cs645 Spring 2014 Final Project: Using restricted Boltzmann Machines to pre-train neural net classifiers on Ground Cover Images

Basir Shariat and Charlie Vollmer
Departments of Computer Science and Statistics
Colorado State University
Fort Collins, CO
b.shariat@gmail.com
charlesv@rams.colostate.edu

Version: May 19, 2014

Abstract

This is the final project for Professor Chuck Anderson's course cs645 Adv Neural Networks. It presents the results of our final semester project for the course. We propose an unsupervised feature extraction approach to classifying ground cover images, provided by Professor Anderson. We trained stacked, restricted Boltzmann Machines (rBM's) to learn a generative model of the data set, to be able to extract feature vectors as well as generate samples from the distributions of the feature vectors, to be used in training deeper layers of neural networks. From these extracted features, we then have a closer approximation to the global minima to facilitate the use of back-propogation algorithms to be able to optimize the weights over the features, to be used in classifying the data set.

1 Introduction

Researchers from all fields are now turning to statistical machine learning methods to help resolve various data problems within their respective disciplines. The data are large, unstructured, and full of valuable insights that are seemingly impossible to tease out with traditional mathematical modeling.

1.1 The Problem

The problem presented to us is no different than the situation described above: classify an aerial-type image of brush-land groundcover.

This is a tedious task to be done by hand for thousands of images, and has obvious motivations for automation. It is time-consuming and resource-intensive.

We propose to automate this task through machine learning techniques, specifically deep learning networks.

1.2 A (Deep) Learning Experience

In the last decade, there has been a lot of energy and excitement over a new field within Machine Learning which attempts to move Machine Learning closer to its origins and foundational motivations of Artificial Intelligence: a field merrily referred to as "Deep Learning."

Deep Learning is about trying to learn multiple levels of representation and abstraction of the data, something akin to how some theorize the human mind to receive and analyze sensory information.

It is difficult to optimize the weights in nonlinear models that have multiple hidden layers (2-4). With large initial weights, autoencoders typically find poor local minima; with small initial weights, the gradients in the early layers are tiny, making it infeasible to train models with many hidden layers. If the initial weights are close to a good solution, gradient descent works well, but finding such initial weights requires a very different type of algorithm that learns one layer of features at a time.

1.3 Why are we interested?

The problem presented to us is a difficult one due to the nature of these images. They are taken by hand, and with little systematic procedure. They are inherently variable, and possibly filled with lots of noise (read: air and light pollution). This task certainly fits the bill when it comes to the fact that out-of-the-box methods just don't fit the data set. The traditional distributional assumptions are nowhere to be found, pre-processing causes headaches, and reliability/consistency of the signal is even out of the question.

This is interesting to us because we are delving into a large and energetically exciting "new field" that seems to be moving fast and doing really innovative and cool things. Deep Learning combines aspects of tried and true modeling methods, such as neural networks, with new biological ideas of their motivations, novel optimization techniques, and in exciting applications relevant to internet-sized data sets and newly-encountered, hard problems in AI.

Ours is an attempt to find higher-level representations of the ground cover data. We are trying to discover if certain ground textures can be learned in our feature vectors in a pre-training phase, that will allow us to model the classification of the images using a network of multiple hidden layers.

1.4 Overview:

The following topics will be discussed in the subsequent sections:

1. a Background of the methods and literature will be presented
2. code and resources which we consulted will be discussed
3. the methods we employed and code used will be provided
4. a short summary of our experiments and results
5. a discussion of our findings
6. main take-aways of our project and concepts we learned

2 Background

We proposed using a deep learning algorithm that extracts feature vectors from the data set using an unsupervised method; learning a generative model that can be used to further extract feature vectors in a deep architecture. The model that we employed is called a restricted Boltzmann machine [4].

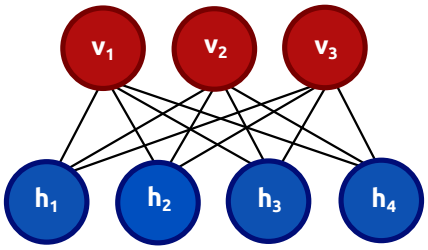


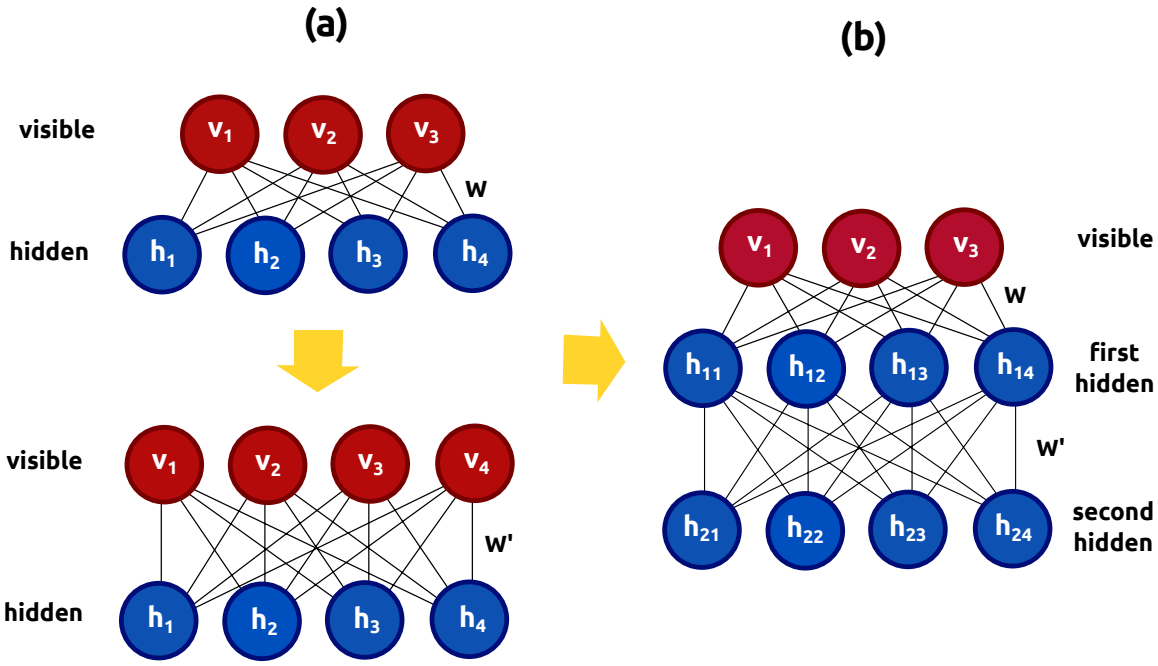
Figure 1: A restricted Boltzmann machine. Restricted because there is "communication" within a layer only through the connections to another layer.

The motivation behind using this as our model comes from the fact that it is unsupervised learning[1]. The idea is that we learn the features and structure within the data, and capture the structure in our feature vectors. Similar to the methods of PCA, which looks for the structure through capturing the directions of most variability in the linear subspaces spanned by the columns of the data matrix, the rBM model tries to capture the non-linear structure in our data. These machines have been shown to perform excellently on a multitude of applications, notwithstanding the infamous MNIST data set[8], as discussed above.

A very nice property of these particular models is that they are "generative." Generative in the sense that we can learn a distribution over the feature space of the features, and actually generate samples from those distributions. This allows us to then create a whole new data set from our learned features and use those generated samples as a new data set to train another layer, creating a stacked model of rBMs (see Figure 2, on next page).

The benefits of stacking these auto-encoders is that they learn different structures in the original data set, something akin to thinking that the deeper you go, the more "primitive" or fundamental the structure learned is to the data.

What we hoped was that using these models, we would learn the structure in our ground cover images reasonably well to then have good starting locations for our feature vectors to then be "tuned" for classification using a back-propogation algorithm [5]. Since there is trouble with gradients propagating through multiple hidden layers in a neural network, it is of the utmost importance to be able to start within reasonably close approximations to your global minima, so that the back propagation optimization does not get stuck in any local minima, and so that the weak gradients that do propagate through the hidden layers have an effect. In the last few years, quite a few learning algorithms have been presented to efficiently perform exactly this task, with different methods better suited to feature and sample generation or to discrimination [9, 10]



The structure that we were hoping to learn really revolved around the textures of the different forms of ground cover. We hoped that some features would be able to learn the textures of each of the different shrubs: like bare ground vs grass.

2.1 The Mathematics of Training:

rBM's are generally trained, and this case notwithstanding, using the technique of contrastive divergence, originally authored by Geoffrey Hinton [2]. Acknowledged by the author himself, contrastive divergence requires a "certain amount of practical experience," to be able to decide the optimal numeric values to assign to different tuning parameters of the model, such as the learning rate, the weight-cost, initial value of weights, and the number of hidden units among many, many others. Many other decisions also need to be made by the implementer regarding whether it is necessary to update the unit states stochastically or deterministically, how many times to update the states of the hidden units for each training case and even what type of units to use. It is also helpful to monitor the progress of the actual training itself and make some subjective decisions as to when to terminate the actual training process, as well [3].

Among all of these myriad decisions lie fine lines which the practitioner must trod, amongst such exist heuristics that one may only acquire through trial and error and through diligent attention and observation paid to a variety of applications. This has been -to be sure- the case which we experienced and perhaps enjoyed the most: learning the subtleties of the model and its "quirks," if we may, through this unique application.

A most useful ability was acquired amongst this process: to be able to relate the failures of learning to the particular decisions which may have caused them.

2.1.1 rBM's as an Energy Model:

Since an rBM is a stochastic neural network(neural network meaning we have neuron-like units whose binary activations depend on the neighbors they're connected to; stochastic meaning these activations have a probabilistic element) in which stochastic, binary pixels are connected to stochastic, binary feature detectors using symmetrically weighted connections, we describe the model as the following: the pixels correspond to "visible" units of the RBM because their states are observed, while the feature detectors correspond to "hidden" units. A joint configuration, $(\mathbf{v}, \mathbf{h})$, of the visible and hidden units is modeled as having an energy [6] given by:

$$E(\mathbf{v}, \mathbf{h}) = -\sum_{i \in pixels} b_i v_i - \sum_{j \in features} b_j h_j - \sum_{i,j} v_i h_j w_{ij} \tag{1}$$

where v_i and h_j are the binary states of pixel i and feature j, b_i and b_j are their biases, and w_{ij} is the weight between them.

The network assigns a probability to every possible image via this energy function:

$$P(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} e^{-E(\mathbf{v},\mathbf{h})} \tag{2}$$

where the "partition function", Z, is given by summing over all possible pairs of visible and hidden vectors:

$$Z = \sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \tag{3}$$

The probability that the network assigns to a visible vector, $\mathbf{v}$, is given by summing over all possible hidden vectors:

$$p(\mathbf{v}) = \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \tag{4}$$

The probability that the network assigns to a training image can be raised by adjusting the weights and biases to lower the energy of that image and to raise the energy of other images, especially those that have low energies and therefore make a big contribution to the partition function [5]. The derivative of the log probability of a training vector with respect to a weight is surprisingly simple.

$$\frac{\partial p(\mathbf{v})}{\partial w_{ij}} = \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model} \tag{5}$$

where the angle brackets denote expectations under the distribution specified by the subscript. This leads to the very convenient learning rule for performing the stochastic gradient ascent in the log probability of the training data:

$$\Delta w_{ij} = \epsilon(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}) \tag{6}$$

where ϵ is the learning rate.

2.2 A Practical Learning Algorithm

Now, there is a very nice result from the fact that these networks are "restricted:" since there are no direct connections between hidden units in an RBM, it becomes very easy to get an unbiased sample of $\langle v_i h_j \rangle_{data}$. Due to the affine form of the energy equation, with respect to h, we readily obtain a tractable expression for the conditional probability. Given a randomly selected training image, $\mathbf{v}$, the binary state, h_j, of each hidden unit, j, is set to 1 with the following probability:

$$p(h_j = 1|\mathbf{v}) = \sigma(b_j + \sum_j v_i w_{ij}) \tag{7}$$

where $\sigma(\cdot)$ is the logistic sigmoidal function $\frac{1}{(1+exp(-x))}$. $v_i h_j$ is then an unbiased sample.

Now, since v and h play a symmetric role in the energy function, a similar derivation allows to efficiently compute and sample $P(v = 1|h)$, an unbiased sample of the state of a visible unit given a hidden vector:

$$p(v_i = 1|\mathbf{h}) = \sigma(a_i + \sum_i h_j w_{ij}) \tag{8}$$

Now, getting an unbiased sample of $\langle v_i h_j \rangle_{model}$, however, is much more difficult. It can be done by starting at any random state of the visible units and performing alternating Gibbs sampling for a very long time. One iteration of alternating Gibbs sampling consists of updating all of the hidden units in parallel using equation (7) followed by updating all of the visible units in parallel using equation (8).

A much faster learning procedure was proposed in [2], by use of Contrastive Divergence. This starts by setting the states of the visible units to a training vector. Then the binary states of the hidden units are all computed in parallel using equation (7). Once binary states have been chosen for the hidden units, a "reconstruction" is produced by setting each v_i to 1 with a probability given by equation (8). The change in a weight is then given by the following approximation:

$$\Delta w_{ij} = \epsilon(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{recon}) \tag{9}$$

In this case, a simplified version of the same learning rule that uses the states of individual units instead of pairwise products is used for the biases.

