猫狗大战

——udacity 机器学习进阶纳米学位

##毕业项目报告

一. 问题的定义

1.1 项目概述

在提供大量有标签的猫狗图片的前提下，我们希望计算机可以从这些图片中学习到猫狗的特征，从而使得计算机可以正确的对不带标签、未曾见过的猫狗图片进行分类。这就涉及到计算机视觉中的图像分类问题。图像分类，计算机视觉研究领域之一，计算机通过学习图像本身的特征将不同类别的图像区分开来。

深度学习，人工智能中机器学习里的一个分支，在近几年大量带标签数据集的产生以及计算机算力得到大幅度提升的背景下迅速发展，深度学习神经网络在计算机视觉中的应用也是大放异彩，各种深度学习网络框架脱颖而出，例如 VGG、ResNet、Inception、DenseNet、NASNet 等。也不乏出现了 tensorflow/keras 等优秀的快速开发工具和接口。

本项目利用 keras 快速开发接口，搭建一个深度学习网络模型，利用 kaggle 比赛《Dogs vs. Cats Redux: Kernels Edition》中的数据集对模型进行训练、优化，利用优化后的模型对未曾见过的猫狗图片进行分类。

1.2 问题陈述

在“猫狗大战”项目中--典型的图像识别二分类问题，需要搭建了一个深度学习网络模型，利用所提供的带有标签（标签为猫或者狗）的猫狗图片对模型进行训练，用最终训练好的模型对不带标签的猫狗图片进行预测分类。

我们期望训练后的模型在测试集上的得分表现 score 可以达到 kaggle 排行榜前 10%，也就是在 Public Leaderboard 上的 logloss 低于 0.06127。（logloss 定义见评价指标）

1.3 评价指标

在此二分类问题中，模型用 Binary Cross Entropy 作为损失函数。

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

其中，

- n 是测试集中图片的个数
- y_i 是图片预测为狗的概率
- y_i 图片为狗时值为 1，为猫时值为 0

损失函数大小对应模型的表现能力：当 logloss 较小时，模型表现能力强，正确预测猫狗图片的能力强；当 logloss 较大时，模型表现能力差，正确预测猫狗图片的能力弱。

二. 分析

2.1 数据的探索

项目中所使用的数据集来源于 kaggle 比赛《Dogs vs. Cats Redux: Kernels Edition》。目前该 kaggle 线上比赛已经结束。（感谢 kaggle 提供 interesting、free 的标签数据集）。

数据集链接:https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition/data

其中，训练集包含 25000 张猫狗图片，随机挑选 10 张图片显示如下：

```
---->image: ./train/dog.3220.jpg  - shape: (242, 299, 3)
---->image: ./train/cat.2505.jpg  - shape: (359, 400, 3)
---->image: ./train/cat.10813.jpg  - shape: (416, 423, 3)
---->image: ./train/dog.10654.jpg  - shape: (53, 59, 3)
---->image: ./train/dog.6002.jpg  - shape: (400, 399, 3)
---->image: ./train/dog.7269.jpg  - shape: (240, 319, 3)
---->image: ./train/dog.2455.jpg  - shape: (499, 375, 3)
---->image: ./train/cat.7538.jpg  - shape: (331, 500, 3)
---->image: ./train/cat.23.jpg  - shape: (256, 334, 3)
---->image: ./train/dog.11559.jpg  - shape: (375, 499, 3)
```

测试集包含 12500 张猫狗图片，随机挑选 10 张图片显示如下：

```
---->image: ./test/6624.jpg  - shape: (375, 499, 3)
---->image: ./test/6836.jpg  - shape: (166, 240, 3)
---->image: ./test/2221.jpg  - shape: (375, 499, 3)
---->image: ./test/2218.jpg  - shape: (215, 249, 3)
---->image: ./test/10512.jpg  - shape: (100, 97, 3)
---->image: ./test/10603.jpg  - shape: (375, 499, 3)
---->image: ./test/10942.jpg  - shape: (270, 206, 3)
---->image: ./test/5152.jpg  - shape: (500, 374, 3)
---->image: ./test/8160.jpg  - shape: (499, 331, 3)
---->image: ./test/6536.jpg  - shape: (299, 400, 3)
```

可以看到，训练集图片名中包含各自的分类（dog、cat），测试集图片是以数字命名的，图片中猫狗所占图片比例大小不一，清晰程度不一，图片大小（size）不统一。对于本项目猫狗分类问题，图片中猫狗比例，清晰程度不需要处理，但在输入模型之前，我们需要对数据集图片进行统一大小 reshape 操作。

随机展示的图片中没有异常值，但整个数据集中是否存在异常值呢？我们看到，各个图片的 size 不同，那么异常值是否和图片的 size 有关？对此我统计了图片 size 的分布情况，如图 2.1 所示：

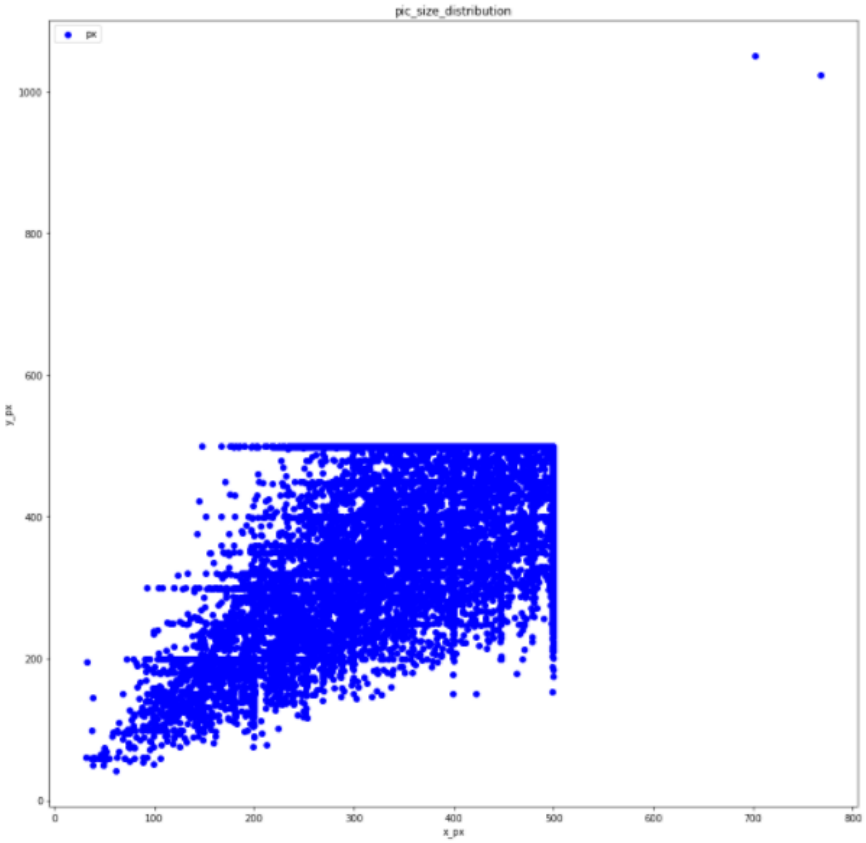


图 2.1 训练集图片尺寸分布情况

我们注意到在图片尺寸分布图的右上角，存在两个明显的尺寸异常值。那么他们是否是图片异常值呢？我们将这两幅图片显示如下：

```
---->image: ./train/cat.835.jpg  - shape: (768, 1023, 3)
---->image: ./train/dog.2317.jpg  - shape: (702, 1050, 3)
```

我们看到，这两张尺寸异常的图片同样是正常的猫狗图片。那么我们如何定义异常值？ImageNet 数据集中包含有猫狗的具体分类，对一个图片在载有 ImageNet 上预训练权值的 xception 模型上进行预测，如果其预测结果 top50 不包含猫狗真实的标签分类（图片预测值前 50 都没有正常分类），那么就将其视为异常值在输入模型之前，我们需要对数据集进行预处理：剔除掉训练集中的异常值。

2.2 算法和技术

卷积神经网络作为计算机视觉领域中最为抢眼的模型框架，近年来随着带标签数据集的出现和计算机算力的提升得到了前所未有的长足发展。不乏出现了像 VGG、ResNet、Inception、DenseNet、NASNet 等优秀的卷积神经网络。

本项目利用 keras 快速开发接口，搭建猫狗图像识别的二分类模型，fine-tune xception 深度神经网络模型，对猫狗数据训练集进行训练，随后对测试集进行预测，并将预测结果提交到 kaggle 进行评分。项目中涉及到以下算法和技术。

2.2.1 卷积神经网络

我们知道图像具有很强的空间相关性，图像的特征也体现在像素与像素之间的关系，图片中距离较近的部分相关性大，而距离比较远的部分相关性较小。CNN 卷积神经网络对图像进行了特征提取同时也保留了图像的空间信息。

一般来说，一个卷积神经网络包含一下几个部分：输入层、卷积层、激活函数、池化层、全连接层等。卷积层和池化层相互配合，逐层提取特征，最终通过几个全连接层，完成图像的分类。

- 输入层：

  输入层主要用于处理原始数据，例如：中心化（将数据各个维度中心来回到坐标系原点）、归一化（减少数据各个维度取值范围带来的差异性）。对数据进行中心化和归一化，使得模型对参数向量不在特别敏感，weights 的变动对分类器分类结果造成的扰动小，并加速模型的训练。

- 卷积层：

  我们选取一个给定大小宽度和高度的滤波器（filter），将图片分成多个小块 patch（patch 大小与 filter 大小相同），用这个 filter 对图片的第一个 patch 做内积运算，得到一个输出；然后我们可以利用这个 filter 在水平方向和垂直方向进行滑动（滑动的步长称为 stride）从而对图片的不同部分进行聚焦，最终得到下一层图像。如图 2.2 所示：假设 image 的 size 是 5*5，现在用一个 3*3 大小的 filter 对其进行卷积，stride 为 1，卷积后最终可以得到一个 3*3 的 convolved feature。如果希望卷积后的图片与输入大小相同，可以在原始图片外外层包裹一圈 0，这个行为称之为 padding。

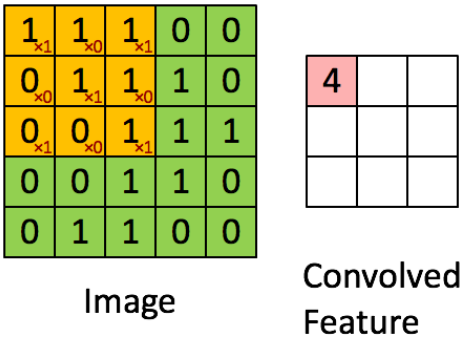


图 2.2 卷积操作

卷积层特点：

- 局部感知：传统神经网络中，每个神经元都要与上层所有像素链接；而卷积神经网络中每个神经元的权重个数均为卷积核的大小，即每个神经元只与图片部分像素相连接。
- 参数共享：一张图片进行卷积操作时，卷积核的权重不变，一幅图片中不同位置的相同目标，提取到的特征是相同的。参数共享在降低了网络模型复杂度的同时与局部感知一样，极大程度的减少了运算量。
- 多核：每个卷积层可以有多个 filter，每个 filter 对上层图片卷积操作后得到的图像就是该 filter 提取到特征的映射，即一个特征图（feature map）。不同的卷积核，提取到不同的特征。

- 激活函数：

  激活函数主要作用，是把卷集成输出结果转化成非线性。

- 池化层：

  池化层，主要是为了降低数据维度，减少数据和参数的数量，并赋予了模型对轻量形变的容忍度，增强了模型的泛化能力，在一定程度上减小过拟合。

- 全链接层

  两层之间所有的神经元都有权重链接，和传统的神经网络链接方式一样。

- Dropout

  典型神经网络的训练流程是，将输入通过网络正向传播，然后将误差进行反向传播，进行权值更新。Dropout 是针对上述过程中，临时随机删除部分神经元（本次不训练临时删除神经元对应的链接权重）。通过 dropout 处理，有效减少了神经元之间的共适应性，增强了模型的鲁棒性，同时也减小了过拟合的发生。

2.2.2 fine-tune

Finetune model 属于迁移学习的一种。在实际运用当中，由于训练时间的限制和训练样本过大的原因，很少有人从头开始训练网络模型，常见的作法有两种：

- 一是把预训练的 CNN 模型当做特征提取器；例如使用在 ImageNet 数据集上的预训练模型，去掉 top 的全链接层，然后将剩下的网络结构当做一个特征提取器。将数据集输入这个特征提取器中，去除 top 全链接层网络的输出即为我们提取到的特征，将得到的特征用线性分类器（例如 svm 或者 softmax 等）来进行分类图像。

- 二是 fine-tune 卷积网络。除了去掉 top 的全链接层，在网络 top 端搭建我们自己的分类器之外，我们需要在网络训练过程中对网络模型的后面几层或者是全部层进行权值更新。通常，由于前面几层提取到的是图像的通用特征（例如例如色彩、边缘、简单的图形等），后面几层是针对特定类别有关的特征，因此一般我们 fine-tune 卷积网络的后边几个层。

2.2.3 优化器

优化器用来更新和计算模型参数，使其更加逼近或者达到最优值，从而使 loss 损失函数最小。

神经网络中最常用优化算法是梯度下降，其核心是：对于每一个变量，按照目标函数在该变量的梯度下降的方向（梯度的反方向）进行更新，学习率决定了每次更新的步长。即在超平面上目标函数沿着斜率下降的方向前进，直到到达超平面的谷底。

- 梯度下降法变体：
  - 批量梯度下降（Batch GradientDescent）：在整个数据集上对每个参数求目标函数的偏导数，其反方向即为此参数变量的梯度下降方向。批量梯度下降中，每次更新都需要计算整个数据集上求出所有参数变量的偏导数，因此速度比较慢。批量梯度下降对于凸函数可以收敛到全局最小值，对于非凸函数可以收敛到局部最小值。

  - 随机梯度下降（Stochastic GradientDescent）：相对于批量梯度下降，随机梯度下降每次更新是针对数据集中的一个样本求损失函数，然后对其求相应的偏导数，SGD 运行速度大大加快。SGD 更新值的方差很大，在频繁的更新之下，目标函数会有剧烈的波动。当降低学习率的时候，SGD 表现出了与批量梯度下降相似的过程。

  - 小批量梯度下降法（Mini-batch GradientDescent）:在每次更新中，对 n 个样本构成的一批数据，计算损失函数，并对相应的参数求导；这种算法降低了参数的方

差，使得收敛过程更稳定。小批量梯度下降法，通常使我们训练神经网络的首先算法。

梯度下降存在一些难题：

学习率大小的选择：太小收敛速度慢，太大学习率会阻碍收敛，并会造成损失函数在最小值处的震荡，甚至导致发散；所有参数都采用相同的学习率：如果数据比较稀疏，我们希望较少出现的特征有更大的学习率；在对神经网络优化非凸函数时，目标函数可能会被困在“鞍点”，鞍点在各个方向的梯度值都为 0，SGD 很难从这些鞍点中脱开。

为了解决以上问题，就有了以下这些梯度下降的优化算法。

- 梯度下降优化算法：
  - 动量法：SGD 很难在陡谷（ravines）中找到正确更新方向，SGD 在陡谷周围震荡想局部极值处缓慢前进。动量法，就像从高坡推下一个小球，小球在滚动过程中积累了动量，在途中他变得越来越快（直到达到峰值速度）。算法中，参数的更新也是如此，动量项在梯度指向方向相同的方向逐渐增大，对梯度指向改变的方向逐渐减小。由此，将会加快收敛以及减小震荡。

  - Adagrad 法：主要功能是，对不同的参数调整学习率，对低频出现的参数进行大的更新，对高频出现的学习率进行小的更新。Adagrad 法大大提升了 SGD 的鲁棒性。Adagrad 主要优势之一是它不需要对每个学习率进行手工调节。劣势在于，Adagrad 会导致学习率不断的缩小，并最终变为一个无限小值，算法将不能从数据中学到额外的信息。

  - Adadelta、RMSprop：adagrad 的改进，解决学习率不断单调下降的问题。

  - 适应性动量估计法（Adam）：另一种对不同参数计算适应性学习率的方法。除了存储类似于 Adadelta 法和 RMSprop 中指数衰减的过去梯度平方均值外，Adam 法也存储像动量法中的指数衰减的过去梯度值均值。

总体来说，如果输入数据比较稀疏，那么使用适应性学习率类型的算法会有助于得到好的结果，此外，使用该方法的另一个好处是，在不调参、直接使用默认值的情况下，就能得到最好的结果。SGD 配合“a simple learning rate annealing schedule”，最终也能找到最小值，但花费时间要远远多于适应性学习率类型的算法的时间。

2.2.4 批量标准化（Batch Normalization）

数据带入模型之前，通常会有预处理过程，将输入数据中心化、归一化处理，以帮助模型进行学习训练，随着参数更新，除了输入层之外的其他各层网络的输入数据分布都会发生变化，这将影响网络的训练速度。

BN 算法就是为了解决再训练过程中，中间层数据分布发生变化的情况下的数据归一化。

批量标准化对每小批数据都从新进行标准化，在模型中加入批量标准化后，能使用更高的学习率而且不在对初始化参数特别敏感。

批量标准化可以看做是一种正则化手段，提高了网络泛化能力。

2.2.5 残差网络

CNN 网络能够提取不同级别的特征，而且特征的等级会随着网络的深度的加深而变高，网络层数越多，也意味着能提取到不同级别的特征越丰富。然而随着网络深度的加深，梯度消失/爆炸成为了训练深层次的网络的障碍，并导致无法收敛。归一初始化和中间归一化在很大程度上解决了这一问题，它使得可收敛的网络层数增加到数十层。

但深度网络收敛时，出现了一个退化问题，随着网络层数的增加，模型的准确率下降了，这种退化并不是由过拟合造成的，在一个合理的深度模型中增加更多的层却导致了更高的错误了。如图 2.3 所示：

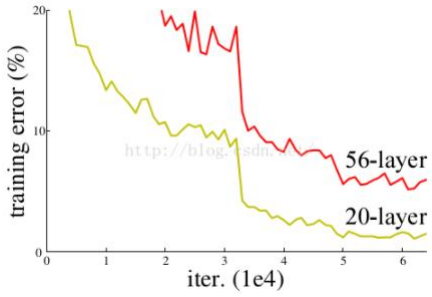


图 2.3 20 层和 56 层的“plain”网络在 CIFAR-10 上的训练错误率

为了解决这个问题，就有了残差网络：通过在一个浅层网络基础上叠加 y=x 层（identity mappings，恒等映射）。残差网络模块如图 2.4 所示：

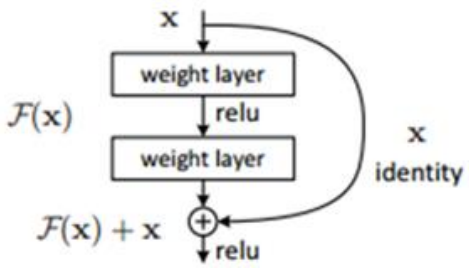


图 2.4 残差网络模块

残差网络有效解决了退化问题，其在 CIFAR-10 测试集上取得了相当好的成绩，如图 2.5：

method			error (%)
Maxout [10]			9.38
NIN [25]			8.81
DSN [24]			8.22
	# layers	# params	
FitNet [35]	19	2.5M	8.39
Highway [42, 43]	19	2.3M	7.54 (7.72±0.16)
Highway [42, 43]	32	1.25M	8.80
ResNet	20	0.27M	8.75
ResNet	32	0.46M	7.51
ResNet	44	0.66M	7.17
ResNet	56	0.85M	6.97
ResNet	110	1.7M	**6.43** (6.61±0.16)
ResNet	1202	19.4M	7.93

图 2.5 CIFAR-10 测试集上分类错误率

2.2.5 xception

Xception 设计思路可以从 inception 模块设计思路开始说起。在卷积神经网络中，每个卷积层会通过卷积操作从输入数据或上一层中提取更高级特征，而卷积核大小的不同选择直接影响了提取到的特征表象，google 提出 Inception 模块：对上一层进行 1*1、3*3、5*5、maxpool 然后进行 concat 操作，从而得到更为全面的特征表象（增加了网络的宽度）。inception 模块结构如图 2.6：

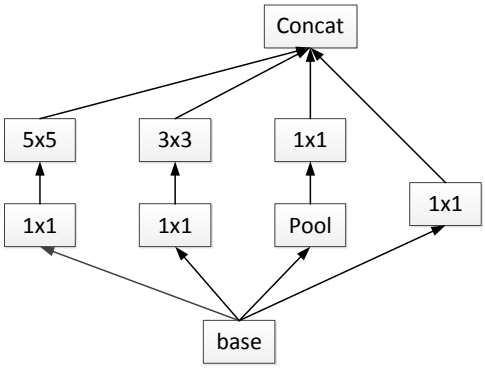


图 2.6 最初的 inception 模块

基于 Inception 模块提出了 InceptionV2，将 5*5 用两个 3*3 卷积核代替，从而降低了参数数量，并提出了 BN 算法。结构如图 2.7：

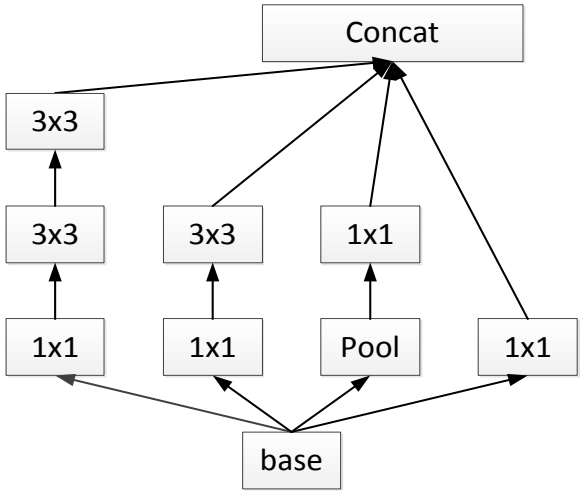


图 2.7　Inception 模块中 5x5 用 2 个 3x3 卷积代替

InceptionV3 中又提出了因式分解，即将 N*N 卷积核用一个 1*N 和一个 N*1 卷积核替代，进一步提快了计算速度，结构如图 2.8 所示：

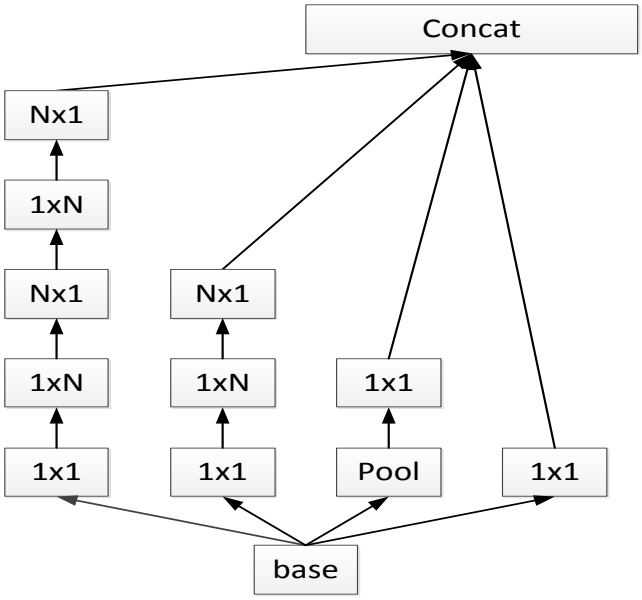


图 2.8　NxN 卷积因子分解后的 Inception 模块

随后又将 ResNet 残差网络加入到网络中，提出了 Inception-ResNet，有效防止了梯度消失的问题。

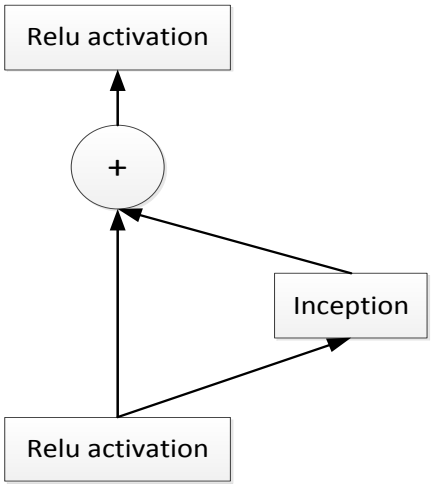


图 2.9 一般 Inception-resnet 模块的架构

相对于图 2.6 的 inception 模块来说，图 2.10 为一个简单化的 inception 模块（没有 pooling）：

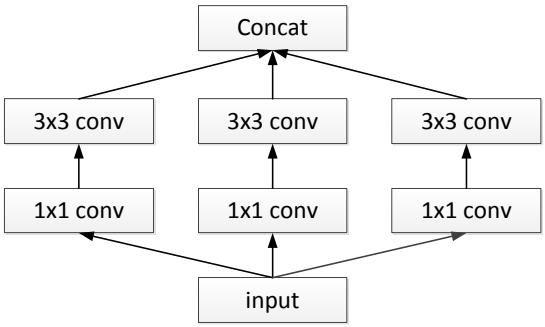


图 2.10 简单的 inception 模块

进而可以延伸出图 2.11，某种角度来看图 2.11 等价于（重构）图 2.10。这次先对 input 进行 1x1 卷积，然后再对 1x1 卷积输出 channels 平均分为 3 部分，随后对每部分进行 3x3 卷积操作，最后对各部分进行 concat 操作：

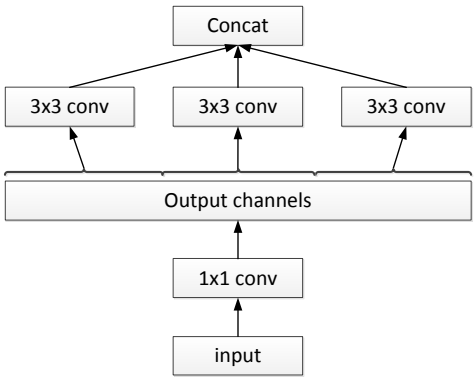


图 2.11 一个严格等价重构的简单的 inception 模块

对于图 2.11，有一种极端的情况，即对 1x1 卷积输出的每个 channel 进行单独的卷积，然后再进行 concat 操作，如图 2.12 所示：

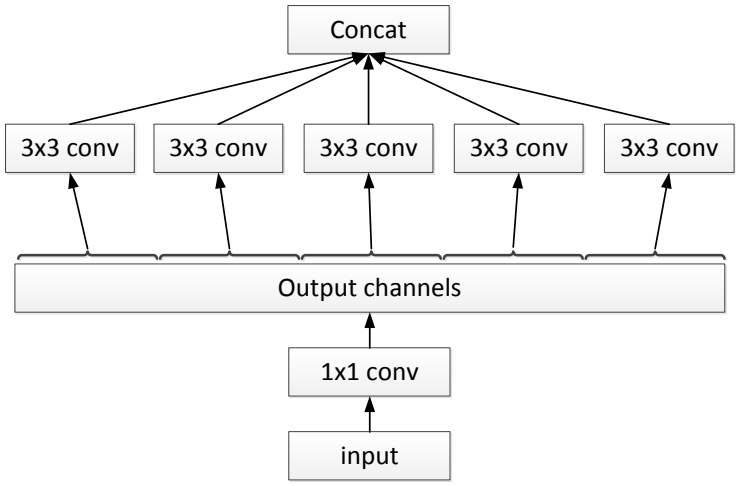


图 2.12 对 1x1 卷积输出的每个 channel 进行独立的卷积的 inception 模块

图 2.12 中所示的“极端的”inception 模块和 depthwise separable convolution 的主要区别有两点：

- 操作顺序不同：depthwise separable convolution 先执行 channel-wise spatial convolution，然后再执行 1x1 卷积；而图 7 是先执行的 1x1 卷积。
- 分线性激活函数是否存在：图 7 中每一步后边都有 Relu 非线性激活函数，而 depthwise separable convolution 后边通常不会设置分线性激活函数。

2017 年 google 基于 depthwise separable convolution 结构重新设计了 Inception 模块，即为 Xception，进一步在减少模型参数的前提下增加了模型的表现能力。下图为 Xception 的网络架构（图中的 SeparableConv 即为 depthwise separable convolution）：主要包括 14 个 block，其中 Entry flow 包括 4 个 block，Middle flow 包括 8 个 block，Exit flow 包括 2 个 block。除了第一个和最后一个 block 之外，其余每个 block 都有 residule connection（图中+号）。

Figure 5. The Xception architecture: the data first goes through the entry flow, then through the middle flow which is repeated eight times, and finally through the exit flow. Note that all Convolution and SeparableConvolution layers are followed by batch normalization [7] (not included in the diagram). All SeparableConvolution layers use a depth multiplier of 1 (no depth expansion).

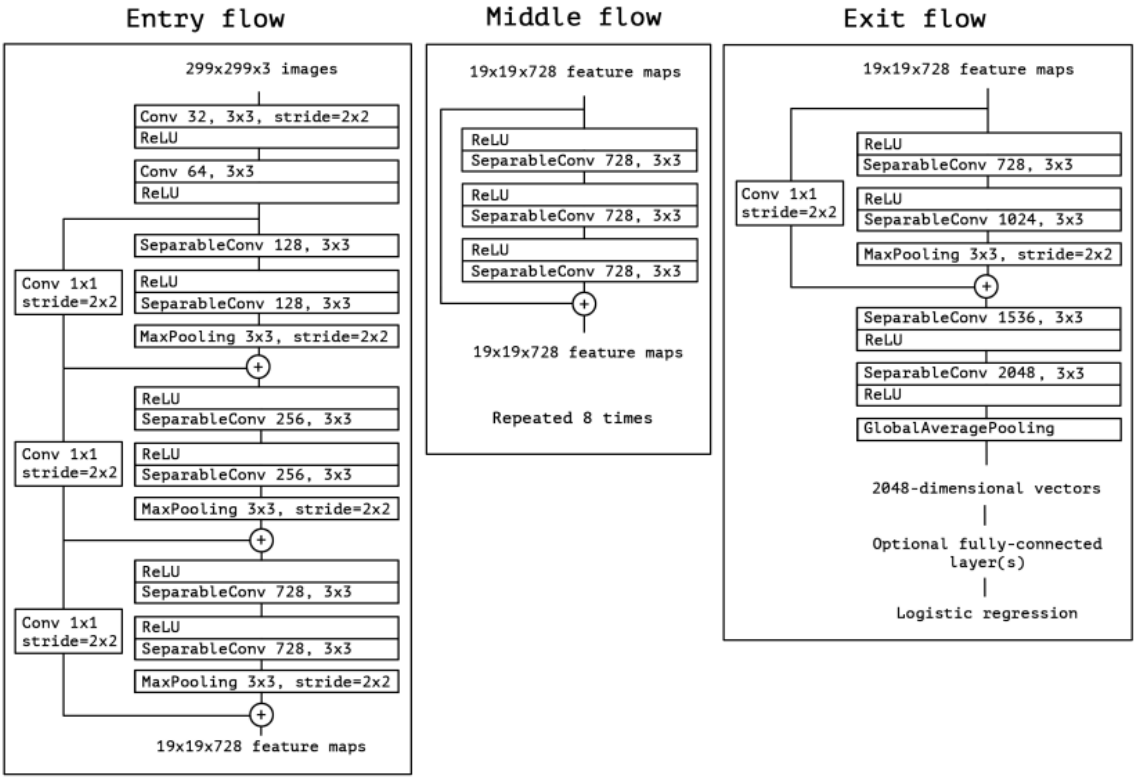


图 2.13 Xcepiton 网络架构

Xception 论文中已经明确对比测试，单模型对比测试数据如下图所示，可以看出 Xception 在 ImageNet 数据集上的表现要优越于其他模型。在猫狗大战项目中，将使用迁移学习，对 Xception 进行 fine-tune，由于 ImageNet 数据集中有猫狗的具体分类，因此我相信，这种做法可以取得不错的成绩。

Table 1. Classification performance comparison on ImageNet (single crop, single model). VGG-16 and ResNet-152 numbers are only included as a reminder. The version of Inception V3 being benchmarked does not include the auxiliary tower.

	Top-1 accuracy	Top-5 accuracy
VGG-16	0.715	0.901
ResNet-152	0.770	0.933
Inception V3	0.782	0.941
Xception	**0.790**	**0.945**

图 2.13 各个模型在 ImageNet 数据机上的准确率

2.3 基准模型

我们期望训练后的模型在测试集上的得分表现 score 可以达到 kaggle 排行榜前 10%，也就是在 Public Leaderboard 上的 logloss 低于 0.06127。

三. 方法

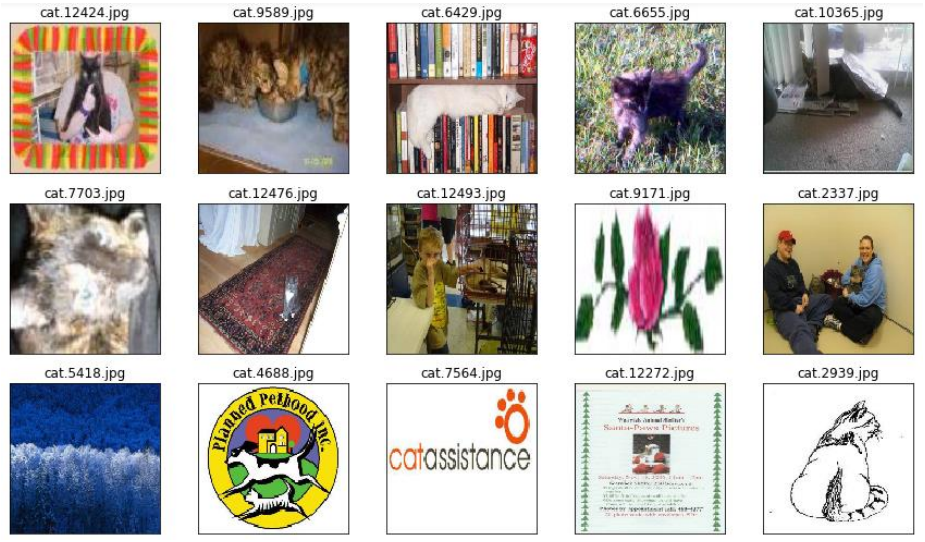
3.1 数据预处理

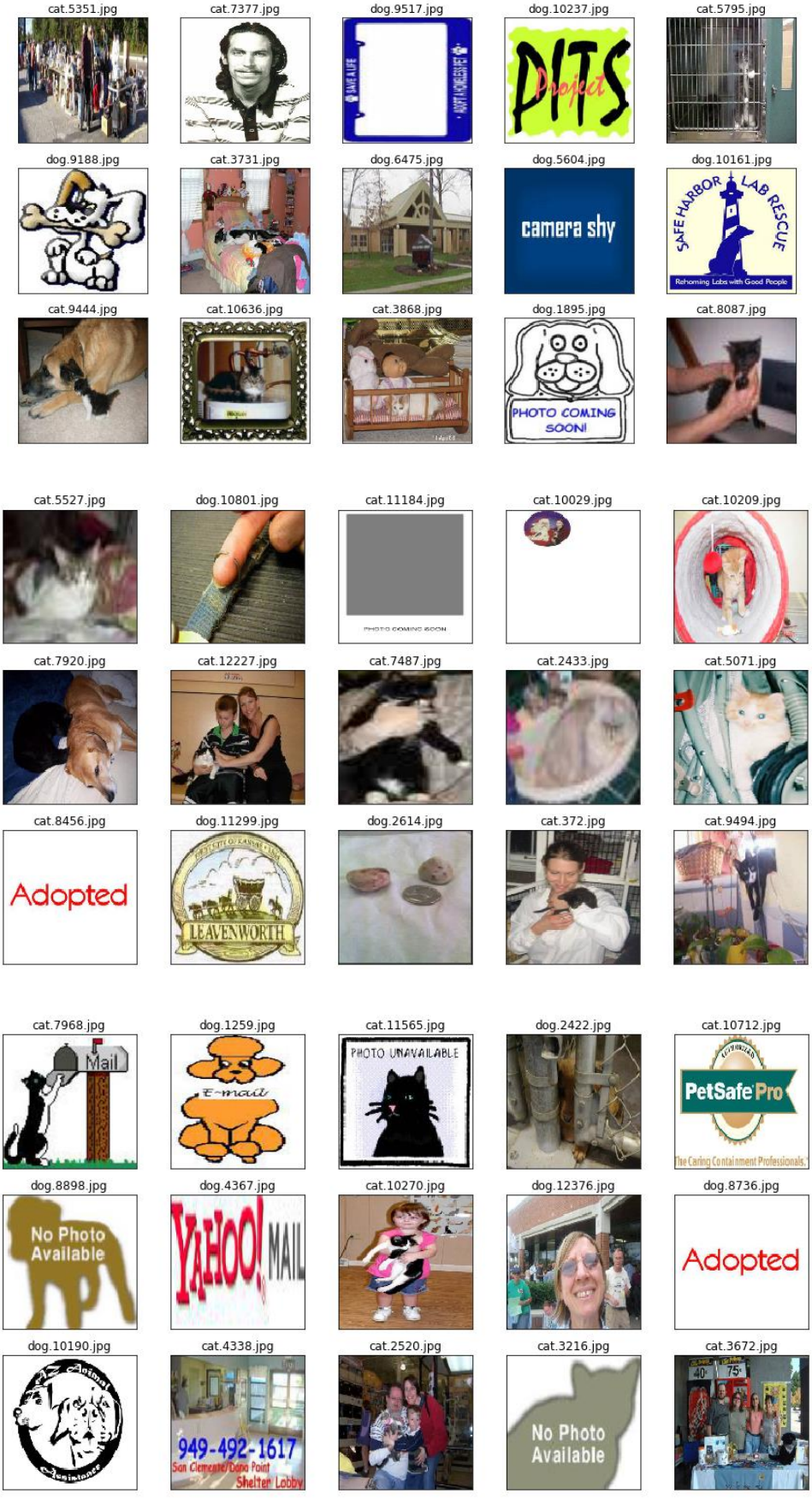
3.1.1 剔除异常值

数据探索时发现，训练集中存在异常值（非猫非狗的图片），在将数据输入模型之前，需要对这些异常值进行处理。

项目中利用装载 Imagenet 数据集上的预训练权值的 Xception 模型来实现异常值的检测。ImageNet 数据集中包含有猫狗的具体分类，对一个图片在载有 ImageNet 上预训练权值的 xception 模型上进行预测，如果其预测结果 top50 不包含猫狗真实的标签分类（图片预测值前 50 都没有正常分类），那么就将其视为异常值。

我们对 25000 张训练集图片进行 top50 预测，得到共 60 张异常值图片，我们可以看出被判定为异常值的图片多为非猫非狗图片、或者是背景特别模糊的猫狗图片、亦或者是猫狗同时存在的图片。不包含 cat/dog 正常标签的图片（异常值）入下图所示：





将异常值从训练集中剔除。

3.1.2 读入图片、统一 size、shuffle

对剔除后的训练集进行 shuffle 操作：

```
import random
random.shuffle(train_image_list)
```

读入内存并 reshape 统一大小（299*299*3）：

```
def read_batch_img(batch_imgpath_list):
    '''read batch img and resize'''
    images = np.zeros((len(batch_imgpath_list), 299, 299, 3), dtype=np.uint8)
    for i in range(len(batch_imgpath_list)):
        img = cv2.imread(batch_imgpath_list[i])
        img = img[:,:,::-1]
        img = cv2.resize(img, (299,299))
        images[i] = img
    return images
```

3.1.3 生成标签、划分训练集/验证集

利用图片的命名获取对应的 label，然后将训练集平均分为 5 份，一份作为验证集其余四份作为训练集。

```
#生成对应 label
def get_lables(image_list):
    labels = np.zeros(len(image_list), dtype=np.uint8)
    for i,item in enumerate(image_list):
        if "dog" in item:
            labels[i] = 1
        else:
            labels[i] = 0
    return labels
Y = get_lables(train_image_list)
#划分成训练集和验证集
val_X = X[:math.ceil(len(train_image_list)/5)]
val_Y = Y[:math.ceil(len(train_image_list)/5)]
train_X = X[math.ceil(len(train_image_list)/5):]
train_Y = Y[math.ceil(len(train_image_list)/5):]
```

3.1.2 中心化

数据集在输入模型训练之前，将利用 xception 自带的 preprocess_input 进行预处理，对输入数据的各个 channel 分别减去各自 channel 的均值，即分别对各个 channel 中心化处理。

3.2 模型构建

base model 选用 Xception，加载在 ImageNet 上预训练的权值，包含顶端的全局平均池化层，不包含 top 的分类器，随后在 base model 后边加一个二分类分类器。

```
# create the base pre-trained model
base_model = xception.Xception(weights='imagenet', input_shape = (299,299,3),
                               include_top=False, pooling='avg')

x = base_model.output

from keras.models import Model
from keras.layers import Dense
#  二分类分类器
predictions = Dense(1, activation='sigmoid')(x)
# this is the model we will train
model = Model(inputs=base_model.input, outputs=predictions)
```

3.3 图像增强

为了更好的训练模型，增加训练集的图片数量，我们利用 keras 官网提供的 ImageDataGenerator 对训练集合图片进行图像增强。

```
#图片数据增强
from keras.preprocessing.image import ImageDataGenerator
#训练数据增强
train_datagen = ImageDataGenerator( preprocessing_function=xception.preprocess_input,
                                    shear_range=0.2,
                                    zoom_range=0.2,
                                    horizontal_flip=True)
#验证数据增强
validation_datagen = ImageDataGenerator(preprocessing_function=xception.preprocess_input)

train_generator = train_datagen.flow(x = train_X,
```

```
                                                y = train_Y,
                                                batch_size = batch_size,
                                                shuffle=True)
validation_generator = validation_datagen.flow( x = val_X,
                                                  y = val_Y,
                                                  batch_size = batch_size,
                                                  shuffle=False)
```

3.4 执行过程

卷积神经网络中前几层学到的是通用特征（例如色彩、边缘、简单的图形等）这些通用特征对很多任务都适用，后面几层提取到的是特定类别相关的特征，因此这里我们只对模型的后边 4 个 block 进行 fine-tune。（以下所有运行数据是在 aws 云的 p2.xlarge 主机训练所得）。

所有训练中，epochs 设置为 30，batch_size 设置为 32，并利用 keras 中的 EarlyStopping 回调函数，当 val_loss 经过 3 个训练轮数不在优化时停止训练（即 patience 设置为 3），并保存每个 epoch 训练后的 model。

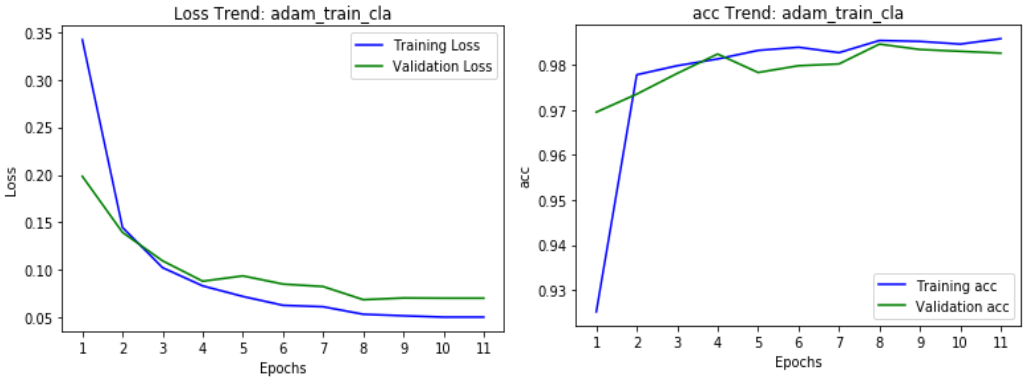
- **模型一：**

起初，我选取了 adam 优化器，lr 设置为 0.0001，fine-tune 模型。模型训练分三个步骤：

第一步，只训练顶端的二分类分类器：

```
Epoch 1/30
624/624 [==============================] - 539s 864ms/step - loss: 0.3420 - acc: 0.9252 - val_loss: 0.1985 - val_acc: 0.9695
Epoch 2/30
624/624 [==============================] - 524s 840ms/step - loss: 0.1448 - acc: 0.9778 - val_loss: 0.1396 - val_acc: 0.9735
Epoch 3/30
624/624 [==============================] - 511s 818ms/step - loss: 0.1025 - acc: 0.9798 - val_loss: 0.1097 - val_acc: 0.9781
Epoch 4/30
624/624 [==============================] - 508s 813ms/step - loss: 0.0834 - acc: 0.9813 - val_loss: 0.0883 - val_acc: 0.9824
Epoch 5/30
624/624 [==============================] - 508s 814ms/step - loss: 0.0723 - acc: 0.9832 - val_loss: 0.0939 - val_acc: 0.9783
Epoch 6/30
624/624 [==============================] - 508s 814ms/step - loss: 0.0630 - acc: 0.9839 - val_loss: 0.0853 - val_acc: 0.9798
Epoch 7/30
624/624 [==============================] - 507s 813ms/step - loss: 0.0615 - acc: 0.9827 - val_loss: 0.0827 - val_acc: 0.9802
Epoch 8/30
624/624 [==============================] - 509s 816ms/step - loss: 0.0536 - acc: 0.9854 - val_loss: 0.0689 - val_acc: 0.9846
Epoch 9/30
624/624 [==============================] - 508s 814ms/step - loss: 0.0520 - acc: 0.9852 - val_loss: 0.0707 - val_acc: 0.9834
Epoch 10/30
624/624 [==============================] - 510s 817ms/step - loss: 0.0506 - acc: 0.9846 - val_loss: 0.0705 - val_acc: 0.9830
Epoch 11/30
624/624 [==============================] - 509s 816ms/step - loss: 0.0474 - acc: 0.9858 - val_loss: 0.0697 - val_acc: 0.9826

<keras.callbacks.History at 0x7ff079f06470>
```

训练历经 11 个 epoch，用时 5641s，loss 与 acc 变化如下图所示：



第二步，选取上一步中表现最好的（val_loss 最低的）epoch 训练后的模型（epoch8 所对应的模型），放开 xception 的 13/14block（顶部 2 个 block）继续训练：

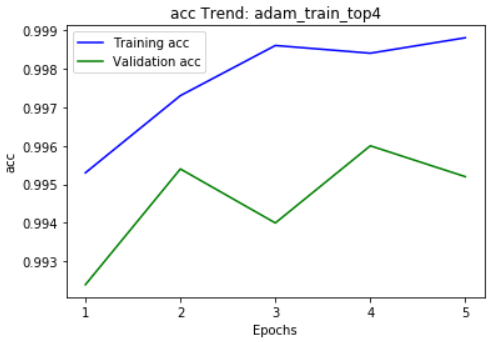
```
Epoch 1/30
624/624 [==============================] - 591s 947ms/step - loss: 0.0314 - acc: 0.9887 - val_loss: 0.0171 - val_acc: 0.9948
Epoch 2/30
624/624 [==============================] - 587s 941ms/step - loss: 0.0161 - acc: 0.9943 - val_loss: 0.0139 - val_acc: 0.9962
Epoch 3/30
624/624 [==============================] - 587s 940ms/step - loss: 0.0088 - acc: 0.9971 - val_loss: 0.0144 - val_acc: 0.9954
Epoch 4/30
624/624 [==============================] - 587s 940ms/step - loss: 0.0065 - acc: 0.9978 - val_loss: 0.0159 - val_acc: 0.9960
Epoch 5/30
624/624 [==============================] - 587s 940ms/step - loss: 0.0056 - acc: 0.9978 - val_loss: 0.0249 - val_acc: 0.9936

<keras.callbacks.History at 0x7ff079ec7fd0>
```

训练历经 5 个 epoch，用时 2939s，loss 与 acc 变化如下图所示：

第三步，选取上一步中表现最好的那个 epoch 训练后的模型（epoch2 所对应的模型），放开 xception 顶部 4 个 blcok 继续训练：

```
Epoch 1/30
624/624 [==============================] - 697s 1s/step - loss: 0.0134 - acc: 0.9953 - val_loss: 0.0265 - val_acc: 0.9924
Epoch 2/30
624/624 [==============================] - 690s 1s/step - loss: 0.0082 - acc: 0.9973 - val_loss: 0.0143 - val_acc: 0.9954
Epoch 3/30
624/624 [==============================] - 691s 1s/step - loss: 0.0040 - acc: 0.9986 - val_loss: 0.0186 - val_acc: 0.9940
Epoch 4/30
624/624 [==============================] - 691s 1s/step - loss: 0.0041 - acc: 0.9984 - val_loss: 0.0171 - val_acc: 0.9960
Epoch 5/30
624/624 [==============================] - 692s 1s/step - loss: 0.0036 - acc: 0.9988 - val_loss: 0.0181 - val_acc: 0.9952
```

训练历经 5 个 epoch，用时 3461s，loss 与 acc 变化如下图所示：



选取 epoch2 对应的模型为本次训练最终模型，这里记为模型一。我们可以看到整个训练用时 200 多分钟，所得模型对应的 val_loss 为 0.0143。

模型取得的 val_loss 比较满意，但用时较长，如果提高学习率是否可以在保证 val_loss 不变或者取得更优异的成绩的前提下缩短训练时间呢？

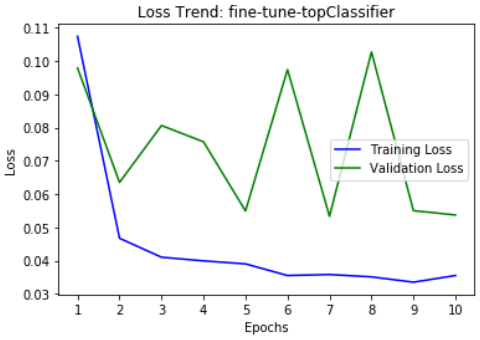
- **模型二：**

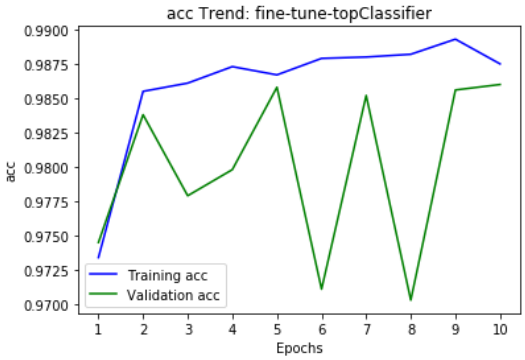
选取 adam 优化器，lr 设置为 0.001，fine-tune 模型。模型训练分三个步骤：

第一步，只训练顶端的二分类分类器：

```
Epoch 1/20
624/624 [==============================] - 496s 795ms/step - loss: 0.1074 - acc: 0.9734 - val_loss: 0.0979 - val_acc: 0.9745
Epoch 2/20
624/624 [==============================] - 486s 778ms/step - loss: 0.0467 - acc: 0.9855 - val_loss: 0.0635 - val_acc: 0.9838
Epoch 3/20
624/624 [==============================] - 488s 782ms/step - loss: 0.0410 - acc: 0.9861 - val_loss: 0.0806 - val_acc: 0.9779
Epoch 4/20
624/624 [==============================] - 484s 775ms/step - loss: 0.0399 - acc: 0.9873 - val_loss: 0.0757 - val_acc: 0.9798
Epoch 5/20
624/624 [==============================] - 492s 788ms/step - loss: 0.0390 - acc: 0.9867 - val_loss: 0.0549 - val_acc: 0.9858
Epoch 6/20
624/624 [==============================] - 483s 775ms/step - loss: 0.0355 - acc: 0.9879 - val_loss: 0.0974 - val_acc: 0.9711
Epoch 7/20
624/624 [==============================] - 491s 787ms/step - loss: 0.0358 - acc: 0.9880 - val_loss: 0.0533 - val_acc: 0.9852
Epoch 8/20
624/624 [==============================] - 484s 776ms/step - loss: 0.0351 - acc: 0.9882 - val_loss: 0.1027 - val_acc: 0.9703
Epoch 9/20
624/624 [==============================] - 484s 776ms/step - loss: 0.0335 - acc: 0.9893 - val_loss: 0.0550 - val_acc: 0.9856
Epoch 10/20
624/624 [==============================] - 491s 787ms/step - loss: 0.0355 - acc: 0.9875 - val_loss: 0.0537 - val_acc: 0.9860

<keras.callbacks.History at 0x7f1d98c55908>
```

训练历经 10 个 epoch，用时 4879s，loss 与 acc 变化如下图所示：

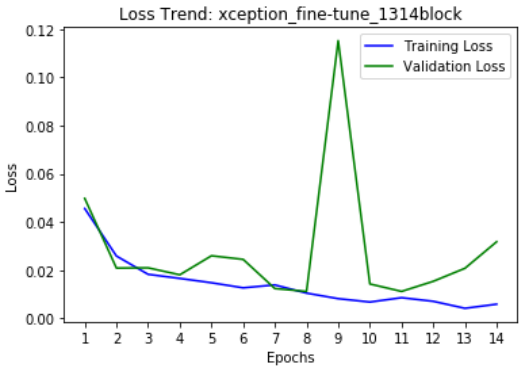


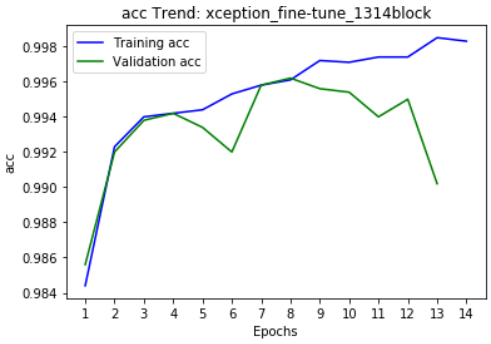


第二步，选取上一步中表现最好的（val_loss 最低的）epoch 训练后的模型（epoch7 所对应的模型），放开 xception 的 13/14block（顶部 2 个 block）继续训练：

```
Epoch 1/20
624/624 [==============================] - 567s 909ms/step - loss: 0.0456 - acc: 0.9844 - val_loss: 0.0498 - val_acc: 0.9856
Epoch 2/20
624/624 [==============================] - 560s 897ms/step - loss: 0.0259 - acc: 0.9923 - val_loss: 0.0209 - val_acc: 0.9920
Epoch 3/20
624/624 [==============================] - 560s 897ms/step - loss: 0.0183 - acc: 0.9940 - val_loss: 0.0210 - val_acc: 0.9938
Epoch 4/20
624/624 [==============================] - 560s 897ms/step - loss: 0.0166 - acc: 0.9942 - val_loss: 0.0181 - val_acc: 0.9942
Epoch 5/20
624/624 [==============================] - 559s 896ms/step - loss: 0.0148 - acc: 0.9944 - val_loss: 0.0260 - val_acc: 0.9934
Epoch 6/20
624/624 [==============================] - 558s 895ms/step - loss: 0.0127 - acc: 0.9953 - val_loss: 0.0245 - val_acc: 0.9920
Epoch 7/20
624/624 [==============================] - 560s 897ms/step - loss: 0.0139 - acc: 0.9958 - val_loss: 0.0124 - val_acc: 0.9958
Epoch 8/20
624/624 [==============================] - 560s 897ms/step - loss: 0.0105 - acc: 0.9961 - val_loss: 0.0112 - val_acc: 0.9962
Epoch 9/20
624/624 [==============================] - 558s 895ms/step - loss: 0.0082 - acc: 0.9972 - val_loss: 0.1153 - val_acc: 0.9761
Epoch 10/20
624/624 [==============================] - 563s 903ms/step - loss: 0.0068 - acc: 0.9971 - val_loss: 0.0143 - val_acc: 0.9956
Epoch 11/20
624/624 [==============================] - 574s 921ms/step - loss: 0.0086 - acc: 0.9974 - val_loss: 0.0112 - val_acc: 0.9954
Epoch 12/20
624/624 [==============================] - 571s 915ms/step - loss: 0.0071 - acc: 0.9974 - val_loss: 0.0154 - val_acc: 0.9940
Epoch 13/20
624/624 [==============================] - 574s 921ms/step - loss: 0.0042 - acc: 0.9985 - val_loss: 0.0208 - val_acc: 0.9950
Epoch 14/20
624/624 [==============================] - 574s 921ms/step - loss: 0.0059 - acc: 0.9983 - val_loss: 0.0318 - val_acc: 0.9902

<keras.callbacks.History at 0x7f157833d470>
```

训练历经 14 个 epoch，用时 7898s，loss 与 acc 变化如下图所示：

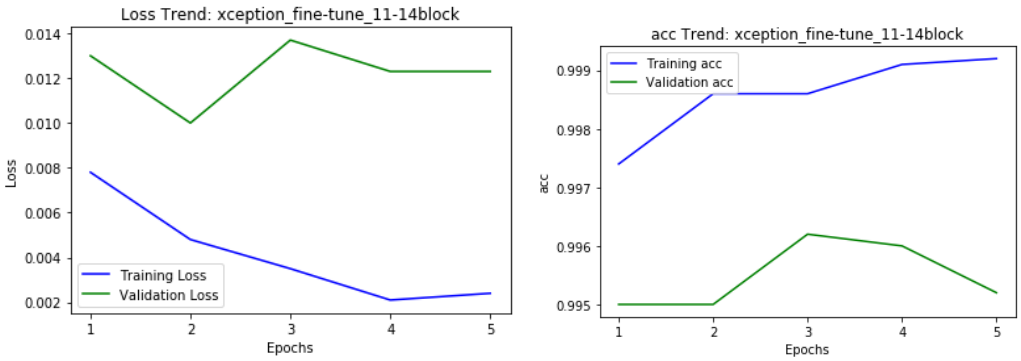




第三步，选取上一步中表现最好的那个 epoch 训练后的模型（epoch8 所对应的模型），放开 xception 顶部 4 个 blcok 继续训练：

```
Epoch 1/20
624/624 [==============================] - 687s 1s/step - loss: 0.0078 - acc: 0.9974 - val_loss: 0.0130 - val_acc: 0.9950
Epoch 2/20
624/624 [==============================] - 676s 1s/step - loss: 0.0048 - acc: 0.9986 - val_loss: 0.0100 - val_acc: 0.9954
Epoch 3/20
624/624 [==============================] - 676s 1s/step - loss: 0.0035 - acc: 0.9990 - val_loss: 0.0137 - val_acc: 0.9962
Epoch 4/20
624/624 [==============================] - 677s 1s/step - loss: 0.0021 - acc: 0.9991 - val_loss: 0.0123 - val_acc: 0.9960
Epoch 5/20
624/624 [==============================] - 677s 1s/step - loss: 0.0024 - acc: 0.9992 - val_loss: 0.0119 - val_acc: 0.9952

<keras.callbacks.History at 0x7f1554b7dc18>
```

训练历经 5 个 epoch，用时 3393s，loss 与 acc 变化如下图所示：



选取 epoch2 对应的模型为本次训练最终模型，这里记为模型二。我们可以看到整个训练用时 269 分钟，所得模型对应的 val_loss 为 0.0100。模型二在验证集上 val_loss 比模型一更低，但在训练过程中呈现锯齿状波动切用时比模型一更长。利用模型二在测试集上进行预测，将结果提交到 kaggle 得到了 0.04734、top3.1%的成绩（1314 排名第 41 名）。

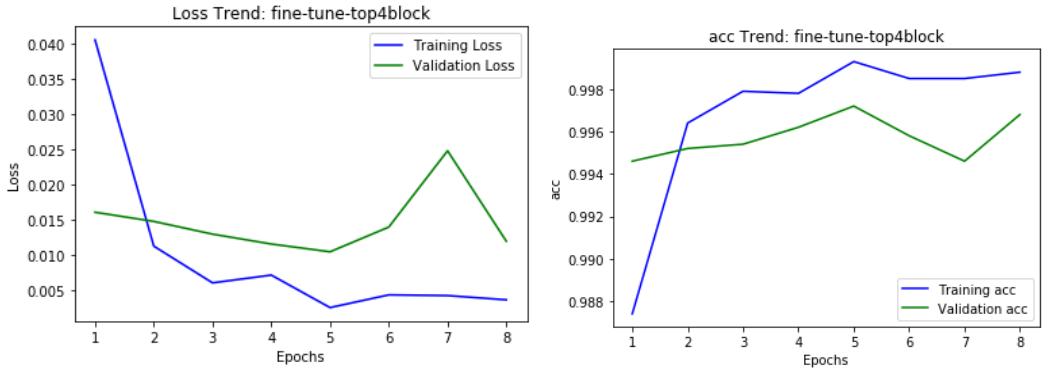
模型一、模型二都是分三步进行训练，每次只多放开 top 的两个 block，如果一次性放开 top4 个 block 进行训练是否可以保证 val_loss 不变或者取得更优异的成绩呢？

- **模型三：**

选取 adam 优化器，lr 设置为 0.0001，fine-tune 模型。一次性放开模型 top 的 4 个 block 进行训练：

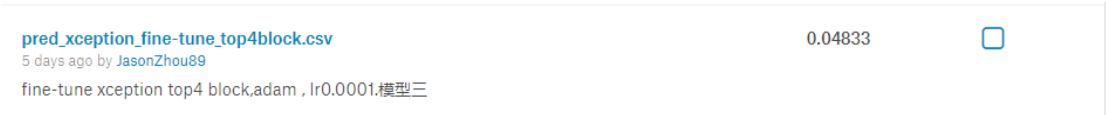
```
Epoch 1/30
624/624 [==============================] - 705s 1s/step - loss: 0.0405 - acc: 0.9874 - val_loss: 0.0161 - val_acc: 0.9946
Epoch 2/30
624/624 [==============================] - 696s 1s/step - loss: 0.0113 - acc: 0.9964 - val_loss: 0.0148 - val_acc: 0.9952
Epoch 3/30
624/624 [==============================] - 697s 1s/step - loss: 0.0061 - acc: 0.9979 - val_loss: 0.0130 - val_acc: 0.9954
Epoch 4/30
624/624 [==============================] - 697s 1s/step - loss: 0.0072 - acc: 0.9978 - val_loss: 0.0116 - val_acc: 0.9962
Epoch 5/30
624/624 [==============================] - 697s 1s/step - loss: 0.0026 - acc: 0.9993 - val_loss: 0.0105 - val_acc: 0.9972
Epoch 6/30
624/624 [==============================] - 696s 1s/step - loss: 0.0044 - acc: 0.9985 - val_loss: 0.0140 - val_acc: 0.9958
Epoch 7/30
624/624 [==============================] - 696s 1s/step - loss: 0.0043 - acc: 0.9985 - val_loss: 0.0248 - val_acc: 0.9946
Epoch 8/30
624/624 [==============================] - 696s 1s/step - loss: 0.0037 - acc: 0.9988 - val_loss: 0.0120 - val_acc: 0.9968

<keras.callbacks.History at 0x7faf62312908>
```

训练历经 8 个 epoch，用时 5580s，loss 与 acc 变化如下图所示：



选取 epoch5 对应的模型为本次训练最终模型，这里记为模型三。我们可以看到整个训练用时 93 分钟，所得模型对应的 val_loss 为 0.0105。

模型三在验证集上 val_loss 比模型二稍微高一点，但在训练过程用时比模型一、模型二要短很多。利用模型三在测试集上进行预测，将结果提交到 kaggle 得到了 0.04833、top3.8%的成绩（1314 排名第 50 名）。



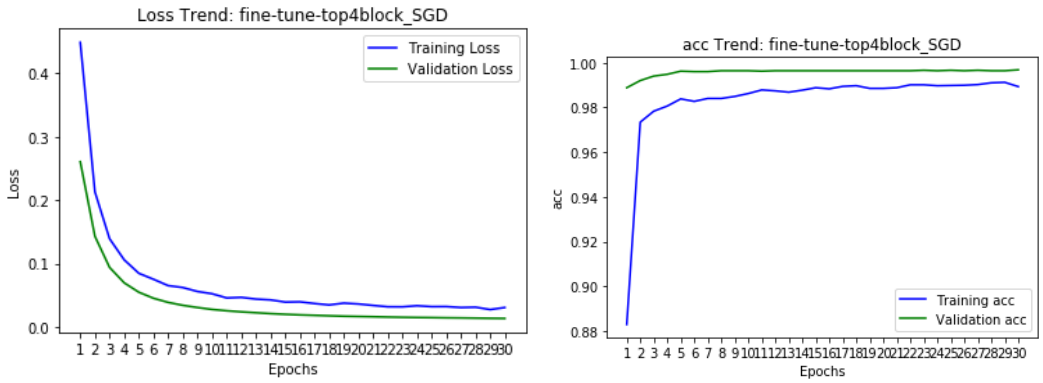
模型一、模型二、模型三都是选用 adam 优化器进行训练，如果选用其他优化器是否可以得到更好的成绩呢？

- **模型四：**

选取 SGD 优化器，lr 设置为 0.0001，momentum 设置为 0.9，fine-tune 模型。一次性放开模型 top 的 4 个 block 进行训练：

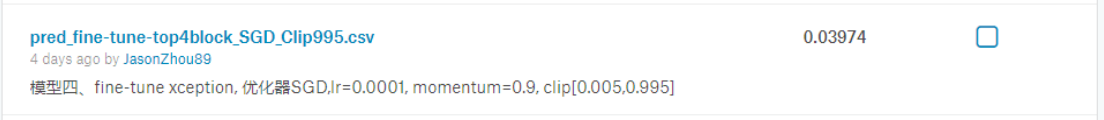
```
Epoch 19/30
624/624 [==============================] - 652s 1s/step - loss: 0.0377 - acc: 0.9885 - val_loss: 0.0171 - val_acc: 0.9964
Epoch 20/30
624/624 [==============================] - 652s 1s/step - loss: 0.0364 - acc: 0.9885 - val_loss: 0.0167 - val_acc: 0.9964
Epoch 21/30
624/624 [==============================] - 652s 1s/step - loss: 0.0341 - acc: 0.9888 - val_loss: 0.0163 - val_acc: 0.9964
Epoch 22/30
624/624 [==============================] - 652s 1s/step - loss: 0.0320 - acc: 0.9901 - val_loss: 0.0158 - val_acc: 0.9964
Epoch 23/30
624/624 [==============================] - 652s 1s/step - loss: 0.0319 - acc: 0.9901 - val_loss: 0.0154 - val_acc: 0.9966
Epoch 24/30
624/624 [==============================] - 652s 1s/step - loss: 0.0336 - acc: 0.9897 - val_loss: 0.0151 - val_acc: 0.9964
Epoch 25/30
624/624 [==============================] - 652s 1s/step - loss: 0.0322 - acc: 0.9898 - val_loss: 0.0149 - val_acc: 0.9966
Epoch 26/30
624/624 [==============================] - 652s 1s/step - loss: 0.0323 - acc: 0.9899 - val_loss: 0.0146 - val_acc: 0.9964
Epoch 27/30
624/624 [==============================] - 652s 1s/step - loss: 0.0308 - acc: 0.9902 - val_loss: 0.0144 - val_acc: 0.9966
Epoch 28/30
624/624 [==============================] - 653s 1s/step - loss: 0.0313 - acc: 0.9910 - val_loss: 0.0141 - val_acc: 0.9964
Epoch 29/30
624/624 [==============================] - 653s 1s/step - loss: 0.0276 - acc: 0.9912 - val_loss: 0.0138 - val_acc: 0.9964
Epoch 30/30
624/624 [==============================] - 653s 1s/step - loss: 0.0308 - acc: 0.9893 - val_loss: 0.0136 - val_acc: 0.9968
```

训练历经 30 个 epoch，用时 19588s，loss 与 acc 变化如下图所示：



选取 epoch30 对应的模型为本次训练最终模型，记为模型四。我们可以看到整个训练用时 326 分钟，所得模型对应的 val_loss 为 0.0136。

模型四在验证集上 val_loss 比模型二、模型三稍微高一点，训练时间要更长，但在训练过程 val_loss 平滑收敛。利用模型四在测试集上进行预测，将结果提交到 kaggle 得到了 0.03974、top1.1%的成绩（1314 排名第 15 名）。



这个成绩比预期的 top10%已经高出很多啦，我们将模型四作为项目最终模型。

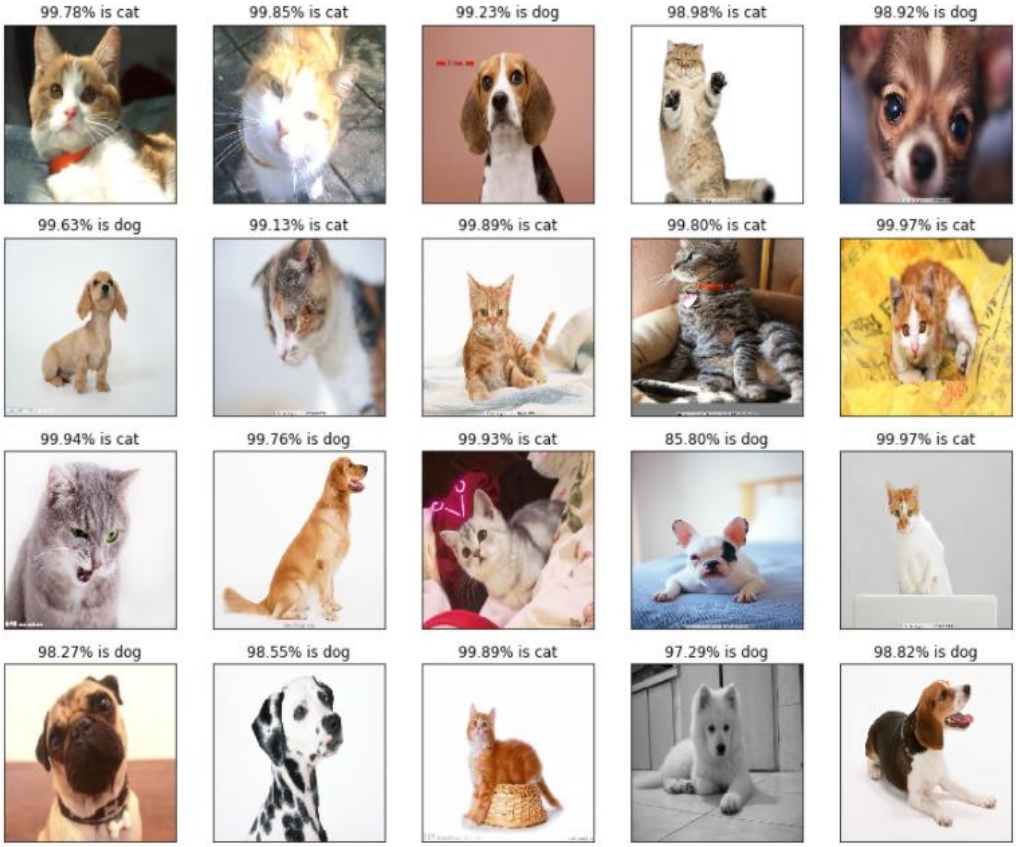
四. 结果

4.1 模型的评价与验证

我们将模型四最为最终模型，其训练过程中 val_loss 很平滑的收敛，并且在 kaggle 上得分达到了 0.003974 的成绩。

为了检验模型的鲁棒性，从网络上随机挑选了 20 张猫狗图片（非训练集、非验证集、非测试集中的图片）用最终模型进行预测，预测结果如下所示：

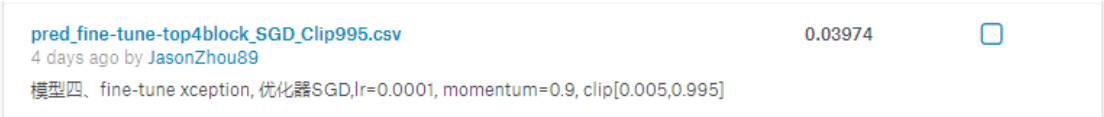
```
display_pred_img(robustness_image_list)
```



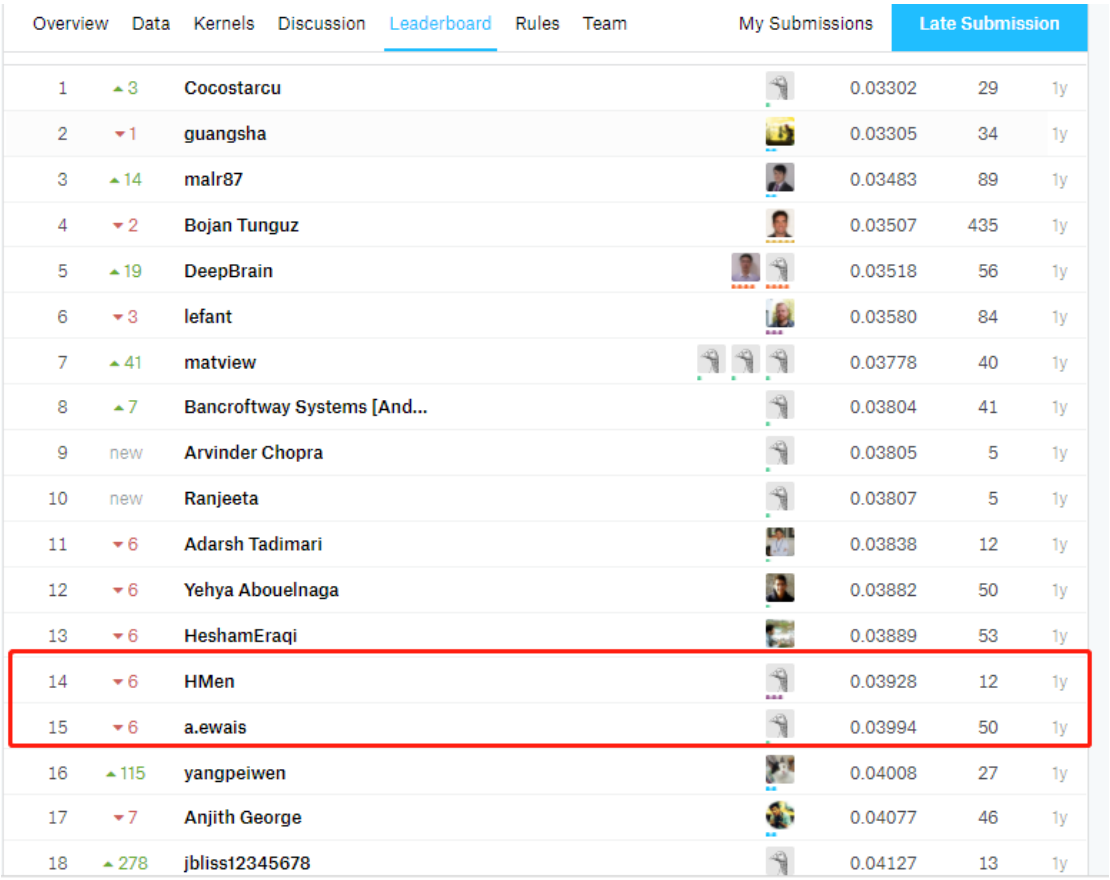
可以看出，20 个张图片全部预测正确，并且预测得分都很高，最终模型具有良好的泛化能力，在预测猫狗上足够稳健可靠。

4.2 合理性分析

最终模型在 kaggle 上得分为 0.03974：



得分超越了 top1.1%的成绩（1314 排名第 15 名），比项目开始制定的基准模型 top10%表现的要好。



最终模型已经完成了项目最初制定的目标，成功完成了“计算机可以正确的对不带标签、未曾见过的猫狗图片进行分类”的期望。

五. 项目结论

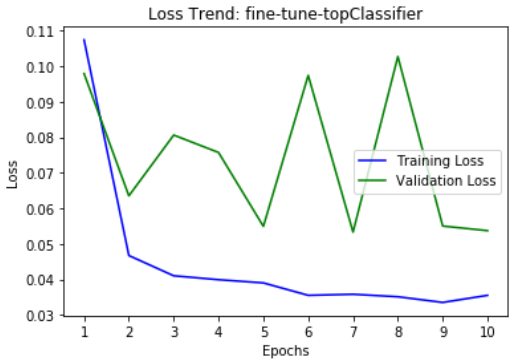
5.1 试验成绩对比结论

- 模型一、模型二对比：

采用相同优化器 adam，模型一 lr=0.0001，模型二 lr=0.01。分三步进行训练：第一步，只训练 top 顶端的分类器；第二步，取上一步最优模型，在此基础上放开 top2

个 blcok 继续训练；第三步，取上一步最优模型，在此基础上放开 top4 个 block 继续训练。

模型一、训练过程 loss 变化

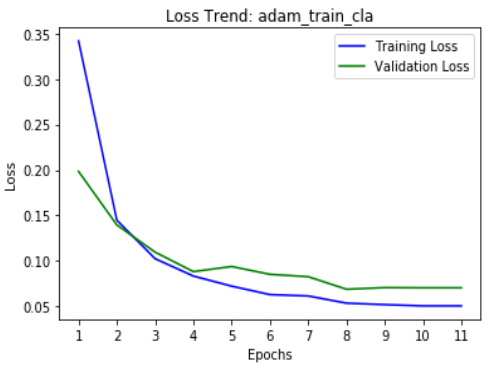


模型二、训练过程 loss 变化

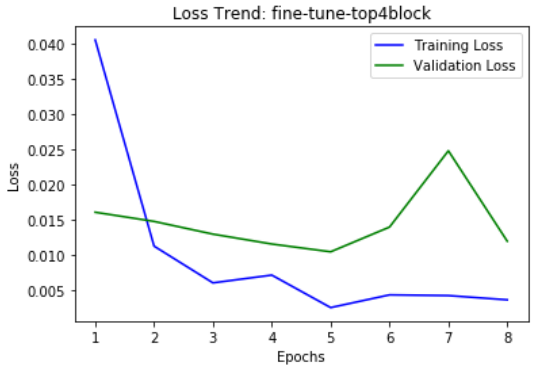
结论：lr 比较小的模型一收敛的比较平滑，lr 相对大的模型二在训练过程中锯齿抖动比较严重。相同模型的前提下，学习率越小模型训练越平滑。

- 模型一、模型三对比：

优化器采用 adam、其他参数完全一致，模型一是分三步进行训练，每次只多放开 top 的两个 block；模型三一次性放开 top4 个 block 进行训练。

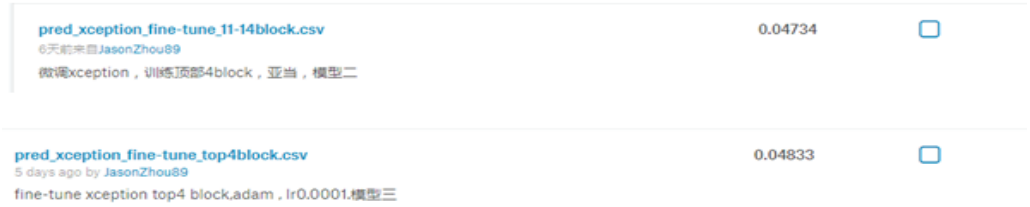


模型一、训练过程 loss 变化



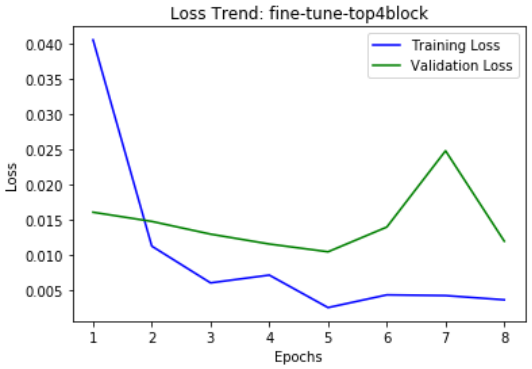
模型三、训练过程 loss 变化

实验一整个训练用时 200 多分钟，所得模型对应的 val_loss 为 0.0143；模型三整个训练用时 93 分钟，所得模型对应的 val_loss 为 0.0105，但 5 个 epoch 过后出现了过拟合现象。提交 kaggle 得分，模型二比模型三稍好一点。

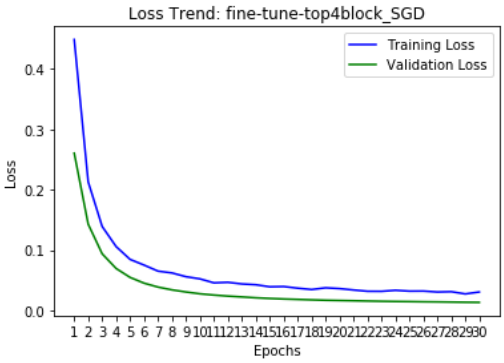


结论：相同优化器，相同参数的情况下，一次性放开顶端 4 个 block 取得的模型在测试集上泛化能力稍差一点，但模型训练用时更短。相对于模型在测试集上的 logloss 相差 0.001 的表现，训练时间大幅度的缩减相当可观。

- 模型三、模型四对比：模型三 adam 优化器、lr=0.0001，模型四 SGD 优化器、lr=0.0001、momentum=0.9，一次性放开 top4 个 block 进行训练。

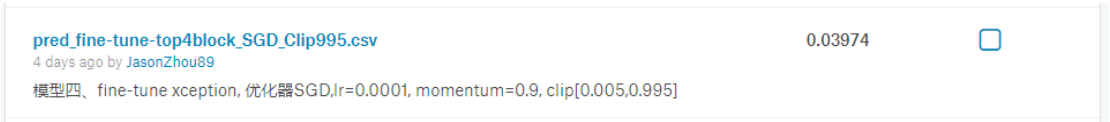


模型三、训练过程 loss 变化



模型四、训练过程 loss 变化

模型四在验证集上 val_loss 比模型三稍微高一点，训练时间要更长，但在训练过程 val_loss 平滑收敛，且没有出现像模型三训练过程中明显的过拟合现象。利用模型四在测试集上进行预测，将结果提交到 kaggle 得到了 0.03974、top1.1%的成绩（1314 排名第 15 名）。



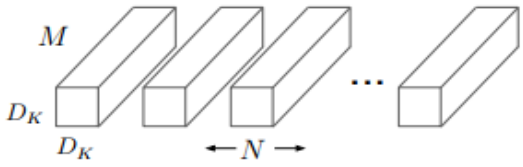
结论：学习率都为相同的情况下，SGD 模型的 loss 收敛更加平滑，其在测试集上的泛化能力更强；而 Adam 模型在取得相对不错成绩的基础上，模型训练所需用时更短。

模型四在测试集上取得的成绩比预期的 top10%已经高出很多，我们将模型四作为项目最终模型。

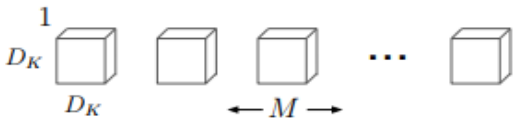
5.1 项目重要技术

本项目中 fine-tune Xception 模型取得了良好的成绩，并且其计算速度之快得力于 Xception 中的 depthwise separable convolution 结构。

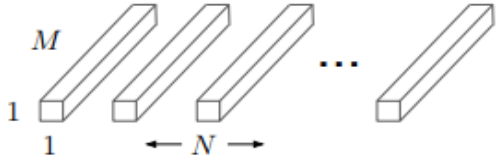
depthwise separable convolution 概念是在 mobileNets 中提出的，其核心概念是原来的卷积操作进行因子分解，即将原来的卷积操作（如下图 a 所示）分为两步：第一步，先用每一个卷积核只对输入的一个 channel 进行卷积，而不进行求和操作（如下图 b 所示）；第二步，对上一步得到的结果进行 pointwise 卷积操作（如下图 c 所示）。



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1 × 1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Figure 2. The standard convolutional filters in (a) are replaced by two layers: depthwise convolution in (b) and pointwise convolution in (c) to build a depthwise separable filter.

上图 a 中标准卷积操作的计算量为 Dk*Dk*M*Df*Df*N(Df*Df 特征图大小、N 为输出的 channels、M 为 input 的 channels、kernel 大小 Dk*Dk)；而 b 图中 Depthwise convolution 的计算量为 Dk*Dk*1*Df*Df*M，图 c 中 pointwise convolution 的计算量为 1*1*M*Df*Df*N。那么，对比标准卷积操作，depthwise separable convolution 的计算量减少了：

$$\frac{D_k * D_k * M * D_f * D_f + M * D_f * D_f * N}{D_k * D_k * M * D_f * D_f * N} = \frac{1}{N} + \frac{1}{{D_k}^2}$$

由上式可以看出，如果卷积核选为 3*3，那么计算量大概减少为标准卷积计算量的八分之一到九分之一。depthwise separable convolution 大大减少了模型中的参数，从而减少了模型的训练时间。

5.2 对项目的思考

项目归属于图像识别类，主要解决的问题是图像分类中的二分类问题。合适的模型选择与搭建是解决问题的关键。当然，无论是解决任何问题，搭建的是哪种模型，采用的是什么算法，对数据集的探索总是少不了的。将数据集规范化的输入、剔除掉部分异常值等预处理，并利用图像增强技术训练搭建的模型，尝试不同的超参数组合并选择表现能力最强的模型作为我们的最终模型。

整个项目过程中也遇到了不少的困难，比如：aws 云主机上环境的搭建、notebook 链接断开导致重新训练模型问题、各个深度学习模型的学习、keras 接口的学习、使用图像增强技术增加训练图片数量、调参等等。

令人兴奋的是，最终模型所取得的成绩大大超出了项目开始时制定的基准线。

5.3 需要作出的改进

项目中只是采用了单模型的训练预测，如果采用多个模型融合可能会取得更好的成绩。此外，虽然取得了不错的成绩，但整个模型的训练过程是漫长的，每次调参的会面临漫长的重新训练过程。如果先利用 model.predict_generator 函数来提取出训练集的特征向量（bottleneck features），然后再利用这些特征向量训练、调参、优化模型，会大大减少训练时间。

参考文献：

1. ImageNet Classification with Deep Convolutional Neural Networks. Alex etc， 2012.
2. Rethinking the Inception Architecture for Computer Vision. Christian Szegedy etc,2015.
3. Deep Residual Learning for Image Recognition. Kaiming He etc,2015.
4. Xception: Deep Learning with Depthwise Separable Convolutions. François Chollet, 2017.
5. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. Andrew G. Howard etc,2017.
6. Keras 中文文档：https://keras.io/zh
7. 项目所用数据集：https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition/data
8. 斯坦福大学课程：CS231n Convolutional Neural Networks for Visual Recognition.
9. An overview of gradient descent optimization algorithms. Sebastian Ruder,2017
10. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Sergey Ioffe etc, 2015.