

**南 开 大 学**

Python语言程序设计

**深度学习大作业报告**

姓名：齐明杰

学号：2113997

专业：信息安全

目录

1. 实验题目
2. 作业要求.................................4
3. 流程分析.................................4
4. 实验工具
5. 编程工具与环境...........................4
6. Pytorch简介..............................4
7. 卷积神经网络的选择
8. 深度残差网络—Resnet50,Resnet101..........5
9. 新型小网络Convmixer......................5
10. 数据集的加载与预处理
11. 图像获取与分类............................6
12. 图像预处理................................6
13. 图像加载..................................7
14. 模型训练与预测
15. 超参数的设置..............................8
16. 损失函数和优化器选择......................9
17. 权重的加载与保存..........................9
18. 模型训练与验证............................9
19. 模型预测..................................11
20. 结果分析
21. 过拟合的出现..............................12
22. 学习率问题................................13
23. 网络深度问题..............................13
24. 批大小问题................................13
25. 解决方案
26. 调整超参数................................14
27. 网络更换..................................14
28. 数据加强..................................14
29. 成果......................................14
30. 实验总结........................................17
31. 实验题目
32. **作业要求**

参考附件论文，在DDR数据集的训练样本上，基于Pytorch框架训练一个用于图像分类的卷积神经网络，并在测试样本上完成测试。发现模型存在的问题，并提出可能的解决方案。

1. **流程分析**

由于DDR数据集已经给出，故我们的任务是写出或选择一个卷积神经网络，

在此数据集上进行训练和验证，得到合适的权重，然后再对该权重进行载入，对测试集进行验证，发现训练和测试过程中存在的问题，并且解决问题，使得最终准确率达到理想的效果。

1. 实验工具
2. **编程工具与环境**
3. 编程平台：Pycharm

2、运行环境：本实验提供了autodl在线云服务器，我将把代码传入服务器中进行训练和验证(首先在服务器上配置torch和cuda环境，安装软件包)，测试既在服务器上进行也在本机上进行。

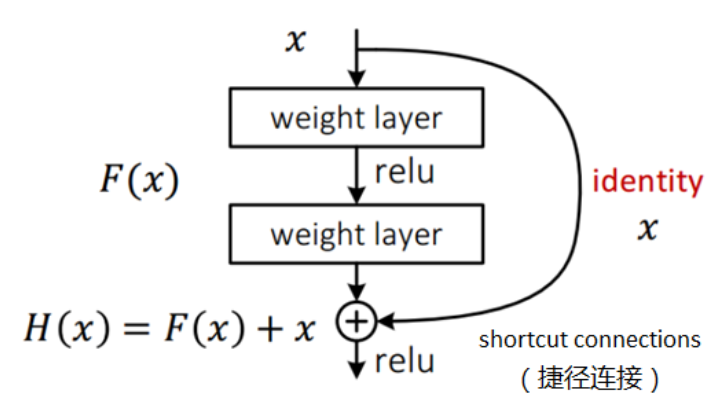
1. **Pytorch简介**

PyTorch的前身是Torch，其底层和Torch框架一样，但是使用Python重新写了很多内容，不仅更加灵活，支持动态图，而且提供了Python接口。它是一个以Python优先的深度学习框架，不仅能够实现强大的GPU加速，同时还支持动态神经网络。

本次实验是基于pytorch框架而搭建的一个深度学习代码，将使用其中诸多模块，我们不必在意pytorch中模块的实现细节，而是把它当成一个整体来使用，注重于代码大局、整体的搭建。

