**猫狗分类项目报告**

## **定义**

### **项目概述**

图像识别或图像分类是属于视觉科学和电脑科学领域的一个问题。这类问题的目的是需要将图片分类至一个或多个类别当中。这个问题可以通过人工手动或借助算法来完成，这些图片可能包含了不同的物体，图片的大小可能不一致，颜色上也千差万别。图片仍可以根据它们所包含的边缘（edges），角（corners），区域（blobs）和脊（ridge）来做分类[1]。

本项目将会采用广泛使用的机器学习算来进行图片识别以及图片分类。机器学习包含许多的分类算法比如，决策树，KNN，朴素贝叶斯和神经网络等等，每个模型都有它们各自的优缺点。这里我们将采用卷积神经网络（简称CNN或ConvNet）来做图片分类。卷积神经网络是一个兼具深度和高识别率的神经网络，已经成功应用于图像识别领域。

本项目将基于Kaggle的一个竞赛项目Dogs vs Cats[2]来进行图像识别。这个项目包含了25000张的训练集图片以及12500张的测试集图片。这些图片将被划分到猫或狗的一个分类中。

### **问题陈述**

猫狗识别的项目属于监督学习中的分类问题，本项目中一共有2个类别，每一张图片只属于其中一个类别，项目的目标是利用CNN将图片划分到它们正确的类别中。因为从零开始训练一个卷积网络需要花费大量的时间和资源，这里会使用已经身经百战的CNN模型，并利用这些模型导出数据集的深度特征。然后基于这些深度特征来训练自定义的模型，并在最后验证自定义的模型。

### **评价指标**

这里Kaggle官方给出的评估标准是LogLoss而且同时我们的测试集是没有标签的。所以我们也使用LogLoss作为我们的评估标准，LogLoss的公式如下，其中n代表的是sample的数量，i代表的是当前sample的索引，代表的是模型预测的结果，代表的是真实的标签。

## **分析**

### **数据的探索**

数据集可以直接从Kaggle[3]上下载。Kaggle一共提供了3个文件，train.zip是训练集，test.zip是测试集以及一个csv格式的submission文件。训练集包含了2个类别的图片，猫和狗的图片各12500张。测试集包含了12500图片。训练集下数据的通过文件名提供了标签，文件名的格式是{类别}.{序号}.jpg，比如猫的图片名称为cat.1.jpg，狗的图片名称为dog.1.jpg。因此我们可以根据文件名，将其进行分类。而测试集的数据中并没有提供标签，文件名的格式是{序号}.jpg，比如1.jpg或2.jpg。同时数据集中的图片大小都是不一致的，需要在后续的步骤中处理这个问题。下面展示的是的训练集和测试集的部分数据。

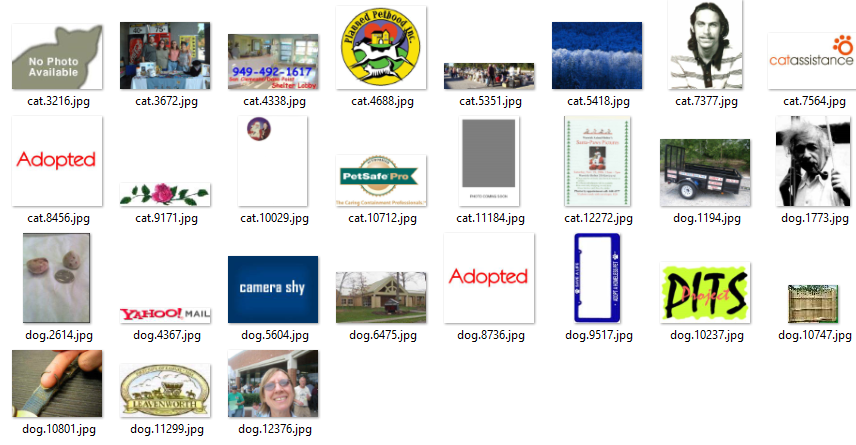
**训练集中猫的图片** [](https://github.com/Qucy/UdacityMachineLearning/blob/master/dog_vs_cat/images/train_set_cat.PNG)

