Denoising Autoencoders And Where To Find Them

Today we're going to train deep autoencoders and apply them to faces and similar images search.

Our new test subjects are human faces from the lfw dataset (http://vis-www.cs.umass.edu/lfw/).

Import stuff

In [1]:

```
import sys
sys.path.append("..")
import grading
```

In [2]:

```
import tensorflow as tf
import keras, keras.layers as L, keras.backend as K
import numpy as np
from sklearn.model_selection import train_test_split
from lfw_dataset import load_lfw_dataset
%matplotlib inline
import matplotlib.pyplot as plt
import download_utils
import keras_utils
import numpy as np
from keras_utils import reset_tf_session
```

```
Using TensorFlow backend.
```

In [3]:

```
# !!! remember to clear session/graph if you rebuild your graph to avoid out-of-memory
 errors !!!
```

Load dataset

Dataset was downloaded for you. Relevant links (just in case):

- http://www.cs.columbia.edu/CAVE/databases/pubfig/download/lfw_attributes.txt (http://www.cs.columbia.edu/CAVE/databases/pubfig/download/lfw_attributes.txt)
- http://vis-www.cs.umass.edu/lfw/lfw-deepfunneled.tgz (http://vis-www.cs.umass.edu/lfw/lfw-deepfunneled.tgz)
- http://vis-www.cs.umass.edu/lfw/lfw.tgz (http://vis-www.cs.umass.edu/lfw/lfw.tgz)

In [4]:

```
# we downloaded them for you, just link them here
download_utils.link_week_4_resources()
```

In [5]:

```
# load images
X, attr = load_lfw_dataset(use_raw=True, dimx=32, dimy=32)
IMG_SHAPE = X.shape[1:]

# center images
X = X.astype('float32') / 255.0 - 0.5

# split
X_train, X_test = train_test_split(X, test_size=0.1, random_state=42)
```

In [6]:

```
def show_image(x):
    plt.imshow(np.clip(x + 0.5, 0, 1))
```

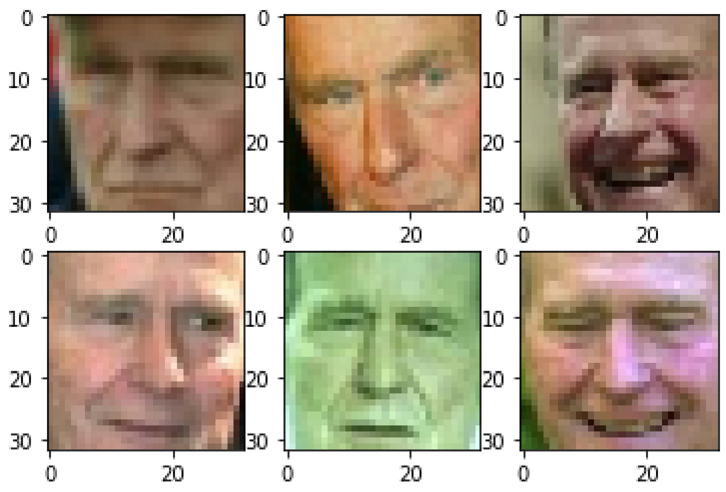
In [7]:

```
plt.title('sample images')

for i in range(6):
    plt.subplot(2,3,i+1)
    show_image(X[i])

print("X shape:", X.shape)
print("attr shape:", attr.shape)

# try to free memory
del X
import gc
gc.collect()
```

```
X shape: (13143, 32, 32, 3)
attr shape: (13143, 73)
```

Out[7]:

```
644
```



Autoencoder architecture

Let's design autoencoder as two sequential keras models: the encoder and decoder respectively.

We will then use symbolic API to apply and train these models.

First step: PCA

Principial Component Analysis is a popular dimensionality reduction method.

Under the hood, PCA attempts to decompose object-feature matrix X into two smaller matrices: W and $\hat{W}$ minimizing *mean squared error*:

$$\|(XW)\hat{W} - X\|_2^2 \rightarrow_{W,\hat{W}} \min$$

- $X \in \mathbb{R}^{n \times m}$ - object matrix (**centered**);
- $W \in \mathbb{R}^{m \times d}$ - matrix of direct transformation;
- $\hat{W} \in \mathbb{R}^{d \times m}$ - matrix of reverse transformation;
- n samples, m original dimensions and d target dimensions;

In geometric terms, we want to find d axes along which most of variance occurs. The "natural" axes, if you wish.

PCA can also be seen as a special case of an autoencoder.

- **Encoder**: X -> Dense(d units) -> code
- **Decoder**: code -> Dense(m units) -> X

Where Dense is a fully-connected layer with linear activaton: $f(X) = W \cdot X + \vec{b}$

Note: the bias term in those layers is responsible for "centering" the matrix i.e. substracting mean.

In [8]:

```python
def build_pca_autoencoder(img_shape, code_size):
    """
    Here we define a simple linear autoencoder as described above.
    We also flatten and un-flatten data to be compatible with image shapes
    """

    encoder = keras.models.Sequential()
    encoder.add(L.InputLayer(img_shape))
    encoder.add(L.Flatten())                  #flatten image to vector
    encoder.add(L.Dense(code_size))           #actual encoder

    decoder = keras.models.Sequential()
    decoder.add(L.InputLayer((code_size,)))
    decoder.add(L.Dense(np.prod(img_shape)))  #actual decoder, height*width*3 units
    decoder.add(L.Reshape(img_shape))         #un-flatten

    return encoder,decoder
```

Meld them together into one model:

In [9]:

```
s = reset_tf_session()

encoder, decoder = build_pca_autoencoder(IMG_SHAPE, code_size=32)

inp = L.Input(IMG_SHAPE)
code = encoder(inp)
reconstruction = decoder(code)

autoencoder = keras.models.Model(inputs=inp, outputs=reconstruction)
autoencoder.compile(optimizer='adamax', loss='mse')

autoencoder.fit(x=X_train, y=X_train, epochs=15,
                validation_data=[X_test, X_test],
                callbacks=[keras_utils.TqdmProgressCallback()],
                verbose=0)
```

```
Epoch 1/15

Epoch 2/15

Epoch 3/15

Epoch 4/15

Epoch 5/15

Epoch 6/15

Epoch 7/15

Epoch 8/15

Epoch 9/15

Epoch 10/15

Epoch 11/15

Epoch 12/15

Epoch 13/15

Epoch 14/15

Epoch 15/15
```

```
Out[9]:
<keras.callbacks.History at 0x7f7ad1369278>
```

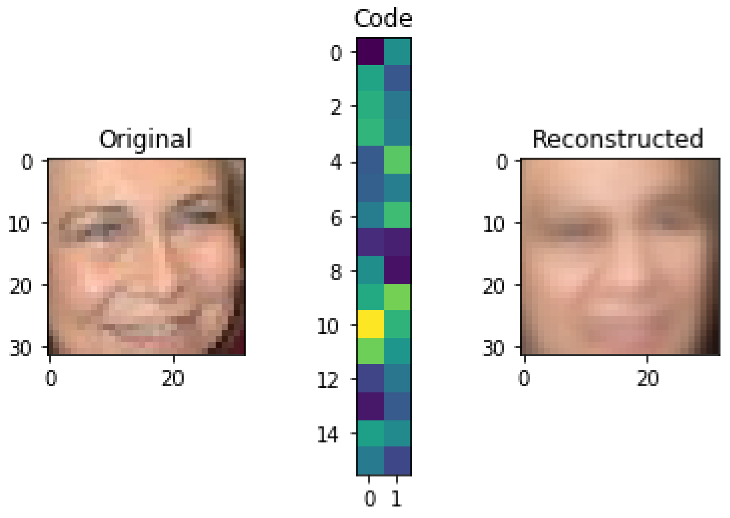
In [10]:

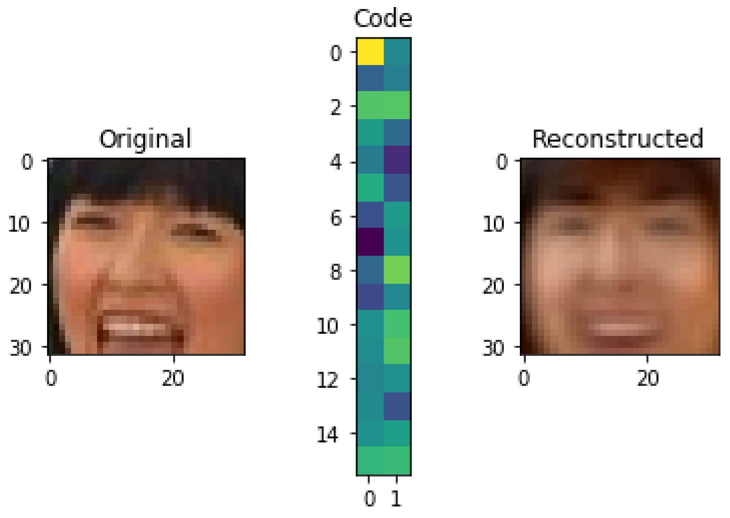
```
def visualize(img,encoder,decoder):
    """Draws original, encoded and decoded images"""
    code = encoder.predict(img[None])[0]  # img[None] is the same as img[np.newaxis, :]
    reco = decoder.predict(code[None])[0]

    plt.subplot(1,3,1)
    plt.title("Original")
    show_image(img)

    plt.subplot(1,3,2)
    plt.title("Code")
    plt.imshow(code.reshape([code.shape[-1]//2,-1]))

    plt.subplot(1,3,3)
    plt.title("Reconstructed")
    show_image(reco)
    plt.show()
```

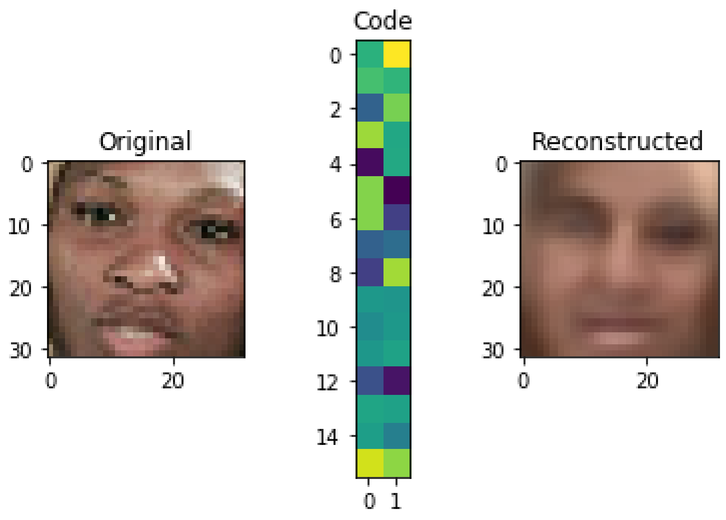
In [11]:

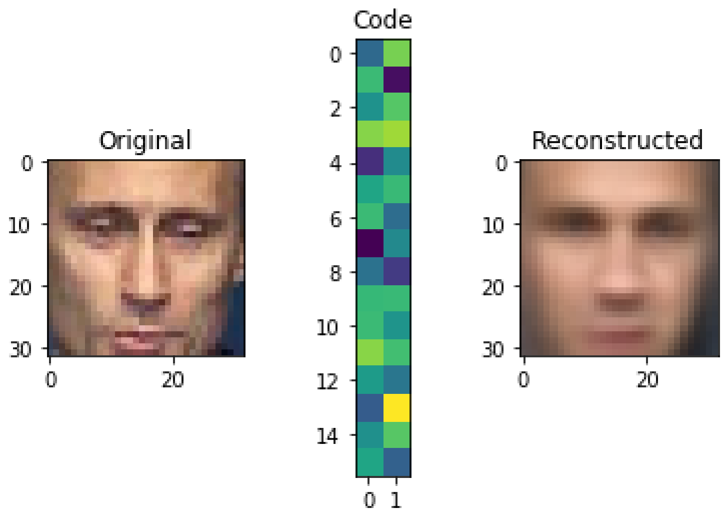
```
score = autoencoder.evaluate(X_test,X_test,verbose=0)
print("PCA MSE:", score)

for i in range(5):
    img = X_test[i]
    visualize(img,encoder,decoder)
```

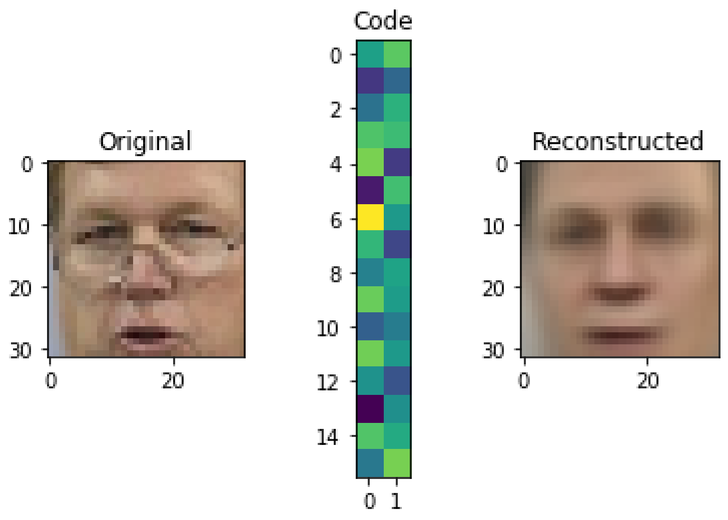
```
PCA MSE: 0.00660536107094
```






Original
Code
Reconstructed


Original
Code
Reconstructed


Original
Code
Reconstructed

Going deeper: convolutional autoencoder

PCA is neat but surely we can do better. This time we want you to build a deep convolutional autoencoder by... stacking more layers.

Encoder

The **encoder** part is pretty standard, we stack convolutional and pooling layers and finish with a dense layer to get the representation of desirable size (`code_size`).

We recommend to use `activation='elu'` for all convolutional and dense layers.

We recommend to repeat (conv, pool) 4 times with kernel size (3, 3), `padding='same'` and the following numbers of output channels: `32, 64, 128, 256`.

Remember to flatten (`L.Flatten()`) output before adding the last dense layer!

Decoder

For **decoder** we will use so-called "transpose convolution".

Traditional convolutional layer takes a patch of an image and produces a number (patch -> number). In "transpose convolution" we want to take a number and produce a patch of an image (number -> patch). We need this layer to "undo" convolutions in encoder. We had a glimpse of it during week 3 (watch this video (https://www.coursera.org/learn/intro-to-deep-learning/lecture/auRqf/a-glimpse-of-other-computer-vision-tasks) starting at 5:41).

Here's how "transpose convolution" works:

In this example we use a stride of 2 to produce 4x4 output, this way we "undo" pooling as well. Another way to think about it: we "undo" convolution with stride 2 (which is similar to conv + pool).

You can add "transpose convolution" layer in Keras like this:

```
L.Conv2DTranspose(filters=?, kernel_size=(3, 3), strides=2, activation='elu', pa
dding='same')
```

Our decoder starts with a dense layer to "undo" the last layer of encoder. Remember to reshape its output to "undo" `L.Flatten()` in encoder.

Now we're ready to undo (conv, pool) pairs. For this we need to stack 4 `L.Conv2DTranspose` layers with the following numbers of output channels: `128, 64, 32, 3`. Each of these layers will learn to "undo" (conv, pool) pair in encoder. For the last `L.Conv2DTranspose` layer use `activation=None` because that is our final image.

In [12]:

```
# Let's play around with transpose convolution on examples first
def test_conv2d_transpose(img_size, filter_size):
    print("Transpose convolution test for img_size={}, filter_size={}:".format(img_size
, filter_size))

    x = (np.arange(img_size ** 2, dtype=np.float32) + 1).reshape((1, img_size, img_size
, 1))
    f = (np.ones(filter_size ** 2, dtype=np.float32)).reshape((filter_size, filter_size
, 1, 1))

    s = reset_tf_session()

    conv = tf.nn.conv2d_transpose(x, f,
                                  output_shape=(1, img_size * 2, img_size * 2, 1),
                                  strides=[1, 2, 2, 1],
                                  padding='SAME')

    result = s.run(conv)
    print("input:")
    print(x[0, :, :, 0])
    print("filter:")
    print(f[:, :, 0, 0])
    print("output:")
    print(result[0, :, :, 0])
    s.close()

test_conv2d_transpose(img_size=2, filter_size=2)
test_conv2d_transpose(img_size=2, filter_size=3)
test_conv2d_transpose(img_size=4, filter_size=2)
test_conv2d_transpose(img_size=4, filter_size=3)
```

```
Transpose convolution test for img_size=2, filter_size=2:
input:
[[ 1.  2.]
 [ 3.  4.]]
filter:
[[ 1.  1.]
 [ 1.  1.]]
output:
[[ 1.  1.  2.  2.]
 [ 1.  1.  2.  2.]
 [ 3.  3.  4.  4.]
 [ 3.  3.  4.  4.]]
Transpose convolution test for img_size=2, filter_size=3:
input:
[[ 1.  2.]
 [ 3.  4.]]
filter:
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
output:
[[  1.   1.   3.   2.]
 [  1.   1.   3.   2.]
 [  4.   4.  10.   6.]
 [  3.   3.   7.   4.]]
Transpose convolution test for img_size=4, filter_size=2:
input:
[[  1.   2.   3.   4.]
 [  5.   6.   7.   8.]
 [  9.  10.  11.  12.]
 [ 13.  14.  15.  16.]]
filter:
[[ 1.  1.]
 [ 1.  1.]]
output:
[[  1.   1.   2.   2.   3.   3.   4.   4.]
 [  1.   1.   2.   2.   3.   3.   4.   4.]
 [  5.   5.   6.   6.   7.   7.   8.   8.]
 [  5.   5.   6.   6.   7.   7.   8.   8.]
 [  9.   9.  10.  10.  11.  11.  12.  12.]
 [  9.   9.  10.  10.  11.  11.  12.  12.]
 [ 13.  13.  14.  14.  15.  15.  16.  16.]
 [ 13.  13.  14.  14.  15.  15.  16.  16.]]
Transpose convolution test for img_size=4, filter_size=3:
input:
[[  1.   2.   3.   4.]
 [  5.   6.   7.   8.]
 [  9.  10.  11.  12.]
 [ 13.  14.  15.  16.]]
filter:
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
output:
[[  1.   1.   3.   2.   5.   3.   7.   4.]
 [  1.   1.   3.   2.   5.   3.   7.   4.]
 [  6.   6.  14.   8.  18.  10.  22.  12.]
 [  5.   5.  11.   6.  13.   7.  15.   8.]
 [ 14.  14.  30.  16.  34.  18.  38.  20.]
 [  9.   9.  19.  10.  21.  11.  23.  12.]
 [ 22.  22.  46.  24.  50.  26.  54.  28.]
 [ 13.  13.  27.  14.  29.  15.  31.  16.]]
```

In [13]:

```
IMG_SHAPE
```

Out[13]:

```
(32, 32, 3)
```

In [16]:

```
def build_deep_autoencoder(img_shape, code_size):
    """PCA's deeper brother. See instructions above. Use `code_size` in layer definitio
ns."""
    H,W,C = img_shape

    # encoder
    encoder = keras.models.Sequential()
    ### YOUR CODE HERE: define encoder as per instructions above ###
    encoder.add(L.InputLayer(img_shape))
    encoder.add(L.Conv2D(filters = 32, kernel_size = (3, 3), activation = 'elu', paddin
g = 'same'))
    encoder.add(L.MaxPooling2D(pool_size = (2, 2)))
    encoder.add(L.Conv2D(filters = 64, kernel_size = (3, 3), activation = 'elu', paddin
g = 'same'))
    encoder.add(L.MaxPooling2D(pool_size = (2, 2)))
    encoder.add(L.Conv2D(filters = 128, kernel_size = (3, 3), activation = 'elu', paddi
ng = 'same'))
    encoder.add(L.MaxPooling2D(pool_size = (2, 2)))
    encoder.add(L.Conv2D(filters = 256, kernel_size = (3, 3), activation = 'elu', paddi
ng = 'same'))
    encoder.add(L.MaxPooling2D(pool_size = (2, 2)))
    encoder.add(L.Flatten())
    encoder.add(L.Dense(units = code_size))

    # decoder
    decoder = keras.models.Sequential()
    ### YOUR CODE HERE: define decoder as per instructions above ###
    decoder.add(L.InputLayer(input_shape = (code_size,)))
    decoder.add(L.Dense(units = 2*2*256))   #final dimension of encoder
    decoder.add(L.Reshape((2, 2, 256)))     #generate 3-D input to decoder
    decoder.add(L.Conv2DTranspose(filters = 128, kernel_size = (3, 3), activation = 'el
u',
                                  strides = 2, padding = 'same'))
    decoder.add(L.Conv2DTranspose(filters = 64, kernel_size = (3, 3), activation = 'el
u',
                                  strides = 2, padding = 'same'))
    decoder.add(L.Conv2DTranspose(filters = 32, kernel_size = (3, 3), activation = 'el
u',
                                  strides = 2, padding = 'same'))
    decoder.add(L.Conv2DTranspose(filters = 3, kernel_size = (3, 3), activation = None,
                                  strides = 2, padding = 'same'))

    return encoder, decoder
```

In [17]:

```
# Check autoencoder shapes along different code_sizes
get_dim = lambda layer: np.prod(layer.output_shape[1:])
for code_size in [1,8,32,128,512]:
    s = reset_tf_session()
    encoder, decoder = build_deep_autoencoder(IMG_SHAPE, code_size=code_size)
    print("Testing code size %i" % code_size)
    assert encoder.output_shape[1:]==(code_size,), "encoder must output a code of requi
red size"
    assert decoder.output_shape[1:]==IMG_SHAPE,    "decoder must output an image of val
id shape"
    assert len(encoder.trainable_weights)>=6,      "encoder must contain at least 3 lay
ers"
    assert len(decoder.trainable_weights)>=6,      "decoder must contain at least 3 lay
ers"

    for layer in encoder.layers + decoder.layers:
        assert get_dim(layer) >= code_size, "Encoder layer %s is smaller than bottlenec
k (%i units)"%(layer.name,get_dim(layer))

print("All tests passed!")
s = reset_tf_session()
```

```
Testing code size 1
Testing code size 8
Testing code size 32
Testing code size 128
Testing code size 512
All tests passed!
```

In [18]:

```
# Look at encoder and decoder shapes.
# Total number of trainable parameters of encoder and decoder should be close.
s = reset_tf_session()
encoder, decoder = build_deep_autoencoder(IMG_SHAPE, code_size=32)
encoder.summary()
decoder.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         (None, 32, 32, 3)         0         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 32)        896       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 64)        18496     
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 8, 8, 64)          0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 128)         73856     
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 128)         0         
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 4, 4, 256)         295168    
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 2, 2, 256)         0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0         
_________________________________________________________________
dense_1 (Dense)              (None, 32)                32800     
=================================================================
Total params: 421,216
Trainable params: 421,216
Non-trainable params: 0
_________________________________________________________________
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_2 (InputLayer)         (None, 32)                0         
_________________________________________________________________
dense_2 (Dense)              (None, 1024)              33792     
_________________________________________________________________
reshape_1 (Reshape)          (None, 2, 2, 256)         0         
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 4, 4, 128)         295040    
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 8, 8, 64)          73792     
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 16, 16, 32)        18464     
_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 32, 32, 3)         867       
=================================================================
Total params: 421,955
Trainable params: 421,955
Non-trainable params: 0
_________________________________________________________________
```

Convolutional autoencoder training. This will take **1 hour**. You're aiming at ~0.0056 validation MSE and ~0.0054 training MSE.

In [19]:

```
s = reset_tf_session()

encoder, decoder = build_deep_autoencoder(IMG_SHAPE, code_size=32)

inp = L.Input(IMG_SHAPE)
code = encoder(inp)
reconstruction = decoder(code)

autoencoder = keras.models.Model(inputs=inp, outputs=reconstruction)
autoencoder.compile(optimizer="adamax", loss='mse')
```

In [20]:

```
# we will save model checkpoints here to continue training in case of kernel death
model_filename = 'autoencoder.{0:03d}.hdf5'
last_finished_epoch = None

#### uncomment below to continue training from model checkpoint
#### fill `last_finished_epoch` with your latest finished epoch
# from keras.models import load_model
# s = reset_tf_session()
# last_finished_epoch = 4
# autoencoder = load_model(model_filename.format(last_finished_epoch))
# encoder = autoencoder.layers[1]
# decoder = autoencoder.layers[2]
```

In [21]:

```
autoencoder.fit(x=X_train, y=X_train, epochs=25,
                validation_data=[X_test, X_test],
                callbacks=[keras_utils.ModelSaveCallback(model_filename),
                           keras_utils.TqdmProgressCallback()],
                verbose=0,
                initial_epoch=last_finished_epoch or 0)
```

```
Epoch 1/25

Model saved in autoencoder.000.hdf5


Epoch 2/25

Model saved in autoencoder.001.hdf5


Epoch 3/25

Model saved in autoencoder.002.hdf5


Epoch 4/25

Model saved in autoencoder.003.hdf5


Epoch 5/25

Model saved in autoencoder.004.hdf5


Epoch 6/25

Model saved in autoencoder.005.hdf5


Epoch 7/25

Model saved in autoencoder.006.hdf5


Epoch 8/25

Model saved in autoencoder.007.hdf5


Epoch 9/25

Model saved in autoencoder.008.hdf5


Epoch 10/25

Model saved in autoencoder.009.hdf5


Epoch 11/25

Model saved in autoencoder.010.hdf5


Epoch 12/25
```

```
Model saved in autoencoder.011.hdf5

Epoch 13/25

Model saved in autoencoder.012.hdf5

Epoch 14/25

Model saved in autoencoder.013.hdf5

Epoch 15/25

Model saved in autoencoder.014.hdf5

Epoch 16/25

Model saved in autoencoder.015.hdf5

Epoch 17/25

Model saved in autoencoder.016.hdf5

Epoch 18/25

Model saved in autoencoder.017.hdf5

Epoch 19/25

Model saved in autoencoder.018.hdf5

Epoch 20/25

Model saved in autoencoder.019.hdf5

Epoch 21/25

Model saved in autoencoder.020.hdf5

Epoch 22/25

Model saved in autoencoder.021.hdf5

Epoch 23/25
```

```
Model saved in autoencoder.022.hdf5


Epoch 24/25

Model saved in autoencoder.023.hdf5


Epoch 25/25

Model saved in autoencoder.024.hdf5
```

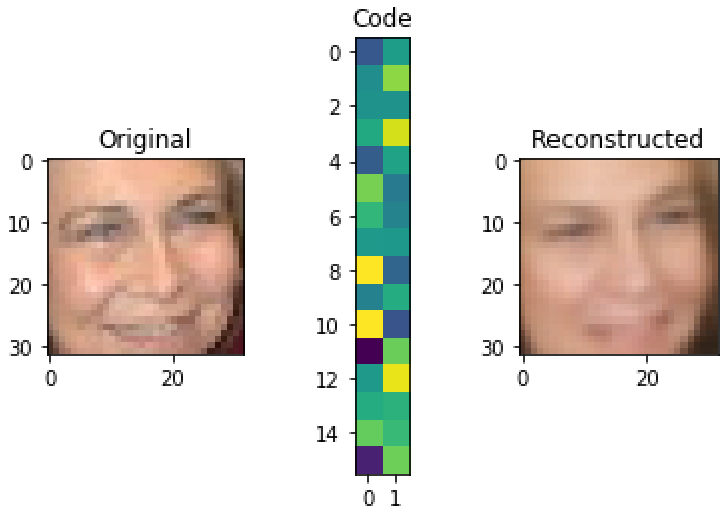
Out[21]:

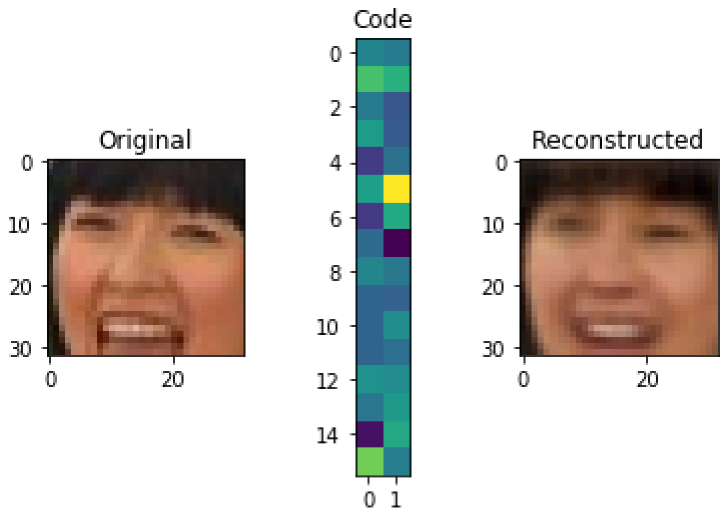
```
<keras.callbacks.History at 0x7f7ad2165a20>
```

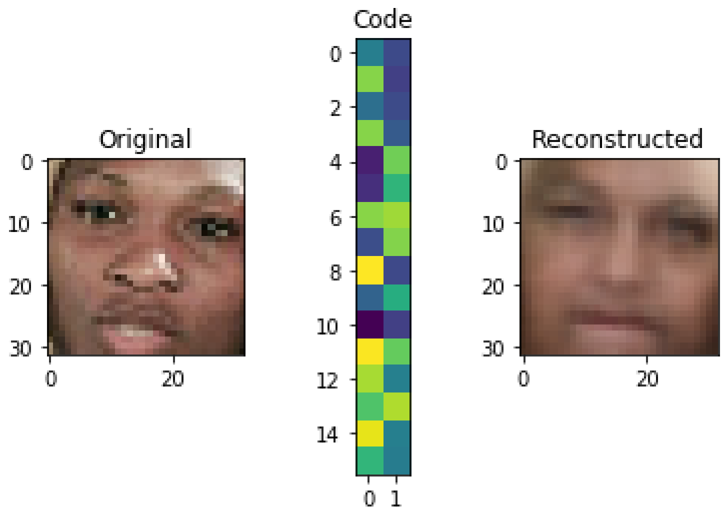
In [22]:

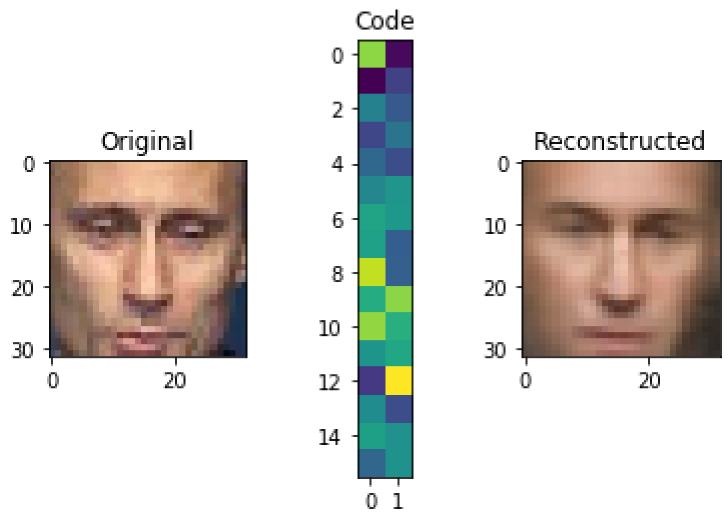
```
reconstruction_mse = autoencoder.evaluate(X_test, X_test, verbose=0)
print("Convolutional autoencoder MSE:", reconstruction_mse)
for i in range(5):
    img = X_test[i]
    visualize(img,encoder,decoder)
```

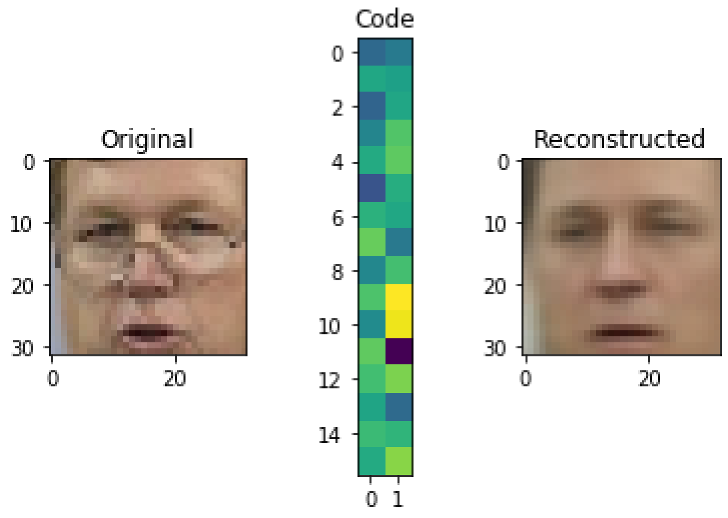
```
Convolutional autoencoder MSE: 0.00552901700582
```











In [23]:

```
# save trained weights
encoder.save_weights("encoder.h5")
decoder.save_weights("decoder.h5")
```

In [25]:

```
# restore trained weights
s = reset_tf_session()

encoder, decoder = build_deep_autoencoder(IMG_SHAPE, code_size=32)
encoder.load_weights("encoder.h5")
decoder.load_weights("decoder.h5")

inp = L.Input(IMG_SHAPE)
code = encoder(inp)
reconstruction = decoder(code)

autoencoder = keras.models.Model(inputs=inp, outputs=reconstruction)
autoencoder.compile(optimizer="adamax", loss='mse')

print(autoencoder.evaluate(X_test, X_test, verbose=0))
print(reconstruction_mse)
```

```
0.00552901700582
0.00552901700582
```

Submit to Coursera

In [26]:

```
from submit import submit_autoencoder
submission = build_deep_autoencoder(IMG_SHAPE, code_size=71)

# token expires every 30 min
COURSERA_TOKEN = 'vUB3fPOBWIE1m6Yj'           ### YOUR TOKEN HERE
COURSERA_EMAIL = 'knowtech94@gmail.com'       ### YOUR EMAIL HERE

submit_autoencoder(submission, reconstruction_mse, COURSERA_EMAIL, COURSERA_TOKEN)
```

```
Submitted to Coursera platform. See results on assignment page!
```

Optional: Denoising Autoencoder

This part is **optional**, it shows you one useful application of autoencoders: denoising. You can run this code and make sure denoising works :)

Let's now turn our model into a denoising autoencoder:

We'll keep the model architecture, but change the way it is trained. In particular, we'll corrupt its input data randomly with noise before each epoch.

There are many strategies to introduce noise: adding gaussian white noise, occluding with random black rectangles, etc. We will add gaussian white noise.

In [28]:

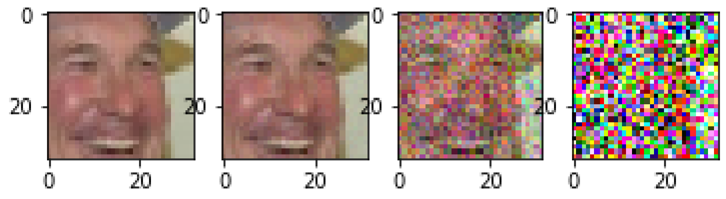
```
def apply_gaussian_noise(X,sigma=0.1):
    """
    adds noise from standard normal distribution with standard deviation sigma
    :param X: image tensor of shape [batch,height,width,3]
    Returns X + noise.
    """
    #noise = ### YOUR CODE HERE ###
    noise = np.random.normal(0, sigma, X.shape)
    return X + noise
```

In [29]:

```
# noise tests
theoretical_std = (X_train[:100].std()**2 + 0.5**2)**.5
our_std = apply_gaussian_noise(X_train[:100],sigma=0.5).std()
assert abs(theoretical_std - our_std) < 0.01, "Standard deviation does not match it's r
equired value. Make sure you use sigma as std."
assert abs(apply_gaussian_noise(X_train[:100],sigma=0.5).mean() - X_train[:100].mean())
< 0.01, "Mean has changed. Please add zero-mean noise"
```

In [30]:

```
# test different noise scales
plt.subplot(1,4,1)
show_image(X_train[0])
plt.subplot(1,4,2)
show_image(apply_gaussian_noise(X_train[:1],sigma=0.01)[0])
plt.subplot(1,4,3)
show_image(apply_gaussian_noise(X_train[:1],sigma=0.1)[0])
plt.subplot(1,4,4)
show_image(apply_gaussian_noise(X_train[:1],sigma=0.5)[0])
```



Training will take **1 hour**.

In [31]:

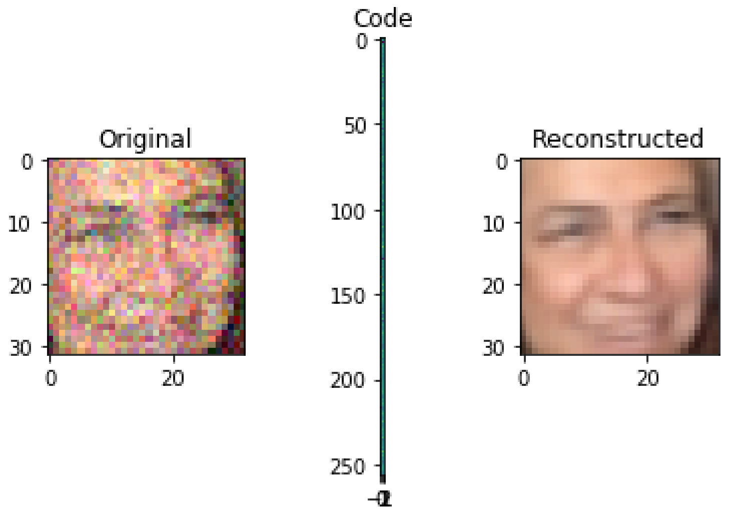
```
s = reset_tf_session()

# we use bigger code size here for better quality
encoder, decoder = build_deep_autoencoder(IMG_SHAPE, code_size=512)
assert encoder.output_shape[1:]==(512,), "encoder must output a code of required size"

inp = L.Input(IMG_SHAPE)
code = encoder(inp)
reconstruction = decoder(code)

autoencoder = keras.models.Model(inp, reconstruction)
autoencoder.compile('adamax', 'mse')

for i in range(25):
    print("Epoch %i/25, Generating corrupted samples..."%(i+1))
    X_train_noise = apply_gaussian_noise(X_train)
    X_test_noise = apply_gaussian_noise(X_test)

    # we continue to train our model with new noise-augmented data
    autoencoder.fit(x=X_train_noise, y=X_train, epochs=1,
                    validation_data=[X_test_noise, X_test],
                    callbacks=[keras_utils.TqdmProgressCallback()],
                    verbose=0)
```

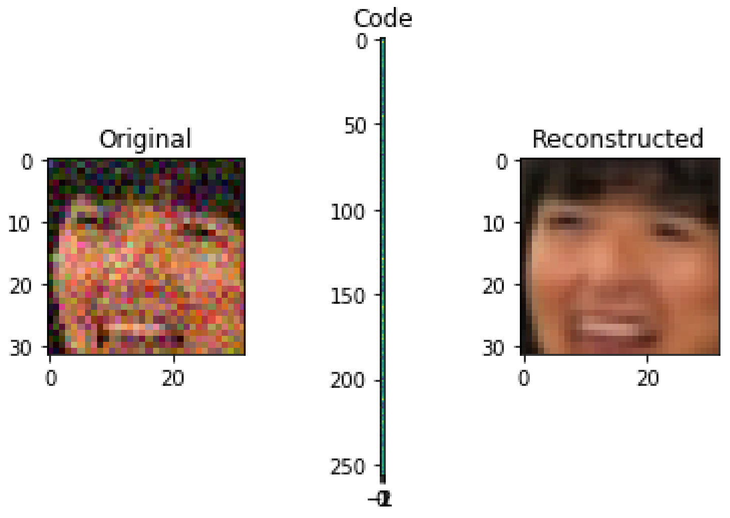
```
Epoch 1/25, Generating corrupted samples...

Epoch 1/1

Epoch 2/25, Generating corrupted samples...

Epoch 1/1

Epoch 3/25, Generating corrupted samples...

Epoch 1/1

Epoch 4/25, Generating corrupted samples...

Epoch 1/1

Epoch 5/25, Generating corrupted samples...

Epoch 1/1

Epoch 6/25, Generating corrupted samples...

Epoch 1/1

Epoch 7/25, Generating corrupted samples...

Epoch 1/1

Epoch 8/25, Generating corrupted samples...

Epoch 1/1

Epoch 9/25, Generating corrupted samples...

Epoch 1/1

Epoch 10/25, Generating corrupted samples...

Epoch 1/1

Epoch 11/25, Generating corrupted samples...

Epoch 1/1

Epoch 12/25, Generating corrupted samples...

Epoch 1/1

Epoch 13/25, Generating corrupted samples...

Epoch 1/1

Epoch 14/25, Generating corrupted samples...

Epoch 1/1
Epoch 15/25, Generating corrupted samples...

Epoch 1/1

Epoch 16/25, Generating corrupted samples...

Epoch 1/1

Epoch 17/25, Generating corrupted samples...

Epoch 1/1

Epoch 18/25, Generating corrupted samples...

Epoch 1/1

Epoch 19/25, Generating corrupted samples...

Epoch 1/1

Epoch 20/25, Generating corrupted samples...

Epoch 1/1

Epoch 21/25, Generating corrupted samples...

Epoch 1/1

Epoch 22/25, Generating corrupted samples...

Epoch 1/1

Epoch 23/25, Generating corrupted samples...

Epoch 1/1

Epoch 24/25, Generating corrupted samples...

Epoch 1/1

Epoch 25/25, Generating corrupted samples...

Epoch 1/1
```

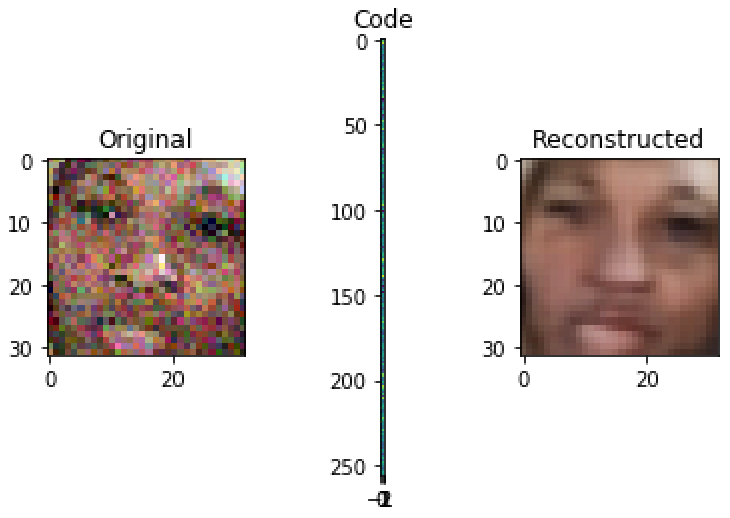
In [32]:

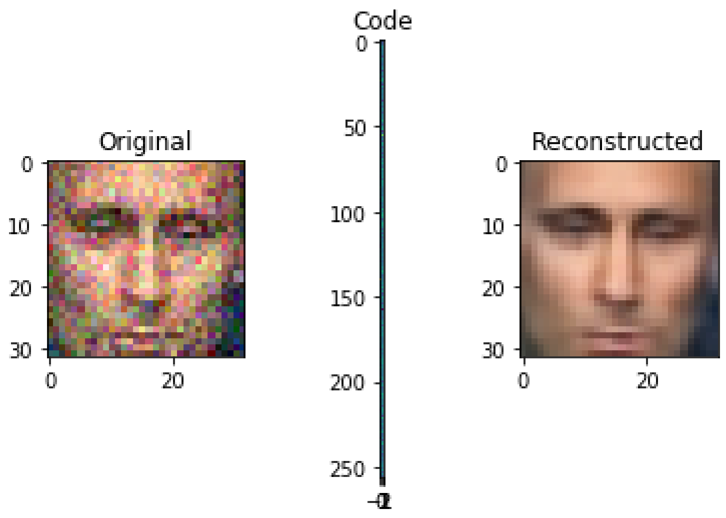
```
X_test_noise = apply_gaussian_noise(X_test)
denoising_mse = autoencoder.evaluate(X_test_noise, X_test, verbose=0)
print("Denoising MSE:", denoising_mse)
for i in range(5):
    img = X_test_noise[i]
    visualize(img,encoder,decoder)
```

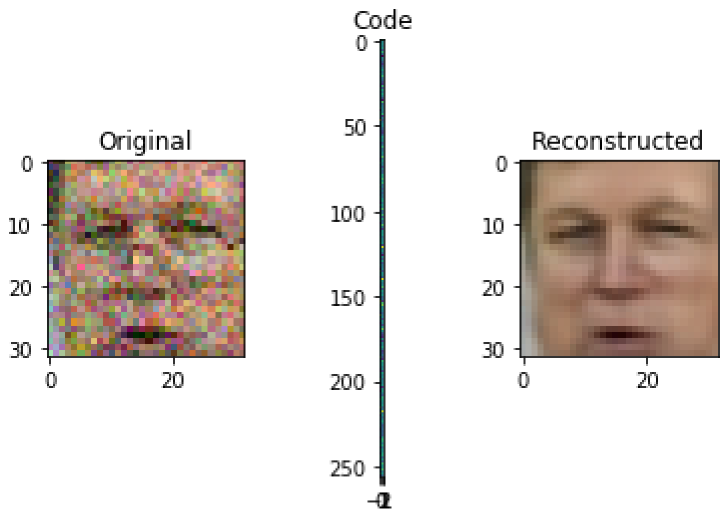
```
Denoising MSE: 0.00284946297115
```











Optional: Image retrieval with autoencoders

So we've just trained a network that converts image into itself imperfectly. This task is not that useful in and of itself, but it has a number of awesome side-effects. Let's see them in action.

First thing we can do is image retrieval aka image search. We will give it an image and find similar images in latent space:

To speed up retrieval process, one should use Locality Sensitive Hashing on top of encoded vectors. This technique (https://erikbern.com/2015/07/04/benchmark-of-approximate-nearest-neighbor-libraries.html) can narrow down the potential nearest neighbours of our image in latent space (encoder code). We will caclulate nearest neighbours in brute force way for simplicity.

In [33]:

```
# restore trained encoder weights
s = reset_tf_session()
encoder, decoder = build_deep_autoencoder(IMG_SHAPE, code_size=32)
encoder.load_weights("encoder.h5")
```

In [34]:

```
images = X_train
#codes = ### YOUR CODE HERE: encode all images ###
codes = encoder.predict(images)
assert len(codes) == len(images)
```

In [35]:

```
from sklearn.neighbors.unsupervised import NearestNeighbors
nei_clf = NearestNeighbors(metric="euclidean")
nei_clf.fit(codes)
```

Out[35]:

```
NearestNeighbors(algorithm='auto', leaf_size=30, metric='euclidean',
         metric_params=None, n_jobs=1, n_neighbors=5, p=2, radius=1.0)
```

In [36]:

```
def get_similar(image, n_neighbors=5):
    assert image.ndim==3,"image must be [batch,height,width,3]"

    code = encoder.predict(image[None])

    (distances,),(idx,) = nei_clf.kneighbors(code,n_neighbors=n_neighbors)

    return distances,images[idx]
```

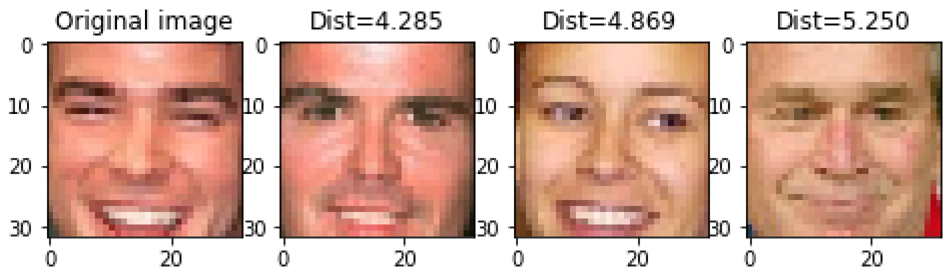
In [37]:

```
def show_similar(image):

    distances,neighbors = get_similar(image,n_neighbors=3)

    plt.figure(figsize=[8,7])
    plt.subplot(1,4,1)
    show_image(image)
    plt.title("Original image")

    for i in range(3):
        plt.subplot(1,4,i+2)
        show_image(neighbors[i])
        plt.title("Dist=%.3f"%distances[i])
    plt.show()
```

Cherry-picked examples:

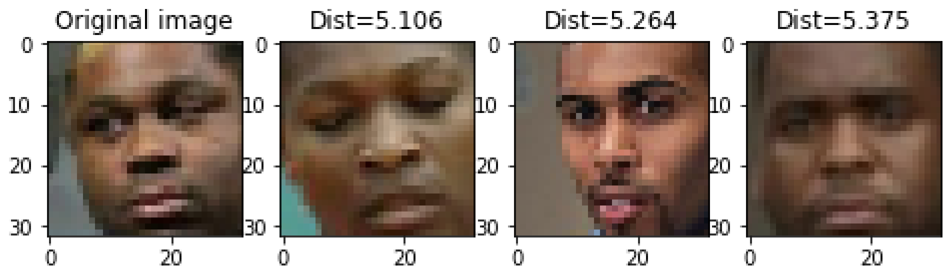
In [38]:

```
# smiles
show_similar(X_test[247])
```



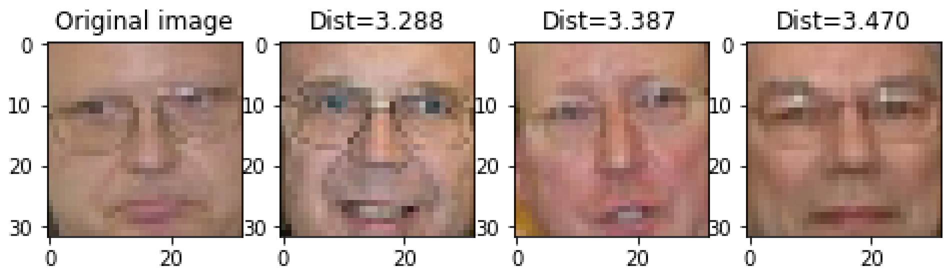
In [39]:

```
# ethnicity
show_similar(X_test[56])
```



In [40]:

```
# glasses
show_similar(X_test[63])
```



Optional: Cheap image morphing

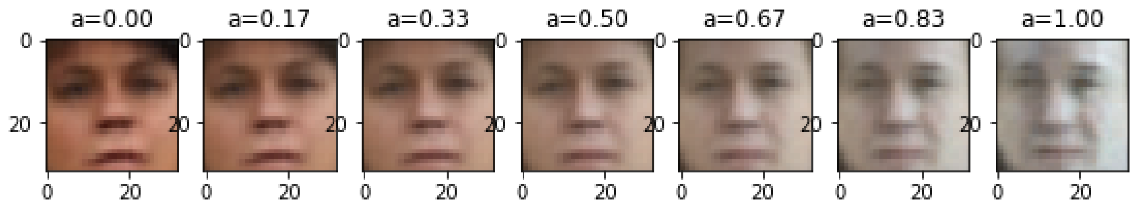
We can take linear combinations of image codes to produce new images with decoder.

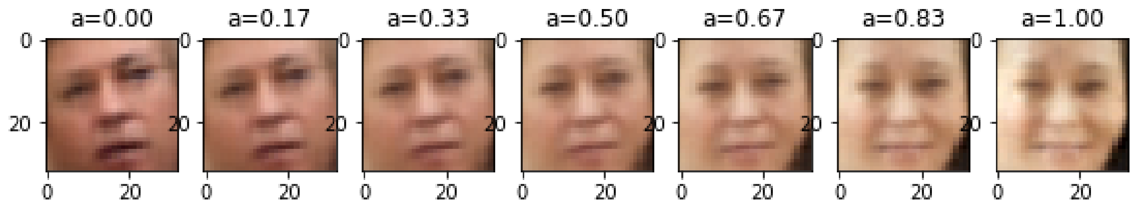
In [41]:

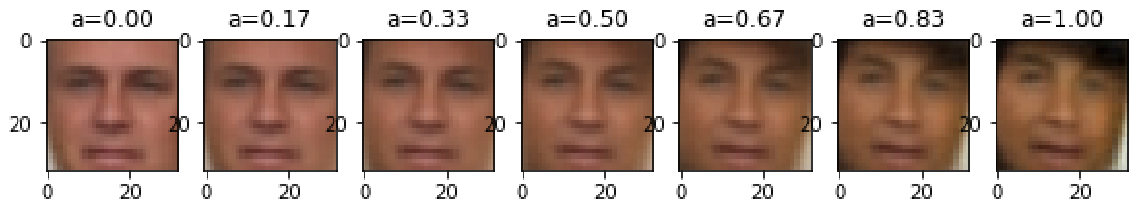
```
# restore trained encoder weights
s = reset_tf_session()
encoder, decoder = build_deep_autoencoder(IMG_SHAPE, code_size=32)
encoder.load_weights("encoder.h5")
decoder.load_weights("decoder.h5")
```

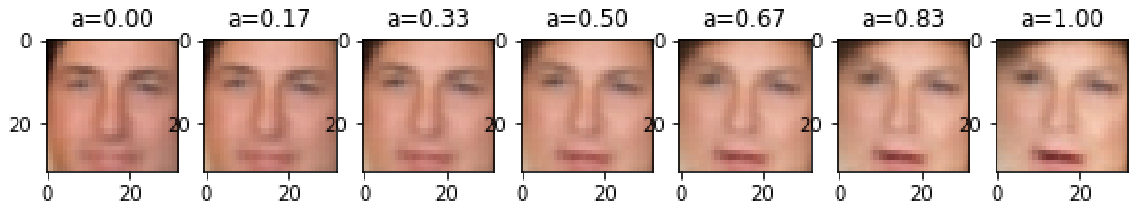
In [42]:

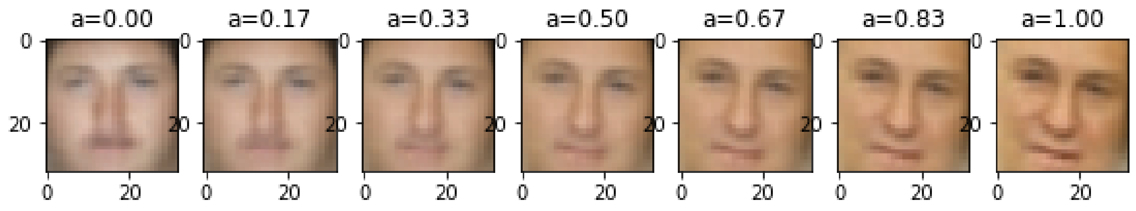
```
for _ in range(5):
    image1,image2 = X_test[np.random.randint(0,len(X_test),size=2)]

    code1, code2 = encoder.predict(np.stack([image1, image2]))

    plt.figure(figsize=[10,4])
    for i,a in enumerate(np.linspace(0,1,num=7)):

        output_code = code1*(1-a) + code2*(a)
        output_image = decoder.predict(output_code[None])[0]

        plt.subplot(1,7,i+1)
        show_image(output_image)
        plt.title("a=%.2f"%a)

    plt.show()
```











That's it!

Of course there's a lot more you can do with autoencoders.

If you want to generate images from scratch, however, we recommend you our honor track on Generative Adversarial Networks or GANs.