Recap: What we have been doing

`1-Tensors-in-PyTorch.ipynb`: introduces a tensor representation in PyTorch

`2-Neural-networks-in-PyTorch.ipynb`: introduces a basic framework for defining neural networks via the `nn` module

`3-Training-neural-networks.ipynb`: introduces loss and backprop to improve predictions

`4-Fashion-MNIST.ipynb`: walks through the feed-forward neural network on Fashion MNIST data

`5-Inference-and-Validation.ipynb`: introduces dropout

`6-Saving-and-Loading-Models.ipynb`: shows how to save and load model so that you don't have to re-train it from scratch

Custom Image Data

In the previous notebooks we have been looking at standard and fairly artificial data, such as MNIST or Fashion MNIST, which you likely won't be using in real projects. Instead you are much more likely to deal with full-sized images, such as those from smart phone cameras. In this notebook we will look at how to load custom images and use them to train neural networks.

We'll be using a dataset of cat and dog photos (https://www.kaggle.com/c/dogs-vs-cats) available from Kaggle. Here are a couple example images:

We'll use this dataset to train a neural network that can differentiate between cats and dogs. These days it doesn't seem like a big accomplishment, but five years ago it was a serious challenge for computer vision systems.

The easiest way to load image data is with `datasets.ImageFolder` from `torchvision` (documentation (http://pytorch.org/docs/master/torchvision/datasets.html#imagefolder)). In general you'll use `ImageFolder`:

```
dataset = datasets.ImageFolder('path/to/data', transform=transforms)
```

where `'path/to/data'` is the file path to the data directory and `transforms` is a list of processing steps built with the `transforms` (http://pytorch.org/docs/master/torchvision/transforms.html) module from `torchvision`. ImageFolder expects the files and directories to be constructed the following way:

```
root/dog/xxx.png
root/dog/xxy.png
root/dog/xxz.png

root/cat/123.png
root/cat/nsdf3.png
root/cat/asd932_.png
```

where each class has its own directory (`cat` and `dog`) for the images.

This structure important because **the images are then labeled with the class taken from the directory name.**

So here, the image `123.png` would be loaded with the class label `cat`. You can download the dataset already structured like this from here (https://s3.amazonaws.com/content.udacity-data.com/nd089/Cat_Dog_data.zip). I've also split it into a training set and test set.

However, if you want to learn how to download the data for yourself, read on:

Explore the data

Let's start by downloading our example data, a .zip of 2,000 JPG pictures of cats and dogs, and extracting it locally in `/tmp`.

NOTE: The 2,000 images used in this exercise are excerpted from the "Dogs vs. Cats" dataset (https://www.kaggle.com/c/dogs-vs-cats/data) available on Kaggle, which contains 25,000 images. Here, we use a subset of the full dataset to decrease training time for educational purposes. But be careful, `/tmp` is a temporary directory and files will be deleted from `/tmp` upon reboot.

In [4]:
```
!wget --no-check-certificate \
    https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip \
    -O /tmp/cats_and_dogs_filtered.zip
```

```
--2018-12-13 14:52:46--  https://storage.googleapis.com/mledu-datasets/cats_and_
dogs_filtered.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 172.217.23.176
Connecting to storage.googleapis.com (storage.googleapis.com)|172.217.23.176|:44
3... connected.
HTTP request sent, awaiting response... 200 OK
Length: 68606236 (65M) [application/zip]
Saving to: '/tmp/cats_and_dogs_filtered.zip'

/tmp/cats_and_dogs_ 100%[===================>]  65.43M  1.76MB/s    in 44s

2018-12-13 14:53:31 (1.48 MB/s) - '/tmp/cats_and_dogs_filtered.zip' saved [68606
236/68606236]
```

In [5]:
```
import os
import zipfile

local_zip = '/tmp/cats_and_dogs_filtered.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('/tmp')
zip_ref.close()
```

The contents of the .zip are extracted to the base directory `/tmp/cats_and_dogs_filtered`, which contains `train` and `validation` subdirectories for the training and validation datasets, which in turn each contain `cats` and `dogs` subdirectories. Let's define each of these directories:

In [6]:
```
base_dir = '/tmp/cats_and_dogs_filtered'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')

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

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

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

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

Now, let's see what the filenames look like in the `cats` and `dogs` `train` directories (file naming conventions are the same in the `validation` directory):

In [8]:
```
train_cat_fnames = os.listdir(train_cats_dir)
print(train_cat_fnames[:10])

train_dog_fnames = os.listdir(train_dogs_dir)
train_dog_fnames.sort()
print(train_dog_fnames[:10])
```

```
['cat.952.jpg', 'cat.946.jpg', 'cat.6.jpg', 'cat.749.jpg', 'cat.991.jpg', 'cat.9
85.jpg', 'cat.775.jpg', 'cat.761.jpg', 'cat.588.jpg', 'cat.239.jpg']
['dog.0.jpg', 'dog.1.jpg', 'dog.10.jpg', 'dog.100.jpg', 'dog.101.jpg', 'dog.102.
jpg', 'dog.103.jpg', 'dog.104.jpg', 'dog.105.jpg', 'dog.106.jpg']
```

Let's find out the total number of cat and dog images in the `train` and `validation` directories:

In [10]:
```
print('total training cat images:', len(os.listdir(train_cats_dir)))
print('total training dog images:', len(os.listdir(train_dogs_dir)))
print('total validation cat images:', len(os.listdir(validation_cats_dir)))
print('total validation dog images:', len(os.listdir(validation_dogs_dir)))
```

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

In [11]:
```
%matplotlib inline

import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# Parameters for our graph; we'll output images in a 4x4 configuration
nrows = 4
ncols = 4

# Index for iterating over images
pic_index = 0
```

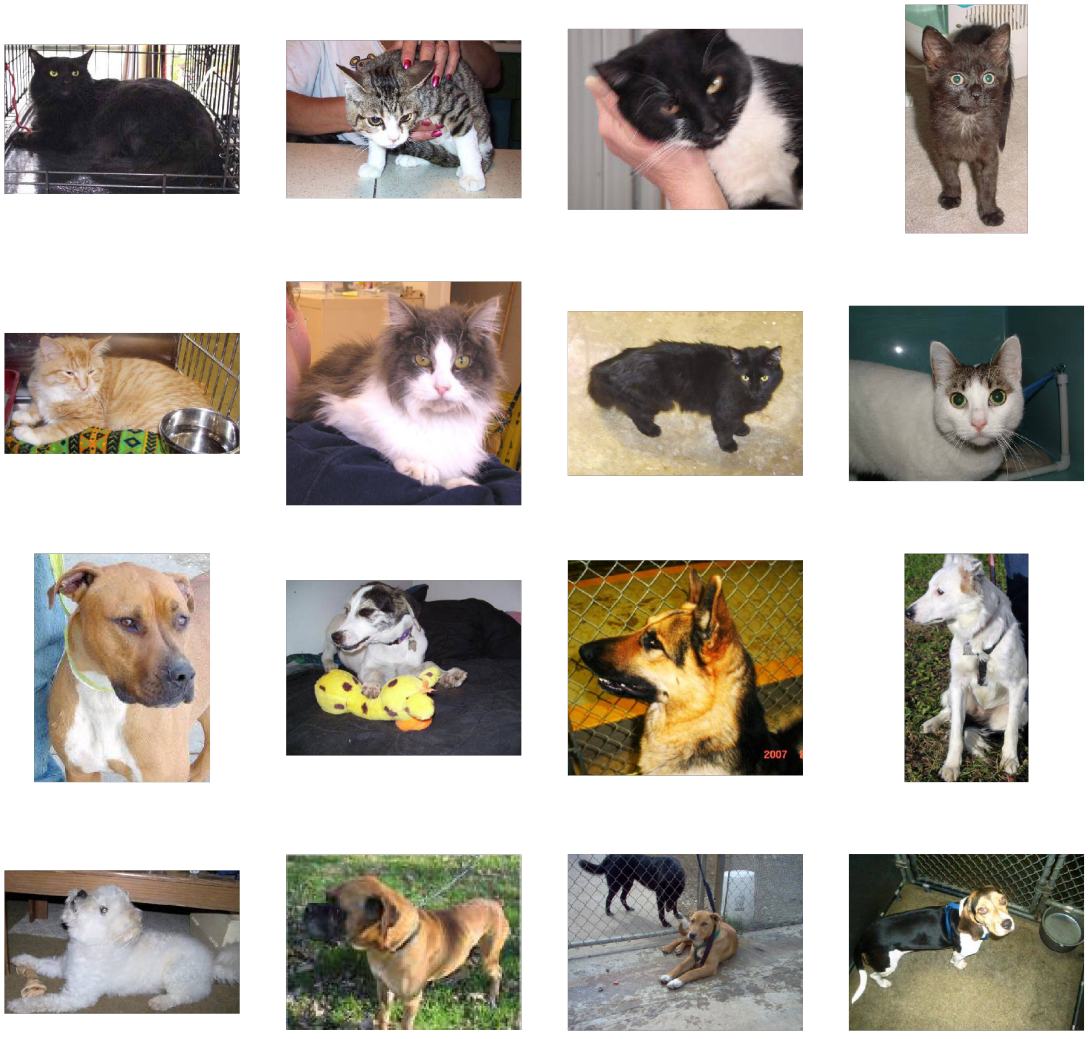
In [13]:
```
# Set up matplotlib fig and size it to fit 4x4 pics
fig = plt.gcf()
fig.set_size_inches(ncols * 4, nrows * 4)

pic_index += 8
next_cat_pix = [os.path.join(train_cats_dir, fname)
                for fname in train_cat_fnames[pic_index-8:pic_index]]
next_dog_pix = [os.path.join(train_dogs_dir, fname)
                for fname in train_dog_fnames[pic_index-8:pic_index]]

for i, img_path in enumerate(next_cat_pix+next_dog_pix):
    # Set up subplot; subplot indices start at 1
    sp = plt.subplot(nrows, ncols, i + 1)
    sp.axis('Off') # Don't show axes (or gridlines)

    img = mpimg.imread(img_path)
    plt.imshow(img)

plt.show()
```



Transforms

When you load in the data with `ImageFolder`, you'll need to define some transforms. For example, the images are different sizes but we'll need them to all be the same size for training. You can either resize them with `transforms.Resize()` or crop with `transforms.CenterCrop()`, `transforms.RandomResizedCrop()`, etc. We will also need to convert the images to PyTorch tensors with `transforms.ToTensor()`. Typically you'll combine these transforms into a pipeline with `transforms.Compose()`, which accepts a list of transforms and runs them in sequence. It looks something like this to scale, then crop, then convert to a tensor:

```
transforms = transforms.Compose([transforms.Resize(255),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor()])
```

There are plenty of transforms available, I'll cover more in a bit and you can read through the documentation (http://pytorch.org/docs/master/torchvision/transforms.html).

Data Loaders

With the `ImageFolder` loaded, you have to pass it to a `DataLoader` (http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader). The `DataLoader` takes a dataset (such as you would get from `ImageFolder`) and returns batches of images and the corresponding labels. You can set various parameters like the batch size and if the data is shuffled after each epoch.

```
dataloader = torch.utils.data.DataLoader(dataset, batch_size=32, shuffle=True)
```

Here `dataloader` is a generator (https://jeffknupp.com/blog/2013/04/07/improve-your-python-yield-and-generators-explained/). To get data out of it, you need to loop through it or convert it to an iterator and call `next()`.

```
# Looping through it, get a batch on each loop
for images, labels in dataloader:
    pass

# Get one batch
images, labels = next(iter(dataloader))
```

> **Exercise:** Load images from the `Cat_Dog_data/train` or `tmp` folder, define a few transforms, then build the dataloader.

In [67]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
import matplotlib.pyplot as plt
import torch
from torchvision import datasets, transforms
import helper
from torch import nn, optim
import torch.nn.functional as F
import fc_model
```

In [35]:
```
train_dir
```

Out[35]:
```
'/tmp/cats_and_dogs_filtered/train'
```

In [36]:
```
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Resize(255),
                                  transforms.CenterCrop(224)])
```

In [48]:
```
# Download and load the training data
trainset = datasets.ImageFolder(train_dir, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
```

In [49]:
```
images, labels = next(iter(dataloader))
```

In [55]:
```
sample_ix = 20
helper.imshow(images[sample_ix], normalize=False)
labels[sample_ix]
```

Out[55]: tensor(1)

Data Augmentation

A common strategy for training neural networks is to introduce randomness in the input data itself. For example, you can randomly rotate, mirror, scale, and/or crop your images during training. This will help your network generalize as it's seeing the same images but in different locations, with different sizes, in different orientations, etc.

To randomly rotate, scale and crop, then flip your images you would define your transforms like this:

```
train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.RandomResizedCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.5, 0.5, 0.5],
                                                            [0.5, 0.5, 0.5])])
```

You'll also typically want to normalize images with `transforms.Normalize`. You pass in a list of means and list of standard deviations, then the color channels are normalized like so

```
input[channel] = (input[channel] - mean[channel]) / std[channel]
```

Subtracting `mean` centers the data around zero and dividing by `std` squishes the values to be between -1 and 1. Normalizing helps keep the network weights near zero which in turn makes backpropagation more stable. Without normalization, networks will tend to fail to learn.

You can find a list of all the available transforms here (http://pytorch.org/docs/0.3.0/torchvision/transforms.html). When you're testing however, you'll want to use images that aren't altered other than normalizing. So, for validation/test images, you'll typically just resize and crop.

> **Exercise:** Define transforms for training data and testing data below. Leave off normalization for now.

In [73]:
```
data_dir = base_dir

# TODO: Define transforms for the training data and testing data
train_transforms = transforms.Compose([transforms.RandomRotation(15),
                                       transforms.RandomResizedCrop(200),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor()])

test_transforms = transforms.Compose([transforms.RandomRotation(10),
                                       transforms.RandomResizedCrop(200),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.RandomVerticalFlip(),
                                       transforms.ToTensor()])


# Pass transforms in here, then run the next cell to see how the transforms look
train_data = datasets.ImageFolder(data_dir + '/train', transform=train_transforms)
test_data = datasets.ImageFolder(data_dir + '/validation', transform=test_transfor
ms)

trainloader = torch.utils.data.DataLoader(train_data, batch_size=32)
testloader = torch.utils.data.DataLoader(test_data, batch_size=32)
```

In [74]:
```
# change this to the trainloader or testloader
data_iter = iter(testloader)

images, labels = next(data_iter)
fig, axes = plt.subplots(figsize=(10,4), ncols=4)
for ii in range(4):
    ax = axes[ii]
    helper.imshow(images[ii], ax=ax, normalize=False)
```

In [75]:
```
images.shape
```

Out[75]:
```
torch.Size([32, 3, 200, 200])
```

At this point you should be able to load data for training and testing. Now, you should try building a network that can classify cats vs dogs. This is quite a bit more complicated than before with the MNIST and Fashion-MNIST datasets. To be honest, you probably won't get it to work with a fully-connected network, no matter how deep. These images have three color channels and at a higher resolution (so far you've seen 28x28 images which are tiny).

In the next part, we will show you how to use a pre-trained network to build a model that can actually solve this problem.

In [79]:
```
model = fc_model.Network(784, 2, [512, 256, 128, 64])
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

In [80]:
```
model
```

Out[80]:
```
Network(
  (hidden_layers): ModuleList(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): Linear(in_features=512, out_features=256, bias=True)
    (2): Linear(in_features=256, out_features=128, bias=True)
    (3): Linear(in_features=128, out_features=64, bias=True)
  )
  (output): Linear(in_features=64, out_features=2, bias=True)
  (dropout): Dropout(p=0.5)
)
```

In [81]:
```
fc_model.train(model, trainloader, testloader, criterion, optimizer, epochs=3)
```

```
Epoch: 1/3..  Training Loss: 7.949..  Test Loss: 1.568..  Test Accuracy: 0.488
Epoch: 2/3..  Training Loss: 0.937..  Test Loss: 0.696..  Test Accuracy: 0.512
Epoch: 2/3..  Training Loss: 0.683..  Test Loss: 0.695..  Test Accuracy: 0.512
Epoch: 3/3..  Training Loss: 0.747..  Test Loss: 0.695..  Test Accuracy: 0.512
```

In [ ]:
```

```