Submission
✔ Ran successfully
Submitted by NinaV 2 days ago

Public Score
0.752

Overview

The goal of this competition is identifying individual whales in images. Despite several whales are well represented in images, most of whales are unique or shown only in a few pictures. In particular, the train dataset includes 25k images and 5k unique whale ids. In addition, ~10k of images show unique whales ('new_whale' label). Checking public kernels suggests that a classical approach for classification problems based on softmax prediction for all classes is working quite well for this particular problem. However, strong class imbalance, handling labels represented by just several images, and 'new_whale' label deteriorates this approach. In addition, form the using this model for production, the above approach doesn't sound right since expansion of the model to identify new whales not represented in the train dataset would require retraining the model with increased softmax size. Meanwhile, the task of this competition could be reconsidered as checking similarities that suggests one-shot based learning algorithm to be applicable. This approach is less susceptible to data imbalance in this competition, can naturally handle 'new_whale' class, and is scalable in terms of a model for production (new classes can be added without retraining the model).

There are several public kernels targeted at using similarity based approach. First of all, it is an amazing kernel posted by Martin Piotte (https://www.kaggle.com/martinpiotte/whale-recognition-model-with-score-0-78563), which discusses Siamese Neural Network architecture in details. A fork of this kernel (https://www.kaggle.com/seesee/siamese-pretrained-0-822/notebook) reports 0.822 public LB score after training for 400 epochs. There is also a quite interesting public kernel (https://www.kaggle.com/ashishpatel26/triplet-loss-network-for-humpback-whale-prediction) discussing Triplet Neural Network architecture, which is supposed to overperform Siamese architecture (check links in this discussion (https://www.kaggle.com/c/humpback-whale-identification/discussion/76012)). Since both positive and negative examples are provided, the gradients are appeared to be more stable, and the network is not only trying to get away from negative or get close to positive example but arranges the prediction to fulfil both.

In this kernel I provide an example of a network inspired by Triplet architecture that is capable to reach **~0.75 public LB score after training within the kernel time limit** in my preliminary test. Training for more epochs is supposed to improve the prediction even further. The main trick of this kernel is **using batch all loss**. If the forward pass is completed for all images in a batch, why shouldn't I compare all of them when calculate the loss function? why should I limit myself by just several triplets? I have designed a loss function in such a way that allows performing all vs. all comparison within each batch, in other words for a batch of size 32 instead of comparing 32 triplets or 64 pairs the network performs processing of 9216 pairs of images at the same time. If training is done on multiple GPUs, the number of compared pares could be boosted even further since it it proportional to bs^2. Such a huge number of processed pairs further stabilizes gradients in comparison with triplet loss and allows more effective mapping of the input into the embedding space since not only pairs or triplets but entire picture is seen at the same time. This approach also allows effective search for hard negative examples at the later stage of training since each image is compared with all images in a batch. I tried to boost the search of the most difficult negative examples even further by selection of most similar negative examples to an anchor image when build triplets.

Moreover, I added **metric learning** (https://en.wikipedia.org/wiki/Similarity_learning) that boosts the performance of the model really a lot. In my preliminary test **for V2 setup I got 0.606->0.655 improvement** after I started calculating distance as d^2 = (v1-v2).T x A x (v1-v2) instead of Euclidian (v1-v2).T x (v1-v2), where A is a trainable matrix parameter. It can be considered as a trainable deformation of the space. However, the above form of the metric is quite slow at the inference time when distances for all image pairs are calculated. Also, it is quite difficult to impose a constrain on A during training to make it positive semi defined. Therefore, I use an alternative approximation formulation for distance calculation that is much faster at the inference time, symmetric and always positive, and have similar (or slightly better) performance with accounting for nonlinear coordinate transformations.

In this version of the kernel I switched to using cropped rescaled square images since they work better. The idea behind rectangular images generated without distortion of images, which I used in the previous versions of the kernel, is the following. Since bounding boxes have different aspect ratio, each image has different degree of distortion when rescaled to square one, which could negatively affect training. However, it looks that the setup when the tail is occupying approximately the same area in the image, no matter what is its orientation and distortion, works better. Looking at the produced images I really do not understand why. In my preliminary test I could get a **boost from ~0.70 to ~0.75 public LB** after this modification. In the current setup, the images are cropped according to bounding boxes (thanks to this fork

(https://www.kaggle.com/suicaokhoailang/generating-whale-bounding-boxes) and to Martin Piotte for posting the original kernel) and rescaled to 224x224 square images.

This kernel is written with using fast.ai 0.7 since a newer version of fast.ai doesn't work well in kaggle: using more than one core for data loading leads to bus error (https://www.kaggle.com/product-feedback/72606) "DataLoader worker (pid 137) is killed by signal: Bus error". Therefore, when I tried to write similar kernel with fast.ai 1.0, it appeared to be much slower, more than 1 hour per epoch vs. 20-30 min with this kernel if ResNet34 and images of size 576x192 are used. People interested in fast.ai 1.0 could check an example of Siamese network here (https://www.kaggle.com/raghavab1992/siamese-with-fast-ai). Also since fast.ai 0.7 is not really designed to build Siamese and Triplet networks, some parts are a little bit far away from a standard usage of the library.

Highlights: Batch all loss, metric learning, mining hard negative examples

In [1]:
```
!pip install fastai==0.7.0 --no-deps
!pip install torch==0.4.1 torchvision==0.2.1
!pip install imgaug
```

Output

In [2]:
```
from fastai.conv_learner import *
from fastai.dataset import *

import pandas as pd
import numpy as np
import os
from sklearn.model_selection import train_test_split
from tqdm import tqdm_notebook
import matplotlib.pyplot as plt
import random
import math
import imgaug as ia
from imgaug import augmenters as iaa
```

In [3]:
```
PATH = './'
# TRAIN = '../input/humpback-whale-identification/train/'
# TEST = '../input/humpback-whale-identification/test/'
LABELS = '../input/humpback-whale-identification/train.csv'
BOXES = '../input/generating-whale-bounding-boxes/bounding_bo
xes.csv'
MODLE_INIT = '../input/pytorch-pretrained-models/'

TRAIN_CROPPED = "whales-cropped/cropped_train/cropped_train/"
TRAIN_CROPPED_IN = '../input/' + TRAIN_CROPPED

TEST_CROPPED = "whales-cropped/cropped_test/cropped_test/"
TEST_CROPPED_IN = '../input/' + TEST_CROPPED

n_embedding = 256
bs = 32
ratio = 3
sz = 224 #increase the image size at the later stage of traini
ng
nw = 2
```

Data

The class Loader creates crops with sizes 224x224 based on the cropped images. In addition, data augmentation based on imgaug library (https://github.com/aleju/imgaug) is applied. This library is quite interesting in the context of the competition since it supports hue and saturation augmentations as well as conversion to gray scale. Following the idea of progressive resizing on the later stage of training one can switch to higher resolution images, but at the beginning low resolution is used to speed up the convergence and improve generalization ability of the model.

In [4]:
```
def open_image(fn):
    flags = cv2.IMREAD_UNCHANGED+cv2.IMREAD_ANYDEPTH+cv2.IMRE
AD_ANYCOLOR
    if not os.path.exists(fn):
        raise OSError('No such file or directory: {}'.format(
fn))
    elif os.path.isdir(fn):
        raise OSError('Is a directory: {}'.format(fn))
    else:
        try:
            im = cv2.imread(str(fn), flags)
            if im is None: raise OSError(f'File not recognize
d by opencv: {fn}')
            return cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
        except Exception as e:
            raise OSError('Error handling image at: {}'.forma
t(fn)) from e

class Loader():
    def __init__(self,path,tfms_g=None, tfms_px=None):
        #tfms_g - geometric augmentation (distortion, small ro
tation, zoom)
        #tfms_px - pixel augmentation and flip
        self.path = path
        self.tfms_g = iaa.Sequential(tfms_g,random_order=Fals
e) \
                        if tfms_g is not None else None
        self.tfms_px = iaa.Sequential(tfms_px,random_order=Fa
lse) \
                        if tfms_px is not None else None
    def __call__(self, fname):
        fname = os.path.basename(fname)
        img = open_image(os.path.join(self.path,fname))

        if self.tfms_g != None:
            img = self.tfms_g.augment_image(img)

        img = cv2.resize(img, (sz,sz))
        if self.tfms_px != None:
            img = self.tfms_px.augment_image(img)
        return img.astype(np.float)/255
```

The pdFilesDataset class below generates triplets of images: original image, different image with the same label, an image with different label (**including new_label images**). Image_selection class performs of selection of the most difficult negative examples used in the last part of the kernel. When this class is created, 64 most similar images with different label are selected for each image. So instead of random sampling of negative examples, sampling of these 64 most difficult images for each anchor image can be used during training.

In [5]:
```
def get_idxs0(names, df, n=64):
    idxs = []
    for name in names:
        label = df[df.Image == name].Id.values[0]
        idxs.append(df[df.Id != label].Image.sample(n).values
)
    return idxs

class Image_selection:
    def __init__(self,fnames,data,emb_df=None,model=None):
        if emb_df is None or model is None:
            df = data.copy()#.set_index('Image')
```

```
            counts = Counter(df.Id.values)
            df['c'] = df['Id'].apply(lambda x: counts[x])
            df['label'] = df.Id
            df.loc[df.c == 1,'label'] = 'new_whale'
            df = df.sort_values(by=['c'])
            df.label = pd.factorize(df.label)[0]
            l1 = 1 + df.label.max()
            l2 = len(df[df.label==0])
            df.loc[df.label==0,'label'] = range(l1, l1+l2) #a
ssign unique ids
            df = df.set_index('label')
            l = len(fnames)
            idxs = Parallel(n_jobs=nw)(delayed(get_idxs0)\
                (fnames[int(i*l/nw):int((i+1)*l/nw)], df) for
 i in range(nw))
            idxs = [y for x in idxs for y in x]
        else:
            data = data.copy().set_index('Image')
            trn_emb = emb_df.copy()
            trn_emb.set_index('files',inplace=True)
            trn_emb['emb'] = [[float(i) for i in s.split()] f
or s in trn_emb['emb']]
            trn_emb = data.join(trn_emb)
            trn_emb = trn_emb.reset_index()
            trn_emb['idx'] = np.arange(len(trn_emb))
            trn_emb = trn_emb.set_index('Id')
            emb = np.array(trn_emb.emb.tolist())
            l = len(fnames)
            idxs = []
            model.eval()
            with torch.no_grad():
                #selection of the most difficult negative exam
ples
                m = model.module if isinstance(model,FP16) el
se model
                emb = torch.from_numpy(emb).half().cuda()
                for name in tqdm_notebook(fnames):
                    label = trn_emb.loc[trn_emb.Image == name
].index[0]
                    v0 = np.array(trn_emb.loc[trn_emb.Image =
= name,'emb'].tolist()[0])
                    v0 = torch.from_numpy(v0).half().cuda()
                    d = m.get_d(v0,emb)
                    ids = trn_emb.loc[trn_emb.index!=label].i
dx.tolist()
                    sorted, indices = torch.sort(d[ids])
                    idxs.append([ids[i] for i in indices[:64
]])
            trn_emb = trn_emb.set_index('idx')
            idxs = [trn_emb.loc[idx,'Image'] for idx in idxs]
        self.df = pd.DataFrame({'Image':fnames,'idxs':idxs}).
set_index('Image')

    def get(self,name):
        return np.random.choice(self.df.loc[name].values[0],1
)[0]
```

In [6]:

```
class pdFilesDataset(FilesDataset):
    def __init__(self, data, path, transform):
        df = data.copy()
        counts = Counter(df.Id.values)
        df['c'] = df['Id'].apply(lambda x: counts[x])
        #in the production runs df.c>1 should be used
        fnames = df[(df.c>2) & (df.Id != 'new_whale')].Image.
tolist()
```

```
        df['label'] = df.Id
        df.loc[df.c == 1,'label'] = 'new_whale'
        df = df.sort_values(by=['c'])
        df.label = pd.factorize(df.label)[0]
        l1 = 1 + df.label.max()
        l2 = len(df[df.label==0])
        df.loc[df.label==0,'label'] = range(l1, l1+l2) #assig
n unique ids
        self.labels = df.copy().set_index('Image')
        self.names = df.copy().set_index('label')
        if path == TRAIN_CROPPED_IN:
            #data augmentation: 8 degree rotation, 10% stratc
h, shear
            tfms_g = [iaa.Affine(rotate=(-8, 8),mode='reflec
t',
                scale={"x": (0.9, 1.1), "y": (0.9, 1.1)}, she
ar=(-8,8))]
            #data augmentation: horizontal flip, hue and statu
ration augmentation,
            #gray scale, blur
            tfms_px = [iaa.Fliplr(0.5), iaa.AddToHueAndSatura
tion((-20, 20)),
                iaa.Grayscale(alpha=(0.0, 1.0)),iaa.GaussianB
lur((0, 1.0))]
            self.loader = Loader(path,tfms_g,tfms_px)
        else: self.loader = Loader(path)
        self.selection = None
        super().__init__(fnames, transform, path)

    def get_x(self, i):
        label = self.labels.loc[self.fnames[i],'label']
        #random selection of a positive example
        for j in range(10): #sometimes loc call fails
            try:
                names = self.names.loc[label].Image
                break
            except: None
        name_p = names if isinstance(names,str) else \
            random.sample(set(names) - set([self.fnames[i]]),
1)[0]
        #random selection of a negative example
        if(self.selection == None):
            for j in range(10): #sometimes loc call fails
                try:
                    names = self.names.loc[self.names.index!=
label].Image
                    break
                except: names = self.fnames[i]
            name_n = names if isinstance(names,str) else name
s.sample(1).values[0]
        else:
            name_n = self.selection.get(self.fnames[i])
        imgs = [self.loader(os.path.join(self.path,self.fname
s[i])),
                self.loader(os.path.join(self.path,name_p)),
                self.loader(os.path.join(self.path,name_n)),
                label,label,self.labels.loc[name_n,'label']]
        return imgs

    def get_y(self, i):
        return 0

    def get(self, tfm, x, y):
        if tfm is None:
            return (*x,0)
        else:
```

```
            x1, y1 = tfm(x[0],x[3])
            x2, y2 = tfm(x[1],x[4])
            x3, y3 = tfm(x[2],x[5])
            #combine all images into one tensor
            x = np.stack((x1,x2,x3),0)
            return x,(y1,y2,y3)

    def get_names(self,label):
        names = []
        for j in range(10):
            try:
                names = self.names.loc[label].Image
                break
            except: None
        return names

    @property
    def is_multi(self): return True
    @property
    def is_reg(self):return True

    def get_c(self): return n_embedding
    def get_n(self): return len(self.fnames)

#class for loading an individual images when embedding is comp
uted
class FilesDataset_single(FilesDataset):
    def __init__(self, data, path, transform):
        self.loader = Loader(path)
        fnames = os.listdir(path)
        super().__init__(fnames, transform, path)

    def get_x(self, i):
        return self.loader(os.path.join(self.path,self.fnames
[i]))

    def get_y(self, i):
        return 0

    @property
    def is_multi(self): return True
    @property
    def is_reg(self):return True

    def get_c(self): return n_embedding
    def get_n(self): return len(self.fnames)
```

In [7]:

```
def get_data(sz,bs,fname_emb=None,model=None):
    tfms = tfms_from_model(resnet34, sz, crop_type=CropType.N
O)
    tfms[0].tfms = [tfms[0].tfms[2],tfms[0].tfms[3]]
    tfms[1].tfms = [tfms[1].tfms[2],tfms[1].tfms[3]]
    df = pd.read_csv(LABELS)
    trn_df, val_df = train_test_split(df,test_size=0.2, rando
m_state=42)
    ds = ImageData.get_ds(pdFilesDataset, (trn_df,TRAIN_CROPP
ED_IN), (val_df,TRAIN_CROPPED_IN), tfms)
    md = ImageData(PATH, ds, bs, num_workers=nw, classes=None
)
    if fname_emb != None and model != None:
        emb = pd.read_csv(fname_emb)
        md.trn_dl.dataset.selection = Image_selection(md.trn_
dl.dataset.fnames,trn_df,
                                                      emb,mod
el)
```

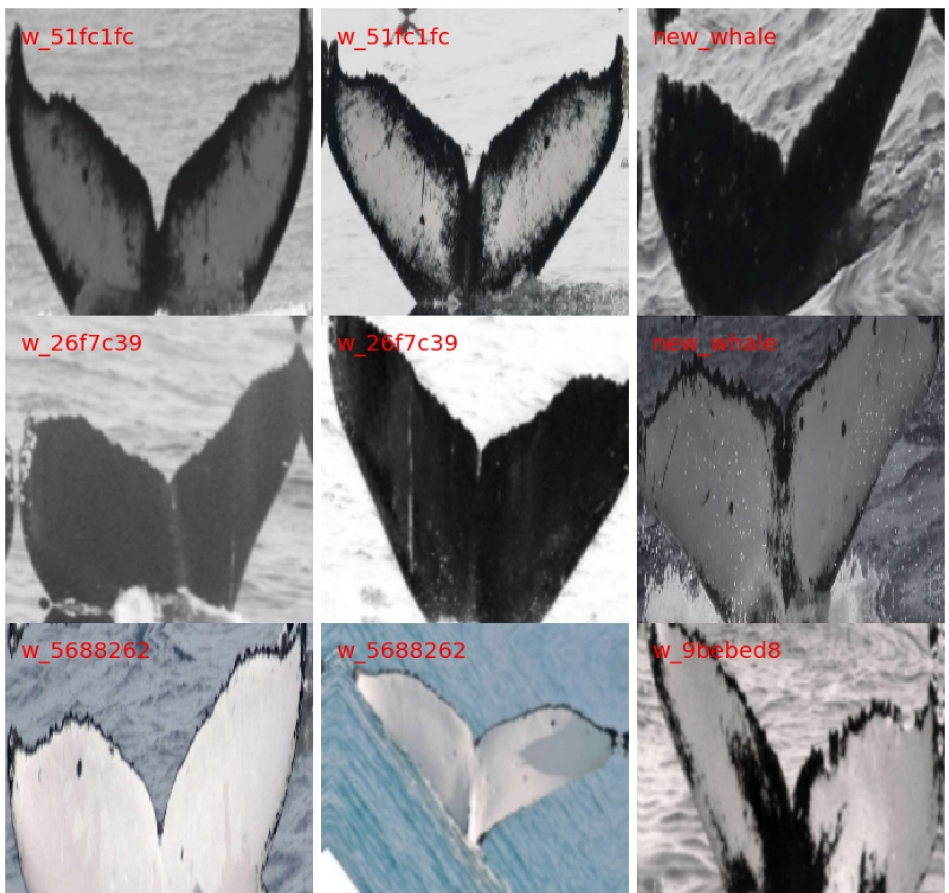
```
        md.val_dl.dataset.selection = Image_selection(md.val_
dl.dataset.fnames,val_df,
                                                      emb,mod
el)
    return md
```

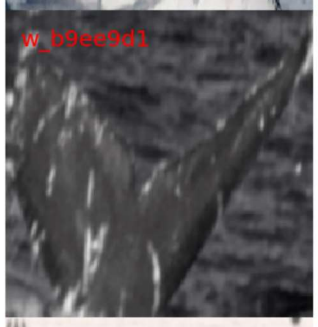
The image below demonstrates an example of triplets of rectangular 576x192 augmented images used for training. To be honest, some of those triplets are quite hard, and I don't think that I could even reach the same performance as the model after training (~99% accuracy in identifications of 2 similar images in a triplet).

In [8]:

```
md = get_data(sz,bs)

x,y = next(iter(md.trn_dl))
print(x.shape, y[0].shape)

def display_imgs(x,lbs=None):
    columns = 3
    rows = min(bs,16)
    ig,ax = plt.subplots(rows, columns, figsize=(columns*5, r
ows*5))
    for i in range(rows):
        for j in range(columns):
            idx = j+i*columns
            ax[i,j].imshow((x[j][i,:,:,:]*255).astype(np.int
))
            ax[i,j].axis('off')
            if lbs is not None:
                ax[i,j].text(10, 25, lbs[j][i], size=20, colo
r='red')
    plt.subplots_adjust(wspace=0, hspace=0)
    plt.show()

to_lb = md.trn_ds.names.Id.to_dict()
lbs = [[to_lb[idx] for idx in y_cur.tolist()] for y_cur in y]
display_imgs((md.trn_ds.denorm(x[:,0,:,:,:]),md.trn_ds.denorm
(x[:,1,:,:,:]),\
              md.trn_ds.denorm(x[:,2,:,:,:])),lbs)
```

```
torch.Size([32, 3, 3, 224, 224]) torch.Size([32])
```




w_b9ee9d1

w_b9ee9d1

w_939a97b

w_5f54045

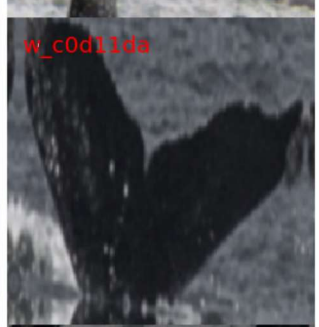
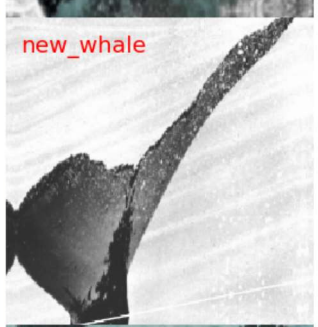
w_5f54045

w_8003858

w_bc7de9f

w_bc7de9f

w_d0475b2

w_c0d11da

w_c0d11da

new_whale

w_5597026

w_5597026

new_whale

w_5597026

w_5597026

new_whale

w_9c506f6

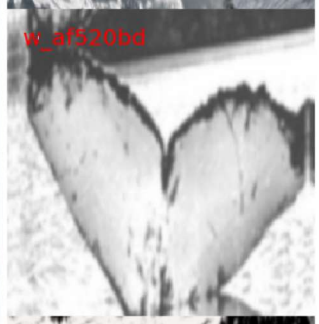
w_9c506f6

w_df32eec

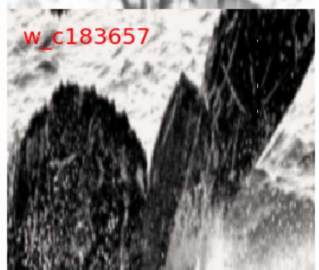
w_6afdf63

w_6afdf63

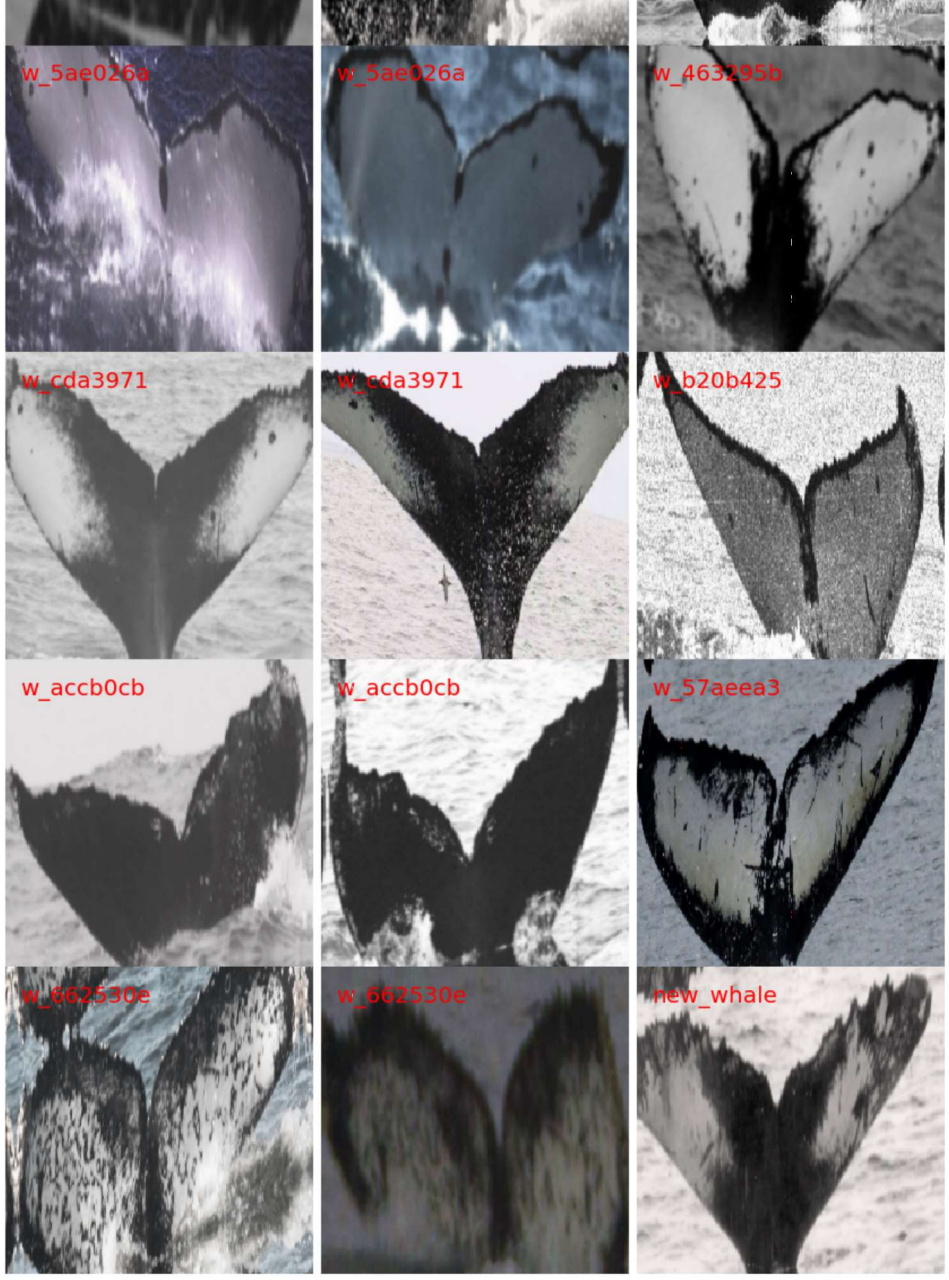
w_53feb91

w_af520bd

w_af520bd

new_whale

w_c183657

w_c183657

w_32eac70



Model

In this kernel I use ResNeXt50 instead of ResNet34 since it gaves slightly better performance after training within kernel time limit. Looking to the images I would expect that larger models should perform better in this competition since there are many really difficult examples that require perception capability surpassing human ones (check examples in the last part of the kernel). The convolutional part is taken from the original ResNeXt50 model pretrained on ImageNet, meanwhile adaptive pooling allows using of images of any sizes and aspect ratios. On the top, 2 fully connected layers are added to convert the prediction of convolutional part into embedding space.

On the top, a learnable metric is applied that converts differences of vector components in the embedding space for all images within a batch into distances. This part replaces calculation of distances in Euclidean space by one in distorted space. The first thing that came me in mind is generalized formula for a distance in a linear space d^2 = (v1-v2).T x A x (v1-v2), where A is a trainable matrix. For Euclidian space d^2 = (v1-v2).T x (v1-v2), i.e. A is a identity matrix. Using of such distance boosted the score of the v2 version of this kernel from 0.606 to 0.655. However, the above form of the metric is quite slow at the inference time when distances for all image pairs are calculated. Also, it is quite difficult to impose a constrain on A during training to make it positive semi defined (distance is always positive). Therefore, I use an alternative approximation formulation for distance calculation that is much faster at the inference time, symmetric and always positive, and have similar (or slightly better) performance. If after calculation of differences between v1 and v2 the result goes through a linear layer, before summing the squares, the distance would include also contribution from mixed terms like dv1 * dv2 (like in an approach with matrix A), though many of these terms are correlated, and results are a little bit worse (it can be viewed as a factorization of the general approach). However, when I included also (v1-v2)^2 terms as an input, the performance of such metric surpassed one for d^2 = (v1-v2).T x A x (v1-v2). It can be explained as an approximation of some nonlinear transformation of a space including square, cubic, and quadratic terms. The concept of metric learning is similar to one in Martin's kernel (https://www.kaggle.com/martinpiotte/whale-recognition-model-with-score-0-78563). However, in that case the metric converts the difference in the probability of two classes to have the same label, while in the current kernel it converts the difference between vectors into a distance in a distorted space.

In [9]:
```
class Metric(nn.Module):
    def __init__(self, emb_sz=64):
        super().__init__()
        self.l = nn.Linear(emb_sz*2,emb_sz*2,False)

    def forward(self,d):
        d2 = d.pow(2)
        d = self.l(torch.cat((d,d2),dim=-1))
        x = d.pow(2).sum(dim=-1)
        return x.view(-1)
```

In [10]:
```
def resnext50(pretrained=True):
    model = resnext_50_32x4d()
    name = 'resnext_50_32x4d.pth'
    if pretrained:
        path = os.path.join(MODLE_INIT,name)
        load_model(model, path)
    return model

class TripletResneXt50(nn.Module):
    def __init__(self, pre=True, emb_sz=64, ps=0.5):
        super().__init__()
        encoder = resnext50(pretrained=pre)
        #add DataParallel to allow support of multiple GPUs
        self.cnn = nn.DataParallel(nn.Sequential(encoder[0],e
ncoder[1],nn.ReLU(),
                           encoder[3],encoder[4],encoder[5],enco
der[6],encoder[7]))
        self.head = nn.DataParallel(nn.Sequential(AdaptiveCon
catPool2d(), Flatten(),
                           nn.Dropout(ps),nn.Linear(4096, 512),
nn.ReLU(),
                           nn.BatchNorm1d(512), nn.Dropout(ps),
nn.Linear(512, emb_sz)))
        self.metric = nn.DataParallel(Metric(emb_sz))

    def forward(self,x):
        x1,x2,x3 = x[:,0,:,:,:],x[:,1,:,:,:],x[:,2,:,:,:]
        x1 = self.head(self.cnn(x1))
        x2 = self.head(self.cnn(x2))
        x3 = self.head(self.cnn(x3))
        x = torch.cat((x1,x2,x3))
        sz = x.shape[0]
        x1 = x.unsqueeze(1).expand((sz,sz,-1))
        x2 = x1.transpose(0,1)
        #matrix of all vs all differencies
        d = (x1 - x2).view(sz*sz,-1)
        return self.metric(d)

    def get_embedding(self, x):
        return self.head(self.cnn(x))

    def get_d(self, x0, x):
        d = (x - x0)
        return self.metric(d)

class ResNeXt50Model():
    def __init__(self,pre=True,name='TripletResneXt50',**kwar
gs):
        self.model = to_gpu(TripletResneXt50(pre=True,**kwarg
s))
        self.name = name
```

```
    def get_layer_groups(self, precompute):
        m = self.model.module if isinstance(self.model,FP16)
else self.model
        if precompute:
            return [m.head] + [m.metric]
        c = children(m.cnn.module)
        return list(split_by_idxs(c,[5])) + [m.head] + [m.met
ric]
```

Loss function

I my tests I have performed a comparison of several loss functions and found that contrastive loss works the best in the current setup. I also, in comparison with the previous version, select only nonzero terms when perform averaging. I found that at a later stage of training many pairs are already well separated and contribute zero to gradients while decrease the magnitude of useful gradients during averaging. The drawback of such an approach is that during training **values of train and validation loss must be ignored, since they are calculated each time based on different number of pairs**, and only the value of metric (like accuracy of identifying a correct pair of images with the same label in a triplet, T_acc, or accuracy of identifying a correct pair of images with the same label within a hardest triplet in batch for each anchor image, BH_acc) must be tracked. However, after performing mining hardest negative examples, even accuracy metrics are not reliable, and the only way to check performance of the model is validation based on the entire validation dataset rather than batches with using the same metric as one in the competition.

In [11]:

```
def Contrastive_loss(d, target, size_average=True, m=10.0):
    d = d.float()
    #matrix of all vs all comparisons
    t = torch.cat(target)
    sz = t.shape[0]
    t1 = t.unsqueeze(1).expand((sz,sz))
    t2 = t1.transpose(0,1)
    y = (t1==t2).view(-1)

    loss_p = d[y==1]
    loss_n = F.relu(m - torch.sqrt(d[y==0]))**2
    loss = torch.cat((loss_p,loss_n),0)
    loss = loss[torch.nonzero(loss).squeeze()]
    loss = loss.mean() if size_average and loss.shape[0] > 0
else loss.sum()
    return loss

#accuracy within a triplet
def T_acc(d, target):
    sz = target[0].shape[0]
    lp = [3*sz*i + i+sz for i in range(sz)]
    ln = [3*sz*i + i+2*sz for i in range(sz)]
    dp, dn = d[lp], d[ln]
    return (dp < dn).float().mean()

#accuracy within a hardest triplet in a batch for each anchor
 image
def BH_acc(d, target):
    t = torch.cat(target)
    sz = t.shape[0]
    t1 = t.unsqueeze(1).expand((sz,sz))
    t2 = t1.transpose(0,1)
    y = (t1==t2)
    d = d.float().view(sz,sz)
    BH = []
    for i in range(sz):
        dp = d[i,y[i,:] == 1].max()
        dn = d[i,y[i,:] == 0].min()
        BH.append(dp < dn)
    return torch.FloatTensor(BH).float().mean()
```

I also tried batch hard triplet loss introduced in https://arxiv.org/pdf/1703.07737.pdf (https://arxiv.org/pdf/1703.07737.pdf) , however I didn't get better results when my current one yet. The problem, I expect, is that triplet loss doesn't try to bring all predictions with the same label to one point in an embedding space, while keeping objects of the same kind together. If would be great if there was no new_whale class. The check I perform is based on the distance to the nearest neighbors: if there are no whales in a sphere of a particular radius, I assign new whale label. It works well if the points are packed with approximately the same density, but if for some classes the spacing between images of the same class is too large, they will be misclassified as a new_whale.

The class below implements such approach. The thing I have introduced here is a parameter n that indicates how many hardest triplets should be selected per batch, which allows gradually going from batch all to batch hard loss during training, when the fraction of hard triplets drops (though, sorting takes additional time).

In [12]:
```
#batch hard loss: https://arxiv.org/pdf/1703.07737.pdf
class BH_loss(nn.Module):
    def __init__(self, n=1, m=0.0):
        super().__init__()
        self.n = n #select n hardest triplets
        self.m = m

    def forward(self, input, target, size_average=True):
        #matrix of all vs all comparisons
        t = torch.cat(target)
        sz = t.shape[0]
        t1 = t.unsqueeze(1).expand((sz,sz))
        t2 = t1.transpose(0,1)
        y = (t1==t2)
        d = input.float().view(sz,sz)
        D = []
        for i in range(sz):
            dp = d[i,y[i,:] == 1].max()
            dn = d[i,y[i,:] == 0]
            dist, idxs = torch.sort(dn)
            n = min(self.n,dn.shape[0])
            for j in range(n):
                D.append((self.m + dp - dn[idxs[j]]).unsqueez
e(0))
        loss = torch.log1p(torch.exp(torch.cat(D)))
        loss = loss.mean() if size_average else loss.sum()
        return loss
```

Training

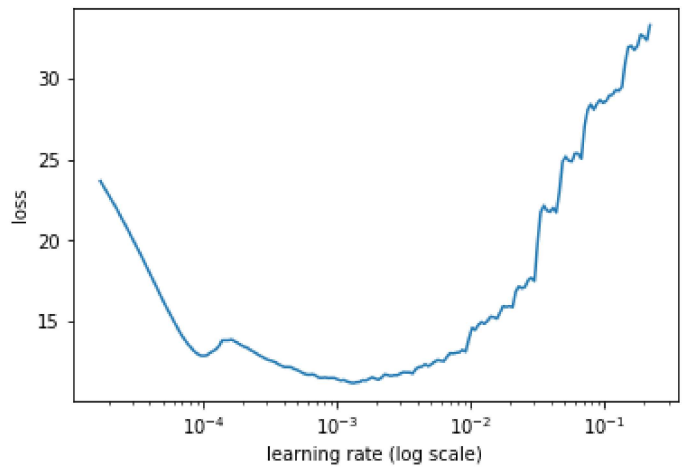
In [13]:
```
learner = ConvLearner(md,ResNeXt50Model(ps=0.0,emb_sz=n
_embedding))
learner.opt_fn = optim.Adam
learner.clip = 1.0 #gradient clipping
learner.crit = Contrastive_loss
learner.metrics = [T_acc,BH_acc]
learner.freeze_to(-2) #unfreez metric and head block
learner #click "output" to see details of the model
```

Output

I begin with finding the optimal learning rate. The following function runs training with different learning rate and records the loss. Increase of the loss indicates onset of divergence of training. The optimal lr lies in the vicinity of the minimum of the curve but before the onset of divergence. Based on the following plot, for the current setup the divergence starts at ~3e-3, and the recommended learning rate is ~5e-4.

In [14]:
```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    learner.lr_find()
learner.sched.plot()
```

```
75%|███████   | 193/257 [06:45<02:11,  2.06s/it, loss=45.2]
```



First, I train only the fully connected part of the model and the metric while keeping the rest frozen. It allows to avoid corruption of the pretrained weights at the initial stage of training due to random initialization of the head layers. So the power of transfer learning is fully utilized when the training is continued.

In [15]:
```
lr = 0.5e-3
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    learner.fit(lr,1)
```

epoch	trn_loss	val_loss	T_acc	BH_acc
0	8.938998	9.609752	0.851388	0.434497

Next, I unfreeze all weights and allow training of entire model. One trick that I use is applying different learning rates in different parts of the model: the learning rate in the fully connected part is still lr, last two blocks of ResNeXt are trained with lr/3, and first layers are trained with lr/10. Since low-level detectors do not vary much from one image data set to another, the first layers do not require substantial retraining compared to the parts of the model working with high level features. Another trick is learning rate annealing. Periodic learning rate increase followed by slow decrease drives the system out of steep minima (when lr is high) towards broader ones (which are explored when lr decreases) that enhances the ability of the model to generalize and reduces overfitting. The length of the cycles gradually increases during training. Usage of half precision doubles the maximum batch size that allows to compare more pairs in each batch.

In [16]:
```
learner.unfreeze() #unfreeze entire model
lrs=np.array([lr/10,lr/3,lr,lr])
learner.half() #half precision
```

Since the loss is calculated as an average of nonzero terms, as mentioned above, it's value is not relaiable and must be ignored. Instead the values of T_acc and BH_acc metrics should be considered.

In [17]:
```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    learner.fit(lrs/2,4,cycle_len=1,use_clr=(10,20))
    learner.fit(lrs/4,4,cycle_len=2,use_clr=(10,20))
learner.save('model0')
```

epoch	trn_loss	val_loss	T_acc	BH_acc
0	9.707771	10.721808	0.974933	0.770516
1	9.419538	10.883561	0.984781	0.871382
2	10.572673	12.657548	0.986571	0.890779
3	10.47062	16.134393	0.988362	0.908087

epoch	trn_loss	val_loss	T_acc	BH_acc
0	9.682955	14.6278	0.987466	0.915249
1	8.888745	20.582505	0.989257	0.913459
2	8.486953	21.414116	0.985676	0.911071
3	7.435253	20.467691	0.990152	0.923008
4	7.739829	18.033643	0.989257	0.925097
5	6.866274	19.669115	0.991047	0.926291
6	7.148704	22.658118	0.989257	0.9245
7	5.948245	21.839638	0.992838	0.934945

Embedding

The following code converts images into embedding vectors, which are used later to generate predictions based on the nearest neighbor search in a space deformed by a metric (see Metric class).

In [18]:
```
def extract_embedding(model,path):
    tfms = tfms_from_model(resnet34, sz, crop_type=CropType.NO)
    tfms[0].tfms = [tfms[0].tfms[2],tfms[0].tfms[3]]
    tfms[1].tfms = [tfms[1].tfms[2],tfms[1].tfms[3]]
    ds = ImageData.get_ds(FilesDataset_single, (None,TRAIN_CROPPED_IN), (None,TRAIN_CROPPED_IN),
         tfms, test=(None,path))
    md = ImageData(PATH, ds, 3*bs, num_workers=nw, classes=None)
    model.eval()
    with torch.no_grad():
        preds = torch.zeros((len(md.test_dl.dataset), n_embedding))
        start=0
        for i, (x,y) in tqdm_notebook(enumerate(md.test_dl,start=0),
                                      total=len(md.test_dl)):
            size = x.shape[0]
            m = model.module if isinstance(model,FP16) else model
            preds[start:start+size,:] = m.get_embedding(x.half())
            start+= size
        return preds, [os.path.basename(name) for name in md.test_dl.dataset.fnames]
```

In [19]:

```
emb, names = extract_embedding(learner.model,TRAIN_CROPPED_IN
)
df = pd.DataFrame({'files':names,'emb':emb.tolist()})
df.emb = df.emb.map(lambda emb: ' '.join(list([str(i) for i i
n emb])))
df.to_csv('train_emb0.csv', header=True, index=False)
```



```
i) for i in emb])))
df_test.to_csv('test_emb0.csv', header=True, index=False)
```


Validation

Find 16 nearest train neighbors in embedding space for each validation image. Since there can be several neighbors with the same label, instead of 5 I use 16 here. The following code will select 5 nearest neighbors with different labels. "new_whale" label can be assigned as a prediction at a distance dcut. In this case, if the number of neighbors at a distance shorter than dcut is less than 5, the image is considered to be different from others, and "new_whale" is assigned.

In [20]:

```
def get_nbs(model,x,y,n=16):
    d, idxs = [], []
    sz = x.shape[0]
    model.eval()
    with torch.no_grad():
        m = model.module if isinstance(model,FP16) else model
        m = m.module if isinstance(m,nn.DataParallel) else m
        for i in tqdm_notebook(range(sz)):
            preds = m.get_d(x[i],y)
            sorted, indices = torch.sort(preds)
            d.append(to_np(sorted[:n]))
            idxs.append(to_np(indices[:n]))
    return np.stack(d), np.stack(idxs)

def get_val_nbs(model,emb_df,out='val.csv',dcut=None):
    emb_df = emb_df.copy()
    data = pd.read_csv(LABELS).set_index('Image')
    emb_df['emb'] = [[float(i) for i in s.split()] for s in e
mb_df['emb']]
    emb_df.set_index('files',inplace=True)
    train_df = data.join(emb_df)
    train_df = train_df.reset_index()
    #the split should be the same as one used for training
    trn_df, val_df = train_test_split(train_df,test_size=0.2,
 random_state=42)
    trn_preds = np.array(trn_df.emb.tolist())
    val_preds = np.array(val_df.emb.tolist())
    trn_df = trn_df.reset_index()
    val_df = val_df.reset_index()

    trn_preds = torch.from_numpy(trn_preds).half().cuda()
    val_preds = torch.from_numpy(val_preds).half().cuda()
    trn_d,trn_idxs = get_nbs(model,val_preds,trn_preds)

    s = []
    for l1 in trn_d.tolist():
        s.append(' '.join([str(l2) for l2 in l1]))
    val_df['d'] = s
```

```
    val_df['nbs'] = [' '.join(trn_df.loc[trn_idxs[index]].Id.
tolist()) \
                     for index, row in val_df.iterrows()]
    val_df[['Image','Id','nbs','d']].to_csv(out, header=True,
 index=False)

    if dcut is not None:
```



```
            nbs = dict()
            for i in range(16): #16 neighbors
                nb = trn_idxs[idx,i]
                l, s = trn_df.loc[nb].Id, trn_d[idx,i]
                if s > dcut and 'new_whale' not in nbs: nbs[
'new_whale'] = dcut
                if l not in nbs: nbs[l] = s
                if len(nbs) >= 5: break
            nbs_sorted = list(nbs.items())
            score = 0.0
            for i in range(min(len(nbs),5)):
                if nbs_sorted[i][0] == l0:
                    score = 1.0/(i + 1.0)
                    break
            scores.append(score)
        print(np.array(scores).mean(), flush=True)
```

In [21]:

```
dcut = 7.5 #fit this parameter based on validation
get_val_nbs(learner.model,df,dcut=dcut,out='val0.csv')
```

```
0.7429627439384979
```

Submission

In [22]:

```
def get_test_nbs(model,trn_emb,test_emb,out='test.csv',\
                 submission='submission.csv',dcut=None):
    trn_emb = trn_emb.copy()
    data = pd.read_csv(LABELS).set_index('Image')
    trn_emb['emb'] = [[float(i) for i in s.split()] for s in
trn_emb['emb']]
    trn_emb.set_index('files',inplace=True)
    train_df = data.join(trn_emb)
    train_df = train_df.reset_index()
    train_preds = np.array(train_df.emb.tolist())
    test_emb = test_emb.copy()
    test_emb['emb'] = [[float(i) for i in s.split()] for s in
 test_emb['emb']]
    test_emb['Image'] = test_emb['files']
    test_emb.set_index('files',inplace=True)
    test_df = test_emb.reset_index()
    test_preds = np.array(test_df.emb.tolist())
    train_preds = torch.from_numpy(train_preds).half().cuda()
    test_preds = torch.from_numpy(test_preds).half().cuda()
    test_d,test_idxs = get_nbs(model,test_preds,train_preds)

    s = []
    for l1 in test_d.tolist():
        s.append(' '.join([str(l2) for l2 in l1]))
    test_df['d'] = s
    test_df['nbs'] = [' '.join(train_df.loc[test_idxs[index]]
```

```
.Id.tolist()) \
                      for index, row in test_df.iterrows()]
    test_df[['Image','nbs','d']].to_csv(out, header=True, ind
ex=False)

    if dcut is not None:
        pred = []
        for idx, row in test_df.iterrows():
            nbs = dict()
            for i in range(0,16):
                nb = test_idxs[idx,i]
                l, s = train_df.loc[nb].Id, test_d[idx,i]
                if s > dcut and 'new_whale' not in nbs: nbs[
'new_whale'] = dcut
                if l not in nbs: nbs[l] = s
                if len(nbs) >= 5: break
            nbs_sorted = list(nbs.items())
            p = ' '.join([lb[0] for lb in nbs_sorted])
            pred.append({'Image':row.files,'Id':p})
        pd.DataFrame(pred).to_csv(submission,index=False)
```

In [23]:

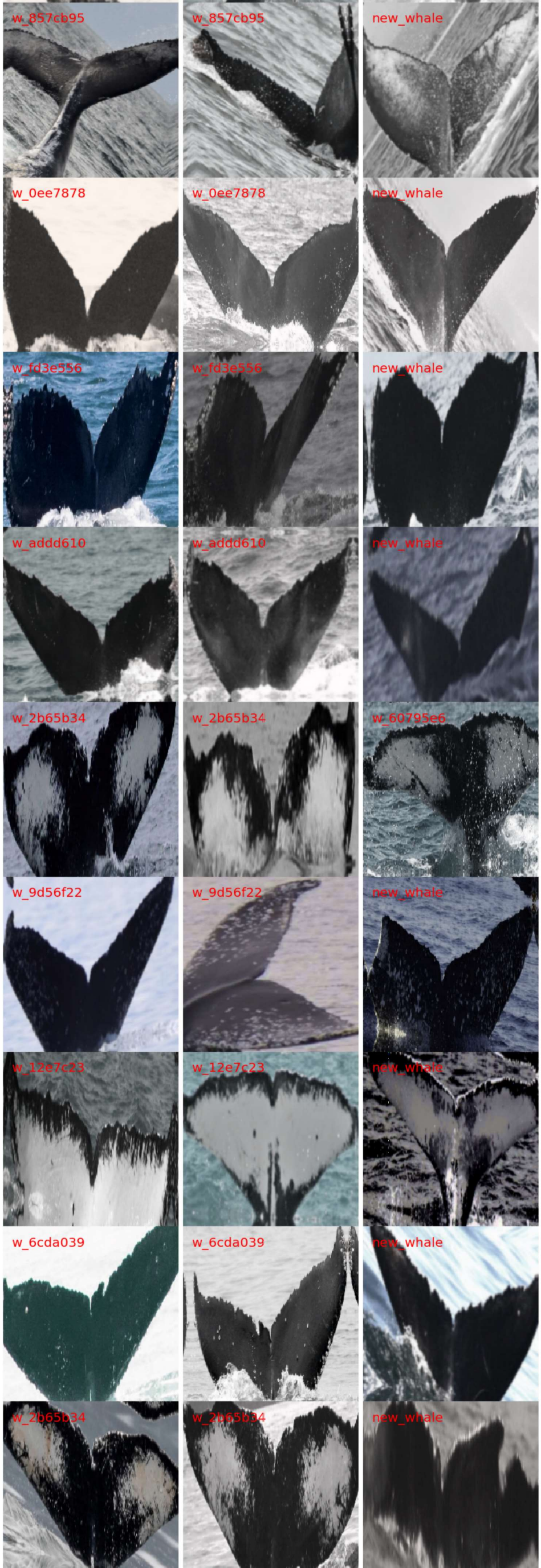
```
get_test_nbs(learner.model,df,df_test,dcut=dcut,out='test0.cs
v',\
    submission='submission0.csv')
```

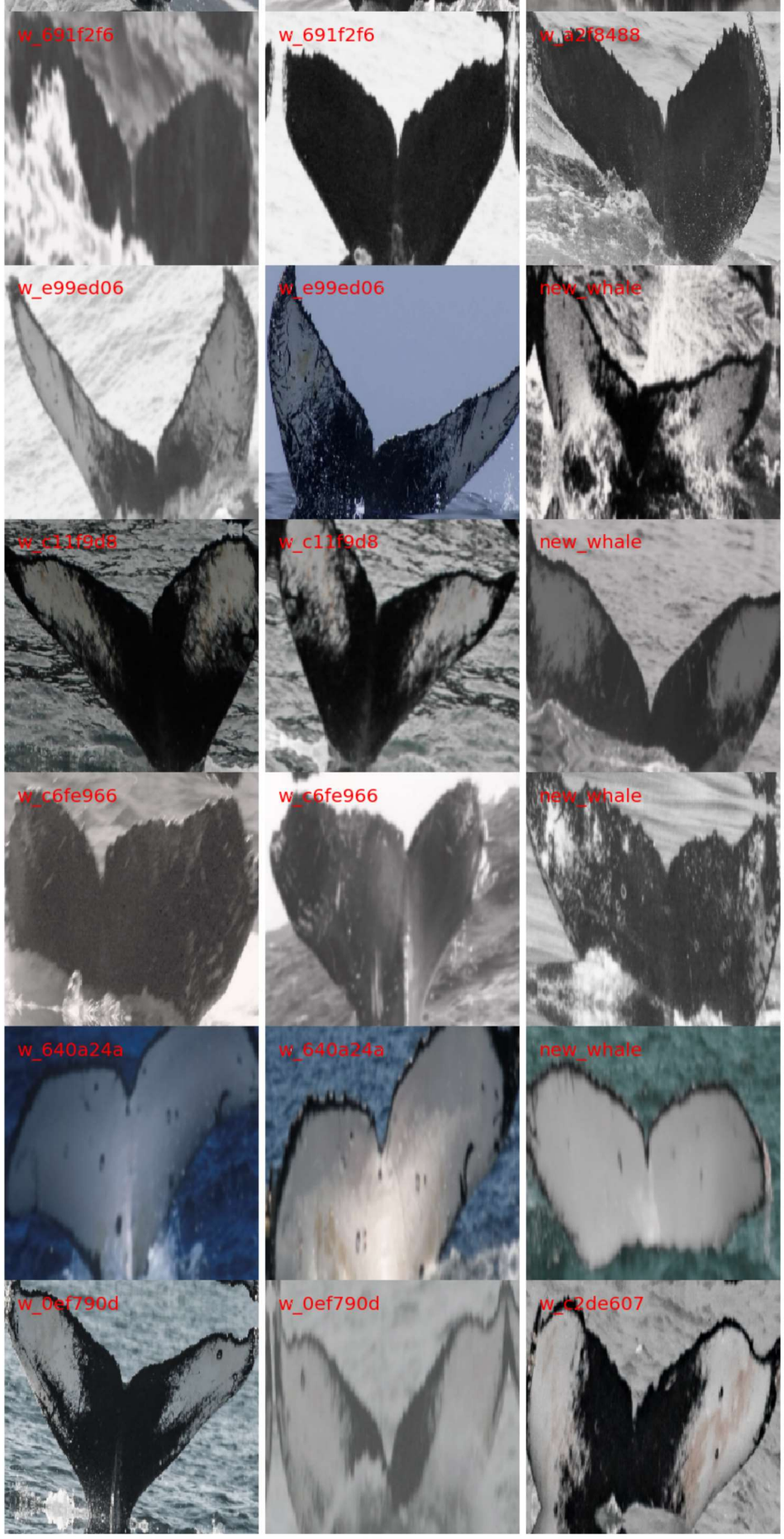
Hard negative example mining

The code below selects the most similar negative examples to images in the dataset. **They are really tough**, and as Haider Alwasiti (https://www.kaggle.com/hwasiti) wrote in comments, I feel really sorry for the network to do such job. The following stage of training is performed on triplets with these hard negative example. Since distribution of these examples is different from one used at the previous stage of training, not only loss, but also metrics (T_acc, BH_acc) don't give a reliable estimation of the model performance, and only a way to check the model is validation based on the entire validation dataset with using the same metric as one in the competition.

In [24]:

```
md = get_data(sz,bs,'train_emb0.csv',learner.model)
learner.set_data(md)
learner.unfreeze()
learner.half()
x,y = next(iter(md.trn_dl))

to_lb = md.trn_ds.names.Id.to_dict()
lbs = [[to_lb[idx] for idx in y_cur.tolist()] for y_cur in y]
display_imgs((md.trn_ds.denorm(x[:,0,:,:,:]),md.trn_ds.denorm
(x[:,1,:,:,:]),\
              md.trn_ds.denorm(x[:,2,:,:,:])),lbs)
```


w_857cb95
w_857cb95
new_whale
w_0ee7878
w_0ee7878
new_whale
w_fd3e556
w_fd3e556
new_whale
w_addd610
w_addd610
new_whale
w_2b65b34
w_2b65b34
w_60795e6
w_9d56f22
w_9d56f22
new_whale
w_12e7c23
w_12e7c23
new_whale
w_6cda039
w_6cda039
new_whale
w_2b65b34
w_2b65b34
new_whale



In [25]:

```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    learner.fit(lrs/16,3,cycle_len=4,use_clr=(10,20))
learner.save('model')
```

epoch	trn_loss	val_loss	T_acc	BH_acc
0	8.85219	41.941863	0.920322	0.912563
1	7.918573	39.363146	0.910474	0.914652
2	7.789009	41.286498	0.907789	0.909281
3	7.36295	38.240309	0.912265	0.921218
4	7.453082	41.458846	0.910474	0.919725
5	7.197116	49.950878	0.91316	0.92271
6	6.863281	48.217589	0.912265	0.920024
7	6.978485	44.182166	0.918532	0.919725
8	6.799551	41.010949	0.918532	0.916741
9	6.556124	50.496629	0.910474	0.917637

10	6.260569	53.335509	0.914951	0.913757
11	6.044919	39.754581	0.919427	0.923306

In [26]:
```
emb, names = extract_embedding(learner.model,TRAIN_CROPPED_IN
)
df = pd.DataFrame({'files':names,'emb':emb.tolist()})
df.emb = df.emb.map(lambda emb: ' '.join(list([str(i) for i i
n emb])))
df.to_csv('train_emb.csv', header=True, index=False)

emb, names = extract_embedding(learner.model,TEST_CROPPED_IN)
df_test = pd.DataFrame({'files':names,'emb':emb.tolist()})
df_test.emb = df_test.emb.map(lambda emb: ' '.join(list([str(
i) for i in emb])))
df_test.to_csv('test_emb.csv', header=True, index=False)
```

In [27]:
```
dcut = 16.0 #fit this parameter based on validation
get_val_nbs(learner.model,df,dcut=dcut,out='val.csv')
get_test_nbs(learner.model,df,df_test,dcut=dcut,out='test.cs
v',\
    submission='submission.csv')
```

```
0.7503843879361324
```

002b4615d.jpg
00600ce17.jpg
00d641885.jpg
00eaedfab.jpg
00fee3975.jpg
010a1f0eb.jpg
01237f1ce.jpg
01dcb420f.jpg
0202dfb29.jpg
020ab0f9b.jpg
1000+ more

- **sample_submission.csv** - a sample submission file in the correct format

Output Files

New Dataset | New Kernel | Download All

Output Files

- submission.csv
- submission0.csv
- test.csv
- test0.csv
- test_emb.csv
- test_emb0.csv
- train_emb.csv
- train_emb0.csv
- val.csv

About this file

Submit to Competition

This file was created from a Kernel, it does not have a description.

submission.csv

	Id	Image
2	w_6ca0970 new_whale w_f01ade6 w_8fda8b2 w_82481c1	aabc5cf3b.jpg
3	w_966b510 new_whale w_dd08014 w_cc09685 w_ac51301	7917b34f8.jpg
4	new_whale w_e8b5980 w_eae373b w_94dcd2b w_aa585b1	b82051bad.jpg
5	new_whale w_c8bbb43 w_9143ee7 w_1ccea93 w_3904b99	2a2c4a661.jpg
6	w_83d10c1 new_whale w_fd4376a w_8fad4d2 w_689e1cd	3ab77e041.jpg
7	new_whale w_b4841bd	50a4d8475.jpg
8	new_whale w_7342df0 w_75f6ffa w_bb2cbcf w_4e88955	c02d5e7b0.jpg
9	w_214e081 new_whale w_cbb15d2 w_9dbdb42 w_72e3fa4	22ef3c6bd.jpg

10	new_whale w_f9bffb5 w_7d09141	c2c4f4ad6 .jpg

Run Info

Succeeded	True	Run Time	27346.1 seconds
Exit Code	0	Queue Time	0 seconds
Docker Image Name	/python(Dockerfile)	Output Size	0
Timeout Exceeded	False	Used All Space	False
Failure Message			

Log

Download Log

```
Time   Line #  Log Message
2.7s         1  [NbConvertApp] Converting notebook __notebook__.ipynb to
                notebook
2.7s         2  [NbConvertApp] Executing notebook with kernel: python3
27342.0s     3  [NbConvertApp] Writing 11664040 bytes to __notebook__.ipynb
27344.5s     4  [NbConvertApp] Converting notebook __notebook__.ipynb to html
27345.6s     5  [NbConvertApp] Support files will be in __results___files/
                [NbConvertApp] Making directory __results___files
27345.6s     6  [NbConvertApp] Making directory __results___files
27345.6s     7  [NbConvertApp] Making directory __results___files
27345.6s     8  [NbConvertApp] Writing 460951 bytes to __results__.html
27345.6s     9
27345.6s    11  Complete. Exited with code 0.
```

Comments (0)

Click here to enter a comment...