实验报告

181840013 陈超

1 基本配置

1. 语言：Pytorch

2. GPU：Kaggle 平台提供的 NVIDIA Tesla P100

3. 数据集：fer2013 from "Challenges in Representation Learning: Facial Expression Recognition Challenge"

(食用本篇实验报告前建议先阅读第五部分开一下胃)

2 代码框架

代码分为三个部分：数据获取及预处理，网络构建，训练及结果输出。

2.1 数据获取及预处理

该部分代码为 load and preprocess data 注释。首先使用 pandas 库读取 train.csv 并将其进行转化为高维 tensor，tensor 的第一维对应了每一组图像及其 label，其中图像是一个三维 tensor，大小，每一维分别对应了 channel、长、宽，label 则是一个 0 6 的整数。下图即为处理后的四维 tensor 的第一个元素：

```
[tensor([[[ 70.,  80.,  82.,  ...,  52.,  43.,  41.],
[ 65.,  61.,  58.,  ...,  56.,  52.,  44.],
[ 50.,  43.,  54.,  ...,  49.,  56.,  47.],
...,
[ 91.,  65.,  42.,  ...,  72.,  56.,  43.],
[ 77.,  82.,  79.,  ..., 105.,  70.,  46.],
[ 77.,  72.,  84.,  ..., 106., 109.,  82.]]]), 0]
```

同时，鉴于原数据中存在较多噪声，我对数据集进行了一定的筛选，丢弃了那些像素中含有一半以上相同像素值的图像。这样能过滤掉 35 张图片，对这些图片一一观察后，可以确认没有误伤到正常的图片。

由于原数据集中并没有验证集，故这里手动将 train.csv 中数据 shuffle 后，取 90% 数据作为训练集，10% 作为验证集。使用 data 库的 Dataloader 函数得到 train_iter 和 valid_iter，batch_size 选取为 512。关于 batch_size 的选取问题，下面还会有讨论。我尝试了使用 torchvision.transforms 对原数据进行增强，例如 RandomSizeCrop 随即裁剪，RandomHorizontalFlip 水平翻转，ColorJitter 改变对比度等 (具体可以参考代码的 version 17)，但是发现并没有什么卵用。虽然很多同学都跟我讲数据增强有用，但是在我看来，网络的初始几个卷积层完全可以替代除裁剪外的其他数据增强，这一点我在 4.5 会具体说明。因此在最后的实现中也就没有采用。

2.2 网络构建

该部分代码为 construct net 注释。在尝试了 github 和 kaggle 上包括 ResNet，GoogLeNet，DenseNet，VGG 等在内的多种经典架后 [2,3,4,5]，我选择了 VGG 作为主要模板，最终采用的网络如下所示 (水个页数)：

Layer (type)	Output Shape	Param #
Conv2d−1	[−1, 64, 48, 48]	640
ReLU−2	[−1, 64, 48, 48]	0
BatchNorm2d−3	[−1, 64, 48, 48]	128
Conv2d−4	[−1, 64, 48, 48]	36,928
ReLU−5	[−1, 64, 48, 48]	0
BatchNorm2d−6	[−1, 64, 48, 48]	128
Conv2d−7	[−1, 64, 48, 48]	36,928
ReLU−8	[−1, 64, 48, 48]	0
BatchNorm2d−9	[−1, 64, 48, 48]	128
MaxPool2d−10	[−1, 64, 24, 24]	0
Dropout−11	[−1, 64, 24, 24]	0
Conv2d−12	[−1, 128, 24, 24]	73,856
ReLU−13	[−1, 128, 24, 24]	0

