Restricted Boltzmann Machines

Project 3 - Math-6373 - Prof. Azencott

Simon Stolarczyk - PSID: 1201831
Adrian Radillo - PSID: 1328335

April 3, 2017

GitHub: The Python code used for this project is available at the public repository, `https://github.com/Spstolar/BMachine.git`

1 Generic Boltzmann Machines

1.1 Gibbs sampler for a Boltzmann machine

For a given configuration $\boldsymbol{X}$ of the machine, let $X(i), X(j) \in \{-1, 1\}$ denote the states of nodes i and j respectively, and $W_{ij} = W_{ji}$ the weight between them ($i, j \in \{1, \ldots, 30\}$).

Let σ be the sigmoid function and define

$$v_i = \sum_{j \neq i} W_{ij} X(j)$$

The Boltzmann machine stochastic dynamics by a Gibbs sampler corresponds to algorithm 1 below.

Algorithm 1 Gibbs sampler

```
1: $totSweeps \leftarrow n$                                  # Set the total number of sweeps
2: for $sweep$ in $\{1, \ldots, totSweeps\}$ do
3:     Select $\alpha$ to be a random permutation of the nodes indices $\{1, 2, \ldots, 30\}$
4:     for $i$ in $\alpha$ do                                  # loop over nodes
5:         $X(i) \leftarrow 1$ with probability $\sigma(v_i)$ and $-1$ otherwise    # refresh state
6:     end for
7: end for
```

1.2 Results of generic BM

In appendix A.3, we include the code written for creating and running a BM according to the Gibbs sampler. We run the machine twice, once with random initial weights, and a second time with these same weights scaled. These two Boltzmann machines were simulated until stabilization, after which we further simulated them for an extra 200 sweeps in order to compute their energy histograms.

1.2.1 Stabilization times

The stabilization times for both sets of initial weights were very similar: $S = 112$ sweeps for the original experiment and $S = 111$ sweeps when the initial weights were rescaled by $\frac{1}{10}$.

In order to visualize the evolution of the empirical means of each node in the machines, for each node j, and for each time step $t < S$ preceding stabilization, we color-plotted in figure 1, the boolean condition:

$$|M_t(j) - M_{t-1}(j)| < \text{THR}.$$

In figure 1, the purple color signifies that the condition is False and yellow that it is True. We observe very comparable stabilization times for both machines, but a great discrepancy in the time course of the stabilization state of the nodes. The machine with larger weights displays a greater proportions of stabilized nodes from the first sweep. On the other hand, the machine with smaller weights spends most of the first 100 sweeps with all its nodes 'un-stabilized'. We are not sure of how to explain this phenomenon.

[A: Anything clever to add?]

1.2.2 Energy of stabilized BM

The Boltzmann energy of a configuration $\boldsymbol{X} = (X(1), \ldots, X(m))$ is defined by:

$$E(\boldsymbol{X}) := -\sum_{i<j} W_{ij} X(i) X(j) - \sum_{i=1}^{m} b_i X(i), \tag{1}$$

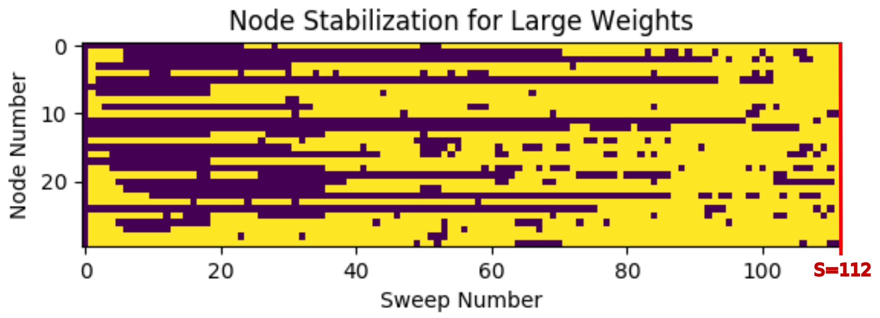
where the first sum is over all pairs of distinct nodes and the $b_i \in \mathbb{R}$ are the thresholds associated to each node.

In figure 2, we present the histograms of the 200 energies, computed over the 200 sweeps following the stabilization time, S, of each machine. We observe that the distribution of energies is narrower and with mean closer to zero, when the initial weights are rescaled by $\frac{1}{10}$.**[A: Explanation?]**

2 Database

We chose the automatic hand-written digits classification task with the MNIST database. Each image is composed of 28 x 28 = 784 pixels. In the original dataset, each pixel is encoded as an 8-bit binary word, which codes for an integer value between 0 and 255, representing the gray scale intensity.

Given an image, each one of these 784 intensity values constitutes a feature, and each image belongs to one of the ten classes $\{0, 1, \ldots, 9\}$.



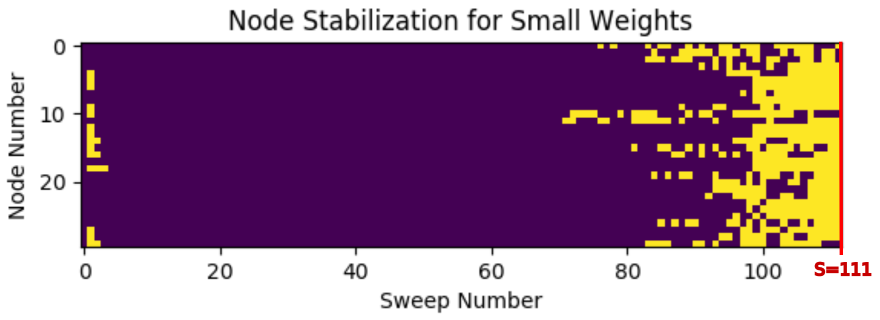
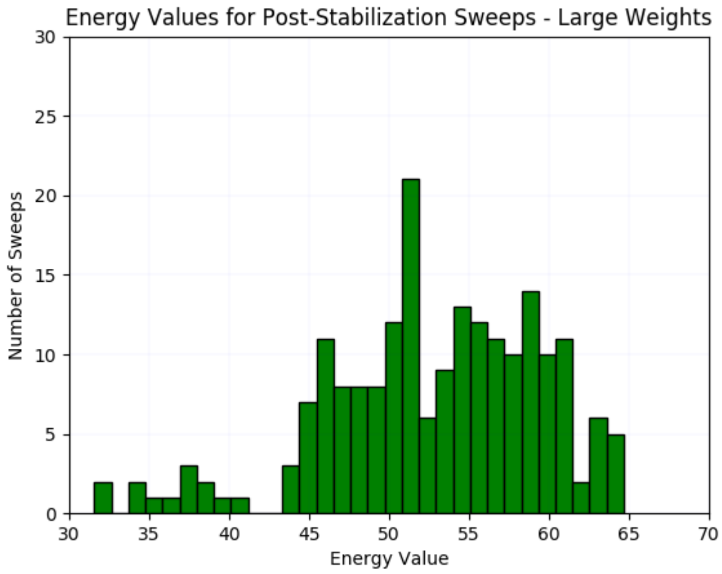


Figure 1: Time course of the stabilization status of each node during training, for the large- (top plot) and small-weight (bottom plot) cases. Purple color (not stabilized) means that the increment in the empirical mean is above the THR=1% threshold, whereas yellow color means that it is below.



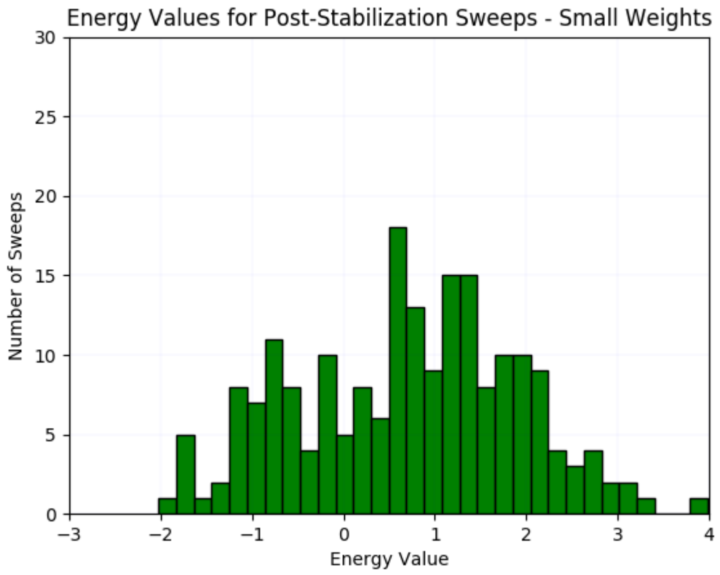


Figure 2: Histograms of energies computed over the 200 sweeps after stabilization times for large (left) and small (right) weights.

To both reduce the input size of our network and render the features interpretable by a Boltzmann machine (which traditionally is made of binary units), we followed the strategy of [1] which consists in setting all the pixels with intensity value smaller than 128 to 0 and all the remaining ones to 1. The code we wrote for this extraction and pre-processing of the data is included in appendix A.1. An example image after this 'binarization' was applied, is presented in figure 3. The code used to produce the image is included in appendix A.2.

Our training and test sets contain 60,000 and 10,000 cases respectively.

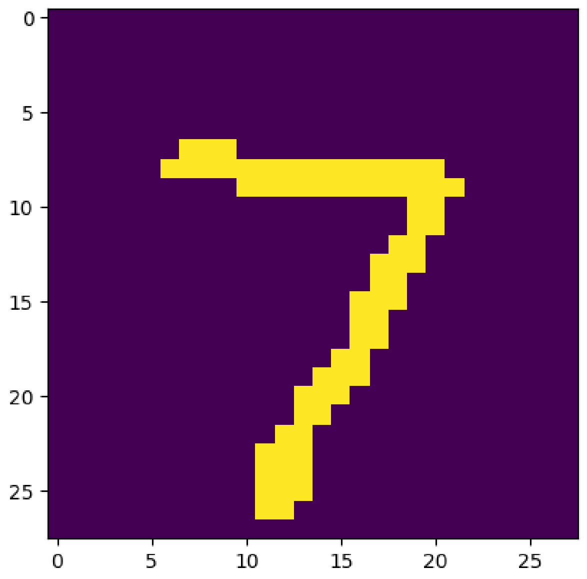


Figure 3: Example from the MNIST database, after undergoing our 'binarization' procedure.

3 Stochastic Auto-encoders based on the restricted Boltzmann machine (RBM)

After our pre-processing stage, each case from our dataset contains 1 bit of information per feature, and 784 features. It is therefore straight-forward to encode such data into the input layer of our RBM. We choose $n_3 = n_1 = 784$ and each bit from the image is mapped onto its RBM unit counterpart according to the rule: $0 \mapsto -1$ & $1 \mapsto 1$.

Our two tentative values of $h < n_1$ are 100 and 500.

3.1 Implement fast RBM learning algorithm to train the 2 auto-encoders

Our fast RBM training algorithm is summarized in Algorithm 2.

3.1.1 Batch construction

We train our RBM in the following way. First, we fix a batch size (500), a batch overlap size (100) and a total number of iterations (20) for the training. We call an iteration a presentation of the whole training set (including some redundancy[1] due to overlap between batches). Our `training()` method produces the batches and is exposed in appendix A.3.

We update the weights between each batch presentation, after having run the machine in the clamped and un-clamped mode for each example in the batch. The weight increment is based on the difference between the coactivities in both modes.

Algorithm 2 Fast RBM training

```
1: initialize RBM
2: $batchSize \leftarrow m$                                   # batch size
3: $batchOverlap \leftarrow \nu$                              # batch overlap size
4: $iterations \leftarrow K$          # number of times training set is presented
5: for $iter$ in $\{1, \dots, iterations\}$ do
6:     produce batches
7:     initialize $coactClamped$ & $coactUnclamped$           # coactivity matrices
8:     for $batch$ in batches do
9:         $timeStep \leftarrow$ number of batches seen since time 0
10:        for $example$ in $batch$ do
11:            clamped_run($example$)                          #1 sweep
12:            update $coactClamped$                           # running average
13:            unclamped_run($example$)                        #1 sweep
14:            update $coactUnclamped$                         # running average
15:        end for
16:    end for
17:    $dW_{ij} \leftarrow \eta \cdot (coactClamped_{ij} - coactUnclamped_{ij})$
18:    $W_{ij} \leftarrow W_{ij} + dW_{ij}$
19: end for
```

[1] Minor detail: for the last batch, if its size goes beyond the last case in the training set, we wrap the batch around the training set and append the first few examples from the training set that are needed to complete the batch.

3.2 Detailed analysis of hidden layer structure and efficiency

3.3 PCA analysis

We include the code for running principal component analysis in appendixA.4. Once we have a stable run of our machine we can easily plug in the data to get images of the hidden layer structure for the activations across inputs.

3.4 Autoencoding efficiency

Our code is quite flexible. So, once we figure out the issues with the machine we can easily tack on this classification layer and run the task.

References

[1] Asja Fischer and Christian Igel. Training restricted Boltzmann machines: An introduction. *Pattern Recognit.*, 47(1):25–39, 2014.

Appendix A Python code

A.1 Convert MNIST binary data into numpy arrays

The following script was used to convert the image datasets into numpy arrays.

```
import numpy as np

# url to format of files: http://yann.lecun.com/exdb/
    mnist/
# definition of constants corresponding to the MNIST
    data sets
num_train_examples = 60000
num_test_examples = 10000
num_classes = 10
pixels_per_image = 784
bits_per_pixel = 8
header_bytes_images = 16
header_bytes_labels = 8

def bits(f):
    bytes = (ord(b) for b in f.read())  # stores all
        bytes in the file
    for b in bytes:
        '''
        The following line only returns the last bit
            of every byte
        The yield command returns a generator as well explained here:
        http://stackoverflow.com/questions/231767/what-does-the-yield-keyword-do-in-python
        '''
        yield (b >> 7) & 1  # I am not quite sure what the & 1 is for


def extract_images_bin2py(num_examples, header_bytes, filename, savefile, pixels_per_image=784):
    dataset = np.zeros((num_examples, pixels_per_image))

    byte_count = 1

    for b in bits(open(filename, 'r')):

        if byte_count <= header_bytes:  # Skip header bytes
            pass

        else:
            corrected_byte_count = byte_count - header_bytes - 1
            image = corrected_byte_count / pixels_per_image  # Which example this belongs to
            pixel = corrected_byte_count % pixels_per_image  # Which pixel this belongs to
            dataset[image, pixel] = b

        byte_count = byte_count + 1
    np.save(savefile, dataset)

# arguments for training images
# filename='train-images-binary'
# savefile= 'trainSet.npy'

#arguments for test images
filename = 'test-images-binary'
savefile = 'testSet.npy'

extract_images_bin2py(num_test_examples, header_bytes_images, filename, savefile)
```

The following script was used to convert the label datasets into numpy arrays.

```
import numpy as np


header_length = 64
num_examples = 60000
pixels_per_pic = 784
bits_per_label = 8
bits_per_pixel = 8
pic_length = pixels_per_pic * bits_per_pixel

test_label_binary = np.zeros((num_examples ,
    bits_per_label))
test_label_classes = np.zeros((num_examples , 10))

def bits(f):
    bytes = (ord(b) for b in f.read())
    for b in bytes:
        for i in xrange(8):
            yield (b >> i) & 1

i = 1
pic_count = 0

for b in bits(open('train-labels-binary', 'r')):
    if i <= header_length:   # Preamble
        pass
    else:
        example = (i - header_length - 1) /
            bits_per_label  # Which example this
            belongs to.
        bit = ((i - header_length - 1) %
            bits_per_label)  # Which column/bit
        test_label_binary[example , bit] = b

    i += 1

for j in range(0,num_examples):
    class_label = 0
    for k in range(0,8):
        class_label += test_label_binary[j,k] * (2 **
            k)
    test_label_classes[j,class_label] = 1  # May want
        to change this to class_label.asType(int) or
        something similar.


np.save("train_labels.npy", test_label_classes)
```

A.2 Visualize digits from our numpy arrays

The following code was used to produce figure 3.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np

length = 28

dog = np.load('stabilization_small.npy')

num_plot = 0

if num_plot == 1:
    for j in range(10):
        img = dog[j,:].reshape((length, length))  #
            for printing numbers
        # img = np.random.randint(0,2,size=(length,
            length))
        imgplot = plt.imshow(img)
        plt.savefig('example' + str(j) + '.png')
else:
    small_plot = np.load('stabilization_small.npy')
    img = small_plot[:112,:].T
    imgplot = plt.imshow(img)
    plt.title('Node Stabilization for Small Weights')
    plt.xlabel('Sweep Number')
    plt.ylabel('Node Number')
    plt.savefig('stabilization_small.png')

    plt.clf()

    large_plot = np.load('stabilization_large.npy')
    img = large_plot[:112, :].T
    imgplot = plt.imshow(img)
    plt.title('Node Stabilization for Large Weights')
    plt.xlabel('Sweep Number')
    plt.ylabel('Node Number')
    plt.savefig('stabilization_large.png')
```

A.3 Create, run and train a BM

The following script defines a *BoltzmannMachine* class in Python which contains several methods and properties. The main ones are described below:

- Create an instance (BoltzmannMachine(Nodes))
- Initialize weights (create_random_weights())
- Run the Gibbs sampler (run_machine())
- Store history of configurations (.history)
- Store history of energies of the system after stabilization (.energy_history)

The end of the script, after the class definition, performs all the computations used in section 1 and stores the energy and stabilization history in two numpy arrays.

```python
import numpy as np
import time

'''
Parameters to modulate:
    Learning rate = how much to change the weights by for each batch
    Batch size = how many examples to use for weight change
    Coactivity sweeps = how many sweeps to compute coactivity for given weights
    Readout sweeps = how many sweeps to run before getting a readout
    Size of training set = how many training examples you'll use
    iterations = how many times to run through the training set
'''

def sigmoid(input_comb):
    return 1.0 / (1 + np.exp(-input_comb))


def rand_bern(length):
    # Return a random vector of -1s and 1s.
    rand_vec = np.random.randint(0, 2, length, dtype=int)  # Begin with a random 0-1 draw.
    return (rand_vec - .5) * 2  # Convert to -1, +1 state.

```

```python

def rand_bern_with_thresh(length, fix1, fix2):
    # Return a random vector of -1s and 1s.
    rand_vec = np.random.randint(0, 2, length, dtype=int)  # Begin with a random 0-1 draw.
    rand_vec[fix1] = 1
    rand_vec[fix2] = 1
    return (rand_vec - .5) * 2  # Convert to -1, +1 state.


def convert_binary_to_pm1(matrix):
    """
    Convert a 0/1 matrix to a -1/1 matrix.
    :param matrix: A binary matrix.
    :return: Converted matrix.
    """
    converted_matrix = (matrix - 0.5) * 2
    return converted_matrix


class BoltzmannMachine(object):
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size + 1
        self.hidden_size = hidden_size + 1
        self.output_size = output_size
        self.total_nodes = self.input_size + self.hidden_size + self.output_size
        self.hidden_ind = self.input_size  # coordinate index where the hidden layer STARTS
        self.out_ind = self.input_size + self.hidden_size  # coordinate index where the output layer STARTS
        self.input_thresh = self.hidden_ind - 1
        self.hidden_thresh = self.out_ind - 1
        self.hidden_nodes = np.arange(self.hidden_ind, self.hidden_thresh)
        self.out_nodes = np.arange(self.out_ind, self.total_nodes)
        self.clamped_visit_list = self.hidden_nodes
        self.unclamped_visit_list = np.hstack((self.hidden_nodes, self.out_nodes))

        self.state = rand_bern_with_thresh(self.
            total_nodes, self.input_thresh, self.
            hidden_thresh)

        self.weights = self.create_random_weights()
        self.correct_weights()

        self.in_to_hidden = np.ix_(np.arange(self.
            hidden_ind), self.hidden_nodes)
        self.hidden_to_hidden = np.ix_(self.
            hidden_nodes, self.hidden_nodes)
        self.hidden_to_out = np.ix_(np.arange(self.
            hidden_ind,self.out_ind), self.out_nodes)

        self.batch_size = 500
        self.inc = 400
        self.learning_rate = .05
        self.rate = self.learning_rate

        self.history = self.state
        self.sweeps = 100
        self.stabilization = np.zeros((self.sweeps,
            self.total_nodes))
        self.threshold = .01
        self.energy_history = np.zeros(200)

    def print_current_state(self):
        print self.state

    def state_energy(self):
        """
        Computes the current energy of the system.
        :return: Current system energy.
        """
        agreement_matrix = np.outer(self.state, self.
            state)  # The (i,j) entry is 1 if i,j agree
            , else -1
        energy_contributions = agreement_matrix * self
            .weights  # Element-wise product.
        energy = 0.5 * np.sum(energy_contributions)  #
             Leaving off bias.
        return energy

    def state_prob(self):
        """
        The (non-normalized) probability of this
            configuration. Does the whole calculation
            rather than just over some
        affected subsets.
        :return: conditional probability of this
        """
        return np.exp(-self.state_energy())

    def conditional_prob(self, node):
        lin_sum_neighbors = np.dot(self.weights[node,
            :], self.state)
        return sigmoid(lin_sum_neighbors)

    def update(self, node):
        """
        Probabilistically update a single node fixing
            all others.
        :param node: The number of the node to update.
        :return: Change the state of the node
            according to the probabilities of the two
            alternatives.
        """
        plus_prob = self.conditional_prob(node)    # P
            ( x_j = 1 |  all other node states)
        coin_flip = np.random.binomial(1, plus_prob)
        result = 2*(coin_flip - .5)  # Convert biased
            coin flip to -1 or 1.
        self.state[node] = result

    def simultaneous_update(self, mle=0):
        """
        Update ALL nodes at once. This currently will
            update everything, including the input/
            output nodes.
        :param mle: Whether to pick the most likely
            state of each node (1), or to use the
            probabilistic update (0).
        :return: Updates the state vector.
        """
        new_state = self.state
        if mle == 1:
            for node in range(0, self.total_nodes):
                new_state[node] = self.mle_update(node
                    )
            self.state = new_state
        elif mle == 0:
            for node in range(0, self.total_nodes):
                plus_prob = self.conditional_prob(node
                    )  # P( x_j = 1 |  all other node states)
                coin_flip = np.random.binomial(1, plus_prob)
                result = 2 * (coin_flip - .5)  # Convert biased coin flip to -1 or 1.
                new_state[node] = result
            self.state = new_state

    def mle_update(self, node, alter=1):
        """
        Update a node to it's highest probability state.
        :param node: Which node to update.
        :param alter: Actually change the node (1), or just return the most likely state (0).
        :return: The most likely state for that given node.
        """
        if alter == 1:
            if self.conditional_prob(node) > .5:
                self.state[node] = 1
            else:
                self.state[node] = -1
        elif alter == 0:
            if self.conditional_prob(node) > .5:
                return 1
            else:
                return -1

    def run_machine(self, sweep_num, stabilized=0):
        """
        For sweep_num passes or until it stabilizes, update each of the nodes, except the inputs and thresholds.
        :param sweep_num: A maximum number of times to update each node.
        :param stabilized: Update the machine until it stabilizes (0), or just run the machine the given amount (1).
        :return:
        """
        for sweep in range(sweep_num):
            np.random.shuffle(self.unclamped_visit_list)
            for node in self.unclamped_visit_list:
                self.update(node)
            if stabilized == 0:
                if self.stabilization_check(sweep) ==
                   1:
                    break
            # if stabilized == 1:
            #     self.history = np.vstack((self.
               history, self.state))
            #     self.energy_history[sweep] = self.
               state_energy()

    def stabilization_check(self, sweep):
        """
        Check to see if the machine has stabilized by
           this sweep.
        :param sweep: Which sweep of updates this is
           checking after.
        :return: Whether or not stabilization of the
           mean activation has occurred for 90% or
           more of the nodes.
        """
        prev_mean = self.empirical_mean()
        self.history = np.vstack((self.history, self.
           state))
        current_mean = self.empirical_mean()
        difference = np.abs(current_mean - prev_mean)
        self.stabilization[sweep, :] = np.less(
           difference, self.threshold)
        minimum_stabilized = np.floor(self.total_nodes
            * .9)
        if (np.sum(self.stabilization[sweep, :]) >
           minimum_stabilized) & (sweep > 100):
            # print sweep
            # print self.stabilization[sweep, :]
            return 1
        else:
            return 0

    def create_random_weights(self):
        weights = np.random.uniform(-1, 1, size=(self.
           total_nodes, self.total_nodes))  # Random
           weights ~ U([-1,1])
        weights = np.triu(weights, k=1)  # discard
           lower diagonal terms (and the diagonal to
           avoid self-connections)
        weights = weights + weights.T  # make the weights symmetric
        return weights

