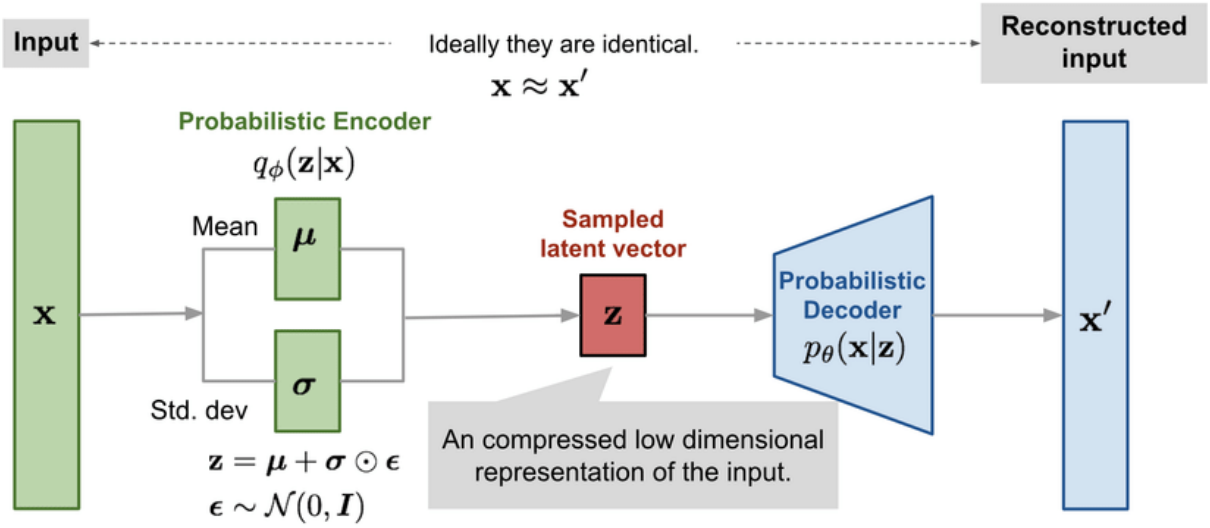
دانشگاه صنعتی شریف
دانشکده‌ی مهندسی کامپیوتر

تمرین سری پنجم
گزارش کار VAE

امیرحسین محمدی
۹۹۲۰۱۰۸۱

بهمن ۱۴۰۰

معماری VAE :



تابع هزینه ی VAE :

$$z = latent\ vector$$
$$x = input\ data$$

$$Encoder = q_\theta(z \mid x)$$
$$Decoder = p_\varphi(x \mid z)$$
$$p(z) = Normal(0, 1)$$

$$Loss = Reconstruction\ Loss + Regularizer$$
$$Loss = E_{q_\theta(z|x)}[log\ p_\varphi(x \mid z)] - KL(q_\theta(z|x) \,||\, p(z))$$

❖ کتابخانه های مورد نیاز:

```
import torch
import torch.nn as nn
import numpy as np
from tqdm import tqdm
from torchvision.utils import save_image, make_grid
```

❖ تنظیم پارامترهای اولیه یعنی ابعاد اولیه، لایه های مخفی و لایه ی latent و تنظیم Devise مورد نظر:

```
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
batch_size = 100
x_dim  = 784
hidden_dim = 400
latent_dim = 200
lr = 1e-3
epochs = 30
```

❖ دانلود دیتاست Mnist و انجام عملیات ترنسفورم بر روی داده ها:

```
from torchvision.datasets import MNIST
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
mnist_transform = transforms.Compose([
        transforms.ToTensor(),
])

train_dataset = MNIST("./mnist/", transform=mnist_transform, train=True, d
ownload=True)
test_dataset  = MNIST("./mnist/", transform=mnist_transform, train=False,
download=True)

train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, sh
uffle=True)
test_loader  = DataLoader(dataset=test_dataset,  batch_size=batch_size, sh
uffle=False)
```

تعریف بخش encoder که در معماری VAE تعریف شده است. این معماری از چهار لایه ی linear تشکیل شده است. دو تا از این چهار لایه برای محاسبه ی واریانس و میانگین توزیع به جود آمده در بخش encoder طراحی شده است. پس از آن از یک تابع فعال ساز Leaky relu استفاد شده است. همچنین از یک تابع forward استفاده شده است که عملیات پیاده سازی شد در بخش encoder را اجرا می کند:

```
class Encoder(nn.Module):

    def __init__(self, input_dim, hidden_dim, latent_dim):
        super(Encoder, self).__init__()

        self.FC_input = nn.Linear(input_dim, hidden_dim)
```

```
        self.FC_input2 = nn.Linear(hidden_dim, hidden_dim)
        self.FC_mean = nn.Linear(hidden_dim, latent_dim)
        self.FC_var = nn.Linear (hidden_dim, latent_dim)
        self.LeakyReLU = nn.LeakyReLU(0.2)
        self.training = True

    def forward(self, x):
        h_= self.LeakyReLU(self.FC_input(x))
        h_= self.LeakyReLU(self.FC_input2(h_))
        mean= self.FC_mean(h_)
        log_var= self.FC_var(h_)
        return mean, log_var
```

❖ تعریف بخش decoder که در معماری VAE تعریف شده است. این معماری از سه لایه ی linear تشکیل شده است. پس از آن از یک تایع فعال ساز Leaky relu استفاد شده است همچنین در انتها از یک تابع Sigmoid استفاده شده است تا روند محاسبه ی خطا و تولید خروجی را تسهیل کند. همچنین از یک تابع forward استفاده شده است که عملیات پیاده سازی شد در بخش encoder را اجرا می کند:

```
class Decoder(nn.Module):
    def __init__(self, latent_dim, hidden_dim, output_dim):
        super(Decoder, self).__init__()
        self.FC_hidden = nn.Linear(latent_dim, hidden_dim)
        self.FC_hidden2 = nn.Linear(hidden_dim, hidden_dim)
        self.FC_output = nn.Linear(hidden_dim, output_dim)
        self.LeakyReLU = nn.LeakyReLU(0.2)

    def forward(self, x):
        h= self.LeakyReLU(self.FC_hidden(x))
        h= self.LeakyReLU(self.FC_hidden2(h))
        x_hat = torch.sigmoid(self.FC_output(h))
        return x_hat
```

❖ نمونه برداری اپسیلون و پیاده سازی تکنیک reparameterization و ایجاد لایه ی Latent و تعریف بخش encoder و decoder در کلاس model:

```
class Model(nn.Module):
    def __init__(self, Encoder, Decoder):
        super(Model, self).__init__()
        self.Encoder = Encoder
        self.Decoder = Decoder

    def reparameterization(self, mean, var):
        epsilon = torch.randn_like(var).to(DEVICE)
        z = mean + var*epsilon
```

```
        return z


    def forward(self, x):
        mean, log_var = self.Encoder(x)
        z = self.reparameterization(mean, torch.exp(0.5 * log_var))
        x_hat= self.Decoder(z)
        return x_hat, mean, log_var
```

❖ تعریف بخش encoder وdecoder و تعریف مدل:

```
encoder = Encoder(input_dim=x_dim, hidden_dim=hidden_dim, latent_dim=laten
t_dim)
decoder = Decoder(latent_dim=latent_dim, hidden_dim = hidden_dim, output_d
im = x_dim)
model = Model(Encoder=encoder, Decoder=decoder).to(DEVICE)
```

❖ تعریف تابع هزینه Binary cross entropy و پیاده سازی تابع هزینه VAE که در بالا به آن اشاره شده و شامل دو بخش KLD و reconstruction است همچنین تعریف تابع بهینه ساز آدام:

```
from torch.optim import Adam

BCE_loss = nn.BCELoss()

def loss_function(x, x_hat, mean, log_var):
    reproduction_loss = nn.functional.binary_cross_entropy(x_hat, x, reduc
tion='sum')
    KLD = - 0.5 * torch.sum(1+ log_var - mean.pow(2) - log_var.exp())

    return reproduction_loss + KLD


optimizer = Adam(model.parameters(), lr=lr)
```

در این بخش گام آموز model پیاده سازی شده است، ابتدا یک batch از داده های دورن train_lader می خوانیم سپس به devise انتقال می دهیم، داده ها به مدل انتقال داده می شود و پس از عبود از بخش decoder و encoder دو مقدار میانگین و واریانس و داده های بازسازی شده را بر می گرداند. و سپس با اساس این سه مقدار، تابع هزینه ی VAE را محاسبه می کنیم که همانطور که گفته شد از دو بخش نشکیل شده است.

```
print("Start training VAE...")
model.train()

for epoch in range(epochs):
    overall_loss = 0
    for batch_idx, (x, _) in enumerate(train_loader):
```

```
        x = x.view(batch_size, x_dim)
        x = x.to(DEVICE)

        optimizer.zero_grad()

        x_hat, mean, log_var = model(x)
        loss = loss_function(x, x_hat, mean, log_var)

        overall_loss += loss.item()

        loss.backward()
        optimizer.step()

    print("\tEpoch", epoch + 1, "complete!", "\tAverage Loss: ", overall_l
oss / (batch_idx*batch_size))

print("Finish!!")
```

❖ پیاده سازی تابعی برای تولید تصاویر از ورودی های نویزی:

```
import matplotlib.pyplot as plt
def show_image(x, idx):
    x = x.view(batch_size, 28, 28)

    fig = plt.figure()
    plt.imshow(x[idx].cpu().numpy())
with torch.no_grad():
    noise = torch.randn(batch_size, latent_dim).to(DEVICE)
    generated_images = decoder(noise)
show_image(generated_images, idx=12)
```

نمونه ای از ورودی های تولید شده: