机器学习大作业

分工

本实验的分工如下：

管昀玫：基础要求和提高部分的GAN部分

徐昕竹：中级要求和提高部分的beta-VAE部分

孙艺齐：中级要求和提高部分的修改数据集部分

初级要求

利用 pytorch (推荐) 或 tensorflow 等神经网络框架编程实现一个变分自编码器 (Variational Auto Encoder, VAE)；分别采用交叉熵损失 (Cross Entropy Loss) 和L2损失 (MSE Loss) 构建损失函数生成新的手写数字,分析结果并展示生成的手写数字；

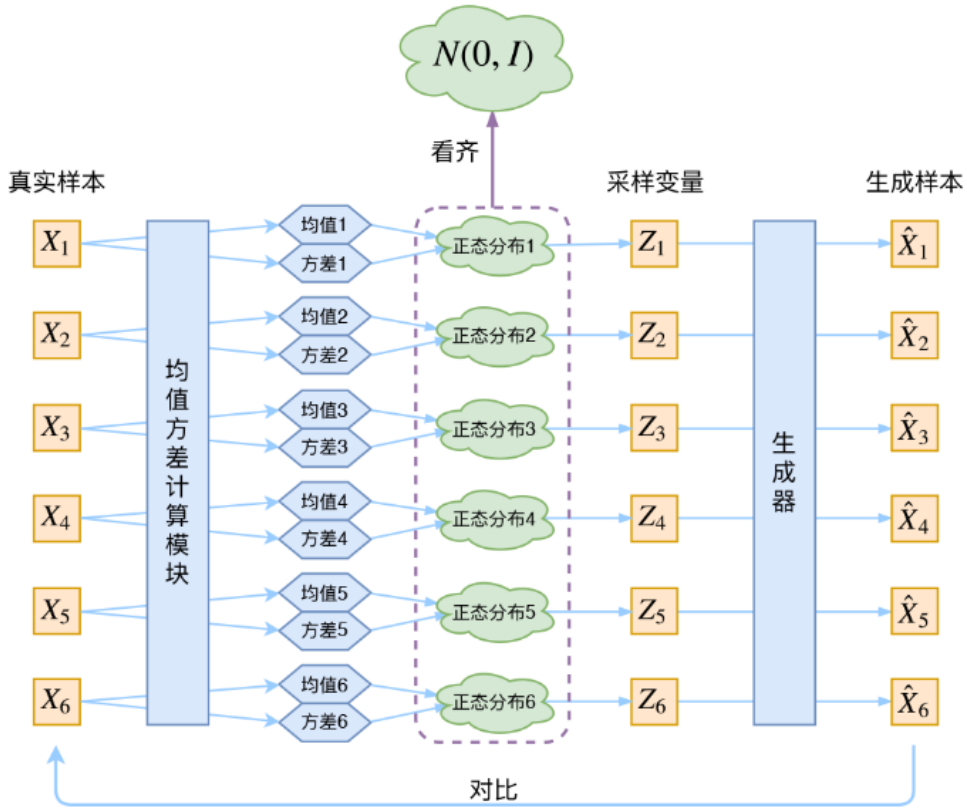
代码实现

1. 导入库和数据集

In [ ]:

```
import torch
import torchvision
import torch.utils.data
import torch.nn
import matplotlib.pyplot as plt
import numpy as np

transform=torchvision.transforms.Compose([
    torchvision.transforms.ToTensor()   #转化为tensor
])

# 下载数据集
train_set=torchvision.datasets.MNIST(
    root='./minst_data/',
    train=False,
    transform=transform,
    download=True
)

train_set_loader=torch.utils.data.DataLoader(
    dataset=train_set,
    shuffle=True,
    batch_size=64   #试试128
)

batch_sz = 64   # 数据集切分，每个batch大小为64/128

# torch.Size([64, 1, 28, 28]), 表示每个batch有64张图，只有一个channel(黑白)，长宽分别是28*28
```

2. 构建神经网络



整个神经网络的框架如图所示。首先经过一个 Encoder 进行前向传播，计算出每个样本专属分布的均值和方差，再从这个分布中采样得到隐变量 z，最后通过一个 Decoder 进行前向传播生成样本。VAE中使用神经网络生成均值 $\mu = f_1(X)$ 和对数方差 $log\sigma^2 = f_2(X)$，方差是非负的，而对数方差是可正可负，更加方便。

- encoder: 由三个相同的部分依次连接，分别为卷积——批处理——激活。添加了一些批处理法线层，使其在潜在空间中具有更强的特性。与标准自动编码器不同，编码器返回均值和方差矩阵，我们使用它们来获得采样的潜在向量。
- decoder: 去卷积与激活层组成

一般来讲， 我们通过 encoder 得到的隐含向量并不是一个标准的正态分布， 为了衡量两种分布的相似程度， 我们使用 KL divergence， 这是用来衡量两种分布相似程度的统计量，它越小，表示两种概率分布越接近。

在实际情况中，需要在模型的准确率和encoder得到的隐含向量服从标准正态分布之间做一个权衡，所谓模型的准确率就是指解码器生成的图片与原始图片的相似程度。可以让神经网络自己做这个决定，只需要将两者都做一个loss，然后求和作为总的loss，这样网络就能够自己选择如何做才能使这个总的loss下降。

$$D_{KL} = KL(q(z|x)||p(z|x)) = \int q(z|x)log\frac{q(z|x)}{p(z|x)}dz$$

为了避免计算 KL divergence 中的积分， 我们使用重参数的技巧， 不是每次产生一个隐含向量， 而是生成两个向量， 一个表示均值， 一个表示标准差， 这里我们默认编码之后的隐含向量服从一个正态分布的之后， 就可以用一个标准正态分布先乘上标准差再加上均值来合成这个正态分布， 即$Z = \mu + \epsilon \times \sigma$。最后 loss 就是希望这个生成的正态分布能够符合一个标准正态分布， 也就是希望均值为 0， 方差为 1。

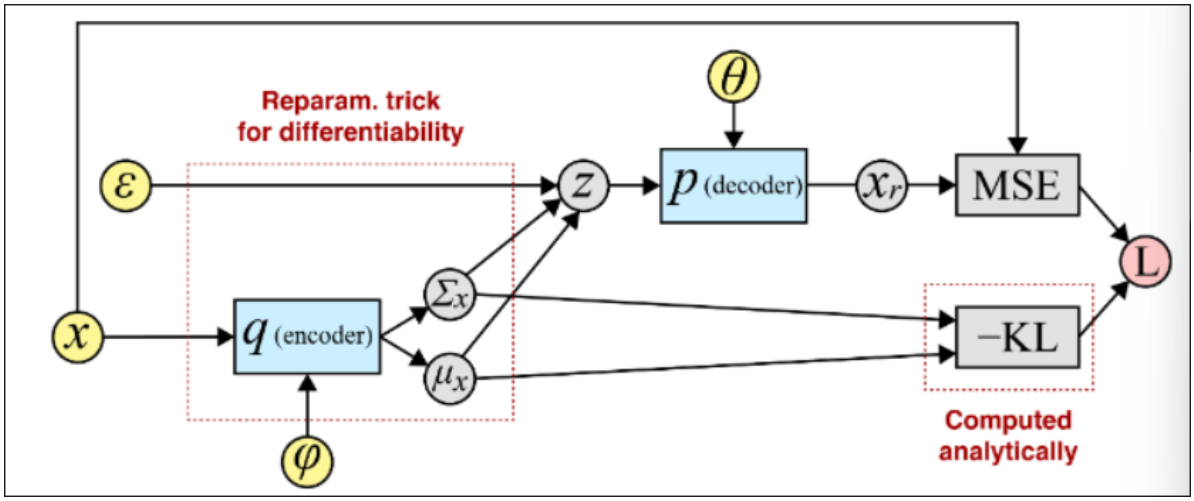
In [3]:

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

        # encoder部分
        self.encoder =torch.nn.Sequential(
        torch.nn.Conv2d(1, 64, kernel_size=4, stride=2, padding=1),     #14*14
        torch.nn.BatchNorm2d(64),
        torch.nn.LeakyReLU(0.2, inplace=True),  # nn.Sigmoid()

        torch.nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),   #7*7
        torch.nn.BatchNorm2d(128),
        torch.nn.LeakyReLU(0.2, inplace=True),

        torch.nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),  #7*7
        torch.nn.BatchNorm2d(128),
        torch.nn.LeakyReLU(0.2, inplace=True),
        )

        # Linear(): 对输入数据进行线性变换
        self.get_mu=torch.nn.Sequential(
            torch.nn.Linear(128 * 7 * 7, 32)    # 128 * 7 * 7 -> 32
        )
        self.get_logvar = torch.nn.Sequential(
            torch.nn.Linear(128 * 7 * 7, 32)    # 128 * 7 * 7 -> 32
        )
        self.get_temp = torch.nn.Sequential(
            torch.nn.Linear(32, 128 * 7 * 7)    # 32 -> 128 * 7 * 7
        )

        # conv_transpose2d: 在由几个输入平面组成的输入图像上应用二维转置卷积，有时也称为“去卷积”。
        self.decoder = torch.nn.Sequential(
        torch.nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1),
        torch.nn.ReLU(inplace=True),

        torch.nn.ConvTranspose2d(64, 1, kernel_size=4, stride=2, padding=1),
        torch.nn.Sigmoid()
        )

    # 重参数化生成隐变量
    def get_z(self,mu,logvar):
        eps=torch.randn(mu.size(0),mu.size(1))  # 64,32 参数采样
        eps=torch.autograd.Variable(eps).cuda()
        z=mu+eps*torch.exp(logvar/2)    # z = mu + epsilon * std
        return z

    # 整个前向传播过程：编码 --> 解码
    def forward(self, x):
        out1=self.encoder(x)
        mu=self.get_mu(out1.view(out1.size(0),-1))              # 64, 128 * 7 * 7 -> 64, 32
        out2=self.encoder(x)
        logvar=self.get_logvar(out2.view(out2.size(0),-1))      # 64, 128 * 7 * 7 -> 64, 32

        z=self.get_z(mu,logvar)
        out3=self.get_temp(z).view(z.size(0),128,7,7)           # 64, 32 -> 64, 128 * 7 * 7

        return self.decoder(out3),mu,logvar
```

3. VAE的损失函数

VAE 的损失由两项组成：

- 第一项是重构项，它是通过比较输入及其对应的重构得到的。
- 另一个是正则化项，也称为编码器返回的分布与标准正态分布之间的Kullback-Leibler 散度。这个KL散度在潜在空间中起着正则化的作用，可能会使编码器返回的分布接近标准正态分布。



较为严格的loss function应该如以下代码所示：

In [4]:

```
def loss_fun(new_x,old_x,mu,logvar):
    # MSE=torch.nn.functional.mse_loss(new_x,old_x,size_average=False)
    BCE=torch.nn.functional.binary_cross_entropy(new_x,old_x,size_average=False)
    KLD=-0.5*torch.sum(1+logvar-mu.pow(2)-logvar.exp())
    # beta-VAE: 修改BCE和KLD之间的权重
    return BCE+KLD
```

可以对loss function进行一些改动。可以把BCE换成L1/L2 loss，或修改二者之间的权重。具体结果将在后文讨论。

将BCE换成MSE的代码如下：

In [5]:

```
def loss_fun(new_x,old_x,mu,logvar):
    MSE=torch.nn.functional.mse_loss(new_x,old_x,size_average=False)
    # 用L1/L2替换 效果也比较好; L1会更好
    # BCE=torch.nn.functional.binary_cross_entropy(new_x,old_x,size_average=False)
    KLD=-0.5*torch.sum(1+logvar-mu.pow(2)-logvar.exp())
    # beta-VAE: 修改BCE和KLD之间的权重
    return MSE+KLD
```

4. 优化器

实现自动反向传播与参数的更新。

In [6]:

```
vae=VAE().cuda()

optimizer=torch.optim.Adam(vae.parameters(),lr=0.001,betas=(0.9,0.999),eps=1e-08,weight_decay=0)
```

5. 进行训练

In [ ]:

```
epoch_n=250
total_loss = 0
for epoch in range(epoch_n):
    for i,(data,_) in enumerate(train_set_loader):
        old_img=torch.autograd.Variable(data).cuda()
        new_img,mu,logvar=vae.forward(old_img)
        loss=loss_fun(new_img,old_img,mu,logvar)

        total_loss+=loss.item()

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()    # 进行单次优化（参数更新）

        if i % 100 == 0:
            sample = torch.autograd.Variable(torch.randn(64, 32)).cuda()
            sample = vae.decoder(vae.get_temp(sample).view(64, 128, 7, 7)).cpu()
            torchvision.utils.save_image(sample.data.view(64, 1, 28, 28), './result_vae_minst/sample_' + str(epoch) + '.png')

            print('Train Epoch:{} -- [{}/{} ({:.0f}%)] -- Loss:{:.6f}'.format(
            epoch, i * len(data), len(train_set_loader.dataset),
            100.* i / len(train_set_loader), loss.data.item() / len(data)))


    print('====> Epoch: {} Average loss: {:.4f}'.format(epoch, total_loss / len(train_set_loader.dataset)))
```

```
F:\Anaconda3\lib\site-packages\torch\nn\_reduction.py:44: UserWarning: size_average and reduce args will be deprecated, plea
se use reduction='sum' instead.
  warnings.warn(warning.format(ret))

Train Epoch:0 -- [0/10000 (0%)] -- Loss:296.260620
Train Epoch:0 -- [6400/10000 (64%)] -- Loss:45.449142
====> Epoch: 0 Average loss: 61.2086
Train Epoch:1 -- [0/10000 (0%)] -- Loss:45.206272
Train Epoch:1 -- [6400/10000 (64%)] -- Loss:43.121449
====> Epoch: 1 Average loss: 102.6111
Train Epoch:2 -- [0/10000 (0%)] -- Loss:36.461872
Train Epoch:2 -- [6400/10000 (64%)] -- Loss:35.746552
====> Epoch: 2 Average loss: 139.6622
Train Epoch:3 -- [0/10000 (0%)] -- Loss:36.100967
Train Epoch:3 -- [6400/10000 (64%)] -- Loss:36.121994
====> Epoch: 3 Average loss: 175.0012
Train Epoch:4 -- [0/10000 (0%)] -- Loss:35.974346
Train Epoch:4 -- [6400/10000 (64%)] -- Loss:33.930527
====> Epoch: 4 Average loss: 209.4033
Train Epoch:5 -- [0/10000 (0%)] -- Loss:34.696812
```

实验结果与分析

1. loss function为BCE+KLD

下列图片依次选取为epoch = 101/161/292 。可以看到，随着训练次数的提升，数字逐渐变得清晰、可辨认。数字有一定的模糊，但整体形态没有较大差异。大致能够看出一些数字的形状，例如"6"，"0"，"9"。

2. loss function为L2 loss+KLD

以下图片分别选取自 epoch = 101/158/196。将其与BCE进行比较可以看出，使用L2 loss生成的图片会比BCE生成的图片更黯淡一些，但是字形上比BCE更加清晰。

3. loss function为L1 loss+KLD

以下图片分别选自 epoch = 102/157/196。将其与BCE与MSE对比可以发现，使用L1 loss的效果比前二者都更好。数字的轮廓更加明显，重构的图片锐度更大，而L2 loss反而会更加模糊。

4. 关于loss function的其他一些讨论

- 如果去掉KLD项：即只考虑重构项，使模型学习到一组完全随机的标准正态分布，即高斯噪声。生成的图片只有轮廓，颜色很淡。
- 如果去掉BCE项：会导致生成效果是图片的叠加。
- 如果修改两者之间的权重：即beta-VAE，将在高级要求中实现。

中级要求

实现VAE的变分推断(最好是手写推导)，描述VAE的由来以及优缺点；

变分推断

VAE的变分推断：

KL散度：$D_{KL} = KL(q(z|x)\|p(z|x))$

$$= \int q(z|x)\log\frac{q(z|x)}{p(z|x)}dz$$

$$= E_{z\sim q(z|x)}[\log\frac{q(z|x)}{p(z|x)}dz]$$

$$= E_{z\sim q(z|x)}[\log q(z|x) - \log p(z|x)]dz$$

根据贝叶斯定理：$p(z|x) = \frac{p(x|z)p(z)}{p(x)}$

$$D_{KL} = E_{z\sim q(z|x)}[\log q(z|x) - \log\frac{p(x|z)p(z)}{p(x)}]dz$$

$$= E_{z\sim q(z|x)}[\log q(z|x) - \log p(x|z) - \log p(z) + \log p(x)]dz$$

$$= E_{z\sim q(z|x)}[\log\frac{q(z|x)}{p(z)}dz] - E_{z\sim q(z|x)}\log p(x|z)dz + \log p(x)$$

$$= KL(q(z|x)\|p(z)) - E_{z\sim q(z|x)}\log p(x|z)dz + \log p(x)$$

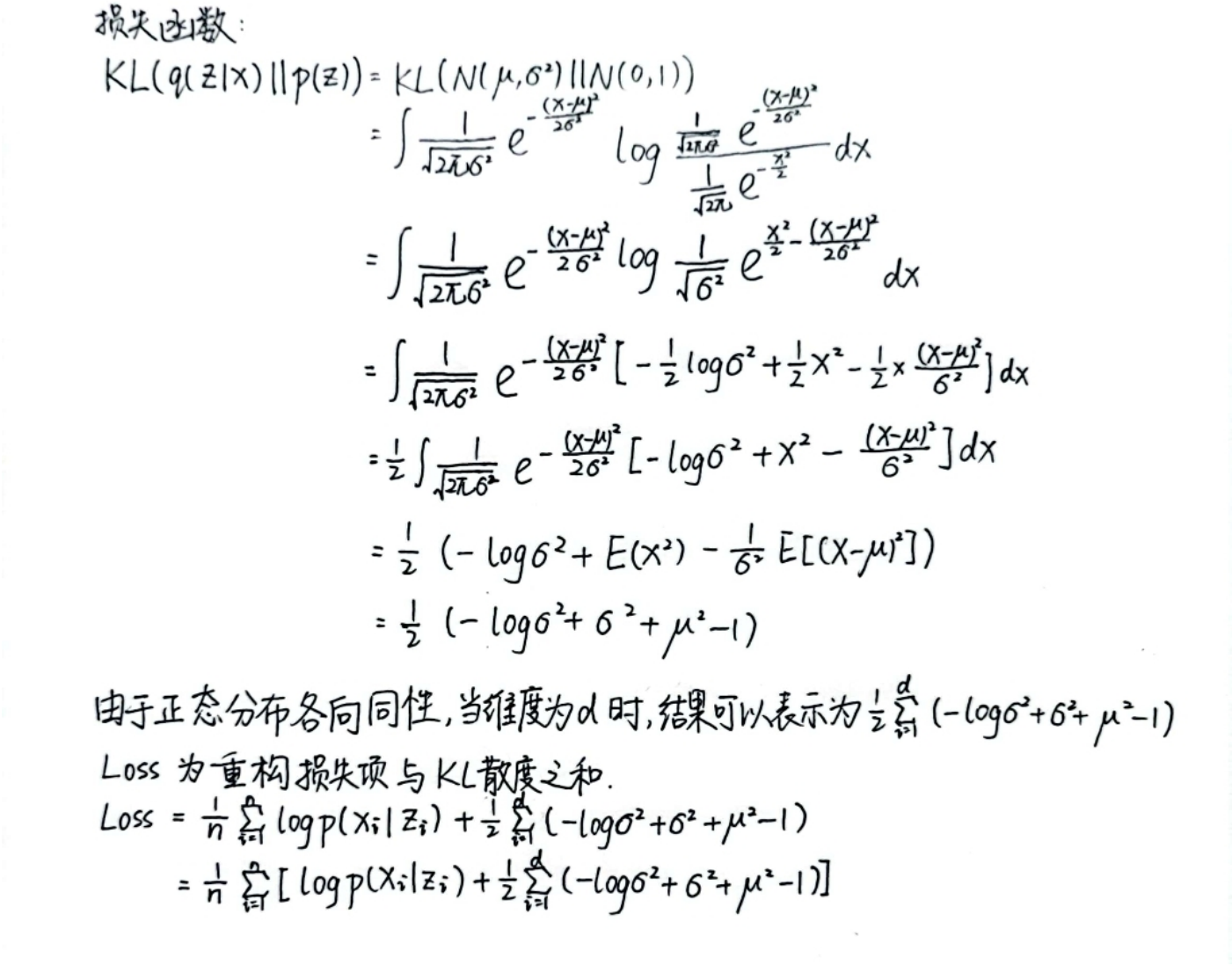
$$\therefore \log p(x) - KL(q(z|x)\|p(z|x)) = E_{z\sim q(z|x)}\log p(x|z)dz - KL(q(z|x)\|p(z))$$

等式左侧的 $\log p(x)$ 为需要最大化的对数似然函数，

$KL(q(z|x)\|p(z|x))$ 表示 $q(z|x)$ 和 $p(z|x)$ 的距离，该项需要最小化，

因此等式左侧整体需要最大化，即等式右侧式子最大化.

$\therefore$ 最大化的目标是 $ELBO = E_{z\sim q(z|x)}\log p(x|z)dz - KL(q(z|x)\|p(z))$



VAE的由来以及优缺点

由来

通常采用的统计建模方法是极大似然估计，但这种方法需要假设P(X)服从某种分布，如果假设的分布和真实的分布不一致则可能会导致结果很差；同时由于在实际问题中，无法真正描述其概率分布形式，因此我们所假设的分布基本都是错误的；在数据集是较高维度的图像时，高斯分布的参数则是高维的向量，需要训练的数据量很大，难以满足要求。

因此，为解决极大似然函数模型的问题而对其进行扩展，引入隐变量模型，将决定数字及其他影响因素用隐变量z来表示，并根据隐变量z成数字图像x，用数学描述为：$p(X) = \int p(x|z,\theta)p(z)dz$

但这个积分式子通常很难计算，而VAE就是克服这个难题而提出的模型。

优点

(1) 可以通过编码解码的步骤直接比较重建图片与原始图片之间存在的差异

(2) 产生结果更加多样化。VAE对于每个隐性参数他不会去只生成固定的一个数，而是会产生一个置信值得分布区间，因此会产生更加多样性的数据，且在隐变量空间中的分布更加均匀。

(3) 除了可以生成新的图片之外，还可以对数据的分布进行建模并学习到数据的隐含表达。

(4) 变分自编码器具有连续的潜在空间，使得随机采样和插值更加方便。

缺点

- 生成结果模糊

由于VAE没有使用对抗网络，而是采用直接计算生成图片和原始图片的均方误差的方法。这种方法存在的问题是，“各分量独立的高斯分布”不能拟合任意复杂的分布，当选定$p(z|x)$的形式后，无论怎样调整参数，都无法使$\int p(z|x)p(x)dx$成为高斯分布，即$KL(p(x,z)||q(x,z))$不可能等于0，因此会使得产生的图片较为模糊。可以采用VAE与对抗网络结合的形式对其进行改进，从而得到更好的效果。

- 存在后验崩溃的问题

z的分布都被假定为单一的高斯分布，可能会存在解码器无法从隐变量中获得正确的分布信息，自行从噪声N(0,1)中采样，原始VAE的后验分布失效的情况。在理想情况下，解码器从前者的后验分布中采样，并自行生成输出，但当解码器过于强大时，其自回归模型中学习到了前者的分布信息，此时后验分布即使有变化，网络也并不会对其采样；此外，当KL散度的约束过强时，由于通常情况下样本中的分布并不清晰，是包含一定噪声的，则无法学习到分布的解码器，会崩溃为从噪声N(0,1)中采样（即独立于后验分布自行采样）。

高级要求

GAN

GAN和VAE的实现上的区别主要在蒙特卡洛采样的时候如何解决采样次数过大的问题：

VAE路线：缩小z的取值空间。缩小p(z)的方差，那么z的采样范围会缩小，采样的次数也不需要那么大。同时，也可能把生成坏样本的z排除，生成更像真实样本的图像。

GAN路线：使用更好的相似度量标准。MSE对图像的平移、旋转等变换敏感，更换能够更好比较相似度的度量标准，不仅会有更多和真实样本相似的生成图像（这也会减少采样数量），还会生成图片的质量更好。但是实际中很难在复杂人物中找到这种度量方式。

代码实现

In [4]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torchvision.utils import save_image
from torch.autograd import Variable

import os
import numpy as np
import math

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

# 超参数部分
Epoch=200                 # 200次
batch_sz=64             # 可128
learning_rate=0.001
save_interval=400     # 一个Epoch中，每400batch存一次图片

transform = transforms.Compose([
    transforms.ToTensor(),
    # 其作用就是先将输入归一化到(0,1)，再使用公式”(x-mean)/std”，将每个元素分布到(-1,1)
    transforms.Normalize([0.5], [0.5]),
])
dataset = torchvision.datasets.MNIST(root='./data', train=True, download=False, transform=transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_sz, shuffle=True)


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

if cuda:
    Tensor = torch.cuda.FloatTensor
else:
    Tensor = torch.FloatTensor

# GAN = Generator + Discriminator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        # 全连接 + 正规化 + 激活函数
        def Normalized_fullconnect_LeakyReLU(input1, output1, normalize=True):
            layers = [nn.Linear(input1, output1)]
            if normalize:
                layers.append(nn.BatchNorm1d(output1))           # 使用正规化可以提高模型性能，避免梯度消失问题
            layers.append(nn.LeakyReLU(0.2, inplace=True))       # LeakyReLU比Sigmoid效果好
            return layers

        self.model = nn.Sequential(
            *Normalized_fullconnect_LeakyReLU(100, 128, normalize=False),
            *Normalized_fullconnect_LeakyReLU(128, 256),
            *Normalized_fullconnect_LeakyReLU(256, 512),
            *Normalized_fullconnect_LeakyReLU(512, 1024),
            nn.Linear(1024, 28*28),
            nn.Tanh()
        )

    # forward中的z是在后面的定义的高斯噪声信号，形状为batch_sz*100
    def forward(self, x):
        img = self.model(x)
        img = img.view(img.size(0), 1, 28, 28)
        return img


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

        self.model = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )

    def forward(self, img):
        imgvectors = img.view(img.size(0), -1)
        discrimination = self.model(imgvectors)
```

