How to do unsupervised learning with an autoencoder

By WJG. Took the code from https://blog.keras.io/building-autoencoders-in-keras.html (https://blog.keras.io/building-autoencoders-in-keras.html)

1. Create model

- The autoencoder architecture is the following: Input --> 32FC layer --> Output FC means 'fully connected layer'.
- In the code, the layers are coded as follows:
  - First part (Input --> 32FC layer) is denoted as `encoder`;
  - Second part (32D layer --> Output) is denoted as `decoder`.
  - Concatenating the encoder and decoder yields the `autoencoder` model.
- The 32FC layer (i.e., output of the encoder) is treated as the encoded representation (aka feature vector).
- The decoder can be seen as a generative model taking as input 32D vectors.

In [1]:

```
from keras.layers import Input, Dense
from keras.models import Model

# this is the size of our encoded representations
encoding_dim = 32  # 32 floats -> compression of factor 24.5, assuming the inp
ut is 784 floats

# this is our input placeholder
input_img = Input(shape=(784,))
# "encoded" is the encoded representation of the input
encoded = Dense(encoding_dim, activation='relu')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded = Dense(784, activation='sigmoid')(encoded)

# this model maps an input to its reconstruction
autoencoder = Model(input=input_img, output=decoded)

# this model maps an input to its encoded representation
encoder = Model(input=input_img, output=encoded)

# create a placeholder for an encoded (32-dimensional) input
encoded_input = Input(shape=(encoding_dim,))
# retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# create the decoder model
decoder = Model(input=encoded_input, output=decoder_layer(encoded_input))

autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

```
Using Theano backend.

Couldn't import dot_parser, loading of dot files will not be possi
ble.

Using gpu device 0: Tesla K80 (CNMeM is disabled, CuDNN 5103)
/usr/prog/pythonML/3.0-goolf-1.5.14-NX-python-2.7.11/lib/python2.7
/site-packages/Theano-0.8.0-py2.7.egg/theano/sandbox/cuda/__init__
.py:600: UserWarning: Your CuDNN version is more recent then Thean
o. If you see problems, try updating Theano or downgrading CuDNN t
o version 4.
  warnings.warn(warn)
```

2. Load MNIST data

In [2]:

```
from keras.datasets import mnist
import numpy as np
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

3. Normalize images

In [3]:

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print x_train.shape
print x_test.shape
```

```
(60000, 784)
(10000, 784)
```

4. Train model

In [4]:

```
autoencoder.fit(x_train, x_train,
                nb_epoch=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

```
WARNING (theano.gof.cmodule): WARNING: your Theano flags `gcc.cxxf
lags` specify an `-march=X` flags.
         It is better to let Theano/g++ find it automatically, but
we don't do it now

Train on 60000 samples, validate on 10000 samples
Epoch 1/50
60000/60000 [==============================] - 1s - loss: 0.3750 -
val_loss: 0.2742
Epoch 2/50
60000/60000 [==============================] - 1s - loss: 0.2683 -
val_loss: 0.2595
Epoch 3/50
60000/60000 [==============================] - 1s - loss: 0.2492 -
val_loss: 0.2360
Epoch 4/50
60000/60000 [==============================] - 1s - loss: 0.2270 -
val_loss: 0.2162
Epoch 5/50
60000/60000 [==============================] - 1s - loss: 0.2103 -
val_loss: 0.2024
Epoch 6/50
60000/60000 [==============================] - 1s - loss: 0.1987 -
val_loss: 0.1924
Epoch 7/50
60000/60000 [==============================] - 1s - loss: 0.1900 -
val_loss: 0.1847
Epoch 8/50
60000/60000 [==============================] - 1s - loss: 0.1829 -
val_loss: 0.1783
Epoch 9/50
60000/60000 [==============================] - 1s - loss: 0.1768 -
val_loss: 0.1726
Epoch 10/50
```

```
60000/60000 [==============================] - 1s - loss: 0.1714 -
val_loss: 0.1674
Epoch 11/50
60000/60000 [==============================] - 1s - loss: 0.1665 -
val_loss: 0.1627
Epoch 12/50
60000/60000 [==============================] - 1s - loss: 0.1620 -
val_loss: 0.1585
Epoch 13/50
60000/60000 [==============================] - 1s - loss: 0.1579 -
val_loss: 0.1545
Epoch 14/50
60000/60000 [==============================] - 1s - loss: 0.1541 -
val_loss: 0.1510
Epoch 15/50
60000/60000 [==============================] - 1s - loss: 0.1507 -
val_loss: 0.1477
Epoch 16/50
60000/60000 [==============================] - 1s - loss: 0.1476 -
val_loss: 0.1447
Epoch 17/50
60000/60000 [==============================] - 1s - loss: 0.1448 -
val_loss: 0.1420
Epoch 18/50
60000/60000 [==============================] - 1s - loss: 0.1421 -
val_loss: 0.1394
Epoch 19/50
60000/60000 [==============================] - 1s - loss: 0.1397 -
val_loss: 0.1370
Epoch 20/50
60000/60000 [==============================] - 1s - loss: 0.1374 -
val_loss: 0.1348
Epoch 21/50
60000/60000 [==============================] - 1s - loss: 0.1352 -
val_loss: 0.1326
Epoch 22/50
60000/60000 [==============================] - 1s - loss: 0.1332 -
val_loss: 0.1306
Epoch 23/50
60000/60000 [==============================] - 1s - loss: 0.1312 -
val_loss: 0.1287
Epoch 24/50
60000/60000 [==============================] - 1s - loss: 0.1293 -
val_loss: 0.1268
Epoch 25/50
60000/60000 [==============================] - 1s - loss: 0.1275 -
val_loss: 0.1250
Epoch 26/50
60000/60000 [==============================] - 1s - loss: 0.1258 -
val_loss: 0.1234
Epoch 27/50
60000/60000 [==============================] - 1s - loss: 0.1241 -
val_loss: 0.1217
Epoch 28/50
60000/60000 [==============================] - 1s - loss: 0.1225 -
val_loss: 0.1202
Epoch 29/50
```

```
60000/60000 [==============================] - 1s - loss: 0.1210 -
val_loss: 0.1187
Epoch 30/50
60000/60000 [==============================] - 1s - loss: 0.1196 -
val_loss: 0.1173
Epoch 31/50
60000/60000 [==============================] - 1s - loss: 0.1182 -
val_loss: 0.1160
Epoch 32/50
60000/60000 [==============================] - 1s - loss: 0.1170 -
val_loss: 0.1147
Epoch 33/50
60000/60000 [==============================] - 1s - loss: 0.1158 -
val_loss: 0.1135
Epoch 34/50
60000/60000 [==============================] - 1s - loss: 0.1146 -
val_loss: 0.1124
Epoch 35/50
60000/60000 [==============================] - 1s - loss: 0.1136 -
val_loss: 0.1114
Epoch 36/50
60000/60000 [==============================] - 1s - loss: 0.1126 -
val_loss: 0.1104
Epoch 37/50
60000/60000 [==============================] - 1s - loss: 0.1117 -
val_loss: 0.1095
Epoch 38/50
60000/60000 [==============================] - 1s - loss: 0.1108 -
val_loss: 0.1087
Epoch 39/50
60000/60000 [==============================] - 1s - loss: 0.1100 -
val_loss: 0.1079
Epoch 40/50
60000/60000 [==============================] - 1s - loss: 0.1092 -
val_loss: 0.1072
Epoch 41/50
60000/60000 [==============================] - 1s - loss: 0.1085 -
val_loss: 0.1065
Epoch 42/50
60000/60000 [==============================] - 1s - loss: 0.1079 -
val_loss: 0.1059
Epoch 43/50
60000/60000 [==============================] - 1s - loss: 0.1072 -
val_loss: 0.1052
Epoch 44/50
60000/60000 [==============================] - 1s - loss: 0.1067 -
val_loss: 0.1047
Epoch 45/50
60000/60000 [==============================] - 1s - loss: 0.1061 -
val_loss: 0.1041
Epoch 46/50
60000/60000 [==============================] - 1s - loss: 0.1056 -
val_loss: 0.1036
Epoch 47/50
60000/60000 [==============================] - 1s - loss: 0.1051 -
val_loss: 0.1032
Epoch 48/50
```

```
60000/60000 [==============================] - 1s - loss: 0.1047 -
val_loss: 0.1027
Epoch 49/50
60000/60000 [==============================] - 1s - loss: 0.1042 -
val_loss: 0.1023
Epoch 50/50
60000/60000 [==============================] - 1s - loss: 0.1038 -
val_loss: 0.1019
```

Out[4]:

```
<keras.callbacks.History at 0x2aab14b0dc50>
```

5. Encode and decode images from test set

First row in the resulting figure are the input images Second row in the resulting figure are the reconstructed images. The 32D codes are in `encoded_imgs`

In [5]:

```
# encode and decode some digits
# note that we take them from the *test* set
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
## Here we can pass our own encodngs to the decoder to predict
```

In [6]:

```
%matplotlib inline
import matplotlib.pyplot as plt
#plt.rcParams['figure.figsize'] = (10, 10)
#plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

n = 10  # how many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i + 1)

    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

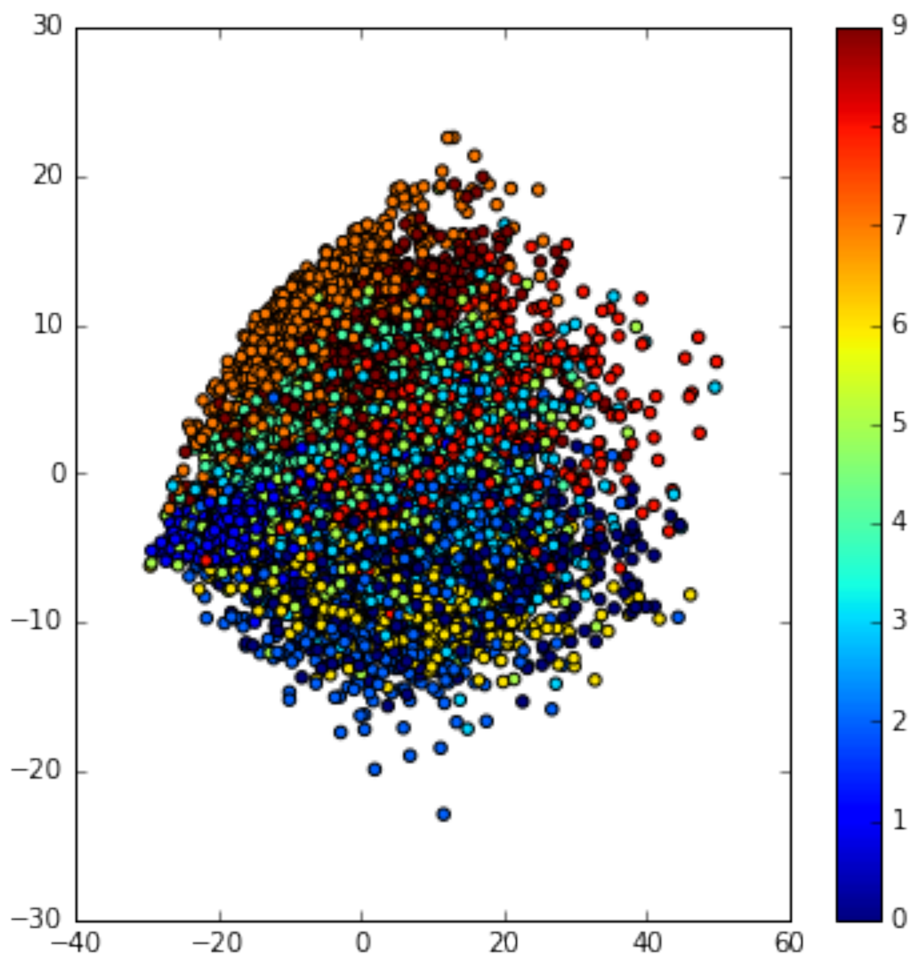
6. Visualize encoded vectors

The encoder generates one 32D vector per image. We can visualize the collection of vectors with,e.g., PCA (Note: Deeper architectures would allow us to map directly the images to 2D vectors; https://blog.keras.io/building-autoencoders-in-keras.html (https://blog.keras.io/building-autoencoders-in-keras.html)). Below we visualize the vectors generated for the test dataset only (10k images). In the scatter plot, one data point corresponds to one image. The colors are the actual digits annotations (which were NOT used during training). Even with this shallow architecture we see that the digits are NOT randomly scattered, but rather the autoencoder managed to extract some structure within the data. The 32D vectors (or the PCA components, if you wish) can be used as feature vectors for, e.g., training other machine-learning methods.

In [34]:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
encoded_imgs = encoder.predict(x_test);
nImages = encoded_imgs.size / 32;
print nImages
data = encoded_imgs.reshape(nImages,32);
pca.fit(data);
scores = pca.transform(data);
plt.rcParams['image.cmap'] = 'jet'
plt.figure(figsize=(6, 6))
plt.scatter(scores[:, 0], scores[:, 1], c=y_test)
plt.colorbar()
plt.show()
```

10000



7. Synthesize some images with the decoder

We'll take a reference encoding vector from the test set and add a noise vector from a zero-mean Gaussian with standard deviation `sigma`. By varying `sigma` we can add a stronger distortion to the encoding, which leads to some interesting synthetically generated patterns. We could also pass directly the noise vector to the decoder, which yields even weirder images.

In [36]:

```
#sigma = 0.01;
#sigma = 0.1
#sigma = 1;
sigma = 3;
#sigma = 10;

n = 10  # how many digits we will display
plt.figure(figsize=(10, 2))
plt.rcParams['image.cmap'] = 'gray'
for i in range(n):
    # synthesize digit from a real image distorted with a random vector
    ax = plt.subplot(2, n, i + 1)
    noise = sigma * np.random.randn(32) # 32 is the encoding dimension
    mu = encoded_imgs[1]; # some reference encoding, to not be so far away in the latent space
    z = mu + noise.reshape(-1,32);
    digitSynth = decoder.predict(z);
    plt.imshow(digitSynth.reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

In [ ]: