Machine Learning Engineer Nanodegree

Capstone Project

Xu Lin
February 8th, 2018

I. Definition

Project Overview

There are lots of things that is easy for humans, dogs, cats, but your computer will find it a bit more difficult. Web services are often protected with a challenge that's supposed to be easy for people to solve, but difficult for computers. Such a challenge is often called a CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) or HIP (Human Interactive Proof). HIPs are used for many purposes, such as to reduce email and blog spam and prevent brute-force attacks on web site passwords.

Identify photographs of cats and dogs. This task is difficult for computers, but studies have shown that people can accomplish it quickly and accurately. Kaggle has redux version for this. 1314 teams have been participated in this project and the leaderboard can be viewed here. Also there are some academic research about this project, like here

My personal motivation for this is improving my web crawler performance. Lots of time my web crawler will stop when it comes with CAPTCHA. I believe this project will help me have betting understanding how to deal with this situation.

Problem Statement

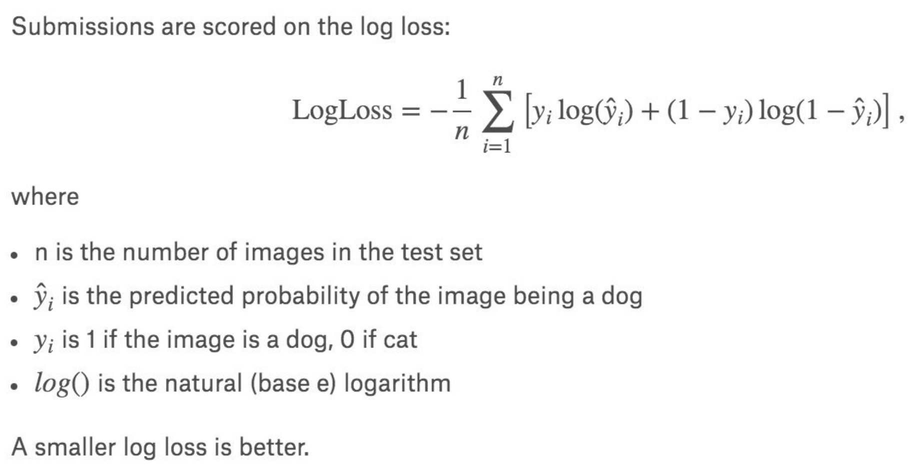
Our basic task is to create an algorithm to classify whether an image contains a dog or a cat. The input for this task are images of dogs or cats from training dataset, while the output is the classification accuracy on test dataset. TFLearn might be applied as one of the solution method.

We will have a data set (images) with label indicated whether it is a dog or cat, we expect we can come up with an algorithm to calculate the probability that the image is a dog (1 = dog, 0 = cat).

We will use Log Loss to value the result.

Metrics

The project is evaluated by log loss



Submissions are scored on the log loss:

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

where

- n is the number of images in the test set
- $\hat{y}_i$ is the predicted probability of the image being a dog
- y_i is 1 if the image is a dog, 0 if cat
- $log()$ is the natural (base e) logarithm

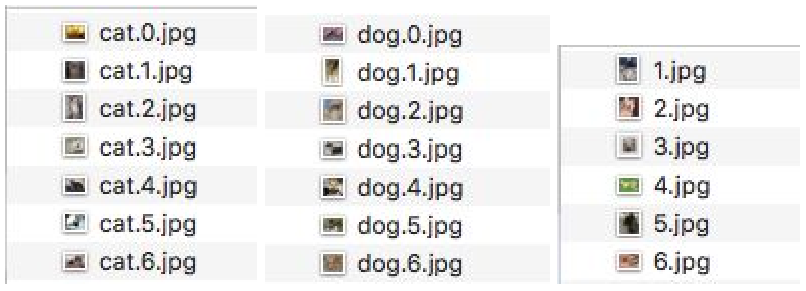
A smaller log loss is better.

II. Analysis

Data Exploration and Exploratory Visualization

The data is coming from here The train folder contains 25,000 images of dogs and cats. Each image in this folder has the label as part of the filename. The test folder contains 12,500 images, named according to a numeric id. For each image in the test set, you should predict a probability that the image is a dog (1 = dog, 0 = cat).

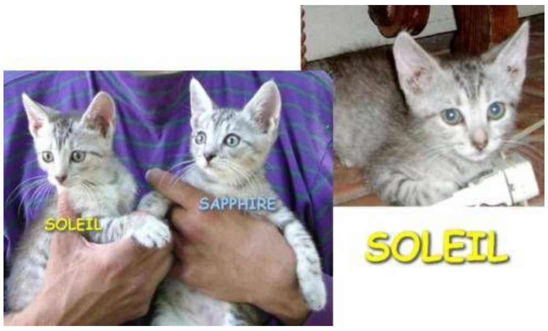
Files screenshots like this:

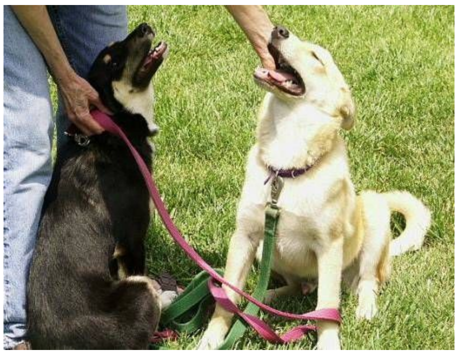
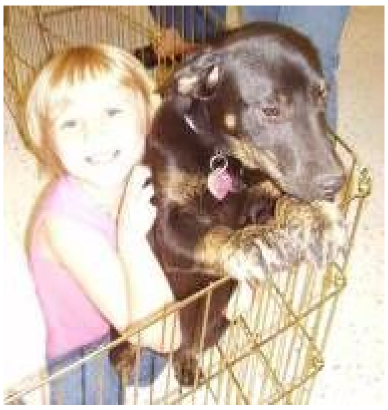
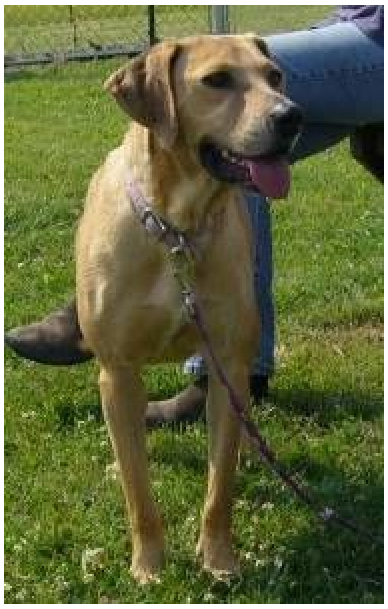


cat.0.jpg	dog.0.jpg	
cat.1.jpg	dog.1.jpg	1.jpg
cat.2.jpg	dog.2.jpg	2.jpg
cat.3.jpg	dog.3.jpg	3.jpg
cat.4.jpg	dog.4.jpg	4.jpg
cat.5.jpg	dog.5.jpg	5.jpg
cat.6.jpg	dog.6.jpg	6.jpg

Regular dog and cat image will be like this:

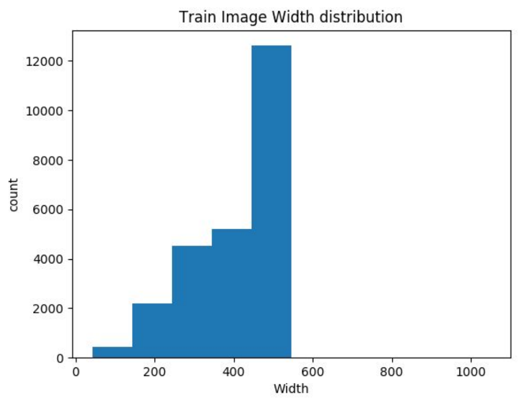
But you will find some special images, which make this classification exercise challenging:

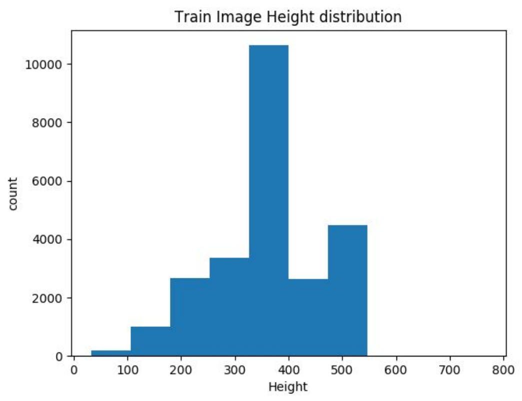


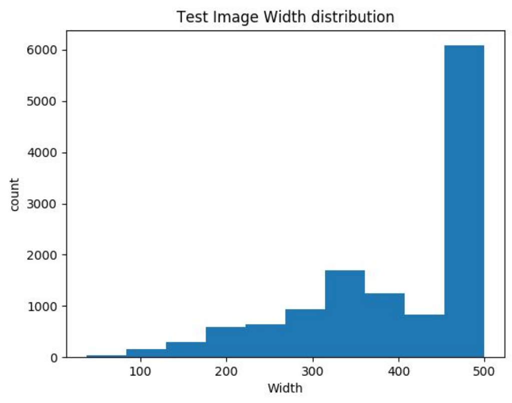




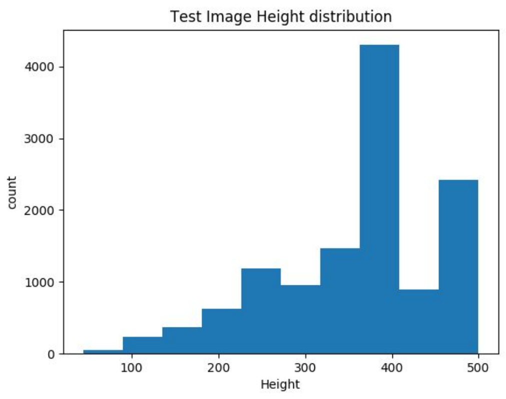
1. image has people or other objects
2. some images have more than 1 dog/cat
3. dog/cat is not always centered

4. weight height ratio is different
5. light amount is different









Algorithms and Techniques

Convolutional neural networks (CNNs) are widely used in pattern- and image-recognition problems as they have a number of advantages compared to other techniques. As for this project, I am trying to go with CNN as follow steps:

1. train a small network from scratch, and see the result
2. trying to improve it with add more layers
3. trying to use some famose model like vgg16, vgg 19, etc
4. trying to combine those famouse model

Benchmark

According to Kaggle Leaderboard, the first position of 1314 teams was taken by team "Cocostarcu", their Log Loss score is 0.03302. I hope I can be the top 20% of this project, which mean the Log Loss result should be lower than 0.08167. After the final result is submitted to Kaggle, it will show the final Log Loss.

III. Methodology

Data Preprocessing

Even though we know there are a few challenging “outliner”s, but since it is a Kaggle competition, we are not allowed to just remove them.
I compared the ImageAugmentation from TFlearn with ImageDataGenerator from Keras, it turns out ImageDataGenerator has more powerful features than ImageAugmentation. This class allows you to:

- configure random transformations and normalization operations to be done on your image data during training
- instantiate generators of augmented image batches (and their labels) via .flow(data, labels) or .flow_from_directory(directory). These generators can then be used with the Keras model methods that accept data generators as inputs, fit_generator, evaluate_generator and predict_generator.

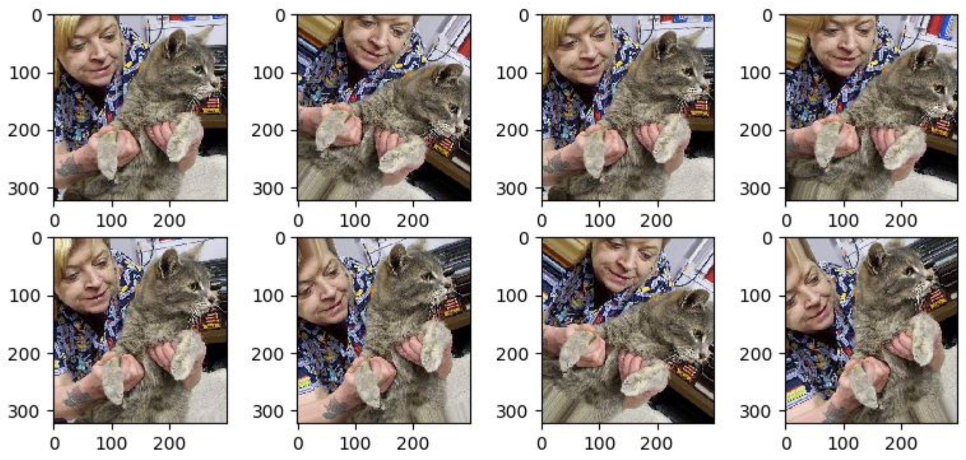
So I picked Keras as out project main training tool.

Let’s go through some parameters of ImageDataGenerator and see if they are useful for this project.

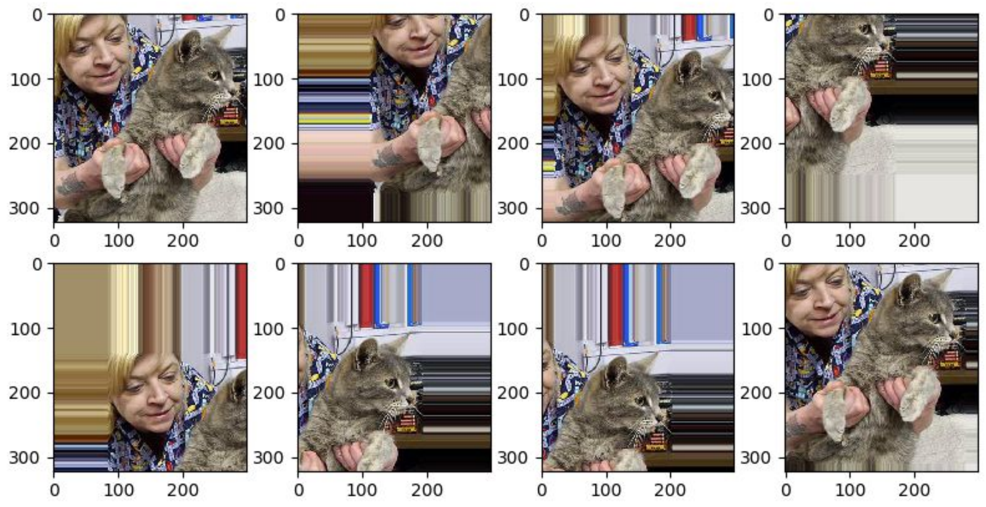
- rotation_range is a value in degrees (0-180), a range within which to randomly rotate pictures
- width_shift and height_shift are ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally
- rescale is a value by which we will multiply the data before any other processing. Our original images consist in RGB coefficients in the 0-255, but such values would be too high for our models to process (given a typical learning rate), so we target values between 0 and 1 instead by scaling with a 1/255. factor.
- shear_range is for randomly applying shearing transformations
- zoom_range is for randomly zooming inside pictures
- horizontal_flip is for randomly flipping half of the images horizontally --relevant when there are no assumptions of horizontal asymmetry (e.g. real-world pictures).
- fill_mode is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift.

Real image after process example (Left top corner image is original one):

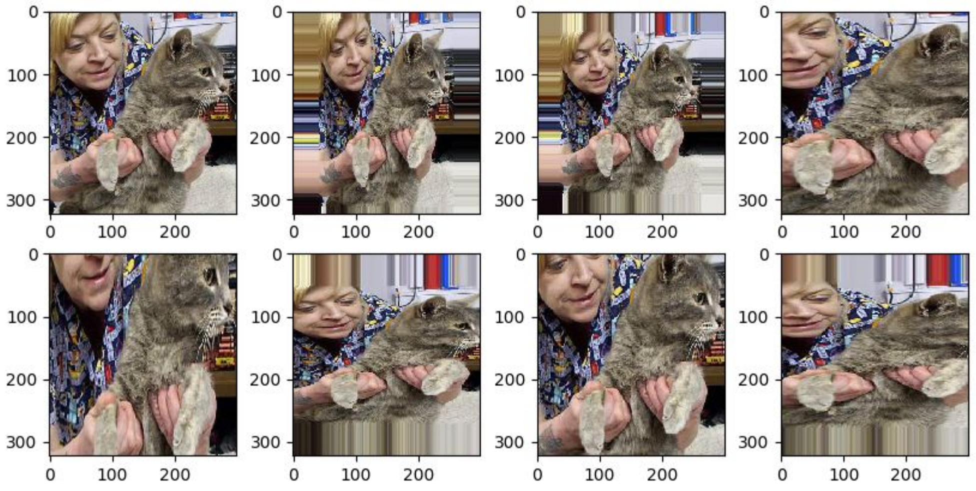
rotation_range = 40



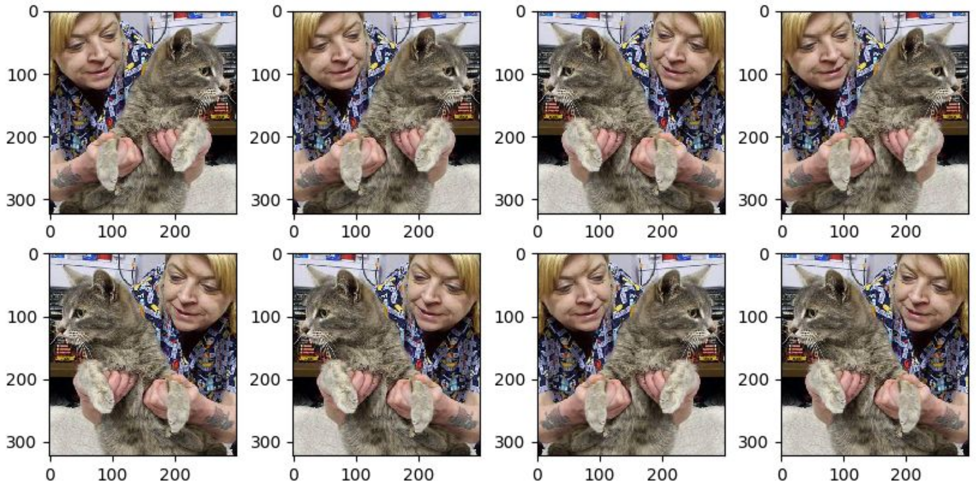
width_shift_range = 0.5, height_shift_range = 0.5



zoom_range = 0.5

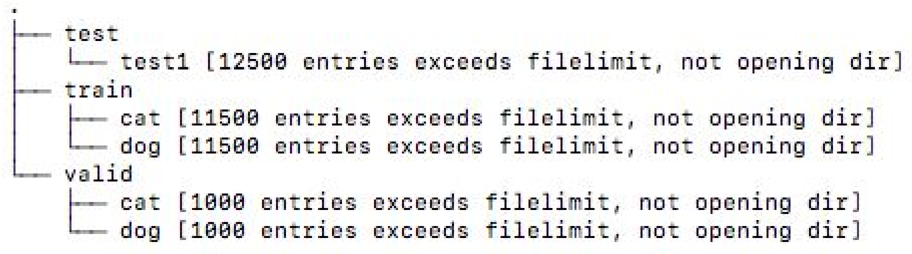


horizontal_flip = True



In the project, I gonna start with rescale = 1/225, shear_range=0.2, zoom_range=0.2, horizontal_flip=True.

In order to use flow_from_directory function of ImageDataGenerator, I need to reorganise the folder structure. it will be like this:



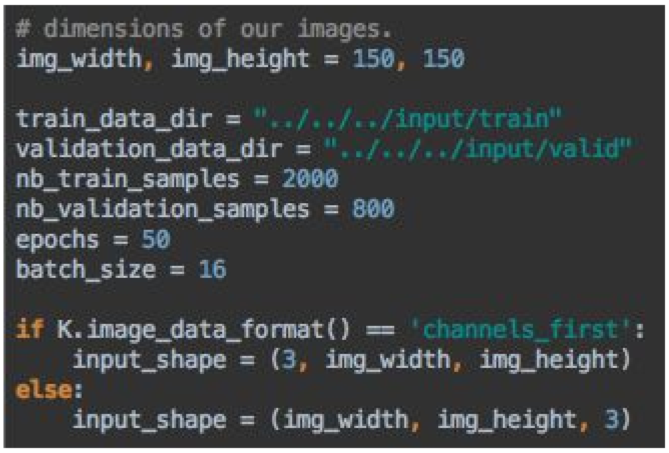
```
.
├── test
│   └── test1 [12500 entries exceeds filelimit, not opening dir]
├── train
│   ├── cat [11500 entries exceeds filelimit, not opening dir]
│   └── dog [11500 entries exceeds filelimit, not opening dir]
└── valid
    ├── cat [1000 entries exceeds filelimit, not opening dir]
    └── dog [1000 entries exceeds filelimit, not opening dir]
```

12500 images of test1 folder are coming from original test folder. 12500 dog/cat training image are divided in to 11500 and 1000 and put into train and valid folder.

Implementation

In our case we will use a very small convnet with few layers and few filters per layer, alongside data augmentation and dropout. Dropout also helps reduce overfitting, by preventing a layer from seeing twice the exact same pattern, thus acting in a way analogous to data augmentation (you could say that both dropout and data augmentation tend to disrupt random correlations occuring in your data).

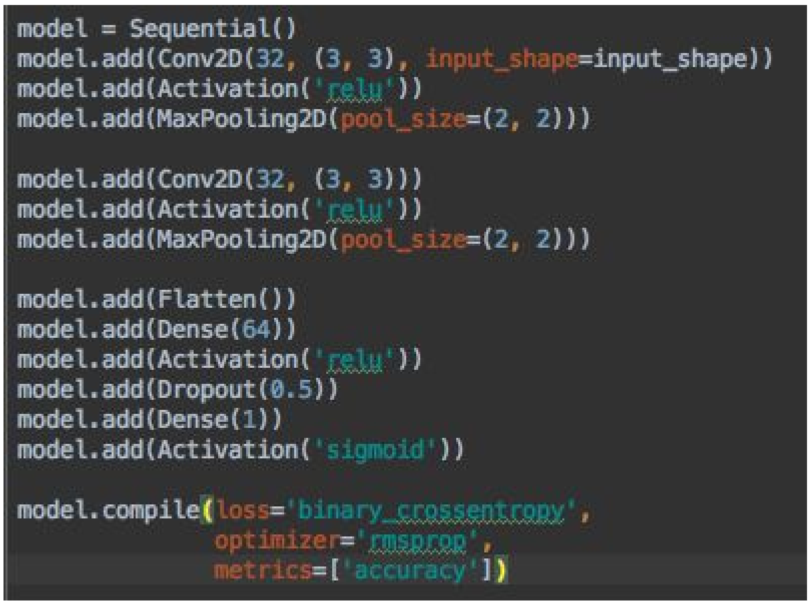
Prepare data set:



```
# dimensions of our images.
img_width, img_height = 150, 150

train_data_dir = "../../../input/train"
validation_data_dir = "../../../input/valid"
nb_train_samples = 2000
nb_validation_samples = 800
epochs = 50
batch_size = 16

if K.image_data_format() == 'channels_first':
    input_shape = (3, img_width, img_height)
else:
    input_shape = (img_width, img_height, 3)
```

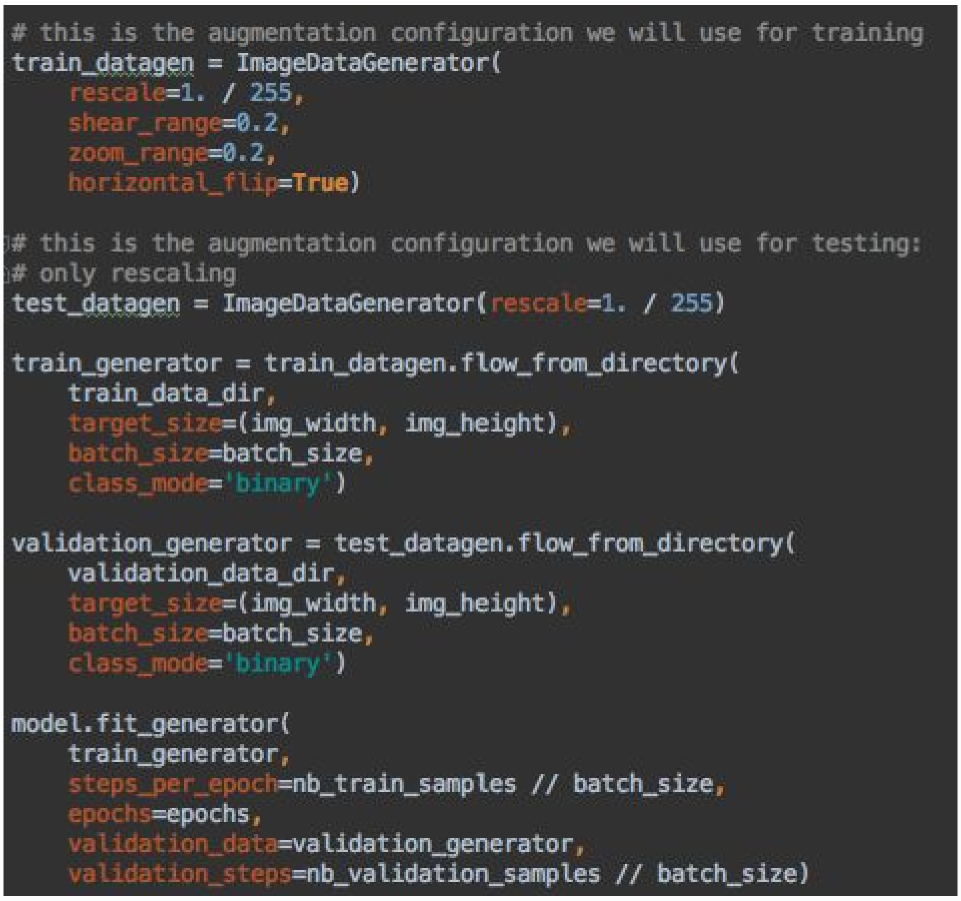
The code snippet below is our first model, a simple stack of 2 convolution layers with a ReLU activation and followed by max-pooling layers.

On top of it we stick two fully-connected layers. We end the model with a single unit and a sigmoid activation, which is perfect for a binary classification. To go with it we will also use the binary_crossentropy loss to train our model.



```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=input_shape))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

Use .flow_from_directory() to generate batches of image data (and their labels) directly from our jpgs in their respective folders



```
# this is the augmentation configuration we will use for training
train_datagen = ImageDataGenerator(
    rescale=1. / 255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True)

# this is the augmentation configuration we will use for testing:
# only rescaling
test_datagen = ImageDataGenerator(rescale=1. / 255)

train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
    validation_data_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='binary')

model.fit_generator(
    train_generator,
    steps_per_epoch=nb_train_samples // batch_size,
    epochs=epochs,
    validation_data=validation_generator,
    validation_steps=nb_validation_samples // batch_size)
```

- loss: 0.5185 - acc: 0.7605 - val_loss: 0.4469 - val_acc: 0.8063

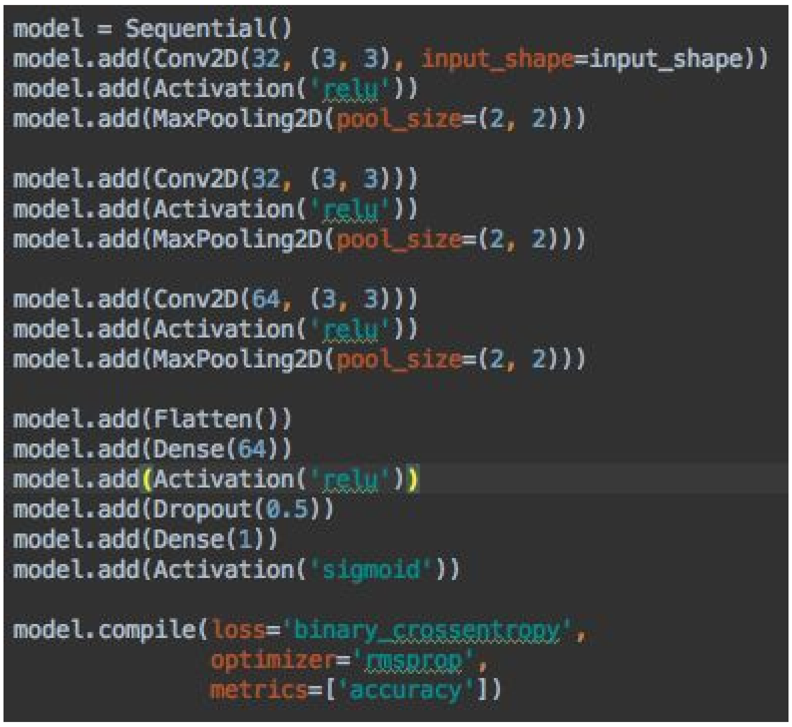
Note that the validation accuracy is 80.63%, because the model is small and uses aggressive dropout, it doesn't seem to be overfitting too much by this point.

Refinement

In this section, I'm maining trying to change training model to achieve better result.

Approach 1:

Add 1 more layer with relu activaition and followed by max-pooling layers.

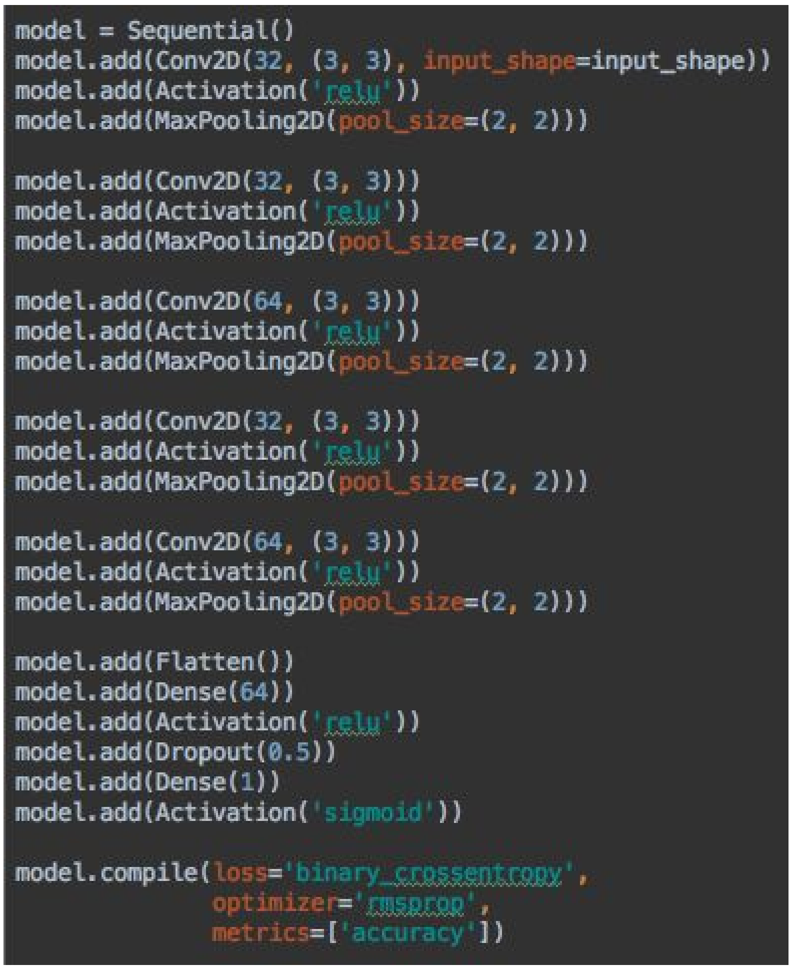


```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=input_shape))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

- loss: 0.4864 - acc: 0.7710 - val_loss: 0.4067 - val_acc: 0.8250
Note that the validation accuracy is 82.50%, it has around 2% improvement.

Approach 2:

Add 3 more layer with relu activaition and followed by max-pooling layers.



```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=input_shape))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

- loss: 0.4731 - acc: 0.7880 - val_loss: 0.4018 - val_acc: 0.8562
Note that the validation accuracy is 85.62%, it also has around 5% improvement.

But when I keep adding layers, the result is not improved a lot, which means adding layers blindly might not be a good solution.

Approach 3:

After trying set up my own network, I would like to try Transfer Learning using some pre-trained networks, like

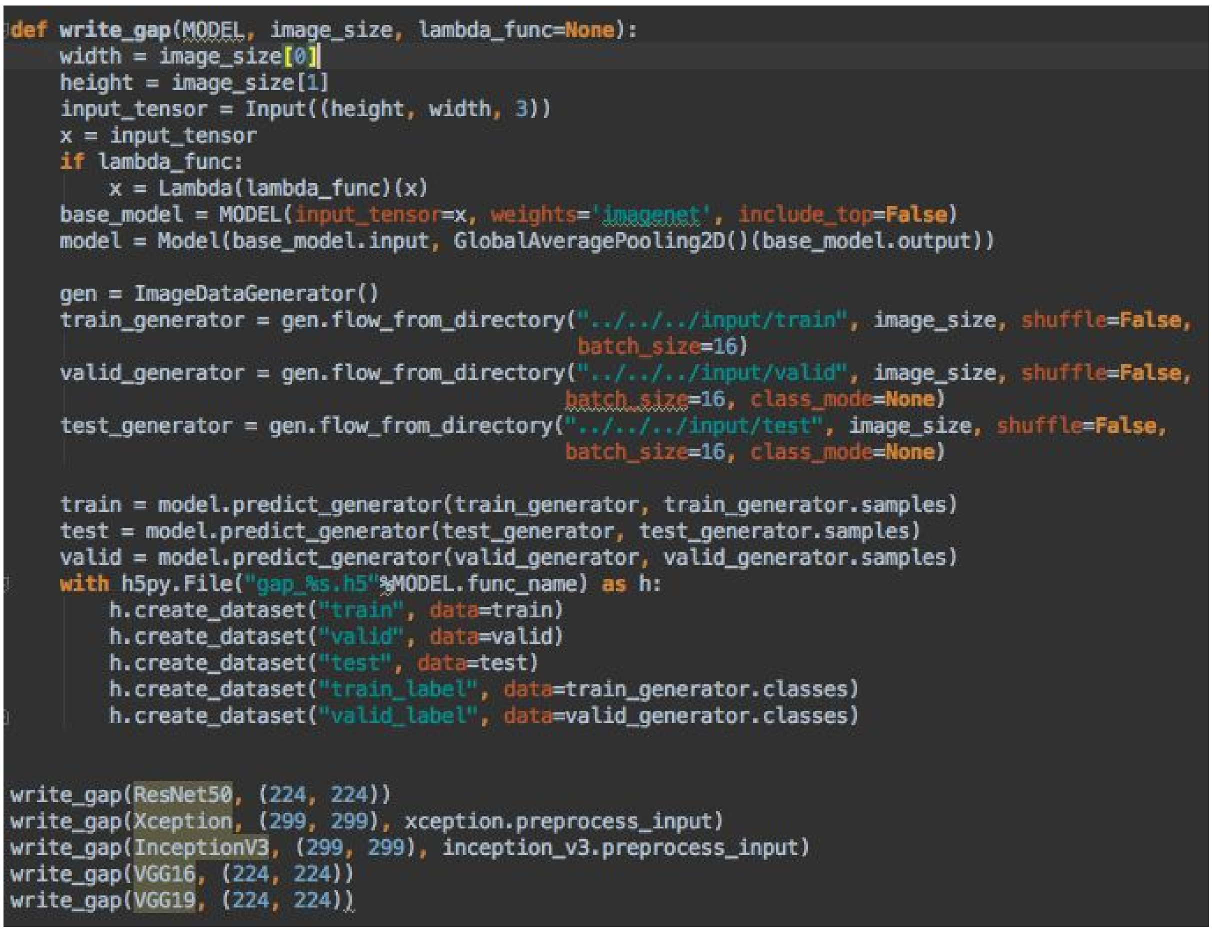
- InceptionV3
- ResNet50
- VGG16
- VGG19
- Xception

Because the ImageNet dataset contains several "cat" classes (persian cat, siamese cat...) and many "dog" classes among its total of 1000 classes, this model will already have learned features that are relevant to our classification problem. That is the reason why we can use transfer learning here.

Thanks to PeiWen Yang provide the idea.

Basic idea for this project is using those pre-trained network to predict our test/train images, get feature vectors and export to .h5 file. In this way, we can reuse those feature vectors to get better weight.

First, let's have a method to generate those feature vectors. Parameters will be pre-trained network name and their default input image size, corresponding image input tensor. All the weight for those pre-trained network gonna be “imagenet”.

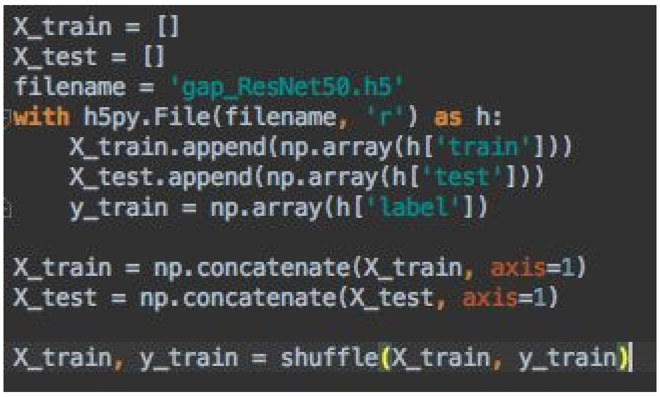


