IBM Developer
SKILLS NETWORK

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?

An RBM are a basic form of autoencoder. 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 an RBM?

RBM is useful for Collaborative Filtering, 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 where 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, or 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 carry out both supervised 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 that a car is an SUV, and $y = 1$ indicates that a car is a sedan, then $p(x|y = 0)$ models the distribution of SUV features, and $p(x|y = 1)$ models the distribution of sedan 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.

[ ]:
```
import urllib.request
with urllib.request.urlopen("https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DL0120EN-SkillsNetwork/labs/Week4/data/utils.py") as url:
    response = url.read()
target = open('utils.py', 'w')
target.write(response.decode('utf-8'))
target.close()
```

Installing TensorFlow

We will installing TensorFlow version 2.2.0 and its required prerequistes. Also installing pillow...

[ ]:
```
!pip install grpcio==1.24.3
!pip install tensorflow==2.2.0
!pip install pillow
```

Notice: This notebook has been created with TensorFlow version 2.2, and might not work with other versions. Therefore we check:

[ ]:
```
import tensorflow as tf
from IPython.display import Markdown, display

def printmd(string):
    display(Markdown('# <span style="color:red">'+string+'</span>'))


if not tf.__version__ == '2.2.0':
    printmd('<<<<<!!!!! ERROR !!!! please upgrade to TensorFlow 2.2.0, or restart your Kernel (Kernel->Restart & Clear Output)>>>>>')
```

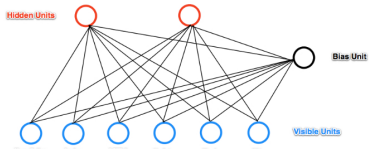
Now, we load in all the packages that we use to create the net including the TensorFlow package:

[ ]:
```
import tensorflow as tf
import numpy as np

from PIL import Image
from utils import tile_raster_images
import matplotlib.pyplot as plt
%matplotlib inline
```

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 $V = 7$ input nodes. The second layer is the hidden layer, which has H neurons in our case. Each hidden node takes on values of either 0 or 1 (i.e., $h_i = 1$ or h_i = 0), with a probability that is a logistic function of the inputs it receives from the other V visible units, called for example, $p(h_i = 1)$. For our toy sample, we'll use 2 nodes in the hidden layer, so $H = 2$.



Each node in the first layer also has a bias. We will denote the bias as v_{bias}, and this single value is shared among the V visible units.

The bias of the second is defined similarly as h_{bias}, and this single value among the H hidden units.

[ ]:
```
v_bias = tf.Variable(tf.zeros([7]), tf.float32)
h_bias = tf.Variable(tf.zeros([2]), tf.float32)
```

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. We define a tensor $\mathbf{W}$ of shape = (7,2), where the number of visible neurons = 7, and the number of hidden neurons = 2.

[ ]:
```
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 of 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, where one node encodes the weight and, and the other encodes the size. In a sense, the different configurations represent different cars, where one is an SUV and the other is Sedan. In a training process, through many forward and backward passes, the 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 an 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.

The Inference Process

RBM has two phases:

- Forward Pass
- Backward Pass or Reconstruction

Phase 1) Forward pass:

Input one training sample (one image) $\mathbf{x}$ 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). First, the probability vector is computed using the input feature vector $\mathbf{x}$, the weight matrix $\mathbf{W}$, and the bias term h_{bias}, as

$$p(h_j|\mathbf{x}) = \sigma(\sum_{i=1}^{V} W_{ij} x_i + h_{bias}),$$

where $\sigma(z) = (1 + e^{-z})^{-1}$ is the logistic function.

So, what does $p(h_j)$ represent? It is the probability distribution of the hidden units. That is, RBM uses inputs x_i to make predictions about hidden node activations. For example, imagine that the hidden node activation values are [0.51 0.84] for the first training item. It tells you that the conditional probability for each hidden neuron for Phase 1 is:

$$p(h_1 = 1|\mathbf{v}) = 0.51$$
$$p(h_2 = 1|\mathbf{v}) = 0.84$$

As a result, for each row in the training set, vector of probabilities is generated. In TensorFlow, this is referred to as a `tensor` with a shape of (1,2).

We then turn unit j with probability $p(h_j|\mathbf{v})$, and turn it off with probability $1 - p(h_j|\mathbf{v})$ by generating a uniform random number vector ξ, and comparing it to the activation probability as

$$\text{If } \xi_j > p(h_j|\mathbf{v}), \text{ then } h_j = 1, \text{ else } h_j = 0.$$

Therefore, the conditional probability of a configuration of $\mathbf{h}$ given $\mathbf{v}$ (for a training sample) is:

$$p(\mathbf{h} \mid \mathbf{v}) = \prod_{j=1}^{H} p(h_j \mid \mathbf{v})$$

where H is the number of hidden units.

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]. Let's see what the output of forward pass would look like:

[ ]:
```
X = tf.constant([[1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]], tf.float32)

v_state = X
print ("Input: ", v_state)
```

```
h_bias = tf.constant([0.1, 0.1])
print ("hb: ", h_bias)
print ("w: ", 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 hidden units
print ("p(h|v): ", 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:", h_state)
```

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. $\mathbf{h}$) becomes the input in the backward pass. The same weight matrix and visible layer biases are used to passed to the sigmoid function. The reproduced output is a reconstruction which is an approximation of the original input.

[ ]:
```
vb = tf.constant([0.1, 0.2, 0.1, 0.1, 0.1, 0.2, 0.1])
print ("b: ", vb)
v_prob = 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: ", v_state)
```

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

$$p(\mathbf{v} \mid \mathbf{h}) = \prod_{i=1}^{V} p(v_i \mid \mathbf{h}),$$

where,

$$p(v_i \mid \mathbf{h}) = \sigma\left(\sum_{j=1}^{H} W_{ji} h_j + v_{bias}\right)$$

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?

[ ]:
```
inp = X
print("input X:" , inp.numpy())

print("probablity vector:" , v_prob[0].numpy())
v_probability = 1

for elm, p in zip(inp[0],v_prob[0]) :
    if elm ==1:
        v_probability *= p
    else:
        v_probability *= (1-p)

print("probability of generating X: " , v_probability.numpy())
```

How similar are vectors $\mathbf{x}$ and $\mathbf{v}$? Of course, the reconstructed values most likely will not look anything like the input vector, because our network has not been 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.

[ ]:
```
#Loading training and test data
mnist = tf.keras.datasets.mnist
(trX, trY), (teX, teY) = mnist.load_data()

# showing an example of the Flatten class and operation
from tensorflow.keras.layers import Flatten
flatten = Flatten(dtype='float32')
trX = flatten(trX/255.0)
trY = flatten(trY/255.0)
```

Lets look at the dimension of the images.

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.

[ ]:
```
vb = tf.Variable(tf.zeros([784]), tf.float32)
hb = tf.Variable(tf.zeros([50]), tf.float32)
```

Let $\mathbf{W}$ be the Tensor of 784x50 (784 - number of visible neurons, 50 - number of hidden neurons) that represents weights between the neurons.

[ ]:
```
W = tf.Variable(tf.zeros([784,50]), tf.float32)
```

Lets define the visible layer:

[ ]:
```
v0_state = tf.Variable(tf.zeros([784]), tf.float32)

#testing to see if the matrix product works
tf.matmul( [v0_state], W)
```

Now, we can define hidden layer:

[ ]:
```
#computing the hidden nodes probability vector and checking shape
h0_prob = tf.nn.sigmoid(tf.matmul([v0_state], W) + hb)  #probabilities of the hidden units
print("h0_state shape: " , tf.shape(h0_prob))

#defining a function to return only the generated hidden states
def hidden_layer(v0_state, W, hb):
    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)))) #sample_h_given_X
    return h0_state


h0_state = hidden_layer(v0_state, W, hb)
print("first 15 hidden states: ", h0_state[0][0:15])
```

Now, we define reconstruction part:

[ ]:
```
def reconstructed_output(h0_state, W, vb):
    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)))) #sample_v_given_h
    return v1_state[0]

v1_state = reconstructed_output(h0_state, W, vb)
print("hidden state shape: ", h0_state.shape)
print("v0 state shape:  ", v0_state.shape)
print("v1 state shape:  ", v1_state.shape)
```

What is the 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.

[ ]:
```
def error(v0_state, v1_state):
    return tf.reduce_mean(tf.square(v0_state - v1_state))

err = tf.reduce_mean(tf.square(v0_state - v1_state))
print("error", err.numpy())
```

Training the Model

Warning... The following part is math-heavy, but you can skip it if you just want to run the cells in the next section.
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 $\mathbf{v}$ (images) in the training set $\mathbf{V}$ (a matrix, where each row of it is treated as a visible vector $\mathbf{v}$)

$$\arg\max_{W} \prod_{\mathbf{v} \in \mathbf{V}_T} p(\mathbf{v}),$$

which is equivalent to maximizing the expectation of the log probability, given as

$$\arg\max_{W} \left[\mathbb{E} \left(\prod_{\mathbf{v} \in \mathbf{V}} \log\left(p(\mathbf{v})\right) \right) \right].$$

So, we have to update the weights W_{ij} to increase $p(\mathbf{v})$ for all $\mathbf{v}$ 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

Gibbs Sampling Step 1

Given an input vector $\mathbf{v}$, we are using $p(\mathbf{h}|\mathbf{v})$ to predict the hidden values $\mathbf{h}$.

$$p(h_j|\mathbf{v}) = \sigma\left(\sum_{i=1}^{V} W_{ij} v_i + h_{bias}\right)$$

The samples are generated from this distribution by generating the uniform random variate vector $\xi \sim U[0,1]$ of length H and comparing to the computed probabilities as

$$\text{If } \xi_j > p(h_j|\mathbf{v}), \text{ then } h_j = 1, \text{ else } h_j = 0.$$

Gibbs Sampling Step 2

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

$$p(v_i|\mathbf{h}) = \sigma\left(\sum_{j=1}^{H} W_{ij}^T h_j + v_{bias}\right)$$

The samples are generated from this distribution by generating a uniform random variate vector $\xi \sim U[0,1]$ of length V and comparing to the computed probabilities as

$$\text{If } \xi_i > p(v_i|\mathbf{h}), \text{ then } v_i = 1, \text{ else } v_i = 0.$$

Let vectors $\mathbf{v}_k$ and $\mathbf{h}_k$ be for the kth iteration. In general, the kth state is generrated as:

Iteration k:

$$\mathbf{v}_{k-1} \Rightarrow p(\mathbf{h}_{k-1}|\mathbf{v}_{k-1}) \Rightarrow \mathbf{h}_{k-1} \Rightarrow p(\mathbf{v}_k|\mathbf{h}_{k-1}) \Rightarrow \mathbf{v}_k$$

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:

$$\Delta \mathbf{W}_k = \mathbf{v}_k \otimes \mathbf{h}_k - \mathbf{v}_{k-1} \otimes \mathbf{h}_{k-1}$$

Contrastive Divergence is actually matrix of values that is computed and used to adjust values of the $\mathbf{W}$ matrix. Changing $\mathbf{W}$ incrementally leads to training of the $\mathbf{W}$ values. Then, on each step (epoch), $\mathbf{W}$ is updated using the following:

$$\mathbf{W}_k = \mathbf{W}_{k-1} + \alpha * \Delta \mathbf{W}_k$$

Reconstruction steps:

- Get one data point from data set, like x, and pass it through the following steps:
  Iteration $k = 1$: Sampling (starting with input image)

  $$\mathbf{x} = \mathbf{v}_0 \Rightarrow p(\mathbf{h}_0|\mathbf{v}_0) \Rightarrow \mathbf{h}_0 \Rightarrow p(\mathbf{v}_1|\mathbf{h}_0) \Rightarrow \mathbf{v}_1$$

  followed by the CD-k step

  $$\Delta \mathbf{W}_1 = \mathbf{v}_1 \otimes \mathbf{h}_1 - \mathbf{v}_0 \otimes \mathbf{h}_0$$
  $$\mathbf{W}_1 = \mathbf{W}_0 + \alpha * \Delta \mathbf{W}_1$$

- $\mathbf{v}_1$ is the reconstruction of $\mathbf{x}$ sent to the next iteration).

Iteration $k = 2$:

Sampling (starting with $\mathbf{v}_1$)

$$\mathbf{v}_1 \Rightarrow p(\mathbf{h}_1|\mathbf{v}_1) \Rightarrow \mathbf{h}_1 \Rightarrow p(\mathbf{v}_2|\mathbf{h}_1) \Rightarrow \mathbf{v}_2$$

followed by the CD-k step

$$\Delta \mathbf{W}_2 = \mathbf{v}_2 \otimes \mathbf{h}_2 - \mathbf{v}_1 \otimes \mathbf{h}_1$$

$$\mathbf{W}_2 = \mathbf{W}_1 + \alpha * \Delta \mathbf{W}_2$$

- $\mathbf{v}_2$ is the reconstruction of $\mathbf{v}_1$ sent to the next iteration).
Iteration $k = K$: Sampling (starting with $\mathbf{v}_{K-1}$)

$$\mathbf{v}_{K-1} \Rightarrow p(\mathbf{h}_{K-1}|\mathbf{v}_{K-1}) \Rightarrow \mathbf{h}_{K-1} \Rightarrow p(\mathbf{v}_K|\mathbf{h}_{K-1}) \Rightarrow \mathbf{v}_K$$

followed by the CD-k step

$$\Delta \mathbf{W}_K = \mathbf{v}_K \otimes \mathbf{h}_K - \mathbf{v}_{K-1} \otimes \mathbf{h}_{K-1}$$

$$\mathbf{W}_K = \mathbf{W}_{K-1} + \alpha * \Delta \mathbf{W}_K$$

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

K is adjustable, and good performance can be achieved with $K = 1$, so that we just take one set of sampling steps per image.

[ ]:
```
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)))) #sample_h_given_X
```

Lets look at the error of the first run:

[ ]:
```
print("error: ", error(v0_state, v1_state))
```

[ ]:
```
#Parameters
alpha = 0.01
epochs = 1
batchsize = 200
weights = []
errors = []
batch_number = 0
K = 1

#creating datasets
train_ds = \
    tf.data.Dataset.from_tensor_slices((trX, trY)).batch(batchsize)

for epoch in range(epochs):
    for batch_x, batch_y in train_ds:
        batch_number += 1
        for i_sample in range(batchsize):----------
            for k in range(K):
                v0_state = batch_x[i_sample]
                h0_state = hidden_layer(v0_state, W, hb)
                v1_state = reconstructed_output(h0_state, W, vb)
                h1_state = hidden_layer(v1_state, W, hb)

                delta_W = tf.matmul(tf.transpose([v0_state]), h0_state) - tf.matmul(tf.transpose([v1_state]), h1_state)
                W = W + alpha * delta_W

                vb = vb + alpha * tf.reduce_mean(v0_state - v1_state, 0)
                hb = hb + alpha * tf.reduce_mean(h0_state - h1_state, 0)

                v0_state = v1_state

            if i_sample == batchsize-1:
                err = error(batch_x[i_sample], v1_state)
                errors.append(err)
                weights.append(W)
                print ( 'Epoch: %d' % epoch,
                       "batch #: %i " % batch_number, "of %i" % int(60e3/batchsize),
                       "sample #: %i" % i_sample,
                       'reconstruction error: %f' % err)
```

Let's take a look at the errors at the end of each batch:

[ ]:
```
plt.plot(errors)
plt.xlabel("Batch Number")
plt.ylabel("Error")
plt.show()
```

What is the final weight matrix W after training?

[ ]:
```
print(W.numpy()) # a weight matrix of shape (50,784)
```

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 transforms the uw (with one flattened image per row of size 784), into an array (of size 28×28) in which images are reshaped and laid out like tiles on a floor.

[ ]:
```
tile_raster_images(X=W.numpy().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=W.numpy().T, img_shape=(28, 28) ,tile_shape=(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.

```
[ ]: from PIL import Image
image = Image.fromarray(tile_raster_images(X =W.numpy().T[10:11], img_shape=(28, 28),tile_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:

```
[ ]: !wget -O destructed3.jpg  https://ibm.box.com/shared/static/vvm1b63uvuxq88vbw9znpwu5ol380mco.jpg
img = Image.open('destructed3.jpg')
img
```

Now let's pass this image through the neural net:

```
[ ]: # convert the image to a 1d numpy array
sample_case = np.array(img.convert('I').resize((28,28)).ravel().reshape((1, -1))/255.0

sample_case = tf.cast(sample_case, dtype=tf.float32)
```

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

```
[ ]: hh0_p = tf.nn.sigmoid(tf.matmul(sample_case, W) + hb)
hh0_s = tf.round(hh0_p)

print("Probability nodes in hidden layer:" ,hh0_p)
print("activated nodes in hidden layer:" ,hh0_s)

# reconstruct
vv1_p = tf.nn.sigmoid(tf.matmul(hh0_s, tf.transpose(W)) + vb)

print(vv1_p)
#rec_prob = sess.run(vv1_p, feed_dict={ hh0_s: hh0_s_val, W: prv_w, vb: prv_vb})
```

Here we plot the reconstructed image:

```
[ ]: img = Image.fromarray(tile_raster_images(X=vv1_p.numpy(), 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.

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.This is the end of this lesson. Thank you for reading this notebook, and good luck on your studies.

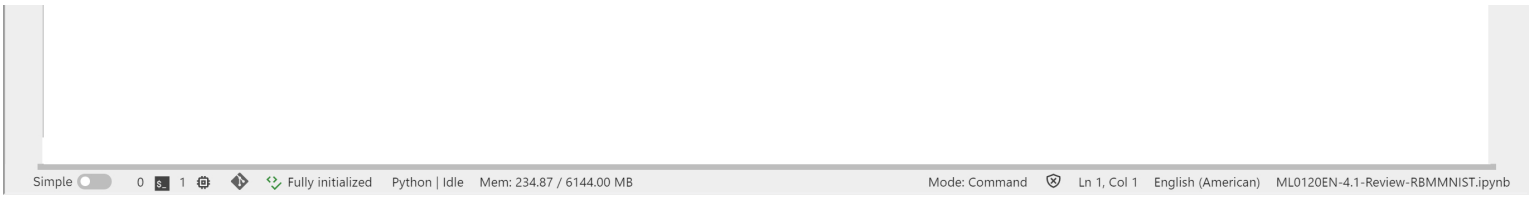
Thanks for completing this lesson!

Notebook created by: Saeed Aghabozorgi

Updated to TF 2.X by Jerome Nilmeier

References:

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


Simple
0
1
Fully initialized
Python | Idle
Mem: 234.87 / 6144.00 MB
Mode: Command
Ln 1, Col 1
English (American)
ML0120EN-4.1-Review-RBMMNIST.ipynb