1. 卷积神经网络的选择
2. **深度残差网络-Resnet50,Resnet101**

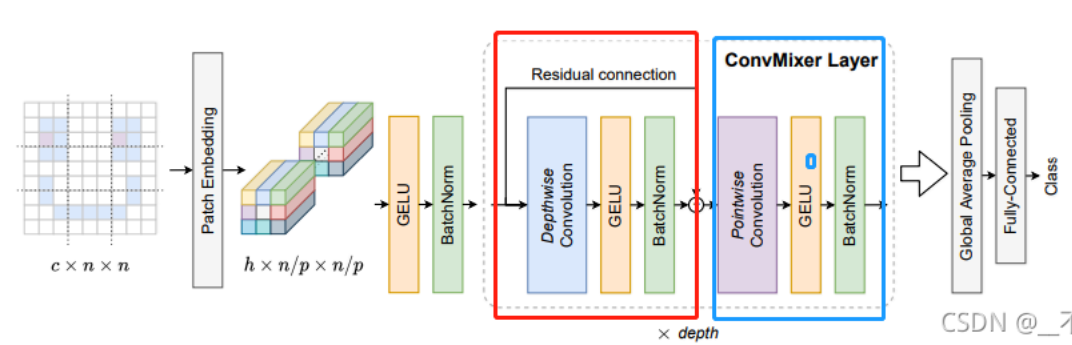
深度学习中常见的深网络问题便是梯度消失。理论上，随着网络越来越深，其“张力”将越来越大，即能够表现的特征将越来越抽象。然而，随着数据随着网络的深入，其梯度下降的问题愈发明显。Resnet采用了深度残差网络的方法（如下图），从根本上解决了深网络梯度消失的问题，在ImageNet2015一举夺冠。因此，我采用Resnet101,Resnet50分别对DDR数据集进行训练，观察结果。



其网络代码过于冗长，不在此展示，存放在model.py中

1. **新型小网络Convmixer**

Convmixer是近年来提出的网络，在各种数据集上表现良好，其采用了“ConvMixer Layer”的结构。在ConvMixer Layer 中, 使用了深度可分离卷积，GELU (高斯误差线性单元)激活函数，取得了良好的效果，网络图如下：



我除了采用Resnet50、Resnet101进行训练之外，还将采用Convmixer进行训练，进行对比，其网络代码也存放在model.py中。

1. 数据集的加载与预处理
2. **图像获取与分类**

我将数据集的加载和预处理写在了一个单独文件dataset.py中。首先导入包。

**import** os

**from** torchvision **import** transforms

**from** torchvision.datasets **import** ImageFolder

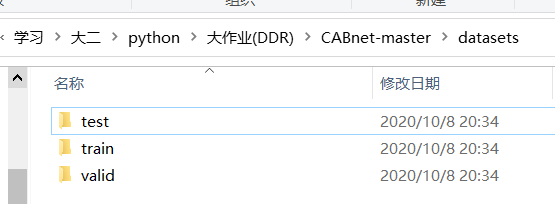
**from** torch.utils.data **import** DataLoader

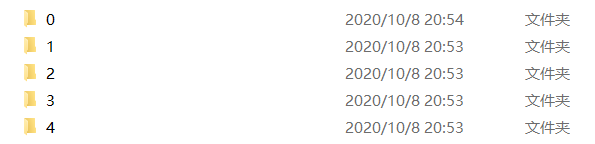
**from** os.path **import** join

然后写入DDR数据集所在的目录

data\_dir = r".\datasets"

其中，datasets文件夹下分为test数据集，train数据集，valid数据集，每个数据集下分为五类，为0，1，2，3，4，如下图





1. **图像预处理**

Torchvision中自带transforms这个对图像进行处理的函数，我们使用它

data\_transform = {

    'train':

        transforms.Compose([

            transforms.Resize([224, 224]),

            transforms.ToTensor(),

            transforms.Normalize(mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225])

        ]),

    'test':

        transforms.Compose([

            transforms.Resize([224, 224]),

            transforms.ToTensor(),

            transforms.Normalize(mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225])

        ]),

    'valid':

        transforms.Compose([

            transforms.Resize([224, 224]),

            transforms.ToTensor(),

            transforms.Normalize(mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225])

        ])

}

其中，我对train，test,valid数据集均做resize(设置大小为224\*224)，totensor（转成tensor张量），Normalize（标准化）的处理，看似并没有太大的不同(后文将指出其缺点并进行修改)

1. **图像加载**

使用torchvision.datasets中的ImageFolder进行文件打开，这个函数的强大之处在于它能自动读取train目录下的01234标签，不必再为每个文件夹说明其属于哪一类，同时，我将batch\_size设置为16。

batch\_size = 16

train\_data = ImageFolder(root = join(data\_dir, "train"), transform = data\_transform['train'])

test\_data = ImageFolder(root = join(data\_dir, "test"), transform = data\_transform['test'])

val\_data = ImageFolder(root = join(data\_dir, "valid"), transform = data\_transform['valid'])

num\_workers = min([os.cpu\_count(), batch\_size]) **if** batch\_size > 1 **else** 0

train\_data\_loader = DataLoader(train\_data, batch\_size = batch\_size, shuffle = True, num\_workers = num\_workers)

test\_data\_loader = DataLoader(test\_data, batch\_size = batch\_size, shuffle = True, num\_workers = num\_workers)

val\_data\_loader = DataLoader(val\_data, batch\_size = batch\_size, shuffle = True, num\_workers = num\_workers)

然后使用DataLoader进行数据加载(根据batch\_size的值)。值得注意的是，DataLoader有一个参数num\_workers，它几乎决定了GPU占用率的大小，我们根据cpu核心数来设定它，以此来达到最大学习速度。由于在前期我未能发现这个参数的作用，刚开始训练的速度非常慢。

1. 模型训练与预测
2. **超参数的设置**

首先将epoch(迭代轮次)，learning（学习率），batch\_size（批大小），模型参数进行设置

epoch = 50

learning = 0.001

batch\_size = 16

device = torch.device("cuda" **if** torch.cuda.is\_available() **else** "cpu")

model\_name = "Convmixer"

model = ConvMixer(n\_classes = 5, dim = 256, depth = 10).to(device)

device根据是否有cuda而选择使用gpu还是cpu，若可用gpu则使用，提高效率。我将ConvMixer的dim(维度)，depth(深度，即含ConvMixer Layer层数)设置为256，10。至于学习率，我将其设置为0.001，这是一个不高不低的选择。

对于Resnet，则有

model\_name = "Resnet50"

model = resnet50(num\_classes = 5).to(device)

以及

model\_name = "Resnet101"

model = resnet101(num\_classes = 5).to(device)

1. **损失函数和优化器选择**

损失函数常用的是交叉熵函数，均方误差函数，这里选择交叉熵函数(在分类问题中更常用)。优化器常用的有SGD(随机梯度下降), Momentum,Adam等，这里我选择Adam。

loss = torch.nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr = learning)

1. **权重的加载与保存**

训练完的权重，如果不保存，那就白训练了，因此我将其保存成pth文件。

如果权重存在，那么先进行加载：

**if** exists(model\_name + ".pth"):

    model.load\_state\_dict(torch.load(model\_name + ".pth"))

我对权重的保存机制进行了一些设定。首先，每轮训练都会进行一次验证，生成一个valid\_accuracy，我将其存放在一个列表valid\_accur\_all中，并将其写入文件Acc\_Resnet50.txt(如果使用的模型是Resnet50的话)里，每行一个Acc。加载模型的同时读取这个文本，每个epoch后，如果验证准确率增大，则对权重进行保存，代码如下：

**if** valid\_accur\_all[-1] == max(valid\_accur\_all) \

**or** (i + 1) % 12 == 0 **and** max(valid\_accur\_all) - valid\_accur\_all[-1] <= 0.03:

  torch.save(model.state\_dict(), model\_name + ".pth")

其中，我后期对条件进行了优化，即若训练了12轮，且acc比最大值小0.03以内，则进行保存，因为经常出现acc到了瓶颈导致训练几十轮都不保存权重的情况(白训练了)。

