Restricted Boltzmann Machines (RBM)

Note

This section assumes the reader has already read through Classifying MNIST digits using Logistic Regression and Multilayer Perceptron. Additionally it uses the following Theano functions and concepts: T.tanh, shared variables, basic arithmetic ops, T.grad, Random numbers, floatX and scan. If you intend to run the code on GPU also read GPU.

Note

The code for this section is available for download here.

Energy-Based Models (EBM)

Energy-based models associate a scalar energy to each configuration of the variables of interest. Learning corresponds to modifying that energy function so that its shape has desirable properties. For example, we would like plausible or desirable configurations to have low energy. Energy-based probabilistic models define a probability distribution through an energy function, as follows:

$$p(x) = \frac{e^{-E(x)}}{Z}. \tag{1}$$

The normalizing factor Z is called the **partition function** by analogy with physical systems.

$$Z = \sum_x e^{-E(x)}$$

An energy-based model can be learnt by performing (stochastic) gradient descent on the empirical negative log-likelihood of the training data. As for the logistic regression we will first define the log-likelihood and then the loss function as being the negative log-likelihood.

$$\mathcal{L}(\theta, \mathcal{D}) = \frac{1}{N} \sum_{x^{(i)} \in \mathcal{D}} \log\ p(x^{(i)})$$
$$\ell(\theta, \mathcal{D}) = -\mathcal{L}(\theta, \mathcal{D})$$

using the stochastic gradient $-\frac{\partial \log p(x^{(i)})}{\partial \theta}$, where θ are the parameters of the model.

EBMs with Hidden Units

In many cases of interest, we do not observe the example x fully, or we want to introduce some non-observed variables to increase the expressive power of the model. So we consider an observed part (still denoted x here) and a **hidden** part h. We can then write:

$$P(x) = \sum_h P(x, h) = \sum_h \frac{e^{-E(x,h)}}{Z}. \tag{2}$$

In such cases, to map this formulation to one similar to Eq. (1), we introduce the notation (inspired from physics) of **free energy**, defined as follows:

$$\mathcal{F}(x) = -\log \sum_h e^{-E(x,h)} \tag{3}$$

which allows us to write,

$$P(x) = \frac{e^{-\mathcal{F}(x)}}{Z} \text{ with } Z = \sum_x e^{-\mathcal{F}(x)}.$$

The data negative log-likelihood gradient then has a particularly interesting form.

$$-\frac{\partial \log p(x)}{\partial \theta} = \frac{\partial \mathcal{F}(x)}{\partial \theta} - \sum_{\tilde{x}} p(\tilde{x}) \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta}. \tag{4}$$

Notice that the above gradient contains two terms, which are referred to as the **positive** and **negative phase**. The terms positive and negative do not refer to the sign of each term in the equation, but rather reflect their effect on the probability density defined by the model. The first term increases the probability of training data (by reducing the corresponding free energy), while the second term decreases the probability of samples generated by the model.

It is usually difficult to determine this gradient analytically, as it involves the computation of $E_P[\frac{\partial \mathcal{F}(x)}{\partial \theta}]$. This is nothing less than an expectation over all possible configurations of the input x (under the distribution P formed by the model) !

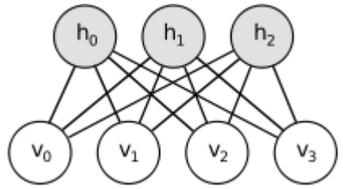
The first step in making this computation tractable is to estimate the expectation using a fixed number of model samples. Samples used to estimate the negative phase gradient are referred to as **negative particles**, which are denoted as $\mathcal{N}$. The gradient can then be written as:

$$-\frac{\partial \log p(x)}{\partial \theta} \approx \frac{\partial \mathcal{F}(x)}{\partial \theta} - \frac{1}{|\mathcal{N}|} \sum_{\tilde{x} \in \mathcal{N}} \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta}. \tag{5}$$

where we would ideally like elements $\tilde{x}$ of $\mathcal{N}$ to be sampled according to P (i.e. we are doing Monte-Carlo). With the above formula, we almost have a pratical, stochastic algorithm for learning an EBM. The only missing ingredient is how to extract these negative particles $\mathcal{N}$. While the statistical literature abounds with sampling methods, Markov Chain Monte Carlo methods are especially well suited for models such as the Restricted Boltzmann Machines (RBM), a specific type of EBM.

Restricted Boltzmann Machines (RBM)

Boltzmann Machines (BMs) are a particular form of log-linear Markov Random Field (MRF), i.e., for which the energy function is linear in its free parameters. To make them powerful enough to represent complicated distributions (i.e., go from the limited parametric setting to a non-parametric one), we consider that some of the variables are never observed (they are called hidden). By having more hidden variables (also called hidden units), we can increase the modeling capacity of the Boltzmann Machine (BM). Restricted Boltzmann Machines further restrict BMs to those without visible-visible and hidden-hidden connections. A graphical depiction of an RBM is shown below.



The energy function $E(v, h)$ of an RBM is defined as:

$$E(v, h) = -b'v - c'h - h'Wv \tag{6}$$

where W represents the weights connecting hidden and visible units and b, c are the offsets of the visible and hidden layers respectively.

This translates directly to the following free energy formula:

$$\mathcal{F}(v) = -b'v - \sum_i \log \sum_{h_i} e^{h_i (c_i + W_i v)}.$$

Because of the specific structure of RBMs, visible and hidden units are conditionally independent given one-another. Using this property, we can write:

$$p(h|v) = \prod_i p(h_i|v)$$

$$p(v|h) = \prod_j p(v_j|h).$$

RBMs with binary units

In the commonly studied case of using binary units (where v_j and $h_i \in \{0, 1\}$), we obtain from Eq. (6) and (2), a probabilistic version of the usual neuron activation function:

$$P(h_i = 1|v) = sigm(c_i + W_i v) \tag{7}$$

$$P(v_j = 1|h) = sigm(b_j + W'_j h) \tag{8}$$

The free energy of an RBM with binary units further simplifies to:

$$\mathcal{F}(v) = -b'v - \sum_i \log(1 + e^{(c_i + W_i v)}). \tag{9}$$

Update Equations with Binary Units

Combining Eqs. (5) with (9), we obtain the following log-likelihood gradients for an RBM with binary units:

$$\begin{aligned} -\frac{\partial \log p(v)}{\partial W_{ij}} &= E_v[p(h_i|v) \cdot v_j] - v_j^{(i)} \cdot sigm(W_i \cdot v^{(i)} + c_i) \\ -\frac{\partial \log p(v)}{\partial c_i} &= E_v[p(h_i|v)] - sigm(W_i \cdot v^{(i)}) \\ -\frac{\partial \log p(v)}{\partial b_j} &= E_v[p(v_j|h)] - v_j^{(i)} \end{aligned} \tag{10}$$

For a more detailed derivation of these equations, we refer the reader to the following page, or to section 5 of Learning Deep Architectures for AI. We will however not use these formulas, but rather get the gradient using Theano T.grad from equation (4).

Sampling in an RBM

Samples of $p(x)$ can be obtained by running a Markov chain to convergence, using Gibbs sampling as the transition operator.

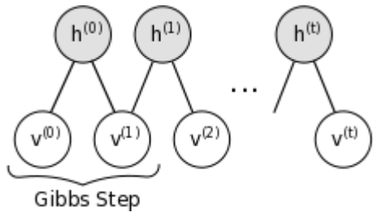
Gibbs sampling of the joint of N random variables $S = (S_1, ..., S_N)$ is done through a sequence of N sampling sub-steps of the form $S_i \sim p(S_i|S_{-i})$ where S_{-i} contains the $N - 1$ other random variables in S excluding S_i.

