Министерство образования Республики Беларусь

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (БГУ)

УДК 681.142.2 (075.8)
№ госрегистрации 20042138

УТВЕРЖДАЮ
Первый проректор,
д-р хим. наук, профессор

_____ С.К. Рахманов

«____» _____ 200__ г.

ОТЧЕТ О НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

РАЗРАБОТКА ПАРАЛЛЕЛЬНЫХ АЛГОРИТМОВ ОБУЧЕНИЯ НЕЙРОННЫХ СЕТЕЙ

(заключительный)

№ темы 448/11

Декан ф-та радиофизики
и электроники,
зав. кафедрой информатики
д-р. техн. наук, профессор — С. Г. Мулярчик

Научный руководитель, исполнитель
мл. науч. сотр., аспирант — А. Е. Верхотуров

Куратор проекта
канд. техн. наук,
доцент кафедры информатики — Г. И. Шпаковский

Нормоконтролер — Т. Н. Долгая

Минск, 2004

РЕФЕРАТ

Отчёт 58 страниц, 18 рисунков, 1 таблица, 15 источников, 1 приложение.
ПАРАЛЛЕЛЬНОЕ ПРОГРАМИРОВАНИЕ, АНАЛИЗ ЭФФЕКТИВНОСТИ, МАСШТАБИРОВАНИЕ, МОДЕЛИРОВАНИЕ, НЕЙРОННЫЕ СЕТИ, ГЕНЕТИЧЕСКИЕ АЛГОРИТМЫ

Объектом исследования являются методы обучения нейронных сетей на параллельных вычислительных системах. Исследуются подходы к реализации параллельного метода обучения нейронных сетей.

Цель работы – исследование эффективности подходов параллельной реализации методов обучения нейронных сетей на кластерах ЭВМ, реализация этих подходов в стандарте MPI.

В процессе работы проводилось моделирование процессов, происходящих при работе нейроэмулятора, анализировались параметры алгоритмов обучения нейронной сети и среды передачи данных.

В результате исследования построенные модели, с помощью которых произведена оценка эффективности параллельных методов обучения нейронных сетей. Реализован нейроэмулятор на языке программирования C++, параллельный генетический алгоритм в стандарте MPI. Показано, что параллельный генетический алгоритм может быть эффективно использован на начальной стадии обучения нейронных сетей, однако, его дальнейшее применение без модификации понижает эффективность процесса обучения.

Проводятся дальнейшие исследования по созданию эффективного метода обучения нейронных сетей на параллельных ЭВМ.

СОДЕРЖАНИЕ

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

MIMD – Multiple Instructions Multiple Data (множественный поток команд, множественный поток данных)

MPI – Message Passing Interface (стандарт передачи сообщений)

SPMD – Single Program Multiple Data (одна программа с множественным потоком данных)

ВС – Вычислительная сеть

ГА – Генетические алгоритмы

НС – Нейронная сеть

ОО – Объектно-ориентированное (программирование)

ОС – Операционная система

СЛАУ – Система линейных алгебраичесих уравнений

ВВЕДЕНИЕ

НС успешно применяются для решения ряда задач: классификация образов, кластеризация (категоризация), аппроксимация функций, предсказание или прогноз, оптимизация и другие. НС, будучи построенными из очень большого числа совсем простых элементов – нейронов, каждый из которых вычисляет взвешенную сумму входных сигналов и передает сигнал дальше, способны решать чрезвычайно сложные задачи. Однако обучение НС является вычислительно сложной задачей.

В работе исследуется возможность применения параллельных вычислений для обучения НС. Цель данной работы – исследовать эффективность подходов параллельной реализации медотов обучения НС на кластерах. Цель достигается решение следующих задач:

1. Изучение принципов функционирования и сущесвующих методов обучения НС
2. Реализация библиотеки классов на языке программирования C++, реализующих искуственные НС и методы их обучения
3. Изучение и формулировка недостатков существующих методов обучения НС и подходов увеличению их эффективности
4. Декомпозиция алгоритмов работы и обучения НС. Поиск подходов к применения параллельных вычислений
5. Моделирования работы параллельных алгоритмов обучения НС пи применении этих подходов
6. Реализация успешных подходов в стандарте MPI
7. Анализ параметров методов

По теме работы опубликована статья [1].

Важной характеристикой для исследования эффективности параллельных алгоритмов является ускорение вычислений. По определению ускорение есть отношение времени решения задачи на однопроцессорной системе ко времени решения на многопроцессорной системе.

$$R = T_1/T_n \tag{1}$$

где T_1 – время решения задачи на однопроцессорной системе, а T_n – на n-процессорной.

Максимальное ускорение от использования параллельных вычислений регламентирует закон Амдала (6). Согласно закону Амдала предельное ускорение определяется свойствами задачи. Однако в формуле (6) не учитываются потери на межпроцессорный обмен. Сетевой закон Амдала [2] учитывает влияние операций передачи данных на производительность через коэффициент сетевой деградации вычислений (9).

Функционирование НС с одной стороны обладает высокой степенью параллелизма, с другой стороны, при параллельной реализации на кластерах ЭВМ, требует большого числа пересылаемых данных. Создание метода обучения НС, в котором минимизировано количество передаваемых данных, позволило бы эффективно использовать имеющиеся вычислительные средства для обучения НС.

1 Параллельное программирование систем с передачей сообщений

Классификация параллельных ЭВМ была предложена Флинном [2]. Основными понятиями классификации являются “поток команд” и “поток данных”. Согласно классификации, ЭВМ делятся на 4 класса:

1. SISD (Single Instructions - Single Data) – последовательные ЭВМ.
2. SIMD (Single Instructions - Multiple Data) – векторные ЭВМ (одновременно может выполняться одна команда над всеми или несколькими элементами вектора).
3. MISD (Multiple Instructions - Single Data) – не нашла применения, считается, что несколько команд одновременно работают с одним элементом данных.
4. MIMD (Multiple Instructions - Multiple Data) – многопроцессорные системы. В таких ЭВМ независимо выполняются несколько программных ветвей, которые обмениваются данными.

Современные суперкомпьютеры и вычислительные сети принадлежат классу MIMD.

1.1 Технология программирования SPMD. Стандарт MPI

Наиболее распространенным интерфейсом параллельного программирования в модели передачи сообщений является MPI (Message Passing Interface). Рекомендуемой бесплатной реализацией MPI является пакет MPICH, разработанный в Аргоннской национальной лаборатории США.

Основой вычислительного кластера является компьютерная сеть. Компьютерная сеть – сложный комплекс взаимосвязанных и согласованно функционирующих программных и аппаратных компонентов.

Элементы сети:

- компьютеры;
- коммуникационное оборудование;
- операционные системы;
- сетевые приложения.

Кластер – параллельный компьютер, все процессоры которого действуют как единое целое для решения одной задачи. Совокупность таких компьютеров составляет метакомпьютер. Примером метакомпьютеров являются глобальные сети, в том числе и Интернет. Однако чтобы преобразовать Интернет в метакомпьютер, необходимо разработать большой объем системного программного обеспечения. В комплексе должны рассматриваться такие вопросы, как средства и модели программирования, распределение и диспетчирование заданий, технология организации доступа к метакомпьютеру, интерфейс с пользователями, безопасность, надежность, политика администрирования, средства доступа и технологии распределенного хранения данных, мониторинг состояния различных подсистем метакомпьютера и др.

Для того чтобы параллельная программа была переносима, необходимо наличие стандарта программирования. Для этих целей и был разработан стандарт MPI (Message Passing Interface) [3]. Система программирования MPI предназначена для ЭВМ с индивидуальной памятью. В модели передачи сообщений процессы, выполняющиеся параллельно, имеют раздельные адресные пространства. Обмен происходит, когда часть адресного пространства одного процесса скопирована в адресное пространство другого процесса.

Эта операция совместная и возможна только, когда первый процесс выполняет операцию передачи сообщения, а второй – операцию его получения.

В MPI используются коллективные операции, которые можно разделить на два вида:

- операции перемещения данных между процессами. Самая простая из них – широковещание. MPI имеет много и более сложных коллективных операций передачи и сбора сообщений;
- операции коллективного вычисления (минимум, максимум, сумма и другие, в том числе и определяемые пользователем операции).

В MPI имеются как блокирующие операции обменов, так и неблокирующие их варианты, благодаря чему окончание этих операций может быть определено явно. MPI также имеет несколько коммуникационных режимов. Синхронный режим требует блокировать посылку на время приема сообщения в противоположность стандартному режиму, при котором посылка блокируется до момента захвата буфера. Режим по готовности (для посылки) – способ, предоставленный программисту, чтобы сообщить системе, что этот прием был зафиксирован. Буферизованный режим позволяет пользователю управлять буферизацией.

Технология программирования SPMD (Single Program Multiple Data) представляет собой способ программирования для MIMD систем, когда для каждого процессора пишется одна программа, но роль одного процесса в программе определяется самим процессом. Этот способ облегчает создание параллельных программ, так как параллельная программа разрабатывается так же, как и последовательная.

1.2 Численное решение СЛАУ

В этом подразделе описывается параллельная программа, реализующая методы Якоби и Гаусса - Зейделя для решения СЛАУ. В программе реализуется итерационный процесс, каждую итерацию которого можно описать схемой, изображённой на рисунке 2.

Задача решения систем линейных алгебраических уравнений (СЛАУ) в векторном виде может быть записана так:

$$A\vec{x} = \vec{b}, \tag{2}$$

Метод простой итерации якоби в координатной форме имеет вид (3).

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^k - \sum_{j=i+1}^{n} a_{ij} x_j^k \right); i = \overline{1,n}; k = 0, 1, 2, ... \tag{3}$$

Метод Гаусса - Зейделя в координатной форме для системы общего вида:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^k \right); \ i = \overline{1,n}; k = 0, 1, 2, \ldots \tag{4}$$

Метод Гаусса - Зейделя отличается метода Якоби лишь тем, что первая сумма в правой части итерационной формулы содержит компоненты вектора решения не на k-й, а на $(k+1)$-й итерации [5]. Т.е. используется часть уже вычисленных на $(k+1)$ итерации значений.

Для реализации параллельных вычислений необходимо выделить независимые части алгоритма. Рассмотрев координатную запись метода, заключаем, что вычисления могут выполняться независимо для каждой координаты $\vec{x}$. Необходимо иметь только значение вектора $\vec{x}$ на предыдущей итерации.

Из итерационной метода Гаусса - Зейделя формулы следует, что значение x_i^{k+1} не может быть вычислено, пока не будут вычислены значения до x_{i-1}^{k+1} включительно. Это означает, что для метода Гаусса - Зейделя невозможно выполнить декомпозицию в таком же виде, как для метода Якоби. Поэтому с целью проведения декомпозиции метод Гаусса - Зейделя был изменён, исходя из специфики разделения задач. Разделение задач в обоих методах происходит следующим образом:

1. Матрица $\mathbf{A}$ со столбцом свободных членов $\vec{b}$ разделяется по строкам примерно поровну между всеми узлами вычислительной сети, участвующими в вычислениях.

2. Задаётся начальное приближение $\vec{x}^0$.

3. Производится одна итерация соответствующего метода, проверяется погрешность вычислений. В методе Гаусса - Зейделя каждый процесс использует только те значения $(k+1)$ итерации, которые вычисляет сам. Такое изменение метода позволяет произвести декомпозицию, т.е. распараллелить метод, но ухудшает сходимость метода.

Программа написана на языке C, согласно стандарту MPI, что делает программу переносимой на любой суперкомпьютер или вычислительную сеть, для которой существует компилятор языка C, и реализация стандарта MPI.

Матрица СЛАУ содержится в классе `CMatrixPart`, искомый вектор – классе `tpdVector`. Для параллельной реализации, так как матрица распределяется по процессам, необходимо несколько расширить её представление.

```
typedef struct
{
    double* m_pdMatrix;
    unsigned long int m_size; // Количество строк в матрице
    /* Первая строка в данном процессе
     * есть строка номер m_firstline в общей матрице */
    unsigned long int m_firstline;
    /*  для ускоренного обмена между процессами */
    int *_sendcounts, *_displs;
} CMatrixPart;
```

По сравнению с последовательной реализацией, добавилось поле `m_firstline`.

```
// Содержит информацию о характеристиках параллельной программы
typedef struct
{
    unsigned int m_iRoot;
    unsigned int m_iMyRank;
    unsigned int m_iSize;
    MPI_Comm m_Comm;
} CThisProcess;

void ThisProcessCreate(CThisProcess* this_Process,
    int* pargc, char** pargv[])
{
    MPI_Init(pargc, pargv);
    this_Process->m_Comm = MPI_COMM_WORLD;
    this_Process->m_iRoot = 0;
    MPI_Comm_rank(this_Process->m_Comm,
```

```
        &this_Process->m_iMyRank);
    MPI_Comm_size(this_Process->m_Comm,
        &this_Process->m_iSize);
}

void ThisProcessDestroy(CThisProcess* this_Process)
{
    MPI_Finalize();
}
```

Чтобы разбить матрицу по процессам примерно поровну (что оптимально для гомогенной распределённой системы [4]), можно воспользоваться формулой:

```
pMatrix->m_size =
    (MATRIX_SIZE / g_Process.m_iSize)
    + ((MATRIX_SIZE % g_Process.m_iSize) >
    g_Process.m_iMyRank ? 1 : 0 );
```

Так задаётся размер в строках части матрицы, которая будет находится в процессе с номером `g_Process.m_iMyRank` при числе процессов `g_Process.m_iSize`.

Для такого построения структур данных у параллельной реализации всего два отличия от последовательной:

1. Вместо индекса `i`, обозначающего номер строки, стоит `_i + pMatrix->m_firstline`, что здесь и является номером строки. Этим достигается, что каждый процесс изменяет только ту часть вектора, которую он вычисляет,

2. В конце итерации выполнен вызов функции MPI:

```
MPI_Allgatherv(
    &pdAnswer[pMatrix->m_firstline],
    pMatrix->m_size, MPI_DOUBLE,
    pdAnswer, pMatrix->_sendcounts,
    pMatrix->_displs, MPI_DOUBLE,
    g_Process.m_Comm);
```

Все процессы должны принять одинаковое решение о том, нужно ли продолжать вычисления, либо точность уже достигнута, и вычисления следует прекратить. Это может происходить как на одном узле вычислительного кластера или на всех узлах, в зависимости от скорости передачи данных в сети. Эксперименты для Fast Ethernet показали, что более оптимально вычисление погрешности в каждом процессе.

Для метода Гаусса - Зейделя отличия заключаются только в том, что в функцию не передаётся вектор на предыдущей итерации, а все значения берутся из вектора, в котором должен быть возвращён ответ. Кроме значений, оставшихся с предыдущей итерации, используются уже вычисленные на данной итерации значения. В этом и заключается идея Зейделя [5]. Однако используется эта идея только в рамках одного процесса. В случае, если число процессов равно числу строк в матрице, по итерационной формуле этот метод будет совпадать с методом Якоби, и только в случае запуска программы на одном процессе итерационная формула этого метода совпадает с итерационной формулой метода Гаусса - Зейделя.

Эффективность программ решения СЛАУ рассмотрена далее.

1.3 Анализ эффективности параллельных вычислений, основанный на экспериментальных данных

Важной задачей является выявление причин, влияющих на эффективность параллельных программ, основанное на экспериментальных данных. Рассматривается стандартный метод оценки эффективности – профилирование, приводятся экспериментальные данные по эффективности параллельных программ.

Эффективность может характеризоваться временем выполнения программы, коэффициентом использования, ускорением (формула (5)). При этом эффективность зависит от выбранного метода решения задачи, реализации этого метода, используемой аппаратной части, её конфигурации, других условий. Для библиотек стандарта MPI – это библиотека профилирования MPE (Multi-Processing Environment).

1.3.1 Профилирование

Появление ошибок при написании любых программ естественно, так как решаемые ими задачи от природы сложны. Параллелизм сам по себе добавляет сложность в программу в виде необходимости декомпозиции задачи, то есть нахождения в ней подзадач, способных выполняться одновременно, а также в обеспечении коммуникации между процессами. Появляется новый класс ошибок – deadlocks. Deadlock – ситуация, когда один процесс запрашивает ресурс у другого процесса, или занятый другим процессом, в то время, как другой процесс ожидает ответа, например, от первого. Часть или все процессы находится в состоянии ожидания, дальнейшее выполнение программы не возможно [6]. Задача отладки параллельных программ в большинстве случаев сводится к отладке каждого процесса по отдельности плюс изучение поведения процессов при их взаимодействии.

Так как основной целью параллельных вычислений является увеличение быстродействия, процесс отладки также должен быть направленным на изучение эффективности. Профилирование позволяет представить программисту информацию о переходах между состояниями. Состояние определяется двумя событиями: переход в данное состояние и выход из него.

Путь решения этой задачи в MPE – сохранение последовательности событий в файлы, затем представить эту информацию в удобном для программиста виде. Библиотека построена в предположении, что часы на узлах распределённой системы можно считать синхронизированными.

1.3.2 Схемы работы программ

Программа – это средство решения поставленной задачи. С этой точки зрения любому алгоритму можно сопоставить модель “чёрного ящика”.

Рисунок 1: Модель алгоритма как чёрного ящика

Модель, представленная на рисунке 1, общая для всех рассматриваемых программ, для которых имеет смысл оценка эффективности. Среди операций, используемые алгоритмом, различаются те, что могут выполняться параллельно, и те, что не могут (скалярные). К операциям, не поддающимся распараллеливанию, можно отнести считывание входных параметров и выдачу результатов исполнения. Это объясняется тем, что задачи ставятся

централизованно, выходные данные являются результатом работы всей программы, а не какой-либо её части (иначе остальные части были бы бессмысленны).

Предположим, что алгоритм преобразования данных хорошо масштабируем, любые операции в нём могут выполняться одновременно. Тогда ускорение, которое будет получено при вычислении преобразования будет равно числу процессоров, на которых производится расчёт. Общее ускорение будет меньшим, и может быть расчитано согласно (6) при подставлении $a = \frac{T_{in}+T_{out}}{T_{in}+T_{tr}+T_{out}}$, где T_{in} – время инициализации, T_{tr} – время преобразования, T_{out} – время выдачи результата (на одном процессе).

Для параллельной программы MIMD каждый процесс приложения запускаясь, получает задание, исходя из своего номера, и работает по схеме, изображённой на рисунке 2.

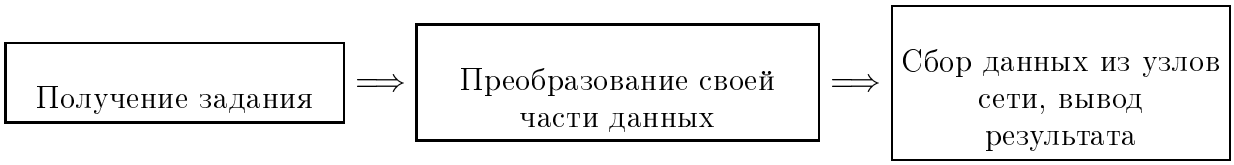

Рисунок 2: Схема работы процессов параллельных программ

По этой схеме работают все процессы параллельных программ MPI, но если процесс итерационный (например, как в программе решения СЛАУ), это схема одной итерации. Итерационные процессы можно представить как неоднократное выполнение схемы на рисунке 2. Падение эффективности в этом случае может произойти также в случае простоя процессоров, которые уже выполнили преобразование своей части данных и ожидают обмена данных от процесса, который ещё находится на этой стадии (см. рисунок 3).

