

# 实验 深度学习的编解码 Encoder-Decoder 实现 1

学号: 21030031009 姓名: 惠欣宇 电话: 18149045867

本实验要求独立完成。实验报告正文要求采用小四号字, 中文使用宋体, 英文使用 Times New Roman。对于实验中出现的代码文字, 可使用 Consolas 字体。实验报告要求提交 pdf 电子版。电子版文件命名格式要求为: 学号-姓名.pdf。

## 1. 实验目的

- 1) 深入理解编解码概念、知识。
- 2) 掌握编解码的基本用法。
- 3) 利用 CNN 实现编解码框架对 MNIST 分类。

## 2. 实验预备知识及实验要求

### 2.1. 实验预备知识

- 1) 编解码、CNN、图像分类相关知识
- 2) MNIST 数据集类型、尺寸、数量
- 3) MNIST 下载链接: <http://yann.lecun.com/exdb/mnist/>

### 2.2. 实验要求:

- 1) 上交可以直接运行的代码(包括数据集)。
- 2) 展示整个实验过程, 可适当截图, 并标注说明。

本次实验在 encoder-decoder 的框架下, 使用 python 实现 CNN 网络对 MNIST 数据集的分类。首先要在 encoder 实现特征的提取, 输出 feature map, 再将 feature map 传入 decoder 通过全连接层实现分类。

下面是实验过程:

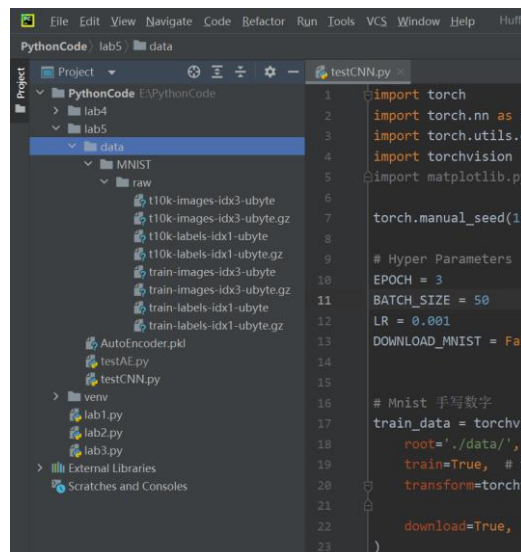
首先下载数据集, pytorch 中包含存在的代码用于 MNIST 数据集的下载。train 为 true 表示加载训练集数据, false 表示加载测试集数据。transform 是将训练集数据转为一个 tensor 并将其扩展一个维度, 将图片从 28x28 的二维数据扩展为 1x28x28 的三位数据, 扩展了一个通道, 并进行归一化。

```
# Mnist 手写数字
train_data = torchvision.datasets.MNIST(
    root='./data/', # 保存或者提取位置
    train=True, # this is training data
    transform=torchvision.transforms.ToTensor(),
    download=True, # 没下载就下载, 下载了就不下载
)
```

测试集数据的读取与下载:

```
test_data = torchvision.datasets.MNIST(root='./data/', train=False)
```

运行下载数据集，在 **data** 下产生一个新的文件夹 **MNIST**，**MNIST** 文件夹中含有实验数据集。



使用 **dataloader** 将训练集一部分数据输入给模型，**dataset** 表示读取训练集还是测试集，**batch\_size** 表示每次读取数据的量，**shuffle** 表示是否随机读取（在训练集中随机地读取 **batch\_size** 个数据输入给模型）：

```
# 批训练 50 samples, 1 channel, 28x28 (50, 1, 28, 28)
train_loader = Data.DataLoader(dataset=train_data, batch_size=BATCH_SIZE, shuffle=True)
```

测试集数据在下载过程中没有进行扩维与归一化操作，则需要下面的操作进行手动扩维与归一，为了节约时间我们只测试前 2000 个数据，**label** 也只取前 2000 个数据：

```
##是对数据维度进行扩充,train_data加载时会自动扩维 为了节约时间,我们测试时只测试前2000个
test_x = torch.unsqueeze(test_data.test_data, dim=1).type(torch.FloatTensor)[:2000]/255
test_y = test_data.test_labels[:2000] #shape (2000,)
```

下面构造 CNN 网络：

CNN 网络包含 **encoder** 与 **decoder** 两部分。

在 **encoder** 中选用卷积操作对数据集进行特征提取，首先使用一个二维卷积 **Conv2d**，经过此操作图片的 **shape** 类型由 **1x28x28** 转换为 **16x28x28**，再使用卷积层与池化层。重复上述操作两遍，最后在 **encoder** 输出后图片的 **shape** 类型就变为了 **32x7x7**。

