Boltzmann Machine

Ankur Ankan(s473828), Kevin Jacobs (s4134621), Zhuoran Liu(s4594851)

Code File: Boltzmann.py

1 Introduction

1.1 Boltzmann Machines

A Boltzmann Machine consists of a set of binary units, $\{\mathbf{s}\}$ and all these units are connected to each other with a weight, $\mathbf{w_{ij}}$ associated with each connection. The global energy, E of the Boltzmann Machine is given by:

$$E = -\left(\sum_{i<j} w_{ij} s_i s_j + \sum_i \theta_i s_i\right)$$

While learning from data, we try to adjust the weight parameters such that the stationary distribution of the Boltzmann machine closely represents our dataset. For determining the stationary distribution of the Boltzmann Machine we start with random states for all the binary units and iterate over them setting them to $+1$ with a probability of $\frac{1}{1+e^{-2a_i}}$, -1 otherwise. After a few iterations the machine converges to its stationary distribution.

Now for comparing this stationary distribution to our dataset we approximate the stationary distribution of the Boltzmann Machine and using that we compute the free expectations using:

$$< s_i >= \sum_s s_i p(s), \quad < s_i s_j >= \sum_s s_i s_j p(s)$$

Similarly we compute clamped expectations from our dataset using:

$$< s_i >_c = \frac{1}{P}\sum_\mu s_i^\mu, \quad < s_i s_j >_c = \frac{1}{P}\sum_\mu s_i^\mu s_j^\mu$$

and then we update our weights based on these values:

$$\begin{aligned} w_{ij}(t+1) &= w_{ij}(t) + \eta \frac{\partial L}{\partial w_{ij}} \\ \theta_i(t+1) &= \theta_i(t) + \eta \frac{\partial L}{\partial \theta_i} \end{aligned} \tag{1}$$

$$\begin{aligned}\frac{\partial L}{\partial \theta_i} &= < s_i >_c - < s_i > \\ \frac{\partial L}{\partial w_{ij}} &= < s_i s_j >_c - < s_i s_j >\end{aligned} \tag{2}$$

2 Research Questions

1. For keeping the matrix C singular in our computation, we need to add noise to the images. We studied the accuracy of our model for different levels of noise.

2. For computing the free statistics, we need to approximate the probability distribution given by the Boltzmann Machine. We studied the results for two approximating methods namely Gibbs Sampling and Mean Field Approximation.

3 Results

3.1 Accuracy of the model with varying levels of noise

Each image in the MNIST dataset is represented by a 28x28 matrix. Each value in the matrix is an integer between 0 and 256 representing the depth of that pixel. Therefore, we first binarize it by setting all the values greater than 126 to 1, 0 otherwise. Then we add random noise to the image so that the matrix C in our computation isn't singular.

For adding the noise to the image we created a noise mask for each pixel. Each pixel in the mask has a value of -1 with probability p and value 1 with probability $1-p$. For the final noisy image, we do an element-wise multiplication of the binary image and the noise mask. Therefore when $p = 0$, there shouldn't be any distortion in the image whereas for $p = 1$ every bit should be flipped but there won't be any change in the structure. For $p = 0.5$, the maximum noise will be obtained and the image will be completely distorted. We can see these effects in Fig. 1.

We ran our implementation using Gibbs Sampling on varying levels of noise and computed the accuracy of our model.

There are 3 main features in the variation of accuracy with noise level:

1. **Low accuracy at low noise levels**: In Fig. 2, we can see that for very low levels of noise (around 0.00 - 0.03) we have an accuracy of around 0.1 which is equivalent to random guessing. This low accuracy can be explained by the fact that due to low noise levels some of the pixels in all the images are the same. For example the top-left pixel or the top-right pixel will be off in almost all the images. And these constant pixels makes the matrix C approach towards singularity resulting in low accuracy.

