МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет: вычислительной техники

Кафедра: «Вычислительная техника»

Направление подготовки: 09.04.01 «Информатика и вычислительная техника»

Магистерская программа: «Прикладной искусственный интеллект»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

ДЕТЕКТИРОВАНИЕ СЕТЕВЫХ АТАК ПРИ ПОМОЩИ НЕЙРОСЕТИ

Студент _____ 30.05.2025 Татаршев Андрей Алексеевич
(подпись, дата) (ФИО полностью)

Руководитель _____ 30.05.2025 Карамышева Н.С.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр _____ 30.05.2025 Бычков А. С.
(подпись, дата) (фамилия, инициалы)

Рецензент зав.каф. "Программирование" ПензГТУ, д.т.н., 30.05.2025 Мартынкин А.Н.
(подпись, дата) (фамилия, инициалы) (должность, место работы)

Работа допущена к защите (протокол зас. кафедры от 28.05.2025 № 16)

Заведующий кафедрой _____ Митрохин М. А.
(подпись) (фамилия, инициалы)

Работа защищена с отметкой отлично (протокол заседания ГЭК от 5.06.2025 № 8)

Секретарь ГЭК _____ Гурин Е. И.
(подпись) (фамилия, инициалы)

Пенза 2025

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет вычислительной техники

Кафедра «Вычислительная техника»

«Утверждаю»
Заведующий кафедрой
М. А. Митрохин
«29» сентября 2023г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ МАГИСТРА

1.Студент Татаршев Андрей Алексеевич гр. 23ВВИм1 факультета вычислительной техники, направления 09.04.01.

2. Тема работы «Детектирование сетевых атак при помощи нейросети».

Тема утверждена приказом ПГУ № 986/о от "29"сентября 2023 г.

3. Руководитель магистерской диссертации к.т.н., доцент кафедры ВТ ПГУ Карамышева Н.С.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

Назначение разработки: создание нейросетевой модели для автоматического обнаружения сетевых атак.

Разработать программное решение, способное классифицировать сетевой трафик с использованием методов глубокого обучения.

5. Перечень подлежащих разработке вопросов:

1) Проведение анализа актуальных подходов к обнаружению сетевых атак и выбор оптимальных архитектур нейросетей.

2) Реализация модели многослойного перцептрона (MLP) и модели сверточной нейронной сети (1D-CNN) для многоклассовой классификации.

3) Построение системы оценки качества моделей с расчётом метрик, визуализацией и генерацией отчётов.

4) Проведение сравнительного анализа результатов MLP и CNN по основным метрикам качества.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1.	Изучить теоретические основы	10%	27.09.2023-15.01.2024	
2.	Анализ предметной области и существующих решений по обнаружению сетевых атак	10%	16.01.2024-22.03.2024	
3.	Выбор инструментов и библиотек, формирование требований к модели	5%	23.03.2024-14.06.2024	
4.	Разработка программной реализации и кода для обработки данных	20%	15.06.2024-10.10.2024	
5.	Спроектировать и обучить нейронную сеть	30%	11.10.2024-05.01.2025	
6.	Сравнительный анализ результатов моделей и визуализация метрик	15%	06.01.2025-17.04.2025	
7.	Тестирование работы моделей и оценка их эффективности	5%	18.04.2025-9.05.2025	
8.	Оформление выпускной работы	5%	10.05.2025-30.05.2025	

Дата выдачи задания "29" сентября 2023 г.

Руководитель магистерской диссертации _____ 29.09.2023 Карамышева Н.С.
(подпись, дата)

Задание к исполнению принял студент _____ 29.09.2023 Татаршев А.А.
(подпись, дата)

Работу к защите допустить
Декан факультета _____ 05.06.2025 Фионова Л.Р.
(подпись, дата)

РЕФЕРАТ

Пояснительная записка содержит 102 листа, 13 рисунков, 3 таблицы, 5 источников и 1 приложение.

НЕЙРОСЕТЕВЫЕ МОДЕЛИ, КИБЕРБЕЗОПАСНОСТЬ, DETECTION, MLP, CNN, СЕТЕВЫЕ АТАКИ, ИНФОРМАЦИОННАЯ БЕЗОПАСНОСТЬ

Дипломная работа посвящена разработке и сравнительному анализу нейросетевых моделей для классификации сетевых атак на основе набора данных Kitsune Network Attack Dataset. Актуальность темы обусловлена ростом числа киберугроз и необходимостью создания интеллектуальных систем обнаружения, способных выявлять как известные, так и новые типы атак в реальном времени.

В работе реализованы две архитектуры глубокого обучения: многослойный перцептрон (MLP) и одномерная сверточная нейронная сеть (1D-CNN). Для повышения качества обучения были применены методы стандартизации данных, устранения выбросов, а также техники балансировки классов, включая SMOTE и случайную выборку.

Проведено сравнение моделей в условиях бинарной и многоклассовой классификации. Оценка качества включала расчёт метрик точности, полноты, F1-меры и ROC-AUC, построение матриц ошибок и графиков обучения. Эксперименты показали, что обе модели достигают высокой точности (более 94%), при этом CNN демонстрирует большую устойчивость к дисбалансу и лучше справляется с редкими типами атак.

Результаты могут быть использованы при создании систем мониторинга трафика в сетях IoT, облачных платформах и критически важных инфраструктурах.

					ПГУ 1.090401.15.001 ПЗ		
Изм.	Лист	№ докум.	Подпись	Дата			
Разраб.		Татаршев А.А.		30.05.25	Детектирование сетевых атак при помощи нейросети Пояснительная записка	Лит.	Лист 2 / Листов 102
Провер.		Карамышева Н.С.		30.05.25			
Н. Контр		Бычков А.С.		30.05.25		ФВТ, каф. ВТ гр. 23ВВИм1	
Утверд.		Митрохин М.А.		30.05.25			

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ 4
ВВЕДЕНИЕ 5
1 Теоретические и технологические основы исследования 6
1.1 Анализ предметной области 6
1.2 Анализ актуальности разработки 7
1.3 Классификация сетевых атак 9
1.4 Постановка задачи . . . . . .
1.5 Выбор инструментов разработки . . . . . .
1.6 Исходная сетевая инфраструктура и условия сбора данных . . . . . .
2 Разработка и реализация моделей обнаружения сетевых атак 20
2.1 Описание датасета и сбор данных 20
2.2 Оценка качества данных 22
2.3 Реализация модели бинарной классификации 26
2.4 Реализация моделей многоклассовой классификации 40
2.4.1 Загрузка данных 40
2.4.2 Архитектура модели MLP 43
2.4.3 Архитектура модели CNN 46
2.4.4 Оценка качества моделей . . . . . .
3 Экспериментальная часть 52
3.1 Бинарная классификация 52

3.2 Многоклассовая классификация . . . . . .
55
ЗАКЛЮЧЕНИЕ . . . . . .
СПИСОК ИСПОЛЬЗОВАНЫХ ИСТОЧНИКОВ . . . . . .
ПРИЛОЖЕНИЕ А Листинг программы . . . . . .
ПРИЛОЖЕНИЕ В Презентация 85

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящей магистерской диссертации применяются следующие сокращения и обозначения

CSV (Comma-Separated Values) – текстовый формат представления табличных данных, разделённых запятыми

CNN (Convolutional Neural Network) – сверточная нейронная сеть, применяемая для анализа структурированных данных

DDoS (Distributed Denial of Service) – распределённая атака отказа в обслуживании

GPU (Graphics Processing Unit) – графический процессор, ускоряющий обучение нейросетей

MLP (Multi-Layer Perceptron) – многослойный перцептрон, базовая архитектура нейронной сети

ReLU (Rectified Linear Unit) – широко используемая функция активации нейронов в нейронных сетях

ROC (Receiver Operating Characteristic) – характеристическая кривая, отражающая соотношение чувствительности и ложноположительных срабатываний

SMOTE (Synthetic Minority Over-sampling Technique) – метод синтетического увеличения числа примеров миноритарного класса

ВВЕДЕНИЕ

Современное развитие цифровых технологий сопровождается ростом количества киберугроз, способных нарушать стабильную работу информационных систем. Атаки становятся всё более разнообразными, масштабными и сложными, что требует применения интеллектуальных методов защиты. Традиционные системы, основанные на сигнатурах и фиксированных правилах, оказываются недостаточно эффективными в условиях появления новых, ранее неизвестных угроз.

Одним из перспективных направлений в области информационной безопасности является использование методов машинного обучения, в частности – нейросетевых моделей. Эти подходы позволяют выявлять аномалии в сетевом трафике на основе скрытых закономерностей, не доступных ручному анализу. Особую актуальность приобретают архитектуры глубокого обучения, способные автоматически извлекать признаки из потоков данных и адаптироваться к изменяющимся условиям.

Целью настоящей работы является построение и сравнение нейросетевых моделей для классификации сетевых атак, а также оценка их применимости в задачах защиты информационных систем

1 Теоретические и технологические основы исследования

1.1 Анализ предметной области

С развитием цифровых технологий и глобализацией сетевой инфраструктуры резко увеличилось количество потенциальных угроз, связанных с кибербезопасностью. Компании, государственные учреждения и частные пользователи ежедневно сталкиваются с попытками несанкционированного доступа, утечки данных, внедрения вредоносного программного обеспечения и другими формами сетевых атак. При этом характер атак продолжает усложняться, и всё чаще используются сложные схемы, которые не поддаются обнаружению классическими сигнатурными методами.

Системы обнаружения вторжений традиционно базируются на двух подходах: сигнатурном и поведенческом. Сигнатурный подход требует заранее заданных шаблонов известных угроз, что делает его бессильным против новых, ранее неизвестных атак. Поведенческий подход основан на анализе аномалий в поведении сетевого трафика и способен выявлять нестандартные события, но при этом может страдать от большого числа ложных срабатываний. В результате возрастает интерес к интеллектуальным методам анализа, в том числе к машинному и глубокому обучению [1].

Особую актуальность приобретает анализ сетевого трафика с помощью нейросетевых моделей. Такие модели способны обрабатывать большие объемы данных, выявлять сложные нелинейные зависимости и эффективно работать даже в условиях зашумлённой или частично размеченной информации. Среди нейросетевых архитектур особое внимание уделяется многослойным перцептронам (MLP) и сверточным нейронным сетям (CNN). Первая отличается простотой реализации и универсальностью, вторая - способностью извлекать локальные признаки и учитывать временные зависимости, что делает её особенно полезной в задачах анализа последовательных данных.

В рамках предметной области исследуются не только архитектурные особенности моделей, но и способы подготовки данных, включая очистку, нормализацию и борьбу с дисбалансом классов. Последний фактор особенно критичен в задачах обнаружения атак, так как количество нормального трафика зачастую значительно превышает объём аномальных записей. В таких случаях без применения специальных техник, таких как SMOTE, модель склонна игнорировать миноритарные классы, что снижает эффективность системы защиты.

Дополнительным вызовом является необходимость работы в условиях ограниченных вычислительных ресурсов. Многие системы мониторинга устанавливаются на периферийных устройствах или в распределённых IoT-сетях, где важны не только точность и надёжность, но и скорость отклика, экономия энергии и устойчивость к задержкам. В связи с этим модель должна быть не только точной, но и компактной, легко обучаемой и способной к работе в реальном времени.

