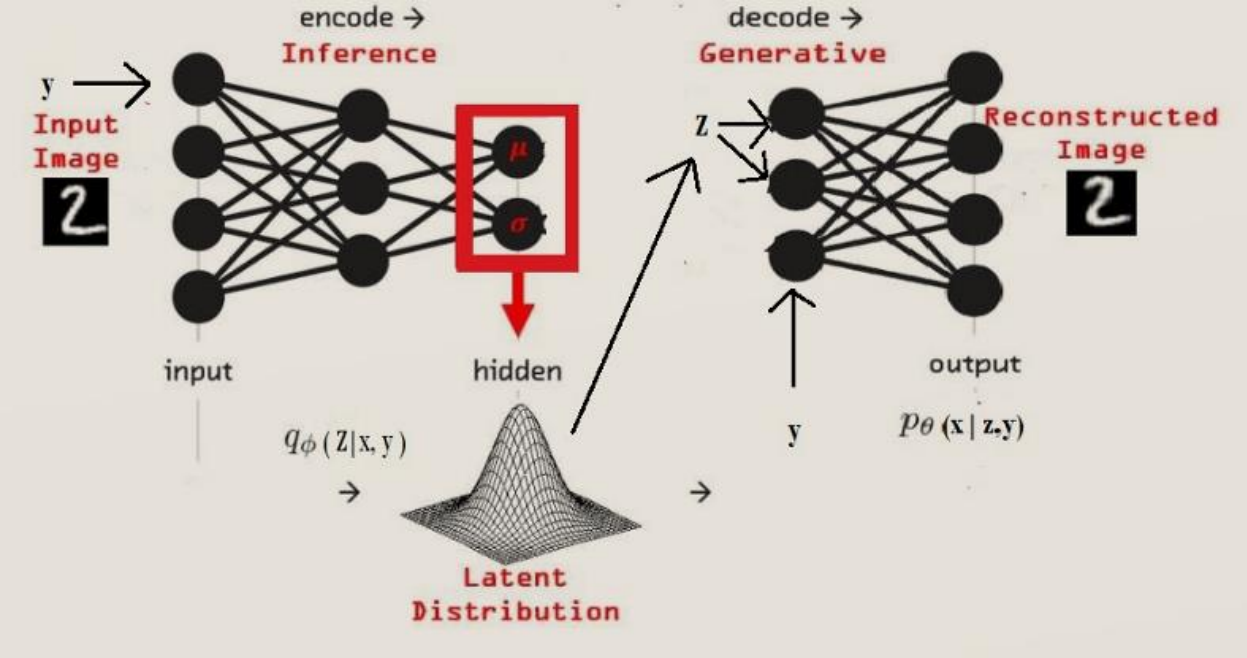
دانشگاه صنعتی شریف
دانشکده‌ی مهندسی کامپیوتر

تمرین سری پنجم
گزارش کار CVAE

امیرحسین محمدی
۹۹۲۰۱۰۸۱

بهمن ۱۴۰۰

معماری CVAE:



❖ کتابخانه های مورد استفاده:

```
❖ import torch
❖ import torch.nn as nn
❖ import torch.nn.functional as F
❖ import torch.optim as optim
❖ from torchvision import datasets, transforms
❖ from torch.autograd import Variable
❖ from torchvision.utils import save_image
```

دانلود مجموعه دادگان mnist و استفاده از dataloader برای مجموعه داده های train و test :

```
bs = 100
train_dataset = datasets.MNIST(root='./mnist_data/', train=True, transform
=transforms.ToTensor(), download=True)
test_dataset = datasets.MNIST(root='./mnist_data/', train=False, transform
=transforms.ToTensor(), download=False)

train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_si
ze=bs, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size
=bs, shuffle=False)
```

تعریف کلاس CVAE که در آن دو بخش encoder و decoder پیاده سازی شده و بسیار شبیه VAE است با این تفاوت که ابعاد c_dim نیز به همراه ورودی به بخش های encoder و decoder داده شده است تا بتواند خروجی را بر اساس condition مسئله تولید کند همچنین تابع sampling برای تولید نمونه ی latent طراحی شده است، لایه ی مورد استفاده برای بخش های encoder و decoder از لایه های linear استفاده شده است که بعد از هرکدام از یک تابع فعال ساز Relu استفاده شده است همچنین به عنوان آخرین لایه ی بخش decoder از یک لایه ی Sigmoid استفاده شده است و خروجی آن میانگین و واریانس و خروجی decoder بر اسا condition مسئله ماست:

```
class CVAE(nn.Module):
    def __init__(self, x_dim, h_dim1, h_dim2, z_dim, c_dim):
        super(CVAE, self).__init__()

        # encoder part
        self.fc1 = nn.Linear(x_dim + c_dim, h_dim1)
        self.fc2 = nn.Linear(h_dim1, h_dim2)
        self.fc31 = nn.Linear(h_dim2, z_dim)
        self.fc32 = nn.Linear(h_dim2, z_dim)
        # decoder part
        self.fc4 = nn.Linear(z_dim + c_dim, h_dim2)
        self.fc5 = nn.Linear(h_dim2, h_dim1)
        self.fc6 = nn.Linear(h_dim1, x_dim)

    def encoder(self, x, c):
        concat_input = torch.cat([x, c], 1)
        h = F.relu(self.fc1(concat_input))
        h = F.relu(self.fc2(h))
        return self.fc31(h), self.fc32(h)

    def sampling(self, mu, log_var):
        std = torch.exp(0.5*log_var)
        eps = torch.randn_like(std)
        return eps.mul(std).add(mu)

    def decoder(self, z, c):
        concat_input = torch.cat([z, c], 1)
        h = F.relu(self.fc4(concat_input))
        h = F.relu(self.fc5(h))
        return F.sigmoid(self.fc6(h))

    def forward(self, x, c):
        mu, log_var = self.encoder(x.view(-1, 784), c)
        z = self.sampling(mu, log_var)
        return self.decoder(z, c), mu, log_var
```