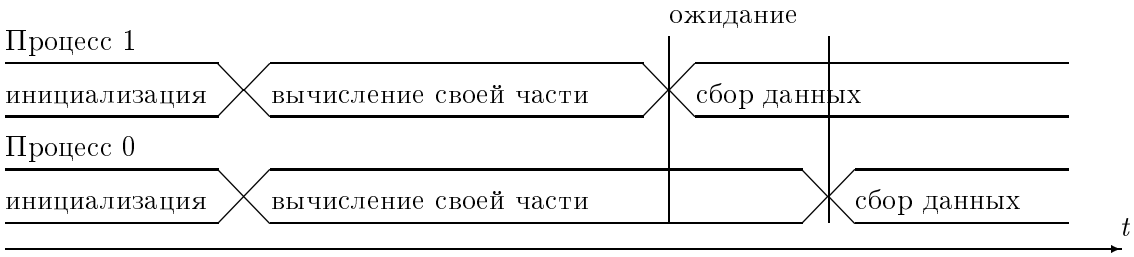

Рисунок 3: Потери из-за простоя процессора 1 при ожидании процесса 0

Это может произойти по причине того, что задача была неравномерно распределена на узлы вычислительной сети, или на одном из узлов в многозадачном режиме выполнялся другой трудоёмкий процесс. Следует отметить, что процесс параллельной программы – не единственный процесс на каждом узле вычислительной сети.

1.3.3 Эксперименты

На кафедре информатики на базе учебной лаборатории факультета радиофизики и электроники создан кластер, состоящий из 6-ти машин следующей конфигурации: Pentium 3-1000MHz с объёмом оперативной памяти 128 MB, дисковым накопителем объёмом 20 GB. Компьютеры соединены локальной сетью Fast Ethernet 100 Mbps посредством концентратора. Компьютеры управляются операционной системой Windows 2000 PE, установлен пакет MPICH 1.2.5.

Данный кластер сравнивался по производительности с учебным кластеров в Международной школе бизнеса (IGSBMT) при БГУ. Цель экспериментов – изучить влияние разных конфигураций на производительность вычислительных сетей.

Тестировались программы вычисления числа π (cpi), вычисление фракталов (mandel), решения СЛАУ методом Якоби. Программы запускались при различных условиях, влия-

ние которых изучалось, фиксировалось время работы программ на разном числе процессоров, а также на одном процессоре с разным числом процессов.

Изучение влияния разных конфигураций проводилось для программы решения СЛАУ методом Якоби на системах уравнений общего вида размерностью 1100×1100, так как именно такая размерность является достаточной для эффективного распараллеливания, и матрицы умещаются в оперативной памяти исследуемых вычислительных систем.

Программа запускалась несколько раз (3-5) для разного числа процессов, не более чем по одному процессу на процессор, для каждого запуска считалось среднее время выполнения одной итерации метода, затем выбиралось среднее, наименьшее и наибольшее время из всех экспериментов.

Более информативным для таких характеристик является минимальное время, поскольку время всегда ограничено снизу параметрами аппаратуры. С другой стороны, при использовании компьютеров более вероятным временем решения задачи является среднее время.

Время работы программы в каждом случае является случайной величиной. Это вызвано тем, что процессы параллельных программ запускаются в многозадачном режиме, и работают вместе с другими программами, используя общие аппаратные ресурсы (в частности центральный процессор). В меньшей степени, но также работа аппаратных ресурсов также подвержена случайным воздействиям внешних факторов (температуры, колебания напряжения питания, и др.).

Производительность параллельных вычислительных систем в первую очередь определяется производительностью узлов, чем она выше, тем меньше время решения задачи. Построение сети влияет на характер зависимости времени вычислений от числа процессов. Это определяется тем, что задача решения СЛАУ, для которой производилось тестирование, обладает малым удельным весом операций передачи данных.

1.3.4 Пути увеличения эффективности

Эффективность может характеризоваться следующими параметрами:

- Время выполнения – максимальное из времен выполнения программы на каждом процессоре.
- Коэффициент использования,
- Потерянное время – разница между временем выполнения, умноженным на число процессов, и временем последовательного выполнения.

Для вычисления коэффициента эффективности распараллеливания следует вычислить два времени:

1. Время, которое потребуется для выполнения программы на однопроцессорной ЭВМ.
2. Суммарное время параллельного выполнения, которое вычисляется как произведение времени выполнения на число процессоров.

Рассчитывать суммарное время параллельного выполнения важно именно таким образом (а не как сумму времен выполнения на всех процессорах) потому, что все процессоры предоставляются программе в момент ее запуска и освобождаются после окончания ее выполнения. Тогда отношению времени последовательного выполнения к суммарному времени параллельного выполнения соответствует коэффициенту использования.

Проведённые исследования позволили выявить причины недостаточно эффективного использования аппаратных ресурсов. При этом различаются следующие независимые компоненты потерянного времени:

1. потери из-за недостатка параллелизма, приводящего к дублированию вычислений на нескольких процессорах,

2. потери из-за выполнения межпроцессорных обменов,

3. потери из-за простоев тех процессоров, на которых выполнение программы завершилось раньше, чем на остальных.

Разбалансировка загрузки процессоров на разных этапах выполнения параллельной программы может приводить к тому, что возрастает время, затрачиваемое на синхронизацию процессоров. Поэтому разбалансировка и рассинхронизация должны выявляться.

Для коллективных операций, требующих синхронизации процессоров (редукция, загрузка буферов и др.), должно определяться время потенциальной рассинхронизации, то есть время, которое может быть потеряно из-за того, что процессоры начали выполнять коллективную операцию не одновременно. Рассинхронизация может возникать не только из-за разбалансировки, но также из-за различий во временах завершения коллективных операций на разных процессорах.

Тем не менее, кластеры на локальных сетях позволяют достичь значения коэффициента использования процессоров в пределах 0.6-0.9, производительность вычислительной системы главным образом зависит от производительности используемых процессоров, так как для исследуемого числа процессоров время вычислений преобладает над временем передачи.

1.4 Разработка формульной модели

Существующие средства профилирования параллельных программ позволяют получить информацию о ходе исполнения программы и оценить эффективность параллельных вычислений. Однако профилирование программы происходит на уровне функций MPI, рассмотреть возможность увеличения эффективности на более низких уровнях не представляется возможным. Такое исследование могло бы позволить проанализировать узкие места в используемом аппаратном обеспечении. Также средства профилирования не позволяют получить характеристики работы программ на вычислительных системах, которые проектируются, либо недоступны для исследований.

Путь решения этих проблем – моделирование работы программ. В этой работе рассматривается формульная модель, построенная на основе закона Амдала [7] и имитационная модель, рассмотренная в подразделе 1.5.

1.4.1 Сетевой закон Амдала

Одной из главных характеристик для оценки эффективности параллельных вычислений является ускорение R параллельной системы, которое определяется выражением (5).

$$R = T_1/T_n \tag{5}$$

где T_1 – время решения задачи на однопроцессорной системе, а T_n – время решения той же задачи на n-процессорной системе.

Пусть $W = W_{ck} + W_{np}$, где W – общее число операций в задаче, W_{np} – число операций, которые можно выполнять параллельно, а W_{ck} – число скалярных (нераспараллеливаемых) операций.

Обозначим также через t время выполнения одной операции. Тогда получаем известный закон Амдала:

$$R = \frac{W \cdot t}{\left(W_{ck} + \frac{W_{np}}{n}\right) \cdot t} = \frac{1}{a + \frac{1-a}{n}}. \tag{6}$$

Здесь $a = W_{ck}/W$ – удельный вес скалярных операций, зависящий от алгоритма.

Основной вариант закона Амдала не отражает потерь времени на межпроцессорный обмен сообщениями. Эти потери могут существенно повлиять на значение ускорения, в том числе и сильно уменьшить. Поэтому необходима некоторая модернизация выражения (6).

$$R = \frac{W \cdot t}{\left(W_{ck} + \frac{W_{np}}{n}\right) \cdot t_{\#} + W_c \cdot t_c} = \frac{1}{\left(a + \frac{1-a}{n}\right) \cdot \frac{t_{\#}}{t} + \frac{W_c \cdot t_c}{W \cdot t}} = \frac{1}{\left(a + \frac{1-a}{n}\right) \cdot \frac{t_{\#}}{t} + c}. \quad (7)$$

Здесь W_c – количество передач данных, t_c – время одной передачи данных, $t_{\#}$ - время одной операции для параллельной системы, которое может быть не равно времени t (например, из-за того, что при распределении исходных данных задачи по процессам в параллельной программе они вмещаются в кэш-память каждого узла, а в последовательной только в оперативную. Но как известно время обращения в кэш на порядок меньше чем в обычную память). Так как учет эффекта кэша в некоторой степени затруднителен, то мы для простоты, а также в связи с тем, что в реальных задачах при больших объемах данных влиянием кэша можно пренебречь, будем считать, что $t_{\#} = t$. Тогда формула (7) перепишется в следующем виде:

$$R = \frac{1}{\left(a + \frac{1-a}{n}\right) + c}. \quad (8)$$

Сетевой закон Амдала должен быть основой оптимальной разработки алгоритма и программирования задач, предназначенных для решения на многопроцессорных ЭВМ [7].

Коэффициент сетевой деградации вычислений c:

$$c = \frac{W_c \cdot t_c}{W \cdot t} = c_A \cdot c_T. \quad (9)$$

определяет объем вычислений, приходящийся на одну передачу данных (по затратам времени). При этом определяет алгоритмическую составляющую коэффициента деградации, обусловленную свойствами алгоритма, а c_T – техническую составляющую, которая зависит от соотношения технического быстродействия процессора и аппаратуры сети. Таким образом, для повышения скорости вычислений следует воздействовать на обе составляющие коэффициента деградации.

1.4.2 Аналитическая оценка времени выполнения параллельной программы

На основе сетевого закона Амдала создадим формульную модель, реализующую приведенный в 1.2 алгоритм Гаусса-Зейделя для СЛАУ.

Т.к. для рассматриваемого метода число итераций зависит от числа используемых процессоров, то будем рассчитывать время выполнения одной итерации в зависимости от числа процессов, а ускорение найдем из (5).

За время t возьмем время выполнения одного такта процессора, тогда, если f – частота процессора в герцах, время $t = 1/f$. В этом случае W будет обозначать не количество параллельных операций, а число тактов требуемых для их выполнения. Т. е. W будет равно:

$$W = \sum_{i=1}^{k} w_i \cdot t_i, \quad (10)$$

где k – количество видов исполняемых в задаче операций (умножение, деление, сложение и т. д.), w_i – число операций данного типа в задаче, t_i – время выполнения операции данного типа в тактах.

Согласно формуле (3) в нашем алгоритме только три основных типа операций: сложение, умножение и деление. Для расчета всего вектора длиной m нам понадобится m операций деления, $m*(m-1)$ операций умножения и столько же операций суммирования (операция вычитания приравнивается к операции сложения). Тогда получаем выражение:

$$W = m \cdot t_{div} + m \cdot (m-1) \cdot t_{mul} + m \cdot (m-1) \cdot t_{add} \tag{11}$$

После выполнения очередной итерации каждый процесс должен переслать всем остальным свою часть вычисленного вектора x^{k+1}, для дальнейшего продолжения работы, т. е. послать m/n элементов. Реализация функции `MPI_Allgather` в MPICH позволяет ограничиться вызовом $\{\log_2 n\}$ операций рассылок каждым процессом.

Такая схема выполнения коллективных операций позволяет равномерно распределить работу по рассылке сообщений на все процессы, тем самым увеличить эффективность использования аппаратных средств. Т.к. при первой посылке передается m/n элементов, при второй $2^*m/n$, при третьей $3^*m/n$ и т. д., применив формулу для суммы первых $\{\log_2 n\}$ членов арифметической прогрессии, получим количество пересылаемых элементов каждым из процессов после очередной итерации N:

$$N = \sum_{i=1}^{\{\log_2 m\}} \frac{im}{n} = \frac{m}{n} \frac{1 + \{\log_2 m\}}{2} \cdot \{\log_2 m\} \tag{12}$$

Окончательная формула для времени T_n принимает вид:

$$T_n = \frac{m \cdot (t_{div} + (m-1)t_{mul} + (m-1)t_{add})}{n} \frac{1}{f} + \frac{8N}{S} + L \cdot \{\log_2 n\} \tag{13}$$

Т.е. ускорение будет определяться как следующая зависимость:

$$R = T_1/T_n = R(t_{div}, t_{mul}, t_{add}, f, S, L, n, m) \tag{14}$$

Чтобы полученная формула (14) могла применяться для реального прогнозирования, необходимо проверить ее на опыте и, если понадобится, выявить и по возможности устранить недостатки. При этом путем подбора количества тактов приходящихся на арифметические операции добивалось максимальное сходство результатов как по времени так и по ускорению.

В случае использования концентратора вместо коммуатора процессы не могут передавать данные все одновременно, а только между какими-либо двумя узлами:

$$T_n = \frac{m \cdot (t_{div} + (m-1)t_{mul} + (m-1)t_{add})}{n} \frac{1}{f} + \left(\frac{8N}{S} + L \cdot \{\log_2 n\}\right) \cdot n \tag{15}$$

1.5 Создание моделирующей программы для оценки производительности кластеров

Рассмотренная в 1.4 модель показывает хорошие результаты для коммутатора и концентратора при небольлшом числе узлов вычислительной сети. Однако данные для концентратора несколько расходятся с результатами эксперимента. Это происходит из-за рассинхронизации процессов, которую не учитывает формульная модель. В этой главе описывается созданная программа для моделирования работы параллельных программ в стандарте MPI.

При создании моделирующей программы ставились следующие задачи:

- Выявление возможности увеличения производительности, составление списка причин, влияющих на эффективность параллельных программ.

- Выявление возможности увеличения производительности вычислительных систем за счёт оптимального использования аппаратных средств путём модификации программного кода (перераспределение вычислительной сложности) или аппаратной части.

1.5.1 Описание поведения моделирующей программы

Модель системы – это такое ее представление, которое состоит из определенного количества организованной информации о ней и построено с целью ее изучения. Поскольку существует очень много вопросов о системе, которые разумно задать, может быть сконструирован ряд различных моделей. Все эти модели представляют одну и ту же систему, но либо рассматривают ее с различных точек зрения и имеют различные цели, либо имеют различную степень детальности.

Моделирующая программа должна содержать сведения о моделируемой аппаратной подсистеме, и осуществлять преобразование

$$\text{операция} \rightarrow \text{время}$$

В этом случае операцией может быть как функция на языке программирования, так и целиком параллельная программа.

Программы запускаются в многозадачном режиме на компьютерах в виде одного или нескольких процессов, где также работают и другие процессы (включая ОС). Все процессы, выполняющиеся на одном компьютере, разделяют системные ресурсы. В итоге при обращении к этим устройствам процессы должны ожидать освобождения ресурсов другими процессами. Переключение задач в многозадачном режиме также занимает некоторое время.

Моделирование всех этих факторов не оправдано, так как

1. модель стала бы неоправданно сложной и её расчёт происходил бы дольше исполнения моделируемой программы,

2. время работы программы зависит от состояния вычислительной системы во время запуска программы, то есть случайным образом от времени запуска программы.

Работу программы на одном компьютере в многозадачном режиме можно рассматривать через взаимодействие процесса и компьютера, то есть аппаратной части и оставшихся процессов. Моделирующая программа должна осуществлять отображение

Рисунок 4: Схема моделирующей программы

Преимущества описания алгоритма с помощью кода программы:

- Обработка кода с помощью уже готового транслятора (компилятора или интерпретатора).

- Не нужно создавать свой язык описания и переписывать на нём исследуемый алгоритм.

Недостаток подхода описания исследуемого алгоритма с помощью кода программы в том, что программа с переопределёнными таким образом функциями и операторами будет работать дольше, чем исследуемая программа, что недопустимо. В принципе в таком виде результат можно получить с помощью библиотек профилирования (например MPE).

Описания с помощью специального языка описания алгоритмов может иметь следующие преимущества:

- Доступность модификации описания алгоритма без перекомпиляции.
- Быстрота исполнения, так как алгоритм на специальном языке описания может быть оптимальным для трансляции.

Недостатки подхода описания с помощью специального языка:

- Необходимость создания этого языка, который должен достаточно полно описывать алгоритм.
- Усложнение программы вследствие создания обработчка нового языка.

Компромиссным вариантом является создание простого языка описания программ (программы придётся описывать (почти переписывать), но этим мы добиваемся быстрой работы программы, имеется открытый, легко модифицируемый код), но этот язык будет совместим с уже существующим языком программирования (не нужно создавать новый транслятор, можно использовать уже готовый).

Отличие SPMD параллельной программы от последовательной в том, что программа запускается на нескольких узлах вычислительной сети и каждый процесс может знать свой номер, и, в соответствии с этим номером, выполнять ту или иную задачу. Под временем работы параллельной программы будем понимать время от первой по времени инициализации одного из процессов до последнего по времени завершения одного из процессов.

$$t = \max_{processes}(finish\_time) - \min_{processes}(start\_time) \tag{16}$$

Если в программе нет связей между процессами, достаточно проанализировать самый трудоёмкий учётом производительности узлов, либо посчитать время выполнения всех процессов, выбрав в качестве решения наибольшее. Если работа программы требует синхронизации процессов, необходимо рассматривать работу программы как последовательность из самых длинных ветвей между синхронизациями.

$$t = \sum_{branches}(\max_{processes}(finish\_branch) - start\_branch) \tag{17}$$

где $branches$ – ветви процессов, $\max_{processes}(finish\_branch)$ – наибольшее время завершения каждой ветви, $start\_branch$ – время начала этой же ветви в этом же процессе. У каждого процесса может быть свой набор ветвей, отличающийся от других процессов даже количеством ветвей.

1.5.2 Разработка структуры программы

Результатом работы моделирующей программы является время работы моделируемой программы. В вырожденном случае модель могла бы просто возвратить измеренное время работы параллельной программы в живом эксперименте. Разумеется, в таком виде

модель не отвечает поставленной задаче, поэтому необходимо иметь более детальные данные о времени исполнения операций, входящих в исследуемую программу. Иными словами, помимо входных и выходных данных, моделирующая программа должна оперировать параметрами аппаратной части.

В модели, как и в реальной ситуации, кластер строится из машин, соединённых в сеть. Необходимо описать поведение каждого узла вычислительной сети, исходя из его характеристик, а также описать работу коммутационной части, исходя из её характеристик. Операции, выполняемые локально на одном компьютере могут моделироваться так, как это описано в 1.4. Для моделирования сетей на концентраторе и коммутаторе учитывались также основные особенности работы этих устройств.

Сеть в модели рассматривается как средство передачи сообщений. Для того, чтобы отличать сообщения друг от друга, сообщение описывается как набор из номера посылающего процесса, номера принимающего процесса, тэга, а также размера сообщения и времени его отправки. Номера посылающего и принимающего процессов, а также тэг позволяют точно идентифицировать сообщение. С помощью времени отправки и размера сообщения вычисляется время прибытия сообщения раньше которого сообщение не может быть принято. Если принимающий процесс начал прием сообщения ранее, чем оно пришло по сети, то этот процесс будет вынужден ожидать прибытие сообщения, простаивая. Именно за счёт уменьшения времени простаивания процессов возможно более эффективное использование машинных ресурсов.

