

基于卷积神经网络的糖尿病眼底病变分类问题

学院：计算机学院

姓名：林帆

学号：2113839

专业：计算机科学与技术

目录

- 一、 摘要
- 二、 思路分析
- 三、 学习过程
- 四、 结果实现
- 五、 失败经验教训
- 六、 预期优化
- 七、 代码及参考文献

一、摘要

糖尿病视网膜病变(DR)分级在医疗诊断中具有一定的现实意义,定期的眼底检查可以预防 DR 引起的失明。彩色眼底图像是 DR 筛查的主要材料,需要通过医生结合个人经验判断病变程度。但数量不断增加的 DR 患者及病变的特异性,和医生数量的有限,给人工实时分析带来一定难度。因此计算机辅助诊断,对 DR 的分级,能一定程度上缓解诊断的难度。

在拥有较大量数据集的基础上,本文提出了一种基于深度学习(Deep Learning)和卷积神经网络(CNN)的糖尿病眼底病变分类方法。该方法基于 PyTorch 框架,对图像数据进行了一定的预处理,使用了在另一种图片分类中性能较好的 CNN 网络,并对比使用 NN、ResNet 等轻量级网络的性能,根据训练集(train)在同等训练次数下不同的准确率,对网络进行了一定的修正和筛选。过程中也重写了训练函数(trainfunc),利用 valid 集测试,输出每次训练的准确率,观察学习成果。另外也控制变量,尝试了多种损失函数(Lossfunction)和优化器(Optimizer),经过大量的实验,对比数据,选出了与所选择网络适配度最高的函数。

本文也对附件论文【1】中所遇到的几个问题提出了一些方法,并进行了不同程度上的尝试。针对五种分级颜色和纹理的相似性过高问题,使用图像增强(Image Enhance)的方法,增大各个数据之间的差异性;针对图片中对分类有负面影响的干扰部分,参考论文【2】中提出的方法,对图像使用高斯滤波(Gauss filtering)进行减噪处理;针对样本数据不平衡问题,以数据过采样为主体思路,尝试使用了两种方法——文件夹扩增和权重随机采样(WeightedRandomSampler),实现对数据的有效采样,提高数据的平衡性。

关键词: DR 分级 卷积神经网络 图像增强 权重随机采样 减噪处理

二、思路分析

本次分级任务是利用所给数据集,和所选择神经网络进行机器学习的过程。所给数据集是对应等级下的.jpg 格式,即所给样例的特征为非结构性数据,因而将图片转化为机器可识别的结构性数据,是较为重要的一步。又因为每个图片的等级唯一且固定,亦即每个样例的标签唯一且固定,因而本次机器学习任务是一个有监督的单一标签分类任务。

初次接触神经网络,可将网络理解为机器学习中的一种特殊的 model。通过尝试使用已经搭建好的网络,学习并更改部分参数,即可尝试将网络实例化。剩下需要做的主体工作为:数据处理、训练函数的实现、准确率测试、神经网络的选择、参数调整、优化器和损失函数的选择、问题解决及优化。

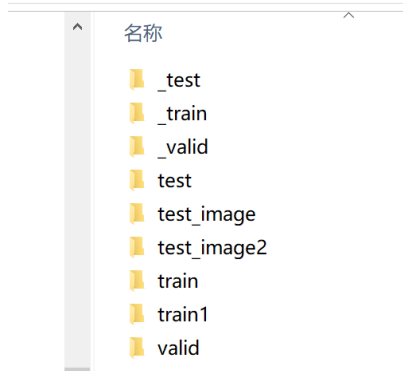
三、学习过程

1. 数据处理

1.1 文件中新数据集的创建

在 DDR 原有三个数据集（train, valid, test）的基础上，创建新数据集

Data (D:) > test_code > _DDR



扩充后，DDR 中各数据集的数据量及其作用如下表所示：

数据集	数据量	作用
train	2992+1638+2238+613+2370	标准训练集
test	1880+189+1344+71+275	标准测试集
valid	1253+126+895+47+182	训练过程中的测试集
_train	361+361+361+361+361	对应上述三个标准集中的部分数据，同等级少量相等取样，规避数据不平衡的影响，同时方便测试训练函数
_test	16+16+16+16+16	
_valid	18+18+18+18+18	
train1	2992+2880+2958+3065+2946	文件夹中图片扩增实现过采样后的样本
test_image	1	图片增强的测试样例
test_image2	0	

1.2 导入工具包

利用 python 中自带的工具包，可以便捷地实现很多功能

```
1 from multiprocessing import freeze_support
2
3 import torch
4 import torchvision
5 import torchvision.transforms as transforms
6 from torchvision.datasets import ImageFolder
7 import torch.nn as nn
8 import torch.nn.functional as F
9 import matplotlib.pyplot as plt
10 import numpy as np
11 import torch.optim as optim
12 from PIL import Image
13 from PIL import ImageEnhance
14
15 import torch
16 import torch.nn as nn
17 from torch.nn import functional as F
```

1.3 使用 ImageFolder 函数提取数据集至 dataset 中

1.3.1

torchvision.datasets 中的 ImageFolder 函数可以将文件按照所给路径提取，并将数据所在文件夹名作为其标签。trainset, testset, validset 分布存储提取和类型转换后三个数据集。

```
print("preparing data...")
# 导入下载
# trainset=torchvision.datasets.CIFAR10(root='D:/test_code/DDR/train/00',train=True,
# ImageFolder将数据按文件夹名字分类贴上标签
trainset = ImageFolder(root='D:/test_code/DDR/train', transform=trans_form)
testset = ImageFolder(root='D:/test_code/DDR/test', transform=trans_form)
validset = ImageFolder(root='D:/test_code/DDR/valid', transform=trans_form)
```

*Tips:一开始的三个路径建议使用_train,_test,_valid, 可以加载少量数据, 节省运行时间, 主要用于检测当前语法是否正确。

1.3.2

调用.class_to_idx 函数, 输出所有类别

调用 len() 函数输出 dataset 的长度

语句:

```
149     print("set is ok")
150     print(trainset.class_to_idx)
151     print(testset.class_to_idx)
152     print(validset.class_to_idx)
153
154     print(len(trainset))
155     print(len(testset))
156     print(len(validset))
```

输出:

```
set is ok
{'0': 0, '1': 1, '2': 2, '3': 3, '4': 4}
{'0': 0, '1': 1, '2': 2, '3': 3, '4': 4}
{'0': 0, '1': 1, '2': 2, '3': 3, '4': 4}
9851
3759
2503
```

可以发现, 类别[0, 1, 2, 3, 4]与 train 中所存储不同等级数据的文件夹名字相同, 9851、3759 和 2503 也分别对应了 train, test, valid 中的图片数据个数

1.3.3

trainset (后两个同上) 数据为 CIFAR10 类型, 包含了图片提取像素所得的多个数组 (特征) 和一个标签值, print(trainset[0]) 观察:

语句:

```
print(trainset[0])
```

输出:

```
(tensor([[[[-2.4291, -2.4291, -2.4291, ..., -2.4291, -2.4291, -2.4291],
          [-2.4291, -2.4291, -2.4291, ..., -2.4291, -2.4291, -2.4291],
          [-2.4291, -2.4291, -2.4291, ..., -2.4097, -2.4291, -2.4291],
          ...,
          [-2.4291, -2.4291, -2.4291, ..., -2.4291, -2.4291, -2.4291],
          [-2.4291, -2.4291, -2.4291, ..., -2.4291, -2.4291, -2.4291],
          [-2.4291, -2.4291, -2.4291, ..., -2.4291, -2.4291, -2.4291]],

        [[[-2.4183, -2.4183, -2.4183, ..., -2.4183, -2.4183, -2.4183],
          [-2.4183, -2.4183, -2.4183, ..., -2.4183, -2.4183, -2.4183],
          [-2.4183, -2.4183, -2.4183, ..., -2.4183, -2.4183, -2.4183],
          ...,
          [-2.4183, -2.4183, -2.4183, ..., -2.4183, -2.4183, -2.4183],
          [-2.4183, -2.4183, -2.4183, ..., -2.4183, -2.4183, -2.4183],
          [-2.4183, -2.4183, -2.4183, ..., -2.4183, -2.4183, -2.4183]],

        [[[-2.2214, -2.2214, -2.2214, ..., -2.2214, -2.2214, -2.2214],
          [-2.2214, -2.2214, -2.2214, ..., -2.2214, -2.2214, -2.2214],
          [-2.2214, -2.2214, -2.2214, ..., -2.2214, -2.2214, -2.2214],
          ...,
          [-2.2214, -2.2214, -2.2214, ..., -2.2214, -2.2214, -2.2214],
          [-2.2214, -2.2214, -2.2214, ..., -2.2214, -2.2214, -2.2214],
          [-2.2214, -2.2214, -2.2214, ..., -2.2214, -2.2214, -2.2214]]]])
```

上图中[]即为数组, 0 为标签, 表明第 0 个 trainset 的类别为 0
至此, 三个数据集都已经成功导入 dataset 中

1.4 调用 torch, 定义图像预处理函数

转换器 transform 实现对图像的放缩、剪裁、翻转、格式转化和归一化处理

```
127 freeze_support()
128 # 图像预处理
129 # compose整合
130 trans_form = transforms.Compose([
131     transforms.Resize((32, 32)),
132     transforms.RandomCrop(32, padding=2), # 随机位置进行裁剪
133     # 32*32 (相当于没有随机)
134     transforms.RandomHorizontalFlip(), # 以0.5的概率水平翻转给定的PIL图像
135     transforms.ToTensor(), # 先由HWC转为CHW格式; 再转为float类型; 最后, 每个像素除以255
136     transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)), # 归一化
137 ])
```

1.5 dataset 加载到数据加载器 dataloader 中

dataloader 可以将数据打乱提取, 多线程分批处理

其中参数定义如下:

batch_size: 每批处理的数据量

shuffle: 是否打乱数据

num_workers: 线程个数

```
158 # 加载
159 trainloader = torch.utils.data.DataLoader(trainset, batch_size=20, shuffle=True, num_workers=4) # 打乱
160 testloader = torch.utils.data.DataLoader(testset, batch_size=12, shuffle=True, num_workers=4)
161 validloader = torch.utils.data.DataLoader(testset, batch_size=4, shuffle=True, num_workers=4)
162 print("loader is ok")
```

可以输出 dataloader 的长度观察:

语句:

```
162     print("loader is ok")
163     print(len(trainloader))
```

输出:

```
loader is ok
493
```

可以发现, $493 \times 20 = 9860 \approx 9851 = \text{len}(\text{trainset})$

成功验证 dataloader 将 dataset 中所有元素按 batch_size 个为一批打包, 共有 batch 个包

至此, 初步的数据处理和转化已经完成, 数据处理优化部分详见该部分后文
7. 问题解决及模型优化

2. 训练函数的实现

2.1 自定义训练函数的优点

(1) 实参方便更改, 形参无变动

```
55     # 训练函数
56     def trainfunc(dataloader, validloader, net, lossfunc, optimizer):
```

如图, trainfunc 函数中除了 dataloader, validloader (用于测试) 以外, 还有所使用的网络参数 net, 所选择的损失函数 lossfunc 和优化器 optimizer 在后续选择时, 只需要改变调用 trainfunc 时所传入的实际参数, 而无须更改 trainfunc 内部的形式参数, 更加方便, 不易出错

(2) 在循环实现多次训练时, 每次循环执行简洁的一句 trainfunc, 使得代码更加整洁美观

(3) 加深个人对机器学习训练过程的理解

2.2 训练函数的实现

2.2.1 训练部分

```
57     # 分batch批次训练, 最终全部数据都经过一次训练
58     # 每一批训练后, 对参数在原有基础上进行一次更新, 对新误差进行计算
59     for batch, (data, target) in enumerate(dataloader):
60         # 现有参数下迭代一次网络
61         output = net(data)
62         # forward: 将数据传入模型, 前向传播求出预测的值
63         # 求误差, 优化, 更新参数
64         loss = 0 # 误差初始化为0
65         optimizer.zero_grad() # 梯度初始化为零, 把loss关于weight的导数变成0
66         loss = lossfunc(output, target) # 用给定损失函数lossfunc求loss
67         loss.backward() # backward: 反向传播求梯度
68         optimizer.step() # optimizer: 更新所有参数
69     # 网络中参数将损失函数和优化器连接
```

关键点:

- (1) 训练与与 dataloader 相适应, 分批 batch 传入数据, 即分批训练网络
- (2) 分割特征值(data)与自身的标签(target)
- (3) 将自变量 data 放入网络中求得网络的预测标签 output
- (4) 每批进入网络时, 误差 loss 和优化器的梯度初始化为 0
- (5) 用损失函数 lossfunc 计算 (预测值 output, 真实值 target) 之间的损失
- (6) Backward() 反向传播求梯度
- (7) 在每一批数据进入网络, 并计算了损失和梯度后, Optimizer.step() 更新网络中的所有参数, 不断的更新会使得新的预测值与真实值越来越接近
- (8) 网络中的参数作为损失函数与优化器的桥梁, 将二者连接起来

2.2.2 测试部分

每加载 100 批后, 用 valid 对训练成果进行检验

```
71 # 全部数据训练完后输出
72 if batch % 100 == 99:
73     correct = 0
74     print(batch, "测试")
75     total = len(validset)
76     for i, (images, labes) in enumerate(validloader):
77         outputs = net(images)
78         _, predicted = torch.max(outputs.data, 1)
79         correct += (predicted == labes).sum().item()
80     print("accuracy:", correct / total)
```

关键点:

- (1) 用相同的循环句式分批分割处理 validloader
- (2) 将标签 images 放入网络中求得预测结果 outputs
- (3) Outputs 是预测结果, 不是预测值! outputs 是一批装有五个元素的数组, 存储对应每个类别的概率。其最大值对应的下标才是预测值 predicted
- (4) Correct 计算分类正确的结果个数, total 是总测试样本数

2.2.3 多次训练

在 main 函数中, 用循环语句实现机器的多次训练学习, 使得参数不断更新

```
168 # 训练网络
169 for epoch in range(20):
170     print(epoch, "train")
171     trainfunc(trainloader, validloader, net, lossfunc, optimizer)
172
173     print('Finished Training')
```

可以看到, 随着训练次数增加, 训练过程中的准确率整体呈现为上升趋势: 经过三次训练, 模型的正确率便从 40%→60%→70%


```

accuracy: 0.4898122253296045
accuracy: 0.738713543747503
1 train
accuracy: 0.6212544946064722
accuracy: 0.6628046344386735
accuracy: 0.73112265281662
accuracy: 0.7083499800239712
2 train
accuracy: 0.7363164202956453
accuracy: 0.7530962844586496
accuracy: 0.7726727926488214
accuracy: 0.7694766280463444

```

3. 准确率测试

在 20 次训练结束后，机器已经学到了最优的参数，此时用与训练过程测试 valid 部分大体相同的代码实现对测试集 test 的测试：

```

175         # 测试集
176         total = len(testset)
177         correct = 0
178         for i, (images, labels) in enumerate(testloader):
179             outputs = net(images)
180             _, predicted = torch.max(outputs.data, 1)
181             correct += (predicted == labels).sum().item()
182         print("accuracy:", correct / total)
183

```

对 test 的测验也是主函数的最后一步，输出最终的准确率，即为机器学习结果和所用网络性能的体现

4. 神经网络的选择

4.1 CNN

4.1.1

一开始选择了网站【3】实现常见生活事物分类所用的 CNN 神经网络，将全连接层的参数改为 5，即五个分类等级，即可建立 class Net

该网络在最基础的 CNN 的框架上，调整参数，增加了多层卷积，减少全连接层数，因而一定程度上提升了网络的性能

参数 nn.Module 表明该自建的网络类也是对 nn.Module 的继承

4.1.2

在主函数中，用类的无参构造函数构建实例化网络对象

```

net = Net() # 构建网络

```

4.1.3 网络结构如下图所示

```
71 # 构建神经网络
72 class Net(nn.Module):
73     def __init__(self):
74         super(Net, self).__init__()
75         self.layer1 = nn.Sequential(
76             nn.Conv2d(3, 64, kernel_size=3, padding=1),
77             nn.BatchNorm2d(64),
78             nn.ReLU(),
79             nn.MaxPool2d(2, 2)
80         )
81         self.layer2 = nn.Sequential(
82             nn.Conv2d(64, 64, kernel_size=3, padding=1),
83             nn.BatchNorm2d(64),
84             nn.ReLU(),
85             nn.MaxPool2d(2, 2)
86         )
87         self.layer3 = nn.Sequential(
88             nn.Conv2d(64, 128, kernel_size=3, padding=1),
89             nn.BatchNorm2d(128),
90             nn.ReLU(),
91             nn.MaxPool2d(2, 2)
92         )
93         self.fc1 = nn.Linear(4 * 4 * 128, 5)
94
95     def forward(self, x):
96         x = self.layer1(x)
97         x = self.layer2(x)
98         x = self.layer3(x)
99         x = x.view(-1, 128 * 4 * 4)
100         x = self.fc1(x)
101         return x
```

经过 3 次训练后，测试准确率如下：

```
Finished Training
accuracy: 0.5461558925246076

Process finished with exit code 0
```

4.2 NN

依然是选用了网站【3】提供的网络 NN，使用了七层全连接神经网络结构如下所示：

```
104 class Net(nn.Module):
105     def __init__(self):
106         super(Net, self).__init__()
107         self.layer1 = nn.Sequential(
108             nn.Linear(3 * 32 * 32, 256),
109             nn.ReLU(),
110             nn.Dropout(0.1)
111         )
112         self.layer2 = nn.Sequential(
113             nn.Linear(256, 500),
114             nn.Dropout(0.1)
115         )
116         self.layer3 = nn.Sequential(
117             nn.Linear(500, 500),
118             nn.ReLU(),
119         )
```

```

120         self.layer4 = nn.Sequential(
121             nn.Linear(500, 256),
122             nn.ReLU(),
123             nn.Dropout(0.5)
124         )
125         self.layer5 = nn.Sequential(
126             nn.Linear(256, 256),
127             nn.ReLU(),
128             nn.Dropout(0.5)
129         )
130         self.layer6 = nn.Sequential(
131             nn.Linear(256, 128),
132             nn.ReLU(),
133         )
134         self.fc1 = nn.Linear(128, 5)
135
136     def forward(self, x):
137         x = x.view(x.size(0), -1)
138         x = self.layer1(x)
139         x = self.layer2(x)
140         x = self.layer3(x)
141         x = self.layer4(x)
142         x = self.layer5(x)
143         x = self.layer6(x)
144
145         # x = x.view(-1, 256*2*2)
146         x = self.fc1(x)
147         return x

```

3 次训练后，测试结果如下：

```

Finished Training
accuracy: 0.4993349295025273

Process finished with exit code 0

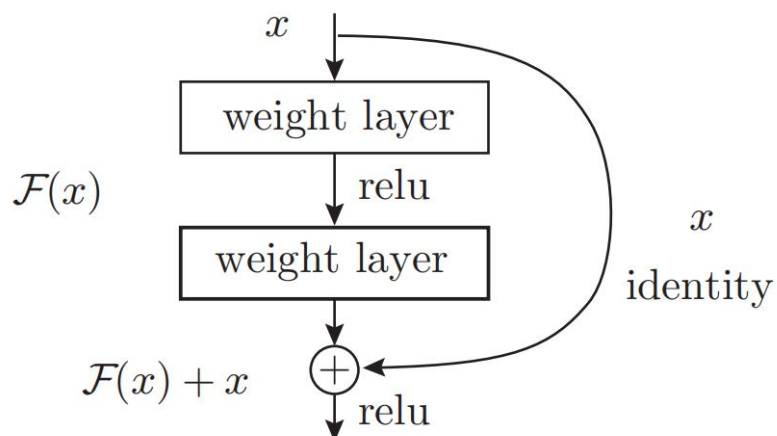
```

效果明显不如 CNN 好

4.3 ResNet

相比于前两个网络，ResNet 的结构更深，其中的快捷结构同时也解决了过度加深层对网络性能的影响

如图所示（图源文献【4】）



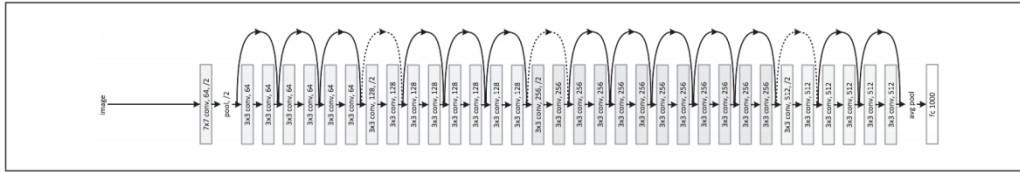


图 8-13 ResNet(引用自文献[24]): 方块对应 3×3 的卷积层, 其特征在于引入了横跨层的快捷结构

通过快捷结构, 反向传播时信号可以无衰减地传递, 因而规避了层加深带来的影响, ResNet 结构如下所示:

```

149 class Net(nn.Module):
150
151     def __init__(self):
152         super(Net, self).__init__()
153         self.layer1 = nn.Sequential(
154             nn.Conv2d(3, 64, kernel_size=3, padding=1),
155             nn.BatchNorm2d(64),
156             nn.ReLU(),
157             nn.MaxPool2d(2, 2)
158         )
159         self.layer2 = nn.Sequential(
160             nn.Conv2d(64, 64, kernel_size=3, padding=1),
161             nn.BatchNorm2d(64),
162             nn.ReLU(),
163             nn.MaxPool2d(2, 2)
164         )
165         self.layer3 = nn.Sequential(
166             nn.Conv2d(64, 128, kernel_size=3, padding=1),
167             nn.BatchNorm2d(128),
168             nn.ReLU(),
169             nn.MaxPool2d(2, 2)
170         )
171         self.fc1 = nn.Linear(4 * 4 * 128, 5)
172
173     def forward(self, x):
174         x = self.layer1(x)
175         x = self.layer2(x)
176         x = self.layer3(x)
177         x = x.view(-1, 128 * 4 * 4)
178         x = self.fc1(x)
179         return x

```

经过 3 次训练, 测试准确率如下:

```

Finished Training
accuracy: 0.4022346368715084

Process finished with exit code 0

```

但性能依旧没有 CNN 好

4.4 其他网络

除了上述三种网络以外，学习过程中也尝试使用 RNN、DNN 和 CBAnet 等较深的网络，但都尝试失败，没能运行出结果

因而综合考虑，最终选择了轻量且性能较好的 CNN 作为主干网络

5. 参数调整

5.1 batch_size 的调整

5.1.1

考虑到训练过程中，按批传数据至实例化的网络训练，因而每批数据越多，理论上参数拟合的越好。但由于电脑运行能力有限，不能同时处理过量数据，因而 batch_size 不能过大。因而需要调整 batch_size，找到最佳性能点

5.1.2

控制其他无关变量不变，包括数据集、训练次数、网络、优化器和损失函数等不变，分别更改 batch_size 的值，得到对应测试正确率，如下组图所示

20:

```
Finished Training
accuracy: 0.5461558925246076
Process finished with exit code 0
```

25:

```
Finished Training
accuracy: 0.5059856344772546
Process finished with exit code 0
```

30:

```
Finished Training
accuracy: 0.5309922851822293
Process finished with exit code 0
```

