Convolutional Neural Networks

Project: Write an Algorithm for a Dog Identification App

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the Jupyter Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.

Why We're Here

In this notebook, you will make the first steps towards developing an algorithm that could be used as part of a mobile or web app. At the end of this project, your code will accept any user-supplied image as input. If a dog is detected in the image, it will provide an estimate of the dog's breed. If a human is detected, it will provide an estimate of the dog breed that is most resembling. The image below displays potential sample output of your finished project (... but we expect that each student's algorithm will behave differently!).

In this real-world setting, you will need to piece together a series of models to perform different tasks; for instance, the algorithm that detects humans in an image will be different from the CNN that infers dog breed. There are many points of possible failure, and no perfect algorithm exists. Your imperfect solution will nonetheless create a fun user experience!

In [1]:
```
import numpy as np
from glob import glob

# load filenames for human and dog images
human_files = np.array(glob("lfw/*/*"))
dog_files = np.array(glob("dogImages/*/*/*"))

# print number of images in each dataset
print('There are %d total human images.' % len(human_files))
print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```

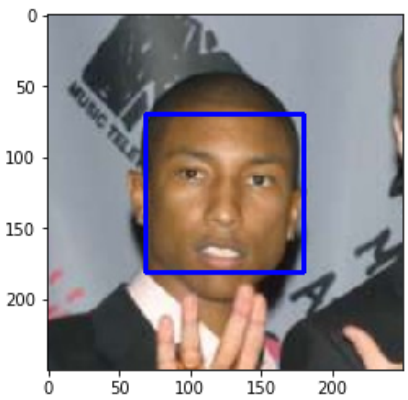
Step 1: Detect Humans

In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers (http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github (https://github.com/opencv/opencv/tree/master/data/haarcascades). We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

In [2]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

# load color (BGR) image
img = cv2.imread(human_files[0])
# convert BGR image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# find faces in image
faces = face_cascade.detectMultiScale(gray)

# print number of faces detected in the image
print('Number of faces detected:', len(faces))

# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

# convert BGR image to RGB for plotting
cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# display the image, along with bounding box
plt.imshow(cv_rgb)
plt.show()
```

```
Number of faces detected: 1
```



Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The `detectMultiScale` function executes the classifier stored in `face_cascade` and takes the grayscale image as a parameter.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

Write a Human Face Detector

We can use this procedure to write a function that returns `True` if a human face is detected in an image and `False` otherwise. This function, aptly named `face_detector`, takes a string-valued file path to an image as input and appears in the code block below.

In [3]:
```
# returns "True" if face is detected in image stored at img_path
def face_detector(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

(IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.

- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays `human_files_short` and `dog_files_short`.

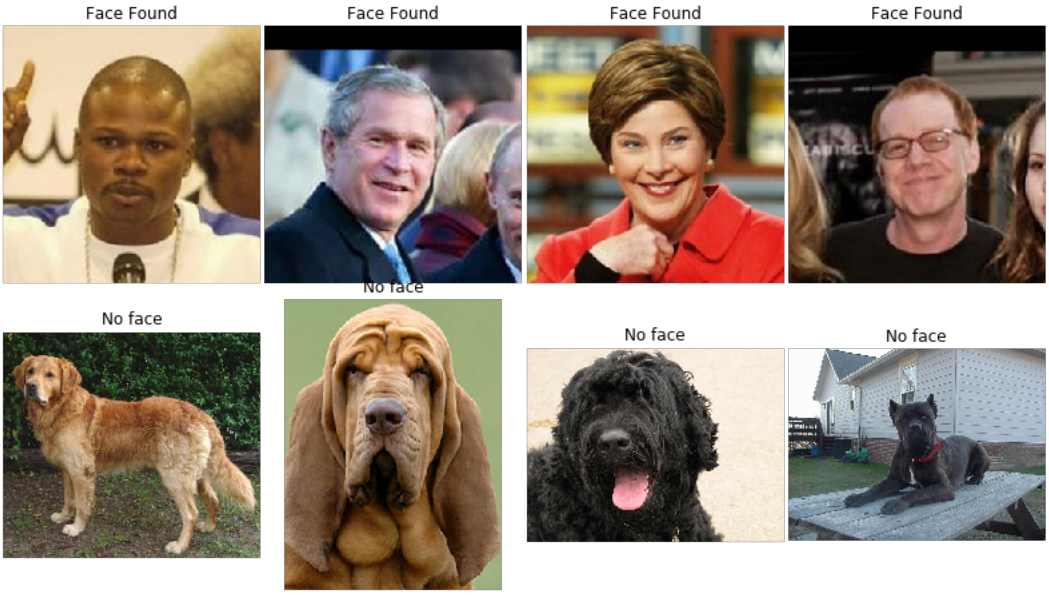
Answer: (You can print out your results and/or write your percentages in this cell)

In 99% of images of humans, a face was found. In 13% of images of dogs, a face was found

In [4]:
```
%matplotlib inline
```

In [5]:
```
# from tqdm import tqdm

human_files_short = human_files[:100]
dog_files_short = dog_files[:100]

#-#-# Do NOT modify the code above this line. #-#-#

## TODO: Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.

fig, ax = plt.subplots(2,4, figsize = (14,8))

for k, files in enumerate([human_files, dog_files]):
    count = 0
    for i in range(4):
        # Random selection
        j = np.random.randint(len(files))

        # Is there a face
        facelog = face_detector(files[j])

        count += 1*facelog
        # Plot the image and set the title accordingly
        img = cv2.imread(files[j])
        cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        ax[k,i].imshow(cv_rgb)
        if facelog:
            ax[k,i].set_title('Face Found')
        else:
            ax[k,i].set_title('No face')
        ax[k,i].set_axis_off()

fig.subplots_adjust(wspace=0.02, hspace=0)

for k, files in enumerate([human_files_short, dog_files_short]):
    count = 0
    for file in files:
        # Is there a face
        facelog = face_detector(file)
        count+= facelog*1

    if k == 0:
        print('In {}% of human images, it found a face'.format(int(count*100
/len(human_files_short))))
    else:
        print('In {}% of dog images, it found a face'.format(int(count*100/l
en(dog_files_short))))
```

```
In 99% of human images, it found a face
In 13% of dog images, it found a face
```



In [6]:
```
dog_files_test = np.array(glob("dogImages/test/*"))
dog_files_valid = np.array(glob("dogImages/valid/*/*"))
dog_files_train = np.array(glob("dogImages/train/*/*"))
print(len(dog_files_test),len(dog_files_valid),len(dog_files_train))
```

