手写字符案例分析

1、模型结构

1、模型代码

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 32, 3, 1, 1),  # 32x28x28
            nn.ReLU(),
            nn.MaxPool2d(2)
        )  # 32x14x14
        self.conv2 = nn.Sequential(
            nn.Conv2d(32, 64, 3, 1, 1),  # 64x14x14
            nn.ReLU(),
            nn.MaxPool2d(2)  # 64x7x7
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(64, 64, 3, 1, 1),  # 64x7x7
            nn.ReLU(),
            nn.MaxPool2d(2)  # 64x3x3
        )

        self.dense = nn.Sequential(
            nn.Linear(64 * 3 * 3, 128),  # fc4 64*3*3 -> 128
            nn.ReLU(),
            nn.Linear(128, 10)  # fc5 128->10
        )

    def forward(self, x):
        conv1_out = self.conv1(x)
        conv2_out = self.conv2(conv1_out)
        conv3_out = self.conv3(conv2_out)  # 64x3x3
        res = conv3_out.view(conv3_out.size(0), -1)  # batch x (64*3*3)
        out = self.dense(res)
        return out

```

2、主要函数

> Conv2d函数简介：
>
> torch.nn.Conv2d(
>
> in_channels,
> out_channels,
> kernel_size,
> stride=1,
> padding=0,
> dilation=1,
> groups=1,

bias=True,
padding_mode='zeros',
device=None,
dtype=None)

in_channels (int) - 输入图像中的通道数

out_channels (int) – 卷积产生的通道数即输出图片的通道数

kernel_size (int or tuple) – 卷积核的大小(可以是个数，也可以是元组)

stride (int or tuple, optional) -- 卷积的步幅。 默认值：1

padding (int, tuple or str, optional) – 填充添加到输入的所有四个边。 默认值：0

padding_mode (string, optional) –填充的几个选择 'zeros', 'reflect', 'replicate' 或 'circular'。 默认值："零"

dilation (int or tuple, optional) – 内核元素之间的间距。 默认值：1

groups (int, optional) – 从输入通道到输出通道的阻塞连接数。 默认值：1

bias (bool, optional) -- 如果为真，则为输出添加可学习的偏差。 默认值：真

MaxPool2d函数简介

先来看源码：

```
class _MaxPoolNd(Module):
 __constants__ = ['kernel_size', 'stride', 'padding', 'dilation',
                  'return_indices', 'ceil_mode']
 return_indices: bool
 ceil_mode: bool
    # 构造函数，这里只需要了解这个初始化函数即可。
 def __init__(self, kernel_size: _size_any_t, stride: Optional[_size_any_t] = None,
              padding: _size_any_t = 0, dilation: _size_any_t = 1,
              return_indices: bool = False, ceil_mode: bool = False) -> None:
     super(_MaxPoolNd, self).__init__()
     self.kernel_size = kernel_size
     self.stride = stride if (stride is not None) else kernel_size
     self.padding = padding
     self.dilation = dilation
     self.return_indices = return_indices
     self.ceil_mode = ceil_mode

 def extra_repr(self) -> str:
     return 'kernel_size={kernel_size}, stride={stride}, padding={padding}' \
         ', dilation={dilation}, ceil_mode={ceil_mode}'.format(**self.__dict__)

class MaxPool2d(_MaxPoolNd):
 kernel_size: _size_2_t
 stride: _size_2_t
 padding: _size_2_t
 dilation: _size_2_t

 def forward(self, input: Tensor) -> Tensor:
```

```python
        return F.max_pool2d(input, self.kernel_size, self.stride,
                            self.padding, self.dilation, self.ceil_mode,
                            self.return_indices)

```

MaxPool2d 这个类的实现十分简单。

我们先来看一下基本参数，一共六个：

kernel_size ：表示做最大池化的窗口大小，可以是单个值，也可以是tuple元组
stride ：步长，可以是单个值，也可以是tuple元组
padding ：填充，可以是单个值，也可以是tuple元组
dilation ：控制窗口中元素步幅
return_indices ：布尔类型，返回最大值位置索引
ceil_mode ：布尔类型，为True，用向上取整的方法，计算输出形状；默认是向下取整。
关于 kernel_size 的详解：

注意这里的 kernel_size 跟卷积核不是一个东西。 kernel_size 可以看做是一个滑动窗口，这个窗口的大小由自己指定，如果输入是单个值，例如 3 33 ，那么窗口的大小就是 3×3 ，还可以输入元组，例如 (3, 2) ，那么窗口大小就是 3×2 。

最大池化的方法就是取这个窗口覆盖元素中的最大值。

关于 stride 的详解：

上一个参数我们确定了滑动窗口的大小，现在我们来确定这个窗口如何进行滑动。如果不指定这个参数，那么默认步长跟最大池化窗口大小一致。如果指定了参数，那么将按照我们指定的参数进行滑动。例如 stride=(2,3) ， 那么窗口将每次向右滑动三个元素位置，或者向下滑动两个元素位置。

关于 padding 的详解：

这参数控制如何进行填充，填充值默认为0。如果是单个值，例如 1，那么将在周围填充一圈0。还可以用元组指定如何填充，例如 padding=(2,1) ，表示在上下两个方向个填充两行0，在左右两个方向各填充一列0。

关于 dilation 的详解：

不会

关于 return_indices 的详解：

这是个布尔类型值，表示返回值中是否包含最大值位置的索引。注意这个最大值指的是在所有窗口中产生的最大值，如果窗口产生的最大值总共有5个，就会有5个返回值。

关于 ceil_mode 的详解：

这个也是布尔类型值，它决定的是在计算输出结果形状的时候，是使用向上取整还是向下取整。怎么计算输出形状，下面会讲到。一看就知道了。

nn.linear()是用来设置网络中的全连接层的，而在全连接层中的输入与输出都是二维张量，一般形状为[batch_size, size]，与卷积层要求输入输出是4维张量不同。
用法与形参见说明如下：

> CLASS `torch.nn.Linear(in_features, out_features, bias=True)`
>
> Applies a linear transformation to the incoming data: $y = xA^T + b$
>
> Parameters
>
> - **in_features** – size of each input sample
> - **out_features** – size of each output sample
> - **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`
>
> nn.Linear
>
> in_features指的是输入的二维张量的大小，即输入的[batch_size, size]中的size。
> batch_size指的是每次训练（batch)的时候样本的大小。比如CNN train的样张图片是60张，设置batch_size=15，那么iteration=4。如果想多训练几次（因为可以每次的batch不是相同的数据），那么就是epoch。
> 所以nn.Linear()中的输入包括有输入的图片数量，同时还有每张图片的维度。
> out_features指的是输出的二维张量的大小，即输出[batch_size，size]中的size是输出的张量维度，而batch_size与输入中的一致。

3、结构分析

- Conv1层

1. 卷积层：输入三通道，输出32通道，卷积核3×3，步长为1，四边填入1
2. 激活函数：ReLu（）
3. 池化层：最大池化，窗口2×2

- Conv2层

1. 卷积层：输入32通道，输出64通道，卷积核3×3，步长为1，四边填入1
2. 激活函数：ReLu（）
3. 池化层：最大池化，窗口2×2

- Conv3层

1. 卷积层：输入64通道，输出64通道，卷积核3×3，步长为1，四边填入1
2. 激活函数：ReLu（）
3. 池化层：最大池化，窗口2×2

- dense层

1. 全连接层1：输入为64×3×3，输出为128
2. 激活函数：ReLu（）
3. 全连接层2：输入为128，输出为10（十个分类）

流程：Conv1->Conv2->Conv3->dense

2、数据加载代码

1、代码总览

