机器学习纳米学位

毕业项目

张鸣一
2017年09月24日

用深度神经网络侦测走神司机
(Using Deep Neural Network to Detect Distracted Driver)

I. 问题的定义

项目概述

驾驶员未能完全将精力集中于道路上。致使他们的判断出现偏差，谓之“分心驾驶(distracted driving)”。导致分心驾驶的原因很多，其中包括：驾驶员用手机通话，传短信， 阅读， 使用全球定位系统，观看视频或电影，吃东西，吸烟，补妆，跟乘客聊天，身心疲劳等.

驾驶期间，驾驶员必须在任何时候都要全神贯注。否则会带来严重后果。研究表明，驾驶过程中使用移动电话的驾驶员发生交通事故的风险大约是不使用者的四倍。在所有交通事故中，近三成涉及分心驾驶.

世界卫生组织WHO的资料显示，随着移动电话用户的增长和新型车载通讯系统的迅速采用，分心驾驶对安全造成的严重威胁正逐年增加。无人驾驶虽然取得了长足进步，但完全替代人类尚需时日。如何有效地侦测司机走神并加以提醒就变得尤为重要。通过车载监视器监测驾驶员状态，通过深度卷积神经网络的分析，可以一定程度上达到这个目的.

问题陈述

项目取自Kaggle的State Farm Distracted Driver Detection比赛。目标是根据车载监视器显示的画面自动判断驾驶员是否走神，若驾驶员走神，则给予提醒。其中根据不同的走神原因又将走神状态进行了细分（具体分类信息将在“分析”一节进行详细描述）。通过对监控镜头中二维图像的分析，我们期望可以判断司机走神与否，如若走神，判断是什么原因导致的。

为了达成项目目标，我将建立一个深度卷积神经网络模型。因为重新训练一个卷积神经网络将是一个费力不讨好的方法，所以我将采用几个预训练模型，综合起来建立模型。训练所用数据集为Kaggle提供的监控图像与其对应的状态。用训练集数据训练模型，精细调节参数。

我期望训练的模型，对于输入的图像，可以分析图像中的特征，将图像内容分入相应的类中，并显示出分类结果。分类的好坏将以准确率和LogLoss函数来衡量.

评价指标

评估指标采用准确率和LogLoss来衡量. 准确率可以粗略评估模型的好坏. 而LogLoss可以更加精确地评估模型对于分类的确定程度. Kaggle的LeaderBoard采用的方式为LogLoss. LogLoss的表达式为

$$\text{LogLoss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\ln p_{ij}$$

其中是测试集的图片数, 是图像类别数(在此项目中), 是自然对数,当第张图属于第类, 是1, 反之则为0. 是预测第张图属于第类的概率, 取值在0到1之间. 从中我们可以看出, 当模型预测得越准确, 置信概率越高, 那么Logloss就越小. 因此

Logloss的值越小表明模型越好. 根据我对Kaggle上面讨论的浏览, 我决定将我的目标定在准确率大于90%, LogLoss小于0.5. 希望可以达成.

II. 分析

数据的探索

数据集来自Kaggle的State Farm Distracted Driver Detection竞赛。数据集分为三个文件。

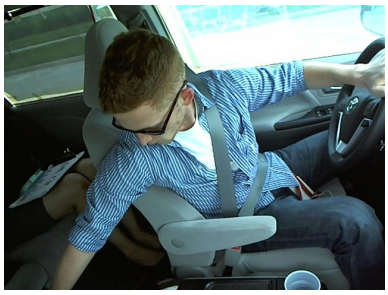
- `imgs.zip` 所有训练集和测试集图片的压缩文件：解压后包含`train`和`test`两个文件夹，其中`train`文件夹包含`c0`到`c9`十个文件夹，每个文件夹对应一种驾驶状态。具体信息统计如下。

```
├── train [22424 images]
│   ├── c0 安全驾驶 [2489 images]
│   ├── c1 右手手机打字 [2267 images]
│   ├── c2 右手打电话 [2317 images]
│   ├── c3 左手手机打字 [2346 images]
│   ├── c4 左手打电话 [2326 images]
│   ├── c5 调收音机 [2312 images]
│   ├── c6 喝水 [2325 images]
│   ├── c7 拿后面的东西 [2002 images]
│   ├── c8 整理头发和化妆 [1911 images]
│   └── c9 和乘客交谈 [2129 images]
│
└── test [79726 images]
```

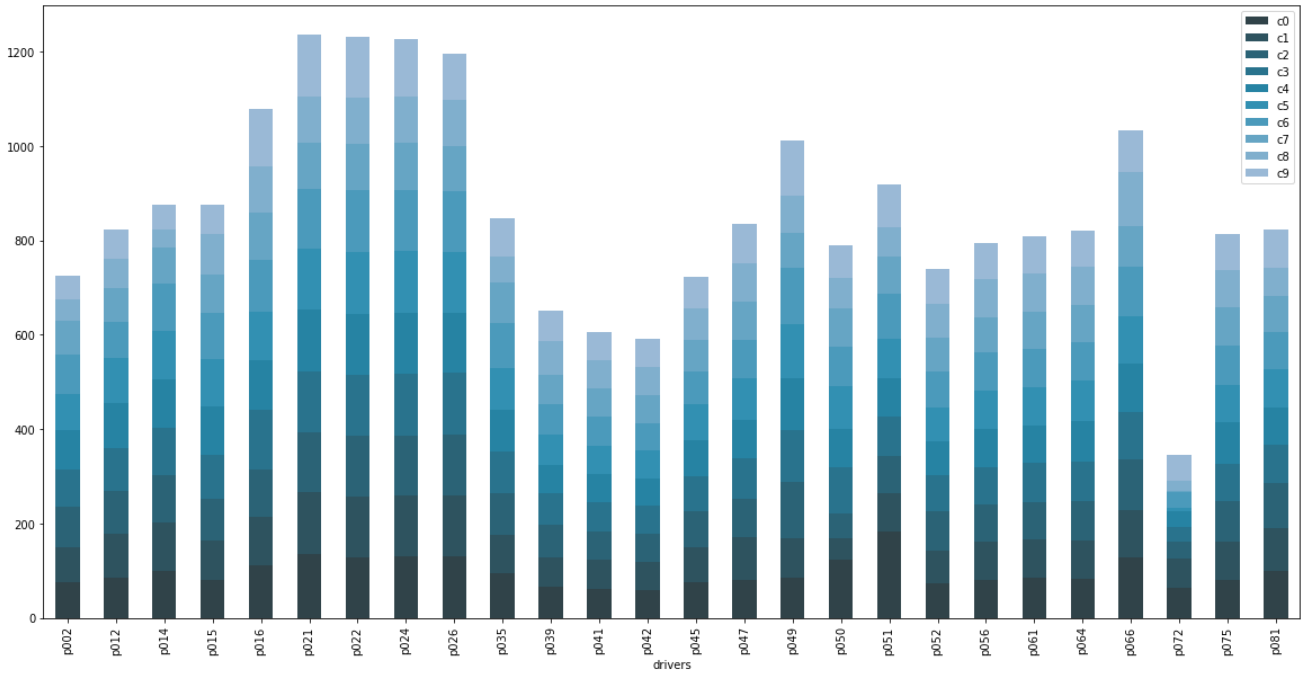
- `driver_imgs_list.csv` 所有训练集图像文件名(img)列表，包含对应的驾驶员(subject)和行为分类(classname)。其中有26位驾驶员。
- `sample_submission.csv` 比赛提交样本。

以下是每类驾驶状态的图片样本：

`c0` 安全驾驶	`c1` 右手手机打字	`c2` 右手打电话
		
`c3` 左手手机打字	`c4` 左手打电话	`c5` 调收音机
		
`c6` 喝水	`c7` 拿后面的东西	`c8` 整理头发和化妆

		
`c9` 和乘客交谈		
	480 × 640 × 3 (height × width × channel)	

将训练集按照驾驶员分类如下图：



可以发现训练集基本比较均衡。每个驾驶员每个类别都有。 通过观察训练图像，我们可以发现：

1. 图像从同一方向同一角度拍摄。
2. 对于同一个驾驶员，训练图像由监视录像逐帧生成。也就是说，若将同一个驾驶员的图片按照`driver_imgs_list.csv`所给的顺序生成`gif`文件，我们将得到一个连贯的小视频。生成的`gif`文件可在文件夹`gifs/`中找到。
3. 测试集图片数远大于训练集，并且训练集和测试集中的驾驶员没有交集。

由于图像的这些特征，同一驾驶员相同的状态的图片将非常相近。也就是说，若将图片 a 和 b 看成两个特征向量 V_a 和 V_b，如果 a 和 b 属于同一个驾驶员的同一个状态，那么 V_a 和 V_b 的差的模就会接近0

$$|V_a - V_b| \sim 0$$

但是同样的，由于图片都从同一角度拍摄，图片中有很多类似的部分。用这样的图片进行训练，很有可能会导致过拟合。这是本数据集最需要处理的问题。

由于测试集中没有训练集中的驾驶员，为了正确地检验模型的有效性，我们将按照司机来分训练集和验证集，这样同一个司机便只会出现在一个数据集里，可以很好地模拟训练集和测试集的情况。

探索性可视化

就像上面所说的，由于数据集是从监视录像中提取出来的，因此我们可以用它们生成一个个连贯的小视频。根据`driver_imgs_list.csv`里提供的顺序，我们可以为每个训练集中的驾驶员生成一个`gif`文件。你可以在文件夹`gifs/`中找到。生成代码参考了Gilberto Titericz Junior的 kernel Just relax and watch some cool movies.

算法和技术

迁移学习

在本项目中最主要的技术是迁移学习。从头训练一个自己构筑的深度卷积神经网络，不但花费大量的时间，而且效果并不好，是很不明智的选择。因此我们的新模型将建立在前人设计并训练好的身经百战的预训练模型的基础上，对其全连接层进行适当改进以符合我们数据集的要求。将这个新模型在自己的数据集上进行训练。实验证明这样可以大幅提高训练效率。本项目中我们只修改用Imagenet数据集预训练的模型的最后一层，使得最后的输出为我们期望的10个类别，而非imagenet的1000个类别。然后用我们的数据集在新的模型上训练。

卷积神经网络

卷积神经网络是专门用来处理图片数据的神经网络。卷积神经网络的结构可以千差万别，但归根结底由一下几个基本单元组成：

1. 输入层：一般是一组图片，一个四维矩阵，其中三维是图片像素，分别是（长，宽，频道）。比如本项目的数据集就是长640，宽480，RGB 红绿蓝三频道的图片集。
2. 卷积层：由一组小的(3x3, 5x5等)卷积过滤单元组成。每一个卷积过滤单元通过学习可以抓取图片的特征信息。一般来说图片通过卷积层后将会生成一组新的，特征得到加强的图片。
3. 整流线性单位（ReLU）层：在神经网络中引入非线性。
4. 池化层：将图片模糊化，以便下一层卷积层提取图片中的长程关联。
5. Dropout层：通过在训练时随机关闭部分神经网络结点，达到预防模型过拟合的作用。
6. 全连接层：全连接层主要出现在模型输出端，模型之前的三维数据将降为一维，多个全连接层组成一个普通的神经网络。其中最后一层全连接层将得到图片分类每一类的概率。其中最常用的激活函数为softmax。

K-fold 交叉验证

我们将数据集根据司机分成五份，每一份大约占20%的数据量。依次将其中的一份作为验证集，其他的作为训练集在我们的卷积神经网络模型上训练，并给出对测试集的预测。最终将五次的预测结果求(算数或加权)平均得到最终预测。

K-Nearest-Neighbor (KNN)

由于数据集有很高的相关性，因此我们可以对测试集进行KNN，找到其最近的K个邻居。也就是说，如前所述，我们可以将图像看成向量 V_a，假设另一图像的特征向量为 V_b，那我们所要找的就是距离 $|V_a - V_b|$ 最小的K个图像。最终我们对某一幅图片的预测将是对其与其相邻的K幅图片的预测值的距离的加权平均值。

集成学习(Ensemble Learning)

由于我们将用不同的预训练模型来训练，因此会有不同的模型和不同的预测。我们将用集成学习将不同模型的结果综合起来，给出我们最后的预测结果。

基准模型

我所使用的基准模型结构是ResNet50。ResNet50用TensorFlow-Slim image classification model library中的预训练模型来实现。这个library由google维护，提供了主流的图像分类模型以及它们在imagenet上训练的权重。我只修改了最后一层，从1000个结点变成10个结点以符合我们数据集的类别数。数据集并未做任何加强修改，也并未按照司机分为训练集和验证集，而是随机分为80%的训练集和20%的验证集。为了使数据集能够用在这个library提供的模型上，数据集必须要转化成`.tfrecord`文件。转换实现方法可在`TFRecord_maker.ipynb`文件中找到。用ResNet50在训练集上训练了30代大约14700步。具体过程和参数设置可以在`Resnet50_baseline_train_validation.ipynb`和`Resnet50_baseline_test.ipynb`中找到。最后在验证集上的Loss为0.046，准确率为0.996。而在LeaderBoard上的Loss为：Private: 0.84495, Public: 0.89158。准确率大约在0.75到0.80之间。排在Private LeaderBoard的约488/1440。验证集上和测试集上的差距主要是由于模型的过拟合。不过作为一个基准模型还可以。有一定的预测能力，准确率还说得过去。

具体结果可在 `Resnet50_baseline_train_validation.ipynb` 和 `Resnet50_baseline_test.ipynb` 中找到。

III. 方法

具体实现过程我们并没有采用基准模型所用的TensorFlow-Slim image classification model library，而是采用了Keras。主要原因有如下几点：

1. 数据占用空间较大。由于要做5-fold交叉验证，而数据都必须是`.tfrecord`文件，因此数据集必须有五份不同的拷贝，对应于不同的训练集和验证集。因为我在FloydHub上进行计算，网络空间以及网速都有限制。并不经济实用。
2. 做实时图片加强不容易。由于数据的性质，为了让模型尽可能不过拟合，图像加强在所难免。可在`dataset_preparation.py`中`load_batch`函数的`preprocessing_fn`前对`raw_image`加入一个图像加强函数。可是由于`raw_image`是一个 tensorflow 的`Tensor`类，所有的操作都必须用 tensorflow 里面的函数来实现… 三思以后觉着还是直接用Keras里的`ImageDataGenerator`函数来实现最方便快捷。
3. 在训练的同时做验证不容易。由于模型载入有`is_training`这个选项，在载入模型时，训练用的模型和验证用的模型是两个不同的graph，但两个图又有相同的scope，这就造成在执行Session的时候，tensorflow会报错。为解决这个问题，可能得将模型的scope设置为`reuse=True`, 但要调试好确实也是非常麻烦，学习曲线不小。若是没有这个功能，那我只能在训练完了以后才知道模型是否过拟合，大大降低了训练的效率。

因此，我最终还是选择用Keras来作为项目的主要工具来实现模型。

数据预处理

1. 将数据上传到FloydHub。将所有原始数据解压以后，放在同一个文件夹中。特别地为了可以用Keras的`flow_from_directory`函数，将测试集装在`test/test`文件夹下。注册安装FloydHub以后在网站上建立一个数据集`drivers_original`，从`shell`进入数据所在文件夹，初始化数据集