1. **模型训练与验证**

我将训练和验证封装成了函数，存在train.py中，以使代码简洁，训练代码如下：

**def** train(model, optimizer, loss, device):

    train\_num, train\_total\_loss, train\_total\_accuracy = 0.0, 0.0, 0.0

    model.to(device)

    model.train()  # 将模型设置成 训练模式

    train\_bar = tqdm(train\_data\_loader, desc = "train")  # 用于进度条显示，没啥实际用处

**for** step, data **in** enumerate(train\_bar):  # 开始迭代跑， enumerate将训练集分为 step是序号，data是数据

        img, target = data  # 将data 分为 img图片，target标签

        img, target = img.to(device), target.to(device)

        optimizer.zero\_grad()  # 清空历史梯度

        outputs = model(img)  # 将图片打入网络进行训练,outputs是输出的结果

        loss\_fn = loss(outputs, target)  # 计算神经网络输出的结果outputs与图片真实标签target的差别-这就是我们通常情况下称为的损失

        outputs = torch.argmax(outputs, 1)  # 最大的值就是我们预测的结果 求最大值

        train\_total\_loss += loss\_fn.item() \* img.size(0)  # 将所有损失的绝对值加起来

        train\_total\_accuracy += torch.eq(outputs, target).sum().item()  # 求训练集的准确率

        train\_num += img.size(0)

        loss\_fn.backward()  # 神经网络反向传播

        optimizer.step()  # 梯度优化 用上面的adam优化

**return** train\_total\_loss / train\_num, train\_total\_accuracy / train\_num

这里对其进行详细的解读。

首先，model.train()将模型设置成训练模式，这是关键的第一步。然后我使用了tqdm模块，用于将训练进度可视化，同时将数据装载器载入训练数据，开始训练。首先将数据分为图片，标签，将其送入gpu中(很重要，否则速度很慢)，然后清空历史梯度(必须)。之后便是计算损失函数的值，然后计入loss中。

torch.eq(outputs, target).sum().item() 这句代码的含义便是比较网络输出和标签值，生成了一个由“True”“False”组成的等维等大小数组(和Numpy的广播功能极为相似)，然后对其进行相加，得到的值即为True的个数，即输出值和标签值相等的个数，之后除以总个数得到最终Acc。之后进行反向传播和梯度优化。

模型验证代码如下：

**def** evaluate(model, loss, device, data\_name):

    data\_num, data\_loss, data\_accuracy = 0.0, 0.0, 0.0

    model.to(device)

    model.eval()

    loader = test\_data\_loader **if** data\_name == "test" **else** val\_data\_loader

    data\_bar = tqdm(loader, desc = data\_name)

    with torch.no\_grad():  # 清空历史梯度，进行测试  与训练最大的区别是测试过程中取消了反向传播

**for** data **in** data\_bar:

            img, target = data

            img, target = img.to(device), target.to(device)

            outputs = model(img)

            loss\_fn = loss(outputs, target)

            outputs = torch.argmax(outputs, 1)

            data\_loss += loss\_fn.item() \* img.size(0)

            data\_accuracy += torch.eq(outputs, target).sum().item()

            data\_num += img.size(0)

**return** data\_loss / data\_num, data\_accuracy / data\_num

首先model.eval()将其转化为评估模式，之后和train的流程基本一致，只是取消了反向传播和梯度优化。由于我将装载的数据集名称做成了参数，故之后测试模型时也将使用此函数(参数data\_name将由”valid”修改成”test”)

训练和验证都完成后，将Acc写入文件：

valid\_accur\_all.append(round(valid\_accuracy, 5))

with open(file\_name, 'a') as file:

    file.write("%.3f" % valid\_accuracy + '\n')

