به نام خدا

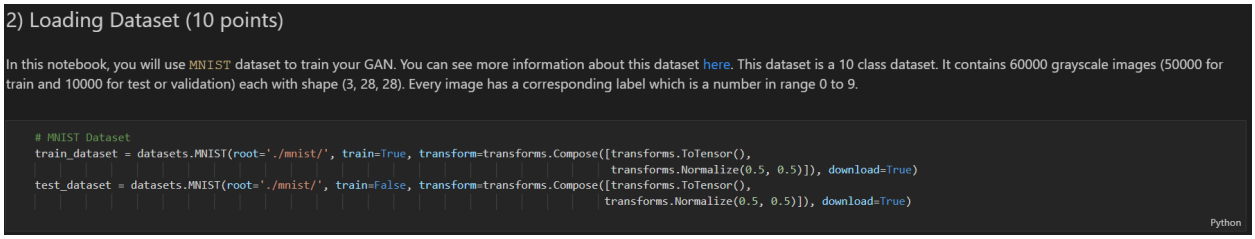
یادگیری عمیق، تمرین ۵، گزارش بخش عملی
مهدی کافی ۹۹۲۱۰۷۵۳

- کدهای بخش عملی تماما توسط بنده و با درک نوشته شده است و با دانشجو دیگری همکاری نداشته‌ام. از اینترنت کمک گرفته‌ام ولی هیچ کپی کردنی صورت نگرفته است.

مسئله‌ی ۶. (۲۰ نمره)

هدف این سوال طراحی یک شبکه ساده GAN می‌باشد که بتواند تصاویر دادگان MNIST را تولید نماید. فایل GAN.ipynb را براساس موارد خواسته شده تکمیل نمایید. دقت کنید که بخش هایی از نمره بستگی به نتایج بدست آمده دارند. باتوجه به نکات تمرین و مواردی که در کلاس عنوان شده، ساختار شبکه و تابع خطا و پارامترهای دیگر را طوری طراحی و انتخاب نمایید که در نهایت تصاویر باکیفیتی توسط Generator تولید شود و فرایند آموزش پایدار باشد. در نهایت فایل تکمیل شده به همراه نتایج را بعلاوه فایل پارامترهای شبکه Generator ای که آموزش داده‌اید به همراه دیگر بخش‌های تمرین ارسال نمایید.

در اولین بخشی که نیاز است کامل کنیم، دیتاست MNIST را لود می‌کنیم. در هنگام لود کردن، دیتاست مقادیر پیکسل‌های تصویر را با transform نوشته شده به بازه [1, -1] می‌بریم.



در بخش بعدی مقادیر هایپرپارامترها را ست می‌کنیم.

```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

################ Problem 01 (5 pts) ################
# define hyper parameters
batch_size = 100
d_lr = 2*1e-4
g_lr = 2*1e-4
n_epochs = 200
######################### End ##########################
z_dim = 100
```

سپس از روی دیتاست لود شده و هایپرپارامتر اندازه بچ، DataLoaderهای مربوط به داده آموزش و تست را می‌سازیم.

```
################ Problem 02 (5 pts) ################
# Define Dataloaders
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
######################## End ########################
```

در این بخش معماری شبکه تمیزدهنده را مشخص می‌کنیم که از لایه‌های Linear و تابع فعال سازی LeakyReLU و تکنیک Dropout استفاده می‌کنیم. در لایه آخر از تابع فعال سازی Sigmoid استفاده می‌کنیم زیرا که می‌خواهیم احتمال واقعی بودن ورودی شبکه را مشخص کنیم.

```
self.discriminator = nn.Sequential(
    ################# Problem 03 (15 pts) #################
    # use linear or convolutional layer
    # use arbitrary techniques to stabilize training
    # transforms.Normalize(0.5, 0.5),
    # nn.Flatten(),
    nn.Linear(784, 1024),
    nn.LeakyReLU(0.2),
    nn.Dropout(0.3),
    nn.Linear(1024, 512),
    nn.LeakyReLU(0.2),
    nn.Dropout(0.3),
    nn.Linear(512, 256),
    nn.LeakyReLU(0.2),
    nn.Dropout(0.3),
    nn.Linear(256, 1),
    nn.Sigmoid()
    ######################## End ########################
)
```

در بخش بعدی معماری شبکه مولد را مشخص می‌کنیم. در این معماری نیز از لایه‌های Linear و تابع فعال سازی LeakyReLU استفاده می‌کنیم. در لایه آخر تابع فعال سازی Tanh را استفاده می‌کنیم زیرا که مقادیر پیکسل‌های تصاویر واقعی بین ۱ و ۱- هستند و تابع Tanh نیز خروجی شبکه مولد را بین ۱ و ۱- قرار می‌دهد.

```
self.generator = nn.Sequential(
    ################ Problem 04 (15 pts) ################
    # use linear or convolutional layer
    # use arbitrary techniques to stabilize training
    nn.Linear(100, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 512),
    nn.LeakyReLU(0.2),
    nn.Linear(512, 1024),
    nn.LeakyReLU(0.2),
    nn.Linear(1024, 784),
    nn.Tanh()
    ######################### End ########################
)
```

در بخش بعدی از هر یک از کلاس‌های شبکه‌های مولد و تمیزدهنده یک نمونه می‌سازیم.

```
# Create instances of modules (discriminator and generator)
# don't forget to put your models on device
discriminator = Discriminator().to(device)
generator = Generator().to(device)
```

در بخش بعدی، بهینه سازهای هر کدام از شبکه‌ها را مشخص می‌کنیم که برای هر دو شبکه از بهینه ساز Adam با هایپرپارامتر مشخص شده در ابتدای کد استفاده می‌کنیم.

```
# Define two optimizer for discriminator and generator
d_optimizer = optim.Adam(discriminator.parameters(), lr=d_lr)
g_optimizer = optim.Adam(generator.parameters(), lr=g_lr)
```

در این بخش تعدادی از متغیرهایی که نیاز داریم را ست می‌کنیم. به طور مثال در ابتدا مقدار batch_size را از روی سایز داده این batch که در حلقه از DataLoader مربوط به داده آموزش خوانده شده است، مشخص می‌کنیم. سپس تصاویر را Flat می‌کنیم که برای ورودی شبکه تمیز دهنده که انتظار گرفتن یک بردار ۷۸۴ بعدی را دارد مناسب باشد. برچسب‌های واقعی را می‌سازیم که به اندازه سایز بچ، مقدار ۱ هستند. سپس بردار تصادفی از توزیع نرمال را می‌سازیم که نیاز داریم با دادن آن به شبکه مولد، تصاویر تقلبی را بسازیم. این بردار را به شبکه مولد می‌دهیم و تصاویر تقلبی را در متغیر fake_images نگهداری می‌کنیم. در نهایت برچسب‌های تقلبی که به اندازه بچ، مقدار ۰ هستند را تولید می‌کنیم.

```
batch_size = images.size(0)
real_images = images.view(batch_size, -1).to(device)
# real_images = images.to(device)
# Scale the real images between -1, 1
# for idx, image in enumerate(real_images):
#     pass
real_labels = torch.ones(batch_size).to(device)
z = torch.randn(batch_size, z_dim, requires_grad=True).to(device)
fake_images = generator(z)
fake_labels = torch.zeros(batch_size).to(device)
```

در این بخش می‌خواهیم شبکه تمیزدهنده را آپدیت کنیم. به این منظور در ابتدا تصاویر واقعی را به شبکه می‌دهیم و نتیجه تصمیم گیری آن برای واقعی بودن یا نبودن تصاویر را در متغیر real_pred می‌ریزیم و با استفاده از برچسب‌های واقعی که از قبل آماده کرده بودیم، مقدار loss را برای تصاویر واقعی محاسبه می‌کنیم. سپس با استفاده از تصاویر خروجی شبکه مولد، بار دیگر نتیجه تخمین شبکه تمیزدهنده را محاسبه و در متغیر fake_pred می‌ریزیم و این بار مقدار loss را برای داده‌های تقلبی با برچسب‌های ۰ محاسبه می‌کنیم. سپس این دو مقدار هزینه را با یکدیگر جمع می‌کنیم و گرادیان را در شبکه تمیزدهنده منتشر می‌کنیم و بهینه سازی پارامترهای شبکه را آپدیت می‌کند.

```
# calculate discriminator loss and update it
# discriminator.zero_grad()
d_optimizer.zero_grad()
# discriminator.zero_grad()

real_pred = discriminator(real_images).flatten()
real_loss = loss_fn(real_pred, real_labels)
# real_error.backward()

fake_pred = discriminator(fake_images).flatten()
fake_loss = loss_fn(fake_pred, fake_labels)
# fake_error.backward()

d_loss = real_loss + fake_loss
d_loss.backward()
d_optimizer.step()
```

در این بخش می‌خواهیم شبکه مولد را آموزش دهیم. به این منظور بار دیگر از گاوسی استاندارد، برداری تصادفی تولید می‌کنیم. با دادن این بردار به شبکه مولد، تصاویر تقلبی را تولید می‌کنیم. تصاویر تقلبی را به شبکه تمیزدهنده که در مرحله قبل کمی بهبود یافته می‌دهیم و نتیجه را در متغیر pred می‌ریزیم. مقدار هزینه را محاسبه می‌کنیم. گرادیان را منتشر می‌کنیم و بهینه ساز، پارامترها را آپدیت می‌کند.

```
# calculate generator loss and update it
z = torch.randn(batch_size, z_dim, requires_grad=True).to(device)
fake_images = generator(z)
pred = discriminator(fake_images).flatten()

g_optimizer.zero_grad()
# generator.zero_grad()
g_loss = loss_fn(pred, real_labels)
g_loss.backward()
g_optimizer.step()
```

در نهایت نیز برای نمایش تعدادی از خروجی‌های شبکه مولد، در اولین epoch، تعداد ۵ بردار تصادفی تولید می‌کنیم و پس از هر ۱۰ epoch این بردارها را به شبکه مولد می‌دهیم و خروجی آنرا چاپ می‌کنیم.

```
if epoch == 0:
    z_plot = torch.randn(5, z_dim).to(device)
if (epoch % plot_frequency == 0):
    fake_images = generator(z_plot).cpu().detach().numpy()
    fig, ax = plt.subplots(1, 5)
    # fig.title(f"Epoch {epoch+1}", fontsize=14)
    for idx in range(5):
        ax[idx].imshow(fake_images[idx].reshape(28, 28), cmap='gray')
        ax[idx].set_xticks([])
        ax[idx].set_yticks([])
    plt.show()
```

خروجی برنامه در حین آموزش مدل GAN به صورت زیر است.

