Image classification with Convolutional Neural Networks

Welcome to the first week of the second deep learning certificate! We're going to use convolutional neural networks (CNNs) to allow our computer to see - something that is only possible thanks to deep learning.

Introduction to our first task: 'Dogs vs Cats'

We're going to try to create a model to enter the Dogs vs Cats competition at Kaggle. There are 25,000 labelled dog and cat photos available for training, and 12,500 in the test set that we have to try to label for this competition. According to the Kaggle web-site, when this competition was launched (end of 2013): "State of the art: The current literature suggests machine classifiers can score above 80% accuracy on this task". So if we can beat 80%, then we will be at the cutting edge as of 2013!

In [1]:
```
# Put these at the top of every notebook, to get automatic reloading and inline plotting
%reload_ext autoreload
%autoreload 2
%matplotlib inline
```

Here we import the libraries we need. We'll learn about what each does during the course.

In [2]:
```
# This file contains all the main external libs we'll use
from fastai.imports import *
```

In [3]:
```
from fastai.transforms import *
from fastai.conv_learner import *
from fastai.model import *
from fastai.dataset import *
from fastai.sgdr import *
from fastai.plots import *
```

In [4]:
```
PATH = "data/dogscats/"
```

In [5]:
```
sz=224
```

Extra steps if NOT using Crestle (e.g. if you're using AWS or your own deep learning box)

The dataset is available at http://files.fast.ai/data/dogscats.zip (http://files.fast.ai/data/dogscats.zip). You can download it directly on your server by running the following line in your terminal. `wget http://files.fast.ai/data/dogscats.zip`. You should put the data in a subdirectory of this notebook's directory, called `data/`.

Extra steps if using Crestle

Crestle has the datasets required for fast.ai in /datasets, so we'll create symlinks to the data we want for this competition. (NB: we can't write to /datasets, but we need a place to store temporary files, so we create our own writable directory to put the symlinks in, and we also take advantage of Crestle's `/cache/` faster temporary storage space.)

In [6]:
```
# os.makedirs('data/dogscats/models', exist_ok=True)

# !ln -s /datasets/fast.ai/dogscats/train {PATH}
# !ln -s /datasets/fast.ai/dogscats/test {PATH}
# !ln -s /datasets/fast.ai/dogscats/valid {PATH}

# os.makedirs('/cache/tmp', exist_ok=True)
# !ln -fs /cache/tmp {PATH}
```

In [7]:
```
os.makedirs('/cache/tmp', exist_ok=True)
!ln -fs /cache/tmp {PATH}
```

```
'ln' is not recognized as an internal or external command,
operable program or batch file.
```

First look at cat pictures

Our library will assume that you have *train* and *valid* directories. It also assumes that each dir will have subdirs for each class you wish to recognize (in this case, 'cats' and 'dogs').

In [8]:
```
os.listdir(PATH)
```

Out[8]:
```
['models', 'sample', 'test', 'test1', 'tmp', 'train', 'valid']
```

In [9]:
```
os.listdir(PATH + "valid")
```

Out[9]:
```
['cats', 'dogs']
```

In [10]:
```
files = os.listdir(PATH + "valid/cats")
files
```

```
Out[10]: ['cat.1001.jpg',
          'cat.10016.jpg',
          'cat.10026.jpg',
          'cat.10048.jpg',
          'cat.10050.jpg',
          'cat.10064.jpg',
          'cat.10071.jpg',
          'cat.10091.jpg',
          'cat.10103.jpg',
          'cat.10104.jpg',
          'cat.10107.jpg',
          'cat.10127.jpg',
          'cat.10131.jpg',
          'cat.10133.jpg',
          'cat.10145.jpg',
          'cat.10152.jpg',
          'cat.10154.jpg',
          'cat.10158.jpg',
          'cat.10166.jpg',
          'cat.10204.jpg',
          'cat.10221.jpg',
          'cat.10265.jpg',
          'cat.10267.jpg',
          'cat.10293.jpg',
          'cat.103.jpg',
          'cat.10304.jpg',
          'cat.1031.jpg',
          'cat.10335.jpg',
          'cat.10337.jpg',
          'cat.10349.jpg',
          'cat.10356.jpg',
          'cat.10367.jpg',
          'cat.10369.jpg',
          'cat.10383.jpg',
          'cat.10434.jpg',
          'cat.1045.jpg',
          'cat.10468.jpg',
          'cat.1047.jpg',
          'cat.10471.jpg',
          'cat.10483.jpg',
          'cat.10485.jpg',
          'cat.10531.jpg',
          'cat.10536.jpg',
          'cat.10547.jpg',
          'cat.10553.jpg',
          'cat.10562.jpg',
          'cat.10564.jpg',
          'cat.10566.jpg',
          'cat.10580.jpg',
          'cat.10588.jpg',
          'cat.10597.jpg',
          'cat.10601.jpg',
          'cat.10602.jpg',
          'cat.10636.jpg',
          'cat.10649.jpg',
          'cat.10660.jpg',
          'cat.10667.jpg',
          'cat.10676.jpg',
          'cat.1068.jpg',
          'cat.10712.jpg',
          'cat.10714.jpg',
          'cat.10729.jpg',
          'cat.10735.jpg',
          'cat.10748.jpg',
          'cat.10749.jpg',
          'cat.10773.jpg',
          'cat.10777.jpg',
          'cat.10782.jpg',
          'cat.10786.jpg',
```