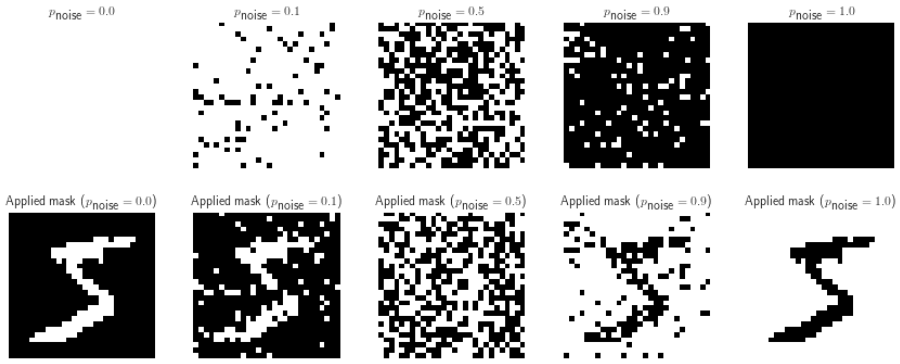

Figure 1: Different noise levels along with the final masked images

2. **Highest accuracy at noise level of 0.08**: The accuracy of the model starts increasing at $p = 0.04$, reaching the maximum accuracy of 97% at $p = 0.08$ and then gradually decreases as we increase the noise level in the data.

3. **Dip at $p = 0.47$ and accuracy of 0.1 at $p = 0.5$**: We see a dip in the accuracy value at $p = 0.47$ where it is almost 0. We couldn't find any good possible explanation for this behaviour. For $p = 0.5$ we again see an accuracy of 0.1. This happens because the images are now completely distorted and the classifier's performance is now equivalent to random guessing.

3.2 Comparison of Sampling Methods and Mean Field Approximation

To train a Boltzmann machine we need to compare the free and the clamped statistics and update the weights according to that. The probability distribution represented by a Boltzmann machine is given by:

$$p(s) = \frac{1}{Z} e^{-E(s)} \tag{3}$$

where:

$$\begin{aligned} E(s) &= -\frac{1}{2} \sum_{ij} w_{ij} s_i s_j + \sum_i \theta_i s_i \\ Z &= \sum exp(-E(s)) \end{aligned} \tag{4}$$

Here if we try to compute the distribution using any exact method we will need to compute the value of Z for which we will need to compute the sum over all the possible 2^n states of the Boltzmann machine. This is not feasible even

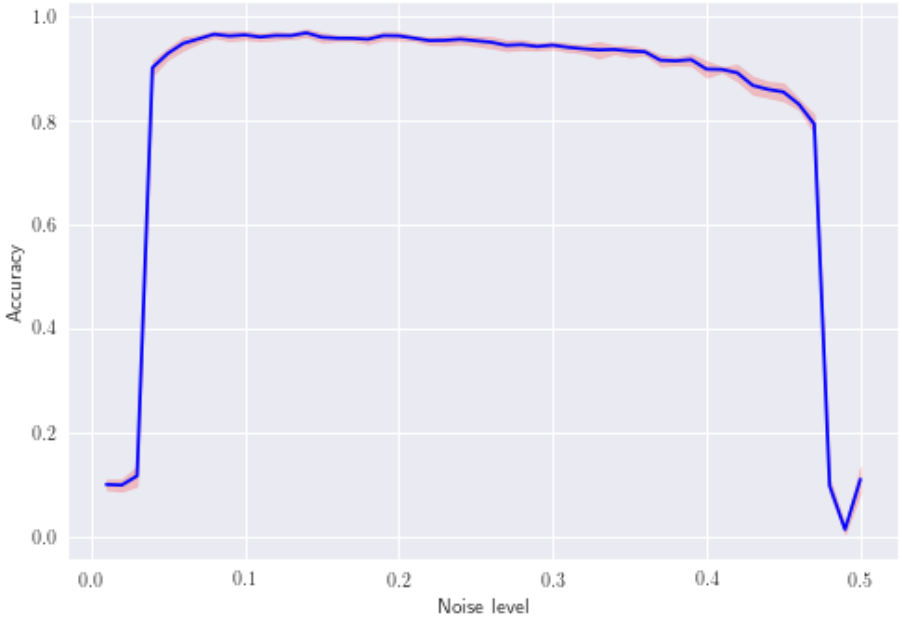

Figure 2: Accuracy of the model with varying noise level for Gibbs Sampling using 400 samples with 100 burn-in samples. Red region represents the standard deviation of the accuracy over multiple runs.

in the case of relatively smaller networks because we need to compute the free statistics in each iteration until convergence. Therefore we need to use some approximate techniques. For the assignment we used Gibbs sampling and Mean Field theory for approximating the free statistics.