使用 **decoder** 的全连接层对信息进行分类，首先使用线性全连接层 **Linear**，输入数据为 **32x7x7** 个神经元，将其映射到 **784** 个神经元，在使用完卷积神经网络之后使用 **ReLU** 激活函数，在调用 **dropout** 函数使得某个神经元的激活值以

0.5 的概率停止工作，增强泛型。最后再进行全连接层的映射，将 784 个神经元映射到 10 个神经元上，得到最终分类。

CNN 的具体流程为：将输入的数据传入 **encoder**，提取特征之后存入 **x**，再对 **x** 的 **size** 进行降维，使用 **view** 函数将四维数据转换为二维数据输入给 **decoder**，经过 **decoder** 得到一个 **output** 分类，**output** 中每张图片有十个值，每个值为每个类别的概率预测。

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.encoder = nn.Sequential( # input shape (1, 28, 28)
            nn.Conv2d(
                in_channels=1, # input height
                out_channels=16, # n_filters
                kernel_size=5, # filter size
                stride=1, # filter movement/step
                padding=2, # 如果想要 conv2d 出来的图片长宽没有变化, padding=(kernel_size-1)/2 当 stride=1
            ), # output shape (16, 28, 28)
            nn.ReLU(), # activation
            nn.MaxPool2d(kernel_size=2), # 在 2x2 空间里向下采样, output shape (16, 14, 14)
            nn.Conv2d(16, 32, 5, 1, 2), # output shape (32, 14, 14)
            nn.ReLU(), # activation
            nn.MaxPool2d(2), # output shape (32, 7, 7)
        )
        self.decoder = nn.Sequential(
            nn.Linear(32 * 7 * 7, 784), # fully connected layer,
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(784, 10) # output 10 classes
        )
    def forward(self, x):
        x = self.encoder(x) # (batch_size, 32, 7, 7)
        x = x.view(x.size(0), -1) # 展平多维的卷积图成 (batch_size, 32 * 7 * 7)
        output = self.decoder(x)
        return output # (batch_size, 10)

cnn = CNN()
print(cnn) # net architecture

optimizer = torch.optim.Adam(cnn.parameters(), lr=LR) # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss() # the target label is not one-hot
```

选取 **Adam** 优化器以及交叉熵作为损失函数，接下来构建训练过程：

将 **image** 图像数据输入给模型得到预测结果，再将预测结果与真实结果进行交叉熵损失函数计算。将优化器梯度置为 0，再把损失函数计算结果进行 **backward** 操作，最后进行梯度更新。

进入 **testing** 过程进行测试验证，再将准确率进行输出，使用 **save** 函数保存 CNN 网络。

```
# training and testing
#training
for epoch in range(EPOCH):
    for step, (b_x, b_y) in enumerate(train_loader): # 分配 batch data, normalize x when iterate train_loader
        output = cnn(b_x) # cnn output
        loss = loss_func(output, b_y) # cross entropy loss
        optimizer.zero_grad() # clear gradients for this training step
        loss.backward() # backpropagation, compute gradients
        optimizer.step() # apply gradients

    #testing
    if step%50 == 0:
        test_output = cnn(test_x)
        pred_y = torch.max(test_output, 1)[1].data.numpy().shape(2000,10) #只返回最大值的每个索引
        accuracy = float((pred_y == test_y.data.numpy()).astype(int).sum()) / float(test_y.size(0))
        print('Epoch: ', epoch, '| train loss: %.4f' % loss.data.numpy(), '| test accuracy: %.2f' % accuracy)

torch.save(cnn, 'cnn_minist.pkl')
print('finish training')
```

下面进行验证过程：

使用 `torch.load` 将模型加载进来，将测试集中的 20 个数据输入进模型得到一个预测结果，再对预测值与真实值进行准确率计算，再将预测值与真实值及他们的准确率输出，进行直观对比。

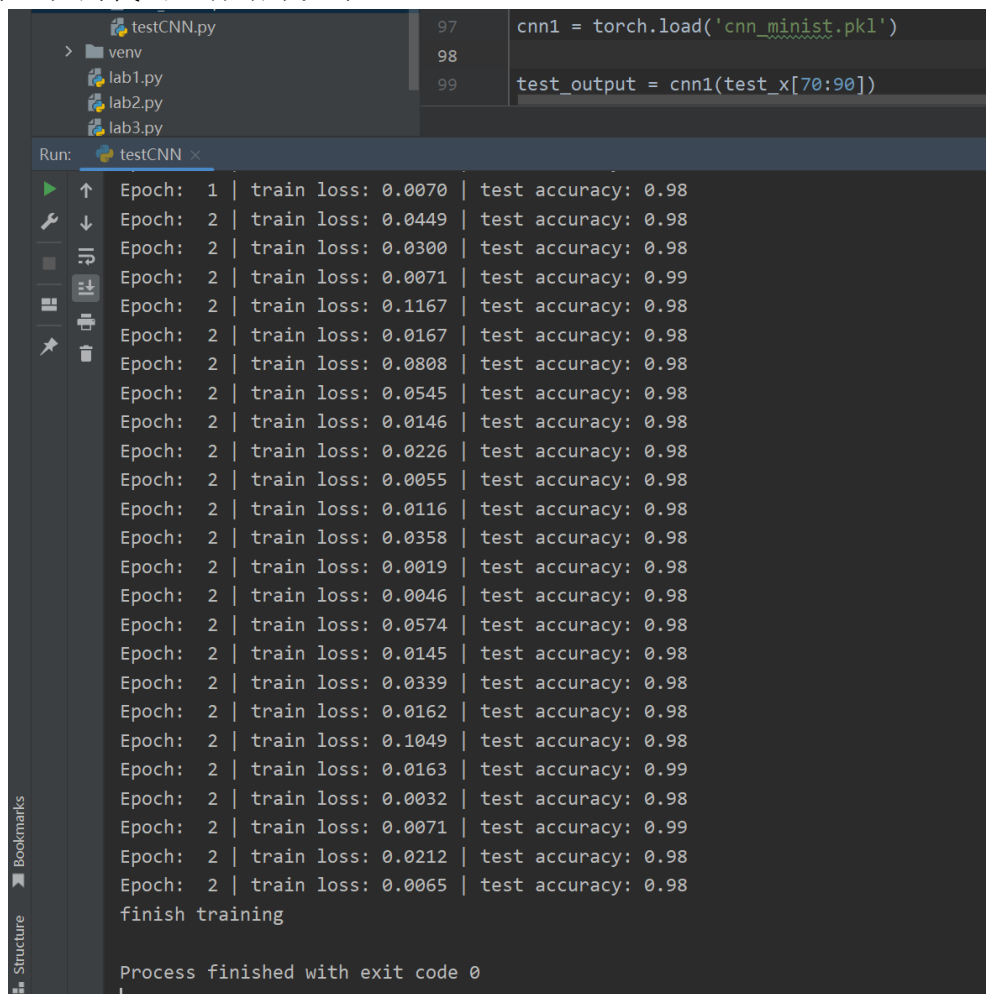
```
#resume test
print('load cnn model')
cnn1 = torch.load('cnn_minist.pkl')

test_output = cnn1(test_x[70:90])
pred_y = torch.max(test_output, 1)[1].data.numpy()
accuracy = float((pred_y == test_y[70:90].data.numpy()).astype(int).sum()) / float(test_y[70:90].size(0))
print(pred_y, 'prediction number')
print(test_y[70:90].numpy(), 'real number')
print('accuracy', accuracy)

#total 2000 test set
total_test_output = cnn1(test_x)
total_pred_y = torch.max(total_test_output, 1)[1].data.numpy()
total_accuracy = float((total_pred_y == test_y.data.numpy()).astype(int).sum()) / float(test_y.size(0))
print('total accuracy', total_accuracy)
```

### 3) 实验结果。

训练过程的代码运行结果如下：

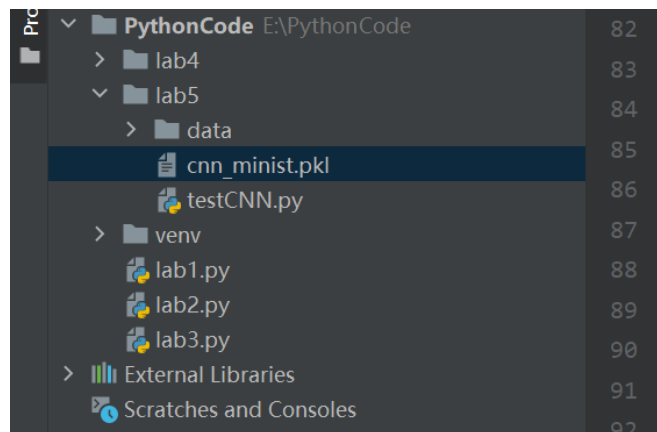


