Deep Learning for Computer Vision: Assignment 5

Computer Science: COMS W 4995 006

Due: April 3, 2018

Problem: Telling Cats from Dogs using VGG16

This assignment is based on the blog post "Building powerful image classification models using very little data" from blog.keras.io. Here you will build a classifier that can distinguish between pictures of dogs and cats. You will use a ConvNet (VGG16) that was pre-trained ImageNet. Your task will be to re-architect the network to solve your problem. To do this you will:

1. Make a training dataset, using images from the link below, with 10,000 images of cats and 10,000 images of dogs. Use 1,000 images of each category for your validation set. The data should be orgainized into folders named ./data/train/cats/ + ./data/train/dogs/ + ./data/validation/cats/ + ./data/validation/dogs/. (No need to worry about a test set for this assignment.)
2. take VGG16 network architecture
3. load in the pre-trained weights from the link below for all layers except the last layers
4. add a fully connected layer followed by a final sigmoid layer to replace the 1000 category softmax layer that was used when the network was trained on ImageNet
5. freeze all layers except the last two that you added
6. fine-tune the network on your cats vs. dogs image data
7. evaluate the accuracy
8. unfreeze all layers
9. continue fine-tuning the network on your cats vs. dogs image data
10. evaluate the accuracy
11. comment your code and make sure to include accuracy, a few sample mistakes, and anything else you would like to add

Downloads:

1. You can get your image data from: https://www.kaggle.com/c/dogs-vs-cats/data (https://www.kaggle.com/c/dogs-vs-cats/data).
2. You can get your VGG16 pre-trained network weights by googling "vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5"

(Note this assignment deviates from blog.keras.io in that it uses more data AND performs the fine-tuning in two steps: first freezing the lower layers and then un-freezing them for a final run of fine-tuning. The resulting ConvNet gets more than 97% accuracy in telling pictures of cats and dogs apart.)

A bunch of code and network definition has been included to to get you started. This is not meant to be a difficult assignment, as you have your final projects to work on! Good luck and have fun!

Here we import necessary libraries.

In [1]:
```
import os
import h5py

import matplotlib.pyplot as plt
import time, pickle, pandas

import numpy as np

from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, load_model
from keras.layers import Convolution2D, MaxPooling2D, ZeroPadding2D, Conv2D
from keras.layers import Activation, Dropout, Flatten, Dense
from keras.callbacks import TensorBoard, ModelCheckpoint
from keras import backend
from keras import optimizers

%matplotlib inline
```

```
Using TensorFlow backend.
```

We only have two classes.

In [2]:
```
nb_classes = 2
class_name = {
    0: 'cat',
    1: 'dog',
}
```

This let's us plot samples.

In [3]:
```
def show_sample(X, y, prediction=-1):
    im = X
    plt.imshow(im)
    if prediction >= 0:
        plt.title("Class = %s, Predict = %s" % (class_name[y], class_name[prediction]))
    else:
        plt.title("Class = %s" % (class_name[y]))

    plt.axis('on')
    plt.show()
```

Here we define where the data comes from and how much we have.

In [4]:
```
# dimensions of our images.
img_width, img_height = 150, 150

train_data_dir = './imagenet_vgg16_fine-tuning/data/train'
validation_data_dir = './imagenet_vgg16_fine-tuning/data/validation'
nb_train_samples = 20000
nb_validation_samples = 2000
batch_size = 32
steps_per_epoch_train = nb_train_samples / batch_size
steps_per_epoch_val = nb_validation_samples / batch_size
```

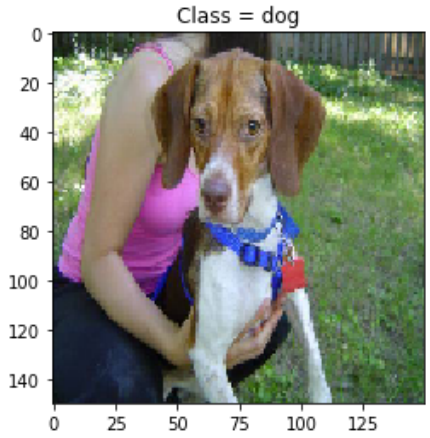
Keras training requires specifying how the data is going to be streamed to to the "fitting" routine. Here the data is augmented with "new" examples which are sheared, zoomed, and flipped versions of the originals.

In [6]:
```
# 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=32,
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_data_dir,
        target_size=(img_width, img_height),
        batch_size=32,
        class_mode='binary')
```

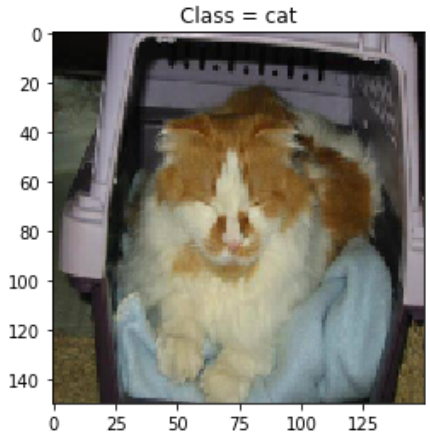
```
Found 20000 images belonging to 2 classes.
Found 2000 images belonging to 2 classes.
```

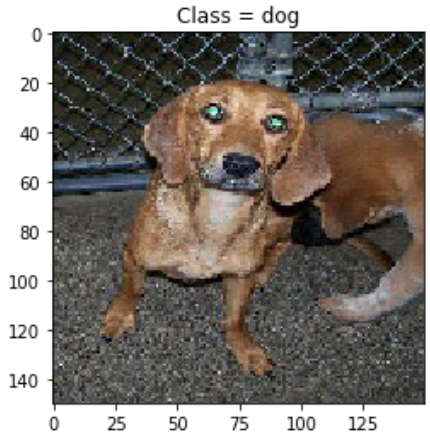
Let's display some sample images from this dataset. Note that these are real images and this is not an easy problem. But also note that it is made simpler by the fact that the animals, for the most part, are relatively large and centered in the photos.

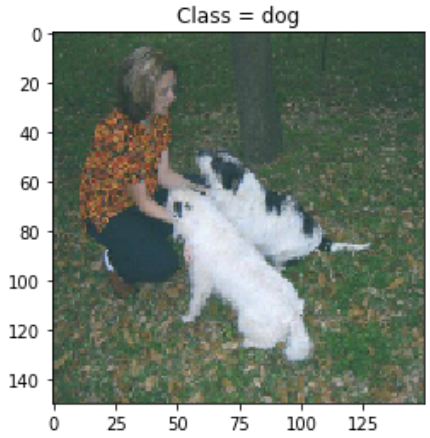
In [7]:
```
for X_batch, Y_batch in validation_generator:
    for i in range(len(Y_batch)):
        show_sample(X_batch[i, :, :, :], Y_batch[i])
    break
```

