# Udacity 机器学习工程师纳米学位

# 毕业项目报告(猫狗大战)

朱斌

04. 05. 2018

## 目录

# 1、问题的定义

## 1.1 项目概述

本毕业设计项目是 Kaggle 平台上于 2016-2017 年举办的一个计算机视觉领域的经典竞赛项目，目的是通过大量已标注图片的训练，使机器能够实现自动判别未标注图片是狗的概率。由于本数据集中的图片均为猫或者狗，本项目实质上也就是一个二分类的问题。

该比赛项目四年前已经在 Kaggle 上举办过一次，当时的评价指标是准确率，最好成绩是冠军的 0.98914，即 1000 张图中会有约 11 张错误。Kaggle 重新举办该比赛也部分是由于近年来机器学习，尤其是深度学习的高速发展，比如在 ILSVRC 分类比赛（共 1000 类）上 top5 错误率，2012 年时候 Geoffrey Hinton 领导的小组率先突破达到 16%[1]，到 2015 年深度学习算法在其上的错误率已经低于 4%[2]，这已经超过了人工标注的 5%错误率水平[3]。

同样受到深度学习在计算机视觉方面快速发展的振奋，同时借 Kaggle 平台重开比赛更可以进一步学习交流，最终决定选取“dogs vs cats”作为本项目的课题。本项目的数据集[4]由 Kaggle 提供，本项目输入数据均为图片，具体说来，数据图片分为两类——猫和狗，一共有 25000 张已经标注好标签的图片（标注是通过图片的名称给出的），其中猫和狗各占一半，另外有 12500 张未标注的测试图片用于测试。

## 1.2 问题描述

这个项目需要解决的问题十分直接明了，即是通过设计算法来对 25000 张标注好猫和狗的图片进行训练，最终预测待测试图片是狗的概率值（由于是二分类，1 为狗， 0 位猫）。

由于这是一个典型的二分类问题，且在计算机视觉方面已有一些经典的网络结构，如 VGG[5]、InceptionV3[6]、ResNet[2]、Xception[7]等可以参考，考虑到实际的计算能力和实现时间，可以先考虑采用迁移学习方案，特别是 ImageNet 数据集上的预训练结果可以供我们参考，当然这其中会涉及到对图片的预处理等问题。之后，我们可以对模型进行微调，还可以通过模型融合来进一步提高模型的预测能力。最后将预测的结果提交到 Kaggle 上计算分数，期望的结果自然是实际是狗的图片预测概率尽量是 1，否则为 0。

## 1.3 评价指标

按照 Kaggle 比赛的官方要求，本次比赛采用以下“对数损失函数”作为评价指标：

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

其中，

n 代表测试集的图片数；

$\hat{y}_i$代表预测图片是狗的概率；

$y_i$代表正确的类别概率（狗为 1，猫为 0）；

log 函数取自然底数。

最终的对数损失越小代表模型预测的效果越好。

# 2、分析

## 2.1 数据的探索

训练集和测试集的数据均为图片，只是训练集会有类别标签（标签是通过图片名给出的），部分训练集图片如图 1。

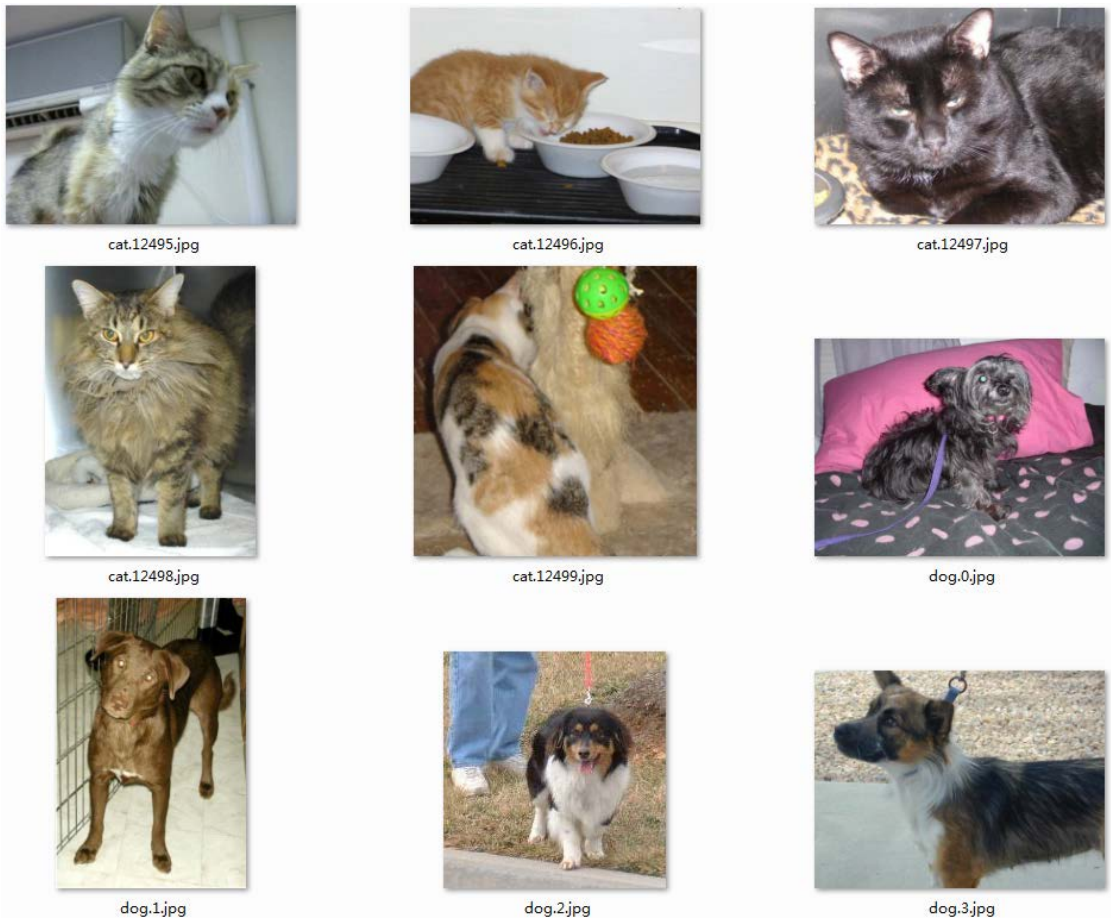


图 1 训练集样图

## 2.2 探索性可视化

具体的训练集和测试集图片类别数目可以参考图 2。值得注意的是，训练集中猫和狗种类数量均衡，不存在倾斜情况。

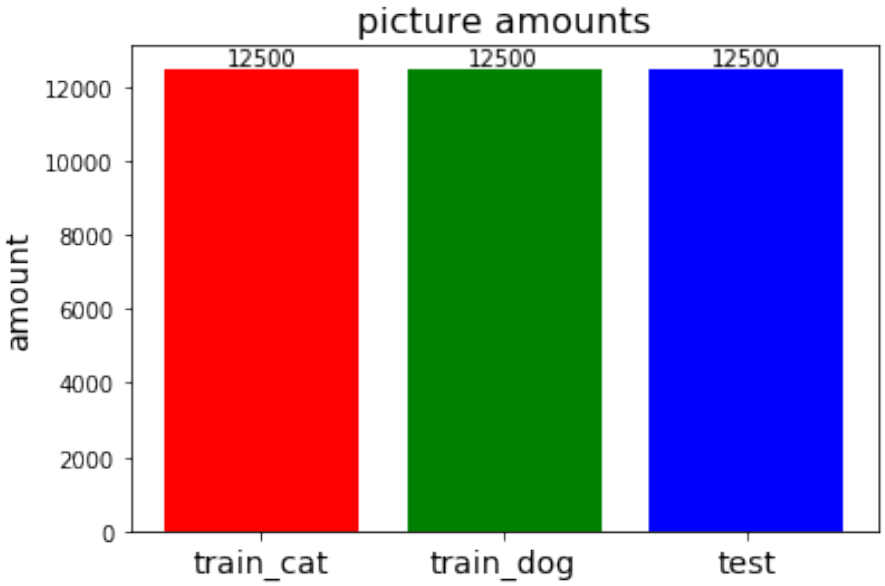


图 2 图片类别数目

另外，从图 1 也可以看出，这些图片为彩色图片，尺寸大小不一，并且图中的猫和狗种类繁多、颜色和姿态各异，图片背景也差异巨大，这些都会给模型的训练带来一定的难度。随机查看几张图片的具体信息如图 3。

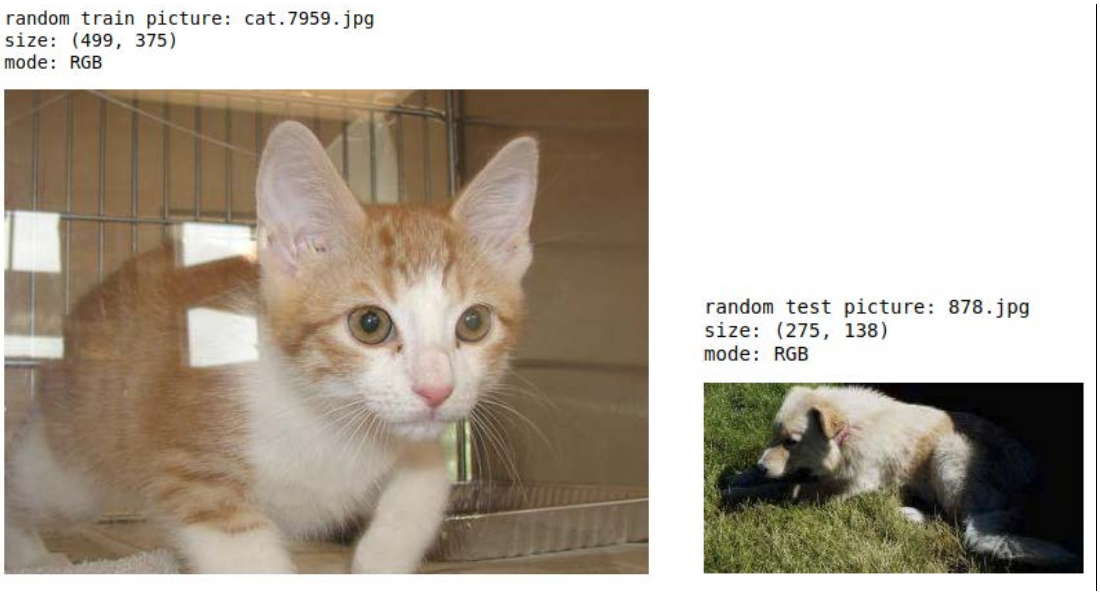


图 3（a）、（b） 随机查看训练集和测试集图片

## 2.3 算法和技术

“深度学习”听起来好像是一个近几年才出现的词汇，实则其历史可以追溯到上世纪 40 年代。彼时，深度学习的雏形出现在控制论中，最早的神经网络数学模型是由 Warren McCulloch 教授和 Walter Pitts 教授于 1943 年在论文[8]中提出的，Frank Rosenblatt 教授在 1958 年又提出了著名的 perceptron 模型，但当时的模型都是线性模型，无法解决异或问题，第一次神经网络浪潮也因此退去；在 80 年代，伴随着联结主义浪潮，神经网络又再次兴起，当时提出的反向传播算法甚至在现在依然是训练深度模型的主导方法，但在 90 年代末，传统的机器学习方法得以突破性进展，比如支持向量机（SVM）在手写体识别的错误率可以降到 0.8%，超过了当时的神经网络，也使得神经网络浪潮第二次退去；最终在 2006 年，Geoffrey Hinton 提出了“深度信念网络”的模型，以及伴随着其他算法、算力和数据的突破，神经网络得以“深度学习”之名第三次复兴[3, 9]。

2009 年，深度学习被引入语音识别领域，产生了巨大的影响，在 TIMIT 数据集上错误率从传统 GMM 模型的 21.7%降到了 17.9%；2012 年，在 ILSVRC 图像分类比赛上，借助于深度学习技术，Geoffrey 教授小组的 AlexNet 模型将 top5 错误率从传统方法的 26%降到 16%。近年来，深度学习计算机视觉、语音识别、机器翻译、医学信息处理、机器人及其控制、自然语言处理、网络安全等领域都取得了超过传统机器学习方法的能力[10]。

由于深度学习的模型都是多层的神经网络，网络结构的构建自然是一个重要的环节，这些年来，也出现了很多经典的神经网络结构，比如 CNN（Convolutional Neural Network，卷积神经网络），RNN（Recurrent Neural Network，循环神经网络），LSTM（Long Short Term Memory，长短期记忆网络），GAN（Generative Adversarial Network，生成式对抗网络），DRL（Deep Reinforcement Learning，深度强化学习）。

其中，CNN 是一种在深度学习中广泛使用的神经网络结构，在计算机视觉领域更是有着出色的效果。在卷积神经网络中，一个卷积层可以有多个不同的卷积核（kernel），而每个卷积核在输入图像上滑动且每次只处理一小块图像，如图 4[11]所示。

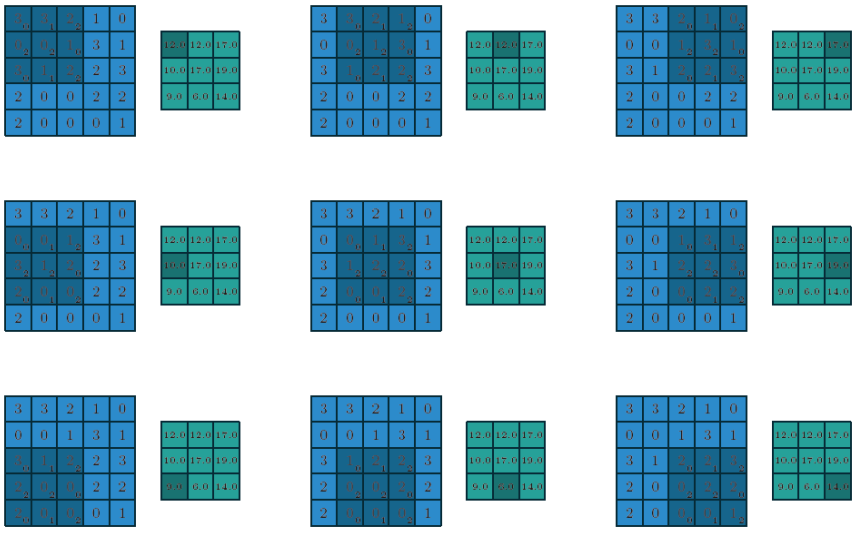

图 4 CNN 卷积操作示例

这样输入端的卷积层可以首先提取到图像中最基础的特征，再逐层组合抽象成更高阶的特征，比如从线段，到形状，再到鼻子，直至一张完整脸，即每个卷积层提取的特征，在后面的层中都会抽象组合成更高阶的特征。

卷积本是一种特殊的线性运算，它主要通过稀疏交互（sparse interactions）、参数共享（parameter sharing）、等变表示（equivarient representations）三个思想来改进机器学习系统。稀疏交互指其层间不同于传统神经网络每个单元间都会有交互，而仅和部分单元发生交互，这样不仅减少了我们的存储需求，也提高了统计和计算效率；权值共享指的是一个模型中的多个函数使用相同的参数，用于一个输入的权重也会绑定到其他权重上，保证了我们只需要学习一个参数合集而不是对每个位置都单独学习一个参数合集，也进一步提高了存储需求；等变性是指如果一个函数输入改变，其输出也会以同样方式发生改变，具体表现在 CNN 中就是如果我们移动输入中的图像，它的表示也会在输出中移动相同的量[9]。

由于卷积操作的特殊性，相比于全连接网络，CNN 也可以看作一个权重加了无限强先验得全连接网络。这个先验是指一个隐藏单元的权重必须和它邻居的权重相同，但可以在空间上移动[9]。

也因此，CNN 相比于全连接网络，不仅存储量小很多，提高了计算效率，还对缩放、平移、旋转等具有不变形，具有很强的泛化能力。CNN 在实际比赛，比如 ImageNet 相关的竞赛中都有着广泛的运用，也取得了优秀的效果，，比如前述 VGG、InceptionV3、ResNet、Xception 均为 CNN 结构，VGG 和 ResNet 为往年 ILSVRC 比赛的冠亚军。本项目也拟采用 CNN 这种神经网络结构。

具体在实现上，TensorFlow 是 Google 研发的第二代人工智能学习系统，也是目前使用人数最多的开发系统，是一个通过计算图的形式来表述计算的编程系统，Tensor 是指其数据结构为多维数组，Flow 直观地表达了 Tensor 间通过计算相互转化的过程，TensorFlow 中每一个计算都是计算图上的一个节点，而节点之间的边则描述了计算之间的依赖关系，如图 5 所示。

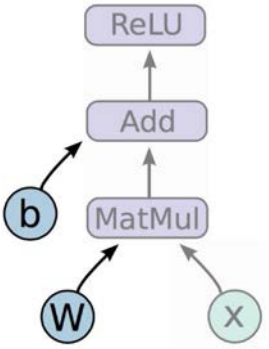


图 5 TensorFlow 计算图示例

Keras 是另一个被广泛使用的深度学习库，可以看作一个高层神经网络 API，由纯 python 写成，支持 Tensorflow 作为后端。Keras 具有友好的操作性和易扩展性，对于迁移学习也具有很好的支持，因此本项目选择使用 Keras 作为深度学习开发的主要依赖库。

## 2.4 基准模型

Kaggle 平台上本比赛的 public leaderboard 的分数可以作为参考，对于本项目至少前 10%（目前有 1314 组队），力争 top50，最好 top20，不设上限。其中第 131 名 logloss 分数为 0.06127，第 50 名为 0.04842，第 20 名为 0.04183，第一名为 0.03302。

# 3、方法

## 3.1 数据预处理

Keras 库目前支持的迁移学习有 VGG16、VGG19、ResNet50、InceptionV3、Inception_resnet_v2[12]、DenseNet[13]、Xception、MobileNet[14]、NasNet[15]共九种预训练模型。考虑到网上[16]已经有利用其中若干模型针对该比赛迁移训练好的特征向量，为了节约大量的时间，第一步可以直接利用这些特征向量连接全连接层后进行预测，有一个初始的成绩，第二步再使用较新的模型进行迁移学习，比如 Inception_resnet_v2，也进行一定的 finetune，最后再将模型进行融合，作为最终结果。

深度学习和传统机器学习的一个关键区别，就在于传统机器学习算法需要手动去提取和转换一些特征，而深度学习是通过多层的网络自动习得的[9]。但是对于本项目而言，一定的预处理还是必要的。由于决定使用迁移学习，因此对本比赛图像数据的预处理也必须和所迁移模型的预处理方式保持一致。

具体而言，在 Keras 中，对于 VGG16、VGG19 和 ResNet50，其默认的预处理模式是：RGB 模式转为 BGR 模式然后每个通道减去 ImageNet 数据集相应通道的均值[103.939, 116.779, 123.68]；对于 InceptionV3、Xception、Inception_resnet_v2、MobileNet，其默认的预处理模式是：归一化到（-1, 1）；对于 DenseNet，其默认预处理模式是：归一化到（0,1）后减去数据集均值[0.485, 0.456, 0.406]，再除以数据集标准差[0.229, 0.224, 0.225]。

默认输入照片分辨率上，Google Inception 系列均采用 299*299，其余一般为 224*224。在输入时也需要将图像 resize 到相应尺寸。

最后，由于需要对模型的效果进行评估，这里选择在训练集中分离出 20%，即 5000 张图片用作验证集。

## 3.2 执行过程

首先构建一个简单的分类网络结构，网络仅由简单的全连接层、dropout 层构成，利用网上已经训练好的特征向量，将其作为网络的输入，然后来训练该网络。在这一简单阶段，

为了加深对深度学习的理解，实际操作时，选择了对网络结构 Dense 层数及节点数目、batch_size 大小、优化器的类型三方面进行改变，也算是对网络结构的简单 finetune。

具体在这一阶段，使用到的几种训练好的特征向量分别是：VGG16、VGG19、ResNet50、InceptionV3、Xception 这五种模型在 ImageNet 上训练好的特征向量。下面简单说明一下这几种模型。VGGNet（包括 VGG16、VGG19）是牛津大学计算机视觉组（Visual Geometry Group）和 Google Deepmind 公司研究员一同研发的深度卷积网络，网络中反复堆叠 3*3 的小型卷积核和 2*2 的最大池化层，其使用多个小的卷积核代替一个大的卷积核，在较大程度较少参数量同时引入了更多的非线性变换。ResNet 是微软研究院提出的深层网络结构，通过引入残差网络解决了传统卷积层存在信息丢失的问题，也因此可以非常深层网络的训练。Google 开发的 Inception 系列则在大网络中设计小网络，后来提出了著名的可以起到一定正则化效果的 Batch Normalization，也综合了 VGG 模型的小卷积核优点。

然后再另外选择一个模型，进行较详细的 finetune，这里选择了 Inception_resnet_v2 模型。Inception_resnet_v2 是 Inception 系列的继续，正如其名，它的出现正是由于作者看到了将 Inception 和 ResNet 结合的可能性，它的一个典型模块如图 6 所示，左侧 relu 激活函数直接引过去的分支就是借鉴 ResNet 残差网路思想的体现。

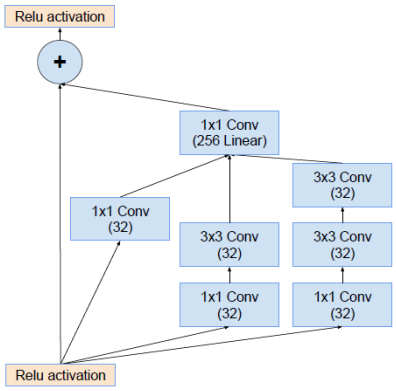


图 6 Inception_resnet_v2 网络中的典型模块

## 3.3 完善

本项目中，模型的完善主要可以分为两部分，一部分是将上述模型的结果进行融合，第二部分是参考[17]的一个小技巧，对预测的数据进行 clip 处理。

由于不同模型在测试集上通常不会在产生完全相同的误差，因此将不同模型进行融合通常会减小模型的泛化误差，并且如果基模型都是“好而不同”的，那么在模型融合后会有更好的泛化性能提升。

具体在 Keras 上，模型融合既可以使用 numpy.concatenate()方法把多个训练好特征向量合成一个，也可以通过 keras.layers.Concatenate()方法把创建不同模型的张量直接进行融合训练，当然可以直接对预测值进行平均（加权）融合。由于第二种方法对算力的要求比较高，在这里指选取了最后一种简单方法。

采用 clip 处理的出发点是由于本比赛采用的评价指标是 logloss，并且考虑到 kaggle 在

处理无穷大时采取的策略是将预测值 pred_value 全部替换为 max[min(pred_value, 1 - e-15), e-15][17]，这样对于错误的预测会有较大的惩罚，提交数据前选择将预测值全部直接 clip 到[0.005, 0.995]的区间，对于单个样本，loglos 计算公式为：$-y_i \log(\hat{y}_i) - (1 - y_i)\log(1 - \hat{y}_i)$。具体对于单个样本的 logloss 计算差别可见下表 1。

| | 真实值 | 预测值（kaggle 处理后） | logloss | Clip 后预测值 | Clip 后 logloss | Logloss 变化值 |
|---|---|---|---|---|---|---|
| 预测正确 | 1 | 1（1 - e-15） | e-15 | 0.995 | 0.005 | 0.005 |
| | 0 | 0（e-15） | e-15 | 0.005 | 0.005 | 0.005 |
| 预测错误 | 1 | 0（e-15） | 34.539 | 0.005 | 5.298 | -29.241 |
| | 0 | 1（1 – e-15） | 34.539 | 0.995 | 5.298 | -29.241 |

表 1 logloss 函数 clip 处理对照表

可以看到经过 clip 处理，对于预测正确的样本 logloss 得分上只会有稍微的降低，但是对于预测错误的样本 logloss 分值会有将大幅度的下拉，使得最终在所有样本上求平均时也会有更好的表现，这种区别本质上是因为 log 函数在接近 0 附近的梯度很大，而在 1 附近则较小。

# 4、结果

## 4.1 模型的评价与验证

### 4.1.1 单模型的调参

实验的第一部分将利用预训练好的权重直接输入到简单的分类网络，实验时一个是调节了 batch_size 的大小，观察对结果的影响，另一个是调节了分类网络的结构，主要是在 dropout 层前增加了一个 dense 层，并且 dense 层的节点数也有所调节。具体结果见表 2-表 5，此处实验用的机器是 cpu 为 i5-3210M，内存为 8G 的笔记本，没有用 gpu 加速。

| model | dropout+dense | batch_size | epochs_total | epoch_vl_0.0800 | epoch_vl_best | best_val_loss | val_acc | loss | acc | total_time |
|---|---|---|---|---|---|---|---|---|---|---|
| VGG19 | 0.5+1('s') | 64 | 30 | 6 | 14 | 0.0737 | 0.9724 | 0.1332 | 0.9504 | 23.15s |
| VGG19 | 0.5+1('s') | 128 | 30 | 8 | 14 | 0.0733 | 0.9728 | 0.1296 | 0.9513 | 13.71s |
| VGG19 | 0.5+1('s') | 256 | 30 | 9 | 14 | 0.0727 | 0.9748 | 0.1287 | 0.9498 | 8.55s |
| VGG19 | 32('r')+0.5+1('s') | 128 | 30 | 4 | 6 | 0.0732 | 0.9740 | 0.0724 | 0.9727 | 15.21s |
| VGG19 | 32('r')+0.5+1('s') | 256 | 30 | 5 | 6 | 0.0729 | 0.9742 | 0.0804 | 0.9697 | 10.49s |

| model | dropout+dense | batch_size | epochs_total | epoch_vl_0.0800 | epoch_vl_best | best_val_loss | val_acc | loss | acc | total_time |
|---|---|---|---|---|---|---|---|---|---|---|
| VGG16 | 0.5+1('s') | 64 | 30 | 8 | 23 | 0.0759 | 0.9704 | 0.1414 | 0.9470 | 22.10s |
| VGG16 | 0.5+1('s') | 128 | 30 | 8 | 25 | 0.0750 | 0.9712 | 0.1345 | 0.9474 | 12.65s |
| VGG16 | 0.5+1('s') | 256 | 30 | 11 | 23 | 0.0767 | 0.9692 | 0.1338 | 0.9490 | 9.04s |
| VGG16 | 32('r')+0.5+1('s') | 128 | 30 | 4 | 8 | 0.0750 | 0.9708 | 0.0694 | 0.9741 | 16.14s |
| VGG16 | 32('r')+0.5+1('s') | 256 | 30 | 8 | 8 | 0.0756 | 0.9712 | 0.0775 | 0.9699 | 10.13s |

表 2 VGG16 & VGG19 调参

| model | dropout+dense | batch_size | epochs_total | epoch_vl_0.0600 | epoch_vl_best | best_val_loss | val_acc | loss | acc | total_time | test_score |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ResNet50 | 0.5+1('s') | 32 | 30 | 6 | 11 | 0.0587 | 0.9770 | 0.0687 | 0.9750 | 52.30s | |
| ResNet50 | 0.5+1('s') | 64 | 30 | 6 | 6 | 0.0574 | 0.9776 | 0.0694 | 0.9734 | 32.23s | |
| ResNet50 | 0.5+1('s') | 128 | 30 | 6 | 20 | 0.0577 | 0.9766 | 0.0665 | 0.9741 | 23.50s | |
| ResNet50 | 0.5+1('s') | 256 | 30 | 12 | 21 | 0.0581 | 0.9760 | 0.0618 | 0.9751 | 18.79s | |
| ResNet50 | 0.5+1('s') | 512 | 30 | 17 | 28 | 0.0582 | 0.9752 | 0.0614 | 0.9752 | 17.48s | |
| ResNet50 | 0.5+1('s') | 1024 | 30 | 21 | 28 | 0.0584 | 0.9752 | 0.0629 | 0.9748 | 15.15s | |
| ResNet50 | 32('r')+0.5+1('s') | 128 | 30 | 4 | 5 | 0.0557 | 0.9780 | 0.0569 | 0.9795 | 26.85s | |
| ResNet50 | 32('r')+0.5+1('s') | 1024 | 30 | 10 | 25 | 0.0560 | 0.9764 | 0.0485 | 0.9818 | 12.88s | |
| ResNet50 | 64('r')+0.5+1('s') | 128 | 30 | 4 | 5 | 0.0567 | 0.9774 | 0.0574 | 0.9778 | 34.36s | |
| ResNet50 | 128('r')+0.5+1('s') | 128 | 30 | 4 | 4 | 0.0562 | 0.9770 | 0.0552 | 0.9784 | 53.48s | |
| ResNet50 | 128('r')+0.5+1('s') | 256 | 30 | 8 | 19 | 0.0542 | 0.9782 | 0.0178 | 0.9936 | 35.91s | |
| ResNet50 | 128('r')+0.5+1('s') | 512 | 30 | 4 | 19 | 0.0539 | 0.9778 | 0.0259 | 0.9897 | 27.58s | |
| ResNet50 | 128('r')+0.5+1('s') | 1024 | 30(60) | 10 | 28 | 0.0530 | 0.9802 | 0.0311 | 0.9885 | 23.27s | |
| ResNet50 | 256('r')+0.5+1('s') | 1024 | 30 | 5 | 20 | 0.0519 | 0.9788 | 0.0365 | 0.9859 | 39.85s | |
| ResNet50 | 256('r')+0.5+1('s') | 2048 | 60 | 9 | 36 | 0.0527 | 0.9784 | 0.0373 | 0.9852 | - | |
| ResNet50 | 512('r')+0.5+1('s') | 512 | 30 | 4 | 15 | 0.0539 | 0.9780 | 0.0270 | 0.9909 | 83.41s | |
| ResNet50 | 512('r')+0.5+1('s') | 1024 | 30 | 8 | 24 | 0.0515 | 0.9784 | 0.0201 | 0.9946 | 67.21s | 0.07089 |
| ResNet50 | 1024('r')+0.5+1('s') | 1024 | 30 | 5 | 18 | 0.0520 | 0.9792 | 0.0274 | 0.9916 | 124.31s | |

表 3 ResNet50 调参

| model | dropout+dense | batch_size | epochs_total | epoch_vl_0.0230 | epoch_vl_best | best_val_loss | val_acc | loss | acc | total_time | test_score |
|---|---|---|---|---|---|---|---|---|---|---|---|
| InceptionV3 | 0.5+1('s') | 64 | 30 | 11 | 20 | 0.0223 | 0.9922 | 0.0162 | 0.9945 | 32.64s | |
| InceptionV3 | 0.5+1('s') | 128 | 30 | 2 | 14 | 0.0222 | 0.9924 | 0.0180 | 0.9942 | 25.11s | |
| InceptionV3 | 0.5+1('s') | 256 | 30 | 11 | 20 | 0.0226 | 0.9926 | 0.0174 | 0.9938 | 19.48s | |
| InceptionV3 | 32('r')+0.5+1('s') | 128 | 30 | 8 | 8 | 0.0230 | 0.9928 | 0.0092 | 0.9972 | 26.42s | |
| InceptionV3 | 32('r')+0.5+1('s') | 256 | 30 | 4 | 6 | 0.0222 | 0.9930 | 0.0141 | 0.9954 | 19.77s | 0.04755 |
| InceptionV3 | 32('r')+0.5+1('s') | 256 | 30 | 8 | 11 | 0.0224 | 0.9924 | 0.0130 | 0.9963 | 15.98s | |
| InceptionV3 | 128('r')+0.5+1('s') | 128 | 30 | - | 4 | 0.0245 | 0.9922 | 0.0126 | 0.9960 | 52.18s | |
| InceptionV3 | 128('r')+0.5+1('s') | 256 | 30 | - | 6 | 0.0242 | 0.9928 | 0.0103 | 0.9970 | 27.65s | |
| InceptionV3 | 64('r')+0.5+1('s') | 128 | 30 | - | 2 | 0.0233 | 0.9918 | 0.0215 | 0.9932 | 36.80s | |
| InceptionV3 | 16('r')+0.5+1('s') | 128 | 30 | - | 4 | 0.0264 | 0.9918 | 0.0246 | 0.9929 | 36.80s | |

表 4 InceptionV3 调参

| model | dropout+dense | batch_size | epochs_total | epoch_vl_0.0200 | epoch_vl_best | best_val_loss | val_acc | loss | acc | total_time | test_score (no clip) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Xception | 0.5+1('s') | 64 | 30 | 16 | 28 | 0.0194 | 0.9940 | 0.0147 | 0.9955 | 33.42s | |
| Xception | 0.5+1('s') | 128 | 30 | 17 | 26 | 0.0193 | 0.9938 | 0.0157 | 0.9944 | 23.64s | |
| Xception | 0.5+1('s') | 256 | 30 | 19 | 28 | 0.0196 | 0.9938 | 0.0150 | 0.9952 | 20.30s | |
| Xception | 32('r')+0.5+1('s') | 128 | 30 | - | 8 | 0.0202 | 0.9928 | 0.0092 | 0.9972 | 28.11s | |
| Xception | 32('r')+0.5+1('s') | 256 | 30 | 19 | 19 | 0.0194 | 0.9944 | 0.0096 | 0.9974 | 19.40s | |
| Xception | 32('r')+0.5+1('s') | 512 | 30 | 14 | 19 | 0.0198 | 0.9944 | 0.0153 | 0.9963 | 16.33s | |
| Xception | 128('r')+0.5+1('s') | 64 | 30 | 4 | 4 | 0.0190 | 0.9944 | 0.0151 | 0.9956 | 89.65s | |
| Xception | 128('r')+0.5+1('s') | 128 | 30 | 6 | 6 | 0.0184 | 0.9942 | 0.0086 | 0.9976 | 54.04s | |
| Xception | 128('r')+0.5+1('s') | 256 | 30 | 1 | 1 | 0.0220 | 0.9948 | 0.0015 | 0.9996 | 35.97s | |
| Xception | 64('r')+0.5+1('s') | 128 | 30 | 6 | 18 | 0.0189 | 0.9944 | 0.0061 | 0.9985 | 38.46s | |
| Xception | 256('r')+0.5+1('s') | 128 | 30 | 4 | 14 | 0.0185 | 0.9946 | 0.0049 | 0.9986 | 97.35s | |
| Xception | 256('r')+0.5+1('s') | 256 | 30 | 4 | 18 | 0.0183 | 0.9944 | 0.0056 | 0.9983 | 59.68s | |
| Xception | 256('r')+0.5+1('s') | 512 | 30 | 4 | 23 | 0.0182 | 0.9944 | 0.0073 | 0.9979 | 47.36s | 0.04213 (0.05868) |
| Xception | 256('r')+0.5+1('s') | 1024 | 50 | 12 | 35 | 0.0186 | 0.9942 | 0.0077 | 0.9979 | 67.75s | |
| Xception | 512('r')+0.5+1('s') | 128 | 30 | 4 | 4 | 0.0190 | 0.9942 | 0.0141 | 0.9955 | 170.00s | |

表 5 Xception 调参

表格的列名分别为模型名、分类网络细节（32('r')表示节点数为 32、使用 relu 激活的 dense 层，1('s')表示节点数为 1、使用 sigmoid 激活的 dense 层，0.5 表示 dropout 层断开比例，需要注意的是在本项目中对所有 base_mdel 的 output 都先进行了 GlobalAveragePooling2D，即将每个特征图池化为一个特征点）、batch_size 数量、epoches 数量、val_loss 达到某值的

epoch 数、val_loss 最好的 epoch 数、最好的 val_loss 值、最好 val_loss 值时的 val_acc\loss\acc、运行耗时，后三个表格还列出了 model 表现最好时预测测试集并提交到 Kaggle 上的 score（默认对提交结果进行 clip 处理，括号中为未 clip 处理的 score）。

这五个模型在横向对比上和他们在 ImageNet 分类上的表现类似，较新的模型的效果会更好。其中值得一提的是 Xception 模型，其在先进行 256 节点 dense，再 dropout0.5，再用 1 节点 dense 分类，训练时 batch_size 取 512，并对结果取 clip 处理，最后的单模型得分可以达到 0.04213，这已经可以在 Kaggle 的 public leaderboard 上排到 21 名，其模型结构如图 7 所示。IncepotionV3 的模型的 0.04755 可以排到 44 名；ResNet50 的 0.07089 可以排到 174 名；VGG 的模型相对较早，网络结构不够复杂，验证集上表现不是很好，也就没有提交。

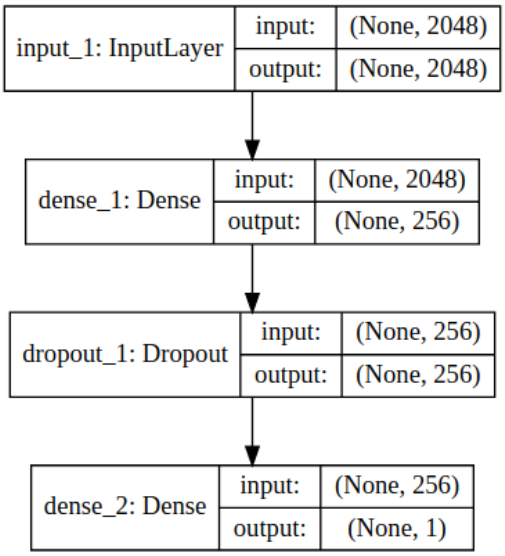


图 7 Xception 调参表现最好时模型结构

几乎所有的深度学习算法都使用到了随机梯度下降（stochastic gradient descent, SGD）及其变种算法，并且机器学习算法的目标函数不同于一般的优化算法，机器学习算法的目标函数通常可以分解为多个训练样本上的求和。传统的梯度下降算法使用全部样本来计算迭代的梯度，虽然这时样本的方差会很小，梯度会更精确，但是每次计算梯度使用全部样本显然计算代价太大，具体而言使用更多样本计算梯度的回报是$\sqrt{n}$倍于样本的数量，低于线性，是不划算的，而且方差太小会使得学习过程中的噪声太小，反而容易陷入鞍点等。另一方面，太小的样本量的梯度误差又会比较大，有可能噪音太大，而且也不易充分利用现在机器普遍的多核架构。因此一般在实际操作中，样本量即 batch_size 的值会取得适中，一般是几十到几百，而且对于二进制的电脑来说，2 的整数幂次会比较好。

从本实验的探索中也可以印证上述部分结论，比如表 2 中，对于 ResNet50 模型的前 6 行，网络结构均为一层 0.5 的 dropout 加一层 sigmoid 的分类层，batch_size 从 32 一直到 1024，每个 epoch 迭代次数更少时间更快，达到相同精度所需的 epoch 需要更多，而且最终精度会陷入不同的局部极值，但是当 batch_size 为 64 时，会有一个最终收敛精度上的最优。其他模型和网络的结论也是类似的。

另外，在这一步，也试着对分类网络的结构进行了小的尝试，发现的规律主要是在

dropout 前加一层 Dense 层会有较好的结果，特别是对于 ResNet50 和 Xception 会有小幅的提高，InceptionV3 虽然二者最好的 val_loss 相同，但是加入了 Dense 层可以更早地达到最优值。猜测主要是因为输入层的维度是 2048，和最终 Dense 到 1 维度差距较大，中间加一层过度的 Dense 之后，特征的抽取会更加平滑。

实验第一部分的另一个环节是对网络的优化器进行了尝试和调参。优化器是控制网络每轮迭代的学习率大小和方向的策略，目前主流的优化器有 SGD、AdaGrad、Adadelta 、RMSprop、Adam 等，这五种 keras 都支持，实验选择了这五种进行调试。对于 SGD 主要参数是学习率，后来引入的动量这一参数可以加速学习，另外也可以对学习率的大小进行控制；AdaGrad 结合了所有历史梯度，可以为每个参数适应不同的学习率；Adadelta 是对 AdaGrad 的扩展，梯度累计方式改为指数衰减；RMSprop 和 Adadelta 一样也使用指数衰减以丢弃遥远过去的历史；Adam 则将动量引入到梯度的估计。表 4 是用 Xception 模型更换不同优化器调参的结果，分类网络结构取上述 Xception 表现最好情况，此处运行使用的机器是 GTX1060 6G 显卡的笔记本，tensorflow 使用 gpu 版本。

| model | optimizer | dropout+dense | batch_size | epochs_total | epoch_vl_0.0200 | epoch_vl_best | best_val_loss | val_acc | total_time | test_score (no clip) |
|---|---|---|---|---|---|---|---|---|---|---|
| Xception | SGD(0.1,0.9,0.9) | 256('r')+0.5+1('s') | 256 | 700 | - | 700 | 0.0302 | 0.9922 | - | |
| Xception | SGD(0.1,0.9,0.1) | 256('r')+0.5+1('s') | 256 | 700 | - | 700 | 0.0230 | 0.9946 | - | |
| Xception | SGD(0.1,0.9, 0) | 256('r')+0.5+1('s') | 256 | 30 | 6 | 12 | 0.0185 | 0.9942 | - | |
| Xception | SGD(0.2,0.9, 0) | 256('r')+0.5+1('s') | 256 | 30 | 3 | 9 | 0.0185 | 0.9942 | - | |
| Xception | SGD(0.01,0.9,0) | 256('r')+0.5+1('s') | 256 | 30(100) | 18 | 55 | 0.0189 | 0.9942 | 21.88s | |
| Xception | SGD(0.1,0.7, 0) | 256('r')+0.5+1('s') | 256 | 30 | 6 | 14 | 0.0184 | 0.9940 | 21.45s | 0.04115(0.05502) |
| Xception | SGD(0.1,0.5, 0) | 256('r')+0.5+1('s' ) | 256 | 30(60) | 8 | 25 | 0.0186 | 0.9944 | 21.40s | |
| Xception | adagrad | 0.5+1('s') | 128 | 30(300) | 41 | 233 | 0.0190 | 0.9942 | 30.03s | |
| Xception | adagrad | 256('r')+0.5+1('s') | 16 | 30 | 4 | 6 | 0.0186 | 0.9942 | 187.52s | |
| Xception | adagrad | 256('r')+0.5+1('s') | 32 | 30 | 4 | 6 | 0.0181 | 0.9940 | 98.37s | 0.04172(0.05625) |
| Xception | adagrad | 256('r')+0.5+1('s') | 64 | 30 | 4 | 9 | 0.0185 | 0.9940 | 54.06s | |
| Xception | adagrad | 256('r')+0.5+1('s') | 128 | 30 | 4 | 14 | 0.0188 | 0.9946 | 31.04s | |
| Xception | adagrad | 256('r')+0.5+1('s') | 256 | 30 | 6 | 12 | 0.0191 | 0.9938 | 21.32s | |
| Xception | adadelta | 0.5+1('s') | 128 | 30 | 17 | 26 | 0.0193 | 0.9938 | 30.31s | |
| Xception | adadelta | 256('r')+0.5+1('s') | 128 | 30 | 7 | 14 | 0.0184 | 0.9946 | 33.64s | |
| Xception | adadelta | 256('r')+0.5+1('s') | 256 | 30 | 4 | 18 | 0.0183 | 0.9942 | 21.46s | |
| Xception | adadelta | 256('r')+0.5+1('s') | 512 | 30(100) | 6 | 22 | 0.0182 | 0.9942 | 16.09s | 0.04201(0.05814) |
| Xception | adadelta | 256('r')+0.5+1('s') | 1024 | 100 | 12 | 48 | 0.0186 | 0.9944 | 29.74s | |
| Xception | RMSprop(0.001) | 256('r')+0.5+1('s') | 512 | 30 | 4 | 12 | 0.0193 | 0.9948 | 12.53s | |
| Xception | RMSprop(0.0001) | 0.5+1('s') | 128 | 30(300) | 118 | 205 | 0.0197 | 0.9940 | 31.44s | |
| Xception | RMSprop(0.0001) | 256('r')+0.5+1('s') | 128 | 30(60) | 7 | 18 | 0.0190 | 0.9938 | 33.31s | |
| Xception | RMSprop(0.0001) | 256('r')+0.5+1('s') | 256 | 30(60) | 8 | 25 | 0.0184 | 0.9940 | 16.44s | |
| Xception | RMSprop(0.0001) | 256('r')+0.5+1('s') | 512 | 30(60) | 12 | 33 | 0.0183 | 0.9938 | 16.36s | 0.04199(0.05613) |
| Xception | RMSprop(0.0001) | 256('r')+0.5+1('s') | 1024 | 30(90) | 15 | 56 | 0.0184 | 0.9942 | 9.07s | |
| Xception | RMSprop(0.00001) | 256('r')+0.5+1('s') | 128 | 30(240) | 69 | 187 | 0.0193 | 0.9938 | 32.76s | |
| Xception | RMSprop(0.00001) | 256('r')+0.5+1('s') | 256 | 30(240) | 74 | 187 | 0.0189 | 0.9936 | 17.31s | |
| Xception | RMSprop(0.00001) | 256('r')+0.5+1('s') | 512 | 30 | - | 1 | 0.0274 | 0.9956 | 11.61s | |
| Xception | adam | 0.5+1('s') | 128 | 30(60) | 12 | 26 | 0.0188 | 0.9940 | 29.96s | |
| Xception | adam | 256('r')+0.5+1('s') | 128 | 30 | 3 | 6 | 0.0189 | 0.9946 | 32.19s | |
| Xception | adam | 256('r')+0.5+1('s') | 256 | 30 | 6 | 7 | 0.0184 | 0.9946 | 20.26s | 0.04224(0.05926) |
| Xception | adam | 256('r')+0.5+1('s') | 512 | 30 | 6 | 14 | 0.0187 | 0.9942 | 15.33s | |

表 6 Xception 应用不同 optimizer 的调参

表格列名的意义基本同表 2-表 5，不同的是 optimizer 列括号内表示的是具体参数，对 SGD 依次是学习率、动量、学习率衰减值，对 RMSprop 仅是学习率。并将五个 optimizer 各自最好的成绩提交 kaggle，它们在验证集上成绩排名是 Adagrad、Adadelta、RMSprop、SGD、Adam，提交测试集 noclip 排名是 SGD、RMSprop、Adagrad、Adadelta、Adam，clip 后是 SGD、Adagrad、RMSprop、Adadelta、Adam，在 kaggle public leaderboard 上名次依次可以达到 18、

20、20、21、22。横向对比来说，Adam 表现较差，SGD 是最老的模型但泛化能力却是最好的，但总体而言实际相差都不大，而且限于时间和我本人能力，每个模型也并不定调到了最好的效果。

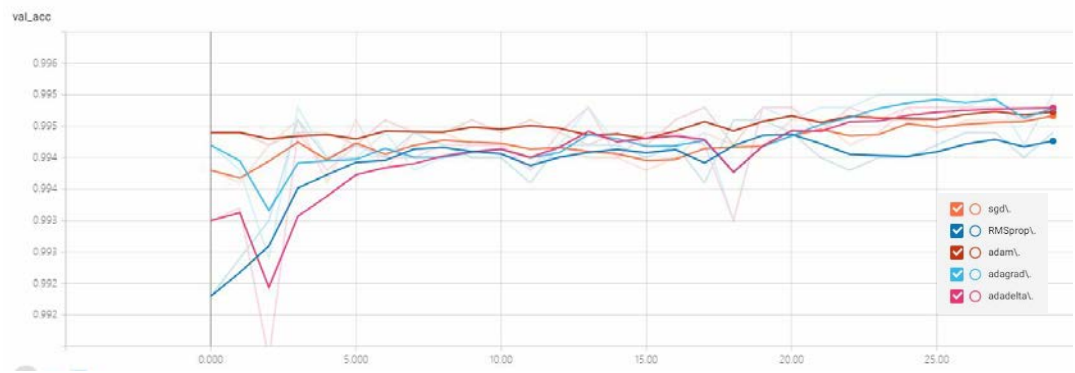


图 8 不同 optimizer 的 val_acc

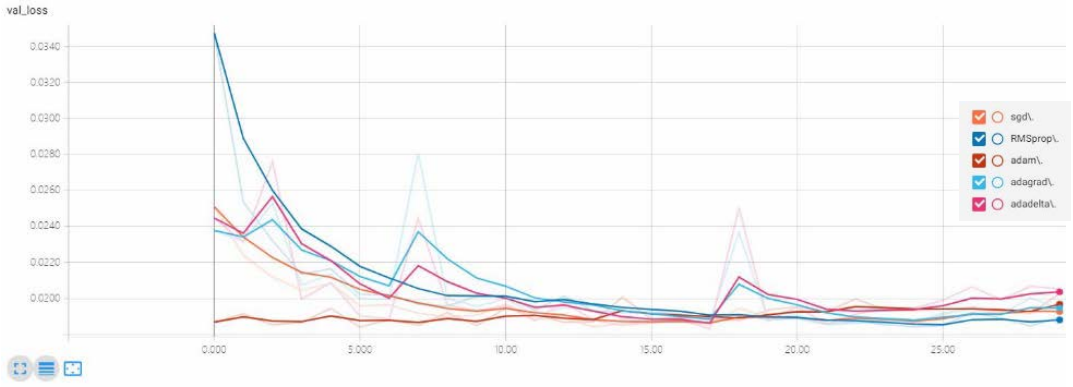


图 9 不同 optimizer 的 val_loss

图 8 和图 9 分别是五种优化器最好情况下在验证集上 30 个 epoch 内的准确率和 logloss 变化曲线。可以发现不论是准确率还是 logloss，Adam 起始时表现都是最好的，也一直比较稳定，而 RMSprop 最开始表现都是最差的，但是有着较明显和稳定的学习曲线。Adadelta 和 Adagrad 的表现非常类似，甚至不论在准确率还是 logloss 上，甚至都有着 2~3 次同步的跳动。相对而言，SGD 模型虽然刚开始表现平平，但是学习过程也比较稳定。

### 4.1.2 进一步调参和融合

实验的第二部分是不利用网上训练好的权重，直接利用 Inception_resnet_v2 模型进行迁移学习，这样可以更进一步调整网络结构。具体结果见表 7，此处运行使用的机器是亚马逊的云服务器，具体为 p2.xlarge，也因此由于 gpu 计算浮点数等原因，每次的实验结果并不能重现，val_loss 也可能出现较大的变化。

| network structure | free layers | optimizer | batch_size | val_losses | best_val_loss | val_acc | epoch_best_val | test_score |
|---|---|---|---|---|---|---|---|---|
| 256('r')+0.5+1('s') | - | sgd(0.1, 0.5) | 256 | 0.0774, 0.0583, 0.0630, 0.0470, 0.0969, 0.0769 | 0.0470 | 0.9886 | 4 | - |
| 256('r')+0.5+1('s') | - | sgd(0.1, 0.9) | 256 | 0.0708, 0.1046, 0.1062 | 0.0708 | 0.9818 | 1 | - |
| 256('r')+0.7+1('s') | - | sgd(0.1, 0.7) | 256 | 0.0551, 0.0324, 0.0712, 0.0396 | 0.0324 | 0.9890 | 2 | - |
| 256('r')+0.8+1('s') | - | sgd(0.1, 0.9) | 256 | 0.0831, 0.0584, 0.0700, 0.0678 | 0.0584 | 0.9852 | 2 | - |
| 256('r')+0.7+1('s') | - | sgd(0.01, 0.9) | 256 | 0.0582, 0.0583, 0.0574 | 0.0582 | 0.9850 | 1 | - |
| 128('r')+0.7+1('s') | 763: | sgd(0.1, 0.9) | 256 | 0.0484, 0.0301, 0.0667, --------- | 0.0301 | 0.9938 | 2 | - |
| 256('r')+0.7+1('s') | - | adam | 256 | 0.0368, 0.0846, 0.0952 | 0.0368 | 0.9898 | 1 | - |
| 256('r')+0.5+1('s') | 763: | adam | 256 | 0.0376, 0.0484, 0.0586 | 0.0376 | 0.9940 | 1 | - |
| 256('r')+0.9+1('s') | 763: | adam | 256 | 0.0794, 0.0394, 0.0497, 0.0695 | 0.0394 | 0.9940 | 2 | - |
| 0.5+128('r')+0.5+1('s') | 763: | adam | 256 | 0.0706, 0.1568, 0.0603, 0.0659, 0.0636 | 0.0603 | 0.9912 | 3 | - |
| 256('r')+0.7+1('s') | 763: | adadelta | 256 | 0.0300, 0.0446, 0.0296, 0.0540, 0.0634 | 0.0296 | 0.9948 | 3 | 0.04662(0.12213) |
| 256('r')+0.7+1('s') | 763: | adagrad | 256 | 0.0647, 0.0394, 0.0554, 0.2774 | 0.0394 | 0.9940 | 2 | - |

表 7 finetune Inception_resnet_v2 结果

列名基本同上，其他的，free layers 表示放开的起始层（这里第 763 层是 keras 上 Inception_resnet_v2 的最后一个 block 起始层），optimizer 中 sgd 只调节了 learning rate 和 momentun 两个参数，为便于比较 batch_size 都设定为 256，这里在回调函数中设置了 earlystopping（监视 val_loss，patience 取 2），val_losses 为所有 epoch 依次的 val_loss 值。

从训练结果表格中可以观察到的，第一个是这些模型的 val_loss 整体都明显低于在第一步中直接利用预训练特征向量的结果，第二点是都较早地出现过拟合，全部都在 4 个 epoch 之内，这可能是因为模型的结构太复杂，很容易学习到数据的特征规律，在 sgd 的尝试中，即是将 dropout 增加到 0.9，仍然没有效果的提升。最后取了表现最好的模型，提交 kaggle，score 是 0.0462，排到 kaggle 第 36 名。

不过由于在这一块实验的参数种类和数量不是很多（aws 的 p3 申请一直有问题，p2 太慢，每一个 epoch 十几分钟的样子），而且由于计算的不稳定性，这次结果很可能不是最好的。

由于对 Inception_resnet_v2 模型的结果感到有些出乎意料，因为这个模型结合了 inception 和 renset，应该说比 InceptionV3 会好一些，但是实际效果却相差无几，并且这一部分 finetune 由于上述原因，也没有进行充分地进行，于是又自己训练出 Inception_resnet_v2 的特征向量，构建分类网络来调参实验。实验结果如表 8 所示，此处运行使用的机器是 GTX1060 6G 显卡的笔记本，tensorflow 使用 gpu 版本。

| model | dropout+dense | batch_size | epochs_total | epoch_vl_0.0180 | epoch_vl_best | best_val_loss | val_acc | loss | acc | total_time | test_score (no clip) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| InceptionResNetV2 | 0.5+1('s') | 256 | 30(60) | - | 40 | 0.0181 | 0.9952 | 0.0170 | 0.9956 | 19.82s | |
| InceptionResNetV2 | 64('r')+0.5+1('s') | 128 | 30 | 6 | 12 | 0.0165 | 0.9958 | 0.0102 | 0.9969 | 27.80s | |
| InceptionResNetV2 | 64('r')+0.5+1('s') | 256 | 30(60) | 9 | 29 | 0.0154 | 0.9950 | 0.0063 | 0.9982 | 19.63s | 0.03745(0.05507) |
| InceptionResNetV2 | 64('r')+0.5+1('s') | 512 | 30 | 11 | 19 | 0.0156 | 0.9952 | 0.0100 | 0.9972 | 14.06s | |
| InceptionResNetV2 | 128('r')+0.5+1('s') | 128 | 30 | 4 | 8 | 0.0159 | 0.9958 | 0.0116 | 0.9969 | 27.72s | |
| InceptionResNetV2 | 128('r')+0.5+1('s') | 256 | 30 | 9 | 20 | 0.0155 | 0.9952 | 0.0074 | 0.9981 | 19.68s | |
| InceptionResNetV2 | 128('r')+0.5+1('s') | 512 | 30 | 12 | 24 | 0.0156 | 0.9952 | 0.0077 | 0.9978 | 13.58s | |
| InceptionResNetV2 | 256('r')+0.5+1('s') | 64 | 30 | 7 | 12 | 0.0166 | 0.9952 | 0.0100 | 0.9970 | 52.38s | |
| InceptionResNetV2 | 256('r')+0.5+1('s') | 128 | 30 | 7 | 12 | 0.0160 | 0.9956 | 0.0084 | 0.9974 | 26.80s | |
| InceptionResNetV2 | 256('r')+0.5+1('s') | 256 | 30 | 4 | 20 | 0.0160 | 0.9956 | 0.0064 | 0.9980 | 19.21s | |
| InceptionResNetV2 | 256('r')+0.5+1('s') | 512 | 30 | 11 | 25 | 0.0159 | 0.9952 | 0.0063 | 0.9981 | 14.00s | |
| InceptionResNetV2 | 256('r')+0.5+1('s') | 1024 | 30(60) | 15 | 40 | 0.0161 | 0.9956 | 0.0061 | 0.9986 | 12.01s | |

表 8 利用特征向量对 Inception_resnet_v2 调参

表格列名的意义基本同表 2-表 5。这一次的结果却令人惊讶，因为最好的情况下的得分 0.03745 在 kaggle 排行榜上已经可以到第 7 名！看来之前的模型确实过于复杂，finetune 起

来调到合适的参数比较困难，而且很容易过拟合，但是直接使用特征向量作为输入来分类，相当于网络隐藏层只有 1-2 层，比较简单，调节起来也比较容易，当然特征向量已经训练集的特征学习的比较好使一个大前提。当然这也只是个人猜测。另外这一步也对 clip 上下界进行了调整测试，证明了(0.005, 0.995)是一个比较好的组合，在此没有列出。

| model1(clip=no) | model2(clip=no) | model3(clip=no) | model4(clip=no) | model5(clip=no) | test_score |
|---|---|---|---|---|---|
| ResNet50 | InceptionV3 | Xception | - | - | 0.04373(0.05532) |
| ResNet50(yes) | InceptionV3(yes) | Xception(yes) | - | - | 0.04386 |
| Xception_sgd | Xception_adagrad | Xception_RMSprop | Xception_adadelta | Xception_adam | 0.04143(0.05608) |
| Xception_sgd(yes) | Xception_adagrad(yes) | Xception_RMSprop(yes) | Xception_adadelta(yes) | Xception_adam(yes) | 0.04140 |
| Xception_sgd | Xception_adagrad | Xception_RMSprop | - | - | 0.04134(0.05532) |
| Xception_sgd | Ince_ResN_v2_finetune | InceptionV3 | - | - | 0.03893(0.05620) |
| Xception_sgd | Ince_ResN_v2_vector | InceptionV3 | - | - | 0.03801(0.05185) |
| Xception_sgd | Ince_ResN_v2_vector | - | - | - | 0.03682(0.04960) |
| Xception_sgd(yes) | Ince_ResN_v2_vector(yes) | - | - | - | 0.03682 |
| Xception_sgd(yes) | Ince_ResN_v2_vector(yes) | Xception_adagrade(yes) | - | - | 0.03751 |
| Xception_RMSprop | Ince_ResN_v2_vector | - | - | - | 0.03685(0.04993) |
| Xception_RMSprop | Ince_ResN_v2_vector | Xception_sgd | - | - | 0.03753(0.04997) |
| Xception_RMSprop | Ince_ResN_v2_vector *2 | Xception_sgd | - | - | 0.03687(0.04962) |
| Ince_ResN_v2_finetune | Ince_ResN_v2_vector | - | - | - | 0.03772(0.06280) |

表 9 模型融合结果

最后，就是对上述所有的模型进行一个融合。若前面 3.3 节所述，由于算力和时间的原因，此处仅将各模型的结果进行了取平均值融合，具体结果参考表 9。

表格中默认取的没有 clip 处理之前的原数据进行的融合，但是前四行两组对比可以看出融合之前数据是否 clip 差别不是很大。另外它们单独模型的得分中，只有 Ince_ResN_v2_vec 有 0.03745，ResNet50 为 0.07089，InceptionV3 为 0.04755，Ince_ResN_v2_finetune 为 0.04662，其余均为 0.0411~0.0423 之间。

从中可以发现，第一个是 Ince_ResN_v2 模型的效果确实是最好的，和其他模型组合的时候都有着小于 0.039 的成绩，第二个是最好成绩出现在 Ince_ResN_v2 和 Xception_sgd 组合时候，之前也提到过，Xception 中 sgd 确实是较好的优化器，第三个是 clip 处理对所有的情况均有着成绩的拉升，说明这种操作确实在这里很适合。

### 4.1.3 异常值问题

由于训练集中的数据难以保证不存在一些噪音，而这些噪音在训练的过程中会诱导模型学习时产生方向上的偏差，因此如果能将这些异常值去除，再来训练模型，那么模型应该会有更好的性能。这一部分本来应该在 4.1.1 小节之前，但是由于在 4.1.1 中用的是网上训练好的特征向量，已经跳过了预处理这一环节，另一方面是由于下述原因。

实际在实验时，刚开始的预想是直接用 4.1.1 节中利用网上特征向量方法中表现最好的模型，将其来预测训练集数据，和实际 label 进行对比（以预测值概率值和真实 label 差值不小于 0.5 为界），准备以此作为基准来对异常值进行筛选。事实证明，这一想法是错误的，仔细一想，也是情理之中，因为模型已经将异常值的特征全部学习记住了，当成是真实的特征，这样并不能筛选出，我们应该用没有在该数据集上训练过的模型来对异常值进行检测。

而实际上，这一步只是检测了模型对于训练集预测错误的图片，换句话说，应该也是训练集本身中比较难以学习的图片。预测错误的图片一共有 49 张，部分图片如图 10。


图 10 预测错误的部分图片

正确筛选异常值的一种方法是借鉴 keras 官网中用 VGG 迁移学习进行图片分类的一个例子，即直接将完整的 VGG 网络迁移过来对训练集图片的类别进行预测，由于 ImageNet 数据集本身包含 1000 中类别，其中有 118 中狗类别和 7 中猫类别，这里取的阈值为 top30，即预测的 top30 个类别中都不含有正确物种，则认定该图片为异常值。并且为了判断的准确性，选用了 Inception_resnet_v2 和 Xception 两个模型进行了实验，最终取两个结果的交集，得到一共 68 张图片，其中猫类异常值 47 张，狗类异常值 21 张。部分图片如图 11 和图 12 所示。


图 11 猫类中部分异常值


图 12 狗类中部分异常值

可以看到，这些筛选出来的异常值大部分都是毫无相关的图片，当然也有一部分中确实含有猫或狗的图片，但是其特征部分在画面中占的尺寸比例太小，反而会带入更多的噪音，因而直接将其作为异常值处理也是合适的。

在得到异常值后，将这些图片从训练集中去除，再利用 Inception_resnet_v2 模型训练出特征向量，构建分类网络进行调参训练，最后得到的结果如表 10，此处运行使用的机器是 GTX1060 6G 显卡的笔记本，tensorflow 使用 gpu 版本。

| model(remove out) | dropout+dense | batch_size | epochs_total | epoch_vl_0.0180 | epoch_vl_best | best_val_loss | val_acc | loss | acc | total_time | score (no clip) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| InceptionResNetV2 | 0.5+1('s') | 64 | 30 | 3 | 7 | 0.0160 | 0.9966 | 0.0180 | 0.9959 | 46.16s | |
| InceptionResNetV2 | 0.5+1('s') | 128 | 30 | 4 | 8 | 0.0157 | 0.9966 | 0.0166 | 0.9955 | 27.47s | |
| InceptionResNetV2 | 0.5+1('s') | 256 | 30 | 6 | 12 | 0.0157 | 0.9968 | 0.0163 | 0.9958 | 18.21s | |
| InceptionResNetV2 | 0.5+1('s') | 512 | 30 | 9 | 18 | 0.0159 | 0.9964 | 0.0163 | 0.9960 | 14.93s | |
| InceptionResNetV2 | 64('r')+0.5+1('s') | 128 | 30 | 3 | 4 | 0.0164 | 0.9964 | 0.0147 | 0.9965 | 26.74s | |
| InceptionResNetV2 | 64('r')+0.5+1('s') | 256 | 30 | 2 | 4 | 0.0155 | 0.9966 | 0.0164 | 0.9961 | 19.43s | 0.03871(0.04816) |
| InceptionResNetV2 | 64('r')+0.5+1('s') | 512 | 30 | 3 | 8 | 0.0155 | 0.9962 | 0.0145 | 0.9963 | 13.74s | |
| InceptionResNetV2 | 64('r')+0.5+1('s') | 1024 | 30 | 5 | 18 | 0.0158 | 0.9962 | 0.0128 | 0.9971 | 10.93s | |
| InceptionResNetV2 | 128('r')+0.5+1('s') | 64 | 30 | 1 | 1 | 0.0157 | 0.9964 | 0.0402 | 0.9874 | 48.20s | |
| InceptionResNetV2 | 128('r')+0.5+1('s') | 128 | 30 | 3 | 4 | 0.0155 | 0.9964 | 0.0135 | 0.9965 | 28.68s | |
| InceptionResNetV2 | 128('r')+0.5+1('s') | 256 | 30 | 1 | 4 | 0.0160 | 0.9962 | 0.0151 | 0.9961 | 19.52s | |
| InceptionResNetV2 | 128('r')+0.5+1('s') | 512 | 30 | 3 | 4 | 0.0157 | 0.9962 | 0.0169 | 0.9956 | 13.78s | |
| InceptionResNetV2 | 128('r')+0.5+1('s') | 1024 | 30 | 4 | 12 | 0.0156 | 0.9964 | 0.0123 | 0.9971 | 11.65s | |
| InceptionResNetV2 | 256('r')+0.5+1('s') | 64 | 30 | 1 | 1 | 0.0172 | 0.9960 | 0.0369 | 0.9890 | 54.01s | |
| InceptionResNetV2 | 256('r')+0.5+1('s') | 128 | 30 | 1 | 3 | 0.0159 | 0.9962 | 0.0152 | 0.9960 | 29.72s | |
| InceptionResNetV2 | 256('r')+0.5+1('s') | 256 | 30 | 3 | 4 | 0.0158 | 0.9958 | 0.0137 | 0.9962 | 18.95s | |
| InceptionResNetV2 | 256('r')+0.5+1('s') | 512 | 30 | 3 | 6 | 0.0159 | 0.9964 | 0.0145 | 0.9963 | 14.55s | |
| InceptionResNetV2 | 256('r')+0.5+1('s') | 1024 | 30 | 4 | 8 | 0.0161 | 0.9964 | 0.0148 | 0.9963 | 11.91s | |

表 10 去除异常值后 Inception_resnet_v2 模型训练调参结果

可能是由于 68 张图片在整个训练集 20000 张图片中占得比例太小，去除异常值之后，模型的效果并没有得到提升，和表 8 基本处在同一水平，因此，在本项目中，也就没有用去除异常值的训练集对其他模型进行训练，也没有进行融合模型的实验。

## 4.2 合理性分析


图 13 本项目最好成绩

图 13 是本项目取得的最好成绩，这相比于之前 2.4 节中基准模型的成绩而言，应该说是一个不错的成绩。如果从 accuracy 的角度，较好的模型基本在训练集上都达到了 0.996 以上，意思是说 1000 张图片中仅有三四张（0.03% ~ 0.04%）识别错误，也基本等于训练集本身 25000 张有 68 张异常值的比例（0.272%），而实际上，4.1.3 节中已经进行了实验，25000 张图片只有 49 张判断失误（0.196%），所以准确率可以说是非常高的。如果是真人肉眼来识别，考虑到有些图片拍摄的并不好，以及人类对动物所有细分类别的不一定完全认知，可以说基本达到了人类水平。还有一点就是，本项目并不是针对准确率进行的优化。

模型的其他方面，比如鲁棒性应该也是很好的，实验中 4.1.3 节的最后相当于已经进行了测试，少量异常值的存在与否，对模型的性能基本没有影响。

# 5、项目结论

## 5.1 结果可视化

如前所述，本项目取得最好的 logloss 成绩为 0.03682，kaggle 上能排到第 7 名。

模型的结构上，由于采用迁移学习方案，卷积网络部分均直接迁移过来，没有改变，分类网络部分的结构基本如图 7 所示，只是在调参过程中有的网络会少一个 Dense 层，这里不再展示。

模型对测试集的部分图片预测效果如下图 14 所示，这里随机取了 20 张图片，可以看到，模型对这些图片的预测全部是正确的，而且都给出了最少 99.95%的预测概率，应该说预测的效果非常满意。

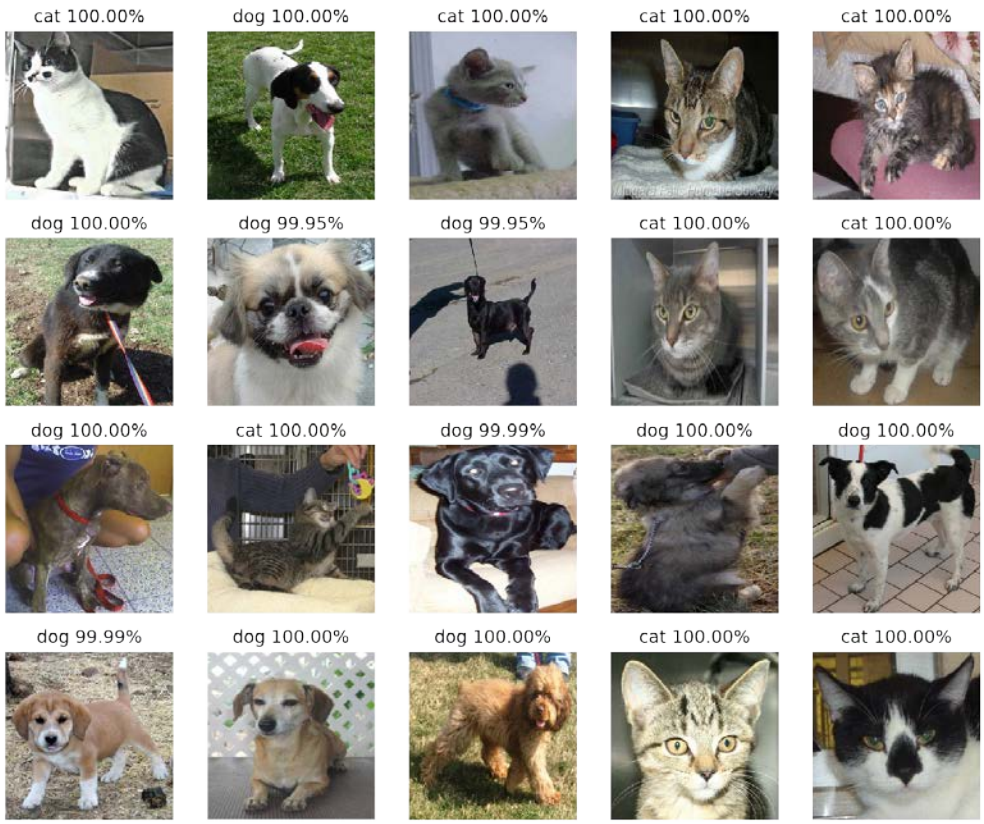


图 14 模型对部分测试集图片的预测

另外，由于好奇模型构建好之后，它在预测图片时究竟是如何作出判断的，在本项目的最后一个部分，利用 CAM (class activation maps，类激活图) [18，19]来进行可视化，将不同权重特征图进行叠加生成热力图，重要的特征图会有较大的权重，也就是网络进行分类的主要“依据”，最后将热力图叠加到原图上。在测试集上对部分图片的实验如图 15 所示。

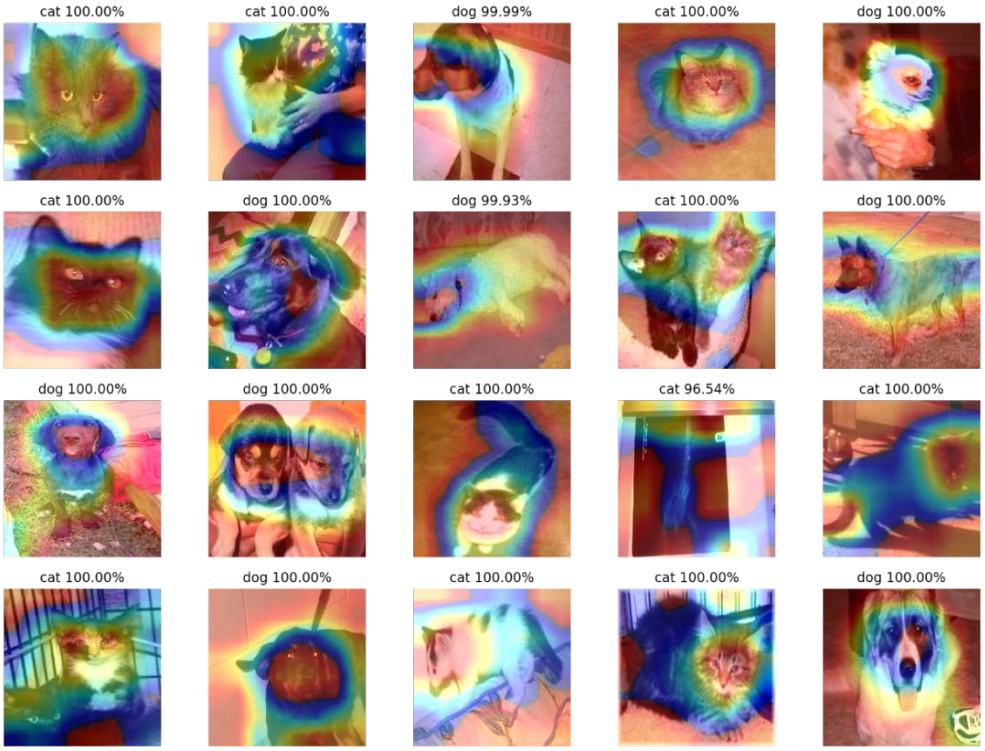


图 15 测试集上 CAM 可视化

可以看到，网络对图片的分类依据和人类还是很类似的，主要是根据面部来判别的，对于身子等特征不够明显的部位并不关注，对其他的背景如草地、地板等则基本完全忽略。比较有意思的是，对于图中有多个动物时，模型也能够都进行关注。

但是这样的可视化也只能看到最终的结果，于是参考[20]的方法，在训练时每个 batch 之后都将权重存储下来，然后利用每个权重都进行可视化成图片，最后将这些图片生成视频，则可以动态地看到整个过程中网络结构作出判断的过程。

实验时刚开始用的 Xception 模型，由于效果在整个过程中都比较好，没有明显的学习过程，后来又使用 VGG16 来训练生成动态 CAM 图，一共 1250（20000/16）个 batch（只训练了一个 epoch），即有 1250 张图片，视频帧率为 25fps，即 50s 长，acc 从大概 0.47 学习到 0.97，随机从训练集选取图片来可视化，图 16 即是模型对 dog.11631.jpg 预测的过程。

从 0s ~ 50s，可以看到，模型初期的判断变化地很快，后面 40s 范围变化很小，这也就是模型在训练时刚开始时候 logloss 和 acc 变化很快的具体体现。有意思的是模型对图中左侧的洋娃娃很快作出判断不是目标（或许是没有鼻子之类原因），但对中间娃娃的嘴部却一直没有舍弃，这可以看做是模型性能不够好的表现之一。

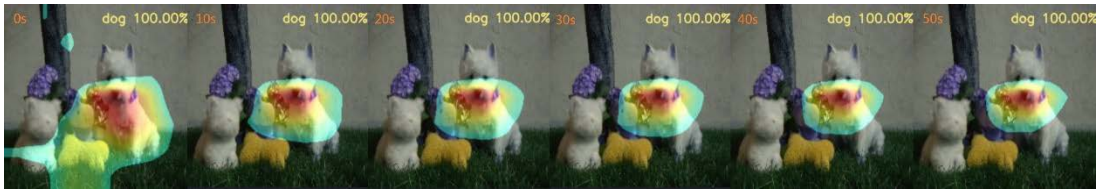


图 16 模型对 dog.11631.jpg 预测过程 CAM 图

## 5.2 对项目的思考

回头再看整个项目，心态上和刚开始还是有一些差别的。虽然刚开始自己大概也知道要做什么，有哪些步骤，会用到什么工具，但是真正只有自己经历了，才会有具体而难忘的印象吧。一个星期前，我就觉得马上要结束了，因为主要实验已经进行得差不多了，但是一晃又过了一周，异常值和可视化等进行地更加完善了，另外也低估了 report 撰写的时间开销。

应该说这个项目对于入门深度学习还是比较合适的，我们可以利用 keras 方便地构建自己的网络，可以利用在 ImageNet 上预训练的模型来 finetune，这本身也会促使我们去看这些模型的文章，而 VGG、ResNet、Inception 系列等基本也是今年来 DL 在计算机视觉上最为经典的模型，便于我们了解整个历史发展的来龙去脉，中间也会了解和利用有关 dropout、BN、CAM 等等技巧。

整个项目中有意思的地方还是比较多的，主要是最后的可视化部分，之前听说深度学习是一个黑匣子，但是观看它的预测过程，发现它和人类的观测过程还是比较类似的，区分猫狗主要是依据其脸部，而且对多目标、不够显著目标、假目标都有一定的鉴别能力。

项目中遇到的难点也是蛮多的，从最开始 ubuntu 安装、tf-gpu 安装等环境搭建，到具体建模时数据的处理、异常值的检测、可视化等，调参当时一段漫长的经历，但其实有一个好的模型基础比调参来的给力很多，调参并不是盲调，需要对深度学习中学习率、batch_size、网络结构等都有着比较清楚地理解，我们才能快速地找到好的参数，这也是整个项目中的宝贵经验之一吧。

## 5.3 需要作出的改进

项目做到这里，需要改进的遗憾也还是有的，首先是成绩是否还可以提高，有一些模型比如 MobileNet、DenseNet 等并没有尝试，inception_resnet_v2 模型的调参也限于时间和算力等原因也没有充分进行，融合也可以有更多的尝试，比如更多模型的组合或者不同权重的组合，以及在网络结构中融合特征而不是直接融合结果。其他比如数据增强等技术在本项目中也值得尝试。

另一方面是，限于时间原因，自己在使用过程中，对模型相关的论文也没有完全熟读，相信如果我更加深入地理解了，最终也一定会有更好的效果。接下来也需要去认真研读。

最后就是，由于模型最后的表现已经基本和人类相差无几，这是否意味着这一算法已经可以进行实用的环节？后面也可以探索使用其来做一些网页或者接入到自己的微信公众号中。

最最后，感谢一路走来 udacity 同学、mentor 的交流和指导以及审阅老师的细心批阅，都让我受益匪浅。

# Reference：


[1]. Krizhevsky, A., Sutskever, I., and Hinton, G. E. *ImageNet classification with deep convolutional neural networks* [J]. In NIPS, pp. 1106–1114, 2012.

[2]. He, Kaiming, et al. *deep residual learning for image recognition* [J]. Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[3]. 才云科技 Caicloud, 郑泽宇, 顾思宇. *TensorFlow 实战 Google 深度学习框架* [M]. 北京：电子工业出版社, 2017: 10.

[4]. https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition/data.

[5]. Simonyan, Karen, and Andrew Zisserman. *very deep convolutional networks for large-scale image recognition* [J]. arXiv preprint arXiv:1409.1556.

[6]. Zbigniew Wojna. *rethinking the inception architecture for computer vision* [J]. arXiv preprint arXiv:1512.00576.

[7]. Francois. Chollet.Xception: *deep learning with depthwise separable convolutions* [J]. arXiv preprint arXiv:1610.02375.

[8]. McCulloch W, Pitts W. *a logical calculus of the ideals immanent in nervous activity* [J]. Bulletin of Mathematical Biophysics. Vol 5, 1943.

[9]. Ian Goodfellow, Yoshua Bengio, Aaron Courville. *深度学习* [M]. 北京:人名邮电出版社, 2017: 8-12, 171, 203-211.

[10]. Md Zahangir Alom, Tarek M. taha, Christopher Yakopcic, et al. *the history began from AlexNet: A comprehensive survey on deep learning approaches* [J]. arXiv preprint arXiv:1803.01164.

[11]. Vincent Dumoulin, Francesco Visin. *a guide to convolution arthmetric for deep learning* [J]. arXiv preprint arXiv:1603.07285.

[12]. Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke. *Inception-v4, inception-resnet and the impact of residual connections on learning* [J]. arXiv preprint arXiv:1602.07261.

[13]. Gao Huang, Zhuang Liu, Laurens van der Maaten, et al. *densely connected convolutional networks* [J]. arXiv preprint arXiv:1608.06993.

[14]. Andrew G. Howard, Menglong Zhu, Bo Chen, et al. *MobileNets: efficient convolutional neural networks for mobile vision applications* [J]. arXiv preprint arXiv:1704.04681.

[15]. Barret Zoph, Vijay Vasudevan, Jonathon Shlens, et al. *learning transferable architecture for scalable image recognition* [J]. arXiv preprint arXiv:1707.07012.

[16]: https://github.com/ypwhs/dogs_vs_cats/releases/tag/gap.

[17]: https://www.kaggle.com/c/state-farm-distracted-driver-detection#evaluation.

[18]. Bolei Zhou, Aditya Khosla, Agata Lapedriza, et al. *learning deep features for discriminative localization* [J]. arXiv preprint arXiv:1512.04150.

[19]. Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, et al. *Grad-CAM: visual explanations from deep networks via gradient-based localization* [J]. arXiv preprint arXiv:1610.02391.

[20]. https://github.com/mtyylx/MLND/blob/master/P6_Dogs_VS_Cats/Class%20Activation%20Map%20Visualizations.ipynb