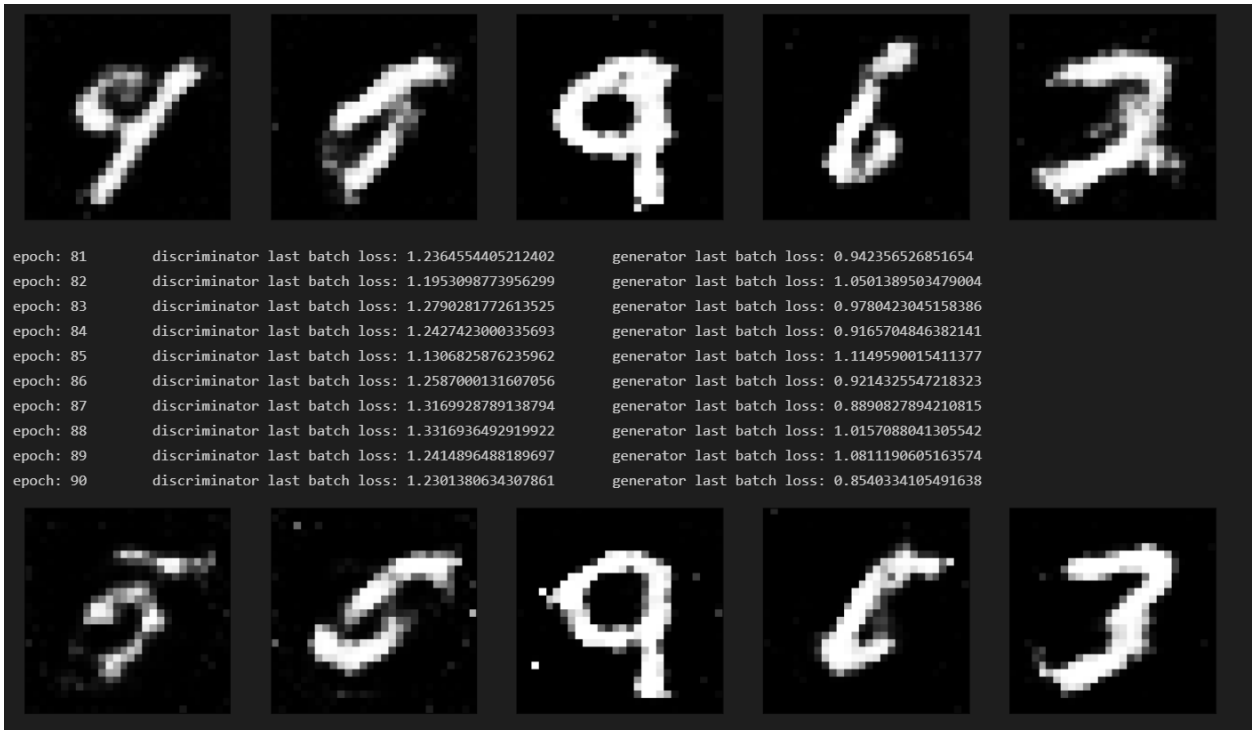




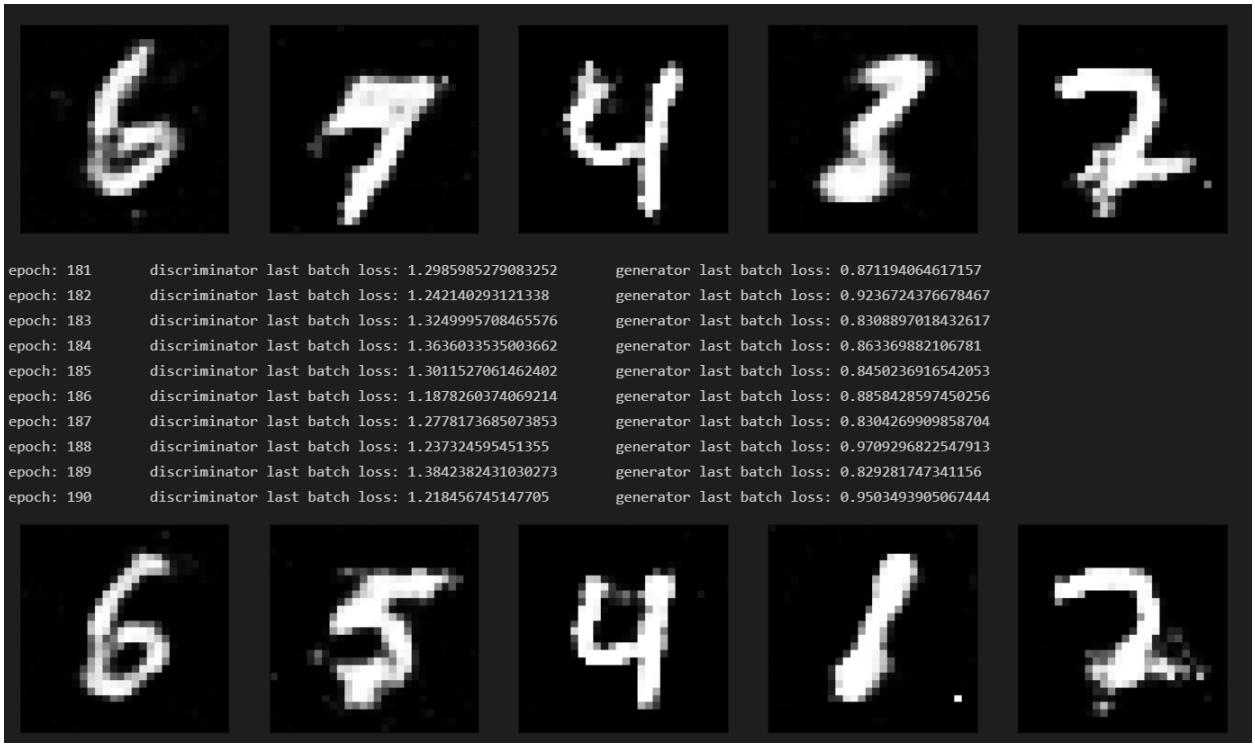
```
epoch: 1        discriminator last batch loss: 0.7932151556015015      generator last batch loss: 2.076763153076172
epoch: 2        discriminator last batch loss: 0.9708606600761414      generator last batch loss: 1.3184319734573364
epoch: 3        discriminator last batch loss: 0.9278310537338257      generator last batch loss: 1.3351424932479858
epoch: 4        discriminator last batch loss: 0.7848531007766724      generator last batch loss: 1.2587809562683105
epoch: 5        discriminator last batch loss: 1.0625941753387451      generator last batch loss: 1.5236469507217407
epoch: 6        discriminator last batch loss: 1.0328850746154785      generator last batch loss: 1.3542637825012207
epoch: 7        discriminator last batch loss: 0.9392848014831543      generator last batch loss: 1.622268795967102
epoch: 8        discriminator last batch loss: 0.8601017594337463      generator last batch loss: 1.3977737426757812
epoch: 9        discriminator last batch loss: 1.0758459568023682      generator last batch loss: 1.1877925395965576
epoch: 10       discriminator last batch loss: 1.0057966709136963      generator last batch loss: 1.226298213005066
```







```
epoch: 81       discriminator last batch loss: 1.2364554405212402       generator last batch loss: 0.942356526851654
epoch: 82       discriminator last batch loss: 1.1953098773956299       generator last batch loss: 1.0501389503479004
epoch: 83       discriminator last batch loss: 1.2790281772613525       generator last batch loss: 0.9780423045158386
epoch: 84       discriminator last batch loss: 1.2427423000335693       generator last batch loss: 0.9165704846382141
epoch: 85       discriminator last batch loss: 1.1306825876235962       generator last batch loss: 1.1149590015411377
epoch: 86       discriminator last batch loss: 1.2587000131607056       generator last batch loss: 0.9214325547218323
epoch: 87       discriminator last batch loss: 1.3169928789138794       generator last batch loss: 0.8890827894210815
epoch: 88       discriminator last batch loss: 1.3316936492919922       generator last batch loss: 1.0157088041305542
epoch: 89       discriminator last batch loss: 1.2414896488189697       generator last batch loss: 1.0811190605163574
epoch: 90       discriminator last batch loss: 1.2301380634307861       generator last batch loss: 0.8540334105491638
```



```
epoch: 181      discriminator last batch loss: 1.2985985279083252      generator last batch loss: 0.871194064617157
epoch: 182      discriminator last batch loss: 1.242140293121338       generator last batch loss: 0.9236724376678467
epoch: 183      discriminator last batch loss: 1.3249995708465576      generator last batch loss: 0.8308897018432617
epoch: 184      discriminator last batch loss: 1.3636033535003662      generator last batch loss: 0.863369882106781
epoch: 185      discriminator last batch loss: 1.3011527061462402      generator last batch loss: 0.8450236916542053
epoch: 186      discriminator last batch loss: 1.1878260374069214      generator last batch loss: 0.8858428597450256
epoch: 187      discriminator last batch loss: 1.2778173685073853      generator last batch loss: 0.8304269909858704
epoch: 188      discriminator last batch loss: 1.237324595451355       generator last batch loss: 0.9709296822547913
epoch: 189      discriminator last batch loss: 1.3842382431030273      generator last batch loss: 0.829281747341156
epoch: 190      discriminator last batch loss: 1.218456745147705       generator last batch loss: 0.9503493905067444
```

در انتها نیز پارامترهای شبکه مولد را روی کامپیوتر و در فایلی ذخیره می‌کنیم.

```
################ Problem 11 (5 pts) ################
# save state dict of your generator
torch.save(generator.state_dict(), "generator.pth")
print("Saved PyTorch Model State to generator.pth")
######################## End ########################
```

مسئله‌ی ۷. (۵ + ۲۰ نمره)

Conditional VAE یکی از ورژن‌های modified شده VAE بوده که بر خلاف VAE کلاسیک، متغیرهای مورد نیاز را به صورت conditioned نسبت به برخی متغیرهای تصادفی تخمین می‌زند. در این تمرین هدف مقایسه خروجی این دو مدل بر روی مجموعه داده MNIST می‌باشد. لطفا کد هر دو روش پیاده‌سازی شده و خروجی آن‌ها از بعد میزان وضوح تصاویر تولید شده مقایسه گردد.

برای طراحی شبکه های VAE و CVAE ، به طور کلی محدودیت چندانی وجود ندارد اما پیشنهاد می شود برای قسمت Encoder ، سه لایه کانولوشن دو بعدی به ترتیب با ۱۶، ۳۲، و ۳۲ لایه به همراه MaxPool دو بعدی ۲ در ۲ پس از هرکدام طراحی شود. برای قسمت Decoder نیز دو لایه خطی به ترتیب با ۳۲ و ۶۴ لایه طراحی گردد. برای تابع هزینه لطفا از Binary Cross Entropy به همراه KL Divergence استفاده شود. برای Optimizer نیز از Adam استفاده گردد. مابقی پارامترها همانند mean می‌تواند به صورت customize شده انتخاب گردد و بسته به خروجی بهتر تغییر کند. برای راهنمایی بیشتر می‌توانید از کد موجود در این لینک استفاده کنید. لطفا کد را کپی نکرده و صرفا برای کمک و الهام‌گیری کد زنی خود از آن استفاده شود. توجه شود حتی ساختار پیشنهادی شبکه بسته به صلاح دید شخصی شما قابل تغییر بوده فقط توجه شود که نوشتن گزارش بخش عملی الزامی بوده و دارای نمره می‌باشد لذا حتما تمامی مراحل اعم از ساختار شبکه‌ها و پارامترها باید به طور کامل در گزارش توضیح داده شوند. برای پیاده‌سازی نیز تنها مجاز به استفاده از کتابخانه pytorch می‌باشید.

برای این بخش در ابتدا بخش VAE را پیاده سازی می‌کنیم و سپس شبکه CVAE را پیاده سازی می‌کنیم.
برای شبکه VAE به صورت زیر عمل می‌کنیم. در ابتدا پکیج‌های مورد نیاز را لود می‌کنیم.

Packages

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

import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```

سپس داده‌های MNIST را لود می‌کنیم.

Loading Dataset

```
train_set = datasets.MNIST(root='./dataset', train=True, transform=transforms.ToTensor(), download=True)
train_loader = DataLoader(train_set, batch_size=100, shuffle=True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./dataset/MNIST/raw/train-images-idx3-ubyte.gz
```

سپس کلاس Encoder برای VAE را می‌نویسیم. برای معماری این شبکه از لایه‌های Convolutional2d، بچ نرمالیزیشن و تابع فعال سازی ReLU استفاده می‌کنیم. سپس که داده‌ها از این شبکه گذشتند نیاز که میانگین و واریانس توزیع لایه میانی را محاسبه کنیم که برای این دو نیز دو شبکه Linear در نظر می‌گیریم. در تابع forward هم داده‌ها را از شبکه اولیه عبور می‌دهیم و از روی خروجی آن، بردار میانگین و بردار واریانس را می‌سازیم.

VAE

Encoder

```python
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.encode = nn.Sequential(
            nn.Conv2d(1, 16, 5), nn.BatchNorm2d(16), nn.ReLU(),
            nn.Conv2d(16, 32, 5), nn.BatchNorm2d(32), nn.ReLU(),
            nn.Flatten()
        )
        self.calc_mean = nn.Linear(12800, 256)
        self.calc_logvar = nn.Linear(12800, 256)

    def forward(self, x):
        x = self.encode(x)
        mean = self.calc_mean(x)
        logvar = self.calc_logvar(x)
        return mean, logvar
```

سپس کلاس Decoder را می‌سازیم که برای این بخش در ابتدا با گرفتن یک نمونه از توزیع latent، این نمونه را از یک لایه Linear عبور می‌دهیم و با دو لایه ConvTransposed2d ابعاد بردار را به ابعاد یک تصویر ۲۸ در ۲۸ مانند داده‌های MNIST می‌رسانیم.