```
def image_list(imageRoot, txt='list.txt'):
    f = open(txt, 'wt')
    for (label, filename) in enumerate(sorted(os.listdir(imageRoot), reverse=False)):
        if os.path.isdir(os.path.join(imageRoot, filename)):
            for imagename in os.listdir(os.path.join(imageRoot, filename)):
                name, ext = os.path.splitext(imagename)
                ext = ext[1:]
                if ext == 'jpg' or ext == 'png' or ext == 'bmp':
                    f.write('%s %d\n' % (os.path.join(imageRoot, filename, imagename), label))
    f.close()


def shuffle_split(listfile, trainfile, valfile):
    with open(listfile, 'r') as f:
        records = f.readlines()
    random.shuffle(records)
    num = len(records)
    trainNum = int(num * 0.8)
    with open(trainfile, 'w') as f:
        f.writelines(records[0:trainNum])
    with open(valfile, 'w') as f1:
        f1.writelines(records[trainNum:])


class MyDataset(Dataset):
    def __init__(self, txt, transform=None, target_transform=None):
        fh = open(txt, 'r')
        imgs = []
        for line in fh:
            line = line.strip('\n')
            line = line.rstrip()
            words = line.split()
            imgs.append((words[0], int(words[1])))
        self.imgs = imgs
        self.transform = transform
        self.target_transform = target_transform

    def __getitem__(self, index):
        fn, label = self.imgs[index]
        img = cv2.imread(fn, cv2.IMREAD_COLOR)
        if self.transform is not None:
            img = self.transform(img)
        return img, label

    def __len__(self):
        return len(self.imgs)
```

2、代码分析

1、image_list

```
 if os.path.isdir(os.path.join(imageRoot, filename)):
```

若在路径中该文件存在，执行下面内容

```
for imagename in os.listdir(os.path.join(imageRoot, filename)):
                name, ext = os.path.splitext(imagename)
                ext = ext[1:]
                if ext == 'jpg' or ext == 'png' or ext == 'bmp':
                    f.write('%s %d\n' % (os.path.join(imageRoot, filename,
imagename), label))
```

遍历文件，os.path.splitext用于分离文件名和拓展名，使用name和ext来储存图片名和后缀。在本案例中，文件均为jpg格式，故ext==".jpg"，所以使用 ext = ext[1:]去除"."

再进行判断，若拓展名为"jpg""png"或者"bmp“,将文件路径、文件名、图片名和标签写入txt文件中

2、shuffle_split

```
with open(listfile, 'r') as f:
        records = f.readlines()
    random.shuffle(records)
```

先读取txt文件中每行的内容，将其打乱

```
num = len(records)
    trainNum = int(num * 0.8)
    with open(trainfile, 'w') as f:
        f.writelines(records[0:trainNum])
    with open(valfile, 'w') as f1:
        f1.writelines(records[trainNum:])
```

计算列表中的行数（即文件数量），前80%写入训练集的txt文件，后20%写入验证集的txt文件

因为文件在前面已经打乱，故在这一步也相当于随机划分

3、MyDataset

```
for line in fh:
            line = line.strip('\n')
            line = line.rstrip()
            words = line.split()
            imgs.append((words[0], int(words[1])))
```

使用strip（）函数去除首尾指定字符

使用rsetrip（）函数去除末尾指定字符，缺省为去除空格

使用split（）函数以输入实参为分割线进行分割，缺省为所有空字符，包括空格、换行和制表符

所以最后word[0]为图片路径，word[1]为标签（即0-9）

```python
 fn, label = self.imgs[index]
        img = cv2.imread(fn, cv2.IMREAD_COLOR)
        if self.transform is not None:
            img = self.transform(img)
        return img, label
```

使用fn和label分别储存图片路径和标签，使用imread读取出来（cv2.IMREAD_COLOR为默认参数，读入一副彩色图片，忽略alpha通道）

返回值为图像和标签

3、训练代码

1、代码总览

```python
def train():
    os.makedirs('./output', exist_ok=True)
    if True: #not os.path.exists('output/total.txt'):
        ml.image_list(args.datapath, 'output/total.txt')
        ml.shuffle_split('output/total.txt', 'output/train.txt', 'output/val.txt')

    train_data = ml.MyDataset(txt='output/train.txt',
                                 transform=transforms.ToTensor())
    val_data = ml.MyDataset(txt='output/val.txt', transform=transforms.ToTensor())
    train_loader = DataLoader(dataset=train_data, batch_size=args.batch_size, shuffle=True)
    val_loader = DataLoader(dataset=val_data, batch_size=args.batch_size)

    model = Net()
    model.to(torch.device("cuda:0"))
    #model = models.resnet18(num_classes=10)  # 调用内置模型
    #model.load_state_dict(torch.load('./output/params_10.pth'))
    #from torchsummary import summary
    #summary(model, (3, 28, 28))

    if args.cuda:
        print('training with cuda')
        model.cuda()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=1e-3)
    scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, [10, 20], 0.1)
    loss_func = nn.CrossEntropyLoss()

    for epoch in range(args.epochs):
        # training-----------------------------------
        model.train()
        train_loss = 0
        train_acc = 0
        # for batch, (batch_x, batch_y) in enumerate(train_loader):
        for (batch_x, batch_y) in tqdm(train_loader,
```

```
                                              desc= 'Epoch:' + str(epoch+1) + '/' + str(args.epochs),
                                              colour='Green'):
            if args.cuda:
                batch_x, batch_y = Variable(batch_x.cuda()), Variable(batch_y.cuda())
            else:
                batch_x, batch_y = Variable(batch_x), Variable(batch_y)
            out = model(batch_x)  # 256x3x28x28  out 256x10
            loss = loss_func(out, batch_y)
            train_loss += loss.item()
            pred = torch.max(out, 1)[1]
            train_correct = (pred == batch_y).sum()
            train_acc += train_correct.item()

            # print('epoch: %2d/%d batch %3d/%d  Train Loss: %.3f, Acc: %.3f'
            #       % (epoch + 1, args.epochs, batch, math.ceil(len(train_data) / args.batch_size),
            #          loss.item(), train_correct.item() / len(batch_x)))

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            scheduler.step()  # 更新learning rate
        print('Train Loss: %.6f, Acc: %.3f' % (train_loss / (math.ceil(len(train_data)/args.batch_size)),
                                               train_acc / (len(train_data))))

        # evaluation--------------------------------
        model.eval()
        eval_loss = 0
        eval_acc = 0
        # for batch_x, batch_y in val_loader:
        for batch_x, batch_y in tqdm(val_loader,
                                     desc='Epoch:' + str(epoch + 1) + '/' + str(args.epochs),
                                     colour='Green'):
            if args.cuda:
                batch_x, batch_y = Variable(batch_x.cuda()), Variable(batch_y.cuda())
            else:
                batch_x, batch_y = Variable(batch_x), Variable(batch_y)

            out = model(batch_x)
            loss = loss_func(out, batch_y)
            eval_loss += loss.item()
            pred = torch.max(out, 1)[1]
            num_correct = (pred == batch_y).sum()
            eval_acc += num_correct.item()
        print('Val Loss: %.6f, Acc: %.3f' % (eval_loss / (math.ceil(len(val_data)/args.batch_size)),
                                             eval_acc / (len(val_data))))
        # save model --------------------------------
        if (epoch + 1) % 1 == 0:
            # torch.save(model, 'output/model_' + str(epoch+1) + '.pth')
```

```
            torch.save(model.state_dict(), 'output/params_' + str(epoch + 1)
+ '.pth')
            #to_onnx(model, 3, 28, 28, 'params.onnx')

if __name__ == '__main__':
    train()
```

2、代码分析

```
ml.image_list(args.datapath, 'output/total.txt')
ml.shuffle_split('output/total.txt', 'output/train.txt', 'output/val.txt')
```

image_list（）函数用来读取路径下的文件

shuffle_split（）函数用来将图片按 4：1 的比例划分为训练集和验证集

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=1e-3)
scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, [10, 20], 0.1)
loss_func = nn.CrossEntropyLoss()
```

第一条设置了优化器函数和学习率等参数

第二条是学习率更新，含义为在第10和第20个epochs时，执行lr = 0.1*lr

第三条是设置损失函数，即交叉熵函数

```
for (batch_x, batch_y) in tqdm(train_loader,
desc= 'Epoch:' + str(epoch+1) + '/' + str(args.epochs),
colour='Green'):
```

深度学习中常用的进度可视化函数，用以取代

```
for batch, (batch_x, batch_y) in enumerate(train_loader):
```

同时依照上面介绍的MyDataset，这段代码中的batch_x为图像，batch_y为标签

```
out = model(batch_x)  # 256x3x28x28  out 256x10
loss = loss_func(out, batch_y)
train_loss += loss.item()
pred = torch.max(out, 1)[1]
train_correct = (pred == batch_y).sum()
train_acc += train_correct.item()
```

将图片输入进行预测，然后通过损失函数计算出损失进行累加

得到的这个out是一个256×10的张量，第二个维度中10个数的和为1，可以理解为0-9各个数字的概率

max函数会返回out[1]中的最大值和最大值索引，我们只需要索引（即0-9），所以在后面加一个[1]

下面两条是累计正确率，若预测结果与标签一致，记为正确

```
 # evaluation--------------------------------
        model.eval()
```

这一段是在验证集上进行验证，故先将模型切换到验证模式，因为不需要进行反向传播，所以可以考虑在前面添加一行with torch.no_grad():

```
# save model --------------------------------
        if (epoch + 1) % 1 == 0:
            # torch.save(model, 'output/model_' + str(epoch+1) + '.pth')
            torch.save(model.state_dict(), 'output/params_' + str(epoch + 1) + '.pth')
```

这一段是每执行一个epoch保存一次，过于消耗空间，可以进行修改

4、代码修改

1、代码总览

```
import torch
import math
import torch.nn as nn
from torch.autograd import Variable
from torchvision import transforms, models
import argparse
import os
from torch.utils.data import DataLoader
from tqdm import tqdm

from dataloader import mnist_loader as ml
from models.cnn import Net
# from toonnx import to_onnx


parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
parser.add_argument('--datapath', required=True, help='data path')
parser.add_argument('--batch_size', type=int, default=1024, help='training batch size')
parser.add_argument('--epochs', type=int, default=200, help='number of epochs to train')
parser.add_argument('--use_cuda', default=False, help='using CUDA for training')

args = parser.parse_args()
# args.cuda = args.use_cuda and torch.cuda.is_available()
args.cuda = True
if args.cuda:
    torch.backends.cudnn.benchmark = True


def train():
    os.makedirs('./output', exist_ok=True)
    if True: #not os.path.exists('output/total.txt'):
        ml.image_list(args.datapath, 'output/total.txt')
        ml.shuffle_split('output/total.txt', 'output/train.txt', 'output/val.txt')

    train_data = ml.MyDataset(txt='output/train.txt', transform=transforms.ToTensor())
```

```python
    val_data = ml.MyDataset(txt='output/val.txt', transform=transforms.ToTensor())
    train_loader = DataLoader(dataset=train_data, batch_size=args.batch_size, shuffle=True, num_workers=4,)
    val_loader = DataLoader(dataset=val_data, batch_size=args.batch_size, num_workers=4,)

    model = Net()
    # model.load_state_dict(torch.load('output/params_mnist.pth'))
     model.to(torch.device("cuda:0"))
    #model = models.resnet18(num_classes=10)  # 调用内置模型
    #model.load_state_dict(torch.load('./output/params_10.pth'))
    #from torchsummary import summary
    #summary(model, (3, 28, 28))

    if args.cuda:
        print('training with cuda')
        model.cuda()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=1e-3)
    # scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, [10, 20], 0.1)
    loss_func = nn.CrossEntropyLoss()
    best_loss = 1000000
    no_optim = 0

    for epoch in range(args.epochs):
        # training-----------------------------------
        model.train()
        train_loss = 0
        train_acc = 0
        # for batch, (batch_x, batch_y) in enumerate(train_loader):
        for (batch_x, batch_y) in tqdm(train_loader,
                                               desc= 'Epoch:' + str(epoch+1) + '/' + str(args.epochs),
                                               colour='Green'):
            if args.cuda:
                batch_x, batch_y = Variable(batch_x.cuda()), Variable(batch_y.cuda())
            else:
                batch_x, batch_y = Variable(batch_x), Variable(batch_y)
            out = model(batch_x)  # 256x3x28x28  out 256x10
            loss = loss_func(out, batch_y)
            train_loss += loss.item()
            pred = torch.max(out, 1)[1]
            train_correct = (pred == batch_y).sum()
            train_acc += train_correct.item()

            # print('epoch: %2d/%d batch %3d/%d  Train Loss: %.3f, Acc: %.3f'
            #       % (epoch + 1, args.epochs, batch, math.ceil(len(train_data) / args.batch_size),
            #          loss.item(), train_correct.item() / len(batch_x)))

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        # scheduler.step()  # 更新learning rate
```

```
        print('Train Loss: %.6f, Acc: %.3f' % (train_loss / (math.ceil(len(train_data)/args.batch_size)),
                                               train_acc / (len(train_data))))
        if loss >= best_loss:
            no_optim += 1
        else:
            no_optim = 0
            best_loss = loss
            torch.save(model.state_dict(), 'output/params_mnist' + '.pth')
        if no_optim >= 3:
            model.load_state_dict(torch.load('output/params_mnist.pth'))
            torch.optim.lr_scheduler.MultiStepLR(optimizer, [(epoch + 1)], 0.1)
            print(("Learn rate changed!!!Best Loss is {}!").format(best_loss))
        if no_optim > 6:
            print('early stop at %d epoch' % (epoch + 1))
            break

        # evaluation--------------------------------
        with torch.no_grad():
            model.eval()
            eval_loss = 0
            eval_acc = 0

            # for batch_x, batch_y in val_loader:
            for batch_x, batch_y in tqdm(val_loader,
                                         desc='Epoch:' + str(epoch + 1) + '/' + str(args.epochs),
                                         colour='Green'):
                if args.cuda:
                    batch_x, batch_y = Variable(batch_x.cuda()), Variable(batch_y.cuda())
                else:
                    batch_x, batch_y = Variable(batch_x), Variable(batch_y)

                out = model(batch_x)
                loss = loss_func(out, batch_y)
                eval_loss += loss.item()
                pred = torch.max(out, 1)[1]
                num_correct = (pred == batch_y).sum()
                eval_acc += num_correct.item()
            print('Val Loss: %.6f, Acc: %.3f' % (eval_loss / (math.ceil(len(val_data)/args.batch_size)),
                                                 eval_acc / (len(val_data))))
        # save model --------------------------------
        # if (epoch + 1) % 1 == 0:
        #     # torch.save(model, 'output/model_' + str(epoch+1) + '.pth')
        #     torch.save(model.state_dict(), 'output/params_' + str(epoch + 1) + '.pth')
            #to_onnx(model, 3, 28, 28, 'params.onnx')

if __name__ == '__main__':
    train()
```

2、主要修改

1、数据加载

```
train_loader = DataLoader(dataset=train_data, batch_size=args.batch_size, shuffle=True)
    val_loader = DataLoader(dataset=val_data, batch_size=args.batch_size)
```

改为

```
train_loader = DataLoader(dataset=train_data, batch_size=args.batch_size, shuffle=True, num_workers=4)
    val_loader = DataLoader(dataset=val_data, batch_size=args.batch_size, num_workers=4)
```

加快了数据读取速度

2、模型保存

```
# save model ------------------------------
        if (epoch + 1) % 1 == 0:
            # torch.save(model, 'output/model_' + str(epoch+1) + '.pth')
            torch.save(model.state_dict(), 'output/params_' + str(epoch + 1) + '.pth')
```

改为

```
 if loss >= best_loss:
            no_optim += 1
        else:
            no_optim = 0
            best_loss = loss
            torch.save(model.state_dict(), 'output/params_mnist' + '.pth')
```

首先定义一个最小损失，当这个epoch的损失小于最小损失时，保存模型同时更新最小损失

这样只需要保存一个模型，同时能够保证保存的模型为最优模型

3、学习率更新

```
scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, [10, 20], 0.1)
```

改为

```
if no_optim >= 3:
            model.load_state_dict(torch.load('output/params_mnist.pth'))
            torch.optim.lr_scheduler.MultiStepLR(optimizer, [(epoch + 1)], 0.1)
            print(("Learn rate changed!!!Best Loss is {}!").format(best_loss))
```

当连续四次模型损失值未下降时，认为在当前学习率下模型已收敛，此时加载最优模型，同时更新学习率为原来的0.1

相较于之前的在固定epoch更新学习率的做法来看，这种方法显然更加智能

4、停止训练

添加

```
if no_optim > 6:
            print('early stop at %d epoch' % (epoch + 1))
            break
```

当连续8次损失未下降时，认为此时模型已经收敛，直接退出训练