Handwritten Digits Classification using Restricted Boltzmann Machines

1 Introduction

Restricted Boltzmann Machine (RBM) has been widely used as an helpful tool to extract the feature of images and construct the learning modules. RBMs are usually trained using the contrastive divergence learning procedure. This requires a certain amount of practical experience to decide how to set the values of numerical meta-parameters such as the learning rate, the momentum, the weight-cost, the sparsity target, the initial values of the weights, the number of hidden units and the size of each mini-batch. [2] There are several ways of using RBMs for discrimination. In this paper, we use RBMs as stand-alone non-linear classifiers, not only as feature extractors. We investigate different algorithms and compare the results when apply them on different test database.

2 Restricted Boltzmann Machines

Restricted Boltzmann Machines are bipartite network, including one layer of hidden units $\{h_i\}$ and one layer of visible units $\left\{v_j\right\}$. A joint configuration, $(\boldsymbol{v}, \boldsymbol{h})$ of the visible and hidden units has an energy [3] given by,

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\sum_{i \in visible} a_i v_i - \sum_{j \in hidden} b_j h_j - \sum_{i,j} v_i h_j w_{ij}$$

where a_i, b_j are their biases and w_{ij} is the weight between them. Therefore, we have the probability of the network (or system) with $(\boldsymbol{v}, \boldsymbol{h})$ as follows [1],

$$p(\boldsymbol{v}, \boldsymbol{h}) = \frac{1}{Z} e^{-E(\boldsymbol{v}, \boldsymbol{h})}$$

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

where $Z = \sum_{\boldsymbol{v}, \boldsymbol{h}}^{e^{-E(\boldsymbol{v}, \boldsymbol{h})}}$ is constant for a single RBM.

We try to adjust the weights and biases to lower the energy of RBM when provided $(\boldsymbol{v}, \boldsymbol{h})$. The derivative of the log probability of $\boldsymbol{v}$ with respect to a weight is given by,

$$\frac{\partial \log p(\boldsymbol{v})}{\partial w_{ij}} = \left\langle v_i h_j \right\rangle_{data} - \left\langle v_i h_j \right\rangle_{model}$$

Hence, while we train a RBM, the change in a weight when training is given by,

$$\Delta w_{ij} = \epsilon \left(\left\langle v_i h_j \right\rangle_{data} - \left\langle v_i h_j \right\rangle_{reconstruction} \right) \quad (1)$$

where ϵ is the learning rate and reconstruction is produced by Gibbs Sampling. It starts by setting the states of the visible units to a training vector. Then the binary states of the hidden units are computed as follows,

$$p\left(h_j = 1 | \boldsymbol{v}\right) = \sigma\left(b_j + \sum_i v_i w_{ij}\right) \quad (2)$$

where $\sigma(x) = 1/(1 + \exp(-x))$ is the logistic function. In the meantime, we can update the states of visible units from those of the hidden units by computing as follows,

$$p\left(v_j = 1 | \boldsymbol{v}\right) = \sigma\left(a_i + \sum_j h_j w_{ij}\right) \quad (3)$$

Therefore, we can get a "reconstruction" by computing equation (2) and (3) in turn.

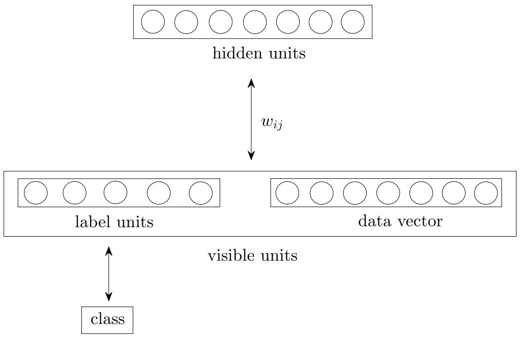


Fig. 1: Restricted Boltzmann Machine modeling the joint distribution of inputs and target classes.

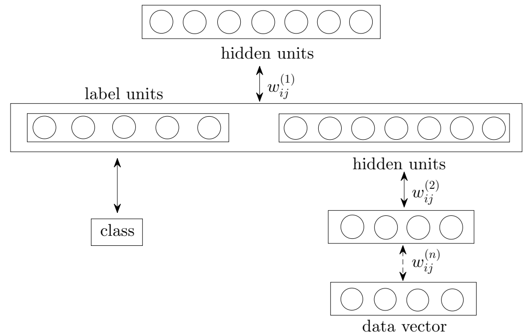


Fig. 2: Multilayer Restricted Boltzmann Machine Model.

It is more efficient to update the weights if we divide the training set into small mini batches where every batch contains examples of all classes. Besides, we use momentum method to increase the speed of learning and weight-decay to penalize large weight. [4]

Eventually, the change in biases $(a_i), \left(b_j\right)$ and weights $\left(w_{ij}\right)$ is given by,

$$\Delta a_i = \beta \cdot \Delta a_i + \epsilon \ (\ v_{i \ data} - v_{i \ reconstruction} \ ) \tag{4}$$

$$\Delta b_j = \beta \cdot \Delta b_j + \epsilon \ (\ h_{i \ data} - h_{i \ reconstruction} \ ) \tag{5}$$

$$\Delta w_{ij} = \beta \cdot \Delta w_{ij} + \epsilon \ \left(\left\langle v_i h_j \right\rangle_{data} - \left\langle v_i h_j \right\rangle_{recon} - \eta \cdot w_{ij} \right) \tag{6}$$

where β is momentum coefficient, ϵ is the learning rate and η is the weight-decay coefficient.

3 Classification methods using RBMs

There are several ways of using RBMs for classification. The most common way is to use the hidden units learned by RBM as input features for other discriminative method. However, this is not into our consideration here. What we concern about is how RBM can be directly used as classifier.

3.1 Classify using a joint density model learned from single RBM

The first method is to train a joint density model using a single RBM whose visible units includes data vector and label units which represents the class. [4] It is shown in Fig 1.

After training, each possible label is tried in turn with a test vector and the one that gives lowest free energy is chosen as the most likely class. The free energy of a visible vector $\boldsymbol{v}$ is defined as follows,

$$F(\boldsymbol{v}) = -\sum_i v_i a_i - \sum_j \log\left(1 + e^{x_j}\right) \tag{7}$$

where $x_j = b_j + \sum_i v_i w_{ij}$.

3.2 Classify using different RBMs trained on each class

The second method is to train a separate RBM on each class. [4] After training, the test vector is tried with each RBM in turn and the one that gives the lowest reconstruction error is chosen as the most likely class. The reconstruction error of visible vector $\boldsymbol{v}$ in RBM $\boldsymbol{R}$ is define as follows,

$$e(\boldsymbol{v}, \boldsymbol{R}) = \|\boldsymbol{v}_{recon}(\boldsymbol{R}) - \boldsymbol{v}\| \tag{8}$$

where $\boldsymbol{v}_{recon}(\boldsymbol{R})$ is a reconstruction from $\boldsymbol{v}$ in RBM $\boldsymbol{R}$ using equation (2) and (3).

3.3 Classify using multilayer RBMs

The third method is a mutation of the first method. We use several RBMs. The output hidden units of the former RBM is the input visible units of the latter RBM. The input of top RBM includes the former RBM's hidden units and the label units. It is shown in Fig 2. The bottom RBMs serve as feature extraction and the top one works as classifier. The classification criteria is the same as the first method (section 3.1).

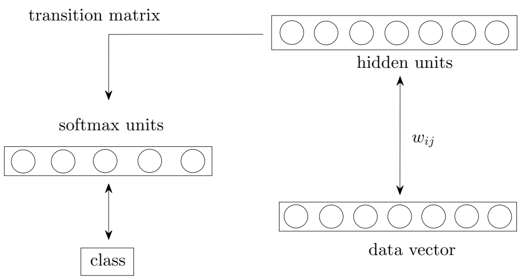


Fig. 3: Single Restricted Boltzmann Machines with transition matrix.

3.4 Classify using single RBM with softmax units

The fourth method seems similar to the first one (section 3.1) but is different in fact. First, we use data vector as input of RBM. Then we multiply the hidden units of RBM with a transition matrix to get the soft units $\{x_i \mid i = 1, 2, \cdots K\}$. The softmax units can represent K classes. The probability of soft units on is computed as follows,

$$p_j = \frac{e^{x_j}}{\sum_{i=1}^{k} e^{x_i}} \tag{9}$$

If the $i - th$ unit of softmax units owns largest probability, then test vector belongs to the $i - th$ class. It is shown in Fig 3

The update procedure of weight of RBM and the transition matrix has changed since this network structure is different from simple RBM in method (section 3.1). First, we initial and preliminarily update the weights of RBM using equation 4, 5 and 6. Then, we use Back Propagation Algorithm to update the transition matrix and the weight of RBM together in order to minimize the loss function.

3.5 Classify using multilayer RBMs with transition matrix

The fifth method is the extension of the fourth method (section 3.4). First, we construct a multilayer RBMs network just like the third method but the input of the top RBM is just the hidden units of the formmer RBM. Then, we multiply the hidden units with a transition matrix to get the label units. The classification criteria is the same as that of the fourth method (section 3.4). It is shown in Fig 3

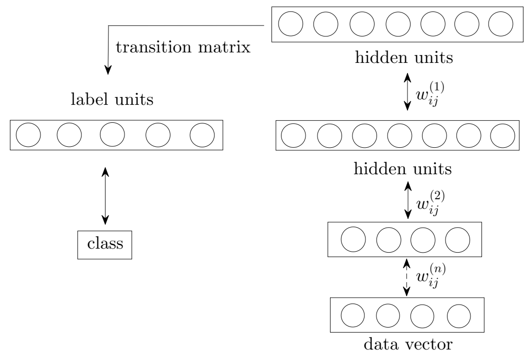


Fig. 4: Multilayer Restricted Boltzmann Machines with transition matrix.

The update procedure of weights of RBMs network and the transition matrix is the same as that in method 4 (section 3.4).

4 Experiment and results

4.1 Data and preprocessing

For our numerical experiments, we use the handwritten digits data set from MNIST database provided by Yann LeCun et al. It contains 60,000 digits ranging from 0 to 9 for training and 10,000 for test. Each digits is normalized and centered in a grey-level image with size 28×28. Professor Tao Linmi and his TA's provide 720 digits for further test experiment, which have been already graded according to the noise condition.

4.2 Testing procedure and parameters settings

We compare five methods: a joint density model learned from single RBM (section 3.1), different RBMs trained on each class (section 3.2), multilayer RBMs (section 3.3), single RBM with softmax units (section 3.4) and multilayer RBMs with transition matrix (section 3.5).

We first use MNIST training dataset for training and MNIST test dataset for test. Then we use data provided by Professor Tao for test in order to investigate the mobility of RBM model.

The parameters of RBM training are set as follows: learning rate $\epsilon = 0.1$, weight-decay coefficient $\eta = 0.0002$, the momentum to start with $\beta = 0.5$ and to be increased to $\beta = 0.9$ when reconstruction error has settled down.

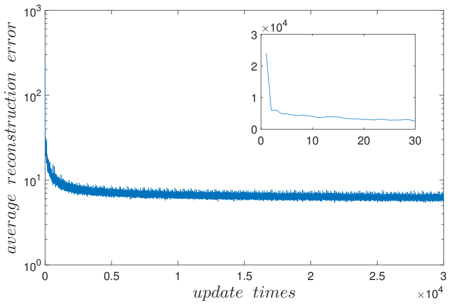


Fig. 5: Average reconstruction error along with every update of weight of RBM.

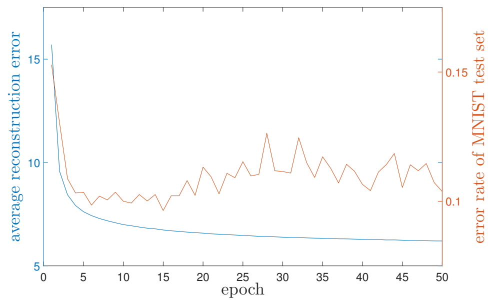


Fig. 6: Average reconstruction error along with every epoch (traversal of all training data) .

For method 1 (section 3.1, single RBM), and method 2 (section 3.2, RBMs on each class), the numbers of hidden units are 500. For method 4 (section 3.4, single RBM with softmax units), the number of hidden units is 1000. For method 3 (section 3.3, RBMs network), we use 2-layer network and the numbers of hidden units of each layer are 500 and 2000. For method 5 (section 3.5, RBMs network with transition matrix), we utilize 3-layer network and the numbers of hidden units of each layer are 500, 500 and 2000.

4.3 Results on MNIST data

4.3.1 *Method 1: single RBM*

Fig 5 shows changes in reconstruction error of the mini train batch while Fig 6 displays reconstruction error of whole training data and the error rate of MNIST test data set. Reconstruction error is defined in equation 8 and there 600 update times in one epoch. It is obvious that reconstruction error decreases every

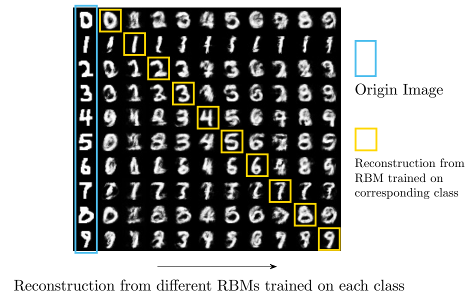


Fig. 7: Reconstruction of different RBMs trained on each class.

time we update the weight of RBM. More specifically, the curve in Fig 5 declines sharply during the first 10 updates and becomes flat then; similarly, the curve in Fig 6 declines sharply during the first 25 epochs and slowly then. However, error rate of MNIST test data set does decrease in the beginning but after 10 epochs, it goes up and down but does not show the trend to decline, which means the model begins to be overfitting.

4.3.2 *Method 2: RBM on each class*

We pick one image randomly from each class and use RBMs trained on different class to reconstruct it. The result is shown in Fig 7. We can find that reconstructions of one image from different RBMs show different styles and look like mixture of origin image and the digit corresponding to RBM. However, some digits share similar structure, such as 3, 5 and 8. Reconstruction from RBM of the digit with similar structure as the origin sometimes does better than reconstruction from RBM of correct digit, which cause certain error when classifying the digits.

4.3.3 *Method 4 and 5: RBMs and transition matrix*

Fig 8 and Fig 9 show the changes of reconstruction error and error rate during the back propagation process. We can observe that no matter whether it is method 4 or method 5, the reconstruction error of training data is decreasing and presents the same characteristics in Fig 6. However, the reconstruction error of test data declines in the beginning and after nearly twenty to thirty epochs, it starts to climbing,

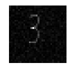
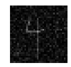
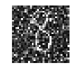
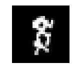
Level	Level 0	Level 1	Level 2	Level 3	Level 4	Level 5
Description	no-noise	subtle-noise	some-noise	medium-noise	more-noise	great-noise
Example	6	2	3	4	8	5
						
After image processing						

TABLE 1: Example images of different levels and results after processing

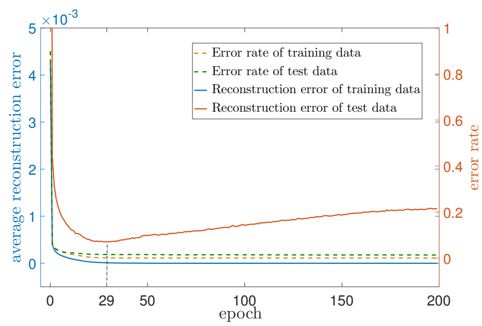


Fig. 8: Reconstruction error and error of training data and test data during back propagation training process in method 4.

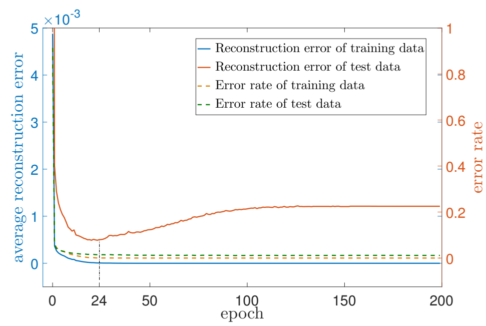


Fig. 9: Reconstruction error and error of training data and test data during back propagation training process in method 5.

which means the RBM model is overfitting afterwards. In addition, both the error rate of training data and test data decrease at first and after ten to twenty epochs it begins to be stable. We can learn that for back propagation process, we need to divide the training data into 2 sets. One is used for training, while the other one is used as validation data in order to avoid the RBM model from being over training.

4.3.4 Comparison among five methods

The Table 2 shows the error rate of test MNIST data using different methods. We can observe that all the error rate are under 10%. The classification effects of method 1 and method 2 nearly the same. The error rate of method 3 is smaller than that of method 1, which means discriminative effect goes better as the layer of RBM increases. Method 4 and method 5 do the best among all five methods, which indicates that back propagation algorithm helps the weights of RBM updated better and improve the classification result greatly.

4.4 Mobility of RBM's model

Digits provided by Professor Linmi Tao and his TA's are graded into 5 levels according their noise condition: no-noise, subtle-noise, some-noise, medium-noise, more-noise and great-noise. Before we classify the test data, we need to do preprocessing work on these images. The processing procedure is introduced in appendix . Table 1 displays example images of different levels and the result images after processing. We can find that as level goes up, the quality of result images becomes worse, which undoubtedly increases difficulty in discrimination.

Table 3 shows vividly the error rate of test data in different files using different methods. The one with smallest error rate is marked with yellow or cyan background in the table 3. We can have several observation.

Method	Method 1	Method 2	Method 3	Method 4	Method 5
Error Rate	9.64%	7.31%	4.85%	1.30%	1.09%

TABLE 2: Error rate of MNIST test data set using different methods

Error Rate		Method 1	Method 2	Method 3	Method 4	Method 5
File 0	level 0	13.33%	30.00%	16.67%	6.67%	3.33%
File digits	level 0	43.33%	26.67%	16.67%	3.33%	0.00%
	level 1	43.33%	26.67%	16.67%	3.33%	0.00%
	level 2	36.67%	23.33%	20.00%	3.33%	3.33%
	level 3	56.67%	70.00%	40.00%	26.67%	10.00%
	level 4	83.33%	80.00%	70.00%	50.00%	56.67%
	level 5	76.67%	73.33%	70.00%	66.67%	66.67%
File hjk_picture	level 0	30.00%	43.33%	23.33%	20.00%	20.00%
	level 1	36.67%	46.67%	23.33%	16.67%	20.00%
	level 2	33.33%	46.67%	26.67%	20.00%	20.00%
	level 3	56.67%	46.67%	46.67%	33.33%	33.33%
	level 4	80.00%	60.00%	76.67%	50.00%	53.33%
	level 5	73.33%	86.67%	56.67%	60.00%	60.00%
File Li Wanjin	level 1	36.67%	33.33%	26.67%	13.33%	13.33%
	level 2	33.33%	36.67%	30.00%	13.33%	10.00%
	level 3	43.33%	63.33%	40.00%	20.00%	16.67%
	level 4	53.33%	73.33%	46.67%	46.67%	40.00%
	level 5	66.67%	73.33%	70.00%	56.67%	53.33%
File number	level 0	36.67%	23.33%	20.00%	3.33%	6.67%
	level 1	36.67%	23.33%	20.00%	3.33%	6.67%
	level 2	33.33%	23.33%	26.67%	10.00%	10.00%
	level 3	50.00%	53.33%	40.00%	26.67%	23.33%
	level 4	76.67%	80.00%	60.00%	60.00%	56.67%
	level 5	70.00%	70.00%	66.67%	53.33%	53.33%
Average	level 0	30.83%	30.83%	19.17%	8.33%	7.50%
	level 1	38.33%	32.50%	21.67%	9.17%	10.00%
	level 2	34.17%	32.50%	25.83%	11.67%	10.83%
	level 3	51.67%	58.33%	41.67%	26.67%	20.83%
	level 4	73.33%	73.33%	63.33%	51.67%	51.67%
	level 5	71.67%	75.83%	65.83%	59.17%	58.33%

TABLE 3: Error rate of test data set provided by Professor Tao Linmi using different methods

Method 1 (section 3.1) and method 2 (section 3.2) behave similarly, and method 3 (section 3.3) owns smaller error rate than those two do. Meanwhile, all the colored area belong to method 4 (section 3.4) and method 5 (section 3.5), which is in accord with Table 2.

However, the error rate of addition test data provided by Professor Tao Linmi are usually higher than that of MNIST test data. This indicates that it is difficult to make real digits images look like MNIST data because the real digits images often generate with unexpected noise and the size and position of digit in one image are unpredictable. Thus, the mobility of RBM model trained by MNIST training data is often not that good. Once we can process the images well, just like level 0 in File 0, can the RBM method perform normal. It is supported by the first data row of table 3, where the error rate of method 4 or 5 is under 10%.

5 Conclusion

In this paper, we first introduce the basic concept of Restricted Boltzmann Machines (RBM) and five methods to do classification work using RBM as classifier. Then we compare the effect of five methods and discuss the mobility of RBM model. In general, method 5 (section 3.5: RBMs network with softmax units using back propagation algorithm) does best in five methods. The mobility of RBM classification methods depends on the quality of test images after processing. We can maintain the performance of RBM method when the quality of images after processing is similar to level 0 in Table 1.

6 Acknowledgments

My study on course 'Pattern Recognition' will soon come to an end and in the end of this thesis, I wish to express my sincere appreciation to Professor Tao Linmi and his TAs, for their patient guidance and great effort. It is they who lead me in pursuit of knowledge of pattern recognition. In the meantime, I wish the course 'Pattern Recognition' will be better in the future.

References

[1] Duda, R. O., Hart, P. E., & Stork, D. G. . *Pattern classification* /. Wiley.

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

[3] Hopeld, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79:2554-2558.

[4] Hinton, G. E. (2010). A practical guide to training restricted Boltzmann machines. *Momentum*, 9(1), 599-619.

[5] Solem, J. E. (2012). *Programming computer vision with Python*. O'Reilly.

Appendix

.1 Python Code

Because of the format requirements, the code cannot be shown in normal way. Code has been uploaded along with this paper.

.1.1 Some Definition

.1.1.1 RBM: single RBM

```
class RBM:
  def __init__(self,num_visible,
  num_hidden,learning_rate=0.1,Path=None):
    """
    Initial Function
    :param num_visible:
          the number of visible units
    :param num_hidden:
          the number of hidden units
    :param learning_rate:
          the learning rate of RBM
    :param Path:
          the path where we store the
          parameters of RBM, and it
          ends with //

    weights£ weights is the matrix of
    ( 1 + num_visible ) * ( 1 + num_hidden)
    the first row of "weights" is the
    hidden bias, the first column of
    "weights" is the visible bias
    the rest part of "weights" is the
    weight matrix of edges between visible
    units and hidden units

    weightsinc: weightsinc is the increase
    (or change) of "weights" in every epoch
    of training
    """

    self.num_hidden = num_hidden
    self.num_visible = num_visible
    self.learning_rate = learning_rate
    self.path = Path

    # Check whether parameter file exists
    # if so, load the data
    import os
```

```
    datafile = self.path + 'weights'
    if os.path.isfile(datafile):
      with open(datafile, 'rb') as fp:
        self.weights = pickle.load(fp)
      print("Load Weights Successfully!")
      datafile = self.path + 'weightsinc'
      with open(datafile, 'rb') as fp:
        self.weightinc = pickle.load(fp)
      print("Load WeightInc Successfully!")
    else:
      # Initialize the weights
      # using a Gaussian distribution
      # mean 0 and standard deviation 0.1.
      self.weights = 0.1 * np.random.randn(
        self.num_visible, self.num_hidden)
      # Insert "weights" for the bias units
      # into the first row and first column
      self.weights = np.insert(
          self.weights, 0, 0, axis=0)
      self.weights = np.insert(
          self.weights, 0, 0, axis=1)
      with open(datafile, 'wb') as fp:
        pickle.dump(self.weights, fp)
      print("Create Weights Successfully!")
      # Initialize the weightsinc with zero
      self.weightinc = np.zeros(
          [self.num_visible + 1,
            self.num_hidden + 1])
      datafile = self.path + 'weightsinc'
      with open(datafile, 'wb') as fp:
        pickle.dump(self.weightinc, fp)
      print("Create WeightInc
          Successfully!")

  def train(self,batch_data,max_epochs=50):
    """
    Train the RBM
    :param batch_data: training data,
      type: list of np.array,
      every np.array is a matrix
      where each row is a training example
      consisting of states of visible units
      i.e. every np.array is a training
      batch
    :param max_epochs: the max epochs of
      training operation
    """
    # Initialization
    # weightcost times weightsinc and then
    # be added to the normal gradient
    # cost ranges from 0.01 to 0.00001
    weightcost = 0.0002
    # Momentum is a simple method for
    # increasing the speed of learning
    # when the objective function contains
    # long, narrow and fairly straight
    # ravines with a gentle but consistent
    # gradient along the floor of ravine
    # and much steeper gradients up the
    # sides of the ravine.
    initialmomentum = 0.5
    finalmomentum = 0.9
    count = 0
    for epoch in range(0, max_epochs):
      errorsum = 0
      for data in batch_data:
        num_examples = data.shape[0]
        # Insert bias into the first column
        data =
            np.insert(data, 0, 1, axis=1)
        # Gibbs Sample
        # (This is the "positive CD phase")
        pos_hidden_activations =
            np.dot(data, self.weights)
        pos_hidden_probs = self._logistic(
            pos_hidden_activations)
        # Fix the bias unit
        pos_hidden_probs[:, 0] = 1
        pos_hidden_states =
            pos_hidden_probs >
              np.random.rand(num_examples,
                self.num_hidden + 1)
        pos_associations = np.dot(data.T,
            pos_hidden_probs)
        # (This is the "negative CD phase")
        neg_visible_activations =
            np.dot( pos_hidden_states,
              self.weights.T)
        neg_visible_probs = self._logistic(
            neg_visible_activations)
        # Fix the bias unit
        neg_visible_probs[:, 0] = 1
        neg_hidden_activations =
            np.dot(neg_visible_probs,
              self.weights)
        neg_hidden_probs = self._logistic(
            neg_hidden_activations)
        # Fix the bias unit
        neg_hidden_probs[:, 0] = 1
        neg_associations =
            np.dot(neg_visible_probs.T,
              neg_hidden_probs)
        error = np.sum(
            (data - neg_visible_probs)**2)
        errorsum = error + errorsum
        # momentum
        if epoch > 5:
          momentum = finalmomentum
        else:
          momentum = initialmomentum
        # Update weights
        delta = (pos_associations -
          neg_associations) / num_examples
        vishid = self.weights
            [1:self.num_visible + 1,
              1:self.num_hidden + 1]
        vishid = np.insert(
            vishid, 0, 0, axis=0)
        vishid = np.insert(
            vishid, 0, 0, axis=1)
        self.weightinc =
            momentum * self.weightinc +
```

```
            self.learning_rate *
              (delta -
                weightcost * vishid)
        self.weightinc[0, 0] = 0
        self.weights += self.weightinc
        self.weights[0, 0] = 0
        count += 1
        print("Count %s: error is %s" %
            (count, error))
        # Save weights and error
        if self.path:
          datafile = self.path+'weights'
          with open(datafile, 'wb') as fp:
            pickle.dump(self.weights, fp)
          datafile = self.path+'count.txt'
          with open(datafile, 'at') as fp:
            fp.write("%s,%s\n" %
                  (count, error))
      if self.path:
        datafile = self.path + 'epoch.txt'
        with open(datafile, 'at') as fp:
          fp.write("%s,%s\n" %
                (epoch, errorsum))

  def run_visible_for_hidden(self,
                             batch_data):
    """
    Assuming the RBM has been trained (so
    that weights for the network have been
    learned), run the network on a set of
    visible units, to get probabilities of
    the hidden units.
    :param batch_data: visible units data,
        type: list of np.array,
        every np.array is a matrix
        where each row is a example
        consisting of the states of
        visible units.
        i.e. every np.array is a batch of
        visible units data set
    :return:  the probabilities of the
        hidden units,
        type: list of np.array,
        every np.array is a batch
        of hidden units data set,
        corresponding to the input
    """
    batch_pos_hidden_probs = []
    for data in batch_data:
      # Insert bias into the first column
      data = np.insert(data, 0, 1, axis=1)
      # Calculate the activations of
      # hidden units.
      hidden_activations =
            np.dot(data, self.weights)
      # Calculate the probabilities of
      # turning the hidden units on.
      hidden_probs = self._logistic(
          hidden_activations)
      pos_hidden_probs =hidden_probs[:, 1:]
      batch_pos_hidden_probs.
          append(pos_hidden_probs)
    return batch_pos_hidden_probs

  def run_hidden_for_visible(self,
                             batch_data):
    """
    Assuming the RBM has been trained (so
    that weights for the network have been
    learned),run the network on a set of
    hidden units, to get probabilities of
    the visible units.
    :param batch_data: hidden units data,
        type: list of np.array,
        every np.array is a matrix
        where each row is a example
        consisting of the states of
        hidden units.
        i.e. every np.array is a batch
        of hidden units data set
    :return:  the probabilities of the
        visible units,
        type: list of np.array,
        every np.array is a batch
        of visible units data set,
        corresponding to the input
    """
    batch_neg_visible_probs = []
    for data in batch_data:
      # Insert bias into the first column
      data = np.insert(data, 0, 1, axis=1)
      # Calculate the activations of
      # the visible units.
      visible_activations =
          np.dot(data, self.weights.T)
      # Calculate the probabilities of
      # turning the visible units on.
      visible_probs = self._logistic(
          visible_activations)
      neg_visible_probs =
          visible_probs[:, 1:]
      batch_neg_visible_probs.
          append(neg_visible_probs)
    return batch_neg_visible_probs

  def predict(self, batch_data,
    soft_max=10):
    """
    Assuming the RBM has been trained (so
    that weights for the network have been
    learned), run the network on a set of
    test data, to get recognition results
    (only perform digits recognition)
    This prediction method is especially
    designed for the visible units
    including the label(softmax)
    :param batch_data: visible units data,
        type: list of np.array,
        every np.array is a matrix
        where each row is a example
        consisting of the states of
        visible units.
```

```
        i.e. every np.array is a batch
        of visible units data set
    :param soft_max:  the dimension of
    label, only can take value of 4
    or 10
        4 means the label is expressed
        as binary
        10 means the state of each
        dimension infer whether it
        belongs to that class
    :return:  the classification result,
        type: list of list of int,
        list2 is a batch of answers,
        corresponding to the input
    """
    final_ans = []
    for data in batch_data:
      ans = []
      num_examples = data.shape[0]
      data = np.insert(data, 0, 1, axis=1)
      data = np.split(data, num_examples)
      for item in data:
        hidden_activations =
            np.dot(item, self.weights)
        vbias_energy =
            hidden_activations[0, 0]
        hidden_probs = self._logfree(
            hidden_activations)
        hidden_probs[:, 0] = 0
        free_energy =
            - np.sum(hidden_probs) -
              vbias_energy
        min_free_energy = free_energy
        tmp_ans = 0
        for number in range(1, 10):
          tmpitem = item.copy()
          if soft_max == 10:
            tmpitem[0,
              self.num_visible - 9:
                self.num_visible + 1] = 0
            tmpitem[0,
              self.num_visible -
                (9 - number)] = 1
          else:
            if soft_max == 4:
              label = bin(number)
              label = label[::-1]
              length = len(label)
              for i in range(0, length -2):
                tmpitem[0,
                  self.num_visible +i-3] =
                  int(label[i])
              if length != 6:
                for i in
                      range(1, 7 - length):
                  tmpitem[0,
                      self.num_visible
                        - (6 - length) +
                              i] = 0
          hidden_activations =
              np.dot(tmpitem, self.weights)
          vbias_energy =
              hidden_activations[0, 0]
          hidden_probs = self._logfree(
              hidden_activations)
          hidden_probs[:, 0] = 0
          free_energy = -
              np.sum(hidden_probs) -
                vbias_energy
          if free_energy < min_free_energy:
            tmp_ans = number
            min_free_energy = free_energy
        ans.append(tmp_ans)
      final_ans.append(ans)
    return final_ans

  def _logistic(self, x):
    # np.tanh is more stable than np.exp
    # return 1.0 / (1 + np.exp(-x))
    return .5 * (1 + np.tanh(.5 * x))

  def _logfree(self, x):
    return np.log(1 + np.exp(x))
```

.1.1.2 RBMs on each class: RBMs are trained separately on each classes

```
class RBM_each:
  def __init__(self, num_visible,
      num_hidden, learning_rate=0.1,
                          Path=None):
    """
    Because we only recognize 10 numbers,
    so the RBM_each consists of 10 RBMs
    :param num_visible:
        the number of visible units
    :param num_hidden:
        the number of hidden units
    :param learning_rate:
        the learning rate of RBM
    :param Path:
        the path where we store the
        parameters of RBM, and it
        ends with //
    """
    self.num_hidden = num_hidden
    self.num_visible = num_visible
    self.learning_rate = learning_rate
    self.path = Path
    self.rbms = []
    for i in range(0, 10):
      tmppath = Path
      tmppath += ('rbm-%d' % i)
      os.mkdir(tmppath)
      tmppath += '\\'
      r = RBM(num_visible=num_visible,
             num_hidden=num_hidden,
               learning_rate=learning_rate,
                 Path=tmppath)
      self.rbms.append(r)
```

```
def train(self, train_data, pieces=100,
                               max_epochs=50):
    """
    Train Function
    :param train_data:
      training data, type: list of np.array
      every np.array is a matrix
      where each row is a training example
      consisting of the states of visible
      units.
      i.e. each np.array is a training set
      of a class
    :param pieces:
      the number of training example in one
      batch of a training set of a class
    :param max_epochs: the max epochs of
      the training operation
    """
    for i in range(0, 10):
      batch_data = np.array_split(
          train_data[i],
            train_data[i].shape[0]/pieces)
      r = self.rbms[i]
      r.train(batch_data,
            max_epochs=max_epochs)
      print(r.weights)
      print("Train RBM%d Successfully" % i)

def predict(self, test):
    """
    Assuming the RBM has been trained (so
    that weights for the network have been
    learned),
    run the network on a set of test data,
    to get recognition results (only
    perform digits recognition)
    :param test: visible units data
        type: list of np.array,
        each np.array consists of
        one row and is a example
        consisting of the states of
        visible units.
    :return: the prediction result
        type:list
    """
    ans=[]
    for item in test:
      minerror = 0
      tmpans = 0
      tmpitem = item.copy()
      tmpitem = [tmpitem]
      for number in range(0, 10):
        r = self.rbms[number]
        hidden_probs =
            r.run_visible_for_hidden(
                                   tmpitem)
        visible_probs_batches =
            r.run_hidden_for_visible(
                              hidden_probs)
        visible_probs =
            visible_probs_batches[0]
        error = np.sum(np.square
            (item - visible_probs))
        if number == 0:
          minerror = error
          tmpans = 0
        else:
          if error < minerror:
            tmpans = number
            minerror = error
      ans.append(tmpans)
    return ans
```

