lab11_2019

November 1, 2019

1 Convolutional Neural Networks for classification

– Prof. Dorien Herremans

We will be demonstrating the use of CNNs for image classification. This tutorial will use the Keras library, built upon Tensorflow version 1.x

Since CNNs can benefit from using GPU power, please change your runtime to GPU. It will make things to 10x as fast. You can do it using the Runtime menu in Google Colab.

In [2]:
```
# If Keras is not installed, please do so with the command below:
!pip install keras

# This notebook is built around using tensorflow 1.0 as the backend for keras
!KERAS_BACKEND=tensorflow python -c "from tensorflow.keras import backend"
```

```
Requirement already satisfied: keras in /usr/local/lib/python3.6/dist-packages (2.2.5)
Requirement already satisfied: keras-preprocessing>=1.1.0 in /usr/local/lib/python3.6/dist-pack
Requirement already satisfied: pyyaml in /usr/local/lib/python3.6/dist-packages (from keras) (3
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.6/dist-packages (from keras
Requirement already satisfied: keras-applications>=1.0.8 in /usr/local/lib/python3.6/dist-packa
Requirement already satisfied: numpy>=1.9.1 in /usr/local/lib/python3.6/dist-packages (from ker
Requirement already satisfied: scipy>=0.14 in /usr/local/lib/python3.6/dist-packages (from kera
Requirement already satisfied: h5py in /usr/local/lib/python3.6/dist-packages (from keras) (2.8
```

We will be performing a 2-class classification problem: classifying dogs versus cats from images. Our dataset is based on the Kaggle dataset https://www.kaggle.com/c/dogs-vs-cats/data. Do **not** use this dataset, however, as it is very big and will take too long to train.

**Please download the reduced dataset from dorienherremans.com/drop -> cds / cnns/ catsanddogs. This reduced dataset contains 1,000 training examples for each class, and 400 validation examples for each class.

In summary, this is our directory structure:

```
data/
    train/
        dogs/
            dog001.jpg
            dog002.jpg
            ...
```

```
        cats/
            cat001.jpg
            cat002.jpg
            ...
    validation/
        dogs/
            dog001.jpg
            dog002.jpg
            ...
        cats/
            cat001.jpg
            cat002.jpg
            ...
preview/
```

Notice how the labels of the images are, in fact, the folders. You can use any other types of labels/images to train this model and it will adapt accordingly...

Let's download the dataset:

```
In [3]: !wget -P / -c "https://dorienherremans.com/drop/CDS/CNNs/cat_dog.zip"
```

```
--2019-11-01 06:45:10--  https://dorienherremans.com/drop/CDS/CNNs/cat_dog.zip
Resolving dorienherremans.com (dorienherremans.com)... 96.127.180.74
Connecting to dorienherremans.com (dorienherremans.com)|96.127.180.74|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 64180857 (61M) [application/zip]
Saving to: cat_dog.zip

at_dog.zip          100%[===================>]  61.21M  17.8MB/s    in 4.5s

2019-11-01 06:45:15 (13.6 MB/s) - cat_dog.zip saved [64180857/64180857]
```

Unzip the file you just downloaded. This will extract everything in the folder structure described above.

```
In [0]: import zipfile
        zip_ref = zipfile.ZipFile('./cat_dog.zip', 'r')
        zip_ref.extractall('./')
        zip_ref.close()
```

Loading the necessary libraries for the lab:

```
In [5]: %tensorflow_version 1.x

        import os
        import numpy as np
        from keras.models import Sequential
```

```
from keras.layers import Activation, Dropout, Flatten, Dense
from keras.preprocessing.image import ImageDataGenerator
from keras.layers import Convolution2D, MaxPooling2D, ZeroPadding2D
from keras import optimizers
from keras import applications
from keras.models import Model
from keras.callbacks import History
```

```
Using TensorFlow backend.
```

Next, we'll store the training and validation data path in two variables. We'll also store the image resolution in two variables:

```
In [0]: img_width, img_height = 150, 150

        train_data_dir = 'data/train'
        validation_data_dir = 'data/validation'
```

2 Simple CNN

Let's preprocess the data before we feed them to the CNN. We will use 'generators' to feed batches of images to the network. All of these are rescaled to 150x150 and the pixel values are normalised to be between 0 and 1 (instead of 0 and 255).

```
In [7]: # rescale the pixel values from [0, 255] to [0, 1] interval
        datagen = ImageDataGenerator(rescale=1./255)
        batch_size = 32

        # automagically retrieve images and their classes for train and validation sets
        train_generator = datagen.flow_from_directory(
                train_data_dir,
                target_size=(img_width, img_height),
                batch_size=batch_size,
                class_mode='binary')

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

```
Found 2000 images belonging to 2 classes.
Found 800 images belonging to 2 classes.
```

2.0.1 Model architecture

Now we are ready to define our model architecture. We'll use a three layered convolutional network with ReLu units and pooling. On top of the three convolutional layers, we add two fully-connected layers.

```
In [8]: # a simple stack of 3 convolution layers with a ReLU activation and followed by max-po

        # in Keras, sequential model means a linear stack of layers. Which is what we are doin
        model = Sequential()

        # We add three convolution layers, each consisting of ReLu
        # units and pooling with a window of (2,2)
        #32 filters, kernel size (3,3)
        model.add(Convolution2D(32, (3, 3), input_shape=(img_width, img_height,3)))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))

        model.add(Convolution2D(32, (3, 3)))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))

        model.add(Convolution2D(64, (3, 3)))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))

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

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1 - keep_prob`.
```

In order to facilitate binary classification, we end the model with a single unit that uses sigmoid activation. Training will be done by minimising the binary_crossentropy loss and using an RmsProp optimizer. Alternatives include Adam optimizer and others, see: https://keras.io/optimizers/

```
In [9]: # specify training loss function
        model.compile(loss='binary_crossentropy',
                      optimizer='rmsprop',
                      metrics=['accuracy'])
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/optimizers.py:793: The nan

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow_core/python/ops/nn_im
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
```

2.0.2 Training

Let's train this simple model for a few epochs. I recommend as many epochs as your computer can handle, but put it to very few the first time you run. (It can be time consuming: about 3-60 seconds an epoch, so definitely use GPU!) We sample (randomly select) 2048 images from the dataset as training, and 832 as validation.

```
In [0]: epochs = 30
        train_samples = 2048
        validation_samples = 832
```

```
In [11]: history = History()  # this will allow us to plot the evolution of the validation loss
         model.fit_generator(
                 train_generator,
                 steps_per_epoch=train_samples // batch_size,
                 epochs=epochs,
                 callbacks=[history],  # save the history so that we can plot it later
                 validation_data=validation_generator,
                 validation_steps=validation_samples// batch_size,)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