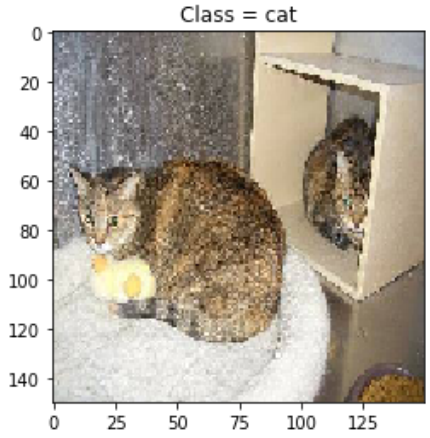

Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

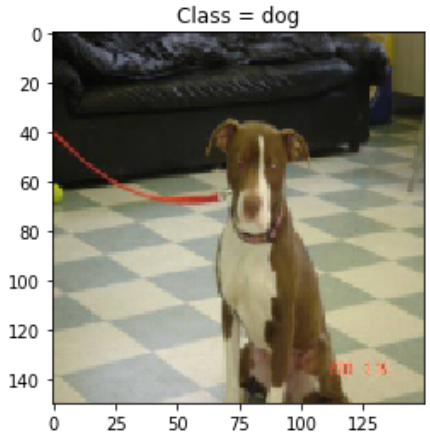

Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

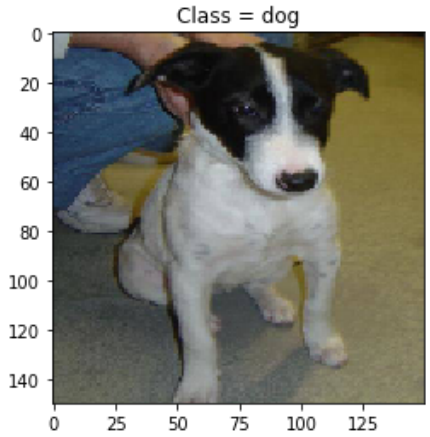

Class = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

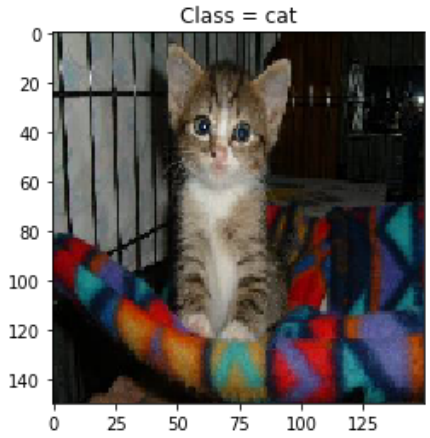

Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

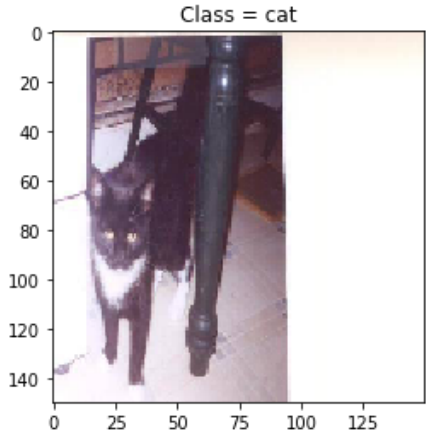

Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

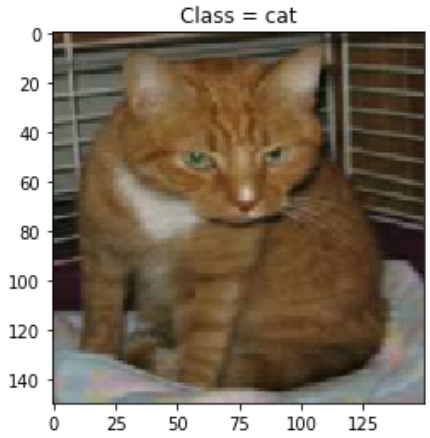

Class = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

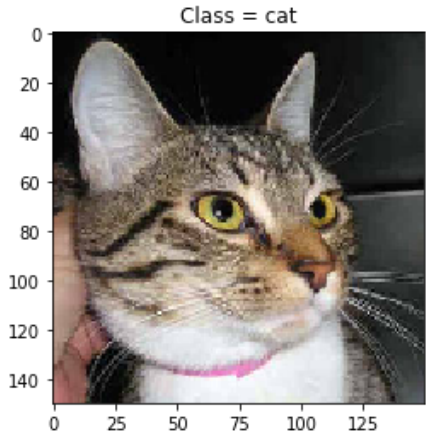

Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

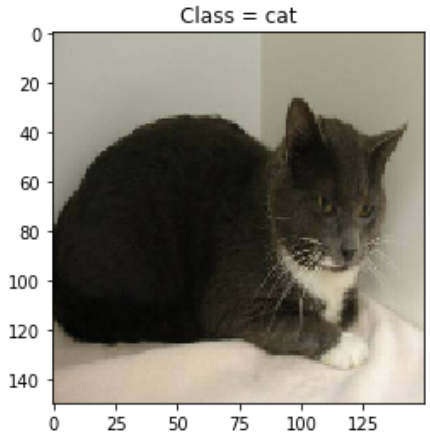

Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

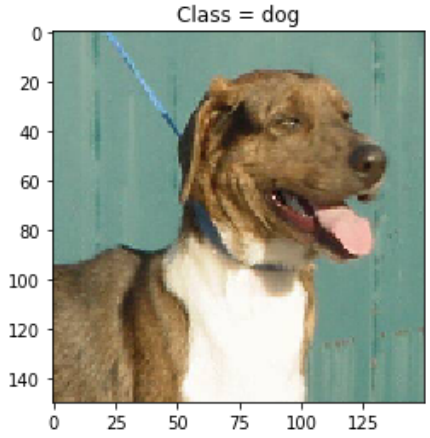

Class = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

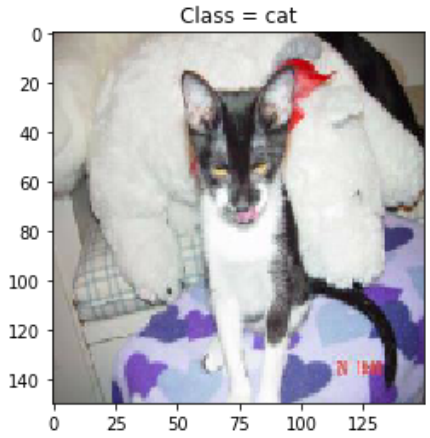

Class = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

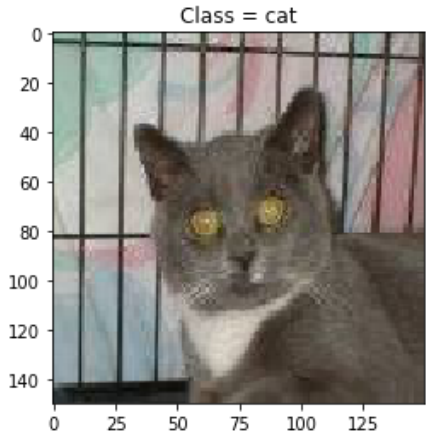

Class = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

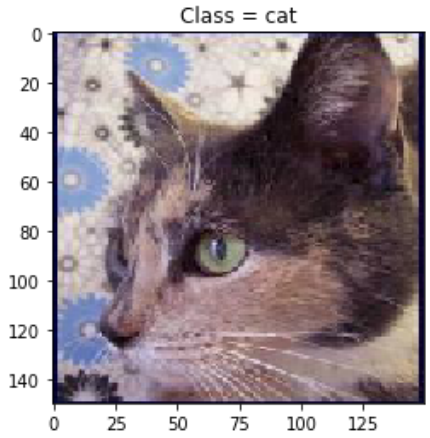

Class = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

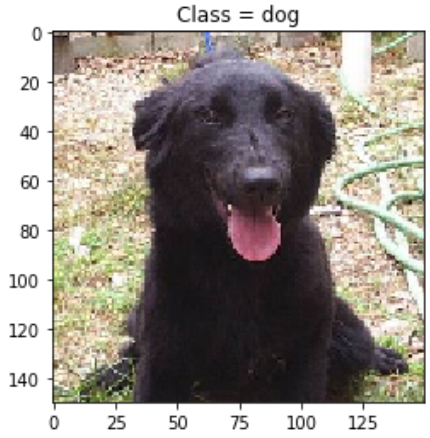

Class = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

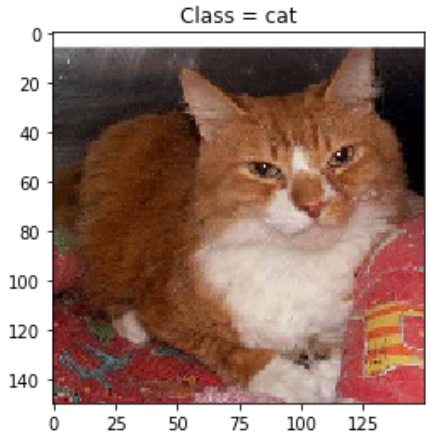

Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

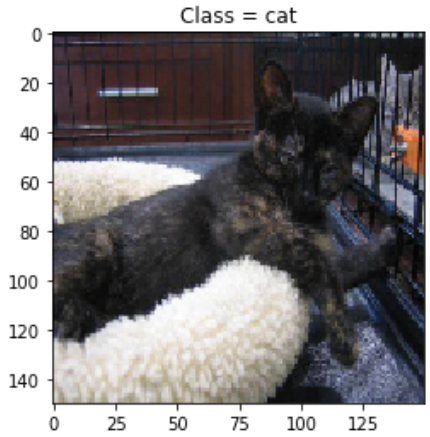

Class = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

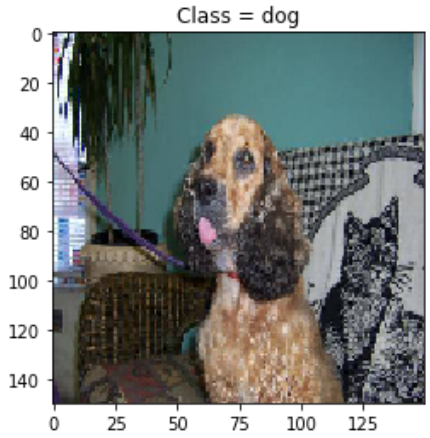

Class = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

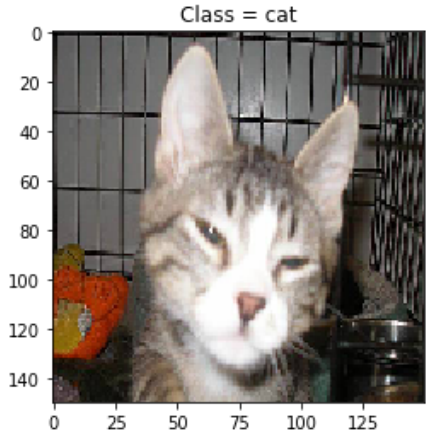

Class = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

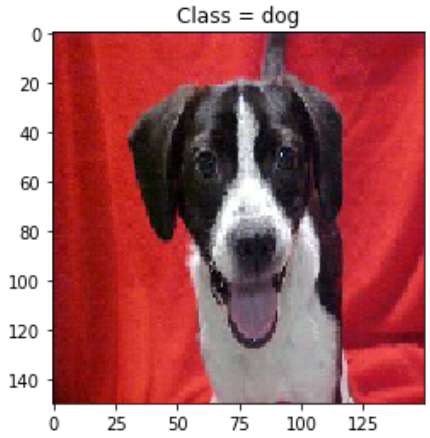

Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

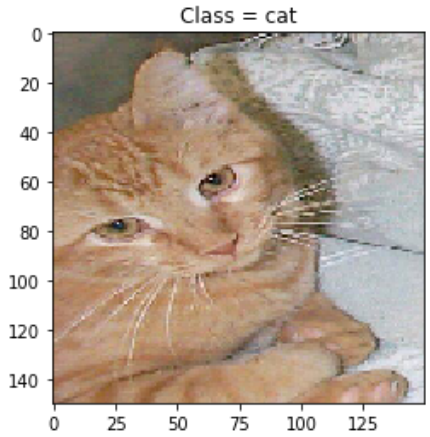

Class = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

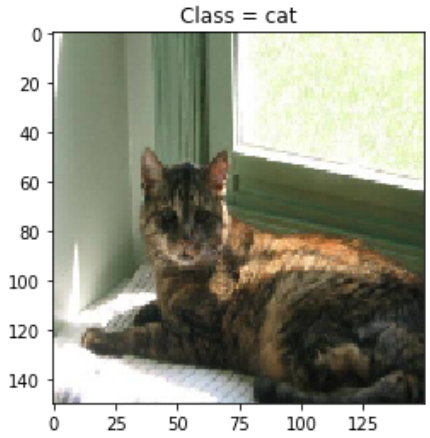

Class = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

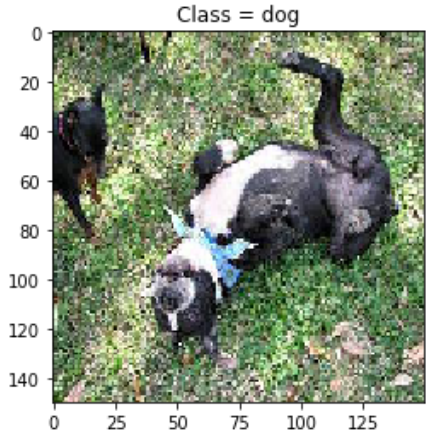

Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

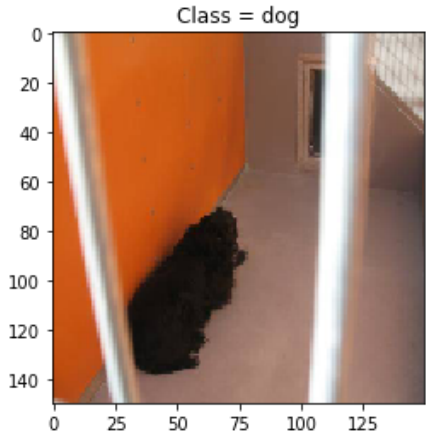

Class = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

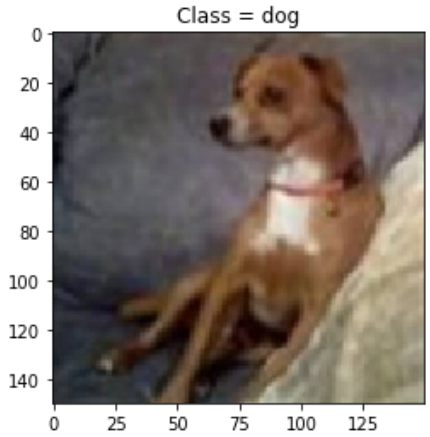

Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

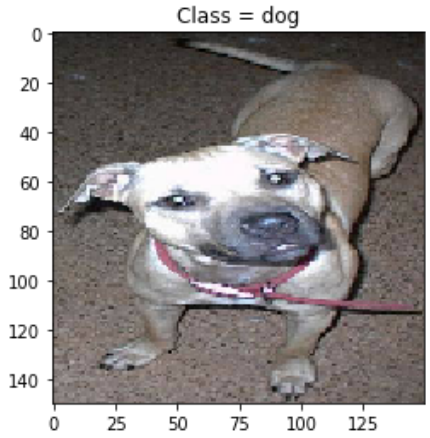

Class = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

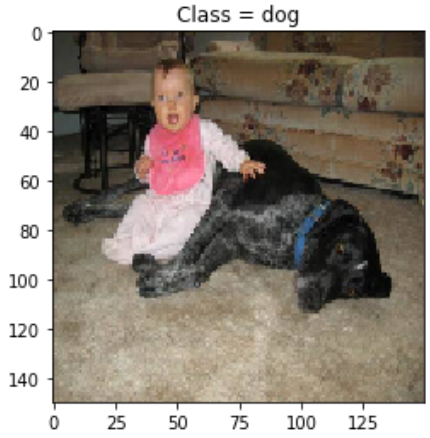

Class = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

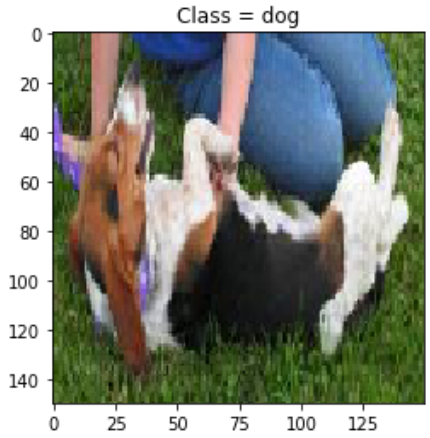

Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

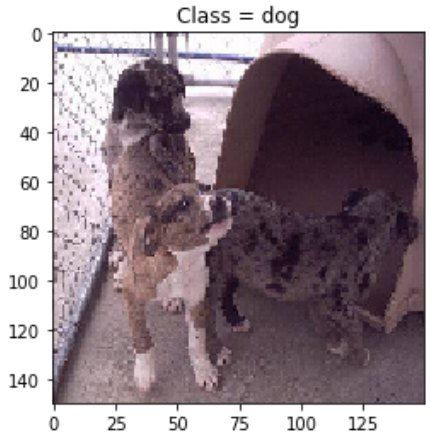

Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125


Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125


Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125


Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125


Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125


Class = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

Here is our first model. It is a small ConvNet which does surprisingly well at the dog vs. cat problem. But we will be able to do much better later on. This network is not particularly deep, just three convolutional layers each with max pooling and a final fully connected layer.

In [8]:
```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(img_width, img_height, 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(63, (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'))
```

```
WARNING:tensorflow:From /home/parita/anaconda3/envs/tensorflow/lib/python3.6/site-packa
ges/keras/backend/tensorflow_backend.py:1259: calling reduce_prod (from tensorflow.pyth
on.ops.math_ops) with keep_dims is deprecated and will be removed in a future version.
Instructions for updating:
keep_dims is deprecated, use keepdims instead
```

Now we compile the network using cross entropy loss on our logistic sigmoid and print out network layers.

In [48]:
```
model.compile(loss = 'binary_crossentropy',
              optimizer = 'rmsprop',
              metrics=['accuracy'])

print(model.summary())
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
activation_1 (Activation)    (None, 148, 148, 32)      0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 32)        9248
_________________________________________________________________
activation_2 (Activation)    (None, 72, 72, 32)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 36, 36, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 34, 34, 63)        18207
_________________________________________________________________
activation_3 (Activation)    (None, 34, 34, 63)        0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 17, 17, 63)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 18207)             0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                1165312
_________________________________________________________________
activation_4 (Activation)    (None, 64)                0
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 65
_________________________________________________________________
activation_5 (Activation)    (None, 1)                 0
=================================================================
Total params: 1,193,728
Trainable params: 1,193,728
Non-trainable params: 0
_________________________________________________________________
None
```

I was havinng permission troubles with creating dirs and writing files, so I jumped out to the shell to make these manually. These dirs will hold the training history, logging, and models created during training.

In [10]:
```
%pushd
%mkdir -p history
%mkdir -p models
%mkdir -p logs
%cd logs
%mkdir -p ./little_convnet
%mkdir -p ./vgg16_fine_tuning
%popd
```

```
/home/parita/4995-06-deep-learning-for-computer-vision/hw5/logs
/home/parita/4995-06-deep-learning-for-computer-vision/hw5
popd -> ~/4995-06-deep-learning-for-computer-vision/hw5
```

Here we design the call backs that will record stuff while training. The first will create the log files that can be viewed using the command line tool Tensorboard. The second saves the "best model" so far, where best is based on the validation accuracy.

In [49]:
```
tensorboard_callback = TensorBoard(log_dir='./logs/little_convnet/', histogram_freq=0, w
rite_graph=True, write_images=False)
checkpoint_callback = ModelCheckpoint('./models/little_convnet_weights.{epoch:02d}-{val_
acc:.2f}.hdf5', monitor='val_acc', verbose=0, save_best_only=True, save_weights_only=Fal
se, mode='auto', period=1)
```

Now we are on to training the network. Note the use of the callbacks. Also note that the learning schedule was set back when we compiled the network.

In [50]:
```
nb_epoch = 10

hist_little_convet = model.fit_generator(train_generator,
              initial_epoch=0,
              verbose=1,
              validation_data=validation_generator,
              steps_per_epoch=steps_per_epoch_train,
              epochs=nb_epoch,
              callbacks=[tensorboard_callback, checkpoint_callback],
              validation_steps=steps_per_epoch_val)


pandas.DataFrame(hist_little_convet.history).to_csv("./history/little_convet.csv")
```

```
Epoch 1/10
625/625 [==============================] - 124s 199ms/step - loss: 0.5478 - acc: 0.7266
- val_loss: 0.5556 - val_acc: 0.7125
Epoch 2/10
625/625 [==============================] - 122s 195ms/step - loss: 0.4999 - acc: 0.7627
- val_loss: 0.4006 - val_acc: 0.8165
Epoch 3/10
625/625 [==============================] - 121s 194ms/step - loss: 0.4809 - acc: 0.7799
- val_loss: 0.3956 - val_acc: 0.8185
Epoch 4/10
625/625 [==============================] - 122s 195ms/step - loss: 0.4577 - acc: 0.7940
- val_loss: 0.3671 - val_acc: 0.8420
Epoch 5/10
625/625 [==============================] - 121s 194ms/step - loss: 0.4379 - acc: 0.8040
- val_loss: 0.3542 - val_acc: 0.8460
Epoch 6/10
625/625 [==============================] - 122s 195ms/step - loss: 0.4215 - acc: 0.8139
- val_loss: 0.5542 - val_acc: 0.7765
Epoch 7/10
625/625 [==============================] - 121s 194ms/step - loss: 0.4161 - acc: 0.8220
- val_loss: 0.3404 - val_acc: 0.8540
Epoch 8/10
625/625 [==============================] - 122s 194ms/step - loss: 0.4061 - acc: 0.8256
- val_loss: 0.3242 - val_acc: 0.8580
Epoch 9/10
625/625 [==============================] - 121s 193ms/step - loss: 0.3895 - acc: 0.8306
- val_loss: 0.3163 - val_acc: 0.8755
Epoch 10/10
625/625 [==============================] - 121s 194ms/step - loss: 0.3838 - acc: 0.8359
- val_loss: 0.6005 - val_acc: 0.7930
```

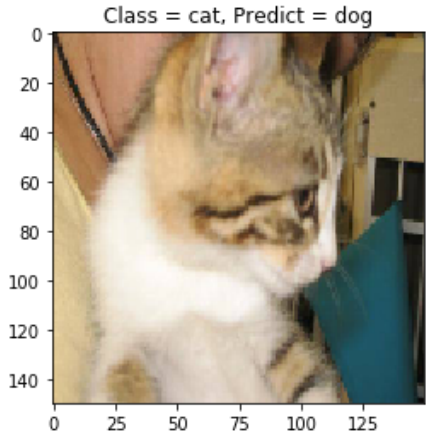
Below we grab some validation batches and evaluate our accuracy. We achieve 80% with this little network, which ain't bad, but we can do MUCH better.

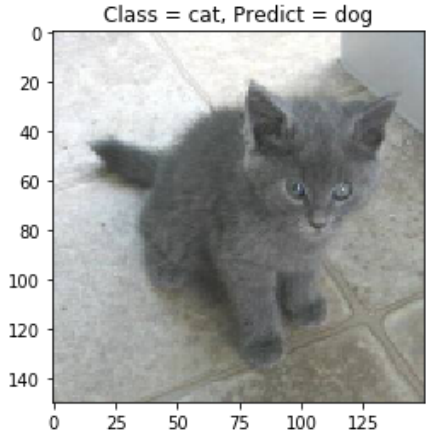
In [51]:
```
accuracies = np.array([])
losses = np.array([])

i=0
for X_batch, Y_batch in validation_generator:
    loss, accuracy = model.evaluate(X_batch, Y_batch, verbose=0)
    losses = np.append(losses, loss)
    accuracies = np.append(accuracies, accuracy)
    i += 1
    if i == 20:
       break

print("Validation: accuracy = %f  ;  loss = %f" % (np.mean(accuracies), np.mean(losses
)))
```

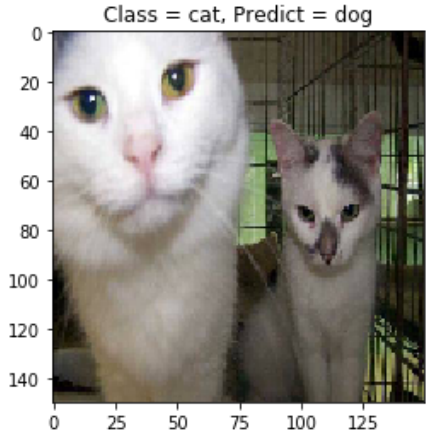
```
Validation: accuracy = 0.803125  ;  loss = 0.639131
```

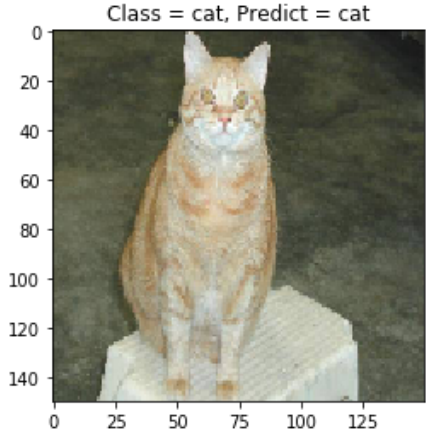
In [52]:
```
X_test, y_test = next(validation_generator)
predictions = model.predict_classes(X_test, batch_size=32, verbose=0)
```

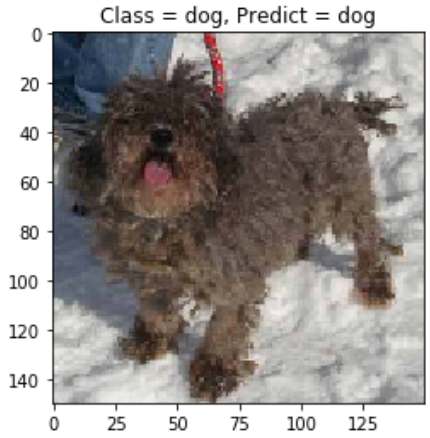
In [53]:
```
for i in range(15):
    show_sample(X_test[i, :, :, :], y_test[i], prediction=predictions[i, 0])
```

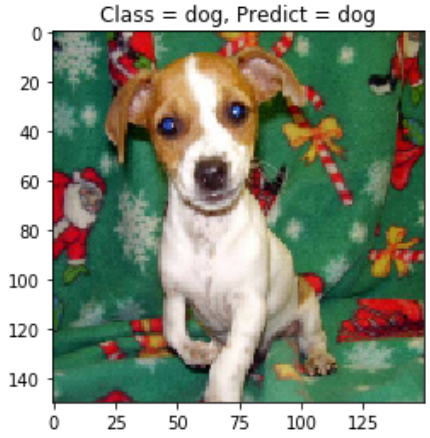

Class = cat, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

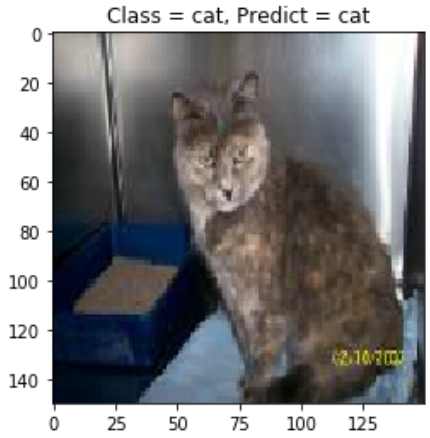

Class = cat, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

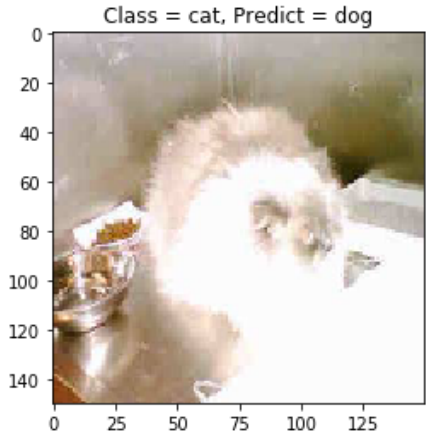

Class = cat, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

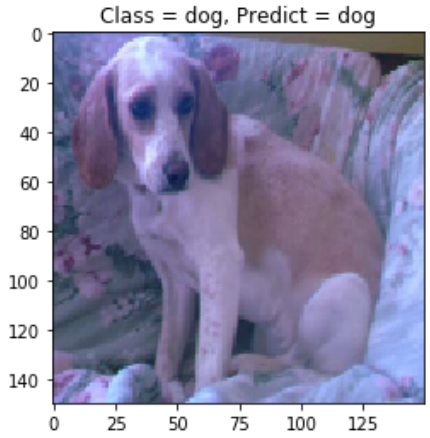

Class = cat, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

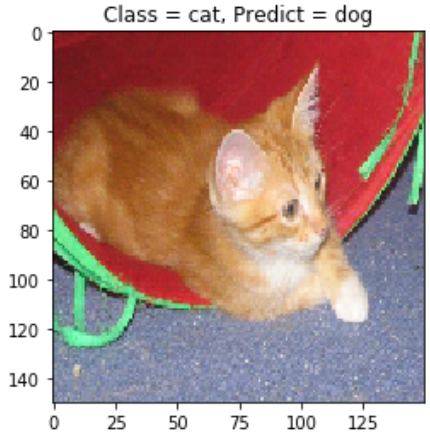

Class = dog, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

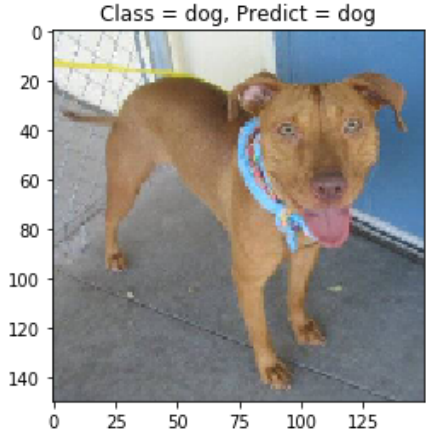

Class = dog, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

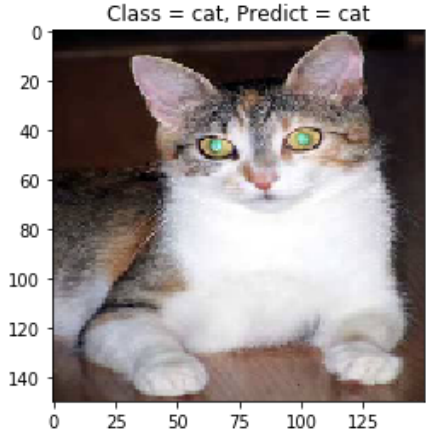

Class = cat, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

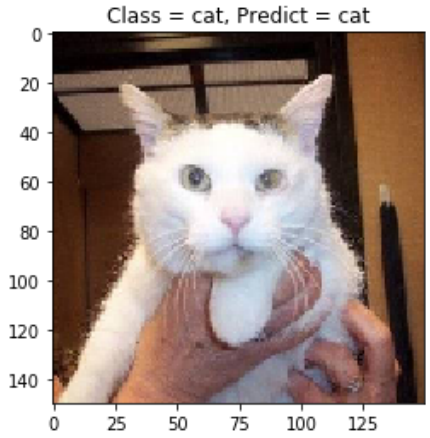

Class = cat, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

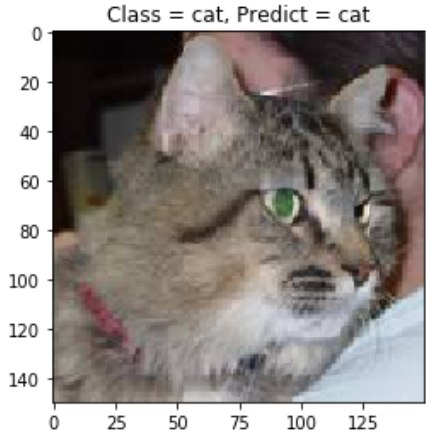

Class = dog, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

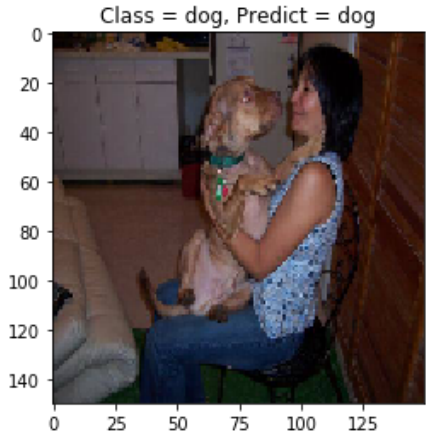

Class = cat, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125


Class = dog, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125


Class = cat, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125







Now we move on to building a more powerful ConvNet: VGG16. We will not train it from scratch but rather load in the weights for the model from a pre-trained version, version that was trained on ImageNet. We will then chop of the last layer and replace it with one that fits our two class cat vs. dog problem. We will freeze the bottom layers of the network and only train the weights of the new last layer. Why? Because if we allow the whole model to train all the weights at once we might do damage to the carefully selected weights in the lower convolutional layers. So instead, we train only the top layer. This training produces a classifier that has 91% accuracy, which is quite an improvement over the last one. However, once this training converges, more or less, we then unfreeze the lower layers and let all the layers train. The final model gets 97% accuracy!

The routine below make a VGG16 network in either Theano or Tensorflow style. The only difference is the ordering of the volume indices. Note that it does not make the last layers as these would have to be removed.

PLEASE NOTE THAT YOUR NUMBERS WILL NOT BE EXACTLY THE SAME AS MINE. SO PLEASE DO NOT ASK ME IF YOUR NUMBERS "ARE GOOD ENOUGH." THIS IS A GUIDELINE FOR WHAT IS POSSIBLE.

In [13]:
```
def build_vgg16(framework='tf'):

    if framework == 'th':
        # build the VGG16 network in Theano weight ordering mode
        backend.set_image_dim_ordering('th')
    else:
        # build the VGG16 network in Tensorflow weight ordering mode
        backend.set_image_dim_ordering('tf')

    model = Sequential()
    if framework == 'th':
        model.add(ZeroPadding2D((1, 1), input_shape=(3, img_width, img_height)))
    else:
        model.add(ZeroPadding2D((1, 1), input_shape=(img_width, img_height, 3)))

    model.add(Conv2D(64, (3, 3), activation='relu', name='conv1_1'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Conv2D(64, (3, 3), activation='relu', name='conv1_2'))
    model.add(MaxPooling2D((2, 2), strides=(2, 2)))

    model.add(ZeroPadding2D((1, 1)))
    model.add(Conv2D(128, (3, 3), activation='relu', name='conv2_1'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Conv2D(128, (3, 3), activation='relu', name='conv2_2'))
    model.add(MaxPooling2D((2, 2), strides=(2, 2)))

    model.add(ZeroPadding2D((1, 1)))
    model.add(Conv2D(256, (3, 3), activation='relu', name='conv3_1'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Conv2D(256, (3, 3), activation='relu', name='conv3_2'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Conv2D(256, (3, 3), activation='relu', name='conv3_3'))
    model.add(MaxPooling2D((2, 2), strides=(2, 2)))

    model.add(ZeroPadding2D((1, 1)))
    model.add(Conv2D(512, (3, 3), activation='relu', name='conv4_1'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Conv2D(512, (3, 3), activation='relu', name='conv4_2'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Conv2D(512, (3, 3), activation='relu', name='conv4_3'))
    model.add(MaxPooling2D((2, 2), strides=(2, 2)))

    model.add(ZeroPadding2D((1, 1)))
    model.add(Conv2D(512, (3, 3), activation='relu', name='conv5_1'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Conv2D(512, (3, 3), activation='relu', name='conv5_2'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Conv2D(512, (3, 3), activation='relu', name='conv5_3'))
    model.add(MaxPooling2D((2, 2), strides=(2, 2)))

    return model
```

Now we build the model using tensorflow format and load the weights.

In [14]:
```
# path to the model weights files.
weights_path = './imagenet_vgg16_fine-tuning/vgg16_weights_tf_dim_ordering_tf_kernels_no
top.h5'
tf_model = build_vgg16('tf')
tf_model.load_weights(weights_path)
```

Next we make the last layer or layers. We flatten the output from the last convolutional layer, and add fully connected layer with 256 hidden units. Finally, we add the output layer which is has a scalar output as we have a binary classifier.

In [17]:
```
# build a classifier model to put on top of the convolutional model
top_model = Sequential()
print(Flatten(input_shape=tf_model.output_shape[1:]))
top_model.add(Flatten(input_shape=tf_model.output_shape[1:]))
top_model.add(Dense(256, activation='relu'))
top_model.add(Dropout(0.5))
top_model.add(Dense(1, activation='sigmoid'))
print(tf_model.summary())
print(top_model.summary())
```

```
<keras.layers.core.Flatten object at 0x7f816dfd46a0>
```

Layer (type)	Output Shape	Param #
zero_padding2d_1 (ZeroPaddin	(None, 152, 152, 3)	0
conv1_1 (Conv2D)	(None, 150, 150, 64)	1792
zero_padding2d_2 (ZeroPaddin	(None, 152, 152, 64)	0
conv1_2 (Conv2D)	(None, 150, 150, 64)	36928
max_pooling2d_4 (MaxPooling2	(None, 75, 75, 64)	0
zero_padding2d_3 (ZeroPaddin	(None, 77, 77, 64)	0
conv2_1 (Conv2D)	(None, 75, 75, 128)	73856
zero_padding2d_4 (ZeroPaddin	(None, 77, 77, 128)	0
conv2_2 (Conv2D)	(None, 75, 75, 128)	147584
max_pooling2d_5 (MaxPooling2	(None, 37, 37, 128)	0
zero_padding2d_5 (ZeroPaddin	(None, 39, 39, 128)	0
conv3_1 (Conv2D)	(None, 37, 37, 256)	295168
zero_padding2d_6 (ZeroPaddin	(None, 39, 39, 256)	0
conv3_2 (Conv2D)	(None, 37, 37, 256)	590080
zero_padding2d_7 (ZeroPaddin	(None, 39, 39, 256)	0
conv3_3 (Conv2D)	(None, 37, 37, 256)	590080
max_pooling2d_6 (MaxPooling2	(None, 18, 18, 256)	0
zero_padding2d_8 (ZeroPaddin	(None, 20, 20, 256)	0
conv4_1 (Conv2D)	(None, 18, 18, 512)	1180160
zero_padding2d_9 (ZeroPaddin	(None, 20, 20, 512)	0
conv4_2 (Conv2D)	(None, 18, 18, 512)	2359808
zero_padding2d_10 (ZeroPaddi	(None, 20, 20, 512)	0
conv4_3 (Conv2D)	(None, 18, 18, 512)	2359808
max_pooling2d_7 (MaxPooling2	(None, 9, 9, 512)	0
zero_padding2d_11 (ZeroPaddi	(None, 11, 11, 512)	0
conv5_1 (Conv2D)	(None, 9, 9, 512)	2359808
zero_padding2d_12 (ZeroPaddi	(None, 11, 11, 512)	0
conv5_2 (Conv2D)	(None, 9, 9, 512)	2359808
zero_padding2d_13 (ZeroPaddi	(None, 11, 11, 512)	0
conv5_3 (Conv2D)	(None, 9, 9, 512)	2359808
max_pooling2d_8 (MaxPooling2	(None, 4, 4, 512)	0

```
Total params: 14,714,688
```

```
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
None
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_5 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_5 (Dense)              (None, 256)               2097408
_________________________________________________________________
dropout_3 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 257
=================================================================
Total params: 2,097,665
Trainable params: 2,097,665
Non-trainable params: 0
_________________________________________________________________
None
```

We add this model to the top of our VGG16 network, freeze all the weights except the top, and compile.

In [18]:
```
# add the model on top of the convolutional base
tf_model.add(top_model)
```

In [19]:
```
# set the first 25 layers (up to the last conv block)
# to non-trainable (weights will not be updated)
for layer in tf_model.layers[:25]:
    layer.trainable = False

# compile the model with a SGD/momentum optimizer
# and a very slow learning rate.
tf_model.compile(loss='binary_crossentropy',
              optimizer=optimizers.SGD(lr=1e-4, momentum=0.9),
              metrics=['accuracy'])
```

Now we train for 5 epochs to get the weights for the top close to where we need them. Essentially, we want the network to be doing the right thing before we unnfreeze the lower weights.

In [21]:
```
tensorboard_callback = TensorBoard(log_dir='./logs/vgg16_fine_tuning/',
                                   histogram_freq=0, write_graph=True,
                                   write_images=False)
checkpoint_callback = \
    ModelCheckpoint(
        './models/vgg16_fine_tuning_weights.{epoch:02d}-{val_acc:.2f}.hdf5',
        monitor='val_acc', verbose=0, save_best_only=True,
        save_weights_only=False, mode='auto', period=1)
```

In [24]:
```
nb_epoch = 5

hist_vgg16_frozen = tf_model.fit_generator(train_generator,
              initial_epoch=0,
              verbose=1,
              validation_data=validation_generator,
              steps_per_epoch=steps_per_epoch_train,
              epochs=nb_epoch,
              callbacks=[tensorboard_callback, checkpoint_callback],
              validation_steps=steps_per_epoch_val)


pandas.DataFrame(hist_vgg16_frozen.history).to_csv("./history/vgg16_fine_tuning_frozen.csv")
```

```
Epoch 1/5
625/625 [==============================] - 147s 236ms/step - loss: 0.1608 - acc: 0.9349
- val_loss: 0.1416 - val_acc: 0.9415
Epoch 2/5
625/625 [==============================] - 148s 236ms/step - loss: 0.1557 - acc: 0.9371
- val_loss: 0.1379 - val_acc: 0.9455
Epoch 3/5
625/625 [==============================] - 148s 236ms/step - loss: 0.1451 - acc: 0.9422
- val_loss: 0.1322 - val_acc: 0.9440
Epoch 4/5
625/625 [==============================] - 146s 234ms/step - loss: 0.1400 - acc: 0.9444
- val_loss: 0.1296 - val_acc: 0.9480
Epoch 5/5
625/625 [==============================] - 147s 234ms/step - loss: 0.1356 - acc: 0.9464
- val_loss: 0.1245 - val_acc: 0.9480
```

Running this, we see that it gets 91% accuracy on the validation set, so we've halved the errors from before.

In [25]:
```
accuracies = np.array([])
losses = np.array([])

i=0
for X_batch, Y_batch in validation_generator:
    loss, accuracy = tf_model.evaluate(X_batch, Y_batch, verbose=0)
    losses = np.append(losses, loss)
    accuracies = np.append(accuracies, accuracy)
    i += 1
    if i == 20:
       break

print("Validation: accuracy = %f  ;  loss = %f" % (np.mean(accuracies), np.mean(losses)))
```

```
Validation: accuracy = 0.939063  ;  loss = 0.139528
```

Now we can unnfreeze the lower layers.

In [26]:
```
# set the first 25 layers (up to the last conv block)
# to non-trainable (weights will not be updated)
for layer in tf_model.layers[:25]:
    layer.trainable = True

# compile the model with a SGD/momentum optimizer
# and a very slow learning rate.
tf_model.compile(loss='binary_crossentropy',
              optimizer=optimizers.SGD(lr=1e-4, momentum=0.9),
              metrics=['accuracy'])
```

We will let this train for 10 epochs.

In [27]:
```
nb_epoch = 10

hist_vgg16_unfreeze = tf_model.fit_generator(train_generator,
              initial_epoch=0,
              verbose=1,
              validation_data=validation_generator,
              steps_per_epoch=steps_per_epoch_train,
              epochs=nb_epoch,
              callbacks=[tensorboard_callback, checkpoint_callback],
              validation_steps=steps_per_epoch_val)


pandas.DataFrame(hist_vgg16_unfreeze.history).to_csv("./history/vgg16_fine_tuning_unfree
ze.csv")
```

```
Epoch 1/10
625/625 [==============================] - 358s 573ms/step - loss: 0.1139 - acc: 0.9538
- val_loss: 0.1070 - val_acc: 0.9560
Epoch 2/10
625/625 [==============================] - 357s 570ms/step - loss: 0.0891 - acc: 0.9652
- val_loss: 0.0784 - val_acc: 0.9660
Epoch 3/10
625/625 [==============================] - 356s 569ms/step - loss: 0.0763 - acc: 0.9716
- val_loss: 0.1371 - val_acc: 0.9440
Epoch 4/10
625/625 [==============================] - 355s 568ms/step - loss: 0.0677 - acc: 0.9729
- val_loss: 0.0737 - val_acc: 0.9720
Epoch 5/10
625/625 [==============================] - 355s 568ms/step - loss: 0.0604 - acc: 0.9773
- val_loss: 0.0785 - val_acc: 0.9720
Epoch 6/10
625/625 [==============================] - 355s 567ms/step - loss: 0.0527 - acc: 0.9797
- val_loss: 0.0677 - val_acc: 0.9745
Epoch 7/10
625/625 [==============================] - 355s 569ms/step - loss: 0.0502 - acc: 0.9808
- val_loss: 0.1058 - val_acc: 0.9620
Epoch 8/10
625/625 [==============================] - 355s 567ms/step - loss: 0.0481 - acc: 0.9819
- val_loss: 0.0605 - val_acc: 0.9765
Epoch 9/10
625/625 [==============================] - 355s 568ms/step - loss: 0.0423 - acc: 0.9842
- val_loss: 0.0644 - val_acc: 0.9755
Epoch 10/10
625/625 [==============================] - 354s 567ms/step - loss: 0.0398 - acc: 0.9856
- val_loss: 0.0646 - val_acc: 0.9750
```

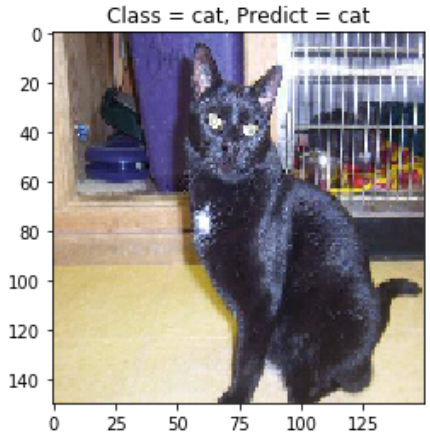
We get to 97.5% accuracy! The network seems to have started overfitting towards the last few epochs

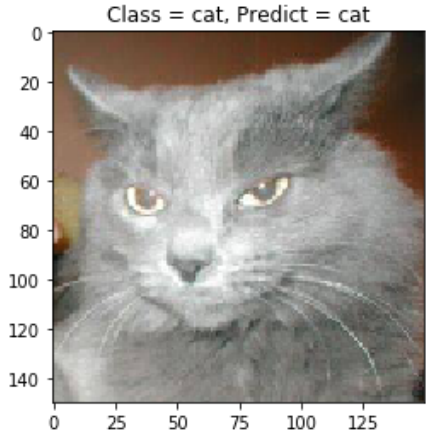
In [29]:
```
accuracies = np.array([])
losses = np.array([])

i=0
for X_batch, Y_batch in validation_generator:
    loss, accuracy = tf_model.evaluate(X_batch, Y_batch, verbose=0)
    losses = np.append(losses, loss)
    accuracies = np.append(accuracies, accuracy)
    i += 1
    if i == 20:
       break

print("Validation: accuracy = %f  ;  loss = %f" % (np.mean(accuracies), np.mean(losses
)))
```

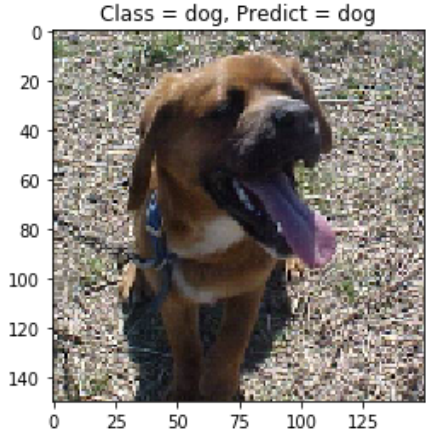
```
Validation: accuracy = 0.970313  ;  loss = 0.070810
```

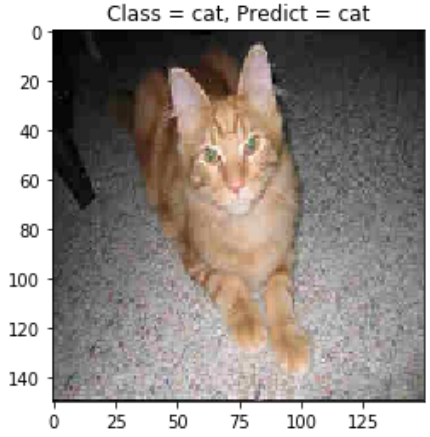
Some sample examples and their predictions from validation set

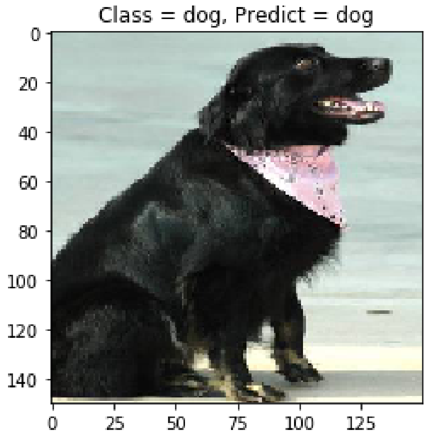
In [31]:
```
X_test, y_test = next(validation_generator)
predictions = tf_model.predict_classes(X_test, batch_size=32, verbose=0)

for i in range(15):
    show_sample(X_test[i, :, :, :], y_test[i], prediction=predictions[i, 0])
```

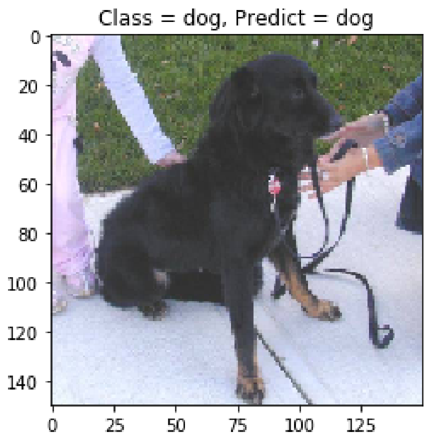

Class = cat, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

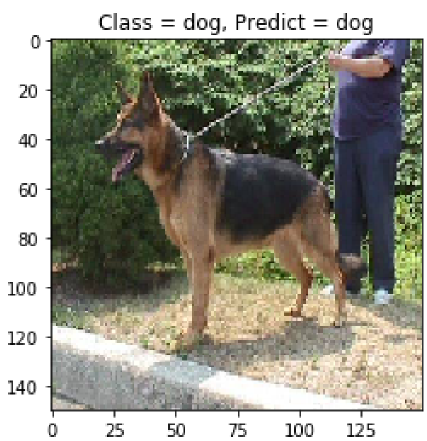

Class = cat, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

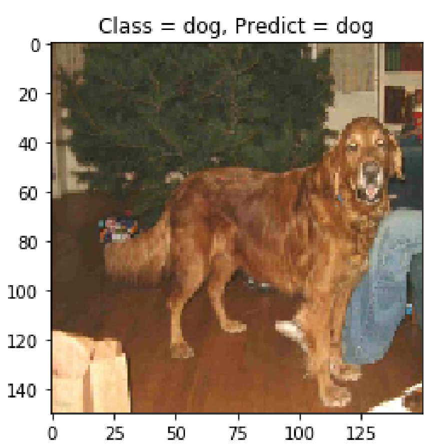

Class = dog, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

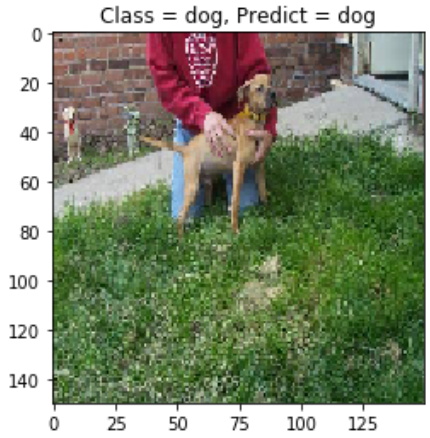

Class = cat, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

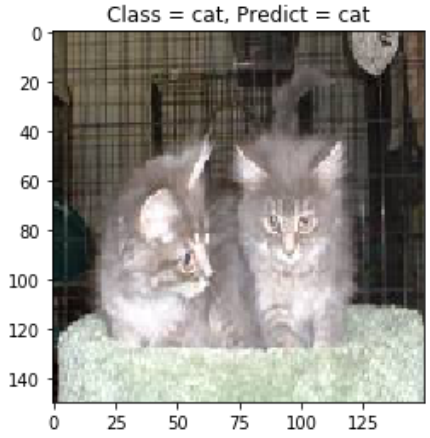

Class = dog, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

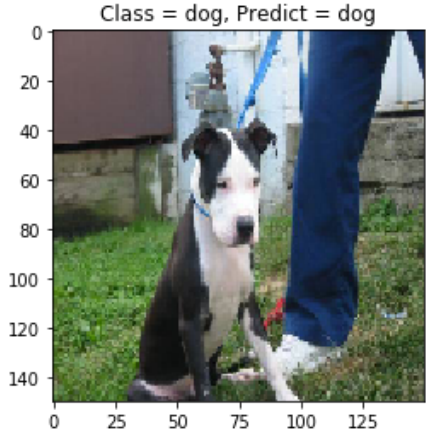

Class = dog, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

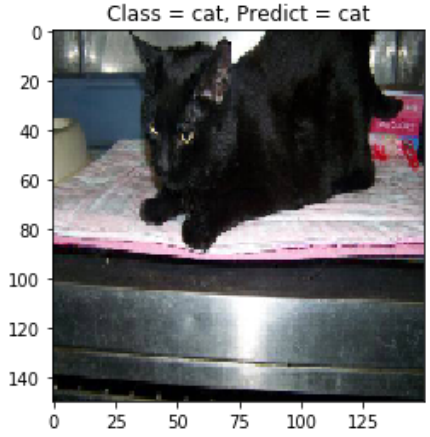

Class = dog, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

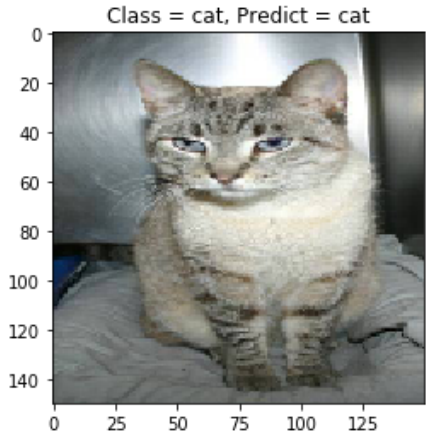

Class = dog, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

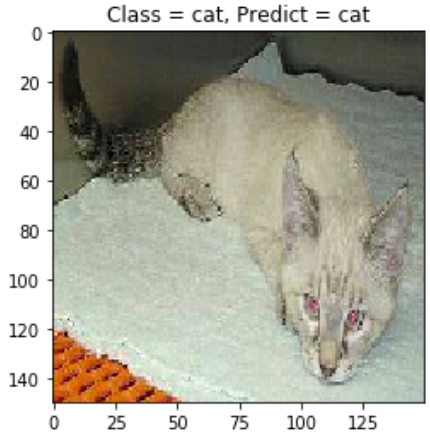

Class = dog, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

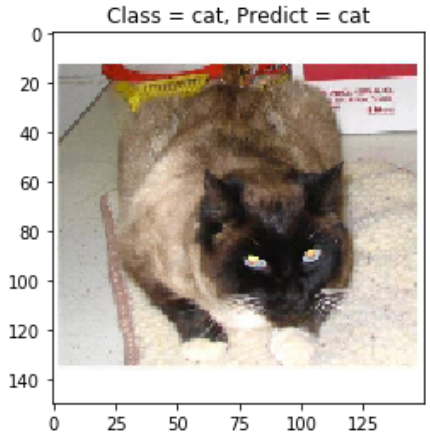

Class = cat, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125


Class = dog, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125


Class = cat, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125







We let it go one last time and see that it pushes up just a bit higher to 97%. Also note that it looks like it is beginning to overfit as the training loss is coming way down and the training accuracy is going well beyond the validation accuracy.

In [32]:
```
nb_epoch = 5

hist_vgg16_unfreeze_push = tf_model.fit_generator(train_generator,
              initial_epoch=0,
              verbose=1,
              validation_data=validation_generator,
              steps_per_epoch=steps_per_epoch_train,
              epochs=nb_epoch,
              callbacks=[tensorboard_callback, checkpoint_callback],
              validation_steps=steps_per_epoch_val)


pandas.DataFrame(hist_vgg16_unfreeze_push.history).to_csv("./history/vgg16_fine_tuning_u
nfreeze_push.csv")
```

```
Epoch 1/5
625/625 [==============================] - 354s 567ms/step - loss: 0.0383 - acc: 0.9856
- val_loss: 0.0567 - val_acc: 0.9785
Epoch 2/5
625/625 [==============================] - 355s 568ms/step - loss: 0.0320 - acc: 0.9886
- val_loss: 0.0931 - val_acc: 0.9645
Epoch 3/5
625/625 [==============================] - 354s 566ms/step - loss: 0.0302 - acc: 0.9892
- val_loss: 0.0657 - val_acc: 0.9745
Epoch 4/5
625/625 [==============================] - 354s 566ms/step - loss: 0.0295 - acc: 0.9890
- val_loss: 0.0712 - val_acc: 0.9715
Epoch 5/5
625/625 [==============================] - 353s 565ms/step - loss: 0.0274 - acc: 0.9902
- val_loss: 0.0683 - val_acc: 0.9755
```

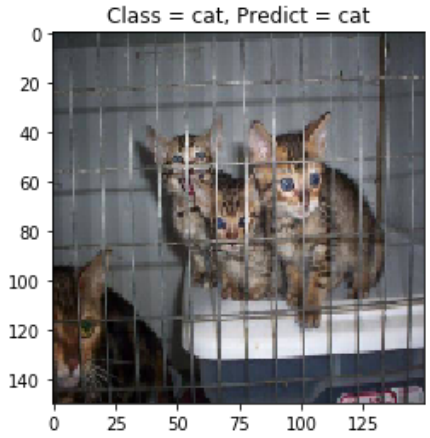
Wow! 97% accuracy! And we are done...

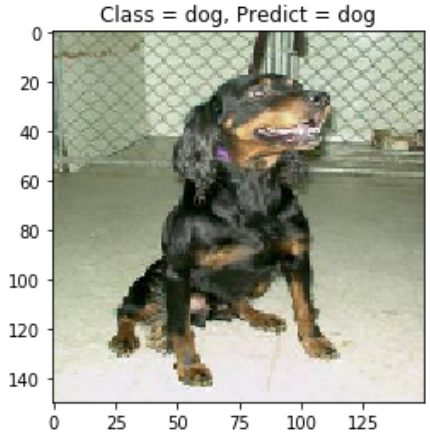
In [33]:
```
accuracies = np.array([])
losses = np.array([])

i=0
for X_batch, Y_batch in validation_generator:
    loss, accuracy = tf_model.evaluate(X_batch, Y_batch, verbose=0)
    losses = np.append(losses, loss)
    accuracies = np.append(accuracies, accuracy)
    i += 1
    if i == 20:
       break

print("Validation: accuracy = %f  ;  loss = %f" % (np.mean(accuracies), np.mean(losses
)))
```

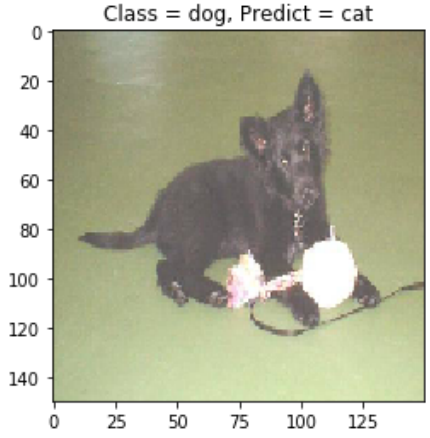
```
Validation: accuracy = 0.981250  ;  loss = 0.057819
```

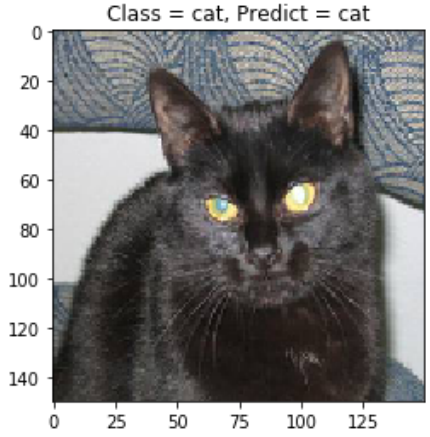
The network has clearly started overfitting a little but it still gives over 98% validation accuracy which is pretty awesome! Further training may not be very wise. Below, there is one sample from the validation data that is incorrectly classified.

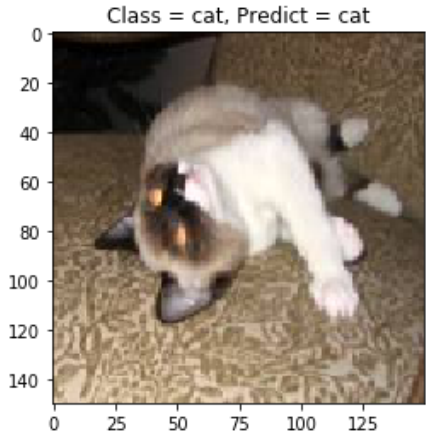
In [34]:
```
X_test, y_test = next(validation_generator)
predictions = tf_model.predict_classes(X_test, batch_size=32, verbose=0)

for i in range(15):
    show_sample(X_test[i, :, :, :], y_test[i], prediction=predictions[i, 0])
```

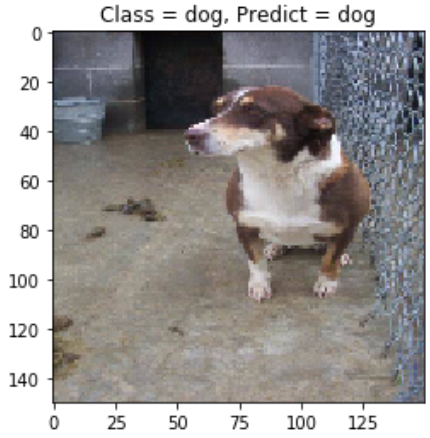

Class = cat, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

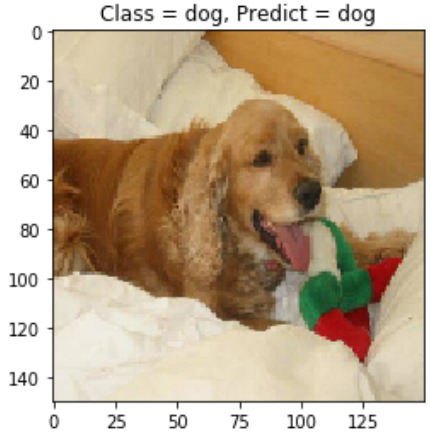

Class = dog, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

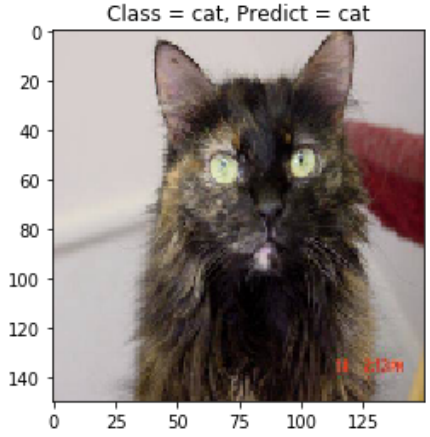

Class = dog, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

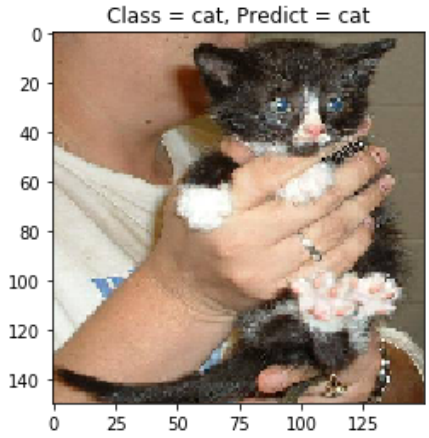

Class = cat, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

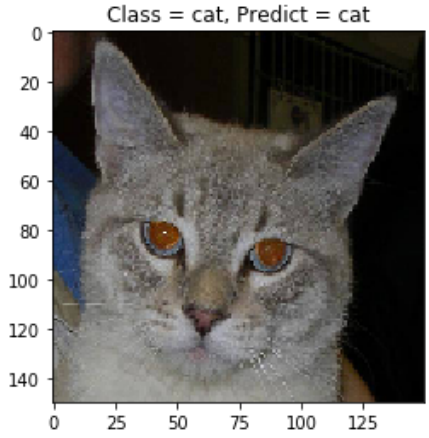

Class = cat, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

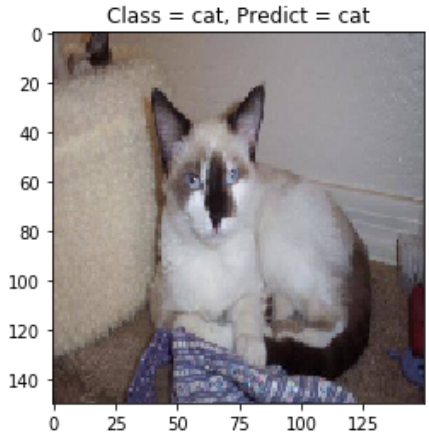

Class = dog, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

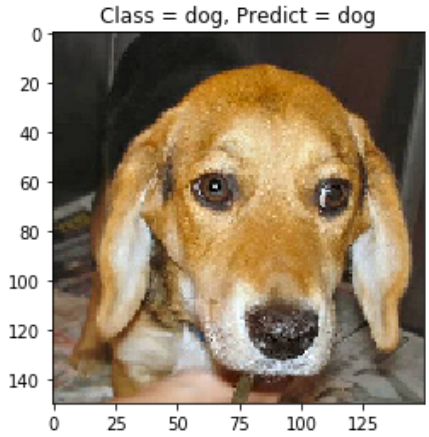

Class = dog, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

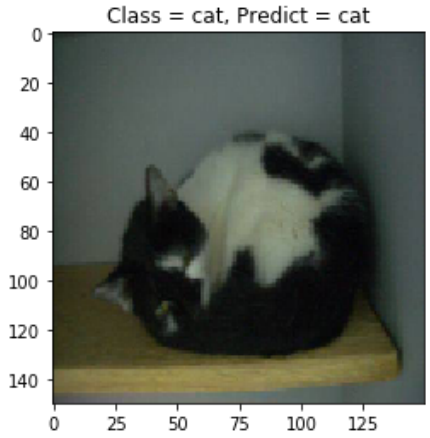

Class = cat, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

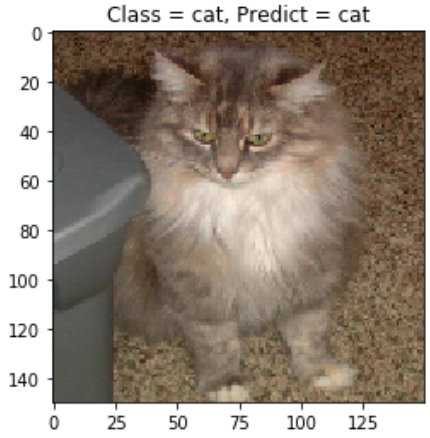

Class = cat, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

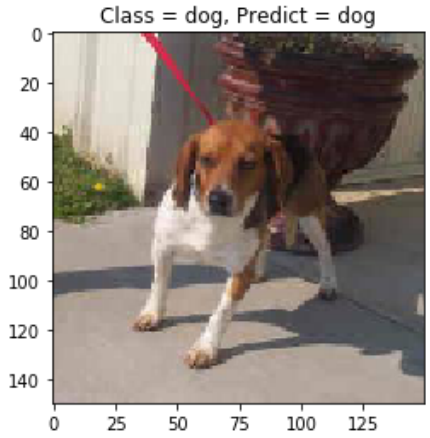

Class = cat, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125


Class = cat, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125


Class = dog, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

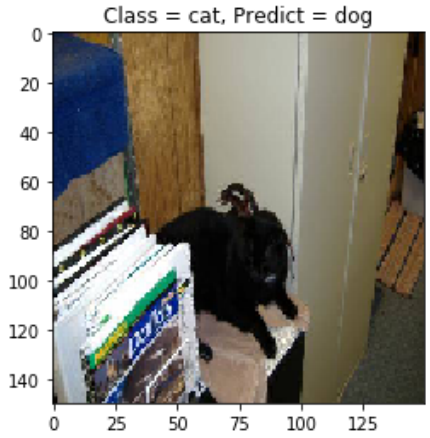


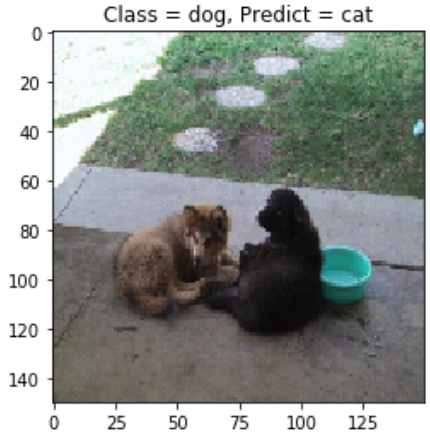


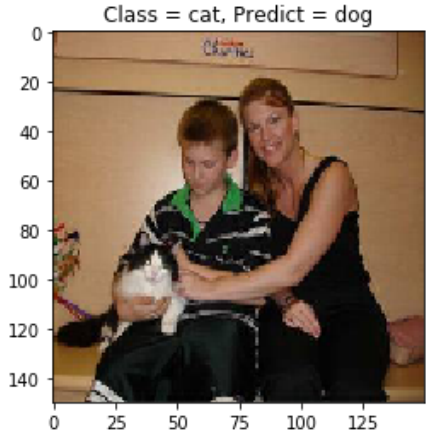


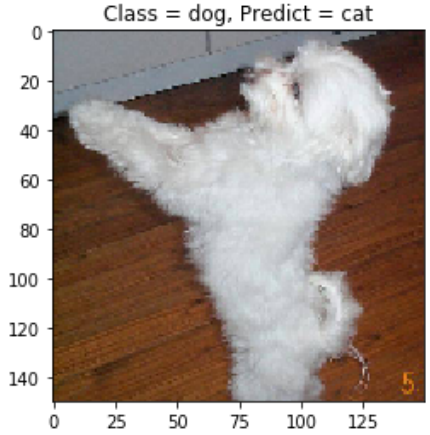
Let's visualize 20 samples that are incorrectly classified to understand if the network gets confused in difficult samples or easy ones compared to human standards

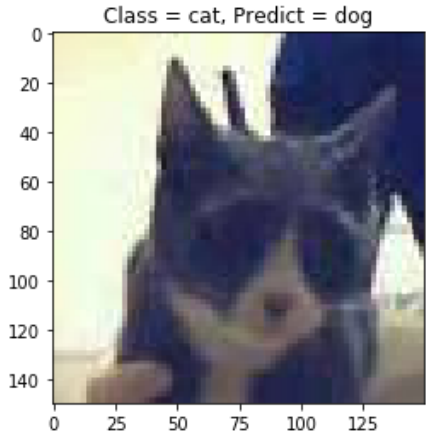
In [46]:
```
accuracies = np.array([])
losses = np.array([])

samples = 0
for X_batch, Y_batch in validation_generator:
    predictions = tf_model.predict_classes(X_batch, batch_size=32, verbose=0)
    idx = np.argwhere(Y_batch != predictions[:, 0])
    idx = idx[:, 0]
    for i in idx:
        show_sample(X_batch[i, :, :, :], Y_batch[i], prediction = predictions[i, 0])
        samples += 1
    if samples == 20:
       break
```

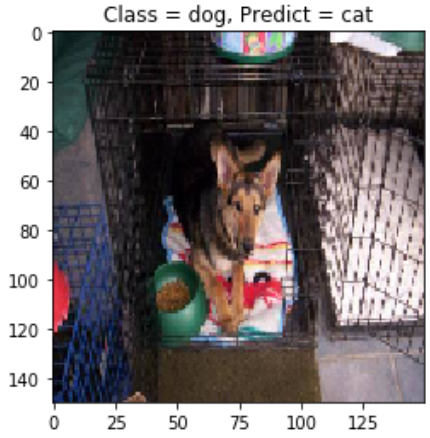

Class = cat, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

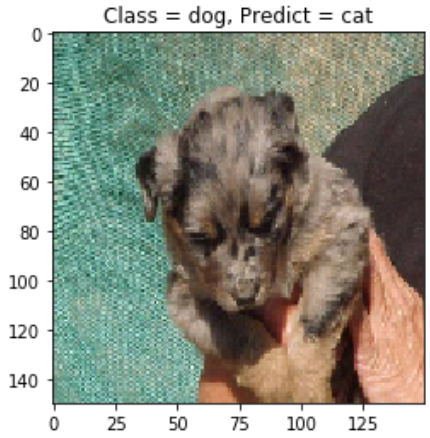

Class = dog, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

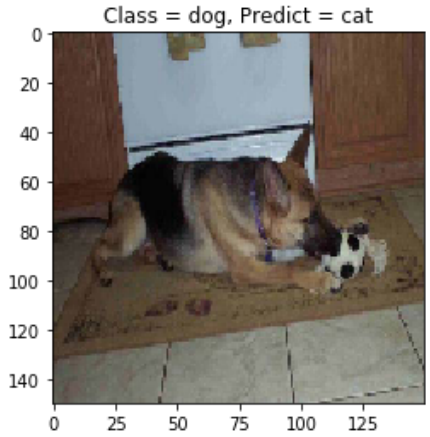

Class = cat, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

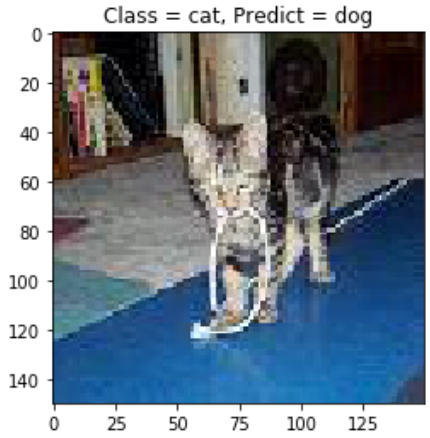

Class = dog, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

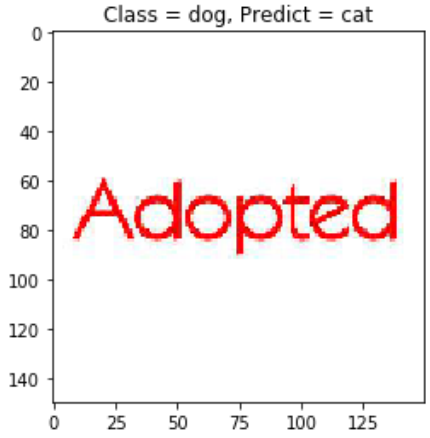

Class = cat, Predict = dog
0
20
40
60
80
100
120
140
0
25
50
75
100
125

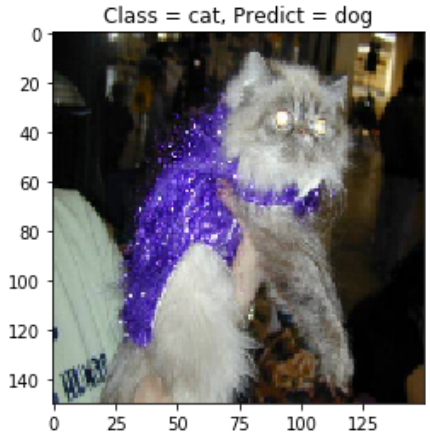

Class = dog, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

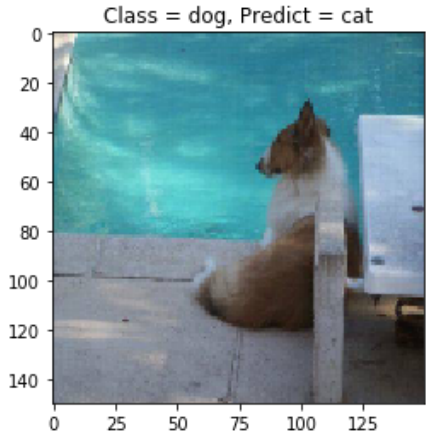

Class = dog, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

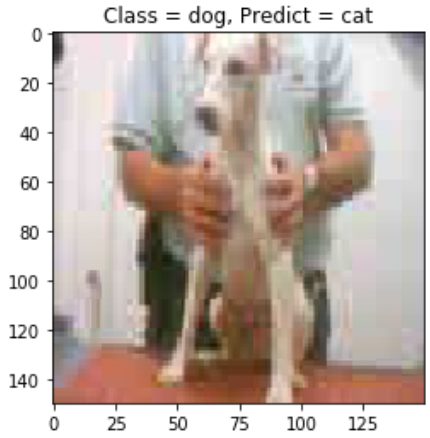

Class = dog, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

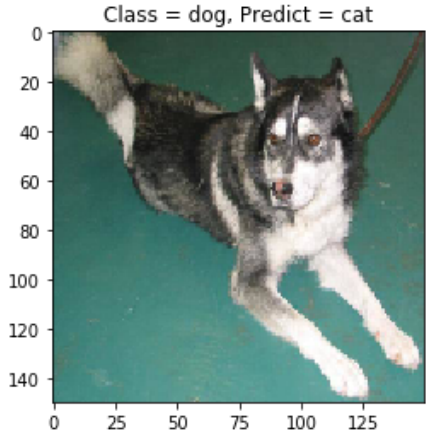

Class = cat, Predict = dog

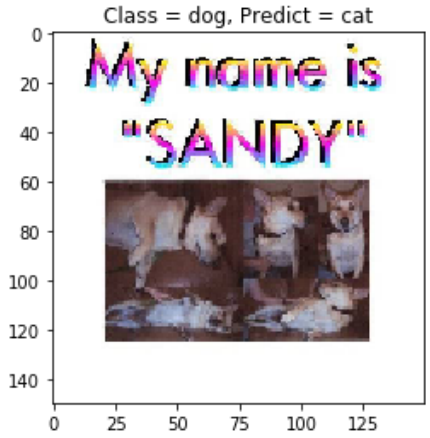

Class = dog, Predict = cat
Adopted

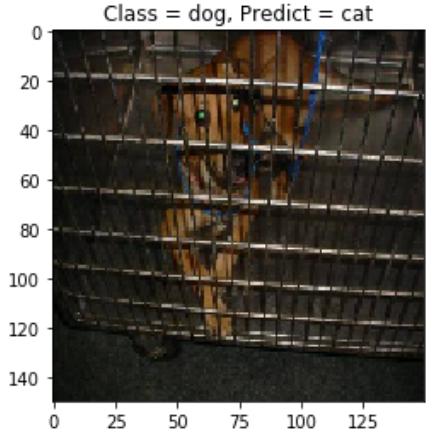

Class = cat, Predict = dog

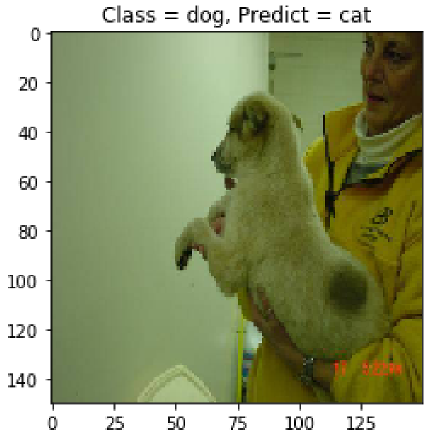

Class = dog, Predict = cat

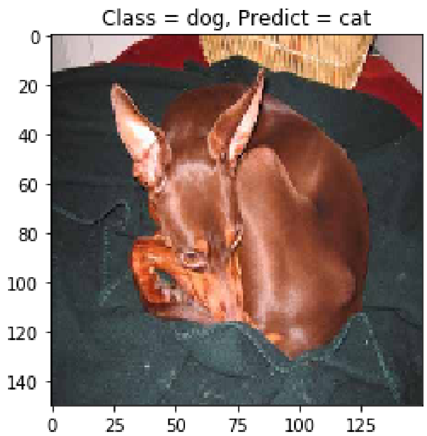

Class = dog, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

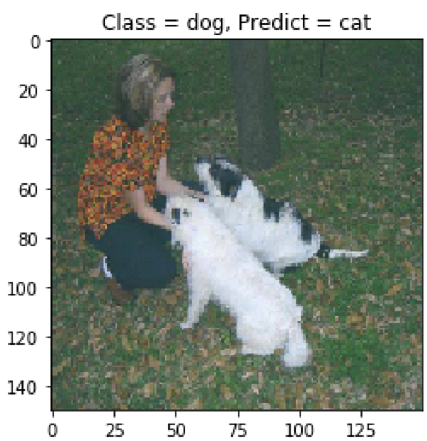

Class = dog, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125

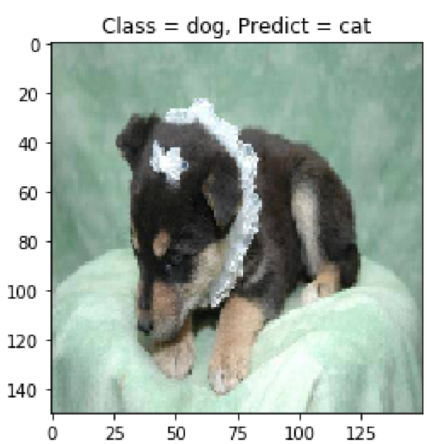

Class = dog, Predict = cat
My name is
"SANDY"
0
20
40
60
80
100
120
140
0
25
50
75
100
125


Class = dog, Predict = cat
0
20
40
60
80
100
120
140
0
25
50
75
100
125


Class = dog, Predict = cat


Class = dog, Predict = cat


Class = dog, Predict = cat


Class = dog, Predict = cat

Some of the above examples look genuinely difficult while some seemed to be missed. So, we could expect the network to do a little better as well.