2.3 Contrastive Divergence

The method of training that we implemented, and which seems to be frequently used, was that of Contrastive Divergence (CD) [2]. Now, it needs to be made clear that by using CD we are only approximating the gradient of the log probability of the training data. In fact, this can actually be though of as only very *crudely* approximating it. Equation (9) is actually an approximation of yet another objective function which is known as Contrastive Divergence: the difference between two Kullback-Liebler divergences. Yet, this learning rule, Equation (9), is ignoring an important and tricky term in this objective function and -as such- is *not* even following the gradient. It has actually been shown by Sutskever and Tieleman that it is *not* following the gradient of *any* function! [11]

It should be noted that while this learning rule does not have the nice properties desired by the pedantic theorist, it seemingly works well enough to achieve success in many significant applications.

Hinton points out that rBM's will typically learn better models if more steps of alternating Gibbs sampling are used before collecting the statistics for the second term in the learning rule, which he calls the negative statistics [3]. We will denote the learning that uses the n full steps of alternating Gibbs sampling by CD_n.

We will also assume all of the units, whether hidden or visible, to be binary. While other types of units have been used, our application of image classification can deal with this most basic assumption. We also approach our modeling process under the pretense that our goal is to learn a good generative model of the set of training vectors. This approach can be relaxed in the case of subsequently fine-tuning the model using back propagation, where the generative model is not the ultimate objective (as is our case), and may be under-fit to save time. Yet, we would like to generate the best possible feature vectors, as well as understand for ourselves what are the paths to achieve such a directive.

2.3.1 Updating the hidden states

Using a one-step contrastive divergence, CD_1, and assuming the hidden units are binary, the units should have a stochastic binary state when being driven by a data vector. The probability of turning on a hidden unit, j, is computed similarly to as stated above, by applying the logistic function to its "total input:"

$$p(h_j = 1) = \sigma(b_j + \sum_i v_i w_{ij}) \tag{10}$$

and we turn this hidden unit on if this probability is greater than a random number generated from a standard continuous uniform distribution.

Hinton stresses [3] that it is very important to make these hidden states binary, rather than using the probabilities themselves, as if the probabilities are used each hidden unit can communicate a real-value to the visible units during the reconstruction. *This* seriously violates the information bottleneck created by the fact that a hidden unit can convey at most one bit (on average). This information bottleneck is acting as a very strong regularizer.

Hinton also suggests [3] that using a stochastic binary state for the lsat update of the hidden units is unwarranted since nothing depends on which state is chosen (it is the last state). He suggests, rather, to use the probability itself to avoid unnecessary sampling noise. When using CD_n, only the final update should use the actual probability.

2.3.2 Updating the visible states

Similarly, we stochastically pick a 1 or a 0 to update the visible states, when generating a reconstruction, with a probability determined by the total top-down input:

$$p_i = p(v_i = 1) = \sigma(a_i + \sum_j h_j w_{ij}) \tag{11}$$

In this case, for practical matters of allowing a faster learning, Hinton suggests [3] using the actual probabilities instead of sampling a binary value since it is not "nearly as problematic" as for the data-driven hidden states.

Hinton has actually seen evidence that this leads to worse density models (using the actual probability), yet this probability does not matter when using an rBM to pretrain a layer of hidden features for use in a deep belief net.

2.3.3 Two methods to "collect the statistics"

When the visible units are using real-valued probabilities instead of stochastic binary values, we can "collect the statistics" for the connection between visible unit i and hidden unit j by either of the following:

$$\langle p_i h_j \rangle_{data} \qquad \text{or} \qquad \langle p_i p_j \rangle_{data}$$

where p_j is a probability and h_j is a binary state that takes value 1 with probability p_j.

Using h_j is closer to the mathematical model of an rBM, as exposed above, but using p_j usually has less sampling noise which allows slightly faster learning, claims Hinton [3]. This is because h_j *always* creates more noise in the positive statistics than using p_j but it can actually create less noise in the $difference$ of the positive and the negative statistics, Equation (9), because the negative statistics depend on the binary decision for the state of j that is used for creating the reconstruction. The probability of j when driven by the reconstruction is highly correlated with the binary decision that was made for j when it was driven by the data.

2.3.4 A recipe for getting the learning signal for CD_1

We took the caution mentioned earlier to the fact that much care is needed when selecting the tuning parameters of the model and this care really only comes from hard-earned experience through many applications. Therefore, we paid close attention to the counsel of Geoffrey Hinton himself and tried to apply his own experiential advice wherever we could. Below are some of the practical pointers that we were able to extract from his writings:

1. Hinton tells us to *always* use the stochastic binary states when the hidden units are being driven by the data [3]. When they are driven by the reconstructions, however, we are to always us the probabilities without sampling.

2. We are also advised to not allow anything random about the generation of the reconstructions given the binary states of the hidden units. When the visible units use the logistic function, we are to use the real-valued probabilities for both the data and the reconstructions.

3. When collecting the pairwise statistics for learning the weights of the individual statistics for learning the biases, we are to use the probabilities and not the binary states, and to ensure that the weights have random initial values, as to break symmetry.

3 Method

Our main focus for this project was on the pre-training phase of the classifiers, rather than on the training ("fine tuning") of the classifier itself. Thus, we placed most of our efforts on doing the unsupervised training of restricted Boltzmann Machines to learn the feature vectors from our data. We also trained a classifier on our set of best pre-training parameters. In the following we will describe the pre-processing of our data, the unsupervised learning of our feature vectors using rBM's, and our classification model.

3.1 Data Overview: From raw to usable

The dataset consists of 1623 aerial-type images of brush-land groundcover, each image in either a portrait or landscape mode ($2272 \times 1704 \approx 4MP$). All the images are taken during the day time but the angle of capturing the image (relative to the surface of the ground) and the light condition is variable among different images. Figure 2 illustrates sample images from the dataset. The images are accompanied by a spreadsheet file which specifies the percent of following covers in each image: Grass, Shrub, Forb, Salsola, Bare, Other. Please note that this is not a location based annotation (Figure 4) of the image but a plain percentage annotation. The challenge is to train an image annotation algorithm, which will compute the percentage of each of these ground covers and show them in the image.

We had to do a variety of pre-processing steps to the data itself just to be able to start to train our model, as the raw data presented many issues that we needed to overcome to practically solve the problem at hand. The data themselves were not directly usable in our model, and we simply did not have the resources to be able to compute on the size of the data we had.

Some of the preprocessing we performed are:

1. Setting the mode of all the images to landscape.
2. Generating thumbnails of different sizes for each image.
3. Converting images to grey-scale.
4. Avoiding digits of date printed in images.

All of these preprocessing steps are done using the Python Imaging Library (PIL).

3.1.1 The need for Patches

One inherent characteristic of our data is that they have these patches of texture of different covers all over the images and in different places within the image in each instance (demonstrated in Figures 3 and 2).

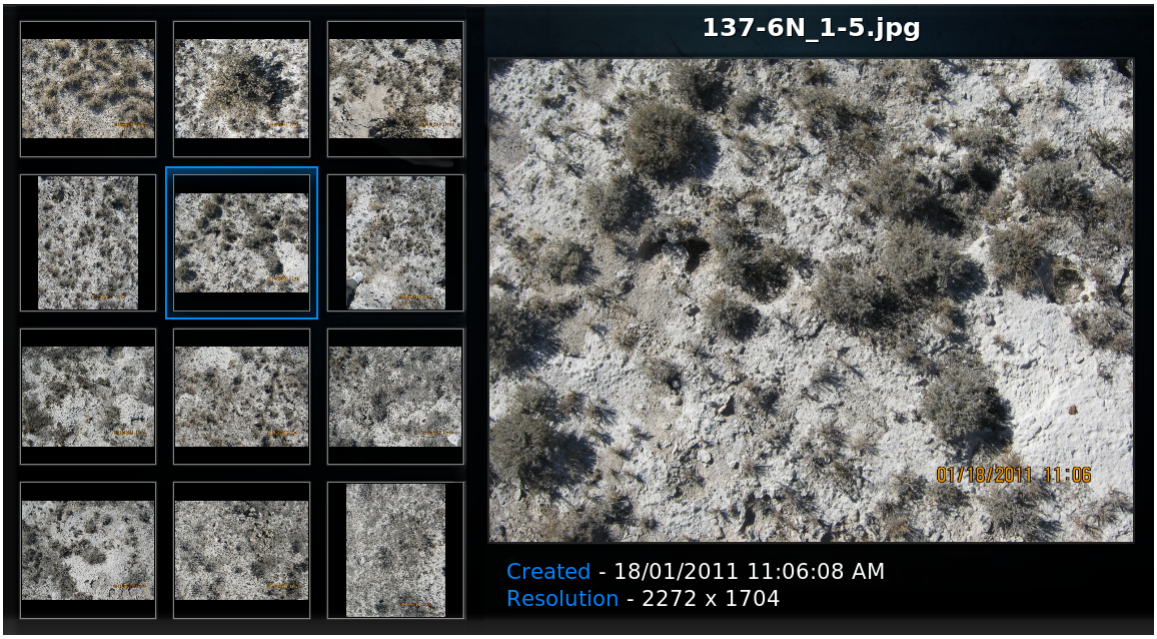


Figure 2: Example images from the groundcover dataset.

After first attempting the naive approach of training an rBM model on the raw gray scale images (thumbnails of 300×225 pixels from the original images), we found that the feature vectors were completely washed in noise, and no relevant information was actually learned. Thus, we decided to "chop" our training images up into patches of different covers. We then used these patches from the images as our training data to train our feature vectors.

This has the effect of training on "images" that are of a consistent texture. We did

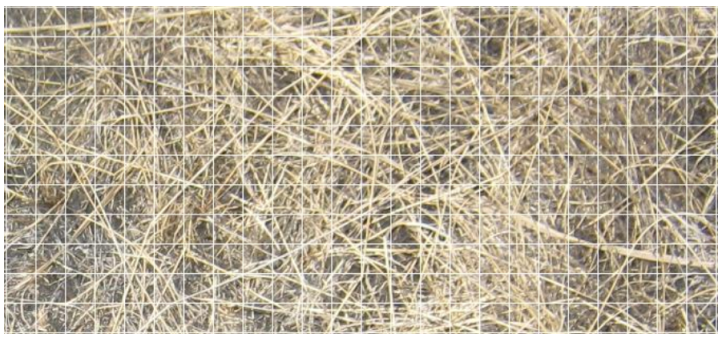


Figure 3: Patches of grass in a sample image

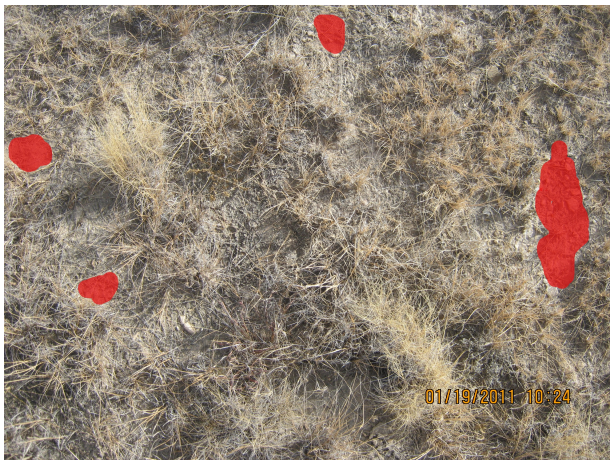


Figure 4: An annotated image with 95% grass and 5% bare. Red areas have cover bare others are grass.

these patches with the hope of isolating the different textures present only within these small patches, such that our features would learn a texture particular to a type of ground cover. One idea is that the features should be able to learn some of this repetitious structure.

As we mentioned earlier, our dataset contains no location-based annotations (please see Figure 4). So in order to generate small patches that only/mostly consists of a specific cover, we either had to extract them manually or do them automatically using images with a high percentage of a certain cover. We took the second approach, and to do so from the spreadsheet file we found images with a high percentage of grass and bare (more than 95%) and we uniformly sampled n small patches (28×28 pixels and $100 \leq n \leq 10000$) from these images and saved the patches into new files.

```
def generate_patches(pictures_folder, csv_file):
    patches = {}
    covers = ['bare','grass']
    index = 0
    nprnd.seed(123)
    for cover in covers:
        index += 1
        count = 0
        patch_folder = 'patches-' + cover
        patch_size = 28
        sample_no_from_each_image = 50
        if not os.path.isdir(patch_folder):
            os.mkdir(patch_folder)
        with open(csv_file, 'rb') as f:
            reader = csv.reader(f)
            try:
                # take maximum over percentage
                for row in reader:
                    if is_number(row[0]):
                        if row[12].lower() == cover and float(row[14]) > 95:
                            count += 1
```

```
                folder_path = pictures_folder + '//' + row[2]

                if os.path.isdir(folder_path):
                    image_path = folder_path + '//' + row[10] + '.jpg'
                    patch_path = patch_folder + '//' + row[1] + "_" + row[1

                    image = Image.open(image_path).convert("L")
                    width, height = image.size
                    xs = nprnd.randint(width - patch_size, size=sample_no_

                    if width > height:
                        # Avoiding the dates on the images
                        ys = nprnd.randint(1420 - patch_size, size=sample_
                    else:
                        # Avoiding the dates on the images
                        ys = nprnd.randint(2000 - patch_size, size=sample_

                    if index == 1:
                        if count == 3:
                            for i in range(2000):
                                patch_file = patch_path + '_' + str(i) + '
                                image.crop((xs[36], ys[36], xs[36] + patch
                                patches[patch_file] = int(index)
                        else:
                            for i in range(sample_no_from_each_image):
                                patch_file = patch_path + '_' + str(i) + '
                                image.crop((xs[i], ys[i], xs[i] + patch_si
                                patches[patch_file] = int(index)
                    else: # cover == 'grass'
                        if count == 2:
                            for i in range(2000):
                                patch_file = patch_path + '_' + str(i) + '
                                image.crop((xs[0], ys[0], xs[0] + patch_si
                                patches[patch_file] = int(index)
                        else:
                            for i in range(sample_no_from_each_image):
                                patch_file = patch_path + '_' + str(i) + '
                                image.crop((xs[i], ys[i], xs[i] + patch_si
                                patches[patch_file] = int(index)
            except csv.Error as e:
                sys.exit('file %s, line %d: %s' % (csv_file, reader.line_num, e))
    return patches, np.prod([patch_size, patch_size])
```

3.1.2 From patches to learning:

After having transformed our original data into a more malleable and suitable type for our learning model, we needed our training data to be placed into numpy objects and Theano shared variables in order to be able to feed them to our models. We first flattened our patch-images of size 28×28 into one dimensional arrays of size 784 and then shuffled them along the labels. Then we divided our dataset into three parts: training (80%), validate (10%), testing (10%). The python code for loading the patches can be seen in the following listing:ded into minibatches of fixed size.

```
1
2  def load_ground_cover(pictures_folder, csv_file):
3      patches, size = generate_patches(pictures_folder, csv_file)
4      n = len(patches)
5      images = np.ndarray(shape=(n, size))
6      labels = np.ndarray(shape=(n, 1))
7      i = 0
8      for thumb_file_name in patches:
9          image = Image.open(thumb_file_name)
10         image_array = np.asarray(image)/255
11         images[i, :] = image_array.reshape(1, size)
12         labels[i] = patches[thumb_file_name]
13         i += 1
14
15     train_ind = int(0.8*n)
16     valid_ind = int(0.9*n)
17     return [(shared(images[:train_ind,:]),shared(labels[:train_ind])),
18       (shared(images[train_ind:valid_ind,:]),shared(labels[train_ind:valid_ind])),
19    (shared(images[valid_ind:, :]),shared(labels[valid_ind:]))]
```

3.2 Unsupervised Learning of Feature Vectors

To train our rBM's we used open-source code that we got from the LISA lab, at the University of Montreal [7] as a base and modified it to fit it to our needs. The code is written in Theano, a symbolic computation package in python tailored specifically for deep learning. We have an RBM class which represents a single layer of a deep belief network. The following list shows this class and the signatures of its methods:

```
1  class RBM(object):
2      """
3         Restricted Boltzmann Machine (RBM)
4         One layer of a Deep Belief Network
5      """
6      def __init__(self, input, n_visible, n_hidden,
7        W, hbias, vbias, numpy_rng, theano_rng):
8      def free_energy(self, v_sample):
9          ...
10     def propup(self, vis):
11         ...
12     def sample_h_given_v(self, v0_sample):
13         ...
14     def propdown(self, hid):
15         ...
16     def sample_v_given_h(self, h0_sample):
17         ...
18     def gibbs_hvh(self, h0_sample):
19         ...
20     def gibbs_vhv(self, v0_sample):
21         ...
22     def get_cost_updates(self, lr, persistent, k):
23         ...
24     def get_pseudo_likelihood_cost(self, updates):
```

```
25          ...
26      def get_reconstruction_cost(self, updates, pre_sigmoid_nv):
27          ...
```

3.2.1 The ctor:

First lets take a look at the constructor of the RBM class. Parameters *n_visible* and *n_hidden* specify the number of visible and hidden units in the RBM. Parameters W, *hbias* and *vbias* are free parameters of the model which will be updated in the pre-training and training phases. These three parameters are converted to Theano shared variables which later will be shared between the different layers of deep belief network and its fine-tuning function. *theano_rng* is used as the random number generator for sampling from the trained distribution.

3.2.2 Activation of the Units:

Two functions *propup* and *propdown* return the symbolic expressions for activations of the visible units (returned to hidden units) and hidden units (returned to visible units), Equations (7), (8), (10), and (11). Functions *sample_h_given_v*/*sample_v_given_h* will return samples from hidden/visible units, given the current state of the visible/hidden units (using *propup* and *propdown*) by a binomial distribution. Also functions *gibbs_hvh* and *gibbs_vhv* return the symbolic expression for one step of Gibbs sampling.

3.2.3 The Energy function:

Given a sample *v_sample* from the visible layer, $free\_energy$ function computes the free energy of the model, Equation (1), which is the linear combination of free parameter W, b and c that was mentioned earlier.

3.2.4 Using Contrastive Divergence:

Finally, function *get_cost_updates* returns the symbolic updates for gradient descent (using the $T.grad$ function of Theano) and monitoring costs for the contrastive divergence (CD) and persistent contrastive divergence (PCD) methods that are used for approximation of Gibbs sampling, Equation (9). The last two function signatures shown above are helper functions which return the monitoring cost for each of the methods PCD and CD, respectively.

Once we have our RBM object we can train it on minibatches by defining a Theano function that takes the minibatches as input and rbm cost as output. The optimization is done using the symbolic expression return by RBM's *get_cost_updates* function. Pre-training is done in multiple epochs and we output the W matrix as an image at the end of each epoch in order to see how the model evolves during the pre-training phase. After the pre-training is finished, we sample from the model by calling *gibbs_vhv* function of the RBM class.

3.3 Classification of Ground Cover

As we mentioned earlier, an ideally trained rBM (or a set of stacked rBMs) can generate samples from the distribution they are representing. Hence the weights and biases of such

an rBM seem to be a good starting point for training a multilayer perceptron with the same architecture. Shared variables in Theano make this task simple. All we have to do is define a hidden layer class for our multilayer perceptron that shares its weight and bias variables with the corresponding hidden layer in the deep belief network. The Theano code for the *HiddenLayer* class can be seen below:

```
class HiddenLayer(object):
    def __init__(self, rng, input, n_in, n_out, W=None, b=None,
                 activation=T.tanh):
        self.input = input
        if W is None:
            W_values = numpy.asarray(rng.uniform(
                    low=-numpy.sqrt(6. / (n_in + n_out)),
                    high=numpy.sqrt(6. / (n_in + n_out)),
                    size=(n_in, n_out)), dtype=theano.config.floatX)
            if activation == theano.tensor.nnet.sigmoid:
                W_values *= 4
            W = theano.shared(value=W_values, name='W', borrow=True)
        if b is None:
            b_values = numpy.zeros((n_out,), dtype=theano.config.floatX)
            b = theano.shared(value=b_values, name='b', borrow=True)
        self.W = W
        self.b = b
        lin_output = T.dot(input, self.W) + self.b
        self.output = (lin_output if activation is None
                       else activation(lin_output))
        # parameters of the model
        self.params = [self.W, self.b]
```

As it can be seen, the constructor of the *HiddenLayer* class accepts symbolic variables W (Weight) and b (Bias). If they are provided to the class, the model parameters are initialized to these variables otherwise they will be initialized to random numbers in a specific interval (Please see [**?**] on how this interval is chosen).

And finally, we need to stack these hidden layers on top of each other and add a logistic regression layer as the final layer. Code snippet below shows how this is done in Theano:

```
for i in xrange(self.n_layers):
            if i == 0:
                input_size = n_ins
            else:
                input_size = hidden_layers_sizes[i - 1]
            if i == 0:
                layer_input = self.x
            else:
                layer_input = self.sigmoid_layers[-1].output
            sigmoid_layer = HiddenLayer(rng=numpy_rng,
                                        input=layer_input,
                                        n_in=input_size,
                                        n_out=hidden_layers_sizes[i],
                                        activation=T.nnet.sigmoid)

```

```
            self.sigmoid_layers.append(sigmoid_layer)
            self.params.extend(sigmoid_layer.params)
            rbm_layer = RBM(numpy_rng=numpy_rng,
                            theano_rng=theano_rng,
                            input=layer_input,
                            n_visible=input_size,
                            n_hidden=hidden_layers_sizes[i],
                            W=sigmoid_layer.W,
                            hbias=sigmoid_layer.b)
            self.rbm_layers.append(rbm_layer)

        self.logLayer = LogisticRegression(
            input=self.sigmoid_layers[-1].output,
            n_in=hidden_layers_sizes[-1],
            n_out=n_outs)
        self.params.extend(self.logLayer.params)
```

This way, after the pre-training phase is done, we can treat the validation set as a new training set and perform the "fine tuning" on the multilayer perceptron. Hence, after the training we have a classifier that recognizes the cover of *patches*.

3.3.1 Classifying a grid within the Image:

In order to solve the original problem of annotating the image with the percentage of each cover we break the image into a grid of patches and then feed each "cell" in the grid to the classifier to decide the dominant cover in that patch. By counting the number of cells of each class, we can compute an approximate coverage percentage. This idea is depicted for a sample image in Figure 5. The python code for the function that performs this visualization can be seen below:

```
def vis_annotation(file_name,helper):
    image = Image.open(file_name).convert("RGBA")
    poly_size = (28, 28)
    for key in helper:
        (i, j) = key
        poly_offset = (i*28, j*28)
        poly = Image.new('RGBA', poly_size)
        pdraw = ImageDraw.Draw(poly)
        if helper[key] == 0:
            pdraw.polygon([(0, 0), (28, 0), (28, 28), (0, 28)],
            fill=(255, 0, 0, 127), outline=(255,255,255,255))
        else:
            pdraw.polygon([(0, 0), (28, 0), (28, 28), (0, 28)],
             fill=(0, 0, 255, 127), outline=(255,255,255,255))
        image.paste(poly, poly_offset, mask=poly)
    image.save("final.jpg")
```

We note that since the sample patches were chosen uniformly from images (with high percentage of each cover) it doesn't really matter how one breaks the image into patches and if higher prediction resolution is needed we can break the image into overlapping patches and take a consensus on prediction of the classifier for overlapping patches.

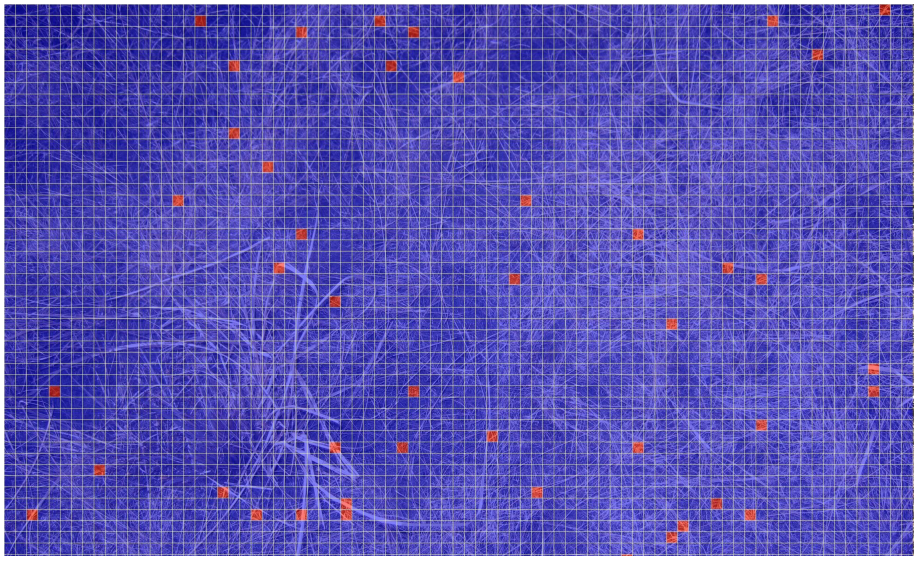


Figure 5: Annotating an image using the trained classifier for patches. The highlight color specifies the class predicted for each patch.

4 Results

4.1 Progression of feature vectors with experimentation:

Our first experiment was to train the RBM on thumbnails of 300×225 pixels from the original images. As we expected, all the filters were pure noise and no pattern could be observed. Thus, we decided to focus the training on a particular type of ground cover and did the pre-training on the patches of grass with different dataset sizes and different network topologies. Table 1 summarizes the results.

Table 2 shows the filter images, for a specific architecture and dataset size, at different epochs during the training.

4.2 Classifying using a deep rBM network:

Next we chose to train a deep belief network on a dataset with 30K grass patches and 30K bare patches and computed the accuracy. The belief network consists of 3 layers of hidden units each with 500 hidden units. Each layer has been trained for 100 epochs. An then fine tuning was performed using early stopping. The best model had accuracy 70.04% on the test dataset.

And finally we tried to annotate an image with grass (95%) and bare (5%) cover using the the trained classifier. Unfortunately we ran out of time and didn't perform the experiment (each run of the DBN took around 26 hours on a dataset of size $50K$). But all the necessary codes (breaking an image into patches and visualizing the predicting on the image) are ready. (Figure 5).

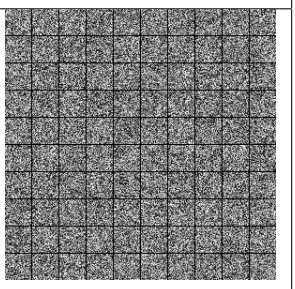
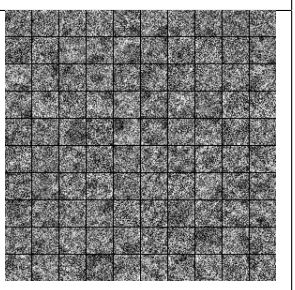
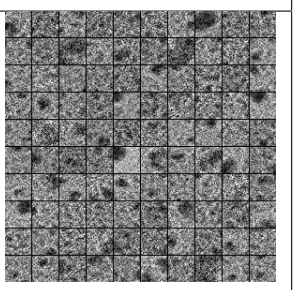
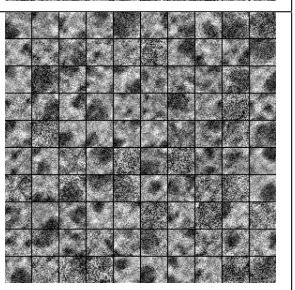
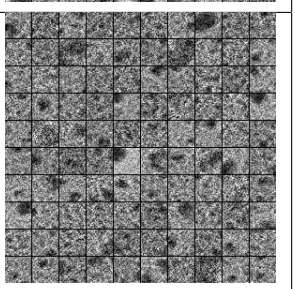
Network Architecture	Number of Samples	Filters
784 visible, 500 hidden	1000	
784 visible, 1000 hidden	20000	
784 visible, 1000 hidden	30000	
784 visible, 100 hidden	60000	
784 visible, 500 hidden	60000	

Table 1: Filters for different topologies of the rBM and different training dataset size. As the number of training samples is increased the patterns in the filters become more clear. Also as the number of hidden units is increased the complexity of patterns is increased.

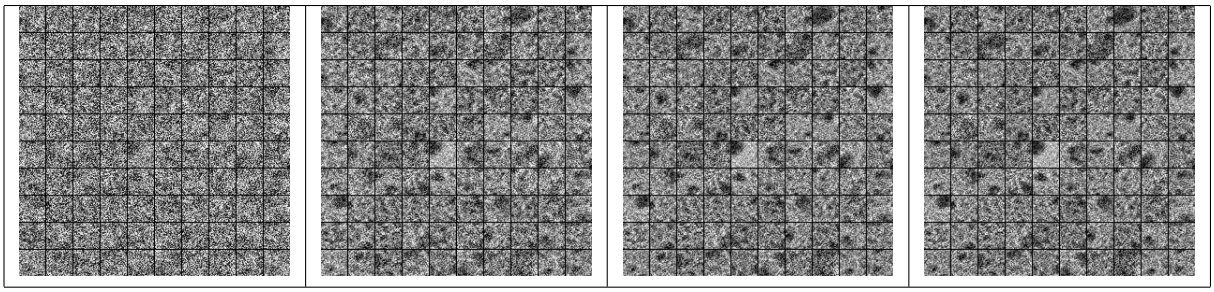


Table 2: Evolution of the filters through out the training. Images are taken at epochs 0,4,9,13 respectively (From left to right) for a dataset with 60000 grass samples and 200 hidden units

5 Conclusion

We started out this project with a difficult problem and finished with a deeply hardened set of tools with which to approach and attack many more.

From incredible python libraries allowing access to powerful processing units to conceptual novelties that represent data in layered abstractions, this small project has deeply engendered our becoming stronger mathematicians and computer scientists.

We take away the practical skills of implementing a novel unsupervised training algorithm that seeks to thwart some of the longest-standing arguments against neural networks: local minimum and its foggy processes. Through rBM's we can seek to approach the global minimum of the feature space in an unsupervised and efficient manner. We can also reproduce sample images of the feature vectors which allows us to see the "thoughts" of the machine as the epochs progress. These allow us to better our attempts at success in both of the mainstream modeling goals: that of predictive accuracy and of explanation of the natural forces behind the data generation process.

Neither of us had exposure to these models before this course, and yet they can be applied to so many fields and in so many ways. As fickle and quirky as these models can be, we now both take intuition of them along with our practical implementation experience. These nuances of intuition are seemingly as important as an understanding of the mechanics, as the unseen behavior of the model is what really needs to be controlled for.

These models have applications for which they are very finely suited, as well as others for which their advantages are of less obvious nature. We believe that they have sufficient flexibility, nonetheless, to adapt to a very wide spectrum and class of problems, and -without a doubt- provide an indispensable tool to the data modeler's toolbox.

References

[1] Yoshua Bengio. Learning deep architectures for ai. *Technical Report 1312*.

[2] G.E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1711–1800, 2002.

[3] G.E. Hinton. A practical guide to training restricted boltzmann machines. *http://learning.cs.toronto.edu*, Version: 1, 2010.

[4] Geoffrey E. Hinton. Learning multiple layers of representation. *TRENDS in Cognitive Sciences*, 11:428–434, 2008.

[5] Osindero S. Hinton, G.E. and Teh Y.W. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006.

[6] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79.

[7] LISA lab. Deep learning tutorial. Release 0.1.

[8] G. Mayraz and G.E. Hinton. Recognizing hand-written digits using hierarchical products of experts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24:189–197, 2001.

[9] R.R. Salakhutdinov and G.E. Hinton. Deep boltzmann machines. *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 12, 2009.

[10] R.R. Salakhutdinov and H. Larochelle. Efficient learning of deep boltzmann machines. *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 13.

[11] I. Sutskever and Tieleman. On the convergence properties of contrastive divergence. *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, Sardinia, Italy, 2010.

Appendices

A RBM Class Code

```
1
2  import time
3  import numpy as np
4  import PIL.Image
5  import numpy
6  import theano
7  import theano.tensor as T
8  import os
9
10 from theano.tensor.shared_randomstreams import RandomStreams
11 from load_data import load_ground_cover
12 from utils import tile_raster_images
13 from logistic_sgd import load_data
14
15
16 class RBM(object):
17     def __init__(self, input=None, n_visible=784, n_hidden=500, \
18         W=None, hbias=None, vbias=None, numpy_rng=None,
19         theano_rng=None):
20
21         self.n_visible = n_visible
22         self.n_hidden = n_hidden
23         if numpy_rng is None:
24             # create a number generator
25             numpy_rng = numpy.random.RandomState(1234)
26         if theano_rng is None:
27             theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))
28         if W is None:
29             initial_W = numpy.asarray(numpy_rng.uniform(
30                       low=-4 * numpy.sqrt(6. / (n_hidden + n_visible)),
31                       high=4 * numpy.sqrt(6. / (n_hidden + n_visible)),
32                       size=(n_visible, n_hidden)),
33                       dtype=theano.config.floatX)
34             W = theano.shared(value=initial_W, name='W', borrow=True)
35
36         if hbias is None:
37             hbias = theano.shared(value=numpy.zeros(n_hidden,
38                                                     dtype=theano.config.floatX),
39                                   name='hbias', borrow=True)
40         if vbias is None:
41             vbias = theano.shared(value=numpy.zeros(n_visible,
42                                                     dtype=theano.config.floatX),
43                                   name='vbias', borrow=True)
44         self.input = input
45         if not input:
46             self.input = T.matrix('input')
47         self.W = W
48         self.hbias = hbias
49         self.vbias = vbias
```

```
50          self.theano_rng = theano_rng
51          self.params = [self.W, self.hbias, self.vbias]
52
53      def free_energy(self, v_sample):
54          wx_b = T.dot(v_sample, self.W) + self.hbias
55          vbias_term = T.dot(v_sample, self.vbias)
56          hidden_term = T.sum(T.log(1 + T.exp(wx_b)), axis=1)
57          return -hidden_term - vbias_term
58
59      def propup(self, vis):
60
61          pre_sigmoid_activation = T.dot(vis, self.W) + self.hbias
62          return [pre_sigmoid_activation, T.nnet.sigmoid(pre_sigmoid_activation)]
63
64      def sample_h_given_v(self, v0_sample):
65
66          pre_sigmoid_h1, h1_mean = self.propup(v0_sample)
67          h1_sample = self.theano_rng.binomial(size=h1_mean.shape,
68                                               n=1, p=h1_mean,
69                                               dtype=theano.config.floatX)
70          return [pre_sigmoid_h1, h1_mean, h1_sample]
71
72      def propdown(self, hid):
73          pre_sigmoid_activation = T.dot(hid, self.W.T) + self.vbias
74          return [pre_sigmoid_activation, T.nnet.sigmoid(pre_sigmoid_activation)]
75
76      def sample_v_given_h(self, h0_sample):
77          pre_sigmoid_v1, v1_mean = self.propdown(h0_sample)
78          v1_sample = self.theano_rng.binomial(size=v1_mean.shape,
79                                               n=1, p=v1_mean,
80                                               dtype=theano.config.floatX)
81          return [pre_sigmoid_v1, v1_mean, v1_sample]
82
83      def gibbs_hvh(self, h0_sample):
84          pre_sigmoid_v1, v1_mean, v1_sample = self.sample_v_given_h(h0_sample)
85          pre_sigmoid_h1, h1_mean, h1_sample = self.sample_h_given_v(v1_sample)
86          return [pre_sigmoid_v1, v1_mean, v1_sample,
87                  pre_sigmoid_h1, h1_mean, h1_sample]
88
89      def gibbs_vhv(self, v0_sample):
90          pre_sigmoid_h1, h1_mean, h1_sample = self.sample_h_given_v(v0_sample)
91          pre_sigmoid_v1, v1_mean, v1_sample = self.sample_v_given_h(h1_sample)
92          return [pre_sigmoid_h1, h1_mean, h1_sample,
93                  pre_sigmoid_v1, v1_mean, v1_sample]
94
95      def get_cost_updates(self, lr=0.1, persistent=None, k=1):
96          pre_sigmoid_ph, ph_mean, ph_sample = self.sample_h_given_v(self.input)
97
98          if persistent is None:
99              chain_start = ph_sample
100         else:
101             chain_start = persistent
102
103         [pre_sigmoid_nvs, nv_means, nv_samples,
```

```
104             pre_sigmoid_nhs, nh_means, nh_samples], updates = \
105                 theano.scan(self.gibbs_hvh,
106                         outputs_info=[None,  None,  None, None, None, chain_start],
107                         n_steps=k)
108
109         chain_end = nv_samples[-1]
110         cost = T.mean(self.free_energy(self.input)) - T.mean(
111             self.free_energy(chain_end))
112         gparams = T.grad(cost, self.params, consider_constant=[chain_end])
113
114         for gparam, param in zip(gparams, self.params):
115             updates[param] = param - gparam * T.cast(lr,
116                                                     dtype=theano.config.floatX)
117         if persistent:
118             updates[persistent] = nh_samples[-1]
119             monitoring_cost = self.get_pseudo_likelihood_cost(updates)
120         else:
121             monitoring_cost = self.get_reconstruction_cost(updates,
122                                                            pre_sigmoid_nvs[-1])
123
124         return monitoring_cost, updates
125
126     def get_pseudo_likelihood_cost(self, updates):
127         bit_i_idx = theano.shared(value=0, name='bit_i_idx')
128         xi = T.round(self.input)
129         fe_xi = self.free_energy(xi)
130         xi_flip = T.set_subtensor(xi[:, bit_i_idx], 1 - xi[:, bit_i_idx])
131         fe_xi_flip = self.free_energy(xi_flip)
132         cost = T.mean(self.n_visible * T.log(T.nnet.sigmoid(fe_xi_flip -
133                                                             fe_xi)))
134         updates[bit_i_idx] = (bit_i_idx + 1) \% self.n_visible
135         return cost
136
137     def get_reconstruction_cost(self, updates, pre_sigmoid_nv):
138         cross_entropy = T.mean(
139                 T.sum(self.input * T.log(T.nnet.sigmoid(pre_sigmoid_nv)) +
140                 (1 - self.input) * T.log(1 - T.nnet.sigmoid(pre_sigmoid_nv)),
141                       axis=1))
142
143         return cross_entropy
144
145
146 def test_rbm(learning_rate=0.1, training_epochs=15,
147              dataset='mnist.pkl.gz', batch_size=20,
148              n_chains=20, n_samples=10, output_folder='rbm_plots',
149              n_hidden=500):
150
151     datasets = load_ground_cover('../data/groundcover', '../data/groundcover/ground_co
152     train_set_x, train_set_y = datasets[0]
153     test_set_x, test_set_y = datasets[1]
154     n_train_batches = train_set_x.get_value(borrow=True).shape[0] / batch_size
155
156     index = T.lscalar()
157     x = T.matrix('x')  # the data is presented as rasterized images
```

```
    rng = numpy.random.RandomState(123)
    theano_rng = RandomStreams(rng.randint(2 ** 30))
    persistent_chain = theano.shared(numpy.zeros((batch_size, n_hidden), dtype=theano.
    rbm = RBM(input=x, n_visible=28 * 28,
              n_hidden=n_hidden, numpy_rng=rng, theano_rng=theano_rng)

    cost, updates = rbm.get_cost_updates(lr=learning_rate,
                                         persistent=persistent_chain, k=25)

    if not os.path.isdir(output_folder):
        os.makedirs(output_folder)
    os.chdir(output_folder)

    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 = time.clock()

    for epoch in xrange(training_epochs):
        mean_cost = []
        for batch_index in xrange(n_train_batches):
            mean_cost += [train_rbm(batch_index)]
        print 'Training epoch %d, cost is ' % epoch, numpy.mean(mean_cost)
        plotting_start = time.clock()
        image = PIL.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 = time.clock()
        plotting_time += (plotting_stop - plotting_start)
    end_time = time.clock()
    pretraining_time = (end_time - start_time) - plotting_time
    print ('Training took %f minutes' % (pretraining_time / 60.))
    number_of_test_samples = test_set_x.get_value(borrow=True).shape[0]

    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))

    plot_every = 1000
    [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)


    updates.update({persistent_vis_chain: vis_samples[-1]})
```

```
    sample_fn = theano.function([], [vis_mfs[-1], vis_samples[-1]],
                                updates=updates,
                                name='sample_fn')

    image_data = numpy.zeros((29 * n_samples + 1, 29 * n_chains - 1),
                             dtype='uint8')
    for idx in xrange(n_samples):
        vis_mf, vis_sample = sample_fn()
        print ' ... plotting sample ', 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))

    image = PIL.Image.fromarray(image_data)
    image.save('samples.png')
    plt.imshow(image_data)
    plt.show()
    os.chdir('../')

if __name__ == '__main__':
    test_rbm()
```