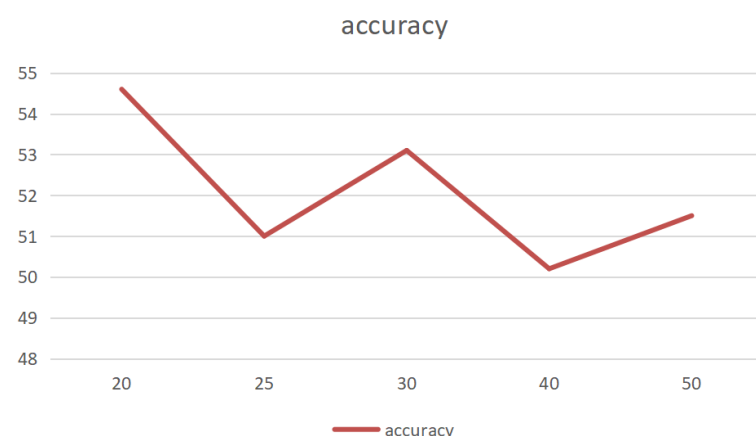
40:

```
Finished Training
accuracy: 0.5022612396914072
Process finished with exit code 0
```

50:

```
Finished Training
accuracy: 0.5150305932428837
Process finished with exit code 0
```

绘制图如下所示:



由此可知，batch_size=20 为最佳选择

5.1.2 网络参数的调整

在多次增加/减少卷次层，改变卷积核数量和大小等尝试后，最终选择了原有参数，实现最高性能

6. 优化器和损失函数的选择

6.1 优化器选择

在确定好网络和参数后，分别选择了 5 种优化器，调用语句如下：

```
288 # optim.  
289 # optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9) # 优化函数（随机梯度优化方法）  
290 # optimizer = optim.Adam(net.parameters(), lr=0.01)  
291 # optimizer=torch.optim.Adagrad(net.parameters(),lr=0.01, lr_decay=0, weight_decay=0)  
292 # optimizer=torch.optim.Adadelta(net.parameters(), lr=1.0, rho=0.9, eps=1e-06, weight_decay=0)  
293 optimizer = torch.optim.RMSprop(net.parameters(), lr=0.01, alpha=0.99, eps=1e-08, weight_decay=0, momentum=0,  
294                                centered=False)
```

不同优化器下，三次训练后的运行结果如下所示：

SGD:

```
Finished Training  
accuracy: 0.535780792764033  
  
Process finished with exit code 0
```

Adadelta:

```
Finished Training  
accuracy: 0.5182229316307528  
  
Process finished with exit code 0
```

Adagrad:

```
Finished Training  
accuracy: 0.5562649640861931  
  
Process finished with exit code 0
```

Adam:

```
Finished Training  
accuracy: 0.5437616387337058  
  
Process finished with exit code 0
```

RMS:

```
Finished Training  
accuracy: 0.5594573024740622  
  
Process finished with exit code 0
```

最终选择了 RMS 算法作为优化器

6.2 损失函数选择

同上所述，损失函数最终选择了交叉熵损失函数，赋值语句如下：

```
# 交叉熵损失函数  
lossfunc = nn.CrossEntropyLoss()
```

7. 问题解决及模型优化

7.1 过采样解决数据不平衡问题

由数据集数量可见，训练集中，第 0 类 2992 个数据，第 3 类只有 613 个数据，相差较大，数据不平衡。因而机器学习会有一定的偏向性，原因是对第 3 类的学习不够充分。因而通过过采样合理选取样本，使得每个类别的训练量相差较小，能够在一定程度上提高机器学习的性能。本文将介绍两种过程中使用的方法——文件夹扩增和权重随机采样(WeightedRandomSampler)。

7.1.1 低样本量的文件夹图片扩增

通过手动操作 control+C, control+V，将训练集中不同类别文件夹里的图片不同数量地扩增，存至 train1 中，实现低样本量数据的扩增。由 1.1 中表格可见，训练集在扩增前后，不同类别的数据个数分别为 [2992, 1638, 2238, 613, 2370]，[2992, 2880, 2958, 3065, 2946]

扩增后明显各个数据量差异非常小。又由于 dataloader 中设置了 shuffle=True, 因而所有的数据都会随机加载, 即可以忽略连续两次学习相同数据 (极小概率情况) 的情况, 因而该方法在一定程度上实现了数据的平衡。

与原代码相比, 只需将 dataset 的加载路径从 train 改为扩增后的 train1 即可。增加新语句, 对此时的 trainset 数据分类计数, 用于检测训练集数据是否平衡

语句:

```
113     n=[0,0,0,0,0,0]
114     for i in range(len(trainset)):
115         num=trainset[i]
116         n[num]+=1
117     print(n[0],n[1],n[2],n[3],n[4])
```

输出:

```
preparing data...
2992 2880 2958 3065 2946
```

平衡验证成功

运行结果如下所示:

```
Finished Training
accuracy: 0.5238095238095238

Process finished with exit code 0
```

但与原来相比, 正确率下降了, 与理论不相符, 只能更换方法, 并且需要在后期的学习中研究思考这个问题。

7.1.2 权重随机采样(WeightedRandomSampler)

pycharm 运行的过程中, 使用空闲态的 jupyter notebook 学习 torch.utils.data 中的 WeightedRandomSampler 采样器

7.1.2.1 采样器功能介绍

WeightedRandomSampler() 函数会根据给定的权重数组, 对数据进行尽可能贴近权重的随机采样。采样器会根据采样的个数和权重, 对不同数量的数据集进行过采样和欠采样等操作, 使得数据尽可能平衡。函数中有三个重要的参数, 如下表所示:

参数	作用
samples_weight	权重数组
samples_num	需要采样的个数
replacement	为 True 时可重复采样, 为 False 时不可

7.1.2.2 采样器运用

在初次尝试陌生函数时, 使用数据量较少的_train 样本(共 985 个)

- (1) replacement 设置为 1, samples_weight 设置为 985

(2) 通过 7.1.1 提到的分类计数的语句，统计样本中各类别的个数，输出结果如下图所示：

```
preparing data...
985
采样
[517, 36, 297, 9, 126]
```

(3) 计算权重数组

为实现数据采样后的平衡，应当使得数量较大的类别采样权重赋小，数量较小的类别权重赋大。因而不同类别数据，权重赋为其数量占比的倒数，是最简单的方法。即 $weight[i] = len(dataset) / n[i]$

赋值之后输出采样结果，如图所示：

```
[985, 0, 0, 0, 0]
```

只采集了类别为 0 的数据，并且多次尝试更换参数后，依然得不到与预期相符的结果。只能放在后期的预期优化中，日后重新尝试。

7.2 图像增强增大数据区分度

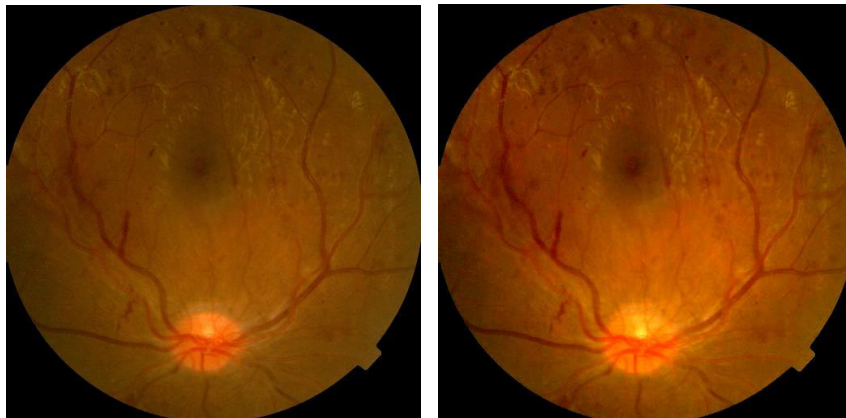
7.2.1 单张图像增强

参考文献【2】中提到，图像因光照影响导致的阴影需要被矫正，同时图像中病变区域和背景之间的对比度需要被提高，因而可采用对比增强的办法。由于【2】所采用的 Ben Graham 等提出的彩色视网膜图像对比增强的方法较为复杂且本人难以实现，故采用了 PIL 库中的 ImageEnhance 来实现最基础的图像增强——对比度增强、色度增强和锐度增强