1. **模型预测**

预测只需进行一轮在测试集上的输出，为了方便切换模型，我将其写入单独的文件predict.py，具体如下：

**from** os.path **import** exists

**from** model **import** \*

**import** torch

**from** train **import** evaluate

loss = torch.nn.CrossEntropyLoss()

device = torch.device("cuda" **if** torch.cuda.is\_available() **else** "cpu")

model\_name = "Convmixer"

model = ConvMixer(n\_classes = 5, dim = 256, depth = 10).to(device)

**if** exists(model\_name + ".pth"):

    model.load\_state\_dict(torch.load(model\_name + ".pth"))

**def** test\_predict(model, loss, device, model\_name):

**print**("--------------- Test Start ---------------")

**if** exists(model\_name + ".pth"):

**print**("The test is using saved weight from model " + model\_name + "!")

    test\_loss, test\_accuracy = evaluate(model, loss, device, "test")

**print**("test-Loss：{:.5f} , test-accuracy：{:.5f}".format(test\_loss, test\_accuracy))

**print**("--------------- Test End ---------------")

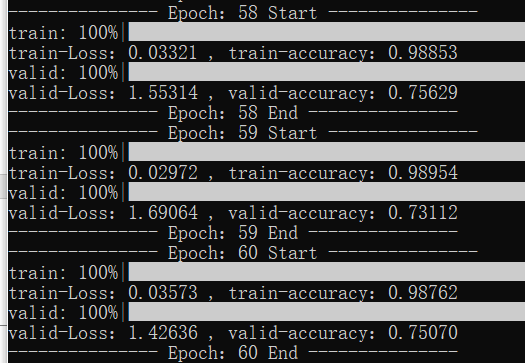
**if** \_\_name\_\_ == "\_\_main\_\_":

    test\_predict(model, loss, device, model\_name)

首先是设置需要测试的模型，然后加载权重，最后对其预测输出Acc即可。

1. 结果分析
2. **过拟合的出现**

Resnet101训练过程中部分截图：



这是Resnet101的训练部分。很明显，trainacc已经达到了0.98，但是validacc仍然在0.7-0.75之间徘徊，出现了过拟合的问题。对这个模型进行测试，结果更是令人大跌眼镜：



Resnet50和Convmixer的训练是根据Resnet101的结果调整超参数，优化完进行的，因此效果比Resnet101好许多，将在后文叙述。

1. **学习率问题**

为何Resnet101如此快的进入了过拟合呢？原因之一便是学习率的设置问题。

学习率0.001看似是一个很小的值，其实不然。在trainacc达到0.85甚至0.9后，学习率应该再小一点，诸如0.0003，否则将出现快速进入过拟合的情况。

1. **网络深度问题**

除了学习率的问题，Resnet101和Resnet50以及Convmixer的重要区别在于其网络的深度。Resnet101的深度更大，能提炼的抽象数据更多，但这是一个只有5类的分类问题，过深的网络不仅学习效率低，而且更容易出现过拟合。因此，后文将使用Resnet50来代替Resnet101

1. **批大小问题**

Batch\_size的设置也是一个重要的问题。增大batch\_size的好处有三点：

1、内存的利用率提高了，大矩阵乘法的并行化效率提高。

2、跑完一次epoch(全数据集)所需迭代次数减少，处理速度进一步加快。

3、一定范围内，batchsize越大，其确定的下降方向就越准，引起训练震荡越小。

因此，我将batch\_size从16修改成了64(服务器能接受的值)

1. 解决方案
2. **调整超参数**

在利用Resnet50和Convmixer进行训练时，我进行了如下调整：

1. 学习率在trainacc增长到0.8后更换为0.0003(**学习率衰减**)
2. Batch\_size由16更改为64
3. **网络更换**

如上所述，Resnet101更换为了Resnet50，同时和Convmixer同步进行训练

1. **数据加强**

数据增强是对抗过拟合问题的一个重要方法。常用的数据加强如对图片的裁剪，缩放，镜像，旋转等。上文中，train的数据处理跟test和valid一样，缺少了训练数据的多样性，因此我采用如下代码进行加强：

'train':

        transforms.Compose([

            transforms.Resize([224, 224]),

            transforms.RandomHorizontalFlip(p = 0.5),

            transforms.RandomVerticalFlip(p = 0.5),

            transforms.RandomRotation(90),

            transforms.ToTensor(),

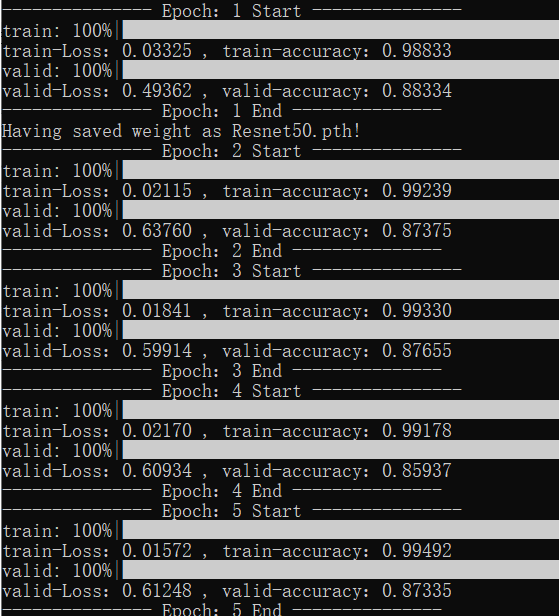
            transforms.Normalize(mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225])

        ]),

RandomHorizontalFlip是对图片进行随机水平翻转，RandomVerticalFlip是对图片进行随机垂直翻转，RandomRotation是对图片进行-90°-90°之间的随机旋转，以此来达到泛化数据的目的。

