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

autoreloads new code

important so all plots occur inline after each code cell

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

```
In [2]:  1  # This file contains all the main external libs we'll use
         2  from fastai.imports import *
```

```
In [3]:  1  from fastai.transforms import *
         2  from fastai.conv_learner import *
         3  from fastai.model import *
         4  from fastai.dataset import *
         5  from fastai.sgdr import *
         6  from fastai.plots import *
```

```
In [4]:  1  PATH = "data/dogscats/"
```

```
In [5]:  1  sz=224
```

*size set to 224 because resnet uses 224 * 224 size*

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. 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]:   1  # os.makedirs('data/dogscats/models', exist_ok=True)
          2
          3  # !ln -s /datasets/fast.ai/dogscats/train {PATH}
          4  # !ln -s /datasets/fast.ai/dogscats/test {PATH}
          5  # !ln -s /datasets/fast.ai/dogscats/valid {PATH}
          6
          7  # os.makedirs('/cache/tmp', exist_ok=True)
          8  # !ln -fs /cache/tmp {PATH}
```

```
In [8]:   1  os.makedirs('/cache/tmp', exist_ok=True)
          2  !ln -fs /cache/tmp {PATH}
```

```
---------------------------------------------------------------------------
PermissionError                           Traceback (most recent call last)
<ipython-input-8-9e4f5d3540b7> in <module>()
----> 1 os.makedirs('/cache/tmp', exist_ok=True)
      2 get_ipython().system('ln -fs /cache/tmp {PATH}')

~/src/anaconda3/envs/fastai/lib/python3.6/os.py in makedirs(name, mode, exist_ok)
    208         if head and tail and not path.exists(head):
    209             try:
--> 210                 makedirs(head, mode, exist_ok)
    211             except FileExistsError:
    212                 # Defeats race condition when another thread created the path

~/src/anaconda3/envs/fastai/lib/python3.6/os.py in makedirs(name, mode, exist_ok)
    218             return
    219     try:
--> 220         mkdir(name, mode)
    221     except OSError:
    222         # Cannot rely on checking for EEXIST, since the operating system

PermissionError: [Errno 13] Permission denied: '/cache'
```

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 [7]:
```
1  !ls {PATH}
```
```
models  sample  test  test1  tmp  train  valid
```

In [8]:
```
1  !ls {PATH}valid
```
good practice to have folder names of the things you are trying to classify
```
cats  dogs
```

In [9]:
```
1  files = !ls {PATH}valid/cats | head
2  files
```

Out[9]:
```
['cat.10016.jpg',
 'cat.1001.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']
```

In [10]:
```
1  img = plt.imread(f'{PATH}valid/cats/{files[0]}')
2  plt.imshow(img);
```


Here is how the raw data looks like

In [11]:
```
1  img.shape
```
L, W, Color channels

Out[11]: (198, 179, 3)

In [12]:
```
1  img[:4,:4]
```
Array Slice

```
Out[12]: array([[[ 29,  20,  23],
                 [ 31,  22,  25],
                 [ 34,  25,  28],
                 [ 37,  28,  31]],

                [[ 60,  51,  54],
                 [ 58,  49,  52],
                 [ 56,  47,  50],
                 [ 55,  46,  49]],

                [[ 93,  84,  87],
                 [ 89,  80,  83],
                 [ 85,  76,  79],
                 [ 81,  72,  75]],

                [[104,  95,  98],
                 [103,  94,  97],
                 [102,  93,  96],
                 [102,  93,  96]]], dtype=uint8)
```

(annotations: R Channel, Green Channel, Blue Channel, "Unsigned integer of 8 bits with a max value of 255 and a min value of 0" pointing to 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](#). We'll be studying them in depth later, but for now we'll focus on using them effectively.

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

```
In [6]:   1  # Uncomment the below if you need to reset your precomputed activation
             s
          2  # !rm -rf {PATH}tmp
```

(annotation: "this LR was picked because of the LR finder graph below")

```
In [129]:  1  arch=resnet34
           2  data = ImageClassifierData.from_paths(PATH, tfms=tfms_from_model(arch,
                 sz))
           3  learn = ConvLearner.pretrained(arch, data, precompute=True)
           4  learn.fit(0.01, 3)
```

(annotations: "architecture of CNN, in this case resnet34 is being used", "resnet34 is good for computer vision")

