Министерство образования и науки РФ

ФГБОУ ВО «Тверской Государственный Университет»

Математический факультет

Специальность «Компьютерная Безопасность»

Специализация «Математические методы защиты информации»

ДИПЛОМНАЯ РАБОТА

*Использование нейронных сетей в алгоритме обмена ключами*

|  |
| --- |
| Автор:  Левачев Дмитрий Вячеславович,  6 курс, группа М-65 |
| Научный руководитель:  д. ф. – м. наук, профессор  Андреева Елена Аркадьевна |

Допущен к защите:

Руководитель ООП:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Заведующий кафедрой Компьютерной безопасности и

математических методов управления (КБиММУ)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Тверь 2017

**Содержание**

[Введение 4](#_Toc500089930)

[Глава 1. Теоретическая часть 5](#_Toc500089931)

[1.1. Основные сведения о нейронных сетях 5](#_Toc500089932)

[1.1.1. Искусственные нейронные сети 5](#_Toc500089933)

[1.1.2. Математическая модель нейрона 6](#_Toc500089934)

[1.1.3. Функция активации 6](#_Toc500089935)

[1.1.4. Веса нейронной сети 7](#_Toc500089936)

[1.1.5. Слои нейронной сети 8](#_Toc500089937)

[1.1.6. Архитектуры нейронных сетей 9](#_Toc500089938)

[1.1.7. Однослойные нейронные сети 9](#_Toc500089939)

[1.1.8. Многослойные нейронные сети 10](#_Toc500089940)

[1.1.9. Нейронные сети прямого распространения 11](#_Toc500089941)

[1.1.10 Нейронные сети с обратной связью 11](#_Toc500089942)

[1.1.11. Процесс обучения 12](#_Toc500089943)

[1.2. Общие сведения из криптографии 13](#_Toc500089944)

[1.2.1. Обмен ключами 13](#_Toc500089945)

[1.3. Нейрокриптография 15](#_Toc500089946)

[1.3.1. Синхронизация нейронных сетей 15](#_Toc500089947)

[1.3.2. Древовидные Машины Четности 16](#_Toc500089948)

[1.3.3. Правила обучения 17](#_Toc500089949)

[1.3.4. Нейросинхронизация 19](#_Toc500089950)

[1.4. Возможные атаки 19](#_Toc500089951)

[1.4.1. Простая атака 19](#_Toc500089952)

[1.4.2. Атака большинством 20](#_Toc500089953)

[1.4.3. Генетическая атака 21](#_Toc500089954)

[Глава 2. Программная реализация протокола. 23](#_Toc500089955)

[2.1. Постановка задачи 23](#_Toc500089956)

[2.2. Транспортный слой 24](#_Toc500089957)

[2.2.1. Декодер и кодировщик данных 26](#_Toc500089958)

[2.3. Модуль безопасности 27](#_Toc500089959)

[2.3.1. Модуль обмена ключами 27](#_Toc500089960)

[2.3.2. Реализация древовидной машины четности 29](#_Toc500089961)

[2.3.3 Процесс обучения 30](#_Toc500089962)

[2.3.4 Алгоритм обмена «рукопожатиями» 31](#_Toc500089963)

[2.3.5 Реализация алгоритма обмена «рукопожатиями» 33](#_Toc500089964)

[2.3.6. Псевдослучайный генератор 34](#_Toc500089965)

[2.3.7 Генератор входных векторов 35](#_Toc500089966)

[2.3.8. Модуль «наблюдатель» 36](#_Toc500089967)

[2.3.9 Криптографический модуль 36](#_Toc500089968)

[Глава 3. Эксперименты с программой 37](#_Toc500089969)

[3.1. Измерение времени синхронизации при изменении L 37](#_Toc500089970)

[3.2. Измерение времени синхронизации при изменении K 39](#_Toc500089971)

[3.3. Измерение времени синхронизации при изменении N 41](#_Toc500089972)

[3.4. Пример рабочего приложения 43](#_Toc500089973)

[Заключение 44](#_Toc500089974)

[Список литературы 45](#_Toc500089975)

[Приложение 46](#_Toc500089976)

[Листинг программы 46](#_Toc500089977)

# Введение

Проблема обмена ключами является одной из ключевых проблем в информационной безопасности. На данный момент, один из самых известных протоколов обмена ключами – протокол Диффи-Хеллмана. Данный протокол базируется на теории чисел, а конкретно на сложности дискретного логарифма. Но в современном мире, при постоянно увеличивающихся мощностях вычислительных машин, и вероятном появлении квантовых компьютеров в ближайшем будущем, данная вычислительная проблема может исчезнуть, что поставит под удар безопасность протоколов обмена ключами, основанных на теории чисел.

Существует и альтернативные решения проблемы обмена ключами, которые не основаны на теории чисел. Одним из таких, решений является нейрокриптографический протокол обмена ключами. При помощи синхронизации двух древовидных машин чётности, можно обеспечить безопасную передачу данных между двумя абонентами через публичный канал.

В первой главе данной работы представлена необходимая для создания протокола информация о нейронных сетях и конкретно о древовидных машинах четности, а также информация о возможных атаках.

Во второй главе представлены особенности реализации протокола передачи данных.

В третьей главе были произведены некоторые эксперименты с древовидными машинами четности для определения зависимости времени синхронизации от параметров машины.

# Глава 1. Теоретическая часть

## 1.1. Основные сведения о нейронных сетях

### 1.1.1. Искусственные нейронные сети

Искусственные нейронные сети или ИНС – математическая модель и ее реализации, вдохновлённые устройством биологического мозга. Как показано на следующем изображении**,** биологический нейрон состоит из ядра, дендритов и аксона. Аксон разделяется на несколько ветвей которые присоединяются, через синапсы, к дендритам других нейронов [4].

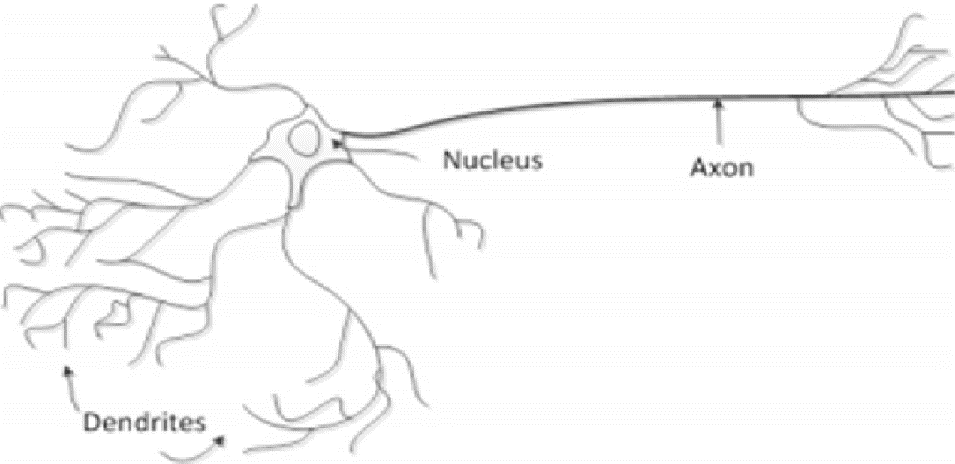


Рис. 1: (Устройство биологического нейрона)

Искусственные нейроны имеют похожую структуру. Они состоят из ядра (вычислительного модуля), нескольких дендритов (входы) и аксон (выход). Как показано на следующем изображении:

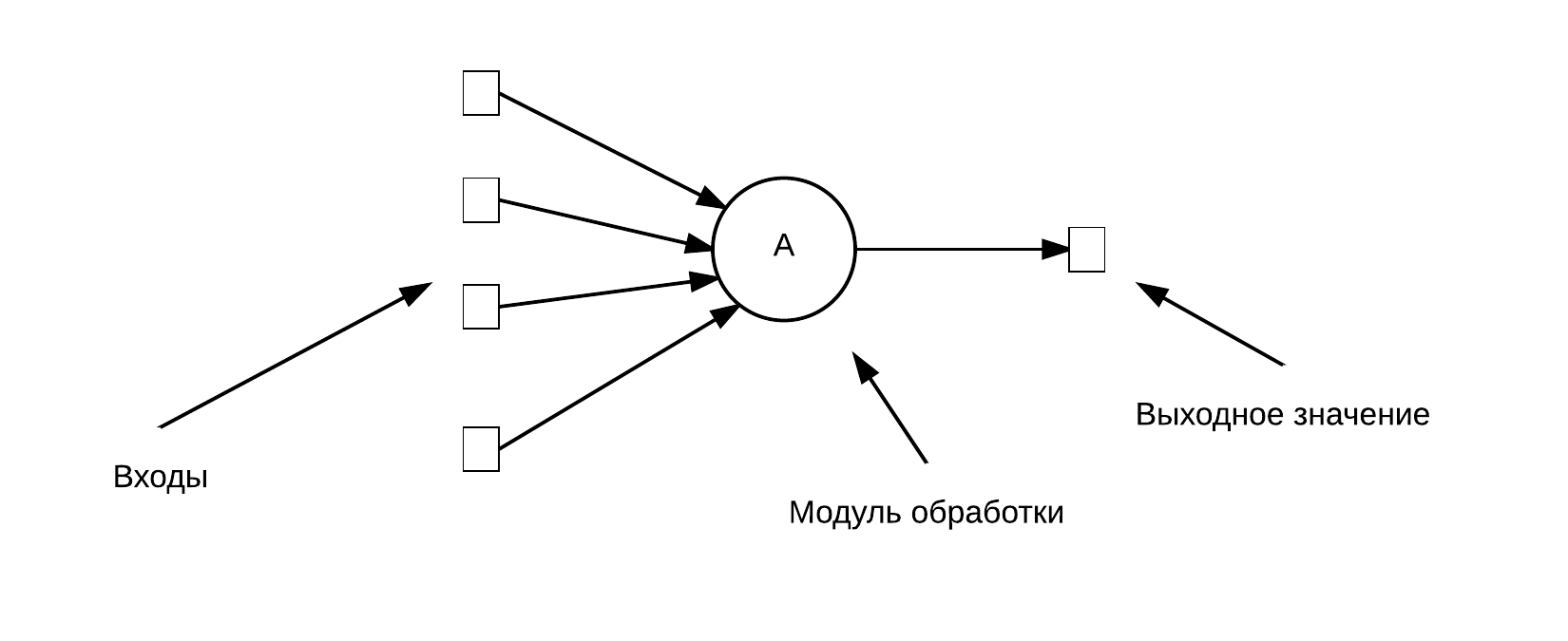


Рис. 2: (Схема устройства нейрона)

Связи между нейронами из нейронной сети являются аналогией синапсов в естественной нейронной сети.

Нейрон выполняет функцию обработчика сигналов, так как он принимает сигналы, приходящие на его входы, после чего, в зависимости от их силы, формирует определенный выходной сигнал. Таким образом, можно сказать, что нейрон имеет сумматор входных сигналов и функцию активации, которая определяет выходной сигнал. Структура искусственного нейрона представлена на следующем изображении:

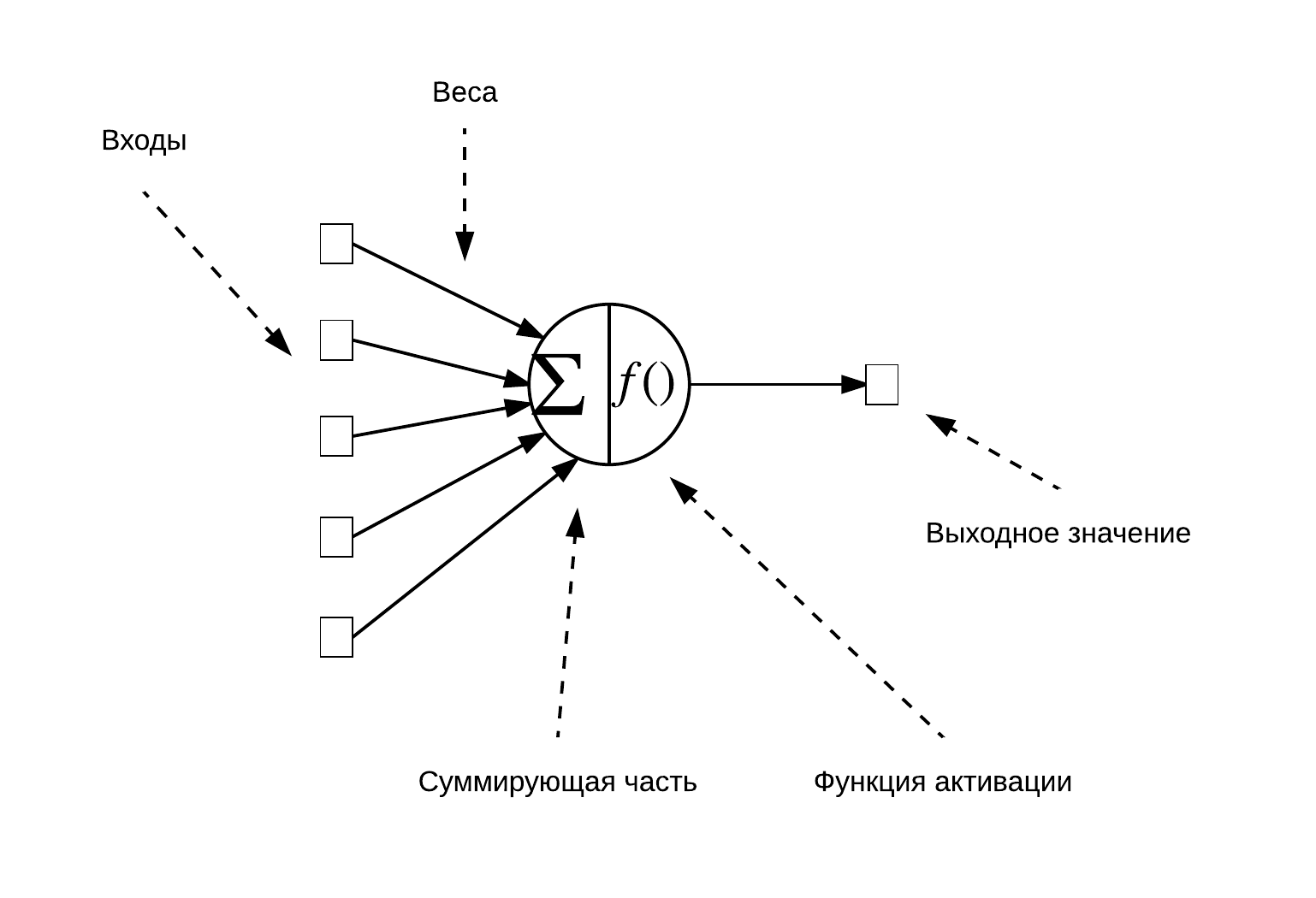


Рис. 3: (Схема элементов нейрона)

### 1.1.2. Математическая модель нейрона

Математический нейрон представляет из себя сумматор, выход которого формируется на основе его входных сигналов и его функции активации следующим образом:

, где

—{\displaystyle x\_{i}} сигналы на входах

{\displaystyle w\_{i}}  —веса входов

f(u) — функция активации

### 1.1.3. Функция активации

Выходной сигнал нейрона вычисляется его функцией активации. Функция активации – это функция, которая однозначно определяет выход нейрона, на основе суммы его входных сигналов [4]. Данная функция призвана добавить нелинейности процессу обработки, которая нужна, так как естественные нейроны имеют нелинейное поведение. Обычно выход данной функции заключен между двух значений. Далее перечислены четыре наиболее используемые функции активации:

1. Сигмоид
2. Гиперболический тангенс
3. Пороговая функция
4. Линейная функция

Уравнения и графики, связанные с функциями, представлены в следующей таблице:

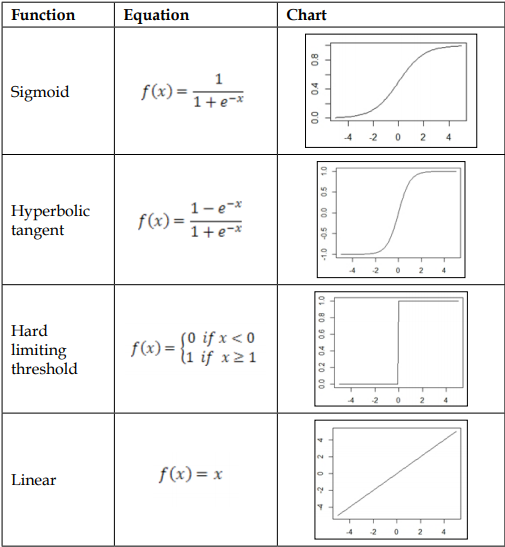


Рис. 4: (Таблица с функциями активации активации)

### 1.1.4. Веса нейронной сети

В нейронной сети, веса представляют соединения между нейронами и способны усиливать или ослаблять нейронный сигнал, например, умножая их на сигналы, тем самым модифицируя их. Изменяя сигнал нейронной сети, веса нейронной сети способны оказывать влияние на выход нейрона, следовательно, активация нейрона зависит от его входов и весов. Входы могут быть получены из выходов других нейронов, либо из внешнего мира. Таким образом, так как веса являются внутренним представлением нейронной сети и влияют на ее выход, мы можем рассматривать их как знания нейронной сети и изменение весов приведет к изменению возможностей нейронной сети.

