组会汇报

2.18

结论

可能原因是loss被困到了局部极值点，解决策略如下：

- 对神经网络的模型参数进行初始化
- 添加BN层加快收敛速度
- 换用对学习率不敏感的优化策略，比如lr为0.001的adam，数据预处理时归一化
- 可以check中间层的输出，查看权重变化
- 检查最后一层是否非线形层，或者加了ReLU激活

动物数据集

```python
class_to_num={}
class_name_list = os.listdir('animals')
print(class_name_list)
for class_name in class_name_list:
    class_to_num[class_name] = len(class_to_num.keys())

image_dir=[]
for class_name in class_name_list:
    image_dir +=glob.glob(os.path.join('animals',class_name,'*.jpg'))
random.shuffle(image_dir) #很重要

print(class_to_num)
with open('animal_data.csv',mode='w',newline='') as f:
    writer = csv.writer(f)
    for image in image_dir:
        class_name = image.split(os.sep)[-2]
        label = class_to_num[class_name]
        writer.writerow([image,label])
```

```
epoch 80, loss 0.0250, train accuracy 0.992, test accuracy 0.766, use time 19.630
epoch 81, loss 0.0628, train accuracy 0.991, test accuracy 0.771, use time 19.633
epoch 82, loss 0.0817, train accuracy 0.977, test accuracy 0.784, use time 19.611
epoch 83, loss 0.0903, train accuracy 0.975, test accuracy 0.771, use time 19.587
epoch 84, loss 0.0332, train accuracy 0.990, test accuracy 0.796, use time 19.643
epoch 85, loss 0.0198, train accuracy 0.997, test accuracy 0.822, use time 19.602
```

Vgg19

```python
class makedataset(Dataset):
    def __init__(self, csv_filename, resize, mode):
        super(makedataset, self).__init__()

        self.csv_filename = csv_filename
        self.resize = resize
        self.image, self.label = self.load_csv()

        if mode == 'train':
            self.image = self.image[:int(0.8 * len(self.image))]
            self.label = self.label[:int(0.8 * len(self.label))]
        elif mode == 'test':
            self.image = self.image[int(0.8 * len(self.image)):]
            self.label = self.label[int(0.8 * len(self.label)):]

    def load_csv(self):
        image, label = [], []
        with open(self.csv_filename) as f:
            reader = csv.reader(f)
            for row in reader:
                i, l = row
                image.append(i)
                label.append(int(l))
        return image, label

    def __len__(self):
        return len(self.image)

    def __getitem__(self, idx):
        tf = transforms.Compose([lambda x:Image.open(x).convert('RGB'),
                                 transforms.Resize((self.resize,self.resize)),
                                 transforms.ToTensor()
```

初始结构

```
epoch 83, loss 0.1116, train accuracy 0.970, test accuracy 0.822, use time 62.638
epoch 84, loss 0.1064, train accuracy 0.971, test accuracy 0.817, use time 62.341
epoch 85, loss 0.1066, train accuracy 0.971, test accuracy 0.817, use time 62.886
epoch 86, loss 0.1058, train accuracy 0.971, test accuracy 0.821, use time 63.869
epoch 87, loss 0.1049, train accuracy 0.971, test accuracy 0.818, use time 63.109
epoch 88, loss 0.1046, train accuracy 0.971, test accuracy 0.811, use time 63.259
epoch 89, loss 0.1028, train accuracy 0.973, test accuracy 0.814, use time 62.481
epoch 90, loss 0.1044, train accuracy 0.971, test accuracy 0.816, use time 62.690
epoch 91, loss 0.1013, train accuracy 0.973, test accuracy 0.811, use time 63.031
epoch 92, loss 0.1026, train accuracy 0.972, test accuracy 0.820, use time 62.374
epoch 93, loss 0.0994, train accuracy 0.973, test accuracy 0.815, use time 62.680
epoch 94, loss 0.1045, train accuracy 0.972, test accuracy 0.815, use time 62.285
epoch 95, loss 0.0953, train accuracy 0.975, test accuracy 0.818, use time 61.993
epoch 96, loss 0.0982, train accuracy 0.973, test accuracy 0.815, use time 62.807
epoch 97, loss 0.0964, train accuracy 0.974, test accuracy 0.814, use time 63.228
epoch 98, loss 0.0991, train accuracy 0.973, test accuracy 0.815, use time 62.370
epoch 99, loss 0.0958, train accuracy 0.974, test accuracy 0.811, use time 63.388
```

