28),
        )

    def discriminate(self, x):
        y = self.discriminator(x)
        return y

    def generate_fake(self, n_samples, device):
        # sample 8-dimensional vectors as random noise
        eps = torch.randn(n_samples, 8).to(device)
        # generate images based on the random vectors
        fake_images = torch.sigmoid(self.generator(eps))
        return fake_images
```

```
model = GAN()
model.to(device) # move model to GPU
```

GAN Model Training and Evaluation

Updating the GAN model is a two-step process with binary cross-entropy (BCE) loss: we first optimize the discriminator and then optimize the generator.

To begin, we get a real batch of MNIST samples from the training loader, called `real_imgs`. Then, we ask the generator to create fake images of the same size, called `fake_imgs`. We label the real images as "1" and the fake images as "0". Next, we ask the discriminator to calculate the classification loss, and minimizing this loss would improve the discriminator's performance. We use the optimizer `opt_D` to update the discriminator.

In the second step, we generate another random batch of fake images, named `fake_imgs` and label them as "1" since at this step, our goal is to let the discriminator believes the generated images are real. We ask the discriminator to calculate the BCE loss and then minimize it. This step helps us improve the generator based on the current discriminator. After loss back-propagation, the optimizer `opt_G` will then update the generator.

Our ultimate goal is to build a better discriminator that can distinguish between real and fake images and an even better generator that can fool this discriminator. The discriminator and generator are optimized alternately to achieve this goal in an adversarial way.

```
# the loss function
loss = torch.nn.BCELoss()

# optimizers
opt_G = torch.optim.AdamW(model.generator.parameters(), lr=2e-4)
opt_D = torch.optim.AdamW(model.discriminator.parameters(), lr=2e-4)

# store the loss curve of D and G
curve_D, curve_G = [], []
for epoch in range(50):

    # store the loss over one epoch
    curve_G.append(0)
    curve_D.append(0)

    for data, label in trainloader:
        """ train discriminator """
        opt_D.zero_grad()

        batch_size = data.shape[0]
        # get real and fake images
        real_imgs = data.view(batch_size, -1).to(device)
        fake_imgs = model.generate_fake(batch_size, device)
        # set their labels (real gets ones, and fake gets zeros)
        ones = torch.ones(batch_size, 1).to(device)
        zeros = torch.zeros(batch_size, 1).to(device)
```

```
        real_loss = loss(model.discriminator(real_imgs), ones)
        fake_loss = loss(model.discriminator(fake_imgs), zeros)
        loss_D = (real_loss + fake_loss) / 2

        loss_D.backward()
        opt_D.step()

        """ train generator """
        opt_G.zero_grad()
        # get fake images and label it as ones
        fake_imgs = model.generate_fake(batch_size, device)
        loss_G = loss(model.discriminator(fake_imgs), ones)

        loss_G.backward()
        opt_G.step()

        curve_G[-1] += loss_G.item()
        curve_D[-1] += loss_D.item()

    print (f"epoch: {epoch} --- \
                loss of G: {curve_G[-1]}, loss of D: {curve_D[-1]}")

"""
epoch: 0 --- loss of G: 746.3518468141556, loss of D: 43.85571137443185
epoch: 1 --- loss of G: 1275.6765661239624, loss of D: 5.175452238880098
epoch: 2 --- loss of G: 1571.3696827888489, loss of D: 4.034340920858085
epoch: 3 --- loss of G: 1540.1672163009644, loss of D: 1.1458109847735614
...
epoch: 44 --- loss of G: 2625.360191345215, loss of D: 1.2749904468219029
epoch: 45 --- loss of G: 2535.6209321022034, loss of D: 1.6519437322713202
epoch: 46 --- loss of G: 2483.5613226890564, loss of D: 2.927129815390799
epoch: 47 --- loss of G: 2380.45946598053, loss of D: 2.1951384948333725
epoch: 48 --- loss of G: 2434.2216053009033, loss of D: 2.180281783337705
epoch: 49 --- loss of G: 2226.480115890503, loss of D: 2.546948796516517
"""
```

Generating synthetic digit images with GAN

As demonstrated in Figure 10.9, similar to a VAE model, a trained GAN model can be utilized to generate synthetic digit images. However, in this particular instance, the quality of images produced by the GAN model is not as good as those generated by the VAE model. It's important to note that this is not a general statement about GAN, as there are numerous examples of successful GAN models that produce realistic data samples.

```
plt.figure(figsize=(10, 3))

model.eval()
with torch.no_grad():
    for l in range(10):
        plt.subplot(2, 5, l+1)

        # use decoder to generate the digit image
        rec = model.generate_fake(1, device).detach().cpu().numpy()
```

```
        rec = rec.reshape((1, 28, 28))[0]
        plt.imshow(rec, cmap="gray")

plt.tight_layout()
plt.show()
```

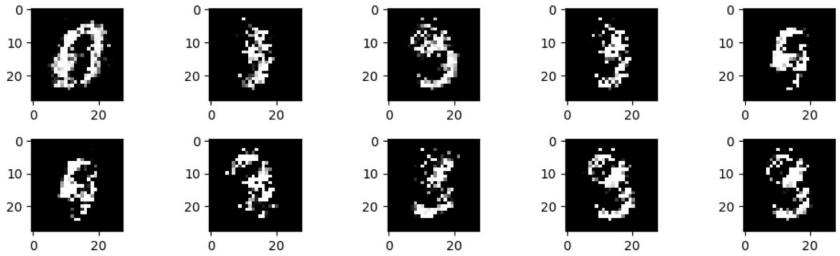

Figure 10.9 Ten "fake" digit images generated by GAN

Conditional GAN

We can also create a model called ConditionalGAN that generates images based on specific labels instead of random digits. The ConditionalGAN model differs from the simple GAN model in two ways: (i) the label embedding is added to the random noise ϵ before being fed into the generator, and (ii) instead of using binary cross-entropy loss, ConditionalGAN uses cross-entropy loss over various classes, including class 0 to class 9, and an additional class for fake images.

To train the discriminator, we assign real images to class labels and use the additional class for fake images. When training the generator, we set the desired class labels for the fake images and minimize the cross-entropy loss. You can find the implementation of the ConditionalGAN model below.

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

        # embedding for each class, initialized randomly
        # it is learned together with the generator and discriminator
        self.digit = nn.Embedding(10, 8)

        # three layer neural network as discriminator
        # output a 11-dim vector represents the probability of each class
        # including the additional fake class
        self.discriminator = nn.Sequential(
            nn.Linear(28*28, 256),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Linear(256, 128),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Linear(128, 11),
        )

        # three layer neural network as generator
```

```
        self.generator = nn.Sequential(
            nn.Linear(8, 128),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Linear(128, 256),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Linear(256, 28*28),
        )

    def discriminate(self, x):
        y = self.discriminator(x)
        return y

    def generate_fake(self, labels, device):
        # generate random vectors
        eps = torch.randn(len(labels), 8).to(device)
        # get the class label embeddings
        digit_emb = self.digit(labels.to(device))
        # add these two and pass to generator
        fake_images = torch.sigmoid(self.generator(eps + digit_emb))
        return fake_images

model = ConditionalGAN()
model.to(device) # move model to GPU
```

We skip the training process of ConditionalGAN here as it is similar to training of GAN. Figure 10.10 shows the digit images from 0 to 9 generated by ConditionalGAN model, the quality of which seems better than GAN model.

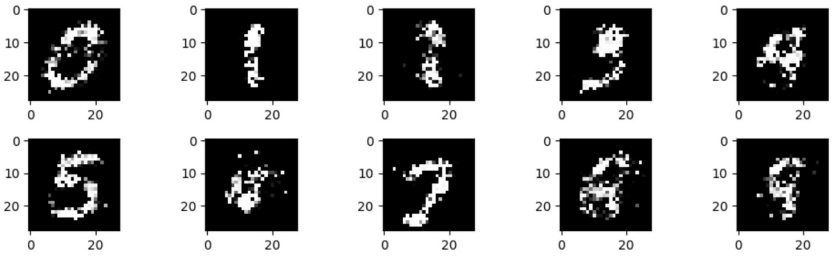

Figure 10.10 Ten digit images generated by ConditionalGAN

A concrete notebook including implementations of GAN and ConditionalGAN could be found in this public repository[10].

Practical improvements of GAN models

The original training procedure of Generative Adversarial Networks (GANs) often runs into issues such as instability and mode collapse, where the generator produces a limited variety of outputs. To address these challenges, Arjovsky and his colleagues introduced a new type of Generative Adversarial Networks (GANs), called Wasserstein GAN (WGAN) [136]. The WGAN model replaces the discriminator with a critic that

[10] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap8-GenAI/notebook

evaluates the generated samples based on the Earth Mover's Distance instead of the KL divergence used in the traditional GANs. The WGAN model is known for its stable and diverse training results.

The Earth Mover's distance (EMD), also known as the Wasserstein distance, in the context of WGAN, is mathematically formulated as the minimum cost of transporting mass to transform one distribution into the other. For two probability distributions, $\mathbb{P}_r$ (real data distribution) and $\mathbb{P}_g$ (generated data distribution), the Wasserstein distance is defined as:

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|].$$

Here, inf denotes the infimum over all joint distributions γ whose marginals are $\mathbb{P}_r$ and $\mathbb{P}_g$, and $\Pi(\mathbb{P}_r, \mathbb{P}_g)$ represents the set of all possible joint distributions (couplings) of $\mathbb{P}_r$ and $\mathbb{P}_g$. The expectation $\mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|]$ calculates the average cost of moving the mass from x to y, representing the amount of work required to transform one distribution into the other.

As for the implementation, WGAN and GAN differ in that:

- WGAN does not use Sigmoid function in the discriminator.
- WGAN uses Wasserstein loss [11] to train the discriminator and the generator.
- WGAN clips the discriminator weights to a limited range after each mini-batch update (such as in $[-0.01, 0.01]$).
- In WGAN, the discriminator is trained more times than the generator (e.g., 5 times).
- WGAN uses RMSProp [12] as the optimizer with a small learning rate and no momentum.

10.3 Diffusion Models

A diffusion model is a generative deep learning technique used for creating new data, such as images, by learning to reverse a gradual noising process. The model is trained to denoise data by learning the reverse process of gradually adding noise to the data. The key steps are:

1. **Forward process.** Take a training dataset and gradually add noise to the data in a series of steps, resulting in a sequence of increasingly noisy versions of the data. This step does not need parameters.
2. **Reverse process.** Train a neural network to reverse this noising process by learning to predict less noisy versions of the data given the noisy input. This steps require a parameterized step-wise denoising model.
3. **Generation.** To generate new data, start with random noise and iteratively apply the trained reverse process model to remove the noise, eventually resulting in a new sample similar to the training data.

[11] https://en.wikipedia.org/wiki/Wasserstein_GAN
[12] https://pytorch.org/docs/stable/generated/torch.optim.RMSprop.html

Diffusion models have shown impressive results in generating high-quality images, audio, and other types of data. They are known for their stability and the ability to produce diverse and realistic outputs [137, 138, 139]. These are a new type of generative model inspired by a physical process. Notable works of diffusion models include diffusion probabilistic models [140], noise-conditioned score network [141], and denoising diffusion probabilistic models [129]. Next, we will introduce diffusion models in detail.

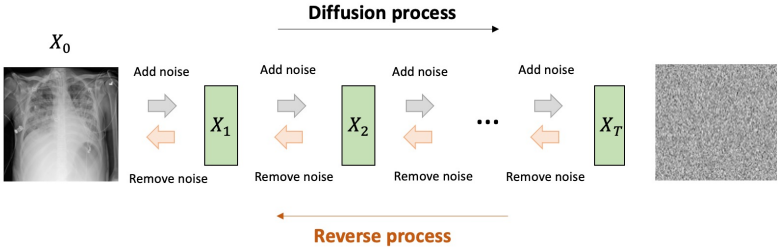

Figure 10.11 Illustration of the diffusion and reverse process in the diffusion model

10.3.1 Forward process of diffusion models

Let's use the example of generating a chest X-ray image to understand the diffusion process. In Figure 10.11, we start with a real chest X-ray image, denoted as $\mathbf{X}_0$, and add a small amount of noise step by step, generating $\mathbf{X}_1, \cdots, \mathbf{X}_T$. Each $\mathbf{X}_t$ contains more Gaussian noise than its predecessor. Finally, we arrive at $\mathbf{X}_T$, which contains 100% noise, following a standard Gaussian distribution.

The degree of noise in each step is controlled by factors $\{\beta_t \in (0,1)\}_{t=1}^{T}$. We model each step in the diffusion process by Gaussian distribution sampling, starting from $\mathbf{X}_0$. Specifically, we have:

$$\mathbf{X}_t \sim q(\mathbf{X}_t|\mathbf{X}_{t-1}) = \mathcal{N}(\sqrt{1-\beta_t}\mathbf{X}_{t-1}, \beta_t\mathbf{I}), \tag{10.37}$$

Mean and Variance of $\mathbf{X}_t$

Assume the initial sample is normalized, meaning that $\mathbb{E}[\mathbf{X}_0] = 0$ and $\mathbb{E}[(\mathbf{X}_0)^2] = \mathbf{I}$. Then, sampled $\mathbf{X}_t$ has the following statistics,

$$mean = \mathbb{E}[\mathbf{X}_t] = \mathbb{E}[\sqrt{1-\beta_t}\mathbf{X}_{t-1}] = \sqrt{1-\beta_t}\mathbb{E}[\mathbf{X}_{t-1}] = 0, \tag{10.38}$$

$$variance = \mathbb{E}[(\mathbf{X}_t)^2] = (\sqrt{1-\beta_t})^2\mathbb{E}[(\mathbf{X}_{t-1})^2] + \beta_t\mathbf{I} = \mathbf{I}. \tag{10.39}$$

Following Markov property, the whole diffusion process can be modeled as a product of the Gaussian distribution,

$$\mathbf{X}_1, \mathbf{X}_2, \cdots, \mathbf{X}_T \sim p(\mathbf{X}_1, \cdots, \mathbf{X}_T \mid \mathbf{X}_0) = \prod_{t=1}^{T} q(\mathbf{X}_t \mid \mathbf{X}_{t-1}). \tag{10.40}$$

As T becomes bigger (if $T \to \infty$), the features/patterns from the original data $\mathbf{X}_0$ gradually becomes indistinguishable and finally fully noisy, which is $\mathbf{X}_T$.

$\mathbf{X}_t$ is Gaussian Distributed

Using the above diffusion process, the intermediate $\mathbf{X}_t$ are all Gaussian distributed. Let us look at this. Given the initial $\mathbf{X}_0$, the first noisy sample is generated by (following Equation 10.37),

$$\mathbf{X}_1 = \sqrt{1-\beta_1}\mathbf{X}_0 + \sqrt{\beta_1}\epsilon_0 \sim \mathcal{N}(\sqrt{1-\beta_1}\mathbf{X}_0, \beta_1\mathbf{I}). \tag{10.41}$$

The second noisy sample is generated by (following Equation 10.37),

$$\begin{aligned}
\mathbf{X}_2 &= \sqrt{1-\beta_2}\mathbf{X}_1 + \sqrt{\beta_2}\epsilon_1 && (10.42)\\
&= \sqrt{1-\beta_2}(\sqrt{1-\beta_1}\mathbf{X}_0 + \sqrt{\beta_1}\epsilon_0) + \sqrt{\beta_2}\cdot\epsilon_1 && (10.43)\\
&= \sqrt{1-\beta_1}\sqrt{1-\beta_2}\mathbf{X}_0 + \sqrt{1-\beta_2}\sqrt{\beta_1}\epsilon_0 + \sqrt{\beta_2}\epsilon_1 && (10.44)\\
&= \sqrt{(1-\beta_1)(1-\beta_2)}\mathbf{X}_0 + \sqrt{1-(1-\beta_1)(1-\beta_2)}\tilde{\epsilon}_1 && (10.45)\\
&\sim \mathcal{N}(\sqrt{(1-\beta_1)(1-\beta_2)}\mathbf{X}_0, (1-(1-\beta_1)(1-\beta_2))\mathbf{I}). && (10.46)
\end{aligned}$$

The last step is given by merging two identical independently distributed Gaussian noise. Since the merge of two Gaussian distribution $\mathcal{N}(\mu_1, \sigma_1^2)$ and $\mathcal{N}(\mu_2, \sigma_2^2)$ is $\mathcal{N}(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2)$. Equivalently, we can get the third noisy sample as

$$\begin{aligned}
\mathbf{X}_3 &= \sqrt{1-\beta_3}\mathbf{X}_3 + \sqrt{\beta_3}\epsilon_2 && (10.47)\\
&= \sqrt{(1-\beta_1)(1-\beta_2)(1-\beta_3)}\mathbf{X}_0 + \sqrt{1-(1-\beta_1)(1-\beta_2)(1-\beta_3)}\tilde{\epsilon}_2 && (10.48)\\
&\sim \mathcal{N}(\sqrt{(1-\beta_1)(1-\beta_2)(1-\beta_3)}\mathbf{X}_0, (1-(1-\beta_1)(1-\beta_2)(1-\beta_3))\mathbf{I}). && (10.49)
\end{aligned}$$

and thus, the last noisy sample is

$$\mathbf{X}_T \sim \mathcal{N}\left(\sqrt{\prod_{t=1}^{T}(1-\beta_t)}\mathbf{X}_0, \left(1-\prod_{t=1}^{T}(1-\beta_t)\right)\mathbf{I}\right). \tag{10.50}$$

In practice, the values of $\prod_{t=1}^{T}(1-\beta_t)$ will gradually vanish over each timestep, reaching 0 at the final timestep. This allows for a smooth transition from the original data distribution to the noisy distribution, ensuring that $\mathbf{X}_T$ is a fully random Gaussian distribution.

In other words, *the diffusion process does not need any learnable parameters*, and it provides a series of noisy samples $\{\mathbf{X}_t,\ t = 1, \ldots, T\}$ where $\mathbf{X}_0$ is the original data, $\mathbf{X}_t$ is more noisy than $\mathbf{X}_{t-1}$, and $\mathbf{X}_T$ is asymptotically randomly gaussian distributed.

10.3.2 Reverse process of diffusion model

After understanding the forward process of the diffusion model gradually adding noise to the original input data, we will next describe the reverse process that gradually denoises the fully noisy image until we recover the initial data in a stochastic way. This

reverse process is motivated by gradient Langevin dynamics [142] where the concept is developed for modeling the molecular systems in physics.

Given that the noise applied at each step is Gaussian distributed, and the sampled intermediate data $\{\mathbf{X}_t\}$ is Gaussian distributed, we thus know the reconstructed intermediate data is also Gaussian distributed. Let us denote the recovering function during the process is $f_\theta(\mathbf{X}_{t-1} \mid \mathbf{X}_t)$ with parameters θ. Thus, the first step is to obtain a $\mathbf{X}_{T-1}$ from $\mathbf{X}_T$.

$$\mathbf{X}_{T-1} \sim f_\theta(\mathbf{X}_{T-1} \mid \mathbf{X}_T) = \mathcal{N}(\mu_\theta(\mathbf{X}_T, T), \Sigma_\theta(\mathbf{X}_T, T)). \tag{10.51}$$

and step two is

$$\mathbf{X}_{T-2} \sim f_\theta(\mathbf{X}_{T-2} \mid \mathbf{X}_{T-1}) = \mathcal{N}(\mu_\theta(\mathbf{X}_{T-1}, T-1), \Sigma_\theta(\mathbf{X}_{T-1}, T-1)). \tag{10.52}$$

until

$$\mathbf{X}_0 \sim f_\theta(\mathbf{X}_0 \mid \mathbf{X}_1) = \mathcal{N}(\mu_\theta(\mathbf{X}_1, 1), \Sigma_\theta(\mathbf{X}_1, 1)). \tag{10.53}$$

Here, we split the parameterized functions f_θ into a function for predicting the mean value $\mu_\theta(\mathbf{X}_t, t)$ and another one for predicting the covariance matrix $\Sigma_\theta(\mathbf{X}_t, t)$. They both take the sample $\mathbf{X}_t$ and the reverse step t as inputs. To train these two functions, we need to know the real distribution of $\mathbf{X}_{t-1}$ from $\mathbf{X}_t$, which can be obtained from the posterior at the forward process.

$$q(\mathbf{X}_{t-1} \mid \mathbf{X}_t, \mathbf{X}_0) = \frac{q(\mathbf{X}_t, \mathbf{X}_{t-1} \mid \mathbf{X}_0)}{q(\mathbf{X}_t \mid \mathbf{X}_0)} = q(\mathbf{X}_t \mid \mathbf{X}_{t-1}, \mathbf{X}_0)\frac{q(\mathbf{X}_{t-1} \mid \mathbf{X}_0)}{q(\mathbf{X}_t \mid \mathbf{X}_0)}. \tag{10.54}$$

Now, we could use the posterior distribution $q(\mathbf{X}_{t-1} \mid \mathbf{X}_t, \mathbf{X}_0)$ as the ground truth to regulate the parameterized posterior $f_\theta(\mathbf{X}_{t-1} \mid \mathbf{X}_t)$ by KL divergence. We do not elaborate on the detailed math derivation of the objective, which could be found in Weng et al. [143].

Objective functions

Thus, the final objective function of the diffusion model contains two parts: one part is still the reconstruction loss, and another part is a series of KL divergence on $q(\mathbf{X}_{t-1} \mid \mathbf{X}_t, \mathbf{X}_0)$ and $f_\theta(\mathbf{X}_{t-1} \mid \mathbf{X}_t)$ to regulate the intermediate variables. Compared to VAE model, we find that the high-level difference here is that VAE uses one hidden variable $\mathbf{h}$ (the bottleneck layer) while diffusion model uses a series of hidden variable $\mathbf{X}^t$ for generation.

10.3.3 PyTorch Implementation of Diffusion Model

In this section, we will use the MNIST data set to show how to implement a simple diffusion model in PyTorch. The model explained in this section is a simpler version, where we do not normalize $\mathbf{X}_t$ to follow $\mathcal{N}(0, \mathbf{I})$ and we use a deterministic form in the reverse process.