### 1.1.5. Слои нейронной сети

Естественные нейроны организованы слоями, каждый из которых обеспечивает определенный уровень обработки. Например, входной слой обеспечивает получение информации из внешнего мира, в то время как выходной слой может оказывать влияние на внешний мир. Между этими слоями может существовать некоторое количество скрытых слоев, которые не взаимодействуют с внешним миром напрямую. Все нейроны на уровне имеют общий входной вектор входной и функцию активации.

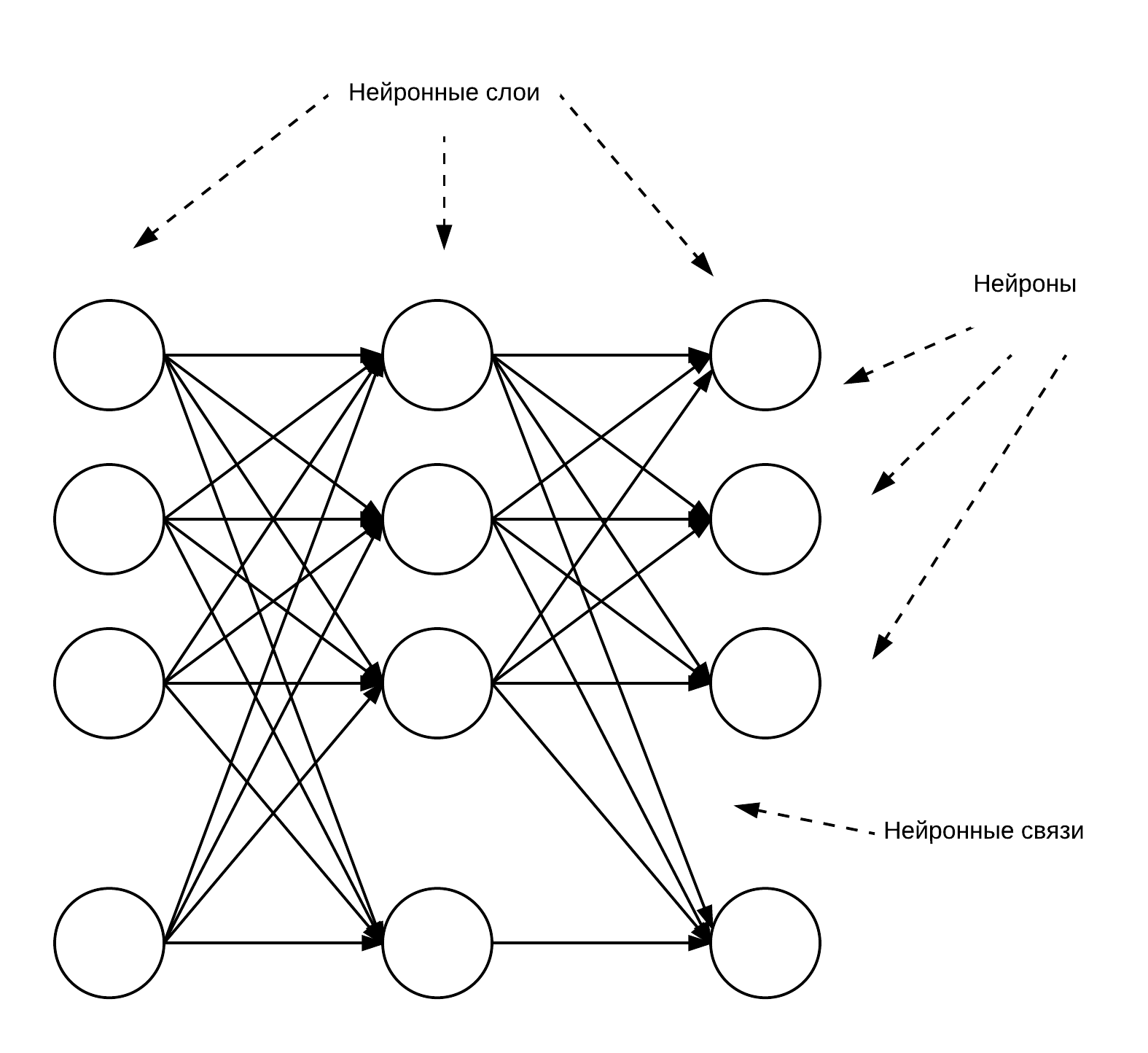


Рис. 5: (Схема слоев нейронной сети)

Нейронные сети могут состоять из нескольких слоев, формируя, так называемые, многослойные нейронные сети. Слои могут быть разделены на три класса:

1. Входной слой
2. Скрытый слой
3. Выходной слой

На практике, дополнительный уровень добавляет еще один уровень абстракции для внешних стимулов, тем самым повышая способность нейронной сети представлять более сложные знания.

### 1.1.6. Архитектуры нейронных сетей

Нейронные сети могут иметь разную архитектуру, в зависимости от того, как нейроны или нейронные слои связаны друг с другом [4]. Архитектура нейронной разрабатывается под определенные нужды. Нейронные сети могут быть использованы для решения большого числа проблем и, в зависимости от природы данной проблемы, нейронная сеть должна быть разработана в с учетом того, чтобы решить данную проблему максимально эффективно. Существуют следующие разновидности архитектур нейронных сетей:

* По нейронным связям
  + Однослойные
  + Многослойные
* По распространению сигнала
  + Прямого распространения
  + С обратной связью

### 1.1.7. Однослойные нейронные сети

В данной архитектуре все нейроны лежат на одном и том же уровне, формируя один единственный слой, как показано на следующем изображении:

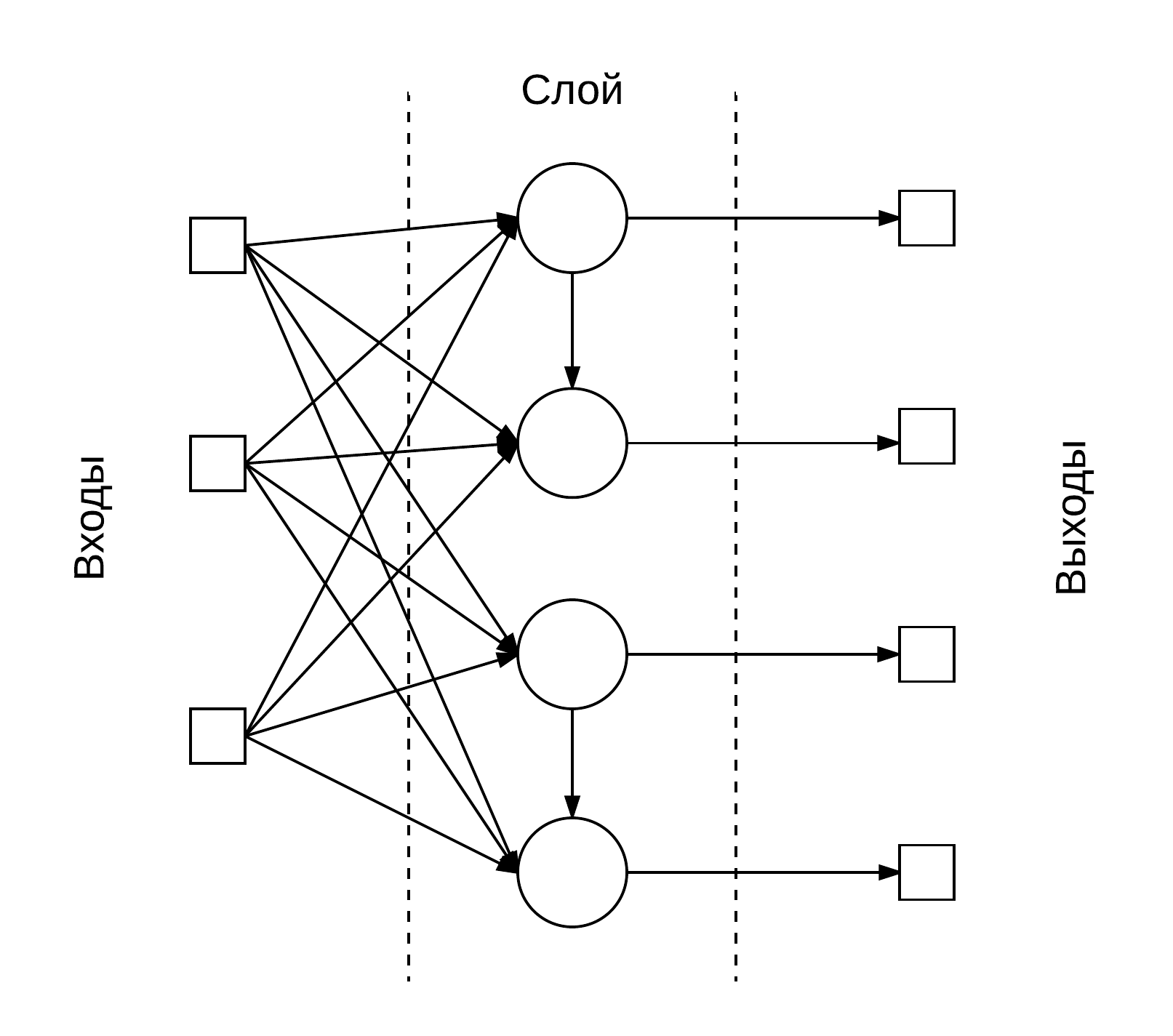


Рис. 6: (Схема однослойной нейронной сети)

Нейронная сеть получает входной сигнал и направляет его нейронам, которые вычисляют выходной сигнал. Нейроны могут быть сильно связаны друг с другом, с рекуррентными связями или без.

### 1.1.8. Многослойные нейронные сети

В этой категории, нейроны разделены на несколько слоев. В рамках уровня, нейроны имеют общий входной сигнал и одну функцию активации. Свои выходные сигналы каждый слой сообщает следующему на вход.

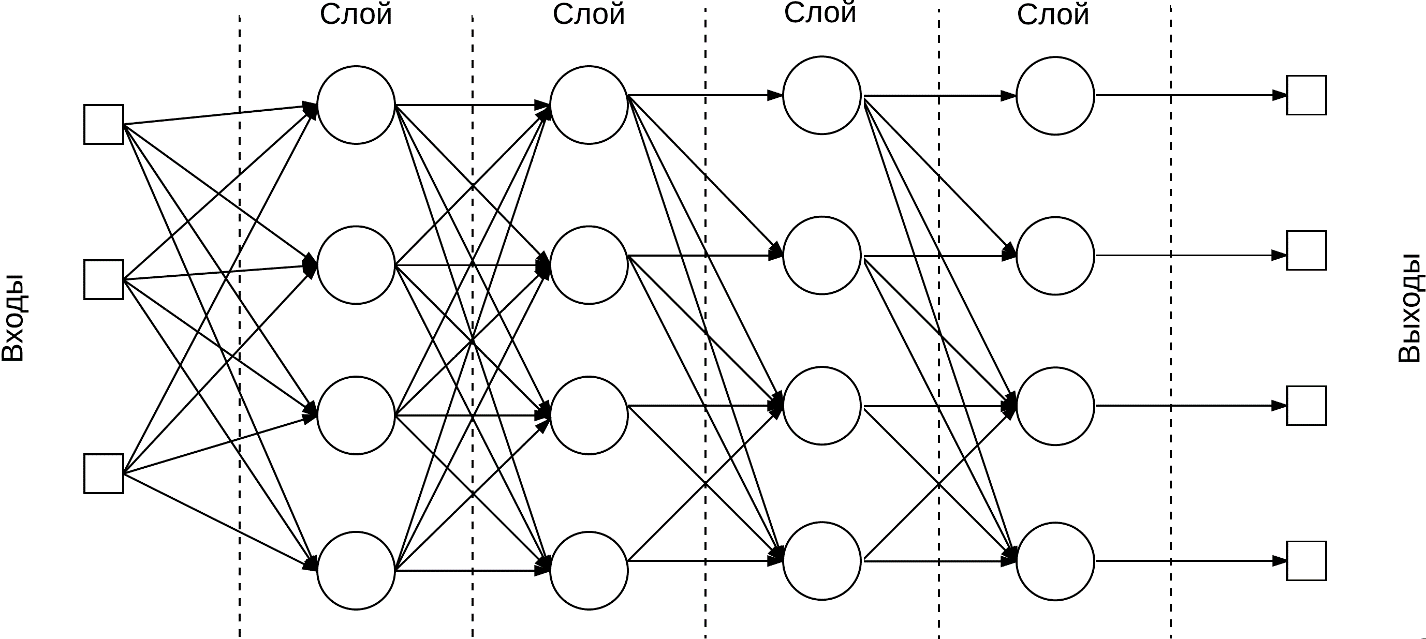


Рис. 7: (Схема многоуровневой нейронной сети)

Радиальные базисные функции и многослойный персептрон, являются хорошими примерами этой архитектуры. Так как эти сети имеют многослойную архитектуру обработки, эти сети адаптируются к обучению по нелинейным данным.

### 1.1.9. Нейронные сети прямого распространения

Сигнал нейронной сити может распространяться либо только в одном направлении, либо иметь рекуррентные зависимости. Первую архитектуру мы называем нейронной сетью прямого распространения. Сначала сигнал подается на входной уровень. Затем, после того как он был обработан, он передается на следующий слой. Многослойный персептрон и радиально базисные функции так же являются хорошим примером нейронных сетей прямого распространения.