NIN

```
epoch 86, loss 0.0457, train accuracy 0.985, test accuracy 0.801, use time 115.214
epoch 87, loss 0.0464, train accuracy 0.984, test accuracy 0.803, use time 113.128
epoch 88, loss 0.0497, train accuracy 0.983, test accuracy 0.805, use time 113.502
epoch 89, loss 0.0408, train accuracy 0.987, test accuracy 0.808, use time 112.787
epoch 90, loss 0.0492, train accuracy 0.983, test accuracy 0.808, use time 113.546
epoch 91, loss 0.0383, train accuracy 0.987, test accuracy 0.805, use time 113.985
epoch 92, loss 0.0385, train accuracy 0.987, test accuracy 0.788, use time 111.756
epoch 93, loss 0.0426, train accuracy 0.986, test accuracy 0.809, use time 114.458
epoch 94, loss 0.0348, train accuracy 0.989, test accuracy 0.805, use time 113.487
epoch 95, loss 0.0229, train accuracy 0.993, test accuracy 0.811, use time 115.391
epoch 96, loss 0.0109, train accuracy 0.997, test accuracy 0.819, use time 114.441
epoch 97, loss 0.0037, train accuracy 1.000, test accuracy 0.825, use time 112.666
epoch 98, loss 0.0017, train accuracy 1.000, test accuracy 0.824, use time 115.159
epoch 99, loss 0.0012, train accuracy 1.000, test accuracy 0.820, use time 112.755
```

googLeNet

优化过程

torch.nn.init.trunc_normal_(*tensor*, *mean=0.0*, *std=1.0*, *a=- 2.0*, *b=2.0*) [SOURCE]

Fills the input Tensor with values drawn from a truncated normal distribution. The values are effectively drawn from the normal distribution $\mathcal{N}(\text{mean}, \text{std}^2)$ with values outside $[a, b]$ redrawn until they are within the bounds. The method used for generating the random values works best when $a \leq \text{mean} \leq b$.

使用trunc_normal截断正太分布初始化以及增加BN层后

```
epoch 36, loss 0.0336, train accuracy 0.990, test accuracy 0.893, use time 136.459
epoch 37, loss 0.0369, train accuracy 0.988, test accuracy 0.894, use time 134.852
epoch 38, loss 0.0342, train accuracy 0.989, test accuracy 0.894, use time 132.137
epoch 39, loss 0.0294, train accuracy 0.991, test accuracy 0.893, use time 134.175
epoch 40, loss 0.0303, train accuracy 0.990, test accuracy 0.898, use time 134.738
epoch 41, loss 0.0247, train accuracy 0.992, test accuracy 0.892, use time 142.660
epoch 42, loss 0.0247, train accuracy 0.992, test accuracy 0.900, use time 139.286
epoch 43, loss 0.0251, train accuracy 0.992, test accuracy 0.900, use time 141.199
```

```
# 随机裁剪图像，所得图像为原始面积的0.08～1之间，高宽比在3/4和4/3之间。
# 然后，缩放图像以创建224x224的新图像
torchvision.transforms.RandomResizedCrop(224, scale=(0.08, 1.0),
                                         ratio=(3.0/4.0, 4.0/3.0)),
torchvision.transforms.RandomHorizontalFlip(),
# 随机更改亮度，对比度和饱和度
torchvision.transforms.ColorJitter(brightness=0.4,
                                   contrast=0.4,
                                   saturation=0.4),
```