For RBMs, S consists of the set of visible and hidden units. However, since they are conditionally independent, one can perform block Gibbs sampling. In this setting, visible units are sampled simultaneously given fixed values of the hidden units. Similarly, hidden units are sampled simultaneously given the visibles. A step in the Markov chain is thus taken as follows:

$$\begin{aligned} h^{(n+1)} &\sim sigm(W' v^{(n)} + c) \\ v^{(n+1)} &\sim sigm(W h^{(n+1)} + b), \end{aligned}$$

where $h^{(n)}$ refers to the set of all hidden units at the n-th step of the Markov chain. What it means is that, for example, $h_i^{(n+1)}$ is randomly chosen to be 1 (versus 0) with probability $sigm(W'_i v^{(n)} + c_i)$, and similarly, $v_j^{(n+1)}$ is randomly chosen to be 1 (versus 0) with probability $sigm(W_{.j} h^{(n+1)} + b_j)$.

This can be illustrated graphically:



As $t \to \infty$, samples are guaranteed to be accurate samples of .

In theory, each parameter update in the learning process would require running one such chain to convergence. It is needless to say that doing so would be prohibitively expensive. As such, several algorithms have been devised for RBMs, in order to efficiently sample from during the learning process.

Contrastive Divergence (CD-k)

Contrastive Divergence uses two tricks to speed up the sampling process:

- since we eventually want (the true, underlying distribution of the data), we initialize the Markov chain with a training example (i.e., from a distribution that is expected to be close to , so that the chain will be already close to having converged to its final distribution).
- CD does not wait for the chain to converge. Samples are obtained after only k-steps of Gibbs sampling. In pratice, has been shown to work surprisingly well.

Persistent CD

Persistent CD [Tieleman08] uses another approximation for sampling from . It relies on a single Markov chain, which has a persistent state (i.e., not restarting a chain for each observed example). For each parameter update, we extract new samples by simply running the chain for k-steps. The state of the chain is then preserved for subsequent updates.

The general intuition is that if parameter updates are small enough compared to the mixing rate of the chain, the Markov chain should be able to "catch up" to changes in the model.

Implementation

We construct an `RBM` class. The parameters of the network can either be initialized by the constructor or can be passed as arguments. This option is useful when an RBM is used as the building block of a deep network, in which case the weight matrix and the hidden layer bias is shared with the corresponding sigmoidal layer of an MLP network.

```
class RBM(object):
    """Restricted Boltzmann Machine (RBM)  """
    def __init__(
        self,
        input=None,
        n_visible=784,
        n_hidden=500,
        W=None,
        hbias=None,
        vbias=None,
        numpy_rng=None,
        theano_rng=None
    ):
        """
        RBM constructor. Defines the parameters of the model along with
        basic operations for inferring hidden from visible (and vice-versa),
        as well as for performing CD updates.

        :param input: None for standalone RBMs or symbolic variable if RBM is
        part of a larger graph.

        :param n_visible: number of visible units

        :param n_hidden: number of hidden units

        :param W: None for standalone RBMs or symbolic variable pointing to a
        shared weight matrix in case RBM is part of a DBN network; in a DBN,
        the weights are shared between RBMs and layers of a MLP

        :param hbias: None for standalone RBMs or symbolic variable pointing
        to a shared hidden units bias vector in case RBM is part of a
        different network

        :param vbias: None for standalone RBMs or a symbolic variable
        pointing to a shared visible units bias
        """

        self.n_visible = n_visible
        self.n_hidden = n_hidden

        if numpy_rng is None:
            # create a number generator
            numpy_rng = numpy.random.RandomState(1234)

        if theano_rng is None:
            theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))

        if W is None:
            # W is initialized with `initial_W` which is uniformely
```

```
            # sampled from -4*sqrt(6./(n_visible+n_hidden)) and
            # 4*sqrt(6./(n_hidden+n_visible)) the output of uniform if
            # converted using asarray to dtype theano.config.floatX so
            # that the code is runable on GPU
            initial_W = numpy.asarray(
                numpy_rng.uniform(
                    low=-4 * numpy.sqrt(6. / (n_hidden + n_visible)),
                    high=4 * numpy.sqrt(6. / (n_hidden + n_visible)),
                    size=(n_visible, n_hidden)
                ),
                dtype=theano.config.floatX
            )
            # theano shared variables for weights and biases
            W = theano.shared(value=initial_W, name='W', borrow=True)

        if hbias is None:
            # create shared variable for hidden units bias
            hbias = theano.shared(
                value=numpy.zeros(
                    n_hidden,
                    dtype=theano.config.floatX
                ),
                name='hbias',
                borrow=True
            )

        if vbias is None:
            # create shared variable for visible units bias
            vbias = theano.shared(
                value=numpy.zeros(
                    n_visible,
                    dtype=theano.config.floatX
                ),
                name='vbias',
                borrow=True
            )

        # initialize input layer for standalone RBM or layer0 of DBN
        self.input = input
        if not input:
            self.input = T.matrix('input')

        self.W = W
        self.hbias = hbias
        self.vbias = vbias
        self.theano_rng = theano_rng
        # **** WARNING: It is not a good idea to put things in this list
        # other than shared variables created in this function.
        self.params = [self.W, self.hbias, self.vbias]
```

Next step is to define functions which construct the symbolic graph associated with Eqs. (7) – (8). The code is as follows:

```
    def propup(self, vis):
        '''This function propagates the visible units activation upwards to
        the hidden units

        Note that we return also the pre-sigmoid activation of the
        layer. As it will turn out later, due to how Theano deals with
        optimizations, this symbolic variable will be needed to write
        down a more stable computational graph (see details in the
        reconstruction cost function)

        '''
        pre_sigmoid_activation = T.dot(vis, self.W) + self.hbias
        return [pre_sigmoid_activation, T.nnet.sigmoid(pre_sigmoid_activation)]
```

```
    def sample_h_given_v(self, v0_sample):
        ''' This function infers state of hidden units given visible units '''
        # compute the activation of the hidden units given a sample of
        # the visibles
        pre_sigmoid_h1, h1_mean = self.propup(v0_sample)
        # get a sample of the hiddens given their activation
        # Note that theano_rng.binomial returns a symbolic sample of dtype
        # int64 by default. If we want to keep our computations in floatX
        # for the GPU we need to specify to return the dtype floatX
        h1_sample = self.theano_rng.binomial(size=h1_mean.shape,
                                             n=1, p=h1_mean,
                                             dtype=theano.config.floatX)
        return [pre_sigmoid_h1, h1_mean, h1_sample]
```

```
    def propdown(self, hid):
        '''This function propagates the hidden units activation downwards to
        the visible units

        Note that we return also the pre_sigmoid_activation of the
        layer. As it will turn out later, due to how Theano deals with
        optimizations, this symbolic variable will be needed to write
        down a more stable computational graph (see details in the
        reconstruction cost function)

        '''
        pre_sigmoid_activation = T.dot(hid, self.W.T) + self.vbias
        return [pre_sigmoid_activation, T.nnet.sigmoid(pre_sigmoid_activation)]
```

```
    def sample_v_given_h(self, h0_sample):
        ''' This function infers state of visible units given hidden units '''
        # compute the activation of the visible given the hidden sample
        pre_sigmoid_v1, v1_mean = self.propdown(h0_sample)
        # get a sample of the visible given their activation
        # Note that theano_rng.binomial returns a symbolic sample of dtype
        # int64 by default. If we want to keep our computations in floatX
        # for the GPU we need to specify to return the dtype floatX
        v1_sample = self.theano_rng.binomial(size=v1_mean.shape,
                                             n=1, p=v1_mean,
                                             dtype=theano.config.floatX)
        return [pre_sigmoid_v1, v1_mean, v1_sample]
```

We can then use these functions to define the symbolic graph for a Gibbs sampling step. We define two functions:

- `gibbs_vhv` which performs a step of Gibbs sampling starting from the visible units. As we shall see, this will be useful for sampling from the RBM.
- `gibbs_hvh` which performs a step of Gibbs sampling starting from the hidden units. This function will be useful for performing CD and PCD updates.

The code is as follows:

```
    def gibbs_hvh(self, h0_sample):
        ''' This function implements one step of Gibbs sampling,
            starting from the hidden state'''
        pre_sigmoid_v1, v1_mean, v1_sample = self.sample_v_given_h(h0_sample)
        pre_sigmoid_h1, h1_mean, h1_sample = self.sample_h_given_v(v1_sample)
        return [pre_sigmoid_v1, v1_mean, v1_sample,
                pre_sigmoid_h1, h1_mean, h1_sample]
```

```
    def gibbs_vhv(self, v0_sample):
        ''' This function implements one step of Gibbs sampling,
            starting from the visible state'''
        pre_sigmoid_h1, h1_mean, h1_sample = self.sample_h_given_v(v0_sample)
        pre_sigmoid_v1, v1_mean, v1_sample = self.sample_v_given_h(h1_sample)
        return [pre_sigmoid_h1, h1_mean, h1_sample,
                pre_sigmoid_v1, v1_mean, v1_sample]
```

Note that we also return the pre-sigmoid activation. To understand why this is so you need to understand a bit about how Theano works. Whenever you compile a Theano function, the computational graph that you pass as input gets optimized for speed and stability. This is done by changing several parts of the subgraphs with others. One such optimization expresses terms of the form log(sigmoid(x)) in terms of softplus. We need this optimization for the cross-entropy since sigmoid of numbers larger than 30. (or even less then that) turn to 1. and numbers smaller than -30. turn to 0 which in terms will force theano to compute log(0) and therefore we will get either -inf or NaN as cost. If the value is expressed in terms of softplus we do not get this undesirable behaviour. This optimization usually works fine, but here we have a special case. The sigmoid is applied inside the scan op, while the log is outside. Therefore Theano will only see log(scan(..)) instead of log(sigmoid(..)) and will not apply the wanted optimization. We can not go and replace the sigmoid in scan with something else also, because this only needs to be done on the last step. Therefore the easiest and more efficient way is to get also the pre-sigmoid activation as an output of scan, and apply both the log and sigmoid outside scan such that Theano can catch and optimize the expression.

The class also has a function that computes the free energy of the model, needed for computing the gradient of the parameters (see Eq. (4)). Note that we also return the pre-sigmoid

```
    def free_energy(self, v_sample):
        ''' Function to compute the free energy '''
        wx_b = T.dot(v_sample, self.W) + self.hbias
        vbias_term = T.dot(v_sample, self.vbias)
        hidden_term = T.sum(T.log(1 + T.exp(wx_b)), axis=1)
        return -hidden_term - vbias_term
```

We then add a `get_cost_updates` method, whose purpose is to generate the symbolic gradients for CD-k and PCD-k updates.

```
    def get_cost_updates(self, lr=0.1, persistent=None, k=1):
        """This functions implements one step of CD-k or PCD-k

        :param lr: learning rate used to train the RBM

        :param persistent: None for CD. For PCD, shared variable
            containing old state of Gibbs chain. This must be a shared
            variable of size (batch size, number of hidden units).

        :param k: number of Gibbs steps to do in CD-k/PCD-k

        Returns a proxy for the cost and the updates dictionary. The
        dictionary contains the update rules for weights and biases but
        also an update of the shared variable used to store the persistent
        chain, if one is used.

        """

        # compute positive phase
        pre_sigmoid_ph, ph_mean, ph_sample = self.sample_h_given_v(self.input)

        # decide how to initialize persistent chain:
        # for CD, we use the newly generate hidden sample
        # for PCD, we initialize from the old state of the chain
        if persistent is None:
            chain_start = ph_sample
        else:
            chain_start = persistent
```

Note that `get_cost_updates` takes as argument a variable called `persistent`. This allows us to use the same code to implement both CD and PCD. To use PCD, `persistent` should refer to a shared variable which contains the state of the Gibbs chain from the previous iteration.