Модель сети представляет собой контейнер отправленных сообщений. Операция отправки сообщения представляется добавлением сообщения в контейнер, плюс увеличение текущего времени процесса на величину, необходимую для отправки сообщения. Это время может быть измерено экспериментально. Приём сообщений – более сложная операция.

Если процесс пытается принять сообщение во время, когда сообщение уже прибыло, сообщение помечается как полученное (чтобы одно сообщение было доставлено только единожды), затем ко времени процесса добавляется время, затрачиваемое им на получение сообщения. Это время также может быть измерено экспериментально.

Если сообщение ещё не пришло, то время работы процесса увеличивается до времени доставки сообщения. Время, которое проходит от момента доставки сообщения до его получения (выхода из функции `MPI_Recv`) также не равно нулю. Оно также измеряется экспериментально и не зависит от свойств сети – оно тратится на уровнях MPI и операционной системой, поэтому для экспериментального измерения этого времени нужно только запустить тест на одной из машин, по конфигурации схожей с теми, что составляют исследуемую вычислительную сеть.

Время доставки сообщений зависит от времени их отправки, размера сообщений, а также свойств сети. Отдельно рассматриваются модели сети на концентраторе и коммутаторе. Если сеть занята передачей сообщения, то все другие сообщения, которые должны быть переданы в данный момент ожидают, пока сеть освободится. Когда сеть освободилось, из готовых к передаче сообщений выбирается одно случайным образом, и т.д. Сеть на коммутаторе считается работающей параллельно, за исключением случаев, когда два сообщения посланы одному процессу.

1.5.3 Результаты моделирования

В качестве параметров модели использовались экспериментальные данные по эффективности аппаратных средств. На рисунке 5 представлены результаты моделирования вместе с экспериментальными данными.

Два нижних графика – время работы метода Якоби на учебном кластере в Международной школе бизнеса при БГУ (IGSBMT).

Используя моделирующую программу, можно исследовать зависимость характеристик вычислительных систем, в зависимости от их конфигурации. Рассмотрим зависимость

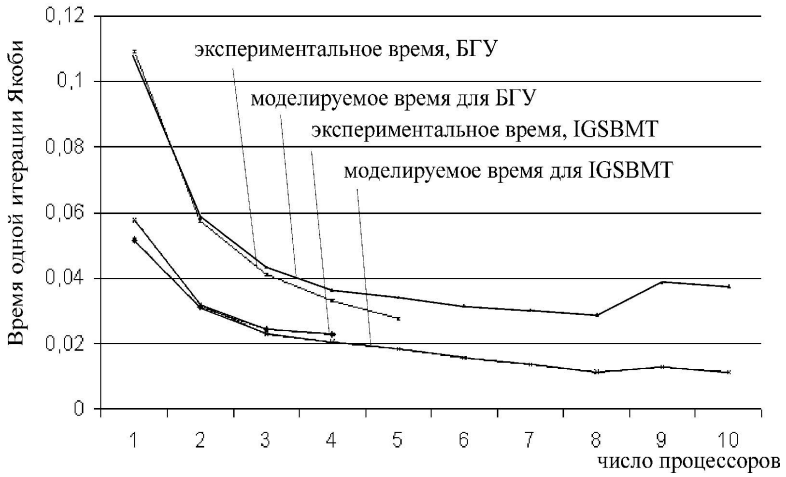

Рисунок 5: Результаты моделирования

времени выполнения программы решения СЛАУ от того, используется ли для построения сети концентратор, или коммутатор, при прочих одинаковых параметрах (см. рисунок 6). Результаты для одного процесса совпадают (передачи данных нет), для двух – различаются незначительно. Далее различия более существенны. Это объясняется методом коллективного доступа, используемого в Fast Ethernet [8].

Рисунок 6: Разница результатов для концентратора и коммутатора

Программная модель позволяет не только расчитывать время работы параллельной программы, но и продемонстрировать причины недостаточной эффективности так же, как это делает программа визуализации логфайлов.

Согласно результатам моделирования, в результате перераспределения загрузки процессоров можно добиться более эффективного использования вычислительных ресурсов за счёт неявного регламентирования порядка обращений к среде передач данных. Для примера с методом Якоби на матрицах порядка 1000 время, потраченное на передачу данных, удалось сократить на 20% для пяти процессов.

Получены формульная и имитационная модель для программы с параллелизмом по данным на примере метода простой итерации Якоби для СЛАУ. На модели показано, что в случае реализации совмещения обменов с вычислениями возможно увеличение эффективности программ за счёт перераспределения нагрузки с учётом реализации операций обмена. Моделирующая программа позволяет проанализировать работу параллельных программ на уровне передачи сообщений. Моделирующая программа может быть использована для построения новых кластеров, а также анализа эффективности существующих.

2 Разработка параллельных алгоритмов обучения НС на кластерах ЭВМ

2.1 Назначение и организация НС

Идея НС родилась в ходе исследований искусственно интеллекта, а именно, в результате попыток воспроизвести способность биологических нервных систем обучаться и исправлять ошибки, моделируя низкоуровневую структуру мозга.

Мозг состоит из очень большого числа (приблизительно 10^{10}) нейронов, соединенных многочисленными связями (в среднем несколько тысяч связей на один нейрон). Нейроны – это особые клетки, способные распространять электрохимические сигналы. Нейрон активируется тогда, когда суммарный уровень сигналов, пришедших в его ядро из дендритов, превысит определенный уровень (порог активации).

Таким образом, будучи построен из очень большого числа совсем простых элементов (каждый из которых берет взвешенную сумму входных сигналов и в случае, если суммарный вход превышает определенный уровень, передает дальше двоичный сигнал), мозг способен решать чрезвычайно сложные задачи.

Чтобы отразить суть биологических нейронных систем, определение искусственного нейрона дается следующим образом (см. рисунок 7)

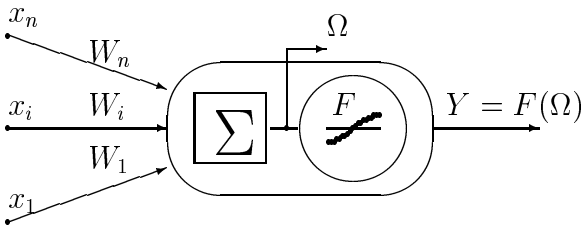

Рисунок 7: Модель искусственного нейрона

Нейрон получает входные сигналы (исходные данные или выходные сигналы других нейронов НС) через несколько входных каналов. Каждый входной сигнал x_i проходит через соединение, имеющее определенную интенсивность (или вес) W_i. Этот вес соответствует синаптической активности биологического нейрона. Положительные веса соответствуют возбуждающим связям, а отрицательные – тормозящим. С каждым нейроном связано определенное пороговое значение. Вычисляется взвешенная сумма входов Ω, из нее вычитается пороговое значение и в результате получается величина активации нейрона (она также называется постсинаптическим потенциалом нейрона – PSP). Сигнал активации преобразуется с помощью функции активации (или передаточной функции) и в результате получается выходной сигнал нейрона.

Математическая модель нейрона:

$$S = \sum_{i=1}^{N} w_i x_i + b \tag{18}$$

$$y = f(S) \tag{19}$$

где w_i – вес синапса (weight), $(i = 1, 2...N)$; b – значение смещения (bias); s – результат суммирования (sum); x_i – компонента входного вектора (входной сигнал), $(i = 1, 2, ...N)$; y – выходной сигнал нейрона; N – число входов нейрона; f – нелинейное преобразование (функция активации).

В общем случае входной сигнал, весовые коэффициенты и значения смещения могут принимать действительные значения. Выход $\vec{y}$ определяется видом функции активации и может быть как действительным, так и целым. Во многих практических задачах входы, веса и смещения могут принимать лишь некоторые фиксированные значения.

Будучи соединенными определенным образом, нейроны образуют нейронную сеть. ИНС можно рассматриваться как направленный граф с взвешенными связями, в котором искусственные нейроны являются узлами. По архитектуре связей ИНС могут быть сгруппированы в два класса: сети прямого распространения, в которых графы не имеют петель, и рекуррентные сети, или сети с обратными связями. Если нейроны каждого слоя НС имеют единую функцию активации, то такую нейронную сеть называют однородной.

Нейрокомпьютеры обладают целым рядом свойств, привлекательных с точки зрения их практического использования [9]:

- сверхвысокое быстродействие за счет использования массового параллелизма обработки информации;
- толерантность к ошибкам: работоспособность сохраняется при повреждении значительного числа нейронов;
- способность к обучению; программирование вычислительной системы заменяется обучением;
- способность к распознаванию образов в условиях сильных помех и искажений.

Использование нейроструктур особо важно с точки зрения производительности ЭВМ. Согласно гипотезе Минского [9], реальная производительность типовой параллельной вычислительной системы из n процессоров растет как $\log(n)$, так как процессоры дольше ждут своей очереди, чем вычисляют. Однако, если использовать для решения задачи НС, то параллелизм может быть использован практически полностью и производительность растет почти пропорционально n. Это возможно только на специализированных нейросетевых архитектурах.

Следовательно, основные преимущества нейрокомпьютеров связаны с массовым параллелизмом обработки, что обуславливает высокое быстродействие, низкие требования к стабильности и точности параметров элементарных узлов, устойчивость к помехам и разрушениям при большой пространственной размерности системы, причем устойчивые и надежные нейросистемы могут создаваться из низконадежных элементов, имеющих большой разброс параметров.

В добавление к классификации Флинна (стр. 6), нейрокомпьютер это вычислительная система с MSIMD-архитектурой, то есть с параллельными потоками одинаковых команд и множественным потоком данных [9]. Отмечается три основных направления развития вычислительных систем с массовым параллелизмом (ВСМП):

- ВСМП на базе каскадного соединения универсальных SISD, SIMD, MISD микропроцессоров: элементная база универсальные RISC или CISC-процессоры (Intel, AMD, Sparc, Alpha, Power PC, MIPS и т.п.);
- на базе процессоров с распараллеливанием на аппаратном уровне: элементная база DSP-процессоры (TMS, ADSP, Motorola);
- ВСМП на специализированной элементной базе от специализированных однобитовых процессоров до нейрочипов.

Для каждого из направлений сегодня существуют решения, реализующие те или иные нейросетевые парадигмы. Нейросетевые системы, реализованные на аппаратных платформах первого направления (пусть и мультипроцессорных) будем относить к нейроэмуляторам, то есть системам, реализующим типовые нейрооперации (взвешенное суммирование и нелинейное преобразование) на программном уровне.

Нейросетевые системы, реализованные на аппаратных платформах второго и третьего направления в виде плат расширения вычислительных систем первого направления,

называются нейроускорителями. Системы, реализованные на аппаратной платформе третьего направления в виде функционально законченных вычислительных устройств, следует относить к нейрокомпьютерам (все операции выполняются в нейросетевом логическом базисе).

Нейрокомпьютеры относятся к вычислительным системам с высоким параллелизмом (MSIMD-архитектуры), реализованным на основе специализированной элементной базы, ориентированной на выполнение нейросетевых операций в нейросетевом логическом базисе. НС находят успешное применение в самых различных областях: бизнесе, медицине, технике, геологии, физике. НС вошли в практику везде, где нужно решать задачи прогнозирования, классификации или управления, поскольку они применимы практически в любой ситуации, когда имеется связь между переменными-предикторами (входами) и прогнозируемыми переменными (выходами), даже если эта связь имеет очень сложную природу и ее трудно выразить в обычных терминах корреляций или различий между группами. НС успешно применяются для решения следующих задач: классификация образов, кластеризация (категоризация), аппроксимация функций, предсказание или прогноз, оптимизация и некоторые другие.

2.2 Постановка задачи обучения НС

Фундаментальным свойством мозга является его способность к обучению. Для ИНС процесс обучения можно рассматривать как настройку архитектуры НС и весов связей для эффективного выполнения специальной задачи. Обычно НС должна настроить веса связей по имеющейся обучающей выборке. По мере итеративной настройки весовых коэффициентов функционирование НС улучшается.

Существуют три парадигмы обучения: “с учителем”, “без учителя” (самообучение) и смешанная [11]. В первом случае НС располагает правильными ответами (выходами НС) на каждый входной пример. Веса настраиваются так, чтобы сеть производила ответы как можно более близкие к известным правильным ответам. Усиленный вариант обучения с учителем предполагает, что известна только критическая оценка правильности выхода НС, но не сами правильные значения выхода. Обучение без учителя не требует знания правильных ответов на каждый пример обучающей выборки. В этом случае раскрывается внутренняя структура данных или корреляции между образцами в системе данных, что позволяет распределить образцы по категориям. При смешанном обучении часть весов определяется посредством обучения с учителем, в то время как остальная получается с помощью самообучения.

Теория обучения рассматривает три фундаментальных свойства, связанных с обучением по примерам: емкость, сложность образцов и вычислительная сложность. Под емкостью понимается, сколько образцов может запомнить сеть, и какие функции и границы принятия решений могут быть на ней сформированы. Сложность образцов определяет число обучающих примеров, необходимых для достижения способности НС к обобщению. Слишком малое число примеров может вызвать “переобученность” сети, когда она хорошо функционирует на примерах обучающей выборки, но плохо на тестовых примерах, подчиненных тому же статистическому распределению.

При обучении НС, в частности многослойных персептронов, существует две не решённые (в общем случае) проблемы:

1. Проблема выбора оптимальной конфигурации сети, т.е. числа скрытых слоёв нейронов и количества нейронов в каждом слое.

2. Проблема выбора начальных весовых коэффициентов. Как правило, начальные веса задаются случайным образом. В то же время эффективность алгоритмов обучения в значительной степени зависит от того, насколько удачным оказался выбор начальных весовых коэффициентов.

При выборе структуры НС решается вопрос ёмкости НС, то есть числа образов, предъявляемых на её входы, которые она способна научиться распознавать. Для сетей с числом слоев больше двух, он остается открытым. Как показано в [10], для НС с двумя слоями, то есть выходным и одним скрытым слоем, детерминистская емкость НС C_d оценивается так:

$$N_w/N_y < C_d < N_w/N_y \cdot \log(N_w/N_y) \tag{20}$$

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

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, обычно больше [10]. Кроме того, фигурирующее в названии ёмкости прилагательное "детерминистский" означает, что полученная оценка ёмкости 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 ёмкость. Так как распределение образов, в общем случае, заранее не известно, мы можем говорить о такой ёмкости только предположительно, но обычно она раза в два превышает ёмкость детерминистскую.

Затронем вопрос о требуемой мощности выходного слоя НС, выполняющего окончательную классификацию образов. 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 НС, организованной таким образом, не очень надёжны. 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.

Оценим способность НС решить задачу. НС вычисляет некоторую вектор-функцию F от входных сигналов. Эта функция зависит от параметров сети. Обучение сети состоит в подборе такого набора параметров сети, чтобы величина $\sum\limits_{i,j}(F_j(x^i) - f_j^i)^2$ была минимальной (в идеале равна нулю). Для того, чтобы НС могла хорошо приблизить заданную таблично функцию f, необходимо, чтобы реализуемая сетью функция F при изменении входных сигналов с x_i на x_j могла изменить значение с f^i на f^j . Очевидно, что наиболее трудным для сети должно быть приближение функции в точках, в которых при малом изменении входных сигналов происходит большое изменение значения функции. Таким образом, наибольшую сложность будет представлять приближение функции f в точках, в которых достигает максимума выражение $\frac{\|f^i - f^j\|}{\|x_i - x_j\|}$. Для аналитически заданных функций величина $\sup\limits_{x,y} \frac{\|f(x)-f(y)\|}{\|x-y\|}$ называется константой Липшица. Исходя из этих соображений, можно дать следующее определение сложности задачи.

Сложность аппроксимации таблично заданной функции f, которая в точках x_i принимает значения f^i, задаётся выборочной оценкой константы Липшица, вычисляемой по следующей формуле:

$$\Lambda_f = \max_{i \neq j} \frac{\|f^i - f^j\|}{\|x_i - x_j\|} \tag{21}$$

Оценка (21) является оценкой константы Липшица аппроксимируемой функции снизу. Для того, чтобы оценить способность сети заданной конфигурации решить задачу,

необходимо оценить константу Липшица сети и сравнить ее с выборочной оценкой (21). Константа Липшица сети вычисляется по следующей формуле:

$$\Lambda_{NN} = \sup_{x,y} \frac{\|F(x) - F(y)\|}{\|x - y\|} \tag{22}$$

В формулах (21) и (22) можно использовать произвольные нормы. Однако для НС наиболее удобной является евклидова норма.

Очевидно, что в случае $\Lambda_f > \Lambda_{NN}$ сеть принципиально не способна решить задачу аппроксимации функции f. Оценка константы Липшица сети строится в соответствии с принципом иерархического устройства сети.

Существует два подхода, помогающие исследователю выбрать оптимальную структуру НС.

- Растущие сети, конструктивный подход

  Алгоритм начинается с выбора небольшой НС, имеющей, допустим, один скрытый нейрон. Сеть обучается, пока изменение ошибки за одну итерацию не станет меньше заранее заданного порога. Затем мы добавляем еще один скрытый нейрон, инициализируем его веса и смещения случайными величинами и продолжаем обучение. Процесс продолжается до тех пор, пока добавление нейрона улучшает обучаемость сети.

- Уменьшающиеся сети, деструктивный подход Реализацией подхода может быть такая процедура: выбирается достаточно большая сеть, из которой последовательно удаляются нейроны. Процесс происходит следующим образом:

  – Обучение большой густо соединенной НС стандартным алгоритмом.
  – Анализ обученной сети на важность каждого веса.
  – Удаление наименее важного веса (весов).
  – Переобучение сети.
  – Повторение шагов 2 - 4.

  Решение о важности веса является непростым, при этом используются несколько эвристических подходов.

Метод случайной стрельбы Идея метода случайной стрельбы состоит в генерации большой последовательности случайных точек и вычисления оценки в каждой из них. При достаточной длине последовательности минимум будет найден.

Остановка данной процедуры производится по команде пользователя или при выполнении условия, что ошибки стала меньше некоторой заданной величины. Существует огромное разнообразие модификаций этого метода. Наиболее простой является метод случайной стрельбы с уменьшением радиуса. Параметры, задаваемые пользователем:

- Число попыток – число неудачных пробных генераций вектора при одном радиусе.

- Минимальный радиус – минимальное значение радиуса, при котором продолжает работать алгоритм.

Идея этого метода состоит в следующем. Новый вектор параметров будем искать как сумму начального и случайного, умноженного на радиус, векторов. Если после заданного числа попыток случайных генераций не произошло уменьшения оценки, то уменьшаем радиус. Если произошло уменьшение оценки, то полученный вектор объявляем начальным и продолжаем процедуру с тем же шагом.

Метод покоординатного спуска Этот метод относят к псевдоградиентным. Идея этого метода состоит в том, что если в задаче сложно или долго вычислять градиент, то можно построить вектор, обладающий приблизительно теми же свойствами, что и градиент следующим путем. Даем малое положительное приращение первой координате вектора. Если оценка при этом увеличилась, то пробуем отрицательное приращение. Далее так же поступаем со всеми остальными координатами. В результате получаем вектор, в направлении которого оценка убывает. Для вычисления такого вектора потребуется, как минимум, столько вычислений функции оценки, сколько координат у вектора. В худшем случае потребуется в два раза большее число вычислений функции оценки. Время же необходимое для вычисления градиента в случае использования двойственных сетей можно оценить как 2-3 вычисления функции оценки. Таким образом, учитывая способность двойственных сетей быстро вычислять градиент, можно сделать вывод о нецелесообразности применения метода покоординатного спуска в обучении НС.