对单张 .jpg 处理的代码如下所示：（在 test_image 中测试）

```
123     img = Image.open('D:/test_code/DDR/test_image/1.jpg')
124     img.show()
125
126     # 对比度增强
127     enh_con = ImageEnhance.Contrast(img)
128     contrast = 1.5
129     img_contrasted = enh_con.enhance(contrast)
130     img_contrasted.show()
131
132     # 色度增强
133     enh_col = ImageEnhance.Color(img)
134     color = 1.5
135     image_colored = enh_col.enhance(color)
136     image_colored.show()
137
138     # 锐度增强
139     enh_sha = ImageEnhance.Sharpness(img)
140     sharpness = 3.0
141     image_sharped = enh_sha.enhance(sharpness)
142     image_sharped.show()
143
144     img_contrasted.save('D:/test_code/DDR/test_image2/2.jpg')]
```


随后在文件夹中生成了增强后的图片(.jpg)
对比如下所示:



原图像

增强后图像

可以观察到,增强后的图像病变区域和背景的分度明显提升,病变更加清晰,且由于光照和仪器的影响产生的阴影也有了一定程度的矫正,因而图像增强更有利于后续卷积神经的学习,和图像间的区分

7.2.2 批量图像增强

由于上述 ImageEnhance 的使用需要加载每张图片的路径,而我们所使用的巨大量的样本,使用这个方法显然不够合理。因而采用了 augly 库中的函数,对三个测试集进行批量增强处理。

augly 库中已知的部分图像增强函数如图所示:

```
imaugs.blur(aug_image, radius=random.randint(1, 2)),          # 图像模糊
imaugs.brightness(aug_image, factor=random.uniform(0.5, 1.5)), # 改变亮度
imaugs.change_aspect_ratio(aug_image, ratio=random.uniform(0.8, 1.5)), # 改变图像宽高比
imaugs.color_jitter(aug_image, brightness_factor=random.uniform(0.8, 1.5), # 颜色晃动
                    contrast_factor=random.uniform(0.8, 1.5), saturation_factor=random.uniform(0.8, 1.5)),
imaugs.crop(aug_image, x1=random.uniform(0, 0.1), y1=random.uniform(0, 0.1), x2=random.uniform(0.9, 1),
            y2=random.uniform(0.9, 1)), # 随机裁剪
imaugs.hflip(aug_image), # 水平翻转
imaugs.opacity(aug_image, level=random.uniform(0.5, 1)), # 改变图像透明度
imaugs.pixelization(aug_image, ratio=random.uniform(0.5, 1)), # 马赛克
imaugs.random_noise(aug_image), # 随机噪声
imaugs.rotate(aug_image, degrees=random.randint(3, 10)), # 随机旋转一定角度
imaugs.shuffle_pixels(aug_image, factor=random.uniform(0, 0.1)), # 随机像素比任意化
imaugs.saturation(aug_image, factor=random.uniform(1, 1.5)), # 改变饱和度
imaugs.contrast(aug_image, factor=random.uniform(1, 1.5)), # 对比度增强
imaugs.grayscale(aug_image) # 转灰度
```

本文选择了其中两个函数:

(1) 对比度增强:

```
imaugs.contrast(aug_image, factor=random.uniform(1, 1.5))
```

(2) 锐化处理:

```
imaugs.sharpen(aug_image, factor=1.5)
```

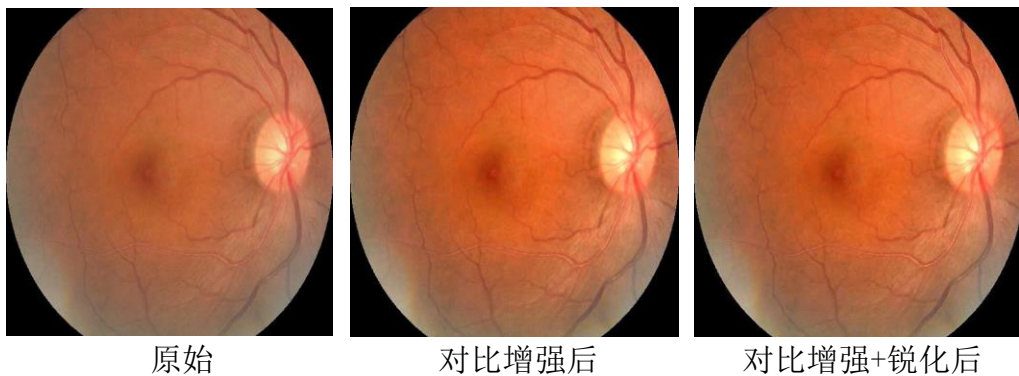
操作具体实现为:

在 jupyter notebook 中先加载原始数据集中的图片,进行对比度增强,存入空文件夹 1 中,再加载文件夹 1 中的图片,进行锐化处理,放入空文件夹 2 中。将 pycharm 中数据集加载的路径更改为新的文件夹。

部分代码如下所示:

```
img_path = "D:/test_code/DDR/final_valid/44" # 需要增强的图像路径
save_path = "D:/test_code/DDR/final_valid/4" # 保存路径
|
def augly_augmentation(aug_image):
    aug = [
        imaugs.contrast(aug_image, factor=random.uniform(1, 1.5)), # 对比度增强
        imaugs.sharpen(aug_image, factor=1.5), # 锐化
    ]
    return aug[1]
```

对比原始文件夹、文件夹 1 和文件夹 2 中同一位置的图片，如图：



可以发现，图片的病变特征被明显放大，有利于后续卷积神经网络的学习

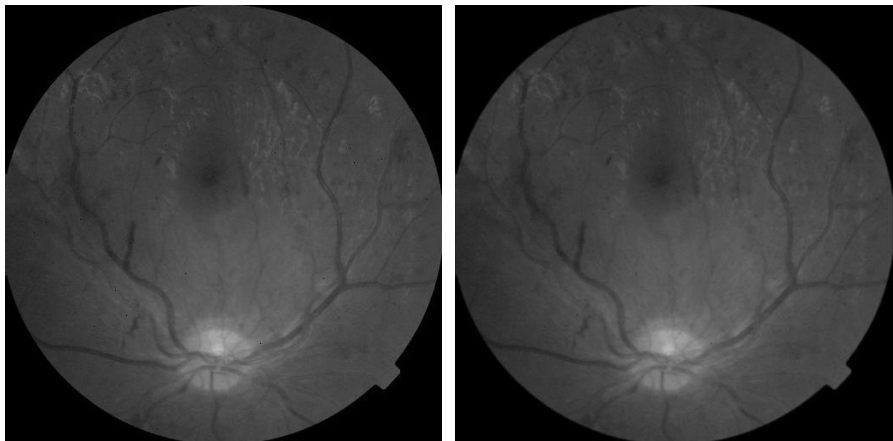
7.3 高斯滤波实现减噪

由于图像中不可避免地存在与分类任务无关，甚至会干扰分类的噪声点，因而使用高斯滤波对其进行减噪，具有一定的意义。高斯滤波器本质是一个卷积核，减噪处理的过程也是一个卷积的过程。使得每一个像素点的值，都由其本身和邻域内的其他像素值经过加权平均后得到，因而一定程度上削弱了无关像素的影响。学习时，在 jupyter notebook 中添加了 CV2 库，参考博客【5】的代码，设置滤波的参数，调用部分库函数，实现了对图像的读取、处理和保存