In [11]:
```
img = plt.imread(f'{PATH}valid/cats/{files[0]}')
plt.imshow(img);
```

Here is how the raw data looks like

In [12]:
```
img.shape
```

Out[12]:
```
(499, 336, 3)
```

In [13]:
```
img[:4,:4]
```

Out[13]:
```
array([[[60, 58, 10],
        [60, 57, 14],
        [61, 56, 18],
        [63, 54, 23]],

       [[56, 54,  6],
        [56, 53, 10],
        [57, 52, 14],
        [60, 51, 20]],

       [[52, 49,  4],
        [52, 49,  6],
        [53, 48, 10],
        [56, 47, 16]],

       [[50, 47,  2],
        [50, 47,  4],
        [51, 45,  9],
        [53, 44, 13]]], dtype=uint8)
```

Our first model: quick start

We're going to use a **pre-trained** model, that is, a model created by some one else to solve a different problem. Instead of building a model from scratch to solve a similar problem, we'll use a model trained on ImageNet (1.2 million images and 1000 classes) as a starting point. The model is a Convolutional Neural Network (CNN), a type of Neural Network that builds state-of-the-art models for computer vision. We'll be learning all about CNNs during this course.

We will be using the **resnet34** model. resnet34 is a version of the model that won the 2015 ImageNet competition. Here is more info on resnet models (https://github.com/KaimingHe/deep-residual-networks). We'll be studying them in depth later, but for now we'll focus on using them effectively.

Here's how to train and evalulate a *dogs vs cats* model in 3 lines of code, and under 20 seconds:

In [14]:
```
# Uncomment the below if you need to reset your precomputed activations
#os.removedirs(PATH + "tmp")
```

In [15]:
```
arch=resnet34
data = ImageClassifierData.from_paths(PATH, tfms=tfms_from_model(arch, sz))
learn = ConvLearner.pretrained(arch, data, precompute=True)
learn.fit(0.01, 3)
```

```
[ 0.       0.04292  0.0246   0.99219]
[ 1.       0.03538  0.02255  0.99365]
[ 2.       0.03308  0.02541  0.99121]
```

How good is this model? Well, as we mentioned, prior to this competition, the state of the art was 80% accuracy. But the competition resulted in a huge jump to 98.9% accuracy, with the author of a popular deep learning library winning the competition. Extraordinarily, less than 4 years later, we can now beat that result in seconds! Even last year in this same course, our initial model had 98.3% accuracy, which is nearly double the error we're getting just a year later, and that took around 10 minutes to compute.

Analyzing results: looking at pictures

As well as looking at the overall metrics, it's also a good idea to look at examples of each of:

1. A few correct labels at random
2. A few incorrect labels at random
3. The most correct labels of each class (ie those with highest probability that are correct)
4. The most incorrect labels of each class (ie those with highest probability that are incorrect)
5. The most uncertain labels (ie those with probability closest to 0.5).

In [16]:
```
# This is the label for a val data
data.val_y
```

Out[16]:
```
array([0, 0, 0, ..., 1, 1, 1])
```

In [17]:
```
# from here we know that 'cats' is label 0 and 'dogs' is label 1.
data.classes
```

Out[17]:
```
['cats', 'dogs']
```

In [18]:
```
# this gives prediction for validation set. Predictions are in log scale
log_preds = learn.predict()
log_preds.shape
```

Out[18]:
```
(2000, 2)
```

In [19]:
```
log_preds[:10]
```

Out[19]:
```
array([[ -0.00009,  -9.30965],
       [ -0.00001, -11.18532],
       [ -0.     , -12.97197],
       [ -0.00014,  -8.86591],
       [ -0.00005,  -9.86501],
       [ -0.00001, -11.79144],
       [ -0.     , -12.54868],
       [ -0.00001, -11.36884],
       [ -0.00001, -11.50155],
       [ -0.00001, -11.99359]], dtype=float32)
```

In [20]:
```
preds = np.argmax(log_preds, axis=1)  # from log probabilities to 0 or 1
probs = np.exp(log_preds[:,1])        # pr(dog)
```

In [21]:
```
def rand_by_mask(mask): return np.random.choice(np.where(mask)[0], 4, replace=Fa
lse)
def rand_by_correct(is_correct): return rand_by_mask((preds == data.val_y)==is_c
orrect)
```

In [22]:
```
def plot_val_with_title(idxs, title):
    imgs = np.stack([data.val_ds[x][0] for x in idxs])
    title_probs = [probs[x] for x in idxs]
    print(title)
    return plots(data.val_ds.denorm(imgs), rows=1, titles=title_probs)
```