BatchNorm2d−14	[−1, 128, 24, 24]	256
Conv2d−15	[−1, 128, 24, 24]	147,584
ReLU−16	[−1, 128, 24, 24]	0
BatchNorm2d−17	[−1, 128, 24, 24]	256
Conv2d−18	[−1, 128, 24, 24]	147,584
ReLU−19	[−1, 128, 24, 24]	0
BatchNorm2d−20	[−1, 128, 24, 24]	256
MaxPool2d−21	[−1, 128, 12, 12]	0
Dropout−22	[−1, 128, 12, 12]	0
Conv2d−23	[−1, 128, 12, 12]	147,584
ReLU−24	[−1, 128, 12, 12]	0
BatchNorm2d−25	[−1, 128, 12, 12]	256
Conv2d−26	[−1, 128, 12, 12]	147,584
ReLU−27	[−1, 128, 12, 12]	0
BatchNorm2d−28	[−1, 128, 12, 12]	256
Conv2d−29	[−1, 128, 12, 12]	147,584
ReLU−30	[−1, 128, 12, 12]	0
BatchNorm2d−31	[−1, 128, 12, 12]	256
MaxPool2d−32	[−1, 128, 6, 6]	0
Dropout−33	[−1, 128, 6, 6]	0
Conv2d−34	[−1, 128, 6, 6]	147,584
ReLU−35	[−1, 128, 6, 6]	0
BatchNorm2d−36	[−1, 128, 6, 6]	256
Conv2d−37	[−1, 128, 6, 6]	147,584
ReLU−38	[−1, 128, 6, 6]	0
BatchNorm2d−39	[−1, 128, 6, 6]	256
Conv2d−40	[−1, 128, 6, 6]	147,584
ReLU−41	[−1, 128, 6, 6]	0
BatchNorm2d−42	[−1, 128, 6, 6]	256
MaxPool2d−43	[−1, 128, 3, 3]	0
Dropout−44	[−1, 128, 3, 3]	0
Conv2d−45	[−1, 256, 3, 3]	295,168
ReLU−46	[−1, 256, 3, 3]	0

BatchNorm2d−47	[−1, 256, 3, 3]	512
Conv2d−48	[−1, 256, 3, 3]	590,080
ReLU−49	[−1, 256, 3, 3]	0
BatchNorm2d−50	[−1, 256, 3, 3]	512
Conv2d−51	[−1, 256, 3, 3]	590,080
ReLU−52	[−1, 256, 3, 3]	0
BatchNorm2d−53	[−1, 256, 3, 3]	512
MaxPool2d−54	[−1, 256, 1, 1]	0
Dropout−55	[−1, 256, 1, 1]	0
Flatten−56	[−1, 256]	0
Linear−57	[−1, 1024]	263,168
ReLU−58	[−1, 1024]	0
Dropout−59	[−1, 1024]	0
Linear−60	[−1, 1024]	1,049,600
ReLU−61	[−1, 1024]	0
Dropout−62	[−1, 1024]	0
Linear−63	[−1, 7]	7,175

Total params: 4,128,519
Trainable params: 4,128,519
Non−trainable params: 0

Input size (MB): 0.01
Forward/backward pass size (MB): 17.91
Params size (MB): 15.75
Estimated Total Size (MB): 33.67

该网络由五个 stage 及最后的三个全连接层组成，每个 stage 结构除了输出的 channel 数逐级增加外基本相同，均由三个 block 和最后的池化加 dropout 组成，而每个 block 又由顺序连接的 $kernel = 3 \times 3, padding = 1$ 卷积层，ReLU 激活函数层和批归一化层组成。

这里有几个地方需要多废话几句：

1. 原始 VGG 网络中并没有加入批归一化，这里加入了批归一化是因

为在训练过程中观察到了过拟合的情况，加入批归一化可以一定程度上缓解这一问题。另一个好处是批归一化可以加快收敛速度，在相同的训练轮数下可以用更小的学习率。

2. 在每个 stage 的最后和全连接层之间加入 dropout 也是为了防止过拟合，尤其是最后的三个全连接层，加入 dropout 与否有很大的影响。dropout 的参数 0.5 则是在测试了多个不同系数后选择的最优系数。

3. 关于卷积核的大小。几乎所有框架都采用的是 3×3 的卷积核，并且也有研究表明这个是所有大小卷积核中表现最好的，所以我这里也就直接沿用了这种卷积核。

4. 我尝试过把 MaxPool2d 换成 conv2d，但是效果并不是很好，我猜可能是因为前者稳定性更高。

5. 一开始我一直执着于增加卷积层和全连接层，试图提高正确率，但很多时候适得其反。终于有一次，我把最后的一个全连接层改成了三个全连接层，正确率上升了 3%，这大抵是我无数次改网络结构的尝试中唯一成功的一次。

2.3 训练及结果输出

该部分代码为 train，fine-tune 和 predict 注释。这一部分参考了从《Dive into Deep Learning》一书的网站：http://www.d2l.ai/下载的 d2l-en 代码包[1](我承认我馋它代码，我下贱)。

我主要使用了其中的 train_ch6 函数及其所调用的函数。在提交的代码中，该函数能够在 GPU 上完成对网络的训练，并且利用在训练过程中记录 loss，train_acc 和 valid_acc 绘制整个训练过程的图像。该函数有以下可选