Метод случайного поиска Этот метод похож на метод случайной стрельбы с уменьшением радиуса, однако в его основе лежит другая идея – сгенерируем случайный вектор и будем использовать его вместо градиента. Этот метод использует одномерную оптимизацию – подбор шага. Параметры метода:

- Число попыток – число неудачных пробных генераций вектора при одном радиусе.
- Минимальный радиус – минимальное значение радиуса, при котором продолжает работать алгоритм.

Идея этого метода состоит в следующем. Новый вектор параметров будем искать как сумму начального и случайного, умноженного на радиус, векторов. Если после заданного числа попыток случайных генераций не произошло уменьшения оценки, то уменьшаем радиус. Если произошло уменьшение оценки, то полученный вектор объявляем начальным и продолжаем процедуру с тем же шагом. Важно, чтобы последовательность уменьшающихся радиусов образовывала расходящийся ряд.

Метод Нелдера-Мида Этот метод является одним из наиболее быстрых и наиболее надежных неградиентных методов многомерной оптимизации. Идея этого метода состоит в следующем. В пространстве оптимизируемых параметров генерируется случайная точка. Затем строится $n-$мерный симплекс с центром в этой точке, и длиной стороны l. Далее в каждой из вершин симплекса вычисляется значение оценки. Выбирается вершина с наибольшей оценкой. Вычисляется центр тяжести остальных n вершин. Проводится оптимизация шага в направлении от наихудшей вершины к центру тяжести остальных вершин. Эта процедура повторяется до тех пор, пока не окажется, что оптимизация не изменяет положения вершины. После этого выбирается вершина с наилучшей оценкой и вокруг нее снова строится симплекс с меньшими размерами. Процедура продолжается до тех пор, пока размер симплекса, который необходимо построить, не окажется меньше требуемой точности.

Однако, несмотря на свою надежность, применение этого метода к обучению НС затруднено большой размерностью пространства параметров.

Метод наискорейшего спуска Наиболее известным среди градиентных методов является метод наискорейшего спуска. Идея этого метода проста: поскольку вектор градиента указывает направление наискорейшего возрастания функции, то минимум следует искать в обратном направлении.

Алгоритм метода:

1. Инициализация всех весов малыми случайными величинами.

2. Повторение

3. Для каждого веса w_{ij} установить $\Delta w_{ij} := 0$

4. Для каждого элемента обучающего множества $(x, t)^p$

   (a) установить вход сети равным x

   (b) рассчитать выход сети

   (c) для каждого веса w_{ij} установить $\Delta w_{ij} := \Delta w_{ij} + (t_i - y_i)y_j$

5. Для каждого веса w_{ij} установить $w_{ij} := w_{ij} + \mu \quad \Delta w_{ij}$

Алгоритм завершается, когда достигнут минимум функции ошибки ($G = 0$). Это означает, что алгоритм сошёлся.

Скорость обучения. Одним из главных параметров алгоритма градиентного спуска является скорость обучения μ, которая определяет, насколько изменяются веса нейронов на каждом шаге. Если μ мало, то требуется много времени для нахождения минимума. Соответственно, если значение μ велико, алгоритм может бесконтрольно скакать по многомерной поверхности функции ошибок – алгоритм расходится

Метод наискорейшего спуска работает, как правило, на порядок быстрее методов случайного поиска. Он имеет два параметра:

- точность – показывает, что если изменение оценки за шаг метода меньше чем точность, то обучение останавливается
- шаг – начальный шаг для оптимизации шага. Заметим, что шаг постоянно изменяется в ходе оптимизации шага.

Недостатки у метода следующие. Во-первых, этим методом находится тот минимум, в область притяжения которого попадет начальная точка. Этот минимум может не быть глобальным. Существует несколько способов выхода из этого положения. Наиболее простой и действенный – случайное изменение параметров с дальнейшим повторным обучение методом наискорейшего спуска. Как правило, этот метод позволяет за несколько циклов обучения с последующим случайным изменением параметров найти глобальный минимум.

Вторым серьезным недостатком метода наискорейшего спуска является его чувствительность к форме окрестности минимума. Для точного достижения минимума потребуется бесконечное число шагов метода градиентного спуска. Этот эффект получил название овражного, а методы оптимизации, позволяющие бороться с этим эффектом – антиовражных.

Алгоритм обратного распространения ошибки (Backpropagation) Когда в НС только один слой, алгоритм ее обучения с учителем 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 персептрона [11]. В многослойных же сетях оптимальные выходные значения нейронов всех слоев, кроме последнего, как правило, не известны, и персептрон с двумя или более слоями 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.

Теория обучения многослойного персептрона была изложена в статье Румельхарта, Хинтона и Уильямся в 1986. Базовый алгоритм метода обратного распространения ошибки был изложен в диссертации Пола Вербоса в 1974 [11]. Идея метода – 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 [10].

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

$$E(w) = \frac{1}{2}\sum_{j,p}(y_{j,p}^{(N)} - d_{j,p})^2 \tag{23}$$

где $y_{j,p}^{(N)}$ – реальное выходное состояние нейрона j выходного слоя N НС 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:

$$\Delta w_{ij}^{(n)} = -\eta \cdot \frac{\partial E}{\partial w_{ij}} \quad . \tag{24}$$

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

Воспользуемся соотношением:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \cdot \frac{dy_j}{ds_j} \cdot \frac{\partial s_j}{\partial w_{ij}} \tag{25}$$

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 $\frac{dy_j}{ds_j}$ 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 сигмойд с экспонентой. В случае гиперболического тангенса

$$\frac{dy}{ds} = 1 - s^2 \tag{26}$$

Третий множитель $\frac{\partial s_j}{\partial w_{ij}}$ есть выход нейрона предыдущего слоя $y_i^{(n-1)}$.

Множитель $\frac{\partial E}{\partial y_j}$ можно представить следующим образом:

$$\frac{\partial E}{\partial y_j} = \sum_k \frac{\partial E}{\partial y_k} \cdot \frac{dy_k}{ds_k} \cdot \frac{\partial s_k}{\partial y_j} = \sum_k \frac{\partial E}{\partial y_k} \cdot \frac{dy_k}{ds_k} \cdot w_{jk}^{(n+1)} \tag{27}$$

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

Введя новую переменную

$$\delta_j^{(n)} = \frac{\partial E}{\partial y_j} \cdot \frac{dy_j}{ds_j} \tag{28}$$

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

$$\delta_j^{(n)} = \left[\sum_k \delta_k^{(n+1)} \cdot w_{jk}^{(n+1)}\right] \cdot \frac{dy_j}{ds_j} \tag{29}$$

Для выходного же слоя

$$\delta_l^{(N)} = (y_l^{(N)} - d_l) \cdot \frac{dy_l}{ds_l} \tag{30}$$

Теперь приращение весов мы можем записать в раскрытом виде:

$$\Delta w_{ij}^{(n)} = -\eta \cdot \delta_j^{(n)} \cdot y_i^{(n-1)} \tag{31}$$

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

$$\Delta w_{ij}^{(n)}(t) = -\eta \cdot (\mu \cdot \Delta w_{ij}^{(n)}(t-1) + (1-\mu) \cdot \delta_j^{(n)} \cdot y_i^{(n-1)}) \tag{32}$$

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

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

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 $f(x)$ – активационная функция $y_q^{(0)} = x_q$, где x_q – q-ая компонента вектора входного образа.

2. Рассчитать $\delta^{(N)}$ для выходного слоя по формуле (30).

Рассчитать по формуле (31) или (32) изменения весов $\Delta w^{(N)}$ слоя N.

3. Рассчитать по формулам (29) и (31) (или (29) и (32)) соответственно $\delta^{(n)}$ и $\Delta w^{(n)}$ для всех остальных слоев, $n = N-1, ...1$.

4. Скорректировать все веса в НС

$$w_{ij}^{(n)}(t) = w_{ij}^{(n)}(t-1) + \Delta w_{ij}^{(n)}(t) \tag{33}$$

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

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

Из выражения (31) 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, 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

$$f(x) = -0.5 + \frac{1}{1+e^{-\alpha \cdot x}} \tag{34}$$

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 (29) и (30) 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 стабилизируются, 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.

Существуют несколько способов предупреждения переобучения:

- Обучение на меньшем количестве итераций. Этот метод получил название ранней остановки
- Уменьшение числа нейронов, что приводит к уменьшению параметров (весов), поэтому сеть не может переобучиться на деталях.
- Начать с большой сети, а затем постепенно ее уменьшать до достижения оптимального результата.
- Наложить ограничение на величины весов и смещений сети, что приводит к уменьшению параметров. Например, метод ослабления веса:

$$E = MSE + \alpha w^2 w(new) = w(old) - \mu \frac{\partial E}{\partial w} - \gamma w \tag{35}$$

В общем случае неизвестна структура, необходимая для идеального решения задачи.

Компромисс между ограниченностью и вариантностью Есть данные, полученые от гладкой функции $h(x)$ с добавлением шума. Требуется получить модель, которая аппроксимирует $h(x)$, имея данные, сгенерированные по закону $y(x) = h(x) + \varepsilon$. Можно описать данные с помощью функции $g(x)$, которая имеет малое количество параметров. Достоинство такой модели – простота, она имеет лишь два свободных параметра. Тем не менее, эта модель плохо описывает данные и, следовательно, будет плохим предсказателем. В таком случае говорят, что модель слишком ограничена. Второй вариант – модель, использующая слишком много свободных параметров. Она хорошо аппроксимирует данные, так как ошибка практически равна 0. Но и эта модель будет плохо предсказывать результат. Говорят, что такая модель обладает слишком большой вариативностью. Нам нужна модель, которая достаточно мощна, чтобы описать структуру данных, но не настолько, чтобы запомнить шумовую компоненту. Компромисс между ограниченностью и вариативностью является реальной проблемой в случае малых размеров обучающего множества. Если же данных достаточно, то такой проблемы не возникает, так как шум каждого элемента играет незначительную роль в процессе обучения. Таким образом, те рекомендации, которые будут даны ниже, относятся к ограниченному набору данных и пакетному режиму обучения.

Наиболее простым методом является разделение данных на два множества: обучающее и проверочное. Обучение происходит только на обучающем множестве. На каждой итерации обученная модель тестируется на независимом проверочном множестве. С каждой новой итерацией качество работы на проверочном множестве будет улучшаться. Как только модель обучится общим закономерностям в данных и начнет запоминать особенности каждого элемента, результат предсказания на проверочном множестве ухудшится. В этот момент времени необходимо прекратить обучение.

Также известен подход, носящий название “ослабления веса”. В случае переобученности функция имеет волнистую форму, в то время как линейная функция максимально сглажена. Регуляризация представляет собой набор методик, которые помогают избежать излишней волнистости функции. Это достигается добавлением штрафной величины к функции ошибки $\tilde{E} = E + \upsilon\Omega$. В случае переобучения веса нейронов имеют очень большие значения.

Мы постараемся уменьшить их, выбирая: $\Omega = \frac{1}{2}\sum_i w_i^2$. Тогда веса будут меняться по закону: $\Delta w_{ij} = -\mu\frac{\partial E}{\partial w_{ij}} - \mu v w_{ij}$, где правое слагаемое будет уменьшать величину веса нейрона в зависимости от самой себя. В отсутствии любого входа все веса будут экспоненциально уменьшаться, отсюда и название метода – ослабления веса.

Последний метод – это добавление незначительного шума (малая случайная величина со нулевым средним) к каждому элементу: $\tilde{x} = x + \epsilon$. Поначалу это может показаться странным принудительно испортить данные, но теперь сети намного сложнее аппроксимировать детали каждой точки. Обучение с добавочным шумом доказало свою полезность на практике. Если обучающее множество ограничено, то альтернативой является обучение в оперативном режиме.

2.3 Программная реализация нейроэмулятора

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 [10].

НС делает преобразование

$$\vec{y}^T = \hat{N}\vec{x}^T \tag{36}$$

где $\vec{x}$ – вектор входов, $\vec{y}$– вектор выходов, $\hat{N}$ – оператор НС. Для линейной однослойной сети N – матрица, каждая строка которой содержит веса нейрона, количество строк равно числу нейронов слоя, количество столбцов – размерность входного вектора. То есть прохождение слоя НС свелось к умножению вектора на матрицу. Эта операция происходит на любом слое любой НС, разница может быть только в последующем вычислении активационной функции, если сеть нелинейная или к конкатенации вектора $\vec{x}$ с дополнительными данными, если сеть имеет обратную связь.

Такой подход может обеспечить 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. Это позволяет создать гибкую, легко перестраиваемую иерархию моделей НС. Такая реализация наиболее прозрачна для программиста, и позволяет конструировать НС даже непрограммистам. Уровень абстрактности программирования, присущий ОО языкам, в будущем будет, по-видимому, расти, и реализация НС с ОО подходом позволит расширить их возможности.

Обычно в ОО реализациях архитектуру классов строят, начиная с класса “нейрон”. Такой подход имеет недостаток: элементы матрицы N некомпактно содержатся в памяти ЭВМ, что замедляет работу кэша процессора. Компактное хранение матрицы делает работу кэша более эффективной. Поэтому была выбрана структура классов, базовым в которой является один слой НС. Слой – объединение входов и весов нейронов, которые могут обрабатываться одновременно, в случае сети с обратной связью данные, идущие на предыдущий слой просто копируются во вход этого слоя.

Нейрон как таковой выполняет вычисление скалярного произведения вектора своих весов и вектора всех входов слоя и активационной функции. Слой есть матрица весов и вектор входных данных. Объект “слой” для всех топологий НС одинаков. В случае параллельной реализации в каждом процессе будет свой слой, но у него будет только часть нейронов, следовательно только часть строк матрицы весов, и результатом будет только часть вектора выходов, в то время как входы будут одинаковы во всех процессах. Сбор всего слоя данных можно выполнять с помощью `MPI_Allgatherv`. Таким образом, объект “слой” одинаков для параллельной и последовательной реализации.

Для сравнения производительности реализация созданная таким образом реализация **pneuro** сравнивалась с известной библиотекой annie (Artificial Neural Network Library)

Version 0.51 Author(s): Asim Shankar, доступной по адресу http://annie.sourceforge.net. Это наиболее полная бесплатная объектно-ориентированная реализация НС. Результаты представлены в таблице 1.

Таблица 1: Соответствие числа нейронов в реализации **pneuro** и **annie** по времени выполнения прямого прохождения

N1	140	170	180	200	220	280
N2	50	70	80	90	100	110
N1/N2	2,79	2,43	2,25	2,22	2,20	2,54

Здесь N1 – число нейронов в реализации pneuro, а N2 – в annie, в соответствующих столбцах время прохождения данных через однослойные сети этих реализаций одинаково. Среднее отношение размеров сетей равно $2,4$. Реализация annie в $2,4$ раза критичнее к памяти при прямом прохождении данных, чем реализация, в которой иерархия классов начинается с класса "слой" (а не нейрон). Этот вывод сделан исходя из того, что результаты annie выдаёт аналогичные временные результаты, что и pneuro с размерностью данных в $2,4$ раза меньшим. Однако для небольших сетей это не критично, следует отдать должное этой реализации, так как она наиболее полная из бесплатно распространяемых в сети Интернет.

Охарактеризуем основные классы в реализации **pneuro**. Класс `CNNLayer` – описывает слой НС. Выделение памяти происходит в функции

```
void CNNLayer::Construct(const unsigned int iThisLayerSize,
    const unsigned int iPrevLayerSize, CActi &a_funcacti);
```

Веса нейронов слоя устанавливаются функцией

```
void CNNLayer::SetRates(
    const SYGDATATYPE *a_pRates, const unsigned int size);
```

Функция

```
void CNNLayer::GetAscons(
    SYGDATATYPE *a_pOutput, const unsigned int size);
```

выполняет прохождение данных через сеть. Входной вектор задаётся в функции

```
void CNNLayer::SetInput(
    const SYGDATATYPE *a_pInput, const unsigned int size);
```

Присутствующий здесь тип данных SYGDATATYPE - макроопределение, фактически означающее double, однако этот тип данных можно заменить на любо другой, поддерживающий операции, которые допустимы над double (например float). У класса CNNLayer есть наследник

```
class CNNLayer_store : public CNNLayer
```

В этом классе переопределены некоторые методы с целью сохранения промежуточных значений взвешенных сумм и значений аксонов НС для более оптимальной реализации метода обратного распространения ошибки. Класс НС определён как шаблон

```
template<class tLayer> class CNeuroNet
```

Это позволяет на этапе компиляции связывать НС с определённым классом слоя, избавляет от дублирования кода либо использования виртуальных функций, требующих дополнительных временных затрат при их вызове. То, что может быть решено до запуска программы не должно занимать время при работе программы. Основная функция CNeuroNet

```
void CNeuroNet::Propagate(
    const SYGDATATYPE *a_pInput, const unsigned int i_size,
    SYGDATATYPE *a_pOutput, const unsigned int o_size)
```

Эта функция определяет то, как работает НС: здесь можно организовать обратную связь или какие-либо операции над данными. Обучение НС происходит в классах-наследниках CNNTraining. Обучающие примеры представлены классом

```
class CTrainingInstances
  : protected std::vector<CTrainingInstance>
```

Каждый обучающий пример есть набор входов и требуемых выходов НС. Есть также средство для оценки константы Липшица.

```
SYGDATATYPE CTrainingInstances::CalcLipshic();
```

Также для удобства хранения информации о примерах и состоянии НС создан класс `CNNIfc` который обеспечивает сохранение и чтение классов `CNeuroNet` и `CTrainingInstances` в файлы, формат которых под ответственностью этого класса.

2.4 Уровни параллелизма при реализации НС на кластере ЭВМ

Главное достоинство нейросетевой архитектуры – параллельная обработка информации. Однако при программной реализации НС на ЭВМ Фон-неймановской архитектуры возможны различные варианты реализации параллельной обработки данных.

При реализации НС можно выделить следующие уровни параллелизма.

1. Суперскалярное вычисление операций, выполняемых нейронами – выполняется компилятором и процессором, автоматизирован.

2. Параллелизм на уровне нейронов, все нейроны одного слоя сети работают параллельно – наиболее общий случай, реализуемый в аппаратных НС. Эффективная реализация требует быстрой пересылки данных при переходе к последующему слою, по сравнению с временными затратами при вычислении данных на одном слое. В случае вычислительной сети ЭВМ – это возможно при большом числе нейронов по сравнению с числом процессоров.

3. На уровне слоёв – конвейерный параллелизм. Применим при обработки многослойной сетью последовательности символов некоторого текста, в том числе, если важен порядок символов. Требует на этапе проектирования сети балансировать загрузку процессоров.

4. На уровне символов текста – текст разбивается на участки, которые отдельно обрабатываются разными сетями. Применим для задач, в которых не важна последовательность обработки символов.

5. На уровне сетей – различно обученные НС применяются для различных задач.

На уровнях 4 и 5 масштабируемость будет зависить только от сбалансированности загрузки, решение о растределении задач принимается заранее. В таком режиме возможна реализация нейросетевых задач для метакомпьютинга.

