С.Короткий

Нейронные сети: обучение без учителя

В статье рассмотрены алгоритмы обучения искусственных нейронных сетей без учителя. Приведена библиотека классов на C++ и тестовый пример.

Рассмотренный в [1] алгоритм обучения нейронной сети с помощью процедуры обратного распространения подразумевает наличие некоего внешнего звена, предоставляющего сети кроме входных так же и целевые выходные образы. Алгоритмы, пользующиеся подобной концепцией, называются алгоритмами обучения с учителем. Для их успешного функционирования необходимо наличие экспертов, создающих на предварительном этапе для каждого входного образа эталонный выходной. Так как создание искусственного интеллекта движется по пути копирования природных прообразов, ученые не прекращают спор на тему, можно ли считать алгоритмы обучения с учителем натуральными или же они полностью искусственны. Например, обучение человеческого мозга, на первый взгляд, происходит без учителя: на зрительные, слуховые, тактильные и прочие рецепторы поступает информация извне, и внутри нервной системы происходит некая самоорганизация. Однако, нельзя отрицать и того, что в жизни человека не мало учителей – и в буквальном, и в переносном смысле, – которые координируют внешние воздействия. Вместе в тем, чем бы ни закончился спор приверженцев этих двух концепций обучения, они обе имеют право на существование.

Главная черта, делающая обучение без учителя привлекательным, – это его "самостоятельность". Процесс обучения, как и в случае обучения с учителем, заключается в подстраивании весов синапсов. Некоторые алгоритмы, правда, изменяют и структуру сети, то есть количество нейронов и их взаимосвязи, но такие преобразования правильнее назвать более широким термином – самоорганизацией, и в рамках данной статьи они рассматриваться не будут. Очевидно, что подстройка синапсов может проводиться только на основании информации, доступной в нейроне, то есть его состояния и уже имеющихся весовых коэффициентов. Исходя из этого соображения и, что более важно, по аналогии с известными принципами самоорганизации нервных клеток[2], построены алгоритмы обучения Хебба.

Сигнальный метод обучения Хебба заключается в изменении весов по следующему правилу:

$$w_{ij}(t) = w_{ij}(t-1) + \alpha \cdot y_i^{(n-1)} \cdot y_j^{(n)} \qquad (1)$$

где $y_i^{(n-1)}$ – выходное значение нейрона i слоя (n-1), $y_j^{(n)}$ – выходное значение нейрона j слоя n; $w_{ij}(t)$ и $w_{ij}(t-1)$ – весовой коэффициент синапса, соединяющего эти нейроны, на итерациях t и t-1 соответственно; α – коэффициент скорости обучения. Здесь и далее, для общности, под n подразумевается произвольный слой сети. При обучении по данному методу усиливаются связи между возбужденными нейронами.

Существует также и дифференциальный метод обучения Хебба.

$$w_{ij}(t) = w_{ij}(t-1) + \alpha \cdot \left[y_i^{(n-1)}(t) - y_i^{(n-1)}(t-1) \right] \cdot \left[y_j^{(n)}(t) - y_j^{(n)}(t-1) \right] \qquad (2)$$

Здесь $y_i^{(n-1)}(t)$ и $y_i^{(n-1)}(t-1)$ – выходное значение нейрона i слоя n-1 соответственно на итерациях t и t-1; $y_j^{(n)}(t)$ и $y_j^{(n)}(t-1)$ – то же самое для нейрона j слоя n. Как видно из формулы (2), сильнее всего обучаются синапсы, соединяющие те нейроны, выходы которых наиболее динамично изменились в сторону увеличения.

Полный алгоритм обучения с применением вышеприведенных формул будет выглядеть так:

1. На стадии инициализации всем весовым коэффициентам присваиваются небольшие случайные значения.

2. На входы сети подается входной образ, и сигналы возбуждения распространяются по всем слоям согласно принципам классических прямопоточных (feedforward) сетей[1], то есть для каждого нейрона рассчитывается взвешенная сумма его входов, к которой затем применяется активационная (передаточная) функция нейрона, в результате чего получается его выходное значение $y_i^{(n)}$, i=0...M_i-1, где M_i – число нейронов в слое i; n=0...N-1, а N – число слоев в сети.

3. На основании полученных выходных значений нейронов по формуле (1) или (2) производится изменение весовых коэффициентов.

4. Цикл с шага 2, пока выходные значения сети не застабилизируются с заданной точностью. Применение этого нового способа определения завершения обучения, отличного от использовавшегося для сети обратного распространения, обусловлено тем, что подстраиваемые значения синапсов фактически не ограничены.

На втором шаге цикла попеременно предъявляются все образы из входного набора.

Следует отметить, что вид откликов на каждый класс входных образов не известен заранее и будет представлять собой произвольное сочетание состояний нейронов выходного слоя, обусловленное случайным распределением весов на стадии инициализации. Вместе с тем, сеть способна обобщать схожие образы, относя их к одному классу. Тестирование обученной сети позволяет определить топологию классов в выходном слое. Для приведения откликов обученной сети к удобному представлению можно дополнить сеть одним слоем, который, например, по алгоритму обучения однослойного перцептрона необходимо заставить отображать выходные реакции сети в требуемые образы.

Другой алгоритм обучения без учителя – алгоритм Кохонена – предусматривает подстройку синапсов на основании их значений от предыдущей итерации.

$$w_{ij}(t) = w_{ij}(t-1) + \alpha \cdot \left[y_i^{(n-1)} - w_{ij}(t-1) \right] \qquad (3)$$

Из вышеприведенной формулы видно, что обучение сводится к минимизации разницы между входными сигналами нейрона, поступающими с выходов нейронов предыдущего слоя $y_i^{(n-1)}$, и весовыми коэффициентами его синапсов.

Полный алгоритм обучения имеет примерно такую же структуру, как в методах Хебба, но на шаге 3 из всего слоя выбирается нейрон, значения синапсов которого максимально походят на входной образ, и подстройка весов по формуле (3) проводится только для него. Эта, так называемая, аккредитация может сопровождаться затормаживанием всех остальных нейронов слоя и введением выбранного нейрона в насыщение. Выбор такого нейрона может

осуществляться, например, расчетом скалярного произведения вектора весовых коэффициентов с вектором входных значений. Максимальное произведение дает выигравший нейрон.

Другой вариант – расчет расстояния между этими векторами в p-мерном пространстве, где p – размер векторов.

$$D_j = \sqrt{\sum_{i=0}^{p-1} (y_i^{(n-1)} - w_{ij})^2}\ , \tag{4}$$

где j – индекс нейрона в слое n, i – индекс суммирования по нейронам слоя (n-1), w_{ij} – вес синапса, соединяющего нейроны; выходы нейронов слоя (n-1) являются входными значениями для слоя n. Корень в формуле (4) брать не обязательно, так как важна лишь относительная оценка различных D_j.

В данном случае, "побеждает" нейрон с наименьшим расстоянием. Иногда слишком часто получающие аккредитацию нейроны принудительно исключаются из рассмотрения, чтобы "уравнять права" всех нейронов слоя. Простейший вариант такого алгоритма заключается в торможении только что выигравшего нейрона.

При использовании обучения по алгоритму Кохонена существует практика нормализации входных образов, а так же – на стадии инициализации – и нормализации начальных значений весовых коэффициентов.

$$x_i = x_i \,/\, \sqrt{\sum_{j=0}^{n-1} x_j^2}\ , \tag{5}$$

где x_i – i-ая компонента вектора входного образа или вектора весовых коэффициентов, а n – его размерность. Это позволяет сократить длительность процесса обучения.

Инициализация весовых коэффициентов случайными значениями может привести к тому, что различные классы, которым соответствуют плотно распределенные входные образы, сольются или, наоборот, раздробятся на дополнительные подклассы в случае близких образов одного и того же класса. Для избежания такой ситуации используется метод выпуклой комбинации[3]. Суть его сводится к тому, что входные нормализованные образы подвергаются преобразованию:

$$x_i = \alpha(t) \cdot x_i + (1 - \alpha(t)) \cdot \frac{1}{\sqrt{n}}, \tag{6}$$

где x_i – i-ая компонента входного образа, n – общее число его компонент, $\alpha(t)$ – коэффициент, изменяющийся в процессе обучения от нуля до единицы, в результате чего вначале на входы сети подаются практически одинаковые образы, а с течением времени они все больше сходятся к исходным. Весовые коэффициенты устанавливаются на шаге инициализации равными величине

$$w_o = \frac{1}{\sqrt{n}}, \tag{7}$$

где n – размерность вектора весов для нейронов инициализируемого слоя.

На основе рассмотренного выше метода строятся нейронные сети особого типа – так называемые самоорганизующиеся структуры – self-organizing feature maps (этот устоявшийся перевод с английского, на мой взгляд, не очень удачен, так как, речь идет не об изменении структуры сети, а только о подстройке синапсов). Для них после выбора из слоя n нейрона j с минимальным расстоянием D_j (4) обучается по формуле (3) не только этот нейрон, но и его соседи, расположенные в окрестности R. Величина R на первых итерациях очень большая, так что обучаются все нейроны, но с течением времени она уменьшается до нуля. Таким образом, чем ближе конец обучения, тем точнее определяется группа нейронов, отвечающих каждому классу образов. В приведенной ниже программе используется именно этот метод обучения.

Развивая объектно-ориентированный подход в моделировании нейронных сетей, рассмотренный в [1], для программной реализации сетей, использующих алгоритм обучения без учителя, были разработаны отдельные классы объектов типа нейрон, слой и сеть, названия которых снабжены суффиксом UL. Они наследуют основные свойства от соответствующих объектов прямопоточной сети, описанной в [1]. Фрагмент заголовочного файла с описаниями классов и функций для таких сетей представлен на листинге 1.

Как видно из него, в классе NeuronUL в отличие от NeuronBP отсутствуют обратные связи и инструменты их поддержания, а по сравнению с NeuronFF здесь появилось лишь две новых переменных – delta и inhibitory. Первая из них хранит расстояние, рассчитываемое по формуле (4), а вторая – величину заторможенности нейрона. В классе NeuronUL существует два конструктора – один, используемый по умолчанию, – не имеет параметров, и к созданным с помощью него нейронам необходимо затем применять метод _allocateNeuron класса NeuronFF. Другой сам вызывает эту функцию через соответствующий конструктор NeuronFF. Метод Propagate является почти полным аналогом одноименного метода из NeuronFF, за исключением вычисления величин delta и inhibitory. Методы Normalize и Equalize выполняют соответственно нормализацию значений весовых коэффициентов по формуле (5) и их установку согласно (7). Метод CountDistance вычисляет расстояние (4). Следует особо отметить, что в классе отсутствует метод IsConverged, что, объясняется, как говорилось выше, различными способами определения факта завершения обучения. Хотя в принципе написать такую функцию не сложно, в данной программной реализации завершение обучения определяется по "телеметрической" информации, выводимой на экран, самим пользователем. В представленном же вместе со статьей тесте число итераций, выполняющих обучение, вообще говоря, эмпирически задано равным 3000.

В состав класса LayerUL входит массив нейронов neurons и переменная с размерностью массивов синапсов – neuronrang. Метод распределения нейронов – внешний или внутренний – определяется тем, как создавался слой. Этот признак хранится в переменной allocation. Конструктор LayerUL(unsigned, unsigned) сам распределяет память под нейроны, что соответствует внутренней инициализации; конструктор LayerUL(NeuronUL _FAR *, unsigned, unsigned) создает слой из уже готового, внешнего массива нейронов. Все методы этого класса

аналогичны соответствующим методам класса LayerFF и, в большинстве своем, используют одноименные методы класса NeuronUL.

В классе NetUL также особое внимание необходимо уделить конструкторам. Один из них – NetUL(unsigned n) создает сеть из n пустых слоев, которые затем необходимо заполнить с помощью метода SetLayer. Конструктор NetUL(unsigned n, unsigned n1, ...) не только создает сеть из n слоев, но и распределяет для них соответствующее число нейронов с синапсами, обеспечивающими полносвязность сети. После создания сети необходимо связать все нейроны с помощью метода FullConnect. Как и в случае сети обратного распространения, сеть можно сохранять и загружать в/из файла с помощью методов SaveToFile, LoadFromFile. Из всех остальных методов класса новыми по сути являются лишь NormalizeNetInputs и ConvexCombination. Первый из них нормализует входные вектора, а второй реализует преобразование выпуклой комбинации (6).

В конце заголовочного файла описаны глобальные функции. SetSigmoidTypeUL, SetSigmoidAlfaUL и SetDSigmaUL аналогичны одноименным функциям для сети обратного распространения. Функция SetAccreditationUL устанавливает режим, при котором эффективность обучения нейронов, попавших в окружение наиболее возбужденного на данной итерации нейрона, пропорциональна функции Гаусса от расстояния до центра области обучения. Если этот режим не включен, то все нейроны попавшие в область с текущим радиусом обучения одинаково быстро подстраивают свои синапсы, причем область является квадратом со стороной, равной радиусу обучения. Функция SetLearnRateUL устанавливает коэффициент скорости обучения, а SetMaxDistanceUL – радиус обучения. Когда он равен 0 – обучается только один нейрон. Функции SetInhibitionUL и SetInhibitionFresholdUL устанавливают соответственно длительность торможения и величину возбуждения, его вызывающего.

Тексты функций помещены в файле neuro_mm.cpp, представленном в листинге 2. Кроме него в проект тестовой программы входят также модули neuron_ff.cpp и subfun.cpp, описанные в [1]. Главный модуль, neuman7.cpp приведен в листинге 3. Программа компилировалась с помощью компилятора Borland C++ 3.1 в модели LARGE.

Тестовая программа демонстрирует христоматийный пример обучения самонастраивающейся сети следующей конфигурации. Входной слой состоит из двух нейронов, значения аксонов которых генерируются вспомогательной функцией на основе генератора случайных чисел. Выходной слой имеет размер 10 на 10 нейронов. В процессе обучения он приобретает свойства упорядоченной структуры, в которой величины синапсов нейронов плавно меняются вдоль двух измерений, имитируя двумерную сетку координат. Благодаря новой функции DigiShow и выводу индексов X и Y выигравшего нейрона, пользователь имеет возможность убедиться, что значения на входе сети достаточно точно определяют позицию точки максимального возбуждения на ее выходе.

Необходимо отметить, что обучение без учителя гораздо более чувствительно к выбору оптимальных параметров, нежели обучение с учителем. Во-первых, его качество сильно зависит от начальных величин синапсов. Во-вторых, обучение критично к выбору радиуса обучения и

скорости его изменения. И наконец, разумеется, очень важен характер изменения собственно коэффициента обучения. В связи с этим пользователю, скорее всего, потребуется провести предварительную работу по подбору оптимальных параметров обучения сети.

Несмотря на некоторые сложности реализации, алгоритмы обучения без учителя находят обширное и успешное применение. Например, в [4] описана многослойная нейронная сеть, которая по алгоритму самоорганизующейся структуры обучается распознавать рукописные символы. Возникающее после обучения разбиение на классы может в случае необходимости уточняться с помощью обучения с учителем. По сути дела, по алгоритму обучения без учителя функционируют и наиболее сложные из известных на сегодняшний день искусственные нейронные сети – когнитрон и неокогнитрон, – максимально приблизившиеся в своем воплощении к структуре мозга. Однако они, конечно, существенно отличаются от рассмотренных выше сетей и намного более сложны. Тем не менее, на основе вышеизложенного материала можно создать реально действующие системы для распознавания образов, сжатия информации, автоматизированного управления, экспертных оценок и много другого.

Литература

1. С.Короткий, Нейронные сети: алгоритм обратного распространения.
2. Ф.Блум, А.Лейзерсон, Л.Хофстедтер, Мозг, разум и поведение, М., Мир, 1988.
3. Ф.Уоссермен, Нейрокомпьютерная техника, М., Мир, 1992.
4. Keun-Rong Hsieh and Wen-Tsuen Chen, A Neural Network Model which Combines Unsupervised and Supervised Learning, IEEE Trans. on Neural Networks, vol.4, No.2, march 1993.

Листинг 1

```
// FILE neuro_mm.h
#include "neuro.h" // описание базовых классов

class LayerUL;
class NetUL;

class NeuronUL: public NeuronFF
{                                friend LayerUL;
                                 friend NetUL;
 float delta;
 unsigned inhibitory;
public:
 NeuronUL(unsigned num_inputs);
 NeuronUL(void){delta=0.;};
 ~NeuronUL();
// int  IsConverged(void); // можно реализовать
 virtual void Propagate(void);
 void Equalize(void);
 void Normalize(void);
 float CountDistance(void);
 void  SetInhibitory(unsigned in){inhibitory=in;};
 unsigned GetInhibitory(void){return inhibitory;};
};

class LayerUL: public LayerFF
{                                friend NetUL;
 NeuronUL _FAR *neurons;
 unsigned neuronrang;
 int  allocation;
 int imax, imaxprevious1;
public:
 LayerUL(unsigned nRang, unsigned nSinapses);
 LayerUL(NeuronUL _FAR *Neu, unsigned nRang,
          unsigned nSinapses);
 LayerUL(void)
 {
  neurons=NULL; neuronrang=0; allocation=EXTERN;
  imax=imaxprevious1=-1;
 };
 ~LayerUL();
 void Propagate(void);
 void Randomize(float);
 void Normalize(void);
 void NormalizeSynapses(void);
 void Equalize(void);
 virtual void Show(void);
 virtual void DigiShow(void);
 virtual void PrintSynapses(int,int);
 virtual void PrintAxons(int x, int y, int
direction);
 void TranslateAxons(void);
 NeuronUL *GetNeuron(unsigned n)
 {
  if(neurons && (n<rang)) return &neurons[n];
  else return NULL;
 };
};

class NetUL: public SomeNet
{
 LayerUL _FAR * _FAR *layers;
 // 1-й слой - входной, без синапсов
public:
 NetUL(void) {layers=NULL;};
 NetUL(unsigned nLayers);
 NetUL(unsigned n, unsigned n1, ...);
 ~NetUL();
 int SetLayer(unsigned n, LayerUL _FAR *pl);
 LayerUL *GetLayer(unsigned n)
  {if(n<rang) return layers[n]; else return NULL; }
 int FullConnect(void);
 void Propagate(void);
 void SetNetInputs(float _FAR *mvalue);
 void Learn(void);
 void Randomize(float);
 void Normalize(void);
 void NormalizeSynapses(void);
 void Equalize(void);
 int  SaveToFile(unsigned char *file);
 int  LoadFromFile(unsigned char *file);
 int  LoadNextPattern(float _FAR *IN);
 void SetLearnCycle(unsigned l){learncycle=l;};
 void virtual PrintSynapses(int x=0,...){};
 float virtual Change(float In);
 void AddNoise(void);
 void NormalizeNetInputs(float _FAR *mv);
 void ConvexCombination(float *In, float step);
 int  Reverse(NetUL **Net);
};

int SetSigmoidTypeUL(int st);
float SetSigmoidAlfaUL(float Al);
float SetLearnRateUL(float lr);
unsigned SetDSigmaUL(unsigned d);
void SetAccreditationUL(int ac);
float SetMaxDistanceUL(float md);
void SetInhibitionUL(int in);
float SetInhibitionFreshold(float f);
```

Листинг 2

```
// FILE neuro_mm.cpp FOR neuro_mm.prj
#include <stdlib.h>
#include <alloc.h>
#include <math.h>
#include <string.h>
#include <stdarg.h>
#include <values.h>

#include "neuro_mm.h"
#include "colour.h"
#define MAXDISTANCE MAXFLOAT

static int      SigmoidType=ORIGINAL;
static int      Accreditation=0;
static float    SigmoidAlfa=1.;
static float    LearnRate=0.25;
static unsigned dSigma=0;
static float    MaxDistance=MAXDISTANCE;
static int      Inhibition=0;
static float    InFreshold=0.0;

int SetSigmoidTypeUL(int st)
{
int i;
 i=SigmoidType;
 SigmoidType=st;
 return i;
}

void SetAccreditationUL(int ac)
{
 Accreditation=ac;
}

// число циклов на которые нейрон тормозится
// после возбуждения
void SetInhibitionUL(int in)
{
 Inhibition=in;
}

// порог возбуждения, инициирующий торможение
float SetInhibitionFreshold(float f)
{
float a;
 a=InFreshold;
 InFreshold=f;
 return a;
}

float SetSigmoidAlfaUL(float Al)
{
float a;
 a=SigmoidAlfa;
 SigmoidAlfa=Al;
 return a;
}

float SetMaxDistanceUL(float md)
{
float a;
 a=MaxDistance;
 if(md<1.0) Accreditation=0;
 MaxDistance=md;
 return a;
}

float SetLearnRateUL(float lr)
{
float a;
 a=LearnRate;
 LearnRate=lr;
 return a;
}

unsigned SetDSigmaUL(unsigned d)
{
unsigned u;
 u=dSigma;
 dSigma=d;
 return u;
}

NeuronUL::~NeuronUL()
{
// dummy
}

NeuronUL::NeuronUL(unsigned num_inputs)
         :NeuronFF(num_inputs)
{
 delta=0.; inhibitory=0;
}

void NeuronUL::Propagate(void)
{
 state=0.;
 for(unsigned i=0;i<rang;i++)
  state+=(*inputs[i]*2)*(synapses[i]*2);
 // поправка на использование логики ±0.5
 state/=2;
 axon=Sigmoid();

 if(Inhibition==0) return;

 if(axon>InFreshold) // возбуждение
 {
  if(inhibitory<=0)  // пока не заторможен
   inhibitory=axon*Inhibition/0.5; // тормозим
 }

 // постепенное восстановление возможности
возбуждаться
 if(inhibitory>0) inhibitory--;
```

```
}

void NeuronUL::Equalize(void)
{
float sq=1./sqrt(rang);
 for(int i=0;i<rang;i++) synapses[i]=sq-0.5;
}

void NeuronUL::Normalize(void)
{
float s=0;
 for(int i=0;i<rang;i++) s+=synapses[i]*synapses[i];
 s=sqrt(s);
 if(s) for(int i=0;i<rang;i++) synapses[i]/=s;
}

LayerUL::LayerUL(unsigned nRang, unsigned nSynapses)
{
 allocation=EXTERN; status=ERROR; neuronrang=0;
 if(nRang==0) return;
 neurons=new NeuronUL[nRang];
 if(neurons==NULL) return;
 for(unsigned i=0;i<nRang;i++)
 {
  neurons[i].InitNeuron(nSynapses);
  neurons[i].SetInhibitory(0.0);
  if(neurons[i].GetStatus()==ERROR)
  {
   status=ERROR;
   return;
  }
 }
 rang=nRang;
 neuronrang=nSynapses;
 allocation=INNER;
 name=NULL; status=OK;
 imax=/*-imaxprevious1=*/-1;
}

LayerUL::LayerUL(NeuronUL _FAR *Neu, unsigned nRang,
                      unsigned nSynapses)
{
 neurons=NULL; neuronrang=0; allocation=EXTERN;
 for(unsigned i=0;i<nRang;i++)
  if(Neu[i].rang!=nSynapses) status=ERROR;
 if(status==OK)
 {
  neurons=Neu;
  rang=nRang;
  neuronrang=nSynapses;
  imax=/*-imaxprevious1=*/-1;
 }
}

LayerUL::~LayerUL(void)
{
 if(allocation==INNER)
 {
  if(neurons!=NULL)
  {
   for(unsigned i=0;i<rang;i++)
    neurons[i]._deallocate();
   delete [] neurons; neurons=NULL;
  }
 }
}

void LayerUL::Randomize(float range)
{
 for(unsigned i=0;i<rang;i++)
  neurons[i].Randomize(range);
}

void LayerUL::Equalize(void)
{
 for(unsigned i=0;i<rang;i++)
  neurons[i].Equalize();
}

void LayerUL::NormalizeSynapses(void)
{
 for(unsigned i=0;i<rang;i++)
  neurons[i].Normalize();
}

void LayerUL::Normalize(void)
{
float s=0.;
 for(unsigned i=0;i<rang;i++)
  s+=(neurons[i].axon+0.5)*(neurons[i].axon+0.5);
 s=sqrt(s);
 if(s) for(i=0;i<rang;i++)
  neurons[i].axon=(neurons[i].axon+0.5)/s-0.5;
}

void LayerUL::Show(void)
{
unsigned char sym[5]={GRAFCHAR_EMPTYBLACK,
GRAFCHAR_DARKGRAY, GRAFCHAR_MIDDLEGRAY,
GRAFCHAR_LIGHTGRAY, GRAFCHAR_SOLIDWHITE };
int i,j;
 if(y && name) for(i=0;i<strlen(name);i++)
  out_char(x+i,y-1,name[i],3);
 out_char(x,y,GRAFCHAR_UPPERLEFTCORNER,15);
 for(i=0;i<2*dx;i++)
  out_char(x+1+i,y,GRAFCHAR_HORIZONTALLINE,15);
 out_char(x+1+i,y,GRAFCHAR_UPPERRIGHTCORNER,15);
 for(j=0;j<dy;j++)
 {
  out_char(x,y+1+j,GRAFCHAR_VERTICALLINE,15);
  for(i=0;i<2*dx;i++)
  {
   int n=(int) ((neurons[j*dx+i/2].axon+0.4999)*5);
   if(n<0) n=0;
   if(n>=5) n=4;
   if(j*dx+i/2<rang)
    out_char(x+1+i, y+1+j, sym[n], 15);
  }
  out_char(x+1+i, y+1+j,GRAFCHAR_VERTICALLINE,15);
 }

 out_char(x,y+j+1,GRAFCHAR_BOTTOMLEFTCORNER,15);
 for(i=0;i<2*dx;i++)
  out_char(x+i+1,y+j+1,GRAFCHAR_HORIZONTALLINE,15);
 out_char(x+1+i,y+j+1,
GRAFCHAR_BOTTOMRIGHTCORNER,15);
}

// вывод уровня возбуждения цветом и цифрами
// одновременно
void LayerUL::DigiShow(void)
{
int i,j;
char cc[3];

 if(y && name) for(i=0;i<strlen(name);i++)
  out_char(x+i,y-1,name[i],3);

 out_char(x,y,GRAFCHAR_UPPERLEFTCORNER,15);
 for(i=0;i<2*dx;i++)
  out_char(x+1+i,y,GRAFCHAR_HORIZONTALLINE,15);
 out_char(x+1+i,y,GRAFCHAR_UPPERRIGHTCORNER,15);

 for(j=0;j<dy;j++)
 {
  out_char(x,y+1+j,GRAFCHAR_VERTICALLINE,15);
  for(i=0;i<2*dx;i++)
  {
   int n=(int)
((neurons[j*dx+i/2].axon+0.4999)*100);
   if(n<0) n=0;
   if(n>=100) n=99;
   sprintf(cc,"%02d",n);

   if(j*dx+i/2<rang)
   {
    int a;
    if(n>=70) a=CGRAY;
    else if(n>=50) a=CCYAN;
    else if(n>=30) a=CBLUE;
    else a=0;

    if(i%2==0)
    out_char(x+1+i, y+1+j, cc[0], 15 | a);
    else
    out_char(x+1+i, y+1+j, cc[1], 15 | a);
   }
  }
  out_char(x+1+i, y+1+j,GRAFCHAR_VERTICALLINE,15);
 }

 out_char(x,y+j+1,GRAFCHAR_BOTTOMLEFTCORNER,15);
 for(i=0;i<2*dx;i++)
  out_char(x+i+1,y+j+1,GRAFCHAR_HORIZONTALLINE,15);
 out_char(x+1+i,y+j+1,
GRAFCHAR_BOTTOMRIGHTCORNER,15);
}

void LayerUL::PrintSynapses(int x, int y)
{
 for(unsigned i=0;i<rang;i++)
  neurons[i].PrintSynapses(x,y+i);
}

void LayerUL::PrintAxons(int x, int y, int
direction)
{
 for(unsigned i=0;i<rang;i++)
  neurons[i].PrintAxons(x+8*i*direction,
                        y+i*(!direction));
}

float NeuronUL::CountDistance(void)
{
int i;
float s=0.0;
 for(i=0;i<rang;i++)
  s+=fabs(*inputs[i]-synapses[i]);
 delta=s;
 return delta;
}

void LayerUL::Propagate(void)
{
unsigned i,j;
float fmax, f;
int cx, cy, nx, ny;

 for(i=0;i<rang;i++)
  neurons[i].Propagate();

  fmax=MAXDISTANCE;
  imax=-1;
  for(i=0;i<rang;i++)
  {
   f=neurons[i].CountDistance();
```

```
   if(f<fmax)
   {
    fmax=f;
    imax=i;
   }
  }

  if(imax==-1)
  {
   out_str(0,13,"minD=???",10);
   return;
  }

  ny=imax/dx;   // вычисление координат X & Y
  nx=imax%dx;

  char buf[40];
  sprintf(buf,"minD=%d(%d,%d)",imax,nx,ny);
  out_str(0,13,buf,10);

  for(i=0;i<rang;i++) neurons[i].delta = 0;


 if(0==Accreditation) //neurons[imax].delta = 1;

 {
  for(cx=max(nx-(int)MaxDistance,0);
      cx<min(nx+(int)MaxDistance+1,dx);cx++)
  {
   if(dy > 0)
   {
    for(cy=max(ny-(int)MaxDistance,0);
        cy<min(ny+(int)MaxDistance+1,dy);cy++)
    {
     // нейрон в зоне обучения
     neurons[cy*dx+cx].delta = 1;
    }
   }
   else
   {
    neurons[cx].delta = 1;
   }
  }
 }

 else //if(Accreditation)
 {
  for(i=0;i<rang;i++)
  {
   int y=i/dx;
   int x=i%dx;

   if(fabs(MaxDistance)>=1.0)
    neurons[i].delta=
    exp(-sqrt((nx-x)*(nx-x)+(ny-y)*(ny-y))
        /MaxDistance);
   else Accreditation=0;

  }
 }
}

NetUL::NetUL(unsigned nLayers)
{
 layers=NULL;
 if(nLayers==0)
 {
  status=ERROR; return;
 }
 layers=new LayerUL _FAR *[nLayers];
 if(layers==NULL) status=ERROR;
 else
 {
  rang=nLayers;
  for(unsigned i=0;i<rang;i++) layers[i]=NULL;
 }
}

NetUL::~NetUL()
{
 if(rang)
 {
  if(layers!=NULL)
  {
   for(unsigned i=0;i<rang;i++) layers[i]-
>~LayerUL();
   delete [] layers; layers=NULL;
  }
 }
}

int NetUL::SetLayer(unsigned n, LayerUL _FAR * pl)
{
unsigned i,p;

 if(n>=rang) return 1;
 p=pl->rang;
 if(p==0) return 2;
 if(n)     // если не первый слой
 {
  if(layers[n-1]!=NULL)
             // если предыдущий слой уже установлен,
  {        // проверяем соответствие числа нейронов
           // в нем и синапсов в добавляемом слое
   for(i=0;i<p;i++)
   if((*pl).neurons[i].rang!=layers[n-1]->rang)
    return 3;
  }
 }

 if(n<rang-1) // если не последний слой
 {
  if(layers[n+1])
  for(i=0;i<layers[n+1]->rang;i++)
  if(p!=layers[n+1]->neurons[i].rang) return 4;
 }

 layers[n]=pl;
 return 0;
}

int NetUL::FullConnect(void)
{
LayerUL *l;
unsigned i,j,k,n;
 for(i=1;i<rang;i++)  // кроме входного слоя
 {
  l=layers[i];               // по слоям
  if(l->rang==0) return 1;
  n=(*layers[i-1]).rang;
  if(n==0) return 2;
  for(j=0;j<l->rang;j++)     // по нейронам
  {
   for(k=0;k<n;k++)          // по синапсам
   {
    l->neurons[j].inputs[k]=
     &(layers[i-1]->neurons[k].axon);
   }
  }
 }
 return 0;
}

void NetUL::Propagate(void)
{
 for(unsigned i=1;i<rang;i++)
 {
  layers[i]->Propagate();
 }
}

void NetUL::SetNetInputs(float _FAR *mv)
{
 for(unsigned i=0;i<layers[0]->rang;i++)
  layers[0]->neurons[i].axon=mv[i];
}

void NetUL::NormalizeNetInputs(float _FAR *mv)
{
float s=0.;
 for(unsigned i=0;i<layers[0]->rang;i++)
  s+=(mv[i]+0.5)*(mv[i]+0.5);
 s=sqrt(s);
 if(s) for(i=0;i<layers[0]->rang;i++)
  mv[i]=(mv[i]+0.5)/s-0.5;
}

int Signum(float a, float b)
{
 if(a<0 && b<0) return -1;
 if(a>0 && b>0) return 1;
 return 0;
}

void LayerUL::TranslateAxons(void)
{
 if(0==Accreditation) return;
 for(int i=0;i<rang;i++)
 {
  neurons[i].axon=neurons[i].delta-0.5;
 }
}

void NetUL::Learn(void)
{
int j;
unsigned i,k;

 for(j=1;j<rang;j++)               // по слоям
 {
  if(Accreditation==0)
  {
   for(i=0;i<layers[j]->rang;i++)
   {                                // по нейронам
    if(layers[j]->neurons[i].delta == 0) continue;

    for(k=0;k<layers[j]->neuronrang;k++)
                                          // по
синапсам
    {
     layers[j]->neurons[i].synapses[k]+=LearnRate*
     (layers[j-1]->neurons[k].axon
     - layers[j]->neurons[i].synapses[k]);
    }
   }
  }
  else
  {
   for(i=0;i<layers[j]->rang;i++)
   {                                 // по нейронам
    if(Inhibition       // заторможенные пропускаем
    && layers[j]->neurons[i].inhibitory>0) continue;

    for(k=0;k<layers[j]->neuronrang;k++)
                                          // по
синапсам
    {
     layers[j]->neurons[i].synapses[k]+=LearnRate
```

```
      *layers[j]->neurons[i].delta
      *(layers[j-1]->neurons[k].axon
      - layers[j]->neurons[i].synapses[k]);
     }
    }
   }
  }
}

void NetUL::Randomize(float range)
{
 for(unsigned i=0;i<rang;i++)
  layers[i]->Randomize(range);
}

void NetUL::Equalize(void)
{
 for(unsigned i=1;i<rang;i++)
  layers[i]->Equalize();
}

void NetUL::Normalize(void)
{
 for(unsigned i=1;i<rang;i++)
  layers[i]->Normalize();
}

int NetUL::SaveToFile(unsigned char *file)
{
FILE *fp;
 fp=fopen(file,"wt");
 if(fp==NULL) return 1;
 fprintf(fp,"%u",rang);
 for(unsigned i=0;i<rang;i++)
 {
  fprintf(fp,"\n+%u",layers[i]->rang);
  fprintf(fp,"\n¦%u",layers[i]->neuronrang);
  for(unsigned j=0;j<layers[i]->rang;j++)
  {
   fprintf(fp,"\n¦+%f",layers[i]->neurons[j].state);
   fprintf(fp,"\n¦¦%f",layers[i]->neurons[j].axon);
   fprintf(fp,"\n¦¦%f",layers[i]->neurons[j].delta);
   for(unsigned k=0;k<layers[i]->neuronrang;k++)
   {
    fprintf(fp,"\n¦¦%f",
     layers[i]->neurons[j].synapses[k]);
   }
   fprintf(fp,"\n¦+");
  }
  fprintf(fp,"\n+");
 }
 fclose(fp);
 return 0;
}

int NetUL::LoadFromFile(unsigned char *file)
{
FILE *fp;
unsigned i,r,nr;
unsigned char bf[12];
 fp=fopen(file,"rt");
 if(fp==NULL) return 1;
 fscanf(fp,"%u\n",&r);

 if(r==0) goto allerr;
 layers=new LayerUL _FAR *[r];
 if(layers==NULL)
 { allerr: status=ERROR; fclose(fp); return 2; }
 else
 {
  rang=r;
  for(i=0;i<rang;i++) layers[i]=NULL;
 }

 for(i=0;i<rang;i++)
 {
  fgets(bf,10,fp);
  r=atoi(bf+1);
  fgets(bf,10,fp);
  nr=atoi(bf+1);
  layers[i] = new LayerUL(r,nr);
  for(unsigned j=0;j<layers[i]->rang;j++)
  {
   fscanf(fp,"¦+%f\n",&(layers[i]-
>neurons[j].state));
   fscanf(fp,"¦¦%f\n",&(layers[i]-
>neurons[j].axon));
   fscanf(fp,"¦¦%f\n",&(layers[i]-
>neurons[j].delta));
   for(unsigned k=0;k<layers[i]->neuronrang;k++)
   {
    fscanf(fp,"¦¦%f\n",
     &(layers[i]->neurons[j].synapses[k]));
   }
   fgets(bf,10,fp);
  }
  fgets(bf,10,fp);
 }
 fclose(fp);
 return 0;
}

NetUL::NetUL(unsigned n, unsigned n1, ...)
{
unsigned i, num, prenum;
va_list varlist;

 status=OK; rang=0; pf=NULL;
learncycle=0;layers=NULL;
 layers=new LayerUL _FAR *[n];
 if(layers==NULL) { allerr: status=ERROR; }
 else
 {
  rang=n;
  for(i=0;i<rang;i++) layers[i]=NULL;

  num=n1;
  layers[0] = new LayerUL(num,0);
  if(layers[0]->GetStatus()==ERROR) status=ERROR;
  va_start(varlist,n1);
  for(i=1;i<rang;i++)
  {
   prenum=num;
   num=va_arg(varlist,unsigned);
   layers[i] = new LayerUL(num,prenum);
   if(layers[i]->GetStatus()==ERROR) status=ERROR;
  }
  va_end(varlist);
 }
}

int NetUL::LoadNextPattern(float _FAR *IN)
{
unsigned char buf[256];
unsigned char *s, *ps;
int i;
 if(imgfile==1)
 {
  restart:
  for(i=0;i<layers[0]->dy;i++)
  {
   if(fgets(buf,256,pf)==NULL)
   {
    if(learncycle)
    {
     rewind(pf);
     learncycle--;
     goto restart;
    }
    else return 2;
   }

   for(int j=0;j<layers[0]->dx;j++)
   {
    if(buf[j]=='x') IN[i*layers[0]->dx+j]=0.5;
    else if(buf[j]=='.') IN[i*layers[0]->dx+j]=-0.5;
   }
  }

  fgets(buf,256,pf);
  return 0;
 }
 else if(imgfile==2 && emuf != NULL)
  return (*emuf)(layers[0]->rang,IN,NULL);
 else if(pf==NULL) return 1;

 // разбор строки доверять функции scanf нельзя
 start:
 if(fgets(buf,250,pf)==NULL)
 {
  if(learncycle)
  {
   rewind(pf);
   learncycle--;
   goto start;
  }
  else return 2;
 }
 s=buf;
 for(;*s==' ';s++);
 for(i=0;i<layers[0]->rang;i++)
 {
  ps=strchr(s,' ');
  if(ps) *ps=0;
  IN[i]=atof(s);
  s=ps+1; for(;*s==' ';s++);
 }

 fgets(buf,256,pf);
 return 0;
}

// функция внесения помех
float NetUL::Change(float In)
{
 return -In;
}

void NetUL::AddNoise(void)
{
unsigned i,k;
 for(i=0;i<dSigma;i++)
 {
  k=random(layers[0]->rang);
  layers[0]->neurons[k].axon=
  Change(layers[0]->neurons[k].axon);
 }
}

void NetUL::ConvexCombination(float *In, float step)
{
float sq=1./sqrt(layers[0]->rang)-0.5;
 if(step<0.) step=0.;
 if(step>1.) step=1.;
 for(int i=0;i<layers[0]->rang;i++)
  In[i]=In[i]*step+sq*(1-step);
}
```

```
void NetUL::NormalizeSynapses(void)
{
 for(unsigned i=0;i<rang;i++)
   layers[i]->NormalizeSynapses();
}
```

Листинг 3

```
// FILE neuman7.cpp for neuro_mm.prj
#include <string.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <bios.h>
#include "neuro_mm.h"
//#define INHIBITION 2

#pragma argsused
int GenFunction(int n, float _FAR *in, float _FAR
*ou)
{
static unsigned loop=0;
static int repeat=0;
int i;
 for(i=0;i<n;i++) in[i]=(float)rand()/RAND_MAX-0.5;
 repeat++;
 if(repeat==232)
 {
  repeat=0;
  loop++;
  srand(loop);
 }
 return 0;
}

main()
{
float Inp[30];
int count;
unsigned char buf[256];
float md=0.0;
int i;

 NetUL N(2,2,100);

 if(N.GetStatus()==ERROR)
 {
  printf("\nERROR: Net Can not Be Constructed!");
  return 1;
 }

 ClearScreen();
 N.GetLayer(0)->SetName("Input");
 N.GetLayer(0)->SetShowDim(1,1,2,1);
 N.GetLayer(1)->SetName("Out");
 N.GetLayer(1)->SetShowDim(17,1,10,10);

 srand(2); // задаем начальное условие для ГСЧ
 SetSigmoidTypeUL(HYPERTAN);
 SetDSigmaUL(2);
 N.FullConnect();
 N.Randomize(5);
 N.NormalizeSynapses();
 // N.Equalize();  // использовать с
ConvexCombination
 N.SetLearnCycle(64000U);

 SetLearnRateUL(1);

 // используем гауссиан для определения формы
 // области обучения и эффективности воздействия
 SetAccreditationUL(1);
// SetInhibitionUL(INHIBITION);

 N.EmulatePatternFile(GenFunction);

 i=13;
 for(count=0;;count++)
 {
  sprintf(buf," Cycle %u ",count);
  out_str(1,23,buf,10 | (1<<4));
  sprintf(buf,"MD=%.1f ",md);
  out_str(14,23,buf,10);
  out_str(1,24," ESC breaks ",11 | (1<<4));
  if(kbhit() || i==13) i=getch();
  if(i==27) break;
  if(i=='s' || i=='S')
  {
   out_str(40,24,"FileConf:",7);
   gotoxy(50,25);
   gets(buf);
   if(strlen(buf)) N.SaveToFile(buf);
   break;
  }

  if(N.LoadNextPattern(Inp)) break;

  // использовать вместе NormalizeSynapses
  // для сложных образов
  // N.NormalizeNetInputs(Inp);

  if(count<3000)
   md=SetMaxDistanceUL(7.0*(3000-count)/3000+1);
  else
   SetMaxDistanceUL(0);

  if(count<3000)
   SetLearnRateUL(0.1*(3000-count)/3000+0.05);
  else
   SetLearnRateUL(0.1);

  // N.ConvexCombination(Inp,(float)count/1000);
  N.SetNetInputs(Inp);
  // в случае ограниченного тренировочного набора
  // варьируем выборку данных
  // N.AddNoise();

  N.Propagate();

  // если нажат Shift, ничего не выводим
  // для ускорения процесса
  if(!(bioskey(2) & 0x03))
  {
   N.GetLayer(0)->DigiShow();
   N.GetLayer(1)->DigiShow(); // состояние
   N.GetLayer(1)->SetShowDim(50,1,10,10);
   N.GetLayer(1)->TranslateAxons();
   N.GetLayer(1)->Show();  // текущая область
обучения
   N.GetLayer(1)->SetShowDim(17,1,10,10);
  }

  // N.NormalizeSynapses();
  N.Learn();

 }
 N.ClosePatternFile();
 return 0;
}
```

Листинг 4

```
// FILE colour.h

// background colours
#define CBLUE (1<<4)
#define CGREEN (1<<5)
#define CRED (1<<6)

#define CCYAN (CGREEN|CBLUE)
#define CYELLOW (CGREEN|CRED)
#define CMAGENTA (CBLUE|CRED)
#define CBLACK 0
#define CGRAY (CGREEN|CBLUE|CRED)
```