In [23]:
```
def plots(ims, figsize=(12,6), rows=1, titles=None):
    f = plt.figure(figsize=figsize)
    for i in range(len(ims)):
        sp = f.add_subplot(rows, len(ims)//rows, i+1)
        sp.axis('Off')
        if titles is not None: sp.set_title(titles[i], fontsize=16)
        plt.imshow(ims[i])
```

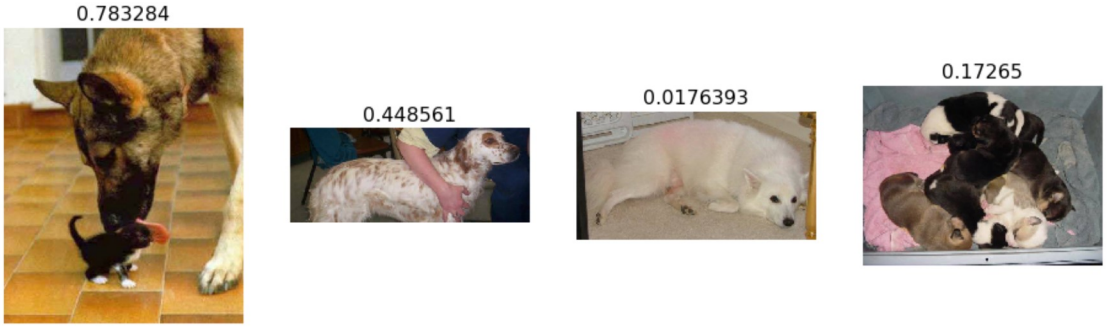
In [24]:
```
def load_img_id(ds, idx): return np.array(PIL.Image.open(PATH+ds.fnames[idx]))

def plot_val_with_title(idxs, title):
    imgs = [load_img_id(data.val_ds,x) for x in idxs]
    title_probs = [probs[x] for x in idxs]
    print(title)
    return plots(imgs, rows=1, titles=title_probs, figsize=(16,8))
```

In [25]:
```
# 1. A few correct labels at random
plot_val_with_title(rand_by_correct(True), "Correctly classified")
```

```
Correctly classified
```

In [26]:
```
# 2. A few incorrect labels at random
plot_val_with_title(rand_by_correct(False), "Incorrectly classified")
```

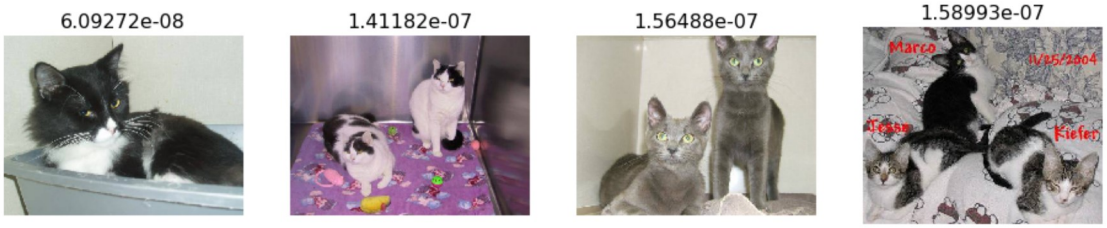
```
Incorrectly classified
```



In [27]:
```
def most_by_mask(mask, mult):
    idxs = np.where(mask)[0]
    return idxs[np.argsort(mult * probs[idxs])[:4]]

def most_by_correct(y, is_correct):
    mult = -1 if (y==1)==is_correct else 1
    return most_by_mask((preds == data.val_y)==is_correct & (data.val_y == y), m
ult)
```

In [28]:
```
plot_val_with_title(most_by_correct(0, True), "Most correct cats")
```

```
Most correct cats
```

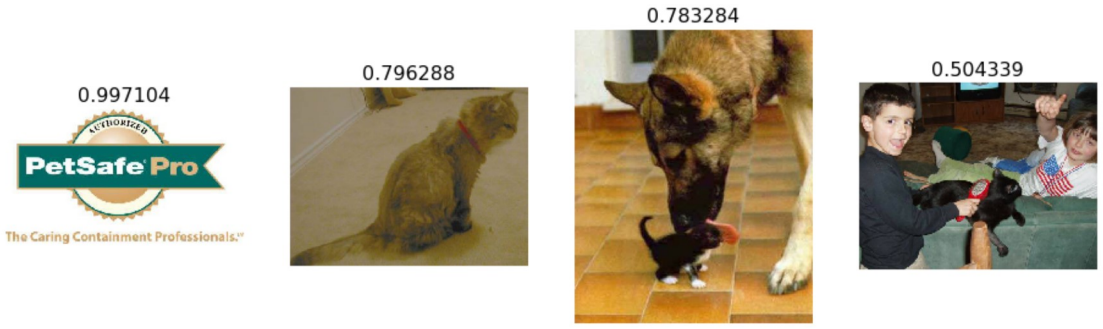


In [29]:
```
plot_val_with_title(most_by_correct(1, True), "Most correct dogs")
```

```
Most correct dogs
```