If `persistent` is `None`, we initialize the Gibbs chain with the hidden sample generated during the positive phase, therefore implementing CD. Once we have established the starting point of the chain, we can then compute the sample at the end of the Gibbs chain, sample that we need for getting the gradient (see Eq. (4)). To do so, we will use the `scan` op provided by Theano, therefore we urge the reader to look it up by following this link.

```
        # perform actual negative phase
        # in order to implement CD-k/PCD-k we need to scan over the
        # function that implements one gibbs step k times.
        # Read Theano tutorial on scan for more information :
        # http://deeplearning.net/software/theano/library/scan.html
        # the scan will return the entire Gibbs chain
```

```
        (
            [
                pre_sigmoid_nvs,
                nv_means,
                nv_samples,
                pre_sigmoid_nhs,
                nh_means,
                nh_samples
            ],
            updates
        ) = theano.scan(
            self.gibbs_hvh,
            # the None are place holders, saying that
            # chain_start is the initial state corresponding to the
            # 6th output
            outputs_info=[None, None, None, None, None, chain_start],
            n_steps=k,
            name="gibbs_hvh"
        )
```

Once we have the generated the chain we take the sample at the end of the chain to get the free energy of the negative phase. Note that the `chain_end` is a symbolical Theano variable expressed in terms of the model parameters, and if we would apply `T.grad` naively, the function will try to go through the Gibbs chain to get the gradients. This is not what we want (it will mess up our gradients) and therefore we need to indicate to `T.grad` that `chain_end` is a constant. We do this by using the argument `consider_constant` of `T.grad`.

```
        # determine gradients on RBM parameters
        # note that we only need the sample at the end of the chain
        chain_end = nv_samples[-1]

        cost = T.mean(self.free_energy(self.input)) - T.mean(
            self.free_energy(chain_end))
        # We must not compute the gradient through the gibbs sampling
        gparams = T.grad(cost, self.params, consider_constant=[chain_end])
```

Finally, we add to the updates dictionary returned by scan (which contains updates rules for random states of `theano_rng`) to contain the parameter updates. In the case of PCD, these should also update the shared variable containing the state of the Gibbs chain.

```
        # constructs the update dictionary
        for gparam, param in zip(gparams, self.params):
            # make sure that the learning rate is of the right dtype
            updates[param] = param - gparam * T.cast(
                lr,
                dtype=theano.config.floatX
            )
        if persistent:
            # Note that this works only if persistent is a shared variable
            updates[persistent] = nh_samples[-1]
            # pseudo-likelihood is a better proxy for PCD
            monitoring_cost = self.get_pseudo_likelihood_cost(updates)
        else:
            # reconstruction cross-entropy is a better proxy for CD
            monitoring_cost = self.get_reconstruction_cost(updates,
                                                           pre_sigmoid_nvs[-1])

        return monitoring_cost, updates
```

Tracking Progress

RBMs are particularly tricky to train. Because of the partition function Z of Eq. (1), we cannot estimate the log-likelihood during training. We therefore have no direct useful metric for choosing the optimal hyperparameters.

Several options are available to the user.

Inspection of Negative Samples

Negative samples obtained during training can be visualized. As training progresses, we know that the model defined by the RBM becomes closer to the true underlying distribution, . Negative samples should thus look like samples from the training set. Obviously bad hyperparameters can be discarded in this fashion.

Visual Inspection of Filters

The filters learnt by the model can be visualized. This amounts to plotting the weights of each unit as a gray-scale image (after reshaping to a square matrix). Filters should pick out strong features in the data. While it is not clear for an arbitrary dataset, what these features should look like, training on MNIST usually results in filters which act as stroke detectors, while training on natural images lead to Gabor like filters if trained in conjunction with a sparsity criteria.

Proxies to Likelihood

Other, more tractable functions can be used as a proxy to the likelihood. When training an RBM with PCD, one can use pseudo-likelihood as the proxy. Pseudo-likelihood (PL) is much less expensive to compute, as it assumes that all bits are independent. Therefore,

Here denotes the set of all bits of x except bit . The log-PL is therefore the sum of the log-probabilities of each bit , conditioned on the state of all other bits. For MNIST, this would involve summing over the 784 input dimensions, which remains rather expensive. For this reason, we use the following stochastic approximation to log-PL:

where the expectation is taken over the uniform random choice of index , and is the number of visible units. In order to work with binary units, we further introduce the notation to refer to x with bit-i being flipped (1->0, 0->1). The log-PL for an RBM with binary units is then written as:

We therefore return this cost as well as the RBM updates in the `get_cost_updates` function of the `RBM` class. Notice that we modify the updates dictionary to increment the index of bit . This will result in bit cycling over all possible values , from one update to another.

Note that for CD training the cross-entropy cost between the input and the reconstruction (the same as the one used for the de-noising autoencoder) is more reliable then the pseudo-loglikelihood. Here is the code we use to compute the pseudo-likelihood:

```
    def get_pseudo_likelihood_cost(self, updates):
        """Stochastic approximation to the pseudo-likelihood"""

        # index of bit i in expression p(x_i | x_{\i})
        bit_i_idx = theano.shared(value=0, name='bit_i_idx')

        # binarize the input image by rounding to nearest integer
        xi = T.round(self.input)

        # calculate free energy for the given bit configuration
        fe_xi = self.free_energy(xi)

        # flip bit x_i of matrix xi and preserve all other bits x_{\i}
        # Equivalent to xi[:,bit_i_idx] = 1-xi[:, bit_i_idx], but assigns
        # the result to xi_flip, instead of working in place on xi.
        xi_flip = T.set_subtensor(xi[:, bit_i_idx], 1 - xi[:, bit_i_idx])

        # calculate free energy with bit flipped
        fe_xi_flip = self.free_energy(xi_flip)

        # equivalent to e^(-FE(x_i)) / (e^(-FE(x_i)) + e^(-FE(x_{\i})))
        cost = T.mean(self.n_visible * T.log(T.nnet.sigmoid(fe_xi_flip -
                                                            fe_xi)))

        # increment bit_i_idx % number as part of updates
        updates[bit_i_idx] = (bit_i_idx + 1) % self.n_visible

        return cost
```

Main Loop

We now have all the necessary ingredients to start training our network.

Before going over the training loop however, the reader should familiarize himself with the function `tile_raster_images` (see Plotting Samples and Filters). Since RBMs are generative models, we are interested in sampling from them and plotting/visualizing these samples. We also want to visualize the filters (weights) learnt by the RBM, to gain insights into what the RBM is actually doing. Bear in mind however, that this does not provide the entire story, since we neglect the biases and plot the weights up to a multiplicative constant (weights are converted to values between 0 and 1).

Having these utility functions, we can start training the RBM and plot/save the filters after each training epoch. We train the RBM using PCD, as it has been shown to lead to a better generative model ([Tieleman08]).

```
    # it is ok for a theano function to have no output
    # the purpose of train_rbm is solely to update the RBM parameters
    train_rbm = theano.function(
        [index],
        cost,
        updates=updates,
        givens={
            x: train_set_x[index * batch_size: (index + 1) * batch_size]
        },
        name='train_rbm'
    )

    plotting_time = 0.
    start_time = timeit.default_timer()

    # go through training epochs
    for epoch in range(training_epochs):

        # go through the training set
        mean_cost = []
        for batch_index in range(n_train_batches):
            mean_cost += [train_rbm(batch_index)]

        print('Training epoch %d, cost is ' % epoch, numpy.mean(mean_cost))

        # Plot filters after each training epoch
        plotting_start = timeit.default_timer()
        # Construct image from the weight matrix
        image = Image.fromarray(
            tile_raster_images(
                X=rbm.W.get_value(borrow=True).T,
                img_shape=(28, 28),
                tile_shape=(10, 10),
                tile_spacing=(1, 1)
            )
        )
        image.save('filters_at_epoch_%i.png' % epoch)
        plotting_stop = timeit.default_timer()
        plotting_time += (plotting_stop - plotting_start)

    end_time = timeit.default_timer()

    pretraining_time = (end_time - start_time) - plotting_time

    print ('Training took %f minutes' % (pretraining_time / 60.))
```