Epoch 1/30
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

64/64 [==============================] - 16s 245ms/step - loss: 0.7290 - acc: 0.5103 - val_loss
Epoch 2/30
64/64 [==============================] - 9s 135ms/step - loss: 0.6932 - acc: 0.5308 - val_loss
Epoch 3/30
64/64 [==============================] - 9s 136ms/step - loss: 0.6592 - acc: 0.6245 - val_loss
Epoch 4/30
64/64 [==============================] - 9s 137ms/step - loss: 0.6128 - acc: 0.6646 - val_loss
Epoch 5/30
64/64 [==============================] - 9s 134ms/step - loss: 0.5827 - acc: 0.6909 - val_loss
Epoch 6/30
64/64 [==============================] - 9s 136ms/step - loss: 0.5515 - acc: 0.7207 - val_loss
Epoch 7/30
64/64 [==============================] - 9s 134ms/step - loss: 0.5144 - acc: 0.7505 - val_loss
Epoch 8/30
64/64 [==============================] - 9s 135ms/step - loss: 0.4694 - acc: 0.7822 - val_loss
Epoch 9/30
64/64 [==============================] - 9s 135ms/step - loss: 0.4415 - acc: 0.7964 - val_loss
Epoch 10/30
64/64 [==============================] - 9s 134ms/step - loss: 0.4176 - acc: 0.8188 - val_loss
Epoch 11/30
64/64 [==============================] - 9s 136ms/step - loss: 0.3830 - acc: 0.8198 - val_loss
Epoch 12/30
64/64 [==============================] - 9s 135ms/step - loss: 0.3464 - acc: 0.8428 - val_loss
Epoch 13/30
64/64 [==============================] - 9s 134ms/step - loss: 0.3135 - acc: 0.8711 - val_loss
Epoch 14/30
64/64 [==============================] - 9s 137ms/step - loss: 0.2799 - acc: 0.8755 - val_loss
Epoch 15/30
64/64 [==============================] - 9s 135ms/step - loss: 0.2651 - acc: 0.8804 - val_loss
Epoch 16/30
64/64 [==============================] - 9s 136ms/step - loss: 0.2241 - acc: 0.9077 - val_loss
Epoch 17/30
64/64 [==============================] - 9s 136ms/step - loss: 0.2075 - acc: 0.9121 - val_loss
Epoch 18/30
64/64 [==============================] - 9s 137ms/step - loss: 0.1790 - acc: 0.9229 - val_loss
Epoch 19/30
64/64 [==============================] - 9s 135ms/step - loss: 0.1689 - acc: 0.9351 - val_loss
Epoch 20/30
64/64 [==============================] - 9s 133ms/step - loss: 0.1357 - acc: 0.9482 - val_loss
Epoch 21/30
64/64 [==============================] - 9s 135ms/step - loss: 0.1222 - acc: 0.9517 - val_loss
Epoch 22/30
64/64 [==============================] - 9s 138ms/step - loss: 0.1213 - acc: 0.9561 - val_loss
```

```
Epoch 23/30
64/64 [==============================] - 9s 139ms/step - loss: 0.0947 - acc: 0.9653 - val_loss
Epoch 24/30
64/64 [==============================] - 9s 139ms/step - loss: 0.1078 - acc: 0.9580 - val_loss
Epoch 25/30
64/64 [==============================] - 9s 140ms/step - loss: 0.0948 - acc: 0.9614 - val_loss
Epoch 26/30
64/64 [==============================] - 9s 135ms/step - loss: 0.0877 - acc: 0.9653 - val_loss
Epoch 27/30
64/64 [==============================] - 9s 135ms/step - loss: 0.0964 - acc: 0.9624 - val_loss
Epoch 28/30
64/64 [==============================] - 9s 136ms/step - loss: 0.0961 - acc: 0.9707 - val_loss
Epoch 29/30
64/64 [==============================] - 9s 136ms/step - loss: 0.0846 - acc: 0.9668 - val_loss
Epoch 30/30
64/64 [==============================] - 9s 137ms/step - loss: 0.0736 - acc: 0.9722 - val_loss
```

```
Out[11]: <keras.callbacks.History at 0x7f61f4bfc9e8>
```

After this long a wait you will want to save your weights! This way you can reuse your model without training it again:

```
In [0]: model.save_weights('basic_cnn_30_epochs.h5')
```

And later, when you want to load the saved model you can just call it like this without having to train it again:

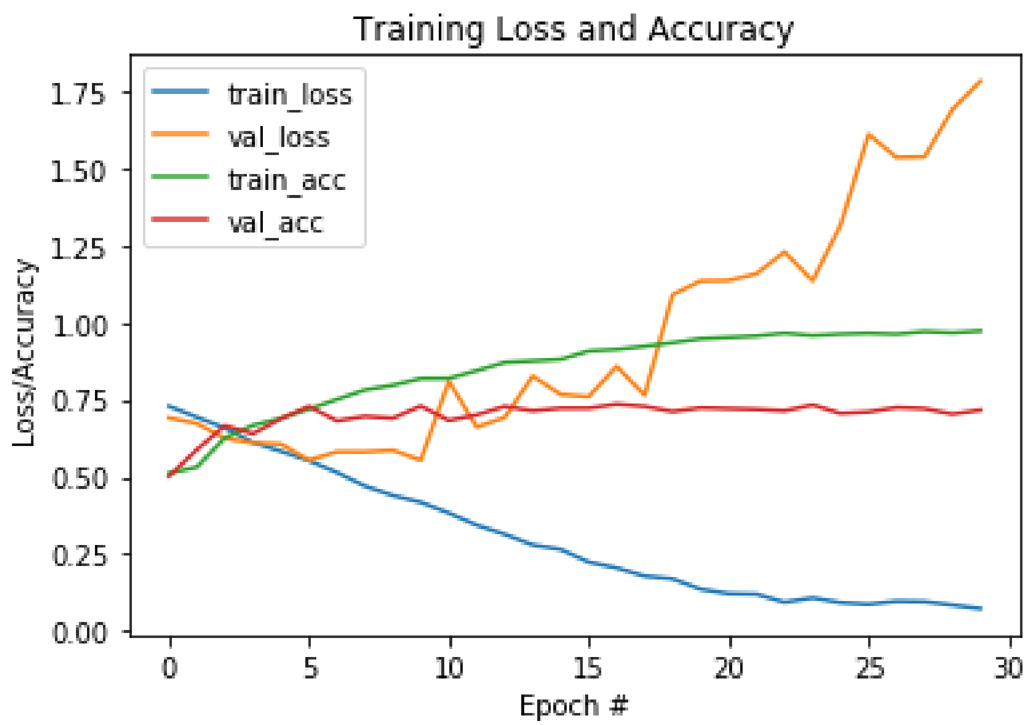
```
In [0]: #model.load_weights('basic_cnn_30_epochs.h5')
```

2.0.3 Evaluating the model during training

Now that we have trained our model, let's see how well it performs. Because we've added the argument callbacks=[history] to our generator object, the loss was saved at each step during training and we can retrieve it from a variable called history. Below we plot the different variables in the history object during training:

```
In [13]: import numpy as np
         import matplotlib.pyplot as plt

         # plot the training loss and accuracy
         def plotResults():
             plt.figure()
             N = epochs

             plt.plot(np.arange(0, N), history.history["loss"], label="train_loss")
             plt.plot(np.arange(0, N), history.history["val_loss"], label="val_loss")
             plt.plot(np.arange(0, N), history.history["acc"], label="train_acc")
             plt.plot(np.arange(0, N), history.history["val_acc"], label="val_acc")
```