**训练集中狗的图片** [](https://github.com/Qucy/UdacityMachineLearning/blob/master/dog_vs_cat/images/train_set_dog.PNG)

**测试集中的图片** [](https://github.com/Qucy/UdacityMachineLearning/blob/master/dog_vs_cat/images/test_set.PNG)

简单的数据探索完成后，我利用Xception对所有的训练集做了预测。根据预测返回的前20的标签获取了Xception模型认为既不是猫也不是狗的图片。因为异常图片并不是非常多，所以我把异常图片拷贝到另外一个文件夹，用肉眼过了一遍。发现有很多猫的图片都误判了，但是狗的图片大部分都对了，只有少部分误判了。因此我再次用了Inception模型和ResNet50模型，对猫的异常图片进行了一个预测，移除了Inception模型和ResNet50模型认为是猫的图片。剩下的异常图片我通过人工的方式做了判断，把我认为正确的图片路径全部放在一个数组中并从异常图片中过滤掉，最终得到了以下异常图片。最后我通过python的API将这些异常文件全部从训练集中移除。

**异常图片**

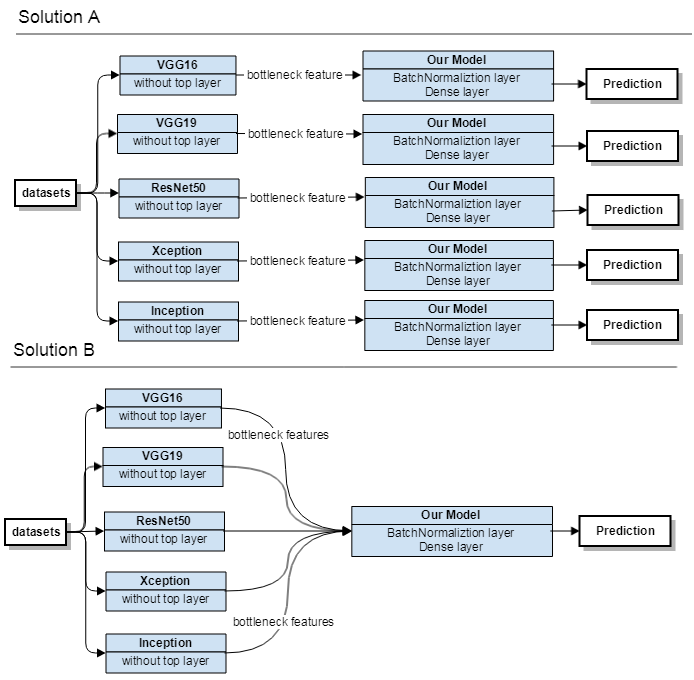


### **算法和技术**

项目中会使用VGG[4]，ResNet[5]，InceptionV3[6]，Xception[7]等模型，它们简单的介绍如下：

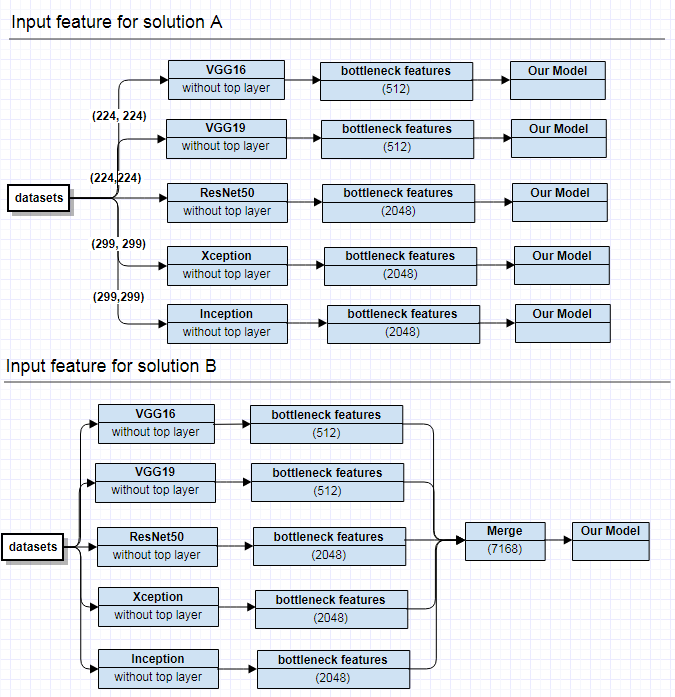
* VGG: VGG网络架构比较简单，遵循基本卷积网络的原型布局，一系列卷积层、最大池化层和激活层，最后还有一些全连接的分类层。而VGG16和VGG19中的16和19分别代表了权重的层数。VGGNet主要有2个缺点，一个训练起来很慢，非常耗时。另外一个是模型weights很多，导致模型很大，训练时也会占用更多的磁盘和带宽。VGG16模型大约为533MB，VGG19模型大约为574MB。
* ResNet：也叫残差网络，诞生于一个美丽而简单的观察：为什么非常深度的网络在增加更多层时会表现得更差？作者将这些问题归结成了一个单一的假设：直接映射是难以学习的。而且他们提出了一种修正方法：不再学习从 x 到 H(x) 的基本映射关系，而是学习这两者之间的差异，也就是「残差（residual）」, 假设残差为 F(x)=H(x)-x，那么现在我们的网络不会直接学习 H(x) 了，而是学习 F(x)+x。ResNet的模型虽然很深但因为使用了很多平均池化层仍然比VGG小许多，大概只有100MB左右。
* Inception：第一版叫做GoogleNet，V3，V4是不同的版本名称。这个网络的特点在同一层同时采用3x3，1x1，5x5的Filter及一个3x3的max pooling来提取多个特征，再将这些不同特征组装。如果ResNet是为了更深，那么 Inception 家族就是为了更宽。Inception的模型比VGGNet和ResNet都要小，大概只有96MB。
* Xception：Xception的意思是extreme inception，而且正如其名字表达的那样，它将 Inception 的原理推向了极致。它的假设是：「跨通道的相关性和空间相关性是完全可分离的，最好不要联合映射它们。」Xception 不再只是将输入数据分割成几个压缩的数据块，而是为每个输出通道单独映射空间相关性，然后再执行 1×1 的深度方面的卷积来获取跨通道的相关性。Xception的模型最小大概只有89MB。