对单张. jpg 处理的部分代码如下所示：（在 test_image 中测试）

```
# 高斯滤波
def fil(imag, delta=0.7):
    k3 = compute(delta)
    k = k3[0]
    H = k3[1]
    k1 = (k - 1) / 2
    [a, b] = imag.shape
    k1 = int(k1)
    new1 = np.zeros((k1, b))
    new2 = np.zeros((a + (k - 1)), k1)
    imag1 = np.r_[new1, imag]
    imag1 = np.r_[imag1, new1]
    imag1 = np.c_[new2, imag1]
    imag1 = np.c_[imag1, new2]
    y = np.zeros((a, b))
    sum2 = sum(sum(H))
    for i in range(k1, (k1 + a)):
        for j in range(k1, (k1 + b)):
            y[(i - k1), (j - k1)] = relate(imag1[(i - k1):(i + k1 + 1), (j - k1):(j + k1 + 1)], H, k) / sum2
    return y
```

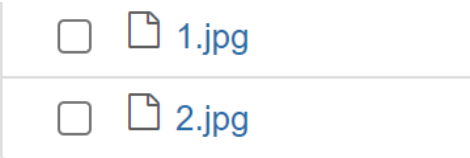
对单张图像灰度处理和使用高斯滤波处理后，结果如下所示：



原图像

减噪后图像

可以观察到，减噪后图像有了一定模糊，因而削弱了无关像素点的干扰。
但本文并没有实现高斯滤波的批量降噪处理，只能在 jupyter notebook 中找到加载和生成的两张图片，格式如下所示：

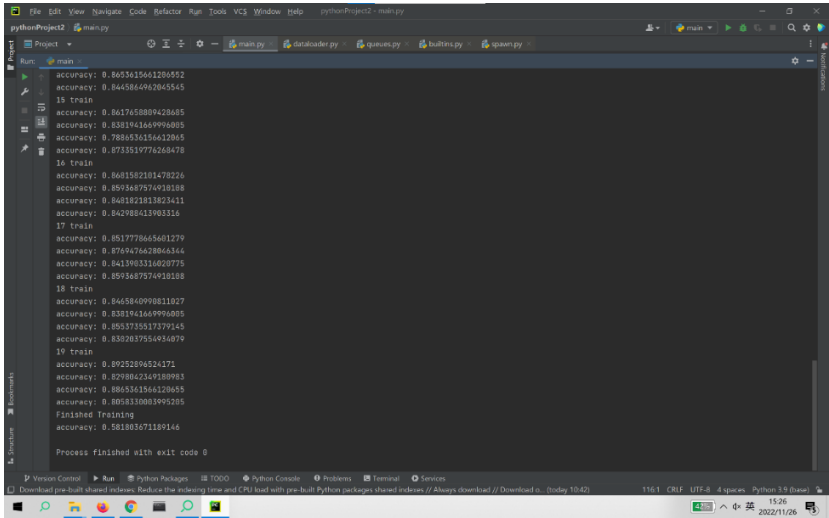


因而在优化预期中，也应当有降噪的批量处理。需要在日后不断完善。

四、结果实现

原始模型：58.18%（20 次训练）

最终经过学习过程，确定使用 CNN 网络，batch_size=20，选择最适优化器和损失函数，在未优化（数据平衡/图像增强/减噪处理）的情况下，经过 20 次训练，得到训练结果如下所示：



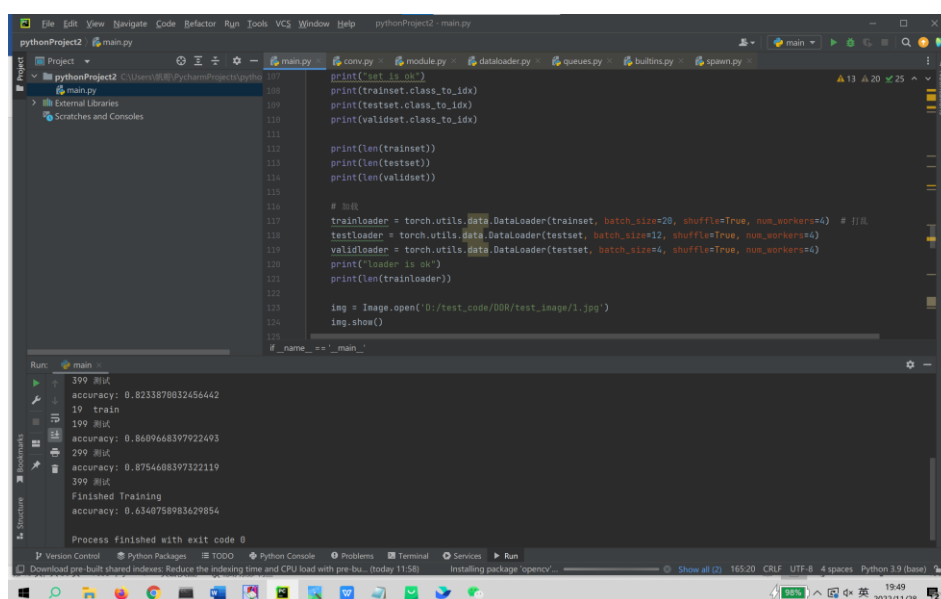
原始模型：58.50%（50 次训练）

```
accuracy: 0.8837395125848981
199 测试
accuracy: 0.8377946464242908
299 测试
accuracy: 0.8513783459848182
399 测试
accuracy: 0.8665601278465841
48 train
99 测试
accuracy: 0.8637634838194167
199 测试
accuracy: 0.8765481422293248
299 测试
accuracy: 0.8637634838194167
399 测试
accuracy: 0.8693567718737515
49 train
99 测试
accuracy: 0.8669596484218938
199 测试
accuracy: 0.8853375948861366
299 测试
accuracy: 0.8921294446664003
399 测试
accuracy: 0.8521773871354374
Finished Training
accuracy: 0.5849960095770151
```

进程已结束，退出代码为 0

优化模型：63.40%（20 次训练）

其他量保持不变，经过图像增强之后的结果如下所示：



```
pythonProject2 - main.py
main.py
101 print('test is ok')
102 print(trainset.class_to_idx)
103 print(testset.class_to_idx)
104 print(validset.class_to_idx)
105
106 print(len(trainset))
107 print(len(testset))
108 print(len(validset))
109
110 # 训练
111 trainloader = torch.utils.data.DataLoader(trainset, batch_size=20, shuffle=True, num_workers=4) # 训练
112 testloader = torch.utils.data.DataLoader(testset, batch_size=12, shuffle=True, num_workers=4)
113 validloader = torch.utils.data.DataLoader(validset, batch_size=4, shuffle=True, num_workers=4)
114 print('loader is ok')
115 print(len(trainloader))
116
117 img = Image.open('D:/test_code/00R/test_image/1.jpg')
118 img.show()
119
120 if __name__ == '__main__':
```

Run: main

```
399 测试
accuracy: 0.8233870812456442
19 train
199 测试
accuracy: 0.8609668397922493
299 测试
accuracy: 0.8754688397322119
399 测试
Finished Training
accuracy: 0.6340758983629854
Process finished with exit code 0
```

五、失败经验教训

在学习的过程中遇到了相当多的问题，本文中只提及以下两个

1. accuracy 计算

accuracy 是预测正确率，公式本身非常简单，即

$$accuracy = correct \div total$$

但由于 dataloader 对数据集进行了分批处理，因而在判断标签与预测值是否相同时，应重点关注二者的形式是否相同。

在一开始，正是因为没有注意到这件事，本人犯了个很愚蠢、但又很巧合的错误，使得第一次跑通代码的时候，一次学习训练后的预测值竟然惊人地高达 98%，甚至在 20 次训练之后就能迅速达到 100%，如下图所示

```
0 train
accuracy: 0.8468085106382979
accuracy: 0.752127659574468
accuracy: 0.8797872340425532
accuracy: 0.8936170212765957
Finished Training
accuracy: 0.9840764331210191
Process finished with exit code 0
```

训练一次后

```
19 train
accuracy: 0.9202127659574468
accuracy: 0.8319148936170213
accuracy: 0.9159574468085107
accuracy: 0.8978723404255319
Finished Training
accuracy: 1.0
Process finished with exit code 0
```

