# Assignment 6 – CS547

Jun Peng

*Nov. 2nd, 2014*

## 1 Introduction

In this assignment, it is asked to implement a RBF neural network with one hidden layer neurons. Two different learning algorithms to determine the center of hidden neurons are studied, one is **Fixed Center Selected at Random**, the other is **Self-Organized Selection of Centers**. For each algorithm, five topologies of different hidden neurons are studied. The five different number of hidden neurons are 2, 5, 10, 20, 50 respectively. The order of presenting data to neural network is also studied in each topology. Root of mean square error is recorded for each training. The **stopping criterion** for training is the error for latest 20 epochs are stable (relative error less than $10^{-5}$). Learning rate is 0.01 throughout the whole report.

**Figure 1** shows the distribution of our training data and testing data in scatter plot. There are four classes shown in each figure. We can see they are not linearly separable. There is no clear boundaries for these classes since lots of data are overlapped.

## 2 Fixed Center Selected at Random

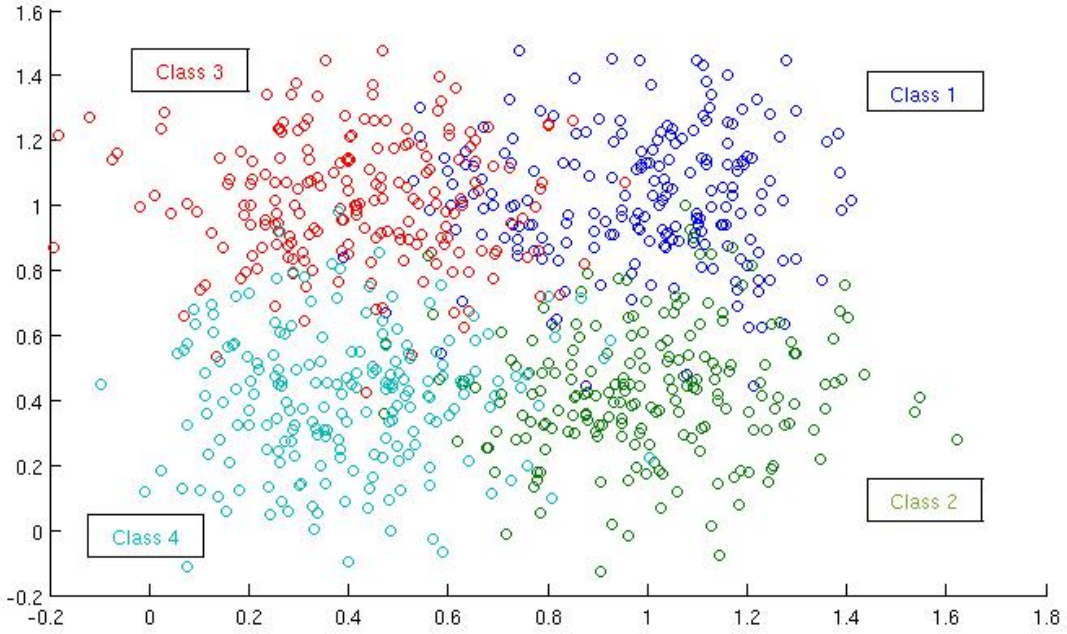
In this learning algorithm, the locations of the centers are chosen randomly from the training data set. Once the location is determined, it would not change in the following training. Then we present data to the neural network, apply Gaussian function to the hidden layer, use LMS to the output layer, updata the weights in the way we do in LMS assignment. See **table 1** for the experimental result. We can see as the number of hidden neurons increases, the convergence speed would decrease. And for each topology, we find that the testing data error doesn't converge with training data. The gap between training data and testing data are evident. The root of mean square error (RMSE) for testing data is roughly double of the training data. Comparing the results to MLP/BP, RBF neural network has a faster convergence speed but a higher RMSE. In MLP/BP, RMSE for testing data would usually converge with training data under no shuffle.

| hidden neurons | epochs | stopping RMSE for training data | stopping RMSE for testing data |
|---|---|---|---|
| 2 | 49 | 0.5063 | 1.1829 |
| 5 | 138 | 0.3753 | 0.7385 |
| 10 | 701 | 0.2985 | 0.8500 |
| 20 | 959 | 0.3578 | 0.7176 |
| 50 | 4528 | 0.3422 | 0.7184 |

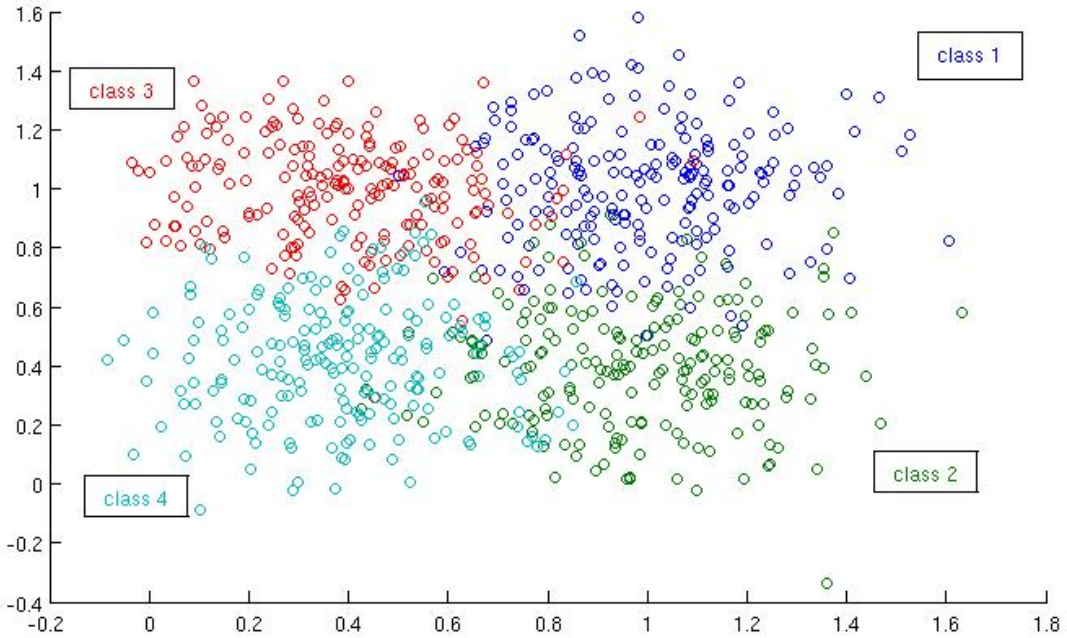
Table 1: Fixed Center - Summary for no shuffle

I take hidden neurons 5 and 50 as representatives of the generalization plot. From **Figure 2**, we can clearly see a gap between the training data and testing data.

We use shuffle to randomize the order of presenting the data to neural network. There are two shuffle strategies. **Shuffle 1** is shuffling the data before training and shuffling once. **Shuffle 2** is shuffling the data every epoch before presenting them to neural network. In general, **Shuffle 1** and **Shuffle 2** have a faster

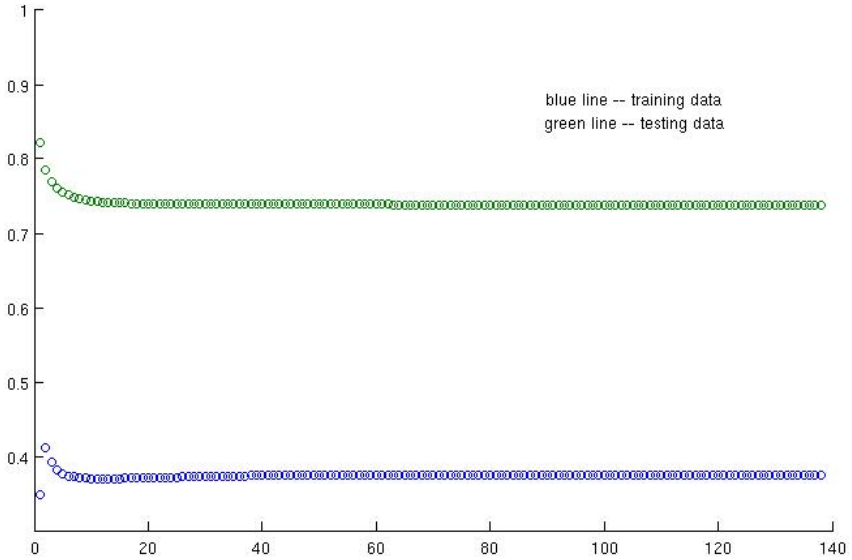



(a) Training data

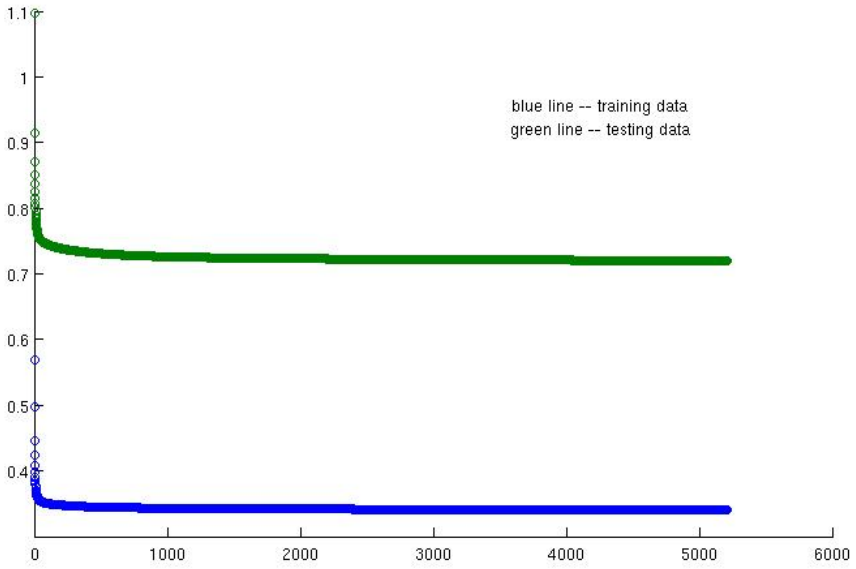



(b) Testing data

Figure 1: Scatter plot for data




(a) 5 hidden neurons




(b) 50 hidden neurons

Figure 2: Generalization graph

convergence speed than no shuffle. The RMSE for training data and testing data are very close to each other. **Shuffle 2** is faster than **Shuffle 1**. In general, the more hidden neurons, the lower RMSE we can get. See **table 2** and **table 3** for the experimental result.

| hidden neurons | epochs | stopping RMSE for training data | stopping RMSE for testing data |
|---|---|---|---|
| 2 | 56 | 0.7835 | 0.7927 |
| 5 | 279 | 0.5712 | 0.5450 |
| 10 | 686 | 0.4078 | 0.3911 |
| 20 | 830 | 0.4052 | 0.4056 |
| 50 | 2520 | 0.3903 | 0.4090 |

Table 2: Fixed Center - Summary for shuffle 1

| hidden neurons | epochs | stopping RMSE for training data | stopping RMSE for testing data |
|---|---|---|---|
| 2 | 48 | 0.7209 | 0.7082 |
| 5 | 189 | 0.4248 | 0.3972 |
| 10 | 113 | 0.4225 | 0.4210 |
| 20 | 362 | 0.4122 | 0.4082 |
| 50 | 279 | 0.4085 | 0.4211 |

Table 3: Fixed Center - Summary for shuffle 2

From the generalization plot (**Figure 3**) for **Shuffle 2** on 20 hidden neurons, we can see The difference between the training data and testing data are small.

# 3 Self-Organized Selection of Centers

In this algorithm, the locations of all hidden neurons are initialized randomly. Then we use *k-mean clustering algorithm*. Draw a sample from training data set and then determine which hidden neuron is closest to the sample, move the location of that hidden neuron closer to sample. After enough training on the locations of hidden neurons, we use them to training the weights to the output layer. The **stopping criterion** for not changing location any more is that the location is not moving any more relatively. I record the minimum distance for every presentation and sum it for one epoch. If the sum of minimum distances is stable between epochs, it can be stopped. After the locations of hidden neurons are determined, we present data to the neural network, apply Gaussian function to the hidden layer, use LMS to the output layer, updata the weights in the way we do in LMS assignment. See **table 4** for the experimental result. We can see as the number of hidden neurons increases, the convergence speed would decrease in general. Interesting thing occurs to the topology with 10 hidden neurons. It converges very quickly, even faster than the topology with 2 hidden neurons. And for each topology, we find that the testing data error doesn't converge with training data. The gap between training data and testing data are evident. The RMSE for testing data is roughly double of the training data. Comparing the results to MLP/BP, RBF neural network has a faster convergence speed but a higher RMSE. In MLP/BP, RMSE for testing data would usually converge with training data under no shuffle.

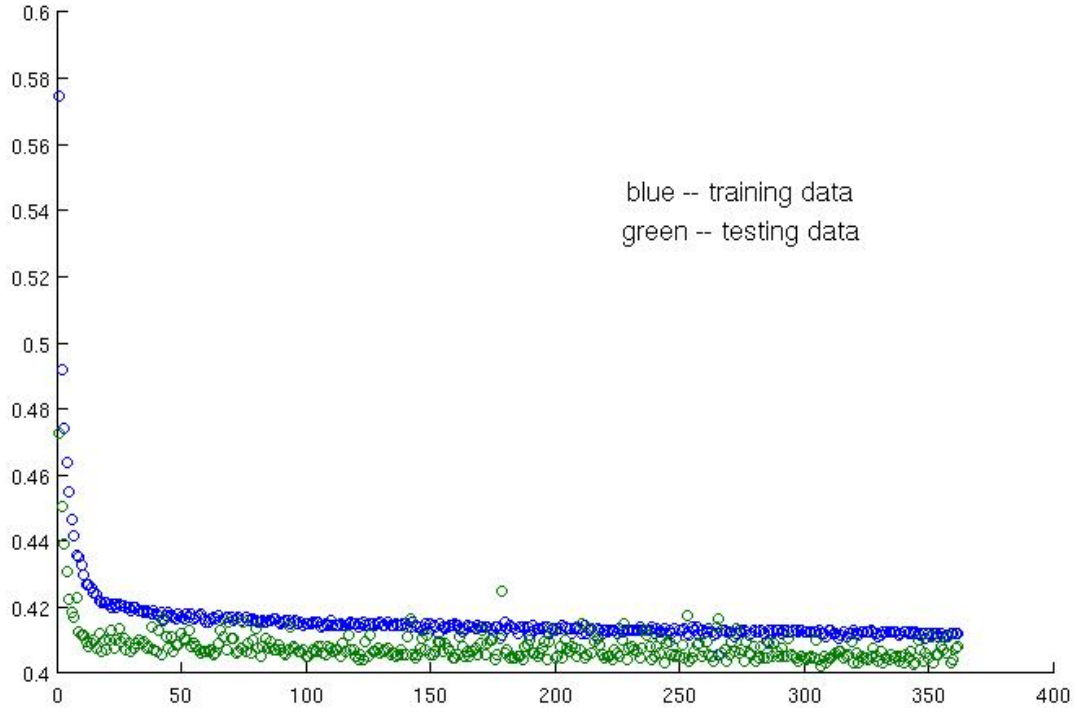


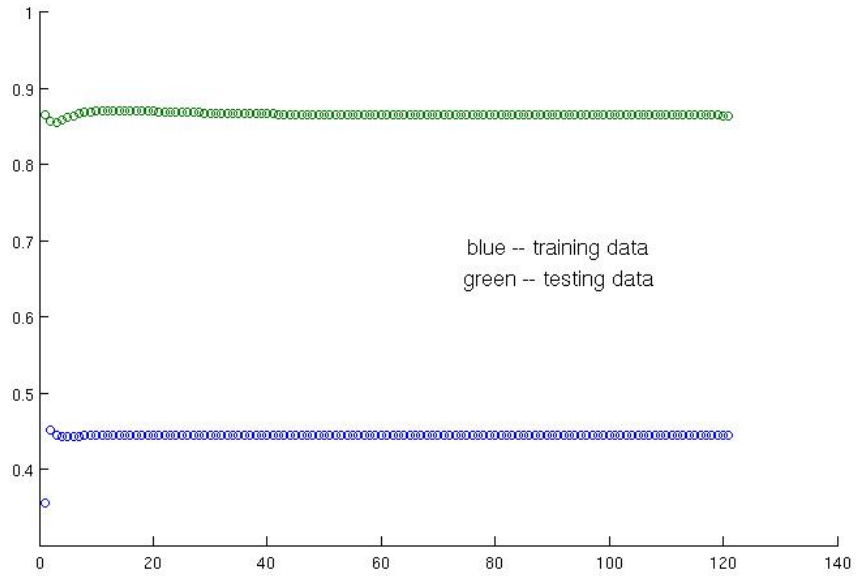

Figure 3: Generalization plot for Shuffle 2 on 20 hidden neurons

| hidden neurons | epochs | stopping RMSE for training data | stopping RMSE for testing data |
|---|---|---|---|
| 2 | 69 | 0.5835 | 1.2289 |
| 5 | 121 | 0.4441 | 0.8648 |
| 10 | 44 | 0.4196 | 0.8026 |
| 20 | 460 | 0.3746 | 0.7427 |
| 50 | 3061 | 0.4017 | 0.7082 |

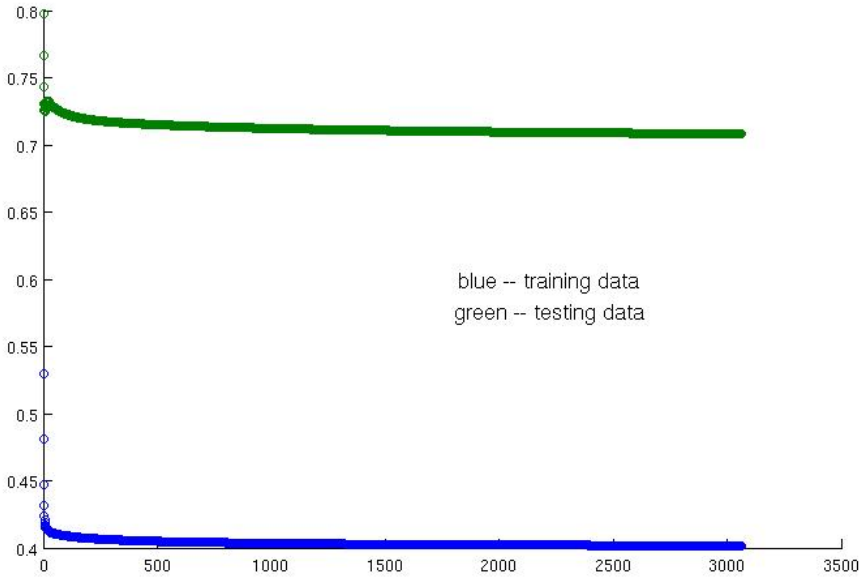
Table 4: K-Mean Clusters - Summary for no shuffle

I take hidden neurons 5 and 50 as representatives of the generalization plot (**Figure 4**). From **Figure 4**, we can clearly see a gap between the training data and testing data.

We use shuffle to randomize the order of presenting the data to neural network. There are two shuffle strategies. **Shuffle 1** is shuffling the data before training and shuffling once. **Shuffle 2** is shuffling the data every epoch before presenting them to neural network. From the experimental result (**table 5** and **table 6**), **Shuffle 1** has a slower convergence speed than no shuffle while **Shuffle 2** has a faster convergence speed. The RMSE for training data and testing data are very close to each other in **Shuffle 1** and **Shuffle 2**. However, from the training data perspective, **Shuffle 1** and **Shuffle 2** are no better than no shuffle.




(a) 5 hidden neurons




(b) 50 hidden neurons

Figure 4: Generalization graph

| hidden neurons | epochs | stopping RMSE for training data | stopping RMSE for testing data |
|---|---|---|---|
| 2 | 33 | 0.5325 | 0.5062 |
| 5 | 544 | 0.5310 | 0.5191 |
| 10 | 499 | 0.4777 | 0.4533 |
| 20 | 2163 | 0.4822 | 0.4840 |
| 50 | 5394 | 0.4450 | 0.4597 |

Table 5: K-Mean Clusters - Summary for shuffle 1

| hidden neurons | epochs | stopping RMSE for training data | stopping RMSE for testing data |
|---|---|---|---|
| 2 | 72 | 0.6822 | 0.6853 |
| 5 | 79 | 0.4761 | 0.4468 |
| 10 | 247 | 0.5015 | 0.5041 |
| 20 | 134 | 0.5417 | 0.5328 |
| 50 | 156 | 0.4658 | 0.4583 |

Table 6: K-Mean Clusters - Summary for shuffle 2

From the generalization plot (**Figure 5**) for **Shuffle 2** on 50 hidden neurons, we can see The difference between the training data and testing data are small.

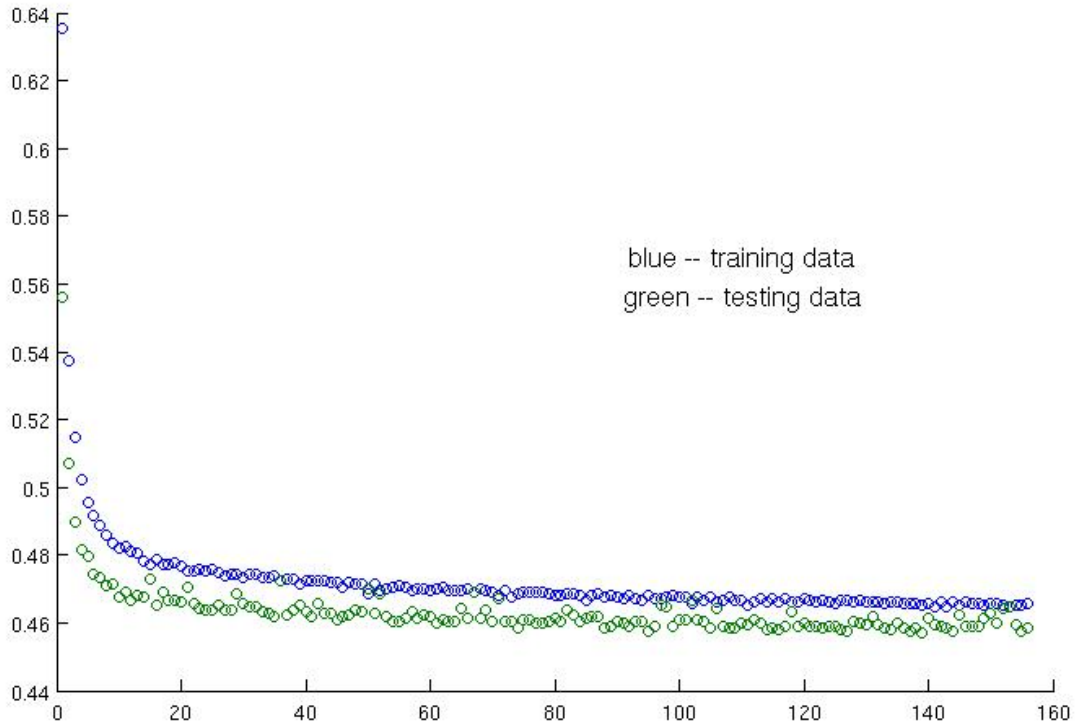



Figure 5: Generalization plot for Shuffle 2 on 50 hidden neurons

# 4 Conclusion

Below are some conclusions I can conclude from my experimental result:
(1) In general, the more hidden neurons, the slower convergence speed, the lower RMSE.
(2) The RMSE for testing data doesn't converge with training data in no shuffle.
(3) In general, shuffling the order of presenting the data to neural network would result in similar RMSEs for training data set and testing data set.
(4) Shuffle doesn't guarantee a faster convergence speed. In particular, **Shuffle 1** algorithm may result in a slower convergence speed. **Shuffle 2** always give us a faster convergence speed than **Shuffle 1** and no shuffle.
(5) Comparing the results to MLP/BP, RBF neural network has a faster convergence speed but a higher RMSE. In MLP/BP, RMSE for testing data would usually converge with training data under no shuffle while RBF not.

# Appendix

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define DATASIZE 800  //size of training data and test data
#define ETA 0.01   //learning rate
#define MAX_EPOCHS 100000

typedef struct ENTRY{   //dataset for each sample data
  double x1;
  double x2;
  double desired_value;
} ENTRY;

typedef struct POINT{
  double x;
  double y;
} POINT;

typedef struct NEURON{
  double output;
  double weight;
  POINT center;
} NEURON;


double gaussian(double distance, double sigma)
{
  return exp(- distance*distance / (2*sigma*sigma));
}

double randomWeight()  //random weight generator between 0 ~ 1
{
  return ((int)rand()\%100000)/(float) 100000;
}

double relativeError(double *error, int len)
{
  len = len - 1;
  if(len < 20)
    return 1;
  //keep track of the last 20 Root of Mean Square Errors
  int start1 = len-20;
  int start2 = len-10;

  double error1 = 0;
  double error2 = 0;
```

```
  int i;

  //calculate the average of the first 10 errors
  for(i = start1; i < start1 + 10; i++)
    {
      error1 += error[i];
    }
  double averageError1 = error1 / 10;

  //calculate the average of the second 10 errors
  for(i = start2; i < start2 + 10; i++)
    {
      error2 += error[i];
    }
  double averageError2 = error2 / 10;

  double relativeErr = (averageError1 - averageError2)/averageError1;
  return (relativeErr > 0) ? relativeErr : -relativeErr;
}

/*****************create hidden neurons************************************/
void createHiddenNeuronsForFixedCenter(NEURON *neurons, int number, ENTRY *data)
{
  int i;
  srand(time(NULL));

  //Bias neuron for output
  neurons[0].output = 1;
  neurons[0].weight= randomWeight();

  for(i = 1; i < number; i++)
    {
      neurons[i].weight= randomWeight();
      //randomly choose one data set as center position
      int index = (int) (randomWeight() * 100000) % DATASIZE;
      neurons[i].center.x = data[index].x1;
      neurons[i].center.y = data[index].x2;
    }
}

void createHiddenNeuronsForKMeanClusters(NEURON *neurons, int number)
{
  int i;
  srand(time(NULL));

  neurons[0].output = 1;
  neurons[0].weight= randomWeight();

  for(i = 1; i < number; i++)
    {
      neurons[i].weight= randomWeight();
```

```
      //randomly initialize the center
      neurons[i].center.x = randomWeight();
      neurons[i].center.y = randomWeight();
    }
}

double gaussianOutput(POINT point, ENTRY data, double sigma)
{
  double d = sqrt(pow(point.x - data.x1, 2) + pow(point.y - data.x2, 2));
  return gaussian(d, sigma);
}


double weightsAdjust(NEURON *neurons, int number, double sigma, ENTRY *data)
{
  int i, j;
  double errorSquareSum = 0;
  for(j = 0; j < DATASIZE;  j++)
    {
      for(i = 1; i < number; i++ )    //calculate the gaussian output
neurons[i].output = gaussianOutput(neurons[i].center, data[j], sigma);

      double y = 0;
      for(i = 0; i < number; i++)    //calculate  the output
{
  y += neurons[i].output * neurons[i].weight;
}

      double error = data[j].desired_value - y;
      errorSquareSum += error * error / 2;

      for(i = 0; i < number; i++)    //adjust the weight
{
  neurons[i].weight += ETA * error * neurons[i].output;
}
    }

  return sqrt(errorSquareSum/DATASIZE);
}


double fixedCenter(NEURON *neurons, int number)
{
  int i, j;
  double dmax = 0;
  for(i = 1; i < number; i++)
    {
      for(j = i + 1; j < number; j++)
{
  double d = sqrt(pow(neurons[i].center.x - neurons[j].center.x, 2)
    + pow(neurons[i].center.y - neurons[j].center.y, 2));
```

```
  if(d > dmax)
    dmax = d;
}
    }

  double sigma = dmax / sqrt(2*(number-1));
  return sigma;
}

double kMeanClusters(NEURON *neurons, int number, ENTRY *data)
{
  int i, j;

  double *error = (double *)malloc(MAX_EPOCHS * sizeof(double));
  int maxlen = 0;
  int epoch = 1;
  do{
    for(i = 0; i < DATASIZE; i++)
      {
int index;
double dmin = 0;
for(j = 1; j < number; j++)  // determine the nearest center
  {
    double d = pow(data[i].x1 - neurons[j].center.x , 2)
        + (data[i].x2 - neurons[j].center.y , 2);
    if(d < dmin)
      {
dmin = d;
index = j;
      }
  }
//move center
neurons[j].center.x += ETA * (data[j].x1 - neurons[j].center.x);
neurons[j].center.y += ETA * (data[j].x2 - neurons[j].center.y);
error[maxlen] += sqrt(dmin);
      }
    maxlen++;
    epoch++;
  }while(epoch < MAX_EPOCHS && relativeError(error, maxlen) > 1e-5);

  double dmax = 0;
  for(i = 1; i < number; i++)
    {
      for(j = i + 1; j < number; j++)
{
  double d = sqrt(pow(neurons[i].center.x - neurons[j].center.x, 2)
    + pow(neurons[i].center.y - neurons[j].center.y, 2));
  if(d > dmax)
    dmax = d;
}
    }
```

```
  double sigma = dmax / sqrt(2*(number-1));
  return sigma;
}


//*****************************testing*************************//
double test(NEURON *neurons, int number, double sigma, ENTRY *data)
{
  int i, j;
  double errorSquareSum = 0;
  for(j = 0; j < DATASIZE;  j++)
    {
      for(i = 1; i < number; i++ )    //calculate the gaussian output
neurons[i].output = gaussianOutput(neurons[i].center, data[j], sigma);

      double y = 0;
      for(i = 0; i < number; i++)    //calculate the output
{
  y += neurons[i].output * neurons[i].weight;
}

      double error = data[j].desired_value - y;
      errorSquareSum += error * error / 2;
    }

  return sqrt(errorSquareSum/DATASIZE);
}

//*******************read training data and test data**********************//

void getTrainingAndTestData(int argc, char **path, ENTRY *training, ENTRY *testing)
{
  if(argc != 3)
    {
      printf("Usage: program training_data_file testing_data_file\n");
      exit(0);
    }

  FILE *fp1, *fp2;
  if((fp1 = fopen(path[1], "r")) == NULL)
    {
      printf("cannot open %s\n", path[1]);
      exit(1);
    }
  if((fp2 = fopen(path[2], "r")) == NULL)
    {
      printf("cannot open %s\n", path[2]);
      exit(1);
    }
```

```
  int i = 0;
  double num;
  while(i < 800)
   {
     fscanf(fp1, "%lf %lf %lf %lf", &training[i].desired_value, &num,
        &training[i].x1, &training[i].x2);
     fscanf(fp2, "%lf %lf %lf %lf", &testing[i].desired_value, &num,
        &testing[i].x1, &testing[i].x2);
     i++;
   }
  fclose(fp1);
  fclose(fp2);
}


//********shuffle the order of presentation to neuron*********//
void swap(ENTRY *data, int i, int j)
{
  ENTRY temp;
  temp.x1 = data[i].x1;
  temp.x2 = data[i].x2;
  temp.desired_value = data[i].desired_value;
  data[i].x1 = data[j].x1;
  data[i].x2 = data[j].x2;
  data[i].desired_value = data[j].desired_value;
  data[j].x1 = temp.x1;
  data[j].x2 = temp.x2;
  data[j].desired_value = temp.desired_value;
}

void shuffle(ENTRY *data, int size)
{
  srand(time(NULL));
  int i;
  for(i = 0; i < size; i++)
    {
      int j = (int)rand()%size;
      swap(data, i, j);
    }
}



//**************************main function***********************//

int main(int argc, char** argv)
{
  int numberOfHiddenNeurons = 2;
  ENTRY *training_data = (ENTRY *)malloc(DATASIZE*sizeof(ENTRY));
  ENTRY *testing_data = (ENTRY *)malloc(DATASIZE*sizeof(ENTRY));
```

```
  //read training data and testing data from file
  getTrainingAndTestData(argc, argv, training_data, testing_data);

  int epoch = 1;

  //output data to a file
  FILE *fout;
  if((fout = fopen("kMean_2.txt", "w")) == NULL)
    {
      fprintf(stderr, "file open failed.\n");
      exit(1);
    }

  //shuffle the order of presenting training data to neural network
  //shuffle(training_data, DATASIZE);

  //train and test neural network
  double *error = (double *)malloc(MAX_EPOCHS * sizeof(double));
  int maxlen = 0;
  //create neural network for backpropagation
  NEURON *hiddenNeurons = (NEURON *)malloc(
    (numberOfHiddenNeurons+1) * sizeof(NEURON));
  //createHiddenNeuronsForFixedCenter(hiddenNeurons,
  //                numberOfHiddenNeurons+1, training_data);
  createHiddenNeuronsForKMeanClusters(hiddenNeurons,
                numberOfHiddenNeurons+1);
  //double sigma = fixedCenter(hiddenNeurons, numberOfHiddenNeurons+1);
  double sigma =  kMeanClusters(hiddenNeurons,
                numberOfHiddenNeurons+1, training_data);
  do{
    //shuffle(training_data, DATASIZE);
    error[maxlen] = weightsAdjust(hiddenNeurons,
                numberOfHiddenNeurons+1, sigma, training_data);
    double testError = test(hiddenNeurons,
                numberOfHiddenNeurons+1, sigma, testing_data);
    //printf("%d %lf %lf\n", epoch, error[maxlen], testError);
    fprintf(fout, "%d %lf %lf\n", epoch, error[maxlen], testError);
    epoch++;
    maxlen++;
  }while(epoch < MAX_EPOCHS && relativeError(error, maxlen) > 1e-5 );

  fclose(fout);
  free(hiddenNeurons);
  free(training_data);
  free(testing_data);
}
```

---