Decoder

```python
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.decode_fc = nn.Sequential(
            nn.Linear(256, 12800), nn.BatchNorm1d(12800), nn.ReLU()
        )
        self.decode_convtrans = nn.Sequential(
            nn.ConvTranspose2d(32, 16, 5), nn.BatchNorm2d(16), nn.ReLU(),
            nn.ConvTranspose2d(16, 1, 5), nn.Sigmoid()
        )
    def forward(self, z):
        out = self.decode_fc(z)
        out = out.view(-1, 32, 20, 20)
        out = self.decode_convtrans(out)
        return out
```

سپس کلاس VAE را می‌سازیم. در این کلاس از دو کلاس قبلی نمونه می‌سازیم و داده ورودی را که تصویر MNIST است از شبکه encoder عبور می‌دهیم. بردارهای میانگین و واریانس را داریم. با تکنیک reparameterization و دو بردار گفته شده، یک نمونه از توزیع میانی می‌سازیم و سپس نمونه را به شبکه decoder می‌دهیم و امیدواریم که تصویر شبیه به تصویر اولیه تولید کنیم. همچنین توزیع میانی را به توزیع گاوسی استاندارد نزدیک کنیم. تابع generate نیز با داشتن شبکه decoder، یک بردار تصادفی تولید می‌کند و این بردار را به شبکه decoder می‌دهد و یک تصویر در خروجی می‌دهد.

VAE

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f"Using {device} device.")

class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()
        self.encoder = Encoder().to(device)
        self.decoder = Decoder().to(device)

    def reparameterize(self, mean, logvar):
        eps = torch.randn_like(logvar).to(device)
        sigma = 0.5 * torch.exp(logvar)
        return mean + eps * sigma

    def forward(self, x):
        mean, logvar = self.encoder(x)
        z = self.reparameterize(mean, logvar)
        # print(z.size())
        return self.decoder(z), mean, logvar

    def generate(self):
        z = torch.randn((1, 256)).to(device)
        # print(z.size())
        return self.decoder(z).squeeze(0)

model = VAE().to(device)
```

```
Using cuda device.
```

تابع هزینه VAEها از دو بخش تشکیل می‌شود که بخش اول سعی می‌کند که خروجی شبکه decoder شبیه به ورودی شبکه encoder باشد (reconstruction loss) و بخش دوم سعی می‌کند که توزیع میانی را به توزیع گاوسی استاندارد نزدیک کند با استفاده از معیار KL divergence.

Loss Function

```python
BCE_loss = nn.BCELoss(reduction='sum')
def loss_fn(x, x_hat, mean, logvar):
    reconstruction_loss = BCE_loss(x_hat, x)
    KLD = 0.5 * torch.sum(-1 - logvar + torch.exp(logvar) + mean**2)
    return reconstruction_loss + KLD
```

آموزش شبکه بسیار ساده است. آموزش به صورت زیر انجام می‌شود. شبکه را برای ۲۰۰ epoch آموزش می‌دهیم.

Train VAE

```python
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
num_epochs = 200
dataset_size = len(train_loader.dataset)
for epoch in range(num_epochs):
    epoch_loss = 0
    for imgs, _ in train_loader:
        imgs = imgs.to(device)
        imgs_hat, mean, logvar = model(imgs)
        loss = loss_fn(imgs, imgs_hat, mean, logvar)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_loss += loss.cpu().item()
    epoch_loss /= dataset_size
    print(f"Epoch {epoch+1}\nLoss: {epoch_loss:.5f}\n{'-'*30}")
```

```
Epoch 1
Loss: 160.22804
------------------------------
Epoch 2
Loss: 97.51267
------------------------------
Epoch 3
Loss: 90.67088
------------------------------
Epoch 4
Loss: 86.67885
------------------------------
Epoch 5
Loss: 84.23999
------------------------------
Epoch 6
Loss: 82.75004
------------------------------
Epoch 7
Loss: 81.90334
------------------------------
Epoch 8
Loss: 81.27004
------------------------------
Epoch 9
...
Loss: 74.92893
------------------------------
Epoch 200
Loss: 74.85350
------------------------------
```

سپس با استفاده از تابع generate در کلاس VAE یک نمونه تصویر تولید می‌کنیم.

```
model.eval()
with torch.no_grad():
    generated_img = model.generate().squeeze(0)
plt.imshow(generated_img.cpu().detach().numpy(), cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f266b22ac90>
```

مشکل VAEها این است که به هنگام تولید داده جدید کنترلی بر روی کلاس داده تولیدی نداریم. به طور مثال نمی‌توانیم مشخص کنیم که می‌خواهیم از کلاس عدد ۱ داده تولید کنیم و به این منظور از CVAEها استفاده می‌کنیم. به دلیل کمبود وقت گزارش این کد را ننوشتم متاسفانه.