SCIT

ASSIGNMENT COVERSHEET

STUDENT SURNAME

STUDENT DETAILS		
STUDENT NAME		
STUDENT NUMBER		
UOW EMAIL		
ASSIGNMENT DETAILS		
SUBJECT NAME		
SUBJECT NUMBER		
ASSIGNMENT NAME		
TUTORIAL / LAB TIME		
LECTURER / TUTOR		
SUBMISSION DETAILS		
DATE DUE		
STUDENT SIGNATURE		
STAFF ONLY		
RECEIVED BY		DATE/TIME LODGED
MARKER		GRADE

By signing this, I declare that:

1. This assignment meets all the requirements of the subject as detailed in the relevant subject outline, which I have read.
2. (a) This assessment item is entirely my work, except where I have included fully-documented references to the work of others.
   (b) The material contained in this assessment item has not previously been submitted for assessment.
3. I acknowledge that:
   (a) The marker of this assessment item may, for the purpose of assessing this assignment, reproduce this assignment and provide a copy to another member of academic staff.
   (b) If required to do so, I will provide an electronic copy of this assessment item to the marker.
4. Where Turnitin is used by the faculty: I acknowledge that the marker of this assessment item may communicate a copy of this assignment to a text-matching service such as Turnitin (which may retain a copy of this assignment on its database for the purpose of future text-matching).
5. I am aware that late submission without an authorised extension from the subject co-ordinator may incur a penalty. (See your subject outline for further information).

Please note: Assignments are not to be submitted by fax and must be submitted during lectures/tutorials/laboratories or directly to the academic. Only under special circumstances will the Administrative Staff collect assignments.

STUDENT REFERENCE			
STUDENT NAME		TUTOR	
STUDENT NUMBER		RECEIVED BY (Academic Signature)	
ASSIGNMENT NAME			
SUBJECT NUMBER		DATE / TIME	

FACULTY OF ENGINEERING & INFORMATION SCIENCES | UNIVERSITY OF WOLLONGONG

Name:_____ Student No: _____

University of Wollongong

School Computing and Information Technology

CSCI464/964 **Computational Intelligence** **Autumn 2019**

Assignment 2 **Marking Sheet** **10 marks**

(Note: Marking allocation is approximate.)

Part 1 SVM – 5 marks

- An introduction of this part and how the data are generated (0.5 mark) [ ]
- Indicate which kind of SVM classifier and justify your choice (1 mark) [ ]
- How you tune the parameters in SVMs (1 mark) [ ]
- The achieved classification accuracy (0.5 mark) [ ]
- Detailed discussion and analysis and compare with neural networks (1.5 marks) [ ]
- A printout of the commands that you (0.5 mark) [ ]
- Comment:

Part 2 SOM – 5 marks

- A brief introduction on the MNIST data set and the examples (0.5 mark) [ ]
- Describe the two training phases and how to set the learning parameters (0.5 mark) [ ]
- Compute the change of weights vs iteration, plot and describe its evolution (1 mark) [ ]
- Plot the weights in image for three stages (0.5 mark) [ ]
- Investigate various settings to train the SOM neural network (0.5 mark) [ ]
- Code can be compiled on UNIX and can run to give reasonable result (2 marks) [ ]
- Comment:

TOTAL [ ]

author

name : zhou fang

studentid : 6286914

Support Vector Machines (part 1)

1 Indroduction

In this part of assignment, we need to use the **Support Vector Machines(SVMs)** to solve the classification and regression problem which appears in assignment one. And as it known to us, The support vector machine(SVM) is a popular classifiaction technique(also support the regression), so in this assignment we use the LIBSVM to solve the three problem with following the steps thich introduced in the Guide. and then analyze the result in critical way.

1. 1 dataset

There are three datasets in this part of assignment : Two Spiral Problem, Abalone Age Problem, SPECT Heart Diagnosis Problem

- 1. Two Spiral Problem

  Before we proceed the svm_train, we need we to **transform the data set into the svm package**, which means that we need to rewrite the format of the input dataset into to the following format :

  ```
  // part of the dataset
  1 1:0.971354 2:0.209317
  0 1:-0.971354 2:-0.209317
  1 1:0.906112 2:0.406602
  0 1:-0.906112 2:-0.406602
  1 1:0.807485 2:0.584507
  0 1:-0.807485 2:-0.584507
  1 1:0.679909 2:0.736572
  ...
  ```

  i write a rewrite code to to reach this format data:

  ```
  # deal the data into the SVM format package
  def deal_spiralData():
      with open('dataset/1-SpiralData') as f:
          datas = f.readlines()
      svm_package = []
      times = 0
      for data in datas:
          if times == 192 :
  ```

```
            break
        if len(data) > 1:
            data_parts = data.split('\t')
            svm_package.append(data_parts[2].strip('\n') + " " + "1:"
+ data_parts[0] + " 2:" + data_parts[1] + "\n")
        times += 1

    for i in range(0, len(svm_package)-1):
        rannum = random.randint(0, len(svm_package)-1)
        # swap
        svm_package[i], svm_package[rannum] = svm_package[rannum],
svm_package[i]

    test = svm_package[:77]
    train = svm_package[77:]
    with open('dataset/svm_SpiralData', 'w') as f:
        for each_svm in train:
            f.write(each_svm)
    with open('dataset/svm_SpiralData_t', 'w') as f:
        for each_svm in test:
            f.write(each_svm)
```

We take the first 115 datas the as the test data and take the first 77 datas the as the test data

- 2. Abalone Age Problem

  This problem also need to rewire the format of the data to generate the data

```
def deal_data2():
    with open('dataset/data2.txt') as f:
        datas = f.readlines()
    svm_package = []
    # reformat
    for data in datas:
            if len(data) > 1:
                data_parts = data.split('\t')
                length = len(data_parts)
                svm_str = ""
                for i in range(0, length-2):
                    svm_str += str(i+1) + ":" + data_parts[i] + " "
                svm_package.append(data_parts[length-1].strip('\n')+"
"+svm_str.strip(' ')+'\n')

    # training data
    with open('dataset/svm_data2', 'w') as f:
        for each_svm in svm_package[:3677]:
            f.write(each_svm)

    # test data
```

```
    with open('dataset/svm_data2_t', 'w') as f:
        for each_svm in svm_package[3678:]:
            f.write(each_svm)
```

And this is the dataset after format:

- part of the training data

```
// part of the training dataset
0.789474 1:0 2:0.455 3:0.365 4:0.095 5:0.514 6:0.2245 7:0.101
8:0.15
0.368421 1:0 2:0.35 3:0.265 4:0.09 5:0.2255 6:0.0995 7:0.0485
8:0.07
0.473684 1:1 2:0.53 3:0.42 4:0.135 5:0.677 6:0.2565 7:0.1415
8:0.21
0.526316 1:0 2:0.44 3:0.365 4:0.125 5:0.516 6:0.2155 7:0.114
8:0.155
0.368421 1:2 2:0.33 3:0.255 4:0.08 5:0.205 6:0.0895 7:0.0395
8:0.055
0.421053 1:2 2:0.425 3:0.3 4:0.095 5:0.3515 6:0.141 7:0.0775
8:0.12
...
```

- part of the test data

```
// part of the test dataset
0.526316 1:1 2:0.605 3:0.49 4:0.15 5:1.1345 6:0.4305 7:0.2525
8:0.35
0.526316 1:0 2:0.61 3:0.45 4:0.19 5:1.0805 6:0.517 7:0.2495
8:0.2935
0.473684 1:1 2:0.61 3:0.495 4:0.165 5:1.0835 6:0.4525 7:0.273
8:0.317
0.578947 1:0 2:0.615 3:0.47 4:0.175 5:1.242 6:0.5675 7:0.287
8:0.317
0.578947 1:0 2:0.62 3:0.5 4:0.18 5:1.3915 6:0.726 7:0.2795 8:0.332
...
```

I use the training and test data predefined in head of the files

which means that we have 3677 training data and 500 test data

- 3. SPECT Heart Diagnosis Problem

  This problem also need to rewire the format of the data to generate the data

```
def deal_data3():

    with open('dataset/data3.txt') as f:
```

```
        datas = f.readlines()
    svm_package = []
    for data in datas:
            if len(data) > 1:
                data_parts = data.split(' ')
                length = len(data_parts)
                svm_str = ""
                for i in range(0, length-1):
                    svm_str += str(i+1) + ":" + data_parts[i] + " "
                svm_package.append(data_parts[length-1].strip('\n')+"
"+svm_str.strip(' ')+'\n')
    # training data
    with open('dataset/svm_data3', 'w') as f:
        for each_svm in svm_package[:3677]:
            f.write(each_svm)

    # test data
    with open('dataset/svm_data3_t', 'w') as f:
        for each_svm in svm_package[3678:]:
            f.write(each_svm)
```

And this is the dataset after format:

- part of the training data :

```
// part of the training dataset
1 1:59 2:52 3:70 4:67 5:73 6:66 7:72 8:61 9:58 10:52 11:72 12:71
13:70 14:77 15:66 16:65 17:67 18:55 19:61 20:57 21:68 22:66 23:72
24:74 25:63 26:64 27:56 28:54 29:67 30:54 31:76 32:74 33:65 34:67
35:66 36:56 37:62 38:56 39:72 40:62 41:74 42:74 43:64 44:67
1 1:72 2:62 3:69 4:67 5:78 6:82 7:74 8:65 9:69 10:63 11:70 12:70
13:72 14:74 15:70 16:71 17:72 18:75 19:66 20:65 21:73 22:78 23:74
24:79 25:74 26:69 27:69 28:70 29:71 30:69 31:72 32:70 33:62 34:65
35:65 36:71 37:63 38:60 39:69 40:73 41:67 42:71 43:56 44:58
1 1:71 2:62 3:70 4:64 5:67 6:64 7:79 8:65 9:70 10:69 11:72 12:71
13:68 14:65 15:61 16:61 17:73 18:71 19:75 20:74 21:80 22:74 23:54
24:47 25:53 26:37 27:77 28:68 29:72 30:59 31:72 32:68 33:60 34:60
35:73 36:70 37:66 38:65 39:64 40:55 41:61 42:41 43:51 44:46
1 1:69 2:71 3:70 4:78 5:61 6:63 7:67 8:65 9:59 10:59 11:66 12:69
13:71 14:75 15:65 16:58 17:60 18:55 19:62 20:59 21:67 22:66 23:74
24:74 25:64 26:60 27:57 28:54 29:70 30:73 31:69 32:76 33:62 34:64
35:61 36:61 37:66 38:65 39:72 40:73 41:68 42:68 43:59 44:63
// ....
```

- part of the test data

```
// part of the test dataset
0 1:64 2:70 3:71 4:69 5:72 6:70 7:75 8:78 9:61 10:66 11:69 12:68
13:68 14:70 15:71 16:70 17:75 18:76 19:73 20:72 21:80 22:78 23:79
24:81 25:74 26:70 27:72 28:79 29:73 30:75 31:77 32:73 33:65 34:64
35:72 36:72 37:59 38:62 39:71 40:74 41:68 42:67 43:58 44:57
0 1:76 2:75 3:68 4:78 5:71 6:72 7:72 8:75 9:61 10:65 11:67 12:70
13:67 14:75 15:60 16:58 17:63 18:67 19:59 20:63 21:67 22:72 23:74
24:73 25:56 26:56 27:52 28:52 29:67 30:68 31:73 32:78 33:65 34:68
35:61 36:67 37:69 38:74 39:77 40:75 41:74 42:70 43:63 44:61
0 1:74 2:73 3:72 4:75 5:63 6:62 7:67 8:67 9:73 10:74 11:75 12:79
13:70 14:71 15:64 16:67 17:65 18:69 19:79 20:78 21:81 22:80 23:71
24:73 25:60 26:62 27:69 28:67 29:69 30:69 31:75 32:75 33:66 34:67
35:67 36:66 37:71 38:73 39:66 40:69 41:62 42:65 43:55 44:56
0 1:65 2:67 3:69 4:76 5:62 6:68 7:65 8:66 9:65 10:64 11:74 12:73
13:60 14:75 15:66 16:63 17:64 18:62 19:73 20:65 21:77 22:74 23:69
24:69 25:66 26:59 27:68 28:59 29:69 30:69 31:76 32:79 33:65 34:63
35:60 36:60 37:69 38:64 39:69 40:74 41:69 42:70 43:62 44:57
```

I use the training and test data predefined in head of the files

which means that we have 299 training data and 50 test data

2. SVM model choose

there are many kind of SVM classifier of regresser such as linear or nonlinear can be choosen, and also have many kernel models such as polynomial, sigmoid, radial basis and so on

2.1 kernel type

According to the Guide, choosing the RBF (radial basis funtion) is a reasonable first choice, This kernel nonlinearly maps samples into a higher dimensional space so it, unlike the linear kernel, can handle the case when the relation between class labels and attributes is nonlinear.Therefore, in this assignment we use the **RBF kernel in this three problem and turn out both have a good accuracy**

2. 2 Classifier or Regresser

2.2.1 Two Spiral Problem

This Problem is a classification problems , and seening from the plot, we can find that it needs a non-linear decision boundary.

so as for this problem we choose the **Classification non-linear model with a RBF kernel**

2.2.1 Abalone Age Problem

This Problem is a regression problems

the Guide indecate that model can be choose as linear only if

- Number of instances ≪ number of features
- Both numbers of instances and features are large
- Number of instances ≫ number of features

is this dataset of problem, we have 8 features and 3677(not too large) instances, if we use the non-liner model, it will be time costed.

so as for this problem we choose the **Regression linear model with a RBF kernel**