```
    # make the graph understandable:
    plt.title("Training Loss and Accuracy")
    plt.xlabel("Epoch #")
    plt.ylabel("Loss/Accuracy")
    plt.legend(loc="upper left")
    plt.show()

plotResults()
```



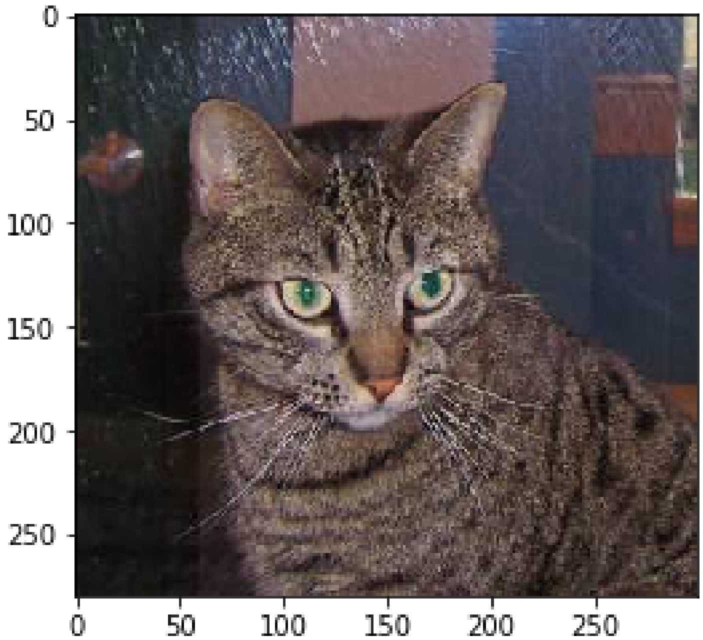
After ~10 epochs the neural network reaches ~75% accuracy. We can witness overfitting, as no progress is made on the validation set in the next epochs.

3 Augmented data model

Overfitting can possibly be remedies by augmenting our dataset so that our model becomes more robust. Let's take a random image of a cat again and display it:

```
In [15]: from keras.preprocessing.image import ImageDataGenerator, array_to_img, img_to_array,
         import matplotlib.pyplot as plt
         import matplotlib.image as mpimg

         img = load_img('./data/train/cats/cat.1.jpg')
```

```
# Show the image:
imgplot = plt.imshow(img)
plt.show()
```



She's cute and looks like my cat Sendai (who is sadly still in Belgium). Unlike Sendai, who likes posing for the camera, we only have one shot of this cat. So let's augment this image into multiple slightly different images...

Keras contains a preprocessing library. This provides us with a number of operations, including:

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

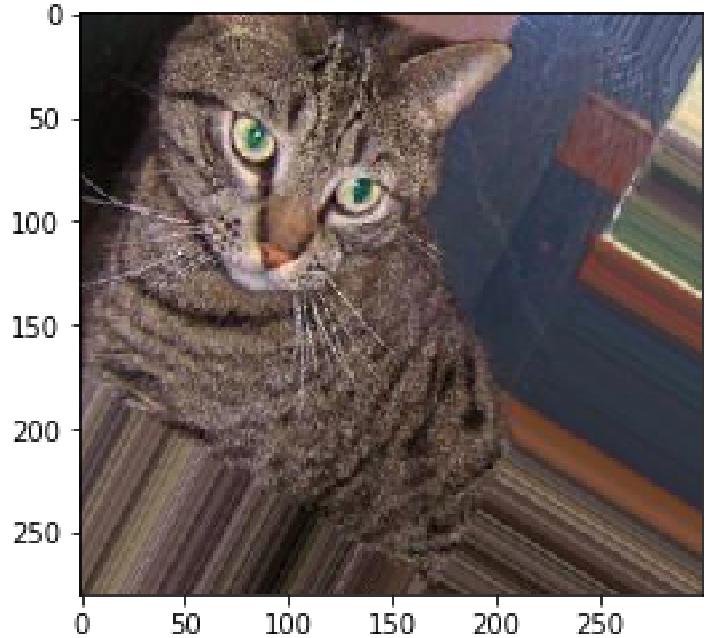
More options available in the documentation: http://keras.io/preprocessing/image/

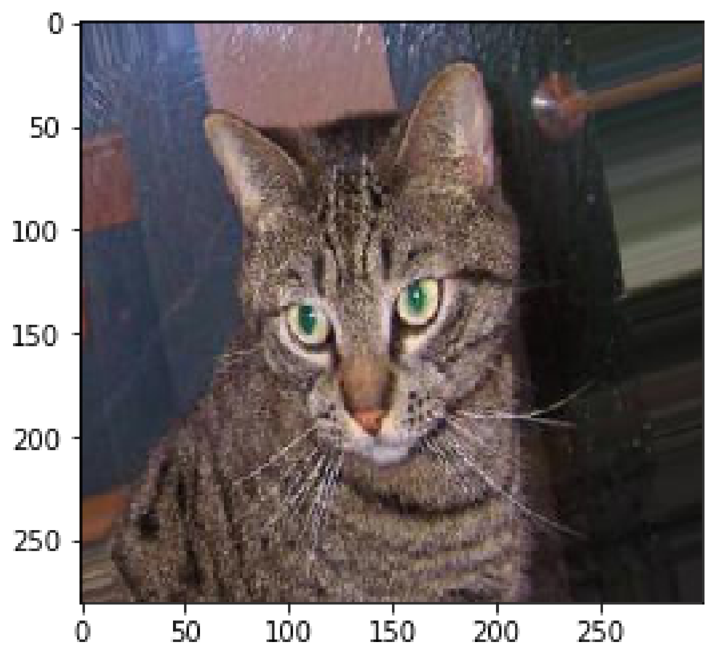
Let's test this out on the cat image:

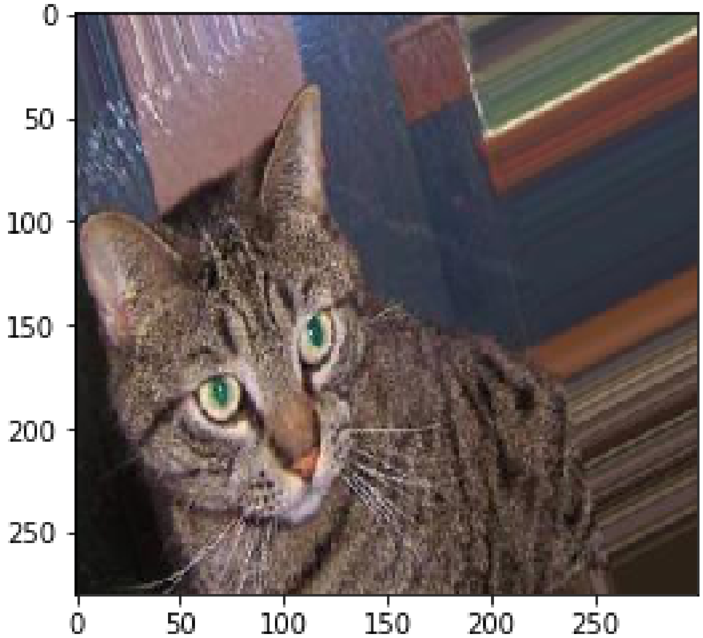
In [0]:
```
# change the characteristics to augment the data in different way if you want to explo
datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')

# load the original image
img = load_img('./data/train/cats/cat.1.jpg')

# reshape the image to a numpy array
x = img_to_array(img)  # this is a Numpy array with shape (3, 150, 150)
x = x.reshape((1,) + x.shape)  # this is a Numpy array with shape (1, 3, 150, 150)