1. **Behaviour in case of Gibbs Sampling** Gibbs sampling is a Markov Chain Monte Carlo algorithm for obtaining a sequence of observations which are approximated from the specified probability distribution. In Gibbs sampling we randomly iterate over each of the components of the random variable and update it according to:

$$q(x' \mid x) = p(x'_i \mid x_{-i})\delta_{x_{-i},x'_{-i}} \tag{5}$$

   We tested the effect of varying the number of samples and burn in period on the convergence of our learning rule. For this we used a network of 10 neurons and trained on 50 random input patterns. We initialized the weights and biases using random samples from a Normal Distribution with mean 0 and variance 1.

   For comparing the rate of convergence, we fitted a straight line using ordinary linear regression to the absolute change in weights with number

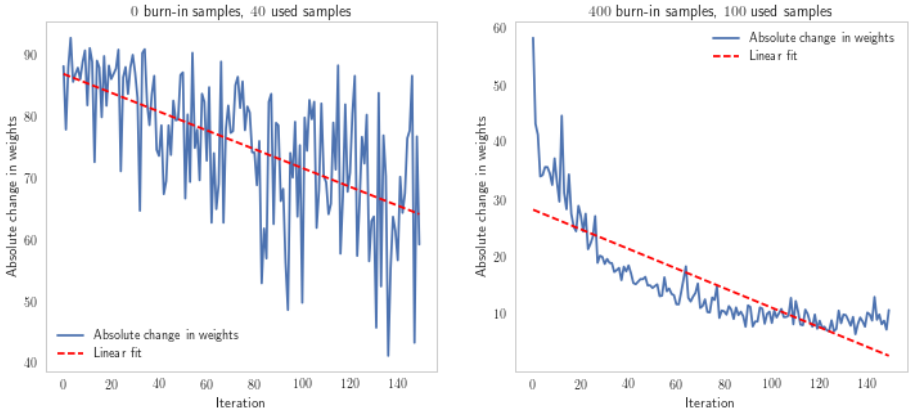

Figure 3: Absolute change in weight with iteration

of iterations as shown in Fig. 3. We also plotted a grid (Fig. 4) of the value of slope of the fitted straight line for different values of burn-in samples and number of used samples.

2. **Approximating using Mean Field Theory**

   Another approach for approximating the probability distribution is to use Mean Field Theory. We can use Mean Field to approximate the value of the normalizing constant, Z using:

$$Z = e^{-F(m)} \tag{6}$$

where

$$\begin{aligned} F(m) = &- \sum_{ij} w_{ij} m_i m_j - \sum_i h_i m_i + \\ &\sum_i \frac{1}{2} \left((1+m_i) \log \frac{1}{2}(1+m_i) + (1-m_i) \log \frac{1}{2}(1-m_i) \right) \end{aligned} \tag{7}$$

As we would expect, the Mean Field Approximation is much faster than Gibbs Sampling since we need to generate at least a few hundred samples while using Gibbs Sampling for good results. Our implementation using Gibbs Sampling took an average of 18.4 seconds for training over the whole dataset whereas the implementation with Mean Field Approximation took 1.74 seconds on average.

Another point to notice is that we got slightly lower accuracy in the case of Mean Field compared to Gibbs sampling. For the noise level of 0.08, we got an accuracy of 96.25% for Mean Field compared to 97.3% in case of Gibbs sampling using 400 samples and 100 burn in samples and the same

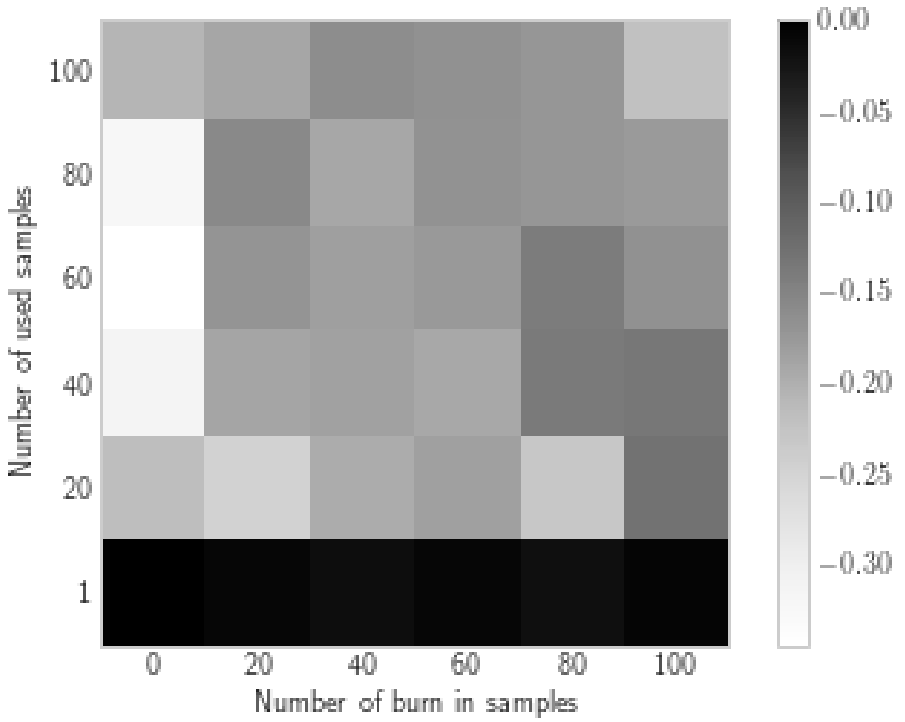

Figure 4: Variation of slope for different values of used samples and burn in samples

noise level. Fig. 5 shows the confusion matrix for the predictions using Mean Field Approximation.

4 Conclusion

We were able to get a good accuracy for classification using the Boltzmann Machine for both the approximation methods. But in the case of Gibbs sampling we have two extra parameters namely the burn-in samples and number of samples, which gives us control over how accurately we want to approximate the free statistics.

Our classifier was slightly confused in a few cases like between 8 and 5, 3 and 5 as we can see in Fig. 5. This happens because the pattern for these digits are similar and should be avoidable using some preprocessing steps like rotation of images. Another possible improvement is to have a more complex Boltzmann Machine for this problem by using hidden variables in the model. Having a more complexer model might increase the accuracy but will also be

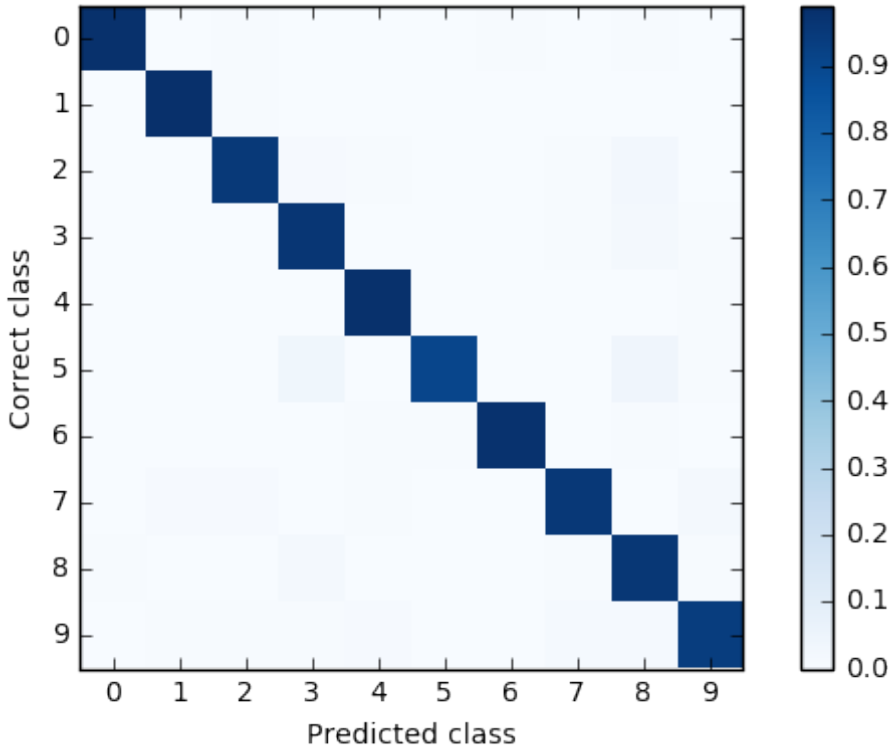

Figure 5: Confusion Matrix for classifying all the 10 digits

more susceptible to over fitting.

5 Appendix

5.1 Code

```
% matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import rc
import seaborn as sns
import scipy.io
import random
rc('text', usetex=True)
sns.set_style("whitegrid", {'axes.grid' : False})

class DataLoader:
```

```
    """
    Class for loading the MatLab data.
    """

    def __init__(self, file):
        """
        Load the data from the given file.

        :param file: Path to file to load the data from
                (without the .mat extension).
        :param flip_rate: Rate for flipping bits randomly,
            0 means no flipping and 1 means all flipped and the
            maximum entropy is obtained for flip_rate = 0.5.
        """
        self.data = scipy.io.loadmat(file, squeeze_me=True,
                                     struct_as_record=False)['mnist']
        self.train_images = self.data.train_images
        self.test_images = self.data.test_images
        self.train_labels = self.data.train_labels
        self.test_labels = self.data.test_labels

    def load_images(self, flip_rate=0):
        self.train_images = self.transform_images(self.train_images)
        self.test_images = self.transform_images(self.test_images)
        if flip_rate > 0:
            noise = np.random.binomial(
                1, flip_rate, self.train_images.shape) * -2 + 1
            self.train_images = np.multiply(self.train_images, noise)

    def transform_images(self, data):
        """
        Convert a (m x n x p) array to a (p x m x n) array and
        apply some additional transformations.

        :param data: Data to transform.
        :return: Transformed data.
        """
        reshaped = data.reshape(data.shape[0] * data.shape[1],
                                data.shape[2])
        swapped_axes = np.swapaxes(reshaped, 0, 1)
        return (swapped_axes > 122) * 2 - 1

def calculate_normalizing_constant(samples, w, theta):
    return np.sum(np.exp(-calculate_energy(samples, w, theta)))

def calculate_probabilities(samples, w, theta, normalizing_constant):
```

```
    return np.exp(-calculate_energy(samples, w, theta)) / \
           normalizing_constant

def calculate_energy(samples, w, theta):
    f = np.dot(samples, np.dot(w, samples.T))
    # Also allow samples consisting of one sample (an array,
    # so f.ndim == 1)
    # Therefore, only take the diagonal in the two dimensional case
    if f.ndim == 2:
        f = np.diagonal(f)
    return np.squeeze(np.asarray(-0.5 * f - np.dot(
        theta.T, samples.T)))

def generate_samples(w, theta, num_burn_in=50, num_samples=500,
                     show_transition_probabilities=False):
    num_neurons = w.shape[0]

    # Initialize a random sample
    s = np.random.binomial(1, 0.5, (num_neurons,)) * 2 - 1

    # Initialize the matrix of generated samples
    X = np.empty((0, num_neurons))

    # Iterate (first generate some samples during the burn-in
    # period and then gather the samples)
    for iteration in range(num_samples):
        for burn_in in range(num_burn_in + 1):
            # Store the original value of s
            s_original = s
            # Calculate the flip probabilities
            p_flip = 0.5 * (1 + np.tanh(np.multiply(
                -s, np.dot(w, s) + theta)))
            # Calculate transition probabilities
            p_transition = p_flip / float(num_neurons)
            p_stay = 1 - np.sum(p_transition)
            # Flip according to the probability distribution of flipping
            if random.random() <= 1 - p_stay:
                # Pick a random neuron
                neuron = random.randint(1, num_neurons) - 1
                if random.random() <= p_flip[neuron]:
                    s[neuron] *= -1
            # Add the state if the sample is not generated
            # during the burn in period
            if burn_in >= num_burn_in:
                if show_transition_probabilities:
                    print('Transition probabilities for ',
```

```
                        s_original,':', p_transition,
                        ' (stay probability: ', p_stay, ')')
                X = np.vstack([X, s])
    return X

def calculate_clamped_statistics(X):
    """
    Calculate <x_i>_c and <x_i x_j>_c given X.
    """
    num_datapoints = X.shape[0]
    return np.sum(X, axis=0) / num_datapoints, \
           np.dot(X.T, X) / num_datapoints

# Function for training using Gibbs Sampling
def training_bm(num_burnin, num_samples):
    num_neurons = 10
    learning_rates = [0.05, 0.05]
    w = np.random.normal(0, 1, (num_neurons, num_neurons))
    w = np.tril(w) + np.tril(w, -1).T
    np.fill_diagonal(w, 0)
    theta = np.random.normal(0, 1, (num_neurons,))

    X_c = np.random.binomial(1, 0.5, (50, num_neurons)) * 2 - 1
    s1_c, s2_c = calculate_clamped_statistics(X_c)

    q = []
    for _ in range(150):
        X = generate_samples(w, theta, num_burnin, num_samples)
        Z = calculate_normalizing_constant(X, w, theta)
        p = calculate_probabilities(X, w, theta, Z)

        p_repeat = np.tile(p, (num_neurons, 1)).T
        Q = np.multiply(p_repeat, X)
        s2 = np.dot(X.T, Q)
        s1 = np.dot(p, X)

        dLdw = s2_c - s2
        dLdtheta = s1_c - s1
        np.fill_diagonal(dLdw, 0)

        delta_w = learning_rates[0] * dLdw
        delta_theta = learning_rates[1] * \
                      np.squeeze(np.asarray(dLdtheta))

        w += delta_w
        theta += delta_theta
```

```
        q.append([np.sum(np.abs(dLdw)), np.sum(np.abs(dLdtheta))])
    return np.matrix(q)


############################################################
##### Absolute change in weight with varying samples ######
############################################################

q = training_bm(0, 40)
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))
fit = np.polyfit(np.arange(150), q[:, 0], 1, full=True)
ax[0].plot(q[:, 0], label='Absolute change in weights')
ax[0].plot(np.arange(150), fit[0][0] * np.arange(150) + fit[0][1],
           '--r', label='Linear fit')
ax[0].legend()
ax[0].set_xlabel('Iteration')
ax[0].set_ylabel('Absolute change in weights')
ax[0].set_title('$0$ burn-in samples, $40$ used samples')

q = training_bm(400, 100)
fit = np.polyfit(np.arange(150), q[:, 0], 1, full=True)
ax[1].plot(q[:, 0], label='Absolute change in weights')
ax[1].plot(np.arange(150), fit[0][0] * np.arange(150) + fit[0][1],
           '--r', label='Linear fit')
ax[1].legend()
ax[1].set_xlabel('Iteration')
ax[1].set_ylabel('Absolute change in weights')
ax[1].set_title('$400$ burn-in samples, $100$ used samples')



#################################
### Mean Field Approximation ####
#################################

num_neurons = 28 * 28
learning_rate = 0.01
w = np.random.normal(0, 1, (num_neurons, num_neurons))
theta = np.random.normal(0, 1, (num_neurons,))
np.fill_diagonal(w, 1)

def calculate_probabilities(samples, w, theta,
                            normalizing_constant):
    return np.exp(-calculate_energy(
        samples, w, theta)) / normalizing_constant
```

```
def calculate_energy(samples, w, theta):
    f = np.dot(samples, np.dot(w, samples.T))
    # Also allow samples consisting of one sample
    # (an array, so f.ndim == 1)
    # Therefore, only take the diagonal in the two dimensional case
    if f.ndim == 2:
        f = np.diagonal(f)
    return np.squeeze(np.asarray(-0.5 * f - np.dot(
        theta.T, samples.T)))


def train_classifiers(samples, labels):
    w = np.zeros((10, 28 * 28, 28 * 28))
    theta = np.zeros((10, 28 * 28))
    Z = np.zeros(10)
    for digit in range(0, 10):
        training_samples = samples[labels == digit]
        s_mean_clamped = np.squeeze(np.asarray(np.mean(
            training_samples, 0)))
        s_cov_clamped = np.cov(training_samples.T)
        m = s_mean_clamped
        C = s_cov_clamped - np.dot(np.asmatrix(
            s_mean_clamped).T, np.asmatrix(s_mean_clamped))
        delta = np.zeros(s_cov_clamped.shape)
        np.fill_diagonal(delta, 1. / (1. - np.multiply(m, m)))
        w[digit, :, :] = delta - np.linalg.inv(C)
        theta[digit, :] = np.arctanh(m) - np.dot(w[digit, :, :], m)
        F = -0.5 * np.dot(np.dot(m, w[digit, :, :]), m) - \
            np.dot(theta[digit, :], m) + 0.5 * np.dot(
            1 + m, np.log(0.5 * (1 + m))) + 0.5 * np.dot(
            1 - m, np.log(0.5 * (1 - m)))
        Z[digit] = np.exp(-F)
    return w, theta, Z

w, theta, Z = train_classifiers(data.train_images, data.train_labels)
```