```
133 835 6680
```

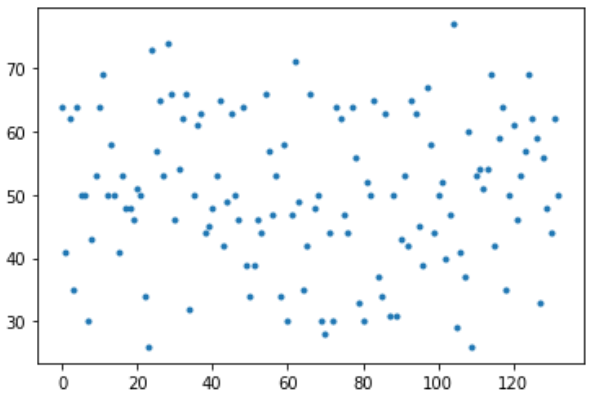
In [7]:
```
ltotal = len(dog_files)
print(len(dog_files_test)/ltotal,len(dog_files_valid)/ltotal,len(dog_files_t
rain)/ltotal)
```

```
0.01592623637887678 0.09998802538618129 0.7999042030894503
```

In [8]:
```
folders1 = []
folders2 = []
for file in dog_files:
    folders1.append(file.split('/')[1])
    folders2.append(file.split('/')[2])

f1 = set(folders1)
f2 = set(folders2)

numberfiles = np.zeros(len(f2))
for i, folder in enumerate(f2):
    files_train = np.array(glob("dogImages/train/"+folder+"/*"))
    numberfiles[i] = len(files_train)
```

In [9]:
```
plt.plot(numberfiles,'.')
print(np.mean(numberfiles), np.std(numberfiles))
print(list(f2)[numberfiles.argmin()],numberfiles.min())
print(list(f2)[numberfiles.argmax()],numberfiles.max())
```

```
50.225563909774436 11.819199711692114
132.Xoloitzcuintli 26.0
005.Alaskan_malamute 77.0
```



We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short` .

In [10]:
```
### (Optional)
### TODO: Test performance of another face detection algorithm.
### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs

In this section, we use a pre-trained model (http://pytorch.org/docs/master/torchvision/models.html) to detect dogs in images.

Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on ImageNet (http://www.image-net.org/), a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a).

In [11]:
```
import torch
import torchvision.models as models
# check if CUDA is available
use_cuda = torch.cuda.is_available()
```

In [12]:
```
import torch
import torchvision.models as models

# define VGG16 model
VGG16 = models.vgg16(pretrained=True)


# move model to GPU if CUDA is available
if use_cuda:
    VGG16 = VGG16.cuda()
```

Given an image, this pre-trained VGG-16 model returns a prediction (derived from the 1000 possible categories in ImageNet) for the object that is contained in the image.

(IMPLEMENTATION) Making Predictions with a Pre-trained Model

In the next code cell, you will write a function that accepts a path to an image (such as `'dogImages/train/001.Affenpinscher/Affenpinscher_00001.jpg'`) as input and returns the index corresponding to the ImageNet class that is predicted by the pre-trained VGG-16 model. The output should always be an integer between 0 and 999, inclusive.

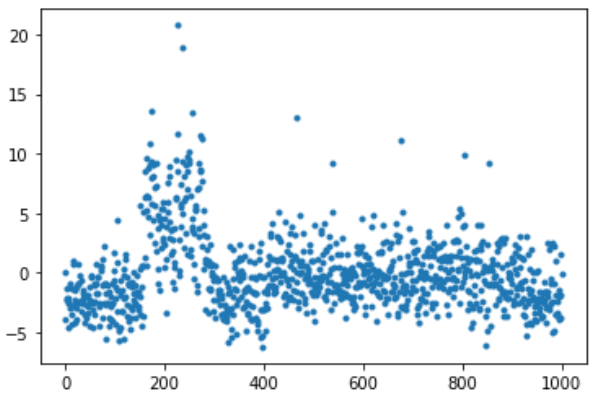
Before writing the function, make sure that you take the time to learn how to appropriately pre-process tensors for pre-trained models in the PyTorch documentation (http://pytorch.org/docs/stable/torchvision/models.html).

Before writing function, let us understand how to provide input data to the network, and what kind of output we receive. As usual the documentation is a bit confusing, but on the web there are some resources where they do go around small details for single image processing:

- We first load the image with Pillow
- Then we construct the transformation for preprocessing:
  - Resize image
  - Crop the central part
  - Transform it to a tensor
  - Normalize according to documentation
  - Unsquezze the array to copy it to the network
  - Move it to the cuda device
- We put VGG16 in evaluation mode
- Then, we evaluate it and take it back to the cpu
- Finally, we need to detach the tensor to forget about the derivatives and transform it into numpy
- The largest value is the label

In [13]:
```
#https://www.learnopencv.com/pytorch-for-beginners-image-classification-using-pre-trained-models/
from PIL import Image
import torchvision.transforms as transforms


img = Image.open(dog_files[0])

normalize = transforms.Normalize(mean = [0.485, 0.456, 0.406],
                                 std = [0.229, 0.224, 0.225])

transform = transforms.Compose([transforms.Resize(256),
                                transforms.CenterCrop(224),
                                transforms.ToTensor(),
                                normalize])

imgtd = transform(img)
batch = torch.unsqueeze(imgtd, 0)
if use_cuda:
    batch = batch.cuda()
#torch.utils.data.DataLoader
VGG16.eval()

output = VGG16(batch).cpu()
outputnp = output.detach().numpy()
plt.plot(outputnp.flatten(),'.')

print('Class is:', outputnp.argmax())
```

```
Class is: 225
```



Now, we are ready to write it

In [14]:
```
from PIL import Image
import torchvision.transforms as transforms

# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

In [15]:
```
def VGG16_predict(img_path):
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

    Returns:
        Index corresponding to VGG-16 model's prediction
    '''

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image

    img = Image.open(img_path)

    # Preprocessing
    # Defining function
    normalize = transforms.Normalize(mean = [0.485, 0.456, 0.406],
                                     std = [0.229, 0.224, 0.225])
    transform = transforms.Compose([transforms.Resize(256),
                                    transforms.CenterCrop(224),
                                    transforms.ToTensor(),
                                    normalize])

    # Transforming
    imgtd = transform(img)
    batch = torch.unsqueeze(imgtd, 0)
    if use_cuda:
        batch = batch.cuda()

    # Network evaluation
    VGG16.eval()

    output = VGG16(batch).cpu()
    outputnp = output.detach().numpy()

    return outputnp.argmax() # predicted class index
```

(IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a), you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

Use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

In [16]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    imgclass = VGG16_predict(img_path)
    isdog = False
    if imgclass>150 and imgclass<269:
        isdog = True

    return isdog # true/false
```

(IMPLEMENTATION) Assess the Dog Detector

Question 2: Use the code cell below to test the performance of your `dog_detector` function.

- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?

Answer: VGG16 classifies 0% of images of humans as dogs, in this subset, and 99% of images of dogs as dogs. This can be compared to mobilenet v2 (in notebook:data exploration.ipynb), where again 0% of images of humans are classified as dogs and 100% of images of dogs are classified as dogs.