的参数：训练轮数 train_epoches，优化器 opt 及其相应参数，同时，鉴于微调和训练代码大致相同，我把微调和训练的代码整合到了一起，用一个 tune 参量指示当前是训练还是微调。在完成训练后会根据训练中过程中记录的数据绘制图像。

训练完成后，会选出训练过程验证集正确率最高的一个模型 (Net.param)，并对其进行微调，学习率采用一个更小的值。同样，最后选取微调过程中验证集正确率最高的模型 (Net (fine-tune).param)。

最后，使用微调后的模型对 test.csv 进行 inference 预测即可。

我还尝试过徐满杰同学的一种憨批训练方法，就是每轮训练过后，如果新的验证集正确率小于更新前的，就 reload 回之前的 (我还考虑了其他很多东西，包括如果连续出现多次 reload 就强制更新参数等，具体可以参考代码的 version 20)。但是实际用起来效果一言难尽，于是我果断放弃了这个憨憨方法 (当事人表示非常后悔.jpg)。

3 模型训练

3.1 超参数选取

如前所述，训练的超参数主要包括 batch_size，train_lr，tune_lr，train_epoches，tune_epoches 和 optimizer，下面将一一进行说明。

1. batch_size: 正常来说，batch_size 不能小于 64，否则会导致批归一化和梯度下降时对噪声敏感而出现较大波动。batch_size 也会影响收敛速度，batch_size 越大，同等学习率下收敛速度越慢，所以改变 batch_size 的同时也要相应改变学习率。一般情况下，batch_size 越大越好，直到塞满显存为止。这里我选取的数值为 256。

2. optimizer: 对于训练过程，因为是从头开始的，所以使用 Adam 方法可以加速收敛，对应的 train_lr 在尝试过后选为 0.001。对于微调过程，为了稳定性，选取 SGD，对应的 tune_lr 在尝试过后选为 0.0001.

3. train_epoches&tune_epoches: 观察训练过程可以发现，在 40 轮左右，验证集正确率就达到了瓶颈，之后开始来回震荡。于是我把 train_epoches 选为 50，这样既不至于过拟合，又可以让模型训练过程中达到一个较高的验证集正确率。至于 train_epoches，50 应该差不多了。

3.2 训练及微调结果

1. 经过 50 轮训练，训练集正确率为 0.779，验证集正确率为 0.658。选取整个训练过程中验证集正确率最高的模型进行下面的微调。

2. 经过 50 轮微调，训练集正确率为 0.777，验证集正确率为 0.662。最后保留整个微调过程中验证集正确率最高的模型，其验证集正确率为 0.666，该模型保存在./Net(upload).param。

3. 提交到 Kaggle 平台，测试集正确率为 0.652。

4. 整个训练过程详见 version 19。

4 结果分析

4.1 显著性图

显著性图的原理其实很简单，就是对输入图像的某个像素改变一个小量，然后观察这个微小改动对最后一层输出结果的影响。影响越大，说明这个像素对模型就越重要。对输入图像的每个像素都重复一遍上面的操作，我们就可以得到一张显著性图。

该部分代码为 salientcy map 注释。具体实现，只需要将图像作为 Variable 变量传入网络正向计算，之后对最后一个全连接层输出的长度为 7 的一维 tensor 进行反向传播，得到的与输入图像大小相同的梯度矩阵的绝对值即为显著性图 [6]。

鉴于不可能把所有图像的显著性图都画出来，我在验证集中挑选了两类图像进行绘制。第一类是被正确分类的，每种表情随机挑选 5 张，一共 7 组，保存在./Figure/Saliency map/Correctly classified。第二类是被错误分类的，我挑选了其中分类错误率最高的 5 类，每一类随机挑选 5 张，一共 7 组，保存在./Figure/Saliency map/Wrongly classified，每张图像的文件名 A(B).png 表示原本属于类别 A 的图像被分为类别 B。

下面展示了标签为 Happy，Surprise 和 Disgust 的被正确分类的图片：

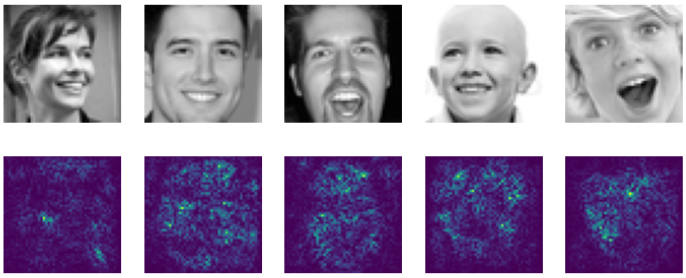

图 1: Happy

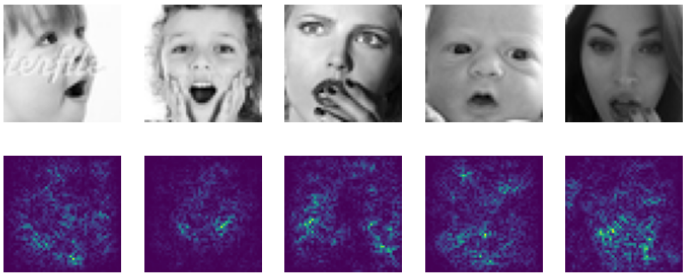

图 2: Surprise

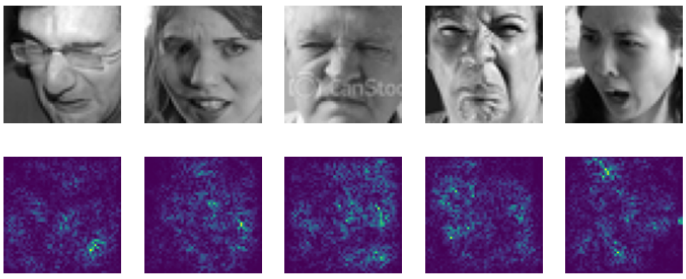

图 3: Disgust

这三张图可以大致反映出我总结的几点模型分类时注重的方面：

- 眉毛: Happy 第五张, Disgust 第五张
- 面部肌肉 (以嘴角和脸颊为主，这应该是最重要的): Digust 第一、二、三张, Happy 第一、三、五张, Surprise 第四张
- 手部动作: Surprise 第二、五张

这个结果还是有些出乎我的意料的，我原本以为眼睛会起重要作用，结果却几乎没有显著性图峰值落在眼睛上。这个仔细一想也是有道理的，毕竟原图像只有 48×48，眼睛所占据的像素可能就只有十几个像素甚至更少，这其中所含的信息自然也就很少。相比之下，上面我所列出的三个方面所占区域及其包含的信息则要大很多。

当然，也有相当一部分显著性图看不出任何规律，比如单纯描一个边框啥的 (或许是我太愚蠢了，理解不了神经网络的深邃思想)，具体可以看./Figure/Saliency map 文件夹中的其他图像。以及，我也看不出错误分类图像和正确分类图像的显著性图有什么区别，很多错误分类图像的显著性图看着也挺有道理的，可它就是分类错了。

4.2 梯度上升方法观察卷积层激活特性

梯度上升也是一个相当精妙的方法。对于某个特定的卷积层，可以用其输出矩阵的元素之和表示其激活情况，和越大表示该卷积层的激活越强。那么如果我们能找到这样一个输入图像，最大化该卷积层的输出元素之和，那么这也就可以理解为该卷积层最为敏感的一种图像。

该部分代码为 gradient ascent 注释。具体实现，可以先随机初始化一个图像，传入网络计算得到特定卷积层的输出矩阵元素和 sum，因为要最大化这个值，所以把 loss 设为-sum，然后再对 loss 做反向传播，更新输入图像即可，当然，更新时需要注意图像不能超过 0~255 的范围。之后重复上述过程，直至收敛。需要稍微注意一下的是学习率的设定，因为 sum 的取值范围较大，一开始需要一个较大的学习率来加快收敛，后期为了则需要一个较小的学习率来取得一个较好的极值。为此，我对其设置了一个 0.99 的衰减率。

由于整个模型中的 channel 数实在太多，显然不可能对所有 channel 都做一遍梯度上升。所以我挑选了每个卷积层的第一个 channel(能大致说明问

题就行)，一共 5 个 stage，每个 stage 包括 MaxPooling 在内又有 4 个卷积层，一共就有 20 个卷积层。对这些卷积层的第一个 channel 的梯度上升结果保存在./Figure/Gradient ascent。下面我挑了几个效果较好的展示一下：

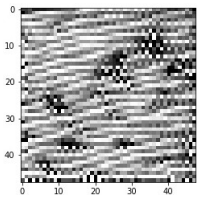

图 4: stage1 layer3

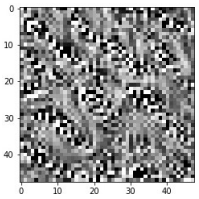

图 5: stage2 layer4

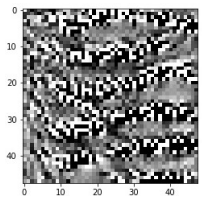

图 6: stage4 layer3

上面呈现的三张图分散在整个网络的不同位置，且都有比较明显的纹理，效果应该是不错的。./Figure/Gradient ascent 文件夹中还有一些看着没有啥规律的图像，这可能是因为我理解不了网络它自己深邃的思想，也有可能是因为网络没有得到充分训练。

4.3 观察不同卷积层的输出

该部分代码为 filter output 注释。这个实现很简单，只要把原始图像传入然后截取特定层的输出即可。同样，我挑选了每个卷积层的第一个 channel 进行绘制，共 20 张图像，保存在./Figure/Gradient ascent。下面展示了三个卷积层的输出，所选卷积层与上面梯度上升挑选的一致。

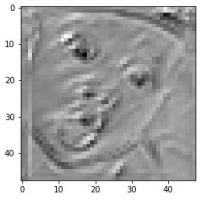

图 7: stage1 layer3

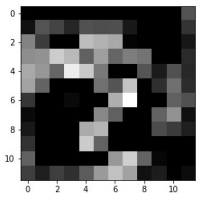

图 8: stage2 layer4

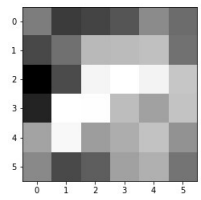

图 9: stage4 layer3

很显然的是，随着卷积层的深入，图像变得越来越模糊，从第 2 个 stage 的第 4 个卷积层开始就看不清人脸了。我本来打算再根据这些图像分析一下网络更注意图像的哪些地方，但却发现了一个有趣的现象：

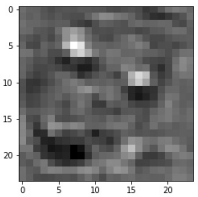

图 10: stage1 layer3

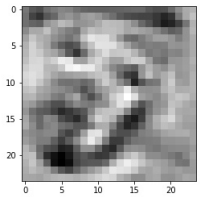

图 11: stage2 layer4

可以看到，这两张图像就好像灰度倒置了一样。前一张图激活峰值落在了眼睛上，而后一张图的激活峰值则落在了面部肌肉上。所以，在整个训练过程中脸上的每一个部位应该是都会被注意到的，也就是说神经网络确实认真分析了整个面部，这点我在 4.4 种还会再次提到。至于最后的显著性图上为什么很少有峰值落在眼睛上，我想可能是在经过多层卷积后眼睛部分权重较少的缘故。

4.4 分析模型对表情的判断方式

首先，我绘制了该模型的 7×7 模糊矩阵，该部分代码为 confusion matrix 注释。结果如下图所示：

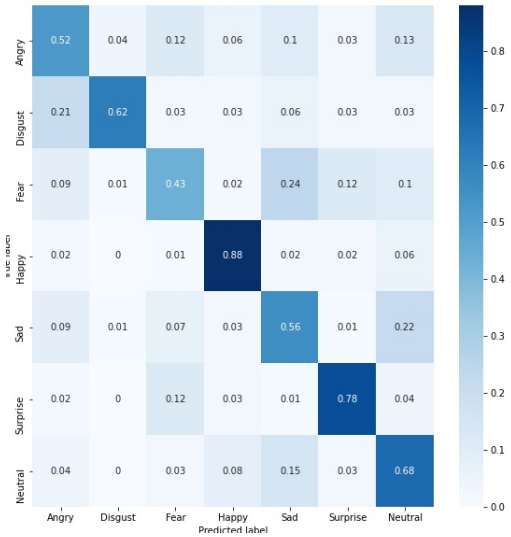

图 12: Confusion matrix

7 种类别中，Happy 分类准确率最高，Fear 分类准确率最低。观察原图像我们不难发现，Happy 往往是最容易区分的，只要依靠大片的面部肌肉就可以了。被标注为 Fear 的表情则有些模棱两可，甚至根本就是错的，很多时候人也有些难区分，很容易和 Disgut 及 Angry 搞混。而且 Fear，Angry 和 Disgust 往往需要依靠眼睛来进行区分，这对于像素较低的图片来说是比较困难的。

