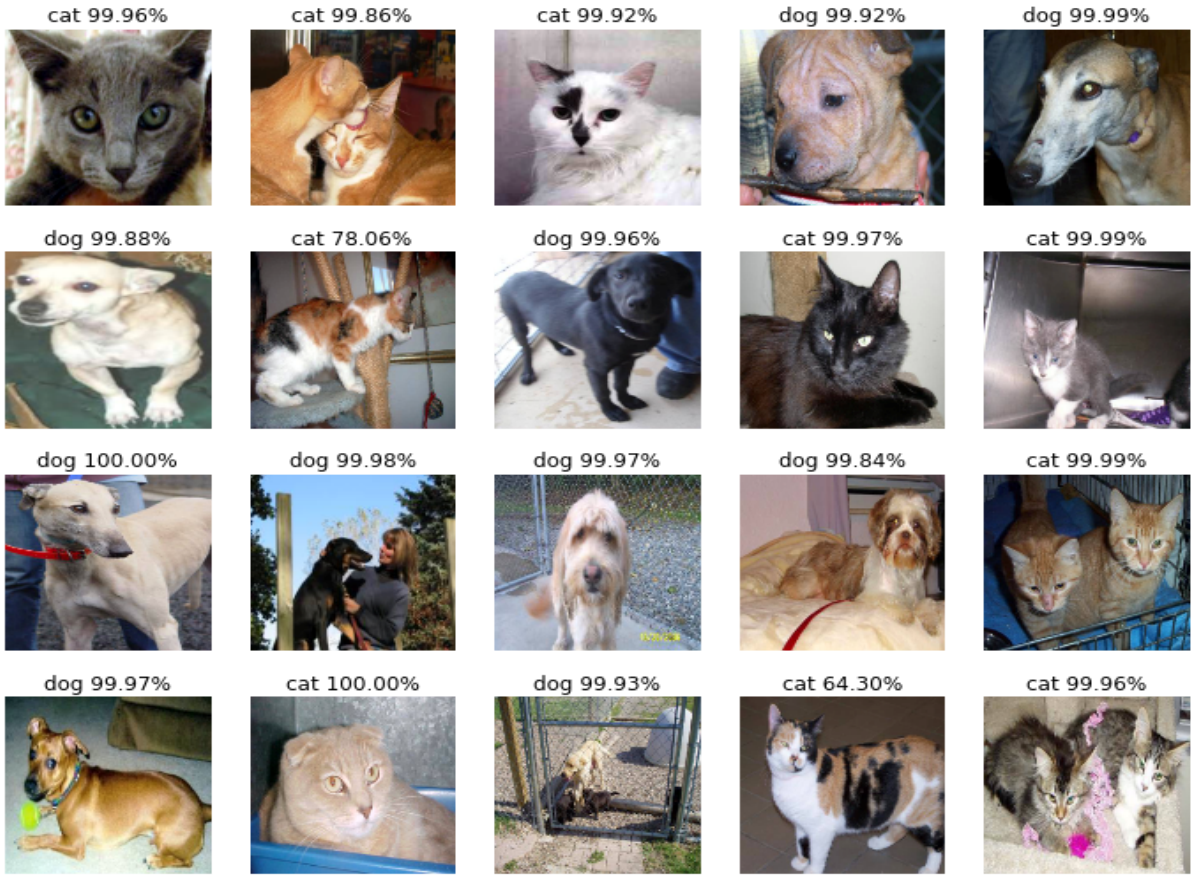


机器学习纳米学位

《猫狗大战》项目实战

2018 年 10 月 10 日

Junjie Huang
SERAPHIC Information Technology (Shanghai) Co., Ltd.
Songhu Road No. 433, Yangpu District, Shanghai

一、定义

1.1 项目背景

该项目来自于 Kaggle 上的一个竞赛：Dogs vs. Cats。参赛者需要训练一个模型去识别给定的图片是猫或者狗，这是计算机视觉领域一个典型的图像分类问题。

计算机视觉是深度学习技术最早实现突破性成就的领域。自 2012 年度深度学习算法 AlexNet 赢得图像分类比赛 ILSVRC 冠军，深度学习开始受到学术界广泛的关注。短短数年间，深度学习算法迅猛发展，图像分类的错误率不断递减。深度学习完全打破了传统机器学习算法在图像分类上的瓶颈，实现了计算机视觉研究领域的重大突破。

在技术革新同时，工业界也将图像分类、物体识别应用于各种产品中了。在 Google，图像分类、物体识别技术已经被广泛应用于无人驾驶汽车、YouTube、谷歌地图、谷歌图像搜索等产品中。

深度学习虽然最早兴起于图像识别，但是在短短几年内迅速推广到了机器学习的各个领域，并且均有出色的表现，在图像识别、语音识别、自然语言处理、机器人、电脑游戏、搜索引擎和金融等各大领域均有应用。

1.2 问题描述

本项目的目标是根据给定数据集训练模型，识别给定的图片是猫或狗，并且识别准确率能到达一个较高的水平。

这是一个典型的二分类问题，可以通过传统的监督学习算法如支持向量机、朴素贝叶斯分类器等算法来完成，也使用深度学习方法如卷积神经网络来完成。

1.3 输入数据

Kaggle 猫狗大战项目提供了测试集文件 train.zip 和训练集文件 test.zip，前者用于训练模型，后者用于模型预测。

训练集包含 25k 张照片，其中猫和狗各占一半，每张图片都带有标签；测试集则包含 12.5K 张照片，没有标记为猫或者狗。

1.4 评估指标

本项目拟采用对数损失函数来评估模型表现，定义如下：

$$\text{LogLoss} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right]$$

其中：

- n 为测试集中的图片数量
- $\hat{y}_i$ 为预测图片内容为狗的概率

- y_i 是类别标签，1 对应狗，0 对应猫

对数损失越小，代表模型的表现越好，预测能力越强。

二、分析

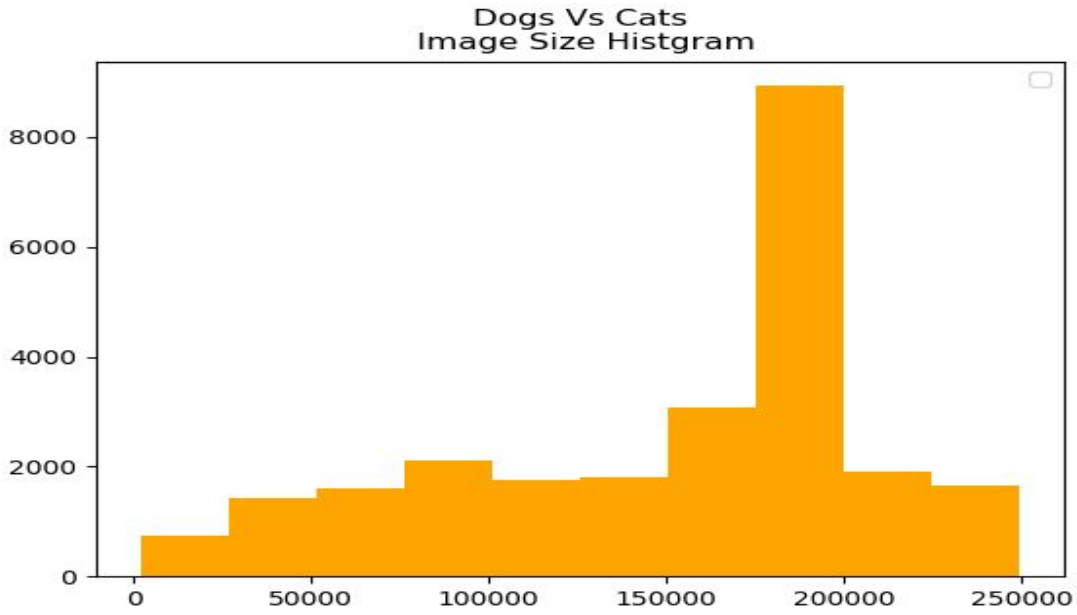
2.1 数据探索

通过研究数据集，发现图片尺寸相差很大，比如测试集中，图片宽度范围为 42px～1050px，高度范围为32px～500px，像素乘积范围为 1900～785664。

训练集中最小的图片为 dog.10747.jpg，尺寸为 50x38：

训练集中最大的图片为 cat.835.jpg，尺寸为 1023x768：

对训练集中的图片按宽高像素乘积，使用 matplotlib 绘制图片大小的统计直方图如下：



可以看出，像素乘积集中在 150k 到 200K 间的图片占比接近 50%，尤其是在 190k 左右比较集中。较大尺寸和较小尺寸图片的分布也都较为均匀。图片大小不一致无法应用于神经网络，因此需要对图片进行缩放处理。

2.2 算法及技术

项目要求使用深度学习方法识别一张彩色图片是猫还是狗，计算各自的概率。这里拟采用迁移学习的方法，使用卷积神经网络（CNN）来解决问题。

2.2.1 卷积神经网络

卷积神经网络（Convolutional Neural Network, CNN）是一种前馈神经网络，它的人工神经元可以响应一部分覆盖范围内的周围单元，对于大型图像处理有出色表现。

卷积神经网络由一个或多个卷积层和顶端的全连通层组成，同时也包括关联权重和池化层。这一结构使得卷积神经网络能够利用输入数据的二维结构。与其他深度学习结构相比，卷积神经网络在图像识别方面能够给出更好的结果。这一模型也可以使用反向传播算法进行训练。相比较其他深度、前馈神经网络，卷积神经网络需要考量的参数更少，使之成为一种颇具吸引力的深度学习结构。

2.2.2 迁移学习

通过使用之前在大数据集上经过训练的预训练模型，我们可以直接使用相应的结构和权重，将它们应用到我们正在面对的问题上，这种方法被称作为“迁移学习”，即将预训练的模型“迁移”到我们正在应对的特定问题中。

这种方式也符合我们我们人类自身的学习和进化模式，我们后人在科技、哲学、艺术、人文等各个领域所取得的进展，无一不是站在前人肩膀上。使用迁移学习，我们就可以使用前人取得的重大进展，而不需要从零开始训练一个神经网络，可以节省大量工夫。

2.3 基准模型

本项目的本质是图像分类，keras 提供了几种开箱即用的用于图像分类的 CNN 模型：

- VGG16
- VGG19
- ResNet50
- Inception V3
- Xception

根据 keras 基准模型的性能对比，拟采用 Top-1 准确率最高、模型参数和深度适中的 Xception 模型作为基准模型。

本文目标是 Kaggle 排行榜（public Leaderboard）前 10%，也就是 LogLoss 要达到 0.06127。

三、方法

3.1 数据预处理

3.1.1 调整文件目录层级

首先从 Kaggle 网站下载本项目所需的训练集文件 train.zip 和测试集文件 test.zip，分别解压放置于 data 目录下。为了使用 sklearn.datasets.load_files 工具，将测试集的图片按猫和狗分为两个目录存储，最终目录层级组织如下：

```
data
├── test
│   ├── 1.jpg
│   ├── 2.jpg
│   ├── .....
│   └── 12500.jpg
└── train
    ├── cat
    │   ├── cat.0.jpg
    │   ├── cat.1.jpg
    │   ├── ......
    │   └── cat.12500.jpg
    └── dog
        ├── dog.0.jpg
        ├── dog.1.jpg
        ├── ......
        └── dog.12500.jpg
```

3.1.2 图片 resize

本文采用预训练模型 Xception，由于其默认的图片输入大小为 299x299，因此先要将图片缩放到默认输入大小。

这里尝试过不同的尺寸：更小的尺寸（如 224 x 224）会加速训练的过程，但是最终在验证集上的得分更低，最终未能到达 Kaggle 目标；更高的尺寸（320 x 320）会加剧内存占用，导致代码报内存错误。

```
1.  rom keras.preprocessing import image
2.  from keras.preprocessing.image import ImageDataGenerator
3.  from tqdm import tqdm
4.
5.  target_image_size = (299, 299)
6.
7.  X = np.array([cv2.resize(cv2.imread(path, cv2.IMREAD_COLOR), #立方差值
8.                           target_image_size,
9.                           interpolation=cv2.INTER_CUBIC)
10.               for path in train_files])
11. Y = np.array([1 if 'dog' in path else 0 for path in train_files]) # for sigmoid
```

3.1.3 划分数据集

将初始训练集按照 4:1 的比例拆分为训练集、验证集。这里开始是按 7:3 的比例拆分，结果在验证集上得分并不高，没有达到预期目标；调整到 4:1, 有一定提高（val_los = 0.04293）；同时尝试过 9:1，可以加快收敛速度，用一半的 epoch 达到相近的分数（val_los = 0.04312）。总结原因，是因为在常规情况下训练集的数量越高，准确率也会越高，但因为总体样本有限，如果验证集过大意味着训练集越小，会造成欠拟合，而如果验证集过小的话，又会造成过拟合。

```
1.  from sklearn.model_selection import train_test_split
2.  from sklearn.utils import shuffle
3.
4.  # random_state设置为0，取保每次分割都一样
5.  X_train, X_validation, Y_train, Y_validation = train_test_split(X, Y, test_size=0.2,
    random_state=0)
6.
7.  # shuffle the train set
8.  X_train, Y_train = shuffle(X_train, Y_train)
```

3.1.4 图片归一化处理与数据增强

使用 keras 的内置 ImageDataGenerator 来对输入的图片进行归一化处理，并进行数据增强。

归一化处理：将每张图像的像素值除以 255，缩放到 0～1 之间。理由是归一化处理会加速梯度下降的收敛速度，否则很难收敛或者不收敛。

数据增强：随机剪切变换，随机缩放，左右、上下翻转，随机旋转一定角度。数据增强是扩充数据集、改善模型效果简单而有效的方法。

```
1.  datagen = ImageDataGenerator(rescale=1.0/255,       #像素值正则化0～1
2.                               shear_range=0.2,       #随机剪切变换强度
```

```
                                zoom_range=0.2,        #随机缩放幅度
                                rotation_range=45,     #随机旋转角度范围
                                horizontal_flip=True, #随机左右翻转
                                vertical_flip=True)    #随机上下翻转

train_generator = datagen.flow(X_train, Y_train, batch_size=batch_size)
validation_generator = datagen.flow(X_validation, Y_validation, batch_size=batch_size)
```

实验发现，这里比较有用的是剪切和缩放变换，对验证集上的准确率有一定改善。由于时间不充足，未能对比不同的剪切、缩放幅度对结果的影响，这里按经验值取 0.2。

3.2 执行过程

3.2.1 构建模型

```
from keras.layers import *
from keras.optimizers import *
from keras.applications import *
from keras.models import Model

base_model = Xception(include_top=False, weights='imagenet')
m = base_model.output
m = GlobalAveragePooling2D(name='avg_pool')(m)
m = Dropout(0.5)(m)
# m = Dense(2, activation='softmax')(m)
m = Dense(1, activation='sigmoid')(m)

model = Model(inputs=base_model.input, outputs=m)
```

在预训练模型 Xception 的基础上，追加一层平均池化层，然后加一层 dropout 层和全连接层。激活函数选用 sigmod，因为本项目是二分类问题；也可选用 softmax，但是需要对输入的训练集和验证集标签进行 one-hot 编码。这里 Dropout 层参数取一般的经验值 0.5，防止过拟合。

3.2.2 编译模型

```
## 编译模型
# model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
metrics=['accuracy'])
# model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.compile(optimizer='adadelta', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```

这里分别尝试了 rmsprop、adam、adadelta 三种不同的优化器，对比下来，adadelta 的收敛速度最快，最后到达的效果也最好。

而损失函数之所以选择 binary_crossentropy，也就是对数损失函数 LogLoss，一是因为它与输出层选用的激活函数 sigmod 相对应，二是因为最终在 Kaggle 上的排名是也是按 Logloss 来评估的，这样我们就能根据验证集上的 val_loss 得分来估计 Kaggle 上的表现。

3.2.3 训练模型

由于计算量比较大，在 AWS EC2 上创建一个 p3.2xlarge 实例，使用 GPU 来训练模型。

```
from keras.callbacks import EarlyStopping
from keras.callbacks import ModelCheckpoint

epochs = 50
batch_size = 16
earlyStopping = EarlyStopping(monitor='val_loss', patience=5, verbose=1, mode='auto')
checkpointer = ModelCheckpoint(filepath='weights.best.xception.hdf5', verbose=1,
save_best_only=True)
callback_list = [checkpointer, earlyStopping]

history = model.fit_generator(
    train_generator,
    steps_per_epoch=len(X_train) // batch_size,
    epochs=epochs,
    callbacks=callback_list,
    validation_data=validation_generator,
    validation_steps=len(X_validation) // batch_size
)

with open('xception.json', 'w') as f:
    f.write(model.to_json())
```

这里的 batch_size 是一个比较关键的超参数。一开始设置为 24，在测试集上的得分未能达到目标，通过反复实验，最终下调到 16，达到不错的分数。这是因为越小的 batch_size，使得训练在更为平缓的局部最小值停止，最终的泛化能力也越好，同时收敛速度也会越慢。batch_size 越大则收敛速度越快，但同时也需要更多的迭代次数来达到较好的成绩。

设置迭代次数为 50 次。为了防止过拟合，设定 val_loss 不再下降后的第 5 个 epochs 后停止训练，同时保存模型架构及最佳的权重值到文件，用于后续的预测。

四、结果

4.1 模型评估

```
Epoch 1/50
1250/1250 [==============================] - 493s 395ms/step - loss: 0.1388 - acc:
0.9464 - val_loss: 0.0906 - val_acc: 0.9631
Epoch 2/50
1250/1250 [==============================] - 441s 352ms/step - loss: 0.0858 - acc:
0.9690 - val_loss: 0.1114 - val_acc: 0.9635
Epoch 3/50
1250/1250 [==============================] - 440s 352ms/step - loss: 0.0695 - acc:
0.9758 - val_loss: 0.0718 - val_acc: 0.9725
Epoch 4/50
1250/1250 [==============================] - 443s 354ms/step - loss: 0.0611 - acc:
0.9781 - val_loss: 0.0581 - val_acc: 0.9777
```

```
Epoch 5/50
1250/1250 [==============================] - 442s 354ms/step - loss: 0.0497 - acc:
0.9831 - val_loss: 0.0864 - val_acc: 0.9731
Epoch 6/50
1250/1250 [==============================] - 443s 354ms/step - loss: 0.0462 - acc:
0.9829 - val_loss: 0.0706 - val_acc: 0.9723
Epoch 7/50
1250/1250 [==============================] - 440s 352ms/step - loss: 0.0442 - acc:
0.9838 - val_loss: 0.0839 - val_acc: 0.9699
Epoch 8/50
1250/1250 [==============================] - 442s 353ms/step - loss: 0.0452 - acc:
0.9841 - val_loss: 0.0650 - val_acc: 0.9753
Epoch 9/50
1250/1250 [==============================] - 441s 352ms/step - loss: 0.0390 - acc:
0.9863 - val_loss: 0.0568 - val_acc: 0.9771
Epoch 10/50
1250/1250 [==============================] - 442s 354ms/step - loss: 0.0376 - acc:
0.9869 - val_loss: 0.0623 - val_acc: 0.9765
Epoch 11/50
1250/1250 [==============================] - 442s 354ms/step - loss: 0.0355 - acc:
0.9883 - val_loss: 0.0572 - val_acc: 0.9787
Epoch 12/50
1250/1250 [==============================] - 442s 353ms/step - loss: 0.0315 - acc:
0.9889 - val_loss: 0.0610 - val_acc: 0.9819
Epoch 13/50
1250/1250 [==============================] - 443s 354ms/step - loss: 0.0311 - acc:
0.9895 - val_loss: 0.1495 - val_acc: 0.9577
Epoch 14/50
1250/1250 [==============================] - 443s 354ms/step - loss: 0.0297 - acc:
0.9896 - val_loss: 0.0551 - val_acc: 0.9813
Epoch 15/50
1250/1250 [==============================] - 444s 355ms/step - loss: 0.0266 - acc:
0.9903 - val_loss: 0.0598 - val_acc: 0.9803
Epoch 16/50
1250/1250 [==============================] - 440s 352ms/step - loss: 0.0304 - acc:
0.9894 - val_loss: 0.0663 - val_acc: 0.9761
Epoch 17/50
1250/1250 [==============================] - 442s 353ms/step - loss: 0.0267 - acc:
0.9906 - val_loss: 0.0539 - val_acc: 0.9801
Epoch 18/50
1250/1250 [==============================] - 442s 354ms/step - loss: 0.0237 - acc:
0.9914 - val_loss: 0.0500 - val_acc: 0.9821
Epoch 19/50
1250/1250 [==============================] - 442s 354ms/step - loss: 0.0284 - acc:
0.9903 - val_loss: 0.0612 - val_acc: 0.9793
Epoch 20/50
1250/1250 [==============================] - 443s 355ms/step - loss: 0.0208 - acc:
0.9929 - val_loss: 0.0645 - val_acc: 0.9801
Epoch 21/50
```

```
1250/1250 [==============================] - 442s 353ms/step - loss: 0.0259 - acc: 0.9903 - val_loss: 0.0496 - val_acc: 0.9831
Epoch 22/50
1250/1250 [==============================] - 442s 354ms/step - loss: 0.0213 - acc: 0.9929 - val_loss: 0.0655 - val_acc: 0.9795
Epoch 23/50
1250/1250 [==============================] - 441s 353ms/step - loss: 0.0199 - acc: 0.9932 - val_loss: 0.0430 - val_acc: 0.9846
Epoch 24/50
1250/1250 [==============================] - 442s 354ms/step - loss: 0.0179 - acc: 0.9940 - val_loss: 0.0659 - val_acc: 0.9797
Epoch 25/50
1250/1250 [==============================] - 443s 354ms/step - loss: 0.0202 - acc: 0.9932 - val_loss: 0.0429 - val_acc: 0.9854
Epoch 26/50
1250/1250 [==============================] - 441s 353ms/step - loss: 0.0203 - acc: 0.9929 - val_loss: 0.0632 - val_acc: 0.9773
Epoch 27/50
1250/1250 [==============================] - 442s 353ms/step - loss: 0.0181 - acc: 0.9941 - val_loss: 0.0832 - val_acc: 0.9773
Epoch 28/50
1250/1250 [==============================] - 442s 353ms/step - loss: 0.0174 - acc: 0.9938 - val_loss: 0.0585 - val_acc: 0.9837
Epoch 29/50
1250/1250 [==============================] - 443s 354ms/step - loss: 0.0177 - acc: 0.9938 - val_loss: 0.0615 - val_acc: 0.9815
Epoch 30/50
1250/1250 [==============================] - 442s 354ms/step - loss: 0.0171 - acc: 0.9943 - val_loss: 0.0867 - val_acc: 0.9763
Epoch 00030: val_loss did not improve from 0.04293
Epoch 00030: early stopping
```

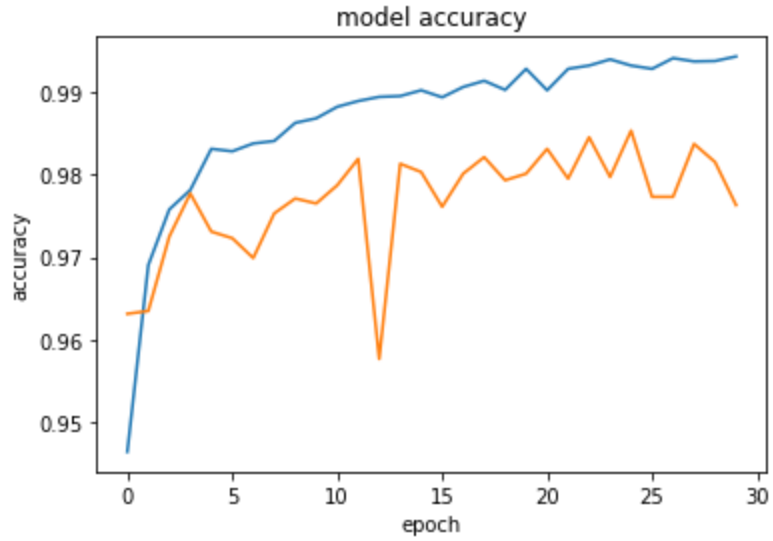
模型在训练过程 30 个 epoch 后停止，在训练第 25 个 epoch 时 val_loss 达到最低，在验证集上准确率为 98.54%，对数损失率为 0.0429。

4.2 模型训练过程可视化

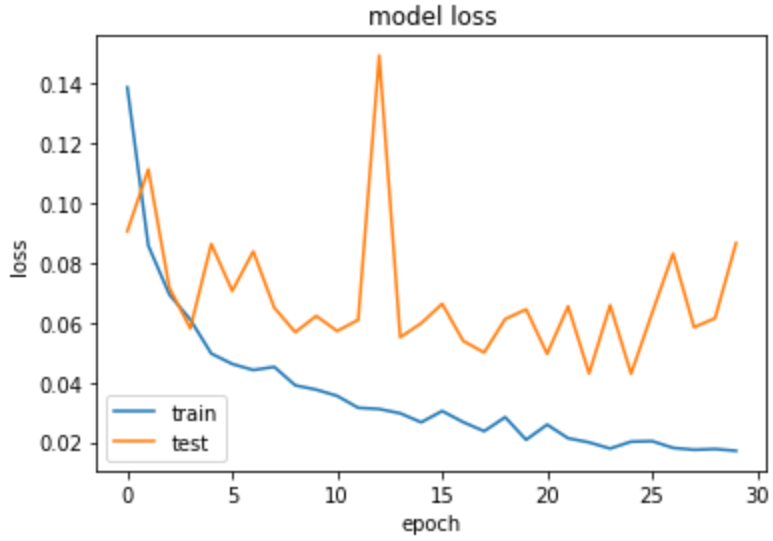
```
fig = plt.figure()
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
```

```
13. plt.xlabel('epoch')
14. plt.legend(['train', 'test'], loc='lower left')
```

训练集与验证集上的准确率：



训练集与验证集上的对数损失率：



可以看出随着迭代次数的增加，训练集上准确率逐渐提升到 99.5%，对数损失率不断下降到0.02以下；在验证集上，准确率达到 98.5% 左右没有再提升，对数损失率在 25 次迭代处降到最低。

4.3 模型预测与验证

```
1.  from tqdm import tqdm
2.  from keras.models import model_from_json
3.
4.  with open('xception.json', 'r') as f:
```

```
    model = model_from_json(f.read())

model.load_weights('weights.best.xception.hdf5')
def load_test_image(index):
    img = cv2.imread('data/test/%d.jpg' % index)
    img = cv2.resize(img, target_image_size)
    img.astype(np.float32)
    img = img / 255.0
    return img

test_num = len(test_files)
T = np.zeros((test_num, 299, 299, 3), dtype=np.float32)

for i in tqdm(range(test_num)):
    T[i] = load_test_image(i + 1)

predictions = model.predict(T, verbose=1)
predictions_clip = predictions.clip(min=0.005, max=0.995)

csv_content = 'id,label\n'
for idx, prob in tqdm(enumerate(predictions_clip)):
    csv_content += '%d,%f\n' % (idx + 1, prob)

with open('KerasResultUsingXceptionNotop.csv', 'w') as f:
    f.write(csv_content)
```

这里借鉴了老师提供的一个小技巧：将预测值限制到了 [0.005, 0.995] 区间内。原因是 kaggle 官方的评估标准是 LogLoss，对于预测正确的样本，0.995 和 1 相差无几，但是对于预测错误的样本，0 和 0.005 的差距非常大。这个技巧最我的得分提升到了 **0.05562**，在 Public Leaderboard 上排名为 **97/1314**，到达了 kaggle 前 10% 的项目目标。

五、结论

5.1 思考

通过完成本项目，对迁移学习和神经网络有了更深入的了解，同时对 keras 等相关工具库的使用也更加熟悉。

本文选取 Xception 模型并使用其在 ImageNet 数据上的权重，思路是没有问题的，但是前期排名一直在 150 名之外。通过调节 batch_size 等超参数，有了很大进展，最终结合别人的一些技巧初步达成了目标。由于时间有限且计算量巨大，很多想法暂时未能一一尝试验证，后续会还会继续跟进。

5.2 改进

后续改进的主要途径：

- 去掉训练集中的异常值

- 尝试将多个模型进行融合以提升效果
- 尝试采用更好的优化算法
- 尝试更小的学习率

5.3 鸣谢

项目的目标虽暂时达到，但是个人在机器学习这条路上才刚刚开始。在这里要感谢 Udacity 的在线课程帮我开启了通往新世界大门，感谢各位 mentor 对我的指导和鼓励，帮我通过了前期的各项关卡，让一个野生程序员找到了家的感觉。最后感谢我的妻子和刚满三个月的儿子，你们给了我在深夜抽空训练模型完成项目的动力。

参考文献

1. 《TensorFlow - 实战 Google 深度学习框架》（第2版）.郑宇才、梁博文、顾思宇 著.
2. 《Deep Learning with Keras》[Italy] Antonio Gulli. [India] Sujit Pal
3. Diederik P.Kingma, Jimmy. Ba.Adam: A Method for Stochastic Optimization
4. 手把手教你如何在Kaggle猫狗大战冲到 Top2 杨培文
5. 基于Pre-trained模型加速模型学习的6点建议 lqfarmer