In [17]:
```
%%time
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

for k, files in enumerate([human_files_short, dog_files_short]):
    count = 0
    for file in files:
        # Is it a dog?
        isdog = dog_detector(file)
        count+= isdog*1

    if k == 0:
        print('It classified {}% of human images as dogs'.format(int(count*1
00/len(human_files_short))))
    else:
        print('It classified {}% of dogs images as dogs'.format(int(count*10
0/len(dog_files_short))))
```

```
It classified 0% of human images as dogs
It classified 99% of dogs images as dogs
CPU times: user 11.7 s, sys: 7.85 ms, total: 11.7 s
Wall time: 2.64 s
```

We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3 (http://pytorch.org/docs/master/torchvision/models.html#inception-v3), ResNet-50 (http://pytorch.org/docs/master/torchvision/models.html#id3), etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

In [18]:
```
### (Optional)
### TODO: Report the performance of another pre-trained network.
### Feel free to use as many code cells as needed
mobilenetv2 = models.mobilenet_v2(pretrained=True)

if use_cuda:
    mobilenetv2 = mobilenetv2.cuda()

def mobilenet_predict(img_path):
    '''
    Use pre-trained mobilenet model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

    Returns:
        Index corresponding to VGG-16 model's prediction
    '''

    img = Image.open(img_path)

    # Preprocessing
    # Defining function
    normalize = transforms.Normalize(mean = [0.485, 0.456, 0.406],
                                     std = [0.229, 0.224, 0.225])
    transform = transforms.Compose([transforms.Resize(256),
                                    transforms.CenterCrop(224),
                                    transforms.ToTensor(),
                                    normalize])

    # Transforming
    imgtd = transform(img)
    batch = torch.unsqueeze(imgtd, 0)
    if use_cuda:
        batch = batch.cuda()

    # Network evaluation
    mobilenetv2.eval()

    output = mobilenetv2(batch).cpu()
    outputnp = output.detach().numpy()

    return outputnp.argmax() # predicted class index

#This will override the previous function
def dog_detector(img_path):
    imgclass = mobilenet_predict(img_path)
    isdog = False
    if imgclass>150 and imgclass<269:
        isdog = True

    return isdog
```

Note. The reported statistics of mobilenet can be found in data exploration notebook: 0% humans are classified as dogs, and 100% of dogs are classified as dogs. We decided to change to this method since is lighter than VGG, and probably more accurate (although results cannot be distinguished statistically, we would need to test them in a larger dataset, not only 100 samples).

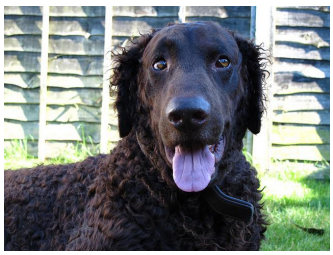
Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

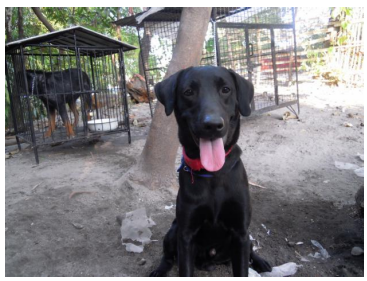
We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between a Brittany and a Welsh Springer Spaniel.

Brittany	Welsh Springer Spaniel
	

It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

Curly-Coated Retriever	American Water Spaniel
	

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

Yellow Labrador	Chocolate Labrador	Black Labrador
		

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

(IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate [data loaders (http://pytorch.org/docs/stable](http://pytorch.org/docs/stable)

In [19]:
```
## Add small gaussian noise to pixel values
class GaussianNoise:
    '''This class is another augmentation method that just add
    small Gaussian noise to the image. The scale of the noise is
    set by the parameter std. Note that if it is additive it should
    be behind normalization function, or it can produce negative
    pixel values, no clipping for the time being.
    '''

    def __init__(self, std = 0.02, noisetype = 'additive'):
        self.std = std
        self.noisetype = noisetype
        if self.noisetype == 'additive':
            self.transform = lambda x: x + torch.randn(x.size())*std
        elif self.noisetype == 'multiplicative':
            self.transform = lambda x: x*(1.0+torch.randn(x.size())*std)
        else:
            print('Noise type not implemented, use either additive or multip
licative. Set to identity')
            self.transform = lambda x: x

    def __call__(self, x):
        return self.transform(x)
```

In [20]:
```
import os
from torchvision import datasets

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes

#Hyperparameters
batch_sizes = 8
max_angle = 20
color_r = (0., 0., 0., 0.0) #(0.5, 0.5, 0.001, 0.001)
max_noise = 0.0 #0.01
# Preprocessing
normalize = transforms.Normalize(mean = [0.485, 0.456, 0.406],
                                 std = [0.229, 0.224, 0.225])

data_transform = {'train': transforms.Compose([transforms.Resize(256),
                          transforms.RandomCrop(224),
                          transforms.RandomRotation(max_angle),
                          transforms.RandomVerticalFlip(),
                          transforms.RandomHorizontalFlip(),
                          transforms.ColorJitter(brightness=color_r[0],
                                                 contrast=color_r[1],
                                                 saturation=color_r[2],
                                                 hue=color_r[3]),
                          transforms.RandomPerspective(),
                          transforms.ToTensor(),
                          normalize,
                          GaussianNoise(max_noise, noisetype='additive')
                          ]),
              'val': transforms.Compose([transforms.Resize(256),
                          transforms.CenterCrop(224),
                          transforms.ToTensor(),
                          normalize,
                          ]),
              'test': transforms.Compose([transforms.Resize(256),
                          transforms.CenterCrop(224),
                          transforms.ToTensor(),
                          normalize,
                          ]),
             }


#Datasets
train_data = datasets.ImageFolder('dogImages/train',
                                  transform = data_transform['train'])
valid_data = datasets.ImageFolder('dogImages/valid',
                                  transform = data_transform['val'])
test_data = datasets.ImageFolder('dogImages/test',
                                  transform = data_transform['test'])



data_loaders = {'train': torch.utils.data.DataLoader(train_data,
                                           batch_size = batch_sizes,
                                           shuffle = True,
                                           num_workers = 1
                                            ),
                'valid':torch.utils.data.DataLoader(valid_data,
                                           batch_size = batch_sizes,
                                           shuffle = True,
                                           num_workers = 1
                                            ),
                'test':torch.utils.data.DataLoader(test_data,
                                           batch_size = batch_sizes,
                                           shuffle = True,
                                           num_workers = 1
                                            )
                }
```

In [21]:
```
class_names = train_data.classes
```

Question 3: Describe your chosen procedure for preprocessing the data.

- How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why?
- Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

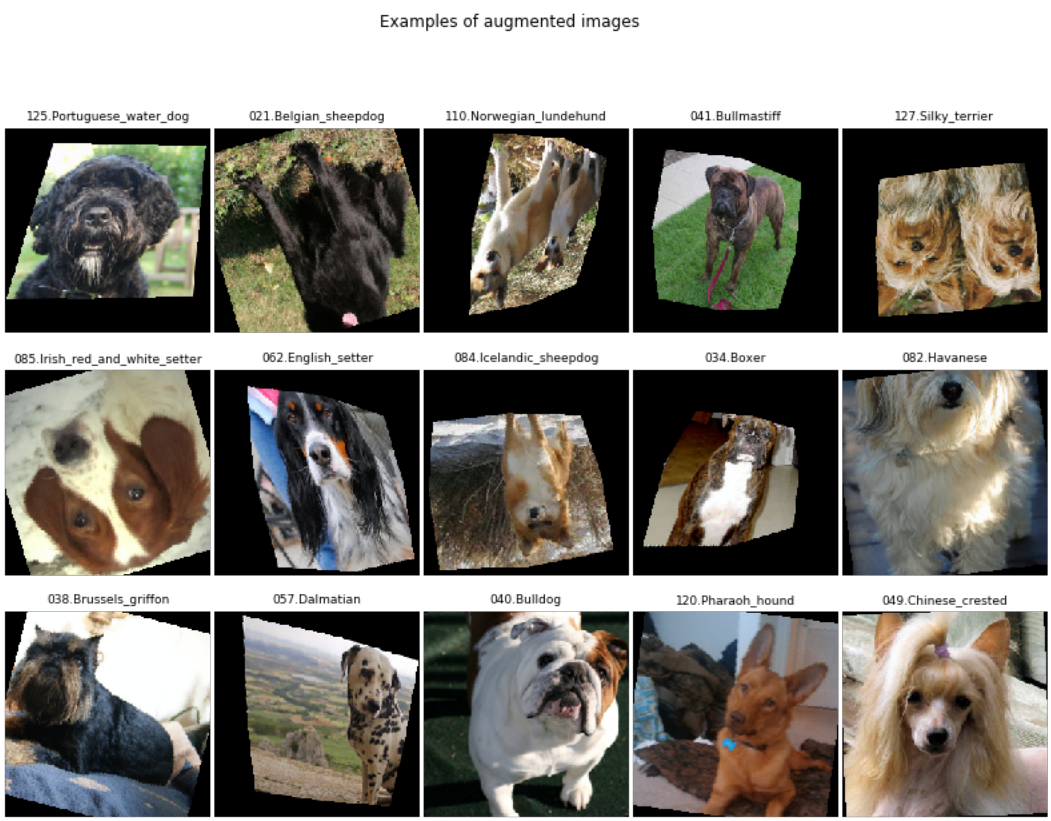
Answer: This is the chain of transformations:

- Image is resized to 256x256 pixels to standardize all the iamges
- A random section of the image of 224x224 pixels is cropped.
- Applied a rotation with a random angle from $-\theta_{\max}$ to $\theta_{\max}$, where $\theta_{\max} = 30$ degrees.
- Applied randomly a vertical and/or a horizontal flip.
- Add color jitter in the image, small enough so that it cannot be seen.
- Normalization of images, using the averages and standard deviations that were used in ImageNet dataset, much larger than our dog dataset.
- Applied small (additive) Gaussian noise, with standard deviation of 0.05.

We use size 224x224 for the input tensor, so we can use it for mobilenetv2 and other pretrained models. We also decided to augment the dataset as we foresee overfitting issues given the small dataset.

In [22]:
```
# https://stackoverflow.com/questions/55179282/display-examples-of-augmented
-images-in-pytorch


def denormalise(img):
    ''' It reverses normalization for img, by applying the inverse normaliza
tion function.
    '''
    # transform PIL image to a normal numpy array
    img = img.numpy().squeeze().transpose(1, 2, 0)
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    img = (img*std + mean).clip(0,1)
    return img

sampler = torch.utils.data.DataLoader(train_data,
                                      batch_size = 1,
                                      shuffle = True,
                                      )


fig, axs = plt.subplots(3, 5, figsize = (14,10) )
axs = axs.flatten()

for ax in axs:
    img, idx = next(iter(sampler))

    ax.imshow(denormalise(img))
    ax.set_axis_off()
    ax.set_title(class_names[idx], fontsize =9)

fig.subplots_adjust(wspace=0.02, hspace=0)
fig.suptitle('Examples of augmented images')
```

Out[22]: Text(0.5, 0.98, 'Examples of augmented images')



(IMPLEMENTATION) Model Architecture

Create a CNN to classify dog breed. Use the template in the code cell below.

In [24]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture

def createblocklist(n0, n1, nf, kernels, nrep = 3, nblocks = 2, pool ='maxpo
ol'):
    '''This function return a list of modules to be implemented as a sequent
ial layer block
    Input variables:
    n0: index in nf array for the layer before this block
    n1: index in nf array, for the output filter dimension
    nf: array with number of filters (or strides for pooling layers) for dif
ferent layers of the network
    kernels: corresponding arrray of kernels to nf
    nrep: number of times layers of convolution+ReLU are repeated in a subbl
ock
    nblocks: number of blocks
    pool: type of 2d pooling it can be maxpool or anything else that will be
considered average pooling.

    Output:
    list of pytorch layers
    '''
    # Initial conv+relu that takes previous number of filters to the new one
    modlist = [nn.Conv2d(nf[n0],nf[n1]
                  ,kernels[n1], padding = kernels[n1]//2),
        nn.ReLU()]

    # Chain of nrep conv+relu
    for i in range(nrep):
        modlist.extend([nn.Conv2d(nf[n1],nf[n1]
                      ,kernels[n1], padding = kernels[n1]//2),
            nn.ReLU()]
        )

    # Only if there are more than one block we do the following block
    if nblocks>1:
        for j in range(nblocks-1):
            # First we compress the number of filters to the initial one
            # This might be useful if we to add skip connections later on.

            modlist.extend([nn.Conv2d(nf[n1],nf[n0]
                          ,kernels[n1], padding = kernels[n1]//2),
                nn.ReLU()]
            )
            # And decompress it
            modlist.extend([nn.Conv2d(nf[n0],nf[n1]
                          ,kernels[n1], padding = kernels[n1]//2),
                nn.ReLU()]
            )

            # Chain of nrep conv+relu
            for i in range(nrep):
                modlist.extend([nn.Conv2d(nf[n1],nf[n1]
                              ,kernels[n1], padding = kernels[n1]//2),
                    nn.ReLU()]
                )


    # Final pooling layer
    n1 += 1
    if pool =='maxpool':
        modlist.append(nn.MaxPool2d(kernels[n1],
                        stride = nf[n1], padding = kernels[n1]//2)
        )
    else:
        modlist.append(nn.AvgPool2d(kernels[n1],
```

In [25]:
```
from torchsummary import summary