    def correct_weights(self):
        self.weights[:self.hidden_ind, :self.hidden_ind] = 0  # forbids connections INP <-> INP
        self.weights[-self.output_size:, -self.output_size:] = 0  # forbids connections OUT <-> OUT
        self.weights[-self.output_size:, :self.input_size] = 0  # forbids connections IN -> OUT
        self.weights[:self.input_size, -self.output_size:] = 0  # forbids connections IN <- OUT
        self.weights[self.hidden_thresh, :self.out_ind] = 0  # forbid hidden_thresh -> hidden & in
        self.weights[:self.out_ind, self.hidden_thresh] = 0  # forbid hidden -> hidden_thresh
        np.fill_diagonal(self.weights, 0)
        # self.weights[self.hidden_ind:, :self.hidden_ind] = 0
        # self.weights[:self.hidden_ind, self.hidden_ind:] = 0

    def check_weights(self):
        w = 0
        w += np.sum(self.weights[:self.hidden_ind, :self.hidden_ind])
        w += np.sum(self.weights[-self.output_size:, -self.output_size:])
        w += np.sum(self.weights[-self.output_size:, :self.input_size])
        w += np.sum(self.weights[:self.input_size, -self.output_size:])
        w += np.sum(self.weights[self.hidden_thresh, self.hidden_ind:self.out_ind])
        w += np.sum(self.weights[self.hidden_ind:self.out_ind, self.hidden_thresh])
        w += np.sum(self.weights.diagonal())
        print 'Sum of the non-connecting weights: ' + str(w)

    def empirical_mean(self, history=0):
        """
        Return the average state of each node for a
            given history.
        :param history: A record of the state of the
            machine over a given number of sweeps,
            defaults to the object
        history.
        :return: The mean activation of each node.
        """
        if history == 0:
            history = self.history
        return np.mean(history, axis=0)

    def clamped_run(self, in_state, out_state,
        sweep_num=1):
        """
        Runs the machine while forcing the input nodes
             and output nodes to stay the same.
        :param in_state: What the input node states
            are.
        :param out_state: What the output node states
            are.
        :param sweep_num: How many times to update the
             non-clamped nodes.
        :return: Updates the hidden states.
        """
        self.state[:self.input_thresh] = in_state
        self.state[self.out_ind:] = out_state
        for sweep in range(sweep_num):
            np.random.shuffle(self.clamped_visit_list)
            for node in self.clamped_visit_list:
                self.update(node)

    def unclamped_run(self, in_state, sweep_num=1):
        """
        Runs the machine while forcing the input nodes
             to stay the same.
        :param in_state: What the input node states
            are.
        :param sweep_num: How many times to update the
             non-clamped nodes.
        :return: Updates the hidden and output states.
        """
        self.state[:self.input_thresh] = in_state
        for sweep in range(sweep_num):
            np.random.shuffle(self.unclamped_visit_list)
            for node in self.unclamped_visit_list:
                self.update(node)

    def clamped_run_mle(self, in_state, out_state):
        # Update the machine using the maximum likelihood states.
        self.state[:self.input_thresh] = in_state
        self.state[self.out_ind:] = out_state
        for node in self.clamped_visit_list:
            self.mle_update(node)

    def unclamped_run_mle(self, in_state):
        self.state[:self.input_thresh] = in_state
        for node in self.clamped_visit_list:
            self.mle_update(node)

    def training(self, example_set, iterations):
        """
        Go through the entire example set for the given number of iterations. For each iteration, you divide the set
        into batches and then pass each batch to the batch_process method to update the weights. After each iteration
        you update the rate at which the weights are changed.
        :param example_set: A set of unlabelled examples.
        :param iterations: How many times to go through the set for the learning.
        :return: The machine's weights are updated.
        """
        # Compute how to go through the batches.
        batch_size = self.batch_size
        inc = self.inc  # This is to allow overlap between batches, let it be about 80% of the batch size.
        set_size = example_set.shape[0]  # How many examples.
        batches_per_iteration = int(set_size / inc) + 1  # How many batches will be needed.
        record_mse = 0
        ramse = np.zeros(iterations*batches_per_iteration)
        testing = 1

        for it in range(iterations):
            np.random.permutation(example_set)
            print "Iteration: " + str(it)
            last_batch_ind = 0  # initialize variable for row index of last batch
            for batch_num, b in enumerate(range(0, set_size - inc, inc)):
                batch = example_set[b:b + batch_size, :]
                # To do learning rate decay change by batch number:
                num_batches_seen = batch_num + batches_per_iteration * it
                if testing == 1:
                    print 'Working on batch ' + str(num_batches_seen)
                    print 'Threshold units are at ' + str(self.state[self.input_thresh]) + ' and ' + str(self.state[self.hidden_thresh])
                    print 'Current weights: ' + str(self.weights)
                    print 'Current sum of weights: ' + str(np.sum(self.weights))
                    self.check_weights()

                self.rate = self.learning_rate / (num_batches_seen + 1)
                self.batch_process(batch)
                if record_mse == 1:
                    ramse[num_batches_seen] = self.average_rmse(batch)  # Compute the batch root mean square error.
                last_batch_ind = b

            # Manually calculate last batch. It includes some of the first and some of the last examples.
            last_batch_ind += inc
            wrap_around_ind = batch_size - (set_size - last_batch_ind)
            batch = np.vstack((example_set[last_batch_ind:, :], example_set[:
                wrap_around_ind, :]))
            self.batch_process(batch)
            if record_mse == 1:
                ramse[batches_per_iteration * (it + 1) -1] = self.average_rmse(batch)