对训练数据进行缩放裁剪、更改亮度等预处理

```
epoch 40, loss 0.5203, train accuracy 0.821, test accuracy 0.897, use time 213.687
epoch 41, loss 0.5120, train accuracy 0.823, test accuracy 0.897, use time 207.006
epoch 42, loss 0.5158, train accuracy 0.824, test accuracy 0.901, use time 206.221
epoch 43, loss 0.5014, train accuracy 0.828, test accuracy 0.899, use time 206.830
epoch 44, loss 0.4945, train accuracy 0.828, test accuracy 0.901, use time 207.446
epoch 45, loss 0.4992, train accuracy 0.828, test accuracy 0.901, use time 206.700
epoch 46, loss 0.4921, train accuracy 0.832, test accuracy 0.904, use time 207.152
epoch 47, loss 0.4885, train accuracy 0.833, test accuracy 0.904, use time 206.712
epoch 48, loss 0.4908, train accuracy 0.830, test accuracy 0.905, use time 206.546
epoch 49, loss 0.4758, train accuracy 0.837, test accuracy 0.899, use time 191.281
```

```
epoch 92, loss 0.3769, train accuracy 0.873, test accuracy 0.919, use time 130.611
epoch 93, loss 0.3714, train accuracy 0.874, test accuracy 0.918, use time 129.655
epoch 94, loss 0.3709, train accuracy 0.873, test accuracy 0.919, use time 128.391
epoch 95, loss 0.3731, train accuracy 0.872, test accuracy 0.916, use time 130.134
epoch 96, loss 0.3806, train accuracy 0.871, test accuracy 0.920, use time 129.456
epoch 97, loss 0.3694, train accuracy 0.874, test accuracy 0.914, use time 126.094
epoch 98, loss 0.3704, train accuracy 0.874, test accuracy 0.924, use time 122.222
epoch 99, loss 0.3679, train accuracy 0.875, test accuracy 0.920, use time 123.831
```

改用预训练模型

```
epoch 23, loss 0.0274, train accuracy 0.991, test accuracy 0.939, use time 130.863
epoch 24, loss 0.0251, train accuracy 0.992, test accuracy 0.938, use time 132.496
epoch 25, loss 0.0244, train accuracy 0.992, test accuracy 0.944, use time 132.956
epoch 26, loss 0.0272, train accuracy 0.991, test accuracy 0.941, use time 129.945
epoch 27, loss 0.0249, train accuracy 0.992, test accuracy 0.922, use time 129.153
epoch 28, loss 0.0266, train accuracy 0.992, test accuracy 0.943, use time 133.338
epoch 29, loss 0.0297, train accuracy 0.990, test accuracy 0.942, use time 133.346
epoch 30, loss 0.0316, train accuracy 0.990, test accuracy 0.941, use time 131.865
```

googLeNet

• GoogLeNet的总体设计目标是什么？（有三个方面）
设计一个神经网络，增加深度和宽度（更深22层），但不增加 计算量（计算量是AlexNet的1/12），精度更好（2014年ImageNet比赛冠军）

• 直接去掉网络中较小权重的稀疏方法有什么缺陷?
从硬件角度分析，稀疏的硬件实现是低效的（大量的查找和缓存缺失）

• 从设计思路讲，Inception是如何实现稀疏的?
卷积核、batch个数大时，卷积也可稠密运算
稀疏矩阵聚类为稠密的子矩阵
用密集的卷积计算近似稀疏结构

• 如何将赫布理论应用到Inception的设计中?
赫布理论：如果两个神经元同步激发，则它们之间的权重增加；如果单独激发，则权重减少。
根据赫布理论，通过分析某些神经元之间激活值的相关性，将相关度高的神经元聚合，获取一个稀疏表示（利用1* 1卷积实现）

• Inception的结构大概是怎么样的？有几个分支，分别是什么?
①用1x1卷积实现赫布理论，把神经元聚在一起
②为了增加聚类时的空间信息，使用patch更大的卷积核，个数相应减少
③为了方便，使用1x1，3x3，5x5卷积，输出 combine
④由于pooling效果很好，也添加并行的pooling分支
改进：用1x1卷积降维,相当于做embedding
An embedding is **a relatively low-dimensional space into which you can translate high-dimensional vectors**.
做法： ①1x1卷积放在3x3和5x5之前降维 ②1x1卷积放在pooling后面降维 ③1x1后面有ReLU，增加非线性

• Inception是如何实现多尺度特征提取的?
Inception采用多尺度并列的结构，捕捉多尺度信息并融合，输出到下一层 这是因为，视觉信息应该在不同的尺度上进行处理，然后进行聚合，以便下一阶段能够同时从不 同的尺度上提取特征。

