alexis_training

September 2, 2018

1 Initial training on reduced sample of images

The network is initially trained using just the first 200 games, split into 80/20 ratio as train/validation sets.

1.1 initial setup:

In [57]:
```
# auto reloading and inline display of matplotlib output:
%reload_ext autoreload
%autoreload 2
%matplotlib inline
```

In [58]:
```
# import fastai external dependencies
from fastai.imports import *
```

In [59]:
```
# import the various fastai functions (mainly wrappers around scikit-learn and pytorc
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 *
```

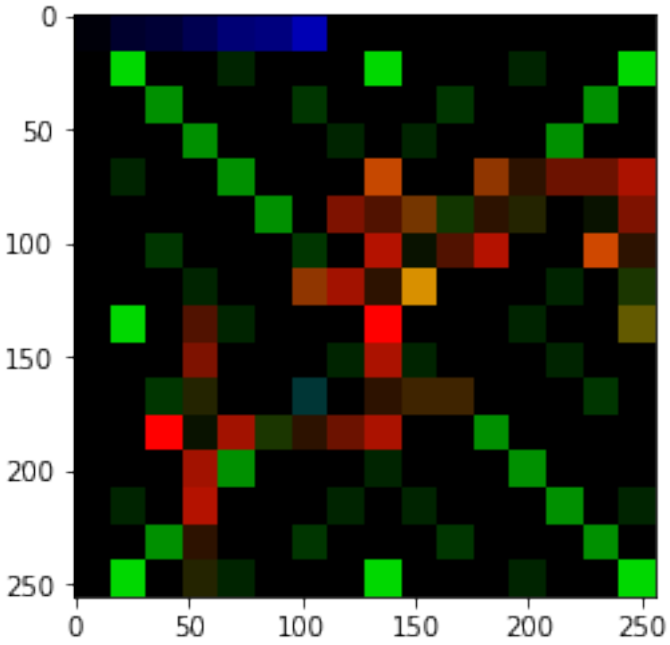
In [5]:
```
# check we're in right directory (should be /c/code/alexis)
!pwd
```

```
/c/code/alexis
```

In [56]:
```
# set root path for our training samples:
PATH = "data/subset/"

# size to scale images to:
sz=256
```

In [37]:
```
# check NVidia GPU is working (should return True):
torch.cuda.is_available()
```

```
Out[37]: True

In [38]: # check CuDNN library is available:
         torch.backends.cudnn.enabled

Out[38]: True

In [39]: # check we have a 'train' and 'valid' directory
         os.listdir(PATH)

Out[39]: ['train', 'valid']

In [40]: # each folder has subfolders for each category
         # (should be 'best' for 'best moves' generated by quackle
         # and 'not_best' for other moves generated by alexis:

         os.listdir(f'{PATH}valid')

Out[40]: ['best', 'not_best']

In [43]: files = os.listdir(f'{PATH}valid/best')
         files[:10] # just look at the first few

Out[43]: ['q_g0001_move02_option2.png',
          'q_g0001_move07_option1.png',
          'q_g0001_move09_option1.png',
          'q_g0001_move10_option2.png',
          'q_g0001_move12_option2.png',
          'q_g0001_move16_option1.png',
          'q_g0001_move17_option2.png',
          'q_g0002_move03_option1.png',
          'q_g0002_move03_option2.png',
          'q_g0002_move04_option1.png']

In [44]: len(files)

Out[44]: 1803

In [46]: # here's a sample image

         img = plt.imread(f'{PATH}valid/best/{files[100]}')
         plt.imshow(img);
```



Here is how the shape of the raw data:

```
In [48]: img.shape

Out[48]: (256, 256, 3)

In [54]: #here's the raw data, triplets of [R, G, B] values (quite sparse - most squares don't
         img

Out[54]: array([[[0.     , 0.     , 0.03529],
                 [0.     , 0.     , 0.03529],
                 [0.     , 0.     , 0.03529],
                 ...,
                 [0.     , 0.     , 0.     ],
                 [0.     , 0.     , 0.     ],
                 [0.     , 0.     , 0.     ]],

                [[0.     , 0.     , 0.03529],
                 [0.     , 0.     , 0.03529],
                 [0.     , 0.     , 0.03529],
                 ...,
                 [0.     , 0.     , 0.     ],
                 [0.     , 0.     , 0.     ],
                 [0.     , 0.     , 0.     ]],

                [[0.     , 0.     , 0.03529],
                 [0.     , 0.     , 0.03529],
```

```
        [0.     , 0.     , 0.03529],
        ...,
        [0.     , 0.     , 0.     ],
        [0.     , 0.     , 0.     ],
        [0.     , 0.     , 0.     ]],

       ...,

       [[0.     , 0.     , 0.     ],
        [0.     , 0.     , 0.     ],
        [0.     , 0.     , 0.     ],
        ...,
        [0.     , 0.84706, 0.     ],
        [0.     , 0.84706, 0.     ],
        [0.     , 0.84706, 0.     ]],

       [[0.     , 0.     , 0.     ],
        [0.     , 0.     , 0.     ],
        [0.     , 0.     , 0.     ],
        ...,
        [0.     , 0.84706, 0.     ],
        [0.     , 0.84706, 0.     ],
        [0.     , 0.84706, 0.     ]],

       [[0.     , 0.     , 0.     ],
        [0.     , 0.     , 0.     ],
        [0.     , 0.     , 0.     ],
        ...,
        [0.     , 0.84706, 0.     ],
        [0.     , 0.84706, 0.     ],
        [0.     , 0.84706, 0.     ]]], dtype=float32)
```

2 Now train a neural network:

2.0.1 download ResNet18 architecture:

In [6]:
```
# Uncomment to reset precomputed activations:
# shutil.rmtree(f'{PATH}tmp', ignore_errors=True)
```

In [55]:
```
arch=resnet18
data = ImageClassifierData.from_paths(PATH, tfms=tfms_from_model(arch, sz))

# learn is the main CNN object containing a description of the architecture and weigh
learn = ConvLearner.pretrained(arch, data, precompute=True)
```

```
Downloading: "https://download.pytorch.org/models/resnet18-5c106cde.pth" to C:\Users\richard/.t
100%|| 46827520/46827520 [00:11<00:00, 4028851.40it/s]
```

```
100%|| 451/451 [03:16<00:00,  2.29it/s]
100%|| 113/113 [00:48<00:00,  2.32it/s]
```

2.0.2 calculate optimal learning rate:

the learning rate is an important hyperparamer. It controls how quickly we adjust the weights in the network when updating them - too small and we will crawl to a solution and potentially get stuck in local minima, too high and we will bounce around the multi-dimensional surface describing our function without settling into a stable minimum point.

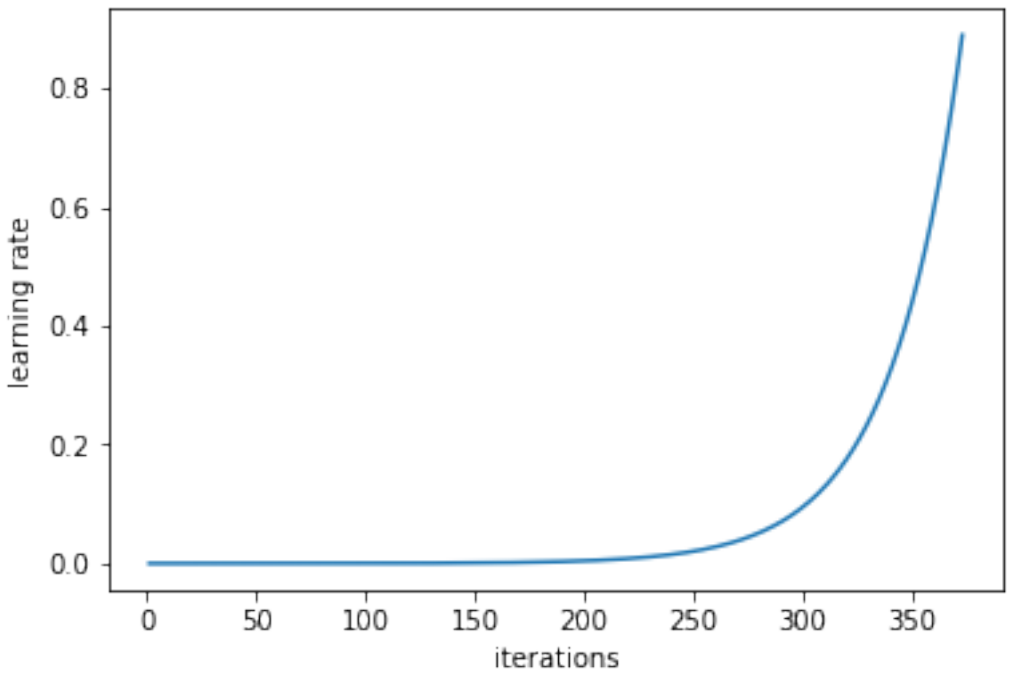
learn.lr_find() uses the technique from Smith (2015) to determine an optimal rate - we keep increasing the learning rate each iteration (i.e. each minibatch), starting from a very small valuem, until the loss is no longer decreasing.

```
In [60]: lrf=learn.lr_find()
```

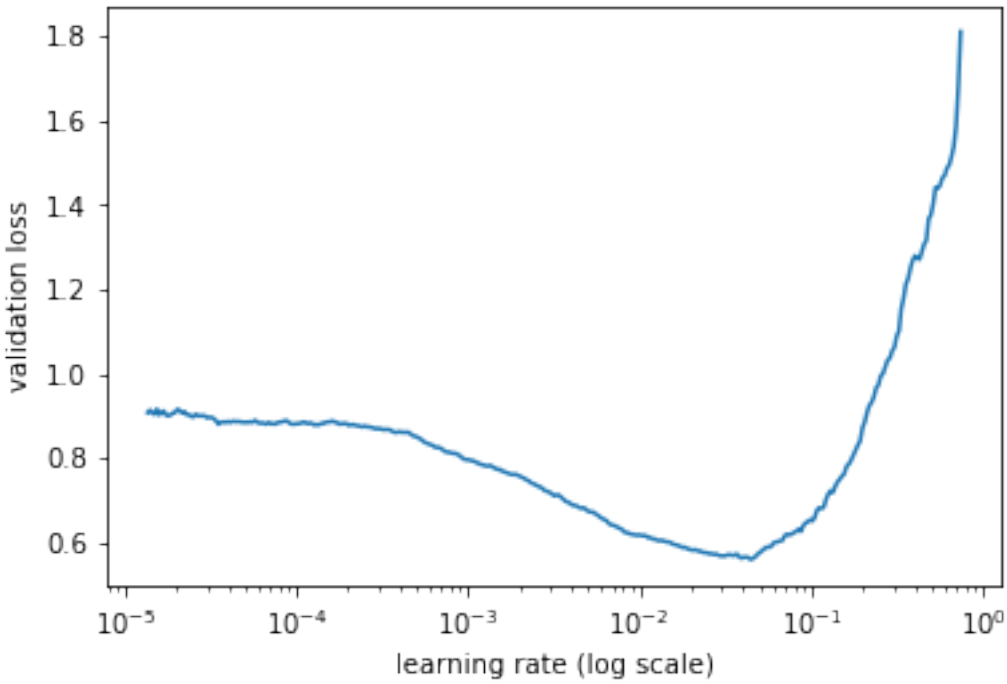
```
HBox(children=(IntProgress(value=0, description='Epoch', max=1), HTML(value='')))
```

```
 82%|            | 371/451 [00:03<00:00, 112.87it/s, loss=1.98]
```

```
In [61]: # here's what the above was doing to the learning rate, increasing it exponentially:
         learn.sched.plot_lr()
```



In [62]:
```
# and here's the effect this had on the loss:
learn.sched.plot()
```



We want to pick a learning rate where the loss is still clearly decreasing. The absolute minimum, the turning point in the above plot, may be a little too high. The loss is still clearly decreasing at 10-2, so we'll use that as the initial learning rate. This may change as the model is changed, so this should be re-run from time to time on a copy of the learn object.

2.0.3 unfreezing layers

We have just calculated the optimal learning rate for the last hidden layer, assuming we are going to fine tune the model - that is, just train the last layer to work with new output categories. This would be fine if we were taking a network trained on ImageNet classification into the 1000 imagenet categories, and retraining it to recognise different but similar categories. We are retraining the network to a markedly different task of acting as a Scrabble move evaluator. This involves some of the same features (an appreciation of relative position, for example), but we probably still want to train the earlier layers. We'll set varying learning rates so that the earlier layers can still re-train, but the weights will move more slowly, since we expect the earlier layers to have learned more features in common with our task.

In [63]:
```
# We also want to start from the initial set of weights,
# not the weights we've been messing with while finding
# a learning rate, so we'll reset the weights:

shutil.rmtree(f'{PATH}tmp', ignore_errors=True)
```

```
In [65]: # We'd previously precomputed the weights in the frozen layers:
         learn.precompute=False

         # now unfreeze all the layers:
         learn.unfreeze()

In [66]: lr = 0.01 # the learning rate we found above

         # here we'll set the early, middle and late layers to different fractions of this lea
         lrs = np.array([lr/4,lr/2,lr])

In [67]: # start fitting on the training samples, validating against the validation set (try 3
         learn.fit(lrs, 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      0.35974    0.341463   0.873821
    1      0.337308   0.325448   0.875208
    2      0.324905   0.31948    0.874376
    3      0.327419   0.322597   0.876317
    4      0.319264   0.32057    0.870632
    5      0.280278   0.327713   0.868691
    6      0.259751   0.3415     0.863422


Out[67]: [array([0.3415]), 0.8634220743205768]
```

The cycle_len parameter used means that instead of doing 3 epochs, we are instead doing 3 cycles of epochs. The cycle_mult parameter means each cycle is twice as long as the last. So we are performing a cycle of 1, 2, and 4 epochs for a total of seven epochs. We are performing SGDR (Stochastic Gradient Descent with Restarts) such that after each cycle we the learning rate is reset.

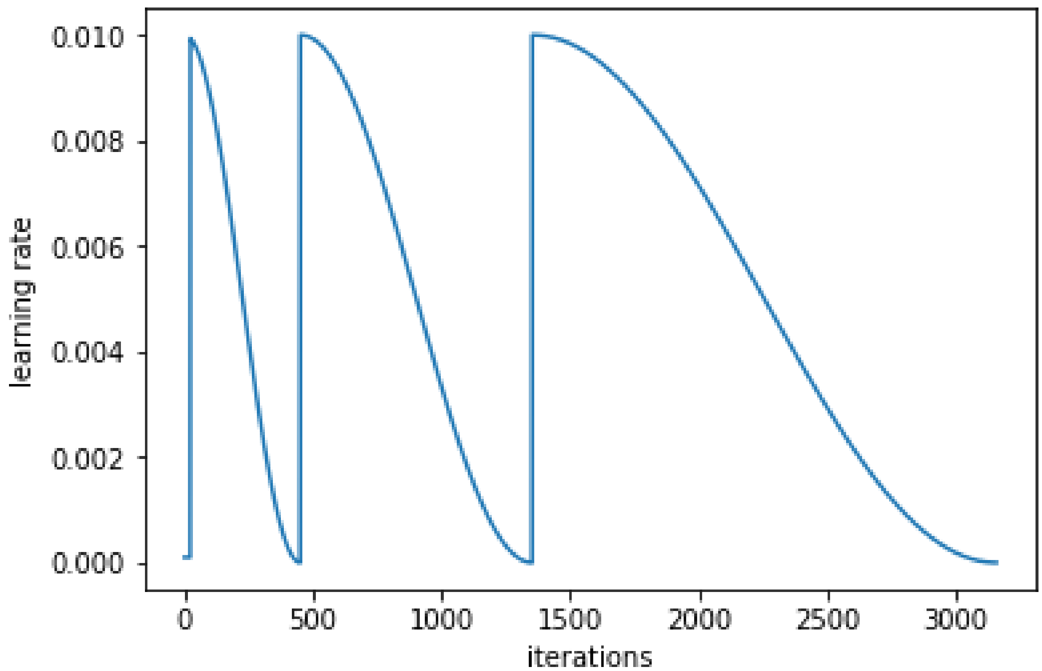
We start off at the initial learning rate (0.01 in this case), and perform learning rate annealing so that the learning rate is reduced each batch. The learning will hopefully slow down as we approach a minimum.

After a cycle, our learning rate will jump up to 0.01 again.

This technique helps to explore multiple minima and hopefully end up in a more stable part of the weight space rather than being stuck in an unstable local minimum.

Our learning rate across epochs looks like this:

```
In [69]: learn.sched.plot_lr()
```




```
In [71]: # let's save our model so far:
         learn.save('initial_trained_model')
```

Now, over the last couple of epochs our accuracy seems to be going down - the loss on the training set is decreasing but the loss on the validation set (the samples we didn't train on) has started to increase, indicating we are starting to overfit to the training data. How are we actually doing?

```
In [75]: # This is the list of labels for our validation set:
         print([y for y in data.val_y])

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, (
```

```
In [78]: # these classes are as follows:
         data.classes

Out[78]: ['best', 'not_best']
```

```
In [80]: data.val_y.shape

Out[80]: (7212,)
```

```
In [85]: # so we know that label 0 represents 'best' moves and label 1 represents 'not_best' mo
         # let's get predictions for our validation set, i.e. perform inference by
         # just making a forward pass through the network:
```

```
        # (predictions are a log scale)
        log_preds = learn.predict()

        # we should have 2 values per sample, a value giving how well
        # the model thinks that sample fits each of the 2 categories:
        log_preds.shape
```

```
Out[85]: (7212, 2)
```

```
In [86]: # let's look at a few:
         log_preds[:-10]
```

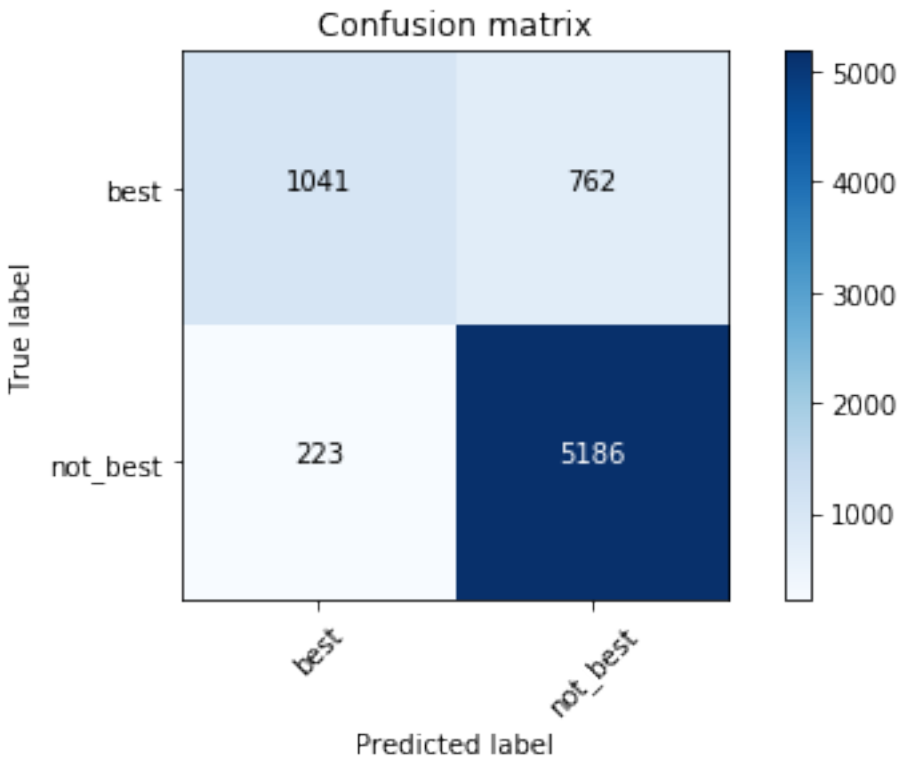
```
Out[86]: array([[-0.00008, -9.4529 ],
                [-0.47529, -0.97208],
                [-0.76165, -0.62903],
                ...,
                [-5.32152, -0.0049 ],
                [-4.37181, -0.01271],
                [-3.07112, -0.04748]], dtype=float32)
```

```
In [112]: # convert these from a log scale to 0 or 1:
          preds = np.argmax(log_preds, axis=1)
          preds
```

```
Out[112]: array([0, 0, 1, ..., 1, 1, 1], dtype=int64)
```

```
In [114]: from sklearn.metrics import confusion_matrix
          cm = confusion_matrix(data.val_y, preds)
          plot_confusion_matrix(cm, data.classes)
```

```
[[1041  762]
 [ 223 5186]]
```



So we can see that for images that were not the 'best' move, our model is labelling them as not the best move around 95.8% of the time. For moves that are supposedly the 'best' move, those generated from Quackle, our model is labelling them as the best move only around 57.7% of the time.

This is still encouraging, since our evaluation method was poor in the first place - we are simply trusting that Quackle will have picked the best move and so the function we are approximating is not really which is the best move but rather out of the moves picked by Quackle and the moves picked by ALexIS, which are the most 'Quackle-like'.

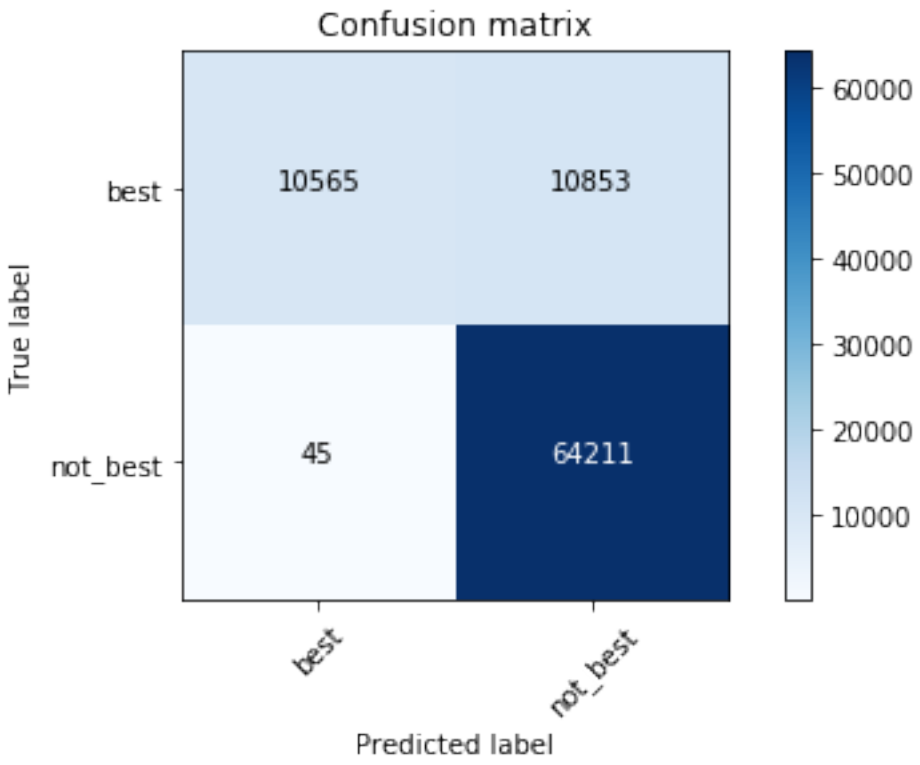
In some situations, such as where it makes no odds to strategy because we are not exposing key squares to our opponent, the optimal choice for Quackle may well be similar to ALexIS' naive move evaluation function which simply picks the highest score, and in such cases there would be little to tell our two Scrabble simulations apart.

3 Training on full data-set

We've performed some quick-and-dirty training on a small subset of the games, the first 200, in order to get a rough neural network. We can likely do better with more data, and so we can now take the rest of the data, discarding the earlier subset we trained on so as not to overfit to it any more. With the remaining 2366 games, we can now repeat the process. Since each has about 20-25 moves and each of those moves has 16 possible plays recorded, we are now training on about 340k images and validating on 85k.

```
In [ ]: # let's copy our saved model to the main data directory:
```

```
In [124]: !cp -r data/subset/models data/gcgs

In [115]: PATH = "data/gcgs/"
          data = ImageClassifierData.from_paths(PATH, tfms=tfms_from_model(arch, sz))

In [127]: # in case any preprocessing is done after instantiating the Learner,
          # let's not just swap in the new data-set, let's just make a new object
          # with the new data and load the old weights into it:

          learn = ConvLearner.pretrained(arch, data, precompute=False)

In [128]: learn.load('initial_trained_model')

In [129]: # retrain with the new data (just 1 epoch):
          learn.fit(lrs, 1)

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


epoch      trn_loss   val_loss   accuracy
    0      0.310383   0.311421   0.872797


Out[129]: [array([0.31142]), 0.8727968812008311]

In [130]: log_preds = learn.predict()
          preds = np.argmax(log_preds, axis=1)
          cm = confusion_matrix(data.val_y, preds)
          plot_confusion_matrix(cm, data.classes)

[[10565 10853]
 [   45 64211]]
```

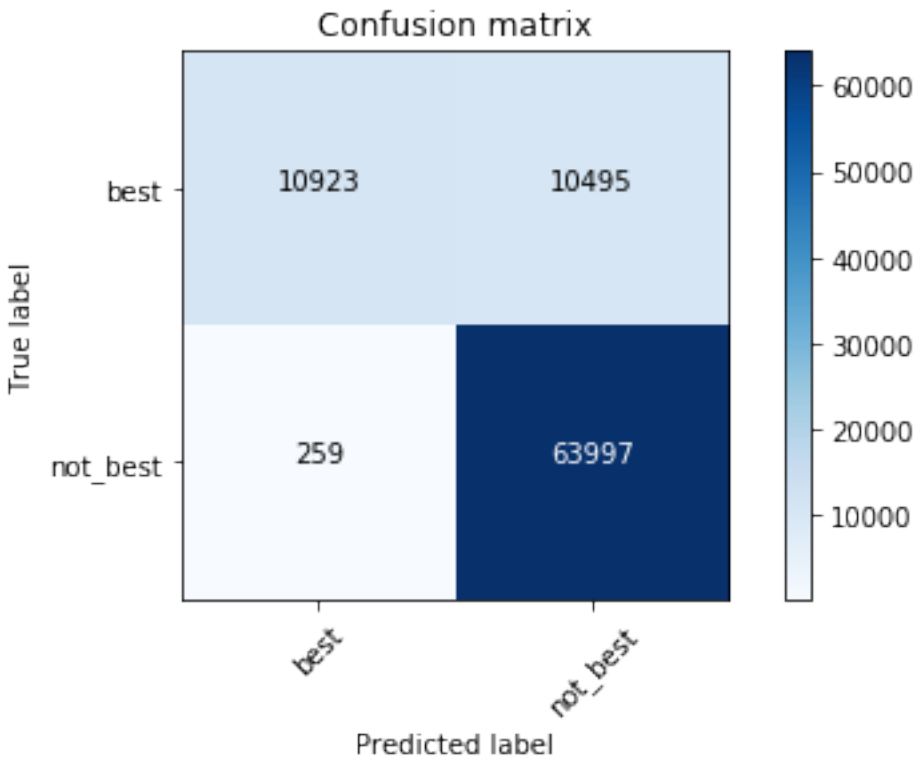



```
In [131]: learn.save('epoch1')

In [132]: # let's do another couple of epochs:
          learn.fit(lrs, 2)
```

```
HBox(children=(IntProgress(value=0, description='Epoch', max=2), HTML(value='')))


epoch      trn_loss   val_loss   accuracy
    0      0.308585   0.307824   0.874209
    1      0.309607   0.306913   0.874478


Out[132]: [array([0.30691]), 0.8744776711721176]
```

```
In [133]: learn.save('epoch3')

In [134]: log_preds = learn.predict()
          preds = np.argmax(log_preds, axis=1)
          cm = confusion_matrix(data.val_y, preds)
          plot_confusion_matrix(cm, data.classes)
```

```
[[10923 10495]
 [  259 63997]]
```



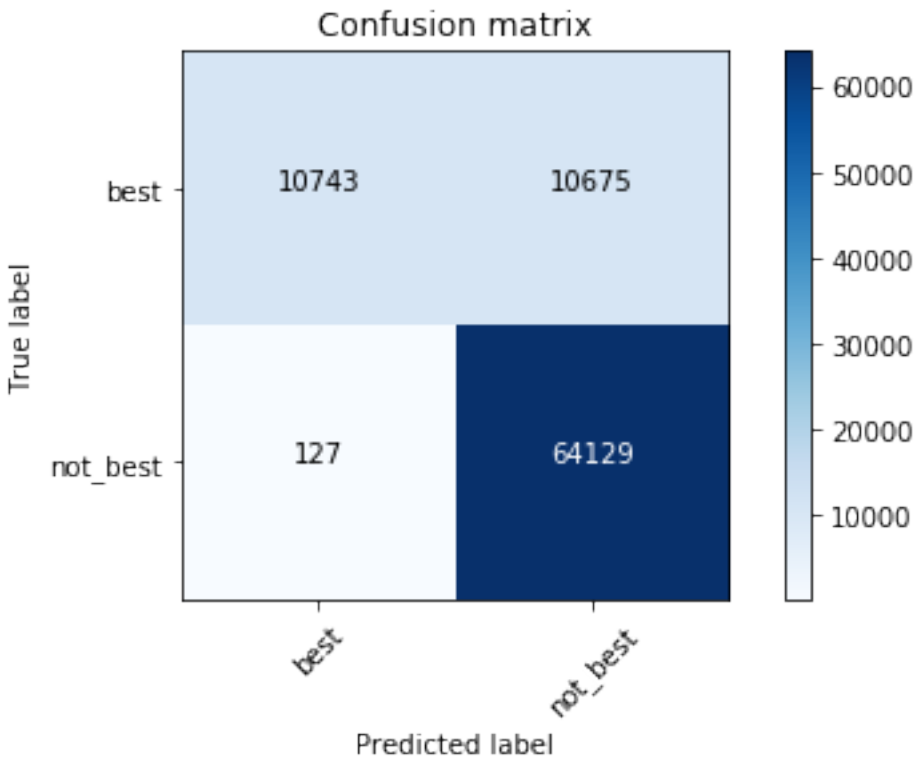

```
In [135]: # and another couple of epochs:
          learn.fit(lrs, 2)
```

```
HBox(children=(IntProgress(value=0, description='Epoch', max=2), HTML(value='')))


epoch      trn_loss   val_loss   accuracy
    0      0.317764   0.307613   0.874244
    1      0.314499   0.306958   0.873917


Out[135]: [array([0.30696]), 0.8739174078483554]
```

```
In [136]: learn.save('epoch5')
```

```
In [137]: log_preds = learn.predict()
          preds = np.argmax(log_preds, axis=1)
          cm = confusion_matrix(data.val_y, preds)
          plot_confusion_matrix(cm, data.classes)
```

```
[[10743 10675]
 [  127 64129]]
```




```
In [138]: # and with restarts:
          learn.fit(lrs, 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      0.315238   0.305415   0.874583
    1      0.315883   0.30582    0.875038
    2      0.309856   0.305224   0.875026
    3      0.308817   0.30503    0.874804
    4      0.297931   0.304889   0.874945
    5      0.318427   0.305293   0.87484
    6      0.313225   0.304956   0.874618
```

```
Out[138]: [array([0.30496]), 0.8746177370030581]
```

```
In [139]: learn.save('epoch10')
```

```
In [140]: log_preds = learn.predict()
          preds = np.argmax(log_preds, axis=1)
          cm = confusion_matrix(data.val_y, preds)
          plot_confusion_matrix(cm, data.classes)
```

```
[[10872 10546]
 [  196 64060]]
```