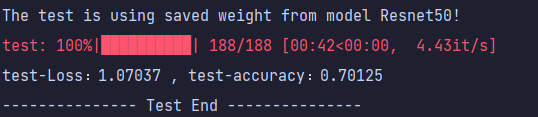
1. **成果**

Resnet50的训练结果：

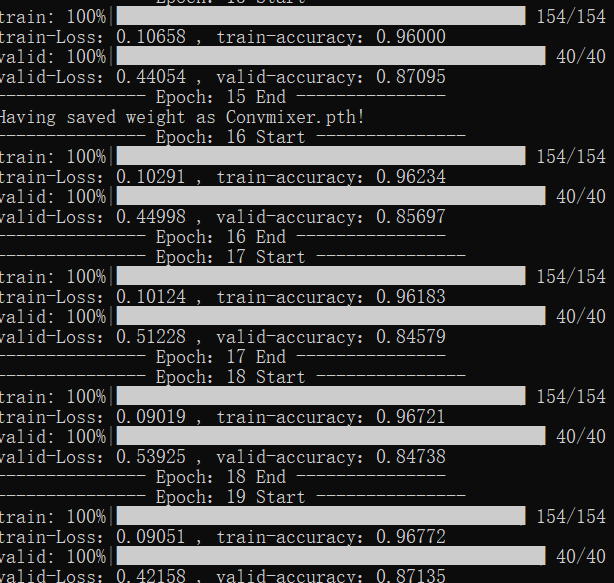


Validacc达到了0.87，其相较于Resnet101提升巨大。

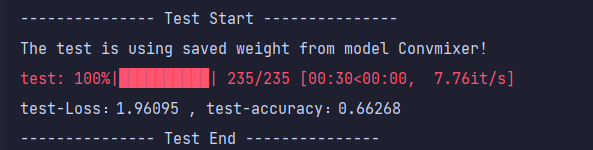
下面来测试一下testacc。Resnet50测试结果：



Convmixer的训练结果：

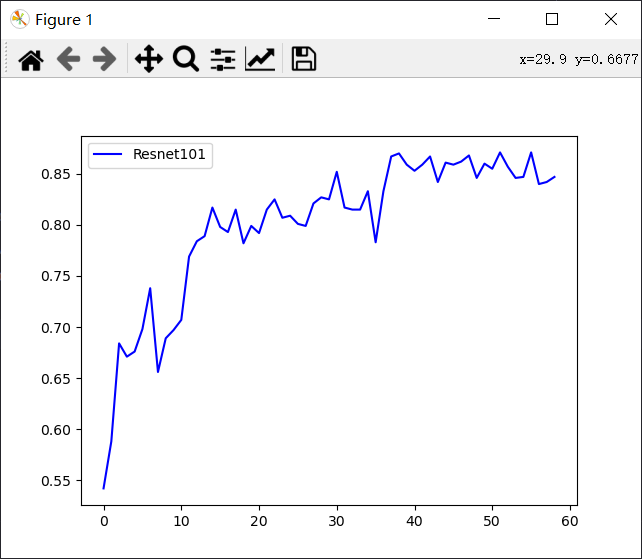


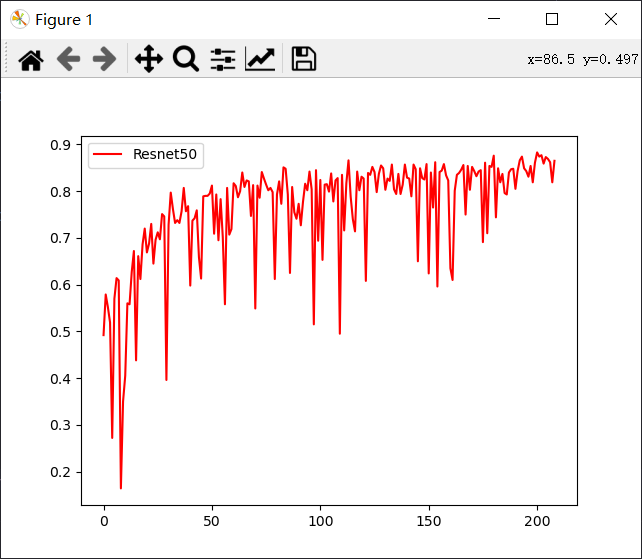
其跟Resnet50的验证结果相差不大，其测试结果：

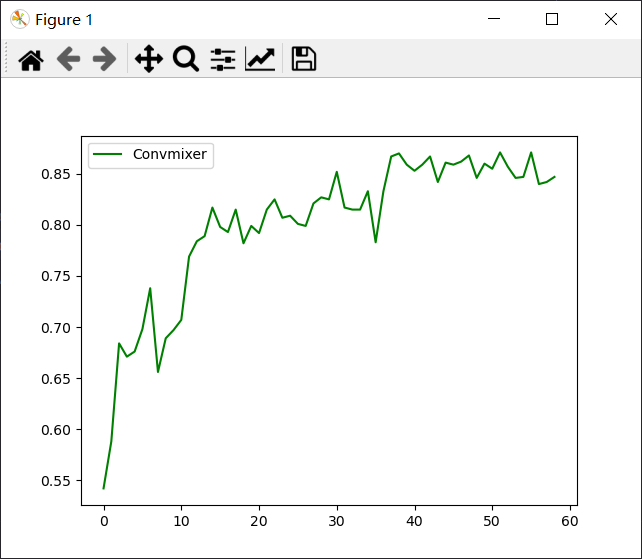


可以看到，**Convmixer的测试准确率为0.66**，这对于小网络来说已经不错，而**Resnet50更是达到了0.70的准确率**。

附图：验证集(valid)准确率上升图(y:valid\_acc, x:迭代次数)







1. 实验总结

我先是采用Resnet101进行训练，发现效果并不是很好，接着查找资料，对超参数，模型网络等进行深入分析后，选择了Resnet50和Convmixer进行训练，均收到了良好的效果，这些都是从实践摸索中所收获的。最终达到了**准确率0.70**收获到了满意的结果。