• Inception中1x1卷积的作用是什么?
降维，减少 计算量 在此基础上，就可以实现在不 增加计算量的情况下，增加卷 积神经网络的深度和宽度

• GoogLeNet中添加辅助loss的目的是什么？测试的时候是否还需要辅助loss?
在中间添加两个额外的softmax（辅助 loss），用于缓解梯度消失，这两个loss 的比重较小(0.3)，在测试时，这两个loss 会被去掉

ResNet

```
epoch 41, loss 0.4923, train accuracy 0.830, test accuracy 0.899, use time 107.033
epoch 42, loss 0.4830, train accuracy 0.834, test accuracy 0.895, use time 107.627
epoch 43, loss 0.4815, train accuracy 0.834, test accuracy 0.906, use time 107.184
epoch 44, loss 0.4779, train accuracy 0.834, test accuracy 0.900, use time 107.087
epoch 45, loss 0.4767, train accuracy 0.834, test accuracy 0.905, use time 107.079
epoch 46, loss 0.4698, train accuracy 0.836, test accuracy 0.905, use time 106.888
epoch 47, loss 0.4648, train accuracy 0.837, test accuracy 0.902, use time 106.808
epoch 48, loss 0.4625, train accuracy 0.840, test accuracy 0.906, use time 107.006
epoch 49, loss 0.4652, train accuracy 0.837, test accuracy 0.908, use time 107.244
```

```
epoch 43, loss 0.4882, train accuracy 0.832, test accuracy 0.900, use time 116.079
epoch 44, loss 0.4822, train accuracy 0.832, test accuracy 0.903, use time 115.952
epoch 45, loss 0.4797, train accuracy 0.833, test accuracy 0.904, use time 116.265
epoch 46, loss 0.4780, train accuracy 0.833, test accuracy 0.904, use time 115.441
epoch 47, loss 0.4816, train accuracy 0.833, test accuracy 0.906, use time 116.467
epoch 48, loss 0.4680, train accuracy 0.837, test accuracy 0.907, use time 116.819
epoch 49, loss 0.4669, train accuracy 0.839, test accuracy 0.911, use time 116.494
epoch 50, loss 0.4617, train accuracy 0.840, test accuracy 0.904, use time 116.169
```

```
epoch 41, loss 0.3532, train accuracy 0.875, test accuracy 0.952, use time 101.852
epoch 42, loss 0.3483, train accuracy 0.879, test accuracy 0.952, use time 101.580
epoch 43, loss 0.3460, train accuracy 0.879, test accuracy 0.950, use time 101.580
epoch 44, loss 0.3416, train accuracy 0.881, test accuracy 0.947, use time 103.048
epoch 45, loss 0.3403, train accuracy 0.882, test accuracy 0.944, use time 101.794
epoch 46, loss 0.3348, train accuracy 0.884, test accuracy 0.947, use time 102.072
epoch 47, loss 0.3365, train accuracy 0.884, test accuracy 0.951, use time 101.175
epoch 48, loss 0.3273, train accuracy 0.886, test accuracy 0.950, use time 101.642
epoch 49, loss 0.3343, train accuracy 0.885, test accuracy 0.949, use time 101.105
```

ResNet

(or unable to do so in feasible time).

In this paper, we address the degradation problem by introducing a *deep residual learning* framework. Instead of hoping each few stacked layers directly fit a desired underlying mapping, we explicitly let these layers fit a residual mapping. Formally, denoting the desired underlying mapping as $\mathcal{H}(\mathbf{x})$, we let the stacked nonlinear layers fit another mapping of $\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) - \mathbf{x}$. The original mapping is recast into $\mathcal{F}(\mathbf{x}) + \mathbf{x}$. We hypothesize that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping. To the extreme, if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers.

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \cdot \frac{\partial g(x)}{\partial x}$$

$$\frac{\partial f(g(x)) + g(x)}{\partial x} = \square + \frac{\partial g(x)}{\partial x}$$

9

A. Object Detection Baselines

In this section we introduce our detection method based

8 i

所以是一个小数加上一个大数