Image Data Augmentation and Generator with Keras

In this notebook, we will perform data augmentation and automatically label the images with the help of Keras' ImageDataGenerator class.

Data augmentation encompasses a wide range of techniques used to generate "new" training samples from the original ones by applying random jitters and perturbations (but at the same time ensuring that the class labels of the data are not changed). Our goal when applying data augmentation is to increase the generalizability of the model. Image data augmentation artificially creates training images through different ways such as random rotation, shifts, shear and flips, etc.

Keras ImageDataGenerator class works by:

- Accepting a batch of images used for training.
- Taking this batch and applying a series of random transformations to each image in the batch (including random rotation, resizing, shearing, etc.).
- Replacing the original batch with the new, randomly transformed batch.
- Training the CNN on this randomly transformed batch (i.e., the original data itself is not used for training).

[Ref (https://www.pyimagesearch.com/2019/07/08/keras-imagedatagenerator-and-data-augmentation/)](https://www.pyimagesearch.com/2019/07/08/keras-imagedatagenerator-and-data-augmentation/)

In [1]:
```
%matplotlib inline
from tensorflow.python.util import deprecation
deprecation._PRINT_DEPRECATION_WARNINGS = False
import matplotlib.pyplot as plt
import os
import zipfile
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import RMSprop, Adam
from keras.preprocessing import image
from tensorflow.keras.models import clone_model
```

```
Using TensorFlow backend.
```

1. Data File Pre-Processing

The following python code will use the OS library to use Operating System libraries, giving us access to the file system, and the zipfile library allowing us to unzip the data.

1.1 File Pre-Processing

In [2]:
```
# training and development dataset

zipfile_path = 'data/cats_dogs_train.zip'
zip_file = zipfile.ZipFile(zipfile_path, 'r') # Open a ZIP file
zip_file.extractall(path = 'data/') # Extract all members from the archive

zipfile_path = 'data/cats_dogs_validation.zip'
zip_file = zipfile.ZipFile(zipfile_path, 'r') # Open a ZIP file
zip_file.extractall(path = 'data/') # Extract all members from the archive

zip_file.close() # Close the archive file
```

The contents of the .zip are extracted to the base directory `data/cats_dogs_train/train` (for training) and `data/cats_dogs_validation/validation` (for development), which in turn each contains `cats` and `dogs` subdirectories.

We do not explicitly label the images as cats or dogs. We'll use Image Generator -- and this is coded to read images from subdirectories, and automatically label them from the name of that subdirectory.

In [3]:
```
# Directory with our training cat pictures
train_cat_dir = os.path.join('data/cats_dogs_train/train/cats')
# Directory with our training dog pictures
train_dog_dir = os.path.join('data/cats_dogs_train/train/dogs')
# Directory with our dev cat pictures
dev_cat_dir = os.path.join('data/cats_dogs_validation/validation/cats')
# Directory with our dev dog pictures
dev_dog_dir = os.path.join('data/cats_dogs_validation/validation/dogs')
```

Now, let's see what the filenames look like in the `cats` and `dogs` directories:

In [4]:
```python
# print some filenames in the cats and dogs directories

train_cat_names = os.listdir(train_cat_dir)
print('training cat names: ' + str(train_cat_names[:5]))

train_dog_names = os.listdir(train_dog_dir)
print('training dog names: ' + str(train_dog_names[:5]))

dev_cat_names = os.listdir(dev_cat_dir)
print('dev cat names: ' + str(dev_cat_names[:5]))

dev_dog_names = os.listdir(dev_dog_dir)
print('dev dog names: ' + str(dev_dog_names[:5]))
```

```
training cat names: ['cat.0.jpg', 'cat.1.jpg', 'cat.10.jpg', 'cat.100.jpg', 'cat.101.jpg']
training dog names: ['dog.0.jpg', 'dog.1.jpg', 'dog.10.jpg', 'dog.100.jpg', 'dog.101.jpg']
dev cat names: ['cat.2000.jpg', 'cat.2001.jpg', 'cat.2002.jpg', 'cat.2003.jpg', 'cat.2004.jpg']
dev dog names: ['dog.2000.jpg', 'dog.2001.jpg', 'dog.2002.jpg', 'dog.2003.jpg', 'dog.2004.jpg']
```

Let's find out the total number of cat and dog images in the directories.

In [5]:
```python
print('total training cat images:', len(train_cat_names))
print('total training dog images:', len(train_dog_names))
print('total dev cat images:', len(dev_cat_names))
print('total dev dog images:', len(dev_dog_names))
```

```
total training cat images: 1000
total training dog images: 1000
total dev cat images: 500
total dev dog images: 500
```