❖ تعریف مدل و در صورت موجود بودن GPU قرار دادن مدل بر روی GPU:

```
cond_dim = train_loader.dataset.train_labels.unique().size(0)
cvae = CVAE(x_dim=784, h_dim1=512, h_dim2=256, z_dim=2, c_dim=cond_dim)
if torch.cuda.is_available():
    cvae.cuda()
```

❖ تعریف بهینه ساز آدام و تابع هزینه VAE که از دو بخش KDL و binary cross entropy که برای بخش reconstruction است:

```
optimizer = optim.Adam(cvae.parameters())
def loss_function(recon_x, x, mu, log_var):
    BCE = F.binary_cross_entropy(recon_x, x.view(-
1, 784), reduction='sum')
    KLD = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())
    return BCE + KLD
```

❖ تابع one-hot که لیبل ها را one-hot می کند:

```
def one_hot(labels, class_size):
    targets = torch.zeros(labels.size(0), class_size)
    for i, label in enumerate(labels):
        targets[i, label] = 1
    return Variable(targets)
```

❖ طراحی تابع Train که مدل تعریف شده را ابتدا در وضعیت train قرار می دهد و سپس به ازای هر batch ماینگین و واریانس و خروجی decoder را بر محاسبه می کند و سپس با استفاده از تابع loss تعریف شده و همچنین تابع بهنیه ساز اجرا شده و سپس عملیات backward:

```
def train(epoch):
    cvae.train()
    train_loss = 0
    for batch_idx, (data, cond) in enumerate(train_loader):
        data, cond = data.cuda(), one_hot(cond, cond_dim).cuda()
        optimizer.zero_grad()

        recon_batch, mu, log_var = cvae(data, cond)
        loss = loss_function(recon_batch, data, mu, log_var)

        loss.backward()
        train_loss += loss.item()
        optimizer.step()

        if batch_idx % 100 == 0:
```

```
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format
(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item() / len(da
ta)))
    print('====> Epoch: {} Average loss: {:.4f}'.format(epoch, train_loss
/ len(train_loader.dataset)))
```

❖ اجرای عملیات train:

```
for epoch in range(1, 5):
    train(epoch)
    test()
```

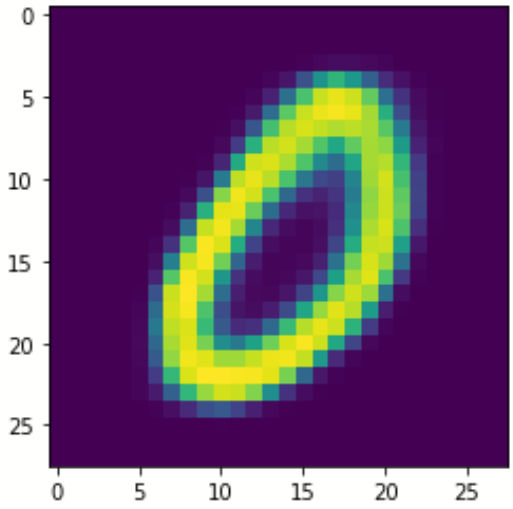
❖ تولید خروجی بر اساس condition:

```
with torch.no_grad():
    z = torch.randn(10, 2).cuda()
    c = torch.eye(10).cuda()

    sample = cvae.decoder(z, c)
```

❖ نمایش خروجی ها به شرط اینکه خروجی صفر باشد:

```
import matplotlib.pyplot as plt
plt.imshow(sample[0].view(28, 28).cpu().numpy())
```



مقایسه ی VAE و CVAE

از نظر بصری، می توان گفت که نتایج بازسازی CVAE بسیار بهتر از VAE اصلی است. در واقع اینوطر می توان استدلال کرد که چون هر داده تحت برچسب خاص، توزیع خاص خود را دارد، بنابراین نمونه‌برداری از داده‌ها با یک برچسب خاص آسان است. اگر به نتیجه VAE اصلی نگاه کنیم، بازسازی ها در لبه ها آسیب می بینند، به عنوان مثال. زمانی که مدل مطمئن نیست که 3، 8 یا 5 است، زیرا آنها بسیار شبیه به نظر می رسند. این در حالی است که در CVAE چنین مشکلی وجود ندارد همانطور که در شکل زیر می بینیم.