Всё это формирует современную предметную область, где ключевыми задачами становятся повышение точности обнаружения угроз, снижение количества ложных срабатываний, устойчивость к изменяющимся паттернам трафика и адаптация под реальные условия эксплуатации. Роль нейросетей в этих задачах становится всё более значимой, особенно с учётом их способности к обучению на больших объёмах сетевых данных без ручного составления правил и эвристик.

1.2 Анализ актуальности разработки

На фоне стремительного роста количества сетевых атак и повышения их сложности вопросы информационной безопасности приобретают стратегическую важность. Злоумышленники используют всё более изощрённые методы проникновения, включая маскировку под легитимный трафик, использование цепочек атак и эксплуатацию нулевых уязвимостей. В этих условиях традиционные системы защиты - межсетевые экраны, сигнатурные антивирусы и простейшие IDS – теряют эффективность. Новые

угрозы требуют гибких и адаптивных решений, способных обнаруживать аномалии без предварительного знания их структуры.

Особенно уязвимыми остаются распределённые IoT-сети, облачные платформы и промышленные системы управления. Все они подвергаются атакующим воздействиям, но в силу особенностей архитектуры и ресурсоограниченности не могут использовать громоздкие решения [2]. Отсюда вытекает необходимость создания интеллектуальных моделей обнаружения вторжений, сочетающих точность, быстродействие и простоту внедрения. Наиболее перспективными в этой области считаются нейросетевые подходы, особенно те, которые опираются на глубокое обучение и умеют адаптироваться к новым типам угроз.

Современные нейросетевые архитектуры демонстрируют отличные результаты в задачах классификации, включая бинарную и многоклассовую детекцию аномального поведения в сетевом трафике. Особую значимость приобретает возможность применения сверточных сетей (CNN), изначально предназначенных для обработки изображений, в задачах анализа временных рядов и последовательностей. Это позволяет эффективно выявлять сложные закономерности, характерные для вредоносной активности, и отличать их от нормального поведения пользователей и устройств.

Большое значение имеет и проблема дисбаланса классов. В реальных условиях количество нормального сетевого трафика многократно превышает объём аномальных данных. Без применения специальных методов, таких как синтетическое дополнение выборки или модифицированные функции потерь, нейросетевые модели склонны игнорировать редкие, но наиболее важные случаи - те, что действительно представляют угрозу. Это делает тему борьбы с дисбалансом особенно актуальной в разработке практических решений.

Kitsune Network Attack Dataset представляет собой уникальный по составу и сложности набор сетевых данных, в котором сочетаются высокая нагрузка, разнообразие протоколов и большое число реальных атак. Он широко используется в научных исследованиях и позволяет моделировать

реальные сценарии вторжений. Использование этого датасета для обучения и сравнения моделей MLP и CNN делает работу особенно актуальной, так как позволяет оценить эффективность подходов в условиях, приближенных к реальности.

Также нельзя игнорировать потребность в интерпретируемости моделей. В условиях растущего регулирования в сфере кибербезопасности требуется не просто обнаруживать угрозы, но и объяснять, на основе каких признаков была зафиксирована аномалия. Это способствует доверию к системам защиты со стороны специалистов и облегчает интеграцию в корпоративные процессы реагирования.

Всё вышеперечисленное свидетельствует о высокой актуальности разработки интеллектуальных моделей обнаружения сетевых атак на основе нейросетевых архитектур, особенно в условиях ограниченных ресурсов, высокого объема данных и быстроменяющихся условий сетевой среды

1.3 Классификация сетевых атак

Сетевые атаки представляют собой действия, направленные на нарушение конфиденциальности, целостности или доступности информационных ресурсов путём вмешательства в сетевое взаимодействие между устройствами. В условиях растущей цифровизации и увеличения числа подключённых устройств проблема защиты сетевого периметра становится всё более актуальной. Актуальные системы мониторинга и защиты строятся на основе понимания типов атак и их особенностей, что делает классификацию сетевых угроз важной частью любой системы обнаружения и реагирования.

Существует множество подходов к классификации атак. Один из распространённых базируется на уровнях модели OSI [4]. Согласно этому подходу, атаки подразделяются на:

– физический уровень: атаки на оборудование, перехват сигнала, электромагнитные помехи,

– канальный уровень: MAC-spoofing, ARP-spoofing, фальсификация адресов,

– сетевой уровень: IP-спуфинг, ICMP-флуд, маршрутизация вредоносных пакетов,

– транспортный уровень: SYN-флуд, UDP-флуд, обрыв TCP-сессий,

– сеансовый и представительный уровни: манипуляции с шифрованием, подделка сертификатов, атаки на SSL/HTTPS,

– прикладной уровень: SQL-инъекции, фаззинг API, атаки на веб-интерфейсы.

Другой важный критерий - вектор вторжения. Атаки могут быть как внешними, исходящими из интернета, так и внутренними, инициируемыми уже скомпрометированными узлами внутри сети. Внутренние угрозы считаются особенно опасными, так как сложно детектируются и нередко исходят от легитимных устройств, уже прошедших авторизацию.

По характеру воздействия атаки делятся на:

– пассивные, направленные на скрытое наблюдение за трафиком без вмешательства в него (например, перехват данных или прослушивание каналов),

– активные, предполагающие вмешательство: изменение, удаление или подмену данных, внедрение вредоносных пакетов, блокировка сервисов.

По цели атаки различают:

– DDoS-атаки (Distributed Denial of Service) - направлены на выведение сервиса из строя путём перегрузки ресурсов,

– Man-in-the-Middle (MitM) - перехват и возможная подмена трафика между двумя сторонами, чаще всего через ARP-спуфинг,

– Port scanning - разведка уязвимостей, поиск открытых портов,

– Botnet-атаки - использование захваченных устройств для организации распределённых атак (типичный пример - Mirai),

– API fuzzing - автоматическое создание запросов к веб-интерфейсам для выявления уязвимостей или вывода из строя [5].

С точки зрения длительности и структуры, атаки делятся на:

– импульсные (burst-type) - кратковременные, но мощные воздействия, например, резкий флуд или сканирование портов за считанные секунды,

– продолжительные (persistent) - протекающие в течение длительного времени с целью незаметного внедрения и сбора информации (часто характерны для APT - advanced persistent threats),

– медленные атаки (low-rate) - сознательно растянутые во времени, с минимальной активностью, чтобы обойти системы обнаружения.

Современные угрозы также могут обладать комбинированным характером, объединяя в себе несколько техник одновременно. Например, атака может начинаться со сканирования сети, переходить в фазу ARP-spoofing, а затем использовать перехваченные данные для внедрения в сеанс связи. В таких случаях особенно важно наличие адаптивных систем, способных выявлять поведенческие паттерны, не привязанные к фиксированным сигнатурам.

Одним из реальных и распространённых примеров является ботнет Mirai, атакующий IoT-устройства. Он активно сканирует сеть на наличие уязвимых устройств, подключается к ним через дефолтные логины и пароли, а затем использует их для организации DDoS-атак. Другая интересная атака - SSL Renegotiation, в которой злоумышленник инициирует многочисленные повторные переговоры протокола SSL, тем самым вызывая перегрузку сервера без необходимости большого объёма данных. Также часто встречается API fuzzing, при котором атака строится на систематическом переборе параметров в HTTP-запросах, что приводит к ошибкам в логике сервера.

Классификация атак необходима не только для анализа инцидентов, но и для построения эффективных датасетов. В исследовании использовался Kitsune Network Attack Dataset, содержащий следующие классы:

– 0 – нормальный трафик,

– 1 – ARP MitM,

- 2 – Active Wiretap,
- 3 – API Fuzzing,
- 4 – Mirai,
- 5 – OS Scan,
- 6 – SSL Renegotiation.

Каждая из этих атак отличается характерной временной структурой, статистическими признаками и распределением по пакетам. Например, ARP MitM сопровождается аномальным количеством ARP-запросов, Mirai - высокой периодичностью ICMP и TCP-запросов, SSL Renegotiation - множеством установок соединения с короткой продолжительностью.

Для задач машинного обучения особенно важно, чтобы классы были хорошо различимы по признакам, но при этом не тривиальны. Простые DDoS-атаки могут быть легко обнаружены по аномальной частоте запросов, а вот API fuzzing требует анализа более тонких закономерностей, таких как комбинации флагов TCP или интервалы между сессиями. В этом контексте нейросетевые модели, такие как MLP и CNN, становятся особенно актуальными, так как способны выявлять неочевидные зависимости в данных и использовать как временные, так и пространственные характеристики трафика.

Понимание структуры и целей различных типов атак позволяет строить более эффективные архитектуры моделей и настраивать процессы обучения, валидации и тестирования с учётом специфики каждого класса. Это особенно критично при работе с несбалансированными данными, где редкие, но опасные атаки могут быть проигнорированы простыми алгоритмами. Классификация сетевых атак - необходимый этап в разработке систем интеллектуального мониторинга и основа для построения устойчивых, интерпретируемых и адаптивных решений в сфере кибербезопасности.

1.4 Постановка задачи

Современные подходы к обеспечению сетевой безопасности требуют разработки интеллектуальных систем, способных в автоматическом режиме

анализировать сетевой трафик, выявлять аномалии и классифицировать потенциальные угрозы. Задача обнаружения сетевых атак усложняется в условиях высокого объёма данных, нестационарности признаков и частой смены моделей атакующих. Возникает необходимость в построении моделей, способных адаптироваться к изменяющейся среде, работать с несбалансированными данными и демонстрировать устойчивую точность при реальной нагрузке.

Основной целью данной работы является разработка и сравнительный анализ нейросетевых моделей обнаружения сетевых атак на основе глубокого обучения. В качестве объектов исследования выбраны многослойный перцептрон (MLP) и одномерная сверточная нейронная сеть (1D-CNN), как две наиболее распространённые архитектуры в задачах классификации. Каждая из них обладает своими преимуществами: MLP отличается простой реализацией и высокой скоростью обучения, а CNN способна выявлять сложные локальные зависимости и временные паттерны в данных.

Исходными данными для обучения и тестирования моделей служит Kitsune Network Attack Dataset – открытый датасет, включающий разнообразные типы атак и богатый набор признаков. Он содержит более 11 миллионов записей, представляющих как нормальный трафик, так и вредоносную активность. Особенность набора заключается в наличии выраженного дисбаланса классов, высокой размерности признаков и наличии атак с временными характеристиками, что делает его подходящим для задач глубокого анализа.

Для выполнения поставленной цели необходимо решить следующие задачи:

– выполнить предварительную обработку данных: стандартизацию признаков, устранение пропущенных значений (при наличии) и анализ корреляции,

– решить проблему дисбаланса классов с помощью метода синтетического увеличения выборки, в частности, с использованием алгоритма SMOTE,

– разработать архитектуру MLP и архитектуру 1D-CNN, адаптированные под специфику сетевого трафика, включая использование регуляризации, нормализации и техники Dropout,

– обучить обе модели на подготовленных данных с применением одинаковых условий: фиксированное количество эпох, контроль переобучения через раннюю остановку и единые метрики качества,

– провести оценку точности, полноты, F1-мера и времени обучения, а также построить матрицы ошибок для каждой модели,

– сравнить результаты и выделить архитектуру, наиболее подходящую для задач обнаружения сетевых атак с учётом доступных вычислительных ресурсов и характеристик трафика.

В рамках работы требуется не просто добиться высокой точности классификации, но и обеспечить устойчивость моделей к изменениям структуры данных и наличии неявных признаков. Это позволит рассматривать разработанные решения как потенциальную основу для построения адаптивных систем мониторинга, применимых в реальных условиях: от защищённых сегментов корпоративных сетей до распределённых IoT-инфраструктур.

1.5 Выбор инструментов разработки

Выбор инструментов разработки играет ключевую роль при реализации моделей машинного обучения, особенно в условиях, когда требуется обеспечить быструю итерацию, воспроизводимость экспериментов и удобство отладки. Для выполнения поставленной задачи был сформирован стек технологий, включающий современные библиотеки Python и интегрированную среду разработки, ориентированную на научные вычисления.

Основным языком реализации выбран Python – интерпретируемый язык с широкой экосистемой библиотек, активно применяемых в задачах анализа данных и построения нейросетевых моделей. Он отличается лаконичным синтаксисом, большим сообществом разработчиков и богатой документацией. Благодаря открытости и активной поддержке, Python стал стандартом де-факто в области машинного обучения и научных исследований [6].

В качестве среды разработки использовалась Spyder (Scientific Python Development Environment). Данная IDE предназначена для работы с научным кодом и сочетает в себе функциональность MATLAB и Jupyter Notebook. Ключевыми преимуществами Spyder являются встроенный редактор кода с подсветкой синтаксиса, мощный интерактивный терминал, отладчик, просмотр переменных и интеграция с библиотеками визуализации. Это делает её удобным инструментом для пошаговой настройки архитектуры моделей и анализа промежуточных результатов. Наличие панели переменных в режиме реального времени позволяет отслеживать структуру и размеры данных, что особенно полезно при работе с многомерными массивами и тензорами [7].

Для построения нейросетевых архитектур использовалась библиотека TensorFlow с высокоуровневым API Keras. TensorFlow обеспечивает эффективное выполнение вычислений как на CPU, так и на GPU, поддерживает автоматическое дифференцирование и содержит встроенные функции оптимизации. Использование Keras позволило описывать модели в декларативной форме, обеспечивая читаемость кода и простоту экспериментов с архитектурой.

Дополнительно применялись следующие библиотеки:

- NumPy – для работы с массивами и выполнения математических операций [8],
- Pandas – для обработки табличных данных, их фильтрации, группировки и анализа [9],

– Matplotlib и Seaborn – для визуализации метрик, построения графиков обучения, матриц ошибок и других аналитических диаграмм,

– scikit-learn – для предварительной обработки данных, расчёта метрик, масштабирования признаков и применения метода SMOTE для балансировки классов.

Выбор инструментов основывался на необходимости обеспечить гибкость при работе с большими объёмами данных, высокую скорость прототипирования и лёгкость отладки. Используемый стек технологий доказал свою эффективность и позволил реализовать все этапы - от подготовки данных до анализа результатов - в едином программном окружении, что значительно ускорило процесс разработки и повысило её надёжность.

1.6 Исходная сетевая инфраструктура и условия сбора данных

Данные, использованные для обучения и оценки нейросетевых моделей, были собраны в специально развернутой экспериментальной сетевой среде, созданной в рамках проекта Kitsune. Архитектура этой среды спроектирована с целью воспроизведения реальных условий эксплуатации сетевых и IoT-устройств, а также моделирования разнообразных сценариев атак, включая как внешние вмешательства, так и внутренние инциденты. Такой подход позволяет получить трафик, максимально приближенный по структуре и поведению к тому, что фиксируется в реальных сетях организаций, домов и промышленных объектов [10].

Сетевая инфраструктура состоит из нескольких логически разделённых сегментов: Deployment A, Deployment B, Remote Site и Office Deployment. Все эти сегменты объединены в единую сеть с помощью VPN-каналов формата site-to-site. В качестве магистральных соединений используются Ethernet-соединения, что обеспечивает стабильную передачу данных между узлами. Центральным элементом инфраструктуры является удалённый узел (Remote Site), где развёрнут сервер для хранения данных и обеспечения доступа клиентов к видеопотоку. Такое построение позволяет симулировать как

централизованные, так и распределённые сценарии организации сетевого взаимодействия.

Deployment A и Deployment B представляют собой две независимые площадки видеонаблюдения, каждая из которых содержит несколько IP-камер разных моделей. Камеры работают с использованием различных протоколов: HTTPS (Sony SNC-EB600), HTTP/TCP (Sony SNC-EB602R), RTP (Sony SNC-EM600, SNC-EM602RC), а также различные модели бытовых камер, также передающих видео по RTP. Камеры подключены через PoE-коммутаторы, что позволяет одновременно передавать данные и питание по одному кабелю. Трафик от каждой камеры поступает в маршрутизаторы и далее отправляется в Remote Site по защищённым VPN-каналам. Разнообразие протоколов и устройств даёт возможность формировать трафик с различной периодичностью, нагрузкой и уровнем зашумлённости.

На схеме инфраструктуры выделены три вектора атак. Первый представляет собой подключение злоумышленника непосредственно в сеть Deployment A, что позволяет проводить атаки на уровне ARP и внедрять вредоносный трафик. Второй сценарий реализуется через соседний сегмент Deployment B и позволяет моделировать перехват пакетов и фаззинг HTTP-интерфейсов. Третий вектор - атака на клиента, подключающегося к системе видеонаблюдения из Remote Site, например, с планшета или смартфона. Здесь симулируются атаки на сеансы, включая SSL Renegotiation и попытки внедрения в цепочку соединения. Такие векторы обеспечивают присутствие в датасете как классических атак, так и более современных, ориентированных на скомпрометированные каналы связи.

Нижняя часть схемы отражает офисный сценарий - Office Deployment. Здесь используются потребительские устройства: ноутбуки, смартфоны, планшеты, точки доступа, сетевые принтеры и недорогие камеры видеонаблюдения. Все устройства работают через беспроводной маршрутизатор, что вносит в сеть дополнительную динамичность и потенциальные уязвимости, связанные с радиоканалом. Трафик в этом

сегменте более разнообразен, включает обмен по HTTP, HTTPS, RTP и локальные соединения, характерные для рабочих станций. Присутствие фоновых сервисов и высокая активность пользователей создают насыщенную среду для обучения моделей, устойчивых к шуму и сложным зависимостям.

Все устройства генерируют как легитимный, так и вредоносный трафик, причём последний был внедрён в контролируемых условиях. Это позволяет точно маркировать данные и использовать их для обучения с учётом классов атак. Среди используемых атак: ARP-spoofing, Active Wiretap, SSL Renegotiation, фаззинг API-интерфейсов, Mirai-ботнет и OS Scan. В результате получен набор данных, включающий как бинарную, так и многоклассовую разметку, пригодную для различных задач детектирования.

Сетевая инфраструктура включает в себя большое количество протоколов передачи данных и устройств с различными характеристиками, что обеспечивает богатое и разнообразное содержимое трафика. Это важно с точки зрения обучения нейросетевых моделей, поскольку позволяет им распознавать сложные паттерны и адаптироваться к новым условиям. Использование реального оборудования и типичных топологий делает трафик репрезентативным и приближенным к тому, что встречается в промышленной и домашней среде.

Условия, в которых осуществлялся сбор данных, способствуют формированию датасета, способного покрыть широкий спектр сценариев атак и нормального поведения. Это создаёт прочную основу для обучения устойчивых моделей обнаружения вторжений, которые могут применяться в практических системах защиты информации. В результате получается надёжная экспериментальная платформа, позволяющая объективно сравнивать различные архитектуры нейронных сетей и оценивать их применимость в реальных условиях. Схема сети представлена на рисунке 1.

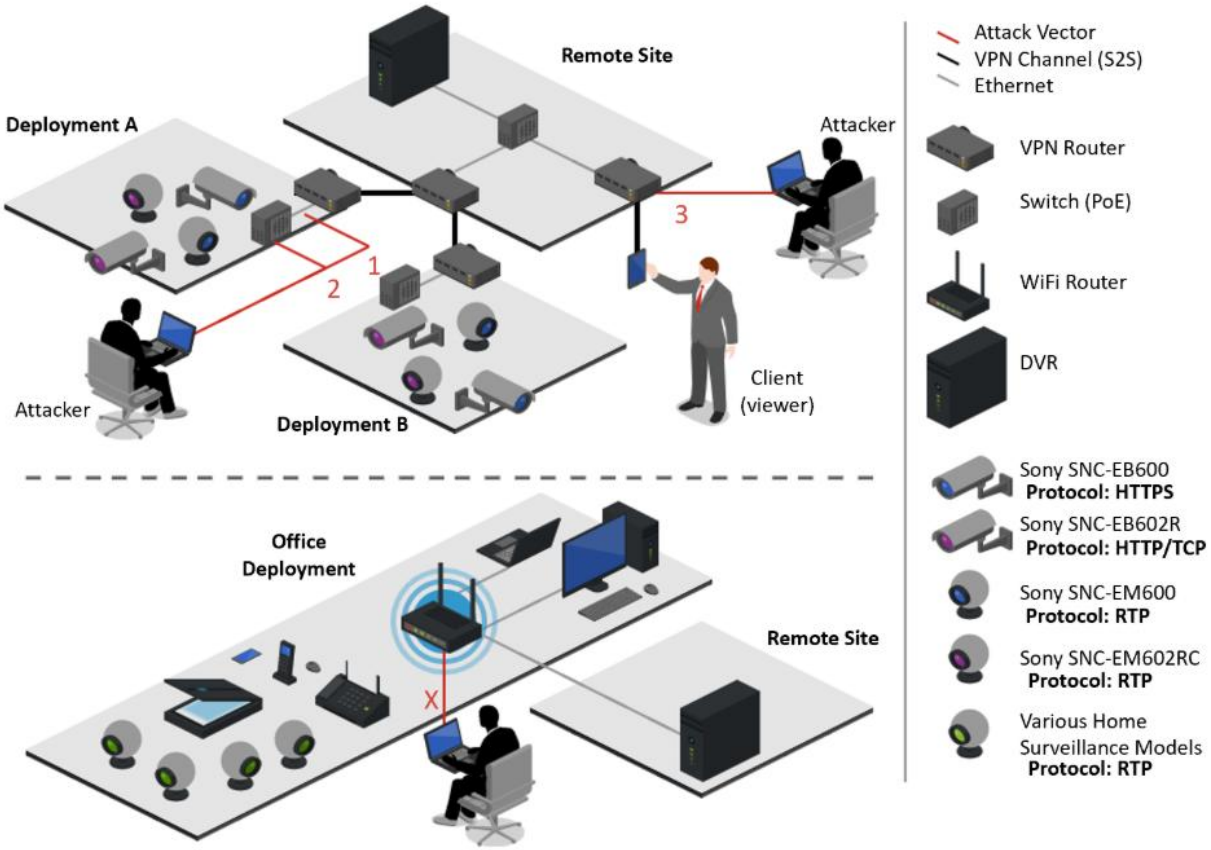


Рисунок 1 – Сетевая инфраструктура

2 Разработка и реализация моделей обнаружения сетевых атак

2.1 Описание датасета и сбор данных

Для обучения и тестирования нейросетевых моделей в исследовании использован Kitsune Network Attack Dataset - один из наиболее известных открытых наборов данных, ориентированных на задачи обнаружения сетевых атак. Данный датасет был собран в условиях, приближенных к реальной эксплуатации сетевых систем, с применением как потребительского, так и промышленного оборудования. Он широко используется в научных публикациях и считается эталонным в области интеллектуального анализа сетевого трафика.

Датасет состоит из девяти подмножеств, каждое из которых представляет отдельный сценарий атаки или комбинацию атак в условиях конкретной сетевой инфраструктуры.

Источниками трафика послужили:

– коммерческая система видеонаблюдения на базе IP-камер,

– локальная сеть с множеством IoT-устройств, подключённых к беспроводной или Ethernet-сети.

Каждый поднабор включает в себя три версии данных:

– готовый к использованию CSV-файл с числовыми признаками (предобработанные данные),

– отдельный CSV-файл с метками классов (нормальный трафик или атака),

– оригинальный PCAP-файл с сетевыми захватами, пригодный для ручного анализа или повторной генерации признаков.

Сбор данных производился в несколько этапов. На начальной фазе в течение определённого времени (обычно 1 млн пакетов) записывался чистый**,** ненарушенный трафик, формируемый реальной сетевой активностью устройств. После этого инициировалась целевая кибератака, направленная на определённые участки сети. Такой подход позволил сформировать чёткую

временную границу между нормальным и вредоносным трафиком, обеспечивая чистоту эксперимента и возможность точной разметки данных.

Каждая строка в предобработанном датасете представляет собой отдельный сетевой пакет, описанный в виде вектора из 115 признаков. Заголовков в файлах нет - структура фиксированная. Все признаки были сгенерированы с помощью специального инструмента AfterImage, разработанного авторами Kitsune. Этот инструмент способен в реальном времени извлекать статистические характеристики сетевых потоков, не прибегая к полному сохранению сессий. Он обрабатывает данные по принципу "скользящего окна", что делает его подходящим для потоковой фильтрации и обнаружения аномалий.

Извлечённые 115 признаков не являются простыми техническими параметрами пакетов (например, длина, IP, TTL), а представляют собой обобщённые статистики, отражающие поведение хостов и их взаимодействие. Признаки включают: частоту пакетов, временные интервалы, количество байт, энтропию данных, повторяемость, долю входящего и исходящего трафика, характеристики по IP-адресам и портам и др. Такая репрезентация позволяет моделям машинного обучения находить как очевидные, так и неявные закономерности в сетевом поведении.

Разметка классов представлена отдельным вектором меток, где:

– 0 соответствует нормальному трафику,

– 1–6 соответствует различным типам атак, в зависимости от поднабора.

В многоклассовой задаче каждая атака получает свой номер, что позволяет модели не просто выявить наличие аномалии, но и классифицировать её по типу.

Среди атак, представленных в датасете:

– ARP MitM – подмена ARP-таблиц с целью перехвата трафика,

– Active Wiretap – внедрение и модификация передаваемых данных,

– API Fuzzing – автоматизированный перебор параметров в API-запросах,

– Mirai – ботнет-атака на устройства IoT,

– OS Scan – сканирование операционных систем с помощью Nmap,

– SSL Renegotiation – отказ в обслуживании через бесконечные SSL-сессии.

Особенность Kitsune Dataset в том, что он содержит как импульсные**,** так и длительные атаки, что важно для обучения устойчивых нейросетевых архитектур. Кроме того, трафик был собран в реальных условиях, с использованием настоящего оборудования, а не симулирован программно. Это придаёт датасету реалистичность, особенно ценную при оценке практической применимости моделей.

Общий объём всех файлов превышает 9 миллионов пакетов, а совокупное количество записей после предобработки - более 11 миллионов. Все признаки числовые и представлены в формате float64. Пропущенные значения отсутствуют, что облегчает работу с данными. Однако, как показал анализ, имеется выраженный дисбаланс классов - около 72% записей относятся к нормальному трафику, и лишь 28% - к атакам. Это потребовало применения методов балансировки при подготовке обучающей выборки.

Таким образом, Kitsune Dataset представляет собой репрезентативную, масштабную и разнообразную коллекцию сетевых данных, позволяющую тестировать модели в условиях, близких к реальным. Использование этого набора обеспечивает достоверную оценку качества архитектур и их способности к обобщению, устойчивости и точному обнаружению сложных сетевых аномалий. Статистика датасета показана в таблице 1.

Таблица 1– Статистика датасета

Параметр	Значение
Общее количество пакетов	~9 млн
Количество признаков	115
Формат признаков	Числовые (статистические)

2.2 Оценка качества данных

Для построения надёжной модели обнаружения сетевых атак была проведена предварительная проверка качества исходного датасета. В выборке содержится около 9 миллионов строк сетевого трафика, каждая из которых описана с помощью 115 числовых признаков. Эти признаки отражают статистические параметры сетевых соединений и отдельных пакетов. Все значения представлены в формате float64, что исключает необходимость приведения типов и упрощает дальнейшую обработку.

Анализ чистоты данных показал отсутствие пропущенных значений. Это положительно сказывается на стабильности обучения, так как отпадает необходимость в заполнении пропусков и исключается риск искажения структуры данных. Работа с полной и непрерывной таблицей позволяет сразу переходить к нормализации и формированию обучающих выборок.

Исходные признаки оказались разномасштабными. Значения некоторых признаков находились в пределах от 10^-5, в то время как другие достигали порядка 10^17. Такая разнородность негативно влияет на сходимость нейросетевых моделей. Чтобы устранить эту проблему, была применена стандартизация с помощью метода StandardScaler. Каждый признак преобразован к нормальному распределению со средним значением 0 и стандартным отклонением 1. Это решение позволило выровнять вклад всех признаков и ускорить обучение моделей. До и после трансформации были построены гистограммы первых десяти признаков, чтобы визуально убедиться в корректности масштабирования. На рисунках 2 и 3 показана гистограмма первых 10 признаков до и после применения StandardScaler.

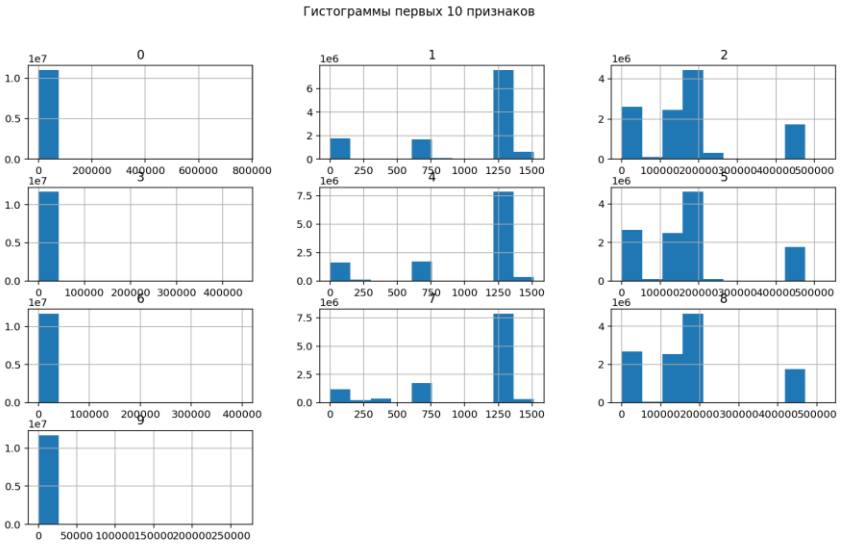


Рисунок 2 – Гистограмма первых 10 признаков до применения StandardScaler

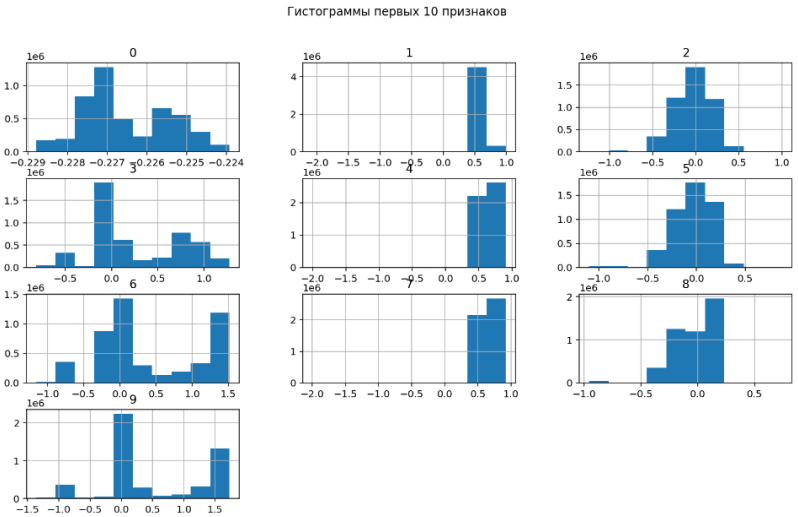


Рисунок 3 – Гистограмма первых 10 признаков после применения StandardScaler

Дополнительно был выполнен корреляционный анализ. Некоторые признаки оказались сильно связаны между собой, например, признаки под номерами 1, 4 и 7 имели почти полную линейную зависимость. Несмотря на это, от удаления коррелированных признаков решено отказаться. Нейросетевые архитектуры способны справляться с избыточной информацией за счёт регуляризации и dropout-слоёв. Это делает возможным обучение даже на частично дублирующих признаках без заметного ухудшения качества. Тепловая карта корреляций показана на рисунке 4

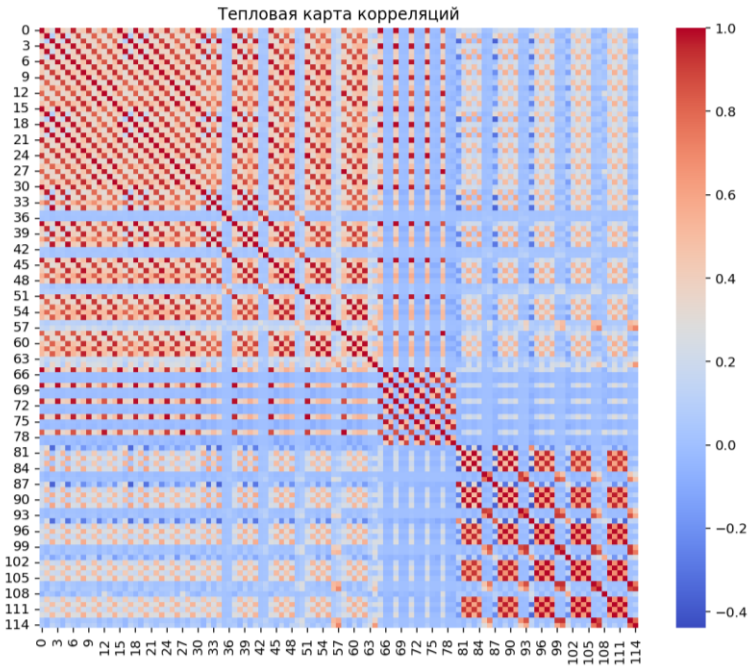


Рисунок 4 – Тепловая карта корреляции

Одним из критических моментов стал дисбаланс классов. Метки в выборке распределены неравномерно: около 72% строк относятся к нормальному трафику, а только 28% - к различным атакам. Такая ситуация приводит к смещению модели в сторону доминирующего класса, в результате чего атакующий трафик может быть ошибочно классифицирован как безопасный. Для нейросетевых моделей это особенно опасно - снижается полнота по важным, но редким классам. Распределение классов показано на рисунке 5.

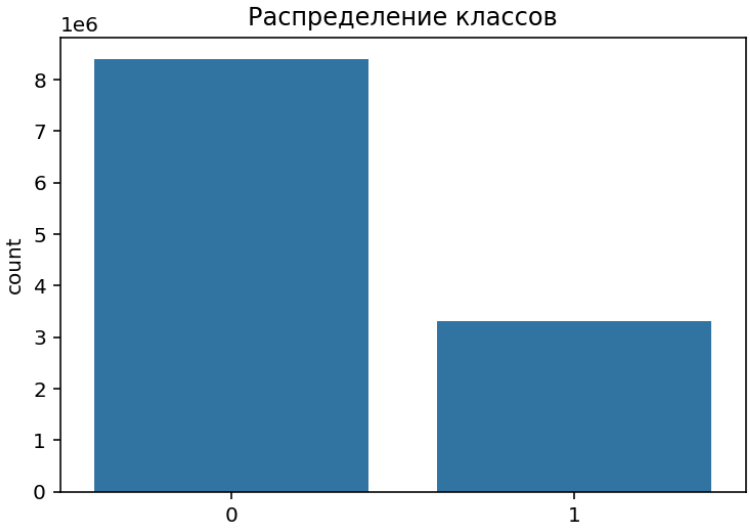


Рисунок 5 – Распределения классов

Для устранения дисбаланса была использована техника SMOTE (Synthetic Minority Over-sampling Technique). Метод синтезирует дополнительные записи меньшинства на основе линейной интерполяции между существующими объектами. SMOTE был применён исключительно к обучающей выборке после её разделения, что позволило избежать утечки информации в тестовую часть. Это дало возможность сбалансировать соотношение классов и сохранить объективность при оценке качества моделей. Распределение классов после применения SMOTE показано на рисунке 6.

Рисунок 6 – Распределения классов после SMOTE

В результате предварительной оценки удалось подтвердить, что датасет пригоден для использования в нейросетевом обучении. Отсутствие пропусков, числовой формат всех признаков, возможность стандартизации и применимость методов балансировки делают его качественной основой для построения устойчивых моделей классификации атак.

2.3 Реализация модели бинарной классификации

Реализация модели бинарной классификации сетевого трафика представлена в виде основной управляющей функции main(), в которой последовательно выполняются этапы подготовки данных, масштабирования признаков, построения и обучения модели, а также визуализации и оценки результатов. Такой подход обеспечивает удобство воспроизводимости

эксперимента и возможность быстрого запуска всех этапов обработки в едином рабочем процессе. Логика main() отражает стандартный пайплайн машинного обучения: от загрузки данных до анализа метрик и формирования финального отчёта.

Первым этапом в реализации модели бинарной классификации является загрузка исходных данных. Для этого используется функция load_binary_data(), отвечающая за считывание всех файлов с признаками и метками, необходимых для обучения. Эта функция проходит через заранее определённый список пар файлов, в которых хранятся признаки и соответствующие им метки. Каждый файл содержит набор сетевых записей, собранных в рамках различных атак или нормального трафика.

Путь к каждому файлу формируется с использованием константы DATA_DIR, указывающей на корневую директорию с датасетом. Перед загрузкой выполняется проверка наличия обоих файлов – как с признаками, так и с метками. Если какой-либо из них отсутствует, в консоль выводится предупреждение, и текущая пара пропускается.

Сами данные загружаются с помощью библиотеки pandas. Признаки считываются без заголовков (header=None) и ограничиваются числом столбцов, соответствующих размерности входного слоя нейросети (INPUT_DIM). Это позволяет игнорировать лишние данные, если они присутствуют в файле. Метки также читаются как одномерный массив, где берётся последний столбец – он содержит информацию о принадлежности записи к определённому классу.

Так как возможна ситуация, при которой число записей в файле с признаками и файле с метками не совпадает, производится уравнивание по минимальной длине. Это необходимо, чтобы каждая строка признаков имела соответствующую метку, и в обучающей выборке не оказалось "висячих" записей. Далее выполняется бинаризация меток: все ненулевые значения заменяются на 1, а все остальные – на 0. Это упрощает задачу и позволяет использовать бинарную функцию потерь в дальнейшем.

После этого данные по всем файлам объединяются в один общий массив: признаки – через вертикальное соединение (vstack), метки – через горизонтальное (hstack). На выходе функция возвращает готовые X и y, которые содержат полный набор сетевых событий, пригодных для дальнейшей предобработки и обучения модели. Код функции представлен ниже

```
    def load_binary_data():
        X_all = []
        y_all = []

        for data_file, label_file in FILES_AND_LABELS.items():
            data_path = os.path.join(DATA_DIR, data_file)
            label_path = os.path.join(DATA_DIR, label_file)

            if not os.path.exists(data_path) or not
os.path.exists(label_path):
                print(f" Missing: {data_file} or {label_file}")
                continue

            # Чтение признаков
            X = pd.read_csv(data_path, header=None).iloc[:,
:INPUT_DIM].values

            # Чтение меток
            y = pd.read_csv(label_path, header=None).iloc[:, -
1].values

            # Уравнивание размеров
            min_len = min(len(X), len(y))
            X = X[:min_len]
            y = y[:min_len]

            y = np.where(y == 1, 1, 0)  # 1 = атака, 0 =
нормальный
```

```
            X_all.append(X)
            y_all.append(y)

    return np.vstack(X_all), np.hstack(y_all)
```

После загрузки данных важным этапом подготовки является их очистка и преобразование, необходимое для корректной работы нейросетевой модели. Функция preprocess_data() выполняет сразу несколько операций: стандартизацию признаков, удаление выбросов и балансировку классов. Это позволяет повысить качество данных, устранить шум и обеспечить равномерное представление классов в обучающей выборке.

На первом шаге производится стандартизация. Входной массив X, представляющий собой признаки всех сетевых пакетов, преобразуется в DataFrame, после чего масштабируется с помощью StandardScaler. Это преобразование выравнивает среднее значение каждого признака до 0 и стандартное отклонение до 1. Стандартизация необходима для корректной работы нейросетей, особенно при использовании активаций ReLU, поскольку устраняет перекос весов и ускоряет сходимость модели.

Далее осуществляется удаление выбросов. Из всего набора признаков выбирается заданное количество – по умолчанию 10 – наиболее чувствительных к аномалиям. Для них рассчитываются квартили и межквартильный размах (IQR). Затем определяется маска, исключающая все строки, в которых хотя бы один из признаков выходит за пределы допустимого диапазона, установленного через коэффициент iqr_multiplier. Это позволяет устранить резкие аномалии в данных, которые могли бы исказить обучение и привести к ложным паттернам.

После фильтрации выбросов оставшиеся данные подвергаются балансировке. В первую очередь применяется быстрый метод – RandomUnderSampler, который уравнивает количество примеров каждого

класса путём случайного удаления избыточных записей. Это снижает объём выборки, но позволяет избежать переобучения на большинстве.

Наконец, для усиления миноритарного класса в дальнейших этапах может быть дополнительно применён SMOTE (в данной функции – опционально). Все преобразованные данные возвращаются вместе с объектом scaler, чтобы в дальнейшем применять ту же трансформацию к тестовой выборке. В консоль выводится количество удалённых строк и финальное распределение классов после балансировки. Код функции представлен ниже.

```
def preprocess_data(X, y, outlier_features=10, iqr_multiplier=1.5, random_state=42):
        """
        1. Стандартизация
        2. Удаление выбросов
        3. Балансировка классов
        """
        df = pd.DataFrame(X)

        # Стандартизация
        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(df)
        df_scaled = pd.DataFrame(X_scaled)

        # Удаление выбросов
        subset = df_scaled.iloc[:, :outlier_features]
        Q1 = subset.quantile(0.25)
        Q3 = subset.quantile(0.75)
        IQR = Q3 - Q1
        mask = ~((subset < (Q1 - iqr_multiplier * IQR)) | (subset > (Q3 + iqr_multiplier * IQR))).any(axis=1)

        X_filtered = df_scaled[mask].values
        y_filtered = np.array(y)[mask]
```

```
        print(f" Удалено выбросов: {len(X) - len(X_filtered)} строк ({(1 - len(X_filtered)/len(X))*100:.2f}%)")

        # Балансировка: быстрый undersampling
        rus = RandomUnderSampler(random_state=random_state)
        X_resampled, y_resampled = rus.fit_resample(X_filtered, y_filtered)

        print(f"    После    UnderSampler:    класс    0    - {np.sum(y_resampled==0)}, класс 1 - {np.sum(y_resampled==1)}")

        print(f"     После     SMOTE:     класс     0     - {np.sum(y_resampled==0)}, класс 1 - {np.sum(y_resampled==1)}")

        return X_resampled, y_resampled, scaler
```

Для предварительного анализа структуры и качества данных была реализована функция analyze_dataset_combined(), позволяющая автоматически формировать текстовый отчёт с ключевыми характеристиками датасета и сохранять наглядные графики. Такая диагностика перед обучением модели помогает лучше понять свойства признаков и выявить потенциальные проблемы, включая дисбаланс, выбросы и мультиколлинеарность.

На вход функция получает матрицу признаков X, вектор меток y, а также директорию для сохранения отчёта и имя текстового файла. Внутри создаётся директория dataset_analysis, где будут сохранены все артефакты. Основной отчёт формируется в виде .txt файла и содержит пять логических блоков.

Первый блок – общая информация о данных. С помощью встроенного метода info() определяется количество строк, типы признаков и наличие пропусков. Вся информация записывается в буфер и сохраняется в текстовом файле. Второй блок – основные статистики. Здесь с помощью метода

describe() рассчитываются значения среднего, стандартного отклонения, минимальных и максимальных значений для каждого признака.

Третий блок – анализ пропущенных значений. Если таковые имеются, выводится список признаков с количеством пропусков. В исследуемом датасете пропуски отсутствуют, что фиксируется в отчёте.

Четвёртый блок – распределение классов. Подсчитывается количество записей для каждого класса (0 и 1) и сохраняется в текстовом отчёте. Это особенно важно для понимания степени дисбаланса в выборке. Пятый блок – корреляционный анализ. Рассчитывается корреляционная матрица по первым десяти признакам, что даёт представление о линейных зависимостях между признаками.

Дополнительно генерируются визуализации. Сначала строится гистограмма распределения классов (countplot), затем гистограммы по первым десяти признакам, тепловая карта корреляций (heatmap) и диаграммы размаха (boxplot). Все графики сохраняются как изображения в указанной директории.

Функция возвращает путь к сформированному текстовому отчёту, что позволяет использовать его в других частях программы или при автоматическом логировании экспериментов. Код функции представлен ниже

```
    def analyze_dataset_combined(X, y,
output_dir="dataset_analysis",
report_file="analysis_report.txt"):
        os.makedirs(output_dir, exist_ok=True)
        df = pd.DataFrame(X)
        report_path = os.path.join(output_dir, report_file)

        with open(report_path, "w", encoding="utf-8") as f:
            # --- 1. Инфо ---
            f.write(" ИНФОРМАЦИЯ О ДАННЫХ:\n")
            buf = StringIO()
            df.info(buf=buf)
```

```
            f.write(buf.getvalue() + "\n")

            # --- 2. Статистика ---
            f.write(" ОСНОВНЫЕ СТАТИСТИКИ:\n")
            desc = df.describe()
            f.write(desc.to_string() + "\n\n")

            # --- 3. Пропущенные значения ---
            f.write(" ПРОПУЩЕННЫЕ ЗНАЧЕНИЯ:\n")
            missing = df.isnull().sum()
            if missing.sum() == 0:
                f.write("Нет пропущенных значений.\n\n")
            else:
                f.write(str(missing[missing > 0]) + "\n\n")

            # --- 4. Распределение классов ---
            f.write(" РАСПРЕДЕЛЕНИЕ КЛАССОВ:\n")
            class_counts = pd.Series(y).value_counts()
            f.write(class_counts.to_string() + "\n\n")

            # --- 5. Корреляционная матрица ---
            f.write("   СТАТИСТИКА   КОРРЕЛЯЦИИ   (первые   10
признаков):\n")
            corr = df.corr()
            f.write(corr.iloc[:10, :10].to_string() + "\n")

        # --- Сохраняем графики ---
        sns.countplot(x=y)
        plt.title("Распределение классов")
        plt.savefig(os.path.join(output_dir,
"class_distribution.png"))
        plt.close()

        df.iloc[:, :10].hist(figsize=(14, 8))
        plt.suptitle("Гистограммы первых 10 признаков")
```

```
        plt.savefig(os.path.join(output_dir,
"feature_histograms.png"))
        plt.close()

        plt.figure(figsize=(10, 8))
        sns.heatmap(corr, cmap='coolwarm', square=True)
        plt.title("Тепловая карта корреляций")
        plt.savefig(os.path.join(output_dir,
"correlation_heatmap.png"))
        plt.close()

        plt.figure(figsize=(12, 6))
        sns.boxplot(data=df.iloc[:, :10])
        plt.title("Boxplot первых 10 признаков")
        plt.savefig(os.path.join(output_dir, "boxplots.png"))
        plt.close()

        print(f"  Анализ  завершён.  Отчёт  сохранён  в:
{report_path}")
        return report_path  # чтобы ты мог отправить его мне
потом
```

Построение модели бинарной классификации реализуется в виде функции create_binary_model(), которая возвращает готовую к обучению нейросетевую архитектуру. Используется последовательная модель (Sequential) из библиотеки Keras, что делает структуру кода прозрачной и читаемой. Архитектура сети была подобрана экспериментально с учётом баланса между сложностью и производительностью.

Входной слой принимает вектор признаков размером input_dim, соответствующий числу характеристик в обработанном датасете (в данном случае 115). Первый скрытый слой содержит 128 нейронов и использует функцию активации ReLU, которая эффективно справляется с задачей выделения нелинейных зависимостей. Следом идёт слой BatchNormalization, стабилизирующий распределение входных данных и способствующий

ускоренной сходимости на этапе обучения. Для борьбы с переобучением применяется Dropout с вероятностью отключения нейронов 0.4.

Второй скрытый слой построен аналогично, но содержит 64 нейрона и использует чуть меньшую вероятность Dropout – 0.3. Такое прогрессивное снижение плотности нейронов позволяет постепенно сжимать информацию, уменьшая размерность и сохраняя при этом важные признаки. Повторное применение BatchNormalization повышает устойчивость к колебаниям градиента.

Выходной слой состоит из одного нейрона с активацией sigmoid, что позволяет интерпретировать результат как вероятность принадлежности к положительному классу – атакующему трафику. Такой подход логически соответствует постановке задачи бинарной классификации.

После определения архитектуры модель компилируется. В качестве оптимизатора выбран Adam – один из самых стабильных и популярных алгоритмов в задачах классификации. Функция потерь – binary_crossentropy, идеально подходящая для двухклассовых задач с вероятностным выходом. В качестве метрик отслеживаются accuracy, precision и recall, что позволяет не только оценивать общее качество, но и понимать баланс между ложными срабатываниями и пропущенными атаками.

Функция возвращает готовую модель, пригодную для обучения и последующей оценки. Код функции представлен ниже

```
def create_binary_model(input_dim):
    model = keras.Sequential([
        layers.Input(shape=(input_dim,)),
        layers.Dense(128, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.4),
        layers.Dense(64, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.3),
        layers.Dense(1, activation='sigmoid')
    ])
```

```
        model.compile(
            optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy',      keras.metrics.Precision(),
keras.metrics.Recall()]
        )
        return model
```

После определения архитектуры и подготовки выборок запускается процесс обучения модели. Для этого используется метод fit(), который принимает вектор признаков и соответствующие метки. Обучение проводится на 80% исходных данных, оставшиеся 20% используются в качестве валидационной выборки. Это позволяет отслеживать поведение модели не только на тех данных, на которых она обучается, но и на тех, которые ей ранее не встречались.

Размер пакета (batch_size) задаётся через константу BATCH_SIZE, которая определяет количество примеров, обрабатываемых за одну итерацию. Параметр epochs указывает общее количество эпох обучения, то есть полных проходов по обучающей выборке. Оптимальные значения этих параметров подбирались экспериментально и обеспечивают баланс между временем обучения и качеством результата.

Важным элементом является использование механизма ранней остановки (EarlyStopping). Он отслеживает валидационную функцию потерь и завершает обучение, если в течение трёх эпох не происходит её улучшения. Это позволяет не только сократить время обучения, но и избежать переобучения. Аргумент restore_best_weights=True гарантирует возврат к состоянию модели с минимальной валидационной ошибкой, даже если оно достигалось не на последней эпохе.

Вывод прогресса обучения осуществляется в консоль благодаря параметру verbose=1, что удобно для мониторинга метрик в реальном времени. Результаты обучения сохраняются в объекте history, который позже

используется для построения графиков точности и функции потерь по эпохам. Код представлен ниже.

```
        history = model.fit(
            X_train, y_train,
            validation_data=(X_test, y_test),
            batch_size=BATCH_SIZE,
            epochs=EPOCHS,
            callbacks=[keras.callbacks.EarlyStopping(patience=3,
restore_best_weights=True)],
            verbose=1
        )
```

После завершения обучения полезно визуально оценить поведение модели по эпохам. Для этого используется функция plot_history(), принимающая на вход объект history, возвращаемый методом fit(). В этом объекте содержатся значения метрик точности и функции потерь на обучающей и валидационной выборках, зафиксированные на каждой эпохе.

Графическая визуализация помогает быстро определить, происходило ли переобучение, насколько устойчиво модель сходилась и в какой момент наблюдалась стабилизация метрик. Визуальный анализ также позволяет оценить необходимость ранней остановки или корректировки параметров обучения.

Функция создаёт два подграфика. Первый отображает изменение точности (accuracy) по эпохам: отдельно для обучающей и валидационной выборок. Второй график аналогично показывает динамику функции потерь (loss). Используются подписи и легенды для наглядности. Графики выравниваются и отображаются рядом, благодаря использованию subplot(1, 2, ...).

Команда tight_layout() обеспечивает корректное распределение пространства между подграфиками, а вызов plt.show() отображает результат на экране. Такая визуализация делает поведение модели понятным даже без

анализа числовых значений и может быть полезна при докладной презентации результатов.

```
    def plot_history(history):
        plt.figure(figsize=(12, 5))
        plt.subplot(1, 2, 1)
        plt.plot(history.history['accuracy'], label='Train Acc')
        plt.plot(history.history['val_accuracy'],     label='Val
Acc')
        plt.legend()
        plt.title('Accuracy')

        plt.subplot(1, 2, 2)
        plt.plot(history.history['loss'], label='Train Loss')
        plt.plot(history.history['val_loss'], label='Val Loss')
        plt.legend()
        plt.title('Loss')

        plt.tight_layout()
        plt.show()
```

Оценка качества модели проводится с помощью функции evaluate_model(), которая сочетает в себе числовой анализ и визуализацию результатов классификации. На вход подаются обученная модель, тестовая выборка признаков и соответствующие метки. Это позволяет объективно оценить способность модели выявлять атаки на данных, которые не участвовали в обучении.

Сначала модель генерирует вероятностные предсказания при помощи метода predict(). Результат приводится к одномерному массиву (flatten()), чтобы упростить дальнейшую обработку. Порог классификации установлен на уровне 0.5: все значения выше интерпретируются как принадлежность к классу "атака", остальные – к классу "нормальный".

Далее строится матрица ошибок (Confusion Matrix). Она позволяет наглядно оценить соотношение истинных положительных, ложных положительных, истинных отрицательных и ложных отрицательных

классификаций. С помощью библиотеки Seaborn создаётся тепловая карта, где визуально отображаются основные ошибки модели. Это помогает выявить смещения предсказаний и критически важные зоны, в которых возможны сбои в обнаружении атак.

После визуализации выводится текстовый отчёт о классификации (Classification Report). Он включает такие метрики, как точность (precision), полнота (recall), F1-мера и общая accuracy – все рассчитаны отдельно по каждому классу. Это даёт возможность оценить, насколько сбалансирована модель в отношении разных типов трафика.

Завершающим этапом является построение ROC-кривой (Receiver Operating Characteristic), отображающей соотношение между чувствительностью (True Positive Rate) и уровнем ложных срабатываний (False Positive Rate) при различных порогах. Также рассчитывается AUC (Area Under Curve) – интегральная метрика качества ранжирования. Значение AUC, близкое к 1, указывает на способность модели эффективно различать классы даже при изменении границы классификации.

Функция сочетает числовые метрики и графическое представление, предоставляя исчерпывающую картину работы модели на тестовой выборке.

Код функции представлен ниже.

```
def evaluate_model(model, X_test, y_test):
    y_pred_prob = model.predict(X_test).flatten()
    y_pred = (y_pred_prob > 0.5).astype(int)

    print("\\nConfusion Matrix:")
    cm = confusion_matrix(y_test, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=['Normal', 'Attack'], yticklabels=['Normal',
'Attack'])
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.show()
```

```
print("\\nClassification Report:")
print(classification_report(y_test, y_pred, digits=4))

auc = roc_auc_score(y_test, y_pred_prob)
fpr, tpr, _ = roc_curve(y_test, y_pred_prob)
plt.figure()
plt.plot(fpr, tpr, label=f'AUC = {auc:.4f}')
plt.plot([0, 1], [0, 1], 'k--')
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
plt.grid()
plt.show()
```

2.4 Реализация моделей многоклассовой классификации

На следующем этапе эксперимента задача классификации была расширена до многоклассовой. Вместо простого деления трафика на "нормальный" и "вредоносный" теперь требуется не только обнаружить факт атаки, но и определить её конкретный тип. Такая постановка ближе к реальным условиям работы систем сетевой безопасности, где важно понимать, с каким именно видом угрозы имеет дело инфраструктура.

Для решения задачи были построены две независимые модели: многослойный перцептрон (MLP) и сверточная нейронная сеть (1D-CNN). Обе архитектуры обучались на одном и том же наборе признаков, но по-разному обрабатывали внутренние зависимости. Это позволило провести сравнение по точности, устойчивости к редким классам, скорости обучения и общему качеству классификации.

2.4.1 Загрузка данных

Для решения задачи многоклассовой классификации была разработана функция load_and_balance_data(), позволяющая загружать данные из разных источников, формировать целевые метки по типам атак и проводить

балансировку классов до их объединения в единый обучающий массив. Это особенно важно в условиях, когда разные типы атак представлены неравномерно, а нормальный трафик доминирует.

Функция принимает на вход словарь, в котором каждому файлу с признаками сопоставлен файл с метками и соответствующий номер атакующего класса. Это позволяет формировать целевой вектор, где нулевое значение соответствует нормальному трафику, а значения от 1 до N – конкретным типам атак.

На первом этапе проверяется наличие файлов. Если один из них отсутствует, выполнение продолжается со следующим набором. После этого производится загрузка ограниченного числа строк (CHUNK_SIZE) из файла с признаками и соответствующих меток из последнего столбца файла с метками. Размер меток приводится к длине данных, чтобы избежать рассинхронизации.

Целевые значения формируются на основе условия: все записи с меткой 1 интерпретируются как атака соответствующего класса, а остальные – как нормальный трафик (метка 0). Далее для каждого класса выполняется балансировка: выбирается одинаковое количество записей атакующего и нормального трафика. Это позволяет добиться симметрии в выборке и улучшить обучение модели.

В случае, если для текущего класса в файле не удалось найти ни одной записи, в консоль выводится сообщение, и обработка продолжается. После формирования сбалансированной выборки записи добавляются в общий массив. Когда все файлы обработаны, данные объединяются и перемешиваются с помощью случайной перестановки индексов.

На выходе формируются два массива: X – с признаками и y – с метками классов. Эти данные используются в дальнейшем при обучении моделей MLP и CNN. Код функции представлен ниже.

```
def load_and_balance_data(file_class_mapping, max_samples_per_class=500000):
```

```
        """Загрузка и балансировка данных"""
        all_data = []
        all_labels = []

        for data_file, (label_file, attack_class) in
file_class_mapping.items():
            data_path = os.path.join(DATA_DIR, data_file)
            label_path = os.path.join(DATA_DIR, label_file)

            if not os.path.exists(data_path) or not
os.path.exists(label_path):
                print(f" Файлы не найдены: {data_path} или
{label_path}")
                continue

            try:
                # Чтение данных
                data_chunk = pd.read_csv(data_path, header=None,
nrows=CHUNK_SIZE)
                labels = pd.read_csv(label_path,
header=None).iloc[:, -1].values  # берем последний столбец

                # Обрезаем метки по размеру данных
                labels = labels[:len(data_chunk)]

                # Создаем метки: 1 → attack_class, 0 →
нормальный трафик
                y = np.where(labels == 1, attack_class, 0)
                X = data_chunk.iloc[:, :INPUT_DIM].values

                # Балансировка классов
                attack_idx = np.where(y == attack_class)[0]
                normal_idx = np.where(y == 0)[0]

                # Берем одинаковое количество примеров
```

```
                n_samples = min(len(attack_idx),
len(normal_idx), max_samples_per_class)

                if n_samples == 0:
                    print(f"    Нет    данных    для    класса
{attack_class} в файле {data_file}")
                    continue

                selected_idx = np.concatenate([
                    np.random.choice(attack_idx,      n_samples,
replace=False),
                    np.random.choice(normal_idx,      n_samples,
replace=False)
                ])

                all_data.append(X[selected_idx])
                all_labels.append(y[selected_idx])

            except Exception as e:
                print(f"  Ошибка  при  обработке  {data_file}:
{str(e)}")
                continue

        if not all_data:
            raise ValueError("Нет данных для обработки!")

        X = np.vstack(all_data)
        y = np.hstack(all_labels)

        # Перемешиваем данные
        indices = np.random.permutation(len(X))
        X, y = X[indices], y[indices]

        return X, y
```

2.4.2 Архитектура модели MLP

Для задачи многоклассовой классификации была реализована модель на основе многослойного перцептрона (MLP), способная эффективно обрабатывать числовые признаки и выявлять сложные нелинейные зависимости. Архитектура сети строится последовательно, что делает её легко настраиваемой и адаптируемой под особенности данных. Конкретная реализация направлена на обеспечение хорошего баланса между глубиной сети, количеством параметров и устойчивостью к переобучению.

На вход подаётся вектор признаков размером input_dim, соответствующий числу извлечённых из сетевого трафика характеристик. Первый скрытый слой содержит 256 нейронов с функцией активации ReLU. Использование ReLU ускоряет обучение и устраняет проблему затухающего градиента. За первым полносвязным слоем следует BatchNormalization, выравнивающий распределение активаций и способствующий стабилизации процесса обучения. Затем применяется Dropout с вероятностью 0.5 – это довольно сильная регуляризация, позволяющая значительно снизить переобучение.

Второй скрытый слой содержит 128 нейронов и сопровождается BatchNormalization и Dropout с вероятностью 0.4. Такое постепенное уменьшение количества нейронов и коэффициентов отсева формирует своеобразную «воронку признаков», позволяя модели постепенно сжимать и обобщать входные данные. Эта стратегия помогает избежать перенасыщения параметрами на глубине модели, сохранив при этом выраженную обучающую способность.

Третий скрытый слой включает 64 нейрона с аналогичной схемой активации, нормализации и Dropout (0.3). Плавное снижение количества нейронов на каждом уровне способствует эффективному выделению ключевых признаков и формированию устойчивых внутренних представлений. За ним следует четвёртый скрытый слой на 32 нейрона, после которого применяется только BatchNormalization, без Dropout. Это позволяет сохранить полезные сигналы на выходе перед последней трансформацией.

Финальный слой содержит n_classes нейронов (по числу классов), каждый из которых соответствует определённому типу сетевого трафика или атаки. Для этого слоя используется активация softmax, позволяющая модели возвращать вероятностное распределение по классам. Такая структура идеально подходит для многоклассовой классификации, поскольку softmax гарантирует, что сумма всех выходных значений равна единице.

После определения архитектуры модель компилируется. В качестве оптимизатора используется Adam с заданной скоростью обучения 0.001. Это один из наиболее устойчивых и адаптивных оптимизаторов, хорошо зарекомендовавший себя при работе с нейросетевыми структурами. Функция потерь выбрана как categorical_crossentropy, что соответствует формату меток, приведённых к one-hot-формату. В качестве метрик мониторинга указываются accuracy, precision и recall – это даёт возможность отслеживать не только общую точность модели, но и её чувствительность к миноритарным классам.

Созданная модель возвращается из функции create_model() и готова к последующему обучению. Такая архитектура легко масштабируется и может быть дополнительно оптимизирована за счёт изменения глубины, количества нейронов или параметров регуляризации.

Код функции представлен ниже

```
def create_model(input_dim, n_classes):
    """Создание модели нейронной сети"""
    model = keras.Sequential([
        layers.Input(shape=(input_dim,)),
        layers.Dense(256, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.5),

        layers.Dense(128, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.4),
```

```
            layers.Dense(64, activation='relu'),
            layers.BatchNormalization(),
            layers.Dropout(0.3),

            layers.Dense(32, activation='relu'),
            layers.BatchNormalization(),

            layers.Dense(n_classes, activation='softmax')
        ])

        model.compile(

optimizer=keras.optimizers.Adam(learning_rate=0.001),
            loss='categorical_crossentropy',
            metrics=['accuracy',
                     keras.metrics.Precision(name='precision'),
                     keras.metrics.Recall(name='recall')]
        )
        return model
```

2.4.3 Архитектура модели CNN

Альтернативным подходом к многоклассовой классификации сетевого трафика стала реализация сверточной нейронной сети на основе одномерных свёрток (1D-CNN). Такая архитектура была выбрана с учётом временной природы сетевых данных, где каждый набор признаков может рассматриваться как упорядоченная последовательность, содержащая локальные зависимости. Модель эффективно сочетает в себе механизмы извлечения локальных признаков и полносвязную классификацию, что делает её особенно ценной при анализе сетевых атак с выраженными временными паттернами.

На вход модель принимает данные в формате input_shape, представляющем собой трёхмерный тензор с размерами (samples, 115, 1). Это необходимо для корректной работы сверточных слоёв, обрабатывающих последовательности по одному измерению. Первый слой – одномерная

свёртка Conv1D с 64 фильтрами и размером ядра 3. Активация ReLU используется для обеспечения нелинейности, а padding='same' сохраняет размерность выходного тензора. Сразу после свёртки применяются BatchNormalization и Dropout с вероятностью 0.3, что стабилизирует градиенты и снижает переобучение.

Второй сверточный слой содержит 128 фильтров с тем же размером ядра и аналогичной конфигурацией. Увеличение количества фильтров позволяет извлекать более абстрактные и сложные признаки из данных. Регуляризация усиливается за счёт Dropout с коэффициентом 0.4, что особенно важно на более глубоких уровнях, где возрастает риск переобучения.

После извлечения признаков сверточные карты преобразуются в вектор с помощью слоя Flatten. Это необходимо для перехода от пространственного представления к векторному, пригодному для подачи в полносвязные слои. Далее следует полносвязный слой из 128 нейронов с функцией активации ReLU. Как и в предыдущих уровнях, применяются BatchNormalization и Dropout с коэффициентом 0.5, поскольку плотность параметров на этом уровне особенно высока.

Финальный слой содержит количество нейронов, равное числу классов (n_classes). Активация softmax используется для получения вероятностного распределения по всем категориям. Такой выходной формат особенно удобен в задачах многоклассовой классификации, так как позволяет интерпретировать предсказания модели как уверенность в принадлежности к каждому из возможных классов.

Компиляция модели выполняется с использованием оптимизатора Adam и функции потерь categorical_crossentropy. Выбор именно этой функции потерь обусловлен тем, что метки классов представлены в one-hot-формате. В список метрик включены accuracy, а также Precision и Recall из keras.metrics, что позволяет отслеживать эффективность модели не только в целом, но и по отношению к качеству предсказаний каждого класса. Эти

метрики особенно важны при наличии редких, но критически важных атакующих классов.

Созданная модель демонстрирует высокую обучаемость, устойчивость к переобучению и способность извлекать полезные зависимости в последовательностях признаков. Применение сверточных слоёв делает возможным автоматическое выявление локальных паттернов, связанных с аномальным поведением сетевых объектов, что особенно актуально в условиях современных киберугроз.

```
    def create_cnn_model(input_shape, n_classes):
        model = keras.Sequential([
            layers.Input(shape=input_shape),  # (1, 115)

            # 1D Convolutional Layers
            layers.Conv1D(64, kernel_size=3, activation='relu',
padding='same'),
            layers.BatchNormalization(),
            layers.Dropout(0.3),

            layers.Conv1D(128, kernel_size=3, activation='relu',
padding='same'),
            layers.BatchNormalization(),
            layers.Dropout(0.4),

            layers.Flatten(),

            # Fully Connected Layers
            layers.Dense(128, activation='relu'),
            layers.BatchNormalization(),
            layers.Dropout(0.5),

            layers.Dense(n_classes, activation='softmax')
        ])

        model.compile(
```

```
            optimizer='adam',
            loss='categorical_crossentropy',
            metrics=['accuracy',
                     Precision(name='precision'),   # Используем
класс Precision
                     Recall(name='recall')]         # Используем
класс Recall
         )
         return model
```

2.4.4 Оценка качества моделей

После завершения обучения модели проводится её итоговая оценка на тестовой выборке. Функция evaluate_model() предназначена для получения ключевых метрик качества, генерации классификационного отчёта и визуализации матрицы ошибок. Такой подход позволяет не только количественно оценить поведение модели, но и выявить проблемные классы или перекосы в предсказаниях.

На вход функция принимает обученную модель и разделённые тестовые данные. Вектор меток преобразуется в категориальный формат с помощью to_categorical(), что необходимо для корректного вычисления потерь и метрик при многоклассовой классификации.

Модель оценивается с помощью метода evaluate(), который возвращает список метрик, указанных при компиляции модели. Обычно это значения функции потерь, точности, точности (precision) и полноты (recall). Если часть метрик не была задана, значения precision и recall инициализируются нулями, чтобы избежать ошибок при логировании.

Дополнительно формируются предсказания модели на тестовой выборке. С помощью argmax() из вероятностного распределения по классам извлекаются итоговые предсказанные метки. На основе этих значений строится матрица ошибок (confusion matrix), где фиксируются все верные и ошибочные классификации по каждому классу.

Для более детальной оценки создаётся classification report с точностью, полнотой, F1-мерой и количеством примеров по каждому классу. Он сохраняется в виде JSON-файла, что упрощает автоматизированную обработку и последующий анализ. Матрица ошибок сохраняется как .npy-массив и также визуализируется в виде тепловой карты. Это даёт возможность наглядно оценить, где чаще всего модель ошибается, и какие классы могут быть перепутаны.

Все ключевые метрики также выводятся в консоль для быстрой интерпретации. В завершение функция возвращает словарь с финальными значениями метрик, матрицей ошибок и классификационным отчётом в пригодном для сериализации виде.

```
    def evaluate_model(model, X_test, y_test):
        """Оценка модели и вывод метрик"""
        y_test_cat = keras.utils.to_categorical(y_test, num_classes=7)

        # Оценка модели
        test_metrics = model.evaluate(X_test, y_test_cat, verbose=0)

        # Для моделей с несколькими метриками
        if len(test_metrics) == 4:  # loss, accuracy, precision, recall
            test_loss, test_acc, test_precision, test_recall = test_metrics
        else:  # Если метрик меньше, задаем значения по умолчанию
            test_loss, test_acc = test_metrics[0], test_metrics[1]
            test_precision, test_recall = 0.0, 0.0
        # Получаем предсказания для дополнительного анализа
        y_pred = model.predict(X_test, verbose=0)
        y_pred_classes = np.argmax(y_pred, axis=1)
        # Confusion Matrix
```

```
        cm = confusion_matrix(y_test, y_pred_classes)

        # Classification Report
        cl_report        =        classification_report(y_test,
y_pred_classes, digits=4, output_dict=True)

        # Сохраняем confusion matrix
        np.save('mlp_confusion_matrix.npy', cm)

        # Сохраняем classification report
        with open('mlp_classification_report.json', 'w') as f:
            json.dump(cl_report, f)

        # Вывод в консоль
        print(f"\n Final Test Metrics:")
        print(f"Accuracy: {test_acc:.4f}")
        print(f"Precision: {test_precision:.4f}")
        print(f"Recall: {test_recall:.4f}")
        print(f"Loss: {test_loss:.4f}")

        # Визуализация confusion matrix
        plt.figure(figsize=(10, 8))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                    xticklabels=['Normal'] + [f'Attack {i}' for
i in range(1, 7)],
                    yticklabels=['Normal'] + [f'Attack {i}' for
i in range(1, 7)])
        plt.title('Confusion Matrix')
        plt.xlabel('Predicted')
        plt.ylabel('True')
        plt.savefig('mlp_confusion_matrix.png')
        plt.close()  # Закрываем график, чтобы не показывался в
консоли
        print("\nClassification Report:")
        print(classification_report(y_test,      y_pred_classes,
digits=4))
```

```
        return {
            'test_metrics': {
                'loss': float(test_loss),
                'accuracy': float(test_acc),
                'precision': float(test_precision),
                'recall': float(test_recall)
            },
            'confusion_matrix':  cm.tolist(),    #  Преобразуем
numpy array в list для JSON
            'classification_report': cl_report
        }
```

3 Экспериментальная часть
3.1 Бинарная классификация

Для решения задачи обнаружения аномального трафика была реализована модель полносвязной нейронной сети (Multi-Layer Perceptron), способная учитывать как линейные, так и нелинейные зависимости между признаками. Входной слой включал 115 нейронов - по числу стандартизированных признаков, описывающих характеристики сетевого пакета. За ним следовали два скрытых слоя: первый состоял из 128 нейронов с функцией активации ReLU и Dropout-регуляризацией 0.3, второй - из 64 нейронов с аналогичными параметрами. Для стабилизации обучения и ускорения сходимости перед каждым скрытым слоем применялась пакетная нормализация. Выходной слой был представлен одним нейроном с сигмоидной активацией, что позволяло интерпретировать выходное значение как вероятность принадлежности пакета к классу «атака».

В качестве функции потерь использовалась бинарная кросс-энтропия, так как задача сводится к разделению объектов на два класса. Оптимизация выполнялась с помощью алгоритма Adam с начальными параметрами: скорость обучения - 0.001, $\beta 1 = 0.9$, $\beta 2 = 0.999$. Чтобы избежать переобучения и улучшить обобщающую способность модели, была реализована стратегия ранней остановки. Если валидационная функция потерь не улучшалась на протяжении трёх эпох подряд, обучение прекращалось автоматически. Обучающая выборка составляла 80% исходных данных, оставшиеся 20% использовались для тестирования.

До подачи в модель все признаки прошли стандартизацию. Из-за выраженного дисбаланса классов - преобладания нормального трафика - к обучающей выборке была применена техника SMOTE. Она позволила синтезировать недостающие записи атакующего трафика и сбалансировать распределение меток. При этом трансформация применялась только к обучающей части, что исключало утечку данных и обеспечивало объективную оценку модели.

Во время обучения отслеживались значения функции потерь и точности на обеих выборках. Графики метрик по эпохам показали

устойчивую динамику - потери последовательно снижались, а точность постепенно росла. Валидирующая точность стабилизировалась в районе 93.5%, без признаков переобучения. Это говорит о том, что модель способна сохранять устойчивость даже при наличии большого числа входных признаков и высокой вариативности трафика. График изменения точности и потерь обучения показан на рисунке 7.

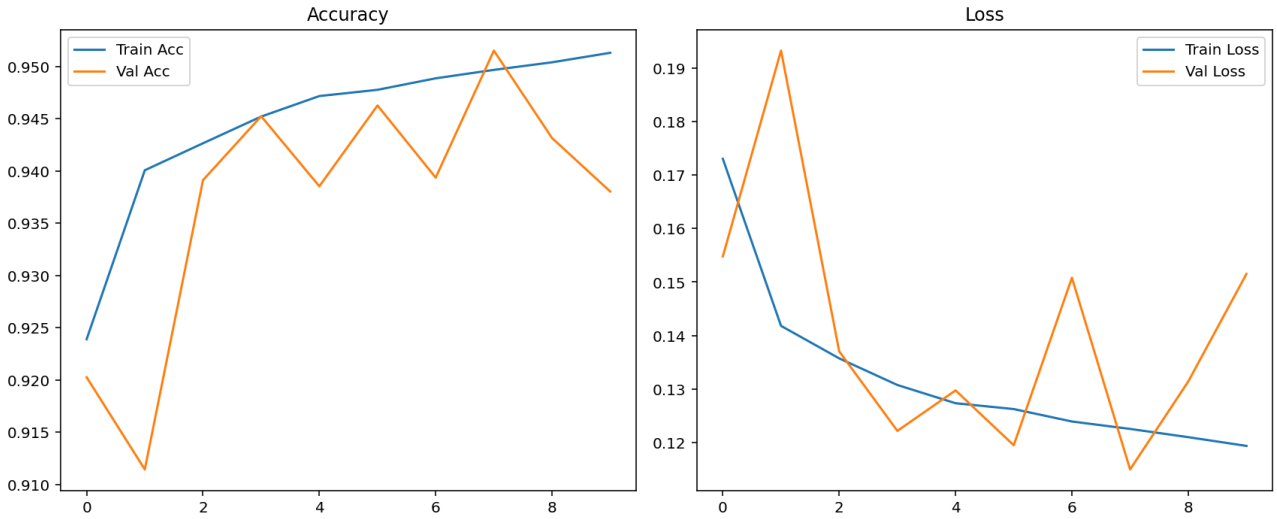


Рисунок 7 – Графики изменения точности и потерь на обучении и валидации

Оценка качества на тестовой выборке продемонстрировала высокие показатели. Общая точность составила 93.80%, что является хорошим результатом для условий с дисбалансом классов. Детальный анализ метрик по классам показал сбалансированное поведение модели. Для нормального трафика (класс 0) точность достигла 97.27%, полнота - 94.00%, F1-мера - 0.9561. Для атакующего трафика (класс 1) точность составила 85.95%, полнота - 93.30%, F1-мера - 0.8947. Это говорит о высокой чувствительности модели к обнаружению атакующих пакетов, при минимальном уровне ложных срабатываний. Метрики показаны на таблице 2.

Таблица 2 – Метрики качества классификации модели

Метрика	Класс	Precision	Recall	F1-score	Support
	0	0.9727	0.9400	0.9561	1 678 904
	1	0.8595	0.9330	0.8947	660 427
Accuracy	–	–	–	0.9380	2 339 331

Macro avg	–	0.9161	0.9365	0.9254	2 339 331
Weighted avg	–	0.9408	0.9380	0.9388	2 339 331

По матрице ошибок видно, что из 1 678 904 записей нормального трафика, правильно классифицировано 1 578 158. Остальные 100 746 ошибочно отнесены к атакующему классу. В то же время, из 660 427 атакующих записей 616 194 были распознаны верно, а 44 233 - ошибочно определены как нормальные. Эти показатели свидетельствуют о высокой полноте по атакующему классу, что особенно критично для систем информационной безопасности, где упущенная угроза может привести к серьёзным последствиям. Матрица ошибок показана на рисунке 8.