有了这些基础模型，我会基于这些基础模型来进行迁移学习，同时我会使用谷歌的TensorFlow[8]作为神经网络的平台，并使用Keras[9]的API来构建并训练CNN。项目中一共采用了2个方案，方案A是基于单个模型的迁移学习，即只采用一个模型导出的深度特征来进行训练；方案B是基于当前所有模型的迁移学习，即合并所有模型的深度特征来进行训练。2种方案的架构图如下：



### **探索性可视化**

我把这一节放在了算法与方法后面，这样看上去会更合理一些。根据算法和方法中的架构图，对于自定义的模型而言输入就是各个模型的深度特征。不同方案下，输入的特征也会不一样。对于方案A，自定义模型的输入就是各个模型导出的深度特征，其中VGG是512个特征，ResNet，Inception，Xception都是2048个特征。对于方案B，输入则是这些深度特征的集合，即把5个模型的深度特征全部相加。可视化结构如下。



### **基准模型**

这里项目的要求是达到Kaggle的排行榜的前10%，所以本项目的要求是在测试集上的LogLoss表现要低于0.06127。

## **方法**

### **数据预处理**

项目中一共对数据做了3次预处理，第一次因为flow\_from\_directory API的要求，对训练集中的2类图片进行了一个分类。根据图片的文件名称将猫和狗的图片分别放在cat和dog的子文件夹中。同时把测试集的数据全部移动到1个子文件夹中。第二次是在数据探索的时候移除了异常的图片。首先利用Xcepton模型对所有训练集做预测，根据排名前20的标签找出Xception模型认为既不是狗也不是猫的图片。然后再利用Inception模型和Resnet模型的预测结果以及人工判断的方法来移除误判的图片。最后把剩下的异常图片全部移除训练集。第三次是在导出深度特征导时，对输入的图片全部加了一个model.preproccess\_input的预处理操作。比如VGG16，用的是vgg16.preprocess\_input方法，而Xception用的是xception.preprocess\_input方法（实际上调用的都是同一个方法，参数不同）这样做的目的是将我们的0-255的RGB值缩小至-1到1的范围，好处是使得模型可以更快的收敛，缩短训练时间。

### **执行过程**

首先需要使用Keras的Application API[10]来获取这些模型，比如通过keras.applications.vgg16.VGG16 (include\_top=False, weights='imagenet') 来获取VGG16的模型，include\_top要设置成False，代表获取的模型不包含最后的全连接层。weights要设置为 ‘imagenet’ 代表获取的权重是pre-train过的。

其次使用Keras的图片预处理API ImageDataGenerator[11]中的方法flow\_from\_directory加载数据集。通过参数target size来调整图片的大小，因为不同的CNN需要的输入的图片大小会有所区别，比如VGG和ResNet要求的图片大小为（224,224）而Xception和Inception需要的图片大小为（299,299）。然后调用model的方法predict\_generator来进行预测，得到所有训练集的深度特征。为了方便后期调试，这里我使用了HDF5 for python API[12]把所有模型的深度特征保存在本地。

项目自定义一个模型来基于深度特征进行迁移学习。自定义模型一共只包含了2层，第一层为BatcNormalization层，目的是为了防止过拟合。第二层为Dense层，激活函数为sigmod，目的是为了做最后的分类。构建完模型后，还需要调用model的compile方法来编译模型。编译模型时，我传递了3个参数，第一个是optimizer代表的是优化器，项目中使用的是Adam[13],这是一款常见的优化器，特点是计算效率较高，对内存要求比较少，默认参数的超参数大多数情况下基本够用了。第二个是loss代表的是损失函数，因为是二分类我使用的loss是binary\_crossentropy。第三个是metrics代表的是衡量指标，这里可以传递的是数组，可以有1个或多个指标，我使用的是['accuracy']。

模型compile后开始训练模型，这里不需手动将训练集划分为训练数据和验证数据，Keras的Model API[14]提供一个训练模型的方法fit。而fit中提供了一个参数validation\_split可以自动帮助我们来划分训练数据和验证数据。比如validation\_split = 0.2时，代表80%的数据用于训练，而剩下20%数据用于验证。

项目中我手动调参并训练了以下场景，记录了以下数据：

* 基于VGG16的迁移学习是我最早调试的，开始时Adam优化器全部采用默认参数并且epoch设置为10。10次以内训练集和验证集的accuracy是不断上升的，loss也是不断下降的。因此我将Epoch加到了50次，虽然最大的accuracy和最小的loss都表现的比上次要好。但是通过verbose=1打印出来的数据显示，test loss和accuracy浮动很大，忽高忽低。推测learning rate稍微偏大了，于是下调了learning rate至0.0001并增加epoch次数，epoch分别尝试了200，300次以及400次。epoch增加至400次时，大概从300次开始出现了轻微的过拟合情况。因此，最终我选择了epoch=300，lr=0.0001的方案。下方图表展示了不同epoch次数下不同learning rate的模型表现。（绿色字体代表最佳方案）

| **epoch** | **optimizer** | **Learning rate** | **train max acc** | **val max acc** | **train min loss** | **val min loss** | **time(sec)** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | Adam | 0.001 | 0.9832 | 0.9694 | 0.0474 | 0.0790 | 6.25 |
| 50 | Adam | 0.001 | 0.9902 | 0.9774 | 0.0284 | 0.0606 | 32.41 |
| 200 | Adam | 0.0001 | 0.9897 | 0.9744 | 0.0310 | 0.0685 | 175.38 |
| 300 | Adam | 0.0001 | 0.9908 | 0.9740 | 0.0287 | 0.0675 | 180.71 |
| 400 | Adam | 0.0001 | 0.9912 | 0.9744 | 0.0269 | 0.0676 | 257.19 |

* 基于VGG19的迁移学习模型，因为VGG19和VGG16非常相似，所以我直接采用了VGG16的最佳方案，设置epoch=300以及lr=0.0001。但是从训练日志从看出，大概从200次开始验证集的acc开始略微下降，loss开始略微上升，出现了一点过拟合的现象。推测可能是因为VGG19比VGG16多3层权重，表达的内容可以更加复杂，所以基于VGG19的迁移学习的模型并不需要那么多的训练次数，否则会出现过拟合的情况。因此，我将epoch设置为200，重新训练。虽然最大acc和最小loss表现的没上次好，但至少没有出现过拟合的情况，这样的模型泛化能力会更好。因此，我最终选择了epoch=200, lr=0.0001的方案。具体表现如下方图表。

| **epoch** | **optimizer** | **Learning rate** | **train max acc** | **val max acc** | **train min loss** | **val min loss** | **time(sec)** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 200 | Adam | 0.0001 | 0.9902 | 0.9752 | 0.0301 | 0.0619 | 188.52 |
| 300 | Adam | 0.0001 | 0.9911 | 0.9774 | 0.0272 | 0.0602 | 188.52 |

* 基于ResNet50的迁移学习模型，ResNet50顾名思义一共有50层，比VGG的层数更多。所以这里我直接尝试了50，100次的epoch以及lr=0.0001。当epoch=100的时候，大概在50次左右的时候开始出现过拟合的情况。因此，我最终选择了epoch=50, lr=0.0001的方案。具体表现如下方图表。

| **epoch** | **optimizer** | **Learning rate** | **train max acc** | **val max acc** | **train min loss** | **val min loss** | **time(sec)** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 50 | Adam | 0.0001 | 0.9940 | 0.9826 | 0.0188 | 0.0484 | 45.58 |
| 100 | Adam | 0.0001 | 0.9980 | 0.9824 | 0.0096 | 0.0459 | 69.26 |

* 基于InceptionV3的迁移学习模型，越后期的模型越成熟，因此learning rate需要设置的更低，这里我直接尝试了lr=0.00005，epoch=15，30，50的场景。从训练日志中可以看出，val loss在15代之后基本不在下降了，不停的上下浮动。因此，我最终选择了epoch=15, lr=0.00005的方案。具体表现如下方图表。

| **epoch** | **optimizer** | **Learning rate** | **train max acc** | **val max acc** | **train min loss** | **val min loss** | **time(sec)** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | Adam | 0.00005 | 0.9934 | 0.9938 | 0.0260 | 0.0285 | 14.50 |
| 30 | Adam | 0.00005 | 0.9946 | 0.9928 | 0.0190 | 0.0256 | 29.68 |
| 50 | Adam | 0.00005 | 0.9961 | 0.9912 | 0.0131 | 0.0285 | 45.40 |

* 基于Xception的迁移学习模型，根据InceptionV3的经验，这里直接尝试了lr=0.00005，epoch=15，30的场景。和Inception类似，Xception也是在15代左右开始val loss基本不在下降，开始不停的上下浮动。因此，我最终选择了epoch=15, lr=0.00005的方案。具体表现如下方图表。

| **epoch** | **optimizer** | **Learning rate** | **train max acc** | **val max acc** | **train min loss** | **val min loss** | **time(sec)** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | Adam | 0.00005 | 0.9939 | 0.9938 | 0.0228 | 0.0251 | 16.02 |
| 30 | Adam | 0.00005 | 0.9948 | 0.9924 | 0.0179 | 0.0263 | 26.92 |

* 基于所有模型的迁移学习，因为集成了所有模型的权重，所以我首先尝试了Inception和Xception的最优方案。即lr=0.00005，epoch=15的场景。结果好的出奇，val的loss第一次降到了0.02以下。因此，我直接选择了epoch=15, lr=0.00005的方案。具体表现如下方图表。

| **epoch** | **optimizer** | **Learning rate** | **train max acc** | **val max acc** | **train min loss** | **val min loss** | **time(sec)** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | Adam | 0.00005 | 0.9961 | 0.9934 | 0.0130 | 0.0196 | 16.02 |

综上，基于所有模型的迁移学习表现是最好的。在训练集上取得了最高的accuracy，0.9961，并在验证集上取得了第二高的accuracy，0.9934。并且在训练集上取得了最低的loss 0.0196，在验证集上也取得了最低的loss 0.0196。同时基于InceptionV3和Xception迁移学习的模型表现的也比较好。

模型训练完成后，可以使用model的predict的函数来进行预测。我首先从本地文件中读取出测试集的深度特征。然后调用自定义模型的predict方法来进行预测，预测完之后函数会返回一个长度为12500的数组，数组中包含了每个测试的预测结果。预测结果为0-1的数字，越接近1说明越可能是狗，越接近0说明越不可能是狗。接着我使用了pandas的dataframe的API[14]read\_csv方法来读取submission文件，然后使用dataframe的at方法将自定义模型的预测值写进去（df.at[index, 'label'] = y）。最后利用dataframe的to\_csv方法生成我自己的提交文件。

在生成我自己的submission文件，利用了2个技巧，第一个是对predict函数生成的数组做了最大和最小值的限制，比如y = y.clip(min=0.005, max=0.995)，具体原因在完善中会介绍。第二个是利用Keras API ImageDataGenerator的flow\_from\_directory方法返回的generator对象，generator对象中有一个属性filenames会返回测试中所有的文件名称，我迭代了这些文件名称来一行行的更新需要提交的submission文件。这样做的目的是，测试集预测的顺序不是按照1.jpg，2.jpg，3.jpg来预测的。实际顺序是1.jpg，10.jpg，100.jpg，1000.jpg，1001.jpg，这个顺序和submission要求的顺序不一样，所以我们不能直接迭代预测结果来更新submission文件，而是要借助generator的帮助，提取文件名的数字部分作为索引来写入自己的预测值。最后submission文件成功生成后就可以提交到Kaggle进行评估了。

