РЕФЕРАТ

Расчетно-пояснительная записка к выпускной квалификационной работе «Метод замещения страниц в разделяемом кэш буфере Postgres с использованием нейронных сетей» содержит 83страниц, 4 раздела, 18 рисунков, 0 таблиц и список используемых источников из 29 наименований.

Ключевые слова: замещение страниц, рекуррентные сети, Postgres.

Объект разработки – метод замещения страниц в разделяемом кэш буфере Postgres.

Цель работы: разработка метода замещения страниц в разделяемом кэш буфере Postgres с использованием нейронных сетей.

В первой части работы выполнен анализ существующих методов замещения страниц. Изучены принципы работы разделяемого кэш буфера PostgreSQL. Проведен сравнительный анализ нейронных сетей, подходящих для решения задачи. Сформулирована цель и формализована постановка задачи в виде IDEF0-диаграммы.

Во второй части разработан метод замещения страниц в разделяемом кэш буфере PostgreSQL. Описаны основные особенности предлагаемого метода. Сформулированы ограничения предметной области. Изложены основные этапы разрабатываемого метода в виде детализированной диаграммы IDEF0 и схем алгоритмов. Выделены функции и структуры СУБД PostgreSQL, которые используются при работе с кэш буфером.

В третьей части обоснован выбор программных средств реализации метода замещения страниц в разделяемом кэш буфере. Создана обучающая выборка с помощью логирования обращений к буферу. Приведены примеры работы программы. Описаны используемые методы тестирования программного обеспечения и приведены его результаты.

В четвертой части проведено исследование разработанного метода и выявлена зависимость коэффициентов попадания и совпадения от количества обращений к страницам на тестовой выборке. Проведено сравнение полученных результатов и значений этих метрик для существующих аналогов.

Разработанный метод замещения страниц может быть использован в СУБД Postgres. Использование метода позволит повысить коэффициент попадания в разделяемом кэш буфере, что должно привести к уменьшению времени отклика системы.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современные базы данных, такие как PostgreSQL, сталкиваются с постоянно растущими требованиями к производительности и эффективности управления ресурсами. Одним из ключевых аспектов работы базы данных является управление памятью, включая работу с разделяемым кэш буфером [1].

Загрузка данных с диска занимает гораздо больше времени, чем из оперативной памяти, поэтому современные системы управления базами данных используют область в оперативной памяти в качестве буфера для кэширования недавно просмотренных страниц, чтобы в будущем запросы к страницам в буфере выполнялись быстрее. Обычно буфер делится на части одинакового размера, где каждая часть может содержать страницу. Когда транзакция базы данных запрашивает страницу, которая в данный момент не хранится в буфере, она должна быть загружена в буфер. Если для кэширования этой страницы больше нет места, то одна из страниц в буфере должна быть вытеснена, чтобы освободить место для новой запрашиваемой страницы. Выбор такой страницы важен для уменьшения задержки доступа. Если все время для вытеснения будет выбираться страница, к которой в скором времени опять произойдет обращение, то производительность может ухудшиться до случая, когда данные в основном берутся с диска.

Для выбора страницы, которую надо исключить из буфера, применяются различные эвристические алгоритмы. Все эти алгоритмы являются приближением оптимального алгоритма и не учитывают структуру конкретной рабочей нагрузки. Если алгоритм замещения страниц будет учитывать особенности рабочей нагрузки, то число операций чтения и записи на диск может быть снижено, что приведет к повышению производительности системы.

Помимо систем управления базами данных методы замещения страниц используются в операционных системах, аппаратном и программным кэше, а также в других местах, где присутствует два типа памяти, один из которых меньше по объему и быстрее по скорости доступа.

Целью данной работы является разработка метода замещения страниц в разделяемом кэш буфере postgres с использованием нейронных сетей.

Для достижения поставленной цели требуется выполнить следующие задачи:

— сравнить существующие методы замещения страниц;

— описать и спроектировать метод замещения страниц с использованием нейронных сетей;

— разработать программное обеспечение для предложенного метода;

— провести сравнение разработанного метода с существующими аналогами по коэффициентам совпадения и попадания.

1 Аналитический раздел

1.1 Особенности управление памятью

1.1.1 Управление памятью в операционных системах

Виртуальная память представляет собой ключевую концепцию в управлении памятью современных компьютерных систем. Она позволяет программам использовать объем оперативной памяти, превышающий физически доступный, за счет автоматического перемещения данных между основной памятью и вторичным хранилищем. Это достигается благодаря использованию виртуальных адресов, которые транслируются в физические адреса с помощью аппаратных средств.

Каждая программа работает с собственным адресным пространством, которое разбивается на страницы, представляющие собой непрерывные диапазоны адресов. Эти страницы не обязательно должны все одновременно находиться в оперативной памяти для выполнения программы, что позволяет эффективно использовать доступную память.

Когда программа обращается к данным, которые уже находятся в физической памяти, аппаратное обеспечение обеспечивает необходимое отображение адресов. Когда программа пытается получить доступ к странице, которая присутствует в виртуальном адресном пространстве, но отсутствует в физической памяти возникает системное прерывания отсутствия страницы после чего управление передается операционной системе.

Операционная система реагирует на ошибку отсутствия страницы, выбирая страницу при помощи алгоритма замещения и сбрасывая её содержимое на диск, если оно уже не находится там. Затем система извлекает нужную страницу с диска и помещает её в освободившееся место в памяти. После этого в таблицы вносятся соответствующие изменения, и прерванная команда выполняется заново.

Таблица страниц содержит сведения о каждой странице, включая номер страничного блока, который является ключевым элементом страничного отображения. Также в информации содержится бит присутствия-отсутствия, если он равен 1, запись активна и может быть использована, иначе соответствую-

щая виртуальная страница в данный момент отсутствует в памяти, и любое обращение к такой записи вызывает ошибку отсутствия страницы. Биты защиты указывают на тип доступа, который разрешен для страницы. В самом простом случае это один бит, который равен 0 для чтения-записи и 1 для только чтения. В более сложных системах могут быть использованы три бита, каждый из которых разрешает чтение, запись или исполнение страницы. Биты модификации и ссылки служат для отслеживания использования страницы. Бит модификации автоматически устанавливается при записи в страницу и помогает операционной системе определить, нужно ли сохранять страницу на диск при ее выгрузке из памяти. Бит ссылки устанавливается при любом обращении к странице и помогает операционной системе определить, какую страницу следует выгрузить при возникновении ошибки отсутствия страницы.

Большинство программ часто обращаются к ограниченному набору страниц. Из-за этого только небольшая часть записей в таблице страниц активно используется, а остальная практически не задействуется. Основываясь на этом наблюдении, для повышения производительности системы было предложены добавить в аппаратуру специальное устройство, которое называется TLB, и отвечает за трансляцию виртуальных адресов в физические для самых используемых страниц.

При использовании TLB существует 2 типа ошибок: программные и аппаратные. Программная ошибка возникает, когда страница отсутствует в TLB, но есть в памяти, и ее можно исправить простым обновлением TLB без обращения к диску. Это занимает 10-20 машинных команд и несколько наносекунд. Аппаратная ошибка возникает, когда страница отсутствует в памяти и требуется обращение к диску, что занимает несколько миллисекунд. Она обрабатывается значительно медленнее программной ошибки.

При возникновении ошибки отсутствия страницы, операционная система должна определить, какую страницу из памяти исключить, чтобы освободить место для загружаемой страницы. Если страница, которую нужно заместить, была изменена с момента загрузки в память, то ее содержимое должно быть обновлено на диске. Если страница не подвергалась изменениям и дисковая копия актуальна, то перезапись не требуется. В этом случае новая страница просто замещает старую.

1.1.2 Управление памятью в PostgreSQL

Разделяемый кэш буфер сохраняет страницы в оперативной памяти, доступ к которой в сотни тысяч раз быстрее, чем к дисковому хранилищу, где содержится вся информация о состоянии базы данных [2].

В операционной системе также есть дисковый кэш, который решает ту же проблему, поэтому системы управления базами данных обычно стараются избежать двойного кэширования, обращаясь к дискам напрямую, а не через кэш ОС. В случае с PostgreSQL это не так: все данные читаются и записываются с помощью обычных файловых операций. Схема взаимодействия разделяемого кэш буфера и кэша уровня операционной системы представлена на рисунке 1.1.

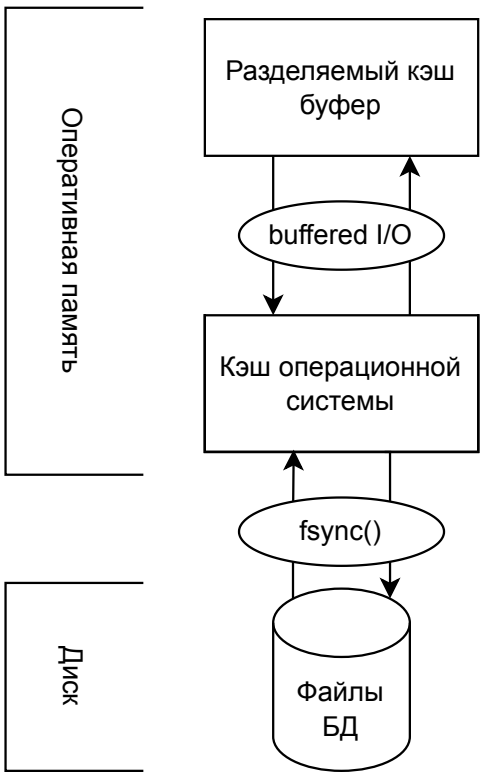

Рисунок 1.1 – Схема взаимодействия разделяемого кэш буфера с ОС

Пои чтении страница сначала ищется в кэш буфере, если она там не

находится, то отправляется запрос операционной системе на чтение страницы. Если операционная система находит ее в своем кэше, то данные сразу копируются в кэш буфер PostgreSQL, иначе операционная система загружает данные с диска в свой кэш и затем они копируются и попадает в разделяемый кэш буфер.

Для записи содержимого буфера на диск используется три процесса:

— user backend;

— bgwriter;

— checkpointer.

Процесс user backend обрабатывает пользовательские соединения. В случае запросов, которые изменяют данные, процесс user backend помечает измененные страницы для дальнейшей записи на диск. В случае, когда процессу надо исключить одну страницу из буфера и записать другую и страница для замещения помечена как измененная, user backend сам инициирует процесс копирования в кэш операционной системы для дальнейшей записи на диск.

Процесс bgwriter работает в фоновом режиме и нужен для снижения нагрузки на user backend и checkpointer. Он запускается раз в некоторый промежуток времени, который устанавливается через конфигурационный файл, просматривает, какие страницы были изменены и записывает их в кэш операционной системы для дальнейшей записи на диск.

Процесс checkpointer отвечает за создание контрольных точек, когда данные гарантированно записываются на диск. В момент вызова контрольной точки все измененные страницы записываются на диск и в лог файл добавляет специальная запись о контрольной точке. Все изменения, сделанные до этой записи, гарантировано записаны на диск.

Для восстановления после аварийного завершения в PostgreSQL используется механизм Write ahead logging. Все изменения перед внесением в страницы в разделяемом кэш буфере записываются в специальный лог файл. После прохождения очередной контрольной точки все изменения гарантировано синхронизированы с диском и записи, предшествующие этой точке, могут быть удалены из лог файла.

Контрольная точка создается в следующих случаях:

— явный вызов команды checkpoint;

— по истечению таймера, который задается в конфигурационном файле, по умолчанию – 300 секунд;

— размер write ahead log файла достиг максимального размера, заданного в конфигурационном файле;

— вызов функции pg_start_backup;

— вызов команды pg_basebackup;

— вызов команды завершения работы СУБД;

— при вызове команд создания и удаления базы данных.

Проблема двойного кэширования отчасти решается тем, что пока страница находится в разделяемом кэш буфере все обращения к ней идут через него и кэш уровня операционной системы задействован не будет. В следствии этого, рано или поздно страница будет исключена из кэша уровня операционной системы, так как к ней не будет обращений.

Буферный кэш является списком блоков. Каждый блок содержит страницу с данными и заголовок. Заголовок содержит:

— номер блока страницы;

— индикатор того, что страница была изменена, но еще не записана на диск;

— число обращений к странице;

— число активных операций или транзакций, которые используют страницу.

При старте все блоки в буферном кэше помещаются в список свободных. Для поиска нужной страницы используется хэш таблица. В качестве ключа используются номер файла и номер страницы в файле.

При обращении к памяти процесс сначала пытается найти страницу в кэше. Если она уже загружена, то счетчик обращений в заголовке соответствующего буфера увеличивается на единицу. До тех пор, пока это счетчик не равен нулю, страница не может быть выгружена из кэша.

Если страница не была найдена в кэше, то она должна быть прочитана с диска в какой-то блок. Если список свободных блоков не пуст, то будет взят первый из него, иначе требуется выбрать страницу, которая будет вытеснена из кэша.

В PostgreSQL для выбора кандидата на замещение анализируются два счетчика – число обращений и количество использований. Алгоритм часы поочередно проходит по всем страницам в кэше и, если оба счетчика равны нулю, то текущая страница будет замещена, иначе оба счетчика уменьшаются на единицу. Для избежания большого числа проходов по всем страницам в поисках кандидата на замещение по умолчанию счетчики не могут быть больше пяти.

Когда кандидат на замещение найден, счетчик использований ассоциированного с ним блока устанавливается в 1, чтобы другие процессы не могли его использовать. Если страница содержит измененную информацию, то запускается процесс переписывания в кэш ОС для дальнейшей записи на диск. После этого новая страница загружается в буфер и для нее выставляется счетчик обращений в единицу.

1.1.3 Вывод

Управление памятью в операционных системах и СУБД, таких как PostgreSQL, основано на схожих принципах минимизации задержек при работе с диском. В операционных системах ключевую роль играет виртуальная память, использующая страничную организацию и механизмы трансляции адресов, что позволяет эффективно распределять физическую память между процессами и избегать её переполнения за счёт выгрузки редко используемых страниц на диск.

В PostgreSQL управление памятью адаптировано под специфику работы с базами данных. Разделяемый буферный кэш служит для хранения часто используемых страниц данных в оперативной памяти, что сокращает количество обращений к диску. Однако это порождает проблему двойного кэширования, так как ОС также использует свой дисковый кэш. PostgreSQL частично решает её, минимизируя взаимодействие с кэшем ОС: пока страница

находится в буфере СУБД, обращения к ней идут напрямую.

Далее в данной работе будут рассмотрены различные методы замещения страниц, которые могут быть использованы как в ядре операционной системы, так и при выборе страниц для замещения в разделяемом кэш буфере PostgreSQL. Также будет предложен новый метод замещения страниц с использованием нейронных сетей, который должен снизить число дисковых операций.

1.2 Методы замещения страниц

1.2.1 Оптимальный алгоритм

Оптимальной алгоритм предлагает вытеснять страницу, которая будет без ссылок в течение самого длительного времени. Этот алгоритм может быть реализован только во втором идентичном прогоне при условии использования истории страниц полученной во время первого запуска. Когда система сталкивается с нагрузкой в режиме реального времени у нее этой истории нет, поэтому этот алгоритм не может быть реализован на практике. Оптимальный алгоритм может быть использован для оценки других алгоритмов замещения страниц, которые могут быть применены и при первом прогоне.

1.2.2 NRU

Для того чтобы собирать статистику использования страниц виртуальной памяти, большинство компьютеров используют два бита состояния для каждой страницы. Бит R устанавливается при обращении к странице, а бит M устанавливается, когда страница изменяется.

Если аппаратура не поддерживает эти биты, то они могут быть созданы с помощью механизмов операционной системы. При запуске процесса все записи в его таблице страниц помечаются как отсутствующие в памяти. Когда происходит обращение к странице, возникает ошибка отсутствия страницы, и операционная система устанавливает бит R, изменяет запись в таблице

страниц, устанавливая режим доступа только для чтения, и перезапускает команду. Если страница впоследствии изменяется, возникает другая ошибка страницы, позволяющая операционной системе установить бит M и изменить режим доступа к странице на чтение-запись.

Аналогичные биты хранятся в заголовке каждого блока в разделяемом кэш буфере PostgreSQL. В качестве бита R можно использовать счетчик обращений, который будет сбрасываться по истечению определенного времени. Для того, чтобы отметить страницы, которые были изменены в заголовке каждого блока присутствует специальный бит модификации.

Идея алгоритма Not Recently Used заключается в следующем: при запуске процесса оба этих бита для всех страниц устанавливаются в 0. При каждом прерывании от таймера бит R сбрасывается, чтобы отличить страницы, к которым не было обращений в последнее время, от тех, к которым были такие обращения.

При возникновении ошибки отсутствия страницы операционная система анализирует все страницы и на основе текущих значений битов R и M разделяет их на четыре категории:

1. К которым не было ни обращений, ни модификаций в последнее время.
2. К которым не было обращений в последнее время, но были модификации.
3. К которым были обращения в последнее время, но не было модификаций.
4. К которым были и обращения, и модификации в последнее время.

Для замещения выбирается произвольная страница из самого низкого непустого класса.

1.2.3 FIFO и его модификации

Система ведет список всех страниц, находящихся в памяти в данный момент. Недавно поступившие страницы находятся в конце списка, а те, что поступили раньше всех, находятся в начале. Если возникает ошибка отсутствия страницы, удаляется страница из начала списка, и в конец добавляется новая страница.

Алгоритм второй шанс является простой модификацией алгоритма FIFO и решает проблему удаления часто востребуемой страницы. Для этого используется проверка бита R самой старой страницы. Если значение этого бита равно нулю, то это означает, что страница не только старая, но и невостребованная, поэтому она сразу же удаляется. Если бит R имеет значение 1, то он сбрасывается, а страница помещается в конец списка страниц, а время ее загрузки обновляется, как будто она только что поступила в память. Затем поиск продолжается.

Алгоритм часы является улучшением алгоритма второй шанс. Он основан на идее использования циклического списка страниц, представленного в виде часов, где стрелка указывает на самую старую страницу.

Принцип работы алгоритма часы следующий:

1. В начале работы алгоритма все страницы помещаются в циклический список в виде часов, где каждая страница имеет бит R, который указывает на ее актуальность.

2. При возникновении ошибки отсутствия страницы проверяется страница, на которую указывает стрелка в циклическом списке.

3. Если бит R этой страницы равен 0, она удаляется из памяти, на ее место загружается новая страница, и стрелка сдвигается вперед на одну позицию.

4. Если бит R равен 1, он сбрасывается, и стрелка перемещается на следующую страницу в списке.

5. Этот процесс повторяется до тех пор, пока не будет найдена страница с битом $R = 0$.

1.2.4 LRU

Алгоритм замещения наименее востребованной страницы основан на идее, что страницы, которые долгое время не были востребованы, скорее всего, останутся невостребованными, в то время как страницы, которые интенсивно

использовались в последнее время, вероятно будут снова востребованы. Поэтому стратегия замещения страниц в этом алгоритме основана на выборе наименее востребованной страницы для удаления.

Для реализации алгоритма Least Recently Used каждая страница в памяти связывается с программным счетчиком, который имеет начальное значение 0. При каждом прерывании от таймера операционная система сканирует все страницы в памяти. Для каждой страницы к счетчику добавляется значение бита R, который равен 0 или 1. Таким образом, счетчики позволяют приблизительно отслеживать частоту обращений к каждой странице.

При возникновении ошибки отсутствия страницы для замещения выбирается та страница, у которой счетчик имеет наименьшее значение, то есть та страница, которая дольше всего не была востребована.

Основная проблема этого алгоритма заключается в том, что он никогда не сбрасывает счетчики и страницы, которые активно использовались в прошлом, и сейчас не востребованы все равное будут оставаться в памяти.

Для борьбы с этой проблемой существует алгоритм старения, который предлагает при каждом прерывании таймера не прибавлять 1 к счетчику, а делать сдвиг вправо и прибавлять 1 к левому биту счетчика.

1.2.5 Алгоритм рабочий набор

Процессы начинают работу без каких-либо страниц в памяти, что приводит к ошибкам отсутствия страниц при первом обращении к данным. Система загружает страницы по мере необходимости. Постепенно процесс получает большинство необходимых ему страниц и начинает работу более стабильно. Рабочий набор страниц, используемых процессом в данный момент, важен для эффективной работы. Многие системы замещения страниц стремятся отслеживать рабочий набор каждого процесса и обеспечивать его присутствие в памяти, перед перезапуском процесса.

Для реализации модели рабочего набора необходимо, чтобы система отслеживала, какие страницы именно входят в рабочий набор. Имея эту информацию, можно использовать следующий алгоритм замещения страниц: при возникновении ошибки отсутствия страницы следует выселить ту страницу,

которая не принадлежит рабочему набору.

Рабочий набор представляет собой набор страниц, используемых в k последних обращениях к памяти. Для реализации алгоритма можно отслеживать страницы, использованные в k последних миллисекундах выполнения, вместо поиска страниц, используемых в k последних обращениях. Для получения этой информации можно добавить специальное поле в таблицу страниц и обновлять его на основе бита R по тику таймера.

Если возраст страницы превышает заранее выбранное значение на момент возникновения ошибки, она становится кандидатом на замену. В противном случае удаляется страница с самым большим возрастом или случайная, если у всех страниц одинаковый параметр.

1.2.6 Алгоритм WSClock

Алгоритм WSClock (Working Set Clock) является модификацией алгоритма рабочего набора и базируется на структуре данных, аналогичной циклическому списку страничных блоков, используемой в алгоритме часы.

Основные принципы работы данного алгоритма следующие:

1. Создается пустой циклический список страничных блоков.

2. При загрузке первой страницы она добавляется список. По мере загрузки следующих страниц они также попадают в список, формируя замкнутое кольцо.

3. В каждой записи списка содержится поле времени последнего использования из базового алгоритма рабочего набора, а также биты R и M.

4. При возникновении ошибки отсутствия страницы сначала проверяется страница, на которую указывает стрелка в списке. Если бит R установлен в 1, это означает, что страница была использована в течение текущего такта и не является идеальным кандидатом на удаление.

5. Затем бит R устанавливается в 0, стрелка перемещается на следующую страницу в списке, и процесс повторяется уже для нее.

6. После того, как бит R у страницы, на которую указывает стрелка, равен 0 и ее возраст превышает заданное значение, а также страница не изменена, происходит замещение этой страницы.

7. Если страница была изменена, то планируется запись на диск.

Если стрелка проходит полный круг и хотя бы одна запись на диск запланирована, поиск может продолжаться до тех пор, пока не будет найдена неизмененная страница. В противном случае все страницы считаются частью рабочего набора, и замещается любая страница, которая не была изменена. Если такой страницы нет, то замещается текущая страница.

1.2.7 Сравнительный анализ методов замещения страниц

Оптимальный алгоритм удаляет страницу с самым отдаленным предстоящим обращением. На практике реализовать такой алгоритм невозможно, но его можно использовать в качестве оценочного критерия.

Алгоритм исключения недавно использовавшейся страницы проводит разбиение всех страниц, основываясь на состоянии битов M и R, на 4 класса и проводит замещение произвольной страницы наименьшего непустого класса.

Алгоритм FIFO работает по принципу очереди и удаляет самую старую страницу. Алгоритм второй шанс борется с недостатками FIFO и перед удалением страницы проверяет не используется ли она в данный момент. Алгоритм часы является разновидностью алгоритма второй шанс, но требует меньше времени на выполнение.

Алгоритм замещения наименее востребованной страницы стремится удалять страницы, которые не были востребованы долгое время. У этого алгоритма есть недостаток, связанный с тем, что страница, которая активно использовалась в прошлом, не обязательно будет востребована сейчас. Для борьбы с этим недостатком был разработан алгоритм старения.

Алгоритм рабочего набора отслеживает набор страниц, используемых за определенный промежуток времени и замещает страницу, которая не относится к рабочему набору. Алгоритм WSClock является оптимизацией алгоритма

рабочего набора.

На практике чаще всего используются алгоритм старения и WSClock. Оба обеспечивают неплохую производительность страничной организации памяти и могут быть эффективно реализованы, но не лишены недостатков на определенном наборе задач.

1.2.8 Вывод

Алгоритмы замещения страниц предлагают различные стратегии баланса между производительностью и ресурсозатратностью. Теоретически оптимальный алгоритм демонстрирует максимальную эффективность, удаляя страницу с самым отдалённым обращением, но его практическая реализация невозможна из-за отсутствия данных о будущих запросах. Более простые методы, такие как NRU и FIFO с модификациями («второй шанс», «часы»), используют биты обращения и циклические списки для минимизации накладных расходов, однако их эффективность ограничена в динамичных сценариях.

Алгоритмы LRU с механизмом старения и WSClock учитывают не только количество обращений, но и время последнего обращения.

Все существующие алгоритмы замещения страниц являются эвристическими и не учитываются специфики конкретной рабочей нагрузки. В данной работе предлагается создать алгоритм замещения страниц, который будет учитывать особенности конкретной рабочей нагрузки для более точного приближения к оптимальному алгоритму. Для обобщения информации о рабочей нагрузки и запоминания истории обращений будут использованы нейронные сети. При обучении модели будет использован оптимальный алгоритм.

1.3 Нейронные сети

Модель нейронной сети основана на биологическом нейроне. У нейрона есть ядро, которое называется телом. В теле накапливается электрический заряд. С телом соединены отростки. Отростки, по которым сигнал поступает в тело, называются дендритами. Отросток, по которому сигнал передается

другим нейронам, называется аксоном. Место, где аксон соединяется с дендритами, называется синапсом. Синапс отвечает за количество заряда, которое перейдет от аксона к дендриту. Синапс может изменяться со временем. Именно с настройкой синапса и связана тренировка биологической нейронной сети.

1.3.1 Математическая модель МакКаллока-Питтса

В математической модели МакКаллока-Питтса, тело нейрона, где накапливается заряд, заменяется на сумматор. Дендриты являются входами сумматора, а выходом – аксоном. Биологический нейрон накапливает заряд до тех пор, пока этот заряд не достигнет какого-то значения, и только после этого этот заряд уходит по аксону к другим нейронам. В математической модели к сигналу после выхода из сумматора применяется функция активации и только после этого сигнал попадает на дендрит следующего нейрона. Синапсы в математической модели заменяются на веса входов нейрона. Математическая модель нейрона выражается зависимостью 1.1

$$y = f\left(\sum_{i=1}^{n}(w_{\mathrm{i}}x_{\mathrm{i}}) + b\right), \tag{1.1}$$

где y – сигнал на выходе из нейрона, f – функция активации, w_{i} – вес i входа, x_{i} – сигнал этого входа, b – некоторое значение смещения, которое задается отдельно для каждого нейрона. Обучение нейронной сети происходит за счет настройки синаптических весов w_{i} и смещения b.

1.3.2 Функции активации

Существует много различных функций активации (фактически любая функция может быть функцией активации). Наиболее популярными считаются логистическую функцию, гиперболический тангенс, ReLU [3]. Важной особенностью функций активации является их дифференцируемость (хотя для некоторых функций это выполняется не всегда), поскольку при обратном распространении ошибки необходимо вычислять градиенты, использующие

производную функции активации.

Логистическая функция преобразовывает поступающие в неё значения в вещественный диапазон [0, 1]. Это означает, что при x>0 выходное значение будет примерно равно единице, а при x<0 будет близким к нулю. Данная функция часто используется в задачах классификации [3]. Логистическая функция определяется зависимостью 1.2.

$$y = \frac{1}{1 + e^{-x}}. \tag{1.2}$$

Гиперболический тангенс схож с логистической функцией, но в отличии от нее может принимать отрицательные значения. Гиперболический тангенс определяется зависимостью 1.3.

$$y = \frac{e^{2x} - 1}{e^{2x} + 1}. \tag{1.3}$$

Функция ReLU возвращает 0, если принимает отрицательный аргумент, в случае же положительного аргумента, функция возвращает само число. Функция ReLU определяется зависимостью 1.4.

$$\mathrm{ReLU}(x) = \begin{cases} x, & \text{если } x > 0, \\ 0, & \text{иначе,} \end{cases} \tag{1.4}$$

ReLU решает проблему обнуления градиента (ситуация, при которой во время обучения градиенты по всем весам становятся близкими или равными нулю) для положительных чисел, также она вычисляется гораздо проще, чем сигмоидальные функции (логистическая функция, гиперболический тангенс) [3].

1.3.3 Составляющие нейронной сети

При обучении нейронной сети используются две подвыборки обучающего множества. Вся обучающая выборка состоит из какого-то количества объектов, для которых известны признаки, на которые должна обучиться нейронная сеть. Первая подвыборка называется тренировочной и используется

для итеративного обучения нейронной сети. Вторая называется тестовой и используется для оценки того, насколько хорошо обучена нейронная сеть.

Нейронную сеть определяют следующие параметры:

— архитектура нейронной сети – отвечает за то, как нейроны связаны между собой;

— функция потерь – определяет насколько точно работает модель [4];

— метод оптимизации – определяет способ уменьшения функции потерь на каждой итерации обучения.

Нейроны делятся на три типа: входной, скрытый и выходной. В том случае, когда нейросеть состоит из большого количества нейронов, вводят термин слоя. Соответственно, есть входной слой, который получает информацию, некоторое количество скрытых, которые ее обрабатывают и выходной слой, который выводит результат [5]. Количество скрытых слоев и число нейронов в каждом из них задают архитектуру нейронной сети.

1.3.4 Методы оптимизации

Один из методов оптимизации – градиентный спуск [6]. Градиентный спуск основан на пошаговом приближении функции к локальному минимуму. На каждой итерации алгоритма новые значения получаются по формуле 1.5

$$w_1 = w_0 - \alpha \Delta f(w_0), \tag{1.5}$$

где w_1 – вектор новых значений, которые подбираются алгоритмом, w_0 – значения параметров на текущем шаге, $\Delta f(w_0)$ – вектор градиентов функции потерь по каждому из параметров на текущем шаге, α – скорость обучения.

На каждой итерации градиентного спуска требуется считать градиент функции потерь, которая зависит от функций активации каждого из нейронов сети. В связи с этим к фукнциям потерь и активации применяются требования по дифференцируемости.

В связи с тем, что градиентный спуск находит только локальный минимум, не всегда полученный результат будет оптимальным. Результат работы

алгоритма зависит от изначальных настроек параметров нейронной сети.

Выделяют три основных типа градиентного спуска [6]:

— мини-пакетный градиентный спуск – в этом случае обучающий набор данных разбивается на небольшие партии, которые используются для расчета ошибки модели и обновления коэффициентов модели;

— стохастический градиентный спуск – в этом случае градиент оптимизируемой функции считается на каждом шаге не как сумма градиентов от каждого элемента выборки, а как градиент от одного, случайно выбранного элемента;

— пакетный градиентный спуск – это разновидность алгоритма градиентного спуска, который вычисляет ошибку для каждого примера в наборе обучающих данных, но обновляет модель только после того, как все обучающие примеры были оценены.

Одной из проблем градиентного спуска является неизменяемая во время обучения скорость обучения. Постоянная скорость обучения может привести к следующим проблемам: если ее значение будет выбрано слишком низким, то модель будет дольше сходиться и требовать большего числа итераций для достижения оптимального решения, если ее значение будет слишком большим, то модель может расходиться и на каждой итерации проходить мимо глобального минимума.

Для решения этой проблемы нужно использовать адаптивно настраивающуюся скорость обучения. Значение скорости обучения для каждого параметра должно настраиваться адаптивно, исходя из правила, что чем больше значение ошибки, тем больше должна быть скорость обучения. Увеличение скорости обучения при больших значениях ошибки дает возможность перескочить через локальные минимумы, а ее уменьшение при малых значениях не дает модели на каждой итерации перескакивать через глобальный минимум.

Для адаптивного изменения весов модели можно использовать алгоритм RMSProps. Этот алгоритм работает по следующим правилам:

— на каждой итерации для каждого параметра считается экспоненциальное скользящее среднее градиента с учетом всей истории обучения;

— при помощи полученных значений для каждого параметра вычисляется скорость обучения и производится обновление весов модели.

Экспоненциальное скользящее среднее на очередной итерации высчитывается по формуле 1.6

$$E_{\mathrm{t}} = \beta * g_{\mathrm{t}}^2 + (1 - \beta) * E_{\mathrm{t}-1}, \tag{1.6}$$

где E_{t} – новое полученное значение экспоненциального скользящего среднего, $E_{\mathrm{t}-1}$ – значение, полученное на предыдущей итерации, β – настраиваемый коэффициент, g_{t} – градиент функции потерь по соответствующему параметру.

После этого веса обновляются с использованием соотношения 1.7

$$w_{\mathrm{t}} = w_{\mathrm{t}-1} - \frac{\eta}{\sqrt{E_{\mathrm{t}}}} g, \tag{1.7}$$

где w_{t} – новое полученное значение параметра модели, $w_{\mathrm{t}-1}$ – предыдущее значение этого параметра, η – скорость обучения, E_{t} – значение экспоненциального скользящего среднего для этого параметра.

Для улучшения сходимости модели при адаптивном обновлении скорости обучения можно считать экспоненциальное скользящее среднее не только по квадрату градиента, но и по самому значению и использовать оба полученных значения при подсчете новой скорости обучения на каждой итерации. Такой подход реализован в алгоритме Adam.

Первый и второй моменты высчитываются по формулам 1.8 и 1.9 соответственно

$$m_{\mathrm{t}} = \beta_1 * g_{\mathrm{t}} + (1 - \beta_1) * m_{\mathrm{t}-1}, \tag{1.8}$$

$$v_{\mathrm{t}} = \beta_2 * g_{\mathrm{t}}^2 + (1 - \beta_2) * v_{\mathrm{t}-1}, \tag{1.9}$$

где m_{t} и v_{t} – первый и второй моменты в соответствующий момент времени, β_1 и β_2 – настраиваемые коэффициента, g – градиент функции потерь по соответствующему параметру.

Для увеличения влияния истинных значений градиента на начальных этапах к моментам применяется корректировка по формулам 1.10 и 1.11.

$$\hat{m}_{\mathrm{t}} = \frac{m_{\mathrm{t}}}{1 - \beta_1^t}, \tag{1.10}$$

$$\hat{v}_{\mathrm{t}} = \frac{v_{\mathrm{t}}}{1 - \beta_2^t}, \tag{1.11}$$

Итоговое обновление весов осуществляется по формуле 1.12

$$w_{\mathrm{t}} = w_{\mathrm{t}-1} - \eta \frac{\hat{m}_{\mathrm{t}}}{\sqrt{\hat{v}_{\mathrm{t}}} + \epsilon}, \tag{1.12}$$

где w_{t} – новое полученной значение параметра модели, $w_{\mathrm{t}-1}$ – предыдущее значение этого параметра, η – скорость обучения, ϵ – поправка, защищающая от деления на ноль.

1.3.5 Функции потерь

Согласно исследованиям [7] для задачи классификации самой эффективной функцией потерь являются категориальная перекрестная энтропия, которая определяется выражением 1.13

$$CM = -\sum_{i=1}^{N} t_{\mathrm{i}} \log p_{\mathrm{i}}, \tag{1.13}$$

где N – число классов классификации, t_{i} – 0 или 1 в зависимости от того принадлежит ли изображение на входе нейронной сети классу, за который отвечает i нейрон выходного слоя, p_{i} – результат на выходе из нейрона.

В задачах классификации используют категориальную перекрестную энтропию в качестве функции потерь. В таком случае на выходном слое нейронной сети создается столько нейронов, сколько возможных классов может иметь объект на входе. В качестве функции активации для каждого из таких нейронов используют софт макс. Софт макс определяется выражением 1.14

$$SM_{\mathrm{i}} = \frac{e^{y_{\mathrm{i}}}}{\sum\limits_{i=1}^{N} e^{y_{\mathrm{j}}}}, \tag{1.14}$$

где y_{i} – результат на выходе из нейрона, к которому применяется функция активации, N – число нейронов в выходном слое, y_{j} – результат на выходе из j нейрона выходного слоя.

Знаменатель в выражении 1.14 отвечает за нормировку. Таким образом, каждый из нейронов выходного слоя показывает вероятность принадлежности объекта на входе нейронной сети к некоторому классу, а сумма всех этих вероятностей будет равна 1.

1.4 Многослойные сети

1.4.1 Перцептрон

Перцептрон – это математическая модель, воспроизводящая принципы обработки информации, схожие с работой человеческого головного мозга. Архитектура перецептрона состоит из трех ключевых элементов [8]:

— сенсоры – получают входные сигналы;

— ассоциативные элементы – обрабатывают данные;

— реагирующие элементы – формирует итоговый отклик.

Эти компоненты организованы в слои нейронной сети: входной, один или несколько скрытых и выходной. Схема трехслойного перцептрона приведена на рисунке 1.2

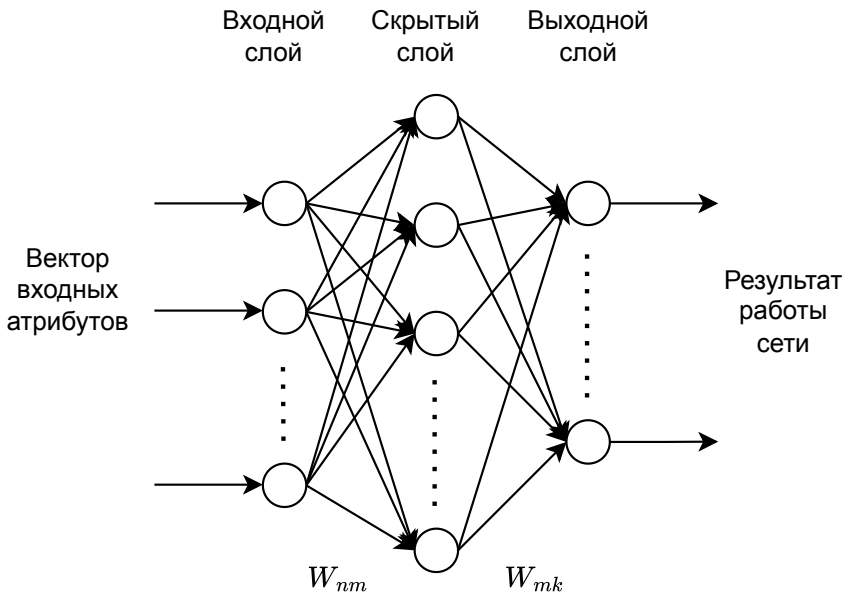

Рисунок 1.2 – Схема перцептрона

W_{nm} и W_{mk} на схеме – матрицы обучаемых весов. n, m, k – число нейронов на входном, скрытом и выходном слоях соответственно.

Результат на выходе из k-го нейрона выходного слоя может быть посчитан по следующей формуле: 1.15

$$Y_k(x) = Y_k(x_1, x_2, ..., x_n) = f_2(\sum_{j=0}^{m}(w_{jk} f_1(\sum_{i=0}^{n}(w_{ij} x_i))), \tag{1.15}$$

где f_1 и f_2 – функции активации на скрытом и выходном слоях соответственно, w_{ij} и w_{jk} – обучаемые веса из матриц W_{nm} и W_{mk}.

В многослойном перцептроне данные последовательно проходят через входной слой, скрытый и выходной. На каждом этапе нейроны слоя выполняют следующие действия:

— линейная комбинация – нейрон умножает сигналы на соответствующие веса и суммирует результаты;

— нелинейное преобразование – полученная сумма пропускается через активационную функцию, которая усиливает характеристики, за которые отвечает соответствующий узел.

Марвин Минский и Сеймур Паперт в своей работе [9] показали, что

однослойные нейронные сети, включающие только входной и выходной слои, способны работать исключительно с линейно разделимыми задачами и обеспечивают лишь линейную аппроксимацию. Этот барьер преодолевается за счёт внедрения скрытых слоёв.

В таких сетях скрытый слой играет ключевую роль: входные данные трансформируются в новое пространство, где выходной слой строит разделяющие поверхности для классификации. Таким образом, модель не только анализирует исходные признаки, но и выявляет признаки признаков, формируемые скрытыми нейронами. Это позволяет сети обучаться сложным нелинейным закономерностям.

Число нейронов на входном слое напрямую зависит от числа входных признаков. Число нейронов на выходном слое зависит от решаемой задачи. Для задач классификации это число равно числу возможных классов и каждый нейрон на выходном слое отвечает за вероятность принадлежности входного параметра одному из классов.

Число нейронов на скрытом слое, как правило, выбирается опытным путем. Слишком большое число нейронов может привести к переобучению модели, а слишком маленькое может не хватить для решения поставленной задачи.

Хехт-Нильсен в своей работе [10], основанной на работе Колмогорова [11] предлагает использовать $2N + 1$ нейронов на скрытом слое, где N – число входов сети.

Баум и Хесслер с своей работе [12] вводят эмпирическое правило для числа весов на скрытом слое для борьбы с переобучением 1.16

$$p > \frac{w}{\epsilon}, \tag{1.16}$$

где p – размер обучающей выборки, w – число весов, ϵ – допустимый уровень ошибки.

Для обучения перцептрона могут быть использованы следующие методы:

— стохастические методы обучения;

— обратное распространение ошибки.

Стохастические методы обучения предполагают обучение по следующему алгоритму:

1. Выбрать значения весов случайным образом.

2. Подсчитать значение функции потерь.

3. Подкорректировать значение случайного весового коэффициента на небольшую величину. Если коррекция уменьшает значение функции потерь, то оставить ее, иначе вернуться к изначальному состоянию.

4. Повторять шаг 3 до тех пор, пока сеть не будет достаточно обучена.

Для реализации обратного распространения ошибки необходимо осуществить минимизацию функции потерь с помощью изменения значений обучаемых весов в направлении обратном градиенту. Для расчета частной производной по каждому из весов необходимо выразить функцию потерь через значения весов и входных аргументов. Для того, чтобы выразить значение на выходе из k-го нейрона через обучаемые веса и входные параметры можно использовать формулу 1.15.

1.4.2 RBF сеть

RBF сеть является многослойным перцептроном, на скрытом слое которого используются RBF нейроны [13]. Потенциал такого нейрона рассчитывается как евклидово расстояние между векторами весовых коэффициентов и входных величин 1.17:

$$V = ||w - x|| = \sqrt{\sum_{i=1}^{N}(w_i - x_i)^2}, \tag{1.17}$$

где w – вектор весовых коэффициентов, x – вектор входных величин, а N – размер этих векторов.

Для RBF нейронов используется следующая функция активации 1.18:

$$f(V) = e^{-(\frac{V}{b})^2}, \tag{1.18}$$

где V – потенциал нейрона, b – коэффициент разброса, который отвечает за плавность функции.

Схема RBF сети приведена на рисунке 1.3

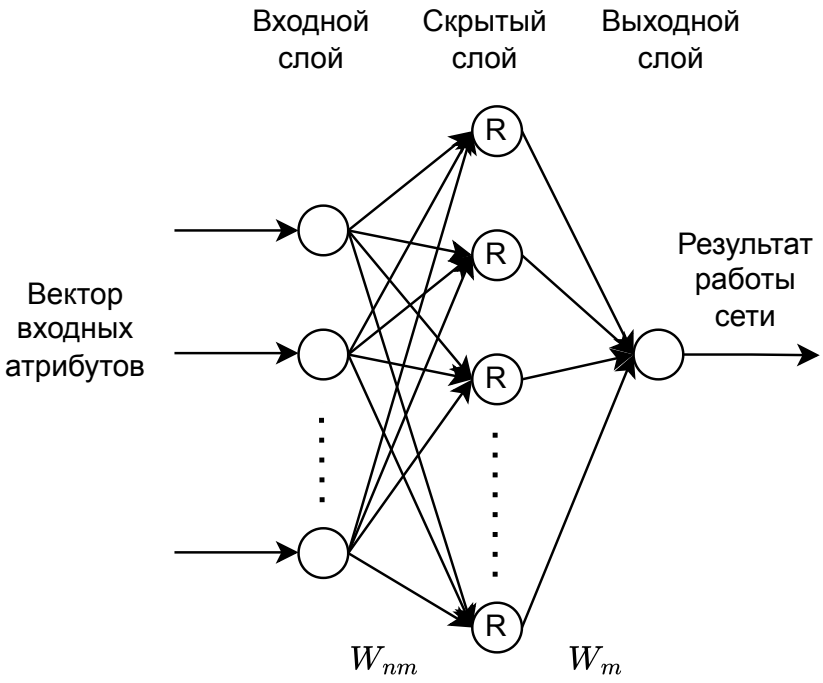

Рисунок 1.3 – Схема RBF сети

На схеме буквой R обозначены RBF нейроны. W_{nm} – матрица обучаемых весов скрытого слоя, W_m – вектор обучаемых весов выходного слоя, n – число нейронов на входном слое, m – число RBF нейронов. Выходной слой является линейной комбинацией активаций RBF нейронов скрытого слоя.

Особенностью такой сети является то, что каждый RBF нейрон активируется только тогда, когда входные данные близки к его эталонному вектору весов w. Таким образом, каждый нейрон скрытого слоя представляет собой центр кластера в пространстве данных, а параметр b отвечает за область охвата этого нейрона. Чем больше значение b, тем большое отклонение от вектора эталонных весов будет приводить к активации нейрона.

Обучение RBF сети состоит из трех этапов:

— выбор центров кластеров;

— расчет параметра разброса b;

— обучения весов выходного слоя.

Для выбора центров кластеров могут быть использованы методы кластеризации, к примеру, k-средний.

Параметр разброса может быть как настроен вручную для каждого кластера, так и задан константой для всех нейронов. К примеру, параметр b может быть задан как среднее расстояние между центрами кластеров.

Для обучения весов выходного слоя может быть использована линейная регрессия.

1.4.3 Вероятностная сеть

Вероятностная модель является развитием RBF сети. Схема такой модели представлена на рисунке 1.4

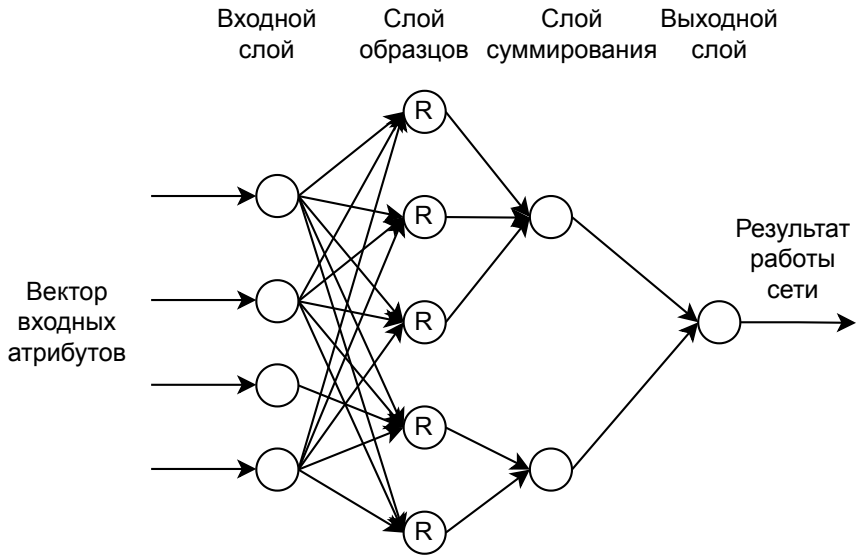

Рисунок 1.4 – Схема вероятностной сети

Вероятностная сеть состоит из следующих слоев:

— входной слой;

— слой образцов;

— суммирующий слой;

— выходной слой.

Входной слой – первый этап работы сети, который принимает входные данные. Каждый нейрон на этом слое соответствует одному входному признаку.

Каждый нейрон на слое образцов соответствует одному образцу из обучающей выборки. Нейроны этого слоя используют RBF функцию активации для вычисления сходства между входным набором данных и образцом, которому они соответствуют.

Каждый нейрон на слое суммирования представляет класс. Нейроны этого слоя суммируют активации RBF нейронов с предыдущего слоя, которые относятся к одному классу.

Выходной слой принимает решение о классификации входных данных, выбирая класс с наибольшей вероятностью.

1.5 Рекуррентные сети

Рекуррентные нейронные сети – нейронные сети с обратной связью между различными слоями нейронов. Их характерная особенность – передача сигналов с выходного или скрытого слоя во входной слой. Рекуррентная нейронная сеть может состоять из любого числа слоев.

Рекуррентные нейронные сети хорошо подходят для обработки последовательностей, например, временные ряды (изменения цен акций, показания датчиков), последовательности с зависимыми элементами (предложения естественного языка), то есть любые данные, где соседние экземпляры (точки выборки) зависят друг от друга и эту зависимость нельзя игнорировать [14].

Общая схема рекуррентной сети приведена на рисунке 1.5.

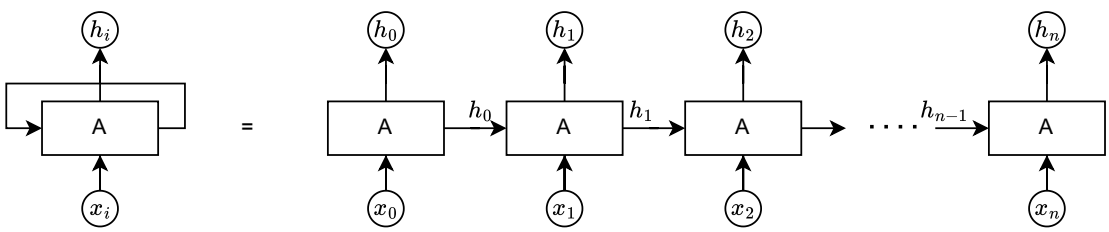

Рисунок 1.5 – Схема рекуррентной сети

Фрагмент нейронной сети A принимает входное значение x_i и возвращает значение h_i. Наличие обратной связи позволяет передавать информацию

от одного шага сети к другому. Рекуррентную сеть можно рассматривать, как несколько копий одной и той же сети, каждая из которых передает информацию последующей копии.

1.5.1 Нейронная сеть Хопфилда

Схема нейронной сети Хопфилда из трех нейронов изображена на рисунке 1.6.

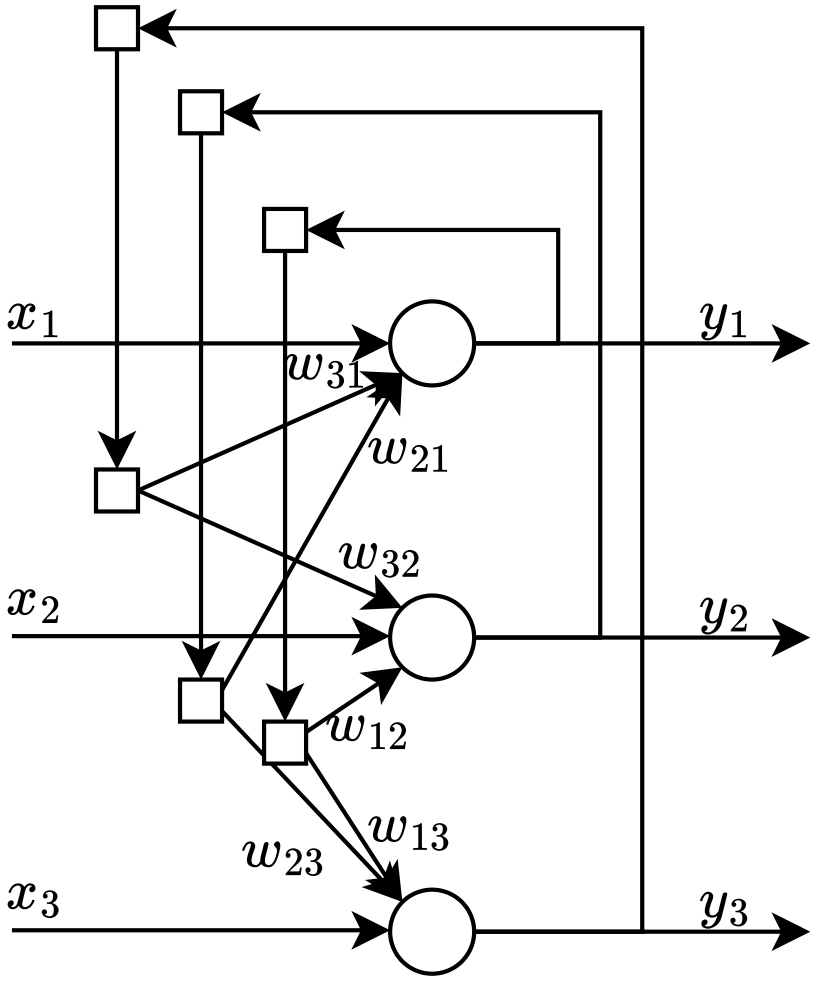

Рисунок 1.6 – Схема нейронной сети Хопфилда

Нейронная сеть Хопфилда – полносвязная однослойная нейронная сеть с симметричной матрицей связей [15]. Функционирование сети продолжается до тех пор, пока не будет достигнуто состояние равновесия, то есть до тех пор, пока новый выход из сети не будет равен предыдущему. Входной образ является начальным состоянием сети, а при равновесии получается выходной.

Сеть состоит из N нейронов, где N – размерность входного и выходного векторов. Каждый нейрон на входе и выходе может принимать одно из двух состояний 1.19:

$$x_i^{(t)}, y_i^{(t)} \in \{-1; +1\}, \tag{1.19}$$

где $x_i^{(t)}$ и $y_i^{(t)}$ – значение на входе и выходе i-го нейрона соответственно в момент времени t.

Работа сети описывается функцией энергии 1.20:

$$E = -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1, j \neq i}^{N} wijx_i x_j, \tag{1.20}$$

где w_{ij} – элемент матрицы взаимодействия W, которая состоит из весовых коэффициентов. В процессе функционирования сети функция энергии должна монотонно уменьшаться.

Состояние сети определяется множеством текущих выходных сигналов y_i от всех нейронов. Таким образом, состояние сети является двоичным числом, так как на выходе нейрона может быть только 2 значения. Каждый бит соответствует значению на выходе конкретного нейрона.

Процесс обучения сети заключается в составлении матрицы взаимодействия W. Матрица строится из m эталонных образов. Каждый образ является бинарным вектором размерности N.

Для расчета весовых коэффициентов применяется следующее выражение 1.21:

$$w_{ij} = \frac{1}{N} \sum_{d=1}^{m} X_{id} X_{jd}, \tag{1.21}$$

где N – размерность векторов, m – число запоминаемых выходных векторов, d – индекс запоминаемого выходного вектора, X_{ij} – i-я компонента запоминаемого выходного j-го вектора. Матрица взаимодействий является симметричной. Элементы на главной диагонали матрицы равны нулю.

Значение выхода i-го нейрона в текущий момент рассчитывается по следующей формуле 1.22:

$$y_i^{(t)} = sign(\sum_{j=1, j\neq i}^{N} w_{ji} y_j^{(t-1)}), \tag{1.22}$$

где $sign$ – функция, возвращающая знак аргумента.

Существует два режима работы сети Хопфилда:

1. Синхронный режим. При таком подходе все нейроны просматриваются последовательно, их состояния запоминаются и не меняются до тех пор, пока не будут обработаны все нейроны. Затем состояние всех нейронов синхронно обновляется.

2. Асинхронный режим. При таком режиме работы состояния нейронов обновляются последовательно, то есть для каждого нейрона поочередно вычисляется новое состояние, для каждого следующего нейрона новое состояние вычисляется с учетом всех изменений состояний рассмотренных ранее нейронов.

В асинхронном режиме работы невозможен динамический аттрактор, то есть вне зависимости от количества запомненных образов и начального состояния сеть придет к устойчивому состоянию.

На число образов, которые может запомнить сеть Хопфилда, накладывается ограничение 1.23:

$$M < \frac{N}{2 \log_2 N}, \tag{1.23}$$

где M – максимально число эталонов, которое может запомнить сеть, N – число нейронов.

Одним из недостатков сети является проблема ложных аттракторов: достижение устойчивого состояния сети не гарантирует правильный ответ.

1.5.2 Двунаправленная ассоциативная память

1.5.3 LSTM

1.6 Проблема переобучения нейронной сети

Проблема переобучения в нейронных сетях заключается в том, что модель запоминает данные только из обучающей выборки, не обобщая свои знания на новые, ранее не встречавшиеся данные. Это происходит из-за того, что модель адаптируется к обучающим примерам, вместо того, чтобы учиться классифицировать новые данные [16]. Признаком переобучения модели является существенно большее значение ошибки распознавания на тренировочной выборке, нежели на тестовой. Зачастую переобучение появляется из-за использования слишком сложных моделей, либо наборов данных, в которых вхождения похожи друг на друга [16].

Недообучение - это противоположная проблема переобучения нейронных сетей. Оно характеризуется тем, что алгоритм обучения не достигает удовлетворительной точности на обучающем множестве. Это может быть связано с тем, что выбрана слишком простая модель или недостаточно обучающих примеров. В результате модель не сможет классифицировать данные в более сложных случаях. [16].

Примеры недообученной, переобученной и оптимально обученной нейронной сети приведены на рисунке 1.7.

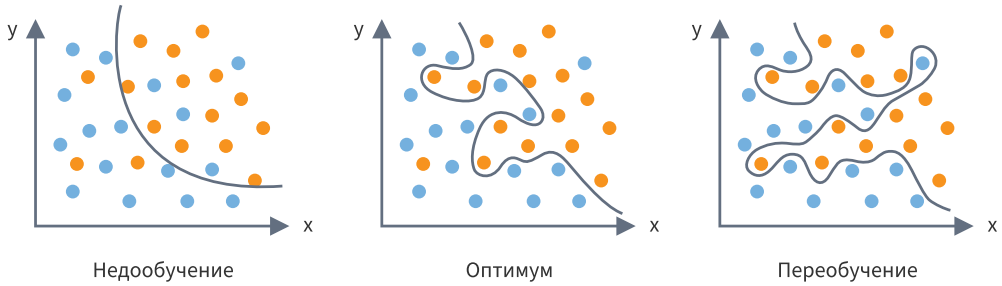

Рисунок 1.7 – Пример переобучения, недообучения и оптимального обучения

В этом примере нейронная сеть используется для разделения входного

множества объектов на два класса. В первом случае нейронная сеть является недообученной, так как вероятность ошибки равна 0.22 и может быть еще уменьшена за счет использование более сложной формы зависимости.

На примере по середине нейронная сеть строит общую зависимость для данного набора данных, не подгоняя значения под аномальные элементы для рыжего и голубого класса слева и справа соответственно.

Пример справа показывает переобученную нейронную сеть, которая строит зависимость, подгоняя параметры под аномальные параметры обучающей выборки.

Для борьбы с переобучением можно использовать следующие способы:

— аугментация обучающей выборки;

— метод раннего останова;

— регуляризация;

— батч нормализация.

1.6.1 Аугментация

Первый способ борьбы с переобучением – аугментация обучающей выборки. Аугментацией называется этап обучения нейронных сетей, состоящий в модификации обучающих изображений (поворот, масштабирование, зеркальное отражение и т. д.) по определенному правилу с целью расширить обучающую выборку и повысить ее разнообразие [17].

Существует три основных вида аугментации:

— геометрическая аугментация – изменение геометрических параметров изображения, таких как поворот, масштабирование, сдвиг и отражение;

— цветовая аугментация – изменение цветовых параметров изображения, таких как яркость, контрастность и насыщенность;

— добавление шума.

Аугментация первого типа обычно улучшает качество работы сверточных нейронных сетей, так как такие сети не инвариантны к масштабу, и изменение масштаба изображения значительно повышает разнообразие данных, позволяя сети обучаться на более разнообразных наборах данных [17]. В статье [18] описывается повышение точности распознавания нейронной сети на 10 процентов за счет использования аугментации масштаба.

Аугментации второго типа предполагают случайное изменение компонент R, G, B цвета пикселей изображения. Это один из самых эффективных методов аугментации данных, потому что нейросети без этой аугментации имеют тенденцию к заучиванию правил вида «сумма цветов пикселей в области». Также такая аугментация может улучшить способность распознавания сети при различных условиях освещенности [17].

Аугментация добавлением шума на изображение повышает устойчивость модели к шуму на реальных изображениях.

1.6.2 Метод раннего останова

Метод раннего останова после каждой эпохи обучения проверяет точность модели на обучающей и тестовой выборках. Обучение нейронной сети начинается при случайных значениях весов, и с каждой эпохой обучения точность на обучающей выборке будет повышаться, а на тестовой точность сначала будет расти, в момент, когда сеть будет достаточно обучена, зафиксируется на некотором значении, а потом начнет падать из-за переобучения модели.

Суть метода раннего останова заключается в отслеживании точности модели на тестовой выборке и остановке обучения в момент, когда она начинает расти.

К преимуществам данного метода можно отнести:

— сокращение времени обучения, так как нейронная сеть не будет обучаться, когда в этом уже нет необходимости;

— отсутствие дополнительных затрат на дополнение обучающей выборки.

1.6.3 Регуляризация

Метод регуляризации заключается в ограничение значений весовых коэффициентов нейронной сети, что делает их распределение более равномерным. Это достигается за счет добавления некоторого штрафа за увеличение весов нейронной сети в функцию потерь.

Существует три основных вида регуляризации [19]:

— L1 регуляризация, которая также называется Лассо регуляризацией, она добавляет штраф от суммы абсолютных значений весов модели;

— L2 регуляризация, которая также называется регуляризацией Тихонова, она добавляет штраф от суммы квадратов весов модели;

— дропаут, который случайным образом удаляет связи между нейронами.

В пером виде регуляризации новая функция потерь описывается выражением 1.24

$$L_{\text{new}} = L_{\text{old}} + \lambda \sum_{i=1}^{N} |w_{\text{i}}|, \tag{1.24}$$

где L_{new} – новое значение функции потерь, полученное после регуляризации, L_{old} – значение функции потерь до проведения регуляризации, λ – коэффициент штрафования весов, N – число весов в модели, w_{i} – значение i-го веса модели.

При коэффициенте λ равном нулю никакой регуляризации не будет и модель переобучится. При повышении коэффициента штрафования модель будет приближаться к оптимальной, то есть значение ошибки на тестовой выборке будет падать, но чем больше значение этого коэффициента, тем ближе значения всех весов будут к нулю, тем дальше модель отклоняется от локального минимума функции потерь до регуляризации и тем больше растет ошибка модели не тренировочной выборке. Таким образом, значение коэффициента λ должно подбираться экспериментально во время обучения. Чем меньше ошибка на тренировочной выборке, тем лучше подобран коэффициент штрафования.

В регуляризации Тихонова штраф считается не по сумме абсолютных

значений, а по сумме квадратов и выражается зависимостью 1.25

$$L_{\text{new}} = L_{\text{old}} + \lambda \sum_{i=1}^{N} w_{\text{i}}^2, \tag{1.25}$$

где все параметры аналогичны параметрам в выражении 1.24.

Главное различие между двумя методам заключается в том, что регуляризация Лассо уменьшает коэффициент менее важной характеристики до нуля, полностью удаляя ее из рассмотрения, а регуляризация Тихонова уменьшает веса, но не делает их равными нулю [19].

Еще одним видом регуляризации является дропаут. Суть метода заключается в том, что на каждой итерации обучения нейронной сети все связи между нейронами удаляются с некоторой вероятностью p. Иными словами это означает, что на каждой итерации обучения модели значение каждого веса w_{i} нейронной сети может быть на одну итерацию приравнено к нулю с некоторой вероятностью p.

Таким образом, регуляризация борется с проблемой переобучения нейронной сети и повышает ее обобщающую способность [19]. К недостатками регуляризации можно отнести то, что:

— добавление штрафа или удаление некоторых связей может привести к ухудшению точности модели на обучающих данных;

— нельзя заранее оптимальным образом определить коэффициент штрафования весов λ и вероятность удаления связи между нейронами p.

1.6.4 Нормализация

Обычно при обучении нейронной сети шаг градиентного спуска делается не по одному конкретному примеру, а сразу по некоторому набору обучающих примеров. Такой подход имеет следующие преимущества [19]:

— усреднение градиента по нескольким примерам представляет собой апроксимацию градиента по всему тренировочному множеству, и чем больше примеров используется в одном мини-батче, тем точнее это при-

ближение, использование всего обучающего множества невозможно в силу ограничений вычислительных ресурсов;

— в глубоких нейронный сетях к каждому примеру в отдельности требуется применить большое число последовательных операций, в случае использования некоторого набора обучающей примеров, можно выполнять эти последовательные операции в параллельном режиме для каждого примера в отдельности.

При таком подходе к обучению и использовании глубоких нейронных сетей возникает проблема, связанная с тем, что изменение распределения активаций выходов первых слоев на очередном шаге градиентного спуска приводит к сдвигу распределения данных во всех последующих слоях, что затрудняет их обучение и может ухудшить результаты. Для борьбы с этой проблемой используется пакетная нормализация, которая позволяет нормализовать выходы каждого слоя в процессе обучения. Это делает распределение данных более стабильным и уменьшает влияние сдвига распределения на последующие слои. Такая проблема получила название внутреннего сдвига переменных.

В исследованиях, приведенных в статье [20], говорится, что процесс обучения сходится быстрее, когда входы нейронной сети нормализованы, то есть их математическое ожидание приведено к нулю, а матрица ковариаций – к единичной. Если применять нормализацию к входам каждого слоя, то удастся избежать проблемы внутреннего сдвига переменных.

Для выполнения нормализации требуется предварительно рассчитать математическое ожидание и дисперсию элементов батча, которые определяются выражениями 1.26 и 1.27 соответственно.

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_{\text{i}}, \tag{1.26}$$

где μ – математическое ожидание элементов бача, N – размер бача, x_{i} – i-ый элемент бача.

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{\text{i}} - \mu)^2, \tag{1.27}$$

где σ – дисперсия элементов бача, N – размер бача, x_{i} – i-ый элемент бача, μ – значение математического ожидания, посчитанное по формуле 1.26.

Тогда нормализацию входов можно проводить, используя выражение 1.28

$$\hat{x_{\text{i}}} = \frac{x_{\text{i}} - \mu}{\sqrt{\sigma^2 + \epsilon}}, \tag{1.28}$$

где $\hat{x_{\text{i}}}$ – нормализованное значение i-го входа, x_{i} – ненормализованное значение i-го входа, μ и σ – математическое ожидание и дисперсия, посчитанные по формулам 1.26 и 1.27 соответственно, ϵ – некоторая константа, которая нужна для предотвращения деления на ноль.

Такая нормализация имеет существенный недостаток: в случае, если в качестве функции активации слоя используется сигмоидальная функция, например логистическая 1.2, то после нормализации нелинейность, которую давала эта функция активации, пропадет, так как большинство значений будут попадать в область, где эта функция ведет себя линейно, и функция активации фактически станет линейной [21].

Для того, чтобы компенсировать этот недостаток, слой нормализации должен быть способен в некоторых случаях практически никак не менять входные значения. Достигается это при помощи введения двух новый коэффициентов: коэффициент масштарбирования и сдвига нормализации. Итоговое выражение для слоя нормализации определяется зависимостью 1.29

$$y_{\text{i}} = \gamma_{\text{i}} \hat{x_{\text{i}}} + \beta_{\text{i}}, \tag{1.29}$$

где y_{i} – i-ый выход слоя нормализации, $\hat{x_{\text{i}}}$ – величина, полученная из выражения 1.28, γ_{i} и β_{i} – коэффициенты масштабирования и сдвига, которые настраиваются во время обучения модели.

Значения математического ожидания и дисперсии во время обучения от батча к батчу будут изменяться, но на этапе тестирования модели все изменяемые параметры должны быть зафиксированы. Для того, чтобы определить значения математического ожидания и дисперсии на этапе тестирования, эти величины накапливаются во время обучения с использованием экспоненциального скользящего среднего, которое определяется зависимостью 1.30

$$EMA_{\text{t}} = \alpha * x_{\text{t}} + (1 - \alpha) * EMA_{\text{t}-1}, \tag{1.30}$$

где EMA_{t} – значение экспоненциального скользящего среднего в точке t, $EMA_{\mathrm{t}-1}$ – значение экспоненциального скользящего среднего в точке t минус 1, причем значение экспоненциального скользящего среднего в нуле EMA_{t} равно x_0, x_{t} – значение исходной функции, в нашем случае это математическое ожидание или дисперсии, в момент времени t, α – коэффициент характеризующий скорость уменьшения весов, принимает значение от 0 и до 1, чем меньше его значение тем больше влияние предыдущих значений на текущую величину среднего.

В случае, когда входной батч описывается кортежем (N, C, H, W), где N – число элементов в батче, C – число каналов в каждом элементе, H и W – высота и ширина каждого изображения, нормализация считается по всем пикселям, всех изображений по каждому из каналов.

1.6.5 Вывод

Переобучение нейронных сетей – это проблема, когда сеть начинает обучаться на шум на тренировочной выборке. Это происходит, когда сеть становится слишком сложной и способна запомнить каждый пример из обучающего набора данных.

Для решения этой проблемы можно использовать различные методы, такие как регуляризация, которая вводит дополнительные ограничения на большие веса модели, нормализация, которая приводит значения входных признаков к одному диапазону, аугментация, которая позволяет увеличить обучающий набор данных, а также метод раннего останова, который отслеживает момент, когда модель начинает переобучаться, и прекращает обучение в этот момент.

1.7 Ансамблевые методы

Ансамблевые методы классификации основаны на том, что несколько классификаторов обучаются на одном и том же наборе обучающих данных, а затем их прогнозы объединяются для классификации элементов тестового

набора данных. Математическим обоснованием этой идеи служит теорема Кондорсье о жюри присяжных [22].

Классификатор называется слабым, если его ошибка на обучающей выборке менее 50 процентов, но больше нуля. Тогда, объединив предсказания нескольких таких классификаторов, можно достичь большей точности классификации на элементах тестовой выборки [22].

Выделяют 3 основных метода ансамблевой классификации [22]:

— бэггинг;

— бустинг;

— стекинг.

Идея бэггинга состоит в том, что если размер обучающей выборки не велик, то можно создать много случайных выборок из исходной путем отбора некоторых элементов, и обучить слабые классификаторы на эти подвыборки. Таким образом, каждая модель имеет свой набор обучающих примеров и старается сделать предсказания на основе своего подмножества данных. Затем результаты всех моделей комбинируются для получения итоговых предсказаний.

Бустинг – это процедура последовательного построения композиции алгоритмов машинного обучения, когда каждый следующий алгоритм стремится компенсировать недостатки композиции всех предыдущих алгоритмов [22]. Таким образом, при бустинге каждая последующая модель обучается на ошибках предыдущей и старается их компенсировать, повышая точность классификации общей модели.

Идея стекинга заключается в введении некоторого алгоритма классификации и его обучении. При стекинге, в отличие от бустинга и бэггинга, классификаторы должны быть разной природы [22]. Обучение модели при стекинге можно свести к следующим трем шагам:

— обучающая выборка разбивается на две непересекающихся подвыборки;

— первая подвыборка используется для обучения классификаторов;

— вторая для обучения алгоритма, который на вход принимает выходы со всех классификаторов.

Главным недостатком стекинга является деление обучающей выборки на две части.

1.8 Формализованная постановка задачи

Цель работы – разработать метод распознавания летательных аппаратов с аэрофотоснимков.

Для достижения поставленной цели требуется выполнить следующие задачи:

— провести анализ и сравнение существующих методов распознавания летательных аппаратов с аэрофотоснимков;

— разработать метод распознавания летательных аппаратов с использованием нейронных сетей;

— разработать программное обеспечение, реализующее метод распознавания летательных аппаратов;

— исследовать характеристики разработанного метода и влияние на них различных подходов к обучению.

На вход методу подается изображение со снимком аэропорта, где находятся летательные аппараты 20 различных классов. Результатом работы метода является распознанные на изображении самолеты и их модели. Требуемая точность работы метода на тестовом наборе данных должна составлять не менее 75 процентов.

На входное изображение накладываются следующие ограничения:

— изображение в формате PNG или JPEG;

— изображение сделано в дневное время суток;

— размер изображения 800 на 800 пикселей;

— размер самолетов на изображении более 70 пикселей.

На рисунке 1.8 приведена IDEF-0 диаграмма уровня A0 метода распознавания летательных аппаратов с аэрофотоснимков с использованием нейронных сетей.

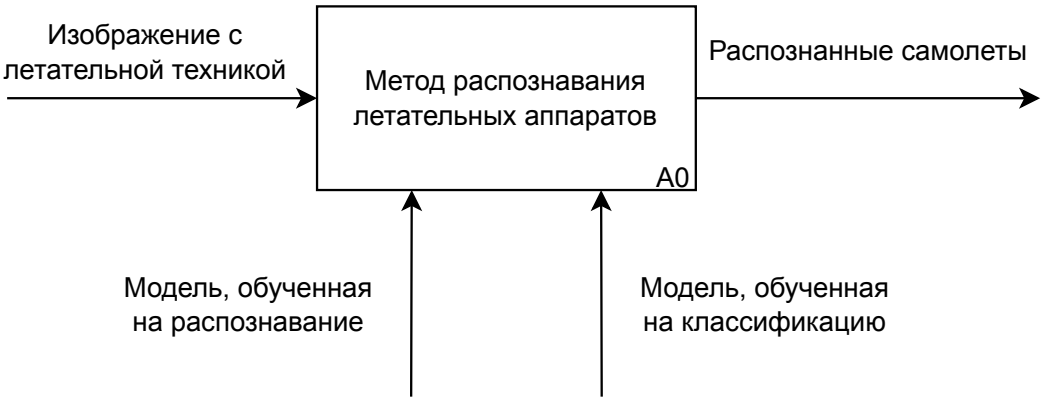

Рисунок 1.8 – IDEF-0 диаграмма метод распознавания летательных аппаратов с аэрофотоснимков

1.9 Вывод

Методы детерминированного подхода плохо подходят для поставленной задачи в силу того, что не устойчивы к шумам.

Дерево решений не применимо для поставленной задачи, так как на этапе построения дерева надо выделять критерии, по которым нужно классифицировать входную информацию.

Среди нейрокомпьютерных алгоритмов выделяют алгоритмы обучения с учителем и без. Задача распознавания объектов на изображении относится к алгоритмам обучения с учителем и может быть решена с использованием нейронных сетей.

Среди основный принципов построения нейронных сетей: перцептрон, сверточные сети и капсульные сети. Большую точность классификации показывают сверточные и капсульные сети. Сверточные нейронные сети обучаются быстрее и требуют меньше обучаемых данных, чем капсульные, а также устойчивы к шумам.

Таким образом, для решения задачи распознавания летательных аппаратов с аэрофотоснимков, лучше всего подходят сверточные нейронные сети за счет большей точности распознавания в сравнении с перцептроном, а

также за счет устойчивости к шуму и большей скорости обучении по сравнению с капсульными сетями. При обучении сверточной нейронной сети нужно использовать регуляризацию и нормализацию для борьбы с проблемой переобучения, а также выполнять аугментацию обучающей выборки для повышения обобщающей способности модели. В случае слабых классификаторов, нужно использовать методы ансамблевой обработки: бустинг, бэггинг и стэкинг.

Среди сверточных нейронных сетей для решения задачи классификации лучше всего подходит SimpLeNet за счет меньших требований к вычислительным ресурсам, большей скорости обучения и меньшим объемом требуемых обучающих данных по сравнению с другими сетями, а для решения задачи детектирования объектов – Yolo.

2 Конструкторский раздел

2.1 Особенности входных данных

На вход метода поступает изображение, на которое накладываются следующие ограничения:

— изображение в формате PNG или JPEG;

— размер изображения 800 на 800 пикселей;

— размер самолетов на изображении больше 70 пикселей;

— изображение сделано под углом 90 градусов к поверхности Земли.

Результатом работы метода являются распознанные самолеты, а именно их количество, расположение на входном изображении и их модели.

Программное обеспечение должно предоставлять интерфейс для разработанного метода со следующими функциями:

— возможность выбора и использования одной из заранее обученных моделей;

— возможность загрузки аэрофотоснимка с летательной техникой и получение информации о распознанных на нем летательных объектах, а именно их количество, местоположение и модели;

— возможность загрузки изображения с одним самолетом и получение результатов о классе летательной техники на нем.

2.2 Проектирование метода

IDEF-0 диаграмма метода распознавания летательной техники с аэрофотоснимков уровня A1 приведена на рисунке 2.1.

Рисунок 2.1 – IDEF-0 диаграмма уровня A1

Решение задачи распознавания самолетов на аэрофотоснимках было поделено на две подзадачи: детектирование объектов на изображении и их классификация.

В качестве модели классификации была выбрана сверточная нейронная сеть из 12 сверточных слоев, выходы которой соединены с двухслойным перцептроном.

Обучаемая модель имеет следующую структуру:

— слой свертки с 64 фильтрами, размером ядра 3 на 3 пикселя, шагом и отступом по одному пикселю;

— три аналогичных слоя с 128 фильтрами;

— слой max pooling, с размером ядра 2 на 2 пикселя и шагом 2, размер выходных матриц после этого слоя становится 48 на 48;

— два слоя с 128 фильтрами, размером ядра 3 на 3 пикселя, шагом и отступом по одному пикселю;

— аналогичный слой с 256 фильтрами;

— слой max pooling, с размером ядра 2 на 2 пикселя и шагом 2, размер выходных матриц после этого слоя становится 24 на 24;

— слой свертки с 256 фильтрами, размером ядра 3 на 3 пикселя, шагом и отступом по одному пикселю;

— слой max pooling, с размером ядра 2 на 2 пикселя и шагом 2, размер выходных матриц после этого слоя становится 12 на 12;

— слой свертки с 512 фильтрами, размером ядра 3 на 3 пикселя, шагом и отступом по одному пикселю;

— слой max pooling, с размером ядра 2 на 2 пикселя и шагом 2, размер выходных матриц после этого слоя становится 6 на 6;

— слой свертки с 2048 фильтрами, размером ядра 3 на 3 пикселя, шагом и отступом по одному пикселю;

— слой свертки с 256 фильтрами, размером ядра 3 на 3 пикселя, шагом и отступом по одному пикселю;

— слой max pooling, с размером ядра 2 на 2 пикселя и шагом 2, размер выходных матриц после этого слоя становится 3 на 3;

— слой свертки с 256 фильтрами, размером ядра 3 на 3 пикселя, шагом один пиксель;

— входной слой перцептрона, состоящий из 256 нейронов;

— выходной слой перцептрона, состоящий из 20 нейронов.

После каждого сверточного слоя применяется слой нормализации, описанный в главе 1.6.4 и функция активации ReLU. Такая архитектура нейронной сети имеет следующие преимущества:

— использование сверточных слоев позволяет выделять различные признаки, такие как границы, формы, вне зависимости от их расположения в входном изображении;

— использование пуллинговых слоев позволяет уменьшить размерность изображения с сохранением отличительных признаков;

— несмотря на количество слоев, такая архитектура за счет использования пуллинговых слоев и сверток с ядром 3 на 3 вместо 5 на 5 и больших имеет относительно немного обучаемых параметров: 4910356 в сравнении с шестьюдесятью миллионами в архитектуре AlexNet, которая была описана в главе **??**.

Использование меньшего числа параметров приводит к уменьшению скорости обучения нейронной сети, а также к меньшим ограничениям на вычислительные ресурсы при использовании модели, что важно, так как все изображения, которые поступают на вход модели, сделаны с беспилотных летательных аппаратов и могут на них же обрабатываться.

Для решения задачи детектирования объектов была выбрана нейронная сеть Yolo v3, состоящая из 106 сверточных слоев, среди которых сверточный слои с размером ядра 3 на 3 и 1 на 1 и шагом свертки 1 или 2 пикселя, а также блоки, в которых используется техника пропуска соединений.

Всего такая сеть выдает три матрицы предсказаний для размеров ячеек 25, 50 и 100 пикселей.

Схемы алгоритмов обучения нейронных сетей и прохождения одной эпохи приведены на рисунках 2.2 и 2.3 соответственно.

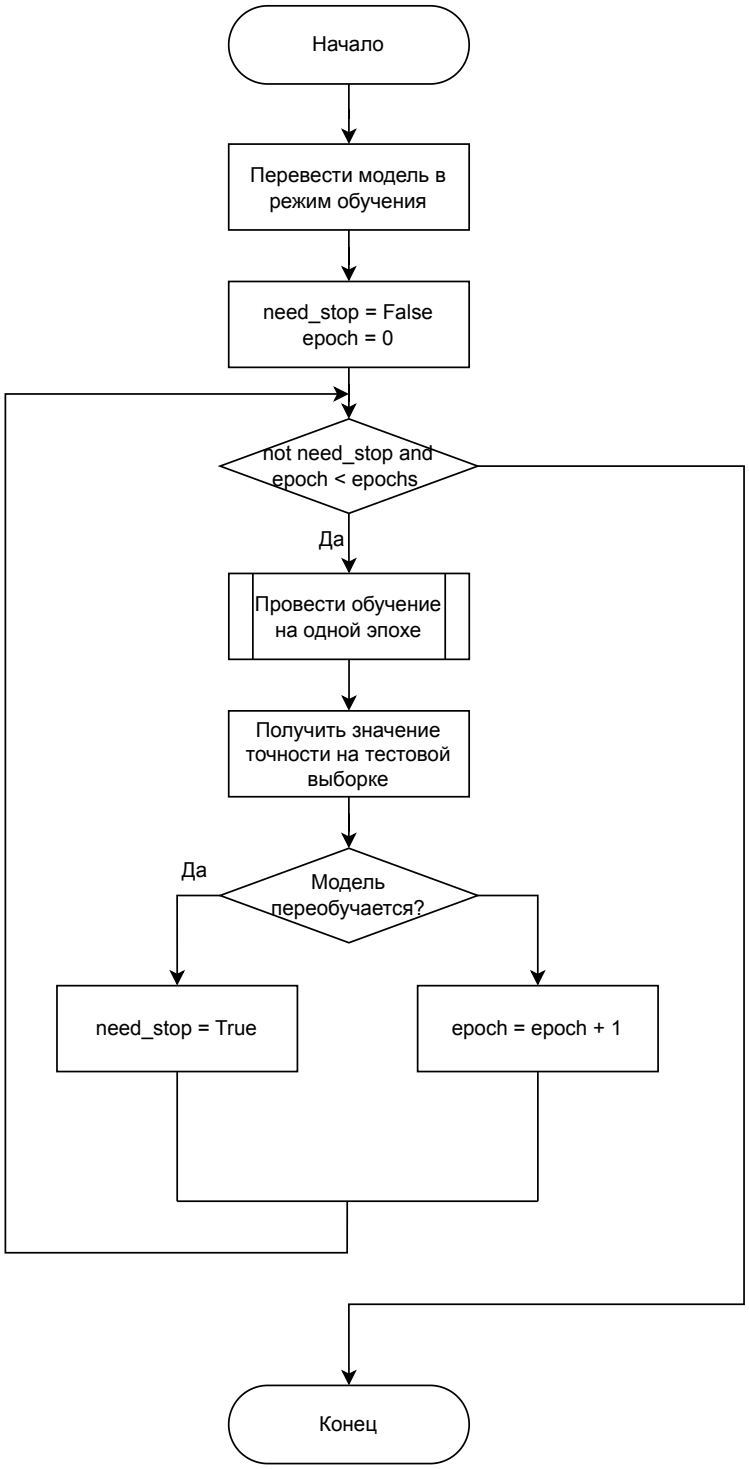

Рисунок 2.2 – Схема алгоритма обучения нейронной сети

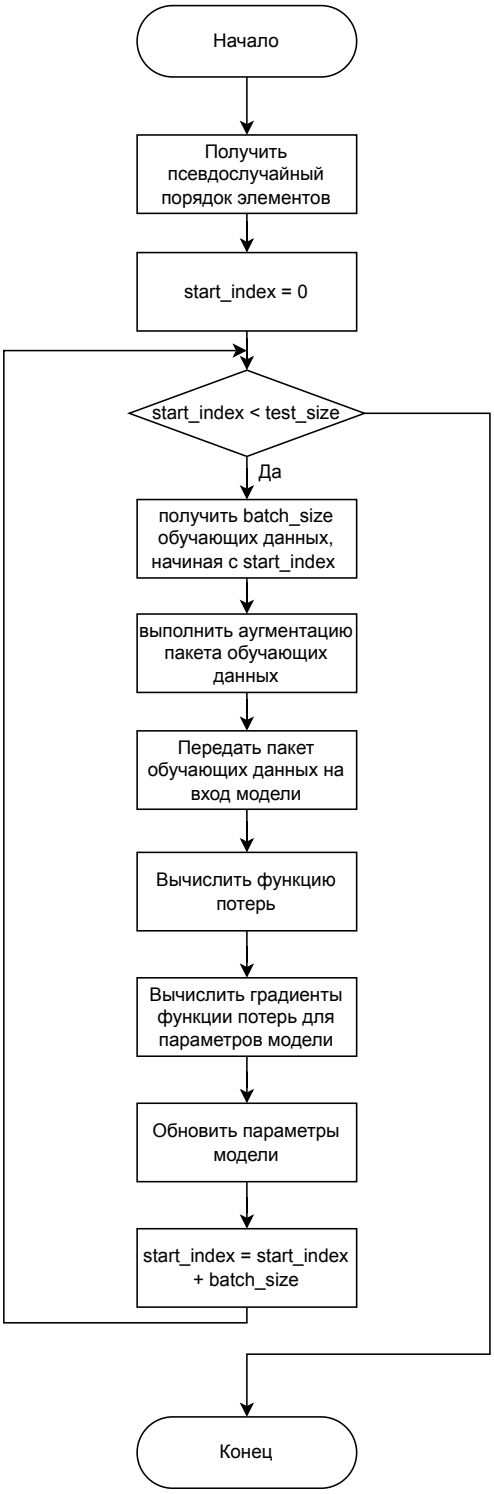

Рисунок 2.3 – Схема алгоритма прохождения одной эпохи

При подсчете точности модели на тестовой выборке важно учитывать ее размер и загружать в оперативную память по частям, каждый раз освобождая выделенные ресурсы.

2.3 Структура программного обеспечения

Программное обеспечение состоит из четырех модулей:

— модуль, реализующий модель нейронной сети детектирования летальной техники;

— модуль, реализующий модель нейронной сети классификации летальной техники;

— модуль, реализующий интерфейс взаимодействия с пользователем;

— модуль пользовательского интерфейса.

Структурная схема взаимодействия модулей разрабатываемого программного обеспечения представлена на рисунке **??**.

Модули детектирования и классификации используются для обучения моделей и взаимодействия с уже обученными. После завершения обучения параметры модели должны быть сохранены в файл для обеспечения в дальнейшем загрузки модели без обучения.

Модуль взаимодействия с пользователем должен обеспечивать следующие возможности:

— загрузка одной из заранее обученных моделей;

— загрузка изображения с летательной техникой и определение ее класса с помощью загруженной или обученной модели;

— загрузка аэрофотоснимка с различными классами летательной техники и определение их классов.

2.4 Набор обучающих данных

Для обучения нейронной сети был выбран набор данных из источника [28], состоящий из 3842 снимков аэропортов, сделанных с беспилотных летательных аппаратов. На этих снимках находится 22341 самолет 20 различных типов.

Пример элемента обучающей выборки приведен на рисунке **??**.

Вся выборка была поделена на обучающую и тестовую. Обучающая выборка содержит 20344 самолета, а тестовая – 1997.

2.5 Вывод

В данном разделе были определены ограничения, которые накладываются на входное изображение, и требования, которые предъявляются к разрабатываемому программному обеспечению.

Была детализирована IDEF-0 диаграмма уровня A0, описанная в разделе формализованной постановки задачи, а также было проведено разбиение программного обеспечения на модули и описание функциональных требований, которые они должны обеспечивать. Задача распознавания самолетов была разбита на две подзадачи: детектирование объектов на изображении и их классификация.

Были определены архитектуры нейронных сетей классификации и детектирования, а также преимущества использования именно таких архитектур. Была определена схема алгоритма обучения составленных нейронных сетей.

Была найдена выборка данных, которые будут использоваться для обучения моделей, и было произведено ее деление на две части: обучающую и тестовую.

3 Технологический раздел

3.1 Средства реализации программного обеспечения

В качестве языка программирования был выбран Python. Данный выбор обусловлен тем, что Python имеет множество библиотек, таких как TensorFlow, Keras, PyTorch и Theano, которые предоставляют множество инструментов для создания и обучения нейронных сетей.

В качестве библиотеки для создания нейронной сети была выбрана библиотека PyTorch версии 2.0.0, так как она имеет следующие возможности и инструменты:

— динамический граф вычислений, использование которого облегчает отладку моделей;

— возможность переноса вычислений на GPU;

— набор инструментов для создания различных слоев, из которых складывается архитектура нейронной сети;

— API на языке C++, что позволяет обучить модель с использованием интерпретируемого языка Python, а использовать ее на компилируемом языке C++.

Для работы с большими данными была выбрана библиотека numpy версии 1.21.0, так как она использует оптимизированный код на C, что позволяет выполнять вычисления быстрее, чем с использованием чистого Python, а также потому что классы этой библиотеки интегрируются с библиотекой PyTorch, которая используется для создания нейронной сети.

Для работы с изображениями была выбрана библиотека Pillow версии 8.2.0, так как она позволяет загружать и трансформировать изображения разных форматов, таких как PNG, JPEG, BMP, а также переводить изображения в объекты, которые передаются на вход нейронной сети.

Для создания графического интерфейса использовалась библиотека PyQt версии 5.0, так как она является кроссплатформенной, имеет базовые виджеты, на которых может быть построен интерфейс, а также предоставляет возможность подписки на события, которые генерируются виджетами.

3.2 Реализация программного комплекса

В качестве модели классификации используется сверточная нейронная сеть с 12 слоями свертки и 5 слоями пуллинга, которая соединяется с двухслойным перцептроном. После каждого сверточного слоя за исключением последнего используется слой нормализации и функция активации ReLU. Реализация модели приведена в листинге А.1 (приложение А).

Перед обучением модели выполняется предобработка всего обучающего набора данных, во время которой нужно разбить все входные изображения по классам, привести все изображения к размеру 96 на 96 пикселей, так как только такое изображение обрабатывается моделью, а также поделить весь набор данных на обучающую и тестовую выборки.

После выполнения предобработки нужно произвести аугментацию для расширения обучающей выборки. К каждому изображению в обучающем множестве применяются следующие преобразования: поворот на 30 и 330 градусов, увеличение яркости в полтора раза, а также гауссово размытие, которое создает фильтр Гаусса с заданным размером и силой размытия, и применяют его к изображению, используя операцию свертки. Коэффициенты в фильтре Гаусса вычисляются по формуле 3.1

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}, \tag{3.1}$$

где x и y – расстояния от центра ядра по горизонтали и вертикали соответственно, σ – сила размытия.

Исходное изображения и изображения после аугментации приведены на рисунках **??**-**??**.

Обучение модели проводилось на машине с процессором Intel Core i9-10900, 64 гигабайтами оперативной памяти и графической картой NVIDIA GeForce RTX 3080 с 16 гигабайтами памяти типа GDDR6. Время прохождения одной эпохи для модели классификации в среднем занимает 5 минут 4 секунды, а для модели детектирования – 6 минут 15 секунд.

В качестве оптимизатора функции потерь был выбран Adam, так как он автоматически адаптирует скорость обучения для каждого параметра в зависимости от его градиента, что позволяет более эффективно использовать

скорость обучения и ускоряет сходимость.

Обучение модели останавливается, когда точность распознавания на тестовой выборке после прохождения очередной эпохи становится меньше, чем на двух предыдущих, так как это свидетельствует о том, что сеть начала переобучаться.

Код обработки и загрузки обучающего набора данных представлен в листинге А.2. Реализация алгоритмов обучения и прохождения одной эпохи представлены на листингах А.3 и А.4 соответственно. Все листинги находятся в приложении А.

Графики зависимостей точности модели классификации на тестовой и обучающей выборках от номера эпохи обучения приведены на рисунке 3.1.

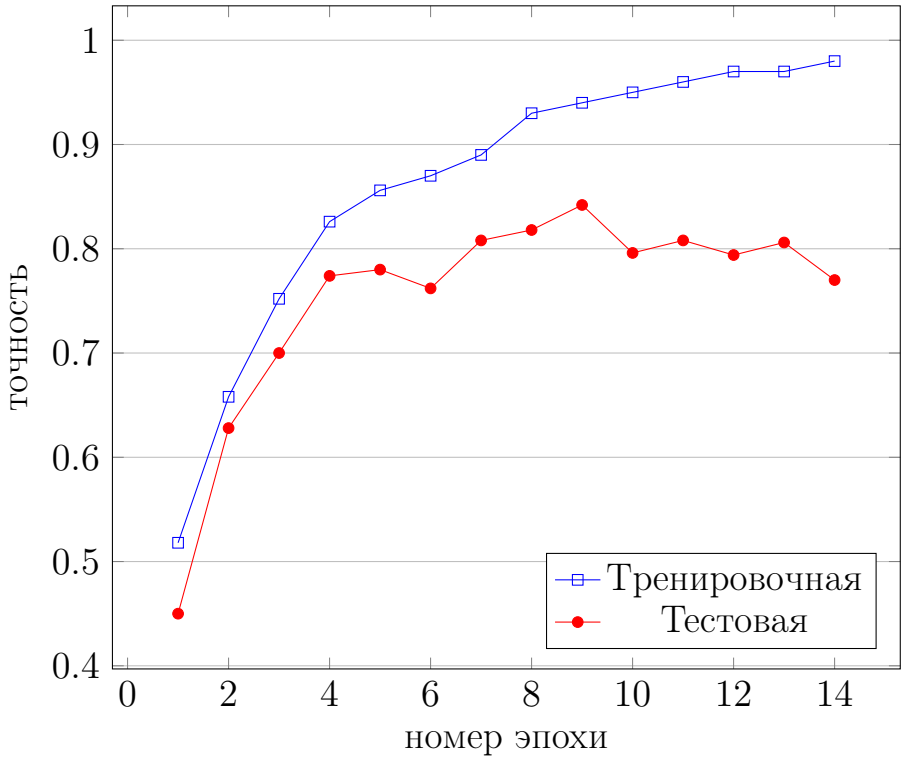

Рисунок 3.1 – Обучение модели

Из графиков видно, что после 9 эпохи модель начинает переобучаться и требуется остановить обучение. Итоговая точность полученной модели на тестовой выборке составила 84 процента.

В качестве модели детектирования была реализована сверточная нейронная сеть, состоящая из 106 слоев. В качестве функции активации была выбрана функция ReLU. Разработанная модель имеет три выходных слоя, каждый из которых имеет размерность (3xSxSx5), где 3 – число якорей, S –

размер ячейки, для 3 выходов размеры ячеей будут 25, 50 и 100 соответственно, вектор из пяти элементов состоит из вероятности нахождения центра объекта внутри ячейки, а также координат его центра, высоты и ширины рамки.

Графики зависимости точности детектирования объектов и вероятности ложного срабатывания представлены на рисунке 3.2. Объект считался корректно распознанным, если отношение площади пересечения полученной рамки с эталоном к площади их объединения составляло больше 0.6. Детектированный объект считался ложным, если отношение площади пересечения его рамки ни с одной из эталонных к площади их объединения не составляло больше 0.1.

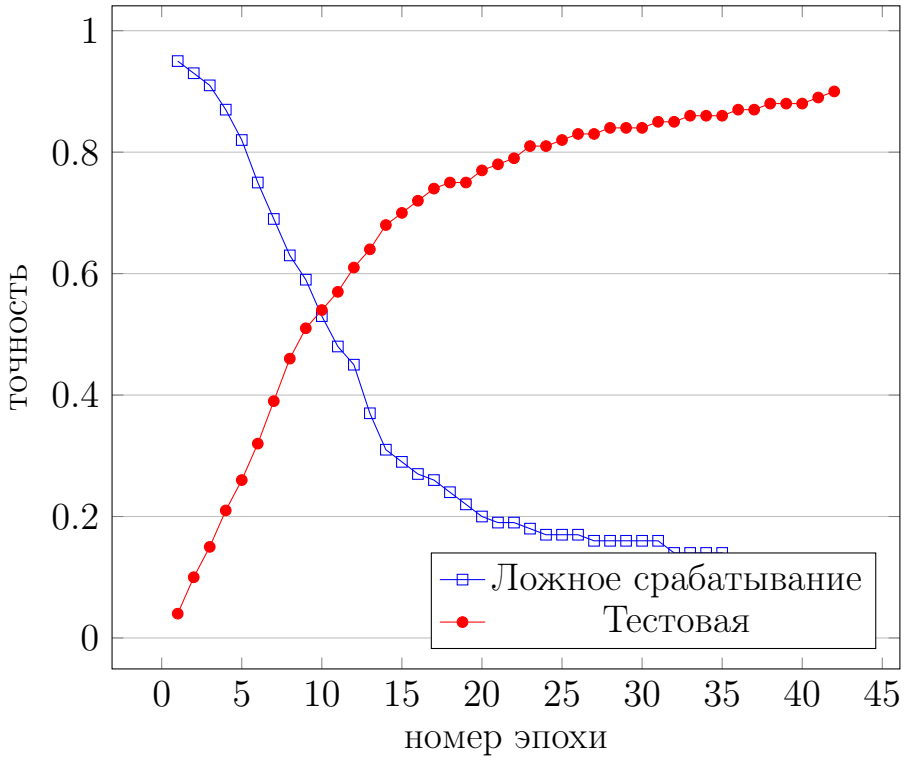

Рисунок 3.2 – Обучение модели

3.3 Взаимодействие с разработанным ПО

Взаимодействие с разработанным программным обеспечение осуществляется через графический пользовательский интерфейс. Интерфейс приложения представлен на рисунке **??**.

Графический интерфейс поделен на две части. Первая часть позволяет

выбрать и загрузить модель обученную на классификацию и проверить ее работу. Вторая позволяет выбрать и загрузить модель, обученную на детектирование, а также проверить ее работу вместе с моделью классификации.

3.4 Тестирование

3.5 Вывод

В рамках данного раздела были выбраны и реализованы средства распознавания летательных аппаратов. Для этого были выбраны методы аугментации обучающих данных, направленные на увеличение количества и разнообразия данных для обучения модели.

Были созданы модели для детектирования и классификации летательных аппаратов и реализован алгоритм их обучения. После этого были проведены тесты для оценки качества моделей. Итоговая точность модели классификации на тестовой выборке составила 84 процента. Точность детектирования на тестовой выборке составила 90 процентов.

Кроме того, был создан графический интерфейс, который позволяет пользователям использовать обученную модель для распознавания типов летательных аппаратов. После этого было проведено ручное тестирование обученных моделей, чтобы убедиться в правильности их работы.

4 Исследовательский раздел

4.1 Подбор параметров сети

4.2 Сравнение с аналогами

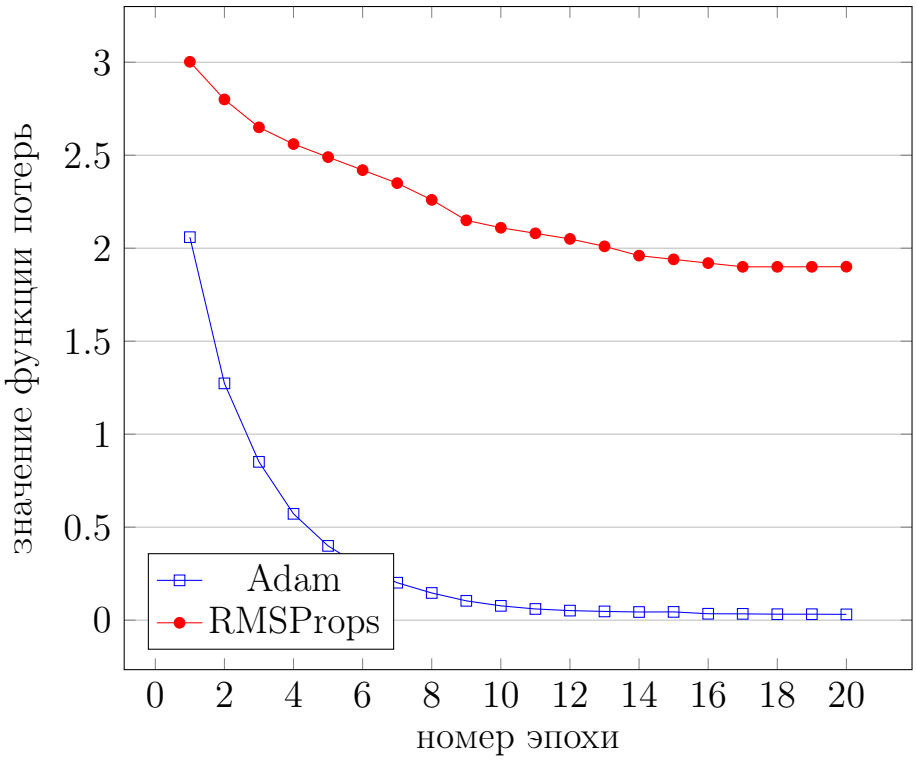

Рисунок 4.1 – Сравнение функций потерь моделей, обученных на RMPSProp и Adam

Из графиков видно, что функция потерь, которая оптимизировалась с помощью алгоритма Adam сходится быстрее и не попадает в локальные минимумы. Итоговая точность модели, которая обучалась с помощью алгоритма Adam, составила 84 процента на тестовой выборке.

После каждого из слоев пуллинга в архитектуре модели классификации был добавлен слой дропаут, который с некоторой заданной вероятностью p исключает нейрон из обучения на одну итерацию. Такой прием приводит к тому, что модель обучается на разных комбинациях активных нейронов, что уменьшает вероятность переобучения модели.

Графики зависимости точности модели на тестовой выборке от номера эпохи обучения с использованием dropout и без приведены на рисунке 4.5.

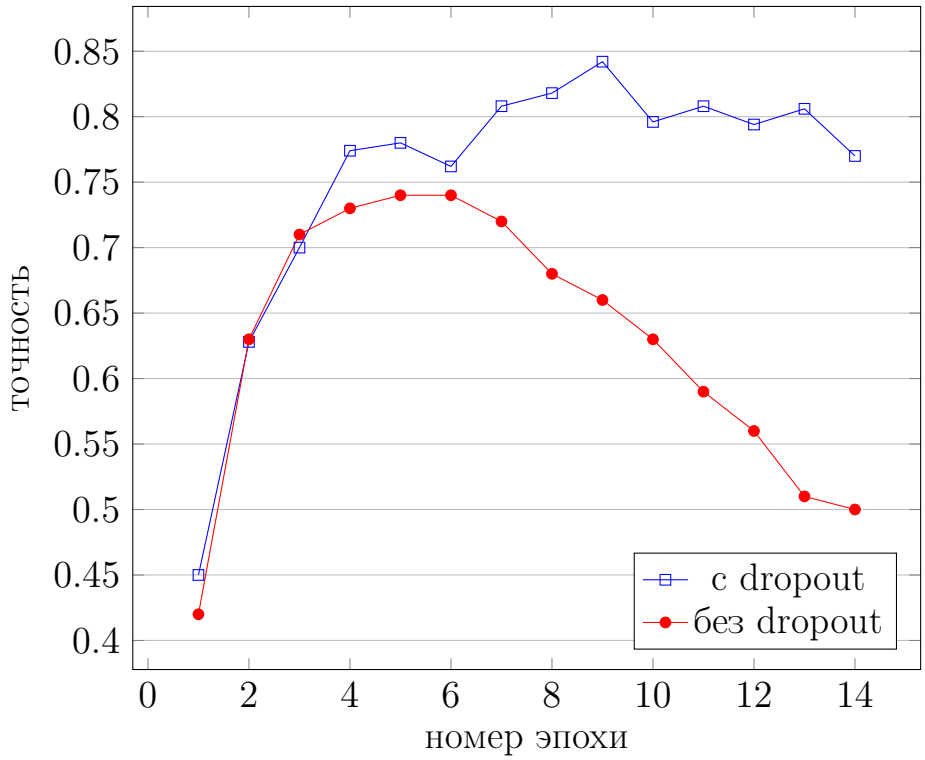

Рисунок 4.2 – Сравнение моделей, обученных с использованием dropout и без, на тестовой выборке

Из графиков видно, что после 6 эпохи обучения, модель которая обучалась без использования dropout стала обучаться на шум на тренировочной выборке, в связи с чем точность модели на тестовой выборке начала падать.

Помимо добавления слоев дропаута из архитектуры сети был убран один из сверточных слоев. Точность модели на тестовой выборке после этого осталась прежней, а число обучаемых параметров было уменьшено на 600 тысяч. Уменьшение число обучаемых параметров увеличивает скорость обучения сети, снижает ограничения на вычислительные ресурсы, а также уменьшает вероятность переобучения модели.

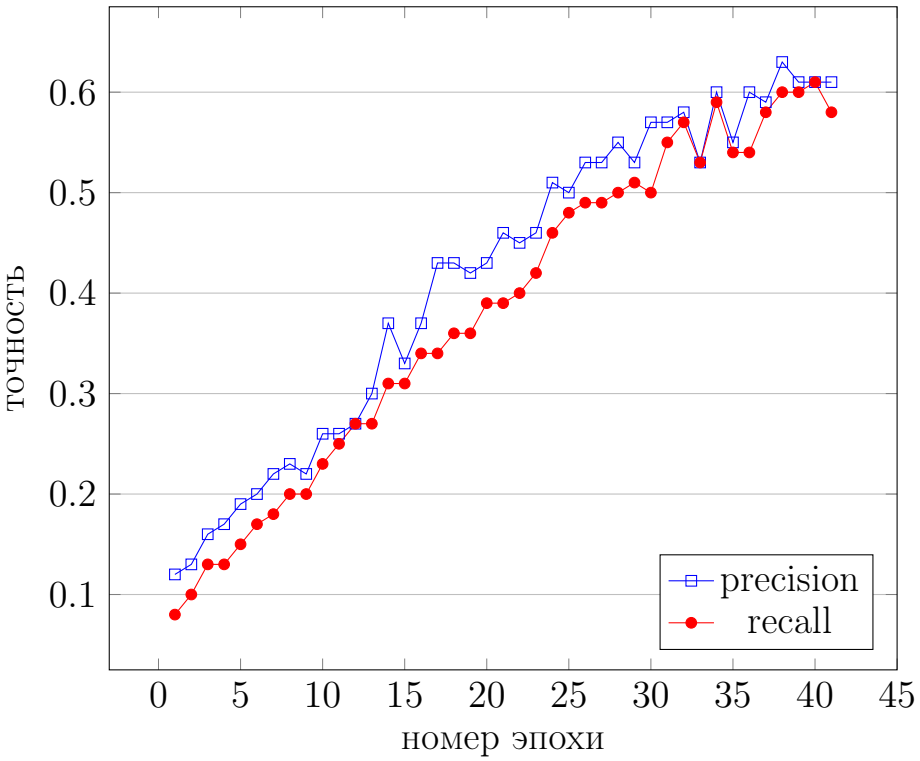

Рисунок 4.3 – Precision и recall yolov3

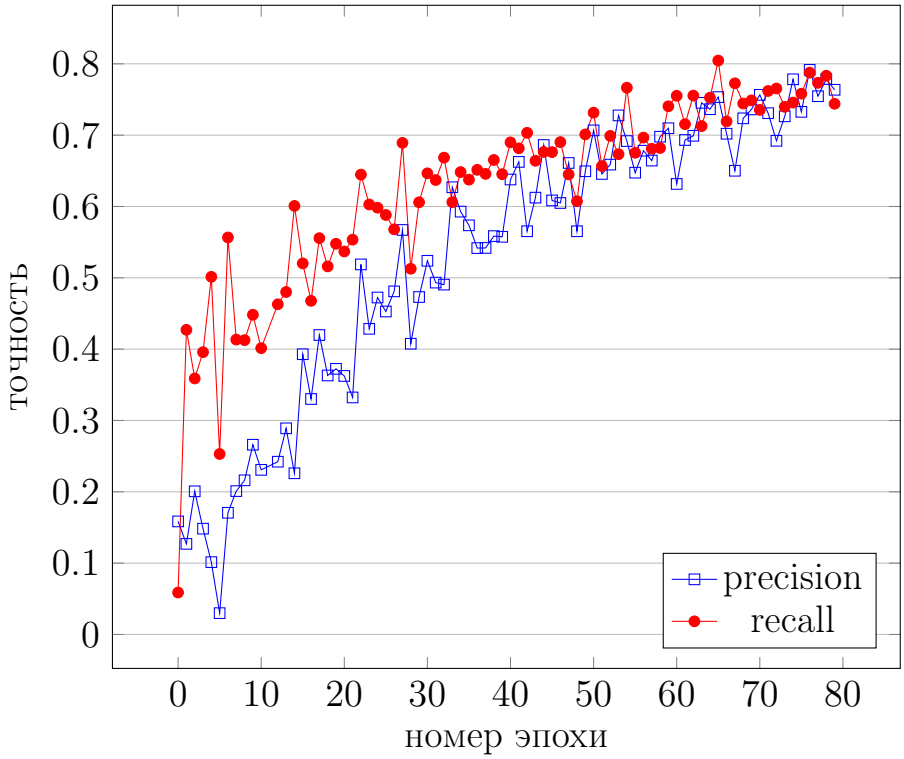

Рисунок 4.4 – Precision и recall yolov7

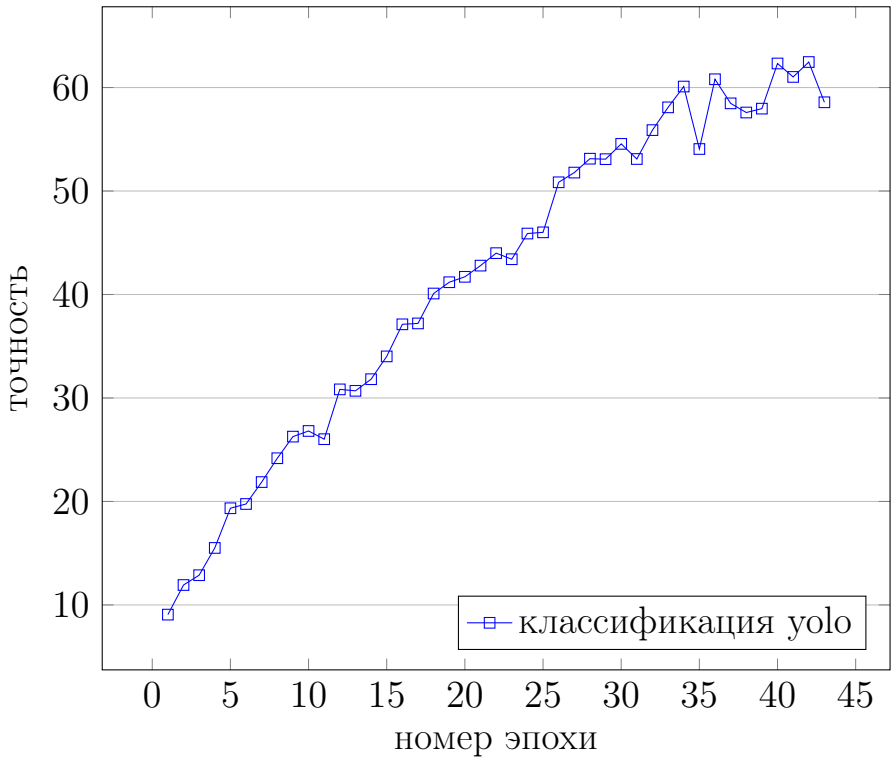

Рисунок 4.5 – Сравнение моделей, обученных с использованием dropout и без, на тестовой выборке

4.3 Вывод

В архитектуру нейронной сети были внесены изменения, которые увеличивают скорость обучения модели, снижают ограничения на вычислительные ресурсы, а также решают проблему переобучения модели.

При сравнении различных оптимизаторов было выяснено, что скорость обучения сети должна адаптивно настраиваться в зависимости от текущего значения ошибки по правилу: чем больше ошибка, тем больше скорость обучения. В ином случае модель может дольше сходиться при малом значении скорости обучения, либо расходиться при больших значениях этого параметра.

Алгоритмы Adam и RMSProps используются для адаптивной настройки скорости обучения. В RMSprops учитывается экспоненциальное скользящее среднее по квадрату градиента для учета истории обучения при обновлении весов модели. В алгоритме Adam также учитывается экспоненциальное скользящее среднее и по первому моменту, что позволяет ему достичь большей сходимости при обучении модели.

Таким образом, разработанный метод показывает наибольшую точность при обучении моделей с помощью алгоритма Adam.

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы был спроектирован и разработан метод распознавания летательных аппаратов с аэрофотоснимков с использование нейронных сетей. В ходе выполнения работы были выполнены следующие задачи:

— проведен анализ и сравнение существующих методов распознавания летательных аппаратов с аэрофотоснимков;

— разработан метод распознавания летательных аппаратов с использованием нейронных сетей;

— разработано программное обеспечение, реализующее метод распознавания летательных аппаратов;

— исследованы характеристики разработанного метода и влияние на них различных подходов к обучению.

Цель работы достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Peiquan Y.* Learned buffer replacement for database systems // Proceedings of the 2022 5th International Conference on Data Storage and Data Engineering. — 2022. — С. 18—25.
2. *Shaik B.* PostgreSQL Configuration: Best Practices for Performance and Security. — Apress, 2020.
3. *Бабушкина Н. Е.* ВЫБОР ФУНКЦИИ АКТИВАЦИИ НЕЙРОННОЙ СЕТИ В ЗАВИСИМОСТИ ОТ УСЛОВИЙ ЗАДАЧИ // Донской государственный технический университет. — 2022. — С. 4.
4. *Антонов Г. В.* ПРОСТАЯ НЕЙРОННАЯ СЕТЬ И ЕЕ ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ // ФГБОУ ВО Великолукская государственная сельскохозяйственная академия. — 2021. — С. 11.
5. *Левченко К. М.* Нейронные сети // Белорусский государственный университет информатики и радиоэлектроники. — 2022. — С. 5.
6. *Барвинский Д. А.* Применение метода градиентного спуска в решении задач оптимизации // Тенденции развития науки и образования. — 2021. — С. 61.
7. *Апарнев А. Н.* Анализ функций потерь при обучении сверточных нейронных сетей с оптимизатором Adam для классификации изображений // ВЕСТНИК МОСКОВСКОГО ЭНЕРГЕТИЧЕСКОГО ИНСТИТУТА. ВЕСТНИК МЭИ. — 2020. — С. 90.
8. *Митина О.* ПЕРЦЕПТРОН В ЗАДАЧАХ БИНАРНОЙ КЛАССИФИКАЦИИ // Национальная ассоциация ученых. — 2021. — С. 6.
9. *Minsky M.* Perceptrons: An Introduction to Computational Geometry. — MIT Press, 1969.
10. *Hecht-Nielsen R.* Kolmogorov's Mapping Neural Network Existence Theorem. — 1987.
11. *Колмогоров А. Н.* О представлении непрерывных функций нескольких переменных в виде суперпозиции непрерывных функций одного переменного и сложения // Доклады академии наук СССР. — 1957. — Т. 114, № 5. — С. 953—956.

12. *Baum E.* What Size Net Gives Valid Generalization? // Neural Computation. — 1989. — Т. 1, № 1. — С. 151—160.

13. *Jiang Q.* An efficient multilayer RBF neural network and its application to regression problems // Neural computing and Applications. — 2022. — С. 1—18.

14. *Дель И. В.* Прогноз приземной температуры воздуха на основе модели рекуррентной нейронной сети // Сборник статей Всероссийской молодежной научной конференции студентов. — 2021.

15. *Ромасенко А. А.* ЗАПОМИНАНИЕ И ОТОБРАЖЕНИИ ОБРАЗОВ НА ОСНОВЕ НЕЙРОННОЙ СЕТИ ХОПФИЛДА // Всероссийская научно-методическая конференция, посвященная 70-летию Оренбургского государственного университета. — 2022. — С. 1384—1392.

16. *Воронецкий Ю. О.* Методы борьбы с переобучением искусственных нейронных сетей // Научный аспект. — 2019. — Т. 13, № 2. — С. 1639—1647.

17. *Парасич А. В.* Формирование обучающей выборки в задачах машинного обучения. Обзор // Информационно-управляющие системы. — 2021. — С. 61—70.

18. *Cai Z.* Cascade R-CNN: High Quality Object Detection and Instance Segmentation // IEEE Transactions on Pattern Analysis and Machine Intelligence. — 2021. — Т. 43, № 5. — С. 1483—1498.

19. *Пырнова О.* МЕТОДЫ И ПРОБЛЕМЫ ПЕРЕОБУЧЕНИЯ МНОГОСЛОЙНОЙ НЕЙРОННОЙ СЕТИ // ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ В СТРОИТЕЛЬНЫХ, СОЦИАЛЬНЫХ И ЭКОНОМИЧЕСКИХ СИСТЕМАХ. — 2020. — С. 101—103.

20. *Lecun Y.* Efficient Backprop // Neural Networks: Tricks of the Trade. — 2019. — С. 99—48.

21. *Николенко С.* Глубокое обучение. Погружение в мир нейронных сетей // Издательство Питер. — 2018. — С. 477.

22. *Кашницкий Ю. С.* Ансамблевый метод машинного обучения, основанный на рекомендации классификаторов // Интеллектуальные системы. Теория и приложения. — 2015. — Т. 19, № 4. — С. 37—55.

23. *LeCun Y.* Gradient-Based Learning Applied to Document Recognition // Proceedings of the IEEE. — 1998. — С. 46.

24. Different types of CNN Architectures [Электронный ресурс]. — Режим доступа: https://vitalflux.com/different-types-of-cnn-architectures-explained-examples/. — (Дата обращения: 28.04.2023).

25. *Szegedy C.* Going Deeper With Convolutions // Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). — 2015. — С. 12.

26. Understanding Capsule Networks [Электронный ресурс]. — Режим доступа: https://www.freecodecamp.org/news/understanding-capsule-networks-ais-allurin-new-architecture-bdb228173ddc. — (Дата обращения: 28.04.2023).

27. *Nitin R.* MobileNets for flower classification using TensorFlow // 2017 international conference on big data, IoT and data science (BID). — IEEE. 2017. — С. 154—158.

28. Military Aircraft Recognition dataset [Электронный ресурс]. — Режим доступа: https://www.kaggle.com/datasets/khlaifiabilel/military-aircraft-recognition-dataset. — (Дата обращения: 28.02.2023).

ПРИЛОЖЕНИЕ А
Модуль модели

Листинг А.1 – Модель для классификации самолетов

```
class simplenet(nn.Module):
    def __init__(self, classes=20, simpnet_name='simplenet'):
        super(simplenet, self).__init__()
        self.features = self._make_layers()
        self.classifier = nn.Linear(256, classes)
        self.drp = nn.Dropout(0.1)

    def forward(self, x):
        out = self.features(x)

        out = F.max_pool2d(out, kernel_size=out.size()[2:])
        out = self.drp(out)

        out = out.view(out.size(0), -1)
        out = self.classifier(out)
        return out

    def _make_layers(self):
        model = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=[3, 3], stride=(1, 1),
               padding=(1, 1)),
            nn.BatchNorm2d(64, eps=1e-05, momentum=0.05,
               affine=True),
            nn.ReLU(inplace=True),

            nn.Conv2d(64, 128, kernel_size=[3, 3], stride=(1,
               1), padding=(1, 1)),
            nn.BatchNorm2d(128, eps=1e-05, momentum=0.05,
               affine=True),
            nn.ReLU(inplace=True),

            nn.Conv2d(128, 128, kernel_size=[3, 3], stride=(1,
               1), padding=(1, 1)),
            nn.BatchNorm2d(128, eps=1e-05, momentum=0.05,
               affine=True),
            nn.ReLU(inplace=True),
```

```
nn.Conv2d(128, 128, kernel_size=[3, 3], stride=(1,
   1), padding=(1, 1)),
nn.BatchNorm2d(128, eps=1e-05, momentum=0.05,
   affine=True),
nn.ReLU(inplace=True),

nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2),
   dilation=(1, 1), ceil_mode=False),
nn.Dropout2d(p=0.1),

nn.Conv2d(128, 128, kernel_size=[3, 3], stride=(1,
   1), padding=(1, 1)),
nn.BatchNorm2d(128, eps=1e-05, momentum=0.05,
   affine=True),
nn.ReLU(inplace=True),

nn.Conv2d(128, 128, kernel_size=[3, 3], stride=(1,
   1), padding=(1, 1)),
nn.BatchNorm2d(128, eps=1e-05, momentum=0.05,
   affine=True),
nn.ReLU(inplace=True),

nn.Conv2d(128, 256, kernel_size=[3, 3], stride=(1,
   1), padding=(1, 1)),
nn.BatchNorm2d(256, eps=1e-05, momentum=0.05,
   affine=True),
nn.ReLU(inplace=True),

nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2),
   dilation=(1, 1), ceil_mode=False),
nn.Dropout2d(p=0.1),

nn.Conv2d(256, 256, kernel_size=[3, 3], stride=(1,
   1), padding=(1, 1)),
nn.BatchNorm2d(256, eps=1e-05, momentum=0.05,
   affine=True),
nn.ReLU(inplace=True),

nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2),
   dilation=(1, 1), ceil_mode=False),
nn.Dropout2d(p=0.1),
```

```
            nn.Conv2d(256, 512, kernel_size=[3, 3], stride=(1,
               1), padding=(1, 1)),
            nn.BatchNorm2d(512, eps=1e-05, momentum=0.05,
               affine=True),
            nn.ReLU(inplace=True),

            nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2),
               dilation=(1, 1), ceil_mode=False),
            nn.Dropout2d(p=0.1),

            nn.BatchNorm2d(2048, eps=1e-05, momentum=0.05,
               affine=True),
            nn.ReLU(inplace=True),

            nn.Conv2d(2048, 256, kernel_size=[1, 1], stride=(1,
               1), padding=(0, 0)),
            nn.BatchNorm2d(256, eps=1e-05, momentum=0.05,
               affine=True),
            nn.ReLU(inplace=True),

            nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2),
               dilation=(1, 1), ceil_mode=False),
            nn.Dropout2d(p=0.1),

            nn.Conv2d(256, 256, kernel_size=[3, 3], stride=(1,
               1)), # , padding=(1, 1)),
            nn.BatchNorm2d(256, eps=1e-05, momentum=0.05,
               affine=True),
            nn.ReLU(inplace=True))

        return model
```

Листинг А.2 – Класс для взаимодействия с обучаемым данными

```
DATASET_PATH: str = "../planes_dataset"
TRAIN_TENSORS_PATH: str = "./train_tensors"
TEST_TENSORS_PATH: str = "./test_tensors"

class DataSetHandler:
    _AUG_ROTATE_ANGLE = 30
    def TrainSize(self):
        files = listdir(TRAIN_TENSORS_PATH)
```

```
        return len(files) - 1 #file with y results

    def TestSize(self):
        files = listdir(TEST_TENSORS_PATH)
        return len(files) - 1 #file with y results

    def GetTrainingBatch(self, batchIndexes, needAug=False):
        xBatch, yBatch = self._GetBatch(batchIndexes,
           f"{TRAIN_TENSORS_PATH}/train")
        if needAug:
            xBatch, yBatch = self._AugmentateBatches(xBatch,
               yBatch)

        return xBatch, yBatch


    def GetTestBatch(self, batchIndexes):
        return self._GetBatch(batchIndexes,
           f"{TEST_TENSORS_PATH}/test")

    def UpdateData(self):
        self._UpdateTrainData()
        self._UpdateTestData()

    def _GetBatch(self, batchIndexes, pathPrefix):
        y = []
        with open(f"{pathPrefix}_results.txt") as classesFile:
            classes = classesFile.read().split('\n')
            for i in batchIndexes:
                y.append(int(classes[i]))

        x = []
        for i in batchIndexes:
            x.append(torch.load(f"{pathPrefix}_{i}.pt"))

        return torch.stack(x), torch.Tensor(y).to(torch.long)

    def _UpdateTrainData(self):
        xTrain = []
        yTrain = []
        with open(f"{DATASET_PATH}/ImageSets/Main/train.txt") as
```

```
            trainImagesFile:
                for imageNumber in trainImagesFile:
                    images, classes =
                        self._FindAllImages(int(imageNumber))
                    xTrain = xTrain + images
                    yTrain = yTrain + classes

        for i in range(len(xTrain)):
            tensor: torch.Tensor =
                self._ConverToTensor(xTrain[i])
            torch.save(tensor,
                f"{TRAIN_TENSORS_PATH}/train_{i}.pt")

        with open(f"{TRAIN_TENSORS_PATH}/train_results.txt",
            "w") as f:
                for i in range(len(yTrain)):
                    f.write(f"{yTrain[i]}\n")

    def _UpdateTestData(self):
        xTest = []
        yTest = []
        with open(f"{DATASET_PATH}/ImageSets/Main/test.txt") as
            testImagesFile:
                for imageNumber in testImagesFile:
                    images, classes =
                        self._FindAllImages(int(imageNumber))
                    xTest = xTest + images
                    yTest = yTest + classes

        for i in range(len(xTest)):
            tensor: torch.Tensor = self._ConverToTensor(xTest[i])
            torch.save(tensor,
                f"{TEST_TENSORS_PATH}/test_{i}.pt")

        with open(f"{TEST_TENSORS_PATH}/test_results.txt", "w")
            as f:
                for i in range(len(yTest)):
                    f.write(f"{yTest[i]}\n")

    def _ConverToTensor(self, imagePath: str) -> torch.Tensor:
        image: Image.Image = Image.open(imagePath)
```

```
        transform = transforms.ToTensor()
        tensor: torch.Tensor = transform(image)
        # remove alpha channel
        if (tensor.size(0) == 4):
            tensor = tensor[:-1]

        return tensor

    def _AugmentateBatches(self, xBatch: torch.Tensor, yBatch:
       torch.Tensor):
        xBatchAug = []
        yBatchAug = []

        for i, tensor in enumerate(xBatch):
            augTensor1 = transforms.functional.rotate(tensor,
               self._AUG_ROTATE_ANGLE)
            augTensor2 = transforms.functional.rotate(tensor,
               -self._AUG_ROTATE_ANGLE)
            augTensor3 =
               transforms.functional.adjust_brightness(tensor,
               1.5)
            augTensor4 =
               transforms.functional.gaussian_blur(tensor,
               kernel_size=(5,9), sigma=3)
            xBatchAug += [tensor, augTensor1, augTensor2,
               augTensor3, augTensor4]
            yBatchAug += [yBatch[i].item()] * 5

        order = np.random.permutation(len(xBatchAug))
        xBatchAug = np.array(xBatchAug)[order]
        yBatchAug = np.array(yBatchAug)[order]

        return torch.stack(list(xBatchAug)),
           torch.Tensor(list(yBatchAug)).to(torch.long)
```

Листинг А.3 – Алгоритм прохода одной эпохи

```
class INetworkController:
    def TrainNetwork(self, epochs: int):
        self.TrainPrepare()

        trainAccs, testAccs = [], []
        for epoch in range(epochs):
```

```
            self.TrainEpoch()
            print(f"================================== {epoch}
               ================================")
            trainAcc, testAcc = self.LogTraining()
            trainAccs.append(trainAcc)
            testAccs.append(testAcc)
            if len(testAccs) >= 3:
                if testAccs[-1] < testAccs[-2] and testAccs[-1]
                   < testAccs[-3]:
                    break

    def LogTraining(self):
        datasetHandler = DataSetHandler()

        predsTest = []
        yBatchTest = []
        for startTestBatch in range(0,
           datasetHandler.TestSize(), 500):
            xBatch, yBatch = datasetHandler.GetTestBatch(list(
                range(startTestBatch, min(startTestBatch + 500,
                   datasetHandler.TestSize()))))
            preds = self.GetResults(xBatch)
            predsTest = predsTest + preds.tolist()
            yBatchTest = yBatchTest + yBatch.tolist()
            gc.collect()

        predsTrain = []
        yBatchTrain = []
        for startTrainBatch in range(0,
           datasetHandler.TrainSize(), 500):
            xBatch, yBatch =
               datasetHandler.GetTrainingBatch(list(range(
                startTrainBatch, min(startTrainBatch + 500,
                   datasetHandler.TrainSize()))))
            preds = self.GetResults(xBatch)
            predsTrain = predsTrain + preds.tolist()
            yBatchTrain = yBatchTrain + yBatch.tolist()
            gc.collect()

        print("====================================")
        testMisses = [0] * 20
```

```
        testClasses = [0] * 20
        recognizedTest = 0
        for i in range(len(predsTest)):
            testClasses[yBatchTest[i]] += 1
            if predsTest[i] == yBatchTest[i]:
                recognizedTest += 1
            else:
                testMisses[yBatchTest[i]] += 1


        trainMisses = [0] * 20
        trainClasses = [0] * 20
        recognizedTrain = 0
        for i in range(len(predsTrain)):
            trainClasses[yBatchTrain[i]] += 1
            if predsTrain[i] == yBatchTrain[i]:
                recognizedTrain += 1
            else:
                trainMisses[yBatchTrain[i]] += 1


        print("".join(map(lambda x: "{:6}".format(x),
           list(range(1,21)))))
        print("".join(map(lambda x: "{:6}".format(x),
           testClasses)))
        print("".join(map(lambda x: "{:6}".format(x),
           testMisses)))
        print("".join(map(lambda x: "{:6}".format(x),
           trainClasses)))
        print("".join(map(lambda x: "{:6}".format(x),
           trainMisses)))

        testAcc = float(recognizedTest) / len(predsTest)
        trainAcc = float(recognizedTrain) / len(predsTrain)
        print(float(recognizedTest) / len(predsTest))
        print(float(recognizedTrain) / len(predsTrain))
        print("====================================")

        return trainAcc, testAcc
```

Листинг А.4 – Класс для взаимодействия с моделью

```
class NetworkController(INetworkController):
```

```
    _TRAINED_MODELS_PATH = "trained_models/"

    def __init__(self, batchSize, learningRate, needAug=True):
        self.m_device = torch.device("cuda" if
           torch.cuda.is_available() else "cpu")
        self.m_planesNetwork = simplenet(20).to(self.m_device)
        self.m_datasetHandler = DataSetHandler()

        self.m_batchSize = batchSize
        self.m_learningRate = learningRate
        self.m_needAug = needAug

        self.m_loss = torch.nn.CrossEntropyLoss()
        self.m_optimizer =
           torch.optim.Adam(self.m_planesNetwork.parameters(),
           lr=self.m_learningRate)
        self.m_datasetLen = self.m_datasetHandler.TrainSize()

    def TrainPrepare(self):
        pass

    def TrainEpoch(self):
        self.m_planesNetwork.train()
        order = np.random.permutation(self.m_datasetLen)
        for startIndex in range(0, self.m_datasetLen,
           self.m_batchSize):
            self.m_optimizer.zero_grad()

            xBatch, yBatch =
               self.m_datasetHandler.GetTrainingBatch(
                     order[startIndex:startIndex+self.m_batchSize],
                        needAug=self.m_needAug)
            xBatch = xBatch.to(self.m_device)
            yBatch = yBatch.to(self.m_device)

            preds = self.m_planesNetwork.forward(xBatch)
            lossValue = self.m_loss(preds, yBatch)

            lossValue.backward()

            self.m_optimizer.step()
```

```
    def GetResults(self, xBatch):
        self.m_planesNetwork.eval()
        results =
           self.m_planesNetwork.forward(xBatch).argmax(dim=1)
        self.m_planesNetwork.train()

        return results

    def GetResult(self, imagePath):
        image: Image.Image = Image.open(imagePath)
        transform = transforms.ToTensor()
        tensor: torch.Tensor = transform(image)
        # remove alpha channel
        if (tensor.size(0) == 4):
            tensor = tensor[:-1]

        t = torch.stack([tensor])
        return self.GetResults(t)[0]

    def SaveModel(self, modelPath):
        torch.save(self.m_planesNetwork,
           f"{self._TRAINED_MODELS_PATH}{modelPath}")

    def LoadModel(self, modelPath):
        self.m_planesNetwork =
           torch.load(f"{self._TRAINED_MODELS_PATH}{modelPath}")
        self.m_planesNetwork.to(self.m_device)

    def GetAllModels(self):
        models = []
        for _, _, filenames in
           os.walk(self._TRAINED_MODELS_PATH):
            for model in filenames:
                models.append(model)

        return models
```

ПРИЛОЖЕНИЕ Б
Презентация