С.Короткий

Нейронные сети: алгоритм обратного распространения

В статье рассмотрен алгоритм обучения нейронной сети с помощью процедуры обратного распространения, описана библиотека классов для C++.

Среди различных структур нейронных сетей (НС) одной из наиболее известных является многослойная структура, в которой каждый нейрон произвольного слоя связан со всеми аксонами нейронов предыдущего слоя или, в случае первого слоя, со всеми входами НС. Такие НС называются полносвязными. Когда в сети только один слой, алгоритм ее обучения с учителем довольно очевиден, так как правильные выходные состояния нейронов единственного слоя заведомо известны, и подстройка синаптических связей идет в направлении, минимизирующем ошибку на выходе сети. По этому принципу строится, например, алгоритм обучения однослойного перцептрона[1]. В многослойных же сетях оптимальные выходные значения нейронов всех слоев, кроме последнего, как правило, не известны, и двух или более слойный перцептрон уже невозможно обучить, руководствуясь только величинами ошибок на выходах НС. Один из вариантов решения этой проблемы – разработка наборов выходных сигналов, соответствующих входным, для каждого слоя НС, что, конечно, является очень трудоемкой операцией и не всегда осуществимо. Второй вариант – динамическая подстройка весовых коэффициентов синапсов, в ходе которой выбираются, как правило, наиболее слабые связи и изменяются на малую величину в ту или иную сторону, а сохраняются только те изменения, которые повлекли уменьшение ошибки на выходе всей сети. Очевидно, что данный метод "тыка", несмотря на свою кажущуюся простоту, требует громоздких рутинных вычислений. И, наконец, третий, более приемлемый вариант – распространение сигналов ошибки от выходов НС к ее входам, в направлении, обратном прямому распространению сигналов в обычном режиме работы. Этот алгоритм обучения НС получил название процедуры обратного распространения. Именно он будет рассмотрен в дальнейшем.

Согласно методу наименьших квадратов, минимизируемой целевой функцией ошибки НС является величина:

$$E(w) = \frac{1}{2}\sum_{j,p}(y_{j,p}^{(N)} - d_{j,p})^2 \qquad (1)$$

где $y_{j,p}^{(N)}$ – реальное выходное состояние нейрона j выходного слоя N нейронной сети при подаче на ее входы p-го образа; d_{jp} – идеальное (желаемое) выходное состояние этого нейрона.

Суммирование ведется по всем нейронам выходного слоя и по всем обрабатываемым сетью образам. Минимизация ведется методом градиентного спуска, что означает подстройку весовых коэффициентов следующим образом:

$$\Delta w_{ij}^{(n)} = -\eta \cdot \frac{\partial E}{\partial w_{ij}} \qquad (2)$$

Здесь w_{ij} – весовой коэффициент синаптической связи, соединяющей i-ый нейрон слоя n-1 с j-ым нейроном слоя n, η – коэффициент скорости обучения, $0<\eta<1$.

Как показано в [2],

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \cdot \frac{dy_j}{ds_j} \cdot \frac{\partial s_j}{\partial w_{ij}} \tag{3}$$

Здесь под y_j, как и раньше, подразумевается выход нейрона j, а под s_j – взвешенная сумма его входных сигналов, то есть аргумент активационной функции. Так как множитель dy_j/ds_j является производной этой функции по ее аргументу, из этого следует, что производная активационной функция должна быть определена на всей оси абсцисс. В связи с этим функция единичного скачка и прочие активационные функции с неоднородностями не подходят для рассматриваемых НС. В них применяются такие гладкие функции, как гиперболический тангенс или классический сигмоид с экспонентой. В случае гиперболического тангенса

$$\frac{dy}{ds} = 1 - s^2 \tag{4}$$

Третий множитель $\partial s_j/\partial w_{ij}$, очевидно, равен выходу нейрона предыдущего слоя $y_i^{(n-1)}$.

Что касается первого множителя в (3), он легко раскладывается следующим образом[2]:

$$\frac{\partial E}{\partial y_j} = \sum_k \frac{\partial E}{\partial y_k} \cdot \frac{dy_k}{ds_k} \cdot \frac{\partial s_k}{\partial y_j} = \sum_k \frac{\partial E}{\partial y_k} \cdot \frac{dy_k}{ds_k} \cdot w_{jk}^{(n+1)} \tag{5}$$

Здесь суммирование по k выполняется среди нейронов слоя n+1.

Введя новую переменную

$$\delta_j^{(n)} = \frac{\partial E}{\partial y_j} \cdot \frac{dy_j}{ds_j} \tag{6}$$

мы получим рекурсивную формулу для расчетов величин $\delta_j^{(n)}$ слоя n из величин $\delta_k^{(n+1)}$ более старшего слоя n+1.

$$\delta_j^{(n)} = \left[\sum_k \delta_k^{(n+1)} \cdot w_{jk}^{(n+1)} \right] \cdot \frac{dy_j}{ds_j} \tag{7}$$

Для выходного же слоя

$$\delta_l^{(N)} = (y_l^{(N)} - d_l) \cdot \frac{dy_l}{ds_l} \tag{8}$$

Теперь мы можем записать (2) в раскрытом виде:

$$\Delta w_{ij}^{(n)} = -\eta \cdot \delta_j^{(n)} \cdot y_i^{(n-1)} \tag{9}$$

Иногда для придания процессу коррекции весов некоторой инерционности, сглаживающей резкие скачки при перемещении по поверхности целевой функции, (9) дополняется значением изменения веса на предыдущей итерации

$$\Delta w_{ij}^{(n)}(t) = -\eta \cdot (\mu \cdot \Delta w_{ij}^{(n)}(t-1) + (1-\mu) \cdot \delta_j^{(n)} \cdot y_i^{(n-1)}) \tag{10}$$

где μ – коэффициент инерционности, t – номер текущей итерации.

Таким образом, полный алгоритм обучения НС с помощью процедуры обратного распространения строится так:

1. Подать на входы сети один из возможных образов и в режиме обычного функционирования НС, когда сигналы распространяются от входов к выходам, рассчитать значения последних. Напомним, что

$$s_j^{(n)} = \sum_{i=0}^{M} y_i^{(n-1)} \cdot w_{ij}^{(n)} \tag{11}$$

где M – число нейронов в слое n-1 с учетом нейрона с постоянным выходным состоянием +1, задающего смещение; $y_i^{(n-1)}=x_{ij}^{(n)}$ – i-ый вход нейрона j слоя n.

$y_j^{(n)} = f(s_j^{(n)})$, где $f()$ – сигмоид (12)

$y_q^{(0)}=I_q$, (13)

где I_q – q-ая компонента вектора входного образа.

2. Рассчитать $\boldsymbol{\delta^{(N)}}$ для выходного слоя по формуле (8).

Рассчитать по формуле (9) или (10) изменения весов $\boldsymbol{\Delta w^{(N)}}$ слоя N.

3. Рассчитать по формулам (7) и (9) (или (7) и (10)) соответственно $\boldsymbol{\delta^{(n)}}$ и $\boldsymbol{\Delta w^{(n)}}$ для всех остальных слоев, n=N-1,...1.

4. Скорректировать все веса в НС

$$w_{ij}^{(n)}(t) = w_{ij}^{(n)}(t-1) + \Delta w_{ij}^{(n)}(t) \tag{14}$$

5. Если ошибка сети существенна, перейти на шаг 1. В противном случае – конец.

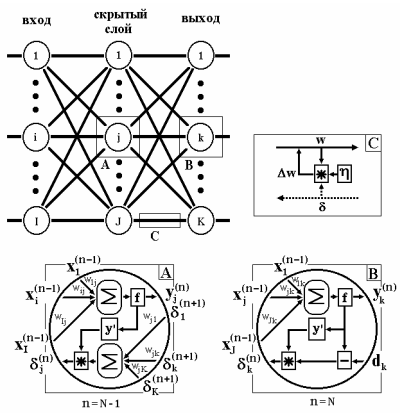

Рис.1 Диаграмма сигналов в сети при обучении по алгоритму обратного распространения

Сети на шаге 1 попеременно в случайном порядке предъявляются все тренировочные образы, чтобы сеть, образно говоря, не забывала одни по мере запоминания других. Алгоритм иллюстрируется рисунком 1.

Из выражения (9) следует, что когда выходное значение $y_i^{(n-1)}$ стремится к нулю, эффективность обучения заметно снижается. При двоичных входных векторах в среднем половина весовых коэффициентов не будет корректироваться[3], поэтому область возможных значений выходов нейронов [0,1] желательно сдвинуть в пределы [-0.5,+0.5], что достигается простыми модификациями логистических функций. Например, сигмоид с экспонентой преобразуется к виду

$$f(x) = -0.5 + \frac{1}{1+e^{-\alpha \cdot x}} \tag{15}$$

Теперь коснемся вопроса емкости НС, то есть числа образов, предъявляемых на ее входы, которые она способна научиться распознавать. Для сетей с числом слоев больше двух, он остается открытым. Как показано в [4], для НС с двумя слоями, то есть выходным и одним скрытым слоем, детерминистская емкость сети C_d оценивается так:

$$N_w/N_y<C_d<N_w/N_y\cdot\log(N_w/N_y) \quad (16)$$

где N_w – число подстраиваемых весов, N_y – число нейронов в выходном слое.

Следует отметить, что данное выражение получено с учетом некоторых ограничений. Во-первых, число входов N_x и нейронов в скрытом слое N_h должно удовлетворять неравенству $N_x+N_h>N_y$. Во-вторых, $N_w/N_y>1000$. Однако вышеприведенная оценка выполнялась для сетей с активационными функциями нейронов в виде порога, а емкость сетей с гладкими активационными функциями, например – (15), обычно больше[4]. Кроме того, фигурирующее в названии емкости прилагательное "детерминистский" означает, что полученная оценка емкости подходит абсолютно для всех возможных входных образов, которые могут быть представлены N_x входами. В действительности распределение входных образов, как правило, обладает некоторой регулярностью, что позволяет НС проводить обобщение и, таким образом, увеличивать реальную емкость. Так как распределение образов, в общем случае, заранее не известно, мы можем говорить о такой емкости только предположительно, но обычно она раза в два превышает емкость детерминистскую.

В продолжение разговора о емкости НС логично затронуть вопрос о требуемой мощности выходного слоя сети, выполняющего окончательную классификацию образов. Дело в том, что для разделения множества входных образов, например, по двум классам достаточно всего одного выхода. При этом каждый логический уровень – "1" и "0" – будет обозначать отдельный класс. На двух выходах можно закодировать уже 4 класса и так далее. Однако результаты работы сети, организованной таким образом, можно сказать – "под завязку", – не очень надежны. Для повышения достоверности классификации желательно ввести избыточность путем выделения каждому классу одного нейрона в выходном слое или, что еще лучше, нескольких, каждый из которых обучается определять принадлежность образа к классу со своей степенью достоверности, например: высокой, средней и низкой. Такие НС позволяют проводить классификацию входных образов, объединенных в нечеткие (размытые или пересекающиеся) множества. Это свойство приближает подобные НС к условиям реальной жизни.

Рассматриваемая НС имеет несколько "узких мест". Во-первых, в процессе обучения может возникнуть ситуация, когда большие положительные или отрицательные значения весовых коэффициентов сместят рабочую точку на сигмоидах многих нейронов в область насыщения. Малые величины производной от логистической функции приведут в соответствие с (7) и (8) к остановке обучения, что парализует НС. Во-вторых, применение метода градиентного спуска не гарантирует, что будет найден глобальный, а не локальный минимум целевой функции. Эта проблема связана еще с одной, а именно – с выбором величины скорости обучения. Доказательство сходимости обучения в процессе обратного распространения основано на производных, то есть приращения весов и, следовательно, скорость обучения должны быть бесконечно малыми, однако в этом случае обучение будет происходить неприемлемо медленно. С другой стороны, слишком большие коррекции весов могут привести к постоянной неустойчивости процесса обучения. Поэтому в качестве η обычно выбирается число меньше 1, но не очень маленькое, например, 0.1, и оно, вообще говоря, может постепенно

уменьшаться в процессе обучения. Кроме того, для исключения случайных попаданий в локальные минимумы иногда, после того как значения весовых коэффициентов застабилизируются, η кратковременно сильно увеличивают, чтобы начать градиентный спуск из новой точки. Если повторение этой процедуры несколько раз приведет алгоритм в одно и то же состояние НС, можно более или менее уверенно сказать, что найден глобальный максимум, а не какой-то другой.

Существует и иной метод исключения локальных минимумов, а заодно и паралича НС, заключающийся в применении стохастических НС, но о них лучше поговорить отдельно.

Теперь мы можем обратиться непосредственно к программированию НС. Следует отметить, что число зарубежных публикаций, рассматривающих программную реализацию сетей, ничтожно мало по сравнению с общим числом работ на тему нейронных сетей, и это при том, что многие авторы опробывают свои теоретические выкладки именно программным способом, а не с помощью нейрокомпьютеров и нейроплат, в первую очередь из-за их дороговизны. Возможно, это вызвано тем, что к программированию на западе относятся как к ремеслу, а не науке. Однако в результате такой дискриминации остаются неразобранными довольно важные вопросы.

Как видно из формул, описывающих алгоритм функционирования и обучения НС, весь этот процесс может быть записан и затем запрограммирован в терминах и с применением операций матричной алгебры, что сделано, например, в [5]. Судя по всему, такой подход обеспечит более быструю и компактную реализацию НС, нежели ее воплощение на базе концепций объектно-ориентированного (ОО) программирования. Однако в последнее время преобладает именно ОО подход, причем зачастую разрабатываются специальные ОО языки для программирования НС[6], хотя, с моей точки зрения, универсальные ОО языки, например C++ и Pascal, были созданы как раз для того, чтобы исключить необходимость разработки каких-либо других ОО языков, в какой бы области их не собирались применять.

И все же программирование НС с применением ОО подхода имеет свои плюсы. Во-первых, оно позволяет создать гибкую, легко перестраиваемую иерархию моделей НС. Во-вторых, такая реализация наиболее прозрачна для программиста, и позволяет конструировать НС даже непрограммистам. В-третьих, уровень абстрактности программирования, присущий ОО языкам, в будущем будет, по-видимому, расти, и реализация НС с ОО подходом позволит расширить их возможности. Исходя из вышеизложенных соображений, приведенная в листингах библиотека классов и программ, реализующая полносвязные НС с обучением по алгоритму обратного распространения, использует ОО подход. Вот основные моменты, требующие пояснений.

Прежде всего необходимо отметить, что библиотека была составлена и использовалась в целях распознавания изображений, однако применима и в других приложениях. В файле neuro.h в листинге 1 приведены описания двух базовых и пяти производных (рабочих) классов: Neuron, SomeNet и NeuronFF, NeuronBP, LayerFF, LayerBP, NetBP, а также описания нескольких общих функций вспомогательного назначения, содержащихся в файле subfun.cpp (см.листинг 4).

Методы пяти вышеупомянутых рабочих классов внесены в файлы neuro_ff.cpp и neuro_bp.cpp, представленные в листингах 2 и 3. Такое, на первый взгляд искусственное, разбиение объясняется тем, что классы с суффиксом _ff, описывающие прямопоточные нейронные сети (feedforward), входят в состав не только сетей с обратным распространением – _bp (backpropagation), но и других, например таких, как с обучением без учителя, которые будут рассмотрены в дальнейшем. Иерархия классов приведенной библиотеки приведена на рисунке 2.

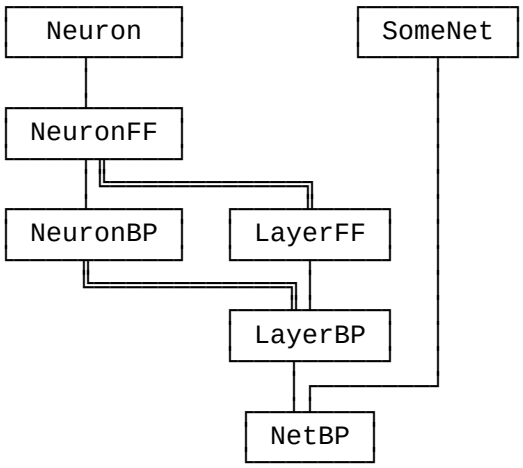

Рис.2 Иерархия классов библиотеки для сетей обратного распространения (одинарная линия – наследование, двойная – вхождение)

В ущерб принципам ОО программирования, шесть основных параметров, характеризующих работу сети, вынесены на глобальный уровень, что облегчает операции с ними. Параметр SigmoidType определяет вид активационной функции. В методе NeuronFF::Sigmoid перечислены некоторые его значения, макроопределения которых сделаны в заголовочном файле. Пункты HARDLIMIT и THRESHOLD даны для общности, но не могут быть использованы в алгоритме обратного распространения, так как соответствующие им активационные функции имеют производные с особыми точками. Это отражено в методе расчета производной NeuronFF::D_Sigmoid, из которого эти два случая исключены. Переменная SigmoidAlfa задает крутизну α сигмоида ORIGINAL из (15). MiuParm и NiuParm – соответственно значения параметров μ и η из формулы (10). Величина Limit используется в методах IsConverged для определения момента, когда сеть обучится или попадет в паралич. В этих случаях изменения весов становятся меньше малой величины Limit. Параметр dSigma эмулирует плотность шума, добавляемого к образам во время обучения НС. Это позволяет из конечного набора "чистых" входных образов генерировать практически неограниченное число "зашумленных" образов. Дело в том, что для нахождения оптимальных значений весовых коэффициентов число степеней свободы НС – N_w должно быть намного меньше числа накладываемых ограничений – $N_y \cdot N_p$, где N_p – число образов, предъявляемых НС во время обучения. Фактически, параметр dSigma равен числу входов, которые будут инвертированы в случае двоичного образа. Если dSigma = 0, помеха не вводится.

Методы Randomize позволяют перед началом обучения установить весовые коэффициенты в случайные значения в диапазоне [-range,+range]. Методы Propagate выполняют вычисления по формулам (11) и (12). Метод NetBP::CalculateError на основе передаваемого в качестве аргумента массива верных (желаемых) выходных значений НС вычисляет величины δ. Метод NetBP::Learn рассчитывает изменения весов по формуле (10), методы Update обновляют весовые коэффициенты. Метод NetBP::Cycle объединяет в себе все процедуры одного цикла обучения, включая установку входных сигналов NetBP::SetNetInputs. Различные методы

PrintXXX и LayerBP::Show позволяют контролировать течение процессов в НС, но их реализация не имеет принципиального значения, и простые процедуры из приведенной библиотеки могут быть при желании переписаны, например, для графического режима. Это оправдано и тем, что в алфавитно-цифровом режиме уместить на экране информацию о сравнительно большой НС уже не удается.

Сети могут конструироваться посредством NetBP(unsigned), после чего их нужно заполнять сконструированными ранее слоями с помощью метода NetBP::SetLayer, либо посредством NetBP(unsigned, unsigned,...). В последнем случае конструкторы слоев вызываются автоматически. Для установления синаптических связей между слоями вызывается метод NetBP::FullConnect.

После того как сеть обучится, ее текущее состояние можно записать в файл (метод NetBP::SaveToFile), а затем восстановить с помощью метода NetBP::LoadFromFile, который применим лишь к только что сконструированной по NetBP(void) сети.

Для ввода в сеть входных образов, а на стадии обучения – и для задания выходных, написаны три метода: SomeNet::OpenPatternFile, SomeNet::ClosePatternFile и NetBP::LoadNextPattern. Если у файлов образов произвольное расширение, то входные и выходные вектора записываются чередуясь: строка с входным вектором, строка с соответствующим ему выходным вектором и т.д. Каждый вектор есть последовательность действительных чисел в диапазоне [-0.5,+0.5], разделенных произвольным числом пробелов (см. листинг 7). Если файл имеет расширение IMG, входной вектор представляется в виде матрицы символов размером dy*dx (величины dx и dy должны быть заблаговременно установлены с помощью LayerFF::SetShowDim для нулевого слоя), причем символ 'x' соответствует уровню 0.5, а точка – уровню -0.5, то есть файлы IMG, по крайней мере – в приведенной версии библиотеки, бинарны (см. листинг 8). Когда сеть работает в нормальном режиме, а не обучается, строки с выходными векторами могут быть пустыми. Метод SomeNet::SetLearnCycle задает число проходов по файлу образов, что в сочетании с добавлением шума позволяет получить набор из нескольких десятков и даже сотен тысяч различных образов.

В листингах 5 и 6 приведены программы, конструирующие и обучающие НС, а также использующие ее в рабочем режиме распознавания изображений.

Особо следует отметить тот нюанс, что в рассматриваемой библиотеке классов НС отсутствует реализация подстраиваемого порога для каждого нейрона. Сеть, вообще говоря, может работать и без него, однако процесс обучения от этого замедляется[3]. Простой, хотя и не самый эффективный способ ввести для нейронов каждого слоя регулируемое смещение заключается в добавлении в класс NeuronFF метода Saturate, который принудительно устанавливал бы выход нейрона в состояние насыщения axon=0.5, с вызовом этого метода для какого-нибудь одного, например, последнего нейрона слоя в конце функции LayerFF::Propagate. Очевидно, что при этом на стадии конструирования в каждый слой НС, кроме выходного необходимо добавить один дополнительный нейрон. Он, в принципе, может не иметь ни

синапсов, ни массива изменений их весов и не вызывать метод Propagate внутри LayerFF::Propagate.

Рассмотренный выше и реализованный в программе алгоритм является, можно сказать, классическим вариантом процедуры обратного распространения, однако известны многие его модификации. Изменения касаются как методов расчетов [7][8], так и конфигурации сети [9][5]. В частности в [5] послойная организация сети заменена на магистральную, когда все нейроны имеют сквозной номер и каждый связан со всеми предыдущими.

Сеть, сконструированная в качестве примера в программе, приведенной на листинге 5, была обучена распознавать десять букв, схематично заданных матрицами 6*5 точек за несколько сотен циклов обучения, которые выполнились на компьютере 386DX40 за время меньше минуты. Обученная сеть успешно распознавала изображения, зашумленные более сильно, чем образы, на которых она обучалась.

Программа компилировалась с помощью Borland C++ 3.1 в моделях Large и Small.

Предложенная библиотека классов позволит создавать сети, способные решать широкий спектр задач, таких как построение экспертных систем, сжатие информации и многих других, исходные условия которых могут быть приведены к множеству парных, входных и выходных, наборов данных.

Литература

1. С.Короткий, Нейронные сети: основные положения.
2. Sankar K. Pal, Sushmita Mitra, Multilayer Perceptron, Fuzzy Sets, and Classification //IEEE Transactions on Neural Networks, Vol.3, N5,1992, pp.683-696.
3. Ф.Уоссермен, Нейрокомпьютерная техника, М., Мир, 1992.
4. Bernard Widrow, Michael A. Lehr, 30 Years of Adaptive NeuralNetworks: Perceptron, Madaline, and Backpropagation //Artificial Neural Networks: Concepts and Theory, IEEE Computer Society Press, 1992, pp.327-354.
5. Paul J. Werbos, Backpropagation Through Time: What It Does and How to Do It //Artificial Neural Networks: Concepts and Theory, IEEE Computer Society Press, 1992, pp.309-319.
6. Gael de La Croix Vaubois, Catherine Moulinoux, Benolt Derot, The N Programming Language //Neurocomputing, NATO ASI series, vol.F68, pp.89-92.
7. H.A.Malki, A.Moghaddamjoo, Using the Karhunen-Loe`ve Transformation in the Back-Propagation Training Algorithm //IEEE Transactions on Neural Networks, Vol.2, N1, 1991, pp.162-165.
8. Harris Drucker, Yann Le Cun, Improving Generalization Performance Using Backpropagation //IEEE Transactions on Neural Networks, Vol.3, N5, 1992, pp.991-997.
9. Alain Petrowski, Gerard Dreyfus, Claude Girault, Performance Analysis of a Pipelined Backpropagation Parallel Algorithm //IEEE Transactions on Neural Networks, Vol.4, N6, 1993, pp.970-981.

Листинг 1

```
// FILE neuro.h
#include <stdio.h>

#define OK    0         // состояния объектов
#define ERROR 1

#define ORIGINAL  0     // типы активационных
функций
```

```
#define HYPERTAN  1
#define HARDLIMIT 2
#define THRESHOLD 3

#define INNER    0      // тип распределения памяти
#define EXTERN   1

#define HORIZONTAL 1
#define VERTICAL 0

#ifndef max
#define max(a,b)    (((a) > (b)) ? (a) : (b))
#define min(a,b)    (((a) < (b)) ? (a) : (b))
#endif

// базовый класс нейронов для большинства сетей
class Neuron
{
protected:
 float state;         // состояние
 float axon;          // выход
 int status;          // признак ошибки
public:
 Neuron(void){ state=0.; axon=0.; status=OK; };
 virtual float Sigmoid(void)=0;
 int GetStatus(void){return status;};
};

class SomeNet
{
protected:
 FILE *pf;
 int imgfile; // 0 - числа; 1 - 2D; 2 - эмуляция
 unsigned rang;
 int status;
 unsigned learncycle;
 int (*emuf)(int n, float _FAR *in, float _FAR *ou);
public:
 SomeNet(void)
 {pf=NULL;imgfile=0;rang=0;status=OK;learncycle=0;};
 unsigned GetRang(void){return rang;};
 void SetLearnCycle(unsigned l){learncycle=l;};
 int  OpenPatternFile(unsigned char *file);
 int  ClosePatternFile(void);
 void EmulatePatternFile(int (*p)(int n,
      float _FAR *, float _FAR *))
      {emuf=p;imgfile=2;};
 int  GetStatus(void){return status;};
};

class LayerBP;
class NetBP;

// нейрон для полносвязной сети прямого
распространения
class NeuronFF: public Neuron
{
protected:
 unsigned rang;        // число весов
 float _FAR *synapses; // веса
 float _FAR * _FAR *inputs;
 // массив указателей на выходы нейронов предыд.
слоя
 void _allocateNeuron(unsigned);
 void _deallocate(void);
public:
 NeuronFF(unsigned num_inputs);
 NeuronFF(void){rang=0; synapses=NULL;
 inputs=NULL; status=OK;};
 ~NeuronFF();
 virtual void Propagate(void);
 void SetInputs(float *massive);
 void InitNeuron(unsigned numsynapses);
 virtual void RandomizeAxon(void);
 virtual void Randomize(float);
 virtual float Sigmoid(void);
 virtual float D_Sigmoid(void);
 virtual void PrintSynapses(int,int);
 virtual void PrintAxons(int, int);
};

class NeuronBP: public NeuronFF
{                              friend LayerBP;
                               friend NetBP;
 float error;
 float _FAR *deltas;      // изменения весов
 void _allocateNeuron(unsigned);
 void _deallocate(void);
public:
 NeuronBP(unsigned num_inputs);
 NeuronBP(void){deltas=NULL; error=0.;};
 ~NeuronBP();
 void InitNeuron(unsigned numsynapses);
 int  IsConverged(void);
};

class LayerFF
{
protected:
 unsigned rang;
 int status;
 int x,y,dx,dy;
 unsigned char *name;   // имя слоя
public:
 LayerFF(void) { rang=0; name=NULL; status=OK; };
 unsigned GetRang(void){return rang;};
 void SetShowDim(int _x, int _y, int _dx, int _dy)
 {x=_x; y=_y; dx=_dx; dy=_dy;};
 void SetName(unsigned char *s) {name=s;};
 unsigned char *GetName(void)
 {if(name) return name;
  else return (unsigned char *)&("NoName");};
 int GetStatus(void){return status;};
 int GetX(void){return x;};
 int GetY(void){return y;};
 int GetDX(void){return dx;};
 int GetDY(void){return dy;};
};

class LayerBP: public LayerFF
{                              friend NetBP;
protected:
 unsigned neuronrang; // число синапсов в нейронах
 int allocation;
 NeuronBP _FAR *neurons;
public:
 LayerBP(unsigned nRang, unsigned nSinapses);
 LayerBP(NeuronBP _FAR *Neu, unsigned nRang,
          unsigned nSinapses);
 LayerBP(void)
 {neurons=NULL; neuronrang=0; allocation=EXTERN;};
 ~LayerBP();
 void Propagate(void);
 void Randomize(float);
 void RandomizeAxons(void);
 void Normalize(void);
 void Update(void);
 int  IsConverged(void);
 virtual void Show(void);
 virtual void PrintSynapses(int,int);
 virtual void PrintAxons(int x, int y, int
direction);
};

class NetBP: public SomeNet
{
 LayerBP _FAR * _FAR *layers;
 // нулевой слой нейронов без синапсов реализует
входы
public:
 NetBP(void) { layers=NULL; };
 NetBP(unsigned nLayers);
 NetBP(unsigned n, unsigned n1, ...);
 ~NetBP();
 int SetLayer(unsigned n, LayerBP _FAR *pl);
 LayerBP *GetLayer(unsigned n)
  {if(n<rang) return layers[n]; else return NULL; }
 void Propagate(void);
 int FullConnect(void);
 void SetNetInputs(float _FAR *mvalue);
 void CalculateError(float _FAR * Target);
 void Learn(void);
 void Update(void);
 void Randomize(float);
 void Cycle(float _FAR *Inp, float _FAR *Out);
 int  SaveToFile(unsigned char *file);
 int  LoadFromFile(unsigned char *file);
 int  LoadNextPattern(float _FAR *IN, float _FAR
*OU);
 int  IsConverged(void);
 void AddNoise(void);
 virtual void PrintSynapses(int x=0,...){};
 virtual float Change(float In);
};

// Сервисные функции
void out_char(int x,int y,int c,int at);
void out_str(int x,int y,unsigned char *s,unsigned
col);
void ClearScreen(void);

// Глобальные параметры для обратного
распространения
int SetSigmoidType(int st);
float SetSigmoidAlfa(float Al);
float SetMiuParm(float Mi);
float SetNiuParm(float Ni);
float SetLimit(float Li);
unsigned SetDSigma(unsigned d);

// Псевдографика
#define GRAFCHAR_UPPERLEFTCORNER 218
#define GRAFCHAR_UPPERRIGHTCORNER 191
#define GRAFCHAR_HORIZONTALLINE 196
#define GRAFCHAR_VERTICALLINE 179
#define GRAFCHAR_BOTTOMLEFTCORNER 192
#define GRAFCHAR_BOTTOMRIGHTCORNER 217

#define GRAFCHAR_EMPTYBLACK 32
#define GRAFCHAR_DARKGRAY 176
#define GRAFCHAR_MIDDLEGRAY 177
#define GRAFCHAR_LIGHTGRAY 178
#define GRAFCHAR_SOLIDWHITE 219
```

Листинг 2

```
//FILE neuro_ff.cpp FOR neuro1.prj & neuro2.prj
#include <stdlib.h>
#include <math.h>
#include "neuro.h"

static int SigmoidType=ORIGINAL;
static float SigmoidAlfa=2.; // > 4 == HARDLIMIT

int SetSigmoidType(int st)
{
int i;
```

```
 i=SigmoidType;
 SigmoidType=st;
 return i;
}

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

void NeuronFF::Randomize(float range)
{
 for(unsigned i=0;i<rang;i++)
  synapses[i]=range*((float)rand()/RAND_MAX-0.5);
}

void NeuronFF::RandomizeAxon(void)
{
 axon=(float)rand()/RAND_MAX-0.5;
}

float NeuronFF::D_Sigmoid(void)
{
 switch(SigmoidType)
 {
  case HYPERTAN: return (1.-axon*axon);
  case ORIGINAL: return SigmoidAlfa*(axon+0.5)*
                        (1.5-axon);
  default:       return 1.;
 }
}

float NeuronFF::Sigmoid(void)
{
 switch(SigmoidType)
 {
  case HYPERTAN: return 0.5*tanh(state);
  case ORIGINAL: return -0.5+1./
                 (1+exp(-SigmoidAlfa*state));
  case HARDLIMIT:if(state>0) return 0.5;
                     else if(state<0) return -0.5;
                     else return state;
  case THRESHOLD:if(state>0.5) return 0.5;
                     else if(state<-0.5) return -0.5;
                 else return state;
  default:       return 0.;
 }
}

void NeuronFF::_allocateNeuron(unsigned num_inputs)
{
 synapses=NULL;inputs=NULL;status=OK;rang=0;
 if(num_inputs==0) return;
 synapses= new float[num_inputs];
 if(synapses==NULL) status=ERROR;
 else
 {
  inputs=new float _FAR * [num_inputs];
  if(inputs==NULL) status=ERROR;
  else
  {
   rang=num_inputs;
   for(unsigned i=0;i<rang;i++)
   { synapses[i]=0.; inputs[i]=NULL; }
  }
 }
}

NeuronFF::NeuronFF(unsigned num_inputs)
{
 _allocateNeuron(num_inputs);
}

void NeuronFF::_deallocate(void)
{
 if(rang && (status==OK))
 {delete [] synapses;delete [] inputs;
  synapses=NULL; inputs=NULL;}
}

NeuronFF::~NeuronFF()
{
 _deallocate();
}

void NeuronFF::Propagate(void)
{
 state=0.;
 for(unsigned i=0;i<rang;i++)
  state+=(*inputs[i]*2)*(synapses[i]*2);
 state/=2;
 axon=Sigmoid();
}

void NeuronFF::SetInputs(float *vm)
{
 for(unsigned i=0;i<rang;i++) inputs[i]=&vm[i];
}

void NeuronFF::InitNeuron(unsigned num_inputs)
{
 if(rang && (status==OK))
 {delete [] synapses;delete [] inputs;}
 _allocateNeuron(num_inputs);
}

void NeuronFF::PrintSynapses(int x=0, int y=0)
{
unsigned char buf[20];
 for(unsigned i=0;i<rang;i++)
 {
  sprintf(buf,"%+7.2f",synapses[i]);
  out_str(x+8*i,y,buf,11);
 }
}

void NeuronFF::PrintAxons(int x=0, int y=0)
{
unsigned char buf[20];
 sprintf(buf,"%+7.2f",axon);
 out_str(x,y,buf,11);
}
```

Листинг 3

```
// FILE neuro_bp.cpp FOR neuro1.prj & neuro2.prj
#include <stdlib.h>
#include <alloc.h>
#include <math.h>
#include <string.h>
#include <stdarg.h>
#include <values.h>

#include "neuro.h"

static float MiuParm=0.0;
static float NiuParm=0.1;
static float Limit=0.000001;
static unsigned dSigma=0;

float SetMiuParm(float Mi)
{
float a;
 a=MiuParm;
 MiuParm=Mi;
 return a;
}

float SetNiuParm(float Ni)
{
float a;
 a=NiuParm;
 NiuParm=Ni;
 return a;
}

float SetLimit(float Li)
{
float a;
 a=Limit;
 Limit=Li;
 return a;
}

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

void NeuronBP::_allocateNeuron(unsigned num_inputs)
{
 deltas=NULL;
 if(num_inputs==0) return;
 deltas=new float[num_inputs];
 if(deltas==NULL) status=ERROR;
 else for(unsigned i=0;i<rang;i++) deltas[i]=0.;
}

NeuronBP::NeuronBP(unsigned num_inputs)
         :NeuronFF(num_inputs)
{
 _allocateNeuron(num_inputs);
}

void NeuronBP::_deallocate(void)
{
 if(deltas && (status==OK))
 {delete [] deltas; deltas=NULL;}
}

NeuronBP::~NeuronBP()
{
 _deallocate();
}

void NeuronBP::InitNeuron(unsigned num_inputs)
{
 NeuronFF::InitNeuron(num_inputs);
 if(deltas && (status==OK)) delete [] deltas;
 _allocateNeuron(num_inputs);
}

int NeuronBP::IsConverged(void)
{
 for(unsigned i=0;i<rang;i++)
  if(fabs(deltas[i])>Limit) return 0;
 return 1;
}

//

LayerBP::LayerBP(unsigned nRang, unsigned nSynapses)
```

```
{
 allocation=EXTERN; status=ERROR; neuronrang=0;
 if(nRang==0) return;
 neurons=new NeuronBP[nRang];
 if(neurons==NULL) return;
 for(unsigned i=0;i<nRang;i++)
  neurons[i].InitNeuron(nSynapses);
 rang=nRang;
 neuronrang=nSynapses;
 allocation=INNER;
 name=NULL; status=OK;
}

LayerBP::LayerBP(NeuronBP _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;
 }
}

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

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

void LayerBP::Update(void)
{
 for(unsigned i=0;i<rang;i++)
 {
  for(unsigned j=0;j<neuronrang;j++)
  neurons[i].synapses[j]-=neurons[i].deltas[j];
 }
}

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

void LayerBP::RandomizeAxons(void)
{
 for(unsigned i=0;i<rang;i++)
  neurons[i].RandomizeAxon();
}

void LayerBP::Normalize(void)
{
float sum;
unsigned i;
 for(i=0;i<rang;i++)
  sum+=neurons[i].axon*neurons[i].axon;
 sum=sqrt(sum);
 for(i=0;i<rang;i++) neurons[i].axon/=sum;
}

void LayerBP::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++) out_char(x+1+i, y+1+j,
  sym[(int) ((neurons[j*dx+i/2].axon+0.4999)*5)],
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);
}

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

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

int LayerBP::IsConverged(void)
{
 for(unsigned i=0;i<rang;i++)
  if(neurons[i].IsConverged()==0) return 0;
 return 1;
}

//

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

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

int NetBP::SetLayer(unsigned n, LayerBP _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 подключен,
про-
  {        // веряем, равно ли число синапсов
каждого
           // его нейрона числу нейронов предыд.
слоя
   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;
}

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

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

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

void NetBP::CalculateError(float _FAR * Target)
{
NeuronBP *n;
float sum;
```

```
unsigned i;
int j;
 for(i=0;i<layers[rang-1]->rang;i++)
 {
  n=&(layers[rang-1]->neurons[i]);
  n->error=(n->axon-Target[i])*n->D_Sigmoid();
 }
 for(j=rang-2;j>0;j--)         // по скрытым слоям
 {
  for(i=0;i<layers[j]->rang;i++)    // по нейронам
  {
   sum=0.;
   for(unsigned k=0;k<layers[j+1]->rang;k++)
   sum+=layers[j+1]->neurons[k].error
       *layers[j+1]->neurons[k].synapses[i];
   layers[j]->neurons[i].error=
   sum*layers[j]->neurons[i].D_Sigmoid();
  }
 }
}

void NetBP::Learn(void)
{
 for(int j=rang-1;j>0;j--)
 {
  for(unsigned i=0;i<layers[j]->rang;i++)
  {                                  // по нейронам
   for(unsigned k=0;k<layers[j]->neuronrang;k++)
                                     // по синапсам
   layers[j]->neurons[i].deltas[k]=NiuParm*
   (MiuParm*layers[j]->neurons[i].deltas[k]+
    (1.-MiuParm)*layers[j]->neurons[i].error
    *layers[j-1]->neurons[k].axon);
  }
 }
}

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

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

void NetBP::Cycle(float _FAR *Inp, float _FAR *Out)
{
 SetNetInputs(Inp);
 if(dSigma) AddNoise();
 Propagate();
 CalculateError(Out);
 Learn();
 Update();
}

int NetBP::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].error);
   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 NetBP::LoadFromFile(unsigned char *file)
{
FILE *fp;
unsigned i,r,nr;
unsigned char bf[12];

 if(layers) return 1;  // возможно использование
только
  // экземпляров класса, сконструированных по
умолчанию
  // с помощью NetBP(void).

 fp=fopen(file,"rt");
 if(fp==NULL) return 1;
 fscanf(fp,"%u\n",&r);

 if(r==0) goto allerr;
 layers=new LayerBP _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 LayerBP(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].error));
   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;
}

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

 status=OK; rang=0; pf=NULL; learncycle=0;
layers=NULL;
 layers=new LayerBP _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 LayerBP(num,0);
  va_start(varlist,n1);
  for(i=1;i<rang;i++)
  {
   prenum=num;
   num=va_arg(varlist,unsigned);
   layers[i] = new LayerBP(num,prenum);
  }
  va_end(varlist);
 }
}

int NetBP::LoadNextPattern(float _FAR *IN,
                           float _FAR *OU)
{
unsigned char buf[256];
unsigned char *s, *ps;
int i;
 if(pf==NULL) return 1;
 if(imgfile)
 {
  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;
   }
  }

  if(fgets(buf,256,pf)==NULL) return 3;
  for(i=0;i<layers[rang-1]->rang;i++)
  {
   if(buf[i]!='.') OU[i]=0.5;
   else            OU[i]=-0.5;
  }
  return 0;
 }

 // "scanf often leads to unexpected results
 // if you diverge from an expected pattern." (!)
 //                  Borland C On-line Help
 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++);
 }
 if(fgets(buf,250,pf)==NULL) return 4;
 s=buf;
 for(;*s==' ';s++);
 for(i=0;i<layers[rang-1]->rang;i++)
 {
  ps=strchr(s,' ');
  if(ps) *ps=0;
  OU[i]=atof(s);
  s=ps+1; for(;*s==' ';s++);
 }
 return 0;
}

int NetBP::IsConverged(void)
{
 for(unsigned i=1;i<rang;i++)
  if(layers[i]->IsConverged()==0) return 0;
 return 1;
}

float NetBP::Change(float In)
{
// для бинарного случая
 if(In==0.5) return -0.5;
 else        return  0.5;
}

void NetBP::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);
 }
}
```

Листинг 4

```
// FILE subfun.cpp FOR neuro1.prj & neuro2.prj
#include <string.h>
#include <math.h>
#include <values.h>
#include "neuro.h"

#define vad(x,y) ((y)*160+(x)*2)

void out_char(int x,int y,int c,int at)
{
unsigned far *p;
 p=(unsigned far *)(0xB8000000L+
   (unsigned long)vad(x,y));
 *p=(c & 255) | (at<<8);
}

void out_str(int x,int y,unsigned char *s,unsigned
col)
{
 for(int i=0;i<strlen(s);i++)
out_char(x+i,y,s[i],col);
}

void ClearScreen(void)
{
 for(int i=0;i<80;i++) for(int j=0;j<25;j++)
  out_char(i,j,' ',7);
}

int matherr(struct exception *pe)
{
 if(strcmp(pe->name,"exp")==0)
 {
  if(pe->type==OVERFLOW) pe->retval=MAXDOUBLE;
  if(pe->type==UNDERFLOW) pe->retval=MINDOUBLE;
  return 10;
 }
 else
 {
  if(pe->type==UNDERFLOW || pe->type==TLOSS) return
1;
  else return 0;
 }
}

int SomeNet::OpenPatternFile(unsigned char *file)
{
 pf=fopen(file,"rt");
 if(strstr(file,".img")) imgfile=1;
 else imgfile=0;
 return !((int)pf);
}

int SomeNet::ClosePatternFile(void)
{
int i;
 if(pf)
 {
  i=fclose(pf);
  pf=NULL;
  return i;
 }
 return 0;
}
```

Листинг 5

```
// FILE neuman1.cpp FOR neuro1.prj
#include <string.h>
#include <conio.h>
#include "neuro.h"

#define N0 30
#define N1 10
#define N2 10

void main()
{
float Inp[N0], Out[N2];
unsigned count;
unsigned char buf[256];
int i;
 NetBP N(3,N0,N1,N2);
 /* первый способ конструирования сети */

 /*** второй способ конструирования сети
 NeuronBP _FAR *H0, _FAR *H1, _FAR *H2;
 H0= new NeuronBP [N0];
 H1= new NeuronBP [N1];
 H2= new NeuronBP [N2];

 for(i=0;i<N1;i++) H1[i].InitNeuron(N0);
 for(i=0;i<N2;i++) H2[i].InitNeuron(N1);

 LayerBP L0(H0,N0,0);
 LayerBP L1(H1,N1,N0);
 LayerBP L2(H2,N2,N1);

 NetBP N(3);
 i=N.SetLayer(0,&L0);
 i=N.SetLayer(1,&L1);
 i=N.SetLayer(2,&L2); // здесь можно проверить i
 ***/

 /* третий способ создания сети см. в листинге 6 */

 ClearScreen();
 N.FullConnect();
 N.GetLayer(0)->SetName("Input");
 N.GetLayer(0)->SetShowDim(1,1,5,6);
 N.GetLayer(1)->SetName("Hidden");
 N.GetLayer(1)->SetShowDim(15,1,2,5);
 N.GetLayer(2)->SetName("Out");
 N.GetLayer(2)->SetShowDim(23,1,10,1);

// srand(1);
// меняем особенность случайной структуры сети
 SetSigmoidType(HYPERTAN);
 SetNiuParm(0.1);
 SetLimit(0.001);
 SetDSigma(1);
 N.Randomize(1);
 N.SetLearnCycle(64000U);

 N.OpenPatternFile("char1.img");
 for(count=0;;count++)
 {
  sprintf(buf,"Cycle %u",count);
  out_str(1,23,buf,10 | (1<<4));
  out_str(1,24,"ESC breaks     ",11 | (1<<4));
  if(kbhit() || i==13) i=getch();
  if(i==27) break;
  if(i=='s' || i=='S') goto save;
  if(N.LoadNextPattern(Inp,Out)) break;
  N.Cycle(Inp,Out);
//  N.Propagate(); // "сквозной канал"
  N.GetLayer(0)->Show();
  N.GetLayer(1)->Show();
  N.GetLayer(2)->Show();
  N.GetLayer(2)->PrintAxons(47,0);
  if(count && N.IsConverged())
  {
   save:
   out_str(40,24,"FileConf:",15 | (1<<4));
   gotoxy(50,25);
   gets(buf);
   if(strlen(buf)) N.SaveToFile(buf);
   break;
  }
 }
 N.ClosePatternFile();
}
```

Листинг 6

```
// FILE neuman2.cpp FOR neuro2.prj
#include <string.h>
#include <conio.h>
#include "neuro.h"

#define N0 30
#define N1 10
#define N2 10

main(int argc, char *argv[])
{
```

```
NetBP N;
static float Inp[N0], Out[N2];

 if(argc!=2) return 1;
 ClearScreen();
 if(N.LoadFromFile(argv[1])) return 1;
 if(N.FullConnect()) return 1;
 N.GetLayer(0)->SetName("Input");
 N.GetLayer(0)->SetShowDim(1,1,5,6);
 N.GetLayer(1)->SetName("Hidden");
 N.GetLayer(1)->SetShowDim(15,1,2,5);
 N.GetLayer(2)->SetName("Out");
 N.GetLayer(2)->SetShowDim(23,1,10,1);
 SetSigmoidType(HYPERTAN);

 if(N.OpenPatternFile("charnois.img")) return 1;
 for(;;)
 {
  if(N.LoadNextPattern(Inp,Out)) break;
  // если все образы кончились, выходим
  N.SetNetInputs(Inp);
  N.Propagate();
  N.GetLayer(0)->Show();
  N.GetLayer(1)->Show();
  N.GetLayer(2)->Show();
  N.GetLayer(2)->PrintAxons(47,0);
  getch();
 }
 N.ClosePatternFile();
 return 0;
}
```

Листинг 7

Файл char1.pat (перенос длинных строк по '\' сделан при верстке)

```
-0.5 -0.5  0.5 -0.5 -0.5 -0.5  0.5 -0.5  0.5 -0.5 -
0.5 \
 0.5 -0.5  0.5 -0.5  0.5 -0.5 -0.5 -0.5  0.5  0.5
0.5 \
 0.5  0.5  0.5  0.5 -0.5 -0.5 -0.5  0.5
 0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5
 0.5 -0.5 -0.5 -0.5  0.5  0.5  0.5 -0.5  0.5  0.5
0.5 \
 0.5 -0.5  0.5  0.5  0.5 -0.5  0.5 -0.5  0.5  0.5 -
0.5 \
 0.5 -0.5  0.5  0.5 -0.5 -0.5 -0.5  0.5
-0.5  0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5
 0.5 -0.5 -0.5 -0.5  0.5  0.5 -0.5 -0.5 -0.5  0.5
0.5 \
 0.5  0.5  0.5  0.5  0.5 -0.5 -0.5 -0.5  0.5  0.5 -
0.5 \
-0.5 -0.5  0.5  0.5 -0.5 -0.5 -0.5  0.5
-0.5 -0.5  0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5
 0.5 -0.5 -0.5 -0.5  0.5  0.5 -0.5 -0.5  0.5  0.5
0.5 \
-0.5  0.5 -0.5  0.5  0.5 -0.5  0.5 -0.5  0.5  0.5
0.5 \
-0.5 -0.5  0.5  0.5 -0.5 -0.5 -0.5  0.5
-0.5 -0.5 -0.5  0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5
 0.5 -0.5 -0.5  0.5  0.5  0.5 -0.5  0.5 -0.5 -0.5
0.5 \
 0.5 -0.5 -0.5 -0.5  0.5 -0.5  0.5 -0.5 -0.5  0.5 -
0.5 \
-0.5  0.5 -0.5  0.5 -0.5 -0.5 -0.5  0.5
-0.5 -0.5 -0.5 -0.5  0.5 -0.5 -0.5 -0.5 -0.5 -0.5
-0.5 -0.5 -0.5  0.5 -0.5 -0.5 -0.5  0.5 -0.5  0.5 -
0.5 \
 0.5 -0.5 -0.5  0.5  0.5 -0.5 -0.5 -0.5  0.5  0.5 -
0.5 \
-0.5 -0.5  0.5  0.5 -0.5 -0.5 -0.5  0.5
-0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5
-0.5  0.5  0.5  0.5 -0.5  0.5 -0.5 -0.5 -0.5  0.5
0.5 \
-0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -
0.5 \
-0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5
-0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5
 0.5  0.5  0.5  0.5  0.5 -0.5 -0.5  0.5 -0.5 -0.5 -
0.5 \
-0.5  0.5 -0.5 -0.5 -0.5 -0.5  0.5 -0.5 -0.5 -0.5 -
0.5 \
 0.5 -0.5 -0.5 -0.5 -0.5  0.5 -0.5 -0.5
-0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5  0.5 -0.5 -0.5
 0.5  0.5  0.5  0.5  0.5  0.5 -0.5 -0.5 -0.5  0.5
0.5 \
-0.5 -0.5 -0.5  0.5  0.5 -0.5 -0.5 -0.5  0.5  0.5 -
0.5 \
-0.5 -0.5  0.5  0.5 -0.5 -0.5 -0.5  0.5
-0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5  0.5 -0.5
 0.5  0.5  0.5  0.5 -0.5  0.5 -0.5 -0.5 -0.5  0.5
0.5 \
 0.5  0.5  0.5 -0.5  0.5 -0.5 -0.5 -0.5  0.5  0.5 -
0.5 \
-0.5 -0.5  0.5  0.5  0.5  0.5  0.5 -0.5
-0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5  0.5
```

Листинг 8

Файл char1.img

```
..x..
.x.x.
.x.x.
x...x
xxxxx
x...x
A.........
x...x
xx.xx
xx.xx
x.x.x
x.x.x
x...x
.M.......
x...x
x...x
xxxxx
x...x
x...x
x...x
..H.......
x...x
x..xx
x.x.x
x.x.x
xx..x
x...x
...N......
x..xx
x.x..
xx...
x.x..
x..x.
x...x
....K.....
...x.
..x.x
.x..x
x...x
x...x
x...x
.....L....
.xxx.
x...x
x....
x....
x...x
.xxx
......C...
xxxxx
..x..
..x..
..x..
..x..
..x..
.......T..
xxxxx
x...x
x...x
x...x
x...x
x...x
........P.
xxxx.
x...x
xxxx
x..xx
x...x
xxxx.
.........B
```