.1.1.3 RBMs network: The Back Propagation Procedure in Function train, Function _classify_init, Function _classify, Function _minimize are Python adaption written by Lin Yujun. The origin Matlab Code are provided by Hinton.

```
class RBM_net:
  def __init__(self, layers=3, dim=None,
          learning_rate=0.1, Path=None,
                                mode=0):
    """
    Initial Function
    :param layers:
      the layers or numbers of RBM
      the hidden units of the former
      RBM is the visible units of
      the latter RBM
    :param dim:
      the visible units number and hidden
      units number of each RBM, type: list
      the i-th elements of list is the the
      visible units number of the i-th RBM
      the i+1-th elements of list is the
      hidden units number of the i-th RBM
    :param learning_rate:
      learning rate of RBM
    :param Path:
      the path where we store the
      parameters of RBM net, and it
      ends with //
    :param mode:
      label is used as visible
      units under mode 1; (method 3)
      otherwise, mode 0 (method 4)

    w_class:   under mode 0,
      (1 + num_visible of the top RBM) *
        the dimension of label(softmax)
      the weight matrix between the hidden
      units of the top RBM and the
      label(softmax)
    """
    self.layers = layers
    if not dim:
      self.dim = [784, 500, 500, 2000, 10]
    else:
      self.dim = dim
```

```
    self.learning_rate = learning_rate
    self.Path = Path
    self.rbms = []
    self.mode = mode
    for i in range(0, layers):
      num_visible = dim[i]
      num_hidden = dim[i+1]
      if i == layers - 1 and mode == 1:
        num_visible += dim[layers+1]
      path = self.Path + 'rbm' +
          ('-%d' % i) + ('-%dh' %
              num_hidden) + ('-%dv' %
                  num_visible)
      import os
      if not os.path.exists(path):
        os.mkdir(path)
      path += '\\'
      r = RBM(num_visible=num_visible,
          num_hidden=num_hidden,
                learning_rate=0.1,
                              Path=path)
      self.rbms.append(r)
    datafile = self.Path + 'w_class'
    import os
    if os.path.isfile(datafile):
      with open(datafile, 'rb') as fp:
        self.w_class = pickle.load(fp)
      print("Load w_class Successfully!")
    else:
      # Initialize the w_class, using a
      # Gaussian distribution
      # mean 0 and standard deviation 0.1.
      self.w_class = 0.1 *
          np.random.randn(dim[layers]+1,
              dim[layers+1])
      with open(datafile, 'wb') as fp:
        pickle.dump(self.w_class, fp)
      print("Create W_class Successfully!")
    print("Create RBM_net Successfully")

  def train_rbms(self, batch_data,
      batch_label=None, max_epochs_1=50,
          max_epochs_2=200, test_set=None,
              test_label_set=None,
                      test_name_set=None):
    """
    Train Function
    Under mode 0, also Prediction Function
    :param batch_data:
      training data, type: list of np.array
      every np.array is a matrix
      where each row is a training example
      consisting of the states of visible
      units.
      i.e. every np.array is a batch of
      training set
    :param batch_label:
      training data label
      type: list of list
      every list is a label of training
      example corresponding to batch_data.
    :param max_epochs_1:
      the max epochs of the RBMs
      training operation
    :param max_epochs_2:
      the max epochs of the w_class
      training operation
      ( weights of each RBM is
      updated either)
      used under mode 0
    :param test_set:
      the set of test data set,
      type: list of list of np.array,
      every list is a test data set
      every np.array is a matrix
      where each row is a example
      consisting of the states of
      visible units.
      i.e. every np.array is a batch of
      visible units data set
      used under mode 0
    :param test_label_set:
      the set of the test data label set,
      type: list of list of list,
      ( we call list 1 of list 2 of list 3)
      every list2 is a test data label set
      every list3 is the label
      corresponding to the row of
      np.array in test_set
      used under mode 0
    :param test_name_set:
      the set of the test data name,
      type: list of string
      every string is name of the test data
      set corresponding to those in
      test_set
      used under mode 0
    """
    train_data = batch_data.copy()

    for i in range(0, self.layers):
      # In mode 1, the visible units of the
      # top RBM consists of the hidden
      # units of the former RBM and
      # the label of the test data
      if i == self.layers - 1 and
                                self.mode == 1:
        train_data = list(map(lambda y:
            np.array(list(map(lambda x:
                x[0].tolist()+x[1],
                  zip(y[0], y[1])))),
                    zip(train_data,
                        batch_label)))
      self.rbms[i].train(train_data,
          max_epochs=max_epochs_1)
      train_data = self.rbms[i].
          run_visible_for_hidden(
                                train_data)
    print("Train RBM_net Successfully")

    if self.mode == 0:
      if not (test_set == None):
```

```
    num_teset_set = len(test_set)
    test_result = [0] * num_teset_set
    test_result_err =
              [0] * num_teset_set
  for epoch in range(0, max_epochs_2):
    num_batches = len(batch_data)
    counter = 0
    err_cr = 0
    for batch in range(0, num_batches):
      data = batch_data[batch]
      label =
          np.array(batch_label[batch])
      hidden_probs = np.insert(
          data, 0, 1, axis=1)
      for i in range(0, self.layers):
        hidden_activations =
            np.dot(hidden_probs,
                self.rbms[i].weights)
        hidden_probs = self._logistic(
            hidden_activations)
        hidden_probs[:, 0] = 1
      label_out = np.exp(np.dot(
          hidden_probs, self.w_class))
      label_out = np.divide(label_out,
          np.array([np.sum(label_out,
                axis=1).tolist()]).T)
      J = np.argmax(label_out, axis=1)
      J1 = np.argmax(label, axis=1)
      counter += np.count_nonzero(J-J1)
      err_cr -= np.sum(np.multiply(
            label, np.log(label_out)))
    if self.Path:
      datafile =
          self.Path+'train_epoch.txt'
      with open(datafile, 'at') as fp:
        fp.write('epoch: %s, wrong num:
            %s, error: %s\n' % (epoch,
                counter,
                  err_cr/num_batches))
    print('epoch: %s \n train: wrong:
        %s, error: %s' % (epoch,
          counter, err_cr/num_batches))

    if not (test_set == None):
      num_teset_set = len(test_set)
      for i in range(0, num_teset_set):
        tmp_result = r.predict(
            batch_test=test_set[i],
                batch_test_label=
                  test_label_set[i],
                    test_name=
                      test_name_set[i])
        if epoch == 0 or tmp_result[1]
            < test_result[i] or
              (tmp_result[1] ==
                test_result[i] and
                  tmp_result[2] <
                    test_result_err
                                  [i]):
          test_result[i] =
              tmp_result[1]
          test_result_err[i] =
              tmp_result[2]
          datafile = self.Path +
              test_name_set[i] +
                '\\w_class'
          with open(datafile, 'wb')
                                as fp:
            pickle.dump(self.w_class,
                                  fp)
          for j in range(0,
                        self.layers):
            datafile = self.Path +
                test_name_set[i] +
                  '\\weights-' +
                      ('%d' % j)
            with open(datafile, 'wb')
                                as fp:
              pickle.dump(self.rbms[j].
                        weights, fp)
          ans = tmp_result[0]
          for j in range(0,
                      ans.__len__()):
            ans[j] = str(ans[j])
          str_convert = ''.join(ans)
          datafile = self.Path +
              test_name_set[i] +
                '\\best_result.txt'
          with open(datafile, 'wt')
                                as fp:
            fp.write('epoch: %d, wrong
                number: %d,error: %d\n'
                  % (epoch,
                    tmp_result[1],
                      tmp_result[2]))
            fp.write('%s\n' %
                        str_convert)
          print('Save Successfully!')
  # combine 10 batches into 1 batch
  # for training
  tt = 0
  for batch in range(0,
              int(num_batches/10)):
    tt += 1
    data = []
    label = []
    for kk in range(0, 10):
      data += batch_data[(tt - 1) *
        10 + kk].tolist()
      label += batch_label[(tt - 1) *
        10 + kk]
    data = np.array(data)
    # max_iter is the time of linear
    # searches we perform conjugate
    # gradient with
    max_iter = 3
    # first update top-level weights
    # (w_class) holding other weights
    # fixed.
    if epoch < 6:
      hidden_probs = np.insert(
        data, 0, 1, axis=1)
```

```
            for i in range(0, self.layers):
              hidden_activations = np.dot(
                hidden_probs,
                    self.rbms[i].weights)
              hidden_probs =self._logistic(
                hidden_activations)
              hidden_probs[:, 0] = 1
            VV = [self.w_class.copy()]
            tmp = self._minimize(func=0,
              x=VV, parameters=
                  [hidden_probs, label],
                        length=max_iter)
            self.w_class = tmp[0]
            import os
            datafile = self.Path+'w_class'
            if os.path.isfile(datafile):
              with open(datafile, 'wb')
                                as fp:
                pickle.dump(self.w_class,
                                          fp)
          else:
            # the update all weights
            # (w_class and weights of
            # each RBMs)
            VV = [0] * (self.layers + 1)
            VV[0] = self.w_class.copy()
            for i in range(0,self.layers):
              VV[i+1] = self.rbms[i].
                                  weights
            tmp = self._minimize(func=1,
                    x=VV, parameters=
                          [data, label],
                            length=max_iter)
            self.w_class = tmp[0]
            for i in range(0,self.layers):
              self.rbms[i].weights =
                                tmp[i+1]
            import os
            datafile = self.Path +
                                'w_class'
            if os.path.isfile(datafile):
              with open(datafile, 'wb')
                                  as fp:
                pickle.dump(self.w_class,
                                      fp)
            for i in range(0,self.layers):
              datafile = self.rbms[i].path
                                + 'weights'
              if os.path.isfile(datafile):
                with open(datafile, 'wb')
                                  as fp:
                  pickle.dump(self.rbms[i].
                              weights, fp)

    def predict(self, batch_test,
              batch_test_label, test_name):
      """
      Prediction Function in mode 1
      :param batch_test:
        visible units data
        type: list of np.array,
        every np.array is a matrix
        where each row is a example
        consisting of the states of
        visible units.
        i.e. every np.array is a batch of
        visible units data set
      :param batch_test_label:
        label, type: list of list,
        every list is a label of example
        corresponding to batch_test.
      :param test_name:
        the name of the test set
        type: string
      :return:
        a list, the first element is also a
        list, consisting of the prediction
        answer,
        corresponding to the batch_test
        the second element is number of the
        wrong prediction
      """
      if self.mode == 1:
        test_data = batch_test.copy()
        for i in range(0, self.layers):
          if i == self.layers - 1:
            test_data = list(map(lambda y:
                np.array(list(map(lambda x:
                  x+[0]*self.dim[self.layers
                      +1], y))), test_data))
            ans = self.rbms[i].predict(
                test_data, soft_max=
                    self.dim[self.layers+1])
          else:
            test_data = self.rbms[i].
                run_visible_for_hidden(
                                  test_data)
        test_num_batches = len(batch_test)
        counter = 0
        err = 0
        for batch in range(0,
                         test_num_batches):
          J = np.array(ans[batch])
          J1 = np.array(
                batch_test_label[batch])
          J1 = np.argmax(J1, axis=1)
          counter += np.count_nonzero(J-J1)
        if self.Path:
          datafile = self.Path + test_name
          if not os.path.exists(datafile):
            os.mkdir(datafile)
          datafile += '\\test_result.txt'
          for i in range(0, ans.__len__()):
            ans[i] = str(ans[i])
          str_convert = ''.join(ans)
          with open(datafile, 'at') as fp:
            fp.write('%s\n' % str_convert)
        print(' %s, wrong: %s' % (test_name,
                                  counter))
        print(ans)
      else:
        test_num_batches = len(batch_test)
```

```
    counter = 0
    err_cr = 0
    ans = []
    for batch in range(0,
                       test_num_batches):
      data = batch_test[batch]
      label = np.array(
                 batch_test_label[batch])
      hidden_probs = np.insert(
                   data, 0, 1, axis=1)
      for i in range(0, self.layers):
        hidden_activations = np.dot(
            hidden_probs, self.rbms[i].
                            weights)
        hidden_probs = self._logistic(
            hidden_activations)
        hidden_probs[:, 0] = 1
      label_out = np.exp(np.dot(
          hidden_probs, self.w_class))
      label_out = np.divide(label_out,
          np.array([np.sum(label_out,
                axis=1).tolist()]).T)
      J = np.argmax(label_out, axis=1)
      J1 = np.argmax(label, axis=1)
      counter += np.count_nonzero(J-J1)
      err_cr -= np.sum(np.multiply(label,
          np.log(label_out)))
      J = J.tolist()
      ans.append(J)
    err = err_cr/test_num_batches
    if self.Path:
      datafile = self.Path + test_name
      if not os.path.exists(datafile):
        os.mkdir(datafile)
      datafile += '\\test_result.txt'
      with open(datafile, 'at') as fp:
        fp.write('%s,%s\n' % (counter,
                            err))
    print(' %s, wrong: %s, error: %s' %
          (test_name, counter, err))
    print(ans)
  return [ans, counter, err]

def _logistic(self, x):
  # return 1.0 / (1 + np.exp(-x))
  return .5 * (1 + np.tanh(.5*x))

def _classify_init(self, w_class,
                   hidden_probs, label):
  """
  the loss function of the RBM net with
  each RBM weights hold
  :param w_class: w_class
  :param hidden_probs:
    the output (hidden units) of the top
    RBM, suppose the input (visible
    units) of RBM net is data
  :param label:  the label of data
  :return: a list,
    the first elements is value of the
    loss function with each RBM weights
    hold
    the second elements is a list,
    consisting the partial derivative
    of the function
  """
  label_out = np.exp(np.dot(hidden_probs,
                            w_class))
  label_out = np.divide(label_out,
      np.array([np.sum(label_out,
                axis=1).tolist()]).T)
  f = - np.sum(np.multiply(label,
                    np.log(label_out)))
  IO = label_out - label
  df = np.dot(hidden_probs.T, IO)
  return [f, [df]]

def _classify(self, w_class, weights,
                          data, label):
"""
the loss function of the RBM net
:param w_class: w_class
:param weights: a list,
  consisting of weights of each RBM
:param data:
  the input (visible units) of first RBM
:param label: the label of the data
:return: a list,
  the first elements is value of
  the loss function
  the second elements is a list,
  consisting the partial derivative
  of the function
  corresponding to w_class and weights[i]
"""
# hidden_probs is a list, the i-th
# elements is the input of the i-th
# RBM or the output of the i-1th RBM
hidden_probs = [0] * (self.layers+1)
hidden_probs[0] = np.insert(
                    data, 0, 1, axis=1)
for i in range(0, self.layers):
  hidden_activations =
      np.dot(hidden_probs[i], weights[i])
  hidden_probs[i+1] = self._logistic(
                  hidden_activations)
  hidden_probs[i+1][:, 0] = 1
label_out = np.exp(np.dot(
    hidden_probs[self.layers], w_class))
label_out = np.divide(label_out,
        np.array([np.sum(label_out,
                axis=1).tolist()]).T)
f = - np.sum(np.multiply(label,
                    np.log(label_out)))
IO = label_out - label
dw_class = np.dot(
        hidden_probs[self.layers].T, IO)
tmp1 = np.dot(IO, w_class.T)
tmp2 = np.subtract(1,
            hidden_probs[self.layers])
Ix = np.multiply(np.multiply(tmp1,
        hidden_probs[self.layers]), tmp2)
```

```
dw = [0] * (self.layers + 1)
dw[0] = dw_class
for i in range(0, self.layers):
  dw[self.layers-i] = np.dot(
      hidden_probs[self.layers-1-i].T,Ix)
  if i < self.layers - 1:
    tmp1 = np.dot(Ix,
        weights[self.layers-1-i].T)
    tmp2 = np.subtract(1,
        hidden_probs[self.layers-1-i])
    Ix = np.multiply(np.multiply(tmp1,
        hidden_probs[self.layers-1-i]),
                                  tmp2)
return [f, dw]

def _minimize(self, func, x, parameters,
                              length):
  """
  Minimize a differentiable multivariate
  function
  :param func: the type of function,
    1 means _classify,
    0 means _classify_init
  :param x: the initial value of the
    variation, here, it is a list,
    if func = 0, then its element
    is w_class
    if func = 1, then its elements
    are w_class, weights of each RBM
  :param parameters: the unchanged
    parameters of function represented
    by func
  :param length:
    the maximum number of line searches
  :return:
    the result with which the function
    value is smaller than before
    here, it is a list
    if func = 0, then its element
    is w_class
    if func = 1, then its elements
    are w_class, weights of each RBM
  """
  INT = 0.1
  EXT = 3.0
  MAX = 20
  RATIO = 10
  SIG = 0.1
  RHO = SIG / 2.0
  i = 0
  Is_failed = 0
  if func:
    tmp = self._classify(w_class=x[0],
        weights=x[1:],data=parameters[0],
        label=parameters[1])
  else:
    tmp = self._classify_init(
        w_class=x[0],
        hidden_probs=parameters[0],
        label=parameters[1])
  f0 = tmp[0]
  df0 = tmp[1]
  s = list(map(lambda x: - x, df0))
  d0 = - sum(list(map(lambda x:
      np.sum(np.multiply(x, x)), s)))
  x3 = 1.0 / (1 - d0)
  while i < length:
    i += 1
    X0 = x.copy()
    F0 = f0
    dF0 = df0.copy()
    M = MAX
    while 1:
      x2 = 0
      f2 = f0
      d2 = d0
      f3 = f0
      df3 = df0.copy()
      success = 0
      while (not success) and M > 0:
        M -= 1
        i += 1
        newx = list(map(lambda x:
            x[0] + x3 * x[1],
            zip(x, s)))
        if func:
          tmp = self._classify(
              w_class=newx[0],
              weights=newx[1:],
              data=parameters[0],
              label=parameters[1])
        else:
          tmp = self._classify_init(
              w_class=newx[0],
              hidden_probs=parameters[0],
              label=parameters[1])
        f3 = tmp[0]
        df3 = tmp[1]
        errf = np.zeros_like(f3)
        errf = np.count_nonzero(
            np.isinf(f3, errf)) > 0
        if errf:
          x3 = (x2 + x3) / 2.0
          continue
        errf = np.zeros_like(f3)
        errf = np.count_nonzero(
            np.isnan(f3, errf)) > 0
        if errf:
          x3 = (x2 + x3) / 2.0
          continue
        for element in df3:
          errdf = np.zeros_like(element)
          errdf = np.count_nonzero(
              np.isinf(element,
              errdf)) > 0
          if errdf:
            x3 = (x2 + x3) / 2.0
            break
          errdf = np.zeros_like(element)
          errdf = np.count_nonzero(
              np.isnan(element,
              errdf)) > 0
```

```
        if errdf:
          x3 = (x2 + x3) / 2.0
          break
      if errdf:
        continue
      success = 1
    if f3 < F0:
      X0 = list(map(lambda x: x[0] +
            x3 * x[1], zip(x,s)))
      F0 = f3
      dF0 = df3.copy()
    d3 = sum(list(map(lambda x:
          np.sum(np.multiply(x[0],
          x[1])), zip(df3, s))))
    if d3 > SIG*d0 or
          f3 > (f0 + x3 * RHO * d0)
          or M == 0:
      break
    x1 = x2
    f1 = f2
    d1 = d2
    x2 = x3
    f2 = f3
    d2 = d3
    A = 6 * (f1 - f2) +
          3 * (d2 + d1) * (x2 - x1)
    B = 3 * (f2 - f1) -
          (2 * d1 + d2) * (x2 - x1)
    x3 = x1 - d1 *
          ((x2 - x1) ** 2.0) /
          (B + ((B * B - A * d1 *
          (x2 - x1)) ** (1/2.0)))
    if (not np.isreal(x3)) or
          (np.isnan(x3)) or
          (np.isinf(x3)) or x3 < 0:
      x3 = x2 * EXT
    else:
      if x3 > x2 * EXT:
        x3 = x2 * EXT
      else:
        if x3 < (x2 + INT *
                                (x2 - x1)) :
          x3 = (x2 + INT * (x2 - x1))

  while (abs(d3) > -SIG * d0
          or f3 > f0 + x3 * RHO * d0)
          and M > 0:
    if d3 > 0 or
            f3 > (f0 + x3 * RHO * d0):
      x4 = x3
      f4 = f3
      d4 = d3
    else:
      x2 = x3
      f2 = x3
      d2 = d3
    if f4 > f0:
      x3 = x2-(0.5 * d2 * ((x4 - x2)
            ** 2)) / (f4 - f2 - d2 *
            (x4 - x2))
    else:
      A = 6 * (f2 - f4) / (x4 - x2) +
            3 * (d4 + d2)
      B = 3 * (f4 - f2) - (2 * d2 +
            d4) * (x4 - x2)
      x3 = x2 + ((B * B - A * d2 *
            ((x4 - x2) ** 2)) **
            (1/2.0) - B) / A
    if np.isnan(x3) or np.isinf(x3):
      x3 = (x2 + x4) / 2.0
    x3 = max(min(x3, (x4 - INT * (x4 -
          x2))), (x2 + INT *
          (x4 - x2)))
    newx = list(map(lambda x: x[0] + x3
          * x[1], zip(x, s)))
    if func:
      tmp = self._classify(
            w_class=newx[0],
            weights=newx[1:],
            data=parameters[0],
            label=parameters[1])
    else:
      tmp = self._classify_init(
          w_class=newx[0],
          hidden_probs=parameters[0],
          label=parameters[1])
    f3 = tmp[0]
    df3 = tmp[1]
    if f3 < F0:
      X0 = list(map(lambda x: x[0] +
          x3 * x[1], zip(x, s)))
      F0 = f3
      dF0 = df3.copy()
    M = M - 1
    d3 = sum(list(map(lambda x:
          np.sum(np.multiply(x[0],
          x[1])), zip(df3, s))))

  if (abs(d3) < -SIG * d0)
        and (f3 < f0 + x3 * RHO * d0):
    x = list(map(lambda x: x[0] + x3 *
          x[1], zip(x, s)))
    f0 = f3
    u33 = sum(list(map(lambda x:
          np.sum(np.multiply(x[0],
          x[1])), zip(df3, df3))))
    u03 = sum(list(map(lambda x:
          np.sum(np.multiply(x[0],
          x[1])), zip(df0, df3))))
    u00 = sum(list(map(lambda x:
          np.sum(np.multiply(x[0],
          x[1])), zip(df0, df0))))
    s = list(map(lambda x: (u33 -
          u03)/u00 * x[0] - x[1],
          zip(s, df3)))
    df0 = df3.copy()
    d3 = d0
    d0 = sum(list(map(lambda x:
          np.sum(np.multiply(x[0],
          x[1])), zip(df0, s))))
    if d0 > 0:
      s = list(map(lambda x: - x, df0))
```

```
        d0 = -sum(list(map(lambda x:
            np.sum(np.multiply(x, x)),
                                s)))
      realmin = np.finfo(np.double).tiny
      x3 = x3 * min(RATIO, (d3 / (d0 -
                            realmin)))
      Is_failed = 0
    else:
      x = X0.copy()
      f0 = F0
      df0 = dF0.copy()
      if Is_failed or i > length:
        break
      s = list(map(lambda x: - x, df0))
      d0 = -sum(list(map(lambda x:
          np.sum(np.multiply(x, x)), s)))
      x3 = 1 / (1 - d0)
      Is_failed = 1
  return x
```

.1.2 Image Process

Reconstruction function code comes from "Book Programming Computer Vision with Python". [5]

```
import numpy as np
from scipy import ndimage
from PIL import Image
from numpy import *
from scipy.misc import imsave

def Reconstruct (im, U_init, tolerance=0.2,
                  tau=0.1, tv_weight=10):
  m, n = im.shape
  # initial
  U = U_init
  # x component in dual domain
  Px = im
  # y component in dual domain
  Py = im
  error = 1
  while (error > tolerance):
    Uold = U
    # gradient of origin
    # x component of gradient of U
    GradUx = roll(U, -1, axis=1)-U
    # y component of gradient of U
    GradUy = roll(U, -1, axis=0)-U
    # update component in dual domain
    PxNew = Px + (tau/tv_weight)*GradUx
    PyNew = Py + (tau/tv_weight)*GradUy
    NormNew = maximum(1, sqrt(PxNew**2+
                            PyNew**2))
    # update x component in dual domain
    Px = PxNew/NormNew
    # update y component in dual domain
    Py = PyNew/NormNew
    # x component translation
    RxPx = roll(Px,1,axis=1)
    # y component translation
    RyPy = roll(Py,1,axis=0)
    # divergence of dual domain
    DivP = (Px-RxPx)+(Py-RyPy)
    # update origin
    U = im + tv_weight*DivP
    # update the error
    error = linalg.norm(U-Uold)/sqrt(n*m)
  return U

def process (im, fig_name):
  # increase contrast
  im = np.array(im)
  im = 255 - im
  im = np.multiply(im, 255/np.max(im))
  im = 255 - im
  # first filtering
  im=ndimage.gaussian_filter(im, sigma=0.5)
  im=ndimage.percentile_filter(im, 20, 2)
  # threshold process
  a = np.mean(im)
  b = np.min(im)
  W1 = im.shape[0]
  W2 = im.shape[1]
  for i in range(0, W1):
    for j in range(0, W2):
      if im[i, j] > (3*a/4+1*b/4):
        im[i, j] = 255
      else:
        im[i, j] = 0
  # filtering for initial value and
  # less noise value for
  # reconstruction function
  M=ndimage.median_filter(im, size=2)
  G=ndimage.gaussian_filter(im, sigma=0.5)
  # Gauss Filtering Image as noisy input
  # Median Filtering Image as initial value
  # reconstruction image
  U = Reconstruct(G, M)
  # thereshold select and maximum filtering
  a = np.mean(U)
  b = np.min(U)
  for i in range(0, W1):
    for j in range(0, W2):
      if U[i, j] > a:
        U[i, j] = 255
  T = ndimage.maximum_filter(U, 1)
  # delete small region in the image
  for i in range(0, W1):
    for j in range(0, W2):
      if T[i, j] > 200:
        T[i, j] = 0
      else:
        T[i, j] = 1
  label_im, nb_labels = ndimage.label(T)
  sizes = ndimage.sum(T, label_im,
                    range(nb_labels + 1))
  a = mean(sizes)
  if size(sizes) > 5:
    mask_size = sizes < 2*a
    remove_pixel = mask_size[label_im]
```

```
      label_im[remove_pixel] = 0
    else:
      if size(sizes) > 2:
        mask_size = sizes < 0.8 * a
        remove_pixel = mask_size[label_im]
        label_im[remove_pixel] = 0
    # coloring the image
    a = max(sizes)
    for i in range(0, W1):
      for j in range(0, W2):
        if label_im[i, j] > 0:
          label_im[i, j] = (1-(sizes
              [label_im[i, j]]/a-1)**2) * 255
        else:
          label_im[i, j] = 0
    # box the digit
    for i in range(0, W1):
      if np.sum(label_im[i]) > 0:
        break
    start_row = i
    for i in range(0, W1):
      if np.sum(label_im[W1-1-i]) > 0:
        break
    end_row = W1-1-i
    for i in range(0, W2):
      if sum(label_im.T[i]) > 0:
        break
    start_column = i
    for i in range(0, W2):
      if sum(label_im.T[W2-1-i]) > 0:
        break
    end_column = W2-1-i
    band_row = end_row - start_row + 1
    band_column = end_column -
                            start_column + 1
    if band_row > band_column:
      mid_column = round((start_column +
                            end_column)/2.0)
      start_column = round(mid_column -
                              band_row/2.0)
      end_column = round(mid_column +
                            band_row/2.0-1)
      if start_column < 0:
        start_column = 0
      if end_column > 31:
        end_column = 31
    else:
      mid_row = round((start_row +
                              end_row)/2.0)
      start_row = round(mid_row -
                            band_column/2.0)
      end_row = round(mid_row +
                          band_column/2.0-1)
      if start_row < 0:
        start_row = 0
      if end_row > 31:
        end_row =31
    # clipping image
    label_im = label_im[start_row:end_row+1,
                  start_column:end_column+1]
    imsave('D:\\Download\\study\\pattern
            recognition\\hw\\data\\2.png',
            label_im)
    baseim = Image.open('D:\\Download\\study
            \\pattern recognition\\hw
            \\data\\1.png')
    floatim = Image.open('D:\\Download\\study
            \\pattern recognition\\hw
            \\data\\2.png')
    # make digits in the center of
    # 28pixels *28 pixels image
    # It is the size of MNIST data image
    floatim = floatim.resize((19, 19),
                              Image.LANCZOS)
    baseim.paste(floatim, (5, 5))
    # save images
    baseim.save('D:\\Download\\study\\pattern
            recognition\\hw\\data\\
            test\\Process2\\'+fig_name)
    return baseim
```