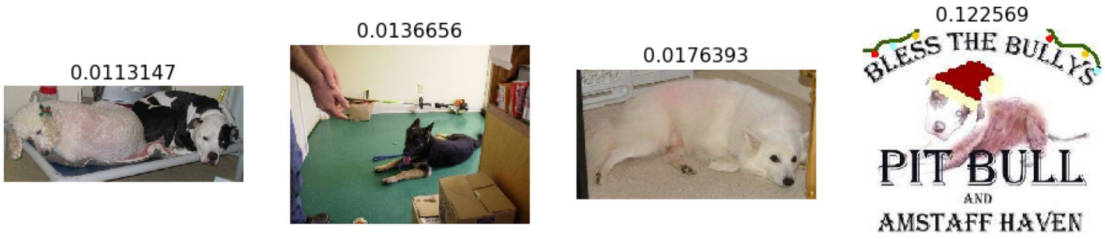
In [30]:
```
plot_val_with_title(most_by_correct(0, False), "Most incorrect cats")
```

```
Most incorrect cats
```



In [31]:
```
plot_val_with_title(most_by_correct(1, False), "Most incorrect dogs")
```

```
Most incorrect dogs
```



In [32]:
```
most_uncertain = np.argsort(np.abs(probs -0.5))[:4]
plot_val_with_title(most_uncertain, "Most uncertain predictions")
```

```
Most uncertain predictions
```

Choosing a learning rate

The *learning rate* determines how quickly or how slowly you want to update the *weights* (or *parameters*). Learning rate is one of the most difficult parameters to set, because it significantly affect model performance.

The method `learn.lr_find()` helps you find an optimal learning rate. It uses the technique developed in the 2015 paper Cyclical Learning Rates for Training Neural Networks (http://arxiv.org/abs/1506.01186), where we simply keep increasing the learning rate from a very small value, until the loss starts decreasing. We can plot the learning rate across batches to see what this looks like.

We first create a new learner, since we want to know how to set the learning rate for a new (untrained) model.

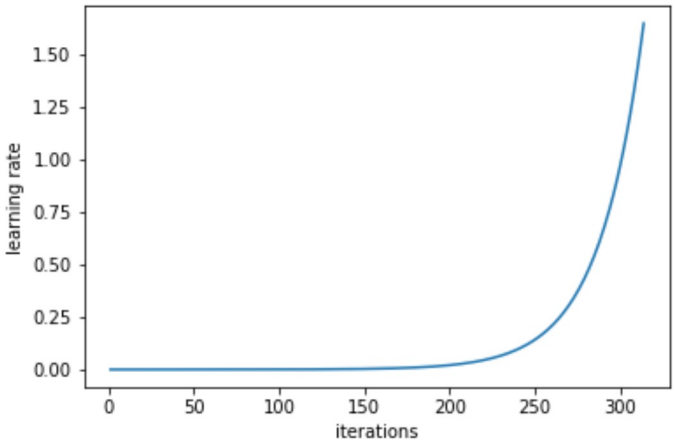
In [33]:
```
learn = ConvLearner.pretrained(arch, data, precompute=True)
```

In [34]:
```
lrf=learn.lr_find()
```

```
 86%|████████████████████████████████████████████████████████████████████████████▍  | 309
/360 [00:04<00:00, 67.18it/s, loss=0.47]
```

Our `learn` object contains an attribute `sched` that contains our learning rate scheduler, and has some convenient plotting functionality including this one:

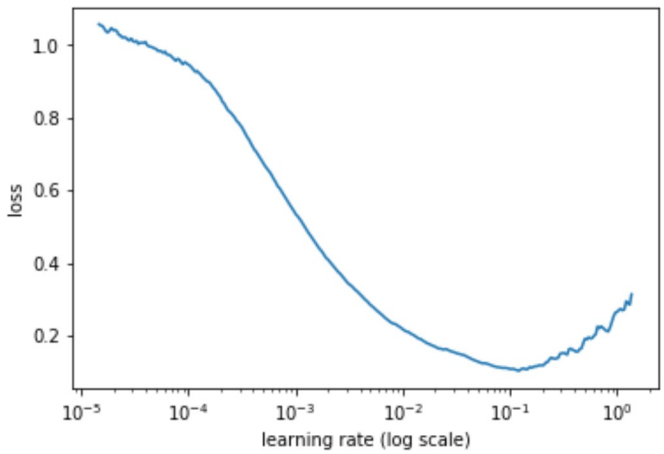
In [35]:
```
learn.sched.plot_lr()
```



Note that in the previous plot *iteration* is one iteration (or *minibatch*) of SGD. In one epoch there are (num_train_samples/num_iterations) of SGD.

We can see the plot of loss versus learning rate to see where our loss stops decreasing:

In [36]:
```
learn.sched.plot()
```



The loss is still clearly improving at lr=1e-2 (0.01), so that's what we use. Note that the optimal learning rate can change as we training the model, so you may want to re-run this function from time to time.

