Variational Autoencoders

Variational autoencoders (VAE) are a powerful and widely-used class of models to learn complex data distributions in an unsupervised fashion.

In [1]:

```
import warnings
warnings.filterwarnings("ignore")

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('/tmp/data/MNIST_data')
```

```
Extracting /tmp/data/MNIST_data/train-images-idx3-ubyte.gz
Extracting /tmp/data/MNIST_data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/MNIST_data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/MNIST_data/t10k-labels-idx1-ubyte.gz
```

In [3]:

```
def lrelu(x, alpha=0.3):
    return tf.maximum(x, tf.multiply(x, alpha))
```

In [4]:

```
tf.reset_default_graph()

batch_size = 64

X_in = tf.placeholder(dtype=tf.float32, shape=[None, 28, 28], name='X')
Y    = tf.placeholder(dtype=tf.float32, shape=[None, 28, 28], name='Y')
Y_flat = tf.reshape(Y, shape=[-1, 28 * 28])
keep_prob = tf.placeholder(dtype=tf.float32, shape=(), name='keep_prob')

dec_in_channels = 1
n_latent = 8#16#

reshaped_dim = [-1, 7, 7, dec_in_channels]
inputs_decoder = 49 * dec_in_channels // 2
print('49 * {} // 2 = {}'.format(dec_in_channels,inputs_decoder ))
```

```
49 * 1 // 2 = 24
```

Defining the encoder

What's most noteworthy is the fact that we are creating two vectors in our encoder, as the encoder is supposed to create objects following a Gaussian Distribution:

- A vector of means
- A vector of standard deviations

You will see later how we "force" the encoder to make sure it really creates values following a Normal Distribution. The returned values that will be fed to the decoder are the z-values. We will need the mean and standard deviation of our distributions later, when computing losses.

In [5]:

```
def encoder(X_in, keep_prob):
    activation = lrelu
    with tf.variable_scope("encoder", reuse=None):
        X = tf.reshape(X_in, shape=[-1, 28, 28, 1])
        x = tf.layers.conv2d(X, filters=64, kernel_size=4, strides=2, padding='same
        print(x.shape) #(?, 14, 14, 64)
        x = tf.nn.dropout(x, keep_prob)
        x = tf.layers.conv2d(x, filters=64, kernel_size=4, strides=2, padding='same
        print(x.shape) #(?, 7, 7, 64)
        x = tf.nn.dropout(x, keep_prob)
        x = tf.layers.conv2d(x, filters=64, kernel_size=4, strides=1, padding='same
        print(x.shape) #(?, 7, 7, 64)
        x = tf.nn.dropout(x, keep_prob)
        x = tf.contrib.layers.flatten(x)
        print(x.shape) #(?, 3136)

        mn = tf.layers.dense(x, units=n_latent) # units: Integer or Long, dimension
        print(mn.shape) #(?, 8)
        sd = 0.5 * tf.layers.dense(x, units=n_latent)
        print(sd.shape) #(?, 8)

        epsilon = tf.random_normal(tf.stack([tf.shape(x)[0], n_latent]))
        print(epsilon.shape) #(?, 8)
        z  = mn + tf.multiply(epsilon, tf.exp(sd))
        print(z.shape) #(?, 8)
        return z, mn, sd
```

In [6]:

```
print('X_in : ', X_in.shape)
sampled, mn, sd = encoder(X_in, keep_prob)
print(sampled.shape) #(?, 8)
print(sampled)
```

```
X_in :  (?, 28, 28)
(?, 14, 14, 64)
(?, 7, 7, 64)
(?, 7, 7, 64)
(?, 3136)
(?, 8)
(?, 8)
(?, 8)
(?, 8)
(?, 8)
Tensor("encoder/add:0", shape=(?, 8), dtype=float32)
```

Defining the decoder

The decoder **does not** care about whether the input values are sampled from some specific distribution that has been defined by us. It simply will try to reconstruct the input images. To this end, we use a series of transpose convolutions.

In [7]:

```
def decoder(sampled_z, keep_prob):
    with tf.variable_scope("decoder", reuse=None):
        x = tf.layers.dense(sampled_z, units=inputs_decoder, activation=lrelu)
        print(x.shape) #(?, 24)
        #x = tf.layers.dense(sampled_z, inputs_decoder, activation=lrelu)
        x = tf.layers.dense(x, units=inputs_decoder * 2 + 1, activation=lrelu)
        print(x.shape) #(?, 49)
        x = tf.reshape(x, reshaped_dim)
        print(x.shape) #(?, 7, 7, 1)
        x = tf.layers.conv2d_transpose(x, filters=64, kernel_size=4, strides=2, pad
        print(x.shape) #(?, 14, 14, 64)
        x = tf.nn.dropout(x, keep_prob)
        x = tf.layers.conv2d_transpose(x, filters=64, kernel_size=4, strides=1, pad
        print(x.shape) #(?, 14, 14, 64)
        x = tf.nn.dropout(x, keep_prob)
        x = tf.layers.conv2d_transpose(x, filters=64, kernel_size=4, strides=1, pad
        print(x.shape) #(?, 14, 14, 64)

        x = tf.contrib.layers.flatten(x)
        print(x.shape) #(?, 12544)

        x = tf.layers.dense(x, units=28*28, activation=tf.nn.sigmoid)
        print(x.shape) #(?, 784)

        img = tf.reshape(x, shape=[-1, 28, 28])
        return img
```

Now, we'll wire together both parts:

In [8]:

```
dec = decoder(sampled, keep_prob)
```

```
(?, 24)
(?, 49)
(?, 7, 7, 1)
(?, 14, 14, 64)
(?, 14, 14, 64)
(?, 14, 14, 64)
(?, 12544)
(?, 784)
```

Computing losses and enforcing a Gaussian latent distribution

For computing the image reconstruction loss, we simply use squared difference (which could lead to images sometimes looking a bit fuzzy). This loss is combined with the Kullback-Leibler divergence, which makes sure our latent values will be sampled from a normal distribution. For more on this topic, please take a look a Jaan Altosaar's great article on VAEs.

In [9]:

```
unreshaped = tf.reshape(dec, [-1, 28*28])
img_loss = tf.reduce_sum(tf.squared_difference(unreshaped, Y_flat), 1)
latent_loss = -0.5 * tf.reduce_sum(1.0 + 2.0 * sd - tf.square(mn) - tf.exp(2.0 * sd
loss = tf.reduce_mean(img_loss + latent_loss)
optimizer = tf.train.AdamOptimizer(0.0005).minimize(loss)
```

In [10]:

```
sess = tf.Session()
sess.run(tf.global_variables_initializer())
```

Training the network

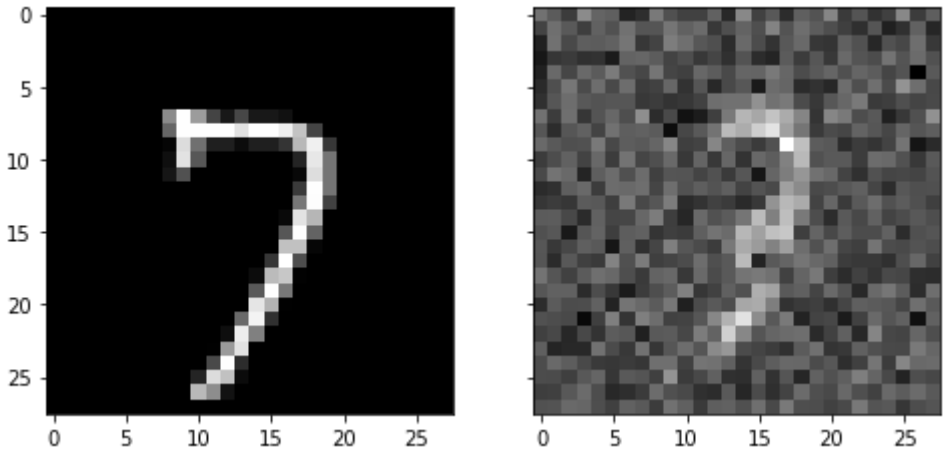
In [11]:

```
vloss = []
for i in range(30000):
    batch = [np.reshape(b, [28, 28]) for b in mnist.train.next_batch(batch_size=bat
    l, _ = sess.run([loss, optimizer], feed_dict = {X_in: batch, Y: batch, keep_pro
    vloss.append(l)

    if not i % 200:
        ls, d, i_ls, d_ls, mu, sigm = sess.run([loss, dec, img_loss, latent_loss, m

        fig, axes = plt.subplots(nrows=1, ncols=2, sharex=True, sharey=True, figsiz

        axes[0].imshow(np.reshape(batch[0], [28, 28]), cmap='gray')
        #plt.show()
        axes[1].imshow(d[0], cmap='gray')
        plt.show()
        print(i, ls, np.mean(i_ls), np.mean(d_ls))
```



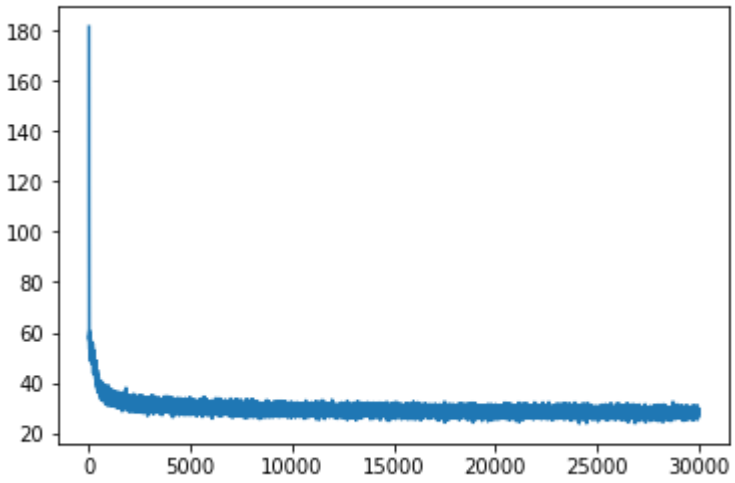

```
0 177.0364 177.03293 0.0034884755
```

In [12]:

```
plt.figure()
plt.plot(vloss)
```

Out[12]:

```
[<matplotlib.lines.Line2D at 0x7f7294066eb8>]
```



Generating new data

The most awesome part is that we are now able to create new characters. To this end, we simply sample values from a unit normal distribution and feed them to our decoder. Most of the created characters look just like they've been written by humans.

In [13]:

```
x1 = np.random.normal(0, 1, n_latent)
print(x1)
```

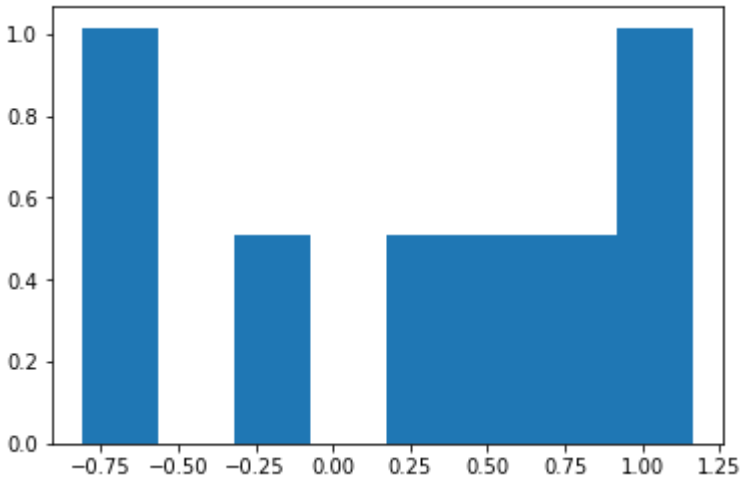
```
[ 1.16114378 -0.8076441  -0.13953766 -0.74217957  0.77082946  0.932676
64
  0.31933542  0.60170802]
```

In [14]:

```
n_bins = n_latent
plt.hist(x1, n_bins, normed=1, histtype='bar')
```

Out[14]:

```
(array([1.01585347, 0.        , 0.50792674, 0.        , 0.50792674,
        0.50792674, 0.50792674, 1.01585347]),
 array([-0.8076441 , -0.56154561, -0.31544713, -0.06934864,  0.1767498
4,
         0.42284832,  0.66894681,  0.91504529,  1.16114378]),
 <a list of 8 Patch objects>)
```



In [15]:

```
img = sess.run(dec, feed_dict = {sampled: [x1], keep_prob: 1.0})
img = np.reshape(img, [28, 28])
plt.figure(figsize=(2,2))
plt.axis('off')
plt.imshow(img, cmap='gray')
```

Out[15]:

```
<matplotlib.image.AxesImage at 0x7f7294775c88>
```

In [16]:

```
randoms = [np.random.normal(0, 1, n_latent) for _ in range(10)]
print(np.shape(randoms))
```

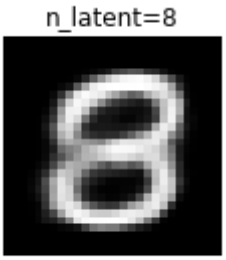
```
(10, 8)
```

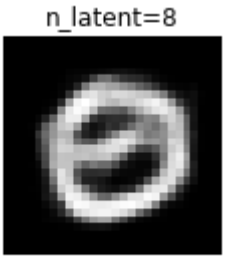
In [17]:

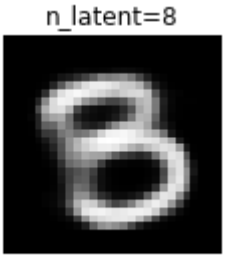
```
import os
if not os.path.exists('out'):
    os.makedirs('out')
```

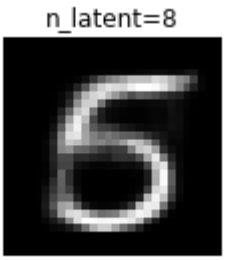
In [18]:

```
imgs = sess.run(dec, feed_dict = {sampled: randoms, keep_prob: 1.0})
imgs = [np.reshape(imgs[i], [28, 28]) for i in range(len(imgs))]
i=0
for img in imgs:
    f1 = plt.figure(figsize=(2,2))
    plt.axis('off')
    plt.title('n_latent={}'.format(n_latent))
    plt.imshow(img, cmap='gray')
    plt.savefig('out/l{}_{}.png'.format(n_latent, i))
    i = i+1
```







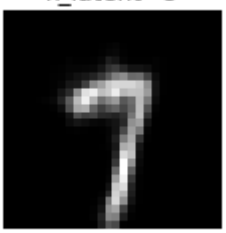


n_latent=8

n_latent=8

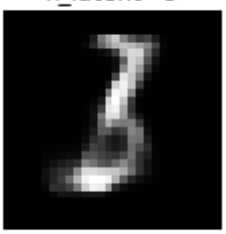
n_latent=8

n_latent=8



n_latent=8

n_latent=8



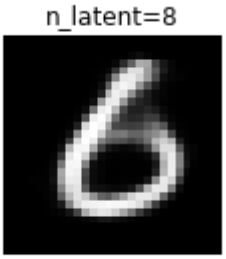
Just for test

In [19]:

```
x2 = [-1.0, 1.0, 0.1, 0.3, -0.2, 0.7, -0.5, -2.5]
#x2 = [-1.0, 1.0, 0.1, 0.3, -0.2, -0.7, -0.5, -2.5, 1.2, 1.8, -0.1, 0.7, -0.2, 0.7,
img = sess.run(dec, feed_dict = {sampled:[x2], keep_prob: 1.0})
img = np.reshape(img, [28, 28])
plt.figure(figsize=(2,2))
plt.axis('off')
plt.title('n_latent={}'.format(n_latent))
plt.imshow(img, cmap='gray')
```

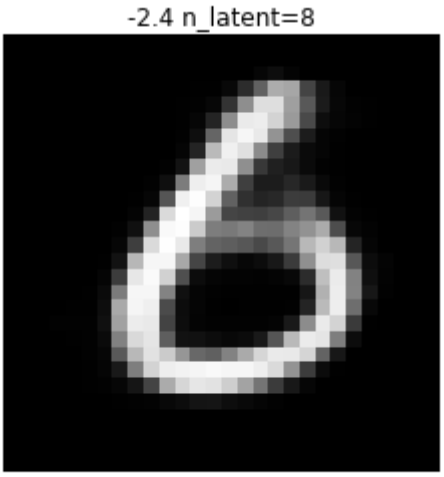
Out[19]:

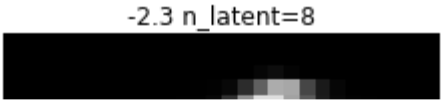
```
<matplotlib.image.AxesImage at 0x7f7294912080>
```



In [20]:

```
x2 = [-1.0, 1.0, 0.1, 0.3, -0.2, 0.7, -0.5, -2.5]
#x2 = [-1.0, 1.0, 0.1, 0.3, -0.2, -0.7, -0.5, -2.5, 1.2, 1.8, -0.1, 0.7, -0.2, 0.7,
for i in range(50):
    x2[7] = x2[7]+0.1
    img = sess.run(dec, feed_dict = {sampled:[x2], keep_prob: 1.0})
    img = np.reshape(img, [28, 28])
    plt.figure()
    plt.axis('off')
    plt.title('{} n_latent={}'.format(round(x2[7], 2), n_latent))
    plt.imshow(img, cmap='gray')
    plt.savefig('out/test_l{}_{}.png'.format(n_latent, i))
```





References

- https://jaan.io/what-is-variational-autoencoder-vae-tutorial/ (https://jaan.io/what-is-variational-autoencoder-vae-tutorial/)
- https://github.com/FelixMohr/Deep-learning-with-Python/blob/master/VAE.ipynb (https://github.com/FelixMohr/Deep-learning-with-Python/blob/master/VAE.ipynb)