(https://www.bigdatauniversity.com)

RESTRICTED BOLTZMANN MACHINES

Introduction

Restricted Boltzmann Machine (RBM): RBMs are shallow neural nets that learn to reconstruct data by themselves in an unsupervised fashion.

Why are RBMs important?
It can automatically extract **meaningful** features from a given input.

How does it work?
RBM is a 2 layer neural network. Simply, RBM takes the inputs and translates those into a set of binary values that represents them in the hidden layer. Then, these numbers can be translated back to reconstruct the inputs. Through several forward and backward passes, the RBM will be trained, and a trained RBM can reveal which features are the most important ones when detecting patterns.

What are the applications of RBM?
RBM is useful for Collaborative Filtering (http://www.cs.utoronto.ca/~hinton/absps/netflixICML.pdf), dimensionality reduction, classification, regression, feature learning, topic modeling and even **Deep Belief Networks**.

Is RBM a generative or Discriminative model?
RBM is a generative model. Let me explain it by first, see what is different between discriminative and generative models:

Discriminative: Consider a classification problem in which we want to learn to distinguish between Sedan cars (y = 1) and SUV cars (y = 0), based on some features of cars. Given a training set, an algorithm like logistic regression tries to find a straight line—that is, a decision boundary—that separates the suv and sedan.
Generative: looking at cars, we can build a model of what Sedan cars look like. Then, looking at SUVs, we can build a separate model of what SUV cars look like. Finally, to classify a new car, we can match the new car against the Sedan model, and match it against the SUV model, to see whether the new car looks more like the SUV or Sedan.

Generative Models specify a probability distribution over a dataset of input vectors. We can do both supervise and unsupervised tasks with generative models:

- In an unsupervised task, we try to form a model for P(x), where P is the probability given x as an input vector.
- In the supervised task, we first form a model for P(x|y), where P is the probability of x given y(the label for x). For example, if y = 0 indicates whether a car is a SUV or y = 1 indicates indicate a car is a Sedan, then p(x|y = 0) models the distribution of SUVs' features, and p(x|y = 1) models the distribution of Sedans' features. If we manage to find P(x|y) and P(y), then we can use `Bayes rule` to estimate P(y|x), because: $$p(y|x) = \frac{p(x|y)p(y)}{p(x)}$$

Now the question is, can we build a generative model, and then use it to create synthetic data by directly sampling from the modeled probability distributions? Lets see.

Table of Contents

Initialization

First we have to load the utility file which contains different utility functions that are not connected in any way to the networks presented in the tutorials, but rather help in processing the outputs into a more understandable way.

In [1]:

```
import urllib.request
with urllib.request.urlopen("http://deeplearning.net/tutorial/code/utils.py") as ur
l:
    response = url.read()
target = open('utils.py', 'w')
target.write(response.decode('utf-8'))
target.close()
```

Now, we load in all the packages that we use to create the net including the TensorFlow package:

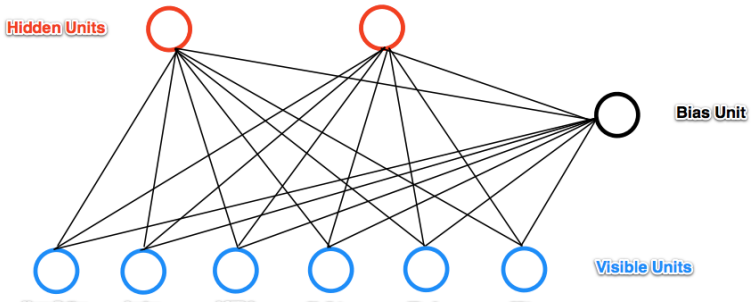
In [2]:

```
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
#!pip install pillow
from PIL import Image
from utils import tile_raster_images
import matplotlib.pyplot as plt
%matplotlib inline
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:519: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:520: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:521: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:522: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:523: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:528: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

RBM layers

An RBM has two layers. The first layer of the RBM is called the **visible** (or input layer). Imagine that our toy example, has only vectors with 7 values, so the visible layer must have j=7 input nodes. The second layer is the **hidden** layer, which possesses i neurons in our case. Each hidden node can have either 0 or 1 values (i.e., si = 1 or si = 0) with a probability that is a logistic function of the inputs it receives from the other j visible units, called for example, p(si = 1). For our toy sample, we'll use 2 nodes in the hidden layer, so i = 2.



Each node in the first layer also has a **bias**. We will denote the bias as “v_bias” for the visible units. The **v_bias** is shared among all visible units.

Here we define the **bias** of second layer as well. We will denote the bias as “h_bias” for the hidden units. The **h_bias** is shared among all hidden units

In [3]:

```
v_bias = tf.placeholder("float", [7])
h_bias = tf.placeholder("float", [2])
```

We have to define weights among the input layer and hidden layer nodes. In the weight matrix, the number of rows are equal to the input nodes, and the number of columns are equal to the output nodes. Let **W** be the Tensor of 7x2 (7 - number of visible neurons, 2 - number of hidden neurons) that represents weights between neurons.

In [4]:

```
W = tf.constant(np.random.normal(loc=0.0, scale=1.0, size=(7, 2)).astype(np.float32
))
```

What RBM can do after training?

Think RBM as a model that has been trained based on images of a dataset of many SUV and Sedan cars. Also, imagine that the RBM network has only two hidden nodes, one for the weight and, and one for the size of cars, which in a sense, their different configurations represent different cars, one represent SUV cars and one for Sedan. In a training process, through many forward and backward passes, RBM adjust its weights to send a stronger signal to either the SUV node (0, 1) or the Sedan node (1, 0) in the hidden layer, given the pixels of images. Now, given a SUV in hidden layer, which distribution of pixels should we expect? RBM can give you 2 things. First, it encodes your images in hidden layer. Second, it gives you the probability of observing a case, given some hidden values.

How to inference?

RBM has two phases:

- Forward Pass
- Backward Pass or Reconstruction

Phase 1) Forward pass: Input one training sample (one image) **X** through all visible nodes, and pass it to all hidden nodes. Processing happens in each node in the hidden layer. This computation begins by making stochastic decisions about whether to transmit that input or not (i.e. to determine the state of each hidden layer). At the hidden layer's nodes, **X** is multiplied by a **W_{ij}** and added to **h_bias**. The result of those two operations is fed into the sigmoid function, which produces the node's output, $p({h_j})$, where j is the unit number.

$p({h_j})= \sigma(\sum_i w_{ij} x_i)$, where $\sigma()$ is the logistic function.

Now lets see what $p({h_j})$ represents. In fact, it is the probabilities of the hidden units. And, all values together are called **probability distribution**. That is, RBM uses inputs x to make predictions about hidden node activations. For example, imagine that the values of h_p for the first training item is [0.51 0.84]. It tells you what is the conditional probability for each hidden neuron to be at Phase 1):

- p(h_{1} = 1|V) = 0.51
- (h_{2} = 1|V) = 0.84

As a result, for each row in the training set, **a vector/tensor** is generated, which in our case it is of size [1x2], and totally n vectors ($p({h})$=[nx2]).

We then turn unit h_j on with probability $p(h_{j}|V)$, and turn it off with probability $1 - p(h_{j}|V)$.

Therefore, the conditional probability of a configuration of h given v (for a training sample) is:
$$p(\mathbf{h} \mid \mathbf{v}) = \prod_{j=0}^H p(h_j \mid \mathbf{v})$$

Now, sample a hidden activation vector **h** from this probability distribution $p({h_j})$. That is, we sample the activation vector from the probability distribution of hidden layer values.

Before we go further, let's look at a toy example for one case out of all input. Assume that we have a trained RBM, and a very simple input vector such as [1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0], lets see what would be the output of forward pass:

In [5]:

```
sess = tf.Session()
X = tf.constant([[1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]])
v_state = X
print ("Input: ", sess.run(v_state))

h_bias = tf.constant([0.1, 0.1])
print ("hb: ", sess.run(h_bias))
print ("w: ", sess.run(W))

# Calculate the probabilities of turning the hidden units on:
h_prob = tf.nn.sigmoid(tf.matmul(v_state, W) + h_bias)  #probabilities of the hidde
n units
print ("p(h|v): ", sess.run(h_prob))

# Draw samples from the distribution:
h_state = tf.nn.relu(tf.sign(h_prob - tf.random_uniform(tf.shape(h_prob)))) #states
print ("h0 states:", sess.run(h_state))
```

```
Input:  [[1. 0. 0. 1. 0. 0. 0.]]
hb:  [0.1 0.1]
w:  [[ 0.62607366  0.79537   ]
 [-0.6646771  -0.09528685]
 [-0.7703524   1.1189759 ]
 [ 1.6733332   0.05102924]
 [ 3.049211    0.5688824 ]
 [ 0.8351101   0.62181854]
 [-0.05040543  0.16212119]]
p(h|v):  [[0.916782   0.72039044]]
h0 states: [[1. 1.]]
```

Phase 2) Backward Pass (Reconstruction): The RBM reconstructs data by making several forward and backward passes between the visible and hidden layers.

So, in the second phase (i.e. reconstruction phase), the samples from the hidden layer (i.e. h) play the role of input. That is, **h** becomes the input in the backward pass. The same weight matrix and visible layer biases are used to go through the sigmoid function. The produced output is a reconstruction which is an approximation of the original input.

In [6]:

```
vb = tf.constant([0.1, 0.2, 0.1, 0.1, 0.1, 0.2, 0.1])
print ("b: ", sess.run(vb))
v_prob = sess.run(tf.nn.sigmoid(tf.matmul(h_state, tf.transpose(W)) + vb))
print ("p(vi|h): ", v_prob)
v_state = tf.nn.relu(tf.sign(v_prob - tf.random_uniform(tf.shape(v_prob))))
print ("v probability states: ", sess.run(v_state))
```

```
b:  [0.1 0.2 0.1 0.1 0.1 0.2 0.1]
p(vi|h):  [[0.820751   0.3635558  0.6103119  0.86108875 0.97629535 0.83
98253
  0.5527321 ]]
v probability states:  [[1. 0. 1. 1. 1. 1. 1.]]
```

RBM learns a probability distribution over the input, and then, after being trained, the RBM can generate new samples from the learned probability distribution. As you know, **probability distribution**, is a mathematical function that provides the probabilities of occurrence of different possible outcomes in an experiment.

The (conditional) probability distribution over the visible units v is given by

$p(\mathbf{v} \mid \mathbf{h}) = \prod_{i=0}^V p(v_i \mid \mathbf{h}),$

where,

$p(v_i \mid \mathbf{h}) = \sigma\left(a_i + \sum_{j=0}^H w_{ji} h_j \right)$

so, given current state of hidden units and weights, what is the probability of generating [1. 0. 0. 1. 0. 0. 0.] in reconstruction phase, based on the above **probability distribution** function?

In [7]:

```
inp = sess.run(X)
print(inp)
print(v_prob[0])
v_probability = 1
for elm, p in zip(inp[0],v_prob[0]) :
    if elm ==1:
        v_probability *= p
    else:
        v_probability *= (1-p)
v_probability
```

```
[[1. 0. 0. 1. 0. 0. 0.]]
[0.820751   0.3635558  0.6103119  0.86108875 0.97629535 0.8398253
 0.5527321 ]
```

Out[7]:

```
0.0002976679144122613
```

How similar X and V vectors are? Of course, the reconstructed values most likely will not look anything like the input vector because our network has not trained yet. Our objective is to train the model in such a way that the input vector and reconstructed vector to be same. Therefore, based on how different the input values look to the ones that we just reconstructed, the weights are adjusted.

MNIST

We will be using the MNIST dataset to practice the usage of RBMs. The following cell loads the MNIST dataset.

In [8]:

```
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
trX, trY, teX, teY = mnist.train.images, mnist.train.labels, mnist.test.images, mni
st.test.labels
```

```
WARNING:tensorflow:From <ipython-input-8-a0c1bc5755ed>:1: read_data_set
s (from tensorflow.contrib.learn.python.learn.datasets.mnist) is deprec
ated and will be removed in a future version.
Instructions for updating:
Please use alternatives such as official/mnist/dataset.py from tensorfl
ow/models.
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.
6/site-packages/tensorflow/contrib/learn/python/learn/datasets/mnist.p
y:260: maybe_download (from tensorflow.contrib.learn.python.learn.datas
ets.base) is deprecated and will be removed in a future version.
Instructions for updating:
Please write your own downloading logic.
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.
6/site-packages/tensorflow/contrib/learn/python/learn/datasets/mnist.p
y:262: extract_images (from tensorflow.contrib.learn.python.learn.datas
ets.mnist) is deprecated and will be removed in a future version.
Instructions for updating:
Please use tf.data to implement this functionality.
Extracting MNIST_data/train-images-idx3-ubyte.gz
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.
6/site-packages/tensorflow/contrib/learn/python/learn/datasets/mnist.p
y:267: extract_labels (from tensorflow.contrib.learn.python.learn.datas
ets.mnist) is deprecated and will be removed in a future version.
Instructions for updating:
Please use tf.data to implement this functionality.
Extracting MNIST_data/train-labels-idx1-ubyte.gz
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.
6/site-packages/tensorflow/contrib/learn/python/learn/datasets/mnist.p
y:110: dense_to_one_hot (from tensorflow.contrib.learn.python.learn.dat
asets.mnist) is deprecated and will be removed in a future version.
Instructions for updating:
Please use tf.one_hot on tensors.
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.
6/site-packages/tensorflow/contrib/learn/python/learn/datasets/mnist.p
y:290: DataSet.__init__ (from tensorflow.contrib.learn.python.learn.dat
asets.mnist) is deprecated and will be removed in a future version.
Instructions for updating:
Please use alternatives such as official/mnist/dataset.py from tensorfl
ow/models.
```

Lets look at the dimension of the images.

In [9]:

```
trX[1].shape
```

Out[9]:

```
(784,)
```

MNIST images have 784 pixels, so the visible layer must have 784 input nodes. For our case, we'll use 50 nodes in the hidden layer, so i = 50.

In [10]:

```
vb = tf.placeholder("float", [784])
hb = tf.placeholder("float", [50])
```

Let **W** be the Tensor of 784x50 (784 - number of visible neurons, 50 - number of hidden neurons) that represents weights between the neurons.

In [11]:

```
W = tf.placeholder("float", [784, 50])
```

Lets define the visible layer:

In [12]:

```
v0_state = tf.placeholder("float", [None, 784])
```

Now, we can define hidden layer:

In [13]:

```
h0_prob = tf.nn.sigmoid(tf.matmul(v0_state, W) + hb)  #probabilities of the hidden
 units
h0_state = tf.nn.relu(tf.sign(h0_prob - tf.random_uniform(tf.shape(h0_prob)))) #sam
ple_h_given_X
```

Now, we define reconstruction part:

In [14]:

```
v1_prob = tf.nn.sigmoid(tf.matmul(h0_state, tf.transpose(W)) + vb)
v1_state = tf.nn.relu(tf.sign(v1_prob - tf.random_uniform(tf.shape(v1_prob)))) #sam
ple_v_given_h
```

What is objective function?

Goal: Maximize the likelihood of our data being drawn from that distribution

Calculate error:
In each epoch, we compute the "error" as a sum of the squared difference between step 1 and step n, e.g the error shows the difference between the data and its reconstruction.

Note: tf.reduce_mean computes the mean of elements across dimensions of a tensor.

In [15]:

```
err = tf.reduce_mean(tf.square(v0_state - v1_state))
```

How to train the model?

Warning!! The following part discuss how to train the model which needs some algebra background. Still, you can skip this part and run the next cells.

As mentioned, we want to give a high probability to the input data we train on. So, in order to train an RBM, we have to maximize the product of probabilities assigned to all rows v (images) in the training set V (a matrix, where each row of it is treated as a visible vector v):

$$\arg\max_W \prod_{v \in V} P(v)$$

Which is equivalent, maximizing the expected log probability of V:

$$\arg\max_W \mathbb{E}\left[\sum_{v \in V} \log P(v)\right]$$

So, we have to update the weights wij to increase p(v) for all v in our training data during training. So we have to calculate the derivative:
$$\frac{\partial \log p(\mathbf v)}{\partial w_{ij}}$$

This cannot be easily done by typical **gradient descent (SGD)**, so we can use another approach, which has 2 steps:

1. Gibbs Sampling
2. Contrastive Divergence

Gibbs Sampling

First, given an input vector v we are using p(h|v) for prediction of the hidden values h.

- $p(h|v) = sigmoid(X \otimes W + hb)$
- h0 = sampleProb(h0)

Then, knowing the hidden values, we use p(v|h) for reconstructing of new input values v.

- $p(v|h) = sigmoid(h0 \otimes transpose(W) + vb)$
- $v1 = sampleProb(v1)$ (Sample v given h)

This process is repeated k times. After k iterations we obtain an other input vector vk which was recreated from original input values v0 or X.

Reconstruction steps:

- Get one data point from data set, like *x*, and pass it through the net
- Pass 0: (x) $\Rightarrow$ (h0) $\Rightarrow$ (v1) (v1 is reconstruction of the first pass)
- Pass 1: (v1) $\Rightarrow$ (h1) $\Rightarrow$ (v2) (v2 is reconstruction of the second pass)
- Pass 2: (v2) $\Rightarrow$ (h2) $\Rightarrow$ (v3) (v3 is reconstruction of the third pass)
- Pass n: (vk) $\Rightarrow$ (hk+1) $\Rightarrow$ (vk+1)(vk is reconstruction of the nth pass)

What is sampling here (sampleProb)?

In forward pass: We randomly set the values of each hi to be 1 with probability $sigmoid(v \otimes W + hb)$.

- To sample h given v means to sample from the conditional probability distribution P(h|v). It means that you are asking what are the probabilities of getting a specific set of values for the hidden neurons, given the values v for the visible neurons, and sampling from this probability distribution. In reconstruction: We randomly set the values of each vi to be 1 with probability $ sigmoid(h \otimes transpose(W) + vb)$.

contrastive divergence (CD-k)

The update of the weight matrix is done during the Contrastive Divergence step.

Vectors v0 and vk are used to calculate the activation probabilities for hidden values h0 and hk. The difference between the outer products of those probabilities with input vectors v0 and vk results in the update matrix:

$\Delta W =v0 \otimes h0 - vk \otimes hk$

Contrastive Divergence is actually matrix of values that is computed and used to adjust values of the W matrix. Changing W incrementally leads to training of W values. Then on each step (epoch), W is updated to a new value W' through the equation below:

$W' = W + alpha * \Delta W$

What is Alpha?
Here, alpha is some small step rate and is also known as the "learning rate".

Ok, lets assume that k=1, that is we just get one more step:

In [16]:

```
h1_prob = tf.nn.sigmoid(tf.matmul(v1_state, W) + hb)
h1_state = tf.nn.relu(tf.sign(h1_prob - tf.random_uniform(tf.shape(h1_prob)))) #sam
ple_h_given_X
```

In [17]:

```
alpha = 0.01
W_Delta = tf.matmul(tf.transpose(v0_state), h0_prob) - tf.matmul(tf.transpose(v1_st
ate), h1_prob)
update_w = W + alpha * W_Delta
update_vb = vb + alpha * tf.reduce_mean(v0_state - v1_state, 0)
update_hb = hb + alpha * tf.reduce_mean(h0_state - h1_state, 0)
```

Let's start a session and initialize the variables:

In [18]:

```
cur_w = np.zeros([784, 50], np.float32)
cur_vb = np.zeros([784], np.float32)
cur_hb = np.zeros([50], np.float32)
prv_w = np.zeros([784, 50], np.float32)
prv_vb = np.zeros([784], np.float32)
prv_hb = np.zeros([50], np.float32)
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
```

Lets look at the error of the first run:

In [19]:

```
sess.run(err, feed_dict={v0_state: trX, W: prv_w, vb: prv_vb, hb: prv_hb})
```

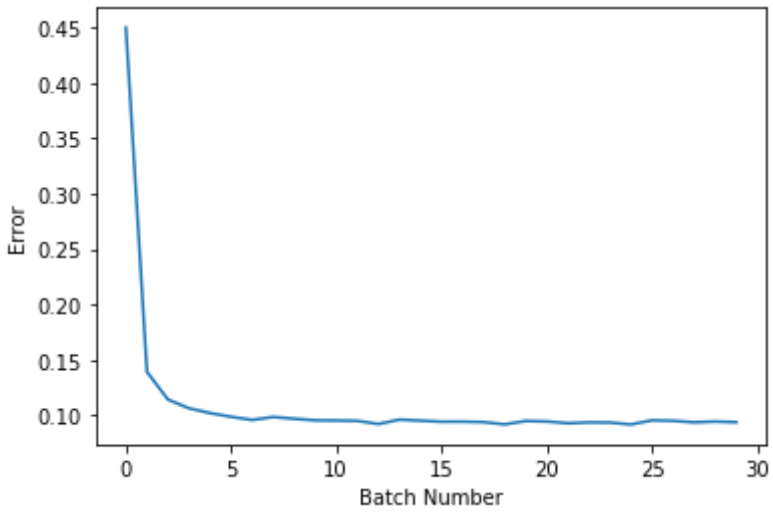
Out[19]:

```
0.48144788
```

In [20]:

```
#Parameters
epochs = 5
batchsize = 100
weights = []
errors = []

for epoch in range(epochs):
    for start, end in zip( range(0, len(trX), batchsize), range(batchsize, len(trX
), batchsize)):
        batch = trX[start:end]
        cur_w = sess.run(update_w, feed_dict={ v0_state: batch, W: prv_w, vb: prv_v
b, hb: prv_hb})
        cur_vb = sess.run(update_vb, feed_dict={v0_state: batch, W: prv_w, vb: prv_
vb, hb: prv_hb})
        cur_hb = sess.run(update_hb, feed_dict={ v0_state: batch, W: prv_w, vb: prv
_vb, hb: prv_hb})
        prv_w = cur_w
        prv_vb = cur_vb
        prv_hb = cur_hb
        if start % 10000 == 0:
            errors.append(sess.run(err, feed_dict={v0_state: trX, W: cur_w, vb: cur
_vb, hb: cur_hb}))
            weights.append(cur_w)
    print ('Epoch: %d' % epoch,'reconstruction error: %f' % errors[-1])
plt.plot(errors)
plt.xlabel("Batch Number")
plt.ylabel("Error")
plt.show()
```

```
Epoch: 0 reconstruction error: 0.098552
Epoch: 1 reconstruction error: 0.094880
Epoch: 2 reconstruction error: 0.093765
Epoch: 3 reconstruction error: 0.093356
Epoch: 4 reconstruction error: 0.093662
```



What is the final weight after training?

In [21]:

```
uw = weights[-1].T
print (uw) # a weight matrix of shape (50,784)
```

```
[[-2.5412369  -2.4994023  -2.4420772  ... -2.5205765  -2.4409456
  -2.4944847 ]
 [-0.95328546 -0.79584736 -0.861638   ... -0.70798224 -0.932385
  -0.8888958 ]
 [-0.31819803 -0.29674518 -0.3250337  ... -0.27715695 -0.32095295
  -0.3127117 ]
 ...
 [-0.84729224 -0.95299274 -0.93308765 ... -0.81702256 -0.89409417
  -0.8816251 ]
 [-0.29673967 -0.31259823 -0.27640024 ... -0.2940992  -0.32855764
  -0.30930287]
 [-2.7765954  -2.5743055  -2.7180948  ... -2.64742    -2.7095854
  -2.7877123 ]]
```

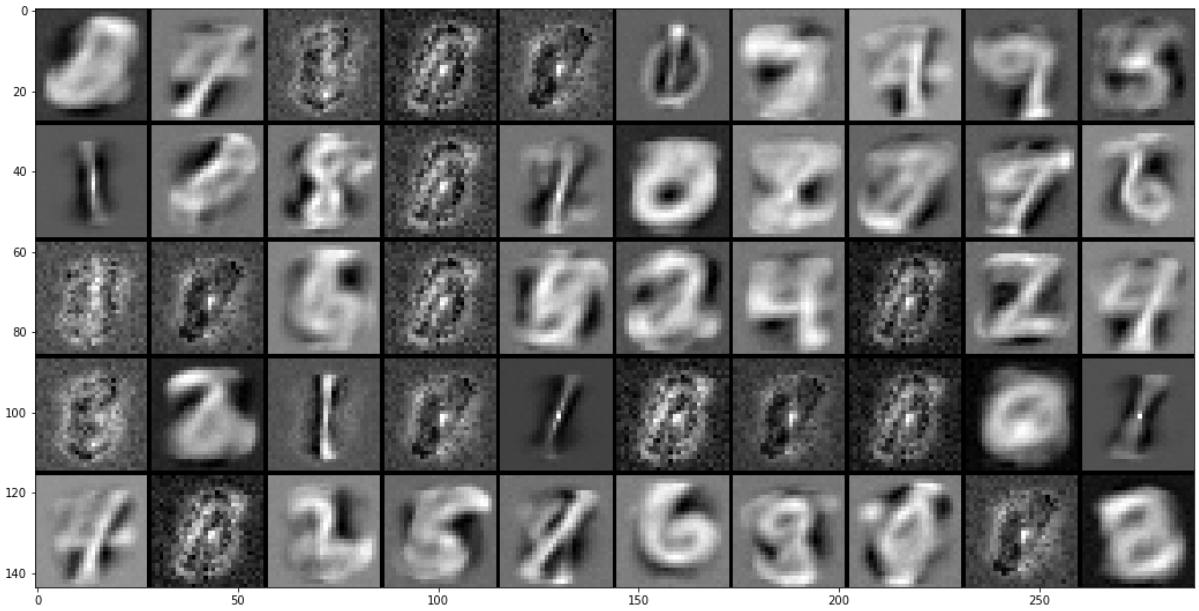
Learned features

We can take each hidden unit and visualize the connections between that hidden unit and each element in the input vector. In our case, we have 50 hidden units. Lets visualize those.

Let's plot the current weights: **tile_raster_images** helps in generating an easy to grasp image from a set of samples or weights. It transform the **uw** (with one flattened image per row of size 784), into an array (of size 25×20) in which images are reshaped and laid out like tiles on a floor.

In [22]:

```
tile_raster_images(X=cur_w.T, img_shape=(28, 28), tile_shape=(5, 10), tile_spacing=
(1, 1))
import matplotlib.pyplot as plt
from PIL import Image
%matplotlib inline
image = Image.fromarray(tile_raster_images(X=cur_w.T, img_shape=(28, 28) ,tile_shap
e=(5, 10), tile_spacing=(1, 1)))
### Plot image
plt.rcParams['figure.figsize'] = (18.0, 18.0)
imgplot = plt.imshow(image)
imgplot.set_cmap('gray')
```

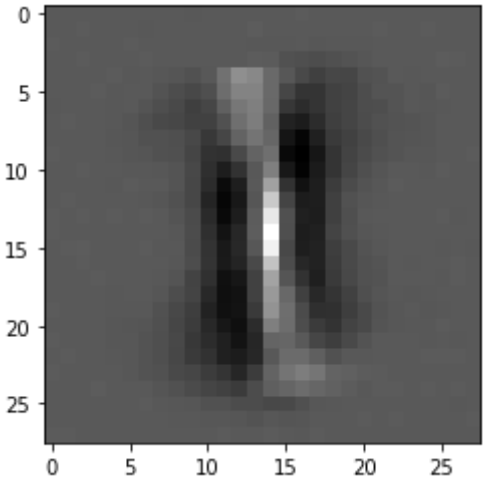


Each tile in the above visualization corresponds to a vector of connections between a hidden unit and visible layer's units.

Let's look at one of the learned weights corresponding to one of hidden units for example. In this particular square, the gray color represents weight = 0, and the whiter it is, the more positive the weights are (closer to 1). Conversely, the darker pixels are, the more negative the weights. The positive pixels will increase the probability of activation in hidden units (after multiplying by input/visible pixels), and negative pixels will decrease the probability of a unit hidden to be 1 (activated). So, why is this important? So we can see that this specific square (hidden unit) can detect a feature (e.g. a "/" shape) and if it exists in the input.

In [23]:

```
from PIL import Image
image = Image.fromarray(tile_raster_images(X =cur_w.T[10:11], img_shape=(28, 28),ti
le_shape=(1, 1), tile_spacing=(1, 1)))
### Plot image
plt.rcParams['figure.figsize'] = (4.0, 4.0)
imgplot = plt.imshow(image)
imgplot.set_cmap('gray')
```



Let's look at the reconstruction of an image now. Imagine that we have a destructed image of figure 3. Lets see if our trained network can fix it:

First we plot the image:

In [24]:

```
!wget -O destructed3.jpg  https://ibm.box.com/shared/static/vvm1b63uvuxq88vbw9znpwu
5ol380mco.jpg
img = Image.open('destructed3.jpg')
img
```

```
--2020-06-08 07:03:03--  https://ibm.box.com/shared/static/vvm1b63uvuxq
88vbw9znpwu5ol380mco.jpg
Resolving ibm.box.com (ibm.box.com)... 107.152.27.197
Connecting to ibm.box.com (ibm.box.com)|107.152.27.197|:443... connecte
d.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: /public/static/vvm1b63uvuxq88vbw9znpwu5ol380mco.jpg [followin
g]
--2020-06-08 07:03:05--  https://ibm.box.com/public/static/vvm1b63uvuxq
88vbw9znpwu5ol380mco.jpg
Reusing existing connection to ibm.box.com:443.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://ibm.ent.box.com/public/static/vvm1b63uvuxq88vbw9znpwu
5ol380mco.jpg [following]
--2020-06-08 07:03:05--  https://ibm.ent.box.com/public/static/vvm1b63u
vuxq88vbw9znpwu5ol380mco.jpg
Resolving ibm.ent.box.com (ibm.ent.box.com)... 107.152.29.201
Connecting to ibm.ent.box.com (ibm.ent.box.com)|107.152.29.201|:443...
connected.
HTTP request sent, awaiting response... 302 Found
Location: https://public.boxcloud.com/d/1/b1!wWJo31A5D_gpw4psDb80AZPREt
87DcrbDUVe4P8bm-TkQq4jSdFye8qg-6nWaHeyE8zD_T9tM7fy4yNLTS1esVGM5HLImA9nE
4wcQXVC7ZEmTmPLAdbFSd0p4pjGp-dRzForIssFN3-rV19RkGfQM41xeh8tn6uCh87aKwyi
fuK0p3Cu6-lLA8VQw_Cvkkzait38ErwbTQX7hhXMnsMOkFnJAvM4oZ7XxWml_oazIb9BXr9
pDiP672CxDPCvQW9MRBOErUc20ko3LqlbplQVl63WQZXdNrpzXKERmK_dO4eZ4PTnFbVOmg
fA60ZZzgq0gp08cO6wRcYK9Z7apuTmDFgkAGx9TaliRr2jJFdQ3TAZ189knG7IjyKKLxuOM
Ow8mHHu6Ue510Y_q7eNgkfTRXB15CrGUeBBtpnVIqaq6QcDip_3Rl6nti9wL0asz5QBKchy
zUpsxBQcsRsr8DI6pbt_QUVCGG1TtPj_bhi0iejCnNGdGLLCbzsQ19nOhIgBveD6whd-VKp
a9hWUvbCKNl6bz1f6frSnreMauEjDMQVZDAp0ESxbE9GCHlrKLxrOAeToi6VJx_bE516ddO
QpeCdFuxx3ocDDt6qq0q-3hj1qNYuJrsNVMf8coriZ2m4vCYcOXtZAWT6eTZtkpkwhl8Xc-
eXr_vnLf2BkM0qxWbOf6YNinel5lYtEfvsJOhZ247zFFz3eCHACSLYRd5vEoW6RqbaAV3fF
ayv7ugyWIRRbMiAqKrH9OgjIYjirGQHMfHvCz4kraS2HsDW3oNmVVhwGwo1vJ6rOp57WeFo
JVtUQ27cjSO2l8qIe1BRS7DeD9fDi-QvFbc70gyyGRUBDvvoQWvdYbAE7CaYwjfpEFppqag
fo0qE-Dg0-9y1WrSs16_T6e2n3JT_pI_KWAwHPcfSSg6t5IScIK1uIvA6_xsb39o14qbdcD
Pnaa-ogvKhA6-UI0U_IMQ6N5ydN7ZXaW3NX84LjyFQ9pMvdF7kwHjr2joPi0BIIBurwZYJM
Nfy8e-Vyc9rlvYop_u18eky2pFB7RYL4R-COUQpHWLW2wL1Z18Botdsoq8FgG5563LvHweW
d7vRIGFw-k_yfXMEagOwec59AUrRy9nU8Q5273OEfimYNuuX9L07UA_Ch1eWjak4_3VydzM
K4TJpsbzlFuPJoULW0XjjS7cncMUn6FL3cRTOwF5jcpZFsCdjOmxWpSw1EKcz28p9qmxjaw
2SauMKSAEaIjHiwBE-pPr09Xw6B-HhOkyGmEmE2a_esUjHz_ba0Nd0zgwhXWFfAecxJWoCj
SOxi4S-Xl60pMT8RarpOb_SUCOJ1YCbxZW6v4Fd2JBhGY9JFfLaI871cR64Oi9pRmSAJTsQ
zLJmxzEW32gN0n8vJ0G3UlCbzVVksjMa7XBPbj7Cv-h0uGcU4gpob687yyCvTzA../downl
oad [following]
--2020-06-08 07:03:06--  https://public.boxcloud.com/d/1/b1!wWJo31A5D_g
pw4psDb80AZPREt87DcrbDUVe4P8bm-TkQq4jSdFye8qg-6nWaHeyE8zD_T9tM7fy4yNLTS
1esVGM5HLImA9nE4wcQXVC7ZEmTmPLAdbFSd0p4pjGp-dRzForIssFN3-rV19RkGfQM41xe
h8tn6uCh87aKwyifuK0p3Cu6-lLA8VQw_Cvkkzait38ErwbTQX7hhXMnsMOkFnJAvM4oZ7X
xWml_oazIb9BXr9pDiP672CxDPCvQW9MRBOErUc20ko3LqlbplQVl63WQZXdNrpzXKERmK_
dO4eZ4PTnFbVOmgfA60ZZzgq0gp08cO6wRcYK9Z7apuTmDFgkAGx9TaliRr2jJFdQ3TAZ18
9knG7IjyKKLxuOMOw8mHHu6Ue510Y_q7eNgkfTRXB15CrGUeBBtpnVIqaq6QcDip_3Rl6nt
i9wL0asz5QBKchyzUpsxBQcsRsr8DI6pbt_QUVCGG1TtPj_bhi0iejCnNGdGLLCbzsQ19nO
hIgBveD6whd-VKpa9hWUvbCKNl6bz1f6frSnreMauEjDMQVZDAp0ESxbE9GCHlrKLxrOAeT
oi6VJx_bE516ddOQpeCdFuxx3ocDDt6qq0q-3hj1qNYuJrsNVMf8coriZ2m4vCYcOXtZAWT
6eTZtkpkwhl8Xc-eXr_vnLf2BkM0qxWbOf6YNinel5lYtEfvsJOhZ247zFFz3eCHACSLYRd
5vEoW6RqbaAV3fFayv7ugyWIRRbMiAqKrH9OgjIYjirGQHMfHvCz4kraS2HsDW3oNmVVhwG
wo1vJ6rOp57WeFoJVtUQ27cjSO2l8qIe1BRS7DeD9fDi-QvFbc70gyyGRUBDvvoQWvdYbAE
7CaYwjfpEFppqagfo0qE-Dg0-9y1WrSs16_T6e2n3JT_pI_KWAwHPcfSSg6t5IScIK1uIvA
6_xsb39o14qbdcDPnaa-ogvKhA6-UI0U_IMQ6N5ydN7ZXaW3NX84LjyFQ9pMvdF7kwHjr2j
```

```
oPi0BIIBurwZYJMNfy8e-Vyc9rlvYop_u18eky2pFB7RYL4R-COUQpHWLW2wL1Z18Botdso
q8FgG5563LvHweWd7vRIGFw-k_yfXMEagOwec59AUrRy9nU8Q5273OEfimYNuuX9L07UA_C
h1eWjak4_3VydzMK4TJpsbzlFuPJoULW0XjjS7cncMUn6FL3cRTOwF5jcpZFsCdjOmxWpSw
1EKcz28p9qmxjaw2SauMKSAEaIjHiwBE-pPr09Xw6B-HhOkyGmEmE2a_esUjHz_ba0Nd0zg
whXWFfAecxJWoCjSOxi4S-Xl60pMT8RarpOb_SUCOJ1YCbxZW6v4Fd2JBhGY9JFfLaI871c
R64Oi9pRmSAJTsQzLJmxzEW32gN0n8vJ0G3UlCbzVVksjMa7XBPbj7Cv-h0uGcU4gpob687
yyCvTzA../download
Resolving public.boxcloud.com (public.boxcloud.com)... 107.152.27.200
Connecting to public.boxcloud.com (public.boxcloud.com)|107.152.27.200
|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 24383 (24K) [image/jpeg]
Saving to: 'destructed3.jpg'

destructed3.jpg     100%[===================>]  23.81K  --.-KB/s    in
0.002s

2020-06-08 07:03:06 (11.0 MB/s) - 'destructed3.jpg' saved [24383/24383]
```

Out[24]:

Now let's pass this image through the net:

In [25]:

```
# convert the image to a 1d numpy array
sample_case = np.array(img.convert('I').resize((28,28))).ravel().reshape((1, -1))/2
55.0
```

Feed the sample case into the network and reconstruct the output:

In [26]:

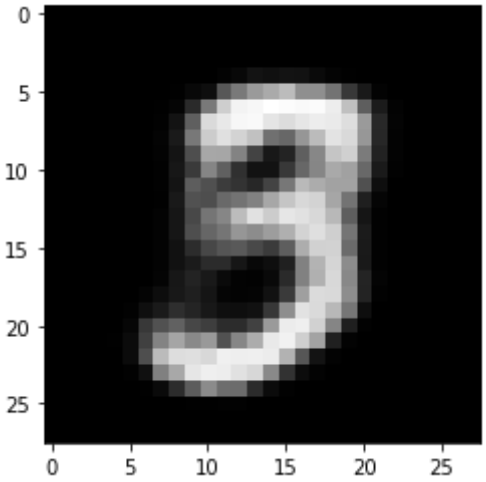
```
hh0_p = tf.nn.sigmoid(tf.matmul(v0_state, W) + hb)
#hh0_s = tf.nn.relu(tf.sign(hh0_p - tf.random_uniform(tf.shape(hh0_p))))
hh0_s = tf.round(hh0_p)
hh0_p_val,hh0_s_val  = sess.run((hh0_p, hh0_s), feed_dict={ v0_state: sample_case,
W: prv_w, hb: prv_hb})
print("Probability nodes in hidden layer:" ,hh0_p_val)
print("activated nodes in hidden layer:" ,hh0_s_val)

# reconstruct
vv1_p = tf.nn.sigmoid(tf.matmul(hh0_s_val, tf.transpose(W)) + vb)
rec_prob = sess.run(vv1_p, feed_dict={ hh0_s: hh0_s_val, W: prv_w, vb: prv_vb})
```

```
Probability nodes in hidden layer: [[3.99175823e-01 4.05607160e-23 1.11
621774e-10 1.03047615e-09
  3.54016426e-12 7.47097772e-07 1.10656746e-19 7.24108553e-29
  1.21747941e-27 9.99989271e-01 0.00000000e+00 0.00000000e+00
  7.89659102e-11 1.79784812e-10 1.79915552e-32 9.80336726e-01
  1.25709889e-07 1.00000000e+00 1.61878633e-25 0.00000000e+00
  2.89394708e-09 4.23358874e-12 8.67023474e-32 1.63434807e-10
  9.99112189e-01 9.99999642e-01 2.78181914e-31 2.61151918e-08
  5.66386712e-17 2.21260279e-24 2.22793485e-11 2.65122187e-24
  8.08389952e-11 7.75331604e-11 1.45123502e-35 4.19289456e-08
  1.95886519e-11 6.45946727e-08 9.99606311e-01 1.76879684e-33
  8.42687843e-25 4.46283615e-08 4.31353801e-36 9.40166598e-22
  2.32222858e-34 0.00000000e+00 3.88985155e-09 3.26782283e-15
  1.04465880e-10 7.13618100e-01]]
activated nodes in hidden layer: [[0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.
0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0.
  1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.
0.
  0. 1.]]
```

Here we plot the reconstructed image:

In [27]:

```
img = Image.fromarray(tile_raster_images(X=rec_prob, img_shape=(28, 28),tile_shape=
(1, 1), tile_spacing=(1, 1)))
plt.rcParams['figure.figsize'] = (4.0, 4.0)
imgplot = plt.imshow(img)
imgplot.set_cmap('gray')
```



Want to learn more?

Running deep learning programs usually needs a high performance platform. **PowerAI** speeds up deep learning and AI. Built on IBM's Power Systems, **PowerAI** is a scalable software platform that accelerates deep learning and AI with blazing performance for individual users or enterprises. The **PowerAI** platform supports popular machine learning libraries and dependencies including TensorFlow, Caffe, Torch, and Theano. You can use PowerAI on IMB Cloud (https://cocl.us/ML0120EN_PAI).

Also, you can use **Watson Studio** to run these notebooks faster with bigger datasets.**Watson Studio** is IBM's leading cloud solution for data scientists, built by data scientists. With Jupyter notebooks, RStudio, Apache Spark and popular libraries pre-packaged in the cloud, **Watson Studio** enables data scientists to collaborate on their projects without having to install anything. Join the fast-growing community of **Watson Studio** users today with a free account at Watson Studio (https://cocl.us/ML0120EN_DSX).This is the end of this lesson. Thank you for reading this notebook, and good luck on your studies.

Thanks for completing this lesson!

Notebook created by: Saeed Aghabozorgi (https://ca.linkedin.com/in/saeedaghabozorgi)

References:

https://en.wikipedia.org/wiki/Restricted_Boltzmann_machine
(https://en.wikipedia.org/wiki/Restricted_Boltzmann_machine)
http://deeplearning.net/tutorial/rbm.html (http://deeplearning.net/tutorial/rbm.html)
http://www.cs.utoronto.ca/~hinton/absps/netflixICML.pdf
(http://www.cs.utoronto.ca/~hinton/absps/netflixICML.pdf)
http://imonad.com/rbm/restricted-boltzmann-machine/ (http://imonad.com/rbm/restricted-boltzmann-machine/)