   ```
   floyd init drivers_original
   ```

   然后上传

   ```
   floyd data upload
   ```

2. 将数据按驾驶员分为5 folds。首先将训练集中的驾驶员列表提取出来，

   ```
   import pandas as pd
   from sklearn.model_selection import KFold
   df = pd.read_csv('driver_imgs_list.csv')
   # drivers' names
   subjects = np.unique(df['subject'])
   # classes
   classnames = np.unique(df['classname'])
   ```

   用`sklearn`中的`KFold`函数将其分为5 folds，分别存于`subjects_t`和`subjects_v`两个列表中。

   ```
   kf = KFold(n_splits=5, random_state=42, shuffle=True)
   subjects_t = []
   subjects_v = []
   for sub_t, sub_v in kf.split(subjects):
       subjects_t.append(list(subjects[sub_t]))
       subjects_v.append(list(subjects[sub_v]))
   ```

   然后定义函数`split_list`按照这个驾驶员的5 folds列表将`driver_imgs_list.csv`的数据列表分为训练集列表`df_t`和验证集列表`df_v`。最后定义函数`symlink_train_val`按照`df_t`和验证集列表`df_v`将图片分别放入`traini`和`validationi`文件夹, 其中 `i = 0, 1, 2, 3, 4`。

   ```
   for fold in range(5):
       df_t, df_v = split_list(subjects_t[fold], subjects_v[fold])
       symlink_train_val(df_t, df_v, fold, train_raw_dir)
   ```

3. 图像零中心化(zero-center)。Keras自带的预训练模型VGG和ResNet50需要对输入图像进行预处理。但由于其自带的预处理函数只能作用在batch好的图像上，而非单张图像，使得其不能直接用在图像生成函数`ImageDataGenerator`中。因此根据其自带的预处理函数，定义新的预处理函数

```
# preprocess function of VGG and ResNet50
def preprocess_fn(x):
    # RGB >> BGR
    x = img_to_array(x)
    x = x[:, :, ::-1]
    # Zero-center by mean pixel
    x[:, :, 0] -= 103.939
    x[:, :, 1] -= 116.779
    x[:, :, 2] -= 123.68
    return x
```

4. 图像加强。用Keras的图像生成函数`ImageDataGenerator`对训练集的图像进行加强。用`flow_from_directory`函数从相应的目录中读取。我们让训练集的图像随机旋转30度以内，水平和垂直平移10%。

```
data_gen_aug = ImageDataGenerator(rotation_range=30.,
                                  width_shift_range=0.1,
                                  height_shift_range=0.1,
                                  preprocessing_function=preprocess_fn)
train_aug_generator = data_gen_aug.flow_from_directory(train_dir,
                        target_size=(img_height, img_width),
                        class_mode='categorical',
                        batch_size=batch_size)
```

其中`train_dir`为训练集所在目录。 而对于验证集和测试集，则不做加强。

```
data_gen = ImageDataGenerator(preprocessing_function=preprocess_fn)
val_generator = data_gen.flow_from_directory(val_dir,
                                             target_size=(img_height, img_width),
                                             class_mode='categorical',
                                             batch_size=batch_size)
test_generator = data_gen.flow_from_directory(test_dir,
                                              target_size=(img_height, img_width),
                                              class_mode=None,
                                              shuffle=False,
                                              batch_size=batch_size)
```

其中训练集和验证集为默认换序shuffle，而测试集则不换序。图像加强可以在一定程度上缓解训练集图像间高度的关联性，在一定程度上可以让模型训练时更难过拟合。当然我只让其做了较小的加强，旋转和移动都不是很大，主要是为了训练可以较快收敛。

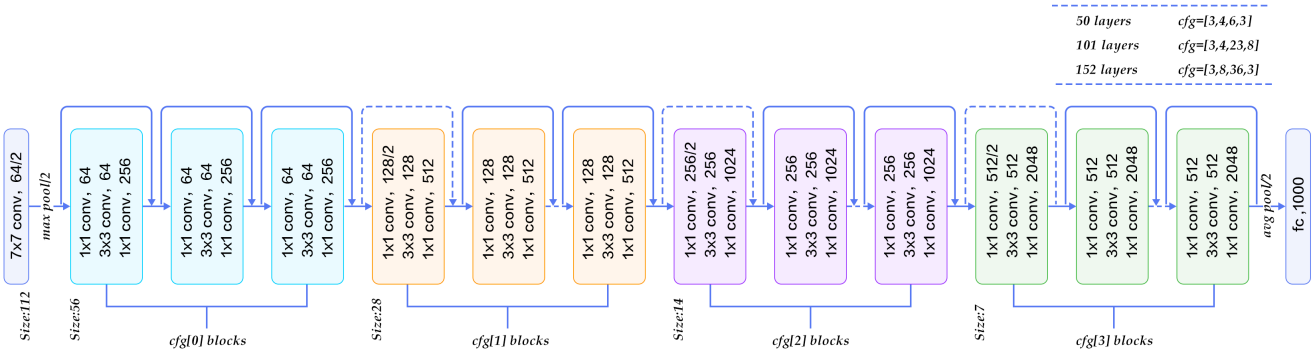
同时用`ImageDataGenerator`来做实时随机加强有一个额外的好处。将在下面一节详述。