```
A Jupyter Widget

[ 0.       0.04726  0.02807  0.99121]
[ 1.       0.04413  0.02372  0.99072]
[ 2.       0.03454  0.02609  0.9917 ]
```

(annotations: "Epochs = how many times you want the model to go through each of your pictures", "validation set loss images we are NOT looking at but are setting them aside to see how accurate we are", "accuracy", "epochs", "training set loss images we are looking at to make a better classifier")

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 [8]:   1  # This is the label for a val data
          2  data.val_y
```
Validation labels

```
Out[8]:  array([0, 0, 0, ..., 1, 1, 1])
```
0 = cats
1 = dogs

```
In [9]:   1  # from here we know that 'cats' is label 0 and 'dogs' is label 1.
          2  data.classes
```

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

```
In [10]:  1  # this gives prediction for validation set. Predictions are in log sca
             le
          2  log_preds = learn.predict()
          3  log_preds.shape
```

```
Out[10]:  (2000, 2)
```

```
In [11]:  1  log_preds[:10]
```

```
Out[11]:  array([[ -0.02109,  -3.86954],
                 [ -0.00005,  -9.91835],
                 [ -0.00001, -11.1595 ],
                 [ -0.00024,  -8.34448],
                 [ -0.00008,  -9.39493],
                 [ -0.00631,  -5.06912],
                 [ -0.00002, -11.01133],
                 [ -0.00004, -10.16935],
                 [ -0.00003, -10.40165],
                 [ -0.00844,  -4.77932]], dtype=float32)
```

```
In [12]:  1  preds = np.argmax(log_preds, axis=1)   # from log probabilities to 0 or
                1
          2  probs = np.exp(log_preds[:,1])         # pr(dog)
```

```
In [13]:  1  def rand_by_mask(mask): return np.random.choice(np.where(mask)[0], 4, re
```

```
    alse)
  2 def rand_by_correct(is_correct): return rand_by_mask((preds == data.val_
    correct)
```

In [14]:
```
  1 def plot_val_with_title(idxs, title):
  2     imgs = np.stack([data.val_ds[x][0] for x in idxs])
  3     title_probs = [probs[x] for x in idxs]
  4     print(title)
  5     return plots(data.val_ds.denorm(imgs), rows=1, titles=title_probs)
```

In [15]:
```
  1 def plots(ims, figsize=(12,6), rows=1, titles=None):
  2     f = plt.figure(figsize=figsize)
  3     for i in range(len(ims)):
  4         sp = f.add_subplot(rows, len(ims)//rows, i+1)
  5         sp.axis('Off')
  6         if titles is not None: sp.set_title(titles[i], fontsize=16)
  7         plt.imshow(ims[i])
```

In [16]:
```
  1 def load_img_id(ds, idx): return np.array(PIL.Image.open(PATH+ds.fnames
    [idx]))
  2
  3 def plot_val_with_title(idxs, title):
  4     imgs = [load_img_id(data.val_ds,x) for x in idxs]
  5     title_probs = [probs[x] for x in idxs]
  6     print(title)
  7     return plots(imgs, rows=1, titles=title_probs, figsize=(16,8))
```

In [17]:
```
  1 # 1. A few correct labels at random
  2 plot_val_with_title(rand_by_correct(True), "Correctly classified")
```
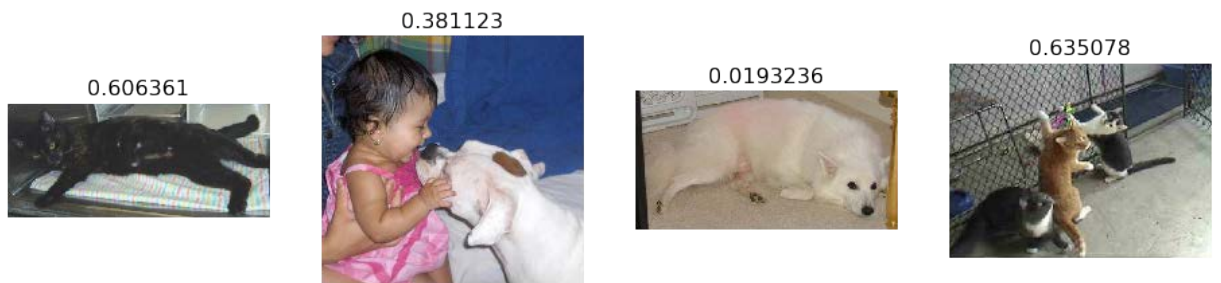
Correctly classified

In [18]:
```
  1 # 2. A few incorrect labels at random
  2 plot_val_with_title(rand_by_correct(False), "Incorrectly classified")
```

Incorrectly classified

0.606361 0.381123 0.0193236 0.635078

```
In [19]:  1  def most_by_mask(mask, mult):
          2      idxs = np.where(mask)[0]
          3      return idxs[np.argsort(mult * probs[idxs])[:4]]
          4
          5  def most_by_correct(y, is_correct):
          6      mult = -1 if (y==1)==is_correct else 1
          7      return most_by_mask((preds == data.val_y)==is_correct & (data.val_y
             ), mult)
```

```
In [20]:  1  plot_val_with_title(most_by_correct(0, True), "Most correct cats")
```

Most correct cats

2.15149e-08 1.28162e-07 1.70913e-07 2.0446e-07

```
In [21]:  1  plot_val_with_title(most_by_correct(1, True), "Most correct dogs")
```

Most correct dogs

1.0 1.0 1.0 1.0

```
In [22]:  1  plot_val_with_title(most_by_correct(0, False), "Most incorrect cats")
```

Most incorrect cats

0.990691   0.927055   0.635078   0.606361

In [23]:
```
1  plot_val_with_title(most_by_correct(1, False), "Most incorrect dogs")
```

Most incorrect dogs

0.0193236   0.0435172   0.105995

probably inaccurate because the classifier is looking at the middle of the picture as GPUs need a square method of looking at things and did not recognize the dog as it did not see the edges where the head of the dog is. Use TTA to improve
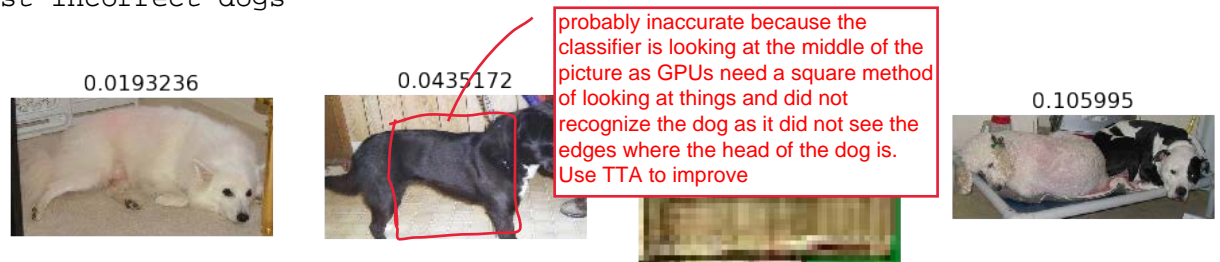
In [24]:
```
1  most_uncertain = np.argsort(np.abs(probs -0.5))[:4]
2  plot_val_with_title(most_uncertain, "Most uncertain predictions")
```

Most uncertain predictions

0.508966   0.482242   0.481406   0.469625

Choosing a learning rate

basic concept of LR = how quickly we will zoom onto the solution
LR too low = take too long
LR too high = overshoot and oscillate
Solution = learning rate finder

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, 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 [49]:
```
1  learn = ConvLearner.pretrained(arch, data, precompute=True)
```

```
In [50]:    1  lrf=learn.lr_find()
```

A Jupyter Widget

82%|■■■■■■■■■| | 295/360 [00:02<00:00, 167.14it/s, loss=0.128]

360 is the number of batches it takes to precompute the activations (~23000 images / 64 batch size) in the train set

Ignore this loss it is not important

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

```
In [51]:    1  learn.sched.plot_lr()
```
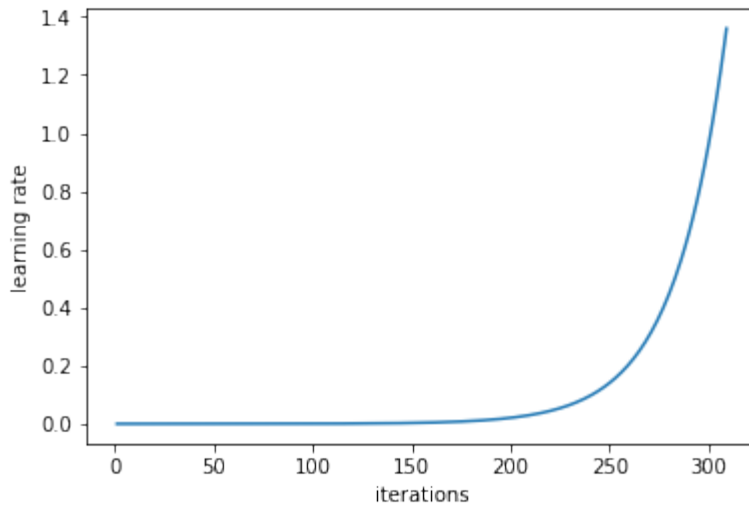

The LR Finder can be considered as a sophisticated form of Grid Search where Grid Search is trying the find the best result for a particular hyper parameter

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 [52]:    1  learn.sched.plot()
```
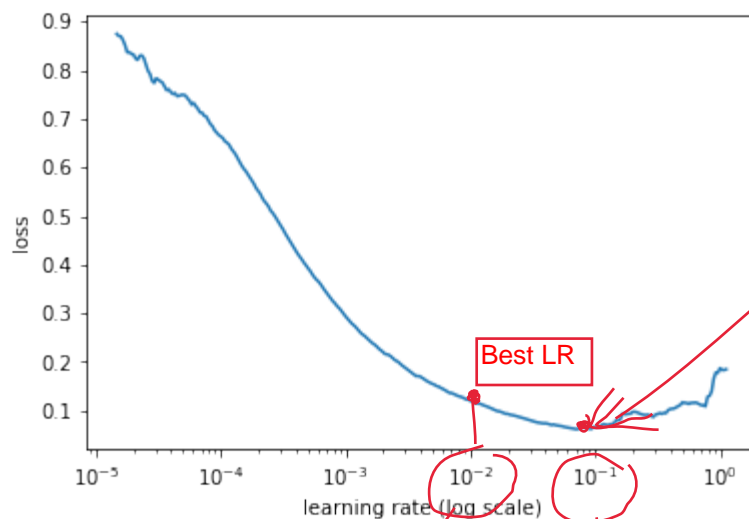

Best LR

lowest portion of graph not the best learning rate as at this point it may be too high so best to have a learning rate that is higher on the curve (ie in this case a lower learning rate 0.01 as opposed to 0,1)

=0.01 or 1e-2

= 0.1 or 1e-1

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 [53]:   1  tfms = tfms_from_model(resnet34, sz, aug_tfms=transforms_side_on, max_z
              om=1.1)
```
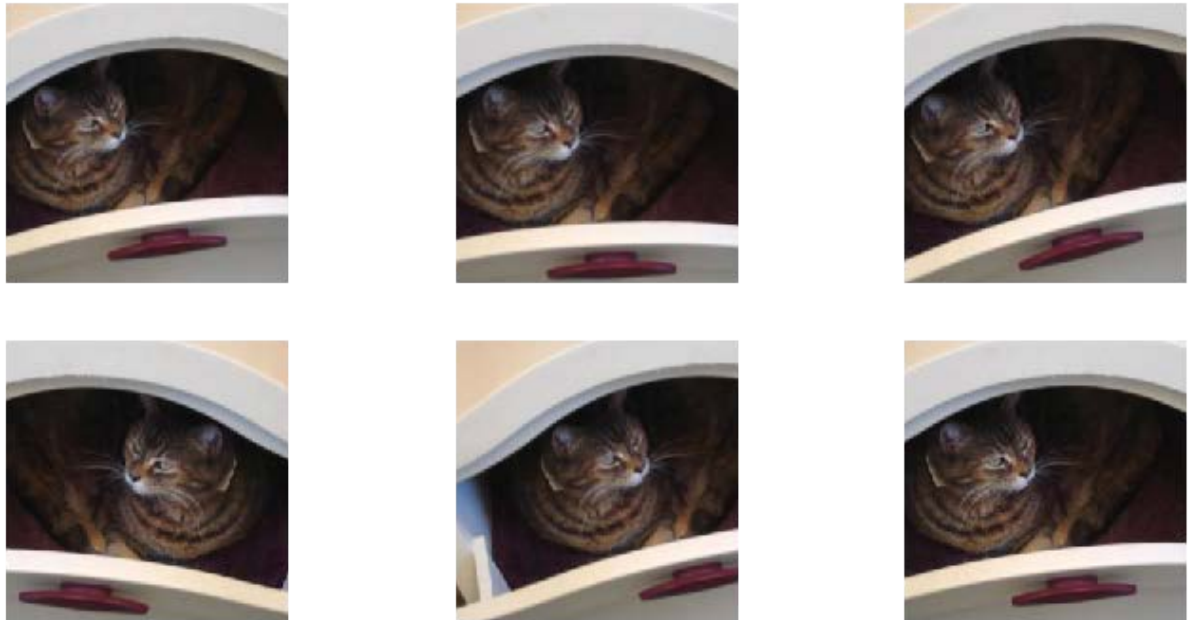
```
In [26]:   1  def get_augs():
           2      data = ImageClassifierData.from_paths(PATH, bs=2, tfms=tfms, num_wo
              kers=1)
           3      x,_ = next(iter(data.aug_dl))
           4      return data.trn_ds.denorm(x)[1]
```

This transform is not creating new data but allows the CCN to look at cats (or what ever picture) from different angle

```
In [30]:   1  ims = np.stack([get_augs() for i in range(6)])
```

```
In [31]:   1  plots(ims, rows=2)
```

Batch Size

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

In [54]:
```
1  data = ImageClassifierData.from_paths(PATH, tfms=tfms)
2  learn = ConvLearner.pretrained(arch, data, precompute=True)
```

precompute activations - this is taking the second from last layer from the pre-trained model

In [55]:
```
1  learn.fit(1e-2, 1)
```

LR same as 0.01

A Jupyter Widget

[0.   0.03651 0.01723 0.99365]

Precompute = Precomputed activations that we feed into the network except for the last layer

In [56]:
```
1  learn.precompute=False
```

In order to use data augmentation precompute needs to be set to False in which case all convolutional layers are frozen

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 [57]:
```
1  learn.fit(1e-2, 3, cycle_len=1)
```

LR is re-set after every epoch so in this case the LR is reset after 1 epoch

A Jupyter Widget

cycle_length is a new feature of DL using socastic gradient descent

[0.   0.03597 0.01879 0.99365]
[1.   0.02605 0.01836 0.99365]
[2.   0.02189 0.0196   0.99316]

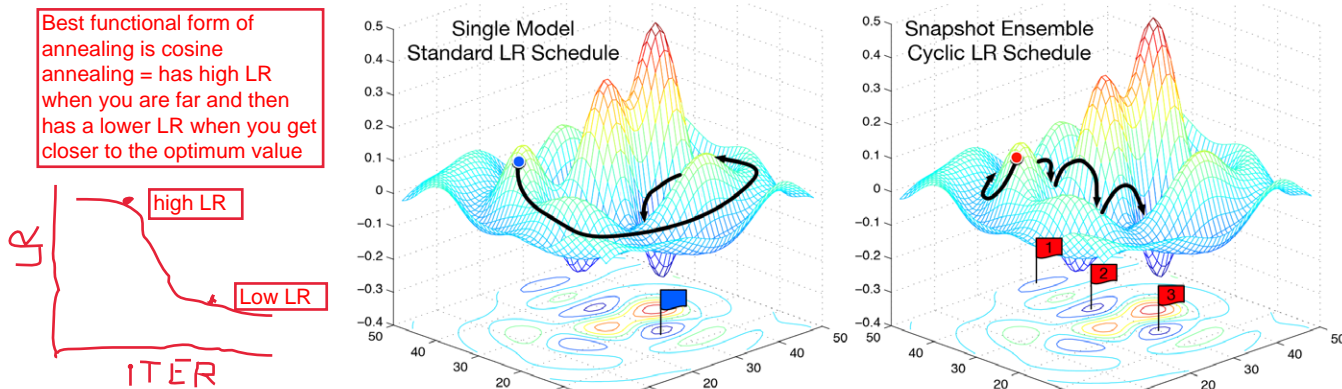*Overfitting = when the Training Loss (TL) is lower than the Validation Loss (VL)*

TL   VL

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.

= decreasing the LR as you train

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
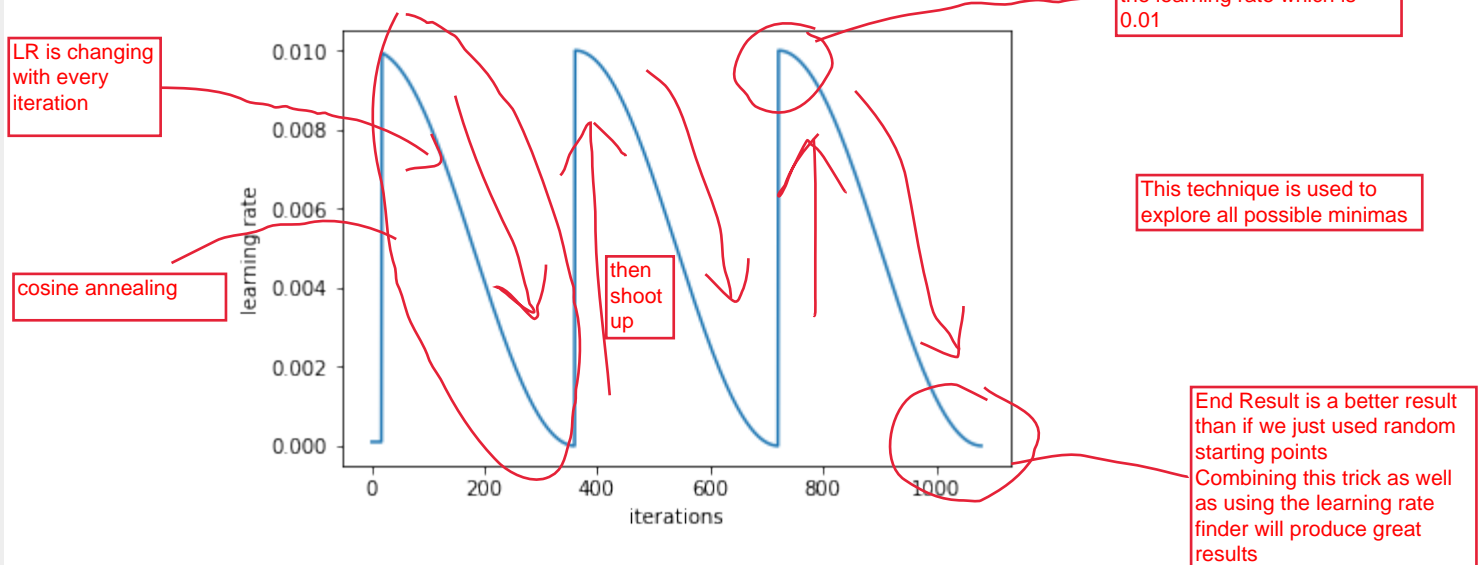
find parts of the weight space that are both accurate and stable. Therefore, from time to time we increase the learning rate (this is the 'restarts' in 'SGDR'), which will force the model to jump to a different part of the weight space if the current area is "spikey". Here's a picture of how that might look if we reset the learning rates 3 times (in this paper they call it a "cyclic LR schedule"):

Best functional form of annealing is cosine annealing = has high LR when you are far and then has a lower LR when you get closer to the optimum value

high LR

Low LR

LR

ITER

(From the paper Snapshot Ensembles).

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 [37]:   1  learn.sched.plot_lr()
```

This peak is set the same as the learning rate which is 0.01

LR is changing with every iteration

cosine annealing

then shoot up

This technique is used to explore all possible minimas

End Result is a better result than if we just used random starting points
Combining this trick as well as using the learning rate finder will produce great results

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.

saves all precomputed activations into the tmp folder and saves all models to the models folder

```
In [31]:   1  learn.save('224_lastlayer')
```

```
In [ ]:    1  learn.load('224_lastlayer')
```

Fine-tuning and differential learning rate annealing

frozen layer = layer that has not been trained

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 [32]:    1  learn.unfreeze()
```

this is now telling the learner that we are ready to change the convolutional pillars
unfreezes all the layers

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.

create an array of learning rates for the layers that we have added on top

```
In [33]:    1  lr=np.array([1e-4,1e-3,1e-2])
```

```
In [34]:    1  learn.fit(lr, 3, cycle_len=1, cycle_mult=2)
```

LR to be used in the middle layers for the more sophisticated convolutional features

```
A Jupyter Widget

[ 0.        0.04538   0.01965   0.99268]
[ 1.        0.03385   0.01807   0.99268]
[ 2.        0.03194   0.01714   0.99316]
[ 3.        0.0358    0.0166    0.99463]
[ 4.        0.02157   0.01504   0.99463]
[ 5.        0.0196    0.0151    0.99512]
[ 6.        0.01356   0.01518   0.9956 ]
```

LR used for the initial layers which comprise of the more basic features closest to the pixels

LR for the additional layers we added on top

This is the number of cycles NOT epochs so in this case learn is doing 3 cycles of 1 epoch (cycle_len = 1)

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 [23]:    1  learn.sched.plot_lr()
```
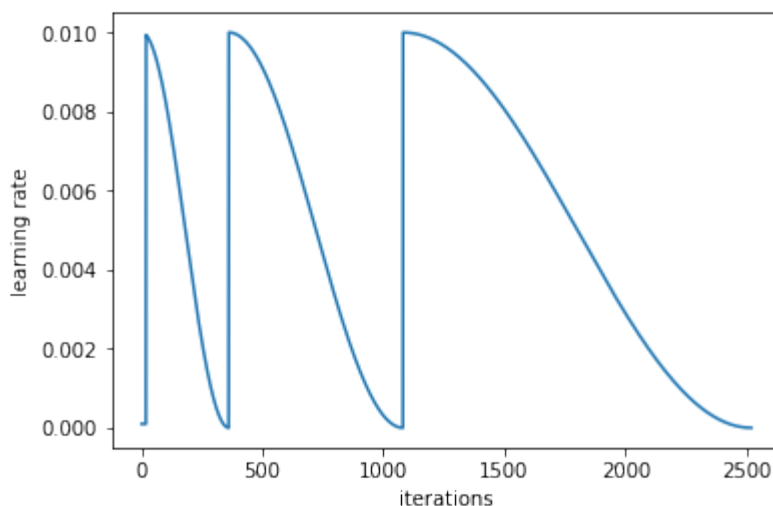
cycle_mult = 2 is doubling the length of each cycle

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 [35]:  1  learn.save('224_all')
```

```
In [9]:  1  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 [36]:  1  log_preds,y = learn.TTA()
          2  accuracy(log_preds,y)
```

```
Out[36]:  0.9965000000000005
```

TTA - training time augmentation ie each picture is then augmented slightly so that the computer gets multiple augmentations of the same picture that is why the accuracy rate goes up to 99.6%

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 [37]:  1  preds = np.argmax(log_preds, axis=1)
          2  probs = np.exp(log_preds[:,1])
```

A common way to analyze the result of a classification model is to use a confusion matrix. Scikit-learn has a convenient function we can use for this purpose:

```
In [38]:  1  from sklearn.metrics import confusion_matrix
          2  cm = confusion_matrix(y, preds)
```

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 [39]:  1  plot_confusion_matrix(cm, data.classes)
```

```
[[998    2]
 [  5  995]]
```

Confusion matrix
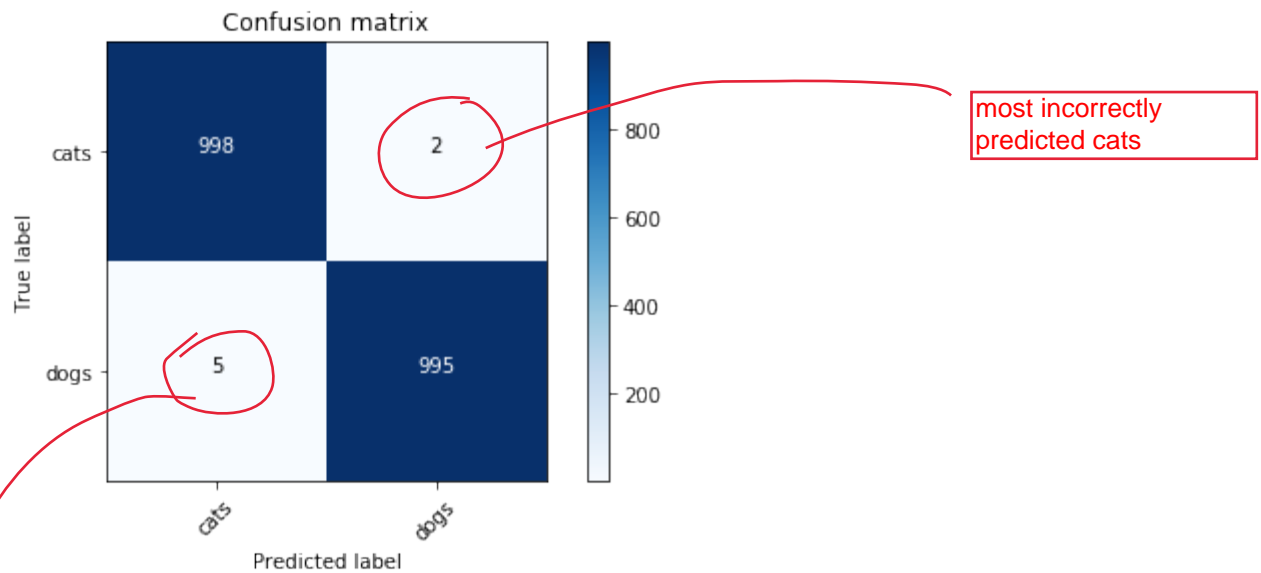
Looking at pictures again

```
In [60]:   1  plot_val_with_title(most_by_correct(0, False), "Most incorrect cats")
```

Most incorrect cats


```
In [61]:   1  plot_val_with_title(most_by_correct(1, False), "Most incorrect dogs")
```

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

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 [29]:    1  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 [30]:    1  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.
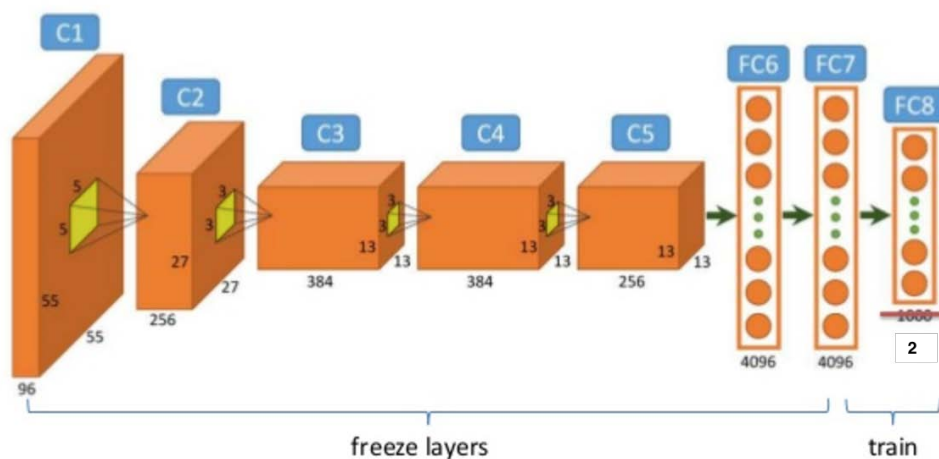
original image

```
In [31]:    1  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 [32]:
```
1  learn.fit(1e-2, 1)
```

A Jupyter Widget

```
[ 0.       0.04153  0.02681  0.98877]
```

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 [34]:
```
1  def binary_loss(y, p):
2      return np.mean(-(y * np.log(p) + (1-y)*np.log(1-p)))
```

In [35]:
```
1  acts = np.array([1, 0, 0, 1])
2  preds = np.array([0.9, 0.1, 0.2, 0.8])
3  binary_loss(acts, preds)
```

Out[35]: 0.164252033486018

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.

In [ ]:   1