In [1]:

```
import keras
keras.__version__
```

```
Using TensorFlow backend.
```

Out[1]:

```
'2.2.4'
```

In [3]:

```
!pwd
```

```
/Users/jlee/Documents/GitHub/Python-Practice/Harvard CSCI E-63/HW3
```

5.2 - Using convnets with small datasets

This notebook contains the code sample found in Chapter 5, Section 2 of [Deep Learning with Python (https://www.manning.com/books/deep-learning-with-python?a_aid=keras&a_bid=76564dff)](https://www.manning.com/books/deep-learning-with-python?a_aid=keras&a_bid=76564dff). Note that the original text features far more content, in particular further explanations and figures: in this notebook, you will only find source code and related comments.

Training a convnet from scratch on a small dataset

Having to train an image classification model using only very little data is a common situation, which you likely encounter yourself in practice if you ever do computer vision in a professional context.

Having "few" samples can mean anywhere from a few hundreds to a few tens of thousands of images. As a practical example, we will focus on classifying images as "dogs" or "cats", in a dataset containing 4000 pictures of cats and dogs (2000 cats, 2000 dogs). We will use 2000 pictures for training, 1000 for validation, and finally 1000 for testing.

In this section, we will review one basic strategy to tackle this problem: training a new model from scratch on what little data we have. We will start by naively training a small convnet on our 2000 training samples, without any regularization, to set a baseline for what can be achieved. This will get us to a classification accuracy of 71%. At that point, our main issue will be overfitting. Then we will introduce *data augmentation*, a powerful technique for mitigating overfitting in computer vision. By leveraging data augmentation, we will improve our network to reach an accuracy of 82%.

In the next section, we will review two more essential techniques for applying deep learning to small datasets: *doing feature extraction with a pre-trained network* (this will get us to an accuracy of 90% to 93%), and *fine-tuning a pre-trained network* (this will get us to our final accuracy of 95%). Together, these three strategies -- training a small model from scratch, doing feature extracting using a pre-trained model, and fine-tuning a pre-trained model -- will constitute your future toolbox for tackling the problem of doing computer vision with small datasets.

The relevance of deep learning for small-data problems

You will sometimes hear that deep learning only works when lots of data is available. This is in part a valid point: one fundamental characteristic of deep learning is that it is able to find interesting features in the training data on its own, without any need for manual feature engineering, and this can only be achieved when lots of training examples are available. This is especially true for problems where the input samples are very high-dimensional, like images.

However, what constitutes "lots" of samples is relative -- relative to the size and depth of the network you are trying to train, for starters. It isn't possible to train a convnet to solve a complex problem with just a few tens of samples, but a few hundreds can potentially suffice if the model is small and well-regularized and if the task is simple. Because convnets learn local, translation-invariant features, they are very data-efficient on perceptual problems. Training a convnet from scratch on a very small image dataset will still yield reasonable results despite a relative lack of data, without the need for any custom feature engineering. You will see this in action in this section.

But what's more, deep learning models are by nature highly repurposable: you can take, say, an image classification or speech-to-text model trained on a large-scale dataset then reuse it on a significantly different problem with only minor changes. Specifically, in the case of computer vision, many pre-trained models (usually trained on the ImageNet dataset) are now publicly available for download and can be used to bootstrap powerful vision models out of very little data. That's what we will do in the next section.

For now, let's get started by getting our hands on the data.

Downloading the data

The cats vs. dogs dataset that we will use isn't packaged with Keras. It was made available by Kaggle.com as part of a computer vision competition in late 2013, back when convnets weren't quite mainstream. You can download the original dataset at: `https://www.kaggle.com/c/dogs-vs-cats/data` (you will need to create a Kaggle account if you don't already have one -- don't worry, the process is painless).

The pictures are medium-resolution color JPEGs. They look like this:

Unsurprisingly, the cats vs. dogs Kaggle competition in 2013 was won by entrants who used convnets. The best entries could achieve up to 95% accuracy. In our own example, we will get fairly close to this accuracy (in the next section), even though we will be training our models on less than 10% of the data that was available to the competitors. This original dataset contains 25,000 images of dogs and cats (12,500 from each class) and is 543MB large (compressed). After downloading and uncompressing it, we will create a new dataset containing three subsets: a training set with 1000 samples of each class, a validation set with 500 samples of each class, and finally a test set with 500 samples of each class.

Here are a few lines of code to do this:

In [2]:

```
import os, shutil
```

In [8]:

```
# The path to the directory where the original
```

```
# dataset was uncompressed
original_dataset_dir = '/Users/jlee/Documents/GitHub/Python-Practice/Harvard CSCI E-

# The directory where we will
# store our smaller dataset
base_dir = '/Users/jlee/Documents/GitHub/Python-Practice/Harvard CSCI E-63/HW3/smal
os.mkdir(base_dir)

# Directories for our training,
# validation and test splits
train_dir = os.path.join(base_dir, 'train')
os.mkdir(train_dir)
validation_dir = os.path.join(base_dir, 'validation')
os.mkdir(validation_dir)
test_dir = os.path.join(base_dir, 'test')
os.mkdir(test_dir)

# Directory with our training cat pictures
train_cats_dir = os.path.join(train_dir, 'cats')
os.mkdir(train_cats_dir)

# Directory with our training dog pictures
train_dogs_dir = os.path.join(train_dir, 'dogs')
os.mkdir(train_dogs_dir)

# Directory with our validation cat pictures
validation_cats_dir = os.path.join(validation_dir, 'cats')
os.mkdir(validation_cats_dir)

# Directory with our validation dog pictures
validation_dogs_dir = os.path.join(validation_dir, 'dogs')
os.mkdir(validation_dogs_dir)

# Directory with our validation cat pictures
test_cats_dir = os.path.join(test_dir, 'cats')
os.mkdir(test_cats_dir)

# Directory with our validation dog pictures
test_dogs_dir = os.path.join(test_dir, 'dogs')
os.mkdir(test_dogs_dir)

# Copy first 1000 cat images to train_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_cats_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 cat images to validation_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_cats_dir, fname)
    shutil.copyfile(src, dst)
```

```
# Copy next 500 cat images to test_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_cats_dir, fname)
    shutil.copyfile(src, dst)

# Copy first 1000 dog images to train_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_dogs_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 dog images to validation_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_dogs_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 dog images to test_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_dogs_dir, fname)
    shutil.copyfile(src, dst)
```

As a sanity check, let's count how many pictures we have in each training split (train/validation/test):

In [9]:

```
print('total training cat images:', len(os.listdir(train_cats_dir)))
```

```
total training cat images: 1000
```

In [10]:

```
print('total training dog images:', len(os.listdir(train_dogs_dir)))
```

```
total training dog images: 1000
```

In [11]:

```
print('total validation cat images:', len(os.listdir(validation_cats_dir)))
```

```
total validation cat images: 500
```

In [12]:

```
print('total validation dog images:', len(os.listdir(validation_dogs_dir)))
```

```
total validation dog images: 500
```

In [13]:

```
print('total test cat images:', len(os.listdir(test_cats_dir)))
```

```
total test cat images: 500
```

In [14]:

```
print('total test dog images:', len(os.listdir(test_dogs_dir)))
```

```
total test dog images: 500
```

So we have indeed 2000 training images, and then 1000 validation images and 1000 test images. In each split, there is the same number of samples from each class: this is a balanced binary classification problem, which means that classification accuracy will be an appropriate measure of success.

Building our network

We've already built a small convnet for MNIST in the previous example, so you should be familiar with them. We will reuse the same general structure: our convnet will be a stack of alternated `Conv2D` (with `relu` activation) and `MaxPooling2D` layers.

However, since we are dealing with bigger images and a more complex problem, we will make our network accordingly larger: it will have one more `Conv2D + MaxPooling2D` stage. This serves both to augment the capacity of the network, and to further reduce the size of the feature maps, so that they aren't overly large when we reach the `Flatten` layer. Here, since we start from inputs of size 150x150 (a somewhat arbitrary choice), we end up with feature maps of size 7x7 right before the `Flatten` layer.

Note that the depth of the feature maps is progressively increasing in the network (from 32 to 128), while the size of the feature maps is decreasing (from 148x148 to 7x7). This is a pattern that you will see in almost all convnets.

Since we are attacking a binary classification problem, we are ending the network with a single unit (a `Dense` layer of size 1) and a `sigmoid` activation. This unit will encode the probability that the network is looking at one class or the other.

In [15]:

```python
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

```
WARNING:tensorflow:From /Users/jlee/anaconda3/lib/python3.7/site-packa
ges/tensorflow/python/framework/op_def_library.py:263: colocate_with (
from tensorflow.python.framework.ops) is deprecated and will be remove
d in a future version.
Instructions for updating:
Colocations handled automatically by placer.
```

Let's take a look at how the dimensions of the feature maps change with every successive layer:

In [16]:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 17, 17, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 6272)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               3211776
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
_________________________________________________________________
```

For our compilation step, we'll go with the `RMSprop` optimizer as usual. Since we ended our network with a single sigmoid unit, we will use binary crossentropy as our loss (as a reminder, check out the table in Chapter 4, section 5 for a cheatsheet on what loss function to use in various situations).

In [17]:

```
from keras import optimizers

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```

Data preprocessing

As you already know by now, data should be formatted into appropriately pre-processed floating point tensors before being fed into our network. Currently, our data sits on a drive as JPEG files, so the steps for getting it into our network are roughly:

- Read the picture files.
- Decode the JPEG content to RBG grids of pixels.
- Convert these into floating point tensors.
- Rescale the pixel values (between 0 and 255) to the [0, 1] interval (as you know, neural networks prefer to deal with small input values).

It may seem a bit daunting, but thankfully Keras has utilities to take care of these steps automatically. Keras has a module with image processing helper tools, located at `keras.preprocessing.image`. In particular, it contains the class `ImageDataGenerator` which allows to quickly set up Python generators that can automatically turn image files on disk into batches of pre-processed tensors. This is what we will use here.

In [18]:

```
from keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        # This is the target directory
        train_dir,
        # All images will be resized to 150x150
        target_size=(150, 150),
        batch_size=20,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')
```

```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
```

Let's take a look at the output of one of these generators: it yields batches of 150x150 RGB images (shape `(20, 150, 150, 3)`) and binary labels (shape `(20,)`). 20 is the number of samples in each batch (the batch size). Note that the generator yields these batches indefinitely: it just loops endlessly over the images present in the target folder. For this reason, we need to `break` the iteration loop at some point.

We need to install pillow package. Pillow is a PIL fork. PIL is Pyhton Image Library

In [19]:

```
!pip install pillow
```

```
Requirement already satisfied: pillow in /Users/jlee/anaconda3/lib/pyt
hon3.7/site-packages (5.3.0)
```

In [20]:

```
for data_batch, labels_batch in train_generator:
    print('data batch shape:', data_batch.shape)
    print('labels batch shape:', labels_batch.shape)
    break
```

```
data batch shape: (20, 150, 150, 3)
labels batch shape: (20,)
```

Let's fit our model to the data using the generator. We do it using the `fit_generator` method, the equivalent of `fit` for data generators like ours. It expects as first argument a Python generator that will yield batches of inputs and targets indefinitely, like ours does. Because the data is being generated endlessly, the generator needs to know example how many samples to draw from the generator before declaring an epoch over. This is the role of the `steps_per_epoch` argument: after having drawn `steps_per_epoch` batches from the generator, i.e. after having run for `steps_per_epoch` gradient descent steps, the fitting process will go to the next epoch. In our case, batches are 20-sample large, so it will take 100 batches until we see our target of 2000 samples.

When using `fit_generator`, one may pass a `validation_data` argument, much like with the `fit` method. Importantly, this argument is allowed to be a data generator itself, but it could be a tuple of Numpy arrays as well. If you pass a generator as `validation_data`, then this generator is expected to yield batches of validation data endlessly, and thus you should also specify the `validation_steps` argument, which tells the process how many batches to draw from the validation generator for evaluation.

In [21]:

```
history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=15,
      validation_data=validation_generator,
      validation_steps=50)
```

```
WARNING:tensorflow:From /Users/jlee/anaconda3/lib/python3.7/site-packa
ges/tensorflow/python/ops/math_ops.py:3066: to_int32 (from tensorflow.
python.ops.math_ops) is deprecated and will be removed in a future ver
sion.
Instructions for updating:
Use tf.cast instead.
Epoch 1/15
```

```
100/100 [==============================] - 101s 1s/step - loss: 0.6921
- acc: 0.5145 - val_loss: 0.6782 - val_acc: 0.5460
Epoch 2/15
100/100 [==============================] - 119s 1s/step - loss: 0.6610
- acc: 0.6095 - val_loss: 0.6416 - val_acc: 0.6350
Epoch 3/15
100/100 [==============================] - 97s 975ms/step - loss: 0.60
29 - acc: 0.6835 - val_loss: 0.6186 - val_acc: 0.6400
Epoch 4/15
100/100 [==============================] - 93s 928ms/step - loss: 0.56
94 - acc: 0.6935 - val_loss: 0.6107 - val_acc: 0.6680
Epoch 5/15
100/100 [==============================] - 102s 1s/step - loss: 0.5338
- acc: 0.7365 - val_loss: 0.6267 - val_acc: 0.6380
Epoch 6/15
100/100 [==============================] - 100s 999ms/step - loss: 0.5
124 - acc: 0.7405 - val_loss: 0.5858 - val_acc: 0.6920
Epoch 7/15
100/100 [==============================] - 104s 1s/step - loss: 0.4816
- acc: 0.7725 - val_loss: 0.5675 - val_acc: 0.6980
Epoch 8/15
100/100 [==============================] - 103s 1s/step - loss: 0.4461
- acc: 0.7835 - val_loss: 0.5857 - val_acc: 0.6880
Epoch 9/15
100/100 [==============================] - 96s 956ms/step - loss: 0.43
21 - acc: 0.8105 - val_loss: 0.5753 - val_acc: 0.7090
Epoch 10/15
100/100 [==============================] - 95s 954ms/step - loss: 0.40
19 - acc: 0.8160 - val_loss: 0.5463 - val_acc: 0.7220
Epoch 11/15
100/100 [==============================] - 83s 829ms/step - loss: 0.36
58 - acc: 0.8470 - val_loss: 0.6058 - val_acc: 0.7060
Epoch 12/15
100/100 [==============================] - 100s 1s/step - loss: 0.3362
- acc: 0.8555 - val_loss: 0.5783 - val_acc: 0.7280
Epoch 13/15
100/100 [==============================] - 104s 1s/step - loss: 0.3141
- acc: 0.8650 - val_loss: 0.5875 - val_acc: 0.7270
Epoch 14/15
100/100 [==============================] - 85s 847ms/step - loss: 0.28
01 - acc: 0.8870 - val_loss: 0.6091 - val_acc: 0.7200
Epoch 15/15
100/100 [==============================] - 84s 840ms/step - loss: 0.26
13 - acc: 0.8925 - val_loss: 0.6065 - val_acc: 0.7300
```

It is good practice to always save your models after training:

In [22]:

```
!pip install h5py
```

```
Requirement already satisfied: h5py in /Users/jlee/anaconda3/lib/pytho
n3.7/site-packages (2.8.0)
Requirement already satisfied: numpy>=1.7 in /Users/jlee/anaconda3/lib
/python3.7/site-packages (from h5py) (1.16.2)
Requirement already satisfied: six in /Users/jlee/anaconda3/lib/python
3.7/site-packages (from h5py) (1.12.0)
```

In [23]:

```
model.save('cats_and_dogs_small_1.h5')
```

Let's plot the loss and accuracy of the model over the training and validation data during training:

In [24]:

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

```
<Figure size 640x480 with 1 Axes>

<Figure size 640x480 with 1 Axes>
```

In [25]:

```
plt.show()
```

These plots are characteristic of overfitting. Our training accuracy increases linearly over time, until it reaches nearly 100%, while our validation accuracy stalls at 70-72%. Our validation loss reaches its minimum after only five epochs then stalls, while the training loss keeps decreasing linearly until it reaches nearly 0.

Because we only have relatively few training samples (2000), overfitting is going to be our number one concern. You already know about a number of techniques that can help mitigate overfitting, such as dropout and weight decay (L2 regularization). We are now going to introduce a new one, specific to computer vision, and used almost universally when processing images with deep learning models: *data augmentation*.

Using data augmentation

Overfitting is caused by having too few samples to learn from, rendering us unable to train a model able to generalize to new data. Given infinite data, our model would be exposed to every possible aspect of the data distribution at hand: we would never overfit. Data augmentation takes the approach of generating more training data from existing training samples, by "augmenting" the samples via a number of random transformations that yield believable-looking images. The goal is that at training time, our model would never see the exact same picture twice. This helps the model get exposed to more aspects of the data and generalize better.

In Keras, this can be done by configuring a number of random transformations to be performed on the images read by our `ImageDataGenerator` instance. Let's get started with an example:

In [26]:

```
datagen = ImageDataGenerator(
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')
```

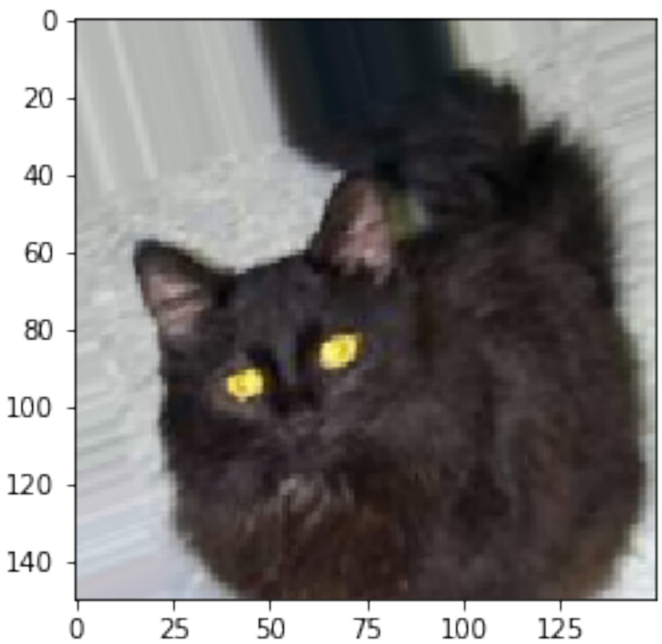
These are just a few of the options available (for more, see the Keras documentation). Let's quickly go over what we just wrote:

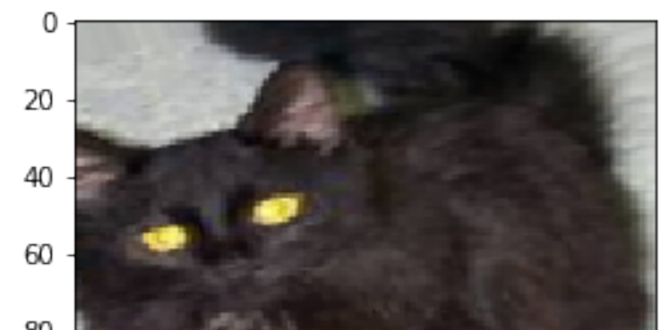
- `rotation_range` is a value in degrees (0-180), a range within which to randomly rotate pictures.
- `width_shift` and `height_shift` are ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally.
- `shear_range` is for randomly applying shearing 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.

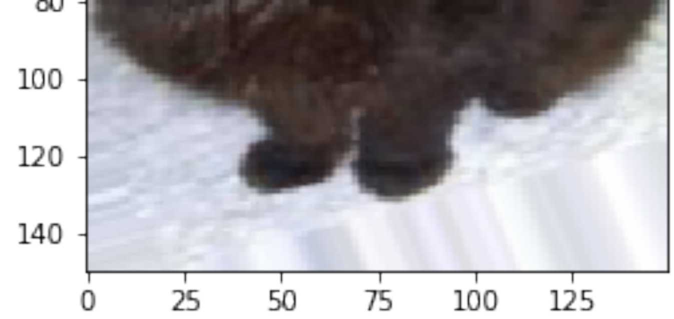
Let's take a look at our augmented images:

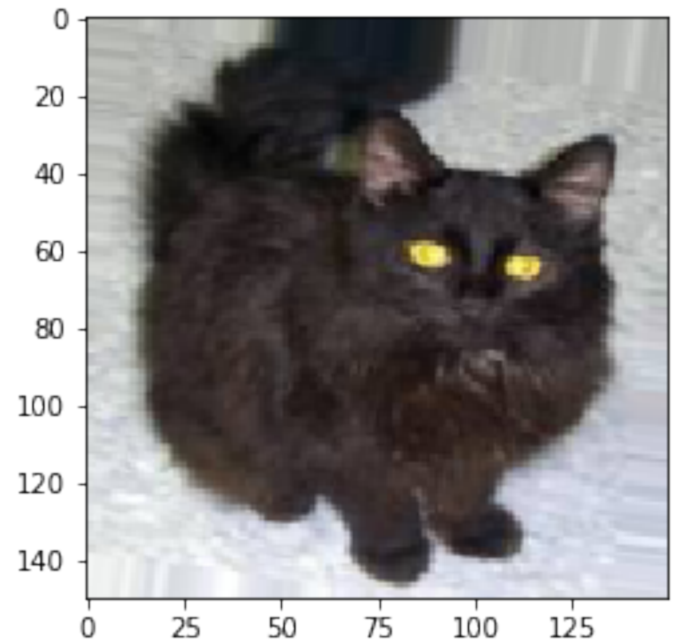
In [27]:

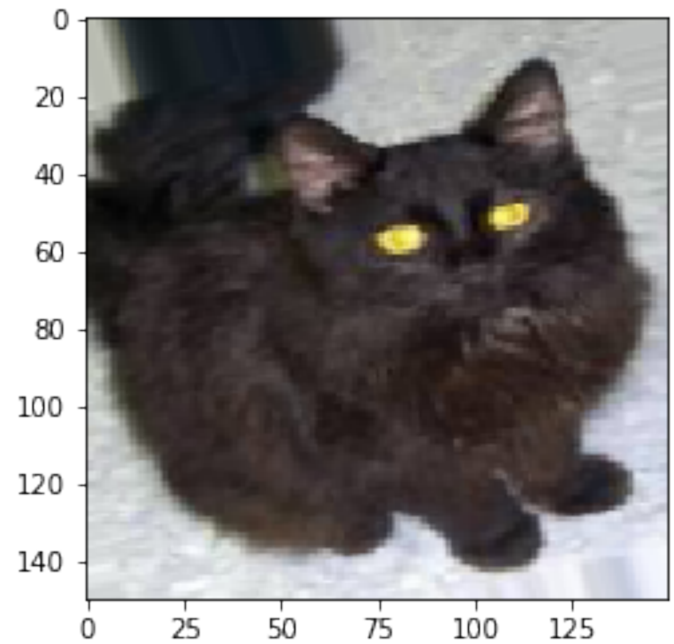
```
# This is module with image preprocessing utilities
from keras.preprocessing import image

fnames = [os.path.join(train_cats_dir, fname) for fname in os.listdir(train_cats_di

# We pick one image to "augment"
img_path = fnames[7]

# Read the image and resize it
img = image.load_img(img_path, target_size=(150, 150))

# Convert it to a Numpy array with shape (150, 150, 3)
x = image.img_to_array(img)

# Reshape it to (1, 150, 150, 3)
x = x.reshape((1,) + x.shape)

# The .flow() command below generates batches of randomly transformed images.
# It will loop indefinitely, so we need to `break` the loop at some point!
i = 0
for batch in datagen.flow(x, batch_size=1):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break

plt.show()
```











If we train a new network using this data augmentation configuration, our network will never see twice the same input. However, the inputs that it sees are still heavily intercorrelated, since they come from a small number of original images -- we cannot produce new information, we can only remix existing information. As such, this might not be quite enough to completely get rid of overfitting. To further fight overfitting, we will also add a Dropout layer to our model, right before the densely-connected classifier:

In [28]:

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```

```
WARNING:tensorflow:From /Users/jlee/anaconda3/lib/python3.7/site-packa
ges/keras/backend/tensorflow_backend.py:3445: calling dropout (from te
nsorflow.python.ops.nn_ops) with keep_prob is deprecated and will be r
emoved in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate
= 1 - keep_prob`.
```

Let's train our network using data augmentation and dropout:

In [29]:

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,)

# Note that the validation data should not be augmented!
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        # This is the target directory
        train_dir,
        # All images will be resized to 150x150
        target_size=(150, 150),
        batch_size=32,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')
```

```
validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=15,
      validation_data=validation_generator,
      validation_steps=50)
```

```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Epoch 1/15
100/100 [==============================] - 131s 1s/step - loss: 0.6941
- acc: 0.5091 - val_loss: 0.6859 - val_acc: 0.5939
Epoch 2/15
100/100 [==============================] - 128s 1s/step - loss: 0.6809
- acc: 0.5550 - val_loss: 0.6625 - val_acc: 0.6237
Epoch 3/15
100/100 [==============================] - 142s 1s/step - loss: 0.6611
- acc: 0.5981 - val_loss: 0.6324 - val_acc: 0.6332
Epoch 4/15
100/100 [==============================] - 156s 2s/step - loss: 0.6395
- acc: 0.6297 - val_loss: 0.6248 - val_acc: 0.6443
Epoch 5/15
100/100 [==============================] - 148s 1s/step - loss: 0.6242
- acc: 0.6384 - val_loss: 0.6010 - val_acc: 0.6751
Epoch 6/15
100/100 [==============================] - 133s 1s/step - loss: 0.6034
- acc: 0.6666 - val_loss: 0.6691 - val_acc: 0.6218
Epoch 7/15
100/100 [==============================] - 129s 1s/step - loss: 0.5949
- acc: 0.6756 - val_loss: 0.5974 - val_acc: 0.6662
Epoch 8/15
100/100 [==============================] - 135s 1s/step - loss: 0.5941
- acc: 0.6878 - val_loss: 0.5751 - val_acc: 0.6894
Epoch 9/15
100/100 [==============================] - 136s 1s/step - loss: 0.5802
- acc: 0.6975 - val_loss: 0.5680 - val_acc: 0.6997
Epoch 10/15
100/100 [==============================] - 129s 1s/step - loss: 0.5767
- acc: 0.6962 - val_loss: 0.5570 - val_acc: 0.6986
Epoch 11/15
100/100 [==============================] - 138s 1s/step - loss: 0.5628
- acc: 0.7187 - val_loss: 0.5882 - val_acc: 0.6798
Epoch 12/15
100/100 [==============================] - 140s 1s/step - loss: 0.5623
- acc: 0.7034 - val_loss: 0.5149 - val_acc: 0.7335
Epoch 13/15
```

```
100/100 [==============================] - 138s 1s/step - loss: 0.5530
- acc: 0.7206 - val_loss: 0.5147 - val_acc: 0.7352
Epoch 14/15
100/100 [==============================] - 136s 1s/step - loss: 0.5527
- acc: 0.7128 - val_loss: 0.5314 - val_acc: 0.7214
Epoch 15/15
100/100 [==============================] - 136s 1s/step - loss: 0.5517
- acc: 0.7116 - val_loss: 0.5488 - val_acc: 0.7030
```

Let's save our model -- we will be using it in the section on convnet visualization.

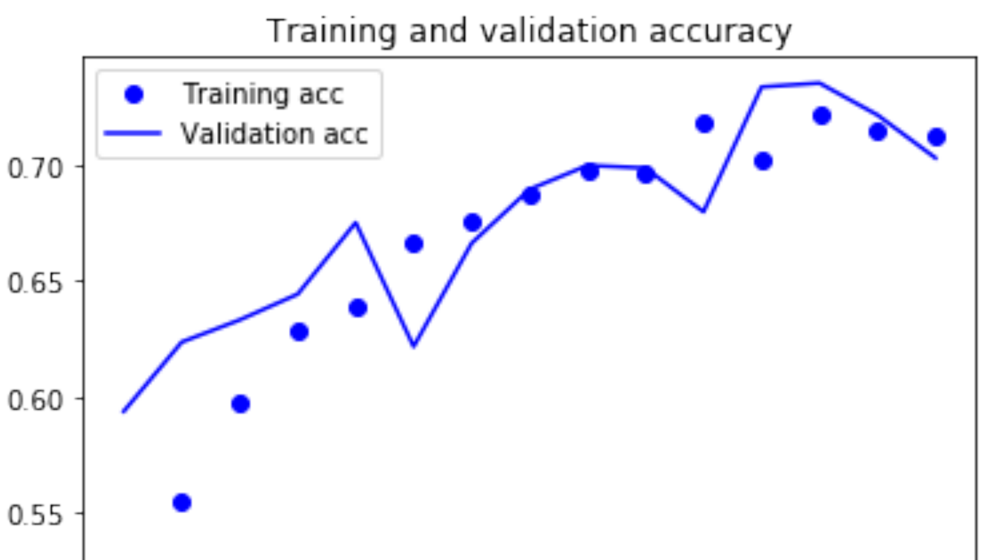
In [30]:

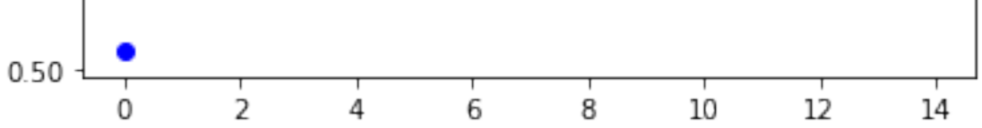
```
model.save('cats_and_dogs_small_2.h5')
```

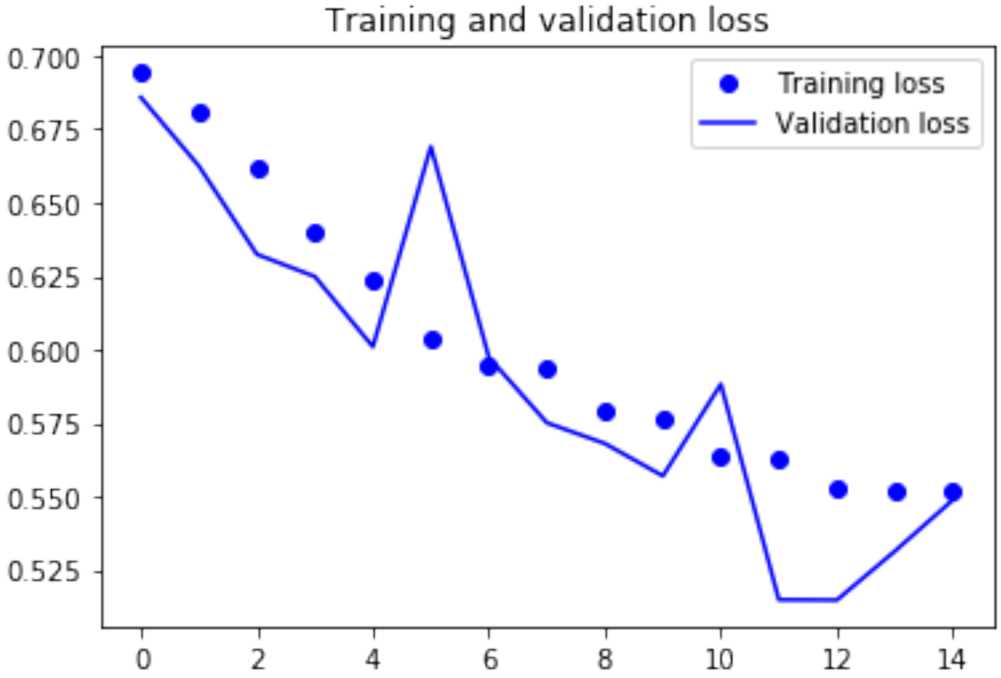
Let's plot our results again:

In [31]:

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```







Thanks to data augmentation and dropout, we are no longer overfitting: the training curves are rather closely tracking the validation curves. We are now able to reach an accuracy of 82%, a 15% relative improvement over the non-regularized model.

By leveraging regularization techniques even further and by tuning the network's parameters (such as the number of filters per convolution layer, or the number of layers in the network), we may be able to get an even better accuracy, likely up to 86-87%. However, it would prove very difficult to go any higher just by training our own convnet from scratch, simply because we have so little data to work with. As a next step to improve our accuracy on this problem, we will have to leverage a pre-trained model, which will be the focus of the next two sections.

Problem 2

Examine jupyter notebook 5.2-using-convnets-with-smalldatasets. ipynb Execute all cells including cell 11. Please explain where are the numbers of parameters 896 and 73856 on the summary display for the conv2d_1 and conv2d_3 in cell 11 (model.summary()) are coming from. Present the actual calculation. (15%)

In general, the equation for the parameters of a filter are defined by this relationship: ((kernel_size) x (# of channels) + 1) x filters)

In the first layer of the CNN the results are:

```
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896
```

Breaking it down, the paramenters in this layer stem from the how we defined the layer:
test_model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)))

The original inputs (k, k) or (3, 3) give us the number of dendrites (3 x 3 = 9) and input is 3. Therefore, 9 dendrites emanate from each of the 32 layers each of the 3 neurons with one bias parameter per filter layer.

So, the number would be calculated by: ((3 x 3) x 3 + 1) x 32 = 896

In the third layer of the CNN the results are:

```
max_pooling2d_2 (MaxPooling2) (None, 36, 36, 64)        0
conv2d_3 (Conv2D)            (None, 34, 34, 128)       73856
```

In this case, we also need to account for the Max Pooling layer:
test_model.add(layers.MaxPooling2D((2, 2)))
test_model.add(layers.Conv2D(128, (3, 3), activation='relu'))

Every one of 128 filter layers in conv2d_3 sees, convolves with every one of 64 output layers of maxpooling2d_2.

So, the number would be calculated by: ((3 x 3) x 64 + 1) x 128 = 73,856

Problem 3

Keep on executing cells of the notebook 5.2-using-convnets-with-smalldatasets. The main point of this notebook is a technique called Data Augmentation. You have small dataset but you still want to train your network. One way to do it is to fudge your samples. You transform samples (images in this case) this way and that way and you generate many more images than you originally had. And, surprisingly, the trick works.

Pickup an image of cat or a dog and demonstrate that you can generate at least 6 different, modified images of that animal applying independently: rotation, width change, high_shift, shear, zoom and horizontal flip. Execute all cells in the notebook 5.2-using-convnets-.., including the code you added. Submit a copy of the working notebook and its PDF image. (25%)

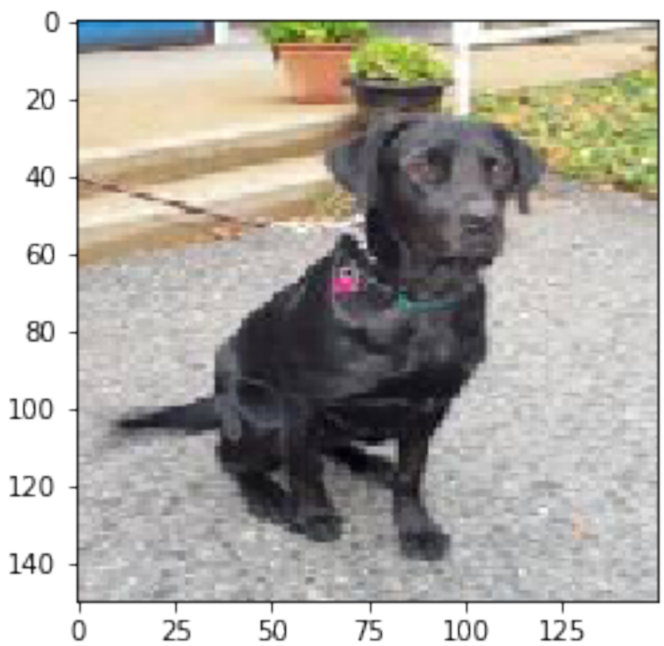
Definitions:

- `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.
- `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.

Original Image

In [99]:

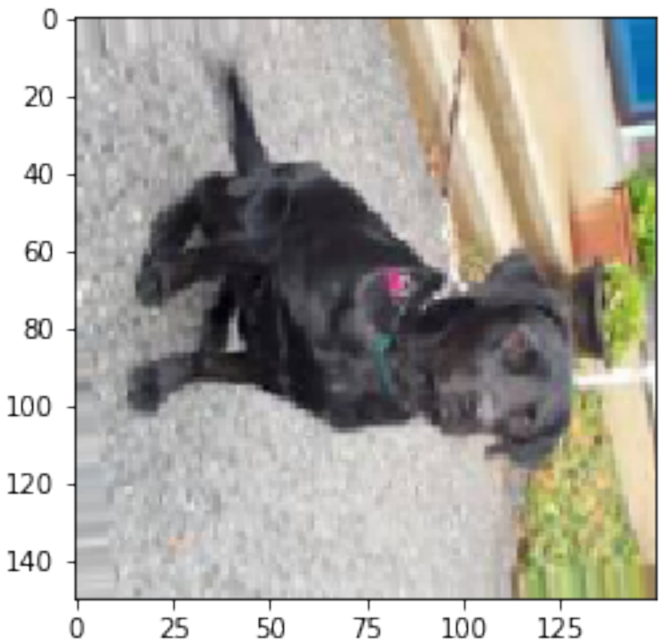
```
from keras.preprocessing import image

fnames = [os.path.join(train_dogs_dir, fname) for fname in os.listdir(train_dogs_di
img_path = fnames[12]
img = image.load_img(img_path, target_size=(150, 150))
imgplot = plt.imshow(img)
```



Rotation

In [76]:

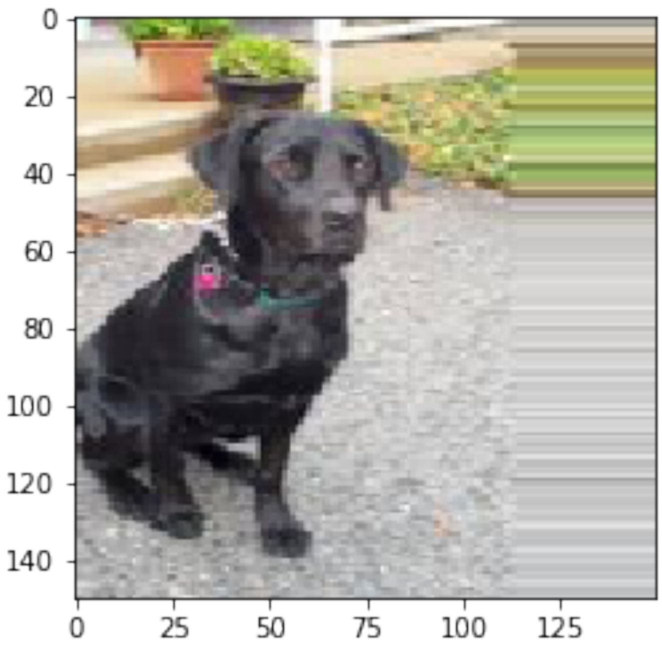
```
datagen = ImageDataGenerator(rotation_range=90)

x = image.img_to_array(img)

# Reshape it to (1, 150, 150, 3)
x = x.reshape((1,) + x.shape)

#Modified for only one image
for batch in datagen.flow(x, batch_size=1):
    plt.figure(0)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    break

plt.show()
```



Width Change

In [75]:

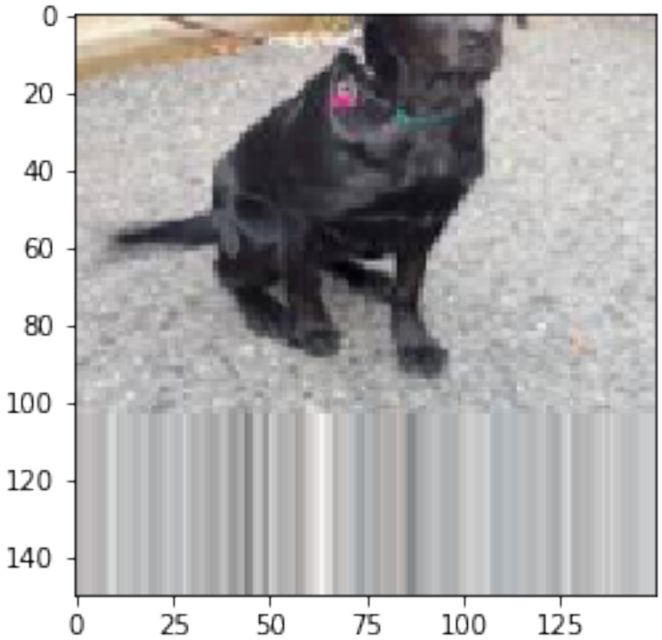
```
datagen = ImageDataGenerator(width_shift_range=0.5)

for batch in datagen.flow(x, batch_size=1):
    plt.figure(0)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    break

plt.show()
```



High Shift

In [70]:

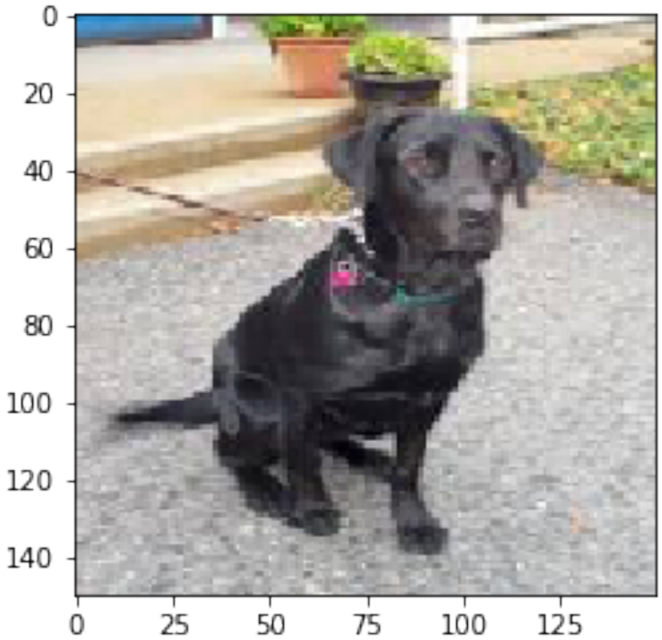
```
datagen = ImageDataGenerator(height_shift_range=0.5)

for batch in datagen.flow(x, batch_size=1):
    plt.figure(0)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    break

plt.show()
```



Shear

In [91]:

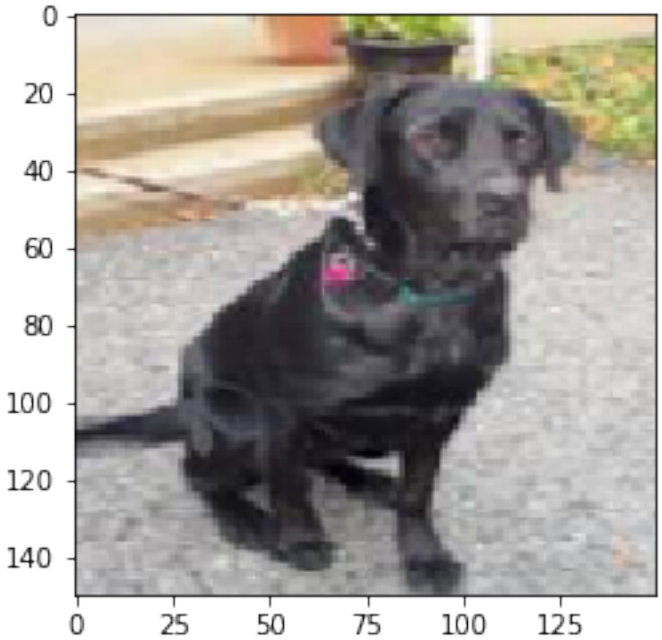
```
datagen = ImageDataGenerator(shear_range=0.5)

for batch in datagen.flow(x, batch_size=1):
    plt.figure(0)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    break

plt.show()
```



Zoom

In [86]:

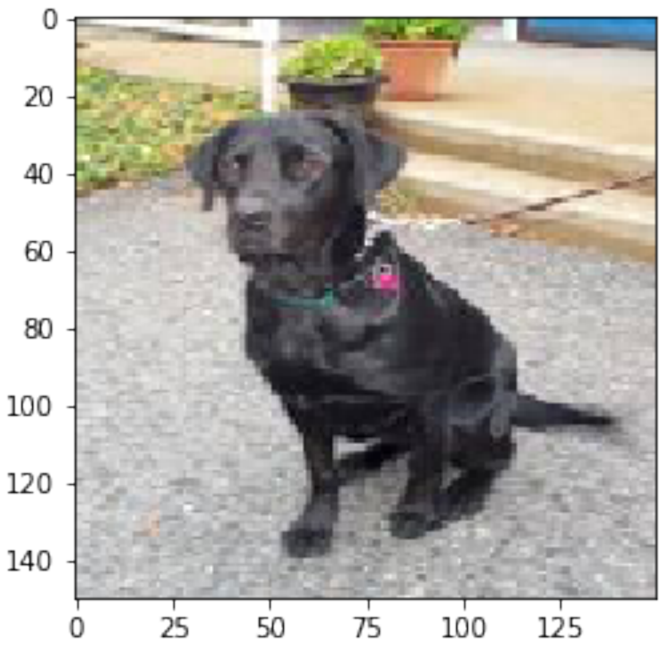
```
datagen = ImageDataGenerator(zoom_range=0.2)

for batch in datagen.flow(x, batch_size=1):
    plt.figure(0)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    break

plt.show()
```



Horizontal Flip

In [68]:

```
datagen = ImageDataGenerator(horizontal_flip=True)

for batch in datagen.flow(x, batch_size=1):
    plt.figure(0)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    break

plt.show()
```



In [1]:

```
```

Using TensorFlow backend.

Out[1]:

'2.2.4'

In [3]:

```
```

/Users/jlee/Documents/GitHub/Python-Practice/Harvard CSCI E-63/HW3

5.2 - Using convnets with small datasets

This notebook contains the code sample found in Chapter 5, Section 2 of [Deep Learning with Python (https://www.manning.com/books/deep-learning-with-python?a_aid=keras&a_bid=76564dff)](https://www.manning.com/books/deep-learning-with-python?a_aid=keras&a_bid=76564dff). Note that the original text features far more content, in particular further explanations and figures: in this notebook, you will only find source code and related comments.

Training a convnet from scratch on a small dataset

Having to train an image classification model using only very little data is a common situation, which you likely encounter yourself in practice if you ever do computer vision in a professional context.

Having "few" samples can mean anywhere from a few hundreds to a few tens of thousands of images. As a practical example, we will focus on classifying images as "dogs" or "cats", in a dataset containing 4000 pictures of cats and dogs (2000 cats, 2000 dogs). We will use 2000 pictures for training, 1000 for validation, and finally 1000 for testing.

In this section, we will review one basic strategy to tackle this problem: training a new model from scratch on what little data we have. We will start by naively training a small convnet on our 2000 training samples, without any regularization, to set a baseline for what can be achieved. This will get us to a classification accuracy of 71%. At that point, our main issue will be overfitting. Then we will introduce *data augmentation*, a powerful technique for mitigating overfitting in computer vision. By leveraging data augmentation, we will improve our network to reach an accuracy of 82%.

In the next section, we will review two more essential techniques for applying deep learning to small datasets: *doing feature extraction with a pre-trained network* (this will get us to an accuracy of 90% to 93%), and *fine-tuning a pre-trained network* (this will get us to our final accuracy of 95%). Together, these three strategies -- training a small model from scratch, doing feature extracting using a pre-trained model, and fine-tuning a pre-trained model -- will constitute your future toolbox for tackling the problem of doing computer vision with small datasets.

The relevance of deep learning for small-data problems

You will sometimes hear that deep learning only works when lots of data is available. This is in part a valid point: one fundamental characteristic of deep learning is that it is able to find interesting features in the training data on its own, without any need for manual feature engineering, and this can only be achieved when lots of training examples are available. This is especially true for problems where the input samples are very high-dimensional, like images.

However, what constitutes "lots" of samples is relative -- relative to the size and depth of the network you are trying to train, for starters. It isn't possible to train a convnet to solve a complex problem with just a few tens of samples, but a few hundreds can potentially suffice if the model is small and well-regularized and if the task is simple. Because convnets learn local, translation-invariant features, they are very data-efficient on perceptual problems. Training a convnet from scratch on a very small image dataset will still yield reasonable results despite a relative lack of data, without the need for any custom feature engineering. You will see this in action in this section.

But what's more, deep learning models are by nature highly repurposable: you can take, say, an image classification or speech-to-text model trained on a large-scale dataset then reuse it on a significantly different problem with only minor changes. Specifically, in the case of computer vision, many pre-trained models (usually trained on the ImageNet dataset) are now publicly available for download and can be used to bootstrap powerful vision models out of very little data. That's what we will do in the next section.

For now, let's get started by getting our hands on the data.

Downloading the data

The cats vs. dogs dataset that we will use isn't packaged with Keras. It was made available by Kaggle.com as part of a computer vision competition in late 2013, back when convnets weren't quite mainstream. You can download the original dataset at: `https://www.kaggle.com/c/dogs-vs-cats/data` (you will need to create a Kaggle account if you don't already have one -- don't worry, the process is painless).

The pictures are medium-resolution color JPEGs. They look like this:

Unsurprisingly, the cats vs. dogs Kaggle competition in 2013 was won by entrants who used convnets. The best entries could achieve up to 95% accuracy. In our own example, we will get fairly close to this accuracy (in the next section), even though we will be training our models on less than 10% of the data that was available to the competitors. This original dataset contains 25,000 images of dogs and cats (12,500 from each class) and is 543MB large (compressed). After downloading and uncompressing it, we will create a new dataset containing three subsets: a training set with 1000 samples of each class, a validation set with 500 samples of each class, and finally a test set with 500 samples of each class.

Here are a few lines of code to do this:

In [2]:

In [8]:

As a sanity check, let's count how many pictures we have in each training split (train/validation/test):

In [9]:

```
total training cat images: 1000
```

In [10]:

```
total training dog images: 1000
```

In [11]:

```
total validation cat images: 500
```

In [12]:

```
total validation dog images: 500
```

In [13]:

```
total test cat images: 500
```

In [14]:

```
total test dog images: 500
```

So we have indeed 2000 training images, and then 1000 validation images and 1000 test images. In each split, there is the same number of samples from each class: this is a balanced binary classification problem, which means that classification accuracy will be an appropriate measure of success.

Building our network

We've already built a small convnet for MNIST in the previous example, so you should be familiar with them. We will reuse the same general structure: our convnet will be a stack of alternated `Conv2D` (with `relu` activation) and `MaxPooling2D` layers.

However, since we are dealing with bigger images and a more complex problem, we will make our network accordingly larger: it will have one more `Conv2D + MaxPooling2D` stage. This serves both to augment the capacity of the network, and to further reduce the size of the feature maps, so that they aren't overly large when we reach the `Flatten` layer. Here, since we start from inputs of size 150x150 (a somewhat arbitrary choice), we end up with feature maps of size 7x7 right before the `Flatten` layer.

Note that the depth of the feature maps is progressively increasing in the network (from 32 to 128), while the size of the feature maps is decreasing (from 148x148 to 7x7). This is a pattern that you will see in almost all convnets.

Since we are attacking a binary classification problem, we are ending the network with a single unit (a `Dense` layer of size 1) and a `sigmoid` activation. This unit will encode the probability that the network is looking at one class or the other.

In [15]:

```
WARNING:tensorflow:From /Users/jlee/anaconda3/lib/python3.7/site-packa
ges/tensorflow/python/framework/op_def_library.py:263: colocate_with (
from tensorflow.python.framework.ops) is deprecated and will be remove
d in a future version.
Instructions for updating:
Colocations handled automatically by placer.
```

Let's take a look at how the dimensions of the feature maps change with every successive layer:

In [16]:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 74, 74, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 64)        18496     
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 36, 36, 64)        0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 34, 34, 128)       73856     
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 17, 17, 128)       0         
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 15, 15, 128)       147584    
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 128)         0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 6272)              0         
```

For our compilation step, we'll go with the `RMSprop` optimizer as usual. Since we ended our network with a single sigmoid unit, we will use binary crossentropy as our loss (as a reminder, check out the table in Chapter 4, section 5 for a cheatsheet on what loss function to use in various situations).

In [17]:

Data preprocessing

As you already know by now, data should be formatted into appropriately pre-processed floating point tensors before being fed into our network. Currently, our data sits on a drive as JPEG files, so the steps for getting it into our network are roughly:

- Read the picture files.
- Decode the JPEG content to RBG grids of pixels.
- Convert these into floating point tensors.
- Rescale the pixel values (between 0 and 255) to the [0, 1] interval (as you know, neural networks prefer to deal with small input values).

It may seem a bit daunting, but thankfully Keras has utilities to take care of these steps automatically. Keras has a module with image processing helper tools, located at `keras.preprocessing.image`. In particular, it contains the class `ImageDataGenerator` which allows to quickly set up Python generators that can automatically turn image files on disk into batches of pre-processed tensors. This is what we will use here.

In [18]:

```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
```

Let's take a look at the output of one of these generators: it yields batches of 150x150 RGB images (shape `(20, 150, 150, 3)`) and binary labels (shape `(20,)`). 20 is the number of samples in each batch (the batch size). Note that the generator yields these batches indefinitely: it just loops endlessly over the images present in the target folder. For this reason, we need to `break` the iteration loop at some point.

We need to install pillow package. Pillow is a PIL fork. PIL is Pyhton Image Library

In [19]:

```
Requirement already satisfied: pillow in /Users/jlee/anaconda3/lib/pyt
hon3.7/site-packages (5.3.0)
```

In [20]:

```
data batch shape: (20, 150, 150, 3)
labels batch shape: (20,)
```

Let's fit our model to the data using the generator. We do it using the `fit_generator` method, the equivalent of `fit` for data generators like ours. It expects as first argument a Python generator that will yield batches of inputs and targets indefinitely, like ours does. Because the data is being generated endlessly, the generator needs to know example how many samples to draw from the generator before declaring an epoch over. This is the role of the `steps_per_epoch` argument: after having drawn `steps_per_epoch` batches from the generator, i.e. after having run for `steps_per_epoch` gradient descent steps, the fitting process will go to the next epoch. In our case, batches are 20-sample large, so it will take 100 batches until we see our target of 2000 samples.

When using `fit_generator` , one may pass a `validation_data` argument, much like with the `fit` method. Importantly, this argument is allowed to be a data generator itself, but it could be a tuple of Numpy arrays as well. If you pass a generator as `validation_data` , then this generator is expected to yield batches of validation data endlessly, and thus you should also specify the `validation_steps` argument, which tells the process how many batches to draw from the validation generator for evaluation.

In [21]:

```
WARNING:tensorflow:From /Users/jlee/anaconda3/lib/python3.7/site-packa
ges/tensorflow/python/ops/math_ops.py:3066: to_int32 (from tensorflow.
python.ops.math_ops) is deprecated and will be removed in a future ver
sion.
Instructions for updating:
Use tf.cast instead.
Epoch 1/15
100/100 [==============================] - 101s 1s/step - loss: 0.6921
- acc: 0.5145 - val_loss: 0.6782 - val_acc: 0.5460
Epoch 2/15
100/100 [==============================] - 119s 1s/step - loss: 0.6610
- acc: 0.6095 - val_loss: 0.6416 - val_acc: 0.6350
Epoch 3/15
100/100 [==============================] - 97s 975ms/step - loss: 0.60
29 - acc: 0.6835 - val_loss: 0.6186 - val_acc: 0.6400
Epoch 4/15
100/100 [==============================] - 93s 928ms/step - loss: 0.56
94 - acc: 0.6935 - val_loss: 0.6107 - val_acc: 0.6680
Epoch 5/15
```

It is good practice to always save your models after training:

In [22]:

```
Requirement already satisfied: h5py in /Users/jlee/anaconda3/lib/pytho
n3.7/site-packages (2.8.0)
Requirement already satisfied: numpy>=1.7 in /Users/jlee/anaconda3/lib
/python3.7/site-packages (from h5py) (1.16.2)
Requirement already satisfied: six in /Users/jlee/anaconda3/lib/python
3.7/site-packages (from h5py) (1.12.0)
```

In [23]:

Let's plot the loss and accuracy of the model over the training and validation data during training:

In [24]:

```
```

```
<Figure size 640x480 with 1 Axes>
```

```
<Figure size 640x480 with 1 Axes>
```

In [25]:

```
```

These plots are characteristic of overfitting. Our training accuracy increases linearly over time, until it reaches nearly 100%, while our validation accuracy stalls at 70-72%. Our validation loss reaches its minimum after only five epochs then stalls, while the training loss keeps decreasing linearly until it reaches nearly 0.

Because we only have relatively few training samples (2000), overfitting is going to be our number one concern. You already know about a number of techniques that can help mitigate overfitting, such as dropout and weight decay (L2 regularization). We are now going to introduce a new one, specific to computer vision, and used almost universally when processing images with deep learning models: *data augmentation*.

Using data augmentation

Overfitting is caused by having too few samples to learn from, rendering us unable to train a model able to generalize to new data. Given infinite data, our model would be exposed to every possible aspect of the data distribution at hand: we would never overfit. Data augmentation takes the approach of generating more training data from existing training samples, by "augmenting" the samples via a number of random transformations that yield believable-looking images. The goal is that at training time, our model would never see the exact same picture twice. This helps the model get exposed to more aspects of the data and generalize better.

In Keras, this can be done by configuring a number of random transformations to be performed on the images read by our `ImageDataGenerator` instance. Let's get started with an example:

In [26]:

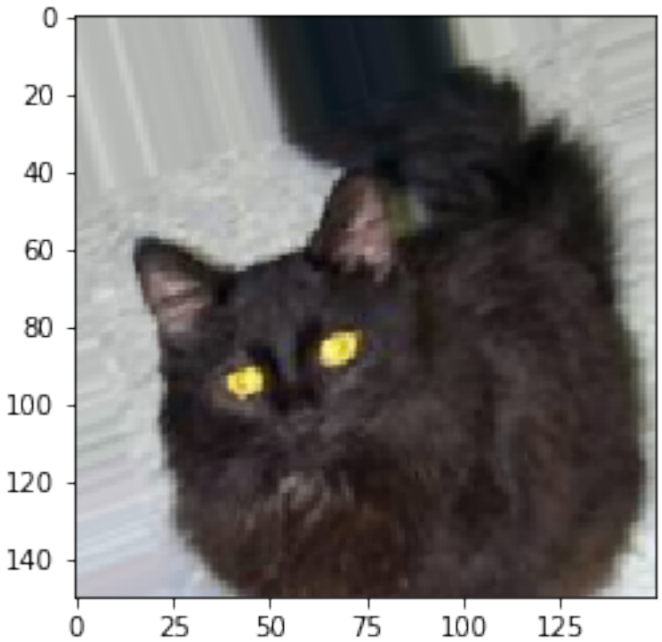
```
```

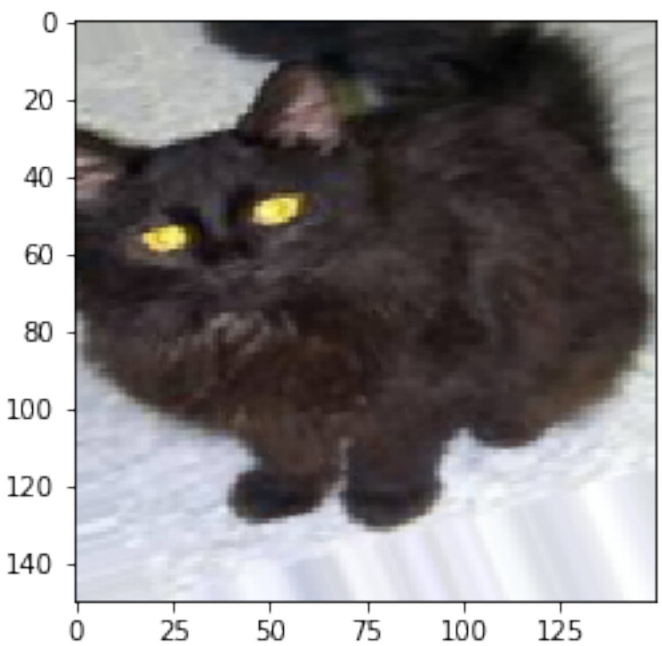
These are just a few of the options available (for more, see the Keras documentation). Let's quickly go over what we just wrote:

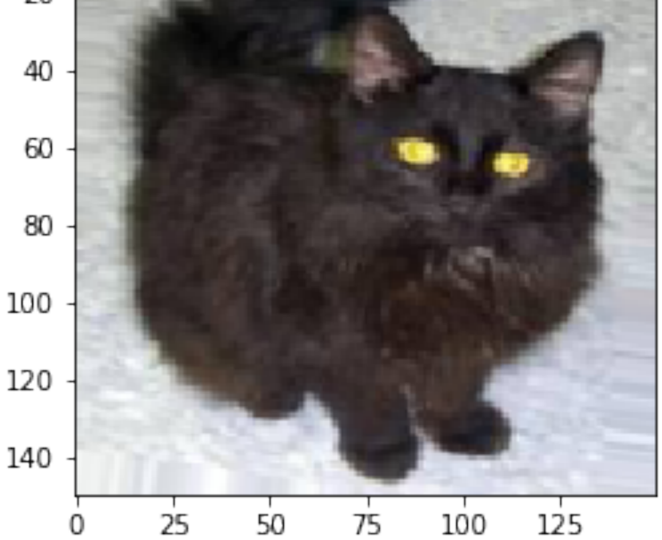
- `rotation_range` is a value in degrees (0-180), a range within which to randomly rotate pictures.
- `width_shift` and `height_shift` are ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally.
- `shear_range` is for randomly applying shearing 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.

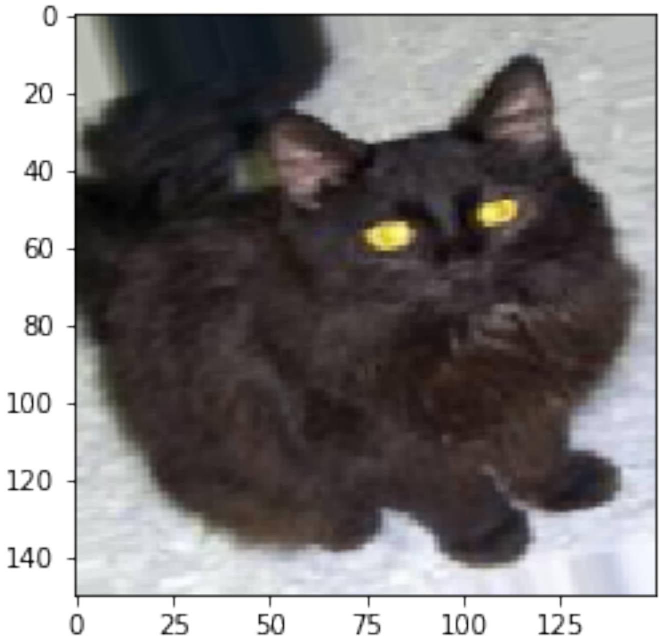
Let's take a look at our augmented images:

In [27]:









If we train a new network using this data augmentation configuration, our network will never see twice the same input. However, the inputs that it sees are still heavily intercorrelated, since they come from a small number of original images -- we cannot produce new information, we can only remix existing information. As such, this might not be quite enough to completely get rid of overfitting. To further fight overfitting, we will also add a Dropout layer to our model, right before the densely-connected classifier:

In [28]:

```
WARNING:tensorflow:From /Users/jlee/anaconda3/lib/python3.7/site-packa
ges/keras/backend/tensorflow_backend.py:3445: calling dropout (from te
nsorflow.python.ops.nn_ops) with keep_prob is deprecated and will be r
emoved in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate
= 1 - keep_prob`.
```

Let's train our network using data augmentation and dropout:

In [29]:

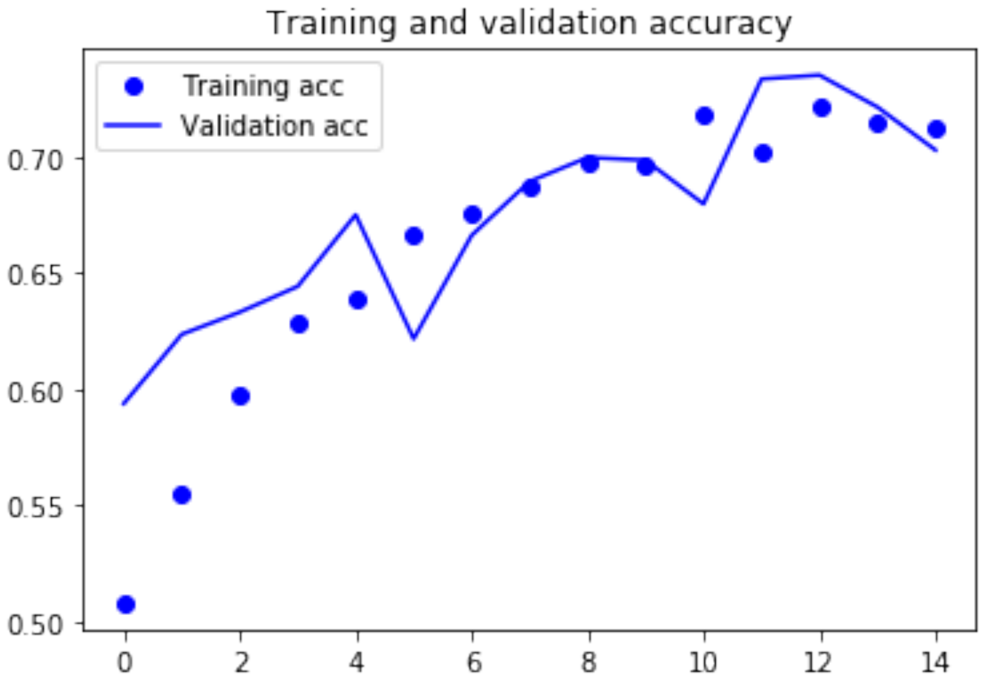
```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Epoch 1/15
100/100 [==============================] - 131s 1s/step - loss: 0.6941
- acc: 0.5091 - val_loss: 0.6859 - val_acc: 0.5939
Epoch 2/15
100/100 [==============================] - 128s 1s/step - loss: 0.6809
- acc: 0.5550 - val_loss: 0.6625 - val_acc: 0.6237
Epoch 3/15
100/100 [==============================] - 142s 1s/step - loss: 0.6611
- acc: 0.5981 - val_loss: 0.6324 - val_acc: 0.6332
Epoch 4/15
100/100 [==============================] - 156s 2s/step - loss: 0.6395
- acc: 0.6297 - val_loss: 0.6248 - val_acc: 0.6443
Epoch 5/15
100/100 [==============================] - 148s 1s/step - loss: 0.6242
- acc: 0.6384 - val_loss: 0.6010 - val_acc: 0.6751
Epoch 6/15
100/100 [==============================] - 133s 1s/step - loss: 0.6034
```

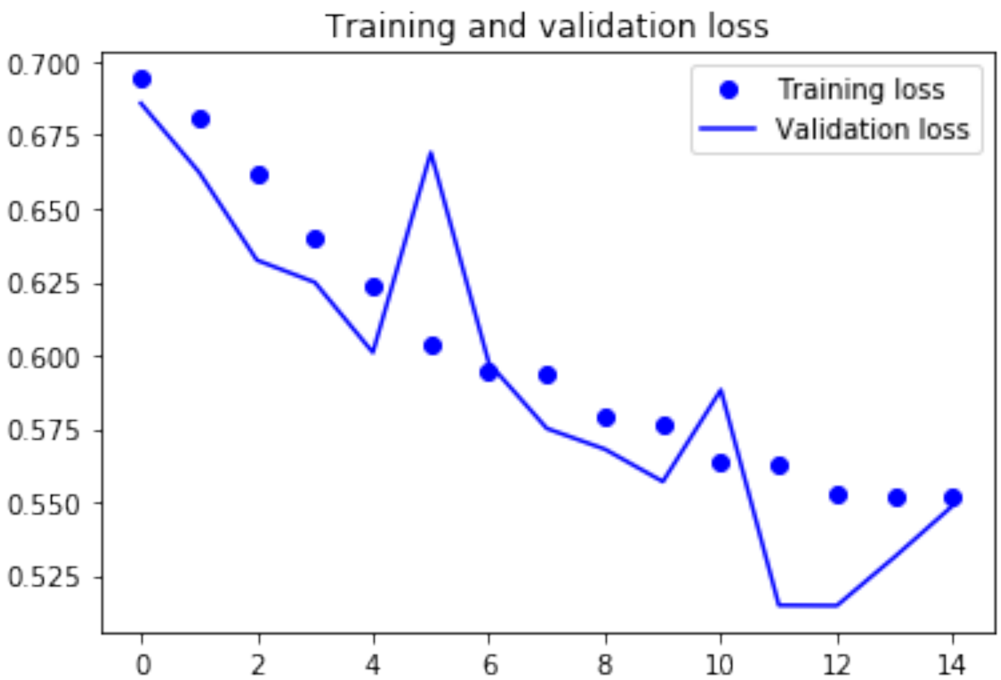
Let's save our model -- we will be using it in the section on convnet visualization.

In [30]:

Let's plot our results again:

In [31]:





Thanks to data augmentation and dropout, we are no longer overfitting: the training curves are rather closely tracking the validation curves. We are now able to reach an accuracy of 82%, a 15% relative improvement over the non-regularized model.

By leveraging regularization techniques even further and by tuning the network's parameters (such as the number of filters per convolution layer, or the number of layers in the network), we may be able to get an even better accuracy, likely up to 86-87%. However, it would prove very difficult to go any higher just by training our own convnet from scratch, simply because we have so little data to work with. As a next step to improve our accuracy on this problem, we will have to leverage a pre-trained model, which will be the focus of the next two sections.

Problem 2

Examine jupyter notebook 5.2-using-convnets-with-smalldatasets. ipynb Execute all cells including cell 11. Please explain where are the numbers of parameters 896 and 73856 on the summary display for the conv2d_1 and conv2d_3 in cell 11 (model.summary()) are coming from. Present the actual calculation. (15%)

In general, the equation for the parameters of a filter are defined by this relationship: ((kernel_size) x (# of channels) + 1) x filters)

In the first layer of the CNN the results are:

```
conv2d_1 (Conv2D)             (None, 148, 148, 32)      896
```

Breaking it down, the paramenters in this layer stem from the how we defined the layer:
test_model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)))

The original inputs (k, k) or (3, 3) give us the number of dendrites (3 x 3 = 9) and input is 3. Therefore, 9 dendrites emanate from each of the 32 layers each of the 3 neurons with one bias parameter per filter layer.

So, the number would be calculated by: ((3 x 3) x 3 + 1) x 32 = 896

In the third layer of the CNN the results are:

```
max_pooling2d_2 (MaxPooling2) (None, 36, 36, 64)        0
conv2d_3 (Conv2D)             (None, 34, 34, 128)       73856
```

In this case, we also need to account for the Max Pooling layer:
test_model.add(layers.MaxPooling2D((2, 2)))
test_model.add(layers.Conv2D(128, (3, 3), activation='relu'))

Every one of 128 filter layers in conv2d_3 sees, convolves with every one of 64 output layers of maxpooling2d_2.

So, the number would be calculated by: ((3 x 3) x 64 + 1) x 128 = 73,856

Problem 3

Keep on executing cells of the notebook 5.2-using-convnets-with-smalldatasets. The main point of this notebook is a technique called Data Augmentation. You have small dataset but you still want to train your network. One way to do it is to fudge your samples. You transform samples (images in this case) this way and that way and you generate many more images than you originally had. And, surprisingly, the trick works.

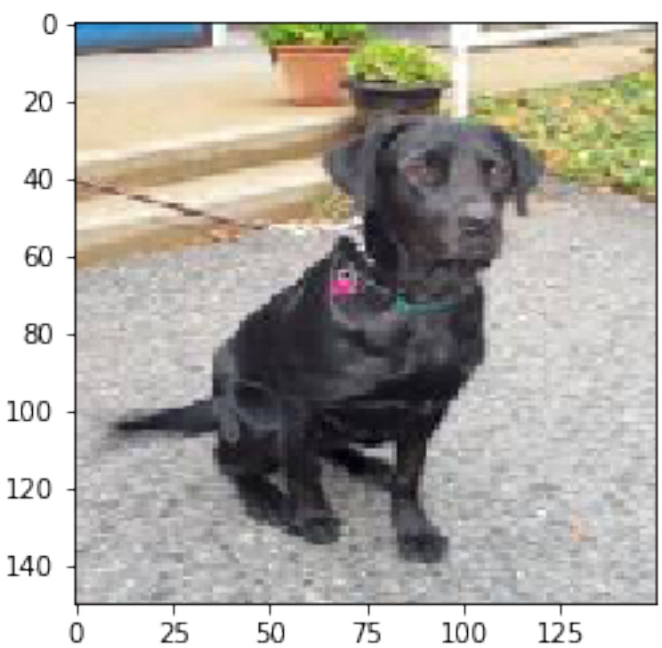
Pickup an image of cat or a dog and demonstrate that you can generate at least 6 different, modified images of that animal applying independently: rotation, width change, high_shift, shear, zoom and horizontal flip. Execute all cells in the notebook 5.2-using-convnets-.., including the code you added. Submit a copy of the working notebook and its PDF image. (25%)

Definitions:

- `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.
- `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.

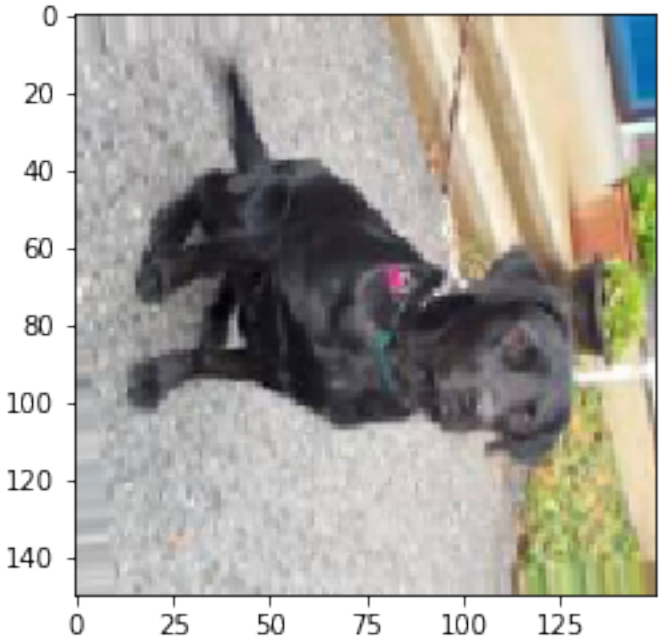
Original Image

In [99]:



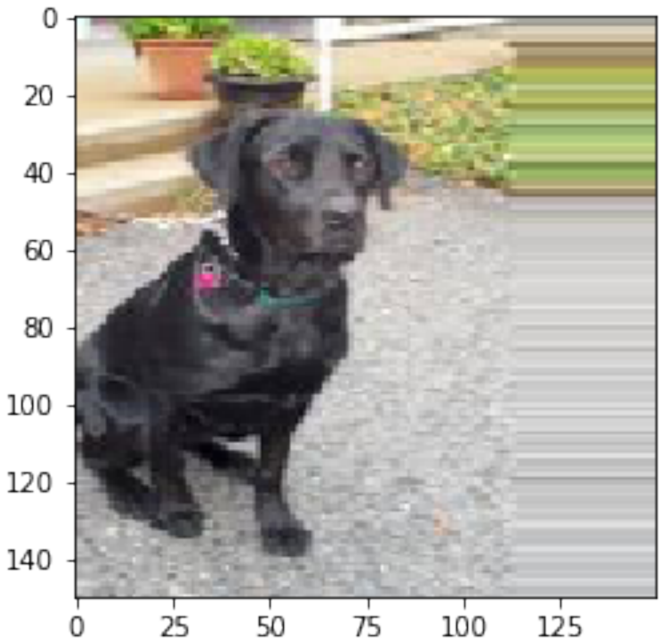
Rotation

In [76]:



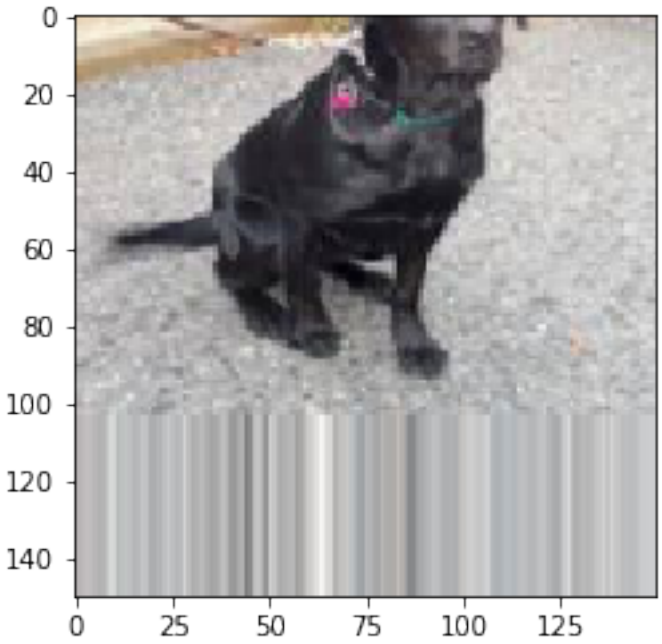
Width Change

In [75]:



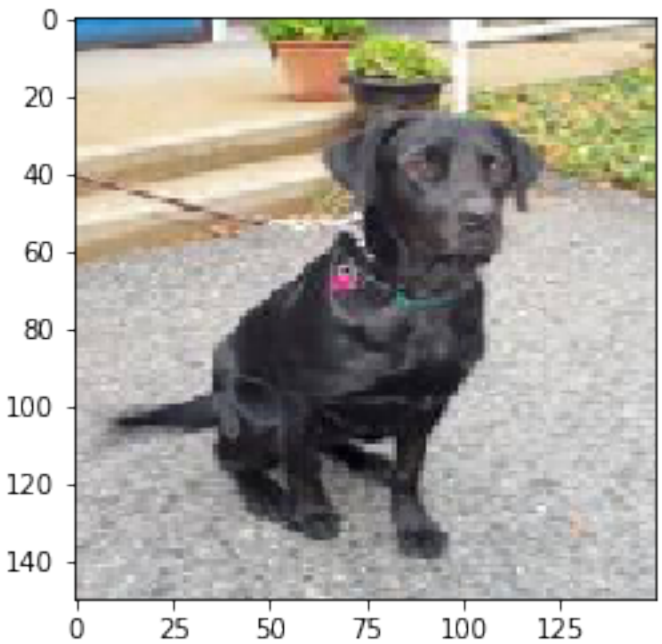
High Shift

In [70]:



Shear

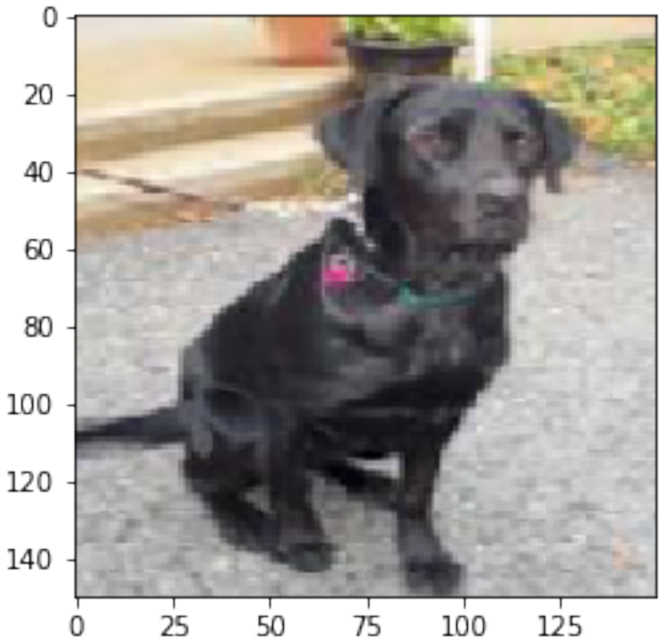
In [91]:



Zoom

In [86]:

```

```



Horizontal Flip

In [68]:

```

```