summary(model_scratch, input_size = (3,224,224))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 32, 224, 224]           2,432
              ReLU-2         [-1, 32, 224, 224]               0
         MaxPool2d-3         [-1, 32, 112, 112]               0
            Conv2d-4         [-1, 64, 112, 112]          51,264
              ReLU-5         [-1, 64, 112, 112]               0
         MaxPool2d-6           [-1, 64, 56, 56]               0
            Conv2d-7          [-1, 128, 56, 56]         204,928
              ReLU-8          [-1, 128, 56, 56]               0
         MaxPool2d-9          [-1, 128, 28, 28]               0
           Conv2d-10          [-1, 128, 28, 28]         409,728
             ReLU-11          [-1, 128, 28, 28]               0
        AvgPool2d-12            [-1, 128, 2, 2]               0
          Dropout-13                  [-1, 512]               0
           Linear-14                  [-1, 500]         256,500
             ReLU-15                  [-1, 500]               0
           Linear-16                  [-1, 133]          66,633
================================================================
Total params: 991,485
Trainable params: 991,485
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 49.78
Params size (MB): 3.78
Estimated Total Size (MB): 54.14
----------------------------------------------------------------
```

Question 4: Outline the steps you took to get to your final CNN architecture and your reasoning at each step.

Answer: I thought about doing a similar architecture to ResNets, that is why there is a subroutine: createblocklist. Training fairly deep (of around 10 conv+relu layers) I realized very quickly that training this network would be really difficult if we do not include skip connections and batch normalization. The main problem was that the network was not decreasing the training loss. To solve this issue, I decided to return to a very simplified version with only 4 conv+relu with max pooling in between, keeping a small number of parameters, and train it. It happened to start converging after a few attempts with different number of filters.

(IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function (http://pytorch.org/docs/stable/nn.html#loss-functions) and optimizer (http://pytorch.org/docs/stable/optim.html). Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

In [26]:
```
import torch.optim as optim

### TODO: select loss function
criterion_scratch = nn.CrossEntropyLoss()

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = .5e-2, moment
um = 0.9)
```

(IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters (http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_scratch.pt'`.

In [27]:
```
# the following import is required for training to be robust to truncated im
ages
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

loaders_scratch = data_loaders

def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_pat
h):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

    for epoch in range(1, n_epochs+1):
        # initialize variables to monitor training and validation loss
        train_loss = 0.0
        valid_loss = 0.0
        train_acc = 0.0
        valid_acc = 0.0
        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data
- train_loss))

            # Computing the loss

            yhat = model(data)
            loss = criterion(yhat, target)

            # Compute the gradients
            loss.backward()

            # Update parameters and gradients to zero
            optimizer.step()
            optimizer.zero_grad()

            train_loss +=((1/(batch_idx + 1)) * (loss.data - train_loss))

            _, maxidcs = torch.max(yhat,1)
            train_acc += (1/(batch_idx + 1))*(-train_acc+
                 (maxidcs == target).sum().data.cpu().numpy()/maxidcs.size
()[0])

            del data
            del target

        ######################
        # validate the model #
        ######################
        model.eval()

        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss

            yhat = model(data)
            loss = criterion(yhat, target)
            valid_loss += ((1/(batch_idx + 1))*(loss.data - valid_loss))
```

In [29]:
```
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

Out[29]:
```
<All keys matched successfully>
```

In [25]:
```
# train the model
model_scratch = train(10, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
Epoch: 1        Training Loss: 4.656472         Validation Loss: 4.574130
Epoch: 2        Training Loss: 4.484364         Validation Loss: 4.431760
Epoch: 3        Training Loss: 4.321619         Validation Loss: 4.222309
Epoch: 4        Training Loss: 4.207127         Validation Loss: 4.193263
Epoch: 5        Training Loss: 4.192065         Validation Loss: 4.107595
Epoch: 6        Training Loss: 4.132681         Validation Loss: 4.268332
Epoch: 7        Training Loss: 4.137611         Validation Loss: 4.123872
Epoch: 8        Training Loss: 4.099572         Validation Loss: 4.131931
Epoch: 9        Training Loss: 4.092736         Validation Loss: 4.123960
Epoch: 10       Training Loss: 4.103441         Validation Loss: 4.124364
```

Out[25]:
```
<All keys matched successfully>
```

In [30]:
```
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = .5e-3, moment
um = 0.9)
```

In [31]:
```
%%time
model_scratch = train(5, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1        Training Loss: 3.294, acc:0.200         Validation Loss: 3.89
4110, acc:0.119
Epoch: 2        Training Loss: 3.318, acc:0.187         Validation Loss: 3.89
0810, acc:0.125
Epoch: 3        Training Loss: 3.297, acc:0.192         Validation Loss: 3.91
5972, acc:0.118
Epoch: 4        Training Loss: 3.321, acc:0.192         Validation Loss: 3.86
4164, acc:0.125
Epoch: 5        Training Loss: 3.300, acc:0.191         Validation Loss: 3.89
3094, acc:0.119
CPU times: user 2min 5s, sys: 4.45 s, total: 2min 9s
Wall time: 6min 8s
```

(IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

In [32]:
```
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

Out[32]:
```
<All keys matched successfully>
```

In [33]:
```
def test(loaders, model, criterion, use_cuda):

    # monitor test loss and accuracy
    test_loss = 0.
    correct = 0.
    total = 0.

    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the m
odel
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_l
oss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu
().numpy())
        total += data.size(0)

    print('Test Loss: {:.6f}\n'.format(test_loss))

    print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
        100. * correct / total, correct, total))

# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.889352


Test Accuracy: 14% (123/836)
```

Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

(IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate [data loaders (http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader)](http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader) for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively).

If you like, **you are welcome to use the same data loaders from the previous step**, when you created a CNN from scratch.

In [34]:
```
## TODO: Specify data loaders
batch_sizes = 8
max_angle = 30
color_r = (0.2, 0.2, 0.001, 0.001)
max_noise = 0.05
path = 'dogImages/'
# Preprocessing
normalize = transforms.Normalize(mean = [0.485, 0.456, 0.406],
                                 std = [0.229, 0.224, 0.225])

data_transform = {'train': transforms.Compose([transforms.Resize(256),
                          transforms.RandomCrop(224),
                          transforms.RandomRotation(max_angle),
                          transforms.RandomVerticalFlip(),
                          transforms.RandomHorizontalFlip(),
                          transforms.ColorJitter(brightness=color_r[0],
                                                 contrast=color_r[1],
                                                 saturation=color_r[2],
                                                 hue=color_r[3]),
                          transforms.ToTensor(),
                          normalize,
                          GaussianNoise(max_noise, noisetype='additive')
                          ]),
              'val': transforms.Compose([transforms.Resize(256),
                          transforms.CenterCrop(224),
                          transforms.ToTensor(),
                          normalize,
                          ]),
              'test': transforms.Compose([transforms.Resize(256),
                          transforms.CenterCrop(224),
                          transforms.ToTensor(),
                          normalize,
                          ]),
             }


#Datasets
train_data = datasets.ImageFolder(path+'train',
                                  transform = data_transform['train'])
valid_data = datasets.ImageFolder(path+'valid',
                                  transform = data_transform['val'])
test_data = datasets.ImageFolder(path+'test',
                                  transform = data_transform['test'])



loaders_transfer = {'train': torch.utils.data.DataLoader(train_data,
                                           batch_size = batch_sizes,
                                           shuffle = True,
                                           num_workers = 1
                                            ),
                'valid':torch.utils.data.DataLoader(valid_data,
                                           batch_size = batch_sizes,
                                           shuffle = True,
                                           num_workers = 1
                                            ),
                'test':torch.utils.data.DataLoader(test_data,
                                           batch_size = batch_sizes,
                                           shuffle = True,
                                           num_workers = 1
                                            )
                }
```

(IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer` .

In [35]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture

model_transfer = models.mobilenet_v2(pretrained=True)

num_classes = len(class_names)
nft = model_transfer.last_channel
model_transfer.classifier = nn.Sequential(
            nn.Dropout(0.2),
            nn.Linear(nft, num_classes),
        )


if use_cuda:
    model_transfer = model_transfer.cuda()
```

Question 5: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. Describe why you think the architecture is suitable for the current problem.