            # self.rate = self.learning_rate / (1.0 + it)  # If we want to decrease the rate more slowly.
        if record_mse == 1:
            print ramse
            np.save('root_avg_mse.npy', ramse)

    def batch_process(self, batch):
        """
        Take in a batch of examples without class labels to be fed into autoencoder training.
        Go through each example in the batch, compute coactivity for clamped and unclamped runs adding to a running
        total that you then average to change the weights.
        :param batch: a matrix whose rows are examples.
        :return: Changes the weights of the machine.
        """
        batch_size = self.batch_size
        batch_coactivity_clamped = np.zeros((self.total_nodes, self.total_nodes))
        batch_coactivity_unclamped = np.zeros((self.total_nodes, self.total_nodes))
        for ex in range(batch_size):
            # First clamp down the input nodes and output nodes and compute coactivity.
            self.state = rand_bern_with_thresh(self.total_nodes, self.input_thresh, self.hidden_thresh)
            self.clamped_run(batch[ex, :], batch[ex, :])
            batch_coactivity_clamped += self.coactivity(clamped=1, sweeps=1)
            # Next clamp down just the input nodes and compute coactivity.
            self.state = rand_bern_with_thresh(self.total_nodes, self.input_thresh, self.hidden_thresh)
```

```python
            self.unclamped_run(batch[ex, :])
            batch_coactivity_unclamped += self.coactivity(clamped=0, sweeps=1)
        dw = (batch_coactivity_clamped - batch_coactivity_unclamped) / batch_size
        self.weights += self.rate * dw  # Not sure if this should be minus. TODO: find the correct rule.
        self.correct_weights()  # Lazy correction.

    def coactivity(self, clamped=1, sweeps=1):
        """
        Computed the coactivity of each node pair averaged over a given number of sweeps.
        :param clamped: If 0, then do unclamped updating. Otherwise, clamp output nodes.
        :param sweeps: how many times to do a full update and compute the coactivities.
        :return: A matrix of coactivity.
        """
        coactivity_matrix = np.zeros((self.total_nodes, self.total_nodes))
        c_in_to_hidden = np.zeros((self.input_size, self.hidden_size - 1))  # minus 1 since threshold doesn't connect
        c_hidden = np.zeros((self.hidden_size - 1, self.hidden_size - 1))  # minus 1 since threshold doesn't connect
        c_hidden_to_out = np.zeros((self.hidden_size, self.output_size))

        for s in range(0, sweeps):
            c_in_to_hidden += np.outer(self.state[:self.hidden_ind], self.state[self.hidden_nodes])
            c_hidden += np.outer(self.state[self.hidden_nodes], self.state[self.hidden_nodes])
            c_hidden_to_out += np.outer(self.state[self.hidden_ind:self.out_ind], self.state[self.out_nodes])

            if clamped == 0 & sweeps > 1:
                self.unclamped_run(self.state[:self.hidden_ind])
            elif clamped == 1 & sweeps > 1:
                self.clamped_run(self.state[:self.hidden_ind], self.state[self.out_ind:])

        coactivity_matrix[self.in_to_hidden] = c_in_to_hidden
        coactivity_matrix[self.hidden_to_hidden] = c_hidden
        coactivity_matrix[self.hidden_to_out] = c_hidden_to_out
        coactivity_matrix = (coactivity_matrix + coactivity_matrix.T) / 2.0
        return coactivity_matrix / sweeps  # TODO: Check how to compute coactivity during training.

    def read_output(self, input_state, print_out=1):
        # Need to fix an input and then run the machine till it has stabilized.
        # Once stabilized, we can return both the maximizer state as well as the
        # averages for 100 or so states.
        sweep_num = 1000
        self.unclamped_run(input_state, sweep_num)
        output = np.zeros(self.output_size, dtype=float)
        post_stab_sweeps = 100
        for i in range(post_stab_sweeps):
            self.unclamped_run(input_state)
            output += self.state[self.out_ind:]
        average_output = output / float(post_stab_sweeps)
        output_state = np.sign(average_output)
        if print_out == 1:
            # print (output_state == input_state)
            print average_output
            print np.sum(np.equal(output_state, input_state))
        return output_state  # TODO: Decide on the exact rule for reading off the state.

    def average_rmse(self, example_set):
        """
        Computes the difference between computed output and input averaged over the examples.
        :param example_set:
        :return: The square root of the average mean
           square error.
        """
        num_ex = example_set.shape[0]
        error = np.zeros(example_set.shape[1])
        for i in range(num_ex):
            input_state = example_set[i,:]
            error += np.abs(input_state - self.
               read_output(input_state, 0))
        total_error = np.sum(error)
        return np.sqrt(total_error / float(num_ex))

    # def modulate_params(self):


def main():
    start_time = time.time()

    # examples = np.load('toy_example_set.npy')
    # np.random.permutation(examples)

    examples = np.load('testSetSimple.npy')
    examples = convert_binary_to_pm1(examples)

    input_size = examples.shape[1]

    # input_size = 3

    BM = BoltzmannMachine(input_size, 30, input_size)