2.2.2 SPECT Heart Diagnosis Problem

This Problem is a classification problems, we have 44 features and 299 instances, does not suit any of the condition above. Therefore we chonse the non-linear model

so as for this problem we choose the **Classification non-linear model with a RBF kernel**

3. Tune the parameter

For classification problem(1, 3), we need to find the best parameter C (cost), γ (gamma)

For Regression problem (2), we need to find the best parameters C(cost), γ (gamma) , ϵ(epsilon)

For best parameter search, we use the tools python grid.py , gridregression.py (which provided in the folder of tools in LIBSVM) corresponds to the classification and regression problem

- Two Spiral Problem

  this is a classfication problem, we use the **grid.py** to search the best C (cost), γ (gamma)

  ```
  $ python grid.py svm_SpiralData
  ```

- Abalone Age Problem

  this is a classfication problem, we use the **gridregression.py** to search the best C (cost), γ (gamma)

```
$python gridregression.py -log2c -10,10,1 -log2g -10,10,1 -log2p
-10,10,1 -v 10 -s 3 -t 0 svm_data2

# '-s 3' refers to this is a regression problem, '-t 0' refers to ues
liner kernel
```

- SPECT Heart Diagnosis Problem

  this is a classfication problem, we use the **grid.py** to search the best C (cost), γ (gamma)

```
$ python grid.py svm_data3
```

4. Achieved classification accurary

4.1 Two Spiral Problem

- Use the tool of grid.py to find the best parameters of this problem

```
$python3 grid.py ../dataset/svm_SpiralData
```

```
[local] 13 -3 53.6458 (best c=2048.0, g=8.0, rate=78.125)
2048.0 8.0 78.125
```

- use the best parameters to test train and test data

```
y, x = svm_read_problem('dataset/svm_SpiralData')
m = svm_train(y, x, '-c 2048 -g 8.0')
y_t, x_t = svm_read_problem('dataset/svm_SpiralData_t')
p_label, p_acc, p_val = svm_predict(y_t, x_t, m)
```

- we can achieve the accurary to **71.4286%**

```
optimization finished, #iter = 96720
nu = 0.191701
obj = -27202.966252, rho = -0.997852
nSV = 80, nBSV = 6
Total nSV = 80
Accuracy = 71.4286% (55/77) (classification)
```

4.2 Abalone Age Problem

- Use the tool of gridregression.py to find the best parameters of this problem

```
$python3 gridregression.py -log2c -1,2,1 -log2g 1,1,1 -t 0
../dataset/svm_data2
```

```
angzhouArkdeMacBook-Pro:tools ark$ python3 gridregression.py -log2c -1,2,1 -log2g 1,1,1 -t 0 ../dataset/svm_data2
local] 1.0 1.0 -4 0.0139996 (best c=2.0, g=2.0, p=0.0625, mse=0.0139996)
local] 1.0 1.0 -6 0.0143152 (best c=2.0, g=2.0, p=0.0625, mse=0.0139996)
```

- use the best parameters to test train and test data (note that we should the regression methon)

```
y, x = svm_read_problem('dataset/svm_data2')
m = svm_train(y, x, '-t 0 -s 3 -c 4.0 -g 2.0 -h 0')
y_t, x_t = svm_read_problem('dataset/svm_data2_t')
p_label, p_acc, p_val = svm_predict(y_t, x_t, m)
```

- we can achieve the Mean squared error to **0.01**

```
.....................................................*
optimization finished, #iter = 53347
nu = 0.301050
obj = -344.677512, rho = -0.214082
nSV = 1113, nBSV = 1099
Mean squared error = 0.0109937 (regression)
Squared correlation coefficient = 0.477704 (regression)
```

4.3 SPECT Heart Diagnosis Problem

- Use the tool of grid.py to find the best parameters of this problem

```
$ python3 grid.py ../dataset/svm_data3
```

```
[local] 13 -3 84.2809 (best c=512.0, g=3.0517578125e-05, rate=87.291)
512.0 3.0517578125e-05 87.291
```

- use the best parameters to test train and test data

```
y, x = svm_read_problem('dataset/svm_data3')
m = svm_train(y, x, '-c 512 -g 3.0517578125e-05')
y_t, x_t = svm_read_problem('dataset/svm_data3_t')
p_label, p_acc, p_val = svm_predict(y_t, x_t, m)
```

- we can achieve the accurary to **96%**

```
.......*....*
optimization finished, #iter = 3465
nu = 0.183367
obj = -18082.550896, rho = -11.211082
nSV = 104, nBSV = 19
Total nSV = 104
Accuracy = 96% (48/50) (classification)

Process finished with exit code 0
```

5. Discussion and analysis the SVM and compare with neural networks

5.1 Discussion and analysis of what i have observed and experience

As for the classification and regression problem, the SVM is very easy to implement by using the LIBSVM, and the kernel function is not too diffcult to choose, because there is only four common kernel function, and we just use the RBF which can handle the problem in non-linear way (more normal is the real world problem) is enough.

For a normal case, we just need to find out two parameter C (cost), γ (gamma) for classification and three parameters C(cost), γ (gamma) , ϵ(epsilon) for regression problems . and we can use the cross-validation to find out the best parameter which is alway the global optimum parameter. if the features is not too large such as the first problem (SpiralData has just two features) the cross-validation and grid search can be very quick. and with the best parameter, we can get a good prediction accuracy to the test data. However thinks might be litter more complicated when meet large features dataset such as second problem (Abalone Age Problem), the process of cross-validation and grid search will be very slow(up to several hours in the second problem). Therefor when have many features, there may be a need to choose a subset of them before giving the data to SVM

In a word, SVM can build a model very fast and predict with high accuracy when deal with a not larger features problem. when facing a larger features problem, we may need to choose a subset to SVM or switch to another algorithm.

5.2 compare with neural networks

Compare with the neural networks, the SVM have the obvious advantages and disadvantages.

As for the adventages,

- SVM has a more simple structure, SVM dose have very deep layer which means that SVM can have a strict theoretical and mathematical basis, based on the principle of structural risk minimization. On the contrary, the neural networks more like a block box, when it comes to deep neural networks, it is very difficult for people to infer why can this model produce such a result.
- SVM always can find out the global optimum parameter through the cross-validation. On the contrary, neural network are more likely to trap into a local minimums, even cannot find a gobal minimums on matter how hard we try
- neural network are more likly to meet overfitting which is not so difficult to be solved in SVM

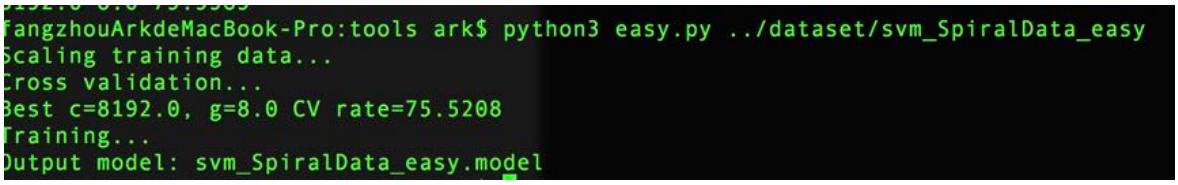
As for the disadvantage

- SVM can not implement large scale of data very well which can be better when use the neural network
- SVM still have lots of problem when face a Multi-classification problem