Answer: I tried two different models: Mobilenetv2 and ResNet16. While ResNet16 was training in Udacity's notebook, Mobilenetv2 was being trained on my laptop. During the training of both algorithms, the training loss of Mobilenetv2 decreased much faster than ResNet16, not only per epoch, but also partially due to the fact it takes less time to train an epoch. Therefore I focused on mobilenet-based architecture

(IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function (http://pytorch.org/docs/master/nn.html#loss-functions) and optimizer (http://pytorch.org/docs/master/optim.html). Save the chosen loss function as `criterion_transfer` , and the optimizer as `optimizer_transfer` below.

In [36]:
```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.Adam(model_transfer.parameters(), lr = 1e-3)
```

(IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters (http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_transfer.pt'` .

In [60]:
```
# train the model
n_epochs = 10
model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer
_transfer, criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
Epoch: 1        Training Loss: 3.328, acc:0.161         Validation Loss: 2.87
7975, acc:0.222
Epoch: 2        Training Loss: 3.277, acc:0.169         Validation Loss: 2.86
5016, acc:0.230
Epoch: 3        Training Loss: 3.224, acc:0.177         Validation Loss: 3.06
2237, acc:0.223
Epoch: 4        Training Loss: 3.153, acc:0.196         Validation Loss: 2.76
9486, acc:0.250
Epoch: 5        Training Loss: 3.066, acc:0.213         Validation Loss: 2.60
3333, acc:0.308
Epoch: 6        Training Loss: 2.971, acc:0.233         Validation Loss: 2.61
7466, acc:0.298
Epoch: 7        Training Loss: 2.833, acc:0.255         Validation Loss: 2.30
8269, acc:0.344
Epoch: 8        Training Loss: 2.686, acc:0.283         Validation Loss: 2.22
6388, acc:0.401
Epoch: 9        Training Loss: 2.644, acc:0.297         Validation Loss: 2.27
9058, acc:0.361
Epoch: 10       Training Loss: 2.538, acc:0.306         Validation Loss: 2.20
1912, acc:0.418
```

In [61]:
```
n_epochs = 20
model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer
_transfer, criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
Epoch: 1        Training Loss: 2.511, acc:0.328         Validation Loss: 2.21
4887, acc:0.375
Epoch: 2        Training Loss: 2.392, acc:0.348         Validation Loss: 1.97
7128, acc:0.439
Epoch: 3        Training Loss: 2.320, acc:0.358         Validation Loss: 2.01
0478, acc:0.425
Epoch: 4        Training Loss: 2.241, acc:0.382         Validation Loss: 1.81
2355, acc:0.484
Epoch: 5        Training Loss: 2.170, acc:0.396         Validation Loss: 1.69
9111, acc:0.517
Epoch: 6        Training Loss: 2.060, acc:0.434         Validation Loss: 1.73
7986, acc:0.498
Epoch: 7        Training Loss: 2.022, acc:0.424         Validation Loss: 1.77
0748, acc:0.506
Epoch: 8        Training Loss: 1.968, acc:0.444         Validation Loss: 1.58
5641, acc:0.553
Epoch: 9        Training Loss: 1.935, acc:0.456         Validation Loss: 1.61
4457, acc:0.545
Epoch: 10       Training Loss: 1.943, acc:0.456         Validation Loss: 1.64
0280, acc:0.515
Epoch: 11       Training Loss: 1.877, acc:0.472         Validation Loss: 1.73
7736, acc:0.503
Epoch: 12       Training Loss: 1.867, acc:0.468         Validation Loss: 1.68
8129, acc:0.544
Epoch: 13       Training Loss: 1.812, acc:0.483         Validation Loss: 1.39
4782, acc:0.588
Epoch: 14       Training Loss: 1.802, acc:0.489         Validation Loss: 1.51
7645, acc:0.578
Epoch: 15       Training Loss: 1.698, acc:0.511         Validation Loss: 1.62
7678, acc:0.525
Epoch: 16       Training Loss: 1.666, acc:0.521         Validation Loss: 1.47
0495, acc:0.567
Epoch: 17       Training Loss: 1.686, acc:0.517         Validation Loss: 1.83
7790, acc:0.487
Epoch: 18       Training Loss: 1.651, acc:0.527         Validation Loss: 1.42
6749, acc:0.598
Epoch: 19       Training Loss: 1.588, acc:0.532         Validation Loss: 1.42
2777, acc:0.588
Epoch: 20       Training Loss: 1.525, acc:0.549         Validation Loss: 1.44
5338, acc:0.605
```

In [62]:
```
n_epochs = 20
model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer
_transfer, criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
Epoch: 1        Training Loss: 1.517, acc:0.557         Validation Loss: 1.39
9182, acc:0.594
Epoch: 2        Training Loss: 1.522, acc:0.556         Validation Loss: 1.34
0911, acc:0.600
Epoch: 3        Training Loss: 1.512, acc:0.562         Validation Loss: 1.32
2531, acc:0.631
Epoch: 4        Training Loss: 1.495, acc:0.567         Validation Loss: 1.35
3712, acc:0.605
Epoch: 5        Training Loss: 1.427, acc:0.585         Validation Loss: 1.38
8041, acc:0.627
Epoch: 6        Training Loss: 1.401, acc:0.586         Validation Loss: 1.34
6849, acc:0.603
Epoch: 7        Training Loss: 1.365, acc:0.600         Validation Loss: 1.28
2675, acc:0.646
Epoch: 8        Training Loss: 1.394, acc:0.590         Validation Loss: 1.27
6777, acc:0.618
Epoch: 9        Training Loss: 1.393, acc:0.592         Validation Loss: 1.40
9579, acc:0.613
Epoch: 10       Training Loss: 1.356, acc:0.599         Validation Loss: 1.27
8390, acc:0.654
Epoch: 11       Training Loss: 1.794, acc:0.502         Validation Loss: 1.42
7038, acc:0.621
Epoch: 12       Training Loss: 1.509, acc:0.564         Validation Loss: 1.38
1229, acc:0.616
Epoch: 13       Training Loss: 1.331, acc:0.602         Validation Loss: 1.18
9366, acc:0.644
Epoch: 14       Training Loss: 1.253, acc:0.629         Validation Loss: 1.26
5083, acc:0.635
Epoch: 15       Training Loss: 1.245, acc:0.631         Validation Loss: 1.16
7597, acc:0.655
Epoch: 16       Training Loss: 1.197, acc:0.642         Validation Loss: 1.18
4683, acc:0.673
Epoch: 17       Training Loss: 1.308, acc:0.610         Validation Loss: 1.21
9509, acc:0.661
Epoch: 18       Training Loss: 1.200, acc:0.646         Validation Loss: 1.25
5322, acc:0.652
Epoch: 19       Training Loss: 1.225, acc:0.634         Validation Loss: 1.22
3307, acc:0.643
Epoch: 20       Training Loss: 1.190, acc:0.640         Validation Loss: 1.22
6062, acc:0.666
```

36 epochs to reach 0.67 in validation set

In [63]:
```
n_epochs = 20
model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer
_transfer, criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
Epoch: 1        Training Loss: 1.211, acc:0.638         Validation Loss: 1.23
3368, acc:0.652
Epoch: 2        Training Loss: 1.119, acc:0.663         Validation Loss: 1.18
5863, acc:0.650
Epoch: 3        Training Loss: 1.100, acc:0.669         Validation Loss: 1.25
1346, acc:0.671
Epoch: 4        Training Loss: 1.098, acc:0.670         Validation Loss: 1.12
7771, acc:0.692
Epoch: 5        Training Loss: 1.137, acc:0.656         Validation Loss: 1.30
6501, acc:0.648
Epoch: 6        Training Loss: 1.099, acc:0.663         Validation Loss: 1.20
1612, acc:0.673
Epoch: 7        Training Loss: 1.065, acc:0.682         Validation Loss: 1.27
4404, acc:0.653
Epoch: 8        Training Loss: 1.051, acc:0.682         Validation Loss: 1.14
1536, acc:0.655
Epoch: 9        Training Loss: 1.135, acc:0.653         Validation Loss: 1.17
3226, acc:0.674
Epoch: 10       Training Loss: 1.010, acc:0.692         Validation Loss: 1.22
3763, acc:0.658
Epoch: 11       Training Loss: 1.025, acc:0.691         Validation Loss: 1.12
0453, acc:0.664
Epoch: 12       Training Loss: 1.006, acc:0.690         Validation Loss: 1.17
7421, acc:0.655
Epoch: 13       Training Loss: 0.992, acc:0.697         Validation Loss: 1.15
8571, acc:0.667
Epoch: 14       Training Loss: 1.107, acc:0.669         Validation Loss: 1.14
6861, acc:0.679
Epoch: 15       Training Loss: 1.022, acc:0.693         Validation Loss: 1.23
1073, acc:0.687
Epoch: 16       Training Loss: 0.945, acc:0.709         Validation Loss: 1.18
5891, acc:0.680
Epoch: 17       Training Loss: 1.019, acc:0.694         Validation Loss: 1.14
5059, acc:0.696
Epoch: 18       Training Loss: 0.917, acc:0.719         Validation Loss: 1.15
4172, acc:0.698
Epoch: 19       Training Loss: 0.944, acc:0.710         Validation Loss: 1.21
2202, acc:0.662
Epoch: 20       Training Loss: 0.907, acc:0.717         Validation Loss: 1.15
5740, acc:0.676
```

In [64]:
```
# load the model that got the best validation accuracy (uncomment the line b
elow)
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

Out[64]:
```
<All keys matched successfully>
```

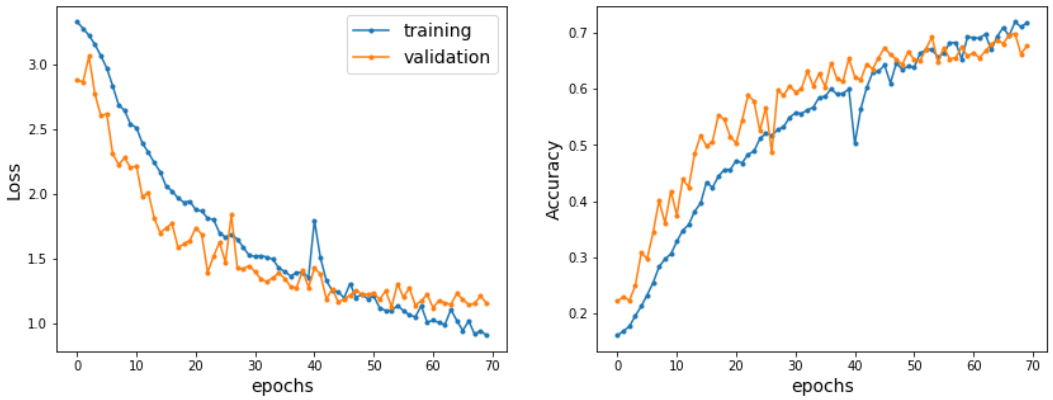
(IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 60%.

In [65]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.228273


Test Accuracy: 68% (571/836)
```

In [66]:
```
lossar = np.loadtxt('lossmobilenetv2_f.txt')
fig, ax = plt.subplots(1,2, figsize = (14,5))
ax[0].plot(lossar[:,0],'.-',label='training')
ax[0].plot(lossar[:,2],'.-',label='validation')
ax[0].set_xlabel('epochs', fontsize = 14)
ax[0].set_ylabel('Loss', fontsize = 14)
ax[1].plot(lossar[:,1],'.-',label='training')
ax[1].plot(lossar[:,3],'.-',label='validation')
ax[1].set_xlabel('epochs', fontsize = 14)
ax[1].set_ylabel('Accuracy', fontsize = 14)
ax[0].legend(fontsize = 14)
fig.savefig('loss_mobilenetv2.png', dpi = 200, tight_layout = True)
```



(IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher` , `Afghan hound` , etc) that is predicted by your model.

In [67]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

# list of class names by index, i.e. a name can be accessed like class_names
[0]
#class_names = [item[4:].replace("_", " ") for item in data_transfer['train
'].classes]
class_names = [item[4:].replace("_", " ") for item in class_names]
```

In [68]:
```
def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    model_transfer.eval()
    img = Image.open(img_path)

    img_transfd = data_transform['test'](img)
    batch = torch.unsqueeze(img_transfd, 0)
    if use_cuda:
        batch = batch.cuda()

    output = model_transfer(batch)
    outputnp = output.detach().cpu().numpy()
    #print(outputnp)
    return class_names[outputnp.argmax()]
```

In [79]:
```
%timeit predict_breed_transfer(dog_files_short[50]), dog_files_short[50]
```

```
9.38 ms ± 51.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

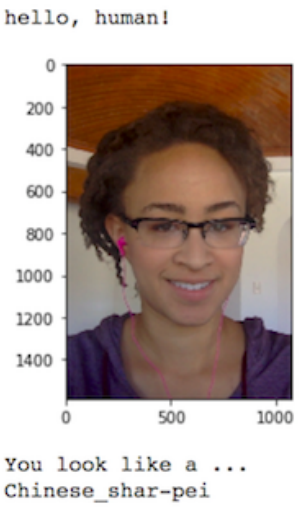
Step 5: Write your Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

- if a **dog** is detected in the image, return the predicted breed.
- if a **human** is detected in the image, return the resembling dog breed.
- if **neither** is detected in the image, provide output that indicates an error.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `dog_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

Some sample output for our algorithm is provided below, but feel free to design your own user experience!



(IMPLEMENTATION) Write your Algorithm

In [70]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path, plot = True):
    ## handle cases for a human face, dog, and neither
    # First we detect what it is:
    ishumanface = face_detector(img_path)
    isdog = dog_detector(img_path)

    if not (ishumanface or isdog):
        output = 'Error, no human or dog found in the picture'
    else:
        if plot:
            img = cv2.imread(img_path)
            cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            fig = plt.figure(figsize = (4,4))
            ax = fig.add_subplot(111)
            ax.imshow(cv_rgb)
            ax.set_axis_off()

        breed = predict_breed_transfer(img_path)
        if not isdog:
            #print('Hello you human!')
            #print('I would say you look like a {}'.format(breed))
            message = 'This human looks like a {}'.format(breed)
            output = (0, breed)
        else:
            #print('This cute dog is a {}'.format(breed))
            message = 'This cute dog is a {}'.format(breed)
            output = (1, breed)

        if plot:
            ax.set_title(message)

    return output
```

Step 6: Test Your Algorithm

In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that *you* look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

(IMPLEMENTATION) Test Your Algorithm on Sample Images!

Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer:

The output is worse. It fails particularly often with Border Collies or Daschunds. In the case of border collies, by inspecting the images one of the reasons is that they only have black and white dogs. With Daschunds it is not clear to me why. One way to improve the algorithm is to provide a more varied dataset, correcting the issues that the previous one has. A second way to improve is to be more aggresive with data augmentation which could extend for a bit longer the period before overfitting. A third way could be using a different approach, by creating a triple GAN, with a generator, a discriminator and a classifier -- this network.

In [78]:
```
%%time
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

new_human_files = ['testimages/fakeGAN.jpg',
                   'testimages/pablo.jpg',
                   'testimages/pablo2.jpg']

new_dog_files = ['testimages/002.jpg',
                   'testimages/001.jpg',
                   'testimages/003.jpg']
## suggested code, below
for file in np.hstack((new_human_files[:3], new_dog_files[:3])):
    run_app(file)
```

```
CPU times: user 14.3 s, sys: 29 ms, total: 14.3 s
Wall time: 2.33 s
```

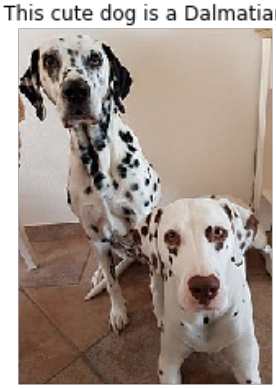
This human looks like a Irish setter

This human looks like a Italian greyhound

This human looks like a Great dane

This cute dog is a Norwegian lundehund

This cute dog is a Dalmatian



This cute dog is a Chow chow

Architecture of Mobilenet v2

In [77]:
```
summary(model_transfer, input_size = (3,224,224))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 32, 112, 112]             864
       BatchNorm2d-2         [-1, 32, 112, 112]              64
             ReLU6-3         [-1, 32, 112, 112]               0
            Conv2d-4         [-1, 32, 112, 112]             288
       BatchNorm2d-5         [-1, 32, 112, 112]              64
             ReLU6-6         [-1, 32, 112, 112]               0
            Conv2d-7         [-1, 16, 112, 112]             512
       BatchNorm2d-8         [-1, 16, 112, 112]              32
  InvertedResidual-9         [-1, 16, 112, 112]               0
           Conv2d-10         [-1, 96, 112, 112]           1,536
      BatchNorm2d-11         [-1, 96, 112, 112]             192
            ReLU6-12         [-1, 96, 112, 112]               0
           Conv2d-13          [-1, 96, 56, 56]             864
      BatchNorm2d-14          [-1, 96, 56, 56]             192
            ReLU6-15          [-1, 96, 56, 56]               0
           Conv2d-16          [-1, 24, 56, 56]           2,304
      BatchNorm2d-17          [-1, 24, 56, 56]              48
 InvertedResidual-18          [-1, 24, 56, 56]               0
           Conv2d-19         [-1, 144, 56, 56]           3,456
      BatchNorm2d-20         [-1, 144, 56, 56]             288
            ReLU6-21         [-1, 144, 56, 56]               0
           Conv2d-22         [-1, 144, 56, 56]           1,296
      BatchNorm2d-23         [-1, 144, 56, 56]             288
            ReLU6-24         [-1, 144, 56, 56]               0
           Conv2d-25          [-1, 24, 56, 56]           3,456
      BatchNorm2d-26          [-1, 24, 56, 56]              48
 InvertedResidual-27          [-1, 24, 56, 56]               0
           Conv2d-28         [-1, 144, 56, 56]           3,456
      BatchNorm2d-29         [-1, 144, 56, 56]             288
            ReLU6-30         [-1, 144, 56, 56]               0
           Conv2d-31         [-1, 144, 28, 28]           1,296
      BatchNorm2d-32         [-1, 144, 28, 28]             288
            ReLU6-33         [-1, 144, 28, 28]               0
           Conv2d-34          [-1, 32, 28, 28]           4,608
      BatchNorm2d-35          [-1, 32, 28, 28]              64
 InvertedResidual-36          [-1, 32, 28, 28]               0
           Conv2d-37         [-1, 192, 28, 28]           6,144
      BatchNorm2d-38         [-1, 192, 28, 28]             384
            ReLU6-39         [-1, 192, 28, 28]               0
           Conv2d-40         [-1, 192, 28, 28]           1,728
      BatchNorm2d-41         [-1, 192, 28, 28]             384
            ReLU6-42         [-1, 192, 28, 28]               0
           Conv2d-43          [-1, 32, 28, 28]           6,144
      BatchNorm2d-44          [-1, 32, 28, 28]              64
 InvertedResidual-45          [-1, 32, 28, 28]               0
           Conv2d-46         [-1, 192, 28, 28]           6,144
      BatchNorm2d-47         [-1, 192, 28, 28]             384
            ReLU6-48         [-1, 192, 28, 28]               0
           Conv2d-49         [-1, 192, 28, 28]           1,728
      BatchNorm2d-50         [-1, 192, 28, 28]             384
            ReLU6-51         [-1, 192, 28, 28]               0
           Conv2d-52          [-1, 32, 28, 28]           6,144
      BatchNorm2d-53          [-1, 32, 28, 28]              64
 InvertedResidual-54          [-1, 32, 28, 28]               0
           Conv2d-55         [-1, 192, 28, 28]           6,144
      BatchNorm2d-56         [-1, 192, 28, 28]             384
            ReLU6-57         [-1, 192, 28, 28]               0
           Conv2d-58         [-1, 192, 14, 14]           1,728
      BatchNorm2d-59         [-1, 192, 14, 14]             384
            ReLU6-60         [-1, 192, 14, 14]               0
           Conv2d-61          [-1, 64, 14, 14]          12,288
      BatchNorm2d-62          [-1, 64, 14, 14]             128
 InvertedResidual-63          [-1, 64, 14, 14]               0
           Conv2d-64         [-1, 384, 14, 14]          24,576
      BatchNorm2d-65         [-1, 384, 14, 14]             768
```

In [ ]: