人工智能实验报告

16337183 孟衍璋

实验内容

设计并实现一个自编码机，对MNIST手写字符数据集进行分类。

实验环境

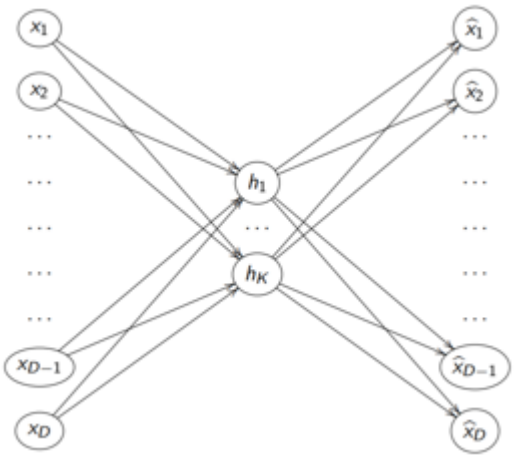
深度学习平台pytorch，在Windows平台上完成。

实验设计

本次实验需要实现自编码机对MNIST手写字符数据集进行分类，首先需要理解自编码机的工作原理。自编码机就是一种神经网络，需要学习一个函数：

$$h(x) \approx x$$

通过梯度下降法与反向传播训练这个神经网络，来重建原始的数据。其中的损失函数就是重建出来的数据与原始数据的差值。此神经网络的结构如下：



可以看出，自编码机的工作过程可以分为两个阶段：压缩与解压。先将原始D维数据压缩成为K维数据（K<D），其中包含了原始数据中关键的信息，然后再使用这K维数据恢复出原本的数据。通过不断训练，中间的如瓶颈一般的部分就是总结出来的原数据的精髓。

自编码机是一种非监督学习形式的神经网络，只需要训练数据，不需要训练标签。在MNIST数据集中，用先压缩再解压对应的图片，再根据压缩的特征来进行非监督分类。

实验过程

安装环境

首先要做的是在电脑上安装深度学习平台pytorch，pytorch是基于python的，所以可以直接用pip安装。在官网上查询到对应的版本之后，在控制台输入如下代码安装：

```
pip3 install http://download.pytorch.org/whl/cpu/torch-0.4.1-cp37-cp37m-win_amd64.whl
pip3 install torchvision
```

之后在python交互式界面输入如下代码验证环境是否已经搭建成功：

```
>>> from __future__ import print_function
>>> import torch
>>> x = torch.rand(5, 3)
>>> print(x)
```

如果输出显示类似如下的结果，便证明pytorch已经安装成功：

```
tensor([[0.9525, 0.9349, 0.4350],
        [0.1367, 0.0771, 0.2426],
        [0.3674, 0.1453, 0.3473],
        [0.4215, 0.5197, 0.7091],
        [0.4686, 0.8101, 0.9795]])
```

准备实验数据

本次实验用到的数据集是MNIST，可以从网站http://yann.lecun.com/exdb/mnist/上获取。下载解压之后放到与源代码相同的文件夹即可。还有一种方法便是在代码中设置其中一个参数*download_mnist*为*True*，在运行过程中便会自动下载MNIST数据集。

然后我们可以设置其中的一些超参数，设置如下：

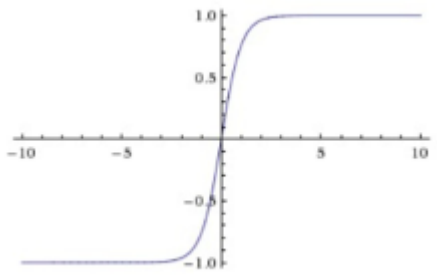
```
# 超参数
epochs = 10
batch_size = 64
batch_size_train = 64
batch_size_test = 1000
learning_rate = 0.005
donwload_mnist = False
n_test_img = 5
momentum = 0.5
log_interval = 100
```

之后便读入训练数据以供*AutoEncoder*学习：

```
train_data = torchvision.datasets.MNIST(
    root='./mnist/',
    train=True,
    transform=torchvision.transforms.ToTensor(),
    download=donwload_mnist,
)
train_load = Data.DataLoader(dataset=train_data,
batch_size=batch_size, shuffle=True)
```

构建自编码机

我们使用一系列全连接层，其中激活函数为*Tanh()*，它将实数值压缩到[-1,1]之间，图像如下：




```
# 构建自编码机
class AutoEncoder(nn.Module):
    def __init__(self):
        super(AutoEncoder, self).__init__()
```

```
        self.encoder = nn.Sequential(
            nn.Linear(28*28, 128),
            nn.Tanh(),
            nn.Linear(128, 64),
            nn.Tanh(),
            nn.Linear(64, 25),
        )
        self.decoder = nn.Sequential(
            nn.Linear(25, 64),
            nn.Tanh(),
            nn.Linear(64, 128),
            nn.Tanh(),
            nn.Linear(128, 28*28),
            nn.Sigmoid(),
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return encoded, decoded
```

此自编码机将数据从*25*25=784*维压缩到*5*5=25*维。

然后再初始化*AutoEncoder*、*optimizer*和*loss_function*:

```
autoencoder = AutoEncoder()
optimizer = torch.optim.Adam(autoencoder.parameters(),
lr=learning_rate)
loss_func = nn.MSELoss()
```

训练模型

针对每个数字图片，我们都迭代一次训练数据。在每次迭代中，我们需要使用*optimizer.zero_grad()*将梯度设置为0。然后在前向传播过程中，我们计算出自编码机的输出和损失函数。接下来我们使用*loss.backward()*将计算新的梯度，再用*optimizer.step()*反向传播回自编码机的参数中。

```
for epoch in range(epochs):
    for step, (x, b_label) in enumerate(train_loader):
```

```
        b_x = x.view(-1, 28*28)
        b_y = x.view(-1, 28*28)

        encoded, decoded = autoencoder(b_x)

        loss = loss_func(decoded, b_y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if step % 100 == 0:  # 每隔100个数据输出一次
            print('Epoch: ', epoch, '| train loss: %.4f' %
loss.data.numpy())

            _, decoded_data = autoencoder(view_data)
            for i in range(n_test_img):
                a[1][i].clear()
                a[1][i].imshow(np.reshape(decoded_data.data.numpy()
[i], (28, 28)), cmap='gray')
                a[1][i].set_xticks(()); a[1][i].set_yticks(())
            plt.draw(); plt.pause(0.05)
```

进行分类

先读取MNIST数据集：

```
# 读取mnist数据集
train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./mnist/', train=True,
download=donwload_mnist,

transform=torchvision.transforms.Compose([
                                torchvision.transforms.ToTensor(),
                                torchvision.transforms.Normalize(
                                  (0.1307,), (0.3081,))
                              ])),
    batch_size=batch_size_train, shuffle=True)
test_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./mnist/', train=False,
download=donwload_mnist,
```

```
transform=torchvision.transforms.Compose([
                                torchvision.transforms.ToTensor(),
                                torchvision.transforms.Normalize(
                                  (0.1307,), (0.3081,))
                               ])),
    batch_size=batch_size_test, shuffle=True)
```

接下来就需要利用AutoEncoder提取出来的信息进行分类，构建如下的网络：

```
# 构建网络
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 25, kernel_size = 2, padding = 2)
        self.conv2 = nn.Conv2d(25, 32, kernel_size = 3, padding = 2)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(288, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 288)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim = 1)
```

训练过程：

```
def train(epoch):
    net.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        (data, _)= autoencoder(Variable(data.view(-1, 28*28)))
        target = Variable(target)
        optimizer.zero_grad()
        output = net(data.view(-1,1,5,5))
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % log_interval == 0:
```

```
            print('Train Epoch: {} \tLoss: {:.6f}'.format(epoch,
loss.item()))
            train_losses.append(loss.item())
            train_counter.append((batch_idx*64) + ((epoch-
1)*len(train_loader.dataset)))
```

测试过程：

```
def test():
    net.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            (data, _)= autoencoder(Variable(data.view(-1, 28*28)))
            target = Variable(target)
            output = net(data.view(-1,1,5,5))
            test_loss += F.nll_loss(output, target,
size_average=False).item()
            pred = output.data.max(1, keepdim=True)[1]
            correct += pred.eq(target.data.view_as(pred)).sum()
    test_loss /= len(test_loader.dataset)
    test_losses.append(test_loss)
    print('\nloss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
      test_loss, correct, len(test_loader.dataset),
      100. * correct / len(test_loader.dataset)))
```

训练之后再进行测试，最后测试的结果为：