下面我使用 lime 包进行了一些分析，该部分代码为 lime 注释。Lime 全称是 Local Interpretable Model-agnostic Explanations，出自 Marco Tulio Ribeiro2016 年发表的论文《"Why Should I Trust You?" Explaining the Predictions of Any Classifier》[8]。该论文提供了一种解释分类器的方法，对任意形式的分类器可以分析出模型到底学习到了什么特征。原理也比较好理解，第一步需要对输入图像进行随机扰动，简单说来就是把图像分成若干互不重叠的随机区域，这些区域我们称为超像素。然后随机选择这些超像素是保留还是去掉 (置为 0)，来生成扰动后的图像。具体如下图所示 [9]：

图 13: Hyperpixel

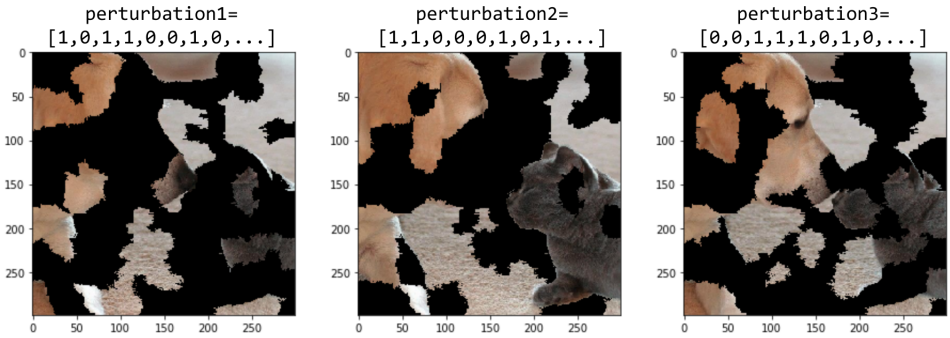

图 14: Perturbation

接着我们把扰动后的图像传入模型，得到分类结果，之后再根据特定的方法 (比如余弦距离) 来计算扰动的大小也就是权重。最后对权重和扰动后的分类结果做一个拟合，就可以得到对每个分类结果，每个超像素所起的作用有多大，正值表示支持该分类，负值表示反对该分类。

具体到 lime 包的使用，我调 bug 真的调到去世了，短短二十几行代码几乎每行都会出 bug，主要问题是出在 lime 包只支持 RGB 三通道图像的分析。explain_instance 函数会自动把传入的图像以三通道形式传入网络，但是我的网络是单通道的，我就需要找到 explain_instance 是什么时候把图像传到网络里面的，截取出来改成单通道的。找了半天终于找到了，改完又出了一堆 tensor 形状的错误，当它能运行的时候我真的泪目了。不过不得不说，整个调试过程让我深刻理解了 lime 的运行过程，也实实在在看到了 lime 对原图像所做的扰动。

有几个参数需要说明一下，top_labels 表示取多少种类别进行分析，num_samples 表示生成多少扰动后的图像，explanation.top_labels[0] 表示对最倾向的分类进行分析，positive_only=False 表示支持的区域将用绿色标出，反对区域将用红色标出，hide_rest=False 表示整张图片会被完整显示。当然还有其他很多参数，比如距离度量函数 distance_metric，图像分块个数 segmentation_fn 等，这里就直接取默认值了。

同 4.1，这里我们挑选两类图像进行绘制，挑选依据也与 4.1 一致。正确分类的图像保存在./Figure/Lime/Correctly classified，错误分类的图像保存在./Figure/Lime/Wrongly classified。下面展示了标签为 Happy，Surprise 和 Disgust 的被正确分类的图片：