### **完善**

项目中一共完善了2个地方来提升了模型得表现及最终得评分。

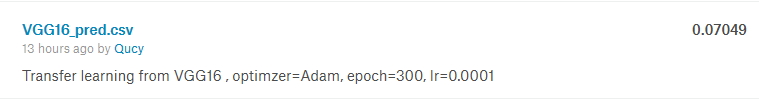
* **参数优化，通过调整epoch以及learning rate来训练出最佳的模型。**以基于VGG16进行迁移学习为例，使用的优化器是Adam，刚开始得时候直接使用了Adam的默认超参数来训练，模型只训练10次，发现模型还有提高的空间。于是增加训练次数。但是当训练次数超过10次以后，模型在验证集的loss上都会出现明显的抖动，一会高，一会低，而不是持续的降低。推测原因是，Adam优化器默认的学习速度在这个场景下还是偏高了。于是调整了模型的超参数learning rate并且适当的增加了训练次数。最终对于基于VGG16进行迁移学习的模型在epoch=300以及lr=0.0001的时候表现比较好。
* **输出完善，限制预测结果的最大最小值。**Kaggle官方使用的Logloss来进行评估，如果预测正确的话，偏差不会特别大。但是当预测结果错误的时候Logloss会出现无穷大的情况，这样会使得Logloss计算出比较大的值，从而降低排名。所以这里对预测值做了一个区间限制[0.005-0.995]。

## **结果**

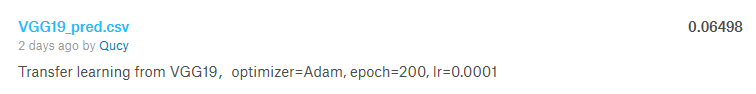
### **模型的评价与验证**

基于测试集进行预测并将结果按照Kaggle的格式要求生成一份csv文件，将文件提交至Kaggle后，得到以下评估结果。

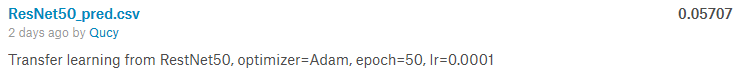
* 基于VGG16迁移学习的模型，测试集的得分为0.07049



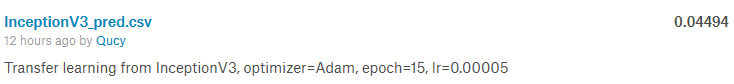
* 基于VGG19迁移学习的模型，测试集的得分为0.06489



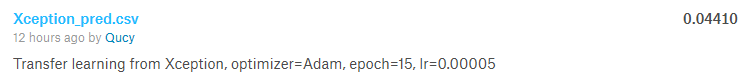
* 基于ResNet50迁移学习的模型，测试集的得分为0.05707



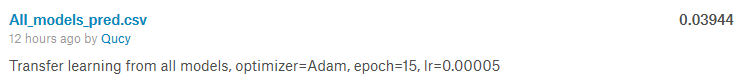
* 基于InceptionV3迁移学习的模型，测试集的得分为0.04494



* 基于Xception迁移学习的模型，测试集的得分为0.04410



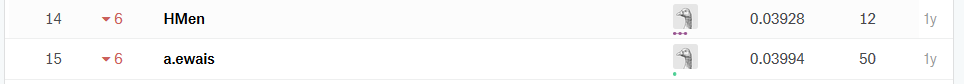
* 基于所有模型的迁移学习，测试集的得分为 0.03944



综上，所有模型的表现都还是挺不错的，其中基于ResNet50，基于InceptionV3，基于Xception以及基于所有模型迁移学习的模型的排名都进入了前10%。

### **合理性分析**

项目的基准要求是测试集的分数低于0.06127。根据上方的评估结果，其中基于ResNet50，基于InceptionV3，基于Xception以及基于所有模型迁移学习的模型都到达了要求。其中基于所有模型的迁移学习的分数最好达到了0.03944，可以排在 leaderboard的第15名。以下为排名14，排名15的分数。

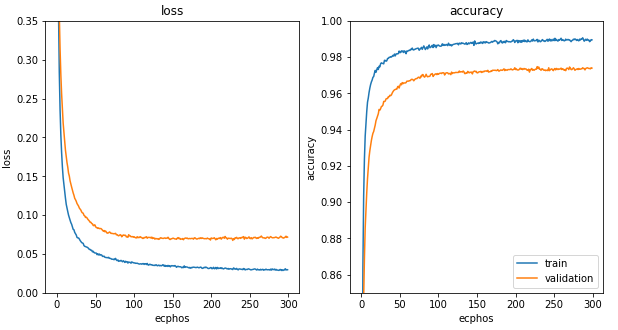


## **结论**

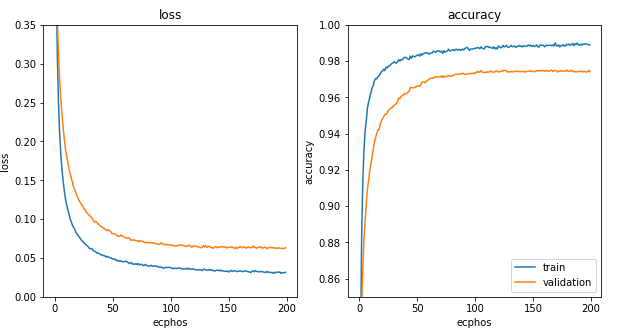
### **结果可视化**

在训练过程中，我对所有方案的最优模型的训练过程进行了可视化，为了确保我的模型没出现过拟合的问题。左边的图为loss在训练过程中的变化，右边的图为accuracy在训练过程中的变化。蓝色的线代表了训练集，橘黄色的线代表的测试集。横坐标代表epoch，纵坐标分别代表loss和accuracy。

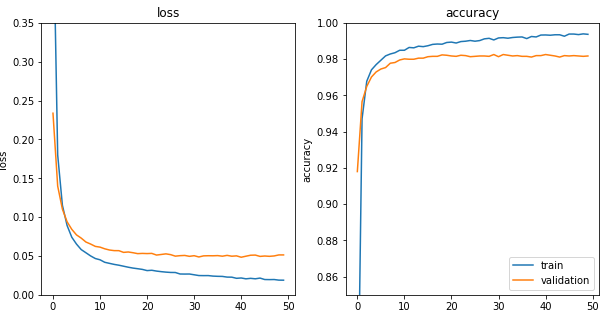
* 基于VGG16的迁移学习，learning rate = 0.0001，在训练300代后loss和accuracy趋于平稳



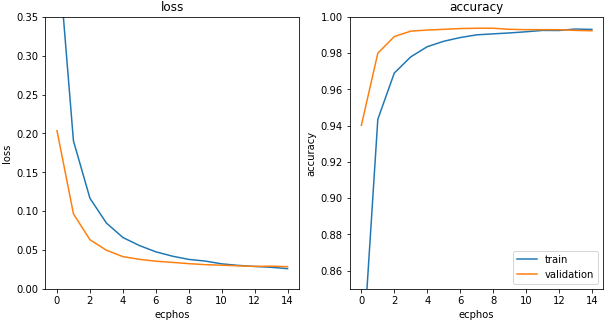
* 基于VGG19的迁移学习，learning rate = 0.0001，在训练200代后loss和accuracy趋于平稳



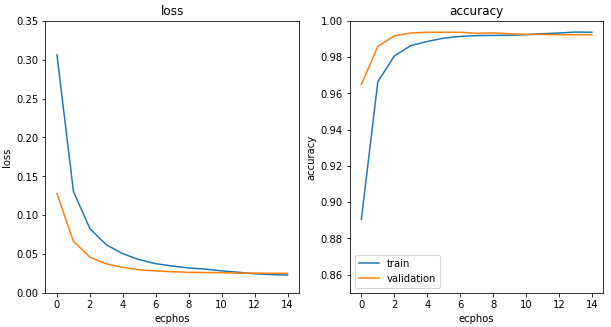
* 基于ResNet50的迁移学习，learning rate = 0.0001，在训练50代后loss和accuracy趋于平稳



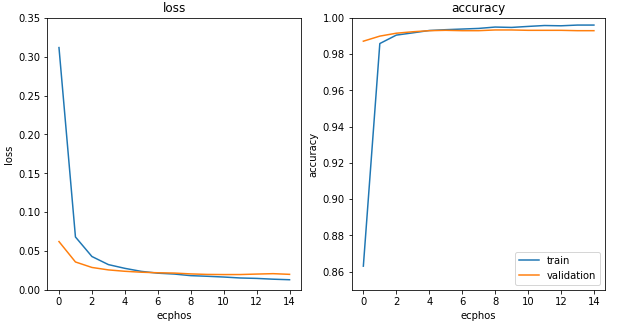
* 基于InceptionV3的迁移学习，learning rate = 0.00005，在训练15代后loss和accuracy趋于平稳



* 基于Xception的迁移学习，learning rate = 0.00005，在训练15代后loss和accuracy趋于平稳



* 基于所有模型的迁移学习，learning rate = 0.00005，在训练15代后loss和accuracy趋于平稳

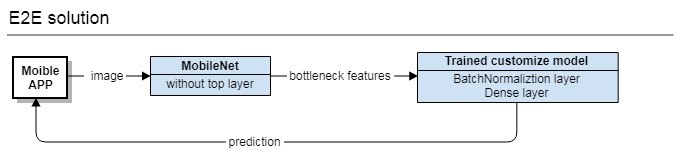


综上，可以看出基于不同模型所需要的超参数很可能是不一样的，所需要训练的次数也是不一样的。基于VGG和基于ResNet50迁移学习的模型需要的learning rate和训练次数稍微高些。而基于Inception，基于Xception以及基于所有模型的迁移学习的模型需要的learning rate更低，训练的次数更低。

### **对项目的思考**

项目主要分成了3个步骤，首先是深度特征的提取，然后是自定义模型的训练，最后是对预测进行评估。由于个人原因，这里我并没有做成一个端到端的方案。如果要做成一个端到端的解决方案的话，步骤会稍微有所不同。

* 第一步是深度特征的提取，深度特征的提取需要考虑用到哪几个基础模型。如果应用是部署在服务器上那么，以准确率和提取时间为第一优先级，可以考虑提取多个CNN的深度特征来增加准确率。如果应用是运行在手机上APP，那么首先要考虑到容量和运行时间的问题，手机上不可能给你几个G的空间去存放好几个CNN模型。需要选择一个占用空间较小运行时间较快，但准确率较高的CNN，比如Inception或Xception，甚至专门为手机定制的MobileNet[15]。
* 第二步是使用已经训练过的自定义模型来进行预测，因此需要保存已经训练好的模型并且部署在服务器或手机中。每当有请求的时候再加载模型并根据深度特征进行预测并返回 预测结果。
* 第三步，考虑到是一个端到端的方案，这里可能需要将第一步和第二步封装成一个API，可能是一个Restful的API，也可能是一个简单的post请求。参数可能是一张或多张图片的路径，也可能是图片转成二进制后的byte数组。返回值可以是true或false，也可以是预测的概率。总之，这些需要根据具体的端到端的需求来决定。
* 补充，如果这是一个web的应用或是一个手机的应用，那么还会有一个前端的用户界面。以手机应用为例，我将一个大致的端到端的方案的架构图画了一下。



通过这个项目，让我感到迁移学习的强大，主要体现在准确率高，loss低以及自定义模型在迁移学习后训练的时间非常短，利用迁移学习可以大大降低训练一个模型的成本。项目中比较有趣的地方是通过集成多个CNN来进行迁移学习可以达到非常不错的效果。我也看了一些参加Kaggle竞赛者排名靠前的采访，他们也集成非常多的CNN才能得到一个非常不错的分数，有些人甚至集成了一些其他的机器学习算法比如，随机森林，XGBOOST等。项目中对我而言比较困难（或麻烦）的地方是环境的准备，我大概花了两天的时间来准备本地环境。因为要用到GPU需要安装许多NVIDIA的包，过程中间还踩了一些坑。

### **需要做出的改进**

当然任何项目都是存在可以进步的空间的，我认为可行的改进还有以下几个方面：

* 数据增强，项目中还提供了另外一个数据集，但是我没有使用。我们可以把这部分数据也作为我们的训练数据或许可以更进一步提升我们模型的表现。
* Fine-tuning，迁移学习还有另外一个利器就是Fine-tuning，我可以继续利用该方法来继续调试，优化模型，得到更好的模型。
* 集成更多的模型，这个项目中我只集成了5个CNN，我们可以集成更多的CNN来得到更全面的信息来提升最终模型的表现。

## **引用**

[1] Wiki page, Feature detection:en.wikipedia.org/wiki/Feature\_detection\_(computer\_vision)

[2] Kaggle Project Dogs vs Cats:[www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition](http://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition)

[3] Dogs vs Cats Datasets: [www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition/data](http://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition/data)

[4] Karen Simonyan, Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556, 2014

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Deep Residual Learning for Image Recognition. arXiv:1512.03385, 2015

[6] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. arXiv:1512.00567, 2015

[7] François Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. arXiv:1610.02357, 2016

[8] TensorFlowofficial site:tensorflow.google.cn

[9] Kerasofficial site:keras.io

[10] Keras applications introduction:keras.io/applications/

[11] Keras ImageDataGenerator introduction:keras.io/preprocessing/image/

[12] H5py Datasets API：docs.h5py.org/en/latest/high/dataset.html

[13] Diederik Kingma, Jimmy Ba. Adam: A Method for Stochastic Optimization. arXiv:1412.6980, 2014

[14] Pandas dataframe API introduction: pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html

[15] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861, 2017