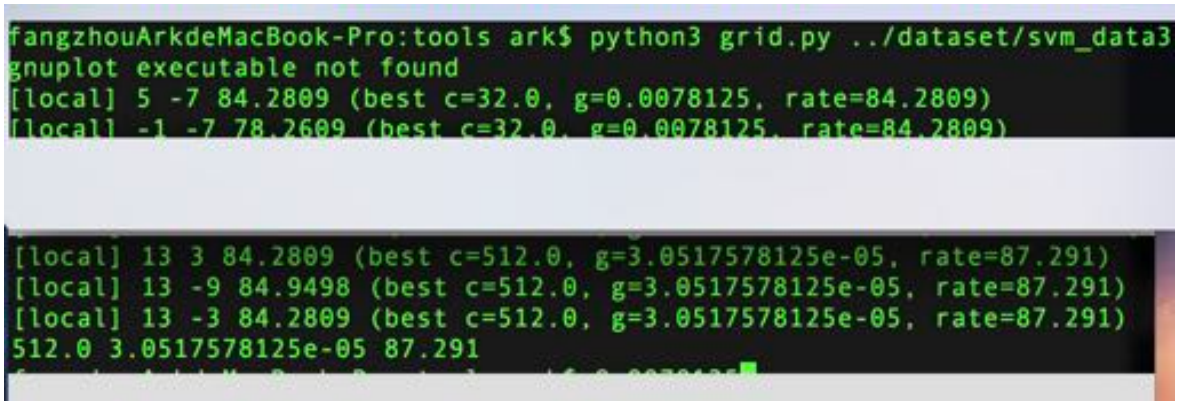
6 A printout of the command to implement the SVM

There are many ways to implement SVM which includes

- using the easy.py for commad

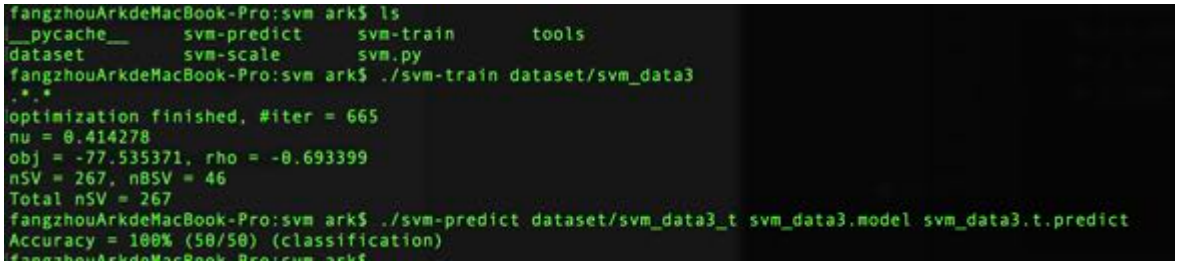


- we can just use the grid.py to find the best parameters and implement this parameters in our python file



```
#dataset 3
y, x = svm_read_problem('dataset/svm_data3')
m = svm_train(y, x, '-c 512 -g 3.0517578125e-05')
y_t, x_t = svm_read_problem('dataset/svm_data3_t')
p_label, p_acc, p_val = svm_predict(y_t, x_t, m)
```

- using the grid.py and svm-predict for command



Self Organizing Map (part 2)

1. Introduction to the MNIST data set

the MINST data set contain 5000 example which represents a 28 * 28 grayscale image. **each column** of the dataset represents a train data, which means that the dataset have 28 * 28 rows, and 5000 column. the number in this dataset are all normalized, which is a float type between 0 to 1. Therefore **we do not need to normalize the data**, and also those data can be plot as a a 28*28 gray image **.svg** just like below :

Therefore the MNIST dataset consist of 5000 handwriting (0-9) examples, which is stored as a 28 * 28 vector . we need to build a model of SOM to do the unsupervised learning to classify those differen (0-9) handwriting

2. Introduction to the steps of SOM training

2.1 training Steps

Kohonen SOM algorithm which used in our experiment is a improved case compared to the (Winner-Take-All) algorithm. It is not just the winner's weight needs to be update in Kohonen SOM algorithm but also the weight of the neighborhood of the winner need to be updated as well, Therefore, the step of our SOM training can be divided into the following step:

- **Step 1 : initialize the weight of the neuron:**

  We initialize the SOM with 100 neurons in the output layer which consists of 10 * 10 maps

  and the number of the neurons also can be modified,and the 100 is the number we use in our experiment

  The weight of each neuron should be the same as the input data's dimension which is 28 * 28, Therefore we should initialize each neuron into a 784 vector and the value in the vector can be initalized to a float between 0 to 1 which is the same as input data

  here is the code in python below

```
    # init the weight of each neuron to (0, 1)
    random.seed()
    for j in range(0, train_class):
        class_vector = []
        for i in range(0, data_dim):
            class_vector.append(round(random.random(), 5))
        neurons.append(class_vector)
```

- **Step 2 : input a sample to the SOM network**

  we read a sample in the data set and make it to a form a vector 784

  here is the code in python below:

```
def read_mnist():
    """
    read the data set and return a list
    Each example column of the file (not row)
    row => 784  (features)
    col => 5000 (example)
    :return:
    examples : the number of train data
    dim      : the dim of one data (features)
    """
    res = []
    with open('SOM_MNIST_data.txt', 'r') as f:
        for line in f.readlines():
            res.append(line.split(' '))
    examples = len(res[0])
    dim = len(res)
    return res, examples, dim
```

- **Step 3 : Compute the Euclidean distance between the sample with each neuron**

  The step can compute the distance between the sample with all the neuron and we pick up the neuron which has the smallest distance with the sample.