Improving our model

Data augmentation

If you try training for more epochs, you'll notice that we start to *overfit*, which means that our model is learning to recognize the specific images in the training set, rather than generalizaing such that we also get good results on the validation set. One way to fix this is to effectively create more data, through *data augmentation*. This refers to randomly changing the images in ways that shouldn't impact their interpretation, such as horizontal flipping, zooming, and rotating.

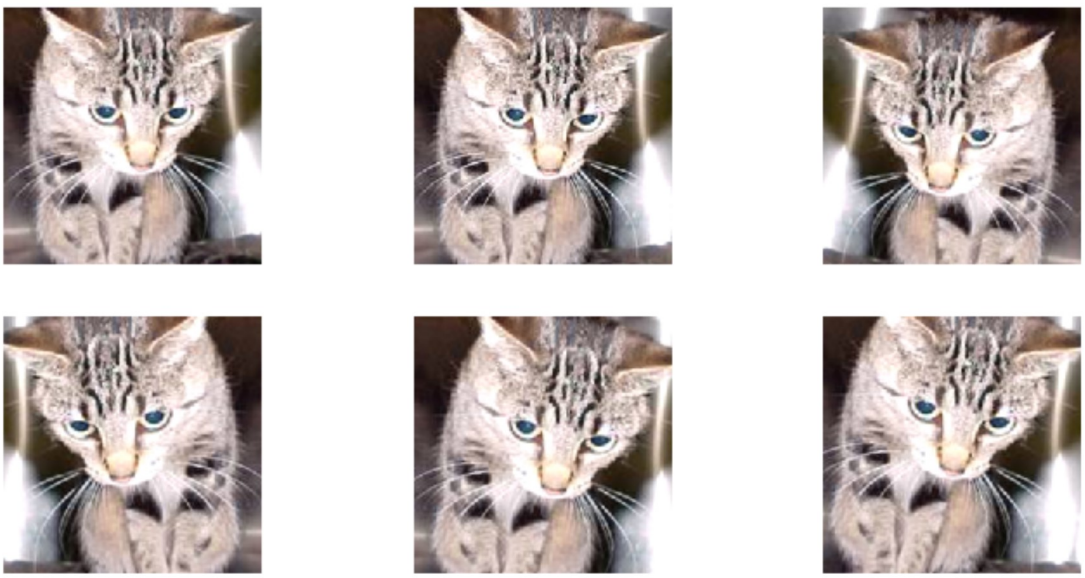
We can do this by passing `aug_tfms` (*augmentation transforms*) to `tfms_from_model`, with a list of functions to apply that randomly change the image however we wish. For photos that are largely taken from the side (e.g. most photos of dogs and cats, as opposed to photos taken from the top down, such as satellite imagery) we can use the pre-defined list of functions `transforms_side_on`. We can also specify random zooming of images up to specified scale by adding the `max_zoom` parameter.

In [37]:
```
tfms = tfms_from_model(resnet34, sz, aug_tfms=transforms_side_on, max_zoom=1.1)
```

In [38]:
```
def get_augs():
    data = ImageClassifierData.from_paths(PATH, bs=2, tfms=tfms, num_workers=1)
    x,_ = next(iter(data.aug_dl))
    return data.trn_ds.denorm(x)[1]
```

In [39]:
```
ims = np.stack([get_augs() for i in range(6)])
```

In [40]:
```
plots(ims, rows=2)
```



Let's create a new `data` object that includes this augmentation in the transforms.

In [41]:
```
data = ImageClassifierData.from_paths(PATH, tfms=tfms)
learn = ConvLearner.pretrained(arch, data, precompute=True)
```

In [ ]:
```
learn.fit(1e-2, 1)
```

```
 30%|███████████████████                                                | 107
/360 [00:33<01:19,  3.17it/s, loss=0.0848]
```

In [43]:
```
learn.precompute=False
```

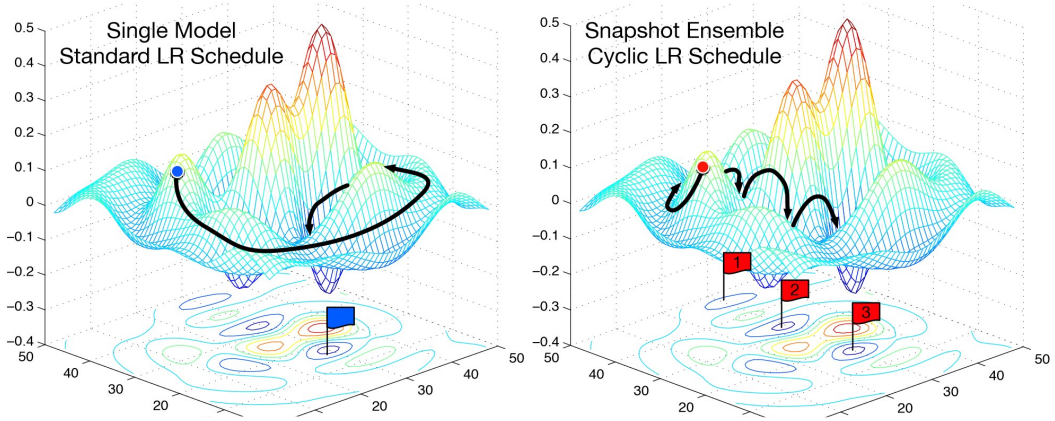
By default when we create a learner, it sets all but the last layer to *frozen*. That means that it's still only updating the weights in the last layer when we call `fit`.

In [44]:
```
learn.fit(1e-2, 3, cycle_len=1)
```

```
[ 0.       0.04793  0.02421  0.99072]
[ 1.       0.03893  0.02363  0.99121]
[ 2.       0.03821  0.02315  0.99072]
```

What is that `cycle_len` parameter? What we've done here is used a technique called *stochastic gradient descent with restarts (SGDR)*, a variant of *learning rate annealing*, which gradually decreases the learning rate as training progresses. This is helpful because as we get closer to the optimal weights, we want to take smaller steps.

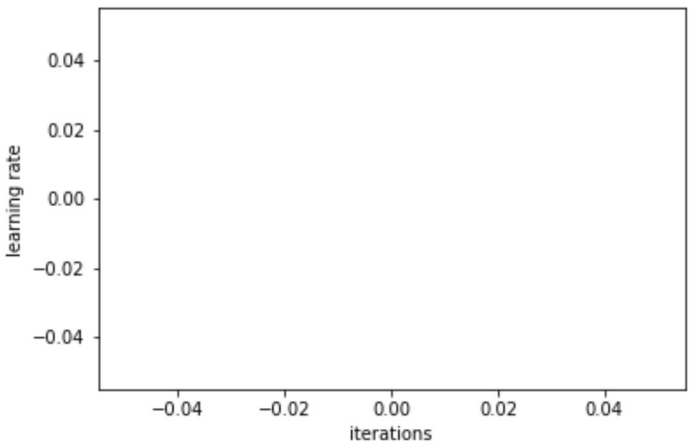
However, we may find ourselves in a part of the weight space that isn't very resilient - that is, small changes to the weights may result in big changes to the loss. We want to encourage our model to find parts of the weight space that are both accurate and stable. Therefore, from time to time we increase the learning rate (this is the 'restarts' in 'SGDR'), which will force the model to jump to a different part of the weight space if the current area is "spikey". Here's a picture of how that might look if we reset the learning rates 3 times (in this paper they call it a "cyclic LR schedule"):



(From the paper Snapshot Ensembles (https://arxiv.org/abs/1704.00109)).

The number of epochs between resetting the learning rate is set by `cycle_len`, and the number of times this happens is refered to as the *number of cycles*, and is what we're actually passing as the 2nd parameter to `fit()`. So here's what our actual learning rates looked like:

In [45]:
```
learn.sched.plot_lr()
```



Our validation loss isn't improving much, so there's probably no point further training the last layer on its own.

Since we've got a pretty good model at this point, we might want to save it so we can load it again later without training it from scratch.

In [46]:
```
learn.save('224_lastlayer')
```

In [47]:
```
learn.load('224_lastlayer')
```

Fine-tuning and differential learning rate annealing

Now that we have a good final layer trained, we can try fine-tuning the other layers. To tell the learner that we want to unfreeze the remaining layers, just call (surprise surprise!) `unfreeze()`.

In [48]:
```
learn.unfreeze()
```

Note that the other layers have *already* been trained to recognize imagenet photos (whereas our final layers where randomly initialized), so we want to be careful of not destroying the carefully tuned weights that are already there.

Generally speaking, the earlier layers (as we've seen) have more general-purpose features. Therefore we would expect them to need less fine-tuning for new datasets. For this reason we will use different learning rates for different layers: the first few layers will be at 1e-4, the middle layers at 1e-3, and our FC layers we'll leave at 1e-2 as before. We refer to this as *differential learning rates*, although there's no standard name for this techique in the literature that we're aware of.

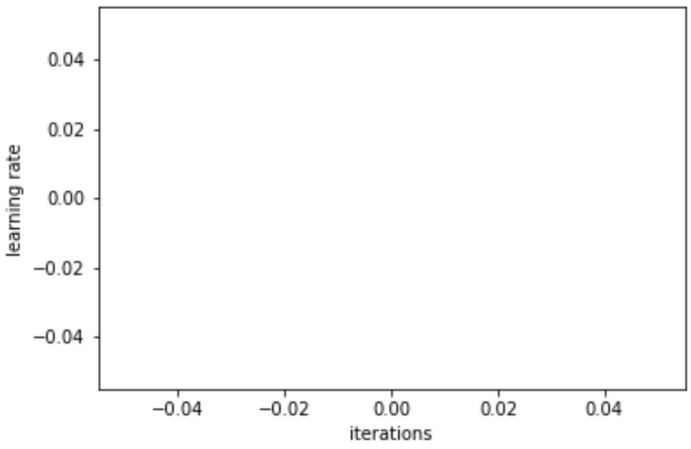
In [49]:
```
lr=np.array([1e-4,1e-3,1e-2])
```