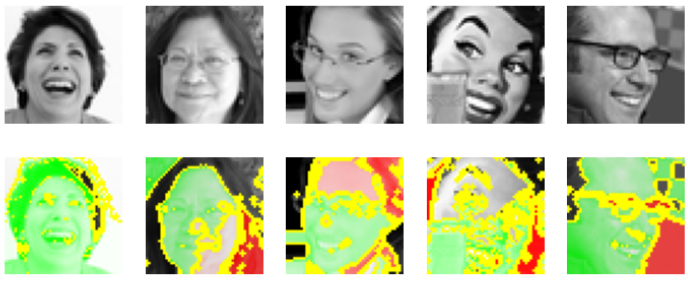

图 15: Happy

图 16: Surprise

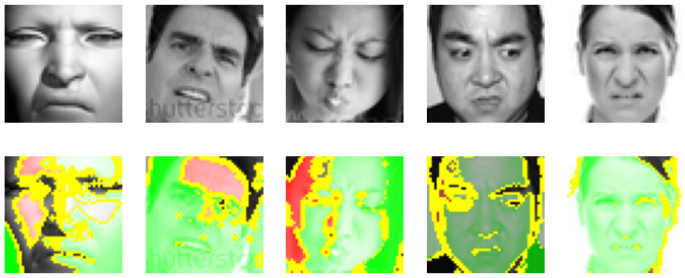

图 17: Disgust

图中绿色是支持首选分类的区域，红色是反对区域，黄色是分界线，黑白是无关区域。下面我们不妨来做一些归纳总结：

- 背景几乎永远被划分为无关区域。
- 额头和头发也几乎永远被划分为无关区域，有时甚至是反对区域。
- 整个面部一般都是支持区域。
- 嘴、眼睛还有鼻子很多时候会出现黄线，之所以只出现黄线是因为图片太小了，黄线所围的无关或反对区域显示不出来。这说明五官对表情分类作用也不大。
- 综上，下半部的面部肌肉对表情分类起主导作用。

- 至于不同表情的分类模式，从 lime 分析图上则很难看出。

4.5 观察模型训练

这边我们从两个角度来进行分析。一是从整个训练过程，二是从整个网络对图像的信息整合。第一个角度很简单，直接记录一下训练过程中的 loss，train_acc 和 test_acc，最后画一下图就行了。下面两张图，左图是训练图像，右图是微调图像。

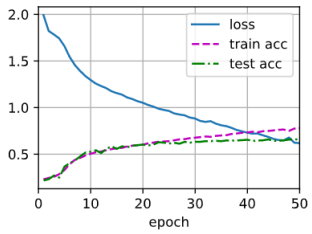

图 18: Train

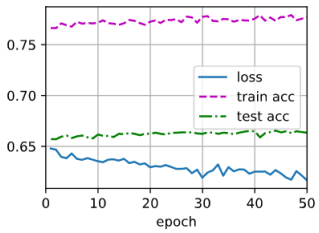

图 19: Fine-tune

可以看到，整个训练和微调过程还是比较平稳的，验证集和训练集正确率都是稳步上升的，说明 batch_size 和 lr 的选取还是比较合适的。

下面我们来分析一下整个网络是如何整合信息的，该部分代码为 filter analyse 注释。其实这个部分的分析和 4.3 的观察不同 filter 的输出差不多，只是这里的分析更加细致一些。首先，我把整个网络 20 个卷积层的输出都画了出来，每个卷积层随机选 5 个 channel。完整图像保存在./Figure/Filter analyse.png，不过鉴于从第四个 stage 开始图像基本就没有任何肉眼可以看懂的信息了，这里就仅展示前三个 stage(奥观海直呼内行)。

图 20: stage1

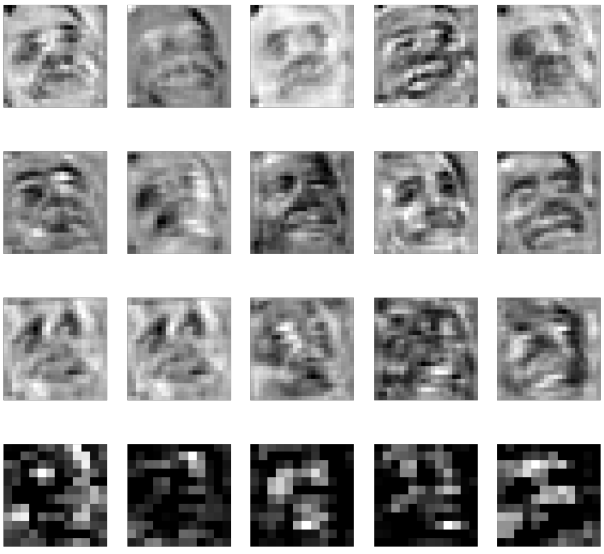

图 21: stage2

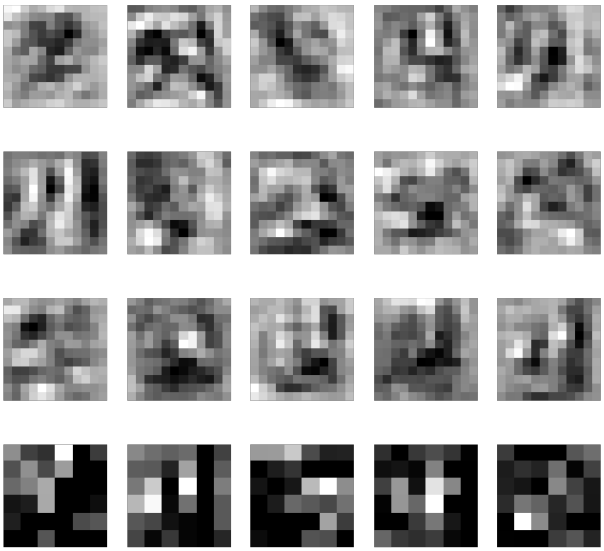

图 22: stage3

下面我就对每个 stage 做一些总结：

- 第一个 stage 的作用感觉和数字图像处理差不多，有些是做了灰度反转 (第一行第三列)，有些是调整了对比度 (第二行第一列)，有些是做了均值滤波 (第二行第五列)，甚至有些还能观察到理想低通滤波器的振铃现象 (第三行第二列)。总而言之，第一个 stage 的输出图像相当于对原图像做了 64 种不同的数字图像处理，而这些处理可以凸显出图像不同的特征。这个 stage 的输出图像打个比方就是学院派画作。
- 第二个 stage 的作用就开始抽象了起来，我将其理解为特征的初步提取。此时图像的不同位置都发生了形变 (第一行第四列的嘴巴，第三行第一列的眼睛)，这种形变我觉得有助于特定部位的特征提取，起到了一个强调的作用。同时，不同位置的高亮也表现出不同卷积层对脸的不同部位的重视 (第二行第四列脸颊肌肉的激活，第三行第五列眼睛和牙齿的激活)。所以这一个 stage 大概是对脸的几乎所有部位都进行了一边初步的特征提取。这个 stage 的输出图像打个比方就是印象派画作。
- 第三个 stage 已经很难看清具体特征了，我也就只能理解为神经网络

更加深邃的思想了 orz。这个 stage 的输出图像打个比方就是现代派画作。

5 舍不得删掉的废话

1. 提交作业时，我已将代码公开，id: shenbiachao，Notebook: fer2013-Net，代码一共 25 个 version，具体的代码更迭过程，不同网络、参数、框架的尝试过程都可以看到 (试图证明自己没有抄作业)。

2. 我的 id 是“天野阳菜天下第一”，不过卷了半天，最后也只卷到了第七名 (呜呜呜)。不过，在我心中天野阳菜永远是天下第一！

3. 这应该是算我第一次调参，这次经历属实让我学到了不少的知识 (好耶)。对于超参数的选取，我积攒了不少经验，不过网络结构如何调整还是一头雾水，今后还得多加锻炼。

4. 提交的 project 中，Figures 文件夹包含了所有原始图像，main.py 为我使用的代码，即 Kaggle 上原封不动下载下来的代码，Net(upload).param 为我提交到 Kaggle 上评分所用的网络，res.csv 为该网络的预测结果。train.py 为训练代码，test.py 为测试代码。train.sh 为训练脚本，test.sh 为测试脚本。

5. 如下运行训练脚本：bash train.sh train.csv。如下运行测试脚本：bash test.sh test.csv Net(upload).param。

6. 鸣谢：感谢徐满杰同学提供的 bash 脚本，感谢孙舒禺同学教会我如何在 Windows 环境下运行脚本，感谢洪亮同学和我一起探讨网络架构和观察方法的实现，感谢钟昊鸣同学和我一起探讨网络架构和训练方法。最重要的是，感谢郭老师宽限半天 ddl，临表涕零，不知所言。

6 参考资料

[1] http://www.d2l.ai/

[2] https://www.kaggle.com/haneenabdelmaguid/facial-expression-detection-2

[3] https://www.kaggle.com/kilean/emotion-detection-accuracy70

[4] https://www.kaggle.com/mohammed94/facial-emotion-detection-cnn

[5] https://github.com/weiaicunzai/pytorch-cifar100 [6] Simonyan, K., Vedaldi, A., & Zisserman, A. (2013). Deep inside convolutional networks: Visualising image classification models and saliency maps. arXiv preprint arXiv:1312.6034.

[7] https://github.com/wmn7/ML_Practice/blob/master/2019_07_08/Saliency%20Maps/.ipynb_checkpoints/Saliency%20Maps%20Picture-checkpoint.ipynb

[8] Ribeiro, M. T., Singh, S., & Guestrin, C. (2016, August). " Why should I trust you?" Explaining the predictions of any classifier. In Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining (pp. 1135-1144).

[9] https://towardsdatascience.com/interpretable-machine-learning-for-image-classification-with-lime-ea947e82ca13