# Image_Classification_CNN

September 30, 2018

## 0.1 Image classification with Convolutional Neural Networks

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

```
In [51]: # This file contains all the main external libs we'll use
         from fastai.imports import *
         warnings.filterwarnings("ignore", message="numpy.dtype size changed")
```

## 0.2 Dogs vs Cats

```
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 *
```

- `PATH` is the path to the data
- `sz` is the size that the images will be resized to be this is a very important value

```
In [4]: PATH = "data/dogscats/"
        sz=224
```

NVidia GPU is currently necessary, I'm using Paperspace currently for this GPU curruntly a M4000 51 c/hr. The framework behind NVidia GPU's is CUDA basically a standard for machine learning backends. The following should return true.

```
In [5]: torch.cuda.is_available()
```

```
Out[5]: True
```

In addition, NVidia provides special accelerated functions for deep learning in a package called CuDNN. Although not strictly necessary, it will improve training performance significantly, and is included by default in all supported fastai configurations. Therefore, if the following does not return True, you may want to look into why.

```
In [6]: torch.backends.cudnn.enabled
```

```
Out[6]: True
```

### 0.2.1 Data Set

You can grab the data from: http://files.fast.ai/data/dogscats.zip
Make sure its available in the same directory as this notebook.

## 0.3 First look at cat pictures

The 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]: # This lists what is under the PATH we set earlier
        os.listdir(PATH)
```

```
Out[7]: ['sample', 'valid', 'models', 'train', 'tmp', 'test1']
```

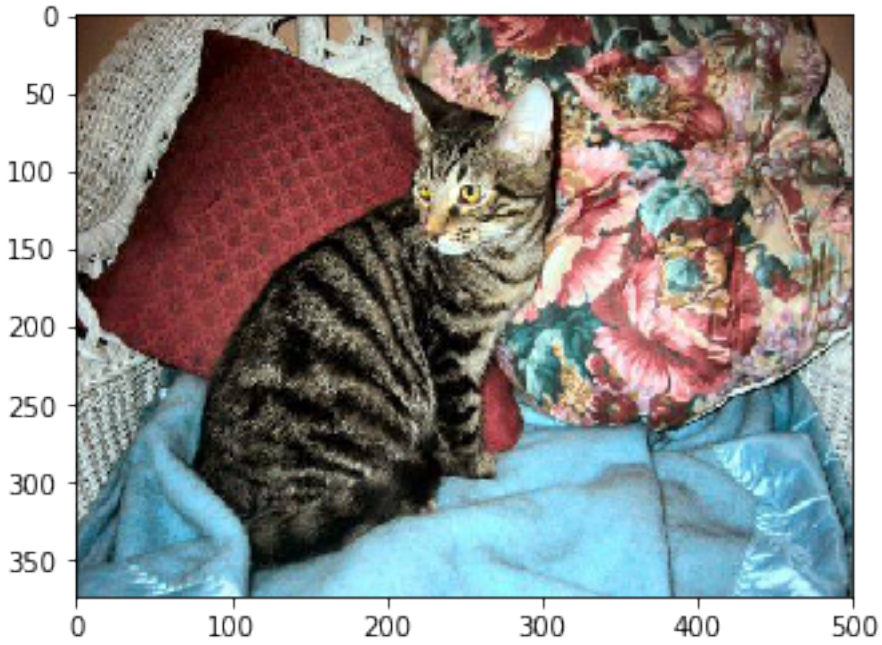
```
In [8]: # These directories are in the valid dir
        # replace valid with train you'll notice the same thing
        os.listdir(f'{PATH}valid')
```

```
Out[8]: ['cats', 'dogs']
```

```
In [9]: # Grab 5 cat files
        # Below is a Python 3.6 format string
        files = os.listdir(f'{PATH}valid/cats')[:5]
        files
```

```
Out[9]: ['cat.11341.jpg',
         'cat.3869.jpg',
         'cat.5089.jpg',
         'cat.7380.jpg',
         'cat.11409.jpg']
```

```
In [10]: # This is cat #5
         img = plt.imread(f'{PATH}valid/cats/{files[4]}')
         plt.imshow(img);
```




Here is what the raw data looks like

```
In [11]: # rank 3 tensor pixel height width and rgb
         # height 374
         # width 500
         # rgb per pixel 3
         img.shape

Out[11]: (374, 500, 3)

In [12]: # rgb values first four rows and columns
         img[:4,:4]

Out[12]: array([[[66, 38, 24],
                 [66, 38, 24],
                 [66, 38, 24],
                 [67, 39, 25]],

                [[68, 40, 26],
                 [68, 40, 26],
                 [68, 40, 26],
                 [68, 40, 26]],

                [[70, 42, 28],
                 [70, 42, 28],
                 [69, 41, 27],
                 [69, 41, 27]],
```

```
[[69, 41, 27],
 [69, 41, 27],
 [68, 40, 26],
 [69, 41, 27]]], dtype=uint8)
```

## 0.4 Train the 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 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.

In [13]:
```
# Uncomment the below if you need to reset your precomputed activations
# shutil.rmtree(f'{PATH}tmp', ignore_errors=True)
```

### 0.4.1 Super high level Training using fast.ai libraries

In [14]:
```
# This may take awhile the first time it runs
# This is due to the pretrained model needing to be downloaded
# We're doing 3 epochs with a step size of 0.01
# An epoch is a full pass through all the images
# The table below tells you what the numbers mean
# Loss train | loss valid | Accuracy
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)
```

```
HBox(children=(IntProgress(value=0, description='Epoch', max=3), HTML(value='')))


epoch      trn_loss   val_loss   accuracy
    0      0.052354   0.024662   0.991
    1      0.047685   0.026102   0.9885
    2      0.044548   0.029434   0.9885
```

Out[14]:
```
[array([0.02943]), 0.9885]
```

80% accuracy used to be state of the art

## 0.5 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 (i.e. those with highest probability that are correct) 4. The most incorrect labels of each class (i.e. those with highest probability that are incorrect) 5. The most uncertain labels (i.e. those with probability closest to 0.5).

### 0.5.1 Objects data & learn

data will hold all of our data contains: * validation data * training data
learn will hold the model

```
In [15]: # This is the label for a val data
         # 0's for cats 1's for dogs
         # validation dependent variable: y
         data.val_y
```

```
Out[15]: array([0, 0, 0, ..., 1, 1, 1])
```

```
In [16]: # from here we know that 'cats' is label 0 and 'dogs' is label 1.
         data.classes
```

```
Out[16]: ['cats', 'dogs']
```

```
In [17]: # this gives prediction for validation set. Predictions are in log scale
         log_preds = learn.predict()
         log_preds.shape
```

```
Out[17]: (2000, 2)
```

```
In [18]: # First 10 predictions
         # most models return the log of the prediction rather than the probabilities
         # labels |dog|cat|
         # to get probabilities use e^x
         log_preds[:10]
```

```
Out[18]: array([[ -0.00006,  -9.68818],
                [ -0.00213,  -6.1551 ],
                [ -0.00082,  -7.10515],
                [ -0.00019,  -8.5506 ],
                [ -0.00006,  -9.65819],
                [ -0.00017,  -8.70505],
                [ -0.00004, -10.19966],
                [ -0.00004, -10.18515],
                [ -0.00173,  -6.36094],
                [ -0.00017,  -8.6512 ]], dtype=float32)
```

```
In [19]: preds = np.argmax(log_preds, axis=1)  # from log probabilities to 0 or 1
         probs = np.exp(log_preds[:,1])        # pr(dog)
```

```
In [20]: def rand_by_mask(mask):
             return np.random.choice(np.where(mask)[0], 4, replace=False)
         def rand_by_correct(is_correct):
             return rand_by_mask((preds == data.val_y)==is_correct)

In [21]: 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 [22]: 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 [23]: # 1. A few correct labels at random
         plot_val_with_title(rand_by_correct(True), "Correctly classified")
```

```
Correctly classified
```


```
In [24]: # 2. A few incorrect labels at random
         plot_val_with_title(rand_by_correct(False), "Incorrectly classified")
```

```
Incorrectly classified
```


```
In [25]: 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), mult)
```

```
In [26]: plot_val_with_title(most_by_correct(0, True), "Most correct cats")
```

Most correct cats


```
In [27]: plot_val_with_title(most_by_correct(1, True), "Most correct dogs")
```

Most correct dogs


In [28]: plot_val_with_title(most_by_correct(0, False), "Most incorrect cats")

Most incorrect cats


In [29]: plot_val_with_title(most_by_correct(1, False), "Most incorrect dogs")

Most incorrect dogs

```
In [30]: most_uncertain = np.argsort(np.abs(probs -0.5))[:4]
         plot_val_with_title(most_uncertain, "Most uncertain predictions")
```

```
Most uncertain predictions
```


## 0.6 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 affects 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 stops 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 [31]: # redefine learn object
         arch=resnet34
         data = ImageClassifierData.from_paths(PATH, tfms=tfms_from_model(arch, sz))
         learn = ConvLearner.pretrained(arch, data, precompute=True)
```

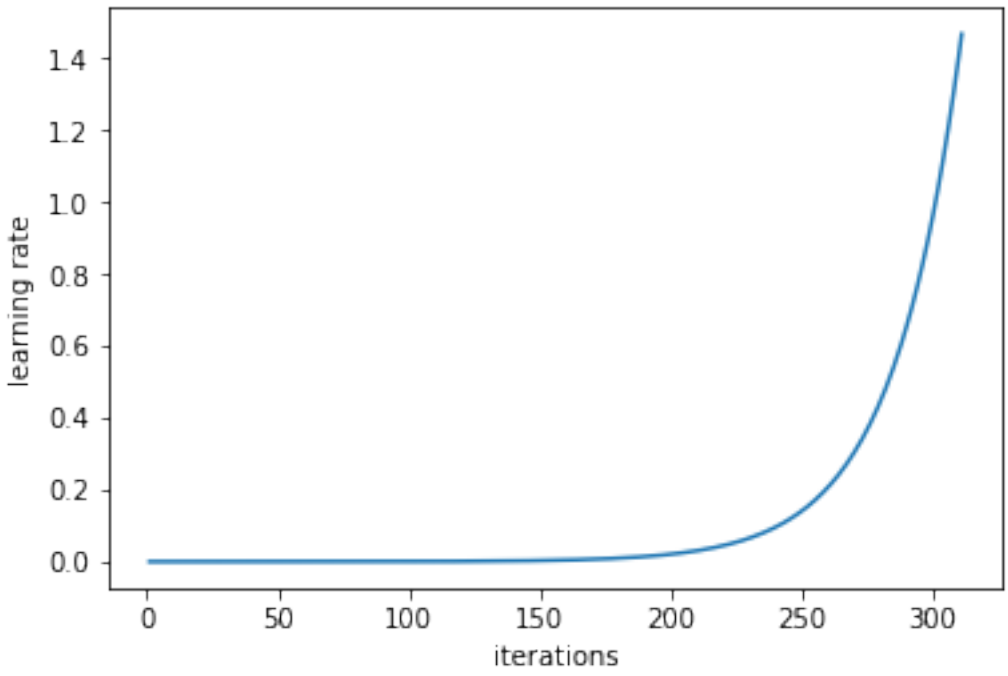
```
In [32]: # Learning rate finder
         # start at very small learning rates and gradually increase
         # you can see this behavior in the graph below
         lrf=learn.lr_find()
```

```
HBox(children=(IntProgress(value=0, description='Epoch', max=1), HTML(value='')))


 76%|  | 273/360 [00:04<00:01, 55.54it/s, loss=0.162]
```

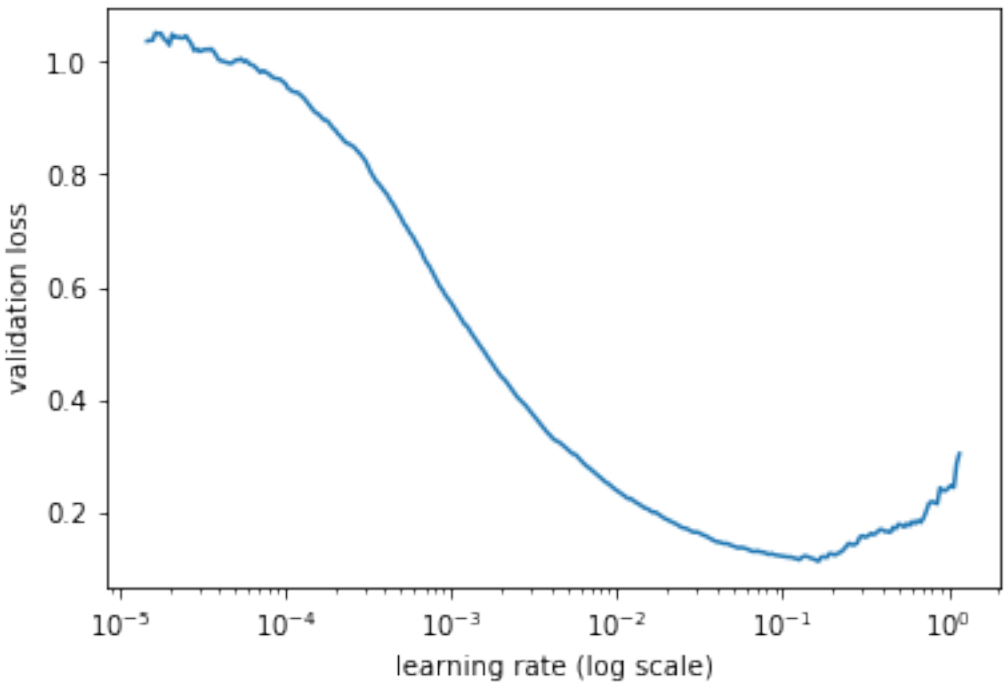
Our `learn` object contains an attribute `sched` that contains our learning rate scheduler, and has some convenient plotting functionality including this one:

```
In [33]: # exponential growth is used to increase learning rate
         learn.sched.plot_lr()
```




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

```
In [34]: # helps choose this particular hyper parameter
         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 train the model, so you may want to re-run this function from time to time.

## 0.7 Improving our model

### 0.7.1 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 generalizing 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 [35]: tfms = tfms_from_model(resnet34, sz, aug_tfms=transforms_side_on, max_zoom=1.1)
```

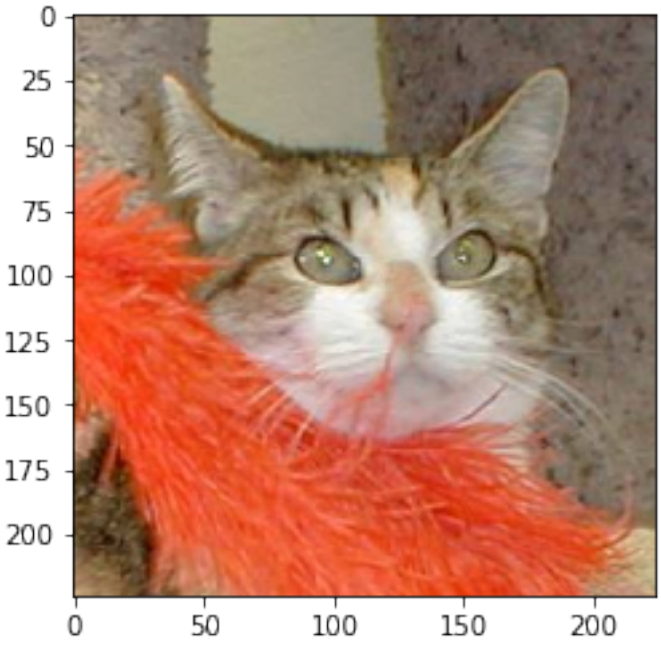
```
In [36]: def get_augs():
             data = ImageClassifierData.from_paths(PATH, bs=2, tfms=tfms, num_workers=1)
             x,_ = next(iter(data.aug_dl))
             return data.trn_ds.denorm(x)[1]
```

In [37]:
```
ims = np.stack([get_augs() for i in range(10)])
```

In [38]:
```
plots(ims, rows=2)
```

In [39]:
```
# every image is 224 x 224 now
# this may pose a problem for long or tall images
# key features may be cut out
# this is why use data augmentation
plt.imshow(ims[0]);
ims[0].shape
```

Out[39]:
```
(224, 224, 3)
```



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

```
In [40]: data = ImageClassifierData.from_paths(PATH, tfms=tfms)
         # pre compute is still equal to true
         learn = ConvLearner.pretrained(arch, data, precompute=True)

In [41]: learn.fit(1e-2, 1)
```

```
HBox(children=(IntProgress(value=0, description='Epoch', max=1), HTML(value='')))


epoch      trn_loss   val_loss   accuracy
    0      0.047777   0.026594   0.99
```

```
Out[41]: [array([0.02659]), 0.99]

In [42]: 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 [43]: # If you're doing much better on the training set than
         # the validation set, that means your model is not generalizing
         # We are not at that point here but we also are not improving
         learn.fit(1e-2, 3, cycle_len=1)
```

```
HBox(children=(IntProgress(value=0, description='Epoch', max=3), HTML(value='')))


epoch      trn_loss   val_loss   accuracy
    0      0.046649   0.027145   0.9915
    1      0.041141   0.026741   0.989
    2      0.045064   0.026627   0.989
```

```
Out[43]: [array([0.02663]), 0.989]
```

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.

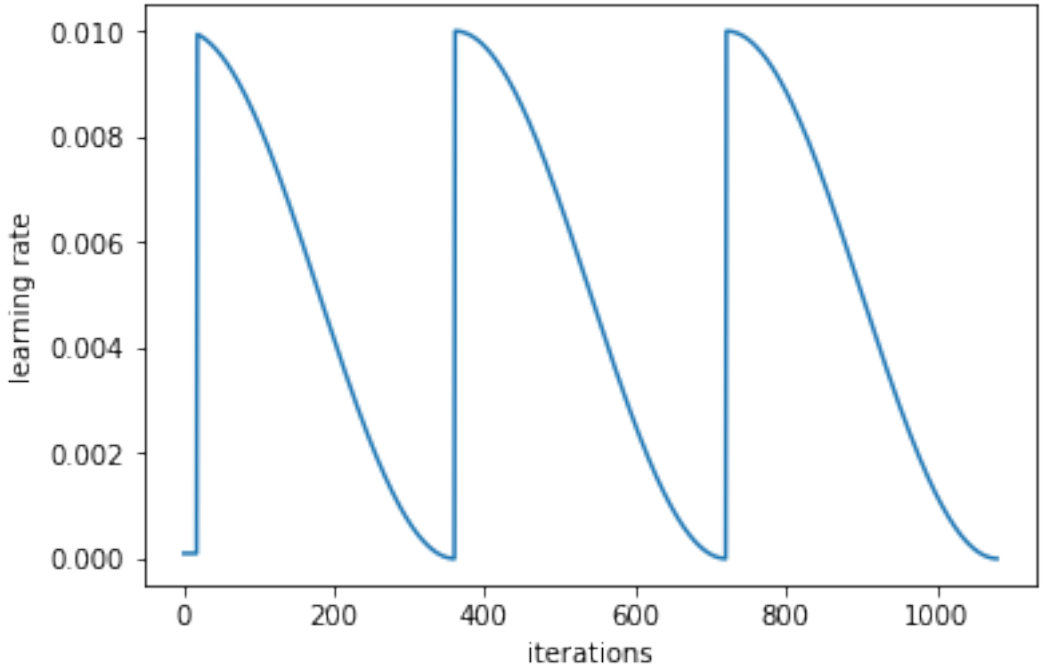
However, we may find ourselves in a part of the weight space that isn't very resilient - that is, small changes to the weights may result in big changes to the loss. We want to encourage our model to find parts of the weight space that are both accurate and stable. Therefore, from time to time we increase 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).

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 [44]: 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 [45]: learn.save('224_lastlayer')
```

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

### 0.7.2 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 [47]: 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 [48]: # Resnet, we are grouping layers into 3 groups
         # smaller learning rates for early layers for fine tuning
         # probably dont even need or want to train them
         # it will use a different learning rate for each layer
         lr=np.array([1e-4,1e-3,1e-2])
```

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

```
HBox(children=(IntProgress(value=0, description='Epoch', max=7), HTML(value='')))


epoch      trn_loss   val_loss   accuracy
    0      0.046665   0.023147   0.9915
    1      0.039462   0.020856   0.9925
    2      0.029003   0.020078   0.991
    3      0.027252   0.024574   0.991
    4      0.022143   0.018684   0.992
    5      0.021846   0.018735   0.9925
    6      0.017602   0.018774   0.991


Out[50]: [array([0.01877]), 0.991]
```