# the .flow() command below generates batches of randomly transformed images
# and saves the results to the `preview/` directory
i = 0
# this will execute the flow function 20 times.
for batch in datagen.flow(x, batch_size=1, save_to_dir='preview', save_prefix='cat', sa
    i += 1
    if i > 19:
        break  # otherwise the generator would loop indefinitely
```

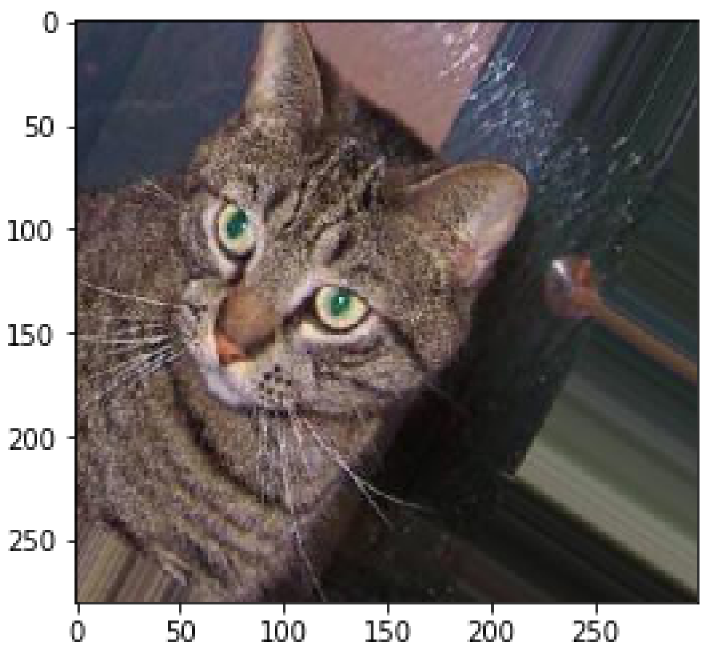
This has put the images in our preview/ folder. You can check in your filemanager or via:

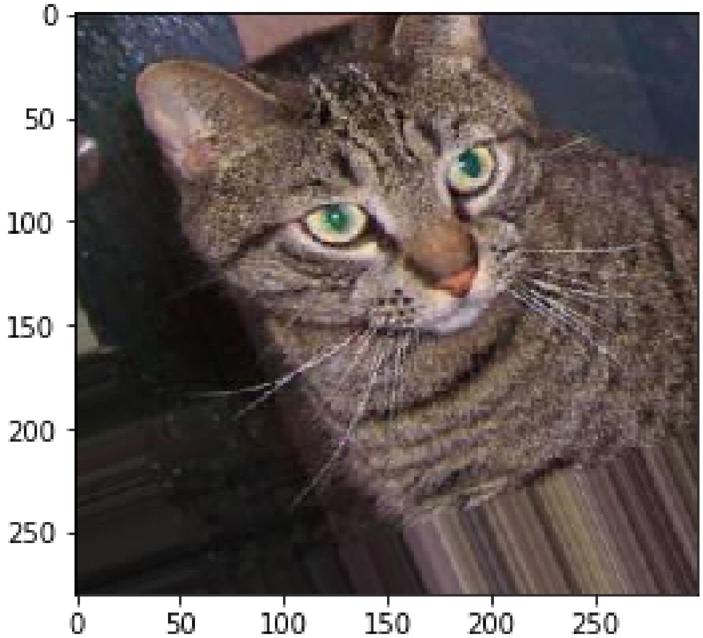
In [19]:
```
from PIL import Image
import glob
for filename in glob.glob('preview/*.jpg'):
    img=Image.open(filename)
    imgplot = plt.imshow(img)
    plt.show()
```

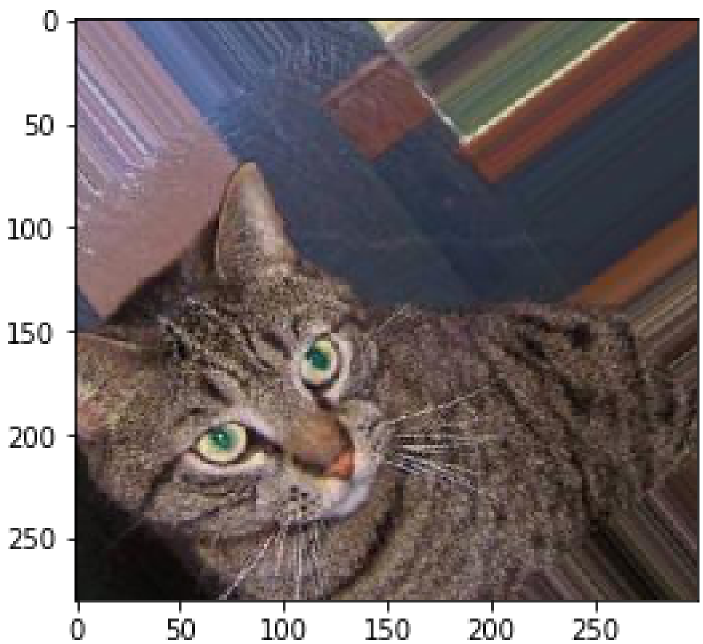

0
50
100
150
200
250
0
50
100
150
200
250

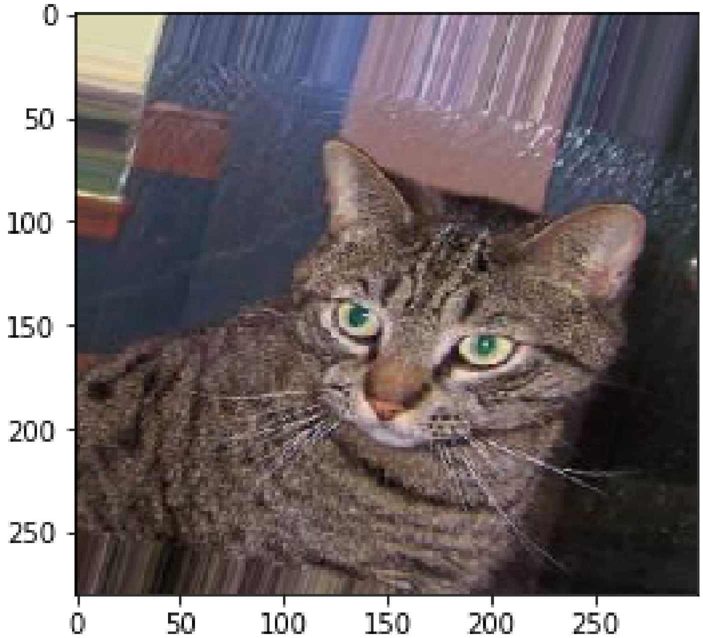

0
50
100
150
200
250
0
50
100
150
200
250

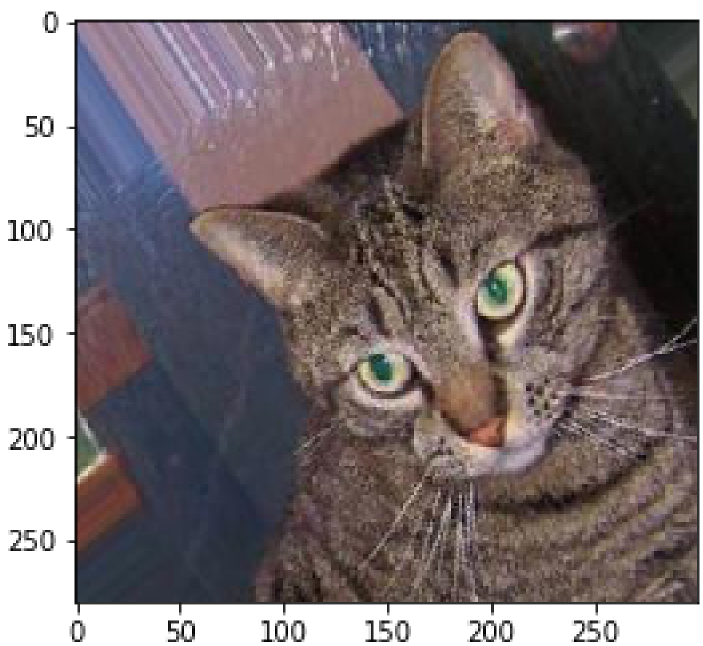

0
50
100
150
200
250
0
50
100
150
200
250

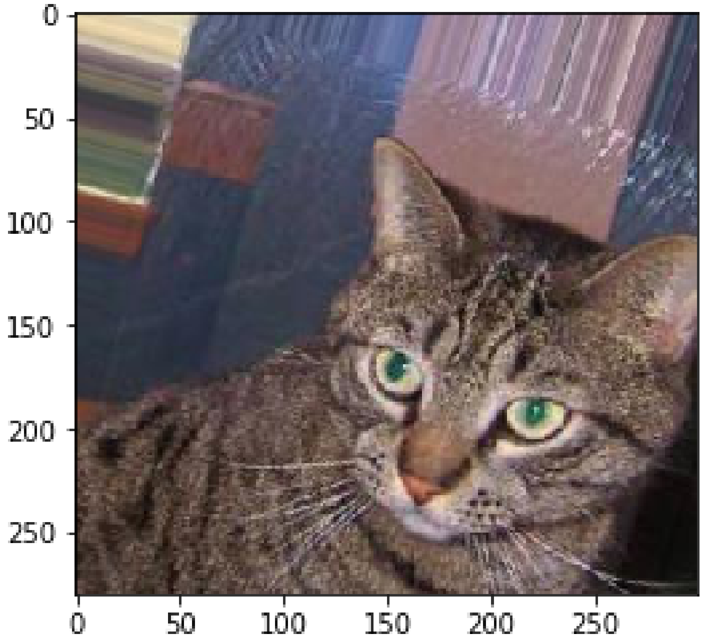

0
50
100
150
200
250
0
50
100
150
200
250

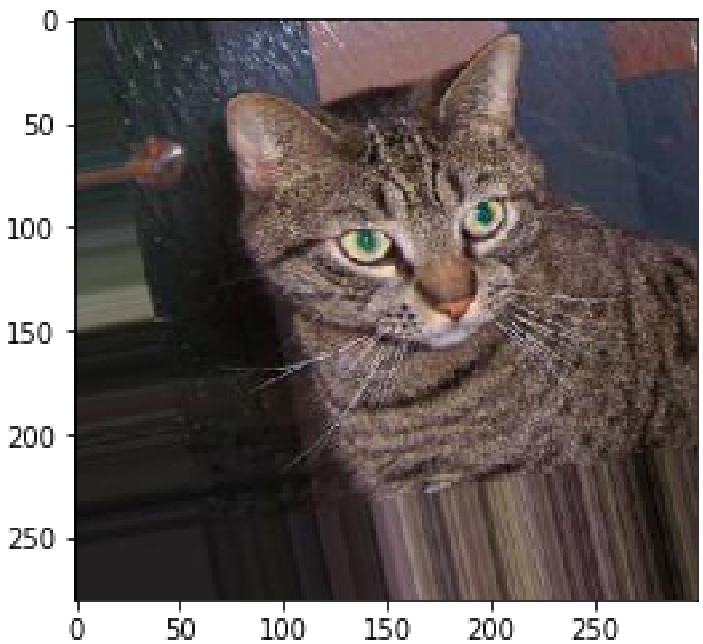

0
50
100
150
200
250
0
50
100
150
200
250

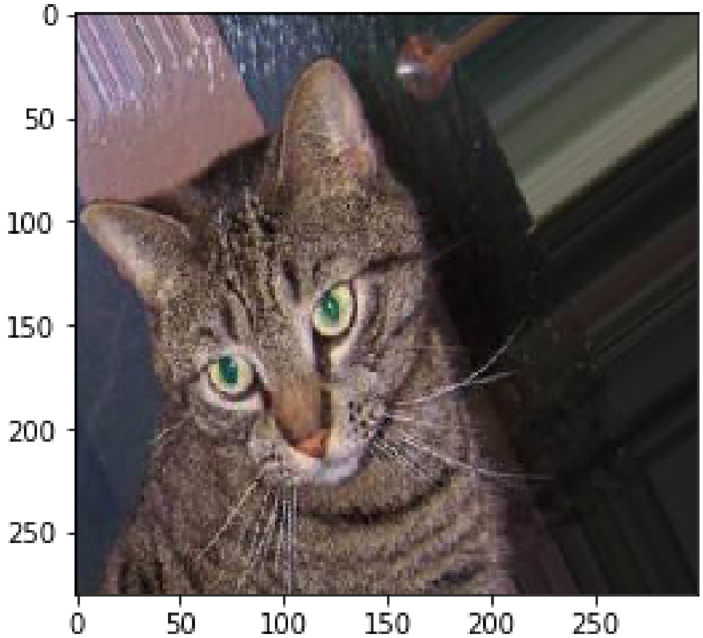

0
50
100
150
200
250
0
50
100
150
200
250

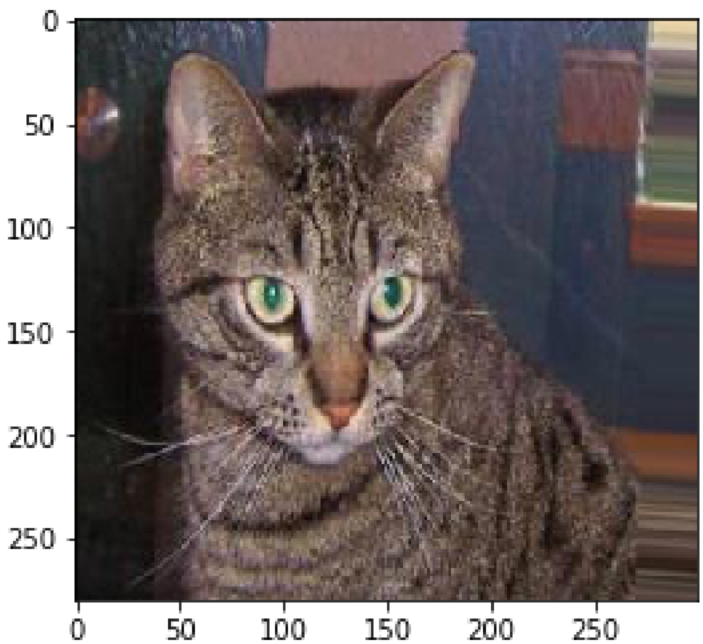

0
50
100
150
200
250
0
50
100
150
200
250

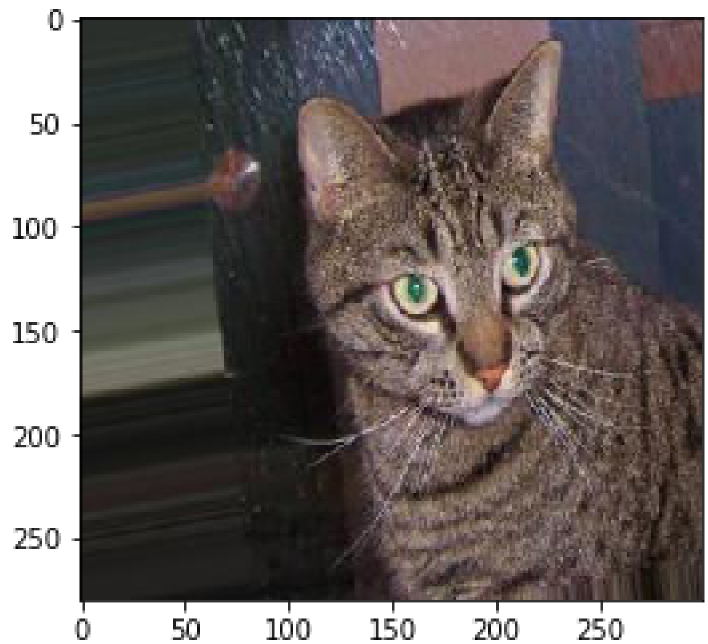

0
50
100
150
200
250
0
50
100
150
200
250

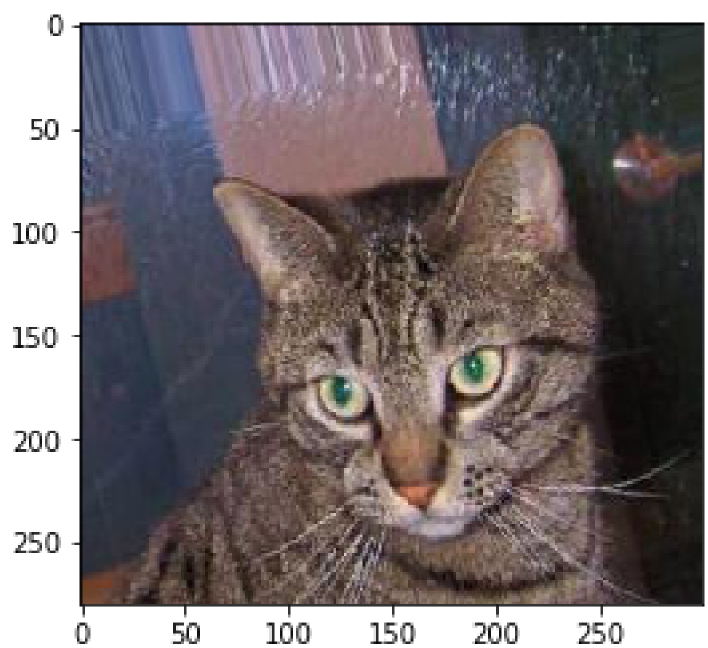

0
50
100
150
200
250
0
50
100
150
200
250

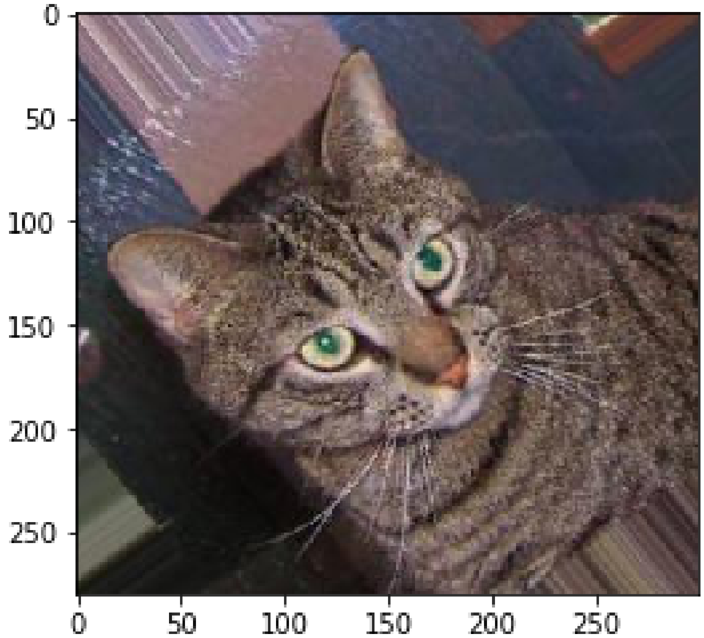

0
50
100
150
200
250
0
50
100
150
200
250

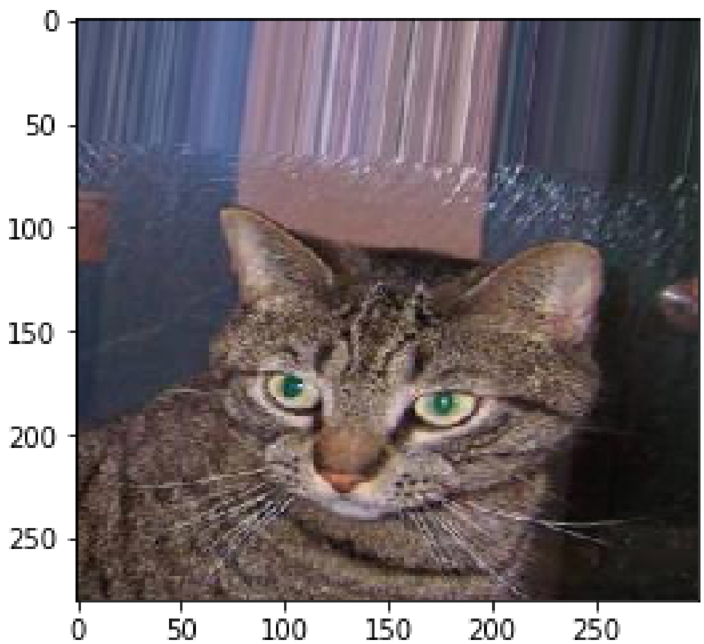

0
50
100
150
200
250
0
50
100
150
200
250

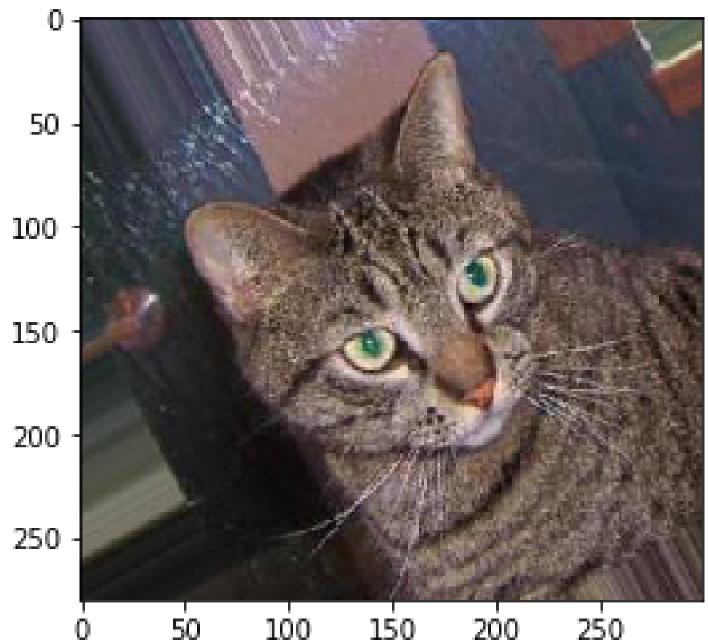

0
50
100
150
200
250
0
50
100
150
200
250

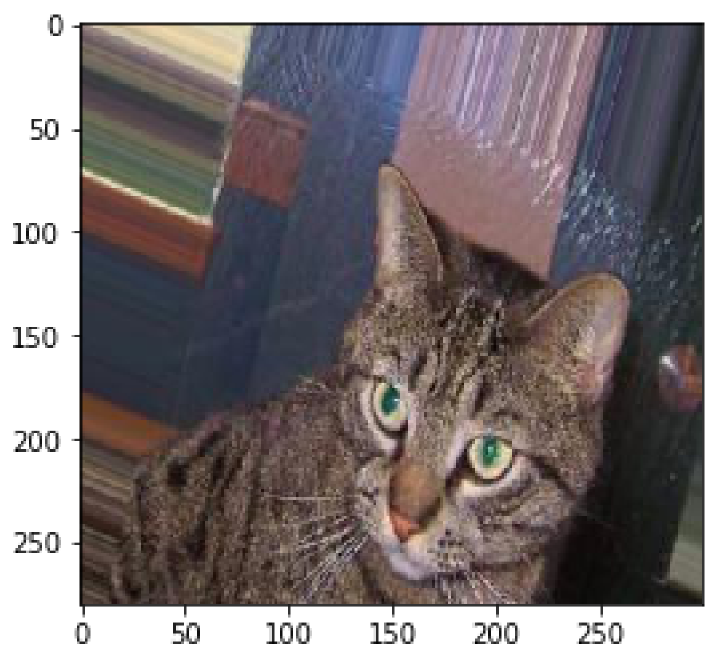

0
50
100
150
200
250
0
50
100
150
200
250

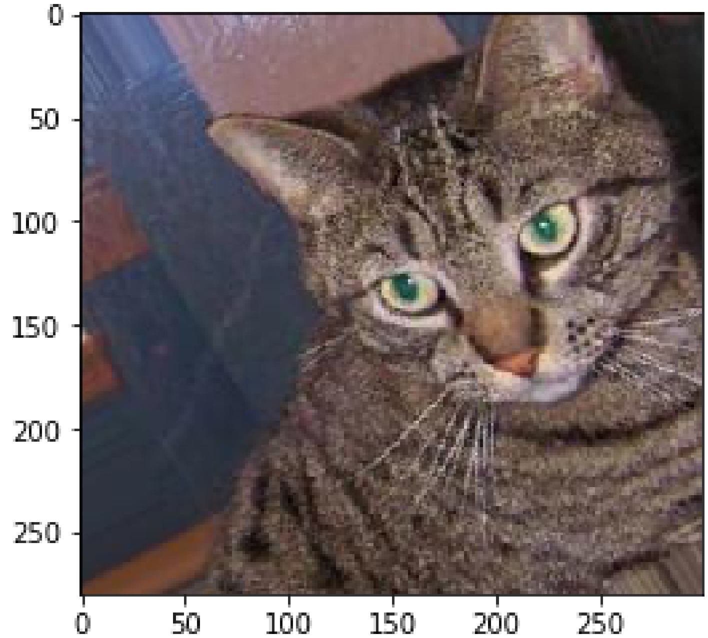

0
50
100
150
200
250
0
50
100
150
200
250

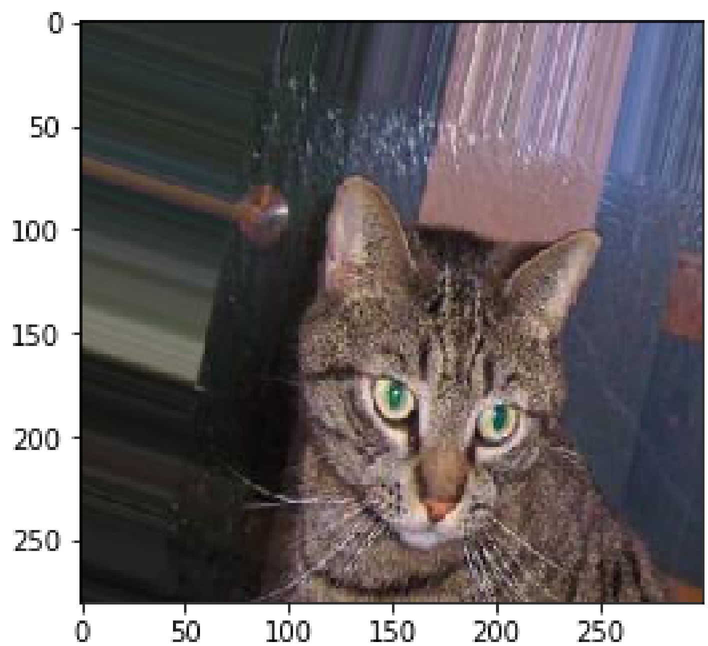

0
50
100
150
200
250
0
50
100
150
200
250


0
50
100
150
200
250
0
50
100
150
200
250


0
50
100
150
200
250
0
50
100
150
200
250


0
50
100
150
200
250
0
50
100
150
200
250





Great! Notice how the cat images are mirrored, skewed and generally deformed. This will allow our model to train more robustly.

Now it's time to fit the new data into the model in Keras. We will be downscaling all the images to 150x150 pixels so that they are all the same size. We will use the same 3 layered model with ReLu nodes and pooling that was previously defined.

Now we can integrate this in our previous data preprocessing as follows:

```
In [20]: # slightly adapted the variables for data augmentation:
         train_datagen_augmented = ImageDataGenerator(
                 rescale=1./255,        # normalize pixel values to [0,1]
                 shear_range=0.2,       # randomly applies shearing transformation
                 zoom_range=0.2,        # randomly applies shearing transformation
                 horizontal_flip=True)  # randomly flip the images

         # almost same code as before exept first line
         train_generator_augmented = train_datagen_augmented.flow_from_directory(
                 train_data_dir,
                 target_size=(img_width, img_height),
                 batch_size=batch_size,
                 class_mode='binary')
```

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

Let's train the model. Remember adjust the (previously set) batch size variable to make the training shorter or longer…

```
In [21]: history = History()  # this will allow us to plot the evolution of the validation los

         model.fit_generator(
                 train_generator_augmented,
                 steps_per_epoch=train_samples // batch_size,
                 epochs=epochs,
                 callbacks=[history],  # save the history so that we can plot it later
                 validation_data=validation_generator,
                 validation_steps=validation_samples // batch_size,)
```

```
Epoch 1/30
64/64 [==============================] - 20s 311ms/step - loss: 0.6146 - acc: 0.6904 - val_los
Epoch 2/30
64/64 [==============================] - 18s 284ms/step - loss: 0.5380 - acc: 0.7476 - val_los
Epoch 3/30
64/64 [==============================] - 18s 285ms/step - loss: 0.5163 - acc: 0.7729 - val_los
Epoch 4/30
64/64 [==============================] - 18s 281ms/step - loss: 0.5200 - acc: 0.7564 - val_los
Epoch 5/30
64/64 [==============================] - 18s 285ms/step - loss: 0.4774 - acc: 0.7886 - val_los
Epoch 6/30
64/64 [==============================] - 18s 282ms/step - loss: 0.4763 - acc: 0.7847 - val_los
Epoch 7/30
64/64 [==============================] - 18s 278ms/step - loss: 0.4757 - acc: 0.7940 - val_los
Epoch 8/30
64/64 [==============================] - 18s 282ms/step - loss: 0.4481 - acc: 0.7949 - val_los
Epoch 9/30
```