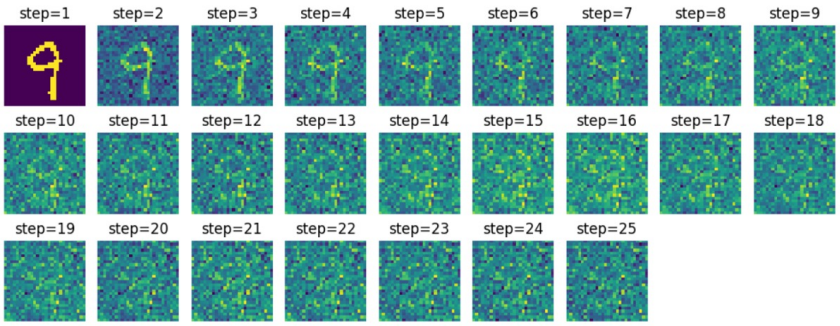

Figure 10.12 Illustration of Diffusion Process (25 steps) on MNIST Digit Image

Simpler diffusion process: adding noise

We first define a `unit_normal` sampler that has the following function `unit_normal.sample([shape])` to sample a Gaussian random variable with any shape. We get the first image from the first batch and its label, named `image` and `label`, and then gradually add noise to it. Given the original image $\mathbf{X}_0$, the diffusion process is shown in Figure 10.12.

```
sigma = 0.3

# Create a normal distribution sampler
unit_normal = torch.distributions.Normal(0, sigma)
unit_normal.loc = unit_normal.loc.to(device)
unit_normal.scale = unit_normal.scale.to(device)

# Demo: Diffusion process
images, labels = next(iter(trainloader))
image = images[0]
label = labels[0]

# Plot the images at different steps of diffusion process
plt.figure(figsize=(10, 4))
for i in range(diffusion_steps):
    plt.subplot(3, 9, i + 1)
    plt.imshow(image)
    plt.axis('off')
    plt.title(f"Step={i+1}")

    # Sample noise from a normal distribution
    noise = unit_normal.sample(image.shape).cpu()
    # Add the noise to the image
    image += noise

plt.tight_layout()
```

The simple diffusion process is explained below:

- **Step 0:** Initial image is $\mathbf{X}_0$, and we do not normalize it.

- **Step 1:** $\mathbf{X}_1 = \mathbf{X}_0 + \epsilon_1$, where $\epsilon_1 \sim \mathcal{N}(0, \Sigma)$ is a random noise. Σ is a diagonal matrix with each diagonal value equals to 0.3, i.e., $\Sigma = 0.3\mathbf{I}$.
- **Step 2:** $\mathbf{X}_2 = \mathbf{X}_1 + \epsilon_2 = \mathbf{X}_0 + \epsilon_1 + \epsilon_2$, where $\epsilon_2 \sim \mathcal{N}(0, \Sigma)$ is a another random noise. In fact, since ϵ_1, ϵ_2 are independent, $\mathbf{X}_2$ is equivalent to $\mathbf{X}_2 = \mathbf{X}_0 + \epsilon_1^*$, where $\epsilon_1^* \sim \mathcal{N}(0, 2\Sigma)$.
- **Step t:** Similarly, we have that $\mathbf{X}_t = \mathbf{X}_{t-1} + \epsilon_t = \mathbf{X}_0 + \epsilon_t^*$, where $\epsilon_t^* \sim \mathcal{N}(0, t\Sigma)$.

In sum, each intermediate image $\mathbf{X}_t$ has two ways to construct: (i) adding a noise sample from $\mathcal{N}(0, \Sigma)$ to $\mathbf{X}_{t-1}$; (ii) adding a noise sample from $\mathcal{N}(0, t\Sigma)$ to the original image $\mathbf{X}_0$.

Simpler Reverse Process and Training: De-noising

In the reverse process, we start with a Gaussian noise sample $\tilde{\mathbf{X}}_{25} \sim \mathcal{N}(0, 25\Sigma)$, assuming we have 25 diffusion steps. Although the final sample at diffusion step 25 should be the original image plus the accumulated noise, we can treat $\tilde{\mathbf{X}}_{25}$ as the final diffusion result since the noise component significantly outweighs the original image.

The reverse process relies on a function f_θ that takes the t-th intermediate image $\mathbf{X}_t$, the diffusion step t, and the data label as inputs. The function aims to infer the additional noise $\tilde{\epsilon}_t$ at diffusion step t:

$$\tilde{\epsilon}_t = f_\theta(\mathbf{X}_t, t, \text{label}) \sim \mathcal{N}(0, \Sigma). \tag{10.55}$$

The output of f_θ is post-processed to follow the same distribution as the real noise ϵ_t, which is $\mathcal{N}(0, \Sigma)$.

Starting from the pure Gaussian noise $\tilde{\mathbf{X}}_{25}$, we generate the estimated noise $\tilde{\epsilon}_{25}$ using f_θ. We then approximate the previous image $\tilde{\mathbf{X}}_{24}$ by subtracting the estimated noise from $\tilde{\mathbf{X}}_{25}$:

$$\tilde{\mathbf{X}}_{24} = \tilde{\mathbf{X}}_{25} - \tilde{\epsilon}_{25}. \tag{10.56}$$

Similarly, we obtain the next noise estimate $\tilde{\epsilon}_{24}$ by feeding $(\tilde{\mathbf{X}}_{24}, 24, \text{label})$ to f_θ. We then perform the de-noising step to approximate $\tilde{\mathbf{X}}_{23}$:

$$\tilde{\mathbf{X}}_{23} = \tilde{\mathbf{X}}_{24} - \tilde{\epsilon}_{24}. \tag{10.57}$$

This process is repeated step by step until we finally obtain the denoised image $\tilde{\mathbf{X}}_0$.

Training Sample Construction

The training objective is to minimize the difference between the estimated noise $\tilde{\epsilon}_t$ and the real noise ϵ_t used in the forward diffusion process. By learning to predict and remove the noise at each step accurately, the model learns to generate realistic images from random Gaussian noise.

To optimize the function f_θ, we can use the following method to construct training samples.

- Get an image and the label from the training set, $\mathbf{X}_0$ and $label$.

- Randomly select a step $t \in \{1, 2, ..., 25\}$ and generate a noise $\epsilon_t^* \sim \mathcal{N}(0, t\Sigma)$.
- Then, randomly generate a noise $\epsilon \sim \mathcal{N}(0, \Sigma)$.
- We let $\mathbf{X}_t = \mathbf{X}_0 + \epsilon_t^*$, and let $\mathbf{X}_{t+1} = \mathbf{X}_t + \epsilon$.
- The ground truth is $\mathbf{X}_t$, and we use $\mathbf{X}_{t+1}$ to predict the noise.

$$\tilde{\epsilon} = f_\theta(\mathbf{X}_{t+1}, t, label). \tag{10.58}$$

- Then, the approximated de-noised version at step t is

$$\tilde{\mathbf{X}}_t = \mathbf{X}_{t+1} - \tilde{\epsilon}. \tag{10.59}$$

- We use mean square error (MSE) loss as the objective between the real noise and the predicted noise,

$$\mathcal{L} = \|\epsilon - \tilde{\epsilon}\|_F^2. \tag{10.60}$$

Next we illustrate the PyTorch implementation.

Load MNIST data

Similar to previous GAN and VAE examples, we load the MNIST training data as the training set.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision

import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt

device = torch.device('cuda:0')

# for the training set, we load the data and label
trainset = torchvision.datasets.MNIST(\
            root='./data', train = True, download = True)
xtrain = trainset.data.numpy()
ytrain = trainset.targets.numpy()

print (xtrain.shape, ytrain.shape)
"""
(60000, 28, 28) (60000,)
"""
```

```
# transform pixel range (0, 255) into {0,1} by splitting at pixel 128
xtrain = np.where(xtrain > 128, 1, 0) #pixel > 128 assigning to 1, otherwise 0.
xtrain = xtrain.astype(np.float32)

batch_size = 512
trainloader = torch.utils.data.DataLoader([[xtrain[i], ytrain[i]] \
            for i in range(len(ytrain))], shuffle=True, batch_size=batch_size)

# sigma is the noise std
sigma = 0.3
```

Define the Diffusion Model

We define the diffusion model f_θ as a two-layer neural network. In the `self.forward()` function, the model takes a batch of samples `x`, the step of the diffusion process `step`, and the label of the images as input `label`. Then, it concatenates the image 28×28 pixels, one hot *step* encoding, and one hot *label* encoding as the features into the two-layer neural network. The output is further normalized into $\mathcal{N}(0, \Sigma)$ distribution.