训练 20 次后

在经过自己和朋友长时间的检查后，发现是 correct 和 total 的计算方法都是错误的，因而巧合地落在 [0, 1] 区间内

错误代码如下所示：

```
70 # 全部数据训练完后输出
71 if batch == 19:
72     correct = 0
73     total = 0
74     for i, (images, lables) in enumerate(validloader):
75         #print(datas.shape)
76         total += 1
77         outputs = net(images)
78         _, predicted = torch.max(outputs.data, 1)
79         if (predicted == lables).sum().item():
80             correct += 1
81     print("accuracy:", correct / total)
```

如图所示，每批数据进入网络后，total+=1;

每当(predicted==lables).sum().item()不为0时，correct+=1

错误在于：

(1) total 本该是 validset 测试集的样本总量，但计算成了 len(validloader)，即 validset 打包的批数。

(2) 由于数据加载器的打包，分割后的 images 和 lables 都是列表类型，每个列表共存放 batch_size 个数据，放入 net 后所得的 outputs 也是列表类型，因而不能直接比较两个列表对 correct 进行计数

将 outputs 和 labels 输出可以观察到：
语句：

```
print("outputs:" outputs)
print("labels:" labels)
```

输出：

```
outputs: tensor([[ 5.8641,  4.5911,  5.2668,  7.2775,  6.1717],
                  [ 51.9502, 103.0016, -4.3745, 111.5232,  83.8173],
                  [  8.2928,  4.5837,  8.4563,  7.2680,  6.7769],
                  [  8.8714,  5.5533,  9.2919,  8.5098,  7.7877]],
               grad_fn=<AddmmBackward0>)
labels: tensor([1, 3, 2, 2])
```

分析可知，由于 validloader 的 batch_size=4，因而 labels 和 outputs 均含有 4 个数据的对应结果。而 labels 中的 [1, 3, 2, 2] 代表每个数据对应的类别，outputs 中的每个元素，都是存储元素个数为 5 的列表，对应每个预测数据分类为 0~5 的可能性，可能性最大的即为我们所预测的值

接下来输出 predicted
语句：

```
print("predicted:" predicted)
```

输出：

```
labels: tensor([0, 0, 4, 0])
predicted: tensor([2, 2, 3, 3])
```

可以观察到 labels 和 predicted 数据类型对应，都是大小为 bratch_size 的列表；可以使用 correct+=(predicted == labels).sum().item() 语句 (list1,list2).sum().item() 返回的是两个列表相同位置元素相同的个数

经过以上修正，得到了正确的 accuracy

2. WeightedRandomSampler 中的权重数组

看了很多篇文章，至今没有明白权重数组是类别的权重，还是样例的权重

六、预期优化

1. 深层网络

期待能在日后深入学习卷积神经网络后，理解各层、各参数的意义，能够跑通并且学会使用更深层、性能更好的网络

2. 增加训练次数

由于 C 盘内存限制，本次机器学习最多只能正常训练到 20 次。而这个学习次数是远远不够训练出拟合较好的参数的。希望在日后能够留足硬盘空间，也

能够学会熟练使用 Linux 系统，增加训练次数，不断优化模型

3. 数据不平衡问题

本文提出的两种解决数据不平衡问题的方法，文件夹扩增法没有达到预期准确率，权重随机取样法最终没能实现。在日后需要研究前者与理论相悖的原因，后者如何实现，以及尝试其他采样器或新方案实现样本数据平衡

4. 降噪批量处理

本次只实现了高斯滤波器的单张降噪处理，批量处理需要在日后完善

5. 分类别计数器的优化

逐个判断，复杂度过高，耗时过长。可以查询相关函数，或分类别装入 5 个 dataset 中，用 len() 计数

七、代码及参考文献

参考文献：

【1】CABNet: Category Attention Block for Imbalanced Diabetic Retinopathy Grading

【2】基于深度学习的糖尿病视网膜病变的研究_任刚 (1)

【3】<https://github.com/DoubleClass/DeepLearning/>

【4】《深度学习入门-基于 Python 的理论与实现》

【5】博客：

https://blog.csdn.net/Ibelievesunshine/article/details/104881204?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522166960845016782425119020%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=166960845016782425119020&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-2-104881204-null-null.142^v66^control,201^v3^add_ask,213^v2^t3_esquery_v2&utm_term=%E9%AB%98%E6%96%AF%E6%BB%A4%E6%B3%A2python&spm=1018.2226.3001.4187

代码：

【6】主干代码：

```
from multiprocessing import freeze_support

import torch
import torchvision
import torchvision.transforms as transforms
from torchvision.datasets import ImageFolder
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
```

```

import numpy as np
import torch.optim as optim
from PIL import Image
from PIL import ImageEnhance

import torch
import torch.nn as nn
from torch.nn import functional as F

# 构建神经网络
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self.fc1 = nn.Linear(4 * 4 * 128, 5)

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = x.view(-1, 128 * 4 * 4)
        x = self.fc1(x)
        return x

# 训练函数

```



```

def trainfunc(dataloader, validloader, net, lossfunc, optimizer):
    # 分 batch 批次训练，最终全部数据都经过一次训练
    # 每一批训练后，对参数在原有基础上进行一次更新，对新误差进行计算
    for batch, (data, target) in enumerate(dataloader):
        # 现有参数下迭代一次网络
        output = net(data)
        # forward: 将数据传入模型，前向传播求出预测的值
        # 求误差，优化，更新参数
        loss = 0 # 误差初始化为 0
        optimizer.zero_grad() # 梯度初始化为零，把 loss 关于 weight 的导数变成 0
        loss = lossfunc(output, target) # 用给定损失函数 lossfunc 求 loss
        loss.backward() # backward: 反向传播求梯度
        optimizer.step() # optimizer: 更新所有参数
        # 网络中参数将损失函数和优化器连接

    # 全部数据训练完后输出
    if batch % 100 == 99:
        correct = 0
        print(batch, "测试")
        total = len(validset)
        for i, (images, lables) in enumerate(validloader):
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            correct += (predicted == lables).sum().item()
        print("accuracy:", correct / total)

def balance_data(dataset):
    return 1

if __name__ == '__main__':

    freeze_support()
    # 图像预处理
    # compose 整合
    trans_form = transforms.Compose([
        transforms.Resize((32, 32)),
        transforms.RandomCrop(32, padding=2), # 随机位置进行裁剪
        # 32*32 (相当于没有随机)
        transforms.RandomHorizontalFlip(), # 以 0.5 的概率水平翻转给定的 PIL 图像
        transforms.ToTensor(), # 先由 HWC 转置为 CHW 格式；再转为 float 类型；最
    后，每个像素除以 255
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)), # 归
    一化

```

```

])

print("preparing data...")
# 导入下载
#
trainset0=torchvision.datasets.CIFAR10(root='D:/test_code/DDR/train/00',train=True,download=True,transform=transform)
# ImageFolder 将数据按文件夹名字分类贴上标签
trainset = ImageFolder(root='D:/test_code/DDR/train', transform=trans_form)
testset = ImageFolder(root='D:/test_code/DDR/test', transform=trans_form)
validset = ImageFolder(root='D:/test_code/DDR/valid', transform=trans_form)

print("set is ok")
print(trainset.class_to_idx)
print(testset.class_to_idx)
print(validset.class_to_idx)

print(len(trainset))
print(len(testset))
print(len(validset))

# 加载
trainloader = torch.utils.data.DataLoader(trainset, batch_size=20, shuffle=True, num_workers=4) # 打乱
testloader = torch.utils.data.DataLoader(testset, batch_size=12, shuffle=True, num_workers=4)
validloader = torch.utils.data.DataLoader(testset, batch_size=4, shuffle=True, num_workers=4)
print("loader is ok")
print(len(trainloader))

net =Net() # 构建网络

print("net is ok")
# 给定使用的损失函数和优化器
# 交叉熵损失函数
lossfunc = nn.CrossEntropyLoss()

# optimizer=torch.optim.ASGD(net.parameters(), lr=0.01, lambd=0.0001, alpha=0.75, t0=1000000.0, weight_decay=0)

# optim.