```
64/64 [==============================] - 18s 281ms/step - loss: 0.4525 - acc: 0.8110 - val_loss
Epoch 10/30
64/64 [==============================] - 18s 280ms/step - loss: 0.4445 - acc: 0.8037 - val_loss
Epoch 11/30
64/64 [==============================] - 18s 275ms/step - loss: 0.4355 - acc: 0.8145 - val_loss
Epoch 12/30
64/64 [==============================] - 18s 279ms/step - loss: 0.4197 - acc: 0.8027 - val_loss
Epoch 13/30
64/64 [==============================] - 18s 278ms/step - loss: 0.3951 - acc: 0.8315 - val_loss
Epoch 14/30
64/64 [==============================] - 18s 277ms/step - loss: 0.4081 - acc: 0.8125 - val_loss
Epoch 15/30
64/64 [==============================] - 18s 277ms/step - loss: 0.3855 - acc: 0.8354 - val_loss
Epoch 16/30
64/64 [==============================] - 18s 277ms/step - loss: 0.4051 - acc: 0.8257 - val_loss
Epoch 17/30
64/64 [==============================] - 17s 272ms/step - loss: 0.3975 - acc: 0.8252 - val_loss
Epoch 18/30
64/64 [==============================] - 18s 278ms/step - loss: 0.3876 - acc: 0.8321 - val_loss
Epoch 19/30
64/64 [==============================] - 18s 274ms/step - loss: 0.3974 - acc: 0.8306 - val_loss
Epoch 20/30
64/64 [==============================] - 17s 273ms/step - loss: 0.3798 - acc: 0.8423 - val_loss
Epoch 21/30
64/64 [==============================] - 18s 276ms/step - loss: 0.3858 - acc: 0.8398 - val_loss
Epoch 22/30
64/64 [==============================] - 18s 274ms/step - loss: 0.3846 - acc: 0.8369 - val_loss
Epoch 23/30
64/64 [==============================] - 18s 276ms/step - loss: 0.3610 - acc: 0.8462 - val_loss
Epoch 24/30
64/64 [==============================] - 18s 274ms/step - loss: 0.3540 - acc: 0.8486 - val_loss
Epoch 25/30
64/64 [==============================] - 17s 273ms/step - loss: 0.3480 - acc: 0.8442 - val_loss
Epoch 26/30
64/64 [==============================] - 17s 264ms/step - loss: 0.3539 - acc: 0.8560 - val_loss
Epoch 27/30
64/64 [==============================] - 17s 272ms/step - loss: 0.3481 - acc: 0.8535 - val_loss
Epoch 28/30
64/64 [==============================] - 17s 273ms/step - loss: 0.3613 - acc: 0.8521 - val_loss
Epoch 29/30
64/64 [==============================] - 17s 273ms/step - loss: 0.3420 - acc: 0.8491 - val_loss
Epoch 30/30
64/64 [==============================] - 17s 267ms/step - loss: 0.3449 - acc: 0.8593 - val_loss
```

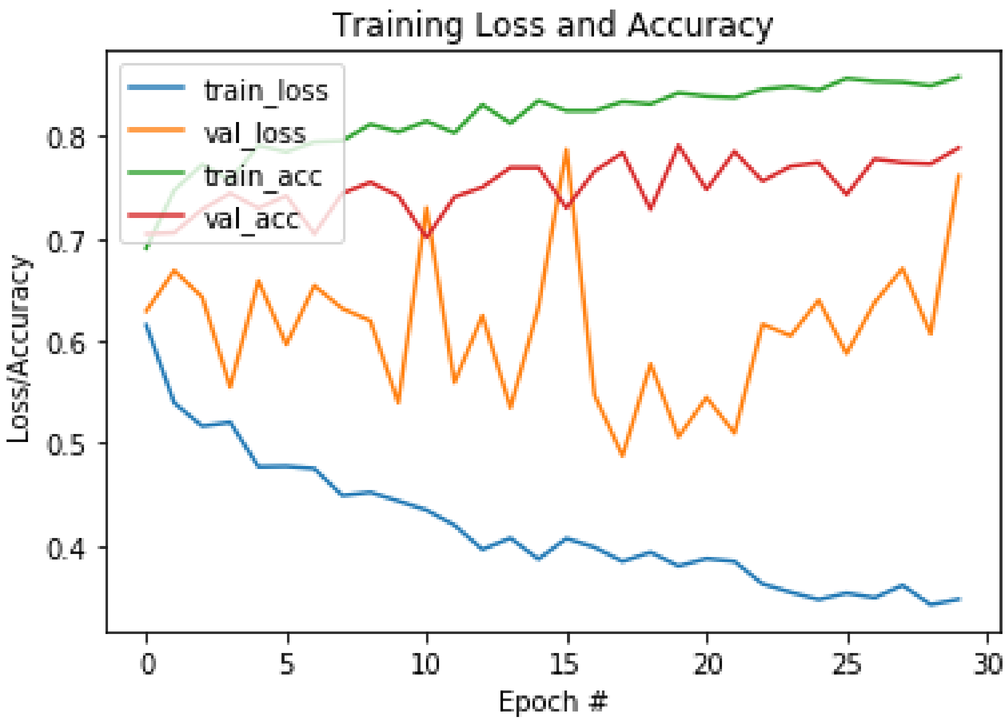
```
Out[21]: <keras.callbacks.History at 0x7f61fb0bccf8>
```

Save the trained model:

```
In [0]: model.save_weights('augmented_30_epochs.h5')
```

How does this model perform? (We use the function we defined earlier.)

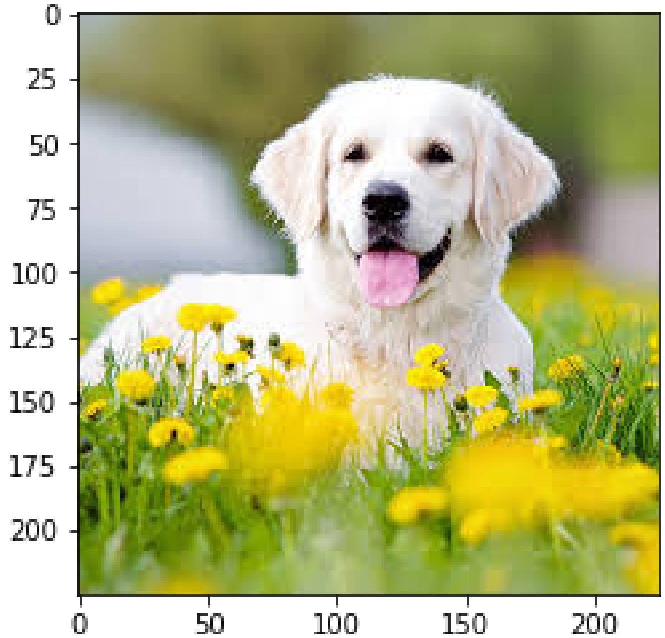
```
In [23]: plotResults()
```



Excellent, you have trained a CNN with augmented data! The accuracy is much higher and surpasses 80%

Now let's see if we can use this model to get the class prediction for one particular image:

```
In [24]: # Is this a dog or a cat?
         img = load_img('./data/test/test.jpg')  # this is a PIL image
         imgplot = plt.imshow(img)
         plt.show()
```



Now let's predict the class this image belongs to:

```
In [25]: # if you need to install cv2 you can use:
         # !pip install opencv-python
         import cv2

         testimg = cv2.imread('data/test/test.jpg')
         testimg = cv2.resize(testimg,(150,150))

         # data preprocessing to get the input in the same shape
         x = img_to_array(testimg)  # this is a Numpy array with shape (3, x, y)
         x = x * 1./255
         x = x.reshape((1,) + x.shape)  # this is a Numpy array with shape (1, 3, x, y)


         predictedclass = model.predict_classes(x)

         # cats are class 0; dogs are class 1 as you can see from command below
         # it's always good to check this for a new dataset
         # print(train_generator.class_indices)

         # making the output a bit nicer.
         if predictedclass == 1:
             prediction = 'dog'
         else:
             prediction = 'cat'
```