DL作业: 自编码器

201300086史浩男

一、训练部分

1、Loss函数

我采用L1范数作为重构损失

```
    def recon_loss(self, x, x_):
        # 实现重构损失函数 (5/100)
        # L1
        loss = nn.L1Loss()  # 必须导这么一手
        recon = loss(x, x_)
        #BCE
        #recon = F.binary_cross_entropy(F.sigmoid(x_),F.sigmoid(x),
reduction='sum')
        return recon
```

经测试，BCELoss收敛极慢，考虑到我电脑的性能，还是LI吧

KL散度的计算方法

```
    def kl_div(self, mu,logvar):
        # 实现kl散度的损失 (5/100)
        kl_mean = -0.5 * torch.mean(1 + logvar - mu.pow(2) - logvar.exp())
        return kl_mean
```

Loss函数的组合：

- AE只需要重构损失，很简单
- VAE和CVAE我采用`recon+kl*annealing`的方法。查阅文献并验证得知，KL散度在训练初期最好调小一点。因此我把`epoch`和`epoch_num`传给Loss，进行针对KL散度的`annealing`方法

```
    def forward(self, x, x_,mu,logvar,epoch,epoch_num, **otherinputs):
        # 实现loss的计算，注意处理 **otherinputs 的解析，以及不同的decode_type对应于不同的loss (10/100)
        recon=Loss.recon_loss(self,x,x_)
        if self.decode_type=="AE":
            total_loss=recon
        elif self.decode_type=="VAE" or self.decode_type=="CVAE":
            kl = Loss.kl_div(self,mu,logvar)
            annealing=epoch/epoch_num
            annealing*=0.003
            total_loss=recon+kl*annealing
        else:
            raise NotImplementedError
        return total_loss
```

2、main的修改

- 我修改了框架代码调用train的方式，改为每个epoch调用一次train和test，好处是方便debug以及可以每一轮都test一下，方便观察模型每个epoch后的效果如何

```
    for epoch in range(args.epoch_num):
        print("\n epoch:", epoch)
        train_epoch(model=auto_encoder, loss=loss, train_loader=train_loader,
                    optimizer=optimizer, epoch=epoch,
type=args.type,epoch_num=args.epoch_num)
        test_epoch(model=auto_encoder, test_loader=test_loader, loss=loss,
                   epoch=epoch, type=args.type, epoch_num=args.epoch_num)
```

- 我合并了`encoder_args`参数，不太理解框架代码中为什么要放两个

```
encoder_args = {
        "x_dim": args.x_dim,
        "hidden_size": args.hidden_size,
        "latent_size": args.latent_size,
        "decode_type": args.type,
        "is_dist": True if args.type in ["VAE", "CVAE"] else False,
    }
```

3、train_epoch

正常调用接口即可，注意正确使用

- optimizer.zero_grad()
- loss_num.backward()
- optimizer.step()

```
def train_epoch(model, train_loader, loss, optimizer,epoch, type,epoch_num):
    model.train()
    train_loss = 0
```

```
    for i, (x, y) in enumerate(train_loader):
        # 训练过程的补全 (20/100) 注意考虑不同类型的AE应该有所区别
        x = x.reshape(128, 1, -1)  # (128,1,784)28*28需要展平
        optimizer.zero_grad()
        if type=="AE":
            z, x_ = model(x)
            loss_num = loss(x, x_)
        elif  type=="VAE":
            x_, mu, log_var, z=model(x)
            loss_num=loss(x, x_,mu,log_var,epoch,epoch_num)
        elif  type=="CVAE":
            x_, mu, log_var, z=model(x,y)
            loss_num=loss(x, x_,mu,log_var,epoch,epoch_num)
        else:
            raise NotImplementedError
        train_loss += loss_num
        loss_num.backward()
        optimizer.step()
        ##########################省略loss的打印
    return
```

4、test_epoch

这部分我用来对不同type问题的保存图像测试

这里我最大的收获是熟练了`make_grid`和`save_image`方法，以后写DL离不开他们

保存示例如下：

```
# 保存一些重构出来的图像用于(写报告)进一步研究 (5/100)
# 保存训练结果部分
                        if not os.path.exists(f"results/{type}/epoch_{epoch}"):
                            os.makedirs(f"results/{type}/epoch_{epoch}")
                        save_x_ = x_.reshape(-1, 1, 28, 28)[:16]
                        save_x_ = make_grid(save_x_, 8, 0)
                        save_image(save_x_,
os.path.join(f"results/{type}/epoch_{epoch}/batch_{i}.png"))
```

二、Model部分

1、AE

AE的model实现很简单，没有新的参数，其实就是压缩一下再还原一下。这一过程主要是熟悉pytorch框架（遇到了很多因为不熟练pytorch产生的bug），熟悉encode和decode的过程，为后续探索进行铺垫。

AE的编码器将高维输入X编码成低维隐变量h，从而让神经网络学习最有信息量的特征

$$\mathbf{h}=g_{\theta_1}(\mathbf{x})=\sigma\left(\mathbf{W}_1\mathbf{x}+\mathbf{b}_1\right) \tag{1}$$

解码器就是将h还原到初始维度得到X^R使得$X^R\approx X$。

$$\hat{\mathbf{x}} = g_{\theta_2}(\mathbf{h}) = \sigma\left(\mathbf{W}_2\mathbf{h} + \mathbf{b}_2\right) \tag{2}$$

AE的forward过程如下：

```
    def forward(self, x):
        z = self.encoder(x)
        # 实现AE的forward过程(5/100)
        x_ = self.decoder(z)
        return z, x_
```

2、VAE

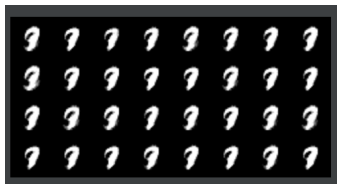
与AE相比，改为输出均值和方差

Encode过程：利用预留好的`self.sigma`层

Decode过程：我修改了神经网络：

```
        elif decode_type == "VAE":
            self.decoder = ...  # 实现VAE的decoder (5/100)
            self.decoder = nn.Sequential(nn.Linear(latent_size, 50), nn.ReLU(),
                                         nn.Linear(50, hidden_size), nn.ReLU(),
                                         nn.Linear(hidden_size, 400), nn.ReLU(),
                                         nn.Linear(400, x_dim)
                                         )
```

我尝试过使用sigmoid进行最后一层的激活，但效果很差：



均值、方差、标准差的计算处理：

```
    def forward(self, xs):
        # 实现VAE的forward过程(10/100)
        mu, log_var = self.encoder(xs)
        std = torch.exp(0.5 * log_var)
        eps = torch.randn_like(std)
        z = eps * std + mu
        x_ = self.decoder(z)
        return x_, mu,log_var, z
```

为了方便生成测试，我添加了inference方法

```
    def inference(self, z):
        x_ = self.decoder(z)
        return x_
```

3、CVAE

模型增加了标记的输入，让生成真正变的鲜活起来

为了处理标记，我使用了独热编码：

```
def idx2onehot(idx, n):
    assert torch.max(idx).item() < n
    if idx.dim() == 1:
        idx = idx.unsqueeze(1)
    onehot = torch.zeros(idx.size(0), n)
    onehot.scatter_(1, idx, 1)
    return onehot
```

完整Encode功能如下：

```
    def forward(self, xs, c=None):
        # 实现编码器的forward过程 (5/100), 注意 is_dist 的不同取值意味着我们需要不同输出的
encoder
        enc_outputs = self.mu(xs)
        if self.is_dist:
            if self.conditional:
                c = idx2onehot(c, n=10)
                xs= torch.cat((xs, c), dim=-1)
            mu = self.sigma(xs)#使用不同网络
            sigma = self.sigma(xs)
            return mu,sigma
        return enc_outputs  # (128,1,10)
```

完整Decode：

```
    def forward(self, zs,c=None, **otherinputs):
        # 实现decoder的decode部分, 注意对不同的decode_type的处理与对**otherinputs的解析
(10/100)
        if self.conditional:
            c = idx2onehot(c, n=10)
            zs = torch.cat((zs, c), dim=-1)
        dec_outputs = self.decoder(zs)
        return dec_outputs  # (128,1,784)
```