- **Step 4 : Choose the winner**

  According to the distance between the neurons with the sample, we choose a winner which has the smallest distance with the sample

  Here is the code in python of this two steps

```
    for each_epoch in range(0, train_times):
        print("now you are at epoch : " + str(each_epoch))
        # randomly order at beginning of each epoch
        for i in range(examples):
            random_num = random.randint(0, examples-1)
            images[i], images[random_num] = images[random_num],
images[i]

        for each_image in images:
            # get the winner
            winner_index = getWinner(each_image)

            # code ...
            # ...
```

```
def getWinner(data):
    """
    find the winner neuron in the neurons
    :param data:
    :return:
    """
    min_value = 100000  # save the min distance
    min_index = -1  # save the min neuron index
    for i in range(len(neurons)):
        dis = caldis(neurons[i], data)
        if dis < min_value:
            min_value = dis
            min_index = i
    return min_index
```

```
def caldis(vector_one, vector_two):
    """
    cal the Euclidean distance between two vector
    :param vector_one:
    :param vector_two:
    :return: distance
    """
    len1 = len(vector_one)
    len2 = len(vector_two)
    if len1 != len2:
        raise RuntimeError("need same dim with the vector")
    dis = 0
    for i in range(len1):
        dis += pow((vector_one[i] - vector_two[i]), 2)
    return math.sqrt(dis)
```

- **Step 5 : Update the weight of each neuron**

  In our experiment, the update strategy is that :

  In the beginning, we update the winner's weight and the neighborhood of the winner as well, **and the area we judge it as a neighborhood will become smaller and smaller as the iteration times raise up**

  In the later period, the area which judge as a neighbor will become just winner itself, which means that the winner will change the weight alone

  And the Lreaning rate is change as well as the iteration times rase up, it will be heigher in the beginning , and become smaller but not less then 0.01

  Here is the code in python below:

```
for i in range(len(neurons)):
    # only the nearly neuron will be updated
    dis = caldis(get_2d_vector(i), get_2d_vector(winner_index))
    if dis < sigma_t:
        for j in range(len(neurons[i])):
            effect = math.exp(-dis / (2 * sigma_t * sigma_t))
            neurons[i][j] += eta_t * effect * (each_image[j] -
neurons[i][j])
```

- **Step 6 : Repeat until the number of trainings is reached**

2.2 two phases

2.2.1 ordering phase

during which the topological ordering of the weight vectors takes place. typically this will take as many as 1000 iterations of the SOM algorithm, and careful consideration needs to be given to the choice of neighbourhood and learning rate parameters.

in my experiment, **I choose the the Learning rate to be 0.6 at the beginning**

2.2.2 Convergence phase

during which the feature map is fine tuned and comes to provide an accurate statistical quantification of the input space. Typically the number of iterations in this phase will be at least 500 times the number of neurons in the network, and again the parameters must be chosen carefully.

In a word, **We need a large learning rate and large neighborhood area in the beginning to leaning faster from the input dataset, However in the later period, we need small learning rate and small neighborhood area to adjust the neuron precisely**

Therefore we change our neighborhood function and learning rate every episode.

```
# eta_0 is the init leanring rate
eta_0 = 0.6
eta_t = eta_0

# sigma_t is the init neighborhood judger
sigma_0 = math.sqrt(pow(0.5*rowNum, 2) + pow(0.5*colNum, 2))
sigma_t = sigma_0
```

```
# Note that the learning rate shall remain above 0.01
# get smaller after each episode
eta_t = max(eta_0 * math.exp(-(each_epoch+1)/tau_2), 0.01)


# Attenuate the sigma value of h() for the next epoch;
# get smaller after each episode
sigma_t = sigma_0 * math.exp(-(each_epoch+1)/tau_1)
```

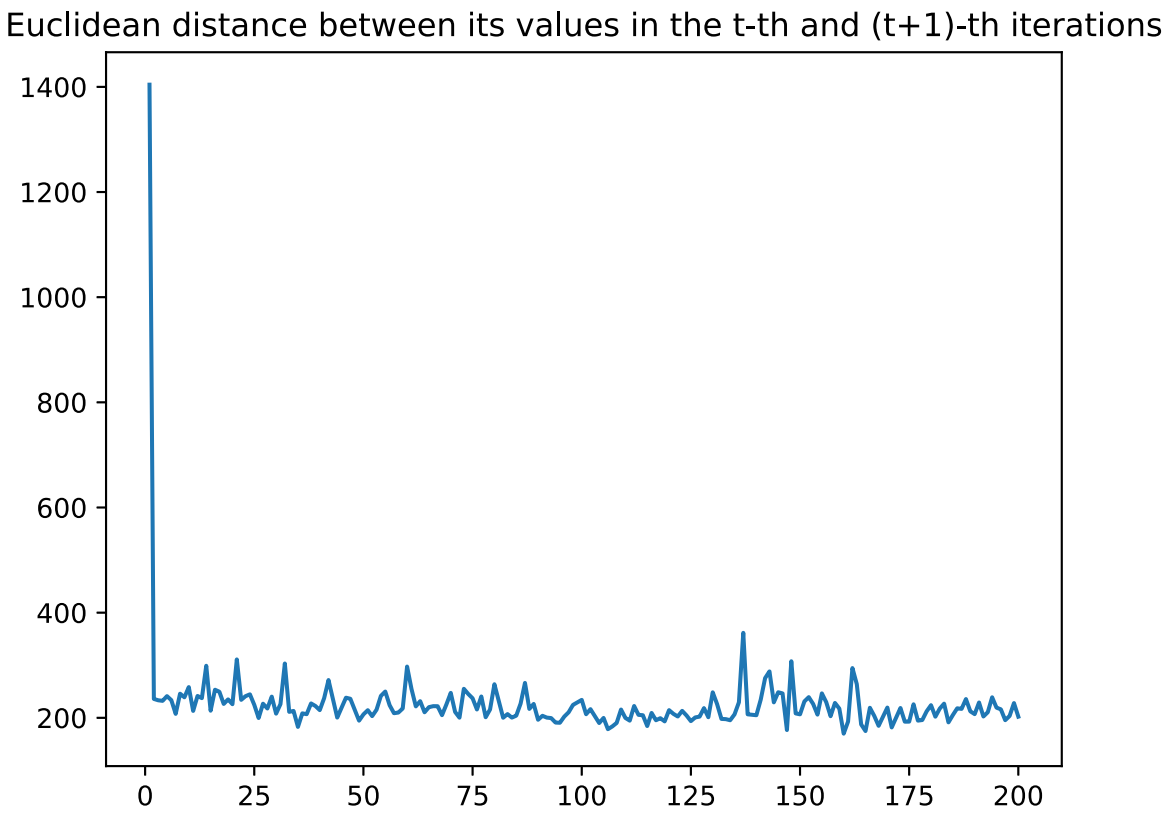
3. the weight change

The change of the weights between two consecutive epochs is indicative of the convergence of the training process.

we record the weight change after each episode, and can plot the the change of the weight to visualize the weight change

```
# differ the weight
dis = 0
for i in range(len(neurons)):
    dis += caldis(neurons[i], old_neurons[i])
    save_diff(dis)
old_neurons = copy.deepcopy(neurons)
```

Here is the weight change plot:

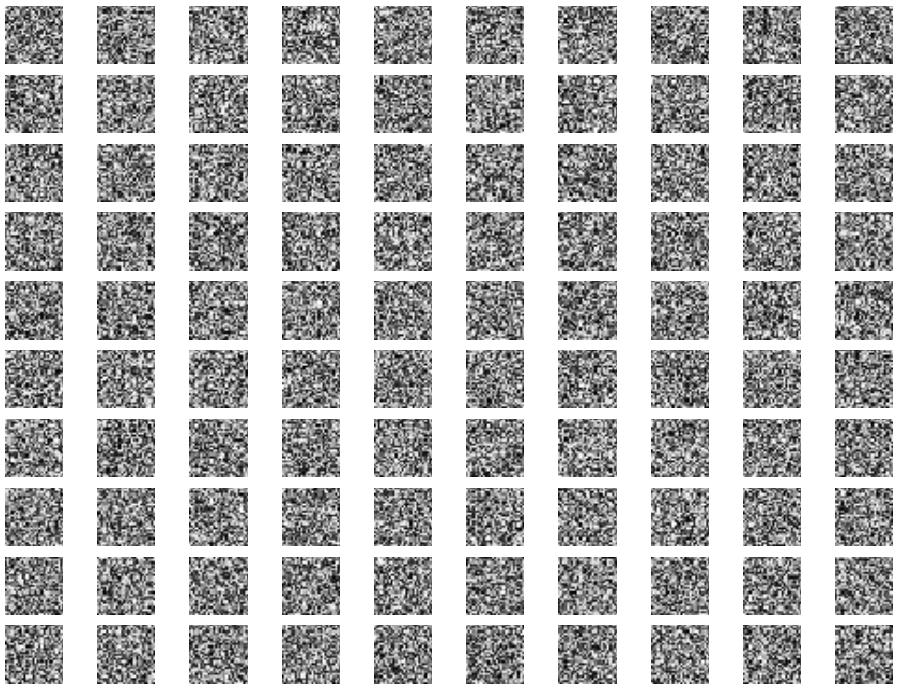


We can see from the plt that the model converge very fast, and become stable at the after 100 times iterations, However the error dynamic gose up at about 140th iteration. then gose stable slowly. it tells us that we can reduce the iteration times to around 130 times when it becomes stable to prevent the dynamic going up.

4. visualize the result

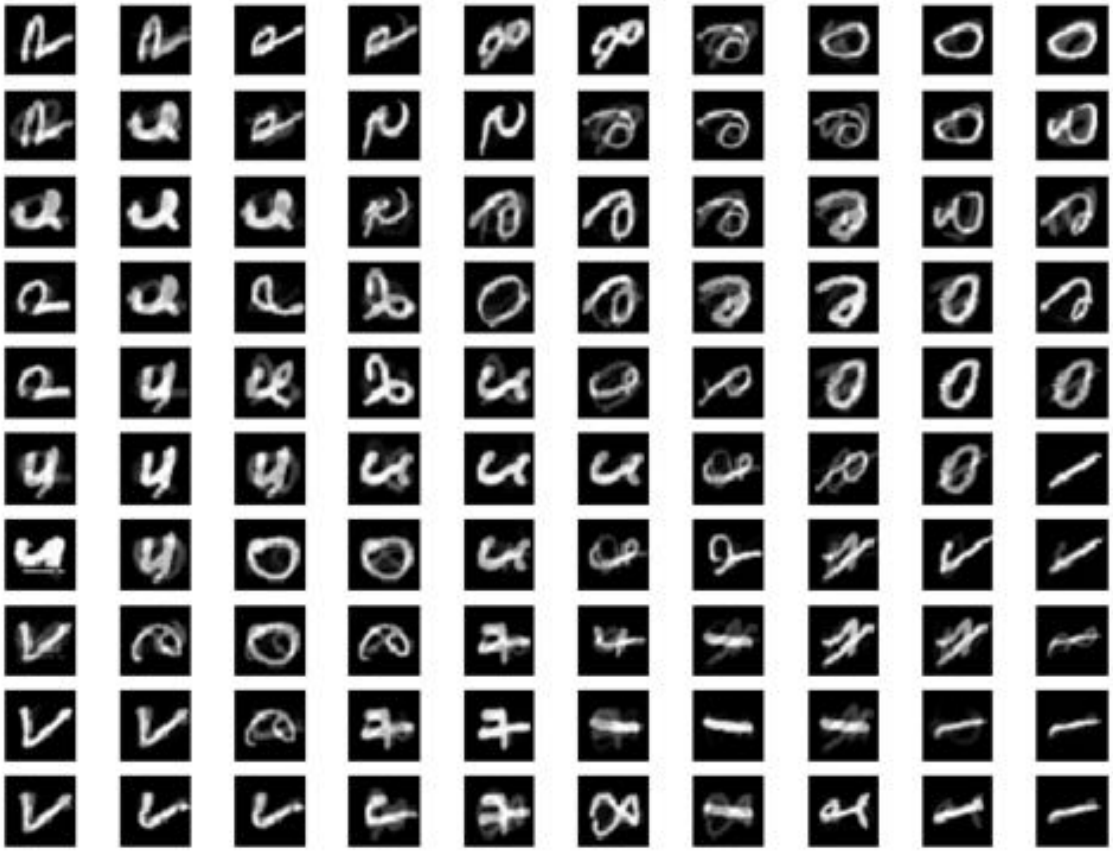
- the weight after initialize

init weight



- the middle weight

- the finnal weight:



5. Invertigare Various settings

the setting to be investigated inlcuding the following :

- the number of training examples
- the size of 2D lattice
- the learning rate
- the size of neighborhood
- number of epochs

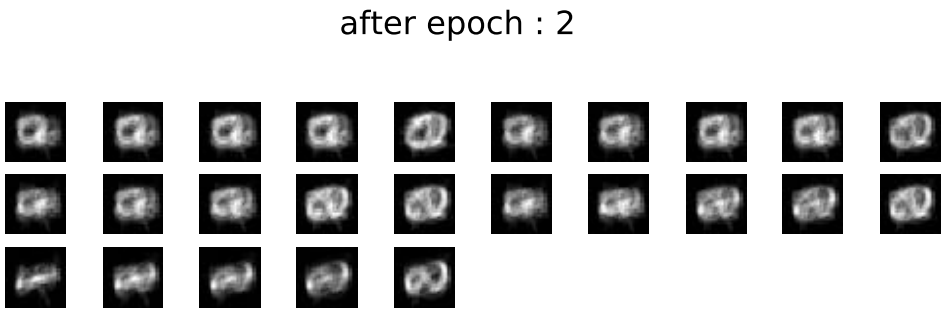
5.1 the number of training examples

the number of training examples in the dataset is 5000 in our experiment, The more examples of learning, the more examples the model can reach, the greater the accuracy of the corresponding unknown data.

5.2 the size of 2D lattice

The size of 2D lattice can be understood as the maximum number of classifications that the SOM model can represent. In this experiment, we need to identify numbers from 0-9, so the size is theoretically greater than 10, but we choose a grid of 10 * 10 The larger the grid selection, the better the similar classification aggregation degree, but the greater the computational pressure that comes with it

- a example to a 5 * 5 lattice



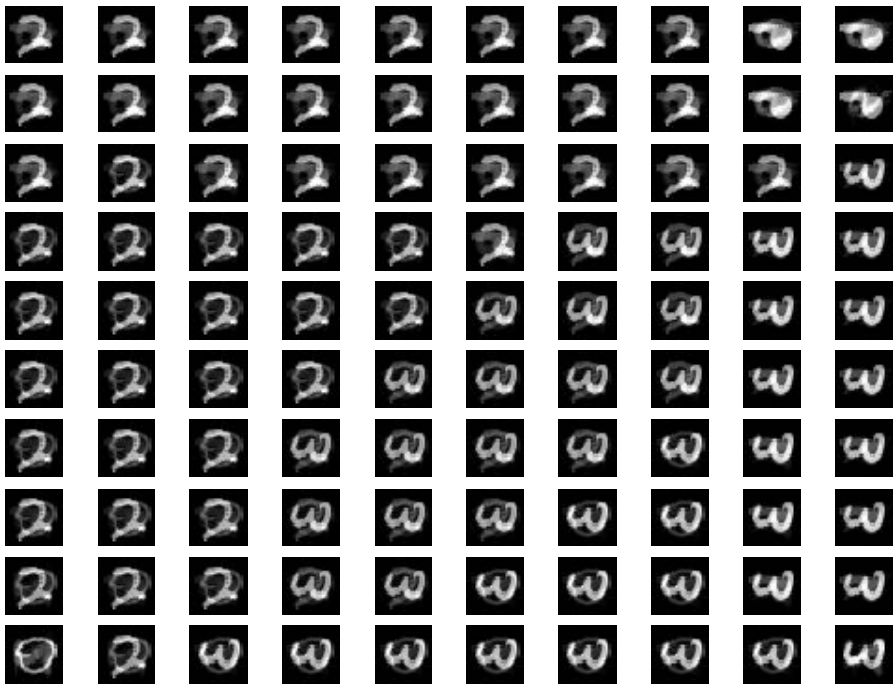
5. 3 the learning rate

The learning rate indicates the impact of the current input on the model. The learning rate is too large or too small. The learning rate is too sensitive to external input. Too small learning rate will cause the model to be too slow.

Here is some of the result with different learning rate

- the learning rate = 0.6

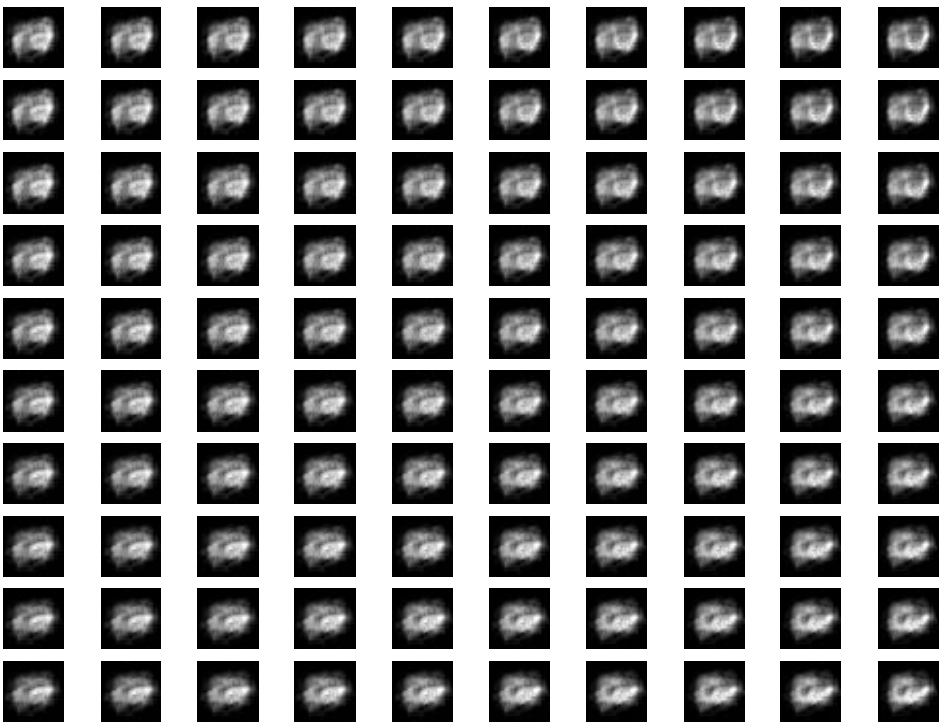
after epoch : 1



we can find that even at the beginning, the model can learn much from the input sample , however in this learning rate, the model is hard to converge

- the learning rate = 0.1

after epoch : 1

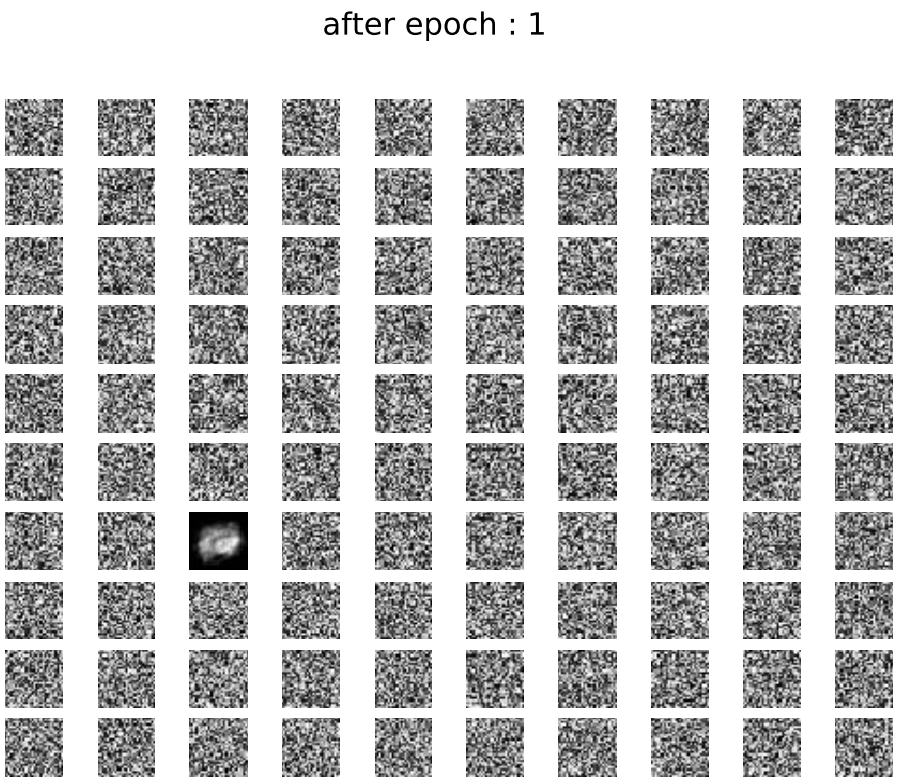


Under this learning rate, model will be learned slower, can the will be lower sensitive to the input sample

5.4 the size of neighborhood

The size of neighborhood represents the definition of neighbors around the winner neuron. These neighbors are also updated with Winner's weight, but relatively speaking, the weight away from Winner will be less updated. The best is at the beginning of the training. At the time, the neighbor function needs to be larger, so that each sample can be fully studied. In the later stage, the neighbor function needs to be smaller. These can accurately adjust these WInners.

- if we take very small neighborhood in the beginning, Only the winner will be updeate



Therefore we use the neighborhood, to get the best result:

```
# sigma_0 is initialised as the "radius" of the lattice;
sigma_0 = math.sqrt(pow(0.5*rowNum, 2) + pow(0.5*colNum, 2))
sigma_t = sigma_0

# get smaller after each episode
sigma_t = sigma_0 * math.exp(-(each_epoch+1)/tau_1)
```

5.5 number of epochs

Describes the number of iterations of the training process. This number is not as good as possible. In the case that the model can already be classified (the vector distances before and after have already converged), it is meaningless to increase the number of times.

6. Code

In this python code, i do not use any machine leaning library, and all the algorithm and be replanted to the C++ as well. However considering the hard of plot processes, i decide to use the python to finish my assignment

```
import random
import math
import matplotlib.pyplot as plt
import copy
# set number of 2d lattice
rowNum = 10
colNum = 10

# the number of training neuron
train_class = rowNum * colNum

# neuron list
neurons = []
old_neurons = []

#
image_x = 28
image_y = 28

# training times
epochNumOrd = 100
epochNumCov = 100
train_times = epochNumOrd + epochNumCov

# training parameters
# Initialise the sigma value of the neighbourhood function h();
# sigma_0 is initialised as the "radius" of the lattice;
sigma_0 = math.sqrt(pow(0.5*rowNum, 2) + pow(0.5*colNum, 2))
sigma_t = sigma_0

# change the tau_1
tau_1 = 1000/math.log(sigma_0)

# Initialise the learning rate and define the rate in each epoch t;
eta_0 = 0.6
eta_t = eta_0
# Define the attenuation speed of the learning rate with epoches;
```

```
tau_2 = 1000


def read_mnist():
    """
    read the data set and return a list
    Each example column of the file (not row)
    row => 784  (features)
    col => 5000 (example)
    :return:
    examples : the number of train data
    dim      : the dim of one data (features)
    """
    res = []
    with open('SOM_MNIST_data.txt', 'r') as f:
        for line in f.readlines():
            res.append(line.split(' '))
    examples = len(res[0])
    dim = len(res)
    return res, examples, dim


def caldis(vector_one, vector_two):
    """
    cal the Euclidean distance between two vector
    :param vector_one:
    :param vector_two:
    :return: distance
    """
    len1 = len(vector_one)
    len2 = len(vector_two)
    if len1 != len2:
        raise RuntimeError("need same dim with the vector")
    dis = 0
    for i in range(len1):
        dis += pow((vector_one[i] - vector_two[i]), 2)
    return math.sqrt(dis)


def getWinner(data):
    """
    find the winner neuron in the neurons
    :param data:
    :return:
    """
    min_value = 100000  # save the min distance
    min_index = -1  # save the min neuron index
    for i in range(len(neurons)):
        dis = caldis(neurons[i], data)
```

```
            if dis < min_value:
                min_value = dis
                min_index = i
        return min_index


    def get_2d_vector(neuron_index):
        """
        pass the index of the neuron
        return the 2d coordinates form
        [
        [0, 1, 2, ..., 9]
        [10,11,12 ... ,19]
        ...
        ]
        e.g.
        2 => [1, 3]
        11 => [1, 2]
        :param neuron_index:
        :return:
        """
        rowIndex = int(neuron_index / rowNum) + 1
        colIndex = (neuron_index % colNum) + 1
        return [rowIndex, colIndex]


    def plot_one(vector, save_name):
        """
        vector => 28 * 28
        :param vector:
        :return:
        """
        img = []
        for i in range(image_x):
            img.append(vector[i * image_x:(i + 1) * image_x])
        plt.imshow(img, cmap='gray')
        plt.axis('off')
        plt.savefig(save_name)


    def plot_neuron(title="", save_name="test.svg"):
        """
        plot all the neuron
        :return:
        """
        times = 1
        for neuron in neurons:
            img = []
            for i in range(image_x):
```

```
                img.append(neuron[i*image_x:(i+1)*image_x])
            plt.subplot(10, 10, times)
            plt.imshow(img, cmap='gray')
            plt.axis('off')
            times += 1
        plt.suptitle(title)
        # if use show, the process will be blocked
        # plt.show()
        plt.savefig(save_name)


def save_diff(differ):
    """
    record the differ by save into a file
    :param differ:
    :return:
    """
    file_name = "differ.txt"
    with open(file_name, 'a') as f:
        f.writelines(str(differ) + "\n")

if __name__ == '__main__':
    DataSet, examples, data_dim = read_mnist()
    # reshape
    # images is DataSet do the Transpose
    images = []
    for i in range(0, examples):
        image = []
        for j in range(0, data_dim):
            image.append(float(DataSet[j][i]))
        images.append(image)

    # init the weight of each neuron to (0, 1)
    random.seed()
    for j in range(0, train_class):
        class_vector = []
        for i in range(0, data_dim):
            class_vector.append(round(random.random(), 5))
        neurons.append(class_vector)
    # old_neurons init
    old_neurons = copy.deepcopy(neurons)
    # plot init weight
    plot_neuron("init weight", "init_weight.svg")
    # start training
    for each_epoch in range(0, train_times):
        print("now you are at epoch : " + str(each_epoch))
        # randomly order at beginning of each epoch
        for i in range(examples):
            random_num = random.randint(0, examples-1)
```

```
            images[i], images[random_num] = images[random_num], images[i]
        times = 0
        for each_image in images:
            # get the winner
            times += 1
            winner_index = getWinner(each_image)
            # get the distance of the winner_index to all the neuron
            # update the weight
            for i in range(len(neurons)):
                # only the nearly neuron will be updated
                dis = caldis(get_2d_vector(i), get_2d_vector(winner_index))
                if dis < sigma_t:
                    for j in range(len(neurons[i])):
                        effect = math.exp(-dis / (2 * sigma_t * sigma_t))
                        neurons[i][j] += eta_t * effect * (each_image[j] -
neurons[i][j])

        # differ the weight
        dis = 0
        for i in range(len(neurons)):
            dis += caldis(neurons[i], old_neurons[i])
        save_diff(dis)
        old_neurons = copy.deepcopy(neurons)

        # after each epoch update the parameters
        # Attenuate the learning rate for the next epoch
        # Note that the learning rate shall remain above 0.01
        eta_t = max(eta_0 * math.exp(-(each_epoch+1)/tau_2), 0.01)

        # Attenuate the sigma value of h() for the next epoch;
        sigma_t = sigma_0 * math.exp(-(each_epoch+1)/tau_1)

        # plot
        plot_neuron("after epoch : " + str(each_epoch + 1),
"epoch_"+str(each_epoch+1)+".svg")
```