In [2]:

```
from __future__ import print_function
#%matplotlib inline
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
```

In [14]:

```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes

batch_size = 16
workers = 2

dataroot = 'data/celebs'

dset = datasets.ImageFolder(root=dataroot,
                               transform=transforms.Compose([
                                   transforms.Resize(256),
                                   transforms.CenterCrop(224),
                                   transforms.RandomHorizontalFlip(), # randoml
y flip and rotate
                                   transforms.RandomRotation(10),
                                   transforms.ToTensor(),
                                   transforms.Normalize(
                                       mean=[0.485, 0.456, 0.406],
                                       std=[0.229, 0.224, 0.225]
                                   )]))
print(dset)
train_set, test_set, val_set = torch.utils.data.random_split(dset, [2500, 1000,
621])

loaders = list(map(lambda x: torch.utils.data.DataLoader(x, batch_size=batch_siz
e,
                                        shuffle=True, num_workers=workers),[tra
in_set, test_set, val_set]))
```

```
Dataset ImageFolder
    Number of datapoints: 4121
    Root location: data/celebs
    StandardTransform
Transform: Compose(
               Resize(size=256, interpolation=PIL.Image.BILINEAR)
               CenterCrop(size=(224, 224))
               RandomHorizontalFlip(p=0.5)
               RandomRotation(degrees=(-10, 10), resample=False, exp
and=False)
               ToTensor()
               Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.2
24, 0.225])
           )
```

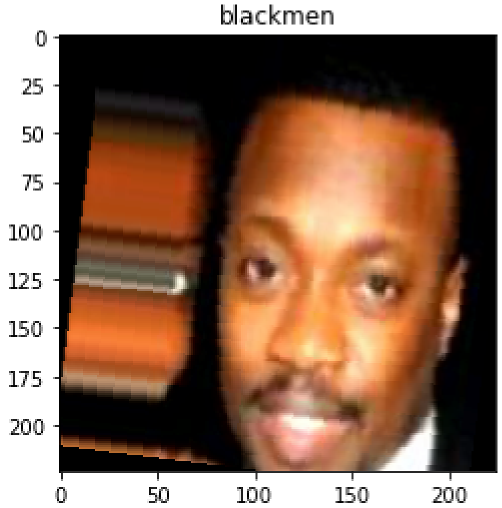
In [15]:

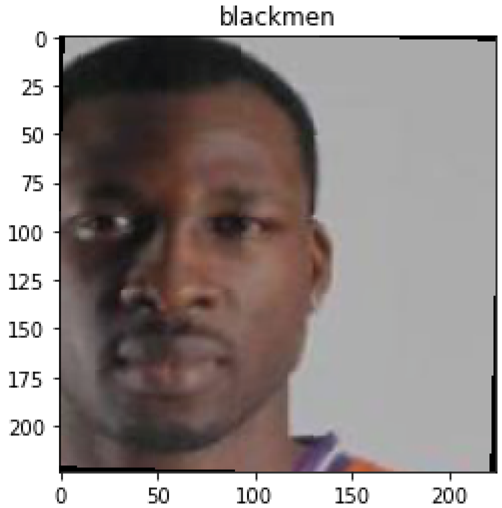
```
class_names = dset.classes
nb_classes = len(class_names)

print("Number of classes:", nb_classes)
print("\nClass names: \n\n", class_names)
```

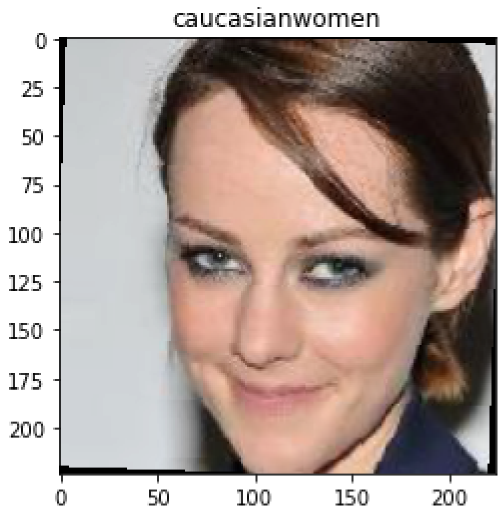
```
Number of classes: 8

Class names:

 ['blackmen', 'blackwomen', 'blondmen', 'blondwomen', 'caucasianme
n', 'caucasianwomen', 'tannedmen', 'tannedwomen']
```

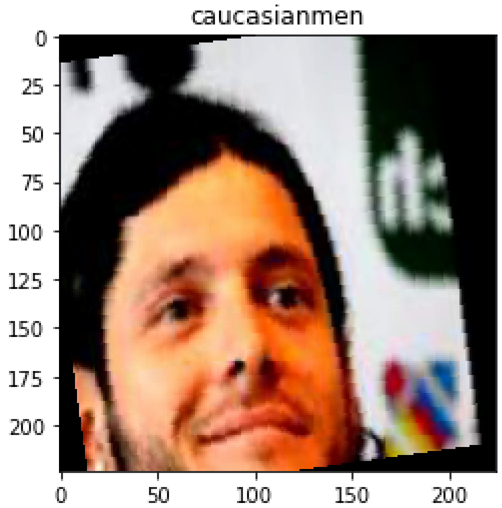
In [18]:

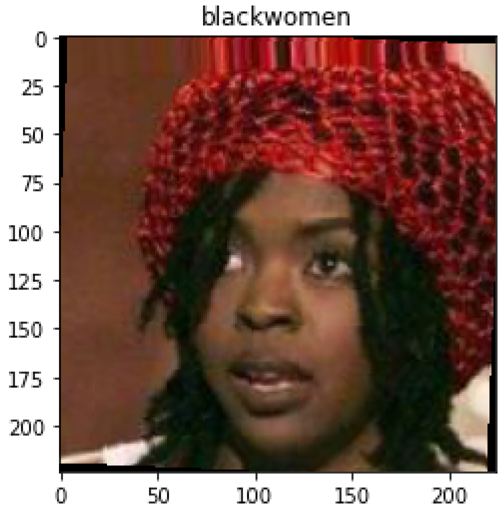
```
# Get a batch of training data
inputs, classes = next(iter(loaders[2]))

for image, label in zip(inputs, classes):
    image = image.to("cpu").clone().detach()
    image = image.numpy().squeeze()
    image = image.transpose(1,2,0)
    image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.
406))
    image = image.clip(0, 1)

    fig = plt.figure(figsize=(16,4))
    plt.imshow(image)
    plt.title(class_names[label])
```

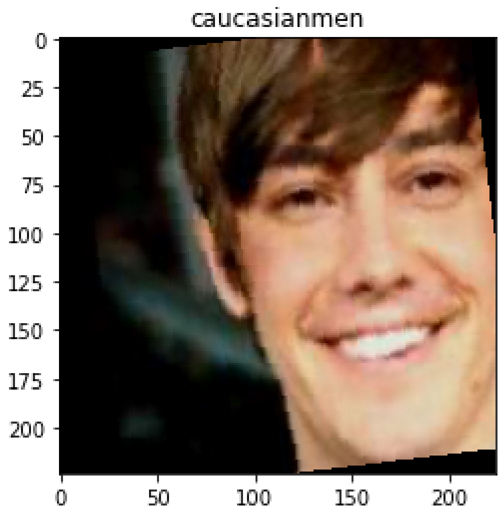

blackmen
0
25
50
75
100
125
150
175
200
0 50 100 150 200

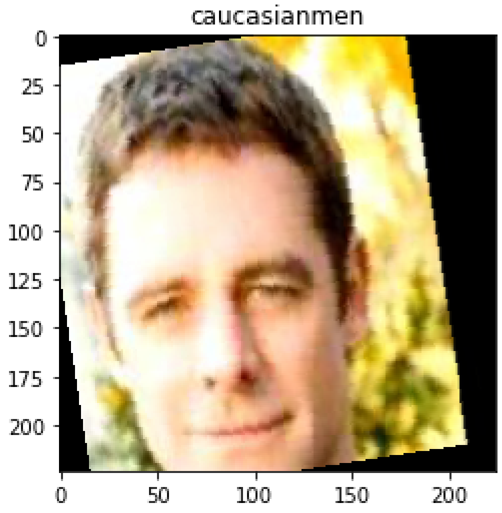

blackmen
0
25
50
75
100
125
150
175
200
0 50 100 150 200

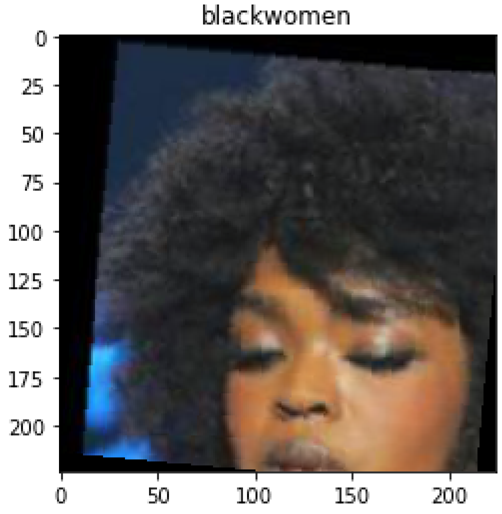

caucasianwomen
0
25
50
75
100
125
150
175
200
0 50 100 150 200

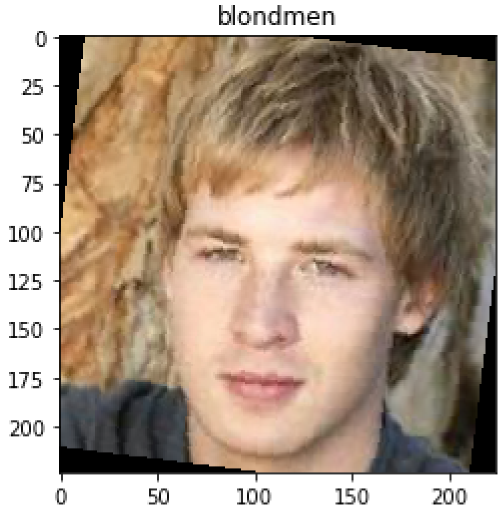

caucasianmen
0
25
50
75
100
125
150
175
200
0 50 100 150 200

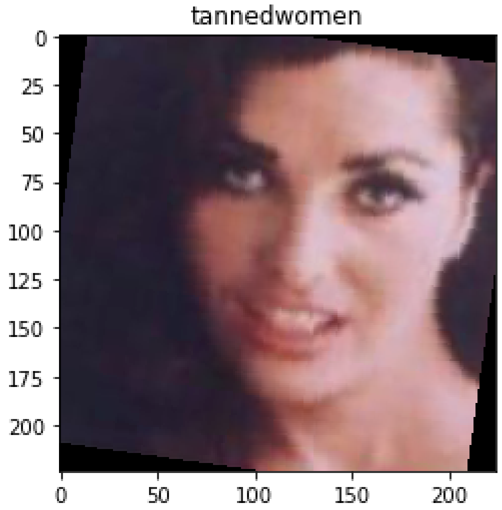

blackwomen
0
25
50
75
100
125
150
175
200
0 50 100 150 200

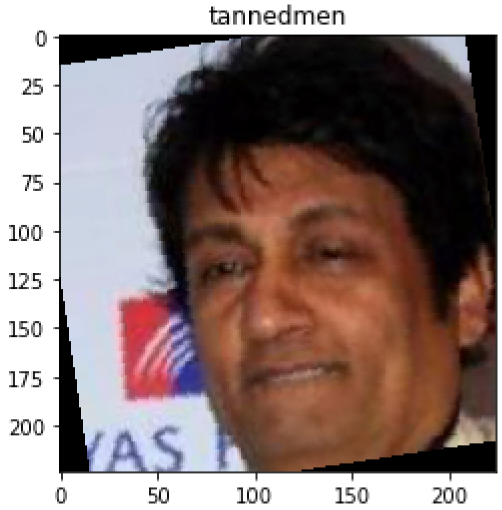

caucasianmen
0
25
50
75
100
125
150
175
200
0 50 100 150 200

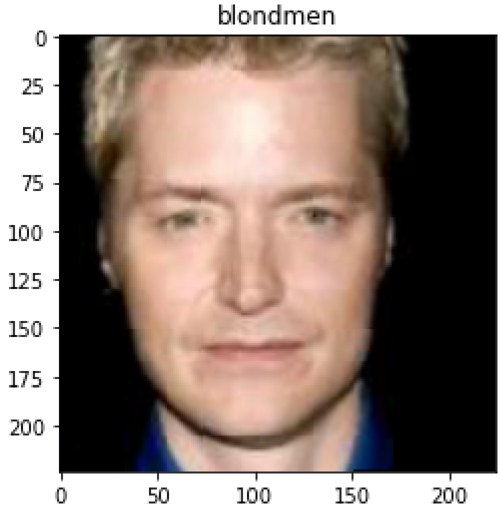

caucasianmen
0
25
50
75
100
125
150
175
200
0 50 100 150 200


blackwomen
0
25
50
75
100
125
150
175
200
0 50 100 150 200

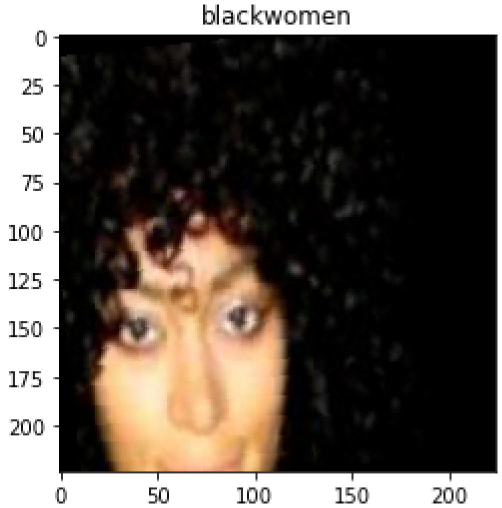

blondmen
0
25
50
75
100
125
150
175
200
0 50 100 150 200

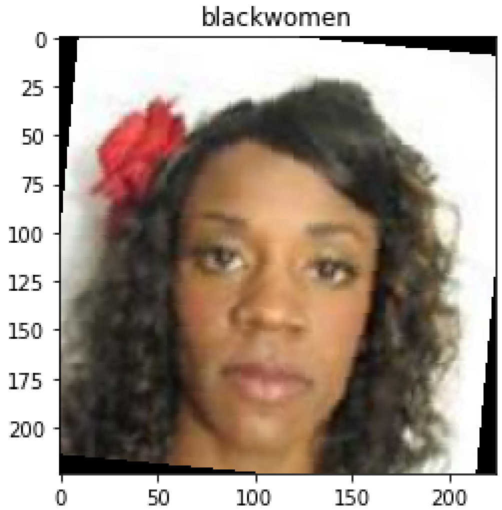

tannedwomen
0
25
50
75
100
125
150
175
200
0 50 100 150 200

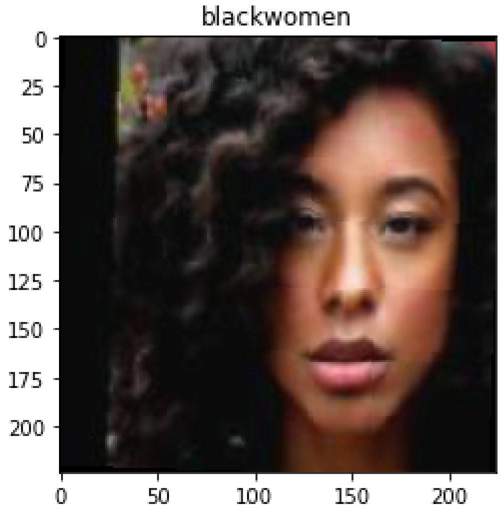

tannedmen
0
25
50
75
100
125
150
175
200
0 50 100 150 200


blondmen
0
25
50
75
100
125
150
175
200
0 50 100 150 200


blackwomen
0
25
50
75
100
125
150
175
200
0
50
100
150
200


blackwomen
0
25
50
75
100
125
150
175
200
0
50
100
150
200


blackwomen
0
25
50
75
100
125
150
175
200
0
50
100
150
200


blackwomen
0
25
50
75
100
125
150
175
200
0
50
100
150
200

In [23]:

```
import torch.nn as nn
import torch.nn.functional as F

# check if CUDA is available
use_cuda = torch.cuda.is_available()

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        self.conv1 = nn.Conv2d(3, 16, 3)
        self.conv2 = nn.Conv2d(16, 32, 3)
        self.conv3 = nn.Conv2d(32, 64, 3)
        self.conv4 = nn.Conv2d(64, 128, 3)
        self.conv5 = nn.Conv2d(128, 256, 3)

        self.fc1 = nn.Linear(256 * 6 * 6, 8)
        self.max_pool = nn.MaxPool2d(2, 2,ceil_mode=True)

        self.dropout = nn.Dropout(0.20)

        self.conv_bn1 = nn.BatchNorm2d(224,3)
        self.conv_bn2 = nn.BatchNorm2d(16)
        self.conv_bn3 = nn.BatchNorm2d(32)
        self.conv_bn4 = nn.BatchNorm2d(64)
        self.conv_bn5 = nn.BatchNorm2d(128)
        self.conv_bn6 = nn.BatchNorm2d(256)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.conv1(x))
        x = self.max_pool(x)
        x = self.conv_bn2(x)

        x = F.relu(self.conv2(x))
        x = self.max_pool(x)
        x = self.conv_bn3(x)

        x = F.relu(self.conv3(x))
        x = self.max_pool(x)
        x = self.conv_bn4(x)

        x = F.relu(self.conv4(x))
        x = self.max_pool(x)
        x = self.conv_bn5(x)

        x = F.relu(self.conv5(x))
        x = self.max_pool(x)
        x = self.conv_bn6(x)

        x = x.view(-1, 256 * 6 * 6)

        x = self.dropout(x)
        x = self.fc1(x)
        return x

#-#-# You do NOT have to modify the code below this line. #-#-#
```

```
# instantiate the CNN
model_scratch = Net()
print(model_scratch)
# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch.cuda()
```

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (conv5): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=9216, out_features=8, bias=True)
  (max_pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation
=1, ceil_mode=True)
  (dropout): Dropout(p=0.2, inplace=False)
  (conv_bn1): BatchNorm2d(224, eps=3, momentum=0.1, affine=True, tra
ck_running_stats=True)
  (conv_bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv_bn3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv_bn4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv_bn5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv_bn6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
)
```

I started out looking for best practices in architecting CNNs, stumbled upon this article [https://algorithmia.com/blog/convolutional-neural-nets-in-pytorch (https://algorithmia.com/blog/convolutional-neural-nets-in-pytorch)](https://algorithmia.com/blog/convolutional-neural-nets-in-pytorch)

After some further searching and combining different setups I decided to use 5 convolution layers with standard values for kernel (3), stride (1) and padding (0), maxpooling, batch normalization, dropout and ReLu in the forward function In the succeeding convolution layers I doubled the input/output ratio's and finally I concluded with a fully connected linear layer of 256 to 8 classes, our 'racial types' ['blackmen', 'blackwomen', 'blondmen', 'blondwomen', 'caucasianmen', 'caucasianwomen', 'tannedmen', 'tannedwomen']

In [24]:

```
import torch.optim as optim

### TODO: select loss function
criterion_scratch = nn.CrossEntropyLoss()

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.001)
```

In [34]:

```
# the following import is required for training to be robust to truncated images
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

    for epoch in range(1, n_epochs+1):
        # initialize variables to monitor training and validation loss
        train_loss = 0.0
        valid_loss = 0.0

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders[0]):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
                        # clear the gradients of all optimized variables
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the m
odel
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model
 parameters
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update training loss
            # update training loss
            train_loss += ((1 / (batch_idx + 1)) * (loss.data - train_loss))

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders[2]):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            with torch.no_grad():
                output = model(data)
            loss = criterion(output, target)
            valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.form
at(
            epoch,
            train_loss,
            valid_loss
```

```
            ))

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            print('Validation loss decreased ({:.6f} --> {:.6f}). Saving mode
l...'.format(valid_loss_min, valid_loss))
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss
    # return trained model
    return model


# train the model
model_scratch = train(30, loaders, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
Epoch: 1        Training Loss: 0.675595         Validation Loss: 1.1
76589
Validation loss decreased (inf --> 1.176589). Saving model...
Epoch: 2        Training Loss: 0.646563         Validation Loss: 1.1
41986
Validation loss decreased (1.176589 --> 1.141986). Saving model...
Epoch: 3        Training Loss: 0.586300         Validation Loss: 1.1
17281
Validation loss decreased (1.141986 --> 1.117281). Saving model...
Epoch: 4        Training Loss: 0.598796         Validation Loss: 1.2
19375
Epoch: 5        Training Loss: 0.578805         Validation Loss: 1.2
04194
Epoch: 6        Training Loss: 0.539694         Validation Loss: 1.1
20571
Epoch: 7        Training Loss: 0.504695         Validation Loss: 1.1
17761
Epoch: 8        Training Loss: 0.495732         Validation Loss: 1.1
36865
Epoch: 9        Training Loss: 0.486607         Validation Loss: 1.0
97667
Validation loss decreased (1.117281 --> 1.097667). Saving model...
Epoch: 10       Training Loss: 0.443001         Validation Loss: 1.0
54710
Validation loss decreased (1.097667 --> 1.054710). Saving model...
Epoch: 11       Training Loss: 0.421394         Validation Loss: 1.1
22078
Epoch: 12       Training Loss: 0.415748         Validation Loss: 1.1
39642
Epoch: 13       Training Loss: 0.398460         Validation Loss: 1.1
03329
Epoch: 14       Training Loss: 0.389481         Validation Loss: 1.1
20838
Epoch: 15       Training Loss: 0.361634         Validation Loss: 1.1
64706
Epoch: 16       Training Loss: 0.355377         Validation Loss: 1.0
83695
Epoch: 17       Training Loss: 0.346736         Validation Loss: 1.0
80034
Epoch: 18       Training Loss: 0.316895         Validation Loss: 1.1
28056
Epoch: 19       Training Loss: 0.312459         Validation Loss: 1.0
25440
Validation loss decreased (1.054710 --> 1.025440). Saving model...
Epoch: 20       Training Loss: 0.309769         Validation Loss: 1.1
04826
Epoch: 21       Training Loss: 0.292527         Validation Loss: 1.2
01774
Epoch: 22       Training Loss: 0.291411         Validation Loss: 1.0
23348
Validation loss decreased (1.025440 --> 1.023348). Saving model...
Epoch: 23       Training Loss: 0.254146         Validation Loss: 1.0
44065
Epoch: 24       Training Loss: 0.248141         Validation Loss: 1.0
47231
Epoch: 25       Training Loss: 0.253125         Validation Loss: 1.1
01015
Epoch: 26       Training Loss: 0.239855         Validation Loss: 1.3
68214
Epoch: 27       Training Loss: 0.245419         Validation Loss: 1.0
08670
```

```
Validation loss decreased (1.023348 --> 1.008670). Saving model...
Epoch: 28       Training Loss: 0.229802         Validation Loss: 1.0
34368
Epoch: 29       Training Loss: 0.212448         Validation Loss: 1.0
35769
Epoch: 30       Training Loss: 0.215868         Validation Loss: 1.0
27102
```

Out[34]:

```
<All keys matched successfully>
```

In [36]:

```
def test(loaders, model, criterion, use_cuda):

    # monitor test loss and accuracy
    test_loss = 0.
    correct = 0.
    total = 0.

    model.eval()
    for batch_idx, (data, target) in enumerate(loaders[1]):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss
))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().n
umpy())
        total += data.size(0)

    print('Test Loss: {:.6f}\n'.format(test_loss))

    print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
        100. * correct / total, correct, total))

# call test function
test(loaders, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 0.947616


Test Accuracy: 68% (686/1000)
```

Because I used data augmentation techniques on small subsets of the CelebA dataset, it seems that there's some overfitting issue, but the accuracy is not that bad. Now trying with a pretrained resnet50 model...

In [48]:

```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture

use_cuda = torch.cuda.is_available()

## TODO: Specify model architecture
model_transfer = models.resnet50(pretrained=True)

for param in model_transfer.parameters():
    param.requires_grad = False

model_transfer.fc = nn.Linear(2048, 8, bias=True)
fc_parameters = model_transfer.fc.parameters()

for param in fc_parameters:
    param.requires_grad = True

if use_cuda:
    model_transfer = model_transfer.cuda()
    print(model_transfer)
```

In [51]:

```
import torch.optim as optim
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.001)
```

In [50]:

```
# train the model

print(use_cuda)
n_epochs = 15

def train(n_epochs, loader, model, optimizer, criterion, use_cuda, save_path):
    valid_loss_min = np.Inf
    for epoch in range(1, n_epochs+1):
        # initialize variables to monitor training and validation loss
        train_loss = 0.0
        valid_loss = 0.0
        #train_loss = 0.0
        model.train()
        for batch_i, (data, target) in enumerate(loader[0]):
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            train_loss = train_loss + ((1 / (batch_i + 1)) * (loss.data - train_
loss))

            if batch_i % 100 == 0:
                print('Epoch %d, Batch %d loss: %.6f' %
                  (epoch, batch_i + 1, train_loss))

        model.eval()
        for batch_i, (data, target) in enumerate(loader[2]):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            output = model(data)
            loss = criterion(output, target)
            valid_loss = valid_loss + ((1 / (batch_i + 1)) * (loss.data - valid_
loss))


        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.form
at(
            epoch,
            train_loss,
            valid_loss
            ))

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            torch.save(model.state_dict(), save_path)
            print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model'
.format(
            valid_loss_min,
            valid_loss))
            valid_loss_min = valid_loss

    return model
```

```
# train the model
model_transfer = train(n_epochs, loaders, model_transfer, optimizer_transfer, cr
iterion_transfer, use_cuda, 'model_transfer.pt')

# load the model that got the best validation accuracy (uncomment the line belo
w)
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

```
False
Epoch 1, Batch 1 loss: 2.054286
Epoch 1, Batch 101 loss: 1.951132
Epoch: 1        Training Loss: 1.917379         Validation Loss: 1.8
71186
Validation loss decreased (inf --> 1.871186).  Saving model
Epoch 2, Batch 1 loss: 1.807552
Epoch 2, Batch 101 loss: 1.799908
Epoch: 2        Training Loss: 1.775935         Validation Loss: 1.7
17255
Validation loss decreased (1.871186 --> 1.717255).  Saving model
Epoch 3, Batch 1 loss: 1.875898
Epoch 3, Batch 101 loss: 1.681712
Epoch: 3        Training Loss: 1.661399         Validation Loss: 1.6
28498
Validation loss decreased (1.717255 --> 1.628498).  Saving model
Epoch 4, Batch 1 loss: 1.751378
Epoch 4, Batch 101 loss: 1.586395
Epoch: 4        Training Loss: 1.572986         Validation Loss: 1.5
46133
Validation loss decreased (1.628498 --> 1.546133).  Saving model
Epoch 5, Batch 1 loss: 1.488634
Epoch 5, Batch 101 loss: 1.532543
Epoch: 5        Training Loss: 1.505317         Validation Loss: 1.4
71715
Validation loss decreased (1.546133 --> 1.471715).  Saving model
Epoch 6, Batch 1 loss: 1.292140
Epoch 6, Batch 101 loss: 1.428732
Epoch: 6        Training Loss: 1.434633         Validation Loss: 1.4
13794
Validation loss decreased (1.471715 --> 1.413794).  Saving model
Epoch 7, Batch 1 loss: 1.635703
Epoch 7, Batch 101 loss: 1.385589
Epoch: 7        Training Loss: 1.381688         Validation Loss: 1.3
62862
Validation loss decreased (1.413794 --> 1.362862).  Saving model
Epoch 8, Batch 1 loss: 1.719517
Epoch 8, Batch 101 loss: 1.359608
Epoch: 8        Training Loss: 1.345917         Validation Loss: 1.3
18248
Validation loss decreased (1.362862 --> 1.318248).  Saving model
Epoch 9, Batch 1 loss: 1.368541
Epoch 9, Batch 101 loss: 1.324173
Epoch: 9        Training Loss: 1.307731         Validation Loss: 1.2
99771
Validation loss decreased (1.318248 --> 1.299771).  Saving model
Epoch 10, Batch 1 loss: 1.477446
Epoch 10, Batch 101 loss: 1.281109
Epoch: 10       Training Loss: 1.271512         Validation Loss: 1.2
56415
Validation loss decreased (1.299771 --> 1.256415).  Saving model
Epoch 11, Batch 1 loss: 1.100479
Epoch 11, Batch 101 loss: 1.270789
Epoch: 11       Training Loss: 1.251881         Validation Loss: 1.2
14532
Validation loss decreased (1.256415 --> 1.214532).  Saving model
Epoch 12, Batch 1 loss: 1.059780
Epoch 12, Batch 101 loss: 1.205252
Epoch: 12       Training Loss: 1.213547         Validation Loss: 1.1
96292
Validation loss decreased (1.214532 --> 1.196292).  Saving model
```

```
Epoch 13, Batch 1 loss: 0.979585
Epoch 13, Batch 101 loss: 1.183514
Epoch: 13       Training Loss: 1.180812         Validation Loss: 1.1
76141
Validation loss decreased (1.196292 --> 1.176141).  Saving model
Epoch 14, Batch 1 loss: 1.406038
Epoch 14, Batch 101 loss: 1.161470
Epoch: 14       Training Loss: 1.166821         Validation Loss: 1.1
70026
Validation loss decreased (1.176141 --> 1.170026).  Saving model
Epoch 15, Batch 1 loss: 1.319971
Epoch 15, Batch 101 loss: 1.127637
Epoch: 15       Training Loss: 1.139749         Validation Loss: 1.1
54948
Validation loss decreased (1.170026 --> 1.154948).  Saving model
```

Out[50]:

```
<All keys matched successfully>
```

In [52]:

```
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

Out[52]:

```
<All keys matched successfully>
```

In [53]:

```
test(loaders, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.113536


Test Accuracy: 66% (662/1000)
```

So, nothing gained as this point. Seems the pretrained model had to learn all over again.