执行过程和完善

由于我是边运行边完善， 因此我将“执行过程”和“完善”两节合并了。

选择预训练模型

Keras 的应用模块`applications`提供了带有预训练权重的 Keras 模型。在浏览了 Kaggle 讨论区和一些博客之后发现，VGG 和 ResNet 在本数据集上的训练效果普遍好于 Inception。因此我选择 VGG16 和 ResNet50 作为我预训练模型的基础模型。



图片来源: Deep Learning 101: Image Classification

建立模型架构

一开始我使用了杨培文的博文中所用的方法：先用去除了 top 层的基础预训练模型导出训练集和验证集的特征向量，然后自己构建 top 模型，用导出的特征向量来训练 top 模型。但训练效果并不好，验证集的 loss 很难达到 1 以下。在 `Keras_models.ipynb` 和 `Keras_transfer_value.ipynb` 可以管窥所做的尝试。（在写报告时发现之前这部分所做的绝大部分尝试被我删了…）。主要原因很可能还是过拟合。训练集的 loss 可降到0.2左右，而验证集的 loss 在尝试了很多不同的 top 模型后一直很难降低… 很有可能模型一直在选择一些不相关的特征。最后我放弃了这个方法。

之后我选择精细调节每一个预训练模型。为此我们定义函数`model_build`来构建模型

```
def model_build(MODEL, optimizer, layer_num_fix, top_dropout=False):
    base_model = MODEL(weights='imagenet', include_top=True)
    if top_dropout:
        x = Dropout(0.4, name='top_dropout')(base_model.layers[-2].output)
        x = Dense(10, activation='softmax', name='top_prediction')(x)
    else:
        x = Dense(10, activation='softmax', name='top_prediction')(base_model.layers[-2].output)

    model_ft = Model(base_model.input, x, name=MODEL.__name__)
    model_ft.compile(optimizer=optimizer,
                     loss='categorical_crossentropy',
                     metrics=['accuracy'])
    for layer in model_ft.layers[:layer_num_fix]:
        layer.trainable = False
    return model_ft
```

其中

1. `MODEL`: 预训练模型。为`VGG16`或`ResNet50`。
2. `optimizer`: 优化器。我尝试了`Adam`和`SGD`，定义如下

   ```
   adam = Adam(lr=1e-4, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=1e-6)
   sgd = SGD(lr=1e-4, decay=1e-6, momentum=0.9, nesterov=True)
   ```

   其中的参数设置有参考Sebastian Ruder的博文和CS231n课程的建议。经过尝试后发现，当初始学习率`lr = 1e-4`时模型收敛得最快。当学习率大于`1e-3`时模型不会收敛，而小于`1e-5`时模型收敛比较慢。`Adam`的其他参数保持默认。`SGD`参考`Adam`设置，并开启`nesterov`动量，以期有更高的收敛效率。在多次尝试以后，我几乎所有的训练都用`adam`作为优化器。因为用`adam`似乎更容易得到更小的loss。
3. `layer_num_fix`: 不训练的隐藏层数。其中 `VGG16` 设为 10，`ResNet50` 设为 142。由于两个预训练模型都是块状结构(block-wise)，如“选择预训练模型”一小节图片所示。以上层数限制意味着我只训练模型的最后一个模块。如此选择的意义主要是为了让模型训练效率高一点。
4. `top_dropout`: top 层中是否加 dropout。如果为真，则加入，反之则不加入。经过试验发现，不加 dropout 的结果更好，效率更高。可能由于 dropout 率设为了 0.4 造成收敛比较缓慢。

因此最终选用的模型架构为

```
model_vgg16_ft = model_build(VGG16, adam, 10, False)
model_resnet50_ft = model_build(ResNet50, adam, 142, False)
```

模型训练

我用`fit_generator`函数来训练模型。`batch_size`选择默认的 32，和`1e-4`的学习率配合收敛速率较快。起初，我采用如下设定

```
model_vgg16_ft.fit_generator(train_aug_generator,
                             train_aug_generator.samples//batch_size+1,
                             num_epochs,
                             validation_data=val_generator,
                             validation_steps=val_generator.samples//batch_size+1,
                             callbacks=[tensorboard_cb, ckpt_cb])
```

其中`num_epochs`设为`5`, `tensorboard_cb`和`ckpt_cb`分别是 `TensorBoard` 和 `ModelCheckpoint` 的 `callbacks`

```
tensorboard_cb = TensorBoard(log_dir='log/vgg16', histogram_freq=0, batch_size=batch_size,
                             write_graph=True, write_images=True)
ckpt_cb = ModelCheckpoint('vgg16_ft0_aug_weights.{epoch:02d}-{val_loss:.3f}.hdf5',
                          monitor='val_loss', verbose=1, save_best_only=True)
```

只保存验证集loss最小的模型的checkpoint。

`train_aug_generator.samples//batch_size+1`是每个`epoch`的步数。在每个`epoch`中所有训练集的图片都会跑且仅跑一遍。但是运行后发现，这样的设定造成必须跑完训练集才能检测验证集的loss，验证频率太低了。基本可以确定，即使保留了让验证集loss最小的模型的权重，它依然有很大的可能不是训练时最佳的选择。

因此我查看了`flow_from_directory`的源代码，确定了其变量`shuffle`指的是`epoch`间的`shuffle`。也就是说，只要`shuffle = True`，每个`epoch`里面，训练集图片的顺序都不一样。也就是说，`epoch`中的步数并没有必要保证所有训练集的图片都跑一遍。不同的`epoch`会让我们跑到不同的图片。同时我们还做了实时随机加强，这样更加保证了`epoch`与`epoch`间的训练集图片基本不可能是一样的。最终我所用的训练函数为

```
model_vgg16_ft.fit_generator(train_aug_generator,
                             50,
                             num_epochs,
                             validation_data=val_generator,
                             validation_steps=val_generator.samples//batch_size+1,
                             callbacks=[tensorboard_cb, ckpt_cb])
```

以及

```
model_resnet50_ft.fit_generator(train_aug_generator,
                                50,
                                num_epochs,
                                validation_data=val_generator,
                                validation_steps=val_generator.samples//batch_size+1,
                                callbacks=[tensorboard_cb, ckpt_cb, stp_cb])
```

其中每个`epoch`的步数设为 50。主要还是效率考量，验证太频繁，效率太低。`num_epochs = 50`，总共训练的步数差不多等价于可以跑完所有的训练图片五次。在训练ResNet50的模型时，还加入了`EarlyStopping`的`callbacks`, `stp_cb`

```
stp_cb = EarlyStopping(monitor='val_loss', min_delta=0.001,
                       patience=10, verbose=1, mode='auto')
```

其中`patience`设为 10, 表明只要有 10 个 epochs 验证集的loss没有改进，那么就退出训练。加入`EarlyStopping`主要是因为在训练ResNet50的模型时发现其收敛速度比较快，往往在 10 个 epochs 后已经开始过拟合了。不过由于训练集有实时随机加强，因此训练过程不太稳定，也有收敛比较慢的时候，因此依然将`num_epochs`设为 50 比较保险。

具体训练过程和结果可在下列文件中找到：

基础模型	运行文件	输出文件	LB-Private	LB-Public
VGG16	`Keras_fine_tuning_aug_fold0.ipynb`	`submission_vgg16_ft0_aug.csv`	0.46152	0.54683
VGG16	`Keras_fine_tuning_aug_fold1.ipynb`	`submission_vgg16_ft1_aug.csv`	0.34059	0.34422
VGG16	`Keras_fine_tuning_aug_fold2.ipynb`	`submission_vgg16_ft2_aug.csv`	0.44347	0.63316
VGG16	`Keras_fine_tuning_aug_fold3.ipynb`	`submission_vgg16_ft3_aug.csv`	0.44602	0.36146

基础模型	运行文件	输出文件	LB-Private	LB-Public
VGG16	`Keras_fine_tuning_aug_fold4.ipynb`	`submission_vgg16_ft4_aug.csv`	0.55130	0.52167
ResNet50	`Keras_fine_tuning_ResNet50_aug_fold0.ipynb`	`submission_resnet50_ft0_aug.csv`	0.35491	0.48216
ResNet50	`Keras_fine_tuning_ResNet50_aug_fold1.ipynb`	`submission_resnet50_ft1_aug.csv`	0.45376	0.42659
ResNet50	`Keras_fine_tuning_ResNet50_aug_fold2.ipynb`	`submission_resnet50_ft2_aug.csv`	0.26854	0.26618
ResNet50	`Keras_fine_tuning_ResNet50_aug_fold3.ipynb`	`submission_resnet50_ft3_aug.csv`	0.41992	0.49777
ResNet50	`Keras_fine_tuning_ResNet50_aug_fold4.ipynb`	`submission_resnet50_ft4_aug.csv`	0.37796	0.36269

集成学习(Ensemble Learning)

集成学习是将不同的模型结果综合起来，给出最终结果。由于可以将每一个fold训练的模型作为一个单独的模型来看待，因此我们将5-fold的结果（加权）平均的过程也可以看成集成学习。

为此我定义了函数`df_sum`

```
def df_sum(df_list, weight):
    df_new = 0
    img_name_df = pd.DataFrame(df_list[0]['img'])
    for i in range(len(df_list)):
        df_new += df_list[i].iloc[:, 1:] / weight[i]
    df_new /= np.sum(weight)
    df_new = pd.concat([img_name_df, df_new], axis=1)
    return df_new
```

其中`df_list`是一组预测结果导入的`DataFrame`的集合。若`df_list`里面的`DataFrame`是由每一个fold所训练的模型所给出的预测导入的，那么函数`df_sum`所返回的`DataFrame`就是5-fold的加权平均。权重`weight`，我将其设为LB-Private的分数。由于是除以分数，因此分数越低，权重越高。不过最后发现，无论是以VGG16为基础的模型，还是以ResNet50为基础的模型，算数平均值的结果更好。经过5-fold交叉验证综合以后的预测，相比于之前有了较大的提高，LB-Private的数值都小于0.25。

输出文件	LB-Private	LB-Public
`submission_resnet50_avg.csv`	0.23491	0.25426
`submission_resnet50_avg_w.csv`	0.23991	0.25918
`submission_vgg16_avg.csv`	0.24844	0.27943
`submission_vgg16_avg_w.csv`	0.25616	0.28510

继而将`submission_resnet50_avg.csv`和`submission_vgg16_avg.csv`两个结果求算数平均，我们得到

输出文件	LB-Private	LB-Public
`submission_vgg16_resnet50_avg.csv`	0.22376	0.24484

`LB-Private`进一步减小。具体执行文件为： `Keras_5fold.ipynb`

KNN

在数据探索一节，我们发现所有的图片数据都截取自监视录像，这样同一个驾驶员的同一状态的图片具有很高的关联性。因此我们对测试集图片进行KNN，找到与每一幅图片最相近的K幅图片。

由于原始图片的维数非常大（480 × 640 × 3 = 921600），且测试集图片数量不小（79726），计算他们之间的距离将非常费时费力。为提高效率，训练集图片被缩小了10倍，长宽变为64和48，并且我们只取黑白图片。这样每幅图片的维度大大减小（48 × 64 = 3072）。大幅降维使得大量图片信息丢失，会造成图片之间距离计算的偏差。但这也是在算力不够的情况下不得不做的妥协。即使是这样，计算每幅图最相似的10幅图也耗费了我笔记本电脑一整晚8小时的时间…

我用`sklearn.neighbors`模块中的`NearestNeighbors`来计算图片之间的距离

```
from sklearn.neighbors import NearestNeighbors
nbrs = NearestNeighbors(n_neighbors=10, algorithm='ball_tree').fit(X)
distances, indices = nbrs.kneighbors(X)
```

其中`X`是 79726x3072 的矩阵，每一行都是一个已经变为行向量的图片。求距离的算法为'ball_tree'，因为其在高维计算上效率较高。`distances`和`indices`都是 79726x10 的矩阵，每一列依距离由小到大，分别给出了离当列图片与其最近的十幅图(包含其自己)的距离和行指标。

接下来，定义函数

```
def knn_out(filename, K, weight, filename_new):
    df = pd.read_csv(filename)
    pdt = np.array(df[cls])
    pdt0 = np.array(df[cls])
    for k in range(1,K):
        pdt += pdt0[idx[:,k]] * weight[k]
    pdt /= np.sum(weight[:K])
    df[cls] = pdt
    df.to_csv(filename_new, index=False)
```

其中`filename`为要进行KNN计算的预测文件；`K`为最近邻居的个数；`weight`为权重；`filename_new`为做了KNN以后新的预测文件名。用此函数我们计算了当`K`为 2 到 10，权重为1, 和权重等差递减 0.1 的结果。发现普遍的，有等差递减权重的结果要好于平权的结果。最终，当`K`为 8 时，LB-Private结果最好，而当`K`为 7 时, LB-Public结果最好。

输出文件	LB-Private	LB-Public
`submission_vgg16_resnet50_avg_7nn_w.csv`	0.19952	0.22181
`submission_vgg16_resnet50_avg_8nn_w.csv`	0.19921	0.22188

这个结果大概可以在LB-Private排第62/1440名，在LB-Public排106/1440名。

具体执行文件：`KNN.ipynb`

IV. 结果

最终的模型建立在VGG16和ResNet50的基础上，进行5-fold交叉验证，综合预测结果，最后进行KNN，K=8，对预测结果加权平均得到最终结果。最终结果为

输出文件	LB-Private	LB-Public
`submission_vgg16_resnet50_avg_8nn_w.csv`	0.19921	0.22188

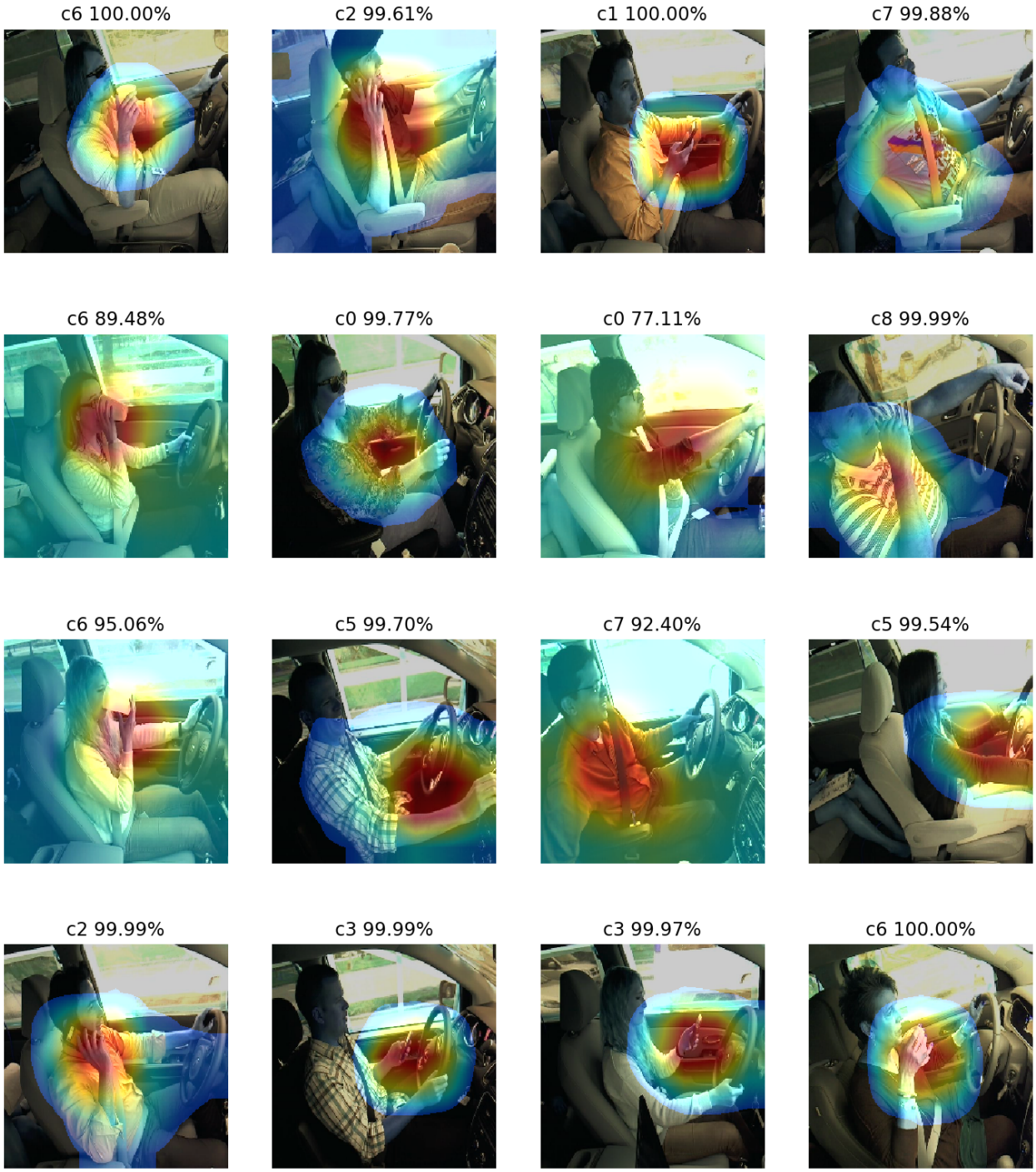
这个结果大概可以在LB-Private排第62/1440名，在LB-Public排106/1440名。其LB-Private排名在所有模型中最高。LB-Public的排名比较靠后，模型应该依然存在过拟合的情况。在不同的数据面前，预测会有不稳定的情况。不过loss在0.22左右，预测准确率应该在90%以上。模型应该是可信的。

最终模型的表现明显比基准模型更好。LB-Private从0.84495提升到了0.19921，提升幅度不小。

V. 项目结论

结果可视化

参考Class Activation Mapping的论文以及杨培文的代码，我们可以非常具体的看出卷积神经网络的关注点在什么地方。如下图



从图中我们可以清楚地看到，我们所训练的模型很好地抓住了图像中的特征点。特别是驾驶员手的位置部分。 比如说第1，5，9，12模型都抓住了水杯这个具体的信息。实在令我印象深刻！

具体实现过程可在`Visualization.ipynb`中找到。

对项目的思考

整个项目一开始是数据探索，在这一部分，最大的发现就是同一个驾驶员的同一个状态的图片有很高的相关性。这个性质主宰了整个模型构造和训练。因为这个原因，我们必须要尽量防止过拟合，因为这个原因，我们必须要谨慎地进行5-fold交叉验证，因为这个原因，KNN才能有效的用于测试集上。因此数据探索这一步是至关重要的，没有这一步作为指导，就很难有好结果。这也是项目中最有意思的地方之一。

接下来是数据预处理。为了得到有效的验证集，我们按照司机来分验证集和训练集。将司机分为五组，准备做5-fold交叉验证。每一个fold的训练集，都进行随机实时加强，以期减小过拟合的几率。

接着就是用Keras进行模型构建。我选择VGG16和ResNet50作为基础模型，只更改其最后一层，得到新的模型架构。然后固定部分权重，只专注于训练模型的最后几层。对模型进行交叉验证，将其在不同的训练集上训练，得到不同的模型。然后综合各个模型的预测结果，给出预测。

最后，对测试集进行KNN，将相近的图片的预测进行加权平均，得到最终结果。

本项目最困难的地方还是在调参上，由于要兼顾效率和好的结果，训练时必须尝试很多次，在找到能让模型可以较快收敛的参数的同时，还必须时刻记住过拟合这个本项目中最大的问题。

最终模型的结果有点超过我的预期。特别是KNN，因为我做了非常大胆的降维操作，丢掉了非常多的原始信息。但在最后，结果居然还有进步，真是有点出乎我的预料。我觉着在实际场景中，这个模型的预测能力还有很大的改进空间。特别是最后还是存在过拟合的问题。将现在这个模型应用在更通用的场景中可能会出现较多的误判。

需要作出的改进

正如上一小节所讲的，本项目还有较大的改进空间。首先对于训练集，我们可以做更为激进的加强，大角度的旋转和较大幅度的平移和一定程度的形变。这样可以在很大程度上防止模型过拟合。

其次是更加精细的训练。在训练中我只用了50步一验证的方法。为了达到更好的效果，可以吧步数减小。这样就能得到更加准确地找到开始过拟合的地方。

再者是可以尝试用更强大一些的模型， 比如ResNet152。本想尝试，但最终没时间做了。