```
def write_gap(MODEL, image_size, lambda_func=None):
    width = image_size[0]
    height = image_size[1]
    input_tensor = Input((height, width, 3))
    x = input_tensor
    if lambda_func:
        x = Lambda(lambda_func)(x)
    base_model = MODEL(input_tensor=x, weights='imagenet', include_top=False)
    model = Model(base_model.input, GlobalAveragePooling2D()(base_model.output))

    gen = ImageDataGenerator()
    train_generator = gen.flow_from_directory("../../../input/train", image_size, shuffle=False,
                                              batch_size=16)
    valid_generator = gen.flow_from_directory("../../../input/valid", image_size, shuffle=False,
                                             batch_size=16, class_mode=None)
    test_generator = gen.flow_from_directory("../../../input/test", image_size, shuffle=False,
                                             batch_size=16, class_mode=None)

    train = model.predict_generator(train_generator, train_generator.samples)
    test = model.predict_generator(test_generator, test_generator.samples)
    valid = model.predict_generator(valid_generator, valid_generator.samples)
    with h5py.File("gap_%s.h5"%MODEL.func_name) as h:
        h.create_dataset("train", data=train)
        h.create_dataset("valid", data=valid)
        h.create_dataset("test", data=test)
        h.create_dataset("train_label", data=train_generator.classes)
        h.create_dataset("valid_label", data=valid_generator.classes)


write_gap(ResNet50, (224, 224))
write_gap(Xception, (299, 299), xception.preprocess_input)
write_gap(InceptionV3, (299, 299), inception_v3.preprocess_input)
write_gap(VGG16, (224, 224))
write_gap(VGG19, (224, 224))
```

The features vector will contrain 3 numpy vector:

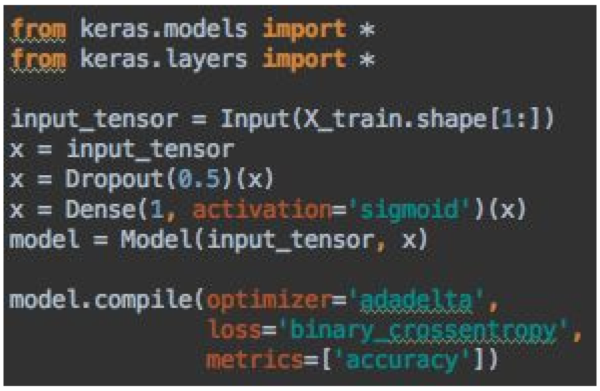
- train(25000, 2048)
- test(12500, 2048)
- label(25000,)

We can get X_train from “train” vector, get Y_train from “label” vector, X_test from “test” vector.



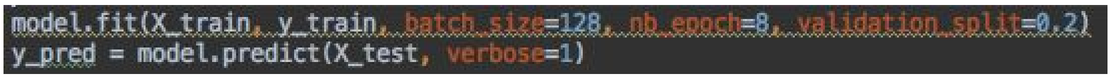
```
X_train = []
X_test = []
filename = 'gap_ResNet50.h5'
with h5py.File(filename, 'r') as h:
    X_train.append(np.array(h['train']))
    X_test.append(np.array(h['test']))
    y_train = np.array(h['label'])

X_train = np.concatenate(X_train, axis=1)
X_test = np.concatenate(X_test, axis=1)

X_train, y_train = shuffle(X_train, y_train)
```

After get this X_train, we can just dropout and create this new model.



```
from keras.models import *
from keras.layers import *

input_tensor = Input(X_train.shape[1:])
x = input_tensor
x = Dropout(0.5)(x)
x = Dense(1, activation='sigmoid')(x)
model = Model(input_tensor, x)

model.compile(optimizer='adadelta',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Then use this model to fit X_train, Y_trian, we can set validation_split = 0.2, which means training set is 20000 image, validation is 5000 images.

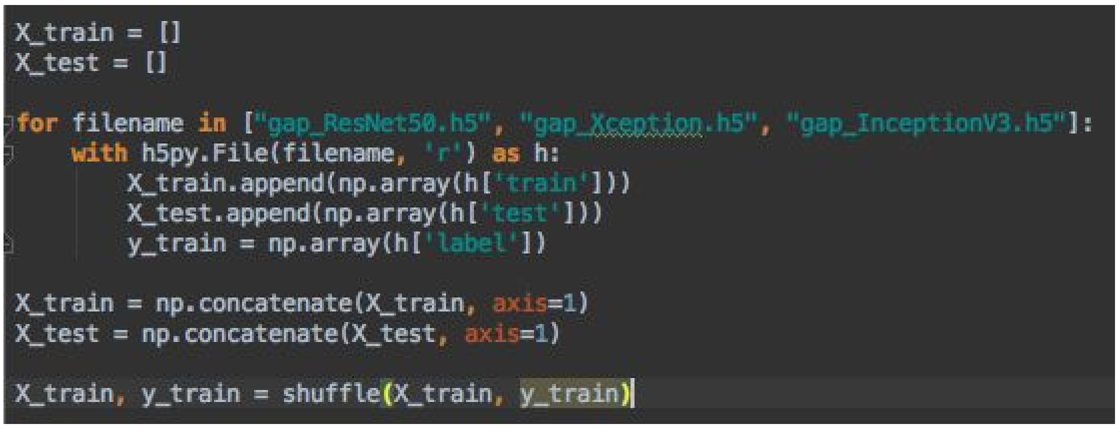


```
model.fit(X_train, y_train, batch_size=128, nb_epoch=8, validation_split=0.2)
y_pred = model.predict(X_test, verbose=1)
```

InceptionV3
- loss: 0.0185 - acc: 0.9937 - val_loss: 0.0266 - val_acc: 0.9912
ResNet50
- loss: 0.0680 - acc: 0.9736 - val_loss: 0.0647 - val_acc: 0.9756
VGG16
- loss: 0.1468 - acc: 0.9460 - val_loss: 0.0788 - val_acc: 0.9700
VGG19
- loss: 0.1354 - acc: 0.9497 - val_loss: 0.0804 - val_acc: 0.9686
Xception
- loss: 0.0210 - acc: 0.9933 - val_loss: 0.0150 - val_acc: 0.9952

We will see those pre-trained networks perform pretty good.

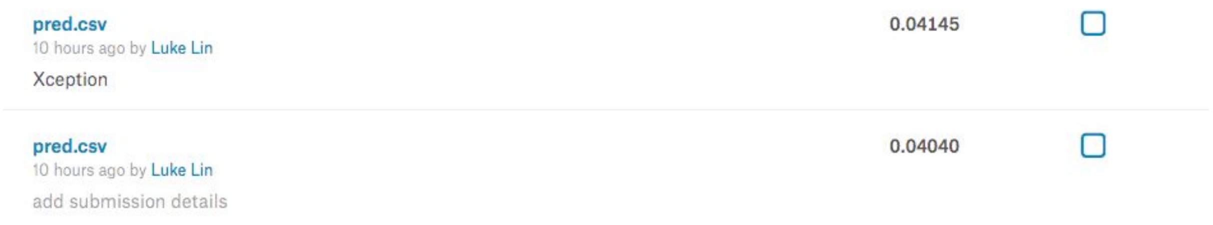
Let's try something new, pick the top 3 networks from those 5, and add their feature vector up, which might have all the benefit from all networks.



```
X_train = []
X_test = []

for filename in ["gap_ResNet50.h5", "gap_Xception.h5", "gap_InceptionV3.h5"]:
    with h5py.File(filename, 'r') as h:
        X_train.append(np.array(h['train']))
        X_test.append(np.array(h['test']))
        y_train = np.array(h['label'])

X_train = np.concatenate(X_train, axis=1)
X_test = np.concatenate(X_test, axis=1)

X_train, y_train = shuffle(X_train, y_train)
```

- loss: 0.0104 - acc: 0.9963 - val_loss: 0.0172 - val_acc: 0.9946
Looks good.

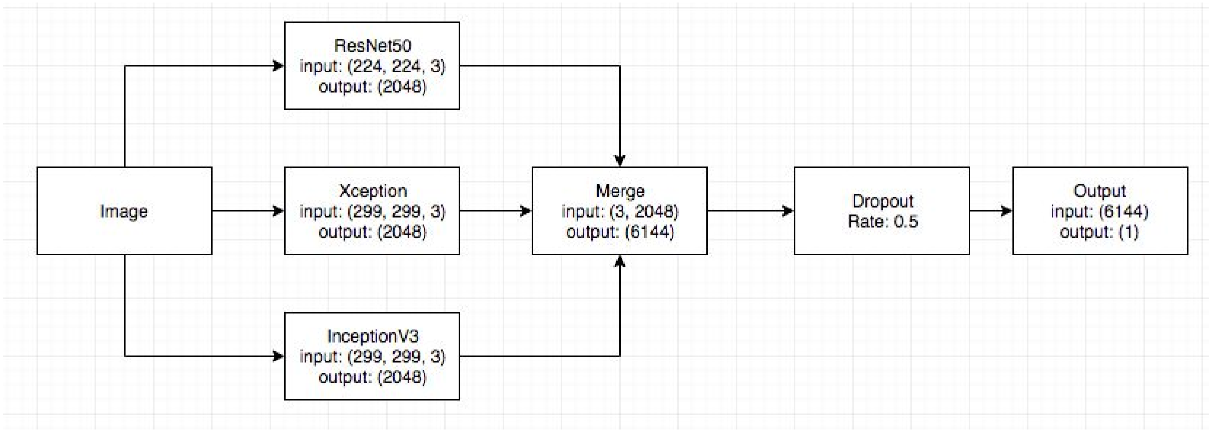
Let's upload the result from Xception and combined model, you will see Xception got 0.04145, combined model got better result 0.04040. The reason might be Xception has better performance on training set, but testing set is kinda more "general", so combined model will do better job here.



pred.csv 10 hours ago by Luke Lin Xception	0.04145	☐
pred.csv 10 hours ago by Luke Lin add submission details	0.04040	☐

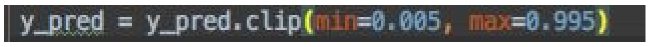
IV. Results

Model Evaluation and Validation



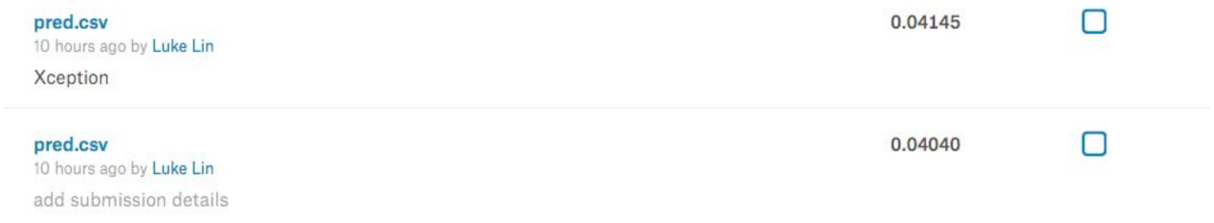
According to final result, combined model have better result than Xception on test set, which proved its robustness.

Justification



```
y_pred = y_pred.clip(min=0.005, max=0.995)
```

There is a small trick here, we clip the prediction value to [0.005, 0.995]. The reason for that is Metrics for this project is LogLoss, so as for correct prediction, 0.995 is almost same with 1, but as for incorrect prediction, 0 is huge difference with 0.005, which is the difference between 15 and 2.



pred.csv 10 hours ago by Luke Lin Xception	0.04145	☐
pred.csv 10 hours ago by Luke Lin add submission details	0.04040	☐

The final result 0.04040 is small the benchmark 0.08167 mentioned above. This solution is good enough to solve this problem. It might better than me.

V. Conclusion

Reflection

I learnt a lot as I worked through this project

- Understanding of deep learning, convolution neural networks and Tensorflow. Working on a real project will help understand those concept in higher level
- ImageDataGenerator is pretty powerful tool, especially you don't have enough data, it will help to enlarge your input data set
- Getting feature vectors will take unknown time on CPU. I compared AWS with Google Cloud to use GPU, I realized AWS is much better than Google Cloud in terms of transparency, documentation, support, resource online.
- Remember to save your work all the time, like saving data/model in to .h5 file. Since training is really time-consuming task, so save your work equals save time.
- Increase layer doesn't mean higher accuracy. It's good to set a stop function when you start to train.
- Don't hesitate to use well-known networks, they all well-trained, got a lots of prizes from different competitions. Since I'm not a researcher from famous lab or a phd who concentrate on building a model. I think the best way of machine learning for me is knowing how to use those existing models, not build on from scratch.

Improvement

- more aggresive data augmentation
- more aggresive dropout
- change weight from different model and combine them