### 1.1.10 Нейронные сети с обратной связью

Когда нейронная сеть имеет своего рода внутренние рекуррентные связи, это значит, что сигнал подается назад нейрону или слою, который уже получал и обрабатывал этот сигнал. Такие нейронные сети называются нейронными сетями с обратной связью. Данная архитектура представлена на рисунке:

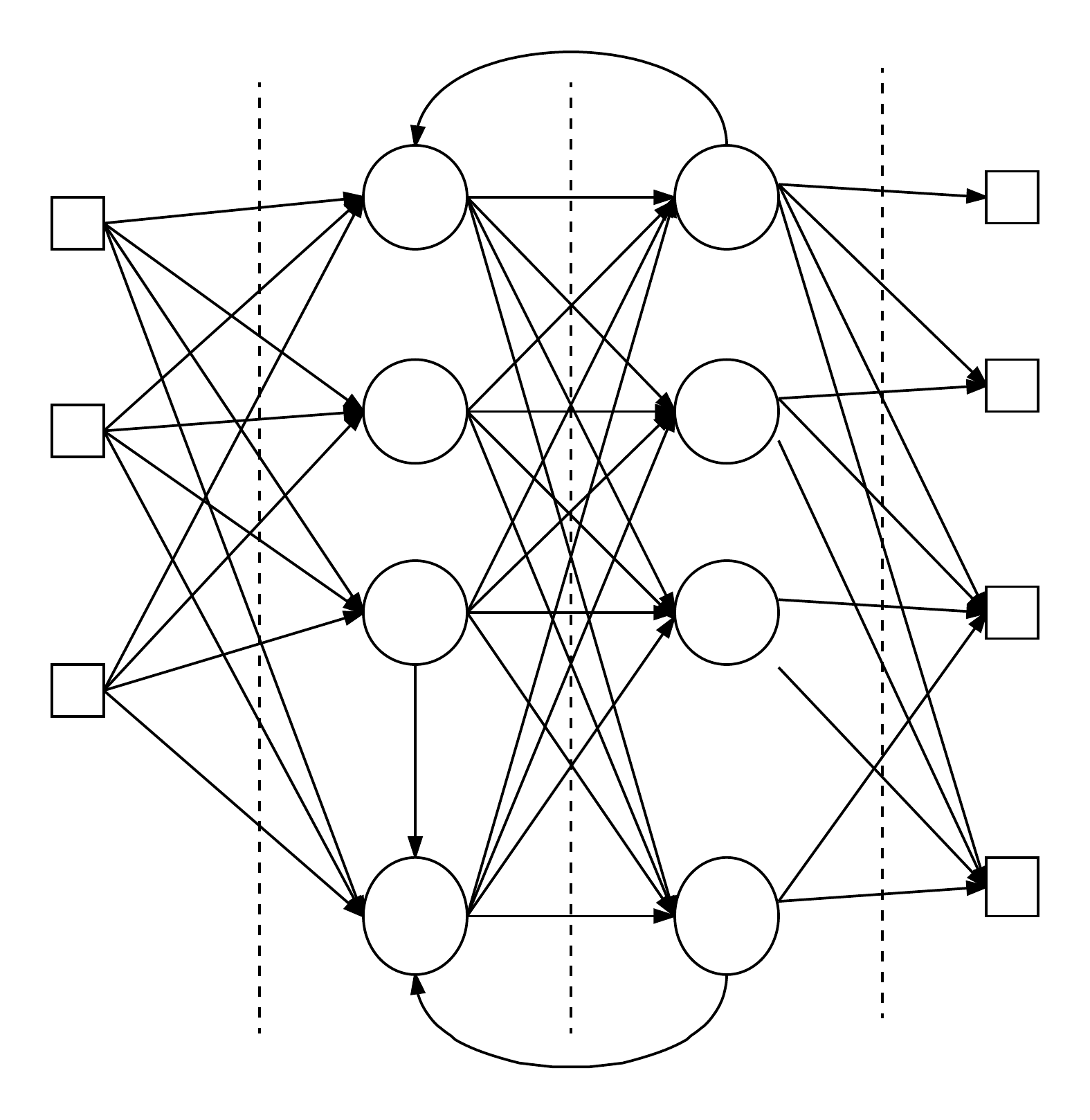


Рис. 8: (Схема нейронной сети с обратной связью)

Особая причинена добавления рекуррентных зависимостей в нейронную сеть – это получение динамичного поведения. На практике, нейронная сеть, предназначенная для проблем, включающих временные ряды или распознавание паттернов, требуют рекуррентные зависимости, для подкрепления обучения. Однако, такую сеть сложно обучать. Большинство нейронных сетей с обратной связью имеют один слой, такие как, сеть Элмана и Хопфилда, но возможно создание многоуровневых рекуррентных сетей, такие как эхо и рекуррентный многоуровневый персептрон.

### 1.1.11. Процесс обучения

В нейронных сетях, все связи между нейронами регулируются нейронами. Как было замечено ранее, веса представляют знания нейронной сети. Различные веса являются причиной того, что нейронные сети производят разный результат для одних и тех же входов. Таким образом, нейронные сети могут улучшать свои результаты регулировкой своих весов согласно правилу обучения. Общая схема процесса обучения представлена на следующем рисунке:

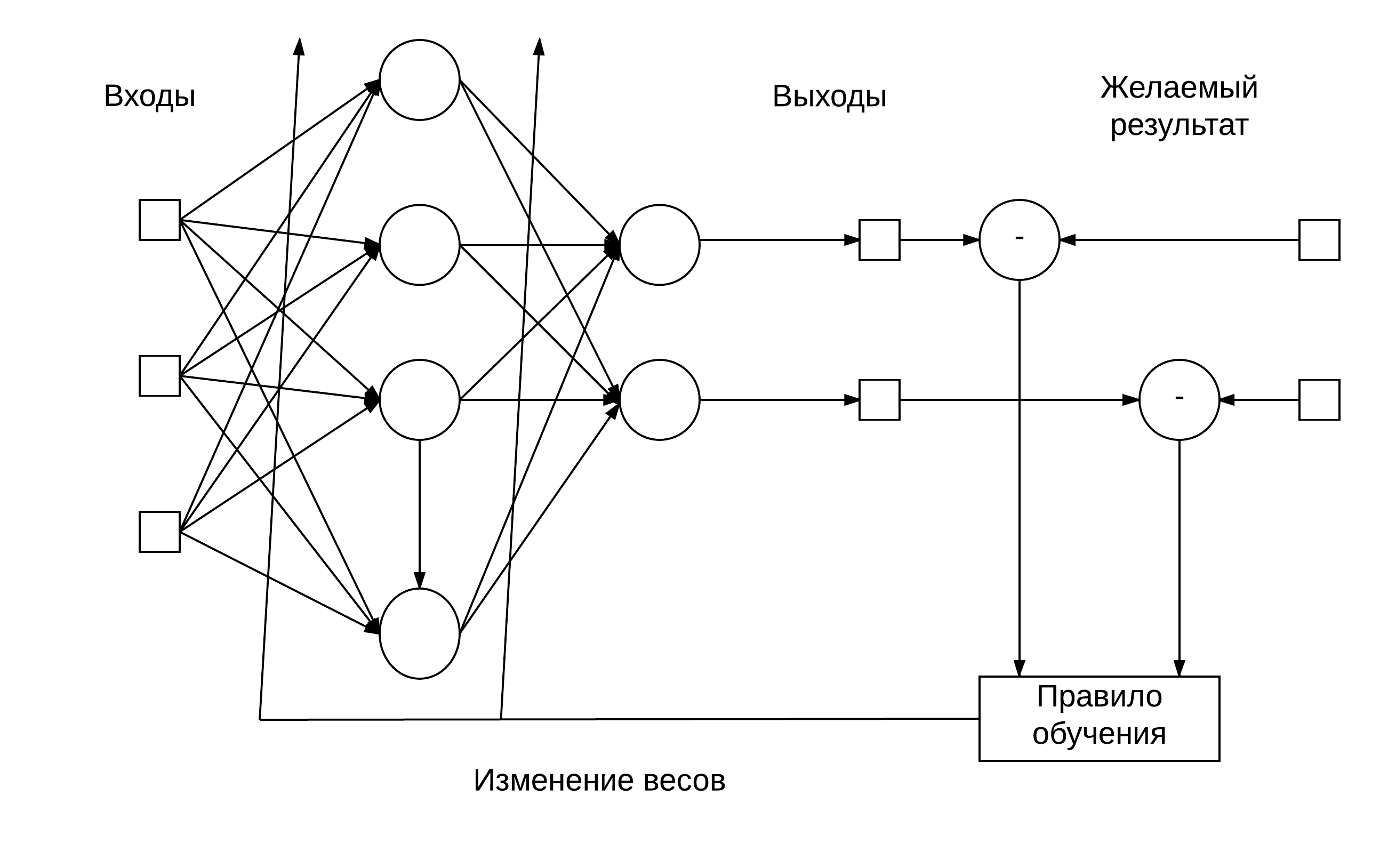


Рис. 9: (Схема процесса обучения)

Процесс, представленный на предыдущем изображении называется обучением с учителем, потому что существует желаемый выход, но нейронные сети также могут учится без учителя, имея только данные для обучения.

## 1.2. Общие сведения из криптографии

### 1.2.1. Обмен ключами

В компьютерной безопасности существует следующая проблема: Два партнера A и B хотят обменятся секретным сообщением через публичный канал [1]. Для обеспечения защиты содержимого сообщения от оппонента E, A шифрует свое сообщение, используя быстрый алгоритм симметричного шифрования. Но теперь B необходимо знать ключ, который принадлежит A для того, чтобы прочитать сообщение. Данная ситуация изображена на рисунке:

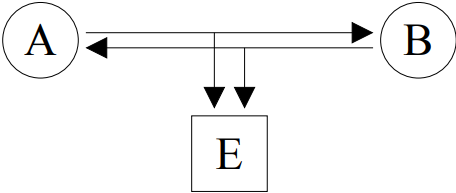


Рис. 10: (Схема обмена ключами)

Для решения данной проблемы обмена ключами существует несколько решений. Первое, A и B могут использовать второй закрытый канал для обмена ключами, также они могут обменятся ключами при личной встрече. Но обычно это крайне сложно или даже невозможно осуществить. В качестве альтернативы, мы можем использовать криптографию с открытым ключом. В данном случае используется алгоритм ассиметричного шифрования, таким образом пара публичных ключей A и B может быть передана между партнерами без необходимости держать их в секрете. Но асимметричный алгоритм шифрование намного более медленный, чем симметричный. Однако, можно достичь такого же результата, используя протокол обмена ключами. В этом случае сообщение передается через публичный канал после чего A и B генерируют закрытый ключ, основываясь на передаваемой информации. Но E не может получить ключ, потому что наблюдения за информацией передаваемой по каналу недостаточно.

## 1.3. Нейрокриптография

### 1.3.1. Синхронизация нейронных сетей

Две нейронные сети начинают с случайно выбранными весовыми векторами. В каждый момент времени они получают общий входной вектор, вычисляют выходное значение и сообщают его друг другу. Если они согласуются, то их веса обновляются по используемому правилу обучения.

В случае, дискретных весов этот процесс ведет к полной синхронизации за конечное число шагов. После чего, полученные веса в обоих сетях имеют одно и то же значение.

Кроме того, третья нейронная сеть может быть обучена, используя в качестве примера входные и выходные значения, генерируемые процессом синхронизации. Так как такая нейронная сеть не может влиять на остальные, обучаемой сети необходимо найти соответствие между входными и выходными значениями, зависящими от времени.

В случае персептрона, который является простейшей нейронной сетью, нет особой разницы между этими двумя ситуациями. Среднее число шагов, необходимое для обучения одинаковое. Но в случае более сложной древовидной машины четности может быть замечено следующее: две нейронных сети, обучаются друг от друга быстрее, чем траться, сеть, которая способна только наблюдать за ними.

Эта разница между однонаправленным и двунаправленным обучением может быть использована для решения проблемы обмена ключами из криптографии. Для этой цели партнеры A и B синхронизируют свои древовидные машины четности. При этом они генерируют их общий сессионный ключ быстрее, чем атакующий способен получить его, обучая другую нейронную сеть. Данное отличие между однонаправленной синхронизацией и двунаправленной является ключевым для безопасности нейросетевого протокола обмена ключами.

### 1.3.2. Древовидные Машины Четности

Древовидные машины четности, которые используются партнерами и атакующими в нейрокриптографии представляют собой многоуровневую нейронной сетью [1]. Общая структура представлена на рисунке.

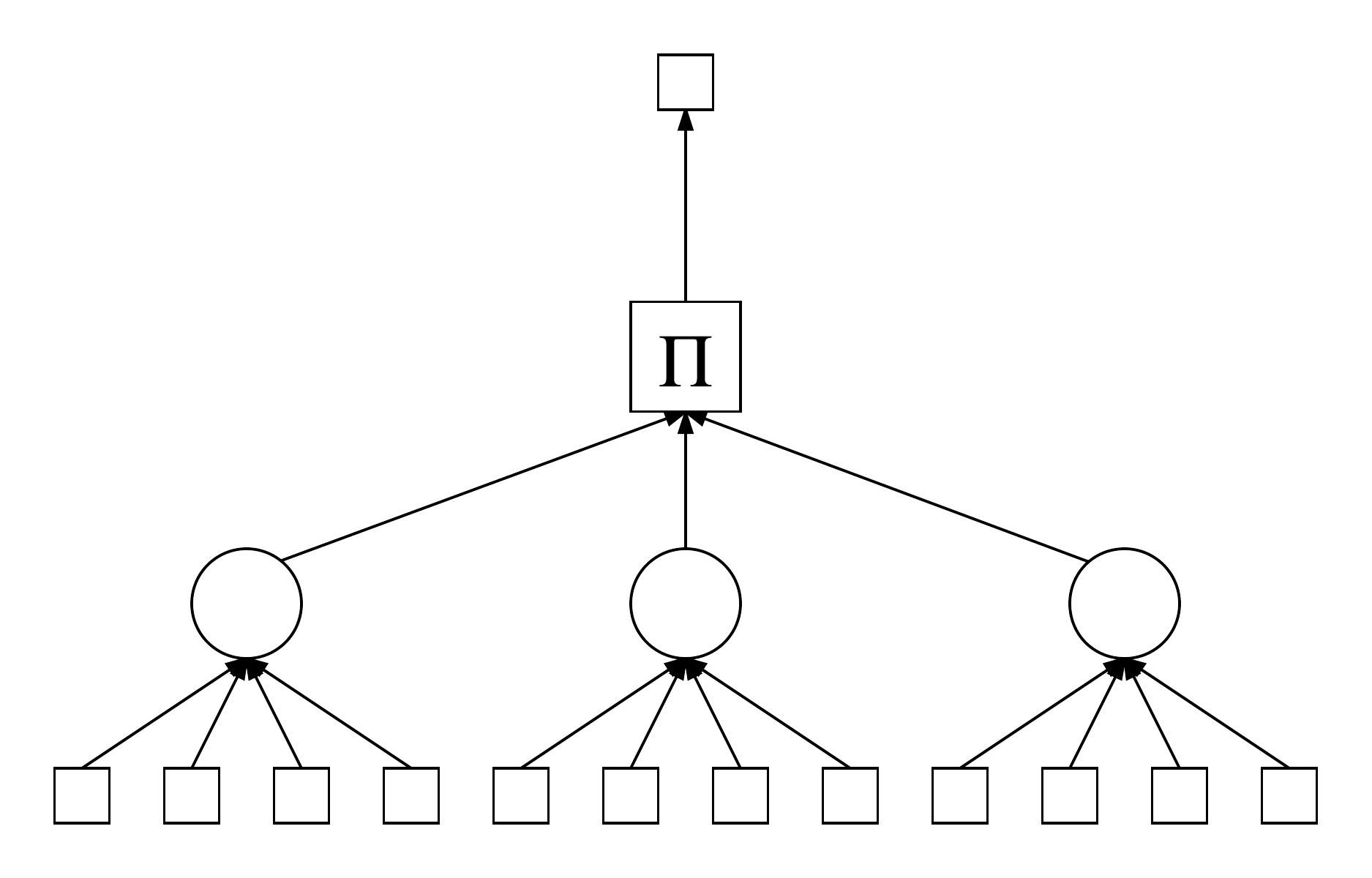


Рис. 11: (Структура древовидной машины четности)

Такие нейронные сети состоят из K скрытых нейронов, являющиеся персептронами. Каждый из них имеет N входных нейронов и один выходной нейрон. Все входные значения являются бинарными.

Веса, которые определяют соответствие между входами и выходами, принимают дискретные значения между *–L* и *L*.

Здесь индексы определяют номер скрытого нейрона древовидной машины четности,входного элементы вектора.

Как и в других нейронных сетях, весовая сумма для текущего входа, определяет выход скрытых модулей. Таким образом полное состояние каждого скрытого модуля может быть представлено локальным полем :

Выходное значение *σi**i*-того нейрона определено как функция sign от hi

Но в особом случае, когда *hi = 0* используется значение *σi = - 1*. Это необходимо для того, чтобы быть уверенными в бинарности выходного значения. Таким образом можно сказать, что когда скрытый нейрон активен, *σi  = + 1*, если весовая сумма положительна, в противном случае, нейрон не активен, *σi  = + 1.*

Таким образом конечное выходное значение древовидной машины четности τ является произведением выходов всех скрытых нейронов,

Можно сказать, что число τ показывает, что число неактивных скрытых нейронов четное или нечетным. В следствии этого, существует различных вариаций , которые ведут к одному и тому же выходному значению τ.

Если в машине есть только один скрытый нейрон, то τ равно *σ1*. Таким образом, машина с имеет то же поведение, что и персептрон.

### 1.3.3. Правила обучения

Находясь в процессе синхронизации, машины A и B начинают со случайно выбранных весов Wi . В каждый момент времени K публичных входных векторов xi генерируются случайным образом, после чего, вычисляются выходные значения τ.

Далее A и B сообщают свои выходные биты друг другу. Если они не равны друг другу, веса не изменяются. В случае равенства, одно из следующих правил обучения можно использовать для изменения весов:

* В случае правила обучения Хэбба, обе нейронных сети обучаются одна от другой.
* Так же возможно, что обе сети обучены на выходах, противоположных им. Этого можно достигнуть используя правило анти-Хэбба.
* Но набор значении выходов не важен для синхронизации до тех пор, пока участвующие сети имеют одинаковый выход. Именно поэтому можно использовать правило случайных блужданий.

При выборе любого из данных правил, веса меняются под их действием только в том случае, когда *σi = τ* . Таким образом, невозможно сказать, без знания значений (*σ1, σ2 , . . . , σK*), какие веса будут изменяться, а какие нет. Эта особенность особенно важна, для реализации криптографии, основанной на нейронных сетях.

Кроме того, правила обучения должны быть реализованы таким образом, чтобы веса оставались в допустимом диапазоне между *– L* и *+ L*. Когда вес заходит за границу данной области, он принимает значение ближайшей границы. Это достигается функцией *g(w)* в каждом обучающем правиле.

После этого, можно считать текущий этап синхронизации завершенным. Этот процесс может быть повторен до тех пор, пока веса машин *A* и *B* не будут равны . Дальнейшее воздействие обучающих правил не способны устранить синхронизацию, потому что, изменения весов зависит только от входного вектора и весов, которые у *A* и *B* идентичные.

### 1.3.4. Нейросинхронизация

Нейросетевой протокол обмена ключами является приложением нейросинхронизации. Оба партнера *A* и *B* используют Древовидные Машины Четности с одинаковой структурой. Параметры *K*, *L* и *N* являются публичными. Каждая нейронная сеть начинает с случайно выбранными весовыми векторами. Эти начальные условия держатся в секрете. В течение процесса синхронизации только входные вектора xi и выходные значения *τA*, τB передаются через публичный канал. Таким образом, каждому участнику известны лишь репрезентация (*σ1, σ2 , . . . , σK*) его собственной Древовидной Машины Четности. Сохранение этой информации в секрете является ключевым фактором для безопасности протокола обмена ключами. После достижения синхронизации *A* и *B* используют свои весовые векторы как общий секретный ключ.

Основная проблема для нападающего E – это то, что внутренняя репрезентация (*σ1, σ2 , . . . , σK*) машин *A* и *B* не известно ему. Так как изменение весов зависит от σi , для успешной атакиявляется важным определить значения скрытых нейронов правильно. Многие известные атаки используют данный метод. Так же существует возможность того, что более эффективные методы атак еще будут найдены, что может полностью уничтожить всю безопасность нейрокриптографии. Однако, такой риск существует практически для всех криптографических алгоритмов.

## 1.4. Возможные атаки

### 1.4.1. Простая атака

Для простой атаки E всего лишь обучает третью Древовидную Машину Четности на основе векторов xi и выходного бита τА . Эти данные могут быть получены простым перехватом сообщений, передаваемых партнерами через публичный канал. E необходимо иметь нейронную сеть той же структуры, что и у A и B и начинать синхронизацию необходимо так же со случайных весовых векторов.

Во время каждой итерации, атакующий вычисляет выход его нейронной сети. После чего, E использует те же правила обучения, что и партнеры, за исключением того, что вместо τА используется τB. Данные правила обучения представлены следующими уравнениями:

* Правило обучения Хэбба:
* Правило обучения анти-Хэбба:
* Правило обучения случайные блуждания:

Таким образом E использует внутреннюю репрезентацию своей собственной сети (*σ1, σ2 , . . . , σK*) для того, чтобы оценить репрезентацию A, даже если их выходы полностью различны. Так как *τА ≠τE*показывает лишь то, что значение хотя бы одного скрытого нейрона *σA ≠ σB* , можно понять, что данный метод, определенно, не лучший для атакующего.

### 1.4.2. Атака большинством

С атакой большинством E может увеличить свои шансы предсказания внутренней репрезентации сети *A*. Для этой цели атакующий использует M Древовидных Машин Четности вместо единственной нейронной сети. В начале процесса синхронизации весовые вектора всех атакующих сетей выбираются случайно.

Так же как в предыдущей атаке, *E* не изменяет веса во время итерации, когда *τА ≠τB*, потому что партнеры пропускают эти входные векторы тоже. Для *τА = τB* обновление обязательно и атакующий вычисляет выходной бит Древовидной Машины Четности . Если выходной бит *m*-той атакующей сети не равен , то *E* находит минимальный модуль локального поля . Затем выходные биты и инвертируются также как и в геометрической атаке. После чего, атакующий подсчитывает количество внутренних репрезентаций его машин и выбирает наиболее общее. Этот голос большинства адаптируется атакующими сетями для применения обучающего правила.

Но эти идентичные обновления создают и усиливают корреляцию между машинами атакующего, что уменьшает эффективность атаки большинством. Если атакующие нейронные сети становятся полностью синхронизированными, то данный метод вырождается в геометрическую атаку.

Для сохранения древовидных машин четности как можно менее коррелированными, атака большинством и геометрическая атака используются попеременно. В четных итерациях для обучения используется голос большинства, а в не четных атакующий применяет геометрическую коррекцию. Таким образом, не все весовые вектора становятся идентичными. Так же, атакующий заменяет атаку большинством геометрической атакой первые 100 итераций процесса синхронизации.

### 1.4.3. Генетическая атака

Генетическая атака предлагает альтернативный метод, который основан не на оптимизации предсказания внутреннего состояния нейронных сетей, а на эволюционном алгоритме. Атакующий начинает с одной случайно инициализированной древовидной машиной четности, но он может увеличит количество сетей до *M*.

В тот момент, когда партнеры обновят веса, из-за того, что в текущей итерации *τА = τB*, следующий генетический алгоритм применяется:

* Пока атакующий имеет, не более , он определяет все возможных репрезентаций которые имеют выходное значение *τА*. После чего они используются, чтобы обновить веса в атакующей сети согласно правилу обучения. Тем самым, атакующий создает вариантов каждой машины на этапе мутаций.
* Но если атакующий уже имеет более чем нейронных сетей, только наиболее подходящие сети будут оставлены. Это достигается путем отбрасывания всех сетей, которые предсказали менее чем *U* выходов *τА* в последние *V* обучающих итераций, для которых *τА = τB* успешно.

Эффективность генетической атаки в большей степени зависит от алгоритма, который выбирает наиболее подходящие нейронные сети. В идеальном случае древовидная машина четности, которая имеет ту же последовательность внутренних репрезентаций, что и *A*, никогда не отбрасывается.

# Глава 2. Программная реализация протокола.

## 2.1. Постановка задачи

Целью данной работы является разработка протокола безопасной передачи данных на основе древовидных машин четности.

Необходимо выполнить программную реализацию протокола передачи данных на основе на основе явления нейросинхронизации. При подключении двух машин друг к другу, необходимо совершить обмен ключами. Далее, после получения общего ключа, все данные, которые передаются через канал необходимо шифровать симметричным шифром с использованием данного ключа. В качестве языка программирования будет использоваться язык Java.



Рис. 12: (Схема уровней приложения)

Таким образом, необходимо реализовать следующие:

1. Транспортный слой, который будет отвечать за преобразования данных, которые отправляются по сети.
2. Слой безопасности, отвечающий за обмен ключами и шифрование сообщений.
3. Пользовательский интерфейс, через который пользователь сможет взаимодействовать с данной программой.

## 2.2. Транспортный слой

Для реализации протокола передачи данных на основе древовидных машин четности необходимо реализовать транспортный слой, который будет определять в каком виде будут представлены сообщения, курсирующие между удаленными машинами по сети. Здесь важно отметить, что дынные, передаваемые по сети обычно представлены в байтовом виде, что несколько усложняет реализацию протокола, так как слой данных, реализующий бизнес логику обычно работает с данными в виде объектов, имеющими поля и методы. Таким образом для реализации трансфера объектов в их байтовое представление, канал передачи данных, был разделен мной на несколько уровней. Данные уровни представлены на рисунке:

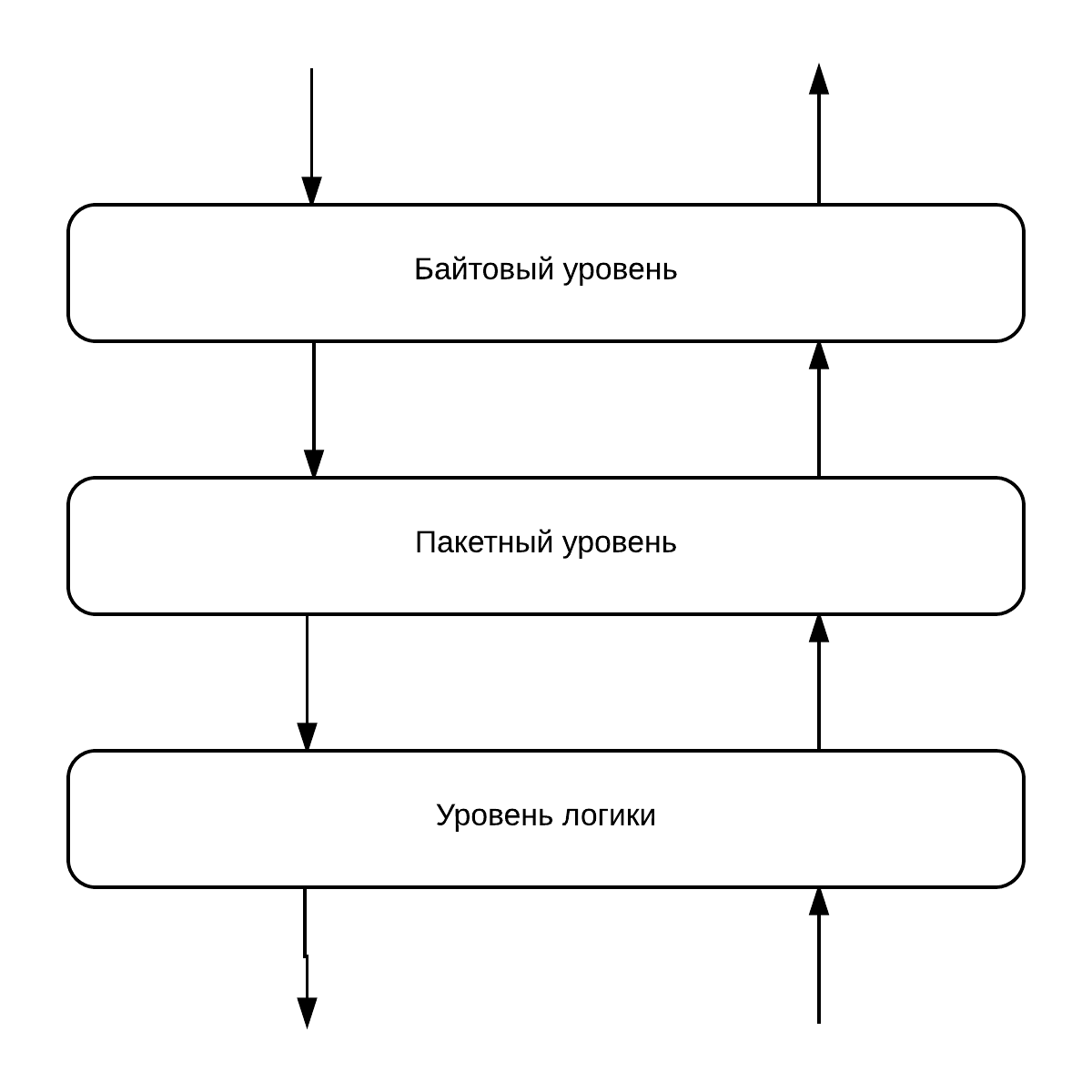


Рис. 13: (Уровни транспортного слоя)

Таким образом, для удобства реализации протокола, в моей программной реализации были выделены следующие уровни данных:

1. Байтовый уровень. На данном уровне все данные представлены набором байт. Данный уровень необходим для того, чтобы осуществлять обмен данными по сети, так как для передачи данных по сети, необходимо, чтобы они были представлены в байтовом виде.
2. Пакетный уровень. На данном уровне осуществляется контроль безопасности протокола. Именно на данном уровне расположены механизмы, запускающие процесс обмена ключами, а также механизмы, осуществляющие шифрование пакетов с данными.
3. Уровень логики. На данном уровне расположены модули, отвечающие за основную бизнес логику приложения.

Для реализации данного разбиения необходимо реализовать набор модулей программы, осуществляющий определенны действия по трансформации данных, для их перевода из одного представления в другое. Таким образом данные, пришедшие из сети будут проходить через определенный набор декодеров, которые будут осуществлять переход данных из байтового представления в представление бизнес объектов и наоборот, когда мы будем посылать наши данные через сеть, будет использован определенный набор кодировщиков для превращения бизнес объектов в байты.

Транспортный слой протокола передачи данных основанного на древовидных машинах четности, можно разбить на следующие модули в соответствии с уровнями, на которых они располагаются:

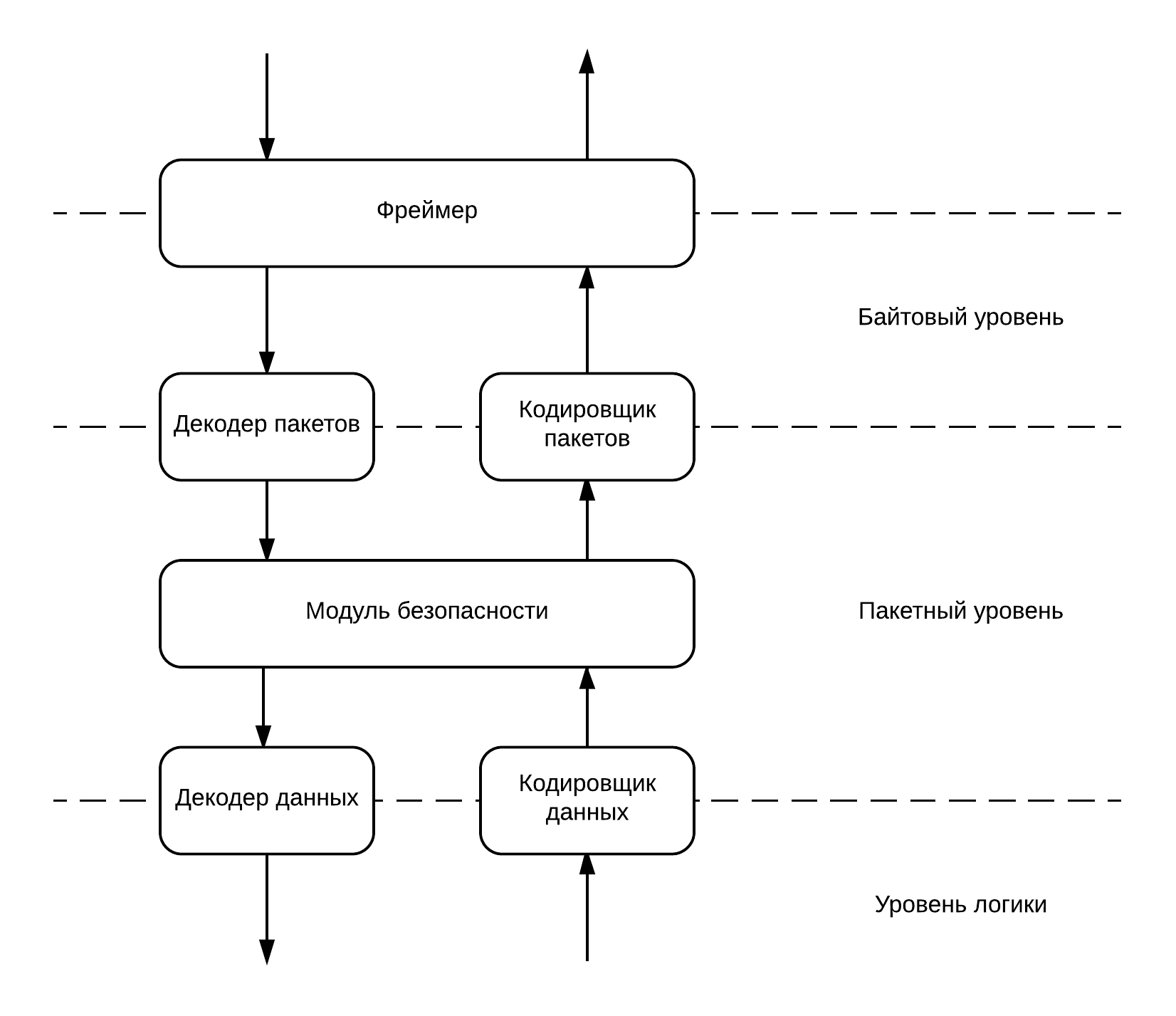


Рис. 14: (Модули транспортного слоя)

Таким образом, транспортный слой состоит из следующих модулей:

1. Декодер и кодировщик данных.
2. Модуль безопасности.
3. Декодер и кодировщик пакетов.
4. Фреймер.

### 2.2.1. Декодер и кодировщик данных

Как уже было сказано ранее, основная логика приложения работает с данными, которые являются специфичными для бизнес логики приложения. Данные модули необходимы для того, чтобы осуществить преобразование бизнес объектов приложения в пакеты, для дальнейшей транспортировки. Так как данные модули напрямую взаимодействуют с бизнес объектами, в случае изменения бизнес объектов, необходимо так же будет изменять и данные модули, чтобы быть способными работать с обновленными бизнес объектами. Конкретно в случае моей программы, бизнес объектом является простое сообщение, несущее в себе простой текст. В процессе перехода объекта на пакетный слой, текст упаковывается в специальный пакет, который необходим для дальнейшего прохождения по каналу.

## 2.3. Модуль безопасности

Именно в данном модуле осуществляется контроль за безопасностью процесса передачи данных. Данный модуль обрабатывает все входящие и исходящие пакеты. Структура модуля безопасности представлена на следующем рисунке:

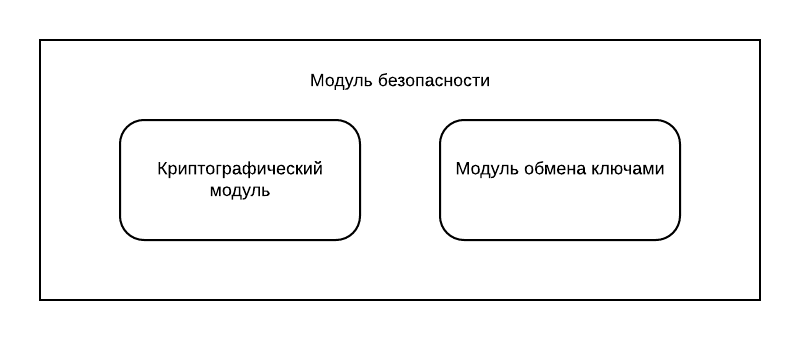


Рис. 15: (Структура модуля безопасности)

Модуль безопасности состоит из следующих подмодулей:

1. Модуль обмена ключами – предназначен для получения на обоих концах публичного канала закрытого ключа таким образом, чтобы через публичный канал его невозможно было перехватить. Данный модуль предназначен для получение сессионных ключей.
2. Криптографический модуль – после того как ключ был сгенерирован во время процедуры обмена ключами, все проходящие через канал данные шифруются и расшифровываются при помощи данного ключа.

### 2.3.1. Модуль обмена ключами

Процесс обмена ключами построен на синхронизации двух древовидных машин четности. При реализации алгоритма синхронизации я придерживался классического алгоритма, за исключением одного момента, касающегося передачи входных векторов.

В классическом алгоритме, на каждую итерацию обучения двух сетей необходимо генерировать общий вектор, затем передавать его второй машине. Это необходимо для получения одинаковых входных векторов для двух машин. Данная реализация плоха тем, что в отличие от выходных значений, которые представляют из себя всего лишь один бит информации, входные вектора имеют гораздо больший размер, который зависит от структуры нейронной сети. Передача входных векторов для каждой итерации обучения может негативно повлиять на общее время синхронизации, так как во время каждой итерации необходимо передавать данные по сети.

В моей реализации, мной был использован псевдослучайный генератор, который в самом начале обмена ключами инициализируется общим начальным значением (сидом). Данное значение передается только один раз, что позволяет сэкономить трафик при обмене данными и время, потраченное на постоянную передачу пакетов с векторами. Модуль синхронизации имеет следующую структуру:

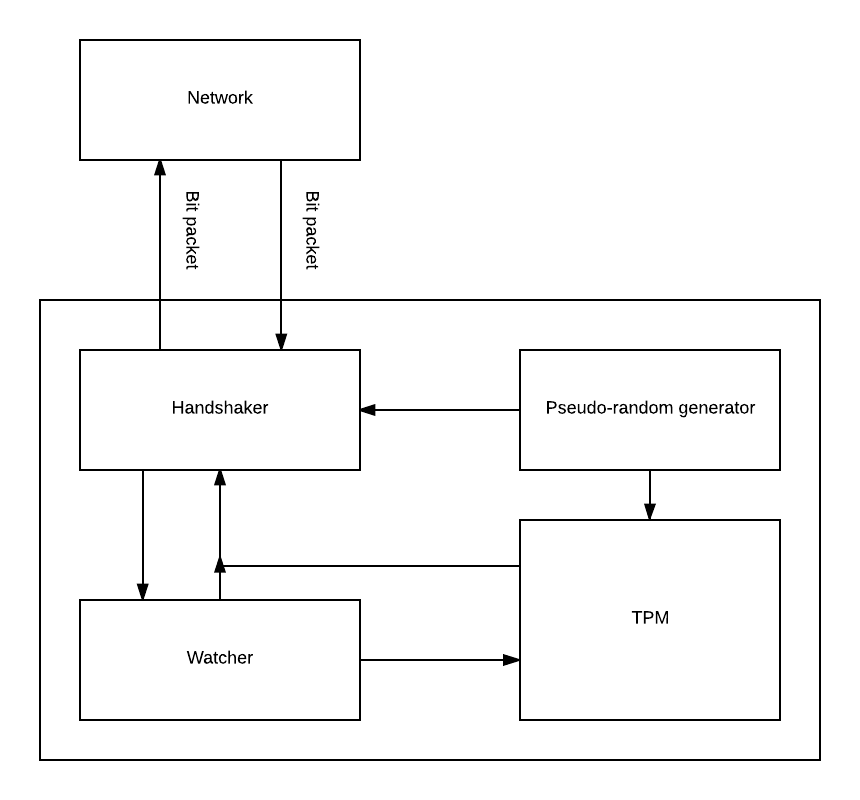


Рис. 16: (Структура модуля синхронизации)

Данный модуль состоит следующих элементов:

1. Древовидная машина четности – служит для реализации синхронизации между двумя абонентами.
2. Псевдослучайный генератор входных векторов – необходим для генерации псевдослучайных векторов, которые затем будут использоваться для обучения древовидных машин четности.
3. Модуль обмена «рукопожатиями» – осуществляет алгоритм обмена «рукопожатиями», во время которого устанавливается безопасное соединение между абонентами.
4. Наблюдатель – модуль, определяющий когда необходимо завершить процесс синхронизации.

### 2.3.2. Реализация древовидной машины четности

Древовидная машина четности является основным компонентом системы обмена ключами. Представляет собой программную реализацию многоуровневой нейронной сети и реализованные для нее алгоритмы обучения. В процессе реализации был реализован следующий пакет классов:

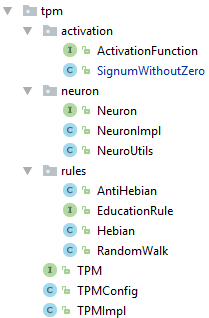
1. *ActivationFunction* – представляет собой интерфейс, который является абстракцией для обеспечения возможности использования разных функций активации.
2. *SignumWithoutZero* – класс реализующий функцию активацию signum, которая при значении 0 выдает -1.
3. *Neuron* – интерфейс, являющийся абстракцией нейрона, необходим для обеспечения возможности использования разных реализаций нейрона.

Рис. 17: (Структура пакета tpm)

1. *NeuronImpl* – класс, являющийся реализацией нейрона.
2. *NeuroUtils* – вспомогательный класс, реализующий часто используемые нейроном методы.
3. *EducationRule* – интерфейс, являющийся абстракцией для возможности реализации разных правил обучения.
4. *Hebian* – класс, являющийся реализацией EducationRule, который представляет из себя правило обучения по Хэббу.
5. *AntiHebian* – класс, являющийся реализацией EducationRule, который представляет из себя правило обучения по анти-Хэббу.
6. *RandomWalks* - класс, являющийся реализацией EducationRule, который представляет из себя правило обучения случайными блужданиями.
7. *TPM* – интерфейс, являющийся абстракцией древовидной машины четности.
8. *TPMImpl* – реализация древовидной машины четности.
9. *TPMConfig* – класс, содержащий настройки древовидной машины четности.

### 2.3.3 Процесс обучения

Основным классом, с которым будет работать пользователь модуля, будет *TPMImpl*. Данный класс содержит в себе список нейронов, которые инициализируются автоматически на основании параметров N, K, L, функция активации, которые передаются пользователем в машину при инициализации. Все параметры, необходимые для инициализации, инкапсулируются в класс *TPMConfig*.

Для того, чтобы начать обучение *TPMImpl*, необходимо совершить следующие действия:

1. Инициализировать *TPMImpl*, путем передачи в конструктор класса *TPMConfig*.
2. Необходимо, используя метод *initInputs(),* задать входной вектор для текущей итерации синхронизации.
3. Вызвать метод *processInputs(),* который вычисляет значение выходного нейрона, на основании входных векторов.
4. В случае, если выходное нейроны обоих машин совпали, вызвать метод *changeWhaights(double output)*, который принимает на вход выходное значение второй машины и производит изменение весовых коэффициентов, согласно правилу обучения.

### 2.3.4 Алгоритм обмена «рукопожатиями»

При подключении клиентского приложения к серверному осуществляется выполнение алгоритма обмена «рукопожатиями». Данный алгоритм представляет из себя определенный набор шагов, который необходимо пройти двум абонентам для достижения безопасного соединения. Данный алгоритм представлен на рисунке:

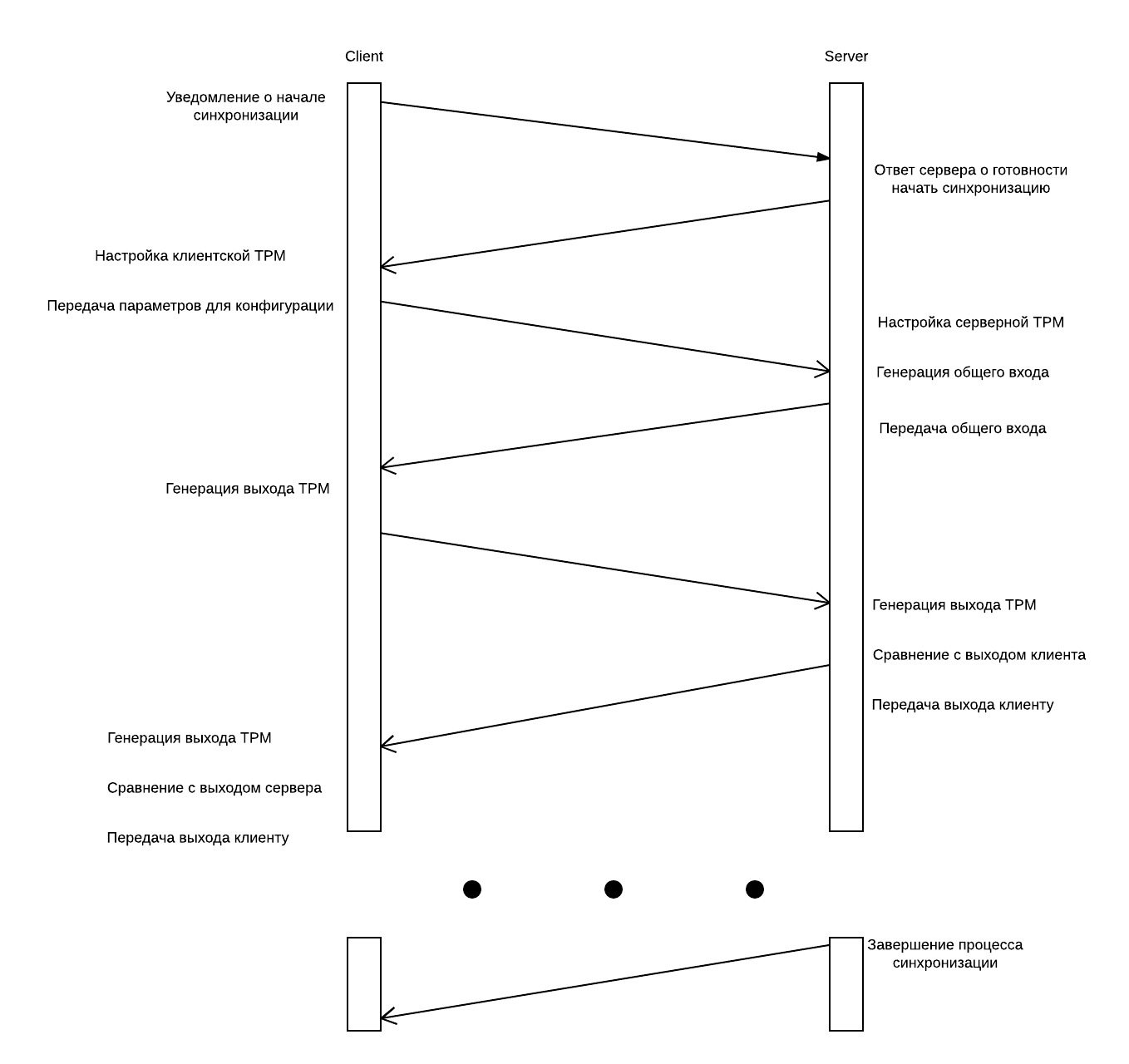


Рис. 18: (Схема алгоритма обмена рукопожатиями)

Алгорит обмена рукопожатиями состоит из следующих шагов:

1. Клиентская сторона инициирует начало установления безопасного соединения. Делается это путем отправки приветственного сообщения серверу. Данное сообщение не содержит каких либо особых данных, так как его основная цель заявить серверной стороне о начале соединения.
2. Получив приветственное сообщение от клиента, сервер начинает процесс соединения с данным клиентом, отправляя серверный пакет приветствия клиенту, чтобы дать понять клиенту, что он готов начинать синхронизацию.
3. После получения сообщения о готовности сервера, клиент начинает подготовку к процессу обмена ключами. На его стороне, модуль безопасности создает экземпляр древовидной машины четности. Параметры данной машины определяются клиентом. После формирования машины, необходимой для проведения обмена ключами, параметры древовидной машины четности отправляются на сервер, для настройки идентичной машины на другом конце канала.
4. Сервер, получив необходимые данные о настройке машины на своей стороне, настраивает идентичную машину. Далее, сервер генерирует общее инициализирующее значение для генератора псевдослучайных векторов. После этого, создается экземпляр генератора псевдослучайных векторов на стороне сервера, а его сид передается клиенту для настройки такого же генератора на своей стороне.
5. Клиент, получив данные о настройке генератора псевдослучайных векторов, конфигурирует его на своей стороне, таким образом, на двух концах канала обмена данными, мы получили два идентичных псевдослучайных генератора, которые будут генерировать идентичные входные вектора, которые необходимы для последующей синхронизации.
6. На данном шаге обе стороны имеют сконфигурированные машины и готовы начать синхронизацию. На данном шаге, клиент генерирует выходное значение, подавая на вход машине сгенерированный псевдослучайный вектор. Далее это значение отправляется на сервер.
7. Сервер получает выходное значение от клиента, после чего генерирует свое выходное значение. Если они равны, происходит изменение весов серверной машины согласно выбранному правилу обучения. Выходное значение серверной машины передается на клиентскую.
8. Клиентская машина получает выходное значение серверной машины, если данное значение равно тому, что было вычислено на шаге 6, то происходит изменение весов серверной машины согласно выбраному правилу обучения.
9. Далее повторяем шаги 6-8 до полной синхронизации машин.
10. После того, как синхронизация была достигнута, сервер отправляет клиенту специальный пакет, который утверждает текущий ключ и означает, что с этого момента, значения весов древовидной машины четности будут использоваться как ключ.

В результате выполнения данного алгоритма мы получаем две синхронизированные древовидные машины четности, которые располагаются на разных концах канала связи, далее мы сможем использовать веса данных машин, которые являются одинаковыми, как общий закрытый ключ. Кроме того, во время выполнения алгоритма, мы не передаем никакой информации, которая могла бы скомпраметировать данные ключи.

### 2.3.5 Реализация алгоритма обмена «рукопожатиями»

Алгоритм обмена рукопожатиями в програмной реализации представлен пакетом классов handshake. Данный пакет имеет следующую структуру:

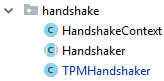
1. Абстрактный класс *Handshaker* – представляет собой абстракцию модуля обмена рукопожатиями.

Рис. 1: (Структура пакета handshake)

Рис. 19: (Структура пакета handshake)

1. Класс *TPMHandshaker* – представляет собой реализацию алгоритма обмена «рукопожатиями», которая основана на синхронизации двух древовидных машин четности. На основе текущего состояния и входного пакета данных определяет следующий шаг алгоритма и генерирует пает с данными для отправки второй машине.
2. Класс *HandshakeContext* – предоставляет собой контекст, для хранения информации о процессе синхронизации.

В данной реализации, основную работу выполняет класс *TPMHandler*. Данный класс содержит метод *handleHandshake(Packet packet)*, который отвечает за обработку пакетов. После передачи пакета, пришедшего через сеть, данный класс генерирует ответ и посылает его второй машине. Содержание отправляемого пакета зависит от текущего шага алгоритма.

### 2.3.6. Псевдослучайный генератор

Псевдослучайный генератор – алгоритм для генерации последовательности псевдослучайных чисел, чьи параметры близки к последовательности случайных чисел. Но последовательность не является в полной мере случайной, потому она в полной мере определяется начальным значением, которое называется сидом.

Одним из типов генераторов является конгруэнтный линейный генератор. Данный генератор получает последующие числа следующим образом:

где  это последовательность псевдослучайных значений и

m - модуль,

- множитель,

- инкремент,

- сид или стартовое значение.

Все перечисленные параметры являются целочисленными константами и определяют генератор.

### 2.3.7 Генератор входных векторов

Для процедуры синхронизации, на каждой итерации, необходимо генерировать входные вектора, которые представляют собой вектора, элементы которых сгенерированы псевдослучайным генератором. В программной реализации данный генератор представлен следующим пакетом:

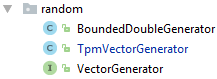
1. Класс *BoundedDoubleGenerator* – является наследником, стандартного в языке Java класса, который позволяет генерировать случайные числа в заданном диапазоне.

Рис. 20: (Структура пакета random)

1. Интерфейс *VectorGenerator* – является абстракцией для генератора случайных векторов.
2. Класс *TpmVectorGenerator* – является реализацией генератора случайных векторов непосредственно для древовидной машины четности. Данный класс использует для инициализации параметры нейронной сети и генерирует случайные векторы для нужного количества нейронов и входов.

Перед началом использования, данный класс необходимо инициализировать начальным значением, которое должно быть общим для двух синхронизируемых машин. Таким образом, все последующие вектора будут генерироваться одинаковыми, даже несмотря на тот факт, что генераторы никак не взаимодействуют друг с другом.

### 2.3.8. Модуль «наблюдатель»

При синхронизации двух удаленных древовидных машин чётности возникает вопрос о том, когда можно считать процесс синхронизации двух машин законченным. Если при синхронизации двух сетей на одном компьютере можно просто сравнить их веса и, в случае их равенства, посчитать процесс завершенным, то при передаче данных по публичному каналу мы не можем передавать веса синхронизируемых нейронных сетей, так как они должны держаться в секрете. Таким образом, нам необходим критерий, по которому можно будет определить, что веса двух машин синхронизировались Данным параметром является количество итераций и количество выходов обеих нейронных сетей, которые совпали. Данные параметры вычисляются для определенной конфигурации древовидной машины четности, затем данная конфигурация заносится в список поддерживаемых сервером и для конкретной реализации задается количество итераций, необходимых для синхронизации, а также количество совпавших выходов. В случае, если клиент пытается начать синхронизацию с неподдерживаемой конфигурацией, будет возвращена ошибка синхронизации. В случае если за заданное количество итераций, количество совпавших выходов превысило заданное, то синхронизацию можно считать успешной. Ключ утверждается и в дальнейшем будет использоваться для шифрования сообщений.

### 2.3.9 Криптографический модуль

Второй основной частью модуля безопасности является криптографический модуль. Данный модуль осуществляет шифрование выходной и дешифрование входной информации. Для этих операций используется ключ, который был сгенерирован путем выполнения алгоритма обмена ключами в модуле обмена ключами. Шифрование осуществляется симметричным блочным шифром. Шифр используемый в данном модуле – AES.

# Глава 3. Эксперименты с программой

## 3.1. Измерение времени синхронизации при изменении L

В данном эксперименте будут проведены измерения скорости синхронизации двух древовидных машин четности при изменении параметра L (диапазон значений весов). Для данного эксперимента мы имеем две древовидных машины четности. Параметры N и K для данного эксперимента будут зафиксированы и принимать значения K = 3 и N = 1000. На следующем графике можно увидеть зависимость количества итераций, необходимых для синхронизации от параметра L.

Рис. 21: (График изменения количества итераций синхронизации в зависимости от L)

Как можно заметить из графика, время синхронизации возрастает нелинейно. При увеличении L, скорость роста времени синхронизации постоянно увеличивается. Так как, данные параметры будут использоваться для протокола обмена ключами, то нам необходимо быстродействие. Для определения изменения быстродействия были сделаны замеры времени при тех же параметрах и представлены следующим графиком:

Рис. 22: (График изменения времени синхронизации в зависимости от L)

На данном графике можно заметить, что график изменения времени синхронизации практически полностью соответствует графику изменения времени синхронизации.

Таким образом можно сделать вывод, что при увеличении параметра L в большую сторону, время синхронизации существенно возрастает. Так как для протокола важно быстродействие, то не желательно, чтобы время процесса синхронизации превышало 1000 мс. Поэтому при выполнении процесса синхронизации не стоит брать значения L, превышающие 10.

## 3.2. Измерение времени синхронизации при изменении K

В данном эксперименте будут проведены измерения скорости синхронизации двух древовидных машин четности при изменении параметра K (количество нейронов). Для данного эксперимента мы имеем две древовидных машины четности. Параметры N и L для данного эксперимента будут зафиксированы и принимать значения L = 3 и N = 1000. На следующем графике можно увидеть зависимость количества итераций, необходимых для синхронизации от параметра K.

Рис. 23: (График изменения количества итераций синхронизации в зависимости от L)

На данном графике можно заметить, что при изменении количество нейронов в древовидной машине четности, скорость синхронизации увеличивается быстрее, чем при изменении диапазона значения, таким образом можно сказать, что количество нейронов в древовидной машине четности влияет на общее время синхронизации сильнее чем диапазон значений.

Как и в случае с предыдущим параметром, произведем замеры изменения времени синхронизации, чтобы выбрать оптимальный по времени синхронизации параметр.

Рис. 24: (График изменения времени синхронизации в зависимости от L)

Из графика видно, что время синхронизации возрастает быстрее при изменении параметра K, чем при изменении параметра L. Эта разница обусловлена разницей в количестве итераций между двумя параметрами, а также увеличением времени, которое необходимо на произведение одной итерации.

Принимая во внимания ограничения, необходимые для быстродействия работы протокола, оптимальное значение для данного параметра не должно превышать 5.

## 3.3. Измерение времени синхронизации при изменении N

В данном эксперименте будут проведены измерения скорости синхронизации двух древовидных машин четности при изменении параметра N (количество входов каждого нейрона). Для данного эксперимента мы имеем две древовидных машины четности. Параметры K и L для данного эксперимента будут зафиксированы и принимать значения L = 3 и K = 3. На следующем графике можно увидеть зависимость количества итераций, необходимых для синхронизации от параметра N.

Рис. 25: (График изменения количества итераций синхронизации в зависимости от N)

Как можно заметить из графика, параметр N, оказывает наименьшее влияние на колличество итераций, необходимых для синхронизации двух древовидных машин четности, из всех трех параметров. Кроме того, можно заметить большой разброс значений при разных колличествах входов.

Далее произведем замеры времени синхронизации, как и в случае с предыдущими параметрами:

Рис. 26: (График изменения времени синхронизации в зависимости от N)

Как видно из графика, увеличение времени синхронизации при изменении параметра N является незначительным и во много раз меньше, чем для предыдущих параметров. Таким образом для данного параметра можно выставлять достаточно большие значения и это не будет оказывать существенного влияния на быстродействие системы.

## 3.4. Пример рабочего приложения

На базе описанного выше протокола было разработано приложение для обмена мнгновенными сообщениями. Данное приложение имеет окно для ввода текста, параметры подключения, параметры древовидной машины четности, которые пользователь может задать при подключении к серверу.

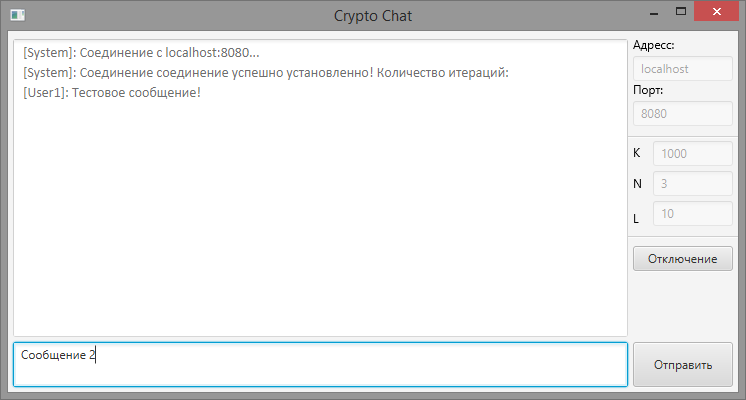


Рис. 27: (Окно программы)

# Заключение

В результате данной работы были исследованы базовые принципы работы древовидных машин четности и их синхронизации.

Была проведена серия экспериментов, для определения влияния параметров древовидной машины четности на время синхронизации.

Был реализован программный комплекс, позволяющий осуществлять безопасную передачу сообщений между абонентами. Данный протокол был реализован на основе явления нейросинхронизации. При реализации процедуры обмена был использован псевдослучайный генератор, который позволил сэкономить трафик и время при передачи входного вектора на каждой итерации.

На основе протокола безопасной передачи данных была построена программа для безопасной передачи сообщений.

Таким образом можно сделать вывод, что на явлении нейросинхронизации возможно реализовать аналог существующих методов обмена ключами.

# Список литературы

1. Andreas Ruttor. «Neural Synchronization and Cryptography» (2006)
2. M. Volkmer, S. Wallner. «Tree parity machine rekeying architectures» (2004).
3. M. Volkmer, A. Schaumburg. «Authenticated tree parity machine key exchange» (2004).
4. Alan M. F. Souza, Fabio M. Soares. «Neural Network Programming with Java» (2016).
5. A. Pikovsky, M. Rosenblum, J. Kurths. «Synchronization». (2001).
6. C.-M. Kim, S. Rim, W.-H. Kye. «Sequential synchronization of chaotic systems with an application to communication» (2002).
7. K. M. Cuomo, A. V. Oppenheim. «Circuit implementation of synchronized chaos with applications to communications»(1993).
8. L. M. Pecora and T. L. Carroll. «Synchronization in chaotic systems» (1990).
9. R. Metzler, W. Kinzel, I. Kanter. «Interacting neural networks». (2000).
10. W. Kinzel, R. Metzler, I. Kanter. «Dynamics of interacting neural networks». (2000).

# Приложение

### Листинг программы

Класс PackageCodec.java

public class PackageCodec extends ByteToMessageCodec<Packet> {

@Override

protected void encode(ChannelHandlerContext ctx, Packet o, ByteBuf byteBuf) throws Exception {

Packet packet = o;

PackageWriter.write(packet, byteBuf);

}

@Override

protected void decode(ChannelHandlerContext channelHandlerContext, ByteBuf byteBuf, List list) throws Exception {

int length = 0;

if(byteBuf.readableBytes() > Integer.SIZE / 8 ) {

length = byteBuf.readInt();

if (byteBuf.readableBytes() >= length) {

System.out.println("readable: " + byteBuf.readableBytes());

System.out.println("length: " + length);

Packet packet = PackageReader.read(byteBuf);

list.add(packet);

}

}

}

}

Класс PackageToMessagesCodec.java

public class PackageToMessagesCodec extends MessageToMessageCodec<Packet, String>{

@Override

protected void encode(ChannelHandlerContext channelHandlerContext, String message, List<Object> list) throws Exception {

ByteData packet = new ByteData();

String s = new String();

packet.setContent(message.getBytes(Charset.forName("UTF-8" )));

list.add(packet);

}

@Override

protected void decode(ChannelHandlerContext channelHandlerContext, Packet packet, List<Object> list) throws Exception {

ByteData bytePacket = (ByteData) packet;

String v = new String(bytePacket.getContent(), Charset.forName("UTF-8") );

System.out.println(new String(((ByteData)packet).getContent(), Charset.forName("UTF-8" )));

list.add(v);

}

}

Класс HandshakeContext.java

public class HandshakeContext {

private PackageType state;

private boolean serverMode;

private TPM tpm;

private double[][] lastInput;

private int numberOfSynch;

private TpmVectorGenerator generator;

private boolean handshakeDone;

private boolean handshakeInProgress;

public void setHandshakeInprogress() {

if(!isHandshakeInProgress()) {

handshakeInProgress = true;

}

}

public void setDoneHandshake() {

if(isHandshakeInProgress()) {

handshakeInProgress = false;

handshakeDone = true;

}

}

public PackageType getState() {

return state;

}

public void setState(PackageType state) {

this.state = state;

}

public boolean isServerMode() {

return serverMode;

}

public void setServerMode(boolean serverMode) {

this.serverMode = serverMode;

}

public TPM getTpm() {

return tpm;

}

public void setTpm(TPM tpm) {

this.tpm = tpm;

}

public double[][] getLastInput() {

return lastInput;

}

public void setLastInput(double[][] lastInput) {

this.lastInput = lastInput;

}

public int getNumberOfSynch() {

return numberOfSynch;

}

public void setNumberOfSynch(int numberOfSynch) {

this.numberOfSynch = numberOfSynch;

}

public TpmVectorGenerator getGenerator() {

return generator;

}

public void setGenerator(TpmVectorGenerator generator) {

this.generator = generator;

}

public boolean isHandshakeDone() {

return handshakeDone;

}

public boolean isHandshakeInProgress() {

return handshakeInProgress;

}

}

Класс Handshaker.java

public abstract class Handshaker {

public ClientHello startHandshake() {

getContext().setHandshakeInprogress();

ClientHello content = new ClientHello();

return content;

}

public HandshakeContent handleHandshake(HandshakeContent content) {

HandshakeContent result = null;

switch (content.getType()) {

case SERVER\_HELLO:

result = handleServerHello((ServerHello) content);

break;

case CLIENT\_HELLO:

getContext().setHandshakeInprogress();

result = handleClientHello((ClientHello) content);

break;

case TPM\_CONFIG:

result = handleTpmConfig((TpmConfigPacket) content);

break;

case INPUT\_SEND:

result = handleSendInput((SendInput) content);

break;

case SERVER\_SYNCH:

if (getContext().getNumberOfSynch() < 4000) {

result = handleServerSynch((ServerSynch) content);

} else {

System.out.println("SUCCESS!!!!");

System.out.println(((TPMImpl)getContext().getTpm()).getNeuronList());

}

break;

case CLIENT\_SYNCH:

if (getContext().getNumberOfSynch() < 4000) {

result = handleClientSynch((ClientSynch) content);

} else {

System.out.println("SUCCESS!!!!");

System.out.println(((TPMImpl)getContext().getTpm()).getNeuronList());

result = new KeyCommit();

getContext().setDoneHandshake();

}

break;

case KEY\_COMMIT:

getContext().setDoneHandshake();

break;

default:

throw new IllegalStateException("Unknown State");

}

return result;

}

abstract TpmConfigPacket handleServerHello(ServerHello content);

abstract ServerHello handleClientHello(ClientHello content);

abstract SendInput handleTpmConfig(TpmConfigPacket content);

abstract ServerSynch handleClientSynch(ClientSynch content);

abstract ClientSynch handleServerSynch(ServerSynch content);

abstract ClientSynch handleSendInput(SendInput content);

abstract KeyCommit handleHeyCommit(HandshakeContent content);

abstract HandshakeContext getContext();

abstract void setContext(HandshakeContext context);

}

Класс TPMHandshaker.java

public class TPMHandshaker extends Handshaker {

private HandshakeContext context = new HandshakeContext();

private int numberOfInputs = 12;

private int numberOfNeuron = 8;

private double weightRange = 5.;

public TPMHandshaker(HandshakeContext context) {

this.context = context;

}

@Override

public ServerHello handleClientHello(ClientHello content) {

getContext().setState(PackageType.SERVER\_HELLO);

ServerHello serverHello = new ServerHello();

return serverHello;

}

@Override

public TpmConfigPacket handleServerHello(ServerHello content) {

TPMConfig config = new TPMConfig();

config.setNumberOfInputs(numberOfInputs);

config.setFunction(new SignumWithoutZero());

config.setNumberOfNeurons(numberOfNeuron);

config.setWeightRange(weightRange);

Hebian rule = new Hebian();

rule.setWeightsRange(weightRange);

config.setRule(rule);

TPM tpm = new TPMImpl(config);

getContext().setTpm(tpm);