```
testCNN.py 97 cnn1 = torch.load('cnn_minist.pkl')
lab1.py 98
lab2.py 99 test_output = cnn1(test_x[70:90])
lab3.py

Run: testCNN x
Epoch: 1 | train loss: 0.0070 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0449 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0300 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0071 | test accuracy: 0.99
Epoch: 2 | train loss: 0.1167 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0167 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0808 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0545 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0146 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0226 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0055 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0116 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0358 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0019 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0046 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0574 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0145 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0339 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0162 | test accuracy: 0.98
Epoch: 2 | train loss: 0.1049 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0163 | test accuracy: 0.99
Epoch: 2 | train loss: 0.0032 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0071 | test accuracy: 0.99
Epoch: 2 | train loss: 0.0212 | test accuracy: 0.98
Epoch: 2 | train loss: 0.0065 | test accuracy: 0.98
finish training

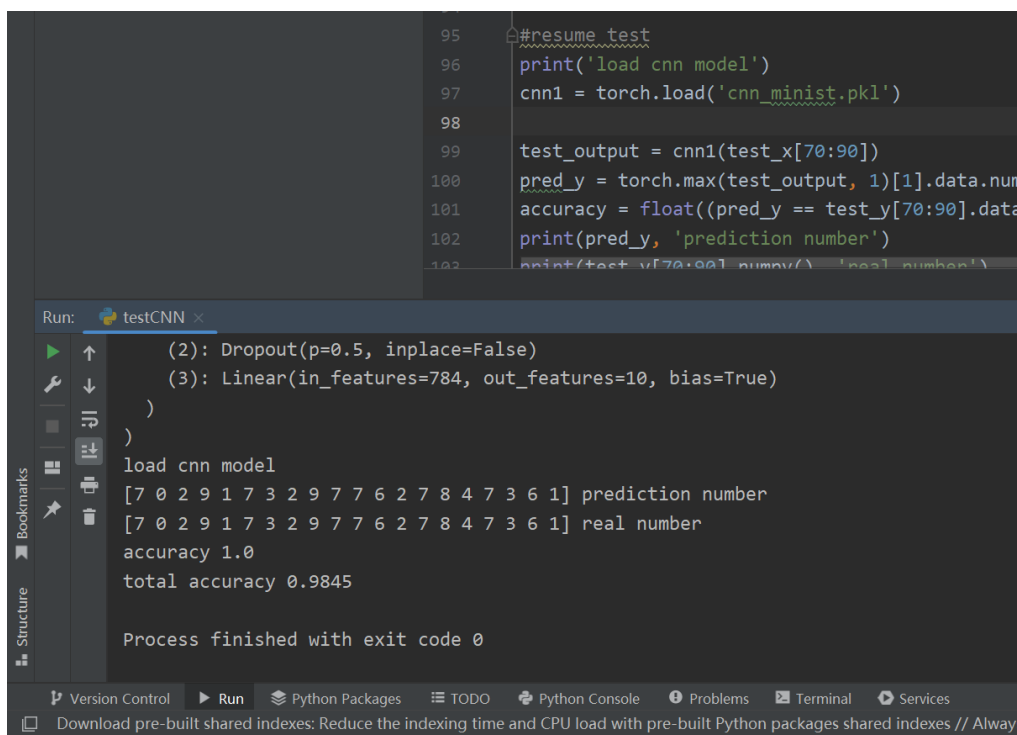
Process finished with exit code 0
```

训练结束会使用 `torch.save` 函数保存模型：



测试集中的 20 个数据预测值与真实值的对比结果如下：

我们发现预测值与真实值相同，模型的预测准确率为 1（100%），测试集的预测准确率为 0.9845（98.45%）。



- 4) 总结分析。总结实现模型搭建和训练过程中所遇到的困难和问题(给出你的解决办法)。总结关键步需要注意的事项。

encoder 采用了一层全连接层，并且采用了 `dropout` 来降低过拟合。我们的输入是三维的 `tensor`，CNN 网络会循环读入，并给出每次循环的网络输出。

decoder 的结构和 encoder 的结构基本一致，区别在于，decoder 每次接受输入的数据为二维的，我们使用 `view` 函数将四维数据转换为二维数据输入给 decoder，经过 decoder 得到一个 `output` 分类。