In [50]:
```
learn.fit(lr, 3, cycle_len=1, cycle_mult=2)
```

```
[ 0.       0.04988  0.02791  0.98975]
[ 1.       0.04185  0.01965  0.99219]
[ 2.       0.02879  0.019    0.99268]
[ 3.       0.02441  0.01824  0.99316]
[ 4.       0.02929  0.01777  0.99414]
[ 5.       0.01841  0.01886  0.99268]
[ 6.       0.01947  0.01764  0.99414]
```

Another trick we've used here is adding the `cycle_mult` parameter. Take a look at the following chart, and see if you can figure out what the parameter is doing:

In [51]:
```
learn.sched.plot_lr()
```



Note that's what being plotted above is the learning rate of the *final layers*. The learning rates of the earlier layers are fixed at the same multiples of the final layer rates as we initially requested (i.e. the first layers have 100x smaller, and middle layers 10x smaller learning rates, since we set `lr=np.array([1e-4,1e-3,1e-2])`.

In [52]:
```
learn.save('224_all')
```

In [53]:
```
learn.load('224_all')
```

There is something else we can do with data augmentation: use it at *inference* time (also known as *test* time). Not surprisingly, this is known as *test time augmentation*, or just *TTA*.

TTA simply makes predictions not just on the images in your validation set, but also makes predictions on a number of randomly augmented versions of them too (by default, it uses the original image along with 4 randomly augmented versions). It then takes the average prediction from these images, and uses that. To use TTA on the validation set, we can use the learner's `TTA()` method.

In [54]:
```
log_preds,y = learn.TTA()
preds = np.mean(np.exp(log_preds),0)
```

In [55]:
```
accuracy(preds, y)
```

Out[55]:
```
0.99550000000000005
```

I generally see about a 10-20% reduction in error on this dataset when using TTA at this point, which is an amazing result for such a quick and easy technique!

Analyzing results

Confusion matrix

In [56]:
```
preds = np.argmax(log_preds, axis=1)
probs = np.exp(log_preds[:,1])
```

A common way to analyze the result of a classification model is to use a confusion matrix (http://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/). Scikit-learn has a convenient function we can use for this purpose:

In [57]:
```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y, preds)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-57-00b8d63a04da> in <module>()
      1 from sklearn.metrics import confusion_matrix
----> 2 cm = confusion_matrix(y, preds)

~\Anaconda3\lib\site-packages\sklearn\metrics\classification.py in confusion_m
atrix(y_true, y_pred, labels, sample_weight)
    248
    249     """
--> 250     y_type, y_true, y_pred = _check_targets(y_true, y_pred)
    251     if y_type not in ("binary", "multiclass"):
    252         raise ValueError("%s is not supported" % y_type)

~\Anaconda3\lib\site-packages\sklearn\metrics\classification.py in _check_targ
ets(y_true, y_pred)
     69     y_pred : array or indicator matrix
     70     """
---> 71     check_consistent_length(y_true, y_pred)
     72     type_true = type_of_target(y_true)
     73     type_pred = type_of_target(y_pred)

~\Anaconda3\lib\site-packages\sklearn\utils\validation.py in check_consistent_
length(*arrays)
    202     if len(uniques) > 1:
    203         raise ValueError("Found input variables with inconsistent numb
ers of"
--> 204                          " samples: %r" % [int(l) for l in lengths])
    205
    206

ValueError: Found input variables with inconsistent numbers of samples: [2000,
5]
```

We can just print out the confusion matrix, or we can show a graphical view (which is mainly useful for dependents with a larger number of categories).

In [ ]:
```
plot_confusion_matrix(cm, data.classes)
```

Looking at pictures again

In [ ]:
```
plot_val_with_title(most_by_correct(0, False), "Most incorrect cats")
```

In [ ]:
```
plot_val_with_title(most_by_correct(1, False), "Most incorrect dogs")
```

Review: easy steps to train a world-class image classifier

1. Enable data augmentation, and precompute=True
2. Use `lr_find()` to find highest learning rate where loss is still clearly improving
3. Train last layer from precomputed activations for 1-2 epochs
4. Train last layer with data augmentation (i.e. precompute=False) for 2-3 epochs with cycle_len=1
5. Unfreeze all layers
6. Set earlier layers to 3x-10x lower learning rate than next higher layer
7. Use `lr_find()` again
8. Train full network with cycle_mult=2 until over-fitting

1. Use `lr_find()` to find highest learning rate where loss is still clearly improving
2. Train last layer with data augmentation (i.e. precompute=False) for 2-3 epochs with cycle_len=1
3. Unfreeze all layers
4. Set earlier layers to 3x-10x lower learning rate than next higher layer
5. Train full network with cycle_mult=2 until over-fitting

Understanding the code for our first model

Let's look at the Dogs v Cats code line by line.

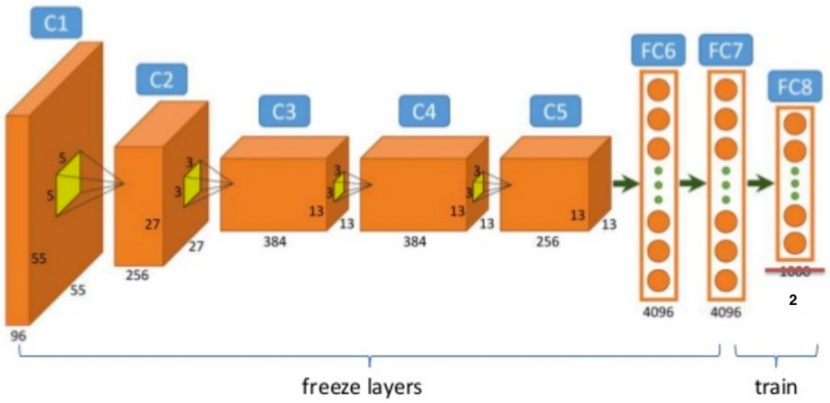
tfms stands for *transformations*. `tfms_from_model` takes care of resizing, image cropping, initial normalization (creating data with (mean,stdev) of (0,1)), and more.

In [ ]:
```
tfms = tfms_from_model(resnet34, sz)
```

We need a **path** that points to the dataset. In this path we will also store temporary data and final results. `ImageClassifierData.from_paths` reads data from a provided path and creates a dataset ready for training.

In [ ]:
```
data = ImageClassifierData.from_paths(PATH, tfms=tfms)
```

`ConvLearner.pretrained` builds *learner* that contains a pre-trained model. The last layer of the model needs to be replaced with the layer of the right dimensions. The pretained model was trained for 1000 classes therfore the final layer predicts a vector of 1000 probabilities. The model for cats and dogs needs to output a two dimensional vector. The diagram below shows in an example how this was done in one of the earliest successful CNNs. The layer "FC8" here would get replaced with a new layer with 2 outputs.



original image (https://image.slidesharecdn.com/practicaldeeplearning-160329181459/95/practical-deep-learning-16-638.jpg)

In [ ]:
```
learn = ConvLearner.pretrained(resnet34, data, precompute=True)
```

Parameters are learned by fitting a model to the data. *Hyparameters* are another kind of parameter, that cannot be directly learned from the regular training process. These parameters express “higher-level” properties of the model such as its complexity or how fast it should learn. Two examples of hyperparameters are the *learning rate* and the *number of epochs*.

During iterative training of a neural network, a *batch* or *mini-batch* is a subset of training samples used in one iteration of Stochastic Gradient Descent (SGD). An *epoch* is a single pass through the entire training set which consists of multiple iterations of SGD.

We can now *fit* the model; that is, use *gradient descent* to find the best parameters for the fully connected layer we added, that can separate cat pictures from dog pictures. We need to pass two hyperameters: the *learning rate* (generally 1e-2 or 1e-3 is a good starting point, we'll look more at this next) and the *number of epochs* (you can pass in a higher number and just stop training when you see it's no longer improving, then re-run it with the number of epochs you found works well.)

In [ ]:
```
learn.fit(1e-2, 1)
```

Analyzing results: loss and accuracy

When we run `learn.fit` we print 3 performance values (see above.) Here 0.03 is the value of the **loss** in the training set, 0.0226 is the value of the loss in the validation set and 0.9927 is the validation accuracy. What is the loss? What is accuracy? Why not to just show accuracy?

Accuracy is the ratio of correct prediction to the total number of predictions.

In machine learning the **loss** function or cost function is representing the price paid for inaccuracy of predictions.

The loss associated with one example in binary classification is given by: `-(y * log(p) + (1-y) * log (1-p))` where `y` is the true label of `x` and `p` is the probability predicted by our model that the label is 1.

In [ ]:
```
def binary_loss(y, p):
    return np.mean(-(y * np.log(p) + (1-y)*np.log(1-p)))
```

In [ ]:
```
acts = np.array([1, 0, 0, 1])
preds = np.array([0.9, 0.1, 0.2, 0.8])
binary_loss(acts, preds)
```

Note that in our toy example above our accuracy is 100% and our loss is 0.16. Compare that to a loss of 0.03 that we are getting while predicting cats and dogs. Exercise: play with `preds` to get a lower loss for this example.

Example: Here is an example on how to compute the loss for one example of binary classification problem. Suppose for an image x with label 1 and your model gives it a prediction of 0.9. For this case the loss should be small because our model is predicting a label 1 with high probability.

`loss = -log(0.9) = 0.10`

Now suppose x has label 0 but our model is predicting 0.9. In this case our loss is should be much larger.

loss = -log(1-0.9) = 2.30

- Exercise: look at the other cases and convince yourself that this make sense.
- Exercise: how would you rewrite `binary_loss` using `if` instead of `*` and `+`?

Why not just maximize accuracy? The binary classification loss is an easier function to optimize.