TpmVectorGenerator random = new TpmVectorGenerator(config.getNumberOfNeurons(),config.getNumberOfInputs(),config.getWeightRange());

getContext().setGenerator(random);

return new TpmConfigPacket(config);

}

@Override

public SendInput handleTpmConfig(TpmConfigPacket content) {

TpmVectorGenerator random = new TpmVectorGenerator(content.getNumberOfNeurons(),content.getNumberOfInputs(),content.getWeightsValues());

Random rand = new Random();

long seed = rand.nextLong();

random.setSeed(seed);

getContext().setGenerator(random);

TPMConfig config = new TPMConfig();

config.setNumberOfInputs(content.getNumberOfInputs());

config.setFunction(new SignumWithoutZero());

config.setNumberOfNeurons(content.getNumberOfNeurons());

config.setWeightRange(content.getWeightsValues());

Hebian rule = new Hebian();

rule.setWeightsRange(content.getWeightsValues());

config.setRule(rule);

TPM tpm = new TPMImpl(config);

getContext().setTpm(tpm);

return new SendInput(seed);

}

@Override

public ServerSynch handleClientSynch(ClientSynch content) {

TpmVectorGenerator generator = getContext().getGenerator();

double[][] input = generator.getNextVector();

TPM tpm = getContext().getTpm();

tpm.initInputs(input);

double output = tpm.processInput();

System.out.println("-----------------------------");

double remoteOutput = content.getOutput();

System.out.println("remote:" + remoteOutput);

System.out.println("this:" + output);

getContext().getTpm().changeWeights(remoteOutput);

System.out.println("S");

getContext().setNumberOfSynch(getContext().getNumberOfSynch() + 1);

System.out.println(getContext().getNumberOfSynch());

for(int i = 0; i < input.length; i++) {

for(int j = 0; j < input[i].length; j++) {

System.out.print(input[i][j] + " ");

}

System.out.println();

}

System.out.println(((TPMImpl)getContext().getTpm()).getNeuronList());

return new ServerSynch(output);

}

@Override

public ClientSynch handleServerSynch(ServerSynch content) {

TpmVectorGenerator generator = getContext().getGenerator();

double[][] input = generator.getNextVector();

TPM tpm = getContext().getTpm();

System.out.println("-----------------------------");

double remoteOutput = content.getOutput();

System.out.println("remote:" + remoteOutput);

System.out.println("this:" + ((TPMImpl) getContext().getTpm()).getOutput());

getContext().getTpm().changeWeights(remoteOutput);

System.out.println("S");

getContext().setNumberOfSynch(getContext().getNumberOfSynch() + 1);

System.out.println(getContext().getNumberOfSynch());

System.out.println(((TPMImpl)getContext().getTpm()).getNeuronList());

tpm.initInputs(input);

double output = tpm.processInput();

System.out.println("++++++++++++++++++++++++++");

for(int i = 0; i < input.length; i++) {

for(int j = 0; j < input[i].length; j++) {

System.out.print(input[i][j] + " ");

}

System.out.println();

}

System.out.println(((TPMImpl)getContext().getTpm()).getNeuronList());

return new ClientSynch(output);

}

@Override

public ClientSynch handleSendInput(SendInput content) {

long seed = content.getSeed();

TpmVectorGenerator generator = getContext().getGenerator();

generator.setSeed(seed);

double[][] input = generator.getNextVector();

TPM tpm = getContext().getTpm();

tpm.initInputs(input);

double output = tpm.processInput();

return new ClientSynch(output);

}

@Override

public KeyCommit handleHeyCommit(HandshakeContent content) {

System.out.println("SUCCESS!!!");

return null;

}

public HandshakeContext getContext() {

return context;

}

public void setContext(HandshakeContext context) {

this.context = context;

}

}

Класс ByteData.java

public class ByteData extends DataContent {

private byte[] content;

public ByteData() {

setType(PackageType.BYTE\_DATA);

}

@Override

public void toBytes(ByteBuf buf) {

super.toBytes(buf);

buf.writeShort(content.length);

for(Byte b : content) {

buf.writeByte(b);

}

}

@Override

public void fromBytes(ByteBuf buf) {

super.fromBytes(buf);

Short length = buf.readShort();

content = new byte[length];

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

content[i] = buf.readByte();

}

}

@Override

public int getSize() {

return super.getSize() + (Byte.SIZE \* content.length + Short.SIZE) / 8;

}

public byte[] getContent() {

return content;

}

public void setContent(byte[] content) {

this.content = content;

}

}

public abstract class DataContent extends ContentPackage {

private boolean encrypted = false;

@Override

public void toBytes(ByteBuf buf) {

super.toBytes(buf);

buf.writeBoolean(encrypted);

}

@Override

public void fromBytes(ByteBuf buf) {

super.fromBytes(buf);

encrypted = buf.readBoolean();

}

@Override

public int getSize() {

return super.getSize() + Byte.SIZE / 8;

}

public boolean isEncrypted() {

return encrypted;

}

public void setEncrypted(boolean encrypted) {

this.encrypted = encrypted;

}

}

Класс ClientHello.java

public class ClientHello extends HandshakeContent {

public ClientHello() {

setType(PackageType.CLIENT\_HELLO);

}

}

Класс ClientSynch.java

public class ClientSynch extends Synch{

public ClientSynch() {

setType(PackageType.CLIENT\_SYNCH);

}

public ClientSynch(double output) {

super(output);

setType(PackageType.CLIENT\_SYNCH);

}

}

Класс KeyCommit.java

public class KeyCommit extends HandshakeContent {

public KeyCommit() {

setType(PackageType.KEY\_COMMIT);

}

}

Класс SendInput.java

public class SendInput extends HandshakeContent {

private long seed;

public SendInput() {

setType(PackageType.INPUT\_SEND);

}

public SendInput(ByteBuf buf) {

fromBytes(buf);

}

public SendInput(long seed) {

setType(PackageType.INPUT\_SEND);

this.seed = seed;

}

@Override

public void toBytes(ByteBuf buf) {

super.toBytes(buf);

buf.writeLong(seed);

}

@Override

public void fromBytes(ByteBuf buf) {

super.fromBytes(buf);

seed = buf.readLong();

}

@Override

public int getSize() {

return super.getSize() + Long.SIZE / 8;

}

public long getSeed() {

return seed;

}

}

Класс ServerHello.java

public class ServerHello extends HandshakeContent {

public ServerHello() {

setType(PackageType.SERVER\_HELLO);

}

}

Класс ServerSynch.java

public class ServerSynch extends Synch{

public ServerSynch() {

setType(PackageType.SERVER\_SYNCH);

}

public ServerSynch(double output) {

super(output);

setType(PackageType.SERVER\_SYNCH);

}

}

Класс Synch.java

public abstract class Synch extends HandshakeContent {

private Byte output;

public Synch() {

}

public Synch(double output) {

this.output = (byte)(output == -1. ? 0 : 1);

}

@Override

public void toBytes(ByteBuf buf) {

super.toBytes(buf);

buf.writeByte(output);

}

@Override

public void fromBytes(ByteBuf buf) {

super.fromBytes(buf);

output = buf.readByte();

}

@Override

public int getSize() {

return super.getSize() + output.SIZE / 8;

}

public double getOutput() {

return output == 0 ? -1. : 1.;

}

}

Класс TpmConfigPacket.java

public class TpmConfigPacket extends HandshakeContent {

private Integer numberOfNeurons;

private Integer numberOfInputs;

private Double weightsValues;

public TpmConfigPacket() {

setType(PackageType.TPM\_CONFIG);

}

public TpmConfigPacket(TPMConfig config) {

setType(PackageType.TPM\_CONFIG);

this.numberOfNeurons = config.getNumberOfNeurons();

this.numberOfInputs = config.getNumberOfInputs();

this.weightsValues = config.getWeightRange();

}

public TpmConfigPacket(Integer numberOfNeurons, Integer numberOfInputs, Double weightsValues) {

this.numberOfNeurons = numberOfNeurons;

this.numberOfInputs = numberOfInputs;

this.weightsValues = weightsValues;

}

@Override

public void toBytes(ByteBuf buf) {

super.toBytes(buf);

buf.writeInt(numberOfNeurons);

buf.writeInt(numberOfInputs);

buf.writeDouble(weightsValues);

}

@Override

public void fromBytes(ByteBuf buf) {

super.fromBytes(buf);

numberOfNeurons = buf.readInt();

numberOfInputs = buf.readInt();

weightsValues = buf.readDouble();

}

@Override

public int getSize() {

return super.getSize() + (numberOfNeurons.SIZE + numberOfInputs.SIZE + weightsValues.SIZE) / 8;

}

public Integer getNumberOfNeurons() {

return numberOfNeurons;

}

public Integer getNumberOfInputs() {

return numberOfInputs;

}

public Double getWeightsValues() {

return weightsValues;

}

}

Класс HandshakeContent.java

public abstract class HandshakeContent extends ContentPackage implements Bytable {

private boolean firstMessage = false;

@Override

public void toBytes(ByteBuf buf) {

super.toBytes(buf);

buf.writeBoolean(firstMessage);

}

@Override

public void fromBytes(ByteBuf buf) {

super.fromBytes(buf);

firstMessage = buf.readBoolean();

}

@Override

public int getSize() {

return super.getSize() + Byte.SIZE / 8;

}

public boolean isFirstMessage() {

return firstMessage;

}

public void setFirstMessage(boolean firstMessage) {

this.firstMessage = firstMessage;

}

}

Класс ContentPackage.java

public abstract class ContentPackage extends Packet{

private long sessionId = -1;

@Override

public void toBytes(ByteBuf buf) {

super.toBytes(buf);

buf.writeLong(sessionId);

}

@Override

public void fromBytes(ByteBuf buf) {

sessionId = buf.readLong();

}

@Override

public int getSize() {

return super.getSize() + Long.SIZE / 8;

}

}

Класс PackageType.java

public enum PackageType {

CLIENT\_HELLO, SERVER\_HELLO, TPM\_CONFIG, INPUT\_SEND, CLIENT\_SYNCH, SERVER\_SYNCH, KEY\_COMMIT, BYTE\_DATA;

public static PackageType byNumber(int i) {

PackageType result;

switch (i) {

case 0:

result = CLIENT\_HELLO;

break;

case 1:

result = SERVER\_HELLO;

break;

case 2:

result = TPM\_CONFIG;

break;

case 3:

result = INPUT\_SEND;

break;

case 4:

result = CLIENT\_SYNCH;

break;

case 5:

result = SERVER\_SYNCH;

break;

case 6:

result = KEY\_COMMIT;

break;

case 7:

result = BYTE\_DATA;

break;

default:

throw new IllegalStateException("Unknown State");

}

return result;

}

}

Класс PackageReader.java

public class PackageReader {

public static Packet read(ByteBuf buf) {

PackageType type = PackageType.byNumber(buf.readShort());

Packet result;

switch (type) {

case SERVER\_HELLO:

result = new ServerHello();

break;

case CLIENT\_HELLO:

result = new ClientHello();

break;

case TPM\_CONFIG:

result = new TpmConfigPacket();

break;

case INPUT\_SEND:

result = new SendInput();

break;

case SERVER\_SYNCH:

result = new ServerSynch();

break;

case CLIENT\_SYNCH:

result = new ClientSynch();

break;

case KEY\_COMMIT:

result = new KeyCommit();

break;

case BYTE\_DATA:

result = new ByteData();

break;

default:

throw new IllegalStateException("Unknown State");

}

result.fromBytes(buf);

return result;

}

}

Класс PackageWriter.java

public class PackageWriter {

public static int write(Packet packet, ByteBuf buf) {

int size = 0;

PackageType type = packet.getType();

switch (type) {

case SERVER\_HELLO:

ServerHello shello = (ServerHello) packet;

buf.writeInt(shello.getSize());

shello.toBytes(buf);

break;

case CLIENT\_HELLO:

ClientHello hello = (ClientHello) packet;

buf.writeInt(hello.getSize());

hello.toBytes(buf);

break;

case TPM\_CONFIG:

TpmConfigPacket tpmConfigPacket = (TpmConfigPacket) packet;

buf.writeInt(tpmConfigPacket.getSize());

tpmConfigPacket.toBytes(buf);

break;

case INPUT\_SEND:

SendInput sendInput = (SendInput) packet;

buf.writeInt(sendInput.getSize());

sendInput.toBytes(buf);

break;

case CLIENT\_SYNCH:

ClientSynch ClientSynch = (ClientSynch) packet;

buf.writeInt(ClientSynch.getSize());

ClientSynch.toBytes(buf);

break;

case SERVER\_SYNCH:

ServerSynch ServerSynch = (ServerSynch) packet;

buf.writeInt(ServerSynch.getSize());

ServerSynch.toBytes(buf);

break;

case KEY\_COMMIT:

KeyCommit KeyCommit = (KeyCommit) packet;

buf.writeInt(KeyCommit.getSize());

KeyCommit.toBytes(buf);

break;

case BYTE\_DATA:

ByteData ByteData = (ByteData) packet;

buf.writeInt(ByteData.getSize());

ByteData.toBytes(buf);

break;

default:

throw new IllegalStateException("Unknown State");

}

return size;

}

}

Класс Bytable.java

public interface Bytable {

void toBytes(ByteBuf buf);

void fromBytes(ByteBuf buf);

int getSize();

}

Класс Packet.java

public abstract class Packet implements Bytable {

private PackageType type;

public PackageType getType() {

return type;

}

protected void setType(PackageType type) {

this.type = type;

}

@Override

public void toBytes(ByteBuf buf) {

buf.writeShort(type.ordinal());

}

@Override

public void fromBytes(ByteBuf buf) {

}

@Override

public int getSize() {

return Short.SIZE / 8;

}

}класс Message.java

public class Message {

private String name;

private String text;

public Message(String name, String text) {

this.name = name;

this.text = text;

}

public String getName() {

return name;

}

public String getText() {

return text;

}

}

Класс BoundedDoubleGenerator.java

public class BoundedDoubleGenerator extends Random {

public BoundedDoubleGenerator() {

}

public BoundedDoubleGenerator(long seed) {

super(seed);

}

public double nextDouble(double leftLimit, double rightLimit) {

double random = super.nextDouble();

double result = leftLimit + (random \* (rightLimit - leftLimit));

return result;

}

}

Класс TpmVectorGenerator.java

public class TpmVectorGenerator extends BoundedDoubleGenerator implements VectorGenerator {

private int numberOfNeurons;

private int numberOfInputs;

private double weightsDiapason;

public TpmVectorGenerator(int numberOfNeurons, int numberOfInputs, double weightsDiapason) {

this.numberOfNeurons = numberOfNeurons;

this.numberOfInputs = numberOfInputs;

this.weightsDiapason = weightsDiapason;

}

public TpmVectorGenerator(long seed, int numberOfNeurons, int numberOfInputs, double weightsDiapason) {

super(seed);

this.numberOfNeurons = numberOfNeurons;

this.numberOfInputs = numberOfInputs;

this.weightsDiapason = weightsDiapason;

}

@Override

public double[][] getNextVector() {

double[][] result = new double[numberOfNeurons][numberOfInputs];

SignumWithoutZero signum = new SignumWithoutZero();

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

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

result[i][j] = signum.run(super.nextDouble(-1\*weightsDiapason, weightsDiapason));

}

}

return result;

}

}

Класс VectorGenerator.java

public interface VectorGenerator {

double [][] getNextVector();

}

Класс SecurityHandler.java

public class SecurityHandler extends MessageToMessageCodec<Packet,Packet> {

private Handshaker handshaker;

private boolean serverMode;

private HandshakeContext context;

private List<Packet> messagesBufer = new ArrayList<Packet>();

public SecurityHandler(boolean serverMode) {

this.serverMode = serverMode;

this.context = new HandshakeContext();

this.context.setServerMode(serverMode);

this.handshaker = new TPMHandshaker(context);

}

@Override

public void channelActive(ChannelHandlerContext ctx) throws Exception {

if(!serverMode) {

ctx.channel().writeAndFlush(handshaker.startHandshake());

}

}

@Override

protected void encode(ChannelHandlerContext channelHandlerContext, Packet packet, List<Object> list) throws Exception {

if((!context.isHandshakeDone() || context.isHandshakeInProgress()) && packet.getType().equals(PackageType.BYTE\_DATA)) {

messagesBufer.add(packet);

} else {

list.add(packet);

}

}

@Override

public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {

for(Packet p : messagesBufer) {

ctx.write(p);