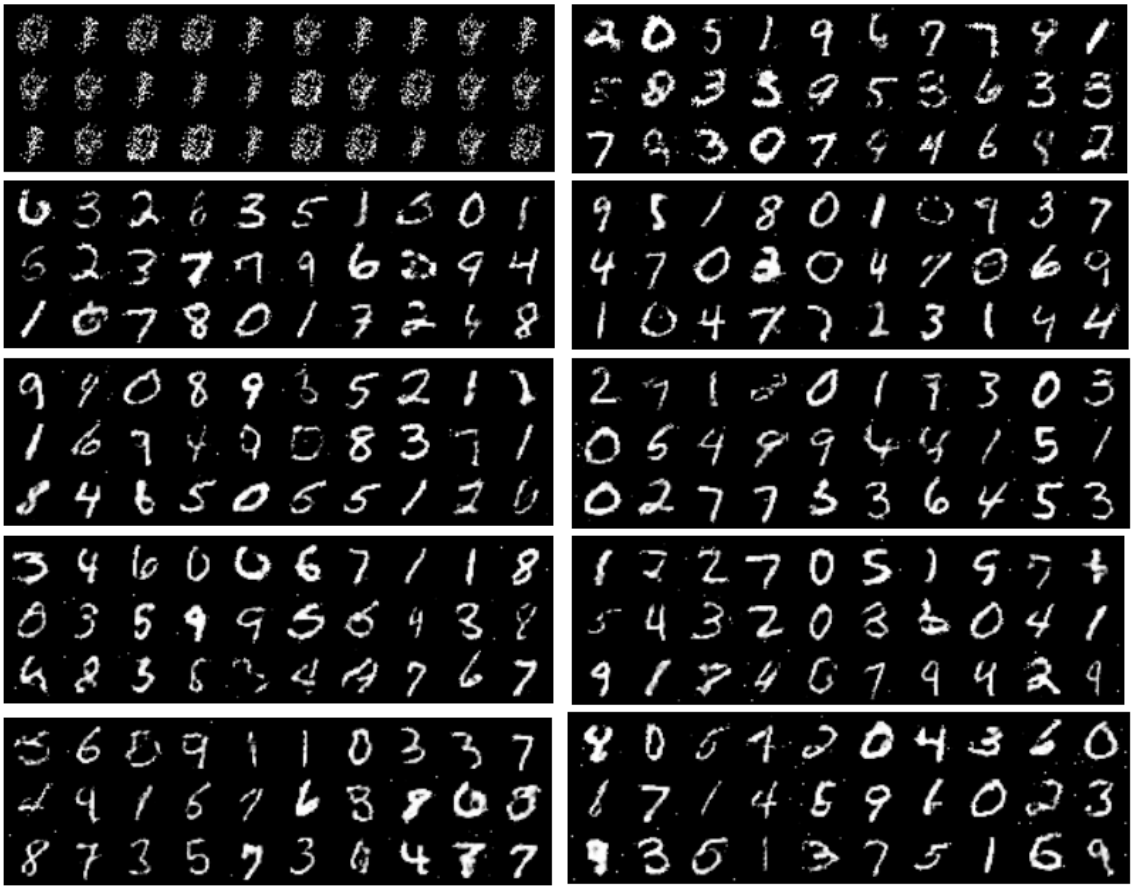
```
        return discrimination


# 使用BCE来替代minimax策略的vanilla loss
Lossfunc = torch.nn.BCELoss()


generator = Generator()
discriminator = Discriminator()
if cuda:
    generator.cuda()
    discriminator.cuda()
    Lossfunc.cuda()



gene_gd = torch.optim.Adam(generator.parameters(), lr=learning_rate, betas=(0.5, 0.999))            # betas=(0.5, 0.999) 效果很好
dis_gd = torch.optim.Adam(discriminator.parameters(), lr=learning_rate, betas=(0.5, 0.999))


g_loss = 0
d_loss = 0
for epoch in range(Epoch):
    for i, (imgs, labels) in enumerate(dataloader):
        # 真实图片的标签全部为1
        True_result = Variable(Tensor(imgs.size(0), 1).fill_(1.0), requires_grad=False)
        # 生成图片的标签全部为0
        False_result = Variable(Tensor(imgs.size(0), 1).fill_(0.0), requires_grad=False)
        original_imgs = Variable(imgs.type(Tensor))

        # 复位梯度值
        gene_gd.zero_grad()
        dis_gd.zero_grad()

        # Sample noise as generator input
        # 生成与真实图片相同batch的输入向量
        # 输入从0到1之间，形状为(imgs.shape[0], 100)的随机高斯数据。
        x = Variable(Tensor(np.random.normal(0, 1, (imgs.shape[0], 100))))
        generated_imgs = generator(x)
        g_loss = Lossfunc(discriminator(generated_imgs), True_result)
        g_loss.backward()
        gene_gd.step()  # 生成器参数更新


        # Loss measures generator's ability to fool the discriminator
        # 目标是：使得当前G网络生成的图片，骗过当前的判决网络D, 判决为真实图片
        # 每次迭代：G网络进化一点点，即使得g_loss降低一点点。
        # 由于每次迭代, D网络也在进化，导致g_loss再提升一点点
        # g_loss反应的是：使用当前的D网络，判断G网络生成的图片，是不是真实图片中的一个
        # g_loss越小，生成的图片越接近真实的图片中的一个

        # 判别器的loss = 判别为真的loss + 判别为假的loss
        d_loss = (Lossfunc(discriminator(original_imgs), True_result) + Lossfunc(discriminator(generated_imgs.detach()), False_result)) / 2
        d_loss.backward()
        dis_gd.step()   # 判别器参数更新

        # print("epoch %d - batch %d " % (epoch, i))


        if i % save_interval == 0:
            save_image(generated_imgs.data[:30], "gan_images/%d-%d.png" % (epoch, i), nrow=10)

    print('====> Epoch: {} generator loss: {:.4f}'.format(epoch, g_loss))
    print('====> Epoch: {} disriminator loss: {:.4f}'.format(epoch, d_loss))
```

```
====> Epoch: 0 generator loss: 0.3530
====> Epoch: 0 disriminator loss: 0.7582
====> Epoch: 1 generator loss: 0.2519
====> Epoch: 1 disriminator loss: 0.8231
====> Epoch: 2 generator loss: 0.2955
====> Epoch: 2 disriminator loss: 0.7824
====> Epoch: 3 generator loss: 0.3153
====> Epoch: 3 disriminator loss: 0.7569
====> Epoch: 4 generator loss: 0.3074
====> Epoch: 4 disriminator loss: 0.7794
====> Epoch: 5 generator loss: 0.2480
====> Epoch: 5 disriminator loss: 0.8442
====> Epoch: 6 generator loss: 0.3233
====> Epoch: 6 disriminator loss: 0.7566
====> Epoch: 7 generator loss: 0.3572
====> Epoch: 7 disriminator loss: 0.7355
====> Epoch: 8 generator loss: 0.2687
====> Epoch: 8 disriminator loss: 0.8341
====> Epoch: 9 generator loss: 0.3559
```

实验结果

可以看到，相对于VAE所得到的较为模糊的结果，GAN所得到的图片看起来更加清晰。这是因为VAE的设计具有强预设性，在其优化过程中会强制性地把数据拟合到有限维度的混合高斯或者其他分布上，会导致信息损失以及将不符合某种分布的数据强制性拟合到该分布上。而GAN在训练的过程中则没有这样的预设，它可以让生成器产生数据的分布直接拟合训练数据的分布。因此，GAN得到的结果会更加清晰。

下面依次选取epoch值为1,21,41,61,81,101,121,141,161,181时的结果进行分析：



可以看到选取epoch=1时的结果中只看得到数字的大致轮廓，而随着epoch的值增大，由GAN网络所生成的图片也更加清晰。

β-VAE

实现思路

β-vae对于传统vae的基础上对于loss部分进行改进，在loss的第二项KL散度项加上一个超参数β。

传统vae的损失函数：

$$Loss = \frac{1}{n}\sum_{i=1}^{n}[logp(x_i) + \frac{1}{2}\sum_{i=1}^{d}(-log\sigma^2) + \sigma^2 + \mu^2 - 1]$$

改进传统vae损失函数的第二项，得到β-VAE的损失函数：

$$Loss = \frac{1}{n}\sum_{i=1}^{n}[logp(x_i) + \frac{1}{2}\beta\sum_{i=1}^{d}(-log\sigma^2) + \sigma^2 + \mu^2 - 1]$$

β-VAE可以增强VAE模型的解纠缠的能力，在VAE中所追求的目标是最大化生成真实数据的概率值和最小化真实和估计后验分布的KL散度，而β-VAE则优化了其损失函数，加入了超参数β。当β=1时，即为标准的VAE，通过增大β的值，则可以使得前变量空间z表示信息的丰富度降低，但同时模型的解纠缠能力增加，可以用于平衡表示能力与解纠缠能力。(其中解纠缠是指将原始数据空间中纠缠着的数据变化，变换到一个好的表征空间中，在这个空间中，不同要素的变化是可以彼此分离的。)

代码实现

In [3]:

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.autograd as autograd
import torch.nn.functional as F
from torchvision import datasets, transforms, models
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
from sklearn.metrics import f1_score
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

rng = np.random.RandomState(1234)
random_state = 42
device = "cuda" if torch.cuda.is_available() else "cpu"

batch_size = 64
n_epochs = 15
beta = 2
z_dim = 10
assert z_dim >= 2

#转化为tensor
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Lambda(lambda x: x.view(-1))])

# 下载数据集
dataloader_train = torch.utils.data.DataLoader(
    datasets.MNIST('./data/MNIST', train=True, download=True, transform=transform),
    batch_size=batch_size,
    shuffle=False
)

dataloader_valid = torch.utils.data.DataLoader(
    datasets.MNIST('./data/MNIST', train=False, download=True, transform=transform),
    batch_size=batch_size,
    shuffle=False
)

def torch_log(x):
    return torch.log(torch.clamp(x, min=1e-10))

# Model definition of BetaVAE
class BetaVAE(nn.Module):
    def __init__(self, z_dim=10, nc=1):
        super(BetaVAE, self).__init__()
        self.z_dim = z_dim
        self.nc = nc
        # Encoder
        self.enc_conv1 = nn.Conv2d(nc, 8, 3, 2, 1)              # B, 8, 14, 14
        self.enc_conv2 = nn.Conv2d(8, 16, 3, 2, 1)              # B, 16, 7, 7
        self.enc_conv3 = nn.Conv2d(16, 32, 3, 2)                # B, 32, 3, 3
        self.enc_fc1 = nn.Linear(32*3*3, 128)                   # B, 128
        self.enc_mean = nn.Linear(128, z_dim)                   # B, z_dim
        self.enc_std = nn.Linear(128, z_dim)                    # B, z_dim
        # Decoder
        self.dec_fc1 = nn.Linear(z_dim, 128)                    # B, 128
        self.dec_fc2 = nn.Linear(128, 32*3*3)                   # B, 32*3*3
        self.dec_convt1 = nn.ConvTranspose2d(32, 16, 3, 2)      # B, 16, 7, 7
        self.dec_convt2 = nn.ConvTranspose2d(16, 8, 3, 2, 1, 1) # B, 8, 14, 14
        self.dec_convt3 = nn.ConvTranspose2d(8, nc, 3, 2, 1, 1) # B, 1, 28, 28
        # BN
        self.bn = nn.BatchNorm2d(16)


    def _encoder(self, x):
        x = F.relu(self.enc_conv1(x.view(-1, self.nc, 28, 28)))
        x = F.relu(self.enc_conv2(x))
        x = self.bn(x)
        x = F.relu(self.enc_conv3(x))
        x = F.relu(self.enc_fc1(x.view(-1, 32*3*3)))
        mean = self.enc_mean(x)
        std = F.softplus(self.enc_std(x))
        return mean, std

    def _sample_z(self, mean, std):
        # 重参数化生成隐变量
        epsilon = torch.randn(mean.shape).to(device)
        return mean + std * epsilon

    def _decoder(self, z):
        x = F.relu(self.dec_fc1(z))
```

```
        x = F.relu(self.dec_fc2(x))
        x = F.relu(self.dec_convt1(x.view(-1, 32, 3, 3)))
        x = self.bn(x)
        x = F.relu(self.dec_convt2(x))
        x = self.dec_convt3(x)
        # 使用sigmoid函数使得输出的结果在[0,1]范围内
        x = torch.sigmoid(x.view(-1, self.nc*28*28))
        return x

    # 前向传播过程，先编码后解码
    def forward(self, x):
        mean, std = self._encoder(x)
        z = self._sample_z(mean, std)
        x = self._decoder(z)
        return x, z

    def loss(self, x):
        mean, std = self._encoder(x)
        KL = -0.5 * torch.mean(torch.sum(1 + torch_log(std**2) - mean**2 - std**2, dim=1))

        z = self._sample_z(mean, std)
        y = self._decoder(z)

        reconstruction = torch.mean(torch.sum(x * torch_log(y) + (1 - x) * torch_log(1 - y), dim=1))

        return KL, -reconstruction

model = BetaVAE(z_dim).to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)

for epoch in range(n_epochs):
    losses = []
    KL_losses = []
    reconstruction_losses = []

    model.train()
    for x, _ in dataloader_train:

        x = x.to(device)

        model.zero_grad()

        # KL_loss, reconstruction_loss, ELBO
        KL_loss, reconstruction_loss = model.loss(x)
        loss = reconstruction_loss + beta*KL_loss

        loss.backward()
        optimizer.step()

        losses.append(loss.cpu().detach().numpy())
        KL_losses.append(KL_loss.cpu().detach().numpy())
        reconstruction_losses.append(reconstruction_loss.cpu().detach().numpy())

    losses_val = []
    model.eval()
    for x, t in dataloader_valid:

        x = x.to(device)

        KL_loss, reconstruction_loss = model.loss(x)

        loss = KL_loss + reconstruction_loss

        losses_val.append(loss.cpu().detach().numpy())

    print('EPOCH: %d    Train Lower Bound: %lf (KL_loss: %lf, reconstruction_loss: %lf)    Valid Lower Bound: %lf' %
          (epoch+1, np.average(losses), np.average(KL_losses), np.average(reconstruction_losses), np.average(losses_val)))
```

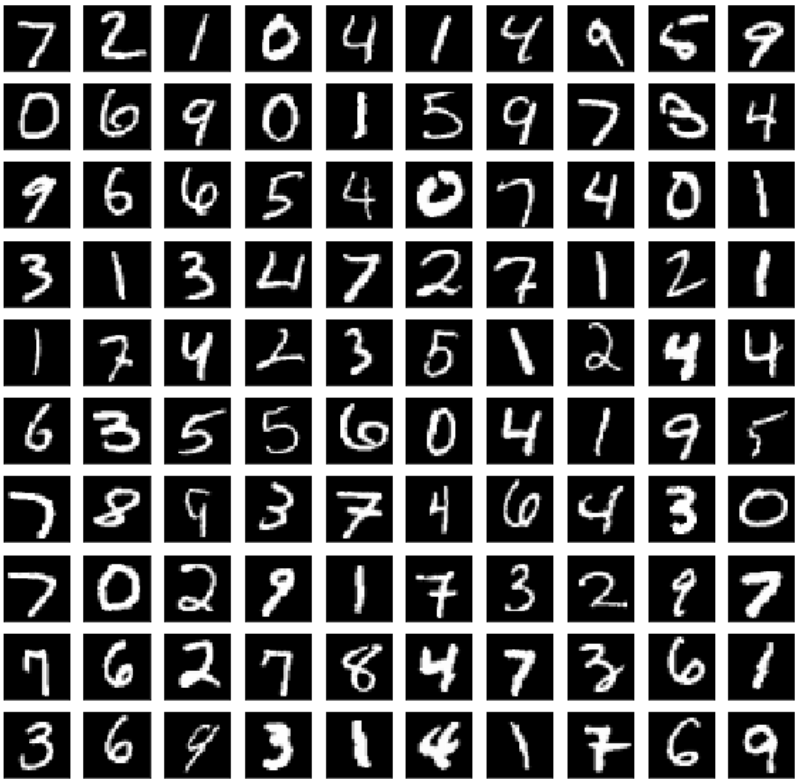
```
EPOCH: 1    Train Lower Bound: 198.513962 (KL_loss: 5.485088. reconstruction_loss: 187.543777)    Valid Lower Bound: 161.35038
8
EPOCH: 2    Train Lower Bound: 144.644485 (KL_loss: 9.560158. reconstruction_loss: 125.524178)    Valid Lower Bound: 167.36265
6
EPOCH: 3    Train Lower Bound: 136.523468 (KL_loss: 10.946808. reconstruction_loss: 114.629868)    Valid Lower Bound: 158.3303
68
EPOCH: 4    Train Lower Bound: 132.751038 (KL_loss: 11.554196. reconstruction_loss: 109.642639)    Valid Lower Bound: 164.3897
09
EPOCH: 5    Train Lower Bound: 130.400589 (KL_loss: 11.883916. reconstruction_loss: 106.632759)    Valid Lower Bound: 176.6950
99
EPOCH: 6    Train Lower Bound: 128.851624 (KL_loss: 12.192620. reconstruction_loss: 104.466400)    Valid Lower Bound: 172.7172
85
EPOCH: 7    Train Lower Bound: 127.621819 (KL_loss: 12.455451. reconstruction_loss: 102.710922)    Valid Lower Bound: 171.4639
74
EPOCH: 8    Train Lower Bound: 126.824028 (KL_loss: 12.657068. reconstruction_loss: 101.509888)    Valid Lower Bound: 183.4630
58
EPOCH: 9    Train Lower Bound: 126.103531 (KL_loss: 12.750308. reconstruction_loss: 100.602913)    Valid Lower Bound: 161.0180
82
EPOCH: 10    Train Lower Bound: 125.541367 (KL_loss: 12.835146. reconstruction_loss: 99.871086)    Valid Lower Bound: 166.5026
40
EPOCH: 11    Train Lower Bound: 125.036758 (KL_loss: 12.911517. reconstruction_loss: 99.213722)    Valid Lower Bound: 160.1084
75
EPOCH: 12    Train Lower Bound: 124.645683 (KL_loss: 12.982285. reconstruction_loss: 98.681107)    Valid Lower Bound: 165.6098
94
EPOCH: 13    Train Lower Bound: 124.282249 (KL_loss: 13.036795. reconstruction_loss: 98.208656)    Valid Lower Bound: 158.3542
94
EPOCH: 14    Train Lower Bound: 123.894539 (KL_loss: 13.076273. reconstruction_loss: 97.741989)    Valid Lower Bound: 175.7341
92
EPOCH: 15    Train Lower Bound: 123.645050 (KL_loss: 13.099656. reconstruction_loss: 97.445732)    Valid Lower Bound: 161.8967
13
```

实验结果

原始图像

In [4]:

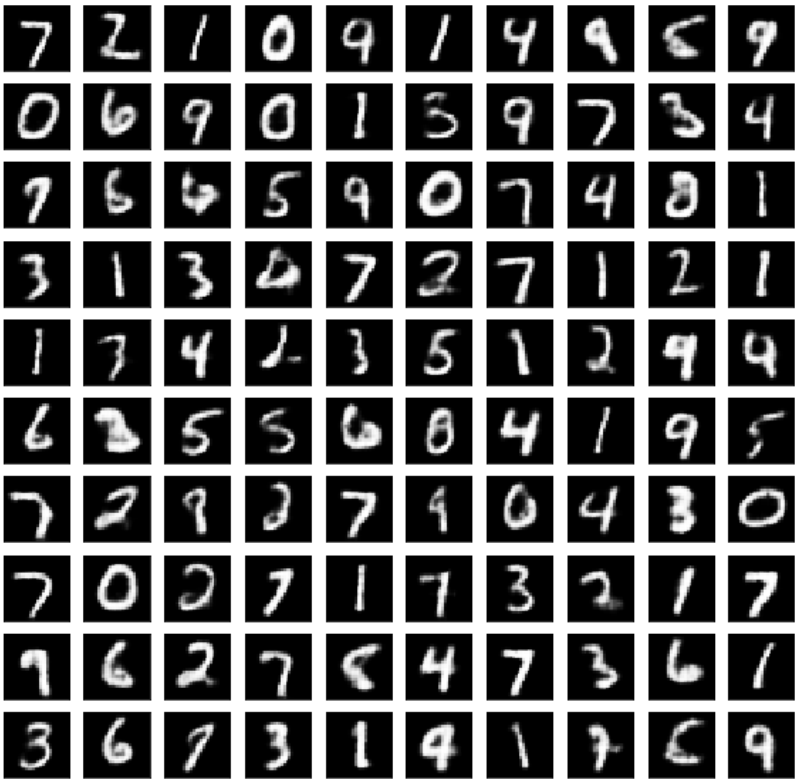
```
valid_dataset = datasets.MNIST('./data/MNIST', train=False, download=True, transform=transform)
fig = plt.figure(figsize=(10, 10))
model.eval()
for i in range(100):
    x, t = valid_dataset[i]
    im = x.view(-1, 28, 28).permute(1, 2, 0).squeeze().numpy()
    ax = fig.add_subplot(10, 10, i+1, xticks=[], yticks=[])
    ax.imshow(im, 'gray')
```



重建的图像

In [5]:

```
valid_dataset = datasets.MNIST('./data/MNIST', train=False, download=True, transform=transform)
fig = plt.figure(figsize=(10, 10))
model.eval()
for i in range(100):
    x, t = valid_dataset[i]
    x = x.to(device)
    x = x.unsqueeze(0)
    y, z = model(x)
    im = y.view(-1, 28, 28).permute(1, 2, 0).cpu().squeeze().detach().numpy()
    ax = fig.add_subplot(10, 10, i+1, xticks=[], yticks=[])
    ax.imshow(im, 'gray')
```

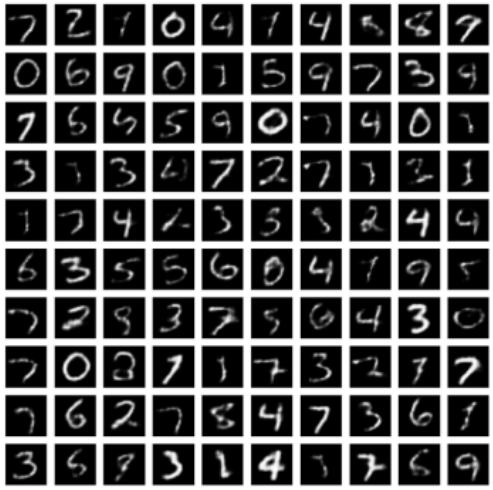


结果分析

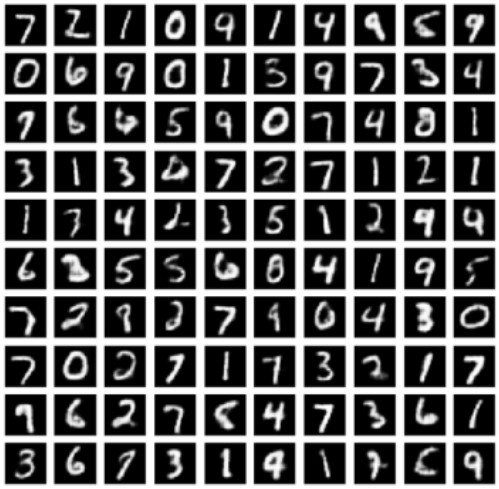
通过观察上面得到的原始图像与生成图像，可以看到，除了个别数字不够清晰之外，生成的图像与原始图像基本保持一致。

对于β-VAE模型，当β=1对应于原始的VAE框架。当β > 1时，模型被推动以学习数据更有效的潜在表示，如果数据至少包含一些独立的潜在变化因素，则可以将其解开。因此相对于VAE，β-VAE可以提高潜在表示中的解缠度。选取不同的β值得到如下结果：

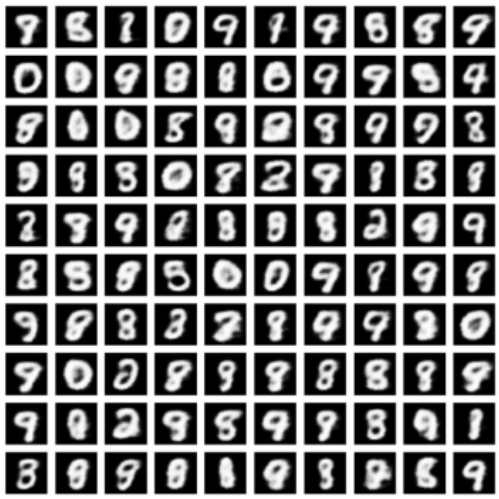
β=1：



β=2：



β=3：



β=4：

β=1时为VAE对应的结果，β=2，3，4为选取不同的β值时β-VAE模型所得到的结果。可以看到选取β=2时，所得到的图片更清晰准确，相对于VAE可以更好的学习到图片的特征。但随着β的值的增大，使得前变量空间z表示信息的丰富度降低，模型的表示能力下降。

修改数据集：加入噪点

对数据集输入的图片进行加入噪点的预处理，通过改变加入噪点的数量，可以探究VAE是否可以使用具有一定噪声的图片进行手写数字的生成

In [2]:

```
import random

def sp_noise(noise_img, proportion):
    '''
    添加椒盐噪声
    proportion的值表示加入噪声的量，可根据需要自行调整
    return: img_noise
    '''
    height, width = noise_img.shape[0], noise_img.shape[1]#获取高度宽度像素值
    num = int(height * width * proportion) #一个准备加入多少噪声小点
    for i in range(num):
        w = random.randint(0, width - 1)
        h = random.randint(0, height - 1)
        if random.randint(0, 1) == 0:
            noise_img[h, w] = 0
        else:
            noise_img[h, w] = 255
    return noise_img
```

In [4]:

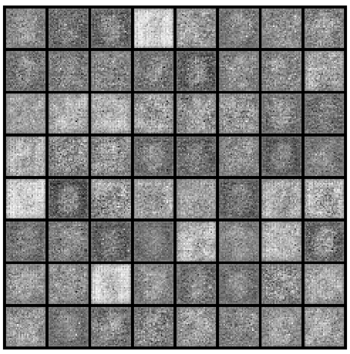
```
import torch
import torchvision
import torch.utils.data
import torch.nn
import matplotlib.pyplot as plt
import numpy as np

transform=torchvision.transforms.Compose([
    torchvision.transforms.ToTensor()   #转化为tensor
])

# 下载数据集
train_set=torchvision.datasets.MNIST(
    root='./minst_data/',
    train=False,
    transform=transform,
    download=True
)

train_set_loader=torch.utils.data.DataLoader(
    dataset=train_set,
    shuffle=True,
    batch_size=64   #试试128
)

batch_sz = 64   # 数据集切分，每个batch大小为64/128

# torch.Size([64, 1, 28, 28]), 表示每个batch有64张图，只有一个channel(黑白)，长宽分别是28*28

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

        # 三层卷积网络
        self.encoder =torch.nn.Sequential(
        torch.nn.Conv2d(1, 64, kernel_size=4, stride=2, padding=1),     #14*14
        torch.nn.BatchNorm2d(64),
        torch.nn.LeakyReLU(0.2, inplace=True),  # nn.Sigmoid()

        torch.nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),   #7*7
        torch.nn.BatchNorm2d(128),
        torch.nn.LeakyReLU(0.2, inplace=True),

        torch.nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),  #7*7
        torch.nn.BatchNorm2d(128),
        torch.nn.LeakyReLU(0.2, inplace=True),
        )

        # Linear(): 对输入数据进行线性变换
        self.get_mu=torch.nn.Sequential(
            torch.nn.Linear(128 * 7 * 7, 32)    # 128 * 7 * 7 -> 32
        )
        self.get_logvar = torch.nn.Sequential(
            torch.nn.Linear(128 * 7 * 7, 32)    # 128 * 7 * 7 -> 32
        )
        self.get_temp = torch.nn.Sequential(
            torch.nn.Linear(32, 128 * 7 * 7)    # 32 -> 128 * 7 * 7
        )

        # conv_transpose2d: 在由几个输入平面组成的输入图像上应用二维转置卷积，有时也称为“去卷积”。
        self.decoder = torch.nn.Sequential(
        torch.nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1),
        torch.nn.ReLU(inplace=True),

        torch.nn.ConvTranspose2d(64, 1, kernel_size=4, stride=2, padding=1),
        torch.nn.Sigmoid()
        )

    # 重参数化生成隐变量
    def get_z(self,mu,logvar):
        eps=torch.randn(mu.size(0),mu.size(1))  # 64,32 参数采样
        eps=torch.autograd.Variable(eps).cuda()
        z=mu+eps*torch.exp(logvar/2)    # z = mu + epsilon * std
        return z

    # 整个前向传播过程：编码 --> 解码
    def forward(self, x):
        out1=self.encoder(x)
        mu=self.get_mu(out1.view(out1.size(0),-1))              # 64, 128 * 7 * 7 -> 64, 32
        out2=self.encoder(x)
        logvar=self.get_logvar(out2.view(out2.size(0),-1))      # 64, 128 * 7 * 7 -> 64, 32

        z=self.get_z(mu,logvar)
        out3=self.get_temp(z).view(z.size(0),128,7,7)           # 64, 32 -> 64, 128 * 7 * 7
```

```python
        return self.decoder(out3),mu,logvar

def loss_fun(new_x,old_x,mu,logvar):
    # MSE=torch.nn.MSELoss(new_x,old_x,size_average=False)
    # 用L1/L2替换 效果比较好; L1会更好
    BCE=torch.nn.functional.binary_cross_entropy(new_x,old_x,size_average=False)
    KLD=-0.5*torch.sum(1+logvar-mu.pow(2)-logvar.exp())
    # beta-VAE: 修改BCE和KLD之间的权重
    return BCE+KLD

vae=VAE().cuda()

optimizer=torch.optim.Adam(vae.parameters(),lr=0.001,betas=(0.9,0.999),eps=1e-08,weight_decay=0)

epoch_n=300
total_loss = 0
for epoch in range(epoch_n):
    for i,(data,_) in enumerate(train_set_loader):
        old_img=torch.autograd.Variable(data).cuda()
        if(epoch==0):
            print(sp_noise(old_img[0:1],0.2))
            for k in range(batch_sz-1):
                old_img[k:k+1] = sp_noise(old_img[k:k+1],0.1)
        new_img,mu,logvar=vae.forward(old_img)
        loss=loss_fun(new_img,old_img,mu,logvar)

        total_loss+=loss.item()

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()    # 进行单次优化（参数更新）

        if i % 100 == 0:
            sample = torch.autograd.Variable(torch.randn(64, 32)).cuda()
            sample = vae.decoder(vae.get_temp(sample).view(64, 128, 7, 7)).cpu()
            torchvision.utils.save_image(sample.data.view(64, 1, 28, 28), './result_vae_minst/sample_' + str(epoch) + '.png')

            print('Train Epoch:{} -- [{}/{} ({:.0f}%)] -- Loss:{:.6f}'.format(
            epoch, i * len(data), len(train_set_loader.dataset),
            100.* i / len(train_set_loader), loss.data.item() / len(data)))

    print('====> Epoch: {} Average loss: {:.4f}'.format(epoch, total_loss / len(train_set_loader.dataset)))
```

```
tensor([[[[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.1608, 0.4000,
           0.4000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
```

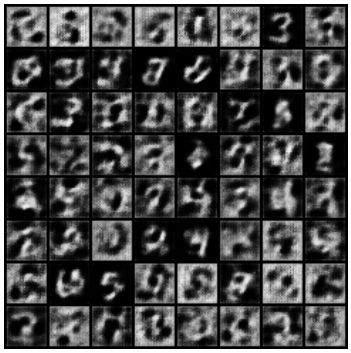
加入噪点效果

如图是对整张图片加入大量随机的黑白噪点的效果。



实验结果

当预处理加入噪声量为图片点数的1%时，得到的实验结果如下图所示：



epoch = 1

epoch = 300

可以看到，生成的手写数字比较清晰，与正常使用VAE生成得到的手写数字结果的完整度没有太大差别

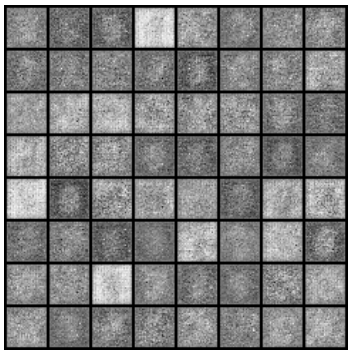
当预处理加入噪声量为图片点数的10%时，得到的实验结果如下图所示：

可以看到，生成的手写数字比较清晰，与正常使用VAE生成得到的手写数字结果的完整度比正常的情况低一些，其中有2个字符出现了字符比较淡的情况。

当预处理加入噪声量为图片点数的20%时，得到的实验结果如下图所示：

可以看到，生成的手写数字不太清晰，与正常使用VAE生成得到的手写数字结果的完整度比加入噪声量为图片点数的10%时更低一些，有一小部分字符出现了比较淡的情况。

当预处理加入噪声量为图片点数的80%时，得到的实验结果如下图所示：



epoch = 1

epoch = 300

可以看到，生成的手写数字比前几种情况更加模糊，与正常使用VAE生成得到的手写数字结果的完整度最低，生成的部分字符难以辨认，部分字符很淡。

通过上述实验结果，可以使用加入噪声点数在10%以下的图片进行手写数字的生成，超过10%以后对手写数字结果的影响较大。同时，VAE也可以用来将具有一定噪声的图片生成更加清晰的图片。