```

```

# optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9) # 优化函数(随机梯度优化方法)
# optimizer = optim.Adam(net.parameters(), lr=0.01)
# optimizer=torch.optim.Adagrad(net.parameters(),lr=0.01, lr_decay=0, weight_decay=0)
# optimizer=torch.optim.Adadelta(net.parameters(), lr=1.0, rho=0.9, eps=1e-06, weight_decay=0)
optimizer = torch.optim.RMSprop(net.parameters(), lr=0.01, alpha=0.99, eps=1e-08, weight_decay=0, momentum=0, centered=False)

# 训练网络
for epoch in range(20):
    print(epoch, "train")
    trainfunc(trainloader, validloader, net, lossfunc, optimizer)

print('Finished Training')

# 测试集
total = len(testset)
correct = 0
for i, (images, labes) in enumerate(testloader):
    outputs = net(images)
    _, predicted = torch.max(outputs.data, 1)
    correct += (predicted == labes).sum().item()
print("accuracy:", correct / total)

```

【7】单张图片增强代码：

```

img = Image.open('D:/test_code/DDR/test_image/1.jpg')
img.show()

# 对比度增强
enh_con = ImageEnhance.Contrast(img)
contrast = 1.5
img_contrasted = enh_con.enhance(contrast)
img_contrasted.show()

# 色度增强
enh_col = ImageEnhance.Color(img)
color = 1.5
image_colored = enh_col.enhance(color)

```

```

image_colored.show()

#锐度增强
enh_sha = ImageEnhance.Sharpness(img)
sharpness = 3.0
image_sharped = enh_sha.enhance(sharpness)
image_sharped.show()

img_contrasted.save("D:/test_code/DDR/test_image2/2.jpg")

```

【8】单张高斯滤波代码：

```

import cv2
import numpy as np
import math
import random

# 计算模板的大小以及模板
def compute(delta):
    k = round(3 * delta) * 2 + 1
    print('模的大小为:', k)
    H = np.zeros((k, k))
    k1 = (k - 1) / 2
    for i in range(k):
        for j in range(k):
            H[i, j] = (1 / (2 * 3.14 * (delta ** 2))) * math.exp(-(i - k1) ** 2 - (j - k1) ** 2) /
(2 * delta ** 2))
    k3 = [k, H]
    print(H)
    print(sum(sum(H)))
    return k3

# 相关
def relate(a, b, k):
    n = 0
    sum1 = np.zeros((k, k))

    for m in range(k):
        for n in range(k):
            sum1[m, n] = a[m, n] * b[m, n]
    return sum(sum(sum1))

# 高斯滤波

```

```

def fil(imag, delta=0.7):
    k3 = compute(delta)
    k = k3[0]
    H = k3[1]
    k1 = (k - 1) / 2
    [a, b] = imag.shape
    k1 = int(k1)
    new1 = np.zeros((k1, b))
    new2 = np.zeros(((a + (k - 1)), k1))
    imag1 = np.r_[new1, imag]
    imag1 = np.r_[imag1, new1]
    imag1 = np.c_[new2, imag1]
    imag1 = np.c_[imag1, new2]
    y = np.zeros((a, b))
    sum2 = sum(sum(H))
    for i in range(k1, (k1 + a)):
        for j in range(k1, (k1 + b)):
            y[(i - k1), (j - k1)] = relate(imag1[(i - k1):(i + k1 + 1), (j - k1):(j + k1 + 1)], H,
k) / sum2
    return y

```

```

# 读取图像
imag = cv2.imread('D:/test_code/DDR/test_image/1.jpg')
imag = cv2.cvtColor(imag, cv2.COLOR_BGR2GRAY)
[a, b] = imag.shape

```

```

# 滤波
imag2 = fil(imag)
imag2 = np.array(imag2, dtype='uint8')

```

```

# 显示并保存图像
cv2.imshow('1', imag)
cv2.imwrite('D:/test_code/DDR/test_image/1.jpg', imag)
cv2.imwrite('D:/test_code/DDR/test_image/2.jpg', imag2)
cv2.imshow("2", imag2)

```

【9】批量图片增强代码（针对 **train** 中类别 1，其他同理）

```

import os
import random
import augly.image as imaugs
import PIL.Image as Image

```

```

img_path = "D:/test_code/DDR/train/1" # 需要增强的图
像路径
save_path = "D:/test_code/DDR/final_train/11" # 保存路径

def augly_augmentation(aug_image):
    aug = [

        imaugs.contrast(aug_image, factor=random.uniform(1, 1.5)), # 对
        比度增强
        imaugs.grayscale(aug_image)
        # 转灰度
    ]
    return aug[0] # 从以上函数
    中随机选其一进行数据增强

for name in os.listdir(img_path):
    aug_image = Image.open(os.path.join(img_path, name))
    count = 1 # 每张图片需要增强的次数
    for i in range(count):
        image = augly_augmentation(aug_image)
        image = image.convert("RGB")
        image.save(os.path.join(save_path, name[:-4]+"_{}.jpg".format(i)))

import os
import random
import augly.image as imaugs
import PIL.Image as Image

img_path = "D:/test_code/DDR/final_train/11" # 需要增
强的图像路径
save_path = "D:/test_code/DDR/final_train/1" # 保存路径

def augly_augmentation(aug_image):
    aug = [

        imaugs.contrast(aug_image, factor=random.uniform(1, 1.5)), # 对比度增强
        imaugs.sharpen(aug_image, factor=1.5), # 锐化
        imaugs.grayscale(aug_image)
        # 转灰度
    ]
    return aug[1] # 从以上函数
    中随机选其一进行数据增强

for name in os.listdir(img_path):

```

```

aug_image = Image.open(os.path.join(img_path, name))
count = 1 # 每张图片需要增强的次数
for i in range(count):
    image = augly_augmentation(aug_image)
    image = image.convert("RGB")
    image.save(os.path.join(save_path, name[:-4]+"_{}.jpg".format(i)))

```

【10】权重随机采样代码：

```

freeze_support()
# 图像预处理
# compose 整合
trans_form = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.RandomCrop(32, padding=2), # 随机位置进行裁剪
    # 32*32 (相当于没有随机)
    transforms.RandomHorizontalFlip(), # 以 0.5 的概率水平翻转给定的 PIL 图像
    transforms.ToTensor(), # 先由 HWC 转置为 CHW 格式；再转为 float 类型；最
后，每个像素除以 255
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)), # 归
一化
])

print("preparing data...")
# 导入下载
#
trainset0=torchvision.datasets.CIFAR10(root='D:/test_code/DDR/train/00',train=True,down
load=True,transform=transform)
# ImageFolder 将数据按文件夹名字分类贴上标签
trainset = ImageFolder(root='D:/test_code/_DDR/_train', transform=trans_form)

print(len(trainset))
print("采样")

#随机采样
#样本权重 weights:
trainloader1=torch.utils.data.DataLoader(trainset, batch_size=20, shuffle=True,
num_workers=4)

# 加载

n1=[0,0,0,0,0]

```

```

for i, (datas, lables) in enumerate(trainloader1):
    for j in range(20):
        b=lables.numpy()
        if j>=len(b):
            continue
        num=b[j].item()
        n1[num]+=1
print(n1)

```

```

weight=[]
for i in range(5):
    weight.append(len(trainloader1)/n1[i])

```

```

sample_num=len(trainset)
Sampler = WeightedRandomSampler([1,1,1,1,1], sample_num, replacement=True)
print(weight)
i=0
for index in Sampler:
    print(i,index)
    i+=1

```

```

trainloader2= torch.utils.data.DataLoader(trainset, batch_size=20, shuffle=False,
num_workers=4,sampler=Sampler)
print("loader is ok")
print(len(trainloader2))

```

```

n2=[0,0,0,0,0]
for i, (datas, lables) in enumerate(trainloader2):
    for j in range(20):
        b=lables.numpy()
        print(b)
        if j>=len(b):
            continue
        num=b[j].item()
        n2[num]+=1
print(n2)

```