Метакомпьютинг строится по архитектуре master-slave: есть master-программа, которая только раздаёт задания и следит за их выполнением, если задание не выполнено, оно даётся другому компьютеру. Slave-программы получают задания, прогоняют данные через свою НС, высылают серверу результат и ждут нового задания. Сервер просто даёт задания и проверяет из выполнение. На slave-программах – много экземпляров НС

(или разных НС). Slave-программы могут их обучать по команде master-программы на предоставленных ею же примерах, могут выдавать результаты прогона данных через сеть master-программе. Например, master-программа посылает каждой slave-программе нераспознанное слово из текста, клиент распознаёт его и возвращает.

Пункт 3 является частным случаем пункта 2, как и распределение нейронов каждого слоя на вычислительную сеть и будет рассматриваться в подразделе 2.5.

Параллельная НС может обучаться последовательно, так же, как и последовательно--работающая, может обучаться параллельно. Для эффективной реализации сети она должна одновременно отвечать всем этим уровням параллелизма, которые позволяет задача и архитектура вычислительной системы. При обучении сети можно выделить такие уровни параллелизма (они могут быть не связаны с уровнями, на которых сеть работает):

1. Суперскалярное вычисление операций, выполняемых нейронами – выполняется компилятором и процессором, автоматизирован.

2. Параллелизм на уровне нейронов, все нейроны одного слоя сети обучаются параллельно.

3. На уровне слоёв. Каждый слой обучается независимо от других, только в конце итерации происходит обмен данными. Применение данного подхода было бы эффективно при возможности минимизировать обмен данными. Для этого хорошо бы иметь возможность заранее предсказать, какие промежуточные этапы пройдут данные между входом и выходом сети (т.е. предсказать выходы нейронов на каждом слое). Тогда обучение сведётся к обучению однослойной НС (на каждом слое), не считая предсказания выходов промежуточных слоёв для набора входных примеров. Решение этой задачи также полезно при выборе начального приближения. Кроме того, требуется на этапе проектирования сети балансировать загрузку процессоров.

4. На уровне символов текста – несколько маленьких сетей обучаются на работу с частью символов, результатирующая сеть строится аддитивно из этих сетей, возможно с предварительной классификацией, к какой части алфавита относится тот или иной символ, либо с последующим выбором подсети по выходным данным. Для применения метода необходимо заранее разделить входной алфавит.

5. На уровне сетей – множество различных НС обучаются на различных примерах, возможно, разными методами, затем используются, либо на их основе стоится итоговая сеть.

Пункт 2 невозможно осуществить при изоляции нейрона – требуются передачи данных. Этот уровень может быть реализован в специализированных архитектурах.

Уровни 4 и 5 затронуты в подразделах 3.2 и 3.3 и являются объектом дальнейших исследований.

В нетривиальном случае функция ошибки сети в пространстве весов имеет невообразимое количество локальных минимумов, которое тем больше, чем больше нейронов. Успешность обучения сети зависит от

1. выбора обучающих пар,

2. выбора архитектуры,

3. начального приближения.

При обучении сложной задаче может применятся следующий подход: делается набор пробных обучений и выбирается лучшая сеть. Для этого сети с разными начальными весами обучаются в течение нескольких десятков итераций, потом выбирается лучшая сеть и она уже “дообучается” до требуемой точности. [12]

Особый интерес представляет пункт 3. Согласно формуле (36), обучение однослойной сети есть нахождение элементов матрицы N по обучающим примерам.

Пусть линейная сеть, тренируемая на l обучающих парах, имеет n входов и m выходов. Матричное уравнение преобразования сигналов для такой сети имеет вид:

$$S_{m,l} = W_{m,n}P_{n,l} + b_{m,1}e_{1,m} \tag{37}$$

где $P_{n,l}$, $T_{m,l}$ – обучающие пары, $W_{m,n}$ – матрица весов, $b_{m,1}$ – вектор смещений нейронного слоя, $e_{1,m}$ – вектор единиц. Если ввести замены

$$X_{n+1,l} = \begin{bmatrix} P_{n,l} \\ 1 \end{bmatrix}, \tag{38}$$

$$N_{m,n+1} = \begin{bmatrix} W_{m,n} & b_{m,1} \end{bmatrix}, \tag{39}$$

то выражение (37) примет вид

$$S_{m,l} = N_{m,n+1}X_{n+1,l}. \tag{40}$$

Тогда матрица $N_{m,n+1}$ может быть выражена

$$N_{m,n+1} = S_{m,l}X^{+}_{l,n+1}, \tag{41}$$

где $X^{+}_{l,n+1}$ – псевдообратная матрица по отношению к матрице (38). Теперь искомые матрицы $W_{m,n}$ и $b_{m,1}$ запишутся в виде

$$W_{m,n} = \begin{bmatrix} w^{1,1} & \cdots & w^{1,n} \\ \vdots & & \vdots \\ w^{m,1} & \cdots & w^{m,n} \end{bmatrix}, \quad b_{m,1} = \begin{bmatrix} w^{1,n+1} \\ \vdots \\ w^{m,n+1} \end{bmatrix}, \tag{42}$$

где $w^{i,j}$ – элементы матрицы $N_{m,n+1}$

Приведём пример решения таким способом задачи распознавания цифр 0, 1, 4, 7 на рецепторном поле размена 3×3. Данная задача описана в [11].

$$\begin{Vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{Vmatrix} = N \begin{Vmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{Vmatrix} \tag{43}$$

Выберем линейно-независимые строки X, чтобы посчитать обратную матрицу:

$$\tilde{X} = \begin{matrix} \text{№6} \\ \text{№5} \\ \text{№1} \\ \text{№9} \end{matrix} \begin{Vmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{Vmatrix} \quad \tilde{N} = \begin{Vmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{Vmatrix} \tag{44}$$

Полученная матрица $\tilde{N}$ является матрицей слоя НС – ответом задачи. Для соответствия размерности умножаемых матриц можно заполнить недостающие столбцы нулями.

$$N = \begin{Vmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{Vmatrix} \tag{45}$$

Умножение матрицы рецепторного поля на N приведёт к правильному результату. Такая реализация не всегда возможна, так как однослойная НС может решать только линейно-разделимые задачи. Нетривиальной задачей является предсказание значений выходов нейронов скрытых слоёв для независимого обучения разных слоёв НС.

2.5 Моделирование работы искусственной НС

Рассмотрим работу сети на уровнях 2 и 3, описанных в подразделе 2.4. Распределение каждого нейрона на узел вычислительной сети (ВС) будет рассматриваться как частный случай распределения одного слоя НС по процессорам ВС.

Согласно уравнению (36) слой НС, распределённый по процессам, будет производить параллельное умножение матрицы на вектор и параллельно вычислять активационную функцию от каждого значения полученного вектора.

Для каждого элемента результирующего вектора $\vec{c}$, получаемого от перемножения матрицы A на вектор $\vec{b}$, нужно выполнить m умножений, и $m-1$ сложений. Имеется m элементов вектора $\vec{c}$, поэтому общее количество операций с плавающей точкой будет

$$m(m+(m-1)) = 2m^2 - m. \tag{46}$$

Для простоты предполагаем, что на сложение и умножение затрачивается одинаковое количество времени, равное T_{comp}, и полное время вычисления в основном зависит от операций с плавающей точкой. Грубая оценка времени вычислений равна

$$(2m^2 - m)T_{comp}. \tag{47}$$

Оценим затраты на коммуникацию. Не учитыая затраты на рассылку $\vec{b}$, количество чисел, которые должны быть переданы, равно $m+1$ (m – чтобы послать строку матрицы A, а 1 – чтобы послать ответ назад) для каждого из m элементов $\vec{c}$, получим

$$m(m+1) = m^2 + m. \tag{48}$$

Если предположить, что время, требуемое на передачу числа с плавающей запятой, равняется T_{comm}, то полное время связи примерно равно $(m^2+m)T_{comm}$. Поэтому отношение времени коммуникаций к времени вычислений согласно (9) равно

$$c = c_a \cdot c_t = \left(\frac{m^2+m}{2m^2-m}\right) \times \left(\frac{T_{comm}}{T_{comp}}\right). \tag{49}$$

Предположим, что:

1. для больших m (именно для таких задач нужны параллельные ЭВМ) можно пренебречь m по сравнению с m^2;
2. в качестве коммуникационной сети используется сеть Fast Ethernet с пропускной способностью $V = 10$ Мбайт/с;
3. объединенное время T_{comp} требует 20 нс при частоте процессора 500 МГц;
4. длина числа с плавающей запятой может составлять 10 байтов.

Тогда, согласно выражению (49), получаем

$$c = c_a \times c_t = \frac{T_{comm}}{2T_{comp}} = \frac{10V}{2T_{comp}} = \frac{10/10^7}{2 \cdot 20 \cdot 10^{-9}} = 25. \tag{50}$$

Следовательно, даже при идеальном распараллеливании:

$$R_c = \frac{1}{1/n + 25} \tag{51}$$

Матрично-векторное умножение всегда будет выполняться в многопроцессорной системе с замедлением по отношению к однопроцессорному варианту.

Уровень 3 может быть представлен как независимое умножение матрицы на вектор и передача данных (соответствующая продвижению данных по сети). Ускорение может быть достигнуто в случае, если передача данных не занимает времени больше, чем операция умножения матрицы на вектор. При этом требуется обеспечить балансировку загрузки ВС. Для обеспечения масштабирования необходимо иметь количество слоёв НС, не меньшее числу узлов ВС, при этом количество нейронов на каджом узле сети должно быть примерно одинаковым. При соблюдении всех этих условий сравним время умножения матрицы на вектор с передачей вектора по сети, оценим параметры сети передачи данных.

При распределении НС по узлам послойно в каждом слое вычисление произведения матрицы на вектор и последующая передача результата происходит синхронно во всех слоях. Для определённости считаем, что НС имеет 2 слоя, которые распределены на разные процессоры ВС. Время получения результата на выходе НС:

$$T_{nap} = a \cdot (t_{mult} \cdot m + t_{add} \cdot (m-1)) + (\frac{m}{R} + L) \tag{52}$$

где a – число нейронов в каждом слое, t_{mult} – время умножения, t_{add} – время сложения, m – размерность входного вектора, R – пропускная способность, L – латентность,

Время получения результата на выходе непараллельной сети:

$$T = a \cdot (t_{mult} \cdot m + t_{add} \cdot (m-1)) \cdot n \tag{53}$$

здесь n – число слоёв НС, и число процессоров ВС. Итоговый результат изображён на рисунке 8. Пропускная способность 100 Мб/с, тактовая частота процессоров взята 1000 МГц. Для таких параметров способ даёт ускорение, если число нейронов в каждом слое превышает 15, даёт ускорение 50% при числе особей в слое более 40. Такой подход не масштабируем, так как количество слёв НС невелико, и применение большего числа процессоров бессмысленно. Следовательно нужны другие подходы к декомпозиции НС.

Рисунок 8: Ускорение от применения конвейерного прохождения слоёв НС

3 Реализация параллельного генетического алгоритма

Рассмотрены генетические алгоритмы (ГА), описана созданная программа, реализующая параллельный ГА в стандарте MPI. Исследованы некоторые свойства ГА, влияющие на масштабируемость. При этом масштабируемость рассматриваться как сохранение эффективности системы при увеличении размера задачи. Полученные результаты использованы при обучении НС.

Хотя модель эволюционного развития, применяемая в ГА, сильно упрощена по сравнению со своим природным аналогом, тем не менее, ГА являются достаточно мощным средством и могут с успехом применяться для широкого класса прикладных задач [13].

Популяция – совокупность особей одного вида, связанных родством, гибридизацией и одним ареалом обитания. Аналог популяции P^t – совокупность особей $(a_1^t, \ldots, a_\nu^t)$. Число ν, характеризующее число особей a_k^t, которые образуют популяцию, будем называть численностью популяции. В общем случае задачи оптимизации популяция $P^t = (a_1^t, \ldots, a_\nu^t)$ соответствует совокупности отдельных допустимых решений $x_n\vec{u}^k \in D, k = \overline{1,\nu}$. Целью эволюционного прогресса популяции является повышение степени приспособленности особей.

В качестве аналога прогресса будем понимать увеличение степени приспособленности популяции к внешней среде в ходе эволюции, например, в смысле обеспечения наибольшего значения средней степени приспособленности по популяции P^t:

$$\mu_{cp}(t) = \frac{1}{\nu}\sum_{i=1}^{\nu}\mu(a_i^t) \tag{54}$$

Кроссинговером в биологии называется процесс случайного обмена гомологичными участками гомологичных хромосом родительских особей при формировании генотипа потомка. В качестве аналога кроссинговера примем обмен частей хромосом двух родительских особей по некоторому закону с элементом случайности [13].

Мутацией называется случайное (и как правило редкое) искажение генотипа особи.

3.1 Декомпозиция генетического алгоритма

Имеется много способов реализации идеи биологической эволюции в рамках ГА. Традиционным считается ГА, представленный на схеме:

```
НАЧАЛО
  Генерация начальной популяции
  ПОКА НЕ достигнуто решение ВЫПОЛНЯТЬ
  НАЧАЛО
    Кроссинговер /* обмен генами */
    Мутация
    Селекция
  КОНЕЦ
КОНЕЦ
```

Создание начальной популяции обычно производится простым заполнением генотипов всех особей популяции случайными данными. Численность популяции является одним из параметров алгоритма. При низкой численности популяции могут перестать работать законы эволюции и ГА не будет способен эффективно работать.

Можно выделить следующие параметры ГА:

1. Численность популяции. Обычно остается постоянной при работе ГА. Является главным ресурсом, за который вынуждены бороться особи.

2. Длина бинарных кодировок (длина генотипов). В первую очередь определяется условиями задачи оптимизации.

3. Количество решений, генерируемых на каждой итерации. Для традиционного ГА совпадает с численностью популяции.

4. Вероятность мутации. Обеспечивает случайные отклонения для расширения области поиска.

5. Правило выбора пары родительских особей. Реализует принцип отбора.

6. Тип используемого оператора кроссинговера. Реализует принцип комбинативной изменчивости.

7. Тип используемого оператора мутации. Реализует принцип мутационной изменчивости.

То, насколько удачным окажется применение ГА при решении той или иной задачи, будет определяться удачным выбором значений параметров. Строгих правил подбора параметров, универсальных для всех задач, нет и быть не может [13].

Наиболее важным при настройке ГА и создании их модификаций является соблюдение баланса между достаточной шириной области поиска (т.е. вероятностью найти очень хорошее решение) и быстрой сходимостью (т.е. приемлемым временем вычислений).

Существует несколько подходов к параллельной реализации ГА. Наиболее простой – параллельное вычисление только функции жизнеспособности, остальные операции проводятся на одном узле вычислительной сети. Такой подход, кроме простоты реализации, обладает большой эффективностью, если вычисление функции жизнеспособности требует больших вычислительных затрат [14]. Так как остальные части алгоритма в таком случае, такие как кроссинговер, мутация и формирование нового поколения, требуют гораздо меньших вычислительных затрат, чем процедура отбора, то такой ГА работает быстрее исходного практически во столько раз, сколько имеется параллельных процессов. Однако, такой подход не применим в случае, если вычисление жизнеспособности всех особей в популяции занимает время, сравнимое со временем, необходимым для пересылки генотипов особей. Такой случай может возникнуть, если функция жизнеспособности проста, или размер популяции небольшой.

Наиболее общий подход к распараллеливанию ГА заключается в создании нескольких популяций, каждая из которых размещается на отдельном узле вычислительной сети (так называемая “островная модель” (Island Model) [15]). При этом, чтобы не получить эффекта, что каждая популяция достигает разными путями одно и то же решение, но при этом занимая больше машинных ресурсов, необходимо ввести ещё одну операцию над популяцией – миграцию. Миграция может производиться с учётом жизнеспособности переселяемой особи, а также носить случайный характер. В случае введения миграции к параметрам параллельных ГА добавляются ещё два:

1. количество переселяемых особей,

2. метод отбора особей для переселения.

Определение оптимальных значений этих параметров определяет эффективность параллельного ГА, но сами параметры зависят от решаемой задачи.

Ещё одним методом создания параллельных ГА является так называемые клеточные ГА (cellular genetic algorithms) [15]. Основная идея этой модификации заключается в том, что особи располагаются на некоторой пространственной сетке, их положение определяется их фенотипом, размножение происходит по определённым правилам, зависящим от относительного расположения особей между собой. При такой реализации приходится решать задачу балансировки загрузки за счёт динамического перераспределения областей

пространства по процессам исходя из количества особей, находящихся в них. Положение особи в этом пространстве, как уже было отмечено, определяется фенотипом особи, например это могут быть вероятностные характеристики генома особи: среднее значение генов и их дисперсия, и.т.п.

Разработку параллельного ГА в стандарте MPI можно свести к следующей последовательности действий:

1. Написание обычного последовательного ГА.

2. При инициализации программы добавить обязательный для MPI вызов `MPI_Init(&argc, &argv)`, перед завершением программы добавить вызов `MPI_Finalize()`. Теперь программа соответствует стандарту MPI, и её можно запускать параллельно, но никакого выигрыша это не даст, так как процессы не используют информацию друг о друге, а просто решают одну и ту же задачу разными путями.

3. Добавление операции миграции в программный код. Итак, одна итерация параллельного ГА:

   (a) Скрещивание (кроссинговер)

   (b) Мутация

   (c) Селекция

   (d) Миграция

Миграция может выполняться с помощью функции MPI `MPI_Sendrecv_replace`, которая выполняет замену содержимого буферов процессов [3].

Был опробован следующий метод: выбираются несколько особей, последовательно хранящихся в стандартном контейнере C++ `std::vector` и отсылаются в следующий по номеру процесс, и так далее. Как показали эксперименты, при малом, по сравнению с количеством особей в популяции, количестве мигрируемых особей эффективность параллельного ГА падает во столько раз, какое количество процессов используется. Это означает, что миграция в таких количествах никак не отражается на переносе генетического материала. Однако, значительное повышение производительности наблюдается, если миграции подвергаются наиболее жизнеспособные особи. Подобный эффект достигается, если количество мигрируемых особей примерно равно половине размера части популяции, находящейся в одном процессе (так сказать, на одном острове).

Если судить об эффективности параллельного ГА через коэффициент использования, то результаты будут схожими с решением СЛАУ (раздел 1.2). Однако параллельная реализация влияет на параметры ГА 37, в частности на размер популяции. При значительном увеличении размера популяции число итераций не изменяется или изменяется незначительно. Это объясняется тем, что в популяции уже набралось достаточное для эволюции количество особей. “Лишние” особи, которые не влияют на сходимость ГА, занимают ресурсы вычислительной системы, снижая эффективность.

3.2 Применение ГА для обучения НС

Для исследования и создания параллельных реализаций методов обучения НС были реализованы последовательные варианты методов обратного распространения ошибки и стохастического обучения.

Исследуем обучение методом обратного распространения ошибки. Первая итерация на графике 9 – обучение первому примеру. Примеры подаются по очереди. Первая итерация уменьшает погрешность НС для первого примера, пр этом уменьшается погрешность второго и увеличивается у остальных. Вторая итерация проделывается со следующим

примером, при этом погрешность увеличивается для всех примеров. С ходом итераций погрешность сети для всех примеров растёт, кроме того, с которого начиналось обучение. Такая же закономерность наблюдалась для всех примеров – НС обучалась только тому примеру, который подавался первым, для остальных погрешность увеличивалась.

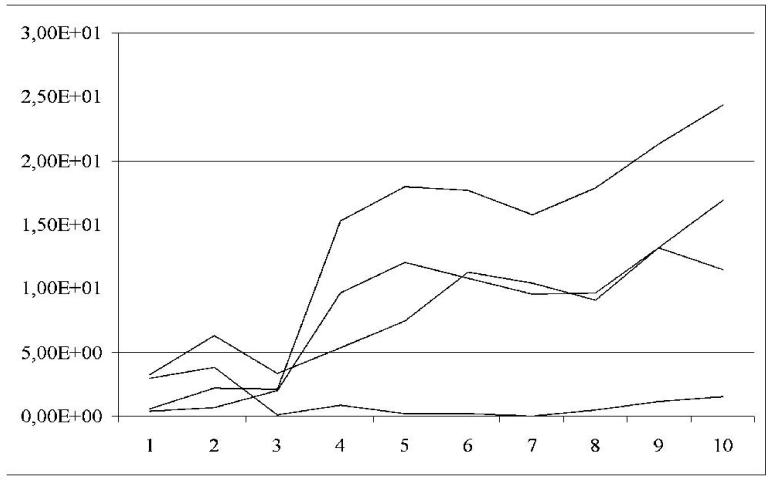

Рисунок 9: Ошибки НС при обучении методом обратного распространения ошибкидля разных обучающих примеров при большой скорости сходимости

Стоило только изменить скорость обучения, как задача сошлась, однако закономерность осталась (см. рисунок 10). Изменение весов, связанное с обучением одному примеру может увеличивать погрешности для других примеров, при этом такой параметр, как скорость обучения определяет сходимость метода. Это делает метод обратного распространения ошибки менее эффективным при обучении сети большому числу примеров.

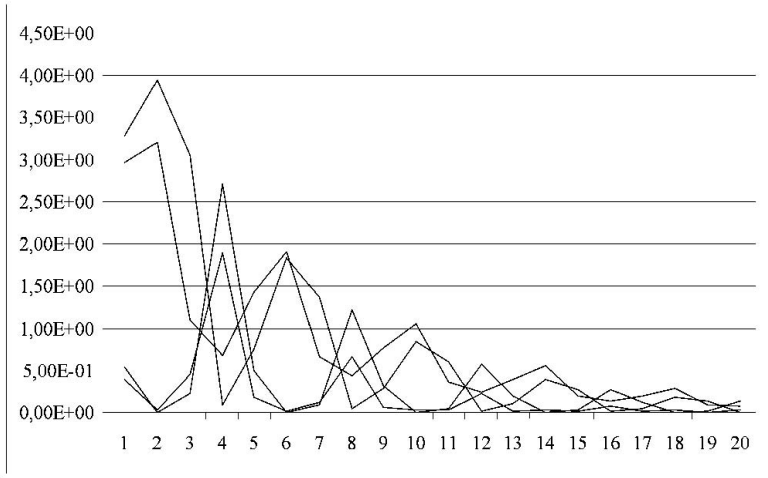

Рисунок 10: Ошибки НС при обучении методом обратного распространения ошибкидля разных обучающих примеров при небольшой скорости сходимости

Часто в таких случаях выгодно использования метода стохастического обучения [11]. Ошибка НС при стохастическом обучении паказана на рисунке 11. Ошибка для различных примеров изменяется немонотонно, однако сам метод предполагает монотонное изменение интегральной ошибки. Метод неградиентный и обладает меньшей сходимостью по сравнению с методом обратного распространения ошибки (см. рисуок 12). В качестве интегрального критерия выбрана сумма квадратов отклонения выходов НС.

Графики на рисунке 12 показывают суммарную ошибку НС для методов обратного распространения и стохастического обучения. Для приведённого примера (обучение на

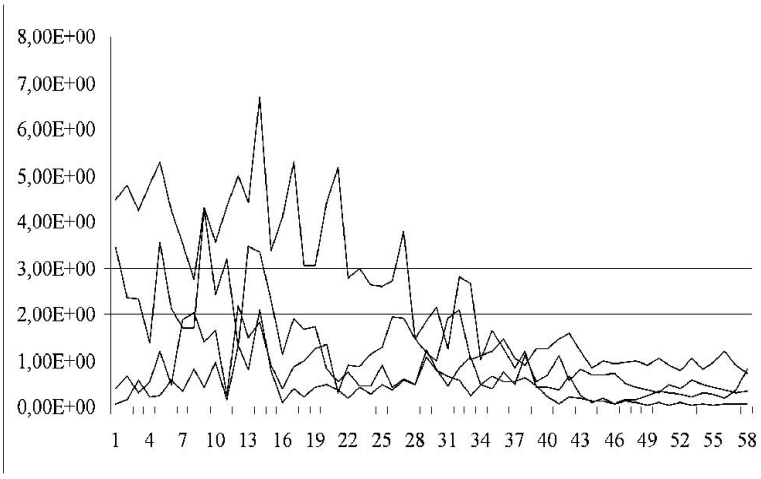

Рисунок 11: Ошибки НС при стохастическом обучении для разных обучающих примеров при небольшой скорости сходимости

четырёх обучающих выборках в указанном диапазоне ошибок) сходимость метода обратного прохождения ошибки больше.

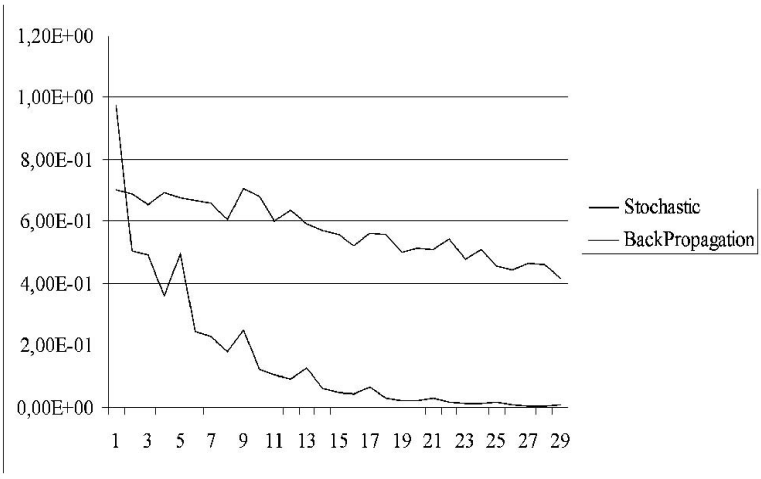

Рисунок 12: Сравнение суммарной ошибки методов обратного распространения и стохастического обучения

Генетический алгоритм может быть представлен как набор из других методов обучения. Например, генерация начальной популяции может представлять собой ни что иное, как метод случайной стрельбы, мутация может представлять собой одну или несколько итераций лыбого другого метода, что соответствует идеологии “hill climbing”. Селекция есть способ уменьшения вычислительной сложности генетического алгоритма. Построим генетический алгоритм с мутацией в виде одной итерации метода обратного распространения ошибки, применённого к наихудшему примеру. На рисунке 13 показаны графики погрешности при применении метода обратного распространения ошибки и с генетическим алгоритмом с мутацией в виде одной итерации этого метода. *GA-BackPr*-4 обозначен вариант генетического алгоритма с популяцией из четырёх особей. Другой вид алгоритма – с 50 особями.

Эффективность генетических алгоритмов тем ниже, чем больше размер популяции, так как каждая особь должна быть обработана. У генетического алгоритма зависмость числа итераций о числа особей нелинейна. Если количество особей достаточно для эволюционного развития, то дальнейшее увеличение размера популяции не позволит добиться большей сходимости, а лишь увеличит вычислительную сложность задачи.

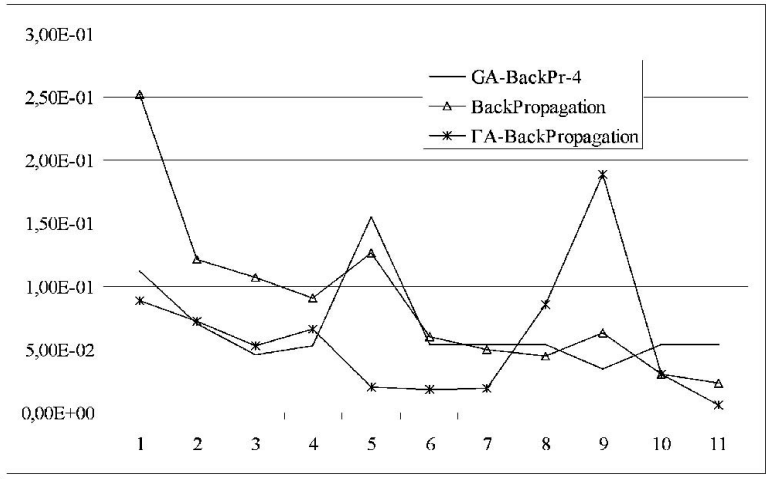

Рисунок 13: Сравнение генетического алгоритма с методом обратного распространения ошибки

А что представляет собой кроссинговер в такой интерпретации? На рисунке 14 показан график изменения погрешности от числа итераций при применении генетического алгоритма с кроссинговером и без него.

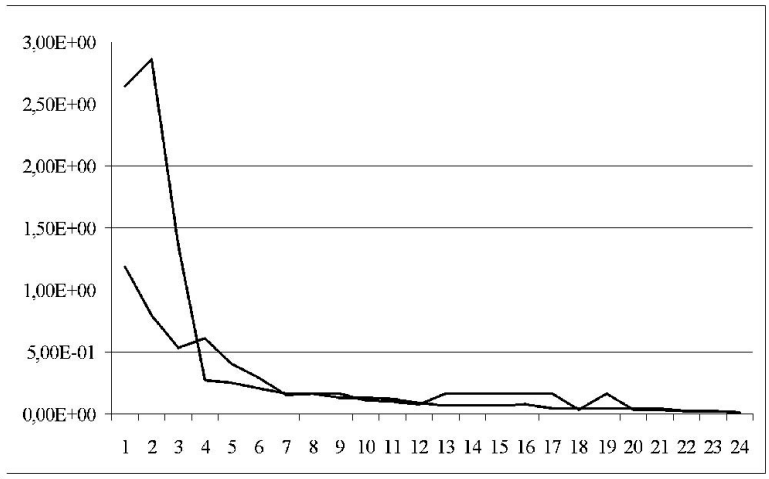

Рисунок 14: Погрешность генетического алгоритма с кроссинговером и без него

Было проведено несколько таких опытов с разными начальными приближениями. В качестве мутации использовалась одна итерация метода обратного растпространения ошибки. Наличие кроссинговера во всех случаях показало более быструю сходимость на начальной стадии обучения и меньшую стабильность на завешющей стадии. Это может означать, что наличие кроссинговера определяет свойство генетических алгоритмов быть применимыми для поиска “достаточно хорошего” решения задачи “достаточно быстро”.

На рисунке 15 показаны графики зависимости погрешности генетического алгоритма от выбора особей для кроссинговера. Выбор особи, наиболее близкой по среднему значению нейронов уменьшает вероятность появления особей в зоне между экстремумами, улучшая сходимость алгоритма. Также проводился эксперимент, где вероятность кроссинговера больше длянаиболее различных особей, но результат оказался отрицательным, это привело к увеличению числа итераций.

Рисунок 16 позволяет продемонстрировать подход выбора примера обучения по максимальой ошибке. Пример выбирается для выполняемого во время мутации метода обратного прохождения ошибки. Такой подход более эффективен, так как исключена вероятность обучеия НС примерам, для которых ошибка в пределах заданной погрешности. Также

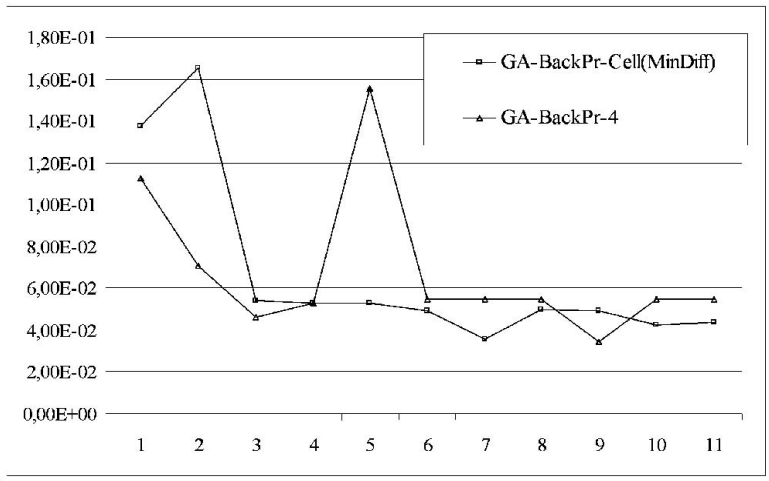

Рисунок 15: Сравнение погрешности генетического алгоритма со случайным выбором особей для кроссинговера и с выбором наиболее близкой по среднему значению весов нейронов

на графике показана зависимость для метода обратного прохождения ошибки. Когда достигается некоторая погрешность, генетическй алгоритм не способен эффективно добится лучшего результата. Нужно учитывать, что в генетическом алгоритме 4 особи, т.е. при равных данных на графике генетический алгорит работал в 4 раза менее эффективно. Эти графики показывают, что при эффективной реализации клеточного генетического алгоритма распределение особей должно происходить с учётом средних значений весов нейонов. При небольшой погрешности (т.е. вблизи глобального экстремума) генетический алгоритм неэффективен по сравнению с классическми последовательныи методами.

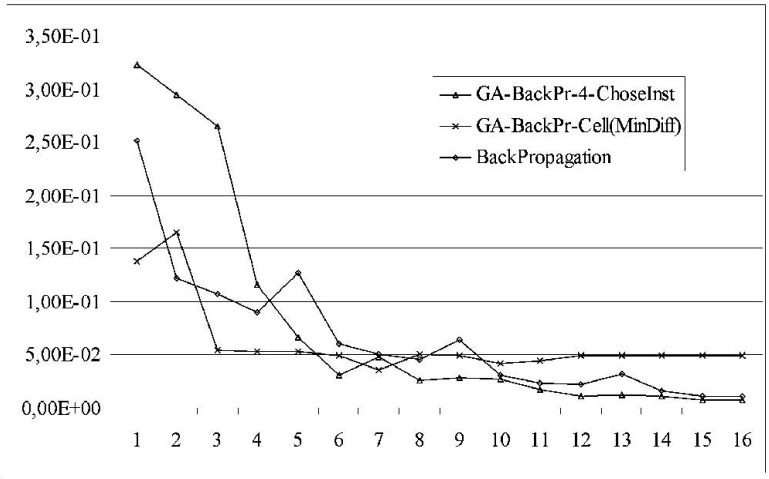

Рисунок 16: Сравнение погрешности генетического алгоритма с последовательным выбором примера обучения и по максимальной погрешности

3.3 Параллельное обучение сети с разделением обучающей выборки

Рассмотрим пункт 5 на стр. 33 в уровнях параллелизма при построении нейроэмулятора. Обучим НС двум из четырёх примеров, затем без изменения структуры сети будем обучать её ещё двум. Примеры независимы, поэтому при обучении одному из примеров не поисходит значительного уменьшения погрешнсти для других. Результат показан на рисунке 17 – при уменьшении погрешноси для новых примеров увеличиваеся для тех, которым уже обучали. Это т.н. “эффект забывания” – сеть теряет информацию о примерах,

которым она обучалась ранее. Для обучения прменялся генетический алгоритм, поэтому мы наблюдаем периодическое изменение погрешности – новые особи имеют меньшую жизнеспособность, поэтому удаляются селекцией. На на рисунке 18 показано дообучение двум новым примерам, но селекция ведётся по функции жизнеспособности, зависящей от четырёх. Такой вариант дообучения обеспечил сходимость ГА.

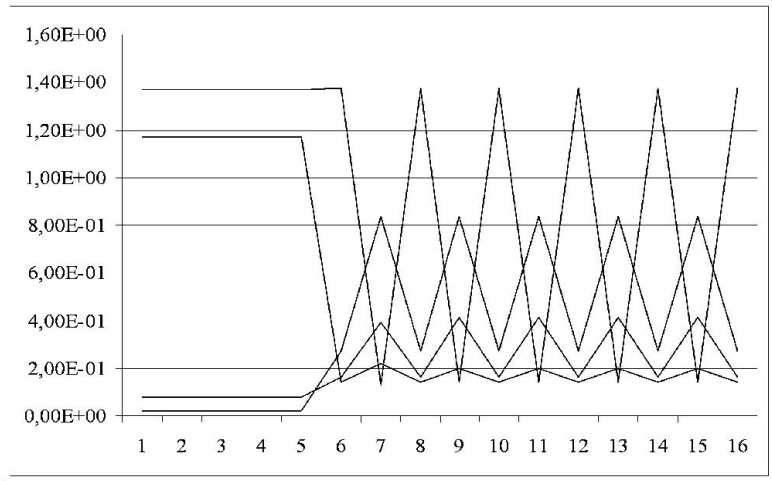

Рисунок 17: Дообучение на двух примерах с селекцией по двум

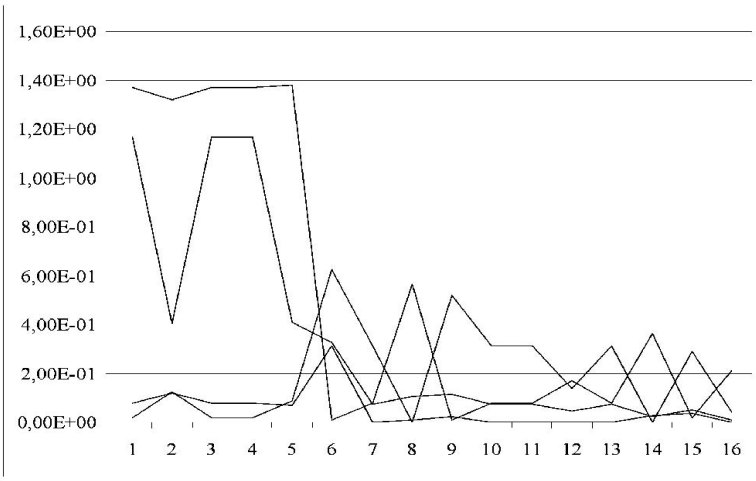

Рисунок 18: Дообучение на двух примерах с селекцией по четырём

ЗАКЛЮЧЕНИЕ

Создан нейроэмулятор, на основе которого исследовались методы параллельного обучения нейроных сетей. Обозначены уровни декомпозиции алгоритмов работы и обучения нейроэмулятора на кластерах ЭВМ.

Созданы модели работы параллельных программ на кластерах ЭВМ, с помощью которых исследовались подходы к реализации параллельного метода обучения НС. С помощью этих моделей можно изучать эффективность кластеров для разных программ в зависимости от их конфигурации, что также может быть полезно при создании новых кластеров. Промоделированы зависимости времени работы программы решения СЛАУ методом простой итерации. Промоделированы случаи модификации вычислительной сети путём замены концентратора на коммутатор. Проведено сравнение этих зависимостей.

Моделирование показало неэффективность некоторых подходов к реализации нейроэмулятора, связанную с большой латентностью и небольшой скоростью передачи данных между узлами вычислительной сети. Именно эта особенность существующих методов обучения НС не позволяет напрямую осуществить их эффективную реализацию на кластерах ЭВМ.

Реализован параллельный генетический алгоритм в стандарте MPI для обучения НС. Исследованы факторы, понижающие масштабируемость ГА при увеличении размеров популяции. Исследована примеимость генетического алгоритма для параллельного обучения НС, и сделан вывод об эффективности его применения на начальных стадиях обучения. ГА позволяют организовать параллельное обучение НС, их применение эффективно на начальной стадии обучения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] **А.Е.Верхотуров, Г.И.Шпаковский.** Применение многопроцессорных кластеров для обучения нейронных сетей на основе генетического алгоритма [Текст] // Материалы II Международной конференции "Информационные системы и технологии"(IST'2004). Минск, 2004. Часть 2

[2] **Г.И.Шпаковский, А.Е.Верхотуров, Н.В.Серикова.** Параллельные вычисления на локальных сетях в стандарте MPI [Текст] // Радиофизика и электроника: Сб.науч.тр. Вып. 6 / Редкол.: Р15 С.Г. Мулярчик (отв. ред.) и др.– Мн.: БГУ, 2003. – 160 с.

[3] MPI [Текст]: Стандарт интерфейса передачи сообщений. Перевод под ред. Г.И. Шпаковского. Интернет адрес http://www.cluster.bsu.by

[4] Интернет адрес: http://www.parallel.ru

[5] **Мулярчик С. Г.** Численные методы [Текст]: Конспект лекций. – Мн., БГУ, 2001. – 127 с.

[6] **W.Gropp, E.Lusk, and A.Skjellum.** Using MPI: Portable Parallel Programming with the Message-Passing Interface. Second edition, published in 1999 by MIT Press, 371 pages

[7] **Г.И.Шпаковский, А.Е.Верхотуров, Д.А.Стрикелев.** Организации эффективных вычислений на локальных сетях в стандарте MPI [Текст] // Материалы I Международной конференции "Информационные системы и технологии"(IST'2002). Минск, 2002. Часть 2

[8] **В.Г. Олифер, Н.А. Олифер.** Компьютерные сети. Принципы, технологии, протоколы [Текст]: Учебник для вузов. 2-е изд. — СПб.: Питер, 2003. — 864 с.: ил.

[9] **В. Шахнов, А. Власов, А. Кузнецов.** Нейрокомпьютеры – архитектура и реализация. [Текст] Интенет адрес: http://www.chipinfo.ru/literature/chipnews/200005/34.html

[10] **Bernard Widrow and Michael A. Lehr.** 30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation. // Proceedings of the IEEE, Vol. 79, No. 9, September 1990

[11] **В.М. Лутковский.** Нейронные сети [Текст]: Конспект лекций – Мн.:БГУ, 2003.–99 с.

[12] **Petr V. Nazarov, Vladimir V. Apanasovich, Vladimir M. Lutkovski, Mikalai M. Yatskou, Rob B. M. Koehorst, and Marcus A. Hemminga.** Artificial Neural Network Modification of Simulation-Based Fitting: Application to a Protein-Lipid System.

[13] **С. Исаев.** Популярно о генетических алгоритмах [Текст]. Интернет адрес: http://softlab.od.ua/algo/neuro/ga-pop/index.htm.

[14] **Каганов Е.Л., Казанский Н.Л., Павельев В.С.** Распараллеливание генетического алгоритма при расчете многопучкового ИМП-модана [Текст] // Труды Всероссийской научной конференции «Высокопроизводительные вычисления и их приложения», Черноголовка, 30 октября – 2 ноября 2000 г. – М.: Изд-во МГУ, 2000, с.112-114.

[15] **Darrel Whitley.** A Genetic Algorithm Tutorial. Technical Report CS-93-103, March 10, 1993. Colorado State University. Интерет адрес: http://carol.wins.uva.nl/ krose/col/cni/papers/tutorial.ps.gz

```
//gsettings.h
#ifndef _LX_GSETTINGS_H_
#define _LX_GSETTINGS_H_
#define SYGDATATYPE double
#endif //_LX_GSETTINGS_H_

//activafunc.h
#ifndef _LX_ACTIVAFUNC_H_INCLUDED_
#define _LX_ACTIVAFUNC_H_INCLUDED_
#include "gsettings.h"
#define ACTIFUNC(f_name) \
SYGDATATYPE f_name(const SYGDATATYPE& a_param)

typedef ACTIFUNC(t_actifunc);

t_actifunc f_sigmoid; //=1/(1+exp(lambda*x)
t_actifunc f_sin;     //=x*sin(x)
t_actifunc f_lin;     //=x
t_actifunc f_htan;    //=1-2/(1+exp(2*(omea-theta)))

// поизводные
t_actifunc dfds_sigmoid;
t_actifunc dfds_sin;
t_actifunc dfds_lin;
t_actifunc dfds_htan;

// А сама функция есть агрегация себя со своей производной, обратной
struct CActi
{
  t_actifunc *f;
   t_actifunc *dfds;
   CActi(t_actifunc f_=f_lin, t_actifunc dfds_=dfds_lin)
   {f=f_; dfds=dfds_;};
};

const CActi acti_sigmoid(f_sigmoid, dfds_sigmoid);
const CActi acti_sin(f_sin, dfds_sin);
const CActi acti_lin(f_lin, dfds_lin);
const CActi acti_htan(f_htan, dfds_htan);

#endif _LX_ACTIVAFUNC_H_INCLUDED_

//activafunc.cpp
#include "activafunc.h"
#include <math.h>

ACTIFUNC(f_sigmoid)
{
const double lambda=0.6;//magic number
   double e = exp(lambda*(double)a_param);
   SYGDATATYPE a = SYGDATATYPE( 1.0f/(1.0f+e));
   return 2*a - 0.5f;
}
ACTIFUNC(f_sin)
{
   return static_cast<SYGDATATYPE>(a_param*sin(a_param));
}
ACTIFUNC(f_lin)
{
   return a_param;
}
ACTIFUNC(f_htan)
{
   return 1-2/(1+exp(2*(a_param)));
}
ACTIFUNC(dfds_sigmoid)
{
const double lambda=0.6;//magic number
   double e = exp(lambda*(double)a_param);
    double _1_e = e + 1.0;
   if (e == _1_e) // округилось?
```

```
      return 0;
   SYGDATATYPE a = -2 * lambda * e / (_1_e * _1_e);
   return a;
}
ACTIFUNC(dfds_sin)
{
   return sin(a_param) + a_param * cos(a_param) ;
}
ACTIFUNC(dfds_lin)
{
   return 1.0;
}
ACTIFUNC(dfds_htan)
{
   return 1.0 - a_param*a_param;
}

//nnlayer.h
#ifndef _LX_NNLAYER_H_
#define _LX_NNLAYER_H_

#include "gsettings.h"
#include "activafunc.h"
#include <assert.h>
#include <memory.h>

class CNNLayer
{
   friend class CNNBackPropagation;
protected:
   /// Матрица
   SYGDATATYPE *m_pRates;
   /// Вектор входов
   SYGDATATYPE *m_pInput;
   /// Смещения входов
   SYGDATATYPE *m_pDispl;
   unsigned int m_iThisLayerSize;
   unsigned int m_iPrevLayerSize;
   CActi m_funcacti;
public:
   // конструктор по умолчанию - требует последующего вызова CNNLayer::Construct
   CNNLayer();
   CNNLayer(const unsigned int iThisLayerSize,
       const unsigned int iPrevLayerSize, CActi &a_funcacti);
   ~CNNLayer();
   void SetRates(const SYGDATATYPE *a_pRates, const unsigned int size);
   void SetInput(const SYGDATATYPE *a_pInput, const unsigned int size);
   void SetDispl(const SYGDATATYPE *a_pDispl, const unsigned int size);
   SYGDATATYPE *GetRates(unsigned int &size);
   SYGDATATYPE *GetDispl(unsigned int &size);
   SYGDATATYPE *GetInput(unsigned int &size);
   void GetAscons(SYGDATATYPE *a_pOutput, const unsigned int size);
   inline CActi GetActiFunc()const{return m_funcacti;};
   inline unsigned int GetThisLayerSize()const{return m_iThisLayerSize;};
   inline unsigned int GetPrevLayerSize()const{return m_iPrevLayerSize;};
protected:
   void Destruct();
public:
   void Construct(const unsigned int iThisLayerSize,
      const unsigned int iPrevLayerSize, CActi &a_funcacti);
};

class CNNLayer_store : public CNNLayer
{
protected:
   SYGDATATYPE *m_pSS; // аргументы акивационой функции

public:
   CNNLayer_store(){m_pSS=0;};
   void Construct(const unsigned int iThisLayerSize,
       const unsigned int iPrevLayerSize, CActi &a_funcacti)
   {
      CNNLayer::Construct(iThisLayerSize, iPrevLayerSize, a_funcacti);
```

```
      m_pInput = new SYGDATATYPE[m_iPrevLayerSize];
      m_pSS    = new SYGDATATYPE[m_iThisLayerSize];
   };
   ~CNNLayer_store()
   {
      if (m_pSS)
         delete[] m_pSS;
      if (m_pInput)
         delete[] m_pInput;
   };
   void SetInput(const SYGDATATYPE *a_pInput, const unsigned int size)
   {
      assert(size == m_iPrevLayerSize);
      memcpy(m_pInput, a_pInput, size*sizeof(SYGDATATYPE));
   };
   void GetAscons(SYGDATATYPE *a_pOutput, const unsigned int size)
   {
      assert(size == m_iThisLayerSize);
      assert(m_pInput != 0);

      unsigned int i,ii;
      for (i=ii=0; i<m_iThisLayerSize; ++i,ii+=m_iPrevLayerSize)
      {
         SYGDATATYPE tempVal=0;
         for (unsigned int j=0;j<m_iPrevLayerSize;++j)
            tempVal+=m_pInput[j]*m_pRates[j+ii];
         m_pSS[i] = tempVal; // вот это изменилось
         a_pOutput[i] = (*m_funcacti.f)(tempVal);
      }
   };

   SYGDATATYPE *GetS()const{return m_pSS;};
};
#endif //_LX_NNLAYER_H_

//nnlayer.cpp
#include "nnlayer.h"
#include <assert.h>
#include <memory.h>

CNNLayer::CNNLayer()
{
}
CNNLayer::CNNLayer(const unsigned int iThisLayerSize,
    const unsigned int iPrevLayerSize, CActi &a_funcacti)
{
   Construct(iThisLayerSize, iPrevLayerSize, a_funcacti);
}
void CNNLayer::Construct(const unsigned int iThisLayerSize,
    const unsigned int iPrevLayerSize, CActi &a_funcacti)
{
   m_iThisLayerSize = iThisLayerSize;
   m_iPrevLayerSize = iPrevLayerSize;

   m_pRates = new SYGDATATYPE[m_iPrevLayerSize*m_iThisLayerSize];
   m_pInput = 0;
   m_pDispl = new SYGDATATYPE[m_iPrevLayerSize];

   m_funcacti = a_funcacti;
}
CNNLayer::~CNNLayer()
{
   Destruct();
}
void CNNLayer::Destruct()
{
   delete[] m_pRates;
   delete[] m_pDispl;
}
void CNNLayer::SetRates(const SYGDATATYPE *a_pRates, const unsigned int size)
{
   assert(size == m_iThisLayerSize * m_iPrevLayerSize);
```

```
   memcpy(m_pRates, a_pRates, size*sizeof(SYGDATATYPE));
}
void CNNLayer::SetDispl(const SYGDATATYPE *a_pDispl, const unsigned int size)
{
   assert(size == m_iPrevLayerSize);
   memcpy(m_pDispl, a_pDispl, size*sizeof(SYGDATATYPE));
}
void CNNLayer::SetInput(const SYGDATATYPE *a_pInput, const unsigned int size)
{
   assert(size == m_iPrevLayerSize);
   m_pInput = const_cast<SYGDATATYPE*>(a_pInput);
}
SYGDATATYPE *CNNLayer::GetRates(unsigned int &size)
{
   size = m_iThisLayerSize * m_iPrevLayerSize;
   return m_pRates;
}
SYGDATATYPE *CNNLayer::GetInput(unsigned int &size)
{
   size = m_iPrevLayerSize;
   return m_pInput;
}
SYGDATATYPE *CNNLayer::GetDispl(unsigned int &size)
{
   size = m_iPrevLayerSize;
   return m_pDispl;
}
void CNNLayer::GetAscons(SYGDATATYPE *a_pOutput, const unsigned int size)
{
   assert(size == m_iThisLayerSize);
   assert(m_pInput != 0);

   unsigned int i,ii;
   for (i=ii=0; i<m_iThisLayerSize; ++i,ii+=m_iPrevLayerSize)
   {
      SYGDATATYPE tempVal=0;
      for (unsigned int j=0;j<m_iPrevLayerSize;++j)
         tempVal+=m_pInput[j]*m_pRates[j+ii];
      a_pOutput[i] = (*m_funcacti.f)(tempVal);
   }
}

//nn1x.h
#ifndef _LX_NN_H_INCLUDED____
#define _LX_NN_H_INCLUDED____

#include "nnlayer.h"

template<class tLayer>
class CNeuroNet
{
   friend class CNNIfc;
   friend class CNNTraining;
   friend class CNNBackPropagation;
   friend class CNeuroNet<CNNLayer_store>;
   friend class CNNStochastic;
   friend class CNNTGA;
   friend int main(int argc, char *argv[]);
protected:

   unsigned int m_numlayers;
   tLayer *m_layers;
   unsigned int m_MAXLAYERSIZE;
   SYGDATATYPE *m_pBuf; // буфер для прогона данных - под ответственность класса

public:
   CNeuroNet(){m_layers=0; m_numlayers=0;};

   virtual ~CNeuroNet(){};
   virtual void Propagate(const SYGDATATYPE *a_pInput, const unsigned int i_size,
      SYGDATATYPE *a_pOutput, const unsigned int o_size)=0;
   unsigned int GetINum()
   {
```

```
      unsigned int Result = 0;
      if (m_layers>0)
         m_layers[0].GetInput(Result);
      return Result;
   };
   unsigned int GetONum()
   {
      unsigned int Result = 0;
      if (m_layers>0)
         m_layers[m_numlayers-1].GetInput(Result);
      return Result;
   };
   SYGDATATYPE *GetBuf(unsigned int & Size)
   {
      Size = m_MAXLAYERSIZE;
      return m_pBuf;
   };
};

#endif //_LX_NN_H_INCLUDED____


//nnperceptron.h
#ifndef _LX_NN_H_INCLUDED___
#define _LX_NN_H_INCLUDED___

#include "gsettings.h"
#include "nn1x.h"

template<class tLayer>
class CNNPerceptron : public CNeuroNet<tLayer>
{
   friend class CNNBackPropagation;
   friend class CNNTraining;
   friend class CNNPerceptron<CNNLayer_store>;
public:

   CNNPerceptron()
      : CNeuroNet<tLayer>()
   {
   };
   explicit CNNPerceptron(unsigned int a_NumLayers,
       unsigned int *a_pNumSizes, CActi &a_acti)
   {// a_NumLayers слоёв, a_pNumSizes[i] - ширина i-го слоя
      m_MAXLAYERSIZE = a_pNumSizes[0];
      m_numlayers = a_NumLayers;
      m_layers = new CNNLayer[a_NumLayers];
      for (unsigned int i=1; i<=a_NumLayers; ++i)
      {
         m_layers[i-1].Construct(a_pNumSizes[i], a_pNumSizes[i-1], a_acti);
         // считаем размер буфера
         if (m_MAXLAYERSIZE < a_pNumSizes[i])
            m_MAXLAYERSIZE = a_pNumSizes[i];
      }
      m_pBuf = new SYGDATATYPE[m_MAXLAYERSIZE];
   };
   CNNPerceptron(const CNNPerceptron<CNNLayer> &a_NN)
   {
      unsigned int ui; SYGDATATYPE *p;
      m_numlayers = a_NN.m_numlayers;
      m_layers = new tLayer[m_numlayers];
      for (unsigned int i=0; i<m_numlayers; ++i)
      {
         m_layers[i].Construct(
            a_NN.m_layers[i].GetThisLayerSize(),
            a_NN.m_layers[i].GetPrevLayerSize(),
            a_NN.m_layers[i].GetActiFunc()    );

         p=a_NN.m_layers[i].GetRates(ui);
         m_layers[i].SetRates(p, ui);

         p=a_NN.m_layers[i].GetDispl(ui);
         m_layers[i].SetDispl(p, ui);
```

```
      }
      m_MAXLAYERSIZE = a_NN.m_MAXLAYERSIZE;
      m_pBuf = new SYGDATATYPE[m_MAXLAYERSIZE];
   };
   CNNPerceptron(char *filename, CNNIfc &a_Ifc, CActi &a_acti)
   {
      a_Ifc.ReadFromFile(filename,this, a_acti);
   };
   virtual ~CNNPerceptron()
   {
      delete[] m_pBuf;
      delete[] m_layers;
   };
   virtual void Propagate(const SYGDATATYPE *a_pInput,
       const unsigned int i_size, SYGDATATYPE *a_pOutput, const unsigned int o_size)
   {
      // прогоняем данные по слоям
      m_layers[0].SetInput(a_pInput, i_size);
      for (unsigned int i=0; i<m_numlayers-1; ++i)
      {   unsigned int iSize;
         m_layers[i+1].GetInput(iSize);

         m_layers[i].GetAscons(m_pBuf, iSize);
         m_layers[i+1].SetInput(m_pBuf, iSize);
      }
      m_layers[m_numlayers-1].GetAscons(a_pOutput, o_size);
   };
   CNNPerceptron<tLayer> & operator = (const CNNPerceptron<tLayer> & a_NN)
   {
      delete[] m_layers;
      delete[] m_pBuf;
      unsigned int ui; SYGDATATYPE *p;
      m_numlayers = a_NN.m_numlayers;
      m_layers = new tLayer[m_numlayers];
      for (unsigned int i=0; i<m_numlayers; ++i)
      {
         m_layers[i].Construct(
            a_NN.m_layers[i].GetThisLayerSize(),
            a_NN.m_layers[i].GetPrevLayerSize(),
            a_NN.m_layers[i].GetActiFunc()    );

         p=a_NN.m_layers[i].GetRates(ui);
         m_layers[i].SetRates(p, ui);

         p=a_NN.m_layers[i].GetDispl(ui);
         m_layers[i].SetDispl(p, ui);
      }
      m_MAXLAYERSIZE = a_NN.m_MAXLAYERSIZE;
      m_pBuf = new SYGDATATYPE[m_MAXLAYERSIZE];

      return *this;
   };
};
#endif //_LX_NN_H_INCLUDED___


//nnTraining.h
#ifndef _LX_NN_TRAINING_H___
#define _LX_NN_TRAINING_H___

#include <vector>
#include <math.h>
#include "nn1x.h"
#include "nnperceptron.h"

class CNNTraining
{
public:
   static SYGDATATYPE GetDiffNorm(const SYGDATATYPE* a_vec1,
        const SYGDATATYPE* a_vec2, const unsigned int a_uiSize)
   {
      SYGDATATYPE Result = 0;
      for (unsigned int ui=0; ui<a_uiSize; ++ui)
```

```
      {
         SYGDATATYPE a = a_vec1[ui]-a_vec2[ui];
         Result += a*a;
      }
      return Result;
   };
};

/// Обучающий пример сети - последовательность входов и выходов
class CTrainingInstance
{
protected:
   SYGDATATYPE *m_pInput;
   unsigned int m_i_size;
   SYGDATATYPE *m_pExpOut;
   unsigned int m_o_size;

public:
   CTrainingInstance(const SYGDATATYPE *a_pInput, const unsigned int i_size,
                const SYGDATATYPE *a_pExpOut, const unsigned int o_size)
   {
      m_i_size  = i_size;  m_pInput  = new SYGDATATYPE[m_i_size];
      m_o_size  = o_size;  m_pExpOut = new SYGDATATYPE[m_o_size];

      m_pExpOut = new SYGDATATYPE[m_o_size];
      memcpy(m_pInput,  a_pInput,  m_i_size*sizeof(SYGDATATYPE));
      memcpy(m_pExpOut, a_pExpOut, m_o_size*sizeof(SYGDATATYPE));
   };
   CTrainingInstance &operator =(const CTrainingInstance &a_TI)
   {
      if (a_TI.m_i_size != this->m_i_size)
      {
         delete[] m_pInput;
         m_pInput = new SYGDATATYPE[m_i_size];
         this->m_i_size  = a_TI.m_i_size;
      }
      if (a_TI.m_o_size != this->m_o_size)
      {
         delete[] m_pExpOut;
         m_pExpOut = new SYGDATATYPE[m_o_size];
         this->m_o_size  = a_TI.m_o_size;
      }
      memcpy(m_pInput,  a_TI.GetInput(),  m_i_size*sizeof(SYGDATATYPE));
      memcpy(m_pExpOut, a_TI.GetOutput(), m_o_size*sizeof(SYGDATATYPE));
      return *this;
   };
   CTrainingInstance(const CTrainingInstance &a_TI)
   {
      m_i_size  = a_TI.m_i_size;      m_pInput  = new SYGDATATYPE[m_i_size];
      m_o_size  = a_TI.m_o_size;      m_pExpOut = new SYGDATATYPE[m_o_size];
      memcpy(m_pInput,  a_TI.GetInput(),  m_i_size*sizeof(SYGDATATYPE));
      memcpy(m_pExpOut, a_TI.GetOutput(), m_o_size*sizeof(SYGDATATYPE));
   };
   ~CTrainingInstance()
   {
      delete[] m_pInput;
      delete[] m_pExpOut;
   };
   SYGDATATYPE *GetInput() const{return m_pInput;};
   unsigned int GetISize() const{return m_i_size;};
   SYGDATATYPE *GetOutput()const{return m_pExpOut;};
   unsigned int GetOSize() const{return m_o_size;};
};

class CTrainingInstances : protected std::vector<CTrainingInstance>
{
   friend class CNNIfc;
public:
   CTrainingInstances(char *filename, CNNIfc &a_Ifc)
   {
      a_Ifc.ReadInstanses(filename, this);
   };
   CTrainingInstances(){};
```

```
   std::vector<CTrainingInstance>::size_type size() const
     {return ((std::vector<CTrainingInstance>*)this)->size();};
   std::vector<CTrainingInstance>::const_reference operator[](size_type _P) const
     {return ((std::vector<CTrainingInstance>*)this)->operator [] (_P); };
   std::vector<CTrainingInstance>::reference operator[](size_type _P)
     {return ((std::vector<CTrainingInstance>*)this)->operator [] (_P); };
   void AddInstance(CTrainingInstance a_Instance)
   {
      this->push_back(a_Instance);
   };
   void RemoveInstance(unsigned int Index)
   {
      assert(Index<this->size());
      erase(begin()+Index);
   };
   CTrainingInstances & operator = (const CTrainingInstances & a_TIs)
   {
      (std::vector<CTrainingInstance>(*this)).operator =
          ((std::vector<CTrainingInstance>)a_TIs);
   };
   SYGDATATYPE CalcLipshic()
   {// константа Липшица есть по пределению inf_i_j(||fi-fj||/||xi-xj||)
      SYGDATATYPE Result = 0;
      SYGDATATYPE fsize = begin()->GetOSize();
      SYGDATATYPE xsize = begin()->GetISize();
      for (iterator Iter1=begin();Iter1!=end();++Iter1)
      for (iterator Iter2=begin();Iter2!=end();++Iter2)
      {   //перебираем все попарно
         if (Iter1==Iter2)
            continue;
         SYGDATATYPE CurLip =
            GetDiffNorm(Iter1->GetOutput(), Iter2->GetOutput(), fsize) /
            GetDiffNorm(Iter1->GetInput(), Iter2->GetInput(), xsize);
         if (Result < CurLip)
            Result = CurLip;
      }
      return Result;
   };
protected:
   SYGDATATYPE GetDiffNorm(const SYGDATATYPE* a_vec1,
        const SYGDATATYPE* a_vec2, const unsigned int a_uiSize)
   {
      return sqrt( CNNTraining::GetDiffNorm(a_vec1, a_vec2, a_uiSize) );
   };
};

class CNNBackPropagation : public CNNTraining
{
public:
   static void Train(CNNPerceptron<CNNLayer> *a_pNN, CTrainingInstances *a_pTI,
     double a_Error, unsigned int a_maxiter=0, unsigned int iter_start=0, SYGDATATYPE metta=-0.1)
   {
#define MY_MAX_VAL 100
      const unsigned int nI = a_pNN->GetINum();
      const unsigned int nO = a_pNN->GetONum();
      const unsigned int _NI = a_pTI->size();
      const unsigned int uiTIsize = a_pTI->size();
      assert(nI && nO && _NI && uiTIsize);
      SYGDATATYPE Error=a_Error*2; // начальное значение - так, чтобы работало
      unsigned int iBufSize;
      SYGDATATYPE *pNetBuf = a_pNN->GetBuf(iBufSize);
      // создадим нс с сохранением промежуточных значений выходов и синаптическх сумм
      // для опимальной реализации метода, а также для сравнения погрешности
      CNNPerceptron<CNNLayer_store> NNs(*a_pNN);

      SYGDATATYPE *pDelta = //new SYGDATATYPE[iBufSize*2];
         (SYGDATATYPE*)malloc( sizeof(SYGDATATYPE) * iBufSize * 2 );

      bool bIsNumItersLimited = (a_maxiter != 0);
      for (unsigned int j=0;Error>a_Error;++j)
        {
         bool IsOwerflowSimilarErrorOccured = false;
         unsigned int j_ = (j+iter_start)%uiTIsize ;
```

```
#ifdef _LX_IS_LOG_ERROR_IN_ITERATIONS
         { // Логируем ошибку
            FILE *faaa=fopen("error-backbprop.log","a");
            Error = 0;
            for (unsigned int ii =0; ii<uiTIsize; ++ii)
            {
               NNs.Propagate(
                  (*a_pTI)[ii].GetInput(), (*a_pTI)[ii].GetISize(),
                  pNetBuf,                 (*a_pTI)[ii].GetOSize() );

               double Error1 = GetDiffNorm(pNetBuf,
                  (*a_pTI)[ii].GetOutput(), (*a_pTI)[ii].GetOSize() );
               fprintf(faaa, "%2.1e\t",Error);
            } fprintf(faaa, "j_=%d",j_);
            fprintf(faaa, "\n");
            fclose(faaa);
         }
#endif //_LX_IS_LOG_ERROR_IN_ITERATIONS

#ifndef _LX_IS_LOG_ERROR_IN_ITERATIONS
         {    Error = 0;
            for (unsigned int ii =0; ii<uiTIsize; ++ii)
            {
               NNs.Propagate(
                  (*a_pTI)[ii].GetInput(), (*a_pTI)[ii].GetISize(),
                  pNetBuf,                 (*a_pTI)[ii].GetOSize() );

               Error += GetDiffNorm(pNetBuf,
                   (*a_pTI)[ii].GetOutput(), (*a_pTI)[ii].GetOSize() );
            }
         }
#endif //_LX_IS_LOG_ERROR_IN_ITERATIONS
         NNs.Propagate(
            (*a_pTI)[j_].GetInput(),  (*a_pTI)[j_].GetISize(),
            pNetBuf,                  (*a_pTI)[j_].GetOSize() );
         {   //для последнего слоя
            unsigned int ui1,ui2, nnn;
            for (ui1=0;ui1<(*a_pTI)[j_].GetOSize();++ui1)
            {
               double y_asterisk_minus_y = ( pNetBuf[ui1] - (*a_pTI)[j_].GetOutput()[ui1] );
               {
               double dy_div_ds = NNs.m_layers[NNs.m_numlayers-1].GetActiFunc().dfds(
                     NNs.m_layers[NNs.m_numlayers-1].GetS()[ui1]   );
               pDelta[ui1+iBufSize*((NNs.m_numlayers-1)%2)] = y_asterisk_minus_y * dy_div_ds;
               }
            }
            for (ui1=0;ui1<NNs.m_layers[NNs.m_numlayers-1].m_iThisLayerSize;++ui1)
            for (ui2=0;ui2<NNs.m_layers[NNs.m_numlayers-1].m_iPrevLayerSize;++ui2)
            {
               double delta_w_i_j = metta *
                  pDelta[ui1+iBufSize*((NNs.m_numlayers-1)%2)] *
                  NNs.m_layers[a_pNN->m_numlayers-1].GetInput(nnn)[ui2];
               a_pNN->m_layers[a_pNN->m_numlayers-1].
                  GetRates(nnn)[ui1 * NNs.m_layers[NNs.m_numlayers-1].
                  m_iPrevLayerSize +ui2] = delta_w_i_j;
            }
         }{   // Заполняем остальные слои
            unsigned int ui1,ui2, nnn;
         for (signed int uiNumLayer=a_pNN->m_numlayers-2; uiNumLayer>=0; --uiNumLayer)
         {
            for (ui1=0;ui1<NNs.m_layers[uiNumLayer].m_iPrevLayerSize;++ui1)
            {// расчёт дельта житого
               double ssss = 0;
               for (unsigned int iiii=0; iiii<NNs.m_layers[uiNumLayer+1].m_iThisLayerSize; ++iiii)
               {
                  unsigned int uiIndex1 =
                      ui1 * NNs.m_layers[uiNumLayer+1].m_iPrevLayerSize+iiii;
                  unsigned int uiIndex2 = iBufSize*((uiNumLayer+1)%2) + iiii;
                  ssss += a_pNN->m_layers[uiNumLayer+1].
                     GetRates(nnn)[uiIndex1] * pDelta[uiIndex2];
               }
               pDelta[ui1+iBufSize*((uiNumLayer+1)%2)] = ssss *
```

```
                NNs.m_layers[uiNumLayer].GetActiFunc().dfds(
                   NNs.m_layers[uiNumLayer].GetS()[ui1]   );
            }
            for (ui1=0;ui1<NNs.m_layers[uiNumLayer].m_iThisLayerSize;++ui1)
            for (ui2=0;ui2<NNs.m_layers[uiNumLayer].m_iPrevLayerSize;++ui2)
            {
               unsigned int Index1 = ui1 * NNs.m_layers[uiNumLayer].m_iPrevLayerSize;
               double delta_mult_y = pDelta[ui1+iBufSize*((uiNumLayer+1)%2)]*
                     NNs.m_layers[uiNumLayer].GetInput(nnn)[ui2];
               a_pNN->m_layers[uiNumLayer].GetRates(nnn)[Index1 + ui2] =
                     metta * delta_mult_y;
            }
         }
         }
         // Пишем результат итерации в NNs
         for (unsigned int uiLas=0; uiLas<NNs.m_numlayers; ++uiLas)
         { // по слоям
            unsigned int nnn, iooooo;
            SYGDATATYPE *pRates = NNs.m_layers[uiLas].GetRates(nnn);
            for (unsigned int uiI = 0; uiI<nnn; ++uiI)
            {
               if (fabs(pRates[uiI]) >= MY_MAX_VAL)
               {
                  IsOwerflowSimilarErrorOccured = true;
                  pRates[uiI] = 0.5-double(rand()%MY_MAX_VAL)/MY_MAX_VAL;
               }
               else
                  pRates[uiI] += a_pNN->m_layers[uiLas].GetRates(iooooo)[uiI];
            }
         }
         if (IsOwerflowSimilarErrorOccured)
         {
            std::cerr<< "IsOwerflowSimilarErrorOccured is true :-(\n";
         }
         if (bIsNumItersLimited)
            if (--a_maxiter == 0)
               break;
        }
      delete[] pDelta;
      // а теперь возвратим значение в a_pNN
      for (unsigned int i=0; i<NNs.m_numlayers; ++i)
      { // по слоям
         unsigned int ui;
         SYGDATATYPE *p=0;
         p=NNs.m_layers[i].GetRates(ui);
         a_pNN->m_layers[i].SetRates(p, ui);

         p=NNs.m_layers[i].GetDispl(ui);
         a_pNN->m_layers[i].SetDispl(p, ui);
      }
   };
};

class CNNStochastic : public CNNTraining
{
public:
   static void Train(CNNPerceptron<CNNLayer> *a_pNN, CTrainingInstances *a_pTI,
      double a_Error, unsigned int a_maxiter=0, SYGDATATYPE SkorostSkhodimosti=0.1)
   {
      const unsigned int nI = a_pNN->GetINum();
      const unsigned int nO = a_pNN->GetONum();
      const unsigned int _NI = a_pTI->size();
      const unsigned int uiTIsize = a_pTI->size();
      assert(nI && nO && _NI && uiTIsize);
      SYGDATATYPE Error[2];
      // Освоим буфер сети
      unsigned int iBufSize;
      SYGDATATYPE *pNetBuf = a_pNN->GetBuf(iBufSize);
      //Считаем, что произвольные малые значения уже присвоены
      CNNPerceptron<CNNLayer> MyNN(*a_pNN);
      CNNPerceptron<CNNLayer> *pMyNNpoiters[2];
      short int best_index = 1, worst_index = (best_index^1)&1;
      pMyNNpoiters[0] = &MyNN; pMyNNpoiters[1] = a_pNN;
```

```
   {//расчёт ошибки для всех примеров
      Error[best_index]=0;
      for (unsigned int j_=0; j_<_NI; ++j_)
      {
         pMyNNpoiters[best_index]->Propagate(
            (*a_pTI)[j_].GetInput(),  (*a_pTI)[j_].GetISize(),
            pNetBuf,                  (*a_pTI)[j_].GetOSize() );

         Error[best_index] +=
            GetDiffNorm(pNetBuf, (*a_pTI)[j_].GetOutput(), (*a_pTI)[j_].GetOSize() );
      }
   }
   bool bIsNumItersLimited = (a_maxiter != 0);
   unsigned int _numiters=0;
   signed int numDullIters=0;
   while (Error[best_index] > a_Error)
   {
      {// Корректируем веса в одном из элементов
         for (unsigned int uiLas=0; uiLas<pMyNNpoiters[worst_index]->m_numlayers; ++uiLas)
         { // по слоям
            unsigned int nnn;
            SYGDATATYPE *pRates_w = pMyNNpoiters[worst_index]->m_layers[uiLas].GetRates(nnn);
            SYGDATATYPE *pRates_b = pMyNNpoiters[best_index]->m_layers[uiLas].GetRates(nnn);
            for (unsigned int uiI = 0; uiI<nnn; ++uiI)
            {// изменяем а случайную величину веса
               pRates_w[uiI] = pRates_b[uiI] +
                 SkorostSkhodimosti*(5e2-rand()%1000)*Error[best_index]/(_NI*500);
            }
         }
      }{//расчёт ошибки для всех примеров нового приближения
         Error[worst_index]=0;
         for (unsigned int j_=0; j_<_NI; ++j_)
         {
            pMyNNpoiters[worst_index]->Propagate(
               (*a_pTI)[j_].GetInput(),  (*a_pTI)[j_].GetISize(),
               pNetBuf,                  (*a_pTI)[j_].GetOSize() );
            Error[worst_index] += GetDiffNorm(pNetBuf,
               (*a_pTI)[j_].GetOutput(), (*a_pTI)[j_].GetOSize() );
         }
      }
#ifdef _LX_IS_LOG_ERROR_IN_ITERATIONS
         { // Логируем ошибку
            FILE *faaa=fopen("error-stoch.log","a");
            fclose(faaa);
         }
#endif //_LX_IS_LOG_ERROR_IN_ITERATIONS
      if (Error[worst_index] < Error[best_index])
      {
         best_index = worst_index;
         worst_index = best_index ^1;
         numDullIters = 0;
      }
      else
      {
         ++numDullIters;
      }
      ++_numiters;
      if (numDullIters > 50)
      {
         SkorostSkhodimosti *= .888;
         numDullIters = 0;
      }
      if (bIsNumItersLimited)
         if (--a_maxiter == 0)
            break;
   }// end of while
   // а теперь возвратим значение в a_pNN
   if (best_index != 1)
   for (unsigned int i=0; i<pMyNNpoiters[best_index]->m_numlayers; ++i)
   { // по слоям
      unsigned int ui;
      SYGDATATYPE *p=0;
      p=pMyNNpoiters[best_index]->m_layers[i].GetRates(ui);
```

```
         a_pNN->m_layers[i].SetRates(p, ui);

         p=pMyNNpoiters[best_index]->m_layers[i].GetDispl(ui);
         a_pNN->m_layers[i].SetDispl(p, ui);
      }
   };
};
#endif //_LX_NN_TRAINING_H___

//nnGATraining.h
#ifndef _LX_CNNTGA_INCLUDED___
#define _LX_CNNTGA_INCLUDED___

#include "nnTraining.h"
#include "../ga/1x_pga.h"
#include <conio.h>

// Последовательный генетический алгоритм
class CNNTGA : public CNNTraining
{
public:
   void Train(CNNPerceptron<CNNLayer> *a_pNN,
      CTrainingInstances *a_pTI, double a_Error)
   {
      const unsigned int nI = a_pNN->GetINum();
      const unsigned int nO = a_pNN->GetONum();
      const unsigned int _NI = a_pTI->size();
      const unsigned int uiTIsize = a_pTI->size();
      assert(nI && nO && _NI && uiTIsize);
      unsigned int iBufSize;
      SYGDATATYPE *pNetBuf = a_pNN->GetBuf(iBufSize);
      // а теперь генерируем начальную популяцию
      CPopulation<CNNIndividual> MyPopulation(/*_NI*/8, CNNIndividual(a_pNN, a_pTI));
      CGenAlgorithm<CNNIndividual, CNNStochastic> MyGenAlg;
      CPopulation<CNNIndividual>::iterator Result = MyPopulation.FindBest();

      while (Result->GetUnfitness() > a_Error)
      {
         MyGenAlg.GenerationNext(MyPopulation);
          { // Логируем ошибку
            double SumError=0;
            FILE *faaai=fopen("error-GA-inst.log","a");
            FILE *faaa=fopen("error-GA.log","a");
            Result = MyPopulation.FindBest();

            for (int ih=0; ih< a_pTI->size()-1; ++ih)
            {   SumError += Result->GetError(ih);
               fprintf(faaai, "%4.1e\t", Result->GetError(ih));
            }
            SumError += Result->GetError(a_pTI->size()-1);
            fprintf(faaai, "%4.1e\n", Result->GetError(a_pTI->size()-1));
            fprintf(faaa, "%1e\n", SumError);

            fprintf(faaai, "\n");
            fclose(faaai);
            fclose(faaa);
         }
      }
      // возвратим значение
      a_pNN->operator =(*MyPopulation.FindBest());
   }
};
#endif //_LX_CNNTGA_INCLUDED___
```