```
class Diffuser(nn.Module):
    def __init__(self, diff_steps, n_labels):
        super(Diffuser, self).__init__()

        # the diffusion model is a two-layer neural network
        self.diffusion = nn.Sequential(
            nn.Linear(28*28 + diff_steps + n_labels, 28*28),
            nn.ReLU(),
            nn.Linear(28*28, 28*28),
        )

        self.diff_steps = diff_steps
        self.n_labels = n_labels

    def forward(self, x, step, label):

        # one hot for diffusion step
        step_encoding = F.one_hot(
            torch.tensor([step]*x.shape[0]), num_classes = self.diff_steps
        ).to(device)

        # one hot for label
        labels_encoding = F.one_hot(label, \
            num_classes = self.n_labels).to(device)

        # concat features and run diffusion process
        emb = torch.cat([x, step_encoding, labels_encoding], 1)
        out = self.diffusion(emb)

        # normalize the output
        MEAN = out.mean(dim=1, keepdim=True)
        STD = out.std(dim=1, keepdim=True) + 1e-8
        out = (out - MEAN) / STD * sigma
        return out

diffusion_steps = 25
model = Diffuser(25, 10)
model.to(device) # move model to GPU
```

Model Training and Evaluation

As part of the model training process, we require an ϵ noise sampler, which we define directly on the device for faster execution. This sampler generates Gaussian distributed data for any given shape using the syntax `unit_normal.sample([shape])`.

In the following Python code snippet, we create a normal distribution sampler and transfer it to the CUDA device:

```
# normal distribution sampler (we could throw to cuda for once)
unit_normal = torch.distributions.Normal(0, sigma)
unit_normal.loc = unit_normal.loc.to(device)
unit_normal.scale = unit_normal.scale.to(device)
```

The evaluation function begins with a 100% noisy sample starting from random noise sampled from $\mathcal{N}(0, 25\Sigma)$ at step 25. The model then progressively predicts the per-step noise and removes it from the current intermediate images. We repeat this process ten times, once for each label, and then print out the resulting images for digits 0 to 9.

```
def evaluate(model, diffusion_steps):

    # 0, 1, ..., 9
    labels = torch.arange(10)

    # random images (assume at step 25)
    noisy_images = unit_normal.sample((len(labels),28,28)) \
                    * np.sqrt(diffusion_steps)

    plt.figure(figsize=(10, 5))

    # we run diffusion process for multiple steps
    # to denoise the noisy images into a digit image
    for diffusion_step in range(diffusion_steps):
        step = diffusion_steps - diffusion_step - 1
        with torch.no_grad():
            noise = model(noisy_images.view(10, 28*28), step, labels)
        noisy_images -= noise.view(len(labels), 28, 28)

    for i in range(10):
        plt.subplot(1, 10, i+1)
        # print the digit one by one
        plt.imshow(noisy_images[i].cpu())
        plt.axis('off')
    plt.show()

evaluate(model, diffusion_steps)
```

We show the generated synthetic digit images before training in Figure 10.13, which are basically random noises.

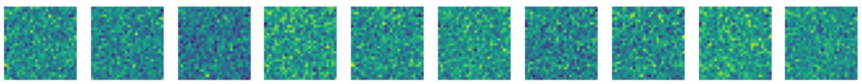

Figure 10.13 Illustration of Generated Digit Images from Diffusion Model (before training)

During the training process, we use Adam as the optimizer and MSE loss. We use 50 epochs to train the diffusion model. For each batch from the training data, we randomly shuffle the diffusion steps and use Equation (10.58) and Equation (10.60) to optimize the parameters. The model will be evaluated every 5 epochs.

```
optimizer = torch.optim.Adam(model.parameters(), 0.1)
criterion = nn.MSELoss()

for epoch in range(50):
    epoch_loss = 0
    for images, labels in tqdm(trainloader):
        images = images.to(device).view(-1, 28*28)

        for step in torch.randperm(diffusion_steps):
            # get a sample at step: original image + sqrt(step) * normal sample
            less_noisy = images + \
                    unit_normal.sample(images.shape) * np.sqrt(step)
            # get a normal sample
            one_step_noise = unit_normal.sample(images.shape)
            # get the sample at (step + 1)
            more_noisy = less_noisy + one_step_noise

            output = model(more_noisy, step, labels)
            loss = criterion(output, one_step_noise)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()
    print(f'Epoch: {epoch} \t Loss: {epoch_loss}')

    if epoch % 5 == 0:
        evaluate(model, diffusion_steps)
```

The final generated digit images drawn from `evaluate()` function are shown in Figure 10.14, which is of high qualilty.

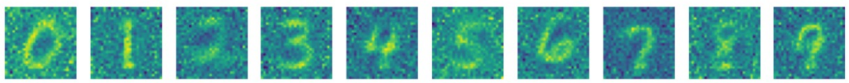

Figure 10.14 Illustration of Generated Digit Images from Diffusion Model

A concrete notebook including PyTorch implementations of diffusion model could be found in this public repository[13].

10.4 Chest X-ray Image Generation with VAE, GAN, and Diffusion Model

Chest X-ray images are useful in diagnosing numerous conditions, including heart diseases and respiratory conditions. Generating synthetic chest X-ray images can be beneficial in improving the accuracy of machine learning models. These artificial images can augment small real datasets or allow other researchers to replicate research

[13] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap8-GenAI/notebook

models without disclosing real patient information. Note that the example requires using pyhealth package in python.

In this section, we will demonstrate how to use the VAE, GAN, and diffusion models in generating chest X-ray images based on the PyHealth package. We will be using the dataset from the COVID-19 Radiography Database[14].

10.4.1 Step 1: data loading

For chest X-ray dataset loading, we use the *COVID*19*CXRDataset* API which will process the COVID19 chest X-ray image dataset. This API only requires the root path to the dataset as the argument, and everything else is handled by the API already.

```
from pyhealth.datasets import COVID19CXRDataset

root = "/srv/local/data/COVID-19_Radiography_Dataset"
base_dataset = COVID19CXRDataset(root)
base_dataset.stat()

"""
Statistics of COVID19CXRDataset:
Number of samples: 21165
Number of classes: 4
Class distribution: Counter({'Normal': 10192, 'Lung Opacity': 6012, \
                                    'COVID': 3616, 'Viral Pneumonia': 1345})
"""
```

We use the `.stat()` function to show the dataset information. In the dataset, there are 21,165 sample images and they belong to 4 different categories: `normal`, `lung opacity`, `covid`, and `viral pneumonia`. The label is not used in unconditional VAE and GAM model training, while they can be used to train the conditional version, such as ConditionalVAE, ConditionalGAN, that generates the data sample under certain labels.

10.4.2 Step 2: define machine learning task

We start by using the `set_task()` function to process the dataset into a dictionary structure. Each sample is formatted as a dictionary, and the dataset's default task is to classify chest X-ray images into different categories. We use the default task to process the dataset.

```
base_dataset.default_task
# COVID19CXRClassification(task_name='COVID19CXRClassification', \
#            input_schema={'path': 'image'}, output_schema={'label': 'label'})

sample_dataset = base_dataset.set_task()
```

Next, we add another sample transformation step to further clean up the image data. The transformation first converts the data into tensor format and scales the image pixel

[14] https://www.kaggle.com/datasets/tawsifurrahman/covid19-radiography-database

intensity into the range of $[0, 1]$. We then use the "Lambda" function to transform all images into three channels. This custom lambda transformation checks if the input image x has three channels (i.e., RGB). If the image already has three channels, it is returned as is. Otherwise, if the image has a single channel (e.g., grayscale), it is repeated three times along the channel dimension to convert it to a three-channel image. We also resize the chest X-ray image from 228×228 to 128×128 to make it smaller for efficiency of the demonstration.

```
from torchvision import transforms

the transformation automatically normalize the pixel intensity into [0, 1]
transform = transforms.Compose([
    # use three channels
    transforms.Lambda(lambda x: x if x.shape[0] == 3 else x.repeat(3, 1, 1),
    transforms.Resize((128, 128)),
    transforms.Normalize(mean=[0.5862785803043838], std=[0.27950088968644304])
])

def encode(sample):
    sample["path"] = transform(sample["path"])
    return sample

sample_dataset.set_transform(encode)
```

After transforming the data, we split it into three parts: training, validation, and test sets. The sizes of these sets are distributed in the ratio of 60% : 20% : 20% respectively. We obtained three data loaders and noticed that each chest X-ray image has a dimension of $1 \times 128 \times 128$. The size of the training set is 12,699, while the validation and test sets have a size of 4,233 each.

```
from pyhealth.datasets import split_by_visit, get_dataloader

# split dataset
train_dataset, val_dataset, test_dataset = split_by_visit(
    sample_dataset, [0.6, 0.2, 0.2]
)
train_dataloader = get_dataloader(train_dataset, batch_size=256, shuffle=True)
val_dataloader = get_dataloader(val_dataset, batch_size=256, shuffle=False)
test_dataloader = get_dataloader(test_dataset, batch_size=256, shuffle=False)

data = next(iter(train_dataloader))
print (data["path"][0].shape)
print(
    "loader size: train/val/test",
    len(train_dataset),
    len(val_dataset),
    len(test_dataset),
)

"""
torch.Size([1, 128, 128])
loader size: train/val/test 12699 4233 4233
"""
```

10.4.3 Step 3: initialize the VAE model

PyHealth offers the VAE API, which has arguments similar to other models. However, in this case, the feature key and the label key are the same, and we use the regression mode. In VAE, the input is the data sample, and the reconstruction supervision is also the same data sample (image "path" in this example), which is treated as a regression task in PyHealth. The hidden dimension represents the dimension of the Gaussian distribution. There are two new arguments: (i) `input_channel` indicates the number of channels in the input, and (ii) `input_size` represents the size of the data (height or weight), and they must be the same. Modifying these two arguments will lead to different model architectures, such as the number of layers, etc.

```
from pyhealth.models import VAE

model = VAE(
    dataset=sample_dataset,
    input_channel=3,
    input_size=128,
    feature_keys=["path"],
    label_key="path",
    mode="regression",
    hidden_dim = 256,
)
```

10.4.4 Step 4&5: model training and inference

To train the model, we use the *pyhealth.trainer* API. The learning task is treated as a regression task, and we record the KL divergence, mean square error (MSE), and mean absolute error (MAE) in this generation application. We monitor the change in KL divergence over each epoch and consider the epoch with the lowest KL divergence as the best trained model.

```
from pyhealth.trainer import Trainer

trainer = Trainer(model=model, device="cuda:0", \
                    metrics=["kl_divergence", "mse", "mae"])
trainer.train(
    train_dataloader=train_dataloader,
    val_dataloader=val_dataloader,
    epochs=20,
    monitor = "kl_divergence",
    monitor_criterion = 'min',
    optimizer_params={"lr": 2e-3},
)
```

We use the evaluation function to assess the reconstruction error on the test data, which is shown below.

```
# evaluate on the test data
print(trainer.evaluate(test_dataloader))

```

```
"""
{'kl_divergence': 0.092681274, 'mse': 2.3712183e-15, \
                'mae': 3.5363602e-08, 'loss': 19134.9755565672}
"""
```

10.4.5 Evaluate the reconstruction

We use the `inference()` function to reconstruct all the samples in the test set and then compare with the original data samples. As shown in Figure 10.15, the reconstructed image can restore the overall shape and some patterns of the original image, however, detailed information is missing.

```
X, X_rec, _ = trainer.inference(test_dataloader)

import matplotlib.pyplot as plt

idx = 4 # choose which image to visualize
plt.figure()
plt.subplot(1, 2, 1)
plt.imshow(X[idx].reshape(128, 128), cmap="gray")
plt.subplot(1, 2, 2)
plt.imshow(X_rec[idx].reshape(128, 128), cmap="gray")
plt.show()
```

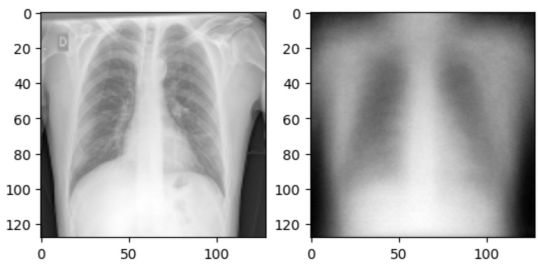

Figure 10.15 Compare the original data (left) and the reconstruction (right)

10.4.6 Generating new chest xray images

After training the VAE model, we can use the decoder to create new chest X-ray images that have a similar style to the ones used for training. To generate these images, we start by selecting a random vector from a multi-variant normal distribution. We have chosen to use 128 as the hidden gaussian dimension, which means we generate a 128-dimensional vector $\mathbf{x}$. We then input this vector into the decoder model, which reshapes the output into a 128x128 image.

```
import torch
import numpy as np
```

```

model = trainer.model

model.eval()
with torch.no_grad():
    # we generate a random 64-dimensional multi-gaussian vector
    x = np.random.normal(0, 1, 256)

    x = x.astype(np.float32)
    x = torch.from_numpy(x).to(trainer.device)\
                    .unsqueeze(0).unsqueeze(2).unsqueeze(3)
    rec = model.decoder(x).detach().cpu().numpy() #convert to numpy array
    rec = rec.reshape((3, 128, 128))[0] #reshape to an image
    plt.imshow(rec, cmap="gray")
    plt.show()
```

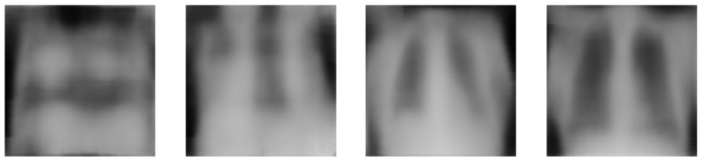

Figure 10.16 A synthesized chest X-ray image from VAE

Remarks

The generated images or the reconstructed images seem to have poor qualities. Improvements are possible with more training data, more training epochs, and better encoder and decoder models. Readers can try the pipeline on their own and tune the models in this repository[15].

10.4.7 Use the GAN model for generation

In this section, we will be using the GAN model from PyHealth. This model will be used to generate Chest X-ray images. The GAN model requires alternative updating steps for training the discriminator and the generator separately. Therefore, we will create our own training pipeline on top of the PyHealth GAN model.

To initialize the GAN model, we set the input image channels to 3, the input image size to 128, and the hidden dimension to 256 (random vector size). The GAN model we are using is based on a deep CNN that uses 3-channel images with a resolution of 128. We will use the same train/val/test loaders from the VAE pipeline.

```
from pyhealth.models import GAN

model = GAN(
    input_channel=3,
    input_size=128,
```

[15] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap8-GenAI/notebook

```
    hidden_dim = 256,
)
```

Training GAN models

In the following steps, we will demonstrate how to train a GAN model. Firstly, we define the loss function for the discriminator and use two different optimizers for the generator and the discriminator.

We keep track of the loss of the generator and discriminator separately and store them in `curve_G` and `curve_D` respectively.

During the training, both real images (with label 1) and generated (fake, with label 0) images are used to train the discriminator. Then, we use the generated images to train the generator, while ignoring the loss function for the discriminators. Through this process, the discriminator learns to distinguish between real and fake images, and the generator becomes stronger in generating fake images that can fool the discriminator.

```
import torch
from tqdm import tqdm

device = torch.device("cuda:0")

# Loss function
loss = torch.nn.BCELoss()
opt_G = torch.optim.AdamW(model.generator.parameters(), lr=1e-3)
opt_D = torch.optim.AdamW(model.discriminator.parameters(), lr=1e-4)

model.to(device)

curve_D, curve_G = [], []

for epoch in range(30):
    curve_G.append(0)
    curve_D.append(0)
    for batch in tqdm(train_dataloader):

        """ train discriminator """
        opt_D.zero_grad()

        real_imgs = torch.stack(batch["path"], dim=0).to(device)
        batch_size = real_imgs.shape[0]
        fake_imgs = model.generate_fake(batch_size, device)

        real_loss = loss(model.discriminator(real_imgs), \
                         torch.ones(batch_size, 1).to(device))
        fake_loss = loss(model.discriminator(fake_imgs), \
                         torch.zeros(batch_size, 1).to(device))
        loss_D = (real_loss + fake_loss) / 2

        loss_D.backward()
        opt_D.step()

        """ train generator """
        opt_G.zero_grad()
```

```
        loss_G = loss(model.discriminator(fake_imgs), \
                            torch.ones(batch_size, 1).to(device))

        loss_G.backward()
        opt_G.step()

        curve_G[-1] += loss_G.item()
        curve_D[-1] += loss_D.item()

    print (f"epoch: {epoch} --- loss of G: {curve_G[-1]}, \
                                    loss of D: {curve_D[-1]}")

"""
epoch: 0 --- loss of G: 223.14766198396683, loss of D: 14.372084847651422]
epoch: 1 --- loss of G: 466.95815896987915, loss of D: 0.5429049517260864
epoch: 2 --- loss of G: 541.9039497375488, loss of D: 0.1549493758357l3
epoch: 3 --- loss of G: 585.4883279800415, loss of D: 0.08368747876374982
...
epoch: 26 --- loss of G: 83.02981567382812, loss of D: 35.7115815281868
epoch: 27 --- loss of G: 79.00199449062347, loss of D: 44.660759538412094
epoch: 28 --- loss of G: 76.95186269283295, loss of D: 39.98225525021553
epoch: 29 --- loss of G: 68.42279094457626, loss of D: 42.02642643451691
"""
```

Synthetic image from GAN

Similarly, we let GAN to generate synthetic chest X-ray image, shown in Figure 10.17.

```
import torch
import numpy as np
import matplotlib.pyplot as plt

model.eval()
with torch.no_grad():
    fake_imgs = model.generate_fake(1, device).detach().cpu()
    plt.imshow(fake_imgs[0][0], cmap="gray")
    plt.show()
```

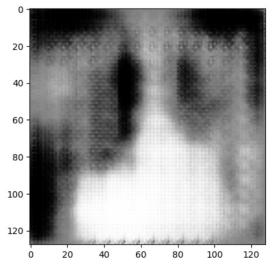

Figure 10.17 A synthesized chest xray image generated by GAN

10.4.8 Use the latent diffusion model for generation

We will be using the diffusion model to create synthetic chest X-ray images. However, simple diffusion models can be difficult to train when working with raw images that have high dimensions. To address this, we use a latent diffusion model (LDM) that operates on a latent space. Here's how it works:

- It first trains a VAE model on the training data, and then the VAE model is completely fixed at later steps.
- Then, on the same training data, we encode the training images into latent space by the VAE encoder. We train a diffusion model on the latent embeddings.
- During data generation, a random vector is drawn from Gaussian distribution, and the vector runs through the reverse process of the diffusion model to generate a latent embedding. The latent embedding is then feeded into the VAE decoder to generate a synthetic chest X-ray image.

Remarks

The LDM combines the strengths of both VAE and the simple diffusion model. VAE is great at encoding data into a low-dimensional latent space, but its generation power is often limited due to the distribution of the latent embeddings not following an independent multi-variate Gaussian. On the other hand, the diffusion model excels at generating synthetic data that follows the distribution in low-dimensional space, but can fail or be hard to train on high-dimensional spaces. The LDM overcomes these issues by leveraging the diffusion model to transform random noise into a low-dimensional distribution that aligns with VAE latent space. Then, it uses the VAE decoder to generate a high-dimensional real image. Below, you can find the code.

Train a VAE model as before

We copy the previous codes below to train a VAE model first.

```
from pyhealth.models import VAE

vae = VAE(
    dataset=sample_dataset,
    input_channel=3,
    input_size=128,
    feature_keys=["path"],
    label_key="path",
    mode="regression",
    hidden_dim = 256,
)
```

```
from pyhealth.trainer import Trainer

trainer = Trainer(model=vae, device=device, metrics=["kl_divergence", "mse", "
                                        mae"])
trainer.train(
    train_dataloader=train_dataloader,
    val_dataloader=val_dataloader,
```

```
    epochs=20,
    monitor = "kl_divergence",
    monitor_criterion = 'min',
    optimizer_params={"lr": 1e-3},
)
```

We generate random vectors and directly use the VAE decoder for generation, shown in Figure 10.18. As we mentioned in the Remark, the latent space of VAE usually does not align with independent multi-variate Gaussian distribution, and thus the generation from a Gaussian random noise does not look good.

```
import torch
import numpy as np

N = 4

vae = trainer.model
vae.eval()
# we generate a random 64-dimensional multi-Gaussian vector
x = np.random.normal(0, 1, (N, 256))
x = x.astype(np.float32)
x = torch.from_numpy(x).to(trainer.device).unsqueeze(2).unsqueeze(3)

with torch.no_grad():
    rec = vae.decoder(x).detach().cpu()

plt.figure(figsize=(7, 5))
for i in range(N):
    plt.subplot(1, N, i+1)
    plt.imshow(rec[i].reshape(3, 128, 128).permute(1,2,0))
    plt.axis('off')
plt.show()
```

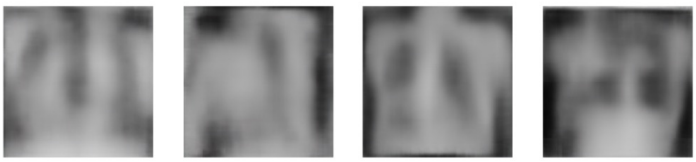

Figure 10.18 Several synthesized chest x-ray images generated by VAE

Build the LDM model

Now, we will train a diffusion model to improve the generation quality. We build the diffusion model, which is almost the same as the one we used for MNIST that takes the image `x`, the diffusion steps `step`, and the image label `label` as input. However, the input data dimension becomes the VAE bottleneck layer dimension 256.

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

```

class Diffuser(nn.Module):
    def __init__(self, diff_steps, n_labels):
        super(Diffuser, self).__init__()

        self.diffusion = nn.Sequential(
            nn.Linear(256 + diff_steps + n_labels, 256),
            nn.ReLU(),
            nn.Linear(256, 256),
        )

        self.diff_steps = diff_steps
        self.n_labels = n_labels

    def forward(self, x, step, label):

        # one hot for diffusion step
        step_encoding = F.one_hot(
            torch.tensor([step]*x.shape[0]), num_classes = self.diff_steps
        ).to(device)

        # one hot for label
        labels_encoding = F.one_hot(label, num_classes = self.n_labels).to(
                                                device)

        # concat features and run diffusion process
        emb = torch.cat([x, step_encoding, labels_encoding], 1)
        out = self.diffusion(emb)

        # normalize the output
        MEAN = out.mean(dim=1, keepdim=True)
        STD = out.std(dim=1, keepdim=True) + 1e-8
        out = (out - MEAN) / STD * sigma
        return out

diffusion_steps = 25
ldm = Diffuser(25, 4)
ldm.to(device) # move model to GPU
```

We use the same random noise sampler as defined below.

