CS 6200 3/13/18

Che Shian Hung

Programming Assignment 2

Backpropagation Report

The program implements backpropagation algorithm to train the network to classify three classes. Each sample is represented as a three-dimensional vector, and there are 30 samples that are randomly generated for each class. The data set is different every time the program runs. In this way, we can fully test the trained network with various dataset.

Among 30 samples for each class, the first half of samples are used for training, and the second half of samples is used for testing the trained network. There are three hard coded sphere centers that are used to generate the data: (0, 15, 0), (7.5, 5, 0), and (-7.5, 5, 0). The data will be randomly generated based on each sphere center with radius 2. There are many global variables listed at the top of the program, which we can just modify the value and test the impact on the result.

Overall, the testing focus on the power of backpropagation network on learning various dataset instead of one data set. In the report, we will train and test the network with 30 different datasets and measure the average of training iteration, the average of accuracy, and the average time performance. While testing, we will also adjust the configuration such as initial weights and number of hidden layer to observe the impact.

First, let’s run one dataset on linearly separable and non-separable problem first. The network is built with 2 hidden layers with 5 nodes for each layer. The number of nodes in input layer and output layer are always 3:

**Linearly separable**

display all samples:

display class 1 samples:

0.219, 16.440, -1.371

-1.902, 15.558, 0.265

-1.395, 16.417, -0.211

-0.710, 13.399, 0.965

0.076, 13.003, 0.086

1.231, 13.867, 1.095

-1.625, 13.930, -0.463

0.420, 16.955, -0.054

-0.028, 16.999, -0.049

0.125, 13.523, 1.343

0.023, 16.999, 0.043

-1.958, 14.959, -0.406

-0.202, 16.958, 0.352

0.458, 16.924, 0.297

-0.021, 17.000, 0.028

0.505, 16.743, -0.841

0.007, 16.950, 0.443

0.910, 13.708, -1.226

-0.409, 13.930, 1.640

-1.622, 14.007, -0.620

-0.507, 14.939, 1.934

0.676, 16.525, -1.103

-1.633, 14.046, 0.650

1.392, 15.179, 1.425

-0.243, 13.019, -0.122

1.507, 16.310, -0.110

0.371, 16.965, 0.052

0.608, 13.430, 1.080

-0.690, 16.853, -0.302

-0.214, 13.012, 0.014

---------------------------------

display class 2 samples:

8.890, 6.430, -0.154

8.002, 3.072, -0.171

8.564, 6.576, 0.619

6.598, 5.379, -1.744

7.776, 3.336, -1.075

6.850, 6.343, -1.331

7.280, 3.453, -1.248

6.507, 4.092, -1.479

5.885, 6.179, -0.006

8.889, 3.852, -0.867

7.078, 4.993, -1.955

5.570, 5.484, 0.199

7.468, 7.000, 0.018

8.622, 5.163, 1.648

6.076, 6.348, 0.395

8.797, 5.097, -1.519

6.514, 4.817, 1.730

8.838, 3.726, -0.766

9.169, 5.175, 1.089

7.460, 6.997, 0.097

9.443, 4.779, -0.421

8.134, 4.239, 1.737

5.927, 3.942, -0.636

7.561, 3.008, 0.170

7.523, 6.381, 1.446

8.672, 3.790, 1.079

6.135, 6.258, 0.743

6.921, 3.749, -1.449

7.722, 4.845, -1.982

6.680, 6.173, -1.398

---------------------------------

display class 3 samples:

-7.423, 3.002, 0.023

-6.733, 6.130, -1.461

-6.201, 4.897, 1.518

-9.026, 5.193, -1.278

-8.424, 4.385, 1.664

-8.278, 6.323, 1.283

-6.831, 3.126, 0.203

-7.229, 3.131, 0.659

-7.470, 3.141, -0.736

-7.871, 5.724, 1.827

-6.765, 3.490, -1.087

-7.639, 6.995, 0.017

-8.148, 4.471, -1.817

-7.911, 6.864, 0.596

-7.603, 6.997, 0.018

-6.763, 3.917, -1.511

-6.672, 3.696, -1.270

-8.905, 3.673, -0.515

-6.986, 4.419, 1.844

-6.121, 3.657, 0.541

-7.472, 7.000, 0.023

-7.121, 3.075, -0.390

-8.263, 5.049, -1.848

-8.140, 3.109, 0.128

-8.877, 5.369, 1.403

-6.218, 5.577, -1.422

-8.041, 6.463, 1.252

-9.391, 4.584, -0.502

-8.363, 6.540, 0.941

-7.508, 6.924, -0.546

---------------------------------

Iteration spend: 414

Print network stats:

inputNum: 3

outputNum: 3

trainIteration: 414

hiddenLayerNum: 2

hiddenLayerHeight: 5

================================

Display error:

0.008 0.008 -0.010

========================

Display targetOutput:

0.000 0.000 1.000

========================

Display biasNodes:

( 1.000, 0.000) ( 1.000, 0.000) ( 1.000, 0.000)

========================

Display networkNodes:

(-7.508, 0.000) ( 6.924, 0.000) (-0.546, 0.000)

( 0.000, -0.000) ( 0.000, -0.000) ( 1.000, 0.000) ( 1.000, 0.000) ( 0.998, -0.000)

( 0.968, -0.000) ( 0.907, -0.000) ( 0.117, 0.000) ( 0.081, 0.000) ( 0.020, 0.000)

( 0.008, 0.000) ( 0.008, 0.000) ( 0.990, -0.000)

========================

Display biasWeights:

(-0.238, 0.000) ( 0.403, 0.000) (-0.471, 0.000) (-0.210, 0.000) (-0.259, 0.000)

( 0.116, 0.000) ( 0.000, 0.000) ( 0.188, 0.000) ( 1.042, 0.000) ( 0.119, 0.000)

(-1.585, 0.000) (-0.686, 0.000) ( 0.265, 0.000)

========================

Display nodeWeights:

Display weights in layer 0:

( 1.051, 1.051) ( 0.872, 0.872) ( -0.790, -0.790) ( -0.897, -0.897) ( -1.345, -1.345)

( -0.370, -0.370) ( -0.366, -0.366) ( 0.386, 0.386) ( 0.375, 0.375) ( -0.555, -0.555)

( -0.011, -0.011) ( -0.299, -0.299) ( 0.514, 0.514) ( 0.233, 0.233) ( -0.284, -0.284)

Display weights in layer 1:

( -2.173, -2.173) ( 0.891, 0.891) ( -1.346, -1.346) ( 0.455, 0.455) ( 1.693, 1.693)

( -1.113, -1.113) ( 0.811, 0.811) ( -1.242, -1.242) ( 0.707, 0.707) ( 1.828, 1.828)

( 1.388, 1.388) ( -0.899, -0.899) ( 0.784, 0.784) ( 0.800, 0.800) ( -1.208, -1.208)

( 1.287, 1.287) ( -1.078, -1.078) ( 0.985, 0.985) ( 0.016, 0.016) ( -1.519, -1.519)

( 0.618, 0.618) ( 4.264, 4.264) ( -3.984, -3.984) ( -4.296, -4.296) ( -1.290, -1.290)

Display weights in layer 2:

( 0.974, 0.974) ( -4.616, -4.616) ( 2.243, 2.243)

( -5.377, -5.377) ( 0.364, 0.364) ( 3.381, 3.381)

( 4.365, 4.365) ( -1.936, -1.936) ( -3.303, -3.303)

( 2.756, 2.756) ( 1.063, 1.063) ( -5.254, -5.254)

( -2.026, -2.026) ( 4.748, 4.748) ( -3.349, -3.349)

========================

Display test result:

Class 1:

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

Class 2:

2

2

2

2

2

2

2

2

2

2

2

2

2

2

2

Class 3:

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

Accuracy: 100.0 %

================================

Average iteration: 414.00

Average accuracy: 100.00

Average training time: 0.07

Press any key to continue . . .

First, we can see the dataset are listed at the top after generated. After the network is trained, we can see the network takes 414 iterations to train with the dataset. The termination rule is that the output error for each trained data has to be less than a threshold ERROR = 0.01.

After finishing training and testing the dataset, it lists a set of network information. The error shows the difference between the output values and target values from the last forward pass. The target output is used for calculating the errors. Since the last sample the network runs belongs to the third class, the target output is (0, 0, 1). The value stored in bias nodes are always one. There is no delta value in bias nodes, so the second term is always 0.

The network nodes are listed layer by layer. Since the input layer has three nodes, there are three set of values in the first line, following by two hidden layers, each consists of five nodes. The last layer tells the information stored in the output nodes. In the input nodes, we can see the nodes contains the values from the last sample data, and the values in the output nodes tell the classify result, which in this case, the third term is the greatest among others, meaning that the network classifies the data into class 3.

In the bias weight section, each line represents the weight values for each bias node. In this case, there are three bias nodes in the network, two for the hidden layers, and one for the output layers. The first term for each weight represents the current weight value, and the second term represents the previous weight before updated. Since there is no need to calculate the delta value for the bias nodes, we do not need to keep track of the old weights connected to bias nodes.

The node weights consist all the weights except bias weights in the network. Each layer of weights represents all the weights between two successive layers. For instance, the node weights in layer 0 are the weights between the input layer and the first hidden layer. The number of lines in each layer represents the number of nodes the weights are connected ‘from’, and the number of weight values in each line represents the number of node the weights are connected ‘to’. For instance, in layer 0, there are three input nodes, and each node connects to five nodes in the next layer with a weight. Therefore, there the first weight in the first line means the weight connected from the first input node to the first node in the next layer, and the third weight in the second line represents the weight connects from the second input node to the third node in the next layer. Since the network is already trained, there is barely any difference between two terms.

Then the test result is printed for the testing data in each class. At the end, the time performance of training the data set is 0.06 seconds. Since we only test with one data set, the average iteration and average accuracy are the same as the test result.

**Non-linearly separable**

display all samples:

display class 1 samples:

0.002, 13.000, -0.023

1.714, 14.040, 0.378

-0.021, 13.001, 0.059

-1.962, 14.989, -0.386

-1.616, 15.816, -0.850

1.317, 13.591, -0.531

1.155, 13.444, 0.494

1.873, 14.756, 0.658

-0.255, 16.960, -0.303

1.508, 16.219, -0.490

-1.421, 15.248, -1.385

-0.885, 13.207, -0.056

-1.261, 16.489, 0.440

1.077, 14.657, -1.650

\*\*\*\* -7.272, 5.579, -1.901

-0.395, 13.734, 1.497

1.599, 16.166, -0.286

-0.312, 13.150, 0.694

-0.004, 16.836, -0.794

-0.160, 13.276, -1.001

0.887, 16.783, 0.184

1.507, 13.969, -0.816

1.164, 15.979, -1.299

1.486, 13.668, 0.142

-0.684, 14.584, 1.833

0.096, 16.997, 0.036

-0.609, 13.715, 1.407

-0.337, 16.466, -1.318

0.393, 16.924, -0.381

\*\*\*\* -5.819, 5.371, 1.019

---------------------------------

display class 2 samples:

6.368, 6.355, -0.939

9.496, 4.879, -0.006

8.253, 3.776, 1.391

6.889, 6.881, -0.300

6.418, 3.927, 1.295

8.669, 4.072, -1.331

9.159, 3.937, -0.343

8.288, 4.154, -1.632

7.487, 3.098, -0.618

7.046, 6.904, 0.409

9.227, 6.008, -0.043

7.312, 6.355, 1.459

8.778, 5.914, 1.237

7.506, 5.963, -1.753

\*\*\*\* -0.884, 15.494, -1.725

6.760, 5.505, 1.788

7.133, 5.756, -1.815

7.708, 3.072, 0.490

7.176, 6.233, 1.541

8.662, 6.626, -0.064

6.279, 3.462, -0.377

7.457, 3.705, -1.523

8.197, 6.869, 0.150

7.225, 6.964, -0.263

7.695, 6.976, 0.240

8.120, 3.598, 1.285

7.461, 6.999, 0.032

5.633, 5.375, 0.612

8.179, 3.133, -0.227

\*\*\*\* 1.165, 16.130, 1.169

---------------------------------

display class 3 samples:

-8.246, 3.144, 0.019

-7.197, 3.023, 0.023

-7.294, 3.017, 0.162

-8.070, 6.640, -0.992

-6.555, 4.594, -1.715

-6.100, 5.675, -1.258

-7.081, 3.974, -1.665

-8.972, 3.693, -0.352

-5.796, 4.907, -1.043

-6.302, 3.401, -0.088

-7.515, 6.988, 0.222

-7.049, 3.745, -1.490

-8.814, 5.426, -1.447

-8.737, 4.949, 1.571

\*\*\*\* 7.415, 3.003, -0.061

-6.091, 4.985, -1.420

-7.021, 3.064, 0.149

-7.371, 3.017, 0.224

-7.139, 3.140, 0.640

-8.957, 6.272, -0.509

-6.524, 3.309, -0.433

-7.083, 6.925, 0.349

-7.841, 3.241, -0.889

-9.473, 4.704, 0.136

-6.687, 3.315, -0.709

-6.322, 5.704, 1.455

-8.493, 6.560, -0.762

-8.181, 6.881, 0.013

-7.697, 6.990, 0.037

\*\*\*\* 9.086, 6.122, 0.475

---------------------------------

Iteration spend: 30001

Print network stats:

inputNum: 3

outputNum: 3

trainIteration: 30001

hiddenLayerNum: 2

hiddenLayerHeight: 5

================================

Display error:

0.000 0.999 -0.999

========================

Display targetOutput:

0.000 0.000 1.000

========================

Display biasNodes:

( 1.000, 0.000) ( 1.000, 0.000) ( 1.000, 0.000)

========================

Display networkNodes:

( 9.086, 0.000) ( 6.122, 0.000) ( 0.475, 0.000)

( 0.000, -0.000) ( 0.000, -0.000) ( 0.000, -0.000) ( 0.000, -0.000) ( 1.000, 0.000)

( 0.000, -0.000) ( 0.616, 0.000) ( 0.914, 0.000) ( 0.011, -0.000) ( 0.075, 0.000)

( 0.000, 0.000) ( 0.999, 0.000) ( 0.001, -0.000)

========================

Display biasWeights:

(-0.382, 0.000) ( 1.025, 0.000) (-0.088, 0.000) (12.404, 0.000) (-2.608, 0.000)

( 0.782, 0.000) ( 1.773, 0.000) (-0.381, 0.000) ( 4.620, 0.000) (-0.630, 0.000)

(-1.495, 0.000) ( 0.602, 0.000) (-5.659, 0.000)

========================

Display nodeWeights:

Display weights in layer 0:

( -1.573, -1.573) ( -3.263, -3.263) ( -1.532, -1.532) ( 9.048, 9.048) ( 7.786, 7.786)

( 0.877, 0.877) ( 1.075, 1.075) ( 0.811, 0.811) (-25.225, -25.225) ( 5.171, 5.171)

( 0.095, 0.095) ( 0.811, 0.811) ( 0.150, 0.150) ( 4.481, 4.481) ( -9.963, -9.963)

Display weights in layer 1:

( -0.992, -0.992) ( -4.442, -4.442) ( -1.874, -1.874) ( -1.575, -1.575) ( 4.194, 4.194)

( 5.745, 5.745) ( -0.176, -0.176) ( -6.269, -6.269) ( 2.395, 2.395) ( -0.591, -0.591)

( -0.734, -0.734) ( -3.719, -3.719) ( -1.731, -1.731) ( -1.015, -1.015) ( 3.092, 3.092)

( 9.888, 9.888) ( -0.979, -0.979) ( -3.331, -3.331) ( 5.347, 5.347) ( -2.463, -2.463)

( -8.796, -8.796) ( -1.301, -1.301) ( 2.750, 2.750) ( -9.133, -9.133) ( -1.879, -1.879)

Display weights in layer 2:

( -1.816, -1.816) ( -9.459, -9.459) ( 6.764, 6.764)

( -7.889, -7.889) ( 3.622, 3.622) ( 3.043, 3.043)

( -1.961, -1.961) ( 4.645, 4.645) ( -2.820, -2.820)

( -3.865, -3.865) ( -1.575, -1.575) ( 8.411, 8.411)

( 4.364, 4.364) ( -3.099, -3.099) ( -6.552, -6.552)

========================

Display test result:

Class 1:

1

1

1

1

1

1

1

1

1

1

1

1

1

1

3

Class 2:

2

2

3

2

2

2

2

2

2

2

2

2

2

3

1

Class 3:

3

3

3

3

3

3

3

3

3

3

3

3

3

3

2

Accuracy: 88.9 %

================================

Average iteration: 30001.00

Average accuracy: 88.89

Average training time: 3.88

Press any key to continue . . .

In this case, the dataset is listed at the top, and the swapped data is marked with ‘\*\*\*\*’ at front. The number of iteration is 30001, and the training terminates due to exceeding the iteration limit 30000.

Since we swap the 15th sample and 30th in each class, the error values are significant since the last sample is the swapped data, which can easily confuse the network and cause the error. If we look at the output layer in network nodes, the network and the output result classify the sample into class 2.

And in the testing result, not all the samples are classified correctly. The accuracy is 88.9 percent. The time performance for training is 3.88, meaning that each iteration takes about 0.0001 seconds to run.

Next, I am going to test with the impact of different parameters by comparing the performance. Each performance factor is collected with testing the network with 30 different datasets. For demonstration purpose, the dataset and network information are omitted in this section, and the network has different configuration for each parameter test.

**Additional Testing**

1. **Sample Scalability**

Number of hidden layer: 1

Number of nodes in each hidden layer: 8

Minimum initial weight: -0.5

Maximum initial weight: 0.5

Learning rate: 1 + 0.2 \* layer depth

**Linearly separable**

|  |  |  |  |
| --- | --- | --- | --- |
| Scale Index | Average Iteration | Average Accuracy (%) | Average Training Time (s) |
| 1 | 454.1 | 100 | 0.04 |
| 10 | 632.4 | 100 | 0.06 |
| 20 | 1629.3 | 98.89 | 0.13 |
| 30 | 1622.37 | 98.89 | 0.13 |
| 40 | 1157.37 | 100.00 | 0.1 |

Overall, the average number of iteration and average training time increase. The main reason is the same learning rate. With the same set of learning rates and increased sample values, it requires more iteration to modify weight to classify the samples. The decreased trend with scale index 40 can be caused by the dataset, for in each run, the data set is randomly generated and changes the difficulty of learning and testing.

**Non-linearly separable**

|  |  |  |  |
| --- | --- | --- | --- |
| Scale Index | Average Iteration | Average Accuracy (%) | Average Training Time (s) |
| 1 | 30001 | 89.26 | 3.19 |
| 10 | 30001 | 93.33 | 3.03 |
| 20 | 30001 | 92.3 | 3.29 |
| 30 | 30001 | 93.26 | 3.38 |
| 40 | 30001 | 90.22 | 3.44 |

The stats appear approximately the same with different scale index. Since in the non-linear separable problem, we also swap the 30th sample, which is part of the testing data. Therefore, the accuracy 93.33 percent means that the network knows the swapped data does not belong to the assigned class. With the interfered data, the network might not classify all other data correctly. In addition, time performance gradually increase as scale index increases, which might be the same reason with linearly separable problem.

1. **Initial Weights**

Number of hidden layer: 1

Number of nodes in each hidden layer: 8

Scale index: 1

Learning rate: 1 + 0.2 \* layer depth

**Linearly separable**

|  |  |  |  |
| --- | --- | --- | --- |
| Initial Weights | Average Iteration | Average Accuracy (%) | Average Training Time (s) |
| [-0.5, 0.5] | 414.8 | 100 | 0.05 |
| [0, 1] | 579.4 | 100 | 0.07 |
| [-1, 0] | 633.23 | 100 | 0.09 |
| [-1, 1] | 500.9 | 100 | 0.08 |
| [0, 2] | 766.37 | 100 | 0.1 |
| [-2, 0] | 3189.27 | 98.89 | 0.35 |
| [-2, 2] | 552.77 | 100 | 0.1 |

Based on the result, we can see that the ideal initial weights are the range that convers positive values and negative values. Closer the range towards 0, the better the performance. Also, the positive-only range performs better than negative-only range. This might be caused by the sphere centers, which has only one negative number and the rest are either 0 or positive values.

**Non-linearly separable**

|  |  |  |  |
| --- | --- | --- | --- |
| Initial Weights | Average Iteration | Average Accuracy (%) | Average Training Time (s) |
| [-0.5, 0.5] | 29860.9 | 89.04 | 3.13 |
| [0, 1] | 30001 | 89.85 | 4.6 |
| [-1, 0] | 30001 | 88 | 4.71 |
| [-1, 1] | 30001 | 88.81 | 3.09 |
| [0, 2] | 30001 | 89.63 | 4.84 |
| [-2, 0] | 30001 | 86.67 | 4.67 |
| [-2, 2] | 30001 | 88.59 | 4.8 |

Like the behavior we observe in the separable problem, the training time increases as the range only takes positive or negative values. The ideal weight range is still the range that includes both signs and closer to 0.

1. **Network Size**

Minimum initial weight: -0.5

Maximum initial weight: 0.5

Scale index: 1

Learning rate: 1 + 0.2 \* layer depth

Next, I am going to test the performance with different network sizes on linearly separable problem. In the result tables, the top row represents number of nodes in each hidden layer, and the first column represents number of hidden layers in the network. Both input and output layers are fixed to three nodes in this problem.

* **Average iteration:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| # of nodes  # of layers | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 4511.83 | 1205.53 | 934.1 | 695.37 | 567.47 | 503.07 | 446.33 | 387.07 |
| 2 | 14199.8 | 6720.97 | 4510.3 | 841.2 | 403.33 | 543 | 297 | 295.87 |
| 3 | 19420.67 | 14937.57 | 9621.8 | 7804.73 | 2895.53 | 2004.4 | 434.03 | 343.93 |
| 4 | 21676.87 | 14306.43 | 8485.1 | 2641.77 | 3008.73 | 2425.8 | 922.8 | 829.97 |

* **Average accuracy (%):**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| # of nodes  # of layers | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 96.67 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| 2 | 85.56 | 93.33 | 95.56 | 100 | 100 | 100 | 100 | 100 |
| 3 | 80 | 86.67 | 91.11 | 94.44 | 97.78 | 98.89 | 100 | 100 |
| 4 | 77.78 | 85.56 | 93.3 | 98.89 | 98.89 | 98.89 | 100 | 100 |

* **Average training time (s):**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| # of nodes  # of layers | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 0.3 | 0.9 | 0.08 | 0.09 | 0.06 | 0.06 | 0.06 | 0.05 |
| 2 | 0.78 | 0.52 | 0.69 | 0.13 | 0.07 | 0.14 | 0.1 | 0.1 |
| 3 | 1.42 | 2.06 | 1.6 | 1.63 | 1.11 | 0.64 | 0.18 | 0.19 |
| 4 | 2.18 | 2.18 | 1.74 | 0.75 | 1.08 | 1.03 | 0.47 | 0.52 |

Overall, the number of iteration increases as we add more hidden layers, and the number of iteration decreases as we add more nodes to each hidden layer. It can be affected by the limitation in backpropagation algorithm since with more hidden layers, the impact of input and the error can be attenuated during both forward pass and backward pass.

For accuracy, adding more hidden layers can potentially decreases the accuracy. This can be caused by the problem, for it is a linearly separable problem, which can be easily classified by a linear classifier. In the network, a hidden layer is like a linear classifier, so multiple layers can easily over train the network and produce inefficient result. As we add more nodes into hidden layers, the accuracy increased. It makes sense because with one more node in a hidden layer, we have one more variable to help classify the data.

For time performance, the training time decreases as we add more nodes into layers, but it increases as we add more layers to the network. We can also observe that time performance has positive relationship with number of training iteration.

After testing, the best network size for this problem happens with one hidden layer and 9 nodes in the hidden layer. Even though the number of training iteration is greater than the same number of nodes with two hidden layers, the average training time is two times faster than the best case with two hidden layers, which may be caused by the reduced number of nodes required to train.

1. **Learning rate**

Number of hidden layer: 2

Number of nodes in each hidden layer: 6

Minimum initial weight: -0.5

Maximum initial weight: 0.5

In this section, we are going to focus on the impact of learning rate on the linearly separable problem. Each result table consists of two parameters: learning base and increase rate. Due to attenuation effect from backpropagation algorithm, I set a base value for all the learning rate, and also set an increase rate that will increase the learning rate layer by layer during backward pass. For instance, if the learning base is 1 and learning rate is 0.2, the learning rate for all the weights connect to the output nodes will be 1, and the weights that connect to the layer before is 1.2, and so on. In order to observe the impact of learning rate and increased rate, the size of hidden layer is set to 2 layers with 6 nodes in each layer.

* **Average iteration:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Inc. rate  Learn base | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 |
| 0.5 | 784 | 858.07 | 30001 | 30001 | 30001 |
| 0.75 | 527.8 | 535.63 | 628.43 | 30001 | 30001 |
| 1 | 381.5 | 1435.97 | 419.6 | 492 | 6309.4 |
| 1.25 | 1322.07 | 324.3 | 346.1 | 359 | 396.37 |
| 1.5 | 269.2 | 247.33 | 274.5 | 387.4 | 1308.47 |

* **Average accuracy (%):**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Inc. rate  Learn base | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 |
| 0.5 | 100 | 100 | 33.33 | 33.33 | 33.33 |
| 0.75 | 100 | 100 | 100 | 33.33 | 33.33 |
| 1 | 100 | 100 | 100 | 100 | 100 |
| 1.25 | 98.89 | 100 | 100 | 100 | 100 |
| 1.5 | 100 | 100 | 100 | 100 | 98.89 |

* **Average training time (s):**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Inc. rate  Learn base | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 |
| 0.5 | 0.11 | 0.11 | 3.19 | 3.26 | 3.26 |
| 0.75 | 0.07 | 0.08 | 0.11 | 3.26 | 3.21 |
| 1 | 0.05 | 0.17 | 0.06 | 0.07 | 0.71 |
| 1.25 | 0.16 | 0.05 | 0.06 | 0.05 | 0.06 |
| 1.5 | 0.04 | 0.04 | 0.04 | 0.05 | 0.16 |

Throughout the test, we can see that if the learning rates for each layer differs more, the network has more potential to fail the test, and larger the learning base, the less the training iteration and training time. However, if we set the learning base value too high, the network can easily fail to learn from the dataset too. In this case, learning base 1.5 with increase rate 0.2 seems to be the best combination for this classification problem.

Source Code:  
  
/\* CS6200 Che Shian Hung 3/12/2018

Programming Assignment 2

Purpose: This program uses backpropagation algorithm to train the network for three classes.

Each sample is represented as a three dimensional vector, and there are 30 sample data

randomly generated for each class everytime the program runs. The first half of data

is used for training, and the second half of data is used for testing the network

after training. There are three fixed sphere centers that are used to generate the data, and

the distances between them have to be greater than 10. The data will then be randomly

generated inside each sphere with radius 2. The user can modify the global variables at the

top to configure the network setting and the dataset. If the problem is configured to be

non-separable, the last sample for both training and testing for each class will

be swap to the next class. For instance, the samples for class 1 will now have the samples

for class 3. In addition, we can also easily modify the variables such as

TEST\_ITERATION and INPUT\_NUM to assess the network performance with different setting.

Architecture: There are mainly three steps in the program: data generation, network training,

and testing. The whole network is a class, which contains all functionalities for running

backpropagation algorithm. Each step has been encapsulated in few functions. Also, for testing purpose,

each function has been designed for reusability. For instance, we can display any information

related to the network. In the network, backpropagation algorithm consists of mainly two functions:

forwardPass and backwardPass, and forwardPass function can be also used in testing phase.

Data Structure: For the data set, the information is generated and stored in statically allocated arrays.

For the nodes and weights in the network, I created two structs to handle the information used in

backpropagation algorithm. Inside the BackpropNetwork class, the network is presented as a bunch of

weights arrays and nodes arrays that are dynamically allocated with the setting, so that we can

easily adjust layer height and number of layers to observe the changes. While training and testing,

the network class requires to take a data set as a parameter to increase cohesion between dataset

and running network.

\*/

#define \_USE\_MATH\_DEFINES

// Import libraries and constants

#include<iostream>

#include<stdlib.h>

#include<time.h>

#include<cmath>

#include<string>

#define TEST\_ITERATION 30 // Number of testing with differnt generated dataset

#define MODIFY\_DATA false // Switch for making the problem linearly separable/ non-separable

#define SCALE\_INDEX 1 // Number and switch to scale the dataset

#define INITIAL\_WEIGHT\_MIN -0.5 // Minimum initial weight for the network

#define INITIAL\_WEIGHT\_MAX 0.5 // Maximum initial weight for the network

#define INPUT\_NUM 3 // Number of input feature for the network

#define HIDDEN\_LAYER\_NUM 1 // Number of hidden layer for the network

#define HIDDEN\_LAYER\_HEIGHT 8 // Number of node in each hidden layer for the network

#define LEARNING\_BASE 1.5 // The greek symbol in gradient decent algorithm (zeta...???)

#define LEARNING\_RATE 0.3 // Amount of increased learning rate after each layer for the network

#define OUTPUT\_NUM 3 // Number of output for the network

#define ERROR 0.01 // The error threshold for training

#define MAX\_ITERATION 30000 // Iteration limit for training with backpropagation network

#define RADIUS 2 // Radius for the class sphere

#define TEST\_SIZE 15 // Number of testing data in each class

#define CLASS\_NUM 3 // Number of classes

#define DIMENSION 3 // Dimensionality for each sample

#define SAMPLE\_SIZE 30 // Number of sample in each class

#define MODIFY\_INDEX 15 // The index of sample data that is chose to be non-separable while training

using namespace std;

struct edge{

double weight; // The current weight

double oldWeight; // Weight before modification (used in backward pass)

};

struct node {

double value; // Input/output value stored in the node

double delta; // Delta value for the current node (used in backward pass)

};

class BackpropNetwork {

private:

int inputNum; // Number of input features

int outputNum; // Number of output nodes

int trainIteration; // Number of iteration for current training

int hiddenLayerNum; // Number of hidden layers

int hiddenLayerHeight; // Number of hidden layers' height

int\*\* testResult; // Test result (class number) of the current test

double accuracy; // Accuracy of the current test

double\* error; // Error vector for each forward pass (Output - target)

double\* targetOutput; // Target output for current training data

node \*biasNodes; // Bias nodes for the network

node \*\*networkNodes; // Input nodes + hidden nodes + output nodes for the network

edge \*\*biasWeights; // Bias weight for each bias node

edge \*\*\*nodeWeights; // All other weights for the rest of the nodes

void destroyNetwork(); // Delete and reset all pointers

void initializePointers(); // Initialize all pointers

void resetPointers(); // Reset the pointer values

void pointersToNull(); // Set all pointers to null

void forwardPass(); // Forward pass

void backwardPass(); // Backward pass

void setInput(double\* inputVector); // Set input values in the input nodes (network nodes)

void setTarget(double\* targetVector); // Set output values in the output nodes (network nodes)

int getClass(); // Determine the maximum value stored in the output nodes and return class number

double sigmoidFunction(double net); // Sigmoid function with an input x

double randomDouble(double min, double max); // Return a random double between a minimum and a maximum value

public:

BackpropNetwork(); // Constructor

~BackpropNetwork(); // Destructor

void trian(double dataSet[][SAMPLE\_SIZE][DIMENSION]); // Train the network with data set

void test(double dataSet[][SAMPLE\_SIZE][DIMENSION]); // Test the network with data set

void displayStats(); // Display none pointer information in the network

void displayAllStats(); // Display all stats about the network

void displayError(); // Display error

void displayTargetOutput(); // Display targetOutput

void displayBias(); // Display values stored in bias nodes

void displayNetwork(); // Display values stored in network nodes

void displayBiasWeight(); // Display values stored in bias weight

void displayNodeWeight(); // Display values stored in nwteork nodes

void displayAccuracy(); // Dispaly accuracy

void displayTestReport(); // Display test report

int getIteration() { return trainIteration; } // Return train iteration for the last training

double getAccuracy() { return accuracy; } // Return accuracy for the last training

};

// Declare global constant variable

const double sphereCenter[CLASS\_NUM][DIMENSION] = { { 0, 15, 0 },{ 7.5, 5, 0 },{ -7.5, 5, 0 } }; // Hard coded sphere centers

// Declare global variables

double classSamples[CLASS\_NUM][SAMPLE\_SIZE][DIMENSION]; // Sample for all three classes, including training data and testing data

int resultClass[CLASS\_NUM][SAMPLE\_SIZE - TEST\_SIZE]; // Captures the testing result after testing with trained classifiers

// Define global fuctions

void generateAllSamples(); // Generate samples randomly for all classes

void generateRandomSamples(double samples[SAMPLE\_SIZE][DIMENSION], int classNum); // Generate samples randomly for a specific class

void displayAllSamples(); // Display samples for all classes

void displaySamples(int classNum); // Display samples for a specific class

void modifyTrainedData(int modifyIndex);

void scaleAllSamples();

int main() {

srand(time(NULL));

double iterationTotal = 0;

double accuracyTotal = 0;

double totalTime = 0;

BackpropNetwork n; // Create a new network

for (int i = 0; i < TEST\_ITERATION; i++) {

generateAllSamples(); // Generate sample data and display for each class

if (SCALE\_INDEX != 1) scaleAllSamples(); // Scale the sample data if required

if (MODIFY\_DATA) { // Swap specific samples to have non-separable problem

modifyTrainedData(MODIFY\_INDEX); // Modify the 15th sample for each class

modifyTrainedData(SAMPLE\_SIZE); // Modify the last sample for each class

}

//displayAllSamples();

clock\_t start = clock();

n.trian(classSamples); // Train the netwrok with dataset

clock\_t end = clock();

totalTime += (double)(end - start) / CLOCKS\_PER\_SEC;

n.test(classSamples); // Test the network with dataset

n.displayAccuracy(); // Display accuracy for testing result

//n.displayAllStats();

//n.displayTestReport();

iterationTotal += n.getIteration();

accuracyTotal += n.getAccuracy();

}

printf("Layer num: %d\nNode num: %d\n", HIDDEN\_LAYER\_NUM, HIDDEN\_LAYER\_HEIGHT);

printf("Average iteration: %5.2f\n", iterationTotal / TEST\_ITERATION);

printf("Average accuracy: %5.2f\n", accuracyTotal / TEST\_ITERATION);

printf("Average training time: %5.2f\n\n", totalTime / TEST\_ITERATION);

iterationTotal = 0;

accuracyTotal = 0;

totalTime = 0;

system("pause");

return 0;

}

void generateAllSamples() {

for (int i = 0; i < 3; i++) {

generateRandomSamples(classSamples[i], i + 1);

}

}

void generateRandomSamples(double sample[SAMPLE\_SIZE][DIMENSION], int classNum) {

for (int j = 0; j < SAMPLE\_SIZE; j++) {

double theta = rand() % 6282 / double(1000);

double phi = rand() % 3141 / double(1000) - 1.5705;

sample[j][0] = sphereCenter[classNum - 1][0] + RADIUS \* cos(theta) \* cos(phi);

sample[j][1] = sphereCenter[classNum - 1][1] + RADIUS \* sin(phi);

sample[j][2] = sphereCenter[classNum - 1][2] + RADIUS \* sin(theta) \* cos(phi);

}

}

void displayAllSamples() {

printf("display all samples:\n");

for (int i = 1; i < DIMENSION + 1; i++) displaySamples(i);

printf("\n\n");

}

void displaySamples(int classNum) {

printf("display class %d samples:\n", classNum);

for (int i = 0; i < SAMPLE\_SIZE; i++) {

if ((i == MODIFY\_INDEX - 1 && MODIFY\_DATA) || (i == SAMPLE\_SIZE - 1 && MODIFY\_DATA)) printf("\*\*\*\*");

for (int j = 0; j < DIMENSION; j++) {

if (j != 2)

printf("%7.3f, ", classSamples[classNum - 1][i][j]);

else

printf("%7.3f\n", classSamples[classNum - 1][i][j]);

}

}

printf("---------------------------------\n\n");

}

void modifyTrainedData(int modifyIndex) {

modifyIndex--;

double class12Sample[2][DIMENSION] = { { classSamples[0][modifyIndex][0], classSamples[0][modifyIndex][1], classSamples[0][modifyIndex][2] },{ classSamples[1][modifyIndex][0], classSamples[1][modifyIndex][1], classSamples[1][modifyIndex][2] } };

for (int i = 0; i < CLASS\_NUM; i++) {

for (int j = 0; j < DIMENSION; j++) {

if (i == 0) classSamples[i][modifyIndex][j] = classSamples[2][modifyIndex][j];

else classSamples[i][modifyIndex][j] = class12Sample[i - 1][j];

}

}

}

void scaleAllSamples() {

for (int i = 0; i < CLASS\_NUM; i++) {

for (int j = 0; j < SAMPLE\_SIZE; j++) {

for (int k = 0; k < DIMENSION; k++) {

classSamples[i][j][k] \*= SCALE\_INDEX;

}

}

}

}

BackpropNetwork::BackpropNetwork() {

inputNum = INPUT\_NUM;

outputNum = OUTPUT\_NUM;

hiddenLayerNum = HIDDEN\_LAYER\_NUM;

hiddenLayerHeight = HIDDEN\_LAYER\_HEIGHT;

accuracy = -1;

trainIteration = -1;

pointersToNull();

initializePointers();

resetPointers();

}

BackpropNetwork::~BackpropNetwork() {

destroyNetwork();

}

void BackpropNetwork::destroyNetwork() {

for (int i = 0; i < CLASS\_NUM; i++)

delete[] testResult[i];

delete[] testResult;

delete[] error;

delete[] targetOutput;

delete[] biasNodes;

for (int i = 0; i < hiddenLayerNum + 2; i++)

delete[] networkNodes[i];

delete[] networkNodes;

for (int i = 0; i < hiddenLayerNum + 1; i++) {

delete[] biasWeights[i];

int height = hiddenLayerHeight;

if (i == 0) height = inputNum;

for (int j = 0; j < height; j++)

delete[] nodeWeights[i][j];

delete[] nodeWeights[i];

}

delete[] biasWeights;

delete[] nodeWeights;

pointersToNull();

}

void BackpropNetwork::initializePointers() {

testResult = new int\*[CLASS\_NUM];

for (int i = 0; i < CLASS\_NUM; i++) testResult[i] = new int[TEST\_SIZE];

error = new double[outputNum];

targetOutput = new double[outputNum];

biasNodes = new node[hiddenLayerNum + 1];

networkNodes = new node\*[hiddenLayerNum + 2];

networkNodes[0] = new node[inputNum];

networkNodes[hiddenLayerNum + 1] = new node[outputNum];

biasWeights = new edge\*[hiddenLayerNum + 1];

nodeWeights = new edge\*\*[hiddenLayerNum + 1];

nodeWeights[0] = new edge\*[inputNum];

for (int i = 0; i < inputNum; i++) nodeWeights[0][i] = new edge[hiddenLayerHeight];

for (int i = 0; i < hiddenLayerNum; i++) {

networkNodes[i + 1] = new node[hiddenLayerHeight];

biasWeights[i] = new edge[hiddenLayerHeight];

nodeWeights[i + 1] = new edge\*[hiddenLayerHeight];

for (int j = 0; j < hiddenLayerHeight; j++)

nodeWeights[i + 1][j] = new edge[hiddenLayerHeight];

}

for (int i = 0; i < hiddenLayerHeight; i++)

nodeWeights[hiddenLayerNum][i] = new edge[outputNum];

biasWeights[hiddenLayerNum] = new edge[outputNum];

}

void BackpropNetwork::resetPointers() {

for (int i = 0; i < CLASS\_NUM; i++)

for (int j = 0; j < TEST\_SIZE; j++) testResult[i][j] = 0;

for (int i = 0; i < inputNum; i++) {

networkNodes[0][i].value = 0;

networkNodes[0][i].delta = 0;

}

for (int i = 0; i < outputNum; i++) {

error[i] = 0;

targetOutput[i] = 0;

networkNodes[hiddenLayerNum + 1][i].value = 0;

networkNodes[hiddenLayerNum + 1][i].delta = 0;

biasWeights[hiddenLayerNum][i].oldWeight = 0;

biasWeights[hiddenLayerNum][i].weight = randomDouble(INITIAL\_WEIGHT\_MIN, INITIAL\_WEIGHT\_MAX);

}

for (int i = 0; i < hiddenLayerNum + 1; i++) {

biasNodes[i].value = 1;

biasNodes[i].delta = 0;

}

for (int i = 0; i < hiddenLayerNum; i++) {

for (int j = 0; j < hiddenLayerHeight; j++) {

networkNodes[i + 1][j].value = 0;

networkNodes[i + 1][j].delta = 0;

biasWeights[i][j].oldWeight = 0;

biasWeights[i][j].weight = randomDouble(INITIAL\_WEIGHT\_MIN, INITIAL\_WEIGHT\_MAX);

}

}

for (int i = 0; i < hiddenLayerNum + 1; i++) {

int fromNodeNum;

if (i == 0) fromNodeNum = inputNum;

else fromNodeNum = hiddenLayerHeight;

for (int j = 0; j < fromNodeNum; j++) {

int toNodeNum = hiddenLayerHeight;

if (i == hiddenLayerNum) toNodeNum = outputNum;

for (int k = 0; k < toNodeNum; k++) {

nodeWeights[i][j][k].oldWeight = 0;

nodeWeights[i][j][k].weight = randomDouble(INITIAL\_WEIGHT\_MIN, INITIAL\_WEIGHT\_MAX);

}

}

}

}

void BackpropNetwork::pointersToNull() {

testResult = NULL;

error = NULL;

targetOutput = NULL;

biasNodes = NULL;

networkNodes = NULL;

biasWeights = NULL;

nodeWeights = NULL;

}

void BackpropNetwork::forwardPass() {

for (int i = 0; i < hiddenLayerNum + 1; i++) {

int toNodeNum = hiddenLayerHeight;

if (i == hiddenLayerNum) toNodeNum = outputNum;

for (int j = 0; j < toNodeNum; j++) {

int fromNodeNum = hiddenLayerHeight;

if (i == 0) fromNodeNum = inputNum;

double net = 0;

for (int k = 0; k < fromNodeNum; k++)

net += networkNodes[i][k].value \* nodeWeights[i][k][j].weight;

net += biasNodes[i].value \* biasWeights[i][j].weight;

networkNodes[i + 1][j].value = sigmoidFunction(net);

}

}

for (int i = 0; i < outputNum; i++)

error[i] = networkNodes[hiddenLayerNum + 1][i].value - targetOutput[i];

}

void BackpropNetwork::backwardPass() {

for (int i = hiddenLayerNum; i >= 0; i--) {

int toNodeNum = hiddenLayerHeight;

if (i == hiddenLayerNum) toNodeNum = outputNum;

double learningRate = (LEARNING\_BASE + (LEARNING\_RATE \* (i - hiddenLayerNum)));

for (int j = 0; j < toNodeNum; j++) {

double outNet = networkNodes[i + 1][j].value \* (1 - networkNodes[i + 1][j].value);

double totalOut = 0;

if (i == hiddenLayerNum) totalOut = error[j];

else {

int nextLayerNodeNum = hiddenLayerHeight;

if (i + 1 == hiddenLayerNum) nextLayerNodeNum = outputNum;

for (int l = 0; l < nextLayerNodeNum; l++)

totalOut += networkNodes[i + 2][l].delta \* nodeWeights[i + 1][j][l].oldWeight;

}

networkNodes[i + 1][j].delta = totalOut \* outNet;

int fromNodeNum = hiddenLayerHeight;

if (i == 0) fromNodeNum = inputNum;

for (int k = 0; k < fromNodeNum; k++) {

nodeWeights[i][k][j].oldWeight = nodeWeights[i][k][j].weight;

nodeWeights[i][k][j].weight -= learningRate \* networkNodes[i + 1][j].delta \* networkNodes[i][k].value;

}

biasWeights[i][j].weight -= learningRate \* networkNodes[i + 1][j].delta;

}

}

}

void BackpropNetwork::setInput(double\* inputVector) {

for (int i = 0; i < inputNum; i++)

networkNodes[0][i].value = inputVector[i];

}

void BackpropNetwork::setTarget(double\* targetVector) {

for (int i = 0; i < outputNum; i++)

targetOutput[i] = targetVector[i];

}

int BackpropNetwork::getClass() {

int maxClass = -1;

double max = -100;

for (int i = 0; i < outputNum; i++) {

if (max < networkNodes[hiddenLayerNum + 1][i].value) {

max = networkNodes[hiddenLayerNum + 1][i].value;

maxClass = i + 1;

}

}

return maxClass;

}

double BackpropNetwork::sigmoidFunction(double net) {

return 1 / (1 + exp(net \* (-1)));

}

double BackpropNetwork::randomDouble(double min, double max) {

return (double(rand()) / float(RAND\_MAX)) \* (max - min) + min;

}

void BackpropNetwork::trian(double dataSet[][SAMPLE\_SIZE][DIMENSION]) {

int iteration = 0;

bool finished = false;

resetPointers();

while (!finished && iteration <= MAX\_ITERATION) {

finished = true;

iteration++;

for (int i = 0; i < CLASS\_NUM; i++) {

double targetOutput[3] = { 0, 0, 0 };

targetOutput[i] = 1;

setTarget(targetOutput);

for (int j = 0; j < TEST\_SIZE; j++) {

setInput(dataSet[i][j]);

forwardPass();

backwardPass();

for (int i = 0; i < outputNum; i++) {

if (finished && abs(error[i]) >= ERROR) {

finished = false;

}

}

}

}

}

printf("Iteration spend: %d\n\n", iteration);

trainIteration = iteration;

}

void BackpropNetwork::test(double dataSet[][SAMPLE\_SIZE][DIMENSION]) {

double correctCounter = 0;

for (int i = 0; i < CLASS\_NUM; i++) {

for (int j = TEST\_SIZE; j < SAMPLE\_SIZE; j++) {

setInput(dataSet[i][j]);

forwardPass();

testResult[i][j - TEST\_SIZE] = getClass();

if (testResult[i][j - TEST\_SIZE] == i + 1) correctCounter++;

}

}

accuracy = correctCounter / (TEST\_SIZE \* CLASS\_NUM) \* 100;

}

void BackpropNetwork::displayStats() {

printf("Print network stats:\n");

printf("inputNum: %d\n", inputNum);

printf("outputNum: %d\n", outputNum);

printf("trainIteration: %d\n", trainIteration);

printf("hiddenLayerNum: %d\n", hiddenLayerNum);

printf("hiddenLayerHeight: %d\n", hiddenLayerHeight);

printf("================================\n\n");

}

void BackpropNetwork::displayAllStats() {

displayStats();

displayError();

displayTargetOutput();

displayBias();

displayNetwork();

displayBiasWeight();

displayNodeWeight();

displayTestReport();

}

void BackpropNetwork::displayError() {

printf("Display error:\n\n");

for (int i = 0; i < outputNum; i++)

printf("%6.3f ", error[i]);

printf("\n========================\n\n");

}

void BackpropNetwork::displayTargetOutput() {

printf("Display targetOutput:\n\n");

for (int i = 0; i < outputNum; i++)

printf("%6.3f ", targetOutput[i]);

printf("\n========================\n\n");

}

void BackpropNetwork::displayBias() {

printf("Display biasNodes:\n\n");

for (int i = 0; i < hiddenLayerNum + 1; i++)

printf("(%6.3f, %6.3f) ", biasNodes[i].value, biasNodes[i].delta);

printf("\n========================\n\n");

}

void BackpropNetwork::displayNetwork() {

printf("Display networkNodes:\n\n");

for (int i = 0; i < hiddenLayerNum + 2; i++) {

int height = hiddenLayerHeight;

if (i == 0) height = inputNum;

else if (i == hiddenLayerNum + 1) height = outputNum;

for (int j = 0; j < height; j++)

printf("(%6.3f, %6.3f) ", networkNodes[i][j].value, networkNodes[i][j].delta);

printf("\n");

}

printf("\n========================\n\n");

}

void BackpropNetwork::displayBiasWeight() {

printf("Display biasWeights:\n\n");

for (int i = 0; i < hiddenLayerNum; i++) {

for (int j = 0; j < hiddenLayerHeight; j++)

printf("(%6.3f, %6.3f) ", biasWeights[i][j].weight, biasWeights[i][j].oldWeight);

printf("\n");

}

for (int i = 0; i < outputNum; i++)

printf("(%6.3f, %6.3f) ", biasWeights[hiddenLayerNum][i].weight, biasWeights[hiddenLayerNum][i].oldWeight);

printf("\n========================\n\n");

}

void BackpropNetwork::displayNodeWeight() {

printf("Display nodeWeights:\n\n");

for (int i = 0; i < hiddenLayerNum + 1; i++) {

printf("Display weights in layer %d:\n", i);

int fromNodeNum = hiddenLayerHeight;

if (i == 0) fromNodeNum = inputNum;

for (int j = 0; j < fromNodeNum; j++) {

int toNodeNum = hiddenLayerHeight;

if (i == hiddenLayerNum) toNodeNum = outputNum;

for (int k = 0; k < toNodeNum; k++)

printf("(%7.3f, %7.3f) ", nodeWeights[i][j][k].weight, nodeWeights[i][j][k].oldWeight);

printf("\n");

}

printf("\n");

}

printf("\n========================\n\n");

};

void BackpropNetwork::displayAccuracy() {

printf("Accuracy: %.1f %% \n\n", accuracy);

}

void BackpropNetwork::displayTestReport() {

printf("\nDisplay test result:\n\n");

for (int i = 0; i < CLASS\_NUM; i++) {

printf("Class %d:\n", i + 1);

for (int j = 0; j < TEST\_SIZE; j++) {

printf("%d\n", testResult[i][j]);

}

printf("\n\n");

}

printf("Accuracy: %.1f %% \n\n", accuracy);

printf("================================\n\n");

}