    # BM.weights = np.ones((11,11))
    # print BM.weights
    # BM.correct_weights()
    # print BM.weights

    BM.run_machine(BM.sweeps)
    BM.training(examples, 3)

    ones_vec = np.ones(5)
    neg_ones_vec = -np.ones(5)

    vec_1 = np.hstack((ones_vec, neg_ones_vec))
    vec_2 = np.hstack((ones_vec, ones_vec))
    vec_3 = np.hstack((neg_ones_vec, ones_vec))
    vec_4 = np.hstack((neg_ones_vec, neg_ones_vec))

    print BM.read_output(vec_1)
    print BM.read_output(vec_2)
    print BM.read_output(vec_3)
    print BM.read_output(vec_4)

    print 'Random vectors: '
    score = 0
    for r in range(10):
        rand = rand_bern(10)
        output_state = BM.read_output(rand)
        score += np.sum(np.equal(output_state, rand))
        print 'In: ' + str(rand) + 'Out: ' + str(
            output_state)
    print str(score) + 'out of 100'

    np.save('trained_weights.npy',BM.weights)

    end_time = time.time()

    print end_time - start_time

if __name__ == "__main__":
    main()
```

The following code was used to make the histograms in figure 2. It opens an energy .npy file produced by the previous script and plots the result.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.load('energy_small_p1.npy')

n, bins, patches = plt.hist(x, 31, normed=0, histtype=
    'bar',facecolor='green', alpha=1, edgecolor='black'
    )

plt.xlabel('Energy Value')
plt.ylabel('Number of Sweeps')
plt.title('Energy Values for Post-Stabilization Sweeps
     - Small Weights')
# plt.axis([30,70,0,30])  # This is for the large
    weights.
plt.axis([-3,4,0,30])  # This is for the small weights
    .
```

```
plt.grid(True)
plt.grid(color='b', linestyle='-', linewidth=.1,alpha
    =.3)

# plt.show()  # If you don't want to save it, but just
     view it, use this line.
plt.savefig('energy_hist_small.png')

plt.clf()

x = np.load('energy_large_p1.npy')

n, bins, patches = plt.hist(x, 31, normed=0, histtype=
    'bar',facecolor='green', alpha=1, edgecolor='black'
    )

plt.xlabel('Energy Value')
plt.ylabel('Number of Sweeps')
plt.title('Energy Values for Post-Stabilization Sweeps
     - Large Weights')
plt.axis([30,70,0,30])  # This is for the large
    weights.
# plt.axis([-3,4,0,30])  # This is for the small
    weights.
plt.grid(True)
plt.grid(color='b', linestyle='-', linewidth=.1,alpha
    =.3)

# plt.show()  # If you don't want to save it, but just
     view it, use this line.
plt.savefig('energy_hist_large.png')
```

A.4 PCA analysis

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.colors as colors

length_of_vecs = 100
my_data = np.zeros((10, length_of_vecs))  # Replace
    with data to be used.

```

```
pca = PCA(n_components=length_of_vecs)  # This creates
     a PCA-doing object.
pca.fit(my_data)

myPCAEigs = pca.explained_variance_  # Creates the
    vector of eigenvalues for the cov matrix.
plt.plot(myPCAEigs,'ro')  # Create a simple plot of
    the eigenvalues.
plt.title('Eigenvalues for Data')
plt.xlabel('Eigenvalue number')
plt.ylabel('Eigenvalue')
plt.savefig('eigPlot.png')  # Save the plot.

plt.clf()  # Clear this figure object for other use.

classes = np.identity(10)  # Replace with class labels
     matrix.

class_labels = np.zeros(classes.shape[0])  # Empty
    vector to condense the class label matrix.
for i in range(classes.shape[0]):
     class_labels[i] = np.argmax(classes[i,:])

num_classes = classes.shape[1]

# Generate a gradient of colors in hsv format.
hsv_colors = [(x*1.0/num_classes, x*0.5/num_classes, (
    num_classes-x)*0.5/num_classes) for x in range(
    num_classes)]
color_list = map(lambda x: colors.rgb2hex(x),
    hsv_colors)  # Convert the colors to hex format.

# This makes a tuple rather than a numpy array out of
    the labels:
label_tuple = []
for i in range(classes.shape[0]):
     label_tuple.append(int(class_labels[i]))

label_colors = np.choose(label_tuple, color_list)  #
    This replaces each label with the corresponding
    color.

fig = plt.figure(1, figsize=(4,3))
ax = fig.add_subplot(111, projection='3d')
pca = PCA(n_components=3)  # Now we force projection
    onto the principal components.
```

```
pca.fit(my_data)
data_proj = pca.transform(my_data)

num_pts = 10  # Let this be 100 or 1000. Too many
    points makes things too messy.

ax.scatter(data_proj[:num_pts,0],data_proj[:num_pts
    ,1], data_proj[:num_pts,2], c=label_colors[:num_pts
    ], cmap=plt.cm.spectral)

plt.axis('on')

# Remove the labels and their silly, imaginary units.
ax.set_xticklabels([])
ax.set_yticklabels([])
ax.set_zticklabels([])

plt.title('PCA Plot for Data')

plt.savefig('pca_example.png')
```