```
### sampler
sigma = 0.24
# normal distribution sampler (we could throw to cuda for once)
unit_normal = torch.distributions.Normal(0, sigma)
unit_normal.loc = unit_normal.loc.to(device)
unit_normal.scale = unit_normal.scale.to(device)
```

let us evaluate the performance of the `ldm` model before training. The generation process starts from a random noise, then it goes through the reverse process to be a latent vector, and next the latent vector is transformed into a Chest X-ray image by the fixed VAE decoder. We shown the initial performance in Figure 10.19. It is interesting that even before model training, the LDM generated images looks better than VAE generated images.

```
def evaluate(model, diffusion_steps):

    # 0, 1, 2, 3
    labels = torch.arange(4)

    # random images
    noisy_images = unit_normal.sample((len(labels),256)) \
                                    * np.sqrt(diffusion_steps)

    plt.figure(figsize=(7, 5))

    # we run diffusion process for multiple steps
    # to denoise the noisy images into a digit image
    for diffusion_step in range(diffusion_steps):
        step = diffusion_steps - diffusion_step - 1
        with torch.no_grad():
            noise = ldm(noisy_images, step, labels)
        noisy_images -= noise.view(len(labels), 256)

    for i in range(4):
        plt.subplot(1, 4, i+1)
        # print the digit one by one
        latent = noisy_images[i:i+1].unsqueeze(-1).unsqueeze(-1)
        with torch.no_grad():
            reconstruction = vae.decoder(latent)[0]
        plt.imshow(reconstruction.cpu().permute(1, 2, 0))
        plt.axis('off')
    plt.show()

evaluate(model, diffusion_steps)
```

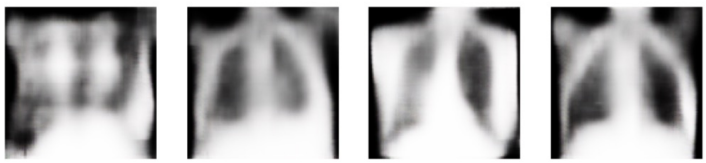

Figure 10.19 Several synthesized chest xray images generated by the LDM model (before training)

Train the LDM model

We train the LDM model before using similar pipeline as we used for training a diffusion model on MNIST, except that before feeding into the model, we transform the images into latent space by VAE encoder.

```
from tqdm import tqdm

optimizer = torch.optim.Adam(ldm.parameters(), 0.1)
criterion = nn.MSELoss()

mapping = {"Normal": 0, "COVID": 1, "Lung Opacity": 2, "Viral Pneumonia":3}
```

```

for epoch in range(25):
    epoch_loss = 0
    for batch in train_dataloader:
        images = torch.stack(batch["path"], dim=0).to(device)
        labels = torch.LongTensor([mapping[item] \
                        for item in batch["label"]]).to(device)
        with torch.no_grad():
            latents = vae.encoder(images)[0]

        for step in torch.randperm(diffusion_steps):
            # get a sample at step: original image + sqrt(step) * normal sample
            less_noisy = latents + \
                    unit_normal.sample(latents.shape) * np.sqrt(step)
            # get a normal sample
            one_step_noise = unit_normal.sample(latents.shape)
            # get the sample at (step + 1)
            more_noisy = less_noisy + one_step_noise

            output = ldm(more_noisy, step, labels)
            loss = criterion(output, one_step_noise)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()
    print(f'Epoch: {epoch} \t Loss: {epoch_loss}')

    if epoch % 5 == 0:
        evaluate(ldm, diffusion_steps)

"""
Epoch: 0   Loss: 144.3587410505861
Epoch: 1   Loss: 136.6373864300549
Epoch: 2   Loss: 135.48954015411437
Epoch: 3   Loss: 134.957813994959
Epoch: 4   Loss: 134.53044639341533
Epoch: 5   Loss: 134.18359554558992
...
Epoch: 21    Loss: 132.3953073080629
Epoch: 22    Loss: 132.3351447628811
Epoch: 23    Loss: 132.276965896599
Epoch: 24    Loss: 132.23989922646433
"""
```

We show the LDM generated images after training in Figure 10.20, which have higher qualities compared to the generated images from VAE or GAN.

The notebook resources could be found at the open repository [16], and readers could play with the latent diffusion model on their own laptop.

[16] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap8-GenAI/notebook

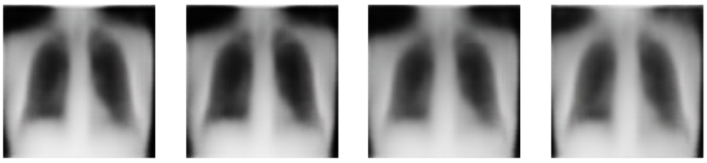

Figure 10.20 Several synthesized Chest X-ray images generated by the LDM model (after training)

10.5 Takeaways

- **Generative Models:** Different from previous supervised models that mainly predict some targeted values or classes based on complex data features, generative models are built mostly on unlabeled data and could synthesize new data samples.
- **Variational Auto-encoder (VAE):** VAE is one type of generative model that leverages the auto-encoder architecture and enforces the middle bottleneck layer to have standard distribution. It leverages MSE reconstruction loss and KL divergence as the objective.
- **Generative Adversarial Network (GAN):** GAN is another type of generative model that employs a discriminator and a generator where the generator strives to generate realistic samples from random noise and the discriminator tries to distinguish the real samples and the generated samples. The model is trained in an adversarial way by optimizing the generator and the discriminator alternatively.
- **Diffusion Models:** Diffusion model is a newly developed generative model that add noise to the real data progressively and then optimizes a model to predict the per-step noise which could gradually generate a realistic data from full noise.
- **Latent Diffusion Models (LDM):** LDM is practically a better choice than diffusion model. LDM needs a well-trained VAE model as predecessor. The trained VAE model first uses its encoder to map high-dimensional samples into latent low-dimensional space where the LDM is trained. During generation, the noise goes through the LDM model and then uses the VAE decoder to synthesize data.

Questions

- What are three generative models introduced in the chapter? When are they proposed?
- What is the main architecture of the AE model? How would you design the AE model for learning image representations from unlabeled image data? Could you specify the model architecture below?
- What are different ways to prevent overfitting in an AE model? Could you elaborate how they works?
- What is VAE? What are the main difference between VAE and common AE models?
- What is evidence lower bound (ELBO) in VAE? Could you derive the inequality?

- Elaborate the architecture of the GAN model and write down its objective functions.
- Discussion the difference between VAE and GAN.
- What are the major problems in training GAN models? Could you come up with some ideas to mitigate or improve the problem? Why they help? (you may search the google and show your answer with citations).
- Explain the diffusion models in a high level. Write down the diffusion forward and reverse process using equations or bullet points.
- What are two different ways to create intermediate noisy sample in diffusion models?
- In the reverse process of diffusion model, please use Bayesian rule to derive the quantity $q(\mathbf{X}_{t-1} \mid \mathbf{X}_t, \mathbf{X}_0)$ by three probability quantity from the diffusion process, $q(\mathbf{X}_t \mid \mathbf{X}_{t-1}, \mathbf{X}_0)$, $q(\mathbf{X}_{t-1} \mid \mathbf{X}_0)$, $q(\mathbf{X}_t \mid \mathbf{X}_0)$. Could you further simplify the results? HINT: by the Markov property, we could further decompose $q(\mathbf{X}_t \mid \mathbf{X}_0)$ into $q(\mathbf{X}_t \mid \mathbf{X}_{t-1})q(\mathbf{X}_{t-1} \mid \mathbf{X}_{t-2}) \cdots q(\mathbf{X}_1 \mid \mathbf{X}_0)$.
- **Programming Question:** Could you use either VAE or GAN or LDM to generate one synthetic image for bird and cat? Your training data is the entire CIFAR-10. You could build VAE or GAN or Diffusion model from scratch using PyTorch or directly use existing models from PyHealth. The CIFAR-10 dataset could be downloaded from [17].

[17] https://www.cs.toronto.edu/ kriz/cifar.html