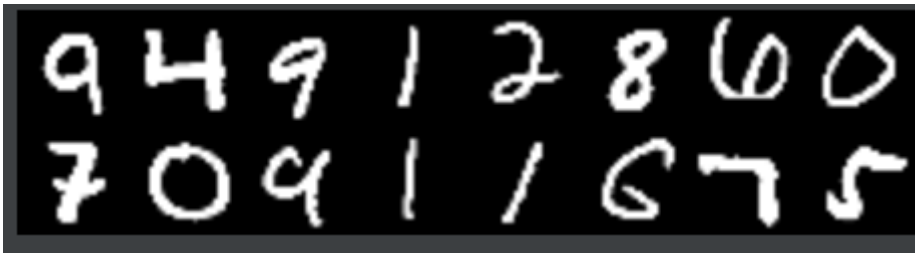
forward方法和inference方法几乎没有改动，只需要多穿一个标记进去就可以了

```
# 实现 CVAE的forward过程(15/100)
#代码略，几乎和VAE相同
```

三、测试效果

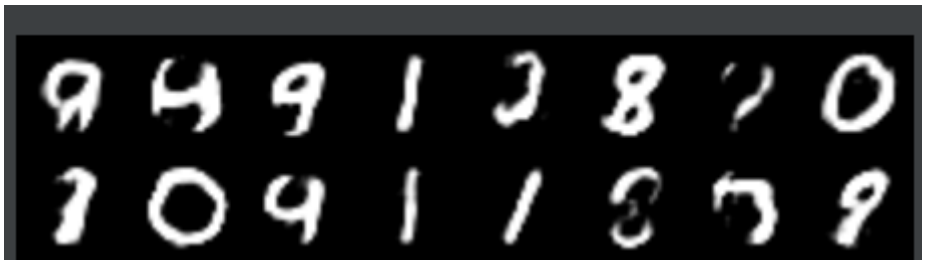
1、AE

原始数据：



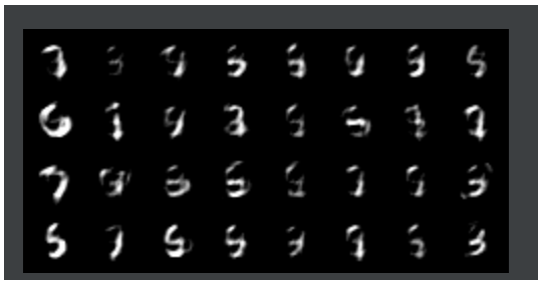
After 4 epoch：

After 16 epoch：



2、VAE

可能是神经网络设计的不好，可能是优化器没有选对，可能是Loss的组合没有选好，目前我的电脑在10个epoch内只能展现出如下结果



3、CVAE

VAE效果都不好，CVAE效果当然不会好，这里我就不放图丢人了...

四、Debug过程

1、batch不对齐

为了解决batch后不对齐问题，我在框架代码的`data.py`中进行如下修改：

增加：`drop_last=True`

```
return td.DataLoader(dataset, batch_size=batch_size,drop_last=True,)
```

2、矩阵不对齐

我遇到了各种各样矩阵不对齐的bug，比如：

- 从dataloader传过来的数据，先reshape一下再传递给网络

```
x = x.reshape(128, 1, -1)
```

- 保存图片时，`save_image`接口对数据格式有特殊要求

```
save_x = x.reshape(-1,1, 28, 28)[:16]
```

我选择输出每个batch前16个数字进行保存图片

3、loss下降但结果不变

这真是一个奇怪的bug，我眼睁睁看着loss逐渐收敛，但输出的图片却都完全一致

```
#optimizer = torch.optim.SGD(auto_encoder.parameters(), lr=0.01)
optimizer =torch.optim.Adam(auto_encoder.parameters(), lr=0.001)
```

更改优化器为Adam后问题解决

五、感悟

探索神经网络真是一件有趣的事

本次实验时间有限，没有尝试足够多的方法和参数组合

通过代码实践，我理解了AE，VAE，CVAE的原理和实现过程，了解了pytorch框架，了解了更多的Loss特点和优化器特点

希望老师以后多布置一些实践作业！