Once the RBM is trained, we can then use the `gibbs_vhv` function to implement the Gibbs chain required for sampling. We initialize the Gibbs chain starting from test examples (although we could as well pick it from the training set) in order to speed up convergence and avoid problems with random initialization. We again use Theano's `scan` op to do 1000 steps before each plotting.

```
    #################################
    #     Sampling from the RBM     #
    #################################
    # find out the number of test samples
    number_of_test_samples = test_set_x.get_value(borrow=True).shape[0]

    # pick random test examples, with which to initialize the persistent chain
    test_idx = rng.randint(number_of_test_samples - n_chains)
```

```
persistent_vis_chain = theano.shared(
    numpy.asarray(
        test_set_x.get_value(borrow=True)[test_idx:test_idx + n_chains],
        dtype=theano.config.floatX
    )
)
```

Next we create the 20 persistent chains in parallel to get our samples. To do so, we compile a theano function which performs one Gibbs step and updates the state of the persistent chain with the new visible sample. We apply this function iteratively for a large number of steps, plotting the samples at every 1000 steps.

```
plot_every = 1000
# define one step of Gibbs sampling (mf = mean-field) define a
# function that does `plot_every` steps before returning the
# sample for plotting
(
    [
        presig_hids,
        hid_mfs,
        hid_samples,
        presig_vis,
        vis_mfs,
        vis_samples
    ],
    updates
) = theano.scan(
    rbm.gibbs_vhv,
    outputs_info=[None, None, None, None, None, persistent_vis_chain],
    n_steps=plot_every,
    name="gibbs_vhv"
)

# add to updates the shared variable that takes care of our persistent
# chain :.
updates.update({persistent_vis_chain: vis_samples[-1]})
# construct the function that implements our persistent chain.
# we generate the "mean field" activations for plotting and the actual
# samples for reinitializing the state of our persistent chain
sample_fn = theano.function(
    [],
    [
        vis_mfs[-1],
        vis_samples[-1]
    ],
    updates=updates,
    name='sample_fn'
)

# create a space to store the image for plotting ( we need to leave
# room for the tile_spacing as well)
image_data = numpy.zeros(
    (29 * n_samples + 1, 29 * n_chains - 1),
    dtype='uint8'
)
for idx in range(n_samples):
    # generate `plot_every` intermediate samples that we discard,
    # because successive samples in the chain are too correlated
    vis_mf, vis_sample = sample_fn()
    print(' ... plotting sample %d' % idx)
    image_data[29 * idx:29 * idx + 28, :] = tile_raster_images(
        X=vis_mf,
        img_shape=(28, 28),
        tile_shape=(1, n_chains),
        tile_spacing=(1, 1)
    )

# construct image
image = Image.fromarray(image_data)
image.save('samples.png')
```

Results

We ran the code with PCD-15, learning rate of 0.1 and a batch size of 20, for 15 epochs. Training the model takes 122.466 minutes on a Intel Xeon E5430 @ 2.66GHz CPU, with a single-threaded GotoBLAS.

The output was the following:

```
... loading data
Training epoch 0, cost is  -90.6507246003
Training epoch 1, cost is  -81.235857373
Training epoch 2, cost is  -74.9120966945
Training epoch 3, cost is  -73.0213216101
Training epoch 4, cost is  -68.4098570497
Training epoch 5, cost is  -63.2693021647
Training epoch 6, cost is  -65.99578971
Training epoch 7, cost is  -68.1236650015
Training epoch 8, cost is  -68.3207365087
Training epoch 9, cost is  -64.2949797113
Training epoch 10, cost is  -61.5194867893
Training epoch 11, cost is  -61.6539369402
Training epoch 12, cost is  -63.5465278086
Training epoch 13, cost is  -63.3787093527
Training epoch 14, cost is  -62.755739271
Training took 122.466000 minutes
 ... plotting sample  0
 ... plotting sample  1
 ... plotting sample  2
 ... plotting sample  3
 ... plotting sample  4
 ... plotting sample  5
 ... plotting sample  6
 ... plotting sample  7
 ... plotting sample  8
 ... plotting sample  9
```

The pictures below show the filters after 15 epochs:

Filters obtained after 15 epochs.

Here are the samples generated by the RBM after training. Each row represents a mini-batch of negative particles (samples from independent Gibbs chains). 1000 steps of Gibbs sampling were taken between each of those rows.