

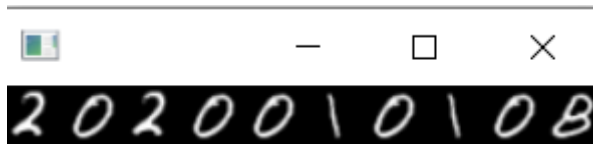
《程序设计实践》作业 3 说明文档

徐浩博 2020010108 软件 02

0 运行环境

python 3.9.12
pytorch 1.8.0 + cuda
opencv-python 4.6.0.66
matplotlib 3.3.3

1 图片可视化



完成方法：修改了 visualizer.py 中的 demo_display_specific_digit_combination 函数，选取对应图片拼接在一起

2 训练模型并使用它进行推理



完成方法：

```
(torch-1.8-python39) PS C:\Users\Xsu1023\Desktop\backend\hw3\LeNet-PyTorch_initial> & C:/Users/Xsu1023/anaconda3/envs/torch-1.8-python39/python  
c:/Users/Xsu1023/Desktop/backend/hw3/LeNet-PyTorch_initial/inference.py --image_path='data/test_data/8.jpg'  
inference label is: 8
```

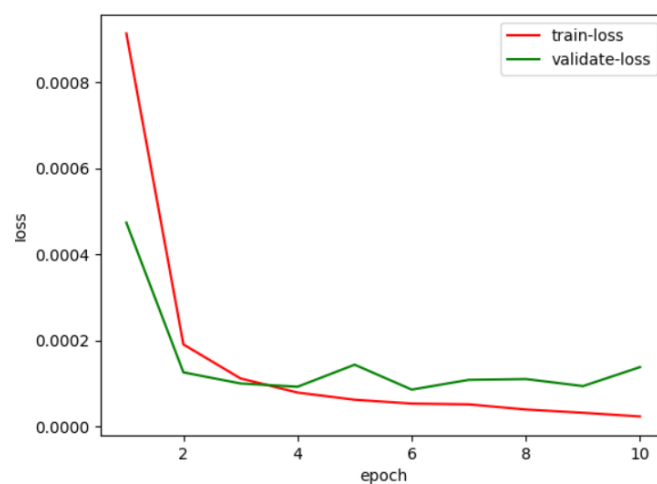
用 bash 运行时附加—image_path 参数，指定为对应图片即可

3 绘制 Loss 曲线

完成方法：使用 **matplotlib 内的 pyplot 进行绘制**。首先记录每个 epoch 进行 train 和 validate 时的 loss 平均值，此即纵坐标（记录在数组 loss_data 中）；然后利用 epoch 数生成横坐标数列，最后利用 plot 函数指定颜色和 label。我将横纵坐标我将验证集和训练集的 loss 平均

值同时绘制在了同一张图上。具体实现方法见 train.py 的 draw_plot 函数。

```
x1 = range(1, len(loss_data['train'])+ 1)
y1 = loss_data['train']
x2 = range(1, len(loss_data['vali']) + 1)
y2 = loss_data['vali']
plt.plot(x1, y1, 'r-', label = 'train-loss')
plt.plot(x2, y2, 'g-', label = 'validate-loss')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```



4 更换优化器

完成方法：修改 train.py 中 train 函数中的 optimizer：

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

结果讨论：

以下为 Adam 和 SGD 的每个 epoch 的 accuracy 比较

epoch	Adam	SGD
1	98.2 %	86.14 %
2	98.58 %	90.57 %
3	98.97 %	92.47 %
4	99.1 %	93.65 %
5	98.99 %	94.44 %
6	99.03 %	95.04 %
7	99.01 %	95.42 %
8	99.19 %	95.6 %
9	99.05 %	95.97 %

10	99.08 %	96.21 %
----	---------	---------

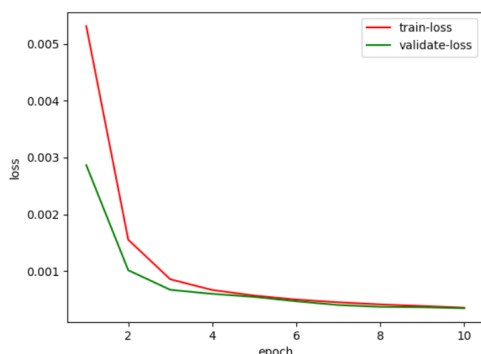


Figure a) SGD

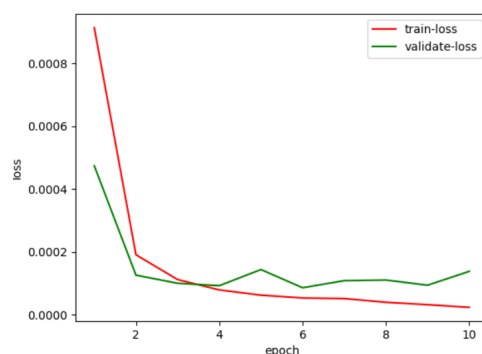


Figure b) Adam

能够看出，与 Adam 相比，SGD 的 performance 略差，对比二者 loss 曲线也可以看出，Adam 收敛速度快于 SGD。因此综合来看，Adam 的性能好于 SGD，这与 Adam 利用了动量机制和固定时间窗口的方式有关。

5 添加数据预处理

实现方法：更改 pre_process.py 中的 data_augment_transform 函数（由于 Adam 的 accuracy 过高，不易看到优化效果，因此以下均采用 SGD）

随机剪裁

```
data_augment = torchvision.transforms.Compose([
    torchvision.transforms.RandomResizedCrop(size = 28),
    torchvision.transforms.ToTensor(),
])
```

注意，RandomResizedCrop 默认值如下：RandomResizedCrop(size, scale=(0.08,1),ratio=(0.75,1.3333)) 而对于 MNIST 来说，scale 下界 0.08 过小，大约只有几个像素，即使是人眼也不可能辨别，为此我改变了 scale 进行对比。以下为**随机剪裁**各个情况的 accuracy：

epoch	原始	scale=(0.08,1)	scale=(0.6,1)	scale=(0.8,1)
1	86.14 %	66.63 %	82.45	86.6 %
2	90.57 %	82.41 %	89.82	91.96 %
3	92.47 %	89.26 %	92.67	94.03 %
4	93.65 %	91.86%	94.48	95.28 %
5	94.44 %	92.2 %	95.22	95.71 %
6	95.04 %	91.81 %	95.89	96.23 %
7	95.42 %	91.86 %	96.26	96.51 %
8	95.6 %	93.01 %	96.4	96.94 %
9	95.97 %	94.34%	96.66	97.07 %
10	96.21 %	93.87%	96.83	97.15 %

可以看到，默认的 scale=(0.08,1)效果反而不如未进行数据增广，而 scale 下界改为 0.6 和 0.8 均提高了 performance，这是因为 MNIST 数据集小，而数据增广提高了数据集的丰富性，优化了模型的泛化能力。

水平翻转

```
data_augment = torchvision.transforms.Compose([
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor(),
])
```

垂直翻转

```
data_augment = torchvision.transforms.Compose([
    torchvision.transforms.RandomVerticalFlip(),
    torchvision.transforms.ToTensor(),
])
```

以下分别为**水平翻转**、**垂直翻转**和原始数据集的 accuracy 的比较：

epoch	原始	水平翻转	垂直翻转
1	86.14 %	77.54 %	77.53 %
2	90.57 %	86.13 %	87.71 %
3	92.47 %	89.34 %	87.31 %
4	93.65 %	90.91 %	91.59 %
5	94.44 %	90.68 %	91.92 %
6	95.04 %	89.78 %	93.14 %
7	95.42 %	91.53 %	93.82 %
8	95.6 %	93.81 %	91.23 %
9	95.97 %	94.32 %	94.74 %
10	96.21 %	93.22 %	93.87 %

能够看到，两种数据增广方式均在不同程度上造成了 performance 的下降，原因可能与 MNIST 数据集的图片类型有关。无论如何，手写数字进行水平或垂直翻转都可能造成一定的混淆，如 6 上下翻转成为 9，即使是人眼也会出错。而对于 CIFAR10、ImageNet 等真实图片的数据集，进行翻转可能会在一定程度上提升 performance

6 添加数据预处理

Module
weight <10×1024>
bias <10>

Module
weight <64>
bias <64>
running_mean <64>
running_var <64>
num_batches_tracked = 235

Module
weight <64×32×5×5>
bias <64>

Module
weight <32>
bias <32>
running_mean <32>
running_var <32>
num_batches_tracked = 235

Module
weight <32×16×5×5>
bias <32>

Module
weight <16>
bias <16>
running_mean <16>
running_var <16>
num_batches_tracked = 235

Module
weight <16×1×5×5>
bias <16>

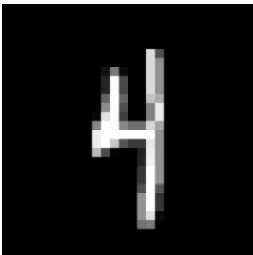
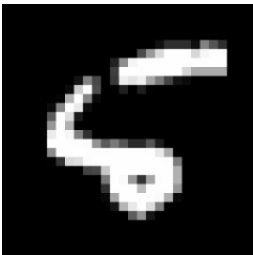
实现方法：更改 lenet.py 中__init__和 forward，具体来说，我在 layer2 和 fc 之间添加了一层卷积层 layer3 并修改了 fc 的参数

```
self.layer3 = torch.nn.Sequential(  
  
    torch.nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2),  
  
    torch.nn.BatchNorm2d(64),  
  
    torch.nn.ReLU(),  
  
    torch.nn.MaxPool2d(kernel_size=4, stride=1))  
self.fc = torch.nn.Linear(4 * 4 * 64, num_classes)
```

最后，我用 [Netron \(lutzroeder.github.io\)](http://lutzroeder.github.io)实现了模型可视化，结果如左图所示。

7 错误样例分析

实现方法：在 train.py 中的 evaluate 函数中调用 draw_wrong_case 函数，在其中寻找到错误样例，然后调用 visualizer.demo_display_single_image 函数将错误传递过去，在该函数中用 cv2 库中的 imwrite 绘制图片。注意，错误样例分析在训练的最后的几个 epoch 进行，绘制的图片位于 train.py 同级目录下，命名方式为" labelX_predictedY.jpg"，意为 label 标记图片为 X 而实际预测为 Y。

	ground truth	predicted	原因
	4	9	手写 4 和 9 区别在于顶上连与不连，模型可能误将不连的 4 认成了 9
	5	2	本样例中 5 写得不规范，加上图片像素较低，即使肉眼辨认也不很容易

	6	4	本样例中 6 写得不规范， 将 6 斜着写， 的确与连笔的 4 十分相像
	7	9	7 书写时中间加一横这个习惯， 并不是人人都有， 可能在训练集中缺少相应例子； 而且 7 加一横以后的确与 9 有几分相似， 使得验证时出现偏差
	9	4	手写 4 和 9 区别在于顶上连与不连， 模型可能误将连的 9 认成了 4

综合以上，有一些例子的图片书写不规范，加之像素太低，即使肉眼辨识也有难度，机器判断出错情有可原；另一些数字如 4 和 9 本身在手写体中就较为相似，加之训练集较小，因此出错率就会变高。