FFR135, Examples sheet 4

Vitalii Iarko (930505-4711, iarko@student.chalmers.se)

October 23, 2014

Contents

1 a one-dimensional Kohonen network. 2

2 a two-dimensional Kohonen network. 8

3 a combination of competitive learning and supervised learning. 14

A File `task1.cpp`: 20

B File `task2.cpp`: 22

C File `task3.cpp`: 23

D File `KohonenNetwork.h`: 28

E File `KohonenNetwork.cpp`: 31

F File `NeuralNetworkClassifier.h`: 38

G File `NeuralNetworkClassifier.cpp`: 40

H File `Output.h`: 45

I File `PseudoRandomGenerator.h`: 46

J File `PseudoRandomGenerator.cpp`: 47

K File `MT19937RNG.h`: 48

L File `MT19937RNG.cpp`: 49

M File `Dataset.h`: 50

N File `Dataset.cpp`: 51

Additional materials.

It was very convenient to present obtained results in form of an animation, but, unfortunately, it is not easy to show an animation on paper, therefore, I uploaded them to my web page. You can find them here: `http://nrg3.github.io/ANN4.htm` (SHA1 hash of this page (its html code): `35734cf08c8e91ba940313504229fbe6abb1e34b`).

1 a one-dimensional Kohonen network.

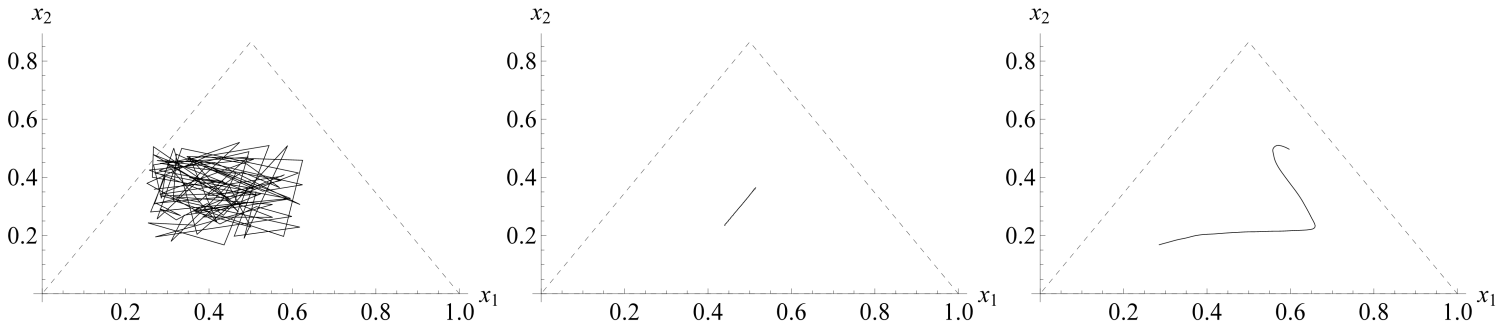
I did 5 experiments for each value of g_0 (Figures 1 - 5 correspond to value 100, 6 - 10 to 10): two with proposed values, but with different seeds (Figures 1, 2, 6, 7), one with bigger quantity of points in the triangle (Figures 3 and 8), one with bigger number of outputs (Figures 4 and 9) and the last one with both of two previous cases (Figures 5 and 10). In addition to this I did 10^7 iterations in convergence phase.

Please note that I did not plot the weights themselves, but a line, that connects them - it is equivalent, but looks neater in case of 100 or more outputs.

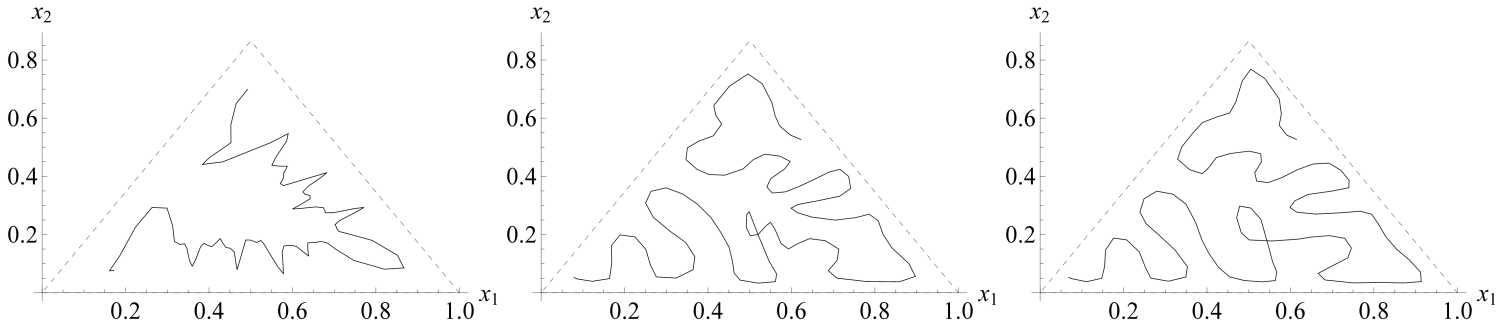
Does the network recognise the triangle? According to obtained experimental results (plotted on Figures 1 - 10 below) it does. Some kind of this structure one can even see after ordering phase and after $5 \cdot 10^4$ iterations of convergence phase it moreover fills the triangle. After this up to 10^7 network does not change dramatically, only obtains more details, becomes smoother.

How do the results with $\sigma_0 = 10$ differ from $\sigma_0 = 100$? For the cases with 100 output units the situation almost the same, except that the network is not that smooth after ordering under $\sigma_0 = 10$. But with 200 output units, one can easily see that the network still has kinks after ordering phase as well as in convergence phase (because here it is harder to remove them). Thus, $\sigma_0 = 10$ is worse in terms of kinks (one can describe this as network converges (removes kinks) slower under $\sigma_0 = 10$).

Which setting works better? We do not like kinks, because they break the idea of mapping - close points in input space should be close in output space. From this points of view, $\sigma_0 = 100$ works better.

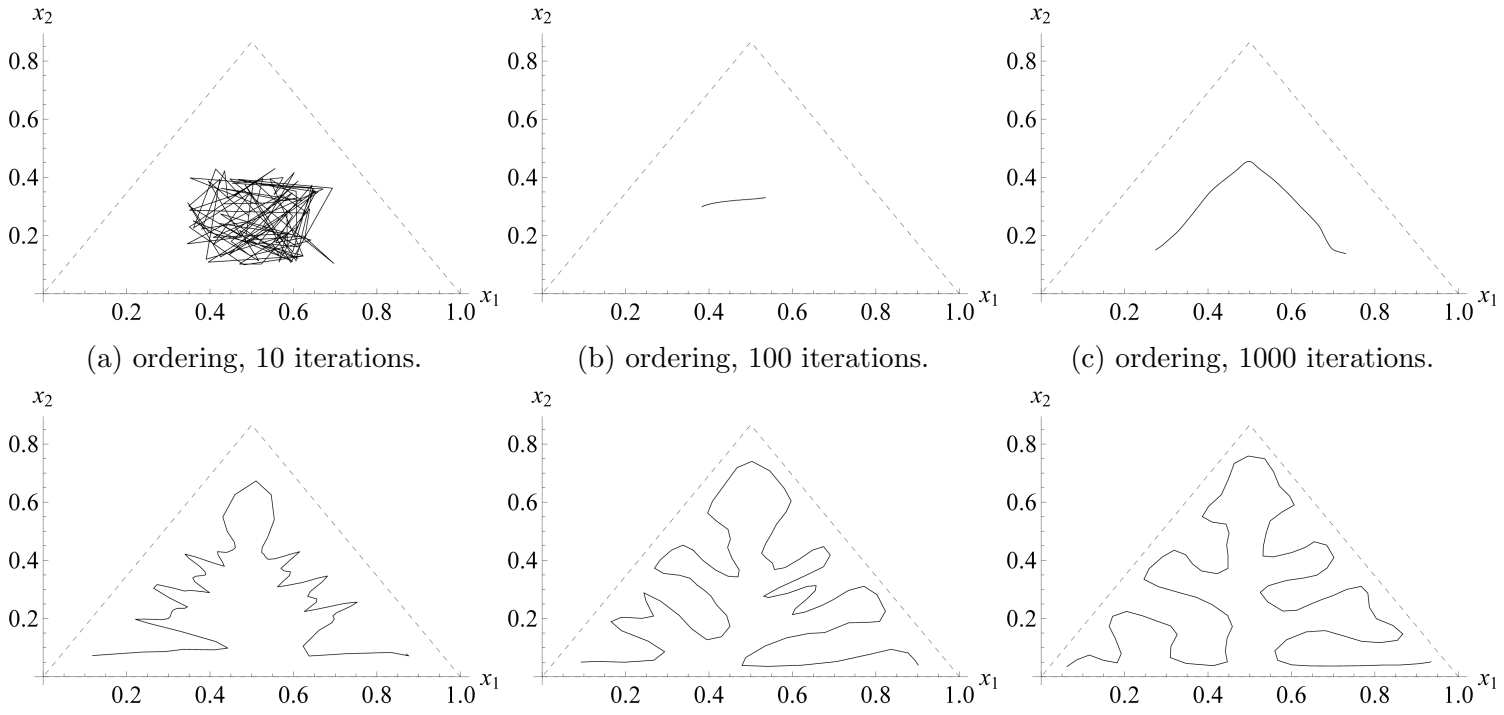


(a) ordering, 10 iterations. (b) ordering, 100 iterations. (c) ordering, 1000 iterations.



(d) convergence, $5 \cdot 10^3$ iterations. (e) convergence, $5 \cdot 10^4$ iterations. (f) convergence, 10^7 iterations.

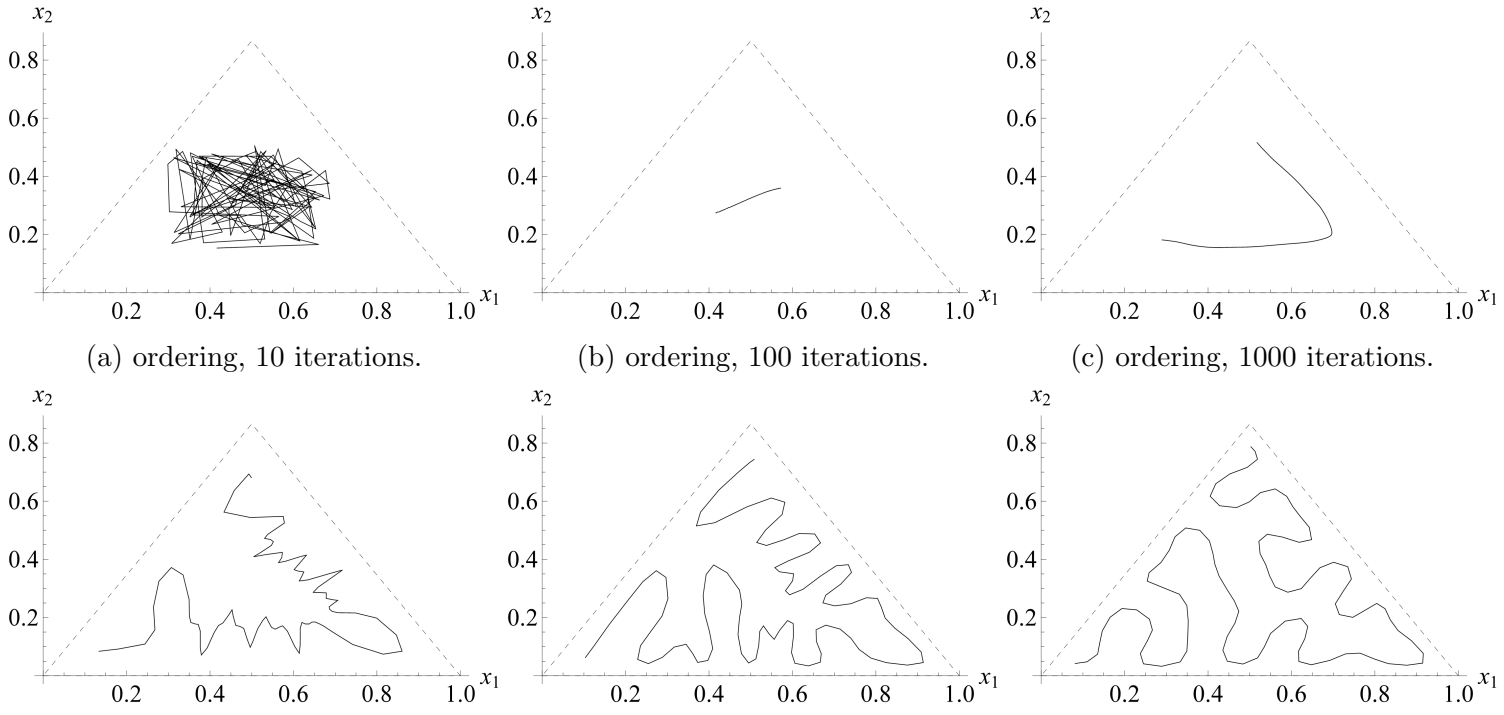
Figure 1: Experiment under $\sigma_0 = 100, seed = 123$ and proposed parameters (number of points in the triangle is 1000, $T_{order} = 1000, \eta_0 = 0.1, \tau_\sigma = T_{order}/\ln[\sigma_0], \sigma_{conv} = 0.9, \eta_{conv} = 0.01, 100$ output nodes).



(a) ordering, 10 iterations. (b) ordering, 100 iterations. (c) ordering, 1000 iterations.

(d) convergence, $5 \cdot 10^3$ iterations. (e) convergence, $5 \cdot 10^4$ iterations. (f) convergence, 10^7 iterations.

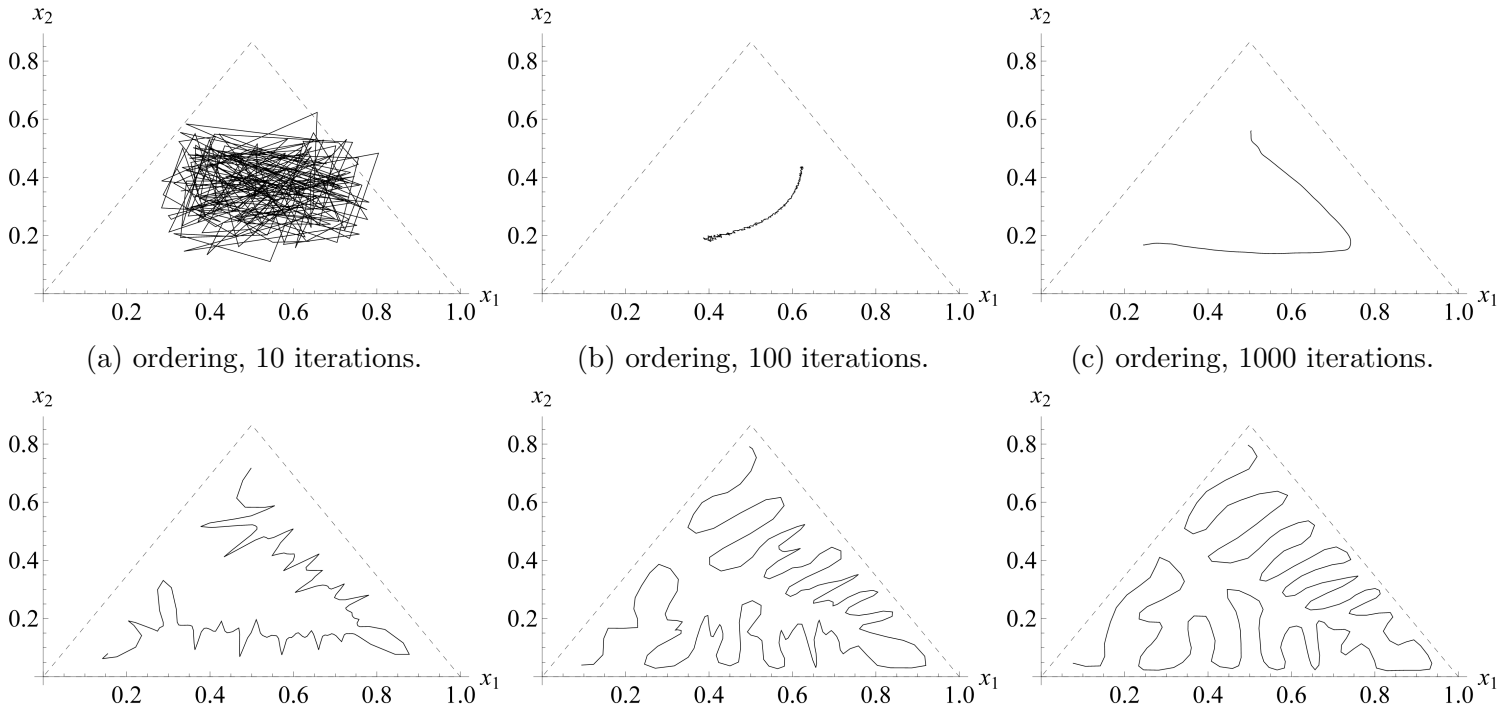
Figure 2: Experiment under $\sigma_0 = 100, seed = 124$ and proposed parameters (number of points in the triangle is 1000, $T_{order} = 1000, \eta_0 = 0.1, \tau_\sigma = T_{order}/\ln[\sigma_0], \sigma_{conv} = 0.9, \eta_{conv} = 0.01, 100$ output nodes).



(a) ordering, 10 iterations. (b) ordering, 100 iterations. (c) ordering, 1000 iterations.

(d) convergence, $5 \cdot 10^3$ iterations. (e) convergence, $5 \cdot 10^4$ iterations. (f) convergence, 10^7 iterations.

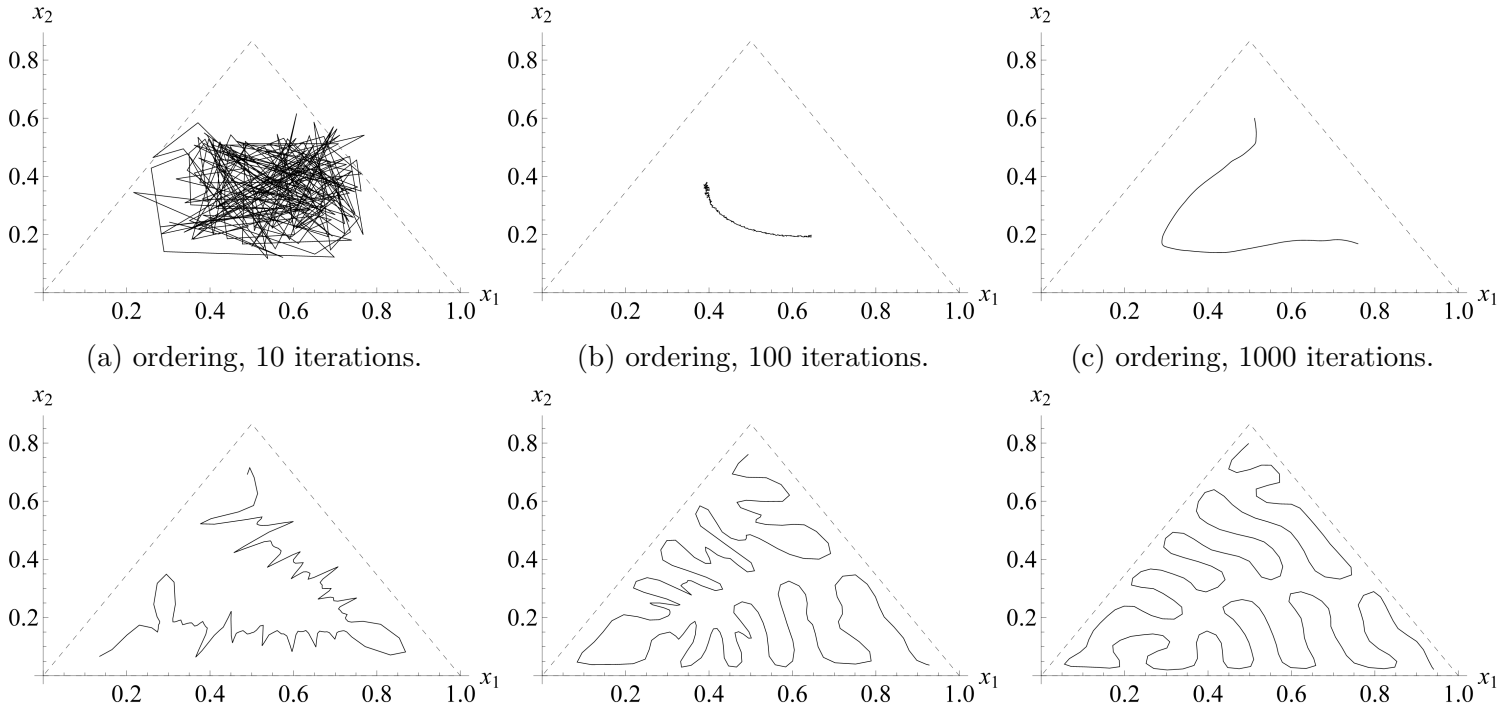
Figure 3: Experiment under $\sigma_0 = 100, seed = 123$, number of points in the triangle 5000 and proposed parameters ($T_{order} = 1000, \eta_0 = 0.1, \tau_\sigma = T_{order}/\ln[\sigma_0], \sigma_{conv} = 0.9, \eta_{conv} = 0.01, 100$ output nodes).



(a) ordering, 10 iterations. (b) ordering, 100 iterations. (c) ordering, 1000 iterations.

(d) convergence, $5 \cdot 10^3$ iterations. (e) convergence, $5 \cdot 10^4$ iterations. (f) convergence, 10^7 iterations.

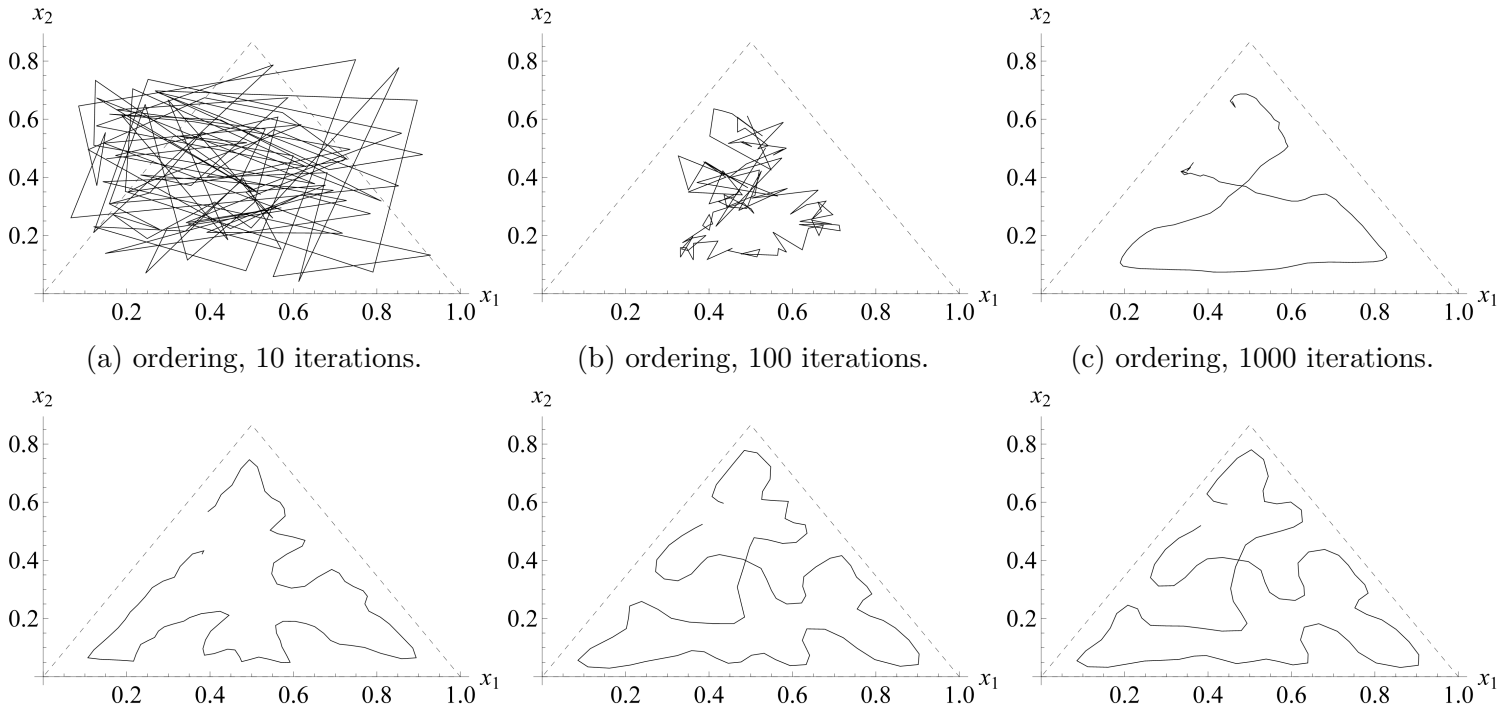
Figure 4: Experiment under $\sigma_0 = 100, seed = 123$, 200 output nodes and proposed parameters (number of points in the triangle is 1000, $T_{order} = 1000, \eta_0 = 0.1, \tau_\sigma = T_{order}/\ln[\sigma_0], \sigma_{conv} = 0.9, \eta_{conv} = 0.01$).



(a) ordering, 10 iterations. (b) ordering, 100 iterations. (c) ordering, 1000 iterations.

(d) convergence, $5 \cdot 10^3$ iterations. (e) convergence, $5 \cdot 10^4$ iterations. (f) convergence, 10^7 iterations.

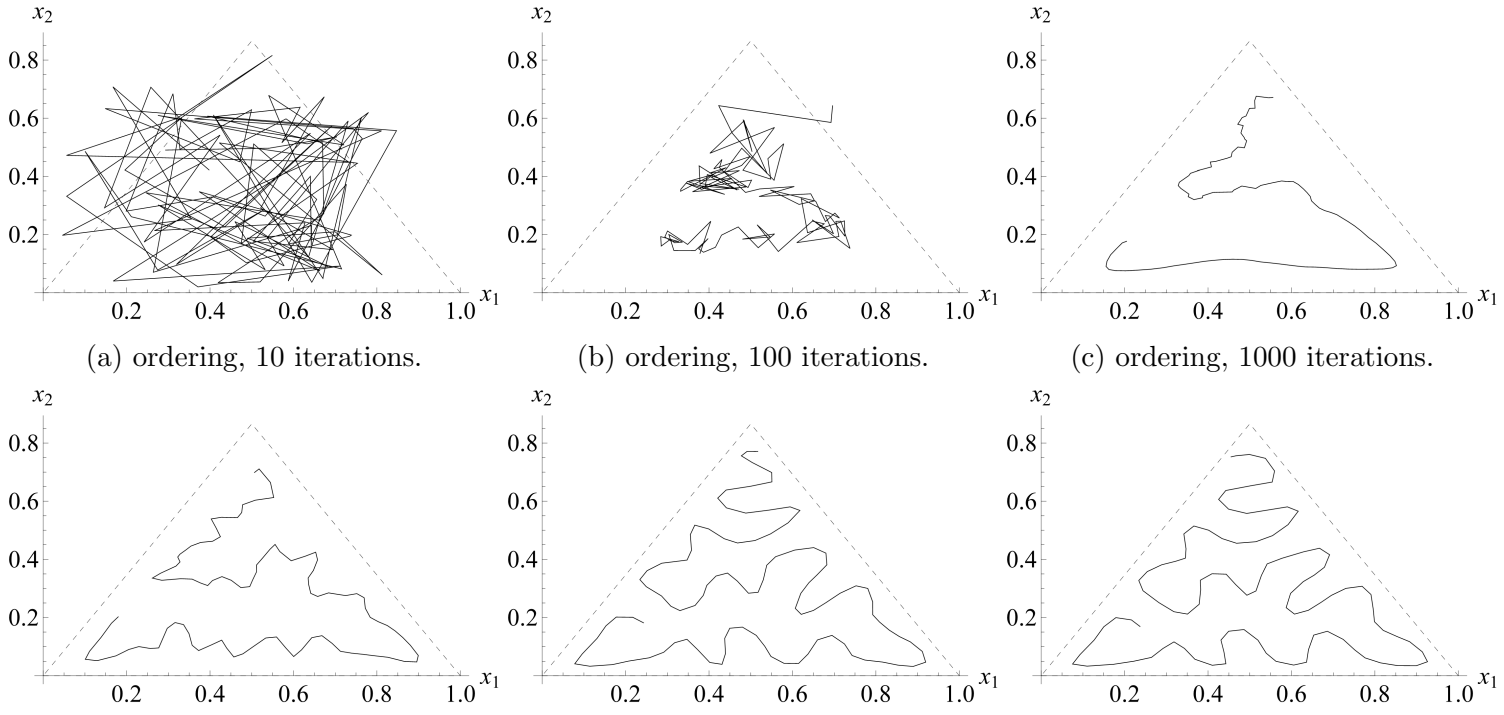
Figure 5: Experiment under $\sigma_0 = 100, seed = 123$, 200 output nodes, number of points in the triangle - 5000 and proposed parameters ($T_{order} = 1000, \eta_0 = 0.1, \tau_\sigma = T_{order}/\ln[\sigma_0], \sigma_{conv} = 0.9, \eta_{conv} = 0.01$).



(a) ordering, 10 iterations. (b) ordering, 100 iterations. (c) ordering, 1000 iterations.

(d) convergence, $5 \cdot 10^3$ iterations. (e) convergence, $5 \cdot 10^4$ iterations. (f) convergence, 10^7 iterations.

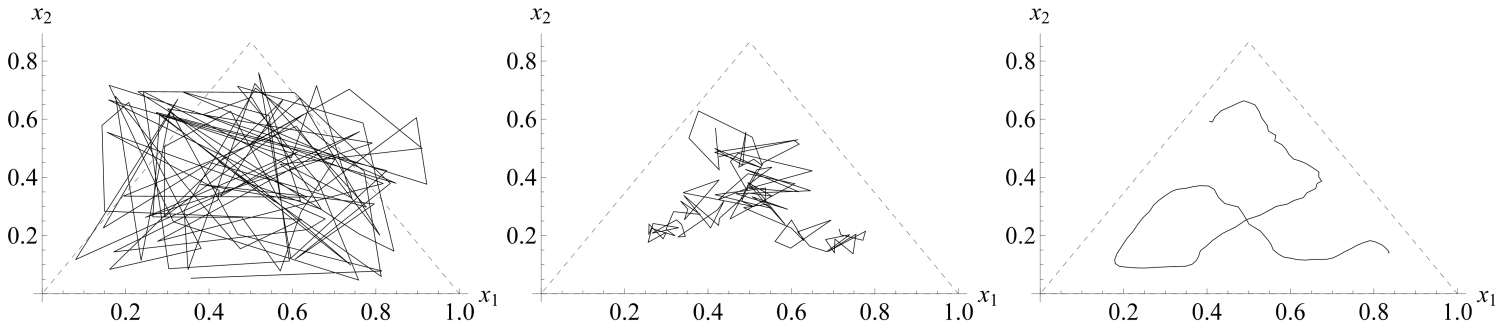
Figure 6: Experiment under $\sigma_0 = 10, seed = 123$ and proposed parameters (number of points in the triangle is 1000, $T_{order} = 1000, \eta_0 = 0.1, \tau_\sigma = T_{order}/\ln[\sigma_0], \sigma_{conv} = 0.9, \eta_{conv} = 0.01, 100$ output nodes).



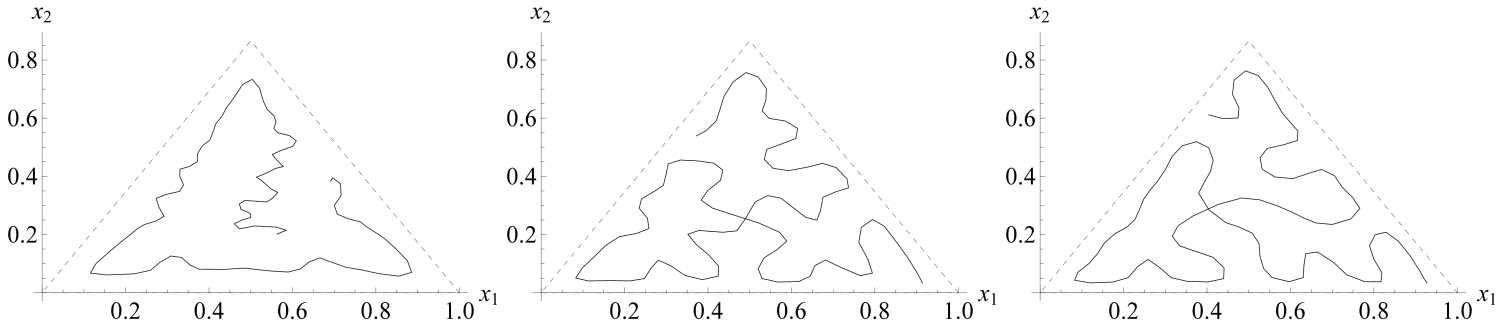
(a) ordering, 10 iterations. (b) ordering, 100 iterations. (c) ordering, 1000 iterations.

(d) convergence, $5 \cdot 10^3$ iterations. (e) convergence, $5 \cdot 10^4$ iterations. (f) convergence, 10^7 iterations.

Figure 7: Experiment under $\sigma_0 = 10, seed = 124$ and proposed parameters (number of points in the triangle is 1000, $T_{order} = 1000, \eta_0 = 0.1, \tau_\sigma = T_{order}/\ln[\sigma_0], \sigma_{conv} = 0.9, \eta_{conv} = 0.01, 100$ output nodes).

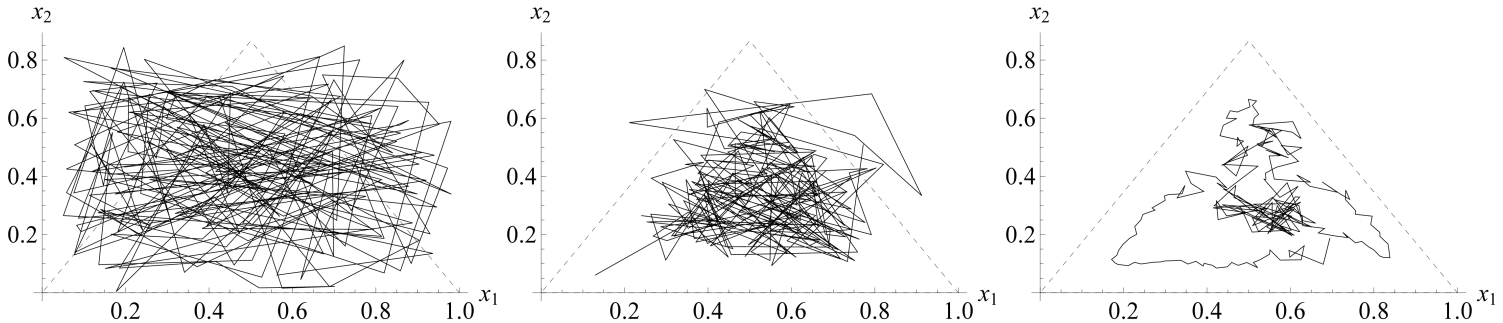


(a) ordering, 10 iterations. (b) ordering, 100 iterations. (c) ordering, 1000 iterations.

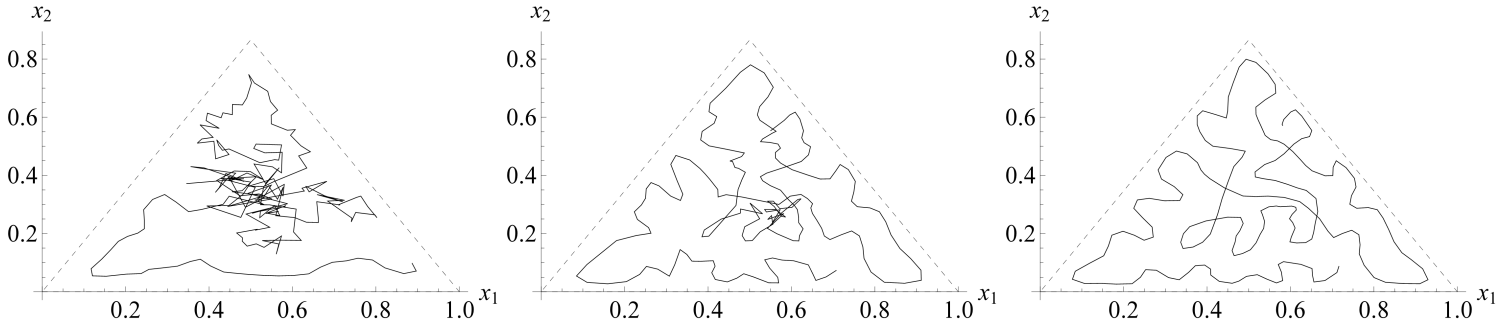


(d) convergence, $5 \cdot 10^3$ iterations. (e) convergence, $5 \cdot 10^4$ iterations. (f) convergence, 10^7 iterations.

Figure 8: Experiment under $\sigma_0 = 10, seed = 123$, number of points in the triangle 5000 and proposed parameters ($T_{order} = 1000, \eta_0 = 0.1, \tau_\sigma = T_{order}/\ln[\sigma_0], \sigma_{conv} = 0.9, \eta_{conv} = 0.01, 100$ output nodes).

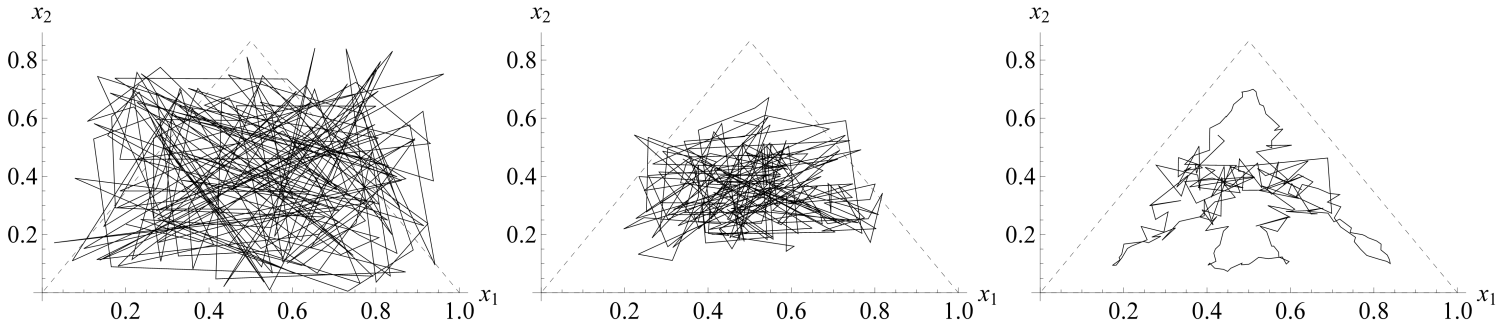


(a) ordering, 10 iterations. (b) ordering, 100 iterations. (c) ordering, 1000 iterations.



(d) convergence, $5 \cdot 10^3$ iterations. (e) convergence, $5 \cdot 10^4$ iterations. (f) convergence, 10^7 iterations.

Figure 9: Experiment under $\sigma_0 = 10, seed = 123$, 200 output nodes and proposed parameters (number of points in the triangle is 1000, $T_{order} = 1000, \eta_0 = 0.1, \tau_\sigma = T_{order}/\ln[\sigma_0], \sigma_{conv} = 0.9, \eta_{conv} = 0.01$).



(a) ordering, 10 iterations. (b) ordering, 100 iterations. (c) ordering, 1000 iterations.

(d) convergence, $5 \cdot 10^3$ iterations. (e) convergence, $5 \cdot 10^4$ iterations. (f) convergence, 10^7 iterations.

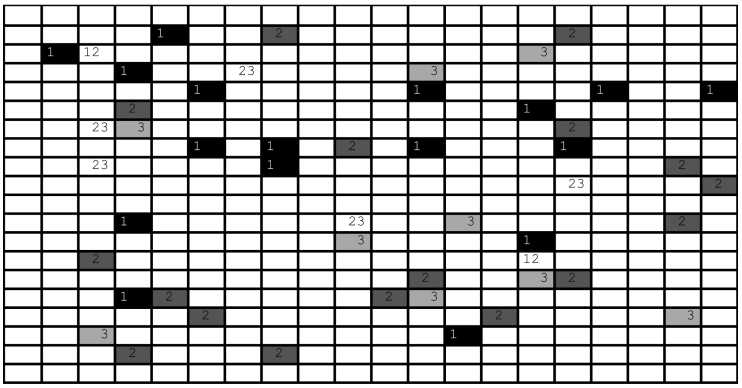
Figure 10: Experiment under $\sigma_0 = 10, seed = 123$, 200 output nodes, number of points in the triangle - 5000 and proposed parameters ($T_{order} = 1000, \eta_0 = 0.1, \tau_\sigma = T_{order}/\ln[\sigma_0], \sigma_{conv} = 0.9, \eta_{conv} = 0.01$).

2 a two-dimensional Kohonen network.

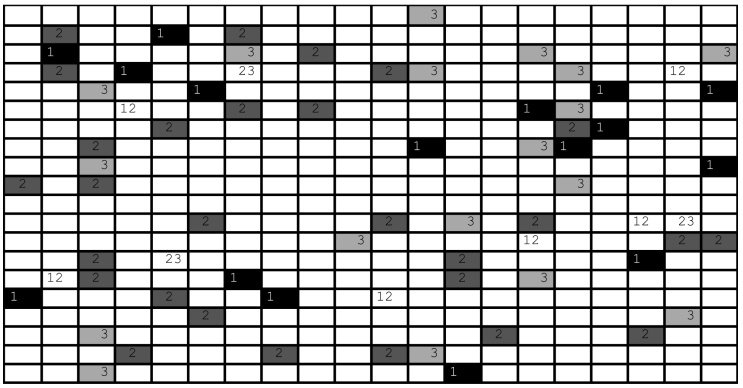
I did 5 runs under proposed parameters (except number of iterations in convergence phase, I did 10^6) with different random seeds. Also I was interested to investigate not only the final classes distribution, but its progress as well.

The distribution of the winning neurons for the different classes are shown on Figures 11 - 15. Each cell of a table corresponds to a neuron. If there is a digit (1, 2 or 3) in a cell, it means that this neuron is a winner for at least one pattern from this class. If a cell consist of 0 or 1 digits, than it has colored background (white - no classes, different levels of gray - 1, 2 or 3-rd class).

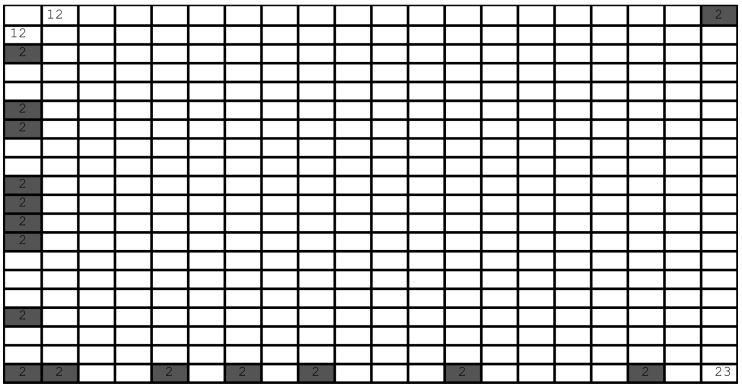
Does the network group wines of the same class together? According to obtained results, it mostly does. Only some few patterns lie in wrong position (but one can find this on all 5 runs, therefore, it seems that they are emissions). Interesting thing that grouping starts to appear even in the ordering phase. Possible way to explain this - each wine class corresponds to some cluster and in ordering phase the network goes to overall average value and classes' clusters located in different directions from this overall average point. Also one can see how network expands and covers the data in convergence phase - at first wining units are located on the boundaries and then they "moves" to the interior as well.



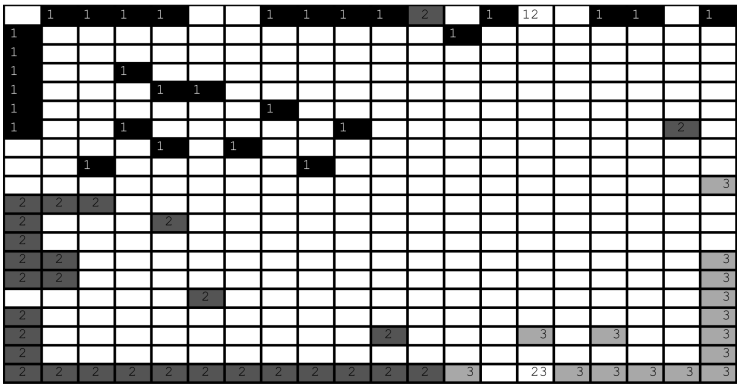
(a) ordering, 1 iteration.



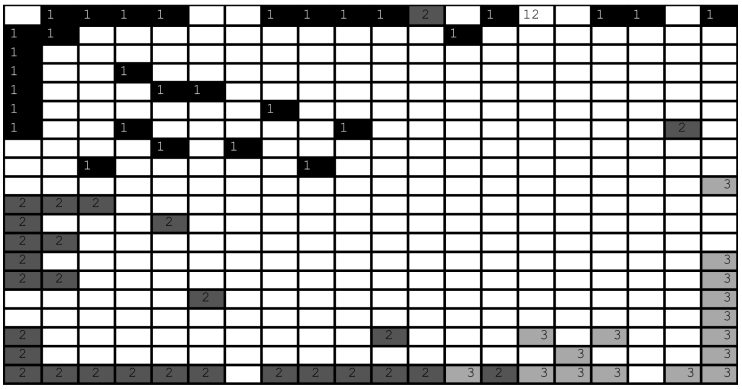
(b) ordering, 10 iterations.



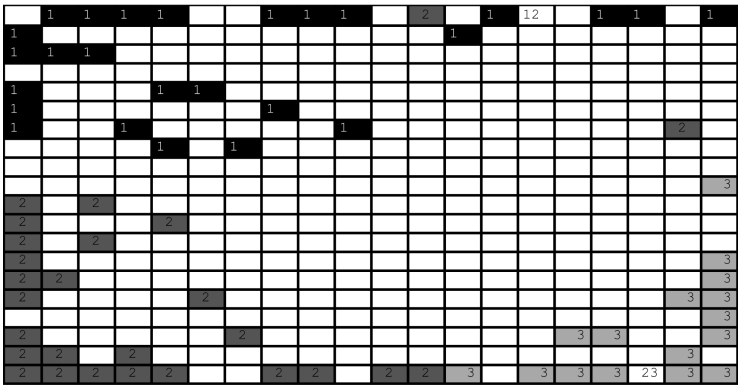
(c) ordering, 100 iterations.



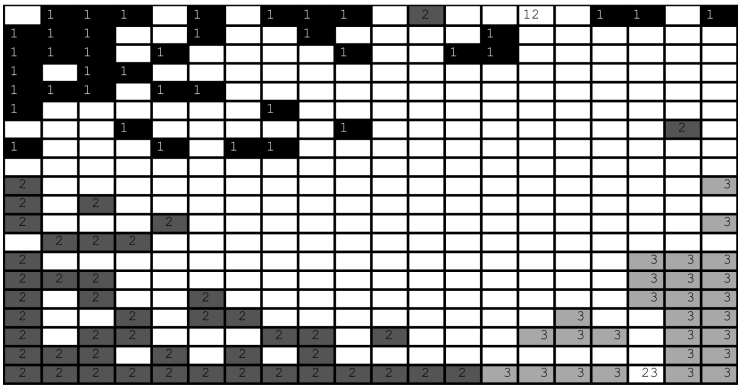
(d) ordering, 1000 iterations.



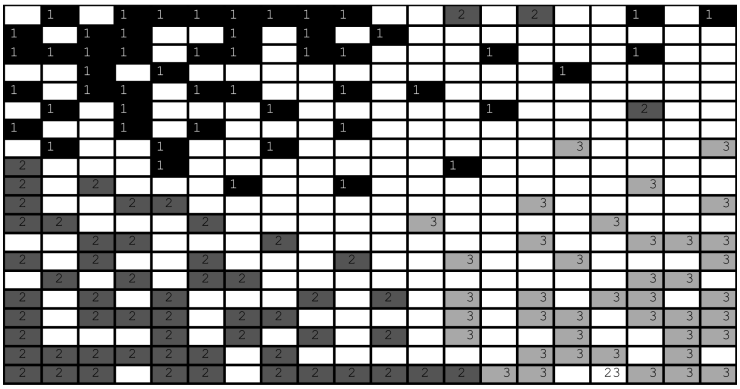
(e) convergence, 200 iterations.



(f) convergence, 2000 iterations.

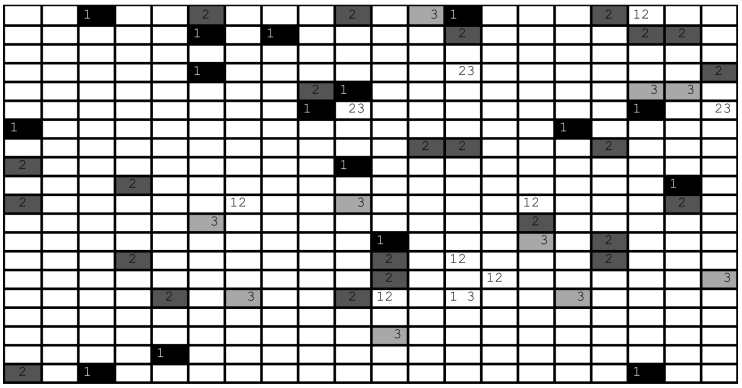


(g) convergence, 20000 iterations.

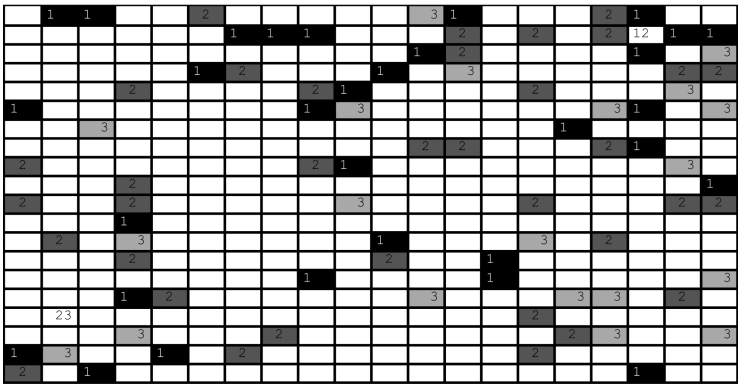


(h) convergence, 1000000 iterations.

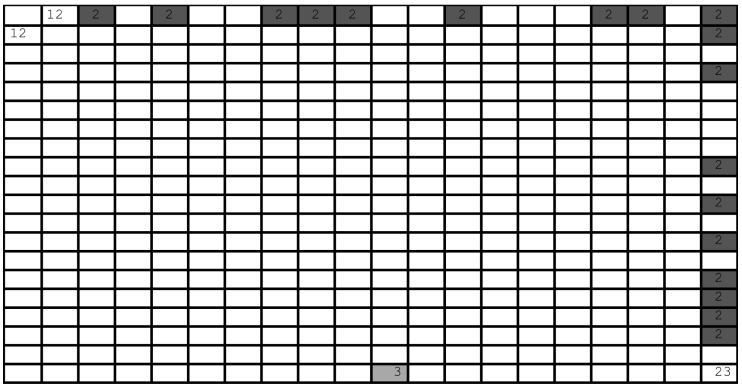
Figure 11: Experiment under $seed = 123$ and proposed parameters (20×20 output units, $T_{order} = 1000, \eta_0 = 0.1, \sigma_0 = 30, \tau_\sigma = T_{order}/\ln[\sigma_0], \sigma_{conv} = 0.9, \eta_{conv} = 0.01$). Each cell corresponds to a neuron. If a cell contains a digit, then it is winning unit for at least one example of this class. If a cell contains only one class, then it is colored in corresponding way (white - no classes, different level of gray - three wine classes).



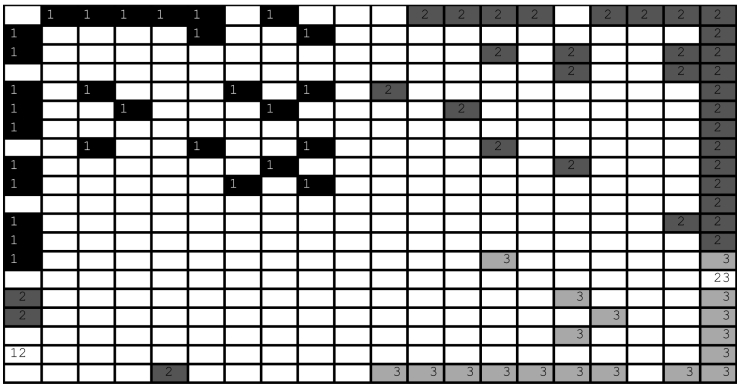
(a) ordering, 1 iteration.



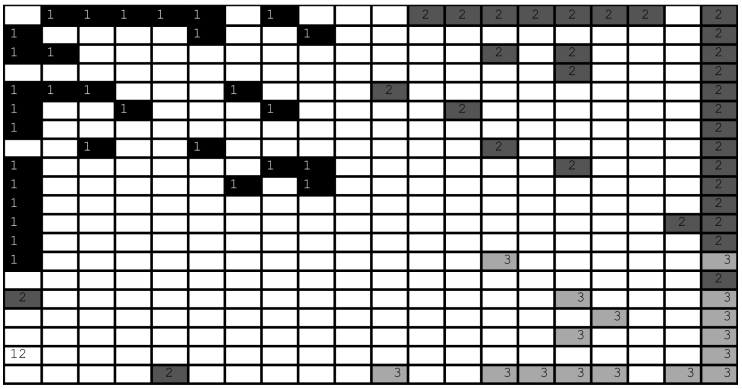
(b) ordering, 10 iterations.



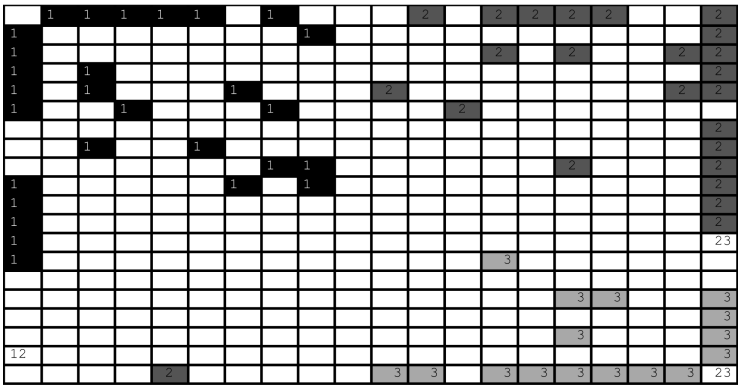
(c) ordering, 100 iterations.



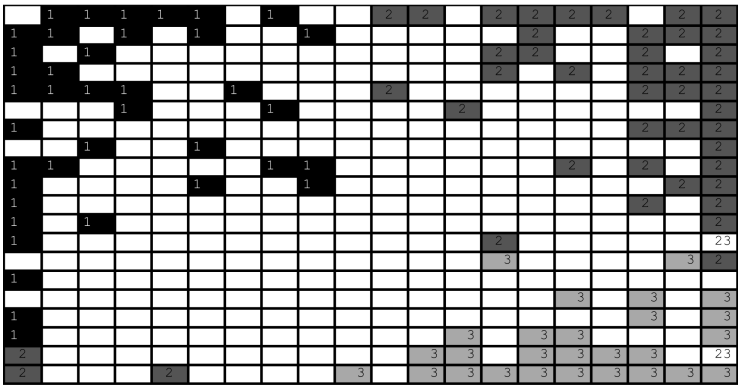
(d) ordering, 1000 iterations.



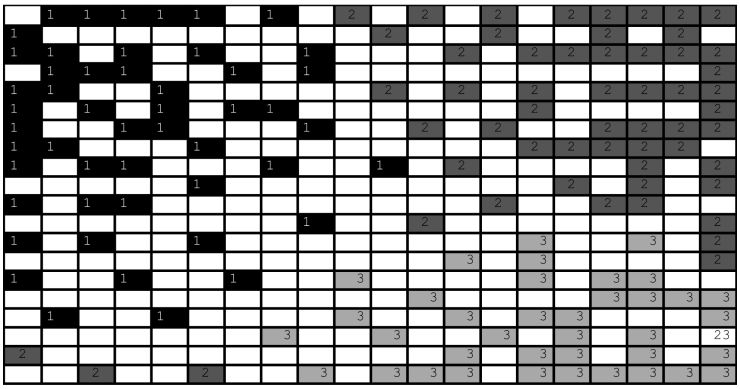
(e) convergence, 200 iterations.



(f) convergence, 2000 iterations.

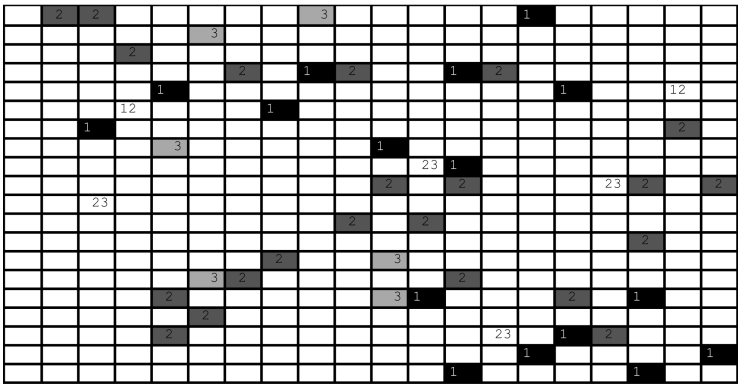


(g) convergence, 20000 iterations.

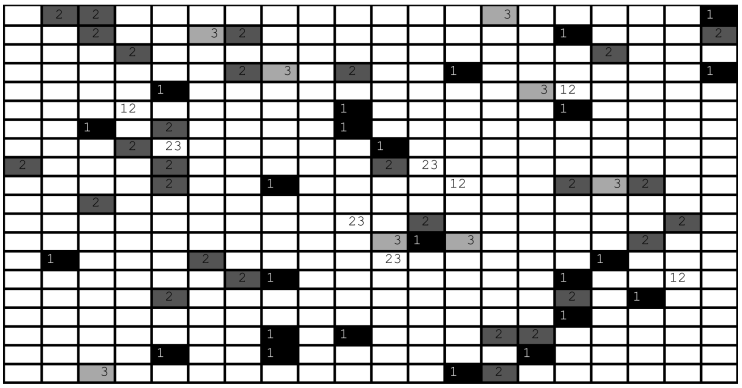


(h) convergence, 1000000 iterations.

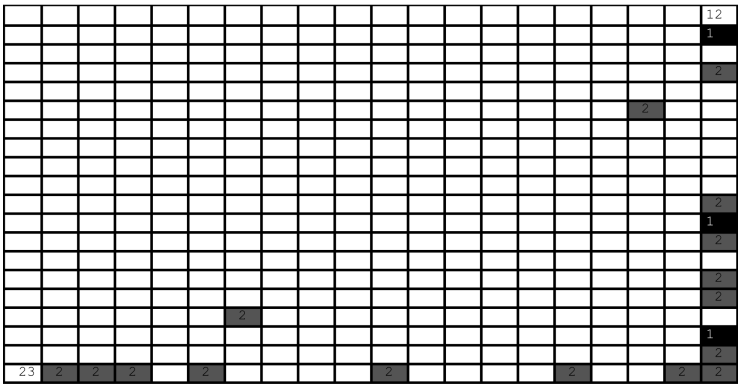
Figure 12: Experiment under $seed = 124$ and proposed parameters (20×20 output units, $T_{order} = 1000, \eta_0 = 0.1, \sigma_0 = 30, \tau_\sigma = T_{order}/\ln[\sigma_0], \sigma_{conv} = 0.9, \eta_{conv} = 0.01$). Each cell corresponds to a neuron. If a cell contains a digit, then it is winning unit for at least one example of this class. If a cell contains only one class, then it is colored in corresponding way (white - no classes, different level of gray - three wine classes).



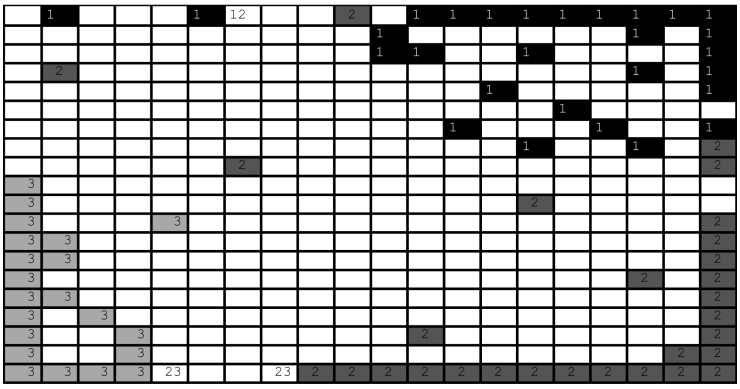
(a) ordering, 1 iteration.



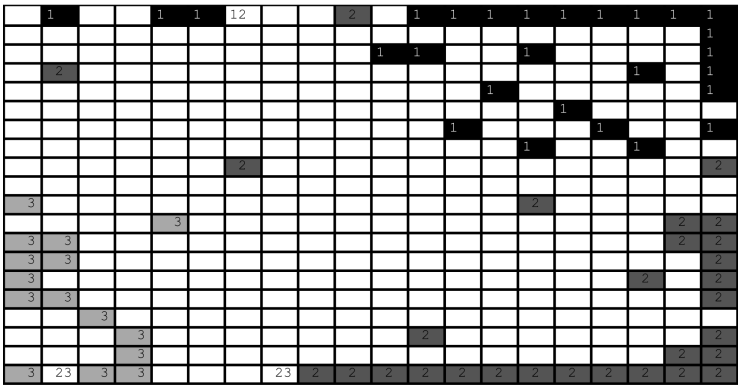
(b) ordering, 10 iterations.



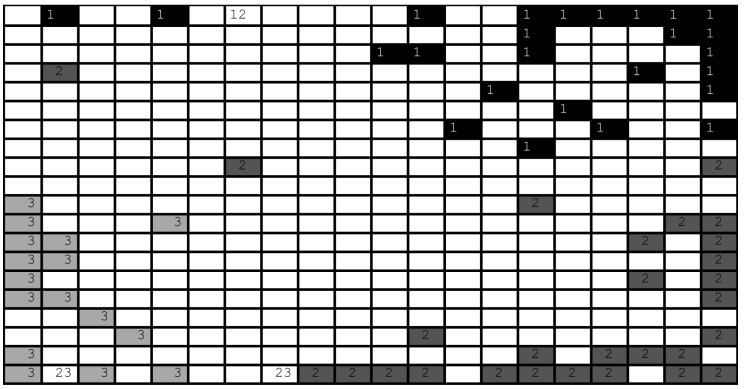
(c) ordering, 100 iterations.



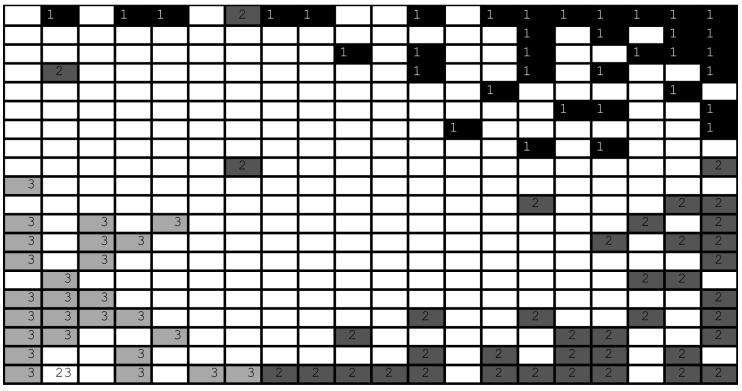
(d) ordering, 1000 iterations.



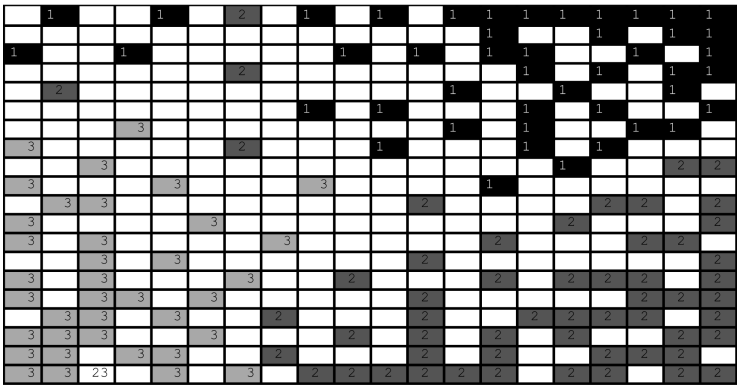
(e) convergence, 200 iterations.



(f) convergence, 2000 iterations.

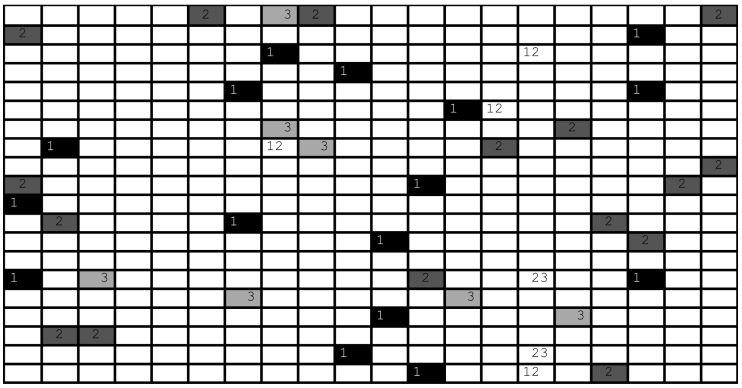


(g) convergence, 20000 iterations.

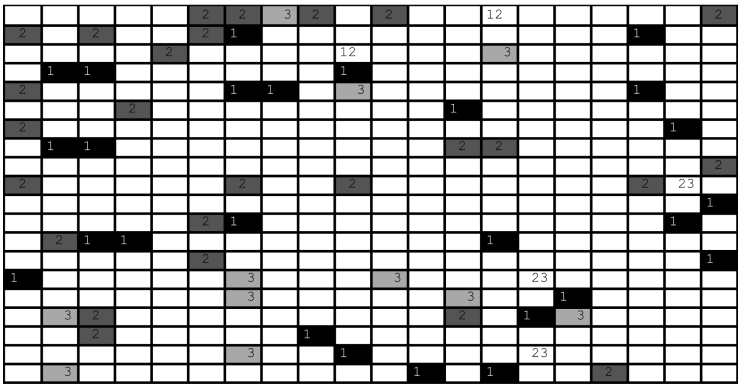


(h) convergence, 1000000 iterations.

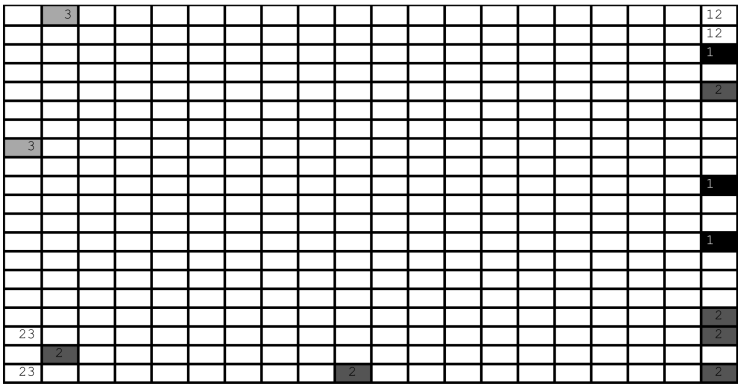
Figure 13: Experiment under $seed = 125$ and proposed parameters (20×20 output units, $T_{order} = 1000, \eta_0 = 0.1, \sigma_0 = 30, \tau_\sigma = T_{order}/\ln[\sigma_0], \sigma_{conv} = 0.9, \eta_{conv} = 0.01$). Each cell corresponds to a neuron. If a cell contains a digit, then it is winning unit for at least one example of this class. If a cell contains only one class, then it is colored in corresponding way (white - no classes, different level of gray - three wine classes).



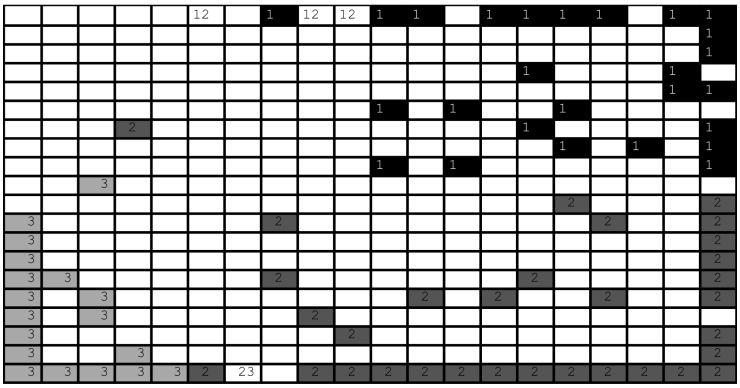
(a) ordering, 1 iteration.



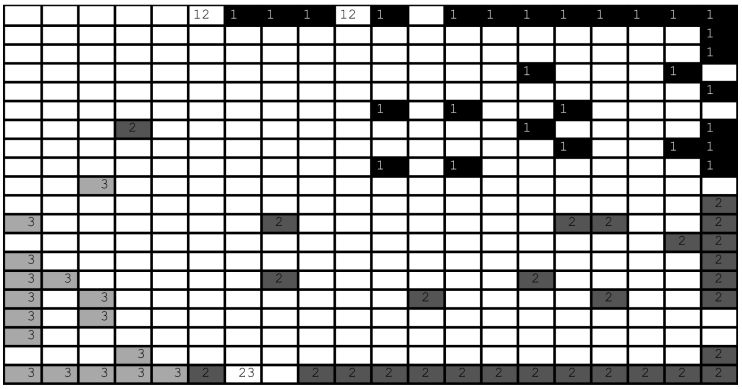
(b) ordering, 10 iterations.



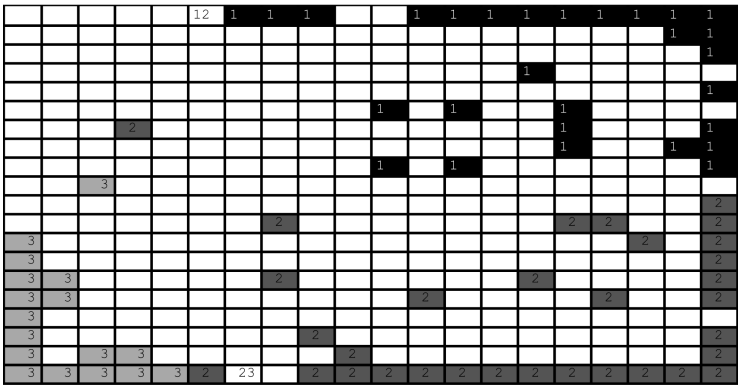
(c) ordering, 100 iterations.



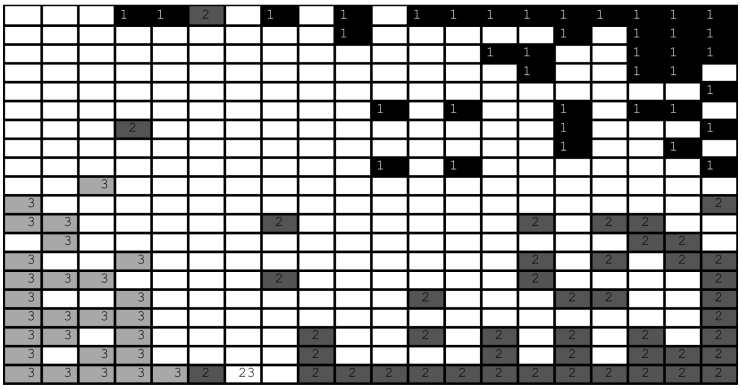
(d) ordering, 1000 iterations.



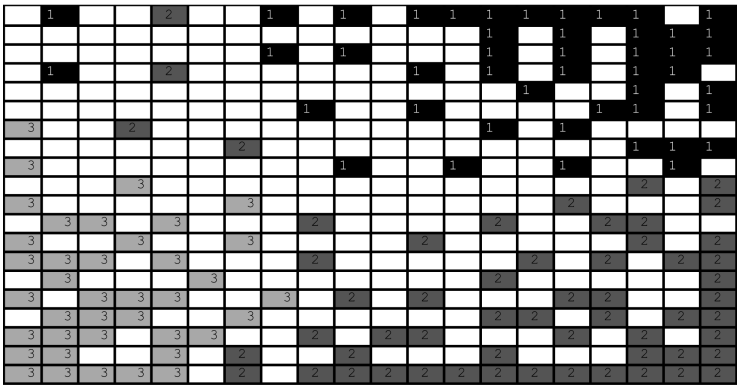
(e) convergence, 200 iterations.



(f) convergence, 2000 iterations.

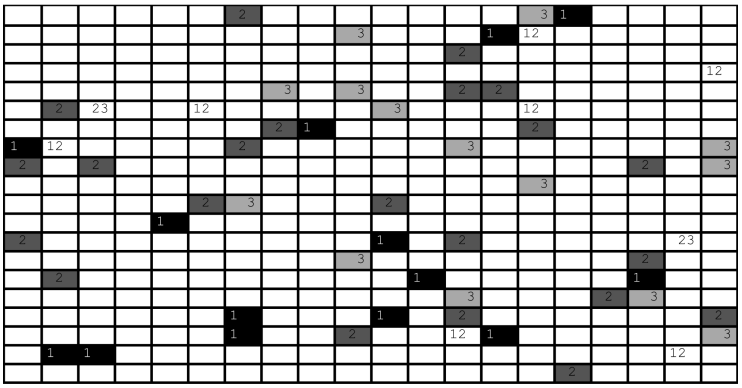


(g) convergence, 20000 iterations.

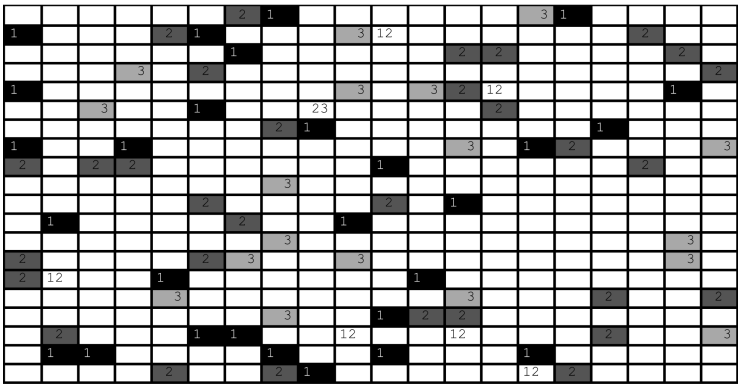


(h) convergence, 1000000 iterations.

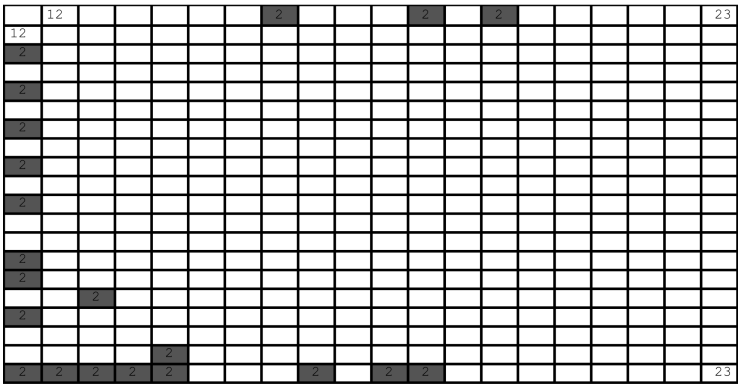
Figure 14: Experiment under $seed = 126$ and proposed parameters (20×20 output units, $T_{order} = 1000, \eta_0 = 0.1, \sigma_0 = 30, \tau_\sigma = T_{order}/\ln[\sigma_0], \sigma_{conv} = 0.9, \eta_{conv} = 0.01$). Each cell corresponds to a neuron. If a cell contains a digit, then it is winning unit for at least one example of this class. If a cell contains only one class, then it is colored in corresponding way (white - no classes, different level of gray - three wine classes).



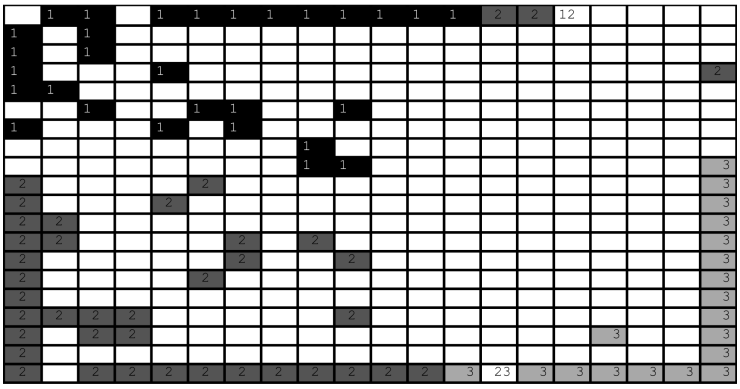
(a) ordering, 1 iteration.



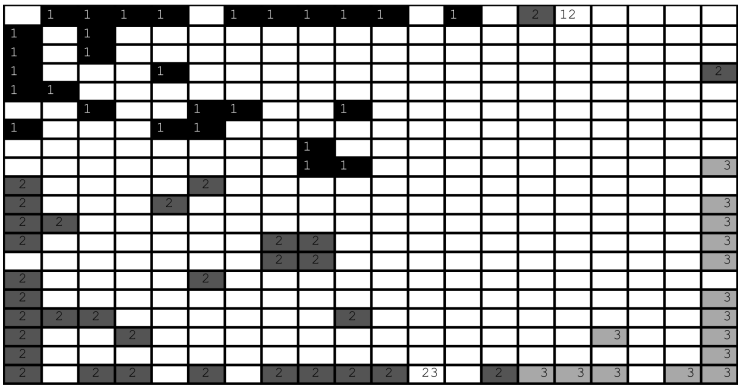
(b) ordering, 10 iterations.



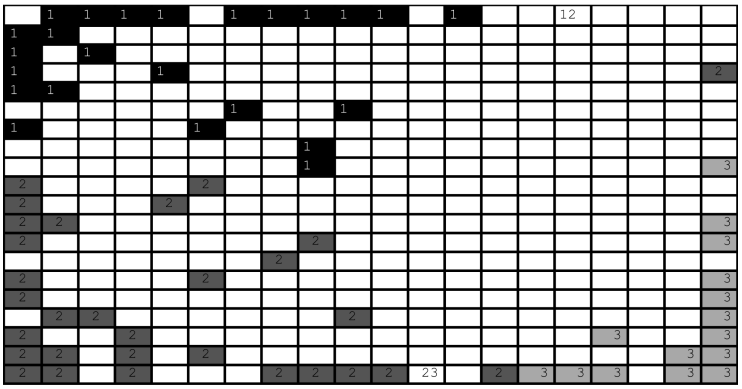
(c) ordering, 100 iterations.



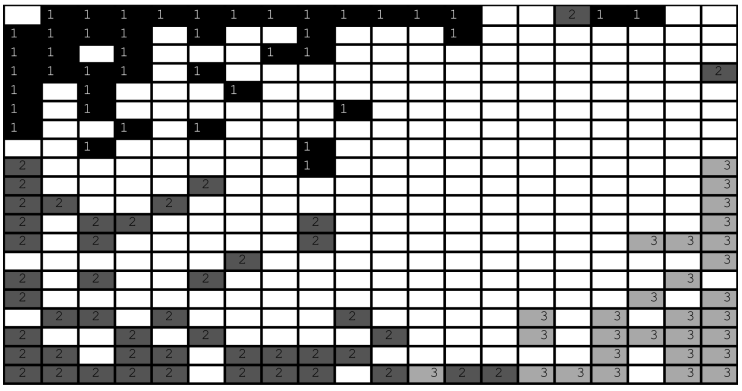
(d) ordering, 1000 iterations.



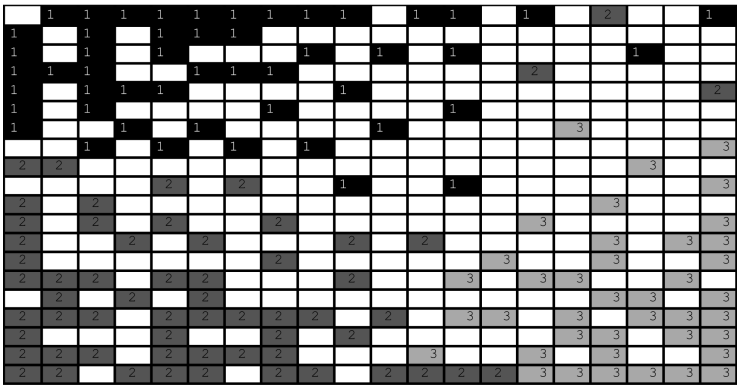
(e) convergence, 200 iterations.



(f) convergence, 2000 iterations.



(g) convergence, 20000 iterations.



(h) convergence, 1000000 iterations.

Figure 15: Experiment under $seed = 127$ and proposed parameters (20×20 output units, $T_{order} = 1000, \eta_0 = 0.1, \sigma_0 = 30, \tau_\sigma = T_{order}/\ln[\sigma_0], \sigma_{conv} = 0.9, \eta_{conv} = 0.01$). Each cell corresponds to a neuron. If a cell contains a digit, then it is winning unit for at least one example of this class. If a cell contains only one class, then it is colored in corresponding way (white - no classes, different level of gray - three wine classes).

3 a combination of competitive learning and supervised learning.

I would like to start with my implementation description. Bernard said that the statement may be ambiguous, but, nevertheless, we can use any meaning - the only we need to do is to describe precisely what we do.

For competitive learning I used Kohonen network with 10 Gaussian nodes with suggested Gaussian activation function. What about standard deviations s_j I used different approaches and I will include the description of this further. Please, note that I call process of obtaining dataset with Kohonen network outputs $g_j(x)$ from raw dataset as "mapping".

For back-propagation I used my implementation from examples sheet 3. As was said in statement I assign to each of the wine classes one of the suggested three-dimensional target outputs. In order to obtain pattern class after classification I take signum function from each of outputs values and then compare obtained vector with target outputs. If no one is equal to it, I understand this as denial to classify, i.e. error. In obtained result I understand classification error as a ratio of incorrectly classified patterns number in the dataset to its size. Also I use normalized energy value, i.e. divided by dataset size, in order to compare its values for datasets with different sizes. In order to make result less biased I did a few runs with different seeds (i.e. different partitions to training and validation set and different pseudo random sequence while classifier training) and then used the average values as a final result. Please, note that by this I mean I did a few runs of classification for only one mapping result. Before working with the classifier I always normalize a dataset to zero mean and unity variance. After mapping I normalize each row (i.e. $g_j(x)$ for given pattern x) to have sum equal to 1.

Let's start with the suggested parameter values and dynamic standard deviations (we will call this case as unbounded). The distribution of the winning neurons for the different wine classes are shown on Figure 16. One can see that classes are distributed, but not fully apart. Normalized energy and classification errors curves for one run are shown on Figure 19. As we expected there is no overfitting (the classifier is too simple). Overall I did 1000 runs under seeds 123 to 1122. Please, note again that I did this runs only for backpropagation, i.e. I used the same mapping for each of the run. As a result of an experiment I took minimum errors rate on validation set through all training process. Average of this values is 0.048 (i.e. 4.8% of errors) with standard deviation 0.023. This means that the network recognises wine classes pretty good.

But Bernard said that variances s_j can become very small. This may lead to numerical instabilities, because we calculate $\exp(\frac{\cdots}{s})$, therefore, let's try to set lower bound 1 for variances. The distribution and curves examples are shown on Figures 17 and 20 correspondingly. As one can see distribution becomes better as well as the average for 1000 runs - 0.025 with standard deviation 0.017.

Finally Bernard said, that we might use fixed variances (i.e. we will just minimize Euclidian distances). Let's try this case with all variances equal to 1. Figures 18 and 21 correspond to this case. Histogram becomes almost perfect, average errors rate is 0.0151 with standard deviation 0.0151.

Ok, let's try to do it without mapping at all, just with raw data (with class values converted to three-dimensional vector as was suggested in the statement). The same procedure gives us 0.00205 in average (standard deviation 0.00588). It is much better, than with any mapping and it seems strange, we would try to explain this as a feature of the dataset, but let's be honest with ourselves - this criterion with best validation score through learning is not that good (i.e. biased). We do not have overfiting, therefore, we do not know when to stop learning, this means that it is not that easy to stop at a moment with best validation score. Maybe, we just investigated unrealistic case. Let's do it in another way - imagine that we will stop somewhen after getting into this steady state (we do not have overfiting). It leads us to another evaluation criterion - to take average validation score on last 100 iterations (i.e. from 4900 to 5000 in our case). Thus, previous criterion corresponds to an average validation score if we stop at the best moment (which is unlikely in real life), current criterion corresponds to an average validation score if we stop somewhere at the end of learning (more realistic situation). Numeric results are presented in Table 1. As you can see mapping with fixed to 1 variances gives 4% errors in this case, when classification on raw data - 8 %. Thus, with more real evaluation criterion we can see that mapping allows us to obtain better classification quality.

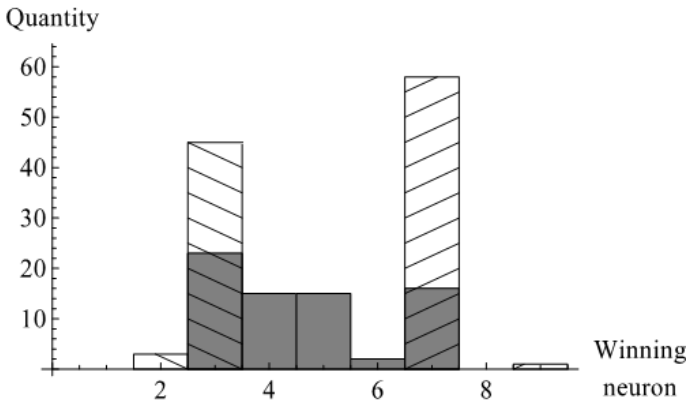


Figure 16: The distribution of the winning neurons for different wine classes. The Kohonen network was trained under suggested parameters (i.e. with unbounded variances). Filled columns correspond to the second class, the columns with hatchings - the first (on left) and the third classes. Note, that columns are not stacked, they overlap.

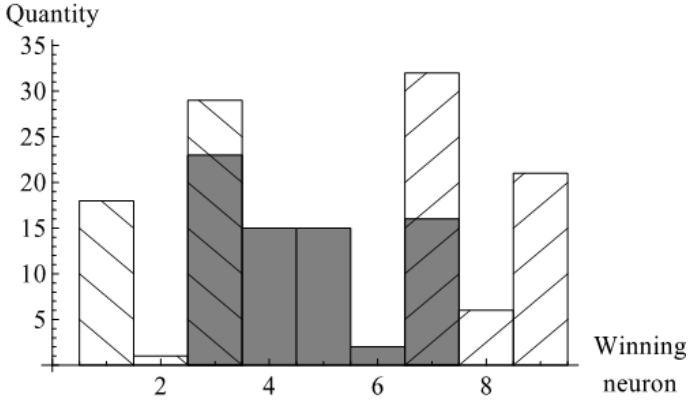


Figure 17: The distribution of the winning neurons for the different wine classes. The Kohonen network was trained under suggested parameters with lower bound 1 for the variances. Filled columns correspond to the second class, the columns with hatchings - the first (on left) and the third classes. Note, that columns are not stacked, they overlap.

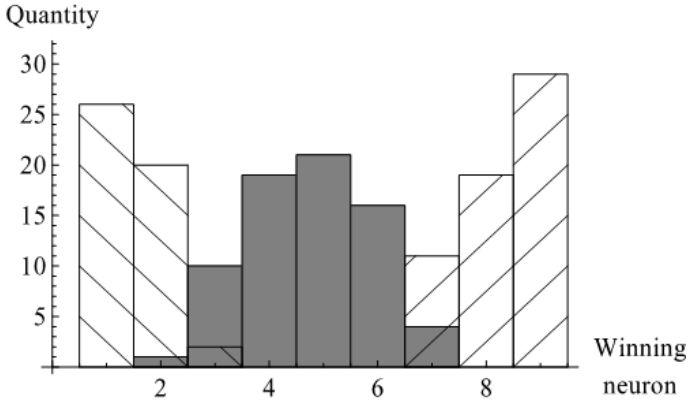
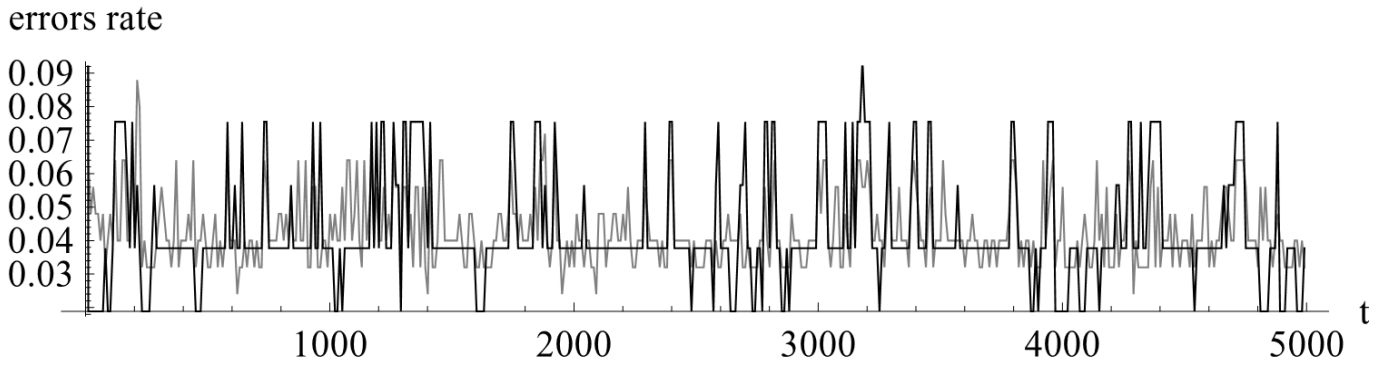
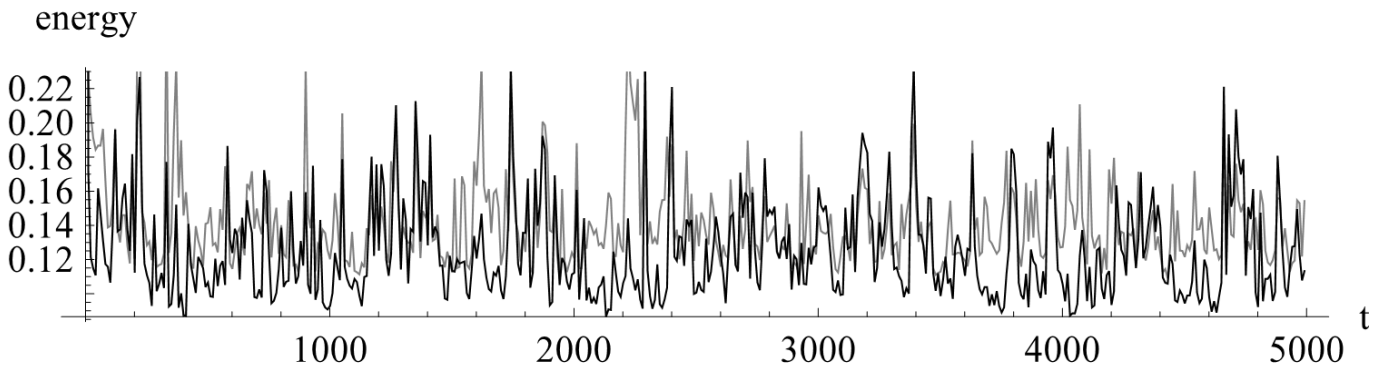


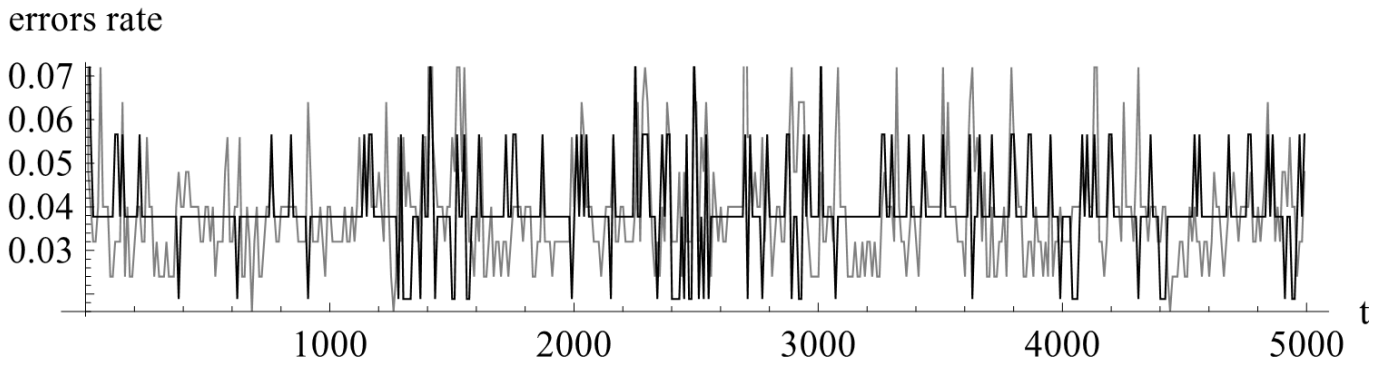
Figure 18: The distribution of the winning neurons for the different wine classes. The Kohonen network was trained under suggested parameters with fixed to 1 variances. Filled columns correspond to the second class, the columns with hatchings - the first (on left) and the third classes. Note, that columns are not stacked, they overlap.



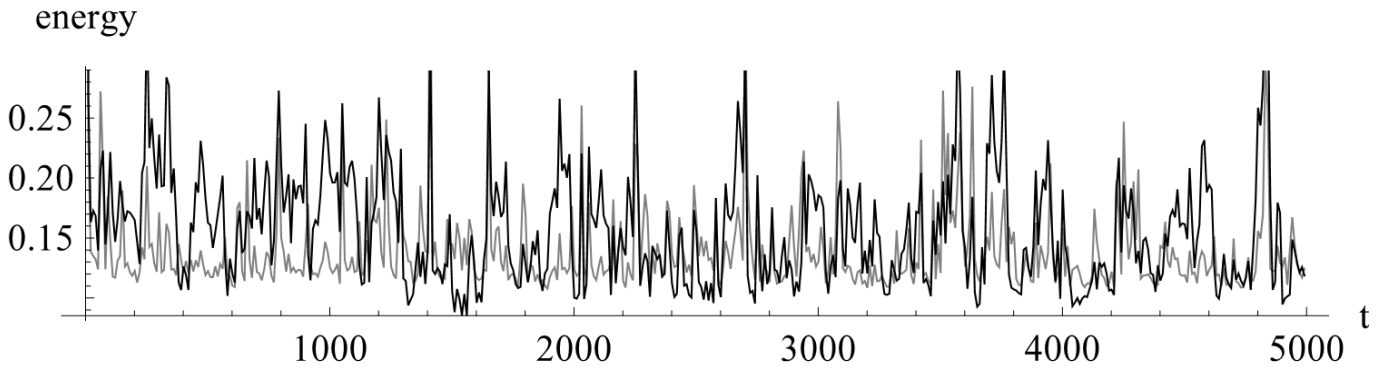
(a) classification error, $seed = 123$



(b) energy, $seed = 123$

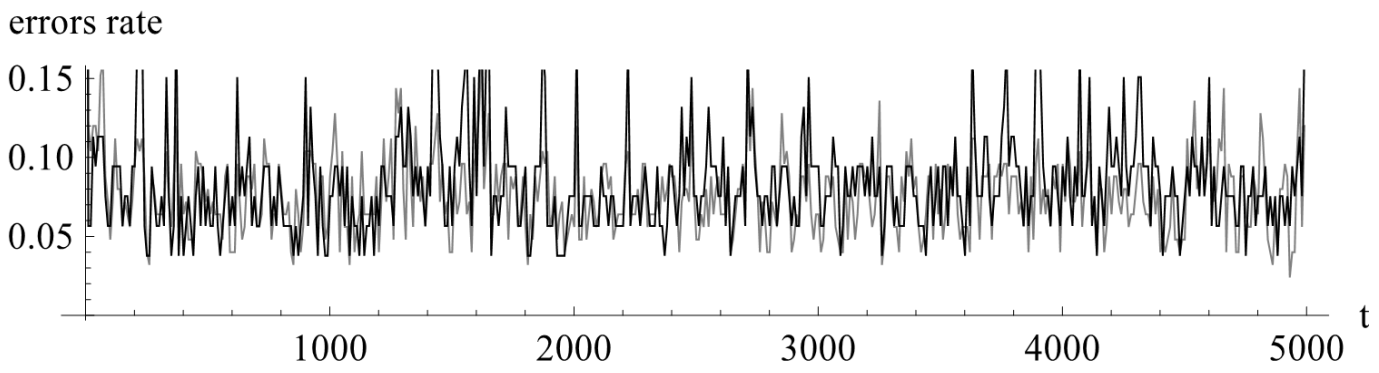


(c) classification error, $seed = 124$

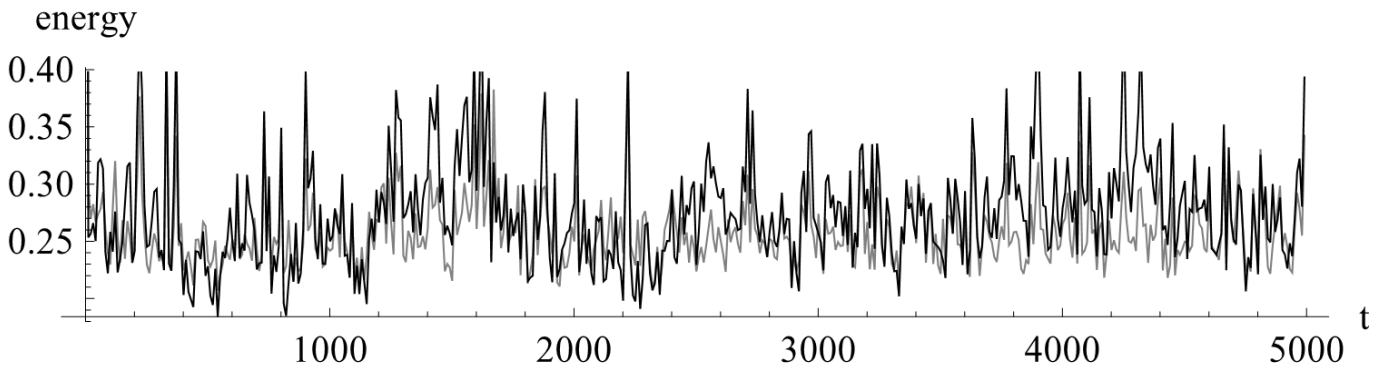


(d) energy, $seed = 124$

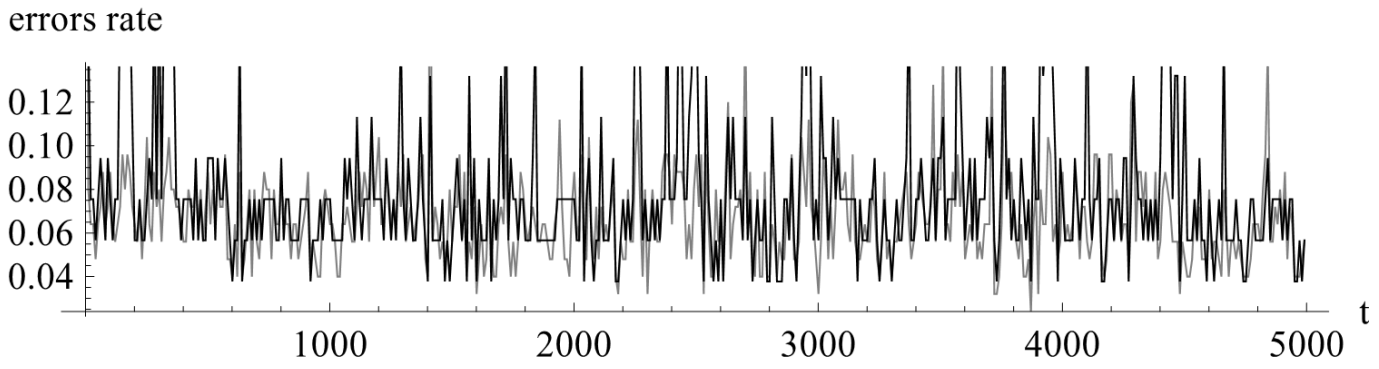
Figure 19: Examples of energy (normalized, i.e. divided by dataset size in order to be comparable for datasets with different sizes) and classification error curves for the case under suggested parameters (i.e. on the dataset after mapping with unbounded variances).These classification experiments were held under $seed = 123, 124$, learning rate is 0.1, 5000 iterations. Please note, that only 500 points per curve are plotted. Gray curve corresponds to results for training set, black - validation.



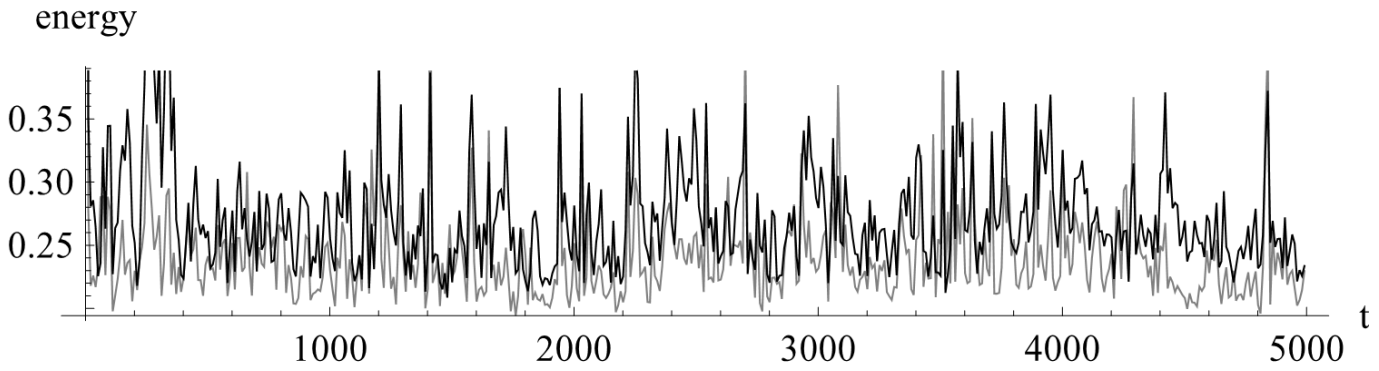
(a) classification error, $seed = 123$



(b) energy, $seed = 123$

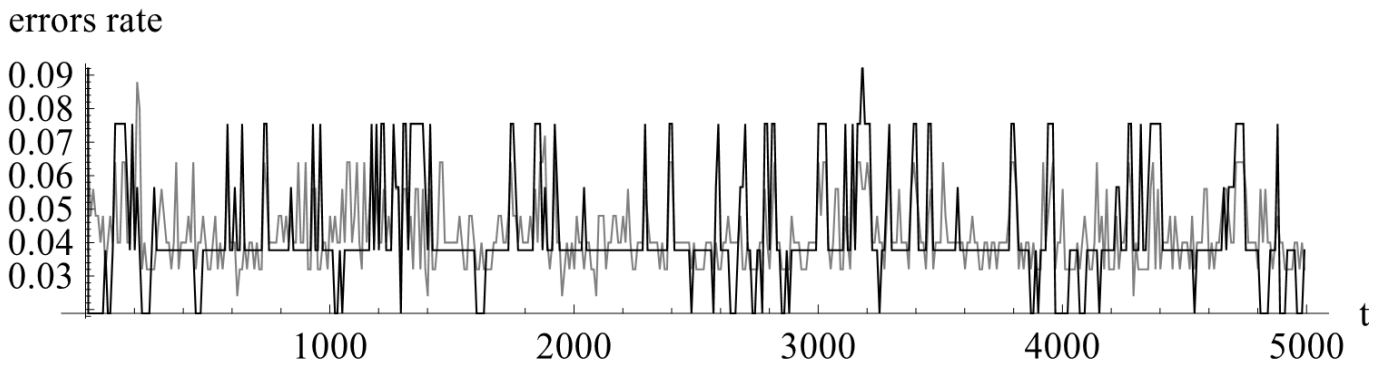


(c) classification error, $seed = 124$

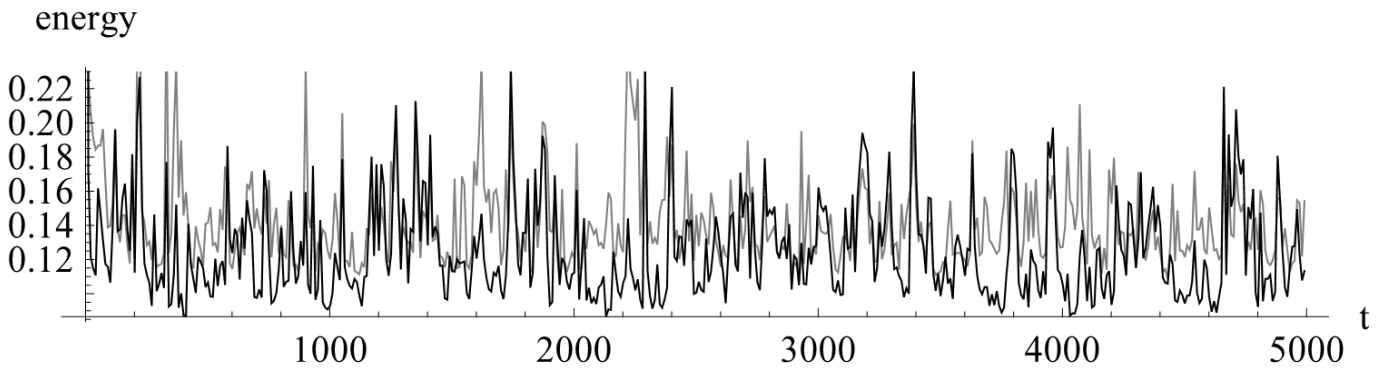


(d) energy, $seed = 124$

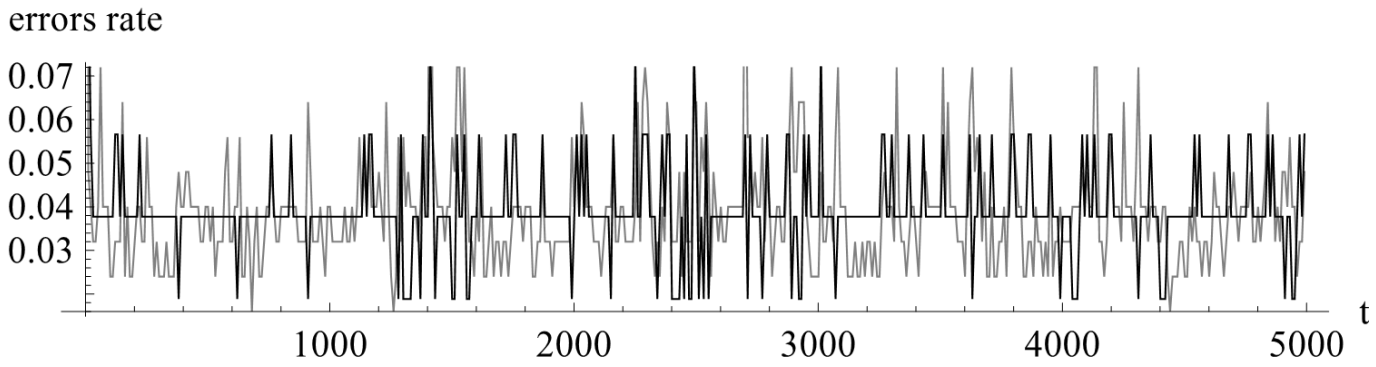
Figure 20: An examples of energy (normalized, i.e. divided by dataset size in order to be comparable for datasets with different sizes) and classification error curves for the case under suggested parameters and on the dataset, obtained after mapping with lower bound 1 for the variances. These classification experiments were held under $seed = 123, 124$, learning rate is 0.1, 5000 iterations. Please note, that only 500 points per curve are plotted. Gray curve corresponds to results for training set, black - validation.



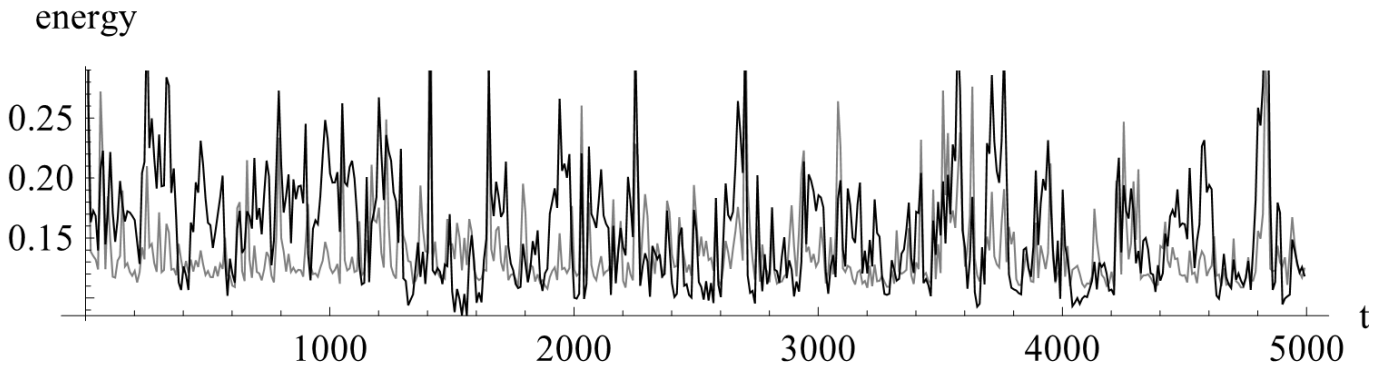
(a) classification error, $seed = 123$



(b) energy, $seed = 123$



(c) classification error, $seed = 124$



(d) energy, $seed = 124$

Figure 21: An examples of energy (normalized, i.e. divided by dataset size in order to be comparable for datasets with different sizes) and classification error curves for the case under suggested parameters and on the dataset, obtained after mapping with fixed to 1 variances. These classification experiments were held under $seed = 123, 124$, learning rate is 0.1, 5000 iterations. Please note, that only 500 points per curve are plotted. Gray curve corresponds to results for training set, black - validation.

mapping (variances)	average of best validation score		average of validation score on last 100 iterations	
	mean	SD	mean	SD
unbounded	0.0487	0.0237	0.2515	0.2709
lower bound of 1	0.0251	0.0179	0.0849	0.0278
fixed to 1	0.0151	0.0151	**0.0465**	0.0201
none (raw data)	**0.0020**	0.0058	0.0834	0.0247

Table 1: Comparison of average classification errors rate on validation set for different mapping cases and evaluation criteria. 1 mapping run and 1000 classification runs were held (where appropriate) for each mapping case and evaluation criterion. Best results in a row are bolded.

A File task1.cpp:

```
1
2  #include <iostream>
3  using std::cin;
4  using std::cout;
5  using std::endl;
6
7  #include <vector>
8  using std::vector;
9
10 #include <string>
11 using std::string;
12
13 #include "PseudoRandomGenerator.h"
14 #include "Output.h"
15
16 #include <math.h>
17
18 #include "KohonenNetwork.h"
19
20 // inside an equilateral triangle with
21 // sides of unit length and
22 // vertices (0; 0), (1; 0), (0.5; sqrt(3) / 2)
23 vector<long double> generate_point() {
24   long double x_in_square = Random::get_random_from_range(0, 1);
25   long double y_in_square = Random::get_random_from_range(0, 1);
26
27   if (x_in_square + y_in_square > 1) {
28     // wrong triangle, map them into the right one.
29     x_in_square = 1 - x_in_square;
30     y_in_square = 1 - y_in_square;
31   }
32
33   long double x_triangle, y_triangle;
34
35   const long double x_first = 0.5, y_first = sqrt(static_cast<long double>(3)) / 2;
36   const long double x_second = 1, y_second = 0;
37
38   x_triangle = x_in_square * x_first + y_in_square * x_second;
39   y_triangle = x_in_square * y_first + y_in_square * y_second;
40
41   vector<long double> result;
42   result.push_back(x_triangle);
43   result.push_back(y_triangle);
44
45   return result;
46 }
47
48 // inside an equilateral triangle with
49 // sides of unit length and
50 // vertices (0; 0), (1; 0), (0.5; sqrt(3) / 2)
51 vector< vector<long double> > generate_points(const unsigned int quantity) {
52   vector< vector<long double> > result(quantity);
53
54   for (unsigned int i = 0; i < quantity; ++i) {
55     result[i] = generate_point();
56   }
57
58   return result;
```

```
}

vector< vector<long double> > calculate_distances_matrix(const unsigned int quantity) {
  vector< vector<long double> > result(quantity, vector<long double>(quantity, 0));

  for (unsigned int i = 0; i < quantity; ++i) {
    for (unsigned int j = 0; j < i; ++j) {
      result[i][j] = result[j][i] = i - j;
    }
  }

  return result;
}

int main(int argc, char** argv) {

  unsigned int seeds[] =                  {123,   124,  123,  123,  123,  /**/  123,   124,  123,  123,  123};
  unsigned int points_quantities[] =      {1000, 1000, 5000, 1000, 5000,  /**/  1000, 1000, 5000, 1000, 5000};
  unsigned int all_outputs[] =            {100,   100,  100,  200,  200,  /**/  100,   100,  100,  200,  200};
  unsigned int ordering_initial_widths[] = {100,   100,  100,  100,  100,  /**/   10,    10,   10,   10,   10};

  for (unsigned int case_id = 0; case_id < 10; ++case_id) {

    unsigned int seed = seeds[case_id];
    unsigned int points_quantity = points_quantities[case_id];
    unsigned int outputs = all_outputs[case_id];

    Random::set_seed(seed);

    vector< vector<long double> > points = generate_points(points_quantity);

    KohonenNetwork network(2, outputs);

    unsigned int ordering_iterations = 1000;
    long double ordering_initial_width = ordering_initial_widths[case_id];
    long double ordering_width_factor = ordering_iterations / log(ordering_initial_width);

    long double ordering_initial_learning_rate = 0.1;

    unsigned int convergence_iterations = 1e7;

    const vector< vector<long double> > distances_matrix = calculate_distances_matrix(outputs);

    network.train(points, ordering_iterations, ordering_initial_width, ordering_width_factor,
            ordering_initial_learning_rate, ordering_width_factor,
            convergence_iterations, 0.9, 0.01,
            distances_matrix,
            1, 10,
            1, 10,
            false,
            "task_1_temp/" + get_string(case_id) + "/"
            );

    network.train(points, ordering_iterations, ordering_initial_width, ordering_width_factor,
            ordering_initial_learning_rate, ordering_width_factor,
```

```
            convergence_iterations, 0.9, 0.01,
            distances_matrix,
            100, 1,
            10000, 1,
            false,
            "task_1_temp/" + get_string(case_id) + "/"
            );
  }

  return 0;
}
```

B File task2.cpp:

```

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <vector>
using std::vector;

#include <string>
using std::string;

#include "PseudoRandomGenerator.h"
#include "Output.h"

#include <math.h>

#include "KohonenNetwork.h"
#include "Dataset.h"

vector< vector<long double> > calculate_distances_matrix_2D(const unsigned int rows,
                                const unsigned int columns) {
  const unsigned int quantity = rows * columns;
  vector< vector<long double> > result(quantity, vector<long double>(quantity, 0));

  for (unsigned int i = 0; i < quantity; ++i) {
    for (unsigned int j = 0; j < i; ++j) {

      unsigned int first_x = i / columns;
      unsigned int first_y = i % columns;

      unsigned int second_x = j / columns;
      unsigned int second_y = j % columns;

      result[i][j] = result[j][i] = sqrt( static_cast<long double>((first_x - second_x) * (first_x - second_x) +
                          (first_y - second_y) * (first_y - second_y)) );
    }
  }

  return result;
}

int main(int argc, char** argv) {

  unsigned int seeds[] = { 123, 124, 125, 126, 127};
```

```
46
47
48    for (unsigned int case_id = 0; case_id < 5; ++case_id) {
49
50      unsigned int seed = seeds[case_id];
51
52      Dataset wine = read_dataset("wine.data.txt", 0);
53      wine.normalize_to_zero_mean();
54      wine.normalize_to_unity_variance();
55
56      unsigned int outputs_rows = 20;
57      unsigned int outputs_columns = 20;
58      unsigned int outputs = outputs_rows * outputs_columns;
59
60      KohonenNetwork network(wine.get_input_variables_quantity(), outputs);
61
62      unsigned int ordering_iterations = 1000;
63      long double ordering_initial_width = 30;
64      long double ordering_width_factor = ordering_iterations / log(ordering_initial_width);
65
66      long double ordering_initial_learning_rate = 0.1;
67
68      unsigned int convergence_iterations = 1e6;
69
70      vector< vector<long double> > distances_matrix = calculate_distances_matrix_2D(
      outputs_rows, outputs_columns);
71
72      Random::set_seed(seed);
73      network.train(wine.inputs, ordering_iterations, ordering_initial_width,
      ordering_width_factor,
74              ordering_initial_learning_rate, ordering_width_factor,
75              convergence_iterations, 0.9, 0.01,
76              distances_matrix,
77              1, 10,
78              1, 10,
79              true,
80              "task_2_temp/" + get_string(case_id) + "/s_" + get_string(seed)
81              );
82
83      network.save_weights_to_file("task_2/"+ get_string(case_id) + "_final_weights.txt");
84      network.save_outputs_to_file(wine.inputs, "task_2/"+ get_string(case_id) + "
      _final_outputs.txt");
85    }
86
87    return 0;
88  }
```

C File task3.cpp:

```
1
2  #include <iostream>
3  using std::cin;
4  using std::cout;
5  using std::endl;
6
7  #include <vector>
8  using std::vector;
9
10 #include <string>
11 using std::string;
```

```
12
13 #include "PseudoRandomGenerator.h"
14 #include "Output.h"
15
16 #include <math.h>
17
18 #include "KohonenNetwork.h"
19 #include "Dataset.h"
20 #include "NeuralNetworkClassifier.h"
21
22 #include <algorithm>
23 using std::pair;
24 using std::make_pair;
25
26 vector< vector<long double> > calculate_distances_matrix(const unsigned int quantity) {
27   vector< vector<long double> > result(quantity, vector<long double>(quantity, 0));
28
29   for (unsigned int i = 0; i < quantity; ++i) {
30     for (unsigned int j = 0; j < i; ++j) {
31       result[i][j] = result[j][i] = i - j;
32     }
33   }
34
35   return result;
36 }
37
38 Dataset get_mapped_dataset(unsigned int case_id, unsigned int seed, long double
    variances_lower_bound,
39                 long double variances_upper_bound, unsigned int ordering_iterations,
40                 long double ordering_initial_width, long double
    ordering_initial_learning_rate,
41                 unsigned int convergence_iterations, long double convergence_constant_width,
42                 long double convergence_constant_learning_rate,
43                 int ordering_reports_saving_period,
44                 int ordering_reports_saving_multiplier,
45                 int convergence_reports_saving_period,
46                 int convergence_reports_saving_multiplier,
47                 bool save_outputs) {
48   Dataset wine = read_dataset("wine.data.txt", 0);
49   wine.normalize_to_zero_mean();
50   wine.normalize_to_unity_variance();
51
52   unsigned int outputs = 10;
53   KohonenNetwork network(wine.get_input_variables_quantity(), outputs);
54
55   long double ordering_width_factor = ordering_iterations / log(ordering_initial_width);
56   vector< vector<long double> > distances_matrix = calculate_distances_matrix(outputs);
57
58   Random::set_seed(seed);
59   network.train_Gaussian(wine.inputs, ordering_iterations, ordering_initial_width,
60                 ordering_width_factor, ordering_initial_learning_rate,
61                 ordering_width_factor, convergence_iterations,
62                 0.9, 0.02, distances_matrix,
63                 variances_lower_bound,
64                 variances_upper_bound,
65                 ordering_reports_saving_period,
66                 ordering_reports_saving_multiplier,
67                 convergence_reports_saving_period,
68                 convergence_reports_saving_multiplier,
69                 save_outputs,
```

```
70                 "task_3/" + get_string(case_id) + "/s_" + get_string(seed)
71                 );
72
73   Dataset mapped_wine;
74
75   mapped_wine.inputs = network.get_neurons_values_Gaussian(wine.inputs, variances_lower_bound
     ,
76                                 variances_upper_bound);
77   mapped_wine.outputs.assign(wine.size(), vector<long double>(3, -1));
78   for (unsigned int i = 0; i < wine.size(); ++i) {
79     unsigned int current_class = static_cast<unsigned int>(wine.outputs[i][0] + 0.5);
80     mapped_wine.outputs[i][current_class - 1] *= -1;
81   }
82
83   return mapped_wine;
84 }
85
86 pair<long double, long double> do_classifier_experiment(const Dataset& dataset,
87                               const vector<unsigned int>& hidden_layers,
88                               unsigned int seed, unsigned int iterations,
89                               long double learning_rate, bool with_report,
90                               const string& path_and_prefix) {
91   Dataset train = dataset;
92   train.normalize_to_zero_mean();
93   train.normalize_to_unity_variance();
94   train.random_shuffle(seed);
95   Dataset validation = train.cutoff_back(0.3 * dataset.size());
96
97   vector<unsigned int> layers;
98   layers.push_back(train.get_input_variables_quantity());
99   for (unsigned int i = 0; i < hidden_layers.size(); ++i) {
100    layers.push_back(hidden_layers[i]);
101  }
102  layers.push_back(train.get_output_variables_quantity());
103
104  NeuralNetworkClassifier classifier(layers);
105
106  vector< pair<unsigned int, long double> > train_error_curve, validation_error_curve;
107  vector< pair<unsigned int, long double> > train_energy_curve, validation_energy_curve;
108
109  long double min_validation_error = 1;
110  long double sum = 0;
111
112  const unsigned int PERIOD = 100;
113
114  for (unsigned int iteration = 1; iteration <= iterations; ++iteration) {
115    classifier.do_one_iteration_of_training(train, 0, learning_rate);
116
117    long double validation_error = classifier.calculate_classification_error(validation);
118    if (validation_error < min_validation_error) {
119        min_validation_error = validation_error;
120    }
121
122    if (iteration + PERIOD > iterations) {
123      sum += validation_error;
124    }
125
126    if (with_report) {
127      long double train_error = classifier.calculate_classification_error(train);
128
```

```
129       long double train_energy = classifier.calculate_energy(train) / train.size();
130       long double validation_energy = classifier.calculate_energy(validation) / validation.
      size();
131
132
133
134       train_error_curve.push_back(make_pair(iteration, train_error));
135       validation_error_curve.push_back(make_pair(iteration, validation_error));
136
137       train_energy_curve.push_back(make_pair(iteration, train_energy));
138       validation_energy_curve.push_back(make_pair(iteration, validation_energy));
139     }
140   }
141
142   if (with_report) {
143     save_to_file(path_and_prefix + get_string(seed) + "_train_error.txt", train_error_curve);
144     save_to_file(path_and_prefix + get_string(seed) + "_validation_error.txt",
      validation_error_curve);
145
146     save_to_file(path_and_prefix + get_string(seed) + "_train_energy.txt", train_energy_curve
      );
147     save_to_file(path_and_prefix + get_string(seed) + "_validation_energy.txt",
      validation_energy_curve);
148   }
149
150   return make_pair(min_validation_error, sum / PERIOD);
151 }
152
153 void do_series_of_classifier_experiments(const Dataset& dataset, const vector<unsigned int>&
      hidden_layers,
154                 unsigned int start_seed, unsigned int end_seed, unsigned int iterations,
155                 long double learning_rate, unsigned int quantity_with_report,
156                 const string& path_and_prefix) {
157
158   vector<long double> min_results;
159   vector<long double> average_results;
160
161   for (unsigned int i = 0; i < end_seed - start_seed + 1; ++i) {
162
163     bool with_report = (i < quantity_with_report);
164
165     pair<long double, long double> result = do_classifier_experiment(dataset,
166                                       hidden_layers, start_seed + i,
167                                       iterations, learning_rate,
168                                       with_report, path_and_prefix);
169
170     min_results.push_back(result.first);
171     average_results.push_back(result.second);
172   }
173
174   save_to_file(path_and_prefix + "series.txt", min_results);
175   save_to_file(path_and_prefix + "series_average.txt", average_results);
176 }
177
178 int main(int argc, char** argv) {
179
180   const unsigned int RUNS_QUANTITY = 1000;
181   const unsigned int START_SEED = 123;
182   const unsigned int END_SEED = START_SEED + RUNS_QUANTITY - 1;
183   const unsigned int ITERATIONS = 5000;
```

```
184 // mapping with unbounded variances (s)
185
186 unsigned int case_id = 0;
187 unsigned int seed = 123 + case_id;
188 long double variances_lower_bound = 0;
189 long double variances_upper_bound = INFINITY;
190
191 unsigned int ordering_iterations = 1e4;
192 long double ordering_initial_width = 10;
193 long double ordering_initial_learning_rate = 0.1;
194
195 unsigned int convergence_iterations = 1e5;
196 long double convergence_constant_width = 0.9;
197 long double convergence_constant_learning_rate = 0.02;
198
199 int ordering_reports_saving_period = 1;
200 int ordering_reports_saving_multiplier = 10;
201 int convergence_reports_saving_period = 1;
202 int convergence_reports_saving_multiplier = 10;
203 bool save_outputs = true;
204
205 Dataset mapped_wine = get_mapped_dataset(case_id, seed, variances_lower_bound,
     variances_upper_bound,
206                         ordering_iterations, ordering_initial_width,
207                         ordering_initial_learning_rate, convergence_iterations,
208                         convergence_constant_width, convergence_constant_learning_rate,
209                         ordering_reports_saving_period, ordering_reports_saving_multiplier,
210                         convergence_reports_saving_period,
     convergence_reports_saving_multiplier,
211                           save_outputs);
212
213 do_series_of_classifier_experiments(mapped_wine, vector<unsigned int>(), START_SEED,
     END_SEED,
214                     ITERATIONS, 0.1, 10, "task_3/class/mapped_unbounded/");
215
216
217 // mapping with lower bound of variances
218
219 case_id = 1;
220 variances_lower_bound = 1;
221 variances_upper_bound = INFINITY;
222
223 mapped_wine = get_mapped_dataset(case_id, seed, variances_lower_bound,
     variances_upper_bound,
224                         ordering_iterations, ordering_initial_width,
225                         ordering_initial_learning_rate, convergence_iterations,
226                         convergence_constant_width, convergence_constant_learning_rate,
227                         ordering_reports_saving_period, ordering_reports_saving_multiplier,
228                         convergence_reports_saving_period,
     convergence_reports_saving_multiplier,
229                           save_outputs);
230
231 do_series_of_classifier_experiments(mapped_wine, vector<unsigned int>(), START_SEED,
     END_SEED,
232                     ITERATIONS, 0.1, 10, "task_3/class/mapped_lower_bound/");
233
234 // mapping with fixed variances
235
236 case_id = 2;
237 variances_lower_bound = 1;
```

```
238   variances_upper_bound = 1;
239
240   mapped_wine = get_mapped_dataset(case_id, seed, variances_lower_bound,
      variances_upper_bound,
241                           ordering_iterations, ordering_initial_width,
242                           ordering_initial_learning_rate, convergence_iterations,
243                           convergence_constant_width, convergence_constant_learning_rate,
244                           ordering_reports_saving_period, ordering_reports_saving_multiplier,
245                           convergence_reports_saving_period,
      convergence_reports_saving_multiplier,
246                             save_outputs);
247
248   do_series_of_classifier_experiments(mapped_wine, vector<unsigned int>(), START_SEED,
      END_SEED,
249                           ITERATIONS, 0.1, 10, "task_3/class/mapped_fixed/");
250
251
252   // raw
253
254   Dataset wine = read_dataset("wine.data.txt", 0);
255   wine.normalize_to_zero_mean();
256   wine.normalize_to_unity_variance();
257
258   for (unsigned int i = 0; i < wine.size(); ++i) {
259     unsigned int current = static_cast<unsigned int>(wine.outputs[i][0] + 0.5);
260     wine.outputs[i].assign(3, -1);
261     wine.outputs[i][current - 1] *= -1;
262   }
263
264   do_series_of_classifier_experiments(wine, vector<unsigned int>(), START_SEED, END_SEED,
265                           ITERATIONS, 0.1, 10, "task_3/class/raw/");
266   return 0;
267 }
```

D File KohonenNetwork.h:

```
1  #pragma once
2  #ifndef KOHONEN_NETWORK_H
3  #define KOHONEN_NETWORK_H
4
5  #include <vector>
6  using std::vector;
7
8  #include <string>
9  using std::string;
10
11 #include <limits>
12
13 const long double INFINITY = std::numeric_limits<long double>::infinity();
14 const long double NONE = -1;
15
16 class KohonenNetwork {
17 private:
18
19   unsigned int dimensions_;
20   unsigned int outputs_quantity_;
21
22   vector< vector<long double> > weights_;
23
24   void initialize_(unsigned int dimensions, unsigned int outputs_quantity);
```

```
25
26   void initialize_random_weights_(const vector<long double>& minimal,
27                     const vector<long double>& maximal);
28
29   void get_bounding_box_(const vector< vector<long double> >& dataset,
30               vector<long double>& minimal, vector<long double>& maximal);
31
32
33
34   unsigned int get_closest_weight_index_(const vector<long double>& pattern) const;
35
36   static long double calculate_time_dependent_value_(long double start, long double factor,
37                             unsigned int time);
38
39   void update_weights_(unsigned int closest_index, long double learning_rate,
40               long double width, const vector<long double>& pattern,
41               const vector< vector<long double> >& outputs_distances);
42   static long double get_Gaussian_value_(long double x, long double variance);
43   static long double calculate_neighbourhood_function_(long double distance, long double
      width);
44
45   vector<long double> calculate_Gaussian_variances_(long double lower_bound,
46                             long double upper_bound) const;
47   unsigned int get_winning_index_Gaussian_(const vector<long double>& pattern,
48                         long double variances_lower_bound,
49                         long double variances_upper_bound,
50                         vector<long double> variances) const;
51
52   unsigned int classify_(const vector<long double>& pattern,
53                bool is_Gaussian_maximisation,
54                long double variances_lower_bound,
55                long double variances_upper_bound,
56                const vector<long double>& variances) const;
57
58   void save_outputs_to_file_(const vector< vector<long double> >& dataset, const string& path
      ,
59                  bool is_Gaussian_maximisation,
60                  long double variances_lower_bound,
61                  long double variances_upper_bound) const;
62
63   void train_(const vector< vector<long double> >& dataset, unsigned int ordering_iterations,
64         long double ordering_initial_width, long double ordering_width_factor,
65         long double ordering_initial_learning_rate,
66         long double ordering_learning_rate_factor,
67         unsigned int convergence_iterations, long double convergence_constant_width,
68         long double convergence_constant_learning_rate,
69         const vector< vector<long double> >& outputs_distances,
70         int ordering_reports_saving_period,
71         int ordering_reports_saving_multiplier,
72         int convergence_reports_saving_period,
73         int convergence_reports_saving_multiplier,
74         bool save_outputs, const string& path_and_prefix,
75         bool is_Gaussian_maximisation, long double variances_lower_bound,
76         long double variances_upper_bound
77         );
78
79 public:
80   KohonenNetwork();
81   KohonenNetwork(unsigned int dimensions, unsigned int outputs_quantity);
82
```

```
83   void save_weights_to_file(const string& path) const;
84   void save_outputs_to_file(const vector< vector<long double> >& dataset, const string& path)
       const;
85
86   static const int NO_REPORTS = -1;
87
88   // based on closest distances
89   void train(const vector< vector<long double> >& dataset, unsigned int ordering_iterations,
90           long double ordering_initial_width, long double ordering_width_factor,
91           long double ordering_initial_learning_rate,
92           long double ordering_learning_rate_factor,
93           unsigned int convergence_iterations, long double convergence_constant_width,
94           long double convergence_constant_learning_rate,
95           const vector< vector<long double> >& outputs_distances,
96           int ordering_reports_saving_period = NO_REPORTS,
97           int ordering_reports_saving_multiplier = NO_REPORTS,
98           int convergence_reports_saving_period = NO_REPORTS,
99           int convergence_reports_saving_multiplier = NO_REPORTS,
100          bool save_outputs = false, const string& path_and_prefix = string("")
101          );
102
103  static long double calculate_squared_distance(const vector<long double>& from,
104                       const vector<long double>& to);
105
106  // based on closest distances
107  unsigned int classify(const vector<long double>& pattern) const;
108
109
110  vector< vector<long double> > get_neurons_values_Gaussian(const vector< vector<long double>
       >& dataset,
111                               long double variances_lower_bound,
112                               long double variances_upper_bound) const;
113
114
115
116  void train_Gaussian(const vector< vector<long double> >& dataset,
117              unsigned int ordering_iterations,
118              long double ordering_initial_width, long double ordering_width_factor,
119              long double ordering_initial_learning_rate,
120              long double ordering_learning_rate_factor,
121              unsigned int convergence_iterations,
122              long double convergence_constant_width,
123              long double convergence_constant_learning_rate,
124              const vector< vector<long double> >& outputs_distances,
125              long double variances_lower_bound = 0,
126              long double variances_upper_bound = INFINITY,
127              int ordering_reports_saving_period = NO_REPORTS,
128              int ordering_reports_saving_multiplier = NO_REPORTS,
129              int convergence_reports_saving_period = NO_REPORTS,
130              int convergence_reports_saving_multiplier = NO_REPORTS,
131              bool save_outputs = false, const string& path_and_prefix = string("")
132              );
133
134
135
136 };
137
138
139 #endif
```

E File KohonenNetwork.cpp:

```
#include "KohonenNetwork.h"

#include "PseudoRandomGenerator.h"
#include <limits>
#include <math.h>
#include "Output.h"
#include <iostream>
#include <ctime>

KohonenNetwork::KohonenNetwork() {}

KohonenNetwork::KohonenNetwork(unsigned int dimensions, unsigned int outputs_quantity) {
  initialize_(dimensions, outputs_quantity);
}

void KohonenNetwork::initialize_(unsigned int dimensions, unsigned int outputs_quantity) {
  dimensions_ = dimensions;
  outputs_quantity_ = outputs_quantity;

  weights_.assign(outputs_quantity, vector<long double>(dimensions_));
}

void KohonenNetwork::initialize_random_weights_(const vector<long double>& minimal,
                          const vector<long double>& maximal) {
  for (unsigned int i = 0; i < outputs_quantity_; ++i) {
    for (unsigned int j = 0; j < dimensions_; ++j) {
      weights_[i][j] = Random::get_random_from_range(minimal[j], maximal[j]);
    }
  }
}

void KohonenNetwork::get_bounding_box_(const vector< vector<long double> >& dataset,
                     vector<long double>& minimal, vector<long double>& maximal) {
  minimal = dataset[0];
  maximal = dataset[0];

  for (unsigned int i = 0; i < dataset.size(); ++i) {
    for (unsigned int j = 0; j < dimensions_; ++j) {
      if (minimal[j] > dataset[i][j]) {
        minimal[j] = dataset[i][j];
      }
      if (maximal[j] < dataset[i][j]) {
        maximal[j] = dataset[i][j];
      }
    }
  }
}

long double KohonenNetwork::calculate_squared_distance(const vector<long double>& from,
                             const vector<long double>& to) {
  long double sum = 0;
  for (unsigned int i = 0; i < from.size(); ++i) {
    sum += (from[i] - to[i]) * (from[i] - to[i]);
  }
  return sum;
}

unsigned int KohonenNetwork::get_closest_weight_index_(const vector<long double>& pattern)
```

```
    const {
  unsigned int closest_index = 0;
  long double best_squared_distance = INFINITY;

  for (unsigned int i = 1; i < weights_.size(); ++i) {
    long double current_squared_distance = calculate_squared_distance(weights_[i], pattern);
    if (current_squared_distance < best_squared_distance) {
      closest_index = i;
      best_squared_distance = current_squared_distance;
    }
  }

  return closest_index;
}

long double KohonenNetwork::calculate_time_dependent_value_(long double start, long double factor,
                                  unsigned int time) {
  return start * exp(-static_cast<long double>(time) / factor);
}

void KohonenNetwork::update_weights_(unsigned int closest_index, long double learning_rate,
                     long double width, const vector<long double>& pattern,
                     const vector< vector<long double> >& outputs_distances) {
  for (unsigned int i = 0; i < weights_.size(); ++i) {
    long double neighbour_value = calculate_neighbourhood_function_(outputs_distances[i][closest_index],
                                       width);
    long double factor = learning_rate * neighbour_value;
    for (unsigned int j = 0; j < weights_[i].size(); ++j) {
      weights_[i][j] += factor * (pattern[j] - weights_[i][j]);
    }
  }
}

long double KohonenNetwork::get_Gaussian_value_(long double x, long double variance) {
  return exp(-x * x / 2 / variance / variance);
}

long double KohonenNetwork::calculate_neighbourhood_function_(long double distance, long double width) {
  return get_Gaussian_value_(distance, width);
}

vector<long double> KohonenNetwork::calculate_Gaussian_variances_(long double lower_bound,
                                   long double upper_bound) const {
  vector<long double> result(weights_.size(), INFINITY);
  for (unsigned int i = 0; i < weights_.size(); ++i) {
    for (unsigned int j = 0; j < i; ++j) {
      long double squared_distance = calculate_squared_distance(weights_[i], weights_[j]);
      if (result[i] > squared_distance) {
        result[i] = squared_distance;
      }
      if (result[j] > squared_distance) {
        result[j] = squared_distance;
      }
    }
  }

  for (unsigned int i = 0; i < weights_.size(); ++i) {
```

```
115     result[i] = sqrt(result[i]);
116
117     if (result[i] < lower_bound) {
118       result[i] = lower_bound;
119     }
120
121     if (result[i] > upper_bound) {
122       result[i] = upper_bound;
123     }
124   }
125
126   return result;
127 }
128
129
130 unsigned int KohonenNetwork::get_winning_index_Gaussian_(const vector<long double>& pattern,
131                                long double variances_lower_bound,
132                                long double variances_upper_bound,
133                                vector<long double> variances) const {
134   unsigned int max_index = 0;
135   long double max_value = -INFINITY;
136
137   if (variances.empty()) {
138     variances = calculate_Gaussian_variances_(variances_lower_bound, variances_upper_bound);
139   }
140
141   for (unsigned int i = 1; i < weights_.size(); ++i) {
142     long double distance = sqrt(calculate_squared_distance(pattern, weights_[i]));
143     long double current = get_Gaussian_value_(distance, variances[i]);
144
145     if (current > max_value) {
146       max_value = current;
147       max_index = i;
148     }
149
150   }
151
152   return max_index;
153 }
154
155 void KohonenNetwork::save_weights_to_file(const string& path) const {
156   save_to_file(path, weights_);
157 }
158
159 void KohonenNetwork::save_outputs_to_file(const vector< vector<long double> >& dataset, const
      string& path) const {
160   const bool IS_GAUSSIAN_MAXIMIZATION = false;
161   save_outputs_to_file_(dataset, path, IS_GAUSSIAN_MAXIMIZATION, NONE, NONE);
162 }
163
164 unsigned int KohonenNetwork::classify_(const vector<long double>& pattern,
165                       bool is_Gaussian_maximisation,
166                       long double variances_lower_bound,
167                       long double variances_upper_bound,
168                       const vector<long double>& variances) const {
169   if (is_Gaussian_maximisation) {
170     return get_winning_index_Gaussian_(pattern, variances_lower_bound,
171                       variances_upper_bound, variances);
172   } else {
173     return get_closest_weight_index_(pattern);
```

```
174   }
175 }
176
177 void KohonenNetwork::save_outputs_to_file_(const vector< vector<long double> >& dataset,
      const string& path,
178                 bool is_Gaussian_maximisation,
179                 long double variances_lower_bound,
180                 long double variances_upper_bound) const {
181   vector<unsigned int> ids(dataset.size());
182
183   vector<long double> variances;
184   if (is_Gaussian_maximisation) {
185     variances = calculate_Gaussian_variances_(variances_lower_bound, variances_upper_bound);
186   }
187
188   for (unsigned int i = 0; i < ids.size(); ++i) {
189     ids[i] = classify_(dataset[i], is_Gaussian_maximisation, variances_lower_bound,
      variances_upper_bound, variances);
190   }
191
192   save_to_file(path, ids);
193 }
194
195
196 void KohonenNetwork::train_(const vector< vector<long double> >& dataset, unsigned int
      ordering_iterations,
197             long double ordering_initial_width, long double ordering_width_factor,
198             long double ordering_initial_learning_rate, long double
      ordering_learning_rate_factor,
199             unsigned int convergence_iterations, long double convergence_constant_width,
200             long double convergence_constant_learning_rate,
201             const vector< vector<long double> >& outputs_distances,
202             int ordering_reports_saving_period, int ordering_reports_saving_multiplier,
203             int convergence_reports_saving_period,  int
      convergence_reports_saving_multiplier,
204             bool save_outputs, const string& path_and_prefix,
205             bool is_Gaussian_maximisation, long double variances_lower_bound,
206             long double variances_upper_bound) {
207
208   vector<long double> minimal, maximal;
209   get_bounding_box_(dataset, minimal, maximal);
210   initialize_random_weights_(minimal, maximal);
211
212
213   if (save_outputs) {
214   save_outputs_to_file_(dataset, path_and_prefix + "_ordering_"
215                 + get_string(0) + "_outputs.txt",
216                 is_Gaussian_maximisation, variances_lower_bound,
217                 variances_upper_bound);
218   }
219
220   save_weights_to_file(path_and_prefix + "_ordering_"
221                   + get_string(0) + "_weights.txt");
222
223
224   long double width = ordering_initial_width;
225   long double learning_rate = ordering_initial_learning_rate;
226
227   unsigned int output_counter = 1;
228   unsigned int quantity_saved = 0;
```

```
229
230   for (unsigned int iteration = 1; iteration <= ordering_iterations; ++iteration) {
231
232     width = calculate_time_dependent_value_(ordering_initial_width, ordering_width_factor,
233                           iteration);
234
235     learning_rate = calculate_time_dependent_value_(ordering_initial_learning_rate,
236                               ordering_learning_rate_factor,
237                               iteration);
238
239     const unsigned int pattern_id = Random::get_random_number(dataset.size());
240
241     unsigned int closest_weight_index;
242     if (is_Gaussian_maximisation) {
243       closest_weight_index = get_winning_index_Gaussian_(dataset[pattern_id],
244                                 variances_lower_bound,
245                                 variances_upper_bound,
246                                 vector<long double>());
247     } else {
248       // closet distance
249       closest_weight_index = get_closest_weight_index_(dataset[pattern_id]);
250     }
251
252
253     update_weights_(closest_weight_index, learning_rate, width, dataset[pattern_id],
254               outputs_distances);
255
256     if (output_counter == ordering_reports_saving_period) {
257
258       quantity_saved++;
259
260       if (quantity_saved == 10) {
261         ordering_reports_saving_period *= ordering_reports_saving_multiplier;
262         quantity_saved = 1;
263       }
264
265       std::cerr << "ord. " << iteration << "\r";
266
267       if (save_outputs) {
268         save_outputs_to_file_(dataset, path_and_prefix + "_ordering_"
269                       + get_string(iteration) + "_outputs.txt",
270                       is_Gaussian_maximisation, variances_lower_bound,
271                       variances_upper_bound);
272       }
273
274       save_weights_to_file(path_and_prefix + "_ordering_"
275                        + get_string(iteration) + "_weights.txt");
276
277       output_counter = 1;
278     } else {
279       if (ordering_reports_saving_period != NO_REPORTS) {
280         ++output_counter;
281       }
282     }
283
284   }
285
286   width = convergence_constant_width;
287   learning_rate = convergence_constant_learning_rate;
288
```

```
289   unsigned int start_time = clock();
290   output_counter = 1;
291
292   quantity_saved = 0;
293
294   for (unsigned int iteration = 1; iteration <= convergence_iterations; ++iteration) {
295
296     const unsigned int pattern_id = Random::get_random_number(dataset.size());
297
298     const unsigned int closest_weight_index = get_closest_weight_index_(dataset[pattern_id]);
299
300     update_weights_(closest_weight_index, learning_rate, width, dataset[pattern_id],
301               outputs_distances);
302
303
304     if (output_counter == convergence_reports_saving_period) {
305
306       quantity_saved++;
307
308       if (quantity_saved == 10) {
309         convergence_reports_saving_period *= convergence_reports_saving_multiplier;
310         quantity_saved = 1;
311       }
312
313       float speed = iteration / (static_cast<float>(clock() - start_time) / CLOCKS_PER_SEC);
314
315       std::cerr << "conv. " << iteration << "  (" << std::fixed << std::setprecision(2) <<
    speed << ")      \r";
316
317       if (save_outputs) {
318         save_outputs_to_file_(dataset, path_and_prefix + "_convergence_"
319                       + get_string(iteration) + "_outputs.txt",
320                       is_Gaussian_maximisation, variances_lower_bound,
321                       variances_upper_bound);
322       }
323
324       save_weights_to_file(path_and_prefix + "_convergence_"
325                        + get_string(iteration) + "_weights.txt");
326
327       output_counter = 1;
328     } else {
329       if (convergence_reports_saving_period != NO_REPORTS) {
330         ++output_counter;
331       }
332     }
333
334
335   }
336
337   std::cerr << "\r                                                    \r";
338 }
339
340
341 void KohonenNetwork::train(const vector< vector<long double> >& dataset, unsigned int
    ordering_iterations,
342               long double ordering_initial_width, long double ordering_width_factor,
343               long double ordering_initial_learning_rate, long double
    ordering_learning_rate_factor,
344               unsigned int convergence_iterations, long double convergence_constant_width,
345               long double convergence_constant_learning_rate,
```

```
346                 const vector< vector<long double> >& outputs_distances,
347                 int ordering_reports_saving_period, int ordering_reports_saving_multiplier,
348                 int convergence_reports_saving_period,  int
    convergence_reports_saving_multiplier,
349                 bool save_outputs, const string& path_and_prefix) {
350   const bool IS_GAUSSIAN_MAXIMIZATION = false;
351   train_(dataset, ordering_iterations, ordering_initial_width, ordering_width_factor,
352       ordering_initial_learning_rate, ordering_learning_rate_factor,
353       convergence_iterations, convergence_constant_width,
354       convergence_constant_learning_rate,
355       outputs_distances,
356       ordering_reports_saving_period, ordering_reports_saving_multiplier,
357       convergence_reports_saving_period, convergence_reports_saving_multiplier,
358       save_outputs, path_and_prefix,
359       IS_GAUSSIAN_MAXIMIZATION, NONE, NONE);
360 }
361
362 unsigned int KohonenNetwork::classify(const vector<long double>& pattern) const {
363   const bool IS_GAUSSIAN_MAXIMIZATION = false;
364   return classify_(pattern, IS_GAUSSIAN_MAXIMIZATION, NONE, NONE, vector<long double>());
365 }
366
367 vector< vector<long double> > KohonenNetwork::get_neurons_values_Gaussian
368                         (const vector< vector<long double> >& dataset,
369                            long double variances_lower_bound,
370                           long double variances_upper_bound) const {
371   vector<long double> variances = calculate_Gaussian_variances_(variances_lower_bound,
372                                   variances_upper_bound);
373
374   vector< vector<long double> > result(dataset.size(), vector<long double>(weights_.size()));
375
376   for (unsigned int i = 0; i < result.size(); ++i) {
377     long double sum = 0;
378     for (unsigned int j = 0; j < result[i].size(); ++j) {
379       long double distance = sqrt(calculate_squared_distance(dataset[i], weights_[j]));
380       long double current = get_Gaussian_value_(distance, variances[j]);
381
382       result[i][j] = current;
383
384       sum += result[i][j];
385     }
386
387     for (unsigned int j = 0; j < result[i].size(); ++j) {
388       result[i][j] /= sum;
389     }
390   }
391
392   return result;
393 }
394
395 void KohonenNetwork::train_Gaussian(const vector< vector<long double> >& dataset,
396                     unsigned int ordering_iterations,
397                     long double ordering_initial_width,
398                     long double ordering_width_factor,
399                     long double ordering_initial_learning_rate,
400                     long double ordering_learning_rate_factor,
401                     unsigned int convergence_iterations,
402                     long double convergence_constant_width,
403                     long double convergence_constant_learning_rate,
404                     const vector< vector<long double> >& outputs_distances,
```

```
405                     long double variances_lower_bound,
406                     long double variances_upper_bound,
407                     int ordering_reports_saving_period,
408                     int ordering_reports_saving_multiplier,
409                     int convergence_reports_saving_period,
410                     int convergence_reports_saving_multiplier,
411                     bool save_outputs , const string& path_and_prefix
412                     ) {
413   const bool IS_GAUSSIAN_MAXIMIZATION = true;
414   train_(dataset, ordering_iterations, ordering_initial_width, ordering_width_factor,
415       ordering_initial_learning_rate, ordering_learning_rate_factor,
416       convergence_iterations, convergence_constant_width,
417       convergence_constant_learning_rate,
418       outputs_distances,
419       ordering_reports_saving_period, ordering_reports_saving_multiplier,
420       convergence_reports_saving_period, convergence_reports_saving_multiplier,
421       save_outputs, path_and_prefix,
422       IS_GAUSSIAN_MAXIMIZATION, variances_lower_bound,
423       variances_upper_bound);
424 }
```

F File NeuralNetworkClassifier.h:

```
1  #pragma once
2  #ifndef NEURAL_NETWORK_CLASSIFIER_H
3  #define NEURAL_NETWORK_CLASSIFIER_H
4
5  #include <vector>
6  using std::vector;
7
8  #include <math.h>
9
10 #include "PseudoRandomGenerator.h"
11 #include "Dataset.h"
12
13
14 // based on "A Gentle Introduction to Backpropagation" by Shashi Sathyanarayana.
15 // http://numericinsight.com/uploads/A_Gentle_Introduction_to_Backpropagation.pdf
16 // pages 11 - 14.
17
18 class NeuralNetworkClassifier {
19 private:
20
21   unsigned int layers_quantity_;
22   unsigned int output_layer_id_;
23
24   static const unsigned int bias_id_ = 0;
25   static const unsigned int input_layer_id_ = 0;
26
27   long double beta_;
28
29   vector<unsigned int> layers_sizes_;
30   // weights_[l][i][j] = weight of edge
31   //   from i-th neuron of layer (l-1)
32   //     to  j-th neuron of layer l
33   vector< vector< vector<long double> > > weights_, inertia_;
34   vector< vector<long double> > inputs_;
35   vector< vector<long double> > errors_;
36
37
```

```
38   long double calculate_g_function_(long double x) const;
39   long double calculate_g_derivative_(long double x) const;
40
41   void initialize_(const vector<unsigned int>& layers_sizes);
42
43   void initialize_random_weights_(const long double range);
44   void initialize_random_thresholds_(const long double range);
45
46   void initialize_input_layer_(const vector<long double>& pattern);
47
48   void propagate_forward_();
49   void calculate_errors_(const vector<long double>& desired_output);
50
51   void update_weights_(const long double learning_rate, const long double alpha = 0);
52
53 public:
54
55
56   NeuralNetworkClassifier();
57   NeuralNetworkClassifier(const vector<unsigned int>& layer_sizes, long double beta = 0.5);
58
59   void prepare_to_training(const unsigned int seed,
60                const long double weight_initialization_range = 0.2,
61                const long double threshold_initialization_range = 1);
62
63   void do_one_iteration_of_training(const vector< vector<long double> >& data,
64                     const vector< vector< long double > >& outputs,
65                     const long double alpha = 0,
66                     const long double learning_rate = 0.01);
67
68   void train(const vector< vector<long double> >& data,
69           const vector< vector< long double > >& outputs,
70           const unsigned int seed,
71           const long long iterations,
72           const long double alpha = 0,
73           const long double weight_initialization_range = 0.2,
74           const long double threshold_initialization_range = 1,
75           const long double learning_rate = 0.01);
76
77   vector<long double> classify(const vector< long double >& pattern);
78   long double NeuralNetworkClassifier::classify_one_output(const vector< long double >&
      pattern);
79
80   long double calculate_classification_error_one_output(const vector< vector<long double> >&
      inputs,
81                           const vector< vector<long double> >& outputs);
82
83   long double calculate_energy_one_output(const vector< vector<long double> >& inputs,
84                   const vector< vector<long double> >& outputs);
85
86   long double calculate_classification_error(const vector< vector<long double> >& inputs,
87                           const vector< vector<long double> >& outputs);
88
89   long double calculate_energy(const vector< vector<long double> >& inputs,
90                   const vector< vector<long double> >& outputs);
91
92   // overloads with struct Dataset
93
94   void do_one_iteration_of_training(const Dataset& dataset,
95                     const long double alpha = 0,
```

```
96                    const long double learning_rate = 0.01);
97
98   void train(const Dataset& dataset,
99           const unsigned int seed,
100          const long long iterations,
101          const long double alpha = 0,
102          const long double weight_initialization_range = 0.2,
103          const long double threshold_initialization_range = 1,
104          const long double learning_rate = 0.01);
105
106  long double calculate_classification_error_one_output(const Dataset& dataset);
107  long double calculate_energy_one_output(const Dataset& dataset);
108  long double calculate_classification_error(const Dataset& dataset);
109  long double calculate_energy(const Dataset& dataset);
110 };
111
112 #endif
```

G File NeuralNetworkClassifier.cpp:

```
1  #include "NeuralNetworkClassifier.h"
2
3  #include <math.h>
4
5  long double NeuralNetworkClassifier::calculate_g_function_(long double x) const {
6    return tanh(beta_ * x);
7  }
8
9  // lecture notes, chapter 4, page 114
10 long double NeuralNetworkClassifier::calculate_g_derivative_(long double x) const {
11   return beta_ * (1 - pow(tanh(beta_ * x), 2));
12 }
13
14 void NeuralNetworkClassifier::initialize_(const vector<unsigned int>& layers_sizes) {
15   layers_sizes_ = layers_sizes;
16
17   // 0-th neuron in each column corresponds to the bias.
18   for (unsigned int i = 0; i < layers_sizes_.size(); ++i) {
19     ++layers_sizes_[i];
20   }
21
22   layers_quantity_ = layers_sizes_.size();
23   output_layer_id_ = layers_sizes_.size() - 1;
24
25   inputs_.resize(layers_quantity_);
26   errors_.resize(layers_quantity_);
27   for (unsigned int i = 0; i < layers_quantity_; ++i) {
28     inputs_[i].resize(layers_sizes_[i]);
29     errors_[i].resize(layers_sizes_[i]);
30   }
31
32   weights_.resize(layers_quantity_ - 1);
33   inertia_.resize(layers_quantity_ - 1);
34   for (unsigned int i = 0; i + 1 < layers_quantity_; ++i) {
35     weights_[i].assign(layers_sizes_[i], vector<long double>(layers_sizes_[i+1]));
36     inertia_[i].assign(layers_sizes_[i], vector<long double>(layers_sizes_[i+1], 0));
37   }
38 }
39
40 void NeuralNetworkClassifier::initialize_random_weights_(const long double range) {
```

```
41   for (unsigned int layer = 0; layer < weights_.size(); ++layer) {
42     for (unsigned int from = 0; from < weights_[layer].size(); ++from) {
43       for (unsigned int to = 0; to < weights_[layer][from].size(); ++to) {
44         if (from == bias_id_ || to == bias_id_) {
45           continue;
46         }
47         weights_[layer][from][to] = Random::get_random_from_range(-range, +range);
48       }
49     }
50   }
51 }
52
53 void NeuralNetworkClassifier::initialize_random_thresholds_(const long double range) {
54   for (unsigned int layer = 0; layer < weights_.size(); ++layer) {
55     for (unsigned int to = 1; to < weights_[layer][bias_id_].size(); ++to) {
56       weights_[layer][bias_id_][to] = Random::get_random_from_range(-range, +range);
57     }
58   }
59 }
60
61 void NeuralNetworkClassifier::initialize_input_layer_(const vector<long double>& pattern) {
62   for (unsigned int i = 1; i < layers_sizes_[input_layer_id_]; ++i) {
63     inputs_[input_layer_id_][i] = pattern[i - 1];
64   }
65 }
66
67 void NeuralNetworkClassifier::propagate_forward_() {
68   for (unsigned int to = 1; to < layers_sizes_[1]; ++to) {
69     inputs_[1][to] = 0;
70     for (unsigned int from = 0; from < layers_sizes_[input_layer_id_]; ++from) {
71       if (from == bias_id_) {
72         inputs_[1][to] += 1 * weights_[input_layer_id_][from][to];
73       } else {
74         inputs_[1][to] += inputs_[input_layer_id_][from] * weights_[input_layer_id_][from][to
    ];
75       }
76     }
77   }
78
79   for (unsigned int layer = 2; layer < layers_quantity_; ++layer) {
80     for (unsigned int to = 1; to < layers_sizes_[layer]; ++to) {
81       inputs_[layer][to] = 0;
82       for (unsigned int from = 0; from < layers_sizes_[layer - 1]; ++from) {
83         if (from == bias_id_) {
84           inputs_[layer][to] += 1 * weights_[layer-1][from][to];
85         } else {
86           inputs_[layer][to] += calculate_g_function_(inputs_[layer - 1][from])
87                                 * weights_[layer - 1][from][to];
88         }
89       }
90     }
91   }
92 }
93
94 void NeuralNetworkClassifier::calculate_errors_(const vector<long double>& desired_output) {
95   // output layer
96   for (unsigned int i = 1; i < layers_sizes_[output_layer_id_]; ++i) {
97     errors_[output_layer_id_][i] = /*2 **/ (inputs_[output_layer_id_][i] - desired_output[i -
      1]);
98   }
```

```
99
100   // other layers
101   for (unsigned int layer = output_layer_id_ - 1; layer >= 1; --layer) {
102     for (unsigned int to = 1; to < layers_sizes_[layer]; ++to) {
103       errors_[layer][to] = 0;
104       for (unsigned int from = 1; from < layers_sizes_[layer + 1]; ++from) {
105         errors_[layer][to] += weights_[layer][to][from] * errors_[layer + 1][from];
106       }
107       errors_[layer][to] *= calculate_g_derivative_(inputs_[layer][to]);
108     }
109   }
110 }
111
112 void NeuralNetworkClassifier::update_weights_(const long double learning_rate,
113                         const long double alpha) {
114   for (unsigned int layer = 0; layer + 1 < layers_quantity_; ++layer) {
115     for (unsigned int from = 0; from < layers_sizes_[layer]; ++from) {
116       for (unsigned int to = 1; to < layers_sizes_[layer + 1]; ++to) {
117         long double delta = 0;
118         if (from == bias_id_) {
119           delta = 1 * errors_[layer + 1][to];
120         } else {
121           delta = calculate_g_function_(inputs_[layer][from]) * errors_[layer + 1][to];
122         }
123
124
125         weights_[layer][from][to] += -learning_rate * delta + alpha * inertia_[layer][from][
    to];
126         inertia_[layer][from][to] = -learning_rate * delta;
127       }
128     }
129   }
130 }
131
132 NeuralNetworkClassifier::NeuralNetworkClassifier() {}
133
134 NeuralNetworkClassifier::NeuralNetworkClassifier(const vector<unsigned int>& layers_sizes,
135                                     long double beta) {
136   beta_ = beta;
137   initialize_(layers_sizes);
138 }
139
140
141 void NeuralNetworkClassifier::prepare_to_training(const unsigned int seed,
142                           const long double weight_initialization_range,
143                           const long double threshold_initialization_range) {
144   Random::set_seed(seed);
145   initialize_random_weights_(weight_initialization_range);
146   initialize_random_thresholds_(threshold_initialization_range);
147 }
148
149 void NeuralNetworkClassifier::do_one_iteration_of_training(const vector< vector<long double>
    >& data,
150                               const vector< vector< long double > >& outputs,
151                               const long double alpha,
152                               const long double learning_rate) {
153   const unsigned int pattern_index = Random::get_random_number(data.size());
154
155   initialize_input_layer_(data[pattern_index]);
156   propagate_forward_();
```

```
157   calculate_errors_(outputs[pattern_index]);
158   update_weights_(learning_rate, alpha);
159 }
160
161 void NeuralNetworkClassifier::train(const vector< vector<long double> >& data,
162                     const vector< vector<long double> >& outputs,
163                     const unsigned int seed,
164                     const long long iterations,
165                     const long double alpha,
166                     const long double weight_initialization_range,
167                     const long double threshold_initialization_range,
168                     const long double learning_rate) {
169
170   prepare_to_training(seed, weight_initialization_range, threshold_initialization_range);
171
172   for (unsigned int iteration = 0; iteration < iterations; ++iteration) {
173     do_one_iteration_of_training(data, outputs, alpha, learning_rate);
174   }
175 }
176
177 vector<long double> NeuralNetworkClassifier::classify(const vector< long double >& pattern) {
178   initialize_input_layer_(pattern);
179   propagate_forward_();
180
181   return vector<long double> (inputs_.back().begin() + 1, inputs_.back().end());
182 }
183
184 long double NeuralNetworkClassifier::classify_one_output(const vector< long double >& pattern
      ) {
185   initialize_input_layer_(pattern);
186   propagate_forward_();
187
188   return inputs_.back()[1];
189 }
190
191 long double NeuralNetworkClassifier::calculate_classification_error_one_output
192                         (const vector< vector<long double> >& inputs,
193                           const vector< vector<long double> >& outputs) {
194   long double error = 0;
195
196   for (unsigned int i = 0; i < inputs.size(); ++i) {
197     long double my_answer = classify_one_output(inputs[i]);
198
199     if (my_answer < 0) {
200       my_answer = -1;
201     } else {
202       my_answer = 1;
203     }
204
205     long double real_answer = outputs[i][0];
206
207     error += fabs(real_answer - my_answer) / 2;
208   }
209
210   return error / inputs.size();
211 }
212
213 // lecture notes, chapter 4, page 110
214 long double NeuralNetworkClassifier::calculate_energy_one_output
215                         (const vector< vector<long double> >& inputs,
```

```
                          const vector< vector<long double> >& outputs) {
  long double energy = 0;

  for (unsigned int i = 0; i < inputs.size(); ++i) {
    long double my_answer = classify_one_output(inputs[i]);
    long double real_answer = outputs[i][0];
    energy += pow(real_answer - my_answer, 2);
  }

  return energy / 2;
}


long double NeuralNetworkClassifier::calculate_classification_error
                         (const vector< vector<long double> >& inputs,
                          const vector< vector<long double> >& outputs) {
  long double error = 0;

  for (unsigned int i = 0; i < inputs.size(); ++i) {
    vector<long double> my_answer = classify(inputs[i]);

    for (unsigned int j = 0; j < my_answer.size(); ++j) {
      if (my_answer[j] < 0) {
        my_answer[j] = -1;
      } else {
        my_answer[j] = 1;
      }
    }

    const vector<long double>& real_answer = outputs[i];

    error += (real_answer != my_answer);
  }

  return error / inputs.size();
}

// lecture notes, chapter 4, page 110
long double NeuralNetworkClassifier::calculate_energy
                         (const vector< vector<long double> >& inputs,
                          const vector< vector<long double> >& outputs) {
  long double energy = 0;

  for (unsigned int i = 0; i < inputs.size(); ++i) {
    vector<long double> my_answer = classify(inputs[i]);
    const vector<long double>& real_answer = outputs[i];

    for (unsigned int j = 0; j < my_answer.size(); ++j) {
      energy += pow(real_answer[j] - my_answer[j], 2);
    }
  }

  return energy / 2;
}



// overloads with struct Dataset

void NeuralNetworkClassifier::do_one_iteration_of_training(const Dataset& dataset,
```

```
276                                 const long double alpha,
277                                 const long double learning_rate) {
278   do_one_iteration_of_training(dataset.inputs, dataset.outputs, alpha, learning_rate);
279 }
280
281 void NeuralNetworkClassifier::train(const Dataset& dataset,
282                     const unsigned int seed,
283                     const long long iterations,
284                     const long double alpha,
285                     const long double weight_initialization_range,
286                     const long double threshold_initialization_range,
287                     const long double learning_rate) {
288   train(dataset.inputs, dataset.outputs, seed, iterations, alpha,
289       weight_initialization_range, threshold_initialization_range,
290       learning_rate);
291 }
292
293 long double NeuralNetworkClassifier::calculate_classification_error_one_output(const Dataset&
      dataset) {
294   return calculate_classification_error_one_output(dataset.inputs, dataset.outputs);
295 }
296
297 long double NeuralNetworkClassifier::calculate_energy_one_output(const Dataset& dataset) {
298   return calculate_energy_one_output(dataset.inputs, dataset.outputs);
299 }
300
301 long double NeuralNetworkClassifier::calculate_classification_error(const Dataset& dataset) {
302   return calculate_classification_error(dataset.inputs, dataset.outputs);
303 }
304
305 long double NeuralNetworkClassifier::calculate_energy(const Dataset& dataset) {
306   return calculate_energy(dataset.inputs, dataset.outputs);
307 }
```

H File `Output.h`:

```
1 #pragma once
2 #ifndef OUTPUT_H
3 #define OUTPUT_H
4
5 #include <vector>
6 using std::vector;
7
8 #include <string>
9 using std::string;
10
11 #include <fstream>
12
13 #include <algorithm>
14 #include <iomanip>
15
16 #include <direct.h>
17 #include <sstream>
18
19 template <class T>
20 std::ostream& operator << (std::ostream& out, const vector<T>& data) {
21   out << "{";
22   for (unsigned int i = 0; i < data.size(); ++i) {
23     if (i != 0) {
24       out << ", ";
```

```
    }
    out << std::fixed << std::setprecision(8) << data[i];
  }
  out << "}";

  return out;
}

template <class T, class U>
std::ostream& operator << (std::ostream& out, const std::pair<T, U>& data) {
  out << "{";
  out <<  std::fixed << std::setprecision(6) << data.first;
  out << ", " << std::fixed << std::setprecision(6) << data.second;
  out << "}";

  return out;
}

template <class T>
void save_to_file(const string& path, const vector< T >& data) {

  for (unsigned int i = 0; i < path.size(); ++i) {
    if (path[i] == '/' || path[i] == '\\') {
      _mkdir(path.substr(0, i + 1).c_str());
    }
  }

  std::fstream out(path, std::ostream::out);
  out << data;
  out.close();
}

template <class T>
string get_string(const T& x) {
  std::stringstream stream;
  stream << x;
  return stream.str();
}

#endif
```

I File PseudoRandomGenerator.h:

```
#pragma once
#ifndef PSEUDO_RANDOM_GENERATOR_H
#define PSEUDO_RANDOM_GENERATOR_H

#include "MT19937RNG.h"
#include <algorithm>

namespace Random {

  PseudoRandomNumberGenerator& get_generator();
  void set_seed(const unsigned int seed);
  unsigned int get_random_number();
  unsigned int get_random_number(const unsigned int upper_bound);
  unsigned int get_random_number_with_upper_bound(const unsigned int upper_bound);

  float get_random_float(); // range [0, 1)
  double get_random_double(); // range [0, 1)
```

```
18   long double get_random_long_double(); // range [0, 1)
19
20   // bounds are included, i.e. [lower_bound, upper_bound]
21   long double get_random_from_range(long double lower_bound, long double upper_bound);
22
23   bool get_true_with_probability(float probability);
24   bool get_true_with_probability(double probability);
25   bool get_true_with_probability(long double probability);
26
27   template<class T>
28   void random_shuffle(T begin, T end) {
29     return std::random_shuffle(begin, end, get_random_number_with_upper_bound);
30   }
31
32 } // namespace Random
33
34 #endif
```

J File PseudoRandomGenerator.cpp:

```
1  #include "PseudoRandomGenerator.h"
2
3  namespace Random {
4
5    PseudoRandomNumberGenerator& get_generator() {
6      static PseudoRandomNumberGenerator generator;
7      return generator;
8    }
9
10   void set_seed(const unsigned int seed) {
11     get_generator().initialize(seed);
12   }
13
14   unsigned int get_random_number() {
15     return get_generator().get_number();
16   }
17
18   unsigned int get_random_number(const unsigned int upper_bound) {
19     return get_generator().get_number() % upper_bound;
20   }
21
22   unsigned int get_random_number_with_upper_bound(const unsigned int upper_bound) {
23     return get_random_number(upper_bound);
24   }
25
26   float get_random_float() {  // range [0, 1)
27     const unsigned int MAX_UNSIGNED_INT = -1;
28
29     unsigned int number;
30     do {
31       number = get_random_number();
32     } while (number == MAX_UNSIGNED_INT);
33
34     return static_cast<float>(number) / MAX_UNSIGNED_INT;
35   }
36
37   double get_random_double() {  // range [0, 1)
38     const unsigned int MAX_UNSIGNED_INT = -1;
39
40     unsigned int number;
```

```
41      do {
42        number = get_random_number();
43      } while (number == MAX_UNSIGNED_INT);
44
45      return static_cast<double>(number) / MAX_UNSIGNED_INT;
46    }
47
48    long double get_random_long_double() {  // range [0, 1)
49      const unsigned int MAX_UNSIGNED_INT = -1;
50
51      unsigned int number;
52      do {
53        number = get_random_number();
54      } while (number == MAX_UNSIGNED_INT);
55
56      return static_cast<long double>(number) / MAX_UNSIGNED_INT;
57    }
58
59    // bounds are included, i.e. [lower_bound, upper_bound]
60    long double get_random_from_range(long double lower_bound, long double upper_bound) {
61      const unsigned int MAX_UNSIGNED_INT = -1;
62      const unsigned int number = get_random_number();
63
64      long double real = static_cast<long double>(number) / MAX_UNSIGNED_INT;
65      real *= (upper_bound - lower_bound);
66      return lower_bound + real;
67    }
68
69    bool get_true_with_probability(float probability) {
70      if (get_random_float() < probability) {
71        return true;
72      } else {
73        return false;
74      }
75    }
76
77    bool get_true_with_probability(double probability) {
78      if (get_random_double() < probability) {
79        return true;
80      } else {
81        return false;
82      }
83    }
84
85    bool get_true_with_probability(long double probability) {
86      if (get_random_long_double() < probability) {
87        return true;
88      } else {
89        return false;
90      }
91    }
92
93  } // namespace Random
```

K File MT19937RNG.h:

```
1  #pragma once
2  #ifndef MT19937RNG_H
3  #define MT19937RNG_H
4
```

```
#include <vector>
using std::vector;

class PseudoRandomNumberGenerator {
private:
    const static size_t length_ = 624;

    size_t index_;
    vector<unsigned int> MT_;

    void generate_numbers_();
public:

    PseudoRandomNumberGenerator();
    explicit PseudoRandomNumberGenerator(const unsigned int seed);
    explicit PseudoRandomNumberGenerator(const vector<unsigned int>& MT);

    void initialize (const unsigned int seed);

    unsigned int get_number();
};


#endif
```

L File MT19937RNG.cpp:

```
#include "MT19937RNG.h"

void PseudoRandomNumberGenerator::generate_numbers_() {
    for (size_t index = 0; index < length_; ++index) {
        unsigned int y = (MT_[index] & (1u << 31)) + (MT_[(index + 1) % length_] & ((1u << 31) - 1));
        MT_[index] = MT_[(index + 397) % length_] ^ (y >> 1);
        if (y % 2 != 0) {
            MT_[index] ^= 2567483615;
        }
    }
}

PseudoRandomNumberGenerator::PseudoRandomNumberGenerator() {
    initialize(0);
}

PseudoRandomNumberGenerator::PseudoRandomNumberGenerator(const unsigned int seed) {
    initialize(seed);
}

PseudoRandomNumberGenerator::PseudoRandomNumberGenerator(const vector<unsigned int>& MT) {
    MT_ = MT;
    index_ = 0;
}

void PseudoRandomNumberGenerator::initialize (const unsigned int seed) {
    index_ = 0;
    MT_.resize(length_);
    MT_[0] = seed;
    for (size_t index = 1; index < length_; ++index) {
        MT_[index] = 1812433253u * (MT_[index - 1] ^ (MT_[index - 1] >> 30)) + index;
    }
```

```
}

unsigned int PseudoRandomNumberGenerator::get_number() {
    if (index_ == 0) {
        generate_numbers_();
    }

    unsigned int result = MT_[index_];
    result ^= (result >> 11);
    result ^= (result << 7) & 2636928640;
    result ^= (result << 15) & 4022730752;
    result ^= (result >> 18);

    index_ = (index_ + 1) % length_;

    return result;
}
```

M File Dataset.h:

```
#pragma once
#ifndef DATASET_H
#define DATASET_H

#include <vector>
using std::vector;

#include <fstream>

#include <sstream>
#include <string>
using std::string;

struct Dataset {
  vector< vector<long double> > inputs, outputs;

  vector<long double> calculate_mean() const;
  vector<long double> calculate_variance(vector<long double> mean = vector<long double>())
   const;


  void shift_mean(const vector<long double>& shift);
  void normalize_to_zero_mean(vector<long double> mean = vector<long double>());
  void normalize_to_unity_variance(vector<long double> variance = vector<long double>());

  void append(const Dataset& other);

  unsigned int get_input_variables_quantity() const;
  unsigned int get_output_variables_quantity() const;

  unsigned int size() const;

  void random_shuffle(const unsigned int seed = 123);

  Dataset cutoff_back(const unsigned int size);
};

Dataset read_dataset(const string& path, unsigned int output_column_index, const char
   delimiter = ',') ;

```

```
39 #endif
```

N File Dataset.cpp:

```
1  #include "Dataset.h"
2
3  #include <vector>
4  using std::vector;
5
6  #include <algorithm>
7  using std::pair;
8  using std::make_pair;
9
10 #include <direct.h>
11 #include <iomanip>
12
13 #include "Output.h"
14
15 #include <sstream>
16 #include "PseudoRandomGenerator.h"
17
18 vector<long double> Dataset::calculate_mean() const {
19   vector<long double> mean(inputs[0].size(), 0);
20   for (unsigned int i = 0; i < inputs.size(); ++i) {
21     for (unsigned int j = 0; j < inputs[i].size(); ++j) {
22       mean[j] += inputs[i][j];
23     }
24   }
25
26   for (unsigned int i = 0; i < mean.size(); ++i) {
27     mean[i] /= inputs.size();
28   }
29
30   return mean;
31 }
32
33 vector<long double> Dataset::calculate_variance(vector<long double> mean) const {
34
35   if (mean.empty()) {
36     mean = calculate_mean();
37   }
38   vector<long double> variance(inputs[0].size(), 0);
39
40   for (unsigned int i = 0; i < inputs.size(); ++i) {
41     for (unsigned int j = 0; j < inputs[i].size(); ++j) {
42       long double difference = inputs[i][j] - mean[j];
43       variance[j] += difference * difference;
44     }
45   }
46
47   for (unsigned int i = 0; i < mean.size(); ++i) {
48     variance[i] /= (inputs.size() - 1);
49   }
50
51   return variance;
52 }
53
54 void Dataset::shift_mean(const vector<long double>& shift) {
55   for (unsigned int i = 0; i < inputs.size(); ++i) {
56     for (unsigned int j = 0; j < inputs[i].size(); ++j) {
```

```
57        inputs[i][j] += shift[j];
58      }
59    }
60  }
61  void Dataset::normalize_to_zero_mean(vector<long double> mean) {
62    if (mean.empty()) {
63      mean = calculate_mean();
64    }
65
66    vector<long double> shift(mean.size());
67    for (unsigned int i = 0; i < shift.size(); ++i) {
68      shift[i] = -mean[i];
69    }
70
71    shift_mean(shift);
72  }
73
74  void Dataset::normalize_to_unity_variance(vector<long double> variance) {
75    if (variance.empty()) {
76      variance = calculate_variance();
77    }
78
79    for (unsigned int i = 0; i < inputs.size(); ++i) {
80      for (unsigned int j = 0; j < inputs[i].size(); ++j) {
81        inputs[i][j] /= sqrt(variance[j]);
82      }
83    }
84  }
85
86  void Dataset::append(const Dataset& other) {
87    for (unsigned int i = 0; i < other.size(); ++i) {
88      inputs.push_back(other.inputs[i]);
89      outputs.push_back(other.outputs[i]);
90    }
91  }
92
93  unsigned int Dataset::get_input_variables_quantity() const {
94    return inputs[0].size();
95  }
96  unsigned int Dataset::get_output_variables_quantity() const {
97    return outputs[0].size();
98  }
99
100 unsigned int Dataset::size() const {
101   return inputs.size();
102 }
103
104 void Dataset::random_shuffle(const unsigned int seed) {
105   Random::set_seed(seed);
106
107   vector<unsigned int> indices(size());
108   for (unsigned int i = 0; i < indices.size(); ++i) {
109     indices[i] = i;
110   }
111
112   Random::random_shuffle(indices.begin(), indices.end());
113
114   vector< vector<long double> > new_inputs(size()), new_outputs(size());
115   for (unsigned int i = 0; i < indices.size(); ++i) {
116     new_inputs[i] = inputs[indices[i]];
```

```
117     new_outputs[i] = outputs[indices[i]];
118   }
119
120   inputs.swap(new_inputs);
121   outputs.swap(new_outputs);
122 }
123
124 Dataset Dataset::cutoff_back(const unsigned int size) {
125   Dataset result;
126   for (unsigned int i = this->size() - size; i < this->size(); ++i) {
127     result.inputs.push_back(inputs[i]);
128     result.outputs.push_back(outputs[i]);
129   }
130
131   const unsigned int old_size = this->size();
132   inputs.resize(old_size - size);
133   outputs.resize(old_size - size);
134
135   return result;
136 }
137
138 Dataset read_dataset(const string& path, unsigned int output_column_index, const char
      delimiter) {
139
140   Dataset result;
141
142   std::ifstream in(path, std::istream::in);
143   std::stringstream stream;
144
145   vector<long double> parsed;
146   string line;
147   while (std::getline(in, line)) {
148
149     if (line.empty()) {
150       break;
151     }
152
153
154     for (unsigned int i = 0; i < line.size(); ++i) {
155       if (line[i] == delimiter) {
156         line[i] = ' ';
157       }
158     }
159
160     stream.clear();
161     stream.str(line);
162
163
164     long double number;
165
166     while (stream >> number) {
167       parsed.push_back(number);
168     }
169
170     long double output = parsed[output_column_index];
171     result.outputs.push_back(vector<long double> (1, output));
172
173     parsed.erase(parsed.begin() + output_column_index);
174     result.inputs.push_back(parsed);
175
```

```
176       parsed.clear();
177     }
178
179     in.close();
180
181     return result;
182   }
```