System.out.println(((ByteData) p).getType());

}

ctx.flush();

}

@Override

protected void decode(ChannelHandlerContext ctx, Packet packet, List<Object> list) throws Exception {

if(!packet.getType().equals(PackageType.BYTE\_DATA) && !packet.getType().equals(PackageType.KEY\_COMMIT)) {

ctx.channel().writeAndFlush(handshaker.handleHandshake((HandshakeContent) packet));

} else if (packet.getType().equals(PackageType.KEY\_COMMIT)) {

System.out.println("SUCCESS!");

System.out.println(((TPMImpl)context.getTpm()).getNeuronList());

context.setDoneHandshake();

} else {

if(context.isHandshakeDone()) {

list.add(packet);

}

}

}

}

Класс ActivationFunction.java

public interface ActivationFunction {

Double run(Double input);

}

Класс SignumWithoutZero/java

public class SignumWithoutZero implements ActivationFunction {

public Double run(Double input) {

Signum signumFunction = new Signum();

return signumFunction.value(input) == 0 ? -1.0: signumFunction.value(input);

}

}

Класс Neuron.java

public interface Neuron {

void processInput();

void initInput(double[] input);

void changeInput();

RealVector getWeights();

double[] getInput();

Double getOutput();

void setWeights(RealVector weights);

void setFunction(ActivationFunction function);

void setInput(double[] input);

void setOutput(Double output);

}

Класс NeuronImpl.java

public class NeuronImpl implements Neuron {

private RealVector weights;

private double[] input;

private Double output;

private ActivationFunction function;

public void initInput(double[] input) {

this.input = input;

}

public NeuronImpl(RealVector weights, ActivationFunction function) {

this.weights = weights;

this.function = function;

}

public NeuronImpl(Double[] weights, ActivationFunction function) {

this.weights = new ArrayRealVector(weights);

this.function = function;

}

public void processInput() {

RealVector vector = new ArrayRealVector(input);

output = function.run(NeuroUtils.sum(vector.ebeMultiply(weights)));

}

public RealVector getWeights() {

return weights;

}

public void setWeights(RealVector weights) {

this.weights = weights;

}

public ActivationFunction getFunction() {

return function;

}

public void setFunction(ActivationFunction function) {

this.function = function;

}

public double[] getInput() {

return input;

}

public void setInput(double[] input) {

this.input = input;

}

public Double getOutput() {

return output;

}

public void setOutput(Double output) {

this.output = output;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

NeuronImpl neuron = (NeuronImpl) o;

return weights != null ? weights.equals(neuron.weights) : neuron.weights == null;

}

public void changeInput() {

double[] newInput = new double[input.length];

newInput[0] = output;

for(int i = 1; i < newInput.length; i++) {

newInput[i] = input[i-1];

}

this.input = newInput;

}

@Override

public int hashCode() {

return weights != null ? weights.hashCode() : 0;

}

@Override

public String toString() {

return "{" + weights +'}';

}

}

Класс NeuroUtils.java

public class NeuroUtils {

public static Double sum(RealVector vector) {

Double sum = 0.;

for(Double d: vector.toArray()) {

sum += d;

}

return sum;

}

public static RealVector getRandomVector(int length, Double range) {

double [] result = new double[length];

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

result[i] = (int) (Math.random()\*2\*range) - range;

result[i] = result[i] ==0 ? 1: result[i];

}

return new ArrayRealVector(result);

}

public static double [] getRandomInput(int length, Double range) {

double [] result = new double[length];

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

result[i] = Math.random() > 0.5 ? 1 : -1;

}

return result;

}

}

Класс EducationRule.java

public interface EducationRule extends Serializable {

void changeWeights(TPM tpm, double remoteOutput);

}

Класс Hebian.java

public class Hebian implements EducationRule {

private Double educationRate;

private Double weightsRange;

public void changeWeights(TPM tpm, double remoteOutput) {

List<Neuron> neurons = ((TPMImpl) tpm).getNeuronList();

double tpmOutput = ((TPMImpl) tpm).getOutput();

for (Neuron n : neurons) {

RealVector input = new ArrayRealVector(n.getInput());

RealVector weights = n.getWeights();

double output = n.getOutput();

n.setWeights(normalize(weights.add(input.mapMultiply(output \* threshold(tpmOutput \* output) \* threshold(tpmOutput \* remoteOutput)))));

}

}

private double threshold(double input) {

double result = 0;

if(input > 0 ) {

result = 1;

} else {

result = 0;

}

return result;

}

private RealVector normalize(RealVector vector) {

double[] array = vector.toArray();

if (vector.getMaxValue() > weightsRange || vector.getMinValue() < -1 \* weightsRange) {

for (int i = 0; i < array.length; i++) {

if (array[i] > weightsRange) {

array[i] = weightsRange;

}

if (array[i] < -1 \* weightsRange) {

array[i] = -weightsRange;

}

}

}

return new ArrayRealVector(array);

}

public Double getWeightsRange() {

return weightsRange;

}

public void setWeightsRange(double weightsRange) {

this.weightsRange = weightsRange;

}

}

Класс TPM.java

public interface TPM {

void initInputs(double[][] inputs);

void changeInputs();

Double processInput();

void changeWeights(double remoteOutput);

double[][] getInputs();

}

Класс TPMConfig.java

public class TPMConfig implements Serializable{

private int numberOfNeurons;

private int numberOfInputs;

private Double weightRange;

private ActivationFunction function;

private int maxErrorNumber;

private int minSuccessNumber;

private EducationRule rule;

public int getNumberOfNeurons() {

return numberOfNeurons;

}

public void setNumberOfNeurons(int numberOfNeurons) {

this.numberOfNeurons = numberOfNeurons;

}

public int getNumberOfInputs() {

return numberOfInputs;

}

public void setNumberOfInputs(int numberOfInputs) {

this.numberOfInputs = numberOfInputs;

}

public Double getWeightRange() {

return weightRange;

}

public void setWeightRange(double weightRange) {

this.weightRange = weightRange;

}

public ActivationFunction getFunction() {

return function;

}

public void setFunction(ActivationFunction function) {

this.function = function;

}

public int getMaxErrorNumber() {

return maxErrorNumber;

}

public void setMaxErrorNumber(int maxErrorNumber) {

this.maxErrorNumber = maxErrorNumber;

}

public int getMinSuccessNumber() {

return minSuccessNumber;

}

public void setMinSuccessNumber(int minSuccessNumber) {

this.minSuccessNumber = minSuccessNumber;

}

public EducationRule getRule() {

return rule;

}

public void setRule(EducationRule rule) {

this.rule = rule;

}

}

Класс TPMImpl.java

public class TPMImpl implements TPM {

private List<Neuron> neuronList = new ArrayList<Neuron>();

private TPMConfig config;

private double output;

public TPMImpl() {

}

public TPMImpl(TPMConfig config) {

this.config = config;

init();

}

private void init() {

for(int i =0; i < config.getNumberOfNeurons(); i++) {

neuronList.add(new NeuronImpl(NeuroUtils.getRandomVector(config.getNumberOfInputs(), config.getWeightRange()),config.getFunction()));

}

}

public void initInputs(double[][] inputs) {

if(inputs.length != neuronList.size()) {

throw new NumberFormatException();

}

for(int i = 0; i < neuronList.size(); i++) {

neuronList.get(i).initInput(inputs[i]);

}

}

public Double processInput() {

Double product = 1.;

for(int i =0; i < neuronList.size(); i++) {

neuronList.get(i).processInput();

product \*= neuronList.get(i).getOutput();

}

setOutput(product);

return product;

}

public void changeWeights(double remoteOutput) {

config.getRule().changeWeights(this, remoteOutput);

}

public Double[][] getWeights() {

Double[][] weights = new Double[config.getNumberOfNeurons()][config.getNumberOfInputs()];

for(int i = 0; i < config.getNumberOfNeurons(); i++) {

for(int j = 0; j < config.getNumberOfInputs(); j++) {

weights[i][j] = neuronList.get(i).getWeights().toArray()[j];

}

}

return weights;

}

public List<Neuron> getNeuronList() {

return neuronList;

}

public void setNeuronList(List<Neuron> neuronList) {

this.neuronList = neuronList;

}

public TPMConfig getConfig() {

return config;

}

public void setConfig(TPMConfig config) {

this.config = config;

}

public double getOutput() {

return output;

}

public void setOutput(double output) {

this.output = output;

}

public void changeInputs() {

for(Neuron n: neuronList) {

n.changeInput();

}

}

public double[][] getInputs() {

double[][] inputs = new double[neuronList.size()][];

for(int i = 0; i < neuronList.size(); i++) {

inputs[i] = neuronList.get(i).getInput();

}

return inputs;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

TPMImpl tpm = (TPMImpl) o;

return neuronList != null ? neuronList.equals(tpm.neuronList) : tpm.neuronList == null;

}

@Override

public int hashCode() {

return neuronList != null ? neuronList.hashCode() : 0;

}

@Override

public String toString() {

return "TPMImpl{" +

"neuronList=" + neuronList +

'}';

}

}

Класс Controller.java

public class Controller {

@FXML

TextArea tfSend;

@FXML

TextArea tfReceive;

@FXML

TextField tfHost;

@FXML

TextField tfPort;

@FXML

Button btnConnect;

@FXML

Button btnSend;

@FXML

Button btnDisconnect;

private BooleanProperty connected = new SimpleBooleanProperty(false);

private StringProperty receivingMessageModel = new SimpleStringProperty("");

private Channel channel;

private EventLoopGroup group;

@FXML

public void initialize() {

btnConnect.disableProperty().bind( connected );

tfHost.disableProperty().bind( connected );

tfPort.disableProperty().bind( connected );

tfSend.disableProperty().bind( connected.not() );

btnSend.disableProperty().bind( connected.not() );

tfReceive.textProperty().bind(receivingMessageModel);

}

@FXML

public void connect() {

if( connected.get() ) {

return;

}

String host = tfHost.getText();

int port = Integer.parseInt(tfPort.getText());

group = new NioEventLoopGroup();

Task<Channel> task = new Task<Channel>() {

@Override

protected Channel call() throws Exception {

updateMessage("Bootstrapping");

updateProgress(0.1d, 1.0d);

Bootstrap b = new Bootstrap();

b

.group(group)

.channel(NioSocketChannel.class)

.remoteAddress( new InetSocketAddress(host, port) )

.handler( new ChatClientInitializer(receivingMessageModel));

ChannelFuture f = b.connect();

Channel chn = f.channel();

updateMessage("Connecting");

updateProgress(0.2d, 1.0d);

connected.set(true);

f.sync();

return chn;

}

@Override

protected void succeeded() {

channel = getValue();

connected.set(true);

}

@Override

protected void failed() {

Throwable exc = getException();

Alert alert = new Alert(Alert.AlertType.ERROR);

alert.setTitle("Client");

alert.setHeaderText( exc.getClass().getName() );

alert.setContentText( exc.getMessage() );

alert.showAndWait();

connected.set(false);

}

};

new Thread(task).start();

}

@FXML

public void disconnect() {

if( !connected.get() ) {

return;

}

Task<Void> task = new Task<Void>() {

@Override

protected Void call() throws Exception {

updateMessage("Disconnecting");

updateProgress(0.1d, 1.0d);

channel.close().sync();

updateMessage("Closing group");

updateProgress(0.5d, 1.0d);

group.shutdownGracefully().sync();

return null;

}

@Override

protected void succeeded() {

connected.set(false);

}

@Override

protected void failed() {

connected.set(false);

Throwable t = getException();

Alert alert = new Alert(Alert.AlertType.ERROR);

alert.setTitle("Client");

alert.setHeaderText( t.getClass().getName() );

alert.setContentText( t.getMessage() );

alert.showAndWait();

}

};

new Thread(task).start();

}

@FXML

public void send() {

if( !connected.get() ) {

return;

}

final String toSend = tfSend.getText();

Task<Void> task = new Task<Void>() {

@Override

protected Void call() throws Exception {

ChannelFuture f = channel.writeAndFlush(toSend);

f.sync();

return null;

}

@Override

protected void failed() {

Throwable exc = getException();

Alert alert = new Alert(Alert.AlertType.ERROR);

alert.setTitle("Client");

alert.setHeaderText( exc.getClass().getName() );

alert.setContentText( exc.getMessage() );

alert.showAndWait();

connected.set(false);

}

};

new Thread(task).start();

}

}

Класс Main.java

public class Main extends Application {

@Override

public void start(Stage primaryStage) throws Exception{

Parent root = FXMLLoader.load(getClass().getClassLoader().getResource("sample.fxml"));

primaryStage.setTitle("Crypto Chat");

primaryStage.setMinWidth(800);

primaryStage.setMinHeight(400);

primaryStage.setScene(new Scene(root, 300, 275));

primaryStage.show();

}

public static void main(String[] args) {

launch(args);

}

}

Класс ChatClientHandler.java

public class ChatClientHandler extends SimpleChannelInboundHandler<String> {

private final StringProperty receivingMessageModel;

public ChatClientHandler(StringProperty receivingMessageModel) {

this.receivingMessageModel = receivingMessageModel;

}

@Override

protected void channelRead0(ChannelHandlerContext arg0, String in) throws Exception {

Platform.runLater( () -> receivingMessageModel.set(receivingMessageModel.getValue() + in) );

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {

ctx.close();

}

}

Класс ChatClientInitializer.java

public class ChatClientInitializer extends ChannelInitializer<SocketChannel> {

private StringProperty receivingMessageModel;

public ChatClientInitializer(StringProperty receivingMessageModel) {

this.receivingMessageModel = receivingMessageModel;

}

@Override

protected void initChannel(SocketChannel socketChannel) throws Exception {

ChannelPipeline pipeline = socketChannel.pipeline();

pipeline.addLast("framer", new LengthFieldBasedFrameDecoder(8192, 0, 2, 0, 0));

pipeline.addLast( new PackageCodec());

pipeline.addLast( new SecurityHandler(false));

pipeline.addLast( new PackageToMessagesCodec());

pipeline.addLast(new ChatClientHandler(receivingMessageModel));

}

}

Класс ChatServer.java

public class ChatServer {

public static void main(String[] args) throws InterruptedException {

new ChatServer(8000).run();

}

private final int port;

public ChatServer(int port){

this.port = port;

}

public void run() throws InterruptedException {

EventLoopGroup bossGroup = new NioEventLoopGroup();

EventLoopGroup workGroup = new NioEventLoopGroup();

try {

ServerBootstrap bootstrap = new ServerBootstrap()

.group(bossGroup, workGroup)

.channel(NioServerSocketChannel.class)

.childHandler(new ChatServerInitializer());

ChannelFuture future = bootstrap.bind(port).sync();

future.channel().closeFuture().sync();

} finally {

bossGroup.shutdownGracefully();

workGroup.shutdownGracefully();

}

}

}

Класс ChatServerHandler.java

public class ChatServerHandler extends SimpleChannelInboundHandler<String> {

private static final List<Channel> channels = new ArrayList<Channel>();

@Override

public void handlerAdded(ChannelHandlerContext ctx) throws Exception {

Channel incoming = ctx.channel();

for (Channel channel : channels) {

channel.writeAndFlush("[SERVER] - " + incoming.remoteAddress() + " has joined\n");

}

channels.add(ctx.channel());

}

@Override

public void handlerRemoved(ChannelHandlerContext ctx) throws Exception {

Channel incoming = ctx.channel();

for (Channel channel : channels) {

channel.writeAndFlush("[SERVER] - " + incoming.remoteAddress() + " has left\n");

}

channels.remove(ctx.channel());

}

@Override

protected void channelRead0(ChannelHandlerContext channelHandlerContext, String s) throws Exception {

Channel incoming = channelHandlerContext.channel();

for (Channel channel : channels) {

if (channel != incoming){

channel.writeAndFlush("[" + incoming.remoteAddress() + "]" + s + "\n");

} else {

channel.writeAndFlush("[ВЫ]" + s + "\n");

}

}

}

}

Класс ChatServerInitializer.java

public class ChatServerInitializer extends ChannelInitializer<SocketChannel> {

@Override

protected void initChannel(SocketChannel socketChannel) throws Exception {

ChannelPipeline pipeline = socketChannel.pipeline();

pipeline.addLast("framer", new LengthFieldBasedFrameDecoder(8192, 0, 2, 0, 0));

pipeline.addLast( new PackageCodec());

pipeline.addLast( new SecurityHandler(true));

pipeline.addLast( new PackageToMessagesCodec());

pipeline.addLast( new ChatServerHandler());

}

}