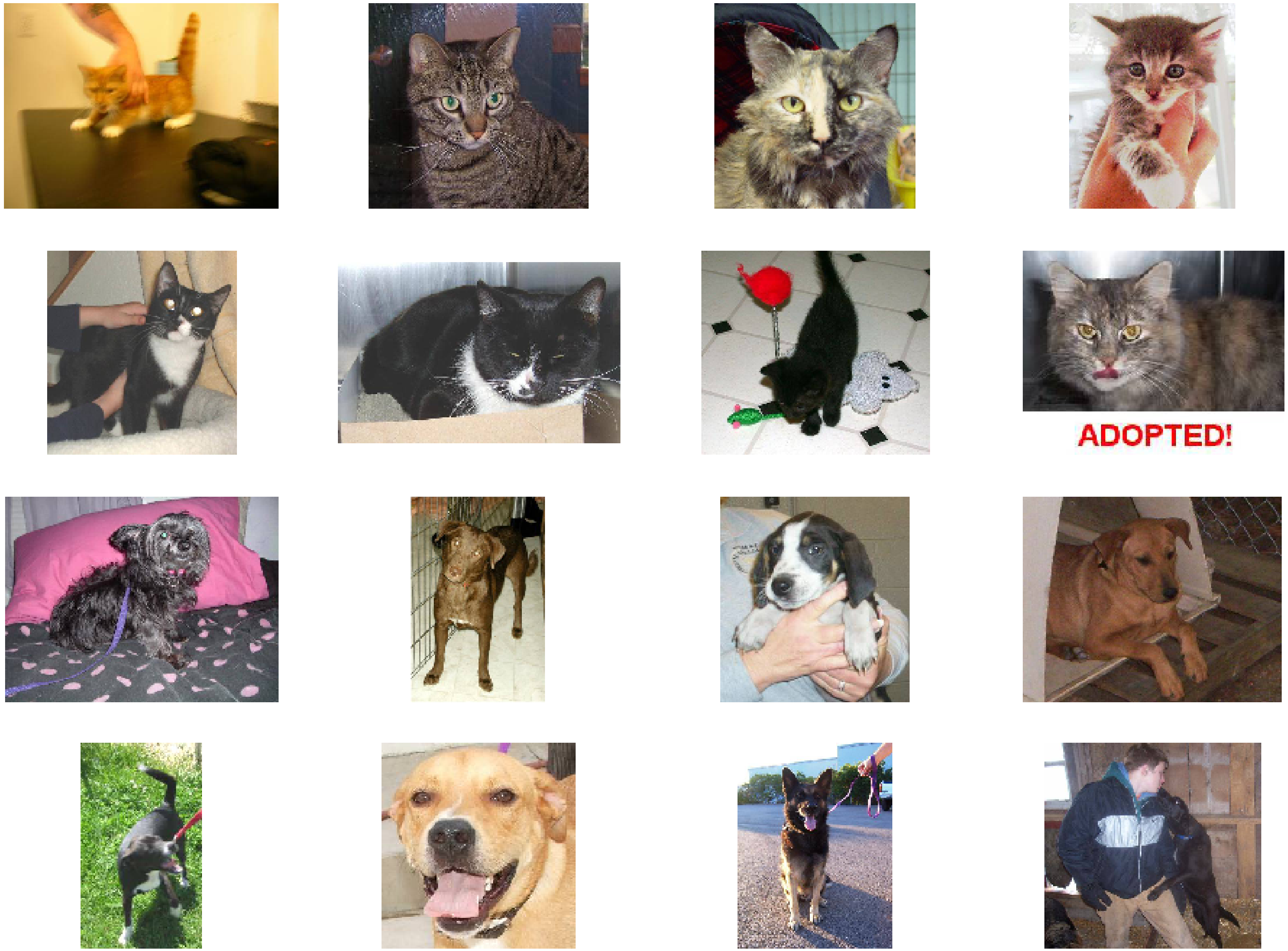
1.2 Image Data Examples

Now let's take a look at a few pictures to get a better sense of what they look like.

In [6]:
```
# we'll output images in a nrows x ncols configuration

nrows = 4
ncols = 4

train_cat_paths = [os.path.join(train_cat_dir, fname) for fname in train_cat_names]
train_dog_paths = [os.path.join(train_dog_dir, fname) for fname in train_dog_names]
dev_cat_paths = [os.path.join(dev_cat_dir, fname) for fname in dev_cat_names]
dev_dog_paths = [os.path.join(dev_dog_dir, fname) for fname in dev_dog_names]
```

Display a batch of 8 cat and 8 dog pictures.

In [7]:
```
fig = plt.gcf()
fig.set_size_inches(ncols * 4, nrows * 3)

# iterating over 16 images
for i, img_path in enumerate(train_cat_paths[0:4] + dev_cat_paths[0:4] + train_dog_paths[0:4] + dev_dog_paths[0:4]):
    sp = plt.subplot(nrows, ncols, i + 1)
    sp.axis('off')
    img = plt.imread(img_path)
    plt.imshow(img)
```


ADOPTED!

2. Image Data Augmentation and Generator

Let's set up data generators that will read pictures in our source folders, convert them to `float32` tensors, and feed them (with their labels) to our network. We'll have one generator for the training images and one for the validation images. Our generators will yield batches of images of size 150x150 and their labels (binary).

Data that goes into neural networks should usually be normalized in some way to make it more amenable to processing by the network (it is uncommon to feed raw pixels into a convnet). In our case, we will preprocess our images by normalizing the pixel values to be in the `[0, 1]` range (originally all values are in the `[0, 255]` range).

In Keras this can be done via the `keras.preprocessing.image.ImageDataGenerator` class using the `rescale` parameter. This `ImageDataGenerator` class allows us to instantiate generators of augmented image batches (and their labels) via `.flow(data, labels)` or `.flow_from_directory(directory)`. These generators can then be used with the Keras model methods that accept data generators as inputs: `fit_generator`, `evaluate_generator`, and `predict_generator`.

Image Augmentation

There are properties on ImageGenerator that we can use to augment the image.

```
# Updated to do image augmentation
train_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. This is relevant when there are no assumptions of horizontal assymmetry (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.

In [8]:

```python
# create image data

# training set without image data augmentation for comparison
train_gen = ImageDataGenerator(rescale = 1/255)
train_generator = train_gen.flow_from_directory(
    'data/cats_dogs_train/train',  # This is the source directory for training images
    target_size = (150, 150),  # All images will be resized to this dimension
    batch_size = 20,
    class_mode = 'binary')

# training set with image data augmentation
train_gen_aug = 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,
    fill_mode = 'nearest')
train_generator_aug = train_gen_aug.flow_from_directory(
    'data/cats_dogs_train/train',  # This is the source directory for training images
    target_size = (150, 150),  # All images will be resized to this dimension
    batch_size = 20,
    class_mode = 'binary')

# dev set
dev_gen = ImageDataGenerator(rescale = 1/255)
dev_generator = dev_gen.flow_from_directory(
    'data/cats_dogs_validation/validation/',  # This is the source directory for dev images
    target_size = (150, 150),  # All images will be resized to this dimension
    batch_size = 20,
    class_mode = 'binary')

# print class indices
print("Class index: "  + str(train_generator.class_indices))
```

```
Found 2000 images belonging to 2 classes.
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Class index: {'cats': 0, 'dogs': 1}
```

3. Build the CNN Model

We add convolutional and pooling layers, and then flatten the result to feed into densely connected layers.

Note that because we are facing a two-class classification problem, i.e. a **binary classification problem**, we will end our network with a **sigmoid** activation, so that the output of our network will be a single scalar between 0 and 1, encoding the probability that the current image is class 1 (as opposed to class 0).

In [9]:
```
model = tf.keras.models.Sequential([
    # The first convolution and pooling
    tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(150, 150, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    # The second convolution and pooling
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # The third convolution and pooling
    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # The fourth convolution and pooling
    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # Flatten the results to feed into a DNN
    tf.keras.layers.Flatten(),
    # 512-neuron hidden layer
    tf.keras.layers.Dense(512, activation='relu'),
    # output neuron
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

In [10]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
conv2d (Conv2D)             (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 74, 74, 32)       0
_________________________________________________________________
conv2d_1 (Conv2D)           (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 36, 36, 64)       0
_________________________________________________________________
conv2d_2 (Conv2D)           (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 17, 17, 128)      0
_________________________________________________________________
conv2d_3 (Conv2D)           (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 7, 7, 128)        0
_________________________________________________________________
flatten (Flatten)           (None, 6272)              0
_________________________________________________________________
dense (Dense)               (None, 512)               3211776
_________________________________________________________________
dense_1 (Dense)             (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
_________________________________________________________________
```

4. Model Training and Comparison

We will compare the performance of the model trained with and without data augmentation.

In [11]:
```python
# A callback is a set of functions to be applied at given stages of the training procedure.
# We can use callbacks to get a view on internal states and statistics of the model during training.
# We can pass a list of callbacks (as the keyword argument callbacks) to the .fit() method of the Sequential
# or Model classes.
# The relevant methods of the callbacks will then be called at each stage of the training.

class myCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        if(logs.get('acc') > 0.98):
            print("\nReached target accuracy so cancelling training!")
            self.model.stop_training = True

callbacks = myCallback()
```

In [12]:
```python
# create function to plot the training progress

def plot_progress(history, title):
    """
    Plot the loss and accuracy of training and validation data, as functions of epochs

    Arguments:
    history: the History object returned by model.fit_generator()
    title: the title of the graph
    """

    train_loss = history.history['loss']
    train_acc = history.history['acc']
    val_loss = history.history['val_loss']
    val_acc = history.history['val_acc']
    epochs = range(len(train_acc))

    fig, axes = plt.subplots(1, 2)
    fig.suptitle(title, fontsize = 20)
    fig.set_size_inches(16, 5)
    axes[0].plot(epochs, train_acc, 'bo', label = 'Training Accuracy')
    axes[0].plot(epochs, val_acc, 'b', label = 'Validation Accuracy')
    axes[0].set_title('Accuracy')
    axes[0].legend()
    axes[1].plot(epochs, train_loss, 'bo', label = 'Training Loss')
    axes[1].plot(epochs, val_loss, 'b', label = 'Validation Loss')
    axes[1].set_title('Loss')
    axes[1].legend()
    plt.show()
```

4.1 Without Data Augmentation

We will train our model with the `binary_crossentropy` loss. We will use the `RMSprop` optimizer with a learning rate of `0.0001`. During training, we will want to monitor classification accuracy.

In [13]:
```
model_no_aug = clone_model(model)
model_no_aug.compile(loss = 'binary_crossentropy',
                     optimizer = RMSprop(lr = 1e-4),
                     metrics = ['acc'])
```

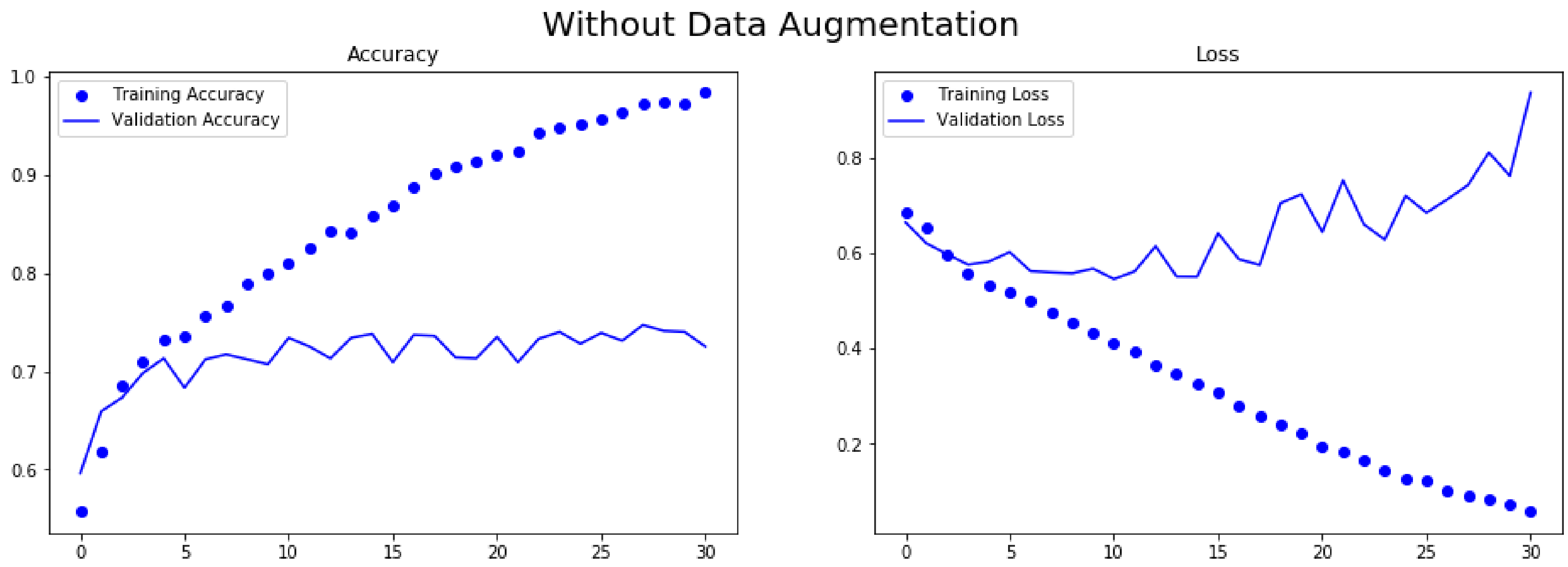
In [14]:
```
history_no_aug = model_no_aug.fit_generator(
      train_generator,
      steps_per_epoch = 100, # number of images = batch_size * steps
      epochs = 100,
      verbose = 1,
      validation_data = dev_generator,
      validation_steps = 50, # number of images = batch_size * steps
      callbacks = [callbacks])
```

```
Epoch 1/100
100/100 [==============================] - 75s 750ms/step - loss: 0.6857 - acc: 0.5565 - val_loss: 0.6648 - val_acc: 0.5960
Epoch 2/100
100/100 [==============================] - 72s 724ms/step - loss: 0.6517 - acc: 0.6185 - val_loss: 0.6209 - val_acc: 0.6590
Epoch 3/100
100/100 [==============================] - 73s 735ms/step - loss: 0.5961 - acc: 0.6855 - val_loss: 0.5979 - val_acc: 0.6730
Epoch 4/100
100/100 [==============================] - 73s 729ms/step - loss: 0.5569 - acc: 0.7095 - val_loss: 0.5760 - val_acc: 0.6980
Epoch 5/100
100/100 [==============================] - 71s 714ms/step - loss: 0.5327 - acc: 0.7320 - val_loss: 0.5824 - val_acc: 0.7130
Epoch 6/100
100/100 [==============================] - 72s 717ms/step - loss: 0.5166 - acc: 0.7360 - val_loss: 0.6024 - val_acc: 0.6830
Epoch 7/100
100/100 [==============================] - 71s 714ms/step - loss: 0.5010 - acc: 0.7555 - val_loss: 0.5624 - val_acc: 0.7120
Epoch 8/100
100/100 [==============================] - 73s 733ms/step - loss: 0.4767 - acc: 0.7665 - val_loss: 0.5597 - val_acc: 0.7170
Epoch 9/100
100/100 [==============================] - 72s 722ms/step - loss: 0.4555 - acc: 0.7890 - val_loss: 0.5579 - val_acc: 0.7120
Epoch 10/100
100/100 [==============================] - 72s 724ms/step - loss: 0.4310 - acc: 0.7990 - val_loss: 0.5676 - val_acc: 0.7070
Epoch 11/100
100/100 [==============================] - 76s 759ms/step - loss: 0.4124 - acc: 0.8100 - val_loss: 0.5460 - val_acc: 0.7340
Epoch 12/100
100/100 [==============================] - 71s 711ms/step - loss: 0.3924 - acc: 0.8260 - val_loss: 0.5618 - val_acc: 0.7250
Epoch 13/100
100/100 [==============================] - 73s 733ms/step - loss: 0.3660 - acc: 0.8430 - val_loss: 0.6148 - val_acc: 0.7130
Epoch 14/100
100/100 [==============================] - 71s 715ms/step - loss: 0.3488 - acc: 0.8410 - val_loss: 0.5514 - val_acc: 0.7340
Epoch 15/100
100/100 [==============================] - 72s 716ms/step - loss: 0.3265 - acc: 0.8575 - val_loss: 0.5510 - val_acc: 0.7380
Epoch 16/100
100/100 [==============================] - 73s 732ms/step - loss: 0.3076 - acc: 0.8680 - val_loss: 0.6418 - val_acc: 0.7090
Epoch 17/100
100/100 [==============================] - 73s 735ms/step - loss: 0.2805 - acc: 0.8875 - val_loss: 0.5873 - val_acc: 0.7370
Epoch 18/100
100/100 [==============================] - 72s 725ms/step - loss: 0.2590 - acc: 0.9020 - val_loss: 0.5753 - val_acc: 0.7360
Epoch 19/100
100/100 [==============================] - 72s 724ms/step - loss: 0.2416 - acc: 0.9085 - val_loss: 0.7050 - val_acc: 0.7140
Epoch 20/100
100/100 [==============================] - 72s 720ms/step - loss: 0.2224 - acc: 0.9145 - val_loss: 0.7235 - val_acc: 0.7130
Epoch 21/100
100/100 [==============================] - 71s 714ms/step - loss: 0.1958 - acc: 0.9210 - val_loss: 0.6444 - val_acc: 0.7350
Epoch 22/100
100/100 [==============================] - 72s 719ms/step - loss: 0.1838 - acc: 0.9235 - val_loss: 0.7529 - val_acc: 0.7090
Epoch 23/100
```

```
100/100 [==============================] - 72s 722ms/step - loss: 0.1663 - acc: 0.9435 - val_loss: 0.6603 - val_acc: 0.7330
Epoch 24/100
100/100 [==============================] - 72s 715ms/step - loss: 0.1449 - acc: 0.9475 - val_loss: 0.6284 - val_acc: 0.7400
Epoch 25/100
100/100 [==============================] - 73s 730ms/step - loss: 0.1282 - acc: 0.9525 - val_loss: 0.7203 - val_acc: 0.7280
Epoch 26/100
100/100 [==============================] - 73s 734ms/step - loss: 0.1214 - acc: 0.9565 - val_loss: 0.6849 - val_acc: 0.7390
Epoch 27/100
100/100 [==============================] - 72s 717ms/step - loss: 0.1024 - acc: 0.9640 - val_loss: 0.7127 - val_acc: 0.7310
Epoch 28/100
100/100 [==============================] - 71s 711ms/step - loss: 0.0912 - acc: 0.9730 - val_loss: 0.7433 - val_acc: 0.7470
Epoch 29/100
100/100 [==============================] - 71s 713ms/step - loss: 0.0846 - acc: 0.9750 - val_loss: 0.8113 - val_acc: 0.7410
Epoch 30/100
100/100 [==============================] - 72s 716ms/step - loss: 0.0740 - acc: 0.9725 - val_loss: 0.7621 - val_acc: 0.7400
Epoch 31/100
 99/100 [============================>.] - ETA: 0s - loss: 0.0567 - acc: 0.9848
Reached target accuracy so cancelling training!
100/100 [==============================] - 72s 724ms/step - loss: 0.0575 - acc: 0.9840 - val_loss: 0.9369 - val_acc: 0.7250
```

Plot the training progress

In [15]:
```
plot_progress(history_no_aug, 'Without Data Augmentation')
```



The Training Accuracy is close to 100%, and the validation accuracy is in the 70%-80% range. This is a great example of overfitting -- which in short means that it can do very well with images it has seen before, but not so well with images it hasn't. One simple method to avoid overfitting is to augment the images a bit. If you think about it, most pictures of a cat are very similar -- the ears are at the top, then the eyes, then the mouth etc. Things like the distance between the eyes and ears will always be quite similar too.

What if we tweak the images -- rotate the image, squash it, etc. That's what image augementation is all about.

4.2 With Data Augmentation

We will train our model with the `binary_crossentropy` loss. We will use the `RMSprop` optimizer with a learning rate of `0.0001`. During training, we will want to monitor classification accuracy.

In [16]:
```
model_aug = clone_model(model)
model_aug.compile(loss = 'binary_crossentropy',
                  optimizer = RMSprop(lr = 1e-4),
                  metrics = ['acc'])
```

In [17]:
```
history_aug = model_aug.fit_generator(
      train_generator_aug,
      steps_per_epoch = 100,   # number of images = batch_size * steps
      epochs = 100,
      verbose = 1,
      validation_data = dev_generator,
      validation_steps = 50,  # number of images = batch_size * steps
      callbacks = [callbacks])
```

```
Epoch 1/100
100/100 [==============================] - 89s 888ms/step - loss: 0.6914 - acc: 0.5275 - val_loss: 0.6764 - val_acc: 0.5630
Epoch 2/100
100/100 [==============================] - 79s 785ms/step - loss: 0.6772 - acc: 0.5685 - val_loss: 0.6490 - val_acc: 0.6240
Epoch 3/100
100/100 [==============================] - 85s 855ms/step - loss: 0.6586 - acc: 0.5875 - val_loss: 0.6392 - val_acc: 0.6030
Epoch 4/100
100/100 [==============================] - 75s 755ms/step - loss: 0.6440 - acc: 0.6245 - val_loss: 0.6100 - val_acc: 0.6500
Epoch 5/100
100/100 [==============================] - 87s 870ms/step - loss: 0.6277 - acc: 0.6470 - val_loss: 0.5881 - val_acc: 0.6940
Epoch 6/100
100/100 [==============================] - 75s 752ms/step - loss: 0.6185 - acc: 0.6455 - val_loss: 0.6122 - val_acc: 0.6390
Epoch 7/100
100/100 [==============================] - 85s 849ms/step - loss: 0.5993 - acc: 0.6695 - val_loss: 0.5740 - val_acc: 0.6920
Epoch 8/100
100/100 [==============================] - 75s 752ms/step - loss: 0.5849 - acc: 0.6955 - val_loss: 0.5594 - val_acc: 0.7030
Epoch 9/100
100/100 [==============================] - 84s 844ms/step - loss: 0.5877 - acc: 0.6865 - val_loss: 0.5527 - val_acc: 0.7090
Epoch 10/100
100/100 [==============================] - 76s 756ms/step - loss: 0.5757 - acc: 0.6920 - val_loss: 0.5386 - val_acc: 0.7290
Epoch 11/100
100/100 [==============================] - 86s 863ms/step - loss: 0.5705 - acc: 0.6870 - val_loss: 0.5414 - val_acc: 0.7260
Epoch 12/100
100/100 [==============================] - 84s 841ms/step - loss: 0.5760 - acc: 0.7010 - val_loss: 0.5345 - val_acc: 0.7270
Epoch 13/100
100/100 [==============================] - 74s 740ms/step - loss: 0.5644 - acc: 0.7025 - val_loss: 0.5195 - val_acc: 0.7230
Epoch 14/100
100/100 [==============================] - 85s 848ms/step - loss: 0.5528 - acc: 0.7080 - val_loss: 0.5320 - val_acc: 0.7170
Epoch 15/100
100/100 [==============================] - 74s 739ms/step - loss: 0.5449 - acc: 0.7240 - val_loss: 0.5705 - val_acc: 0.7150
Epoch 16/100
100/100 [==============================] - 84s 841ms/step - loss: 0.5421 - acc: 0.7260 - val_loss: 0.5079 - val_acc: 0.7430
Epoch 17/100
100/100 [==============================] - 76s 756ms/step - loss: 0.5294 - acc: 0.7365 - val_loss: 0.5548 - val_acc: 0.7080
Epoch 18/100
100/100 [==============================] - 88s 881ms/step - loss: 0.5334 - acc: 0.7280 - val_loss: 0.5281 - val_acc: 0.7350
Epoch 19/100
100/100 [==============================] - 73s 731ms/step - loss: 0.5472 - acc: 0.7205 - val_loss: 0.5066 - val_acc: 0.7340
Epoch 20/100
100/100 [==============================] - 86s 860ms/step - loss: 0.5360 - acc: 0.7290 - val_loss: 0.5218 - val_acc: 0.7230
Epoch 21/100
100/100 [==============================] - 86s 855ms/step - loss: 0.5167 - acc: 0.7360 - val_loss: 0.4952 - val_acc: 0.7490
Epoch 22/100
100/100 [==============================] - 76s 757ms/step - loss: 0.5204 - acc: 0.7465 - val_loss: 0.4942 - val_acc: 0.7510
Epoch 23/100
```

```
100/100 [==============================] - 89s 888ms/step - loss: 0.5207 - acc: 0.7440 - val_loss: 0.5208 - val_acc: 0.7390
Epoch 24/100
100/100 [==============================] - 77s 766ms/step - loss: 0.5207 - acc: 0.7385 - val_loss: 0.4940 - val_acc: 0.7670
Epoch 25/100
100/100 [==============================] - 87s 868ms/step - loss: 0.5196 - acc: 0.7410 - val_loss: 0.4863 - val_acc: 0.7520
Epoch 26/100
100/100 [==============================] - 74s 741ms/step - loss: 0.5046 - acc: 0.7490 - val_loss: 0.4730 - val_acc: 0.7670
Epoch 27/100
100/100 [==============================] - 87s 866ms/step - loss: 0.4996 - acc: 0.7605 - val_loss: 0.4878 - val_acc: 0.7580
Epoch 28/100
100/100 [==============================] - 84s 844ms/step - loss: 0.5027 - acc: 0.7540 - val_loss: 0.4820 - val_acc: 0.7650
Epoch 29/100
100/100 [==============================] - 77s 770ms/step - loss: 0.5019 - acc: 0.7505 - val_loss: 0.5178 - val_acc: 0.7300
Epoch 30/100
100/100 [==============================] - 89s 891ms/step - loss: 0.4889 - acc: 0.7620 - val_loss: 0.5133 - val_acc: 0.7460
Epoch 31/100
100/100 [==============================] - 78s 781ms/step - loss: 0.4929 - acc: 0.7615 - val_loss: 0.4646 - val_acc: 0.7750
Epoch 32/100
100/100 [==============================] - 88s 879ms/step - loss: 0.4896 - acc: 0.7640 - val_loss: 0.4780 - val_acc: 0.7550
Epoch 33/100
100/100 [==============================] - 78s 781ms/step - loss: 0.4860 - acc: 0.7495 - val_loss: 0.4624 - val_acc: 0.7810
Epoch 34/100
100/100 [==============================] - 91s 911ms/step - loss: 0.4728 - acc: 0.7650 - val_loss: 0.5071 - val_acc: 0.7390
Epoch 35/100
100/100 [==============================] - 77s 769ms/step - loss: 0.4861 - acc: 0.7700 - val_loss: 0.4607 - val_acc: 0.7770
Epoch 36/100
100/100 [==============================] - 88s 882ms/step - loss: 0.4785 - acc: 0.7745 - val_loss: 0.4630 - val_acc: 0.7690
Epoch 37/100
100/100 [==============================] - 78s 775ms/step - loss: 0.4838 - acc: 0.7670 - val_loss: 0.4865 - val_acc: 0.7640
Epoch 38/100
100/100 [==============================] - 89s 890ms/step - loss: 0.4780 - acc: 0.7735 - val_loss: 0.4859 - val_acc: 0.7560
Epoch 39/100
100/100 [==============================] - 76s 764ms/step - loss: 0.4565 - acc: 0.7835 - val_loss: 0.4457 - val_acc: 0.7840
Epoch 40/100
100/100 [==============================] - 87s 866ms/step - loss: 0.4717 - acc: 0.7670 - val_loss: 0.4871 - val_acc: 0.7500
Epoch 41/100
100/100 [==============================] - 79s 786ms/step - loss: 0.4737 - acc: 0.7705 - val_loss: 0.5786 - val_acc: 0.7010
Epoch 42/100
100/100 [==============================] - 89s 893ms/step - loss: 0.4649 - acc: 0.7785 - val_loss: 0.4552 - val_acc: 0.7810
Epoch 43/100
100/100 [==============================] - 77s 773ms/step - loss: 0.4559 - acc: 0.7940 - val_loss: 0.4659 - val_acc: 0.7770
Epoch 44/100
100/100 [==============================] - 90s 897ms/step - loss: 0.4404 - acc: 0.7865 - val_loss: 0.4515 - val_acc: 0.7880
Epoch 45/100
100/100 [==============================] - 78s 779ms/step - loss: 0.4529 - acc: 0.7850 - val_loss: 0.4601 - val_acc: 0.7760
```

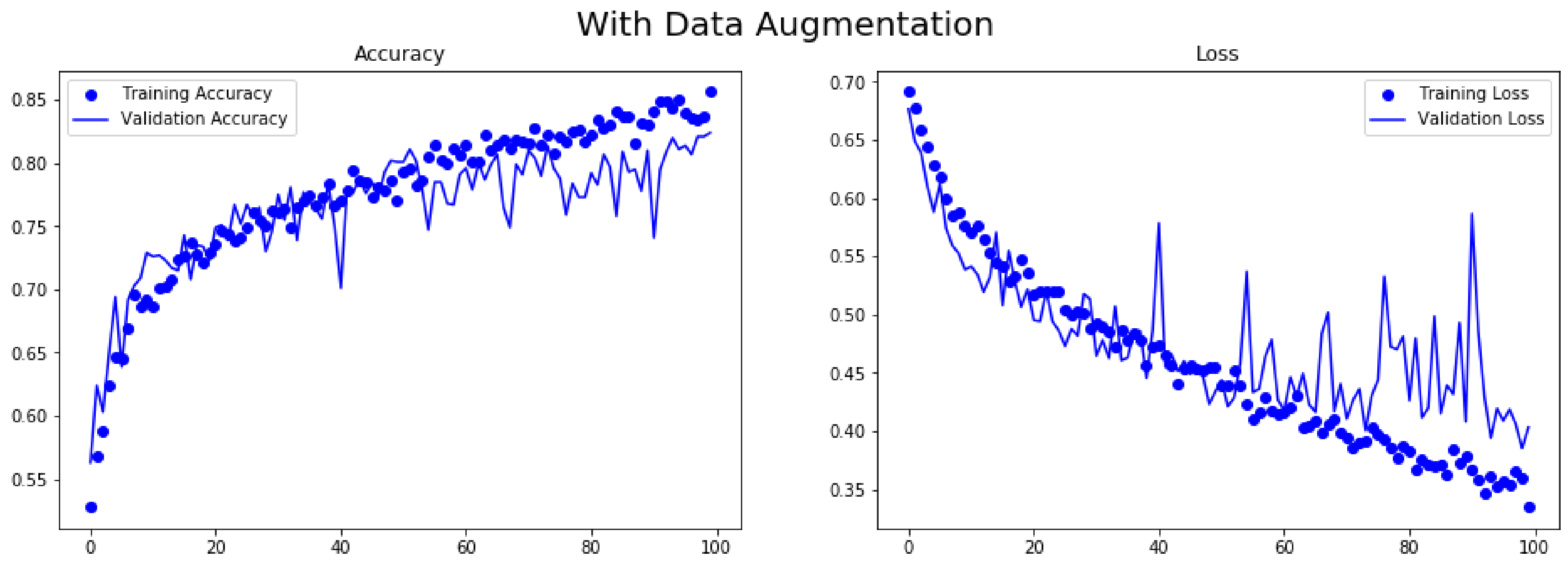
```
Epoch 46/100
100/100 [==============================] - 86s 860ms/step - loss: 0.4563 - acc: 0.7730 - val_loss: 0.4488 - val_acc: 0.7850
Epoch 47/100
100/100 [==============================] - 77s 771ms/step - loss: 0.4534 - acc: 0.7815 - val_loss: 0.4558 - val_acc: 0.7740
Epoch 48/100
100/100 [==============================] - 90s 897ms/step - loss: 0.4525 - acc: 0.7785 - val_loss: 0.4482 - val_acc: 0.7930
Epoch 49/100
100/100 [==============================] - 87s 875ms/step - loss: 0.4554 - acc: 0.7860 - val_loss: 0.4232 - val_acc: 0.8020
Epoch 50/100
100/100 [==============================] - 78s 783ms/step - loss: 0.4553 - acc: 0.7705 - val_loss: 0.4339 - val_acc: 0.8010
Epoch 51/100
100/100 [==============================] - 89s 891ms/step - loss: 0.4395 - acc: 0.7925 - val_loss: 0.4437 - val_acc: 0.8010
Epoch 52/100
100/100 [==============================] - 87s 868ms/step - loss: 0.4394 - acc: 0.7960 - val_loss: 0.4211 - val_acc: 0.8110
Epoch 53/100
100/100 [==============================] - 76s 760ms/step - loss: 0.4525 - acc: 0.7830 - val_loss: 0.4287 - val_acc: 0.8020
Epoch 54/100
100/100 [==============================] - 88s 880ms/step - loss: 0.4391 - acc: 0.7870 - val_loss: 0.4581 - val_acc: 0.7790
Epoch 55/100
100/100 [==============================] - 80s 796ms/step - loss: 0.4238 - acc: 0.8045 - val_loss: 0.5370 - val_acc: 0.7470
Epoch 56/100
100/100 [==============================] - 88s 878ms/step - loss: 0.4096 - acc: 0.8145 - val_loss: 0.4333 - val_acc: 0.7850
Epoch 57/100
100/100 [==============================] - 78s 783ms/step - loss: 0.4161 - acc: 0.8025 - val_loss: 0.4361 - val_acc: 0.7850
Epoch 58/100
100/100 [==============================] - 93s 931ms/step - loss: 0.4291 - acc: 0.8000 - val_loss: 0.4642 - val_acc: 0.7680
Epoch 59/100
100/100 [==============================] - 76s 758ms/step - loss: 0.4180 - acc: 0.8120 - val_loss: 0.4786 - val_acc: 0.7670
Epoch 60/100
100/100 [==============================] - 93s 925ms/step - loss: 0.4151 - acc: 0.8065 - val_loss: 0.4269 - val_acc: 0.7910
Epoch 61/100
100/100 [==============================] - 76s 761ms/step - loss: 0.4153 - acc: 0.8150 - val_loss: 0.4175 - val_acc: 0.7960
Epoch 62/100
100/100 [==============================] - 87s 871ms/step - loss: 0.4202 - acc: 0.8005 - val_loss: 0.4461 - val_acc: 0.7790
Epoch 63/100
100/100 [==============================] - 79s 788ms/step - loss: 0.4306 - acc: 0.8015 - val_loss: 0.4294 - val_acc: 0.8000
Epoch 64/100
100/100 [==============================] - 87s 867ms/step - loss: 0.4028 - acc: 0.8225 - val_loss: 0.4496 - val_acc: 0.7870
Epoch 65/100
100/100 [==============================] - 76s 762ms/step - loss: 0.4047 - acc: 0.8110 - val_loss: 0.4228 - val_acc: 0.7990
Epoch 66/100
100/100 [==============================] - 90s 896ms/step - loss: 0.4087 - acc: 0.8140 - val_loss: 0.4161 - val_acc: 0.8070
Epoch 67/100
100/100 [==============================] - 79s 787ms/step - loss: 0.3981 - acc: 0.8180 - val_loss: 0.4838 - val_acc: 0.7650
Epoch 68/100
```

```
100/100 [==============================] - 90s 900ms/step - loss: 0.4061 - acc: 0.8115 - val_loss: 0.5017 - val_acc: 0.7490
Epoch 69/100
100/100 [==============================] - 76s 762ms/step - loss: 0.4096 - acc: 0.8190 - val_loss: 0.4168 - val_acc: 0.7990
Epoch 70/100
100/100 [==============================] - 87s 869ms/step - loss: 0.3989 - acc: 0.8170 - val_loss: 0.4407 - val_acc: 0.7910
Epoch 71/100
100/100 [==============================] - 76s 761ms/step - loss: 0.3937 - acc: 0.8160 - val_loss: 0.4106 - val_acc: 0.8100
Epoch 72/100
100/100 [==============================] - 90s 905ms/step - loss: 0.3853 - acc: 0.8275 - val_loss: 0.4268 - val_acc: 0.8040
Epoch 73/100
100/100 [==============================] - 79s 791ms/step - loss: 0.3898 - acc: 0.8145 - val_loss: 0.4361 - val_acc: 0.7900
Epoch 74/100
100/100 [==============================] - 89s 894ms/step - loss: 0.3914 - acc: 0.8230 - val_loss: 0.4005 - val_acc: 0.8140
Epoch 75/100
100/100 [==============================] - 79s 788ms/step - loss: 0.4037 - acc: 0.8080 - val_loss: 0.4308 - val_acc: 0.7960
Epoch 76/100
100/100 [==============================] - 88s 877ms/step - loss: 0.3974 - acc: 0.8215 - val_loss: 0.4439 - val_acc: 0.7880
Epoch 77/100
100/100 [==============================] - 79s 789ms/step - loss: 0.3926 - acc: 0.8170 - val_loss: 0.5327 - val_acc: 0.7590
Epoch 78/100
100/100 [==============================] - 89s 886ms/step - loss: 0.3858 - acc: 0.8245 - val_loss: 0.4724 - val_acc: 0.7840
Epoch 79/100
100/100 [==============================] - 76s 760ms/step - loss: 0.3775 - acc: 0.8265 - val_loss: 0.4702 - val_acc: 0.7730
Epoch 80/100
100/100 [==============================] - 89s 888ms/step - loss: 0.3870 - acc: 0.8165 - val_loss: 0.4815 - val_acc: 0.7730
Epoch 81/100
100/100 [==============================] - 74s 740ms/step - loss: 0.3831 - acc: 0.8225 - val_loss: 0.4262 - val_acc: 0.7920
Epoch 82/100
100/100 [==============================] - 85s 854ms/step - loss: 0.3674 - acc: 0.8350 - val_loss: 0.4797 - val_acc: 0.7830
Epoch 83/100
100/100 [==============================] - 75s 752ms/step - loss: 0.3760 - acc: 0.8275 - val_loss: 0.4116 - val_acc: 0.8070
Epoch 84/100
100/100 [==============================] - 87s 874ms/step - loss: 0.3708 - acc: 0.8310 - val_loss: 0.4198 - val_acc: 0.7970
Epoch 85/100
100/100 [==============================] - 73s 735ms/step - loss: 0.3694 - acc: 0.8415 - val_loss: 0.4987 - val_acc: 0.7580
Epoch 86/100
100/100 [==============================] - 89s 893ms/step - loss: 0.3709 - acc: 0.8370 - val_loss: 0.4151 - val_acc: 0.8090
Epoch 87/100
100/100 [==============================] - 79s 788ms/step - loss: 0.3624 - acc: 0.8370 - val_loss: 0.4392 - val_acc: 0.7930
Epoch 88/100
100/100 [==============================] - 87s 869ms/step - loss: 0.3844 - acc: 0.8160 - val_loss: 0.4319 - val_acc: 0.7950
Epoch 89/100
100/100 [==============================] - 79s 792ms/step - loss: 0.3730 - acc: 0.8320 - val_loss: 0.4929 - val_acc: 0.7780
Epoch 90/100
100/100 [==============================] - 89s 886ms/step - loss: 0.3786 - acc: 0.8300 - val_loss: 0.4080 - val_acc: 0.8100
```

```
Epoch 91/100
100/100 [==============================] - 75s 751ms/step - loss: 0.3668 - acc: 0.8415 - val_loss: 0.5867 - val_acc: 0.7410
Epoch 92/100
100/100 [==============================] - 87s 867ms/step - loss: 0.3580 - acc: 0.8485 - val_loss: 0.4861 - val_acc: 0.7950
Epoch 93/100
100/100 [==============================] - 74s 744ms/step - loss: 0.3474 - acc: 0.8485 - val_loss: 0.4291 - val_acc: 0.8090
Epoch 94/100
100/100 [==============================] - 92s 924ms/step - loss: 0.3607 - acc: 0.8430 - val_loss: 0.3942 - val_acc: 0.8200
Epoch 95/100
100/100 [==============================] - 74s 743ms/step - loss: 0.3526 - acc: 0.8500 - val_loss: 0.4192 - val_acc: 0.8110
Epoch 96/100
100/100 [==============================] - 92s 922ms/step - loss: 0.3564 - acc: 0.8390 - val_loss: 0.4088 - val_acc: 0.8140
Epoch 97/100
100/100 [==============================] - 87s 866ms/step - loss: 0.3541 - acc: 0.8355 - val_loss: 0.4187 - val_acc: 0.8070
Epoch 98/100
100/100 [==============================] - 87s 869ms/step - loss: 0.3651 - acc: 0.8350 - val_loss: 0.4058 - val_acc: 0.8210
Epoch 99/100
100/100 [==============================] - 73s 730ms/step - loss: 0.3602 - acc: 0.8370 - val_loss: 0.3852 - val_acc: 0.8210
Epoch 100/100
100/100 [==============================] - 89s 886ms/step - loss: 0.3345 - acc: 0.8565 - val_loss: 0.4033 - val_acc: 0.8240
```

Plot the training progress

In [18]:
```
plot_progress(history_aug, 'With Data Augmentation')
```



Compared with the model trained without data augmentation, we can see that overfitting has been resolved.

5. Model Testing

Let's now take a look at actually running a prediction using the model. This code will allow us to choose 1 or more files from our file system, it will then upload them, and run them through the model, giving an indication of whether the object is a cat or a dog.

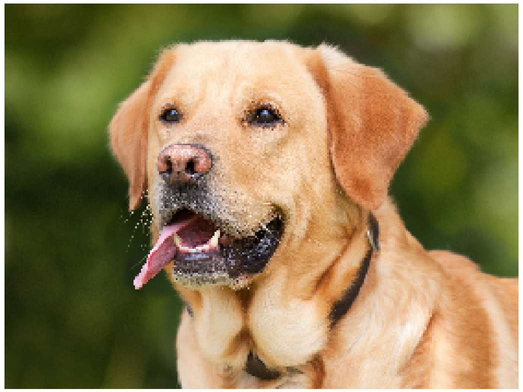
Our images are downloaded from [https://pixabay.com/ (https://pixabay.com/)](https://pixabay.com/)

In [19]:
```python
model_test = model_aug # choose which model to use
test_path = 'test/' # the folder that saves the test images
test_files = [os.path.join(test_path, fname) for fname in os.listdir(test_path)]

for file in test_files:
    img = image.load_img(file, target_size = (150, 150))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis = 0)
    x /= 255

    # predict
    classes = model_test.predict(x)

    # show the figure with prediction
    image_cur = plt.imread(file)
    plt.figure(figsize = (5, 5))
    plt.imshow(image_cur)
    plt.axis('off')
    if classes[0] > 0.5:
        plt.title(file.split('/')[-1] + ": dog")
    else:
        plt.title(file.split('/')[-1] + ": cat")
```

cat-1.jpg: cat

cat-2.jpg: cat

dog-1.jpg: dog



dog-2.jpg: dog

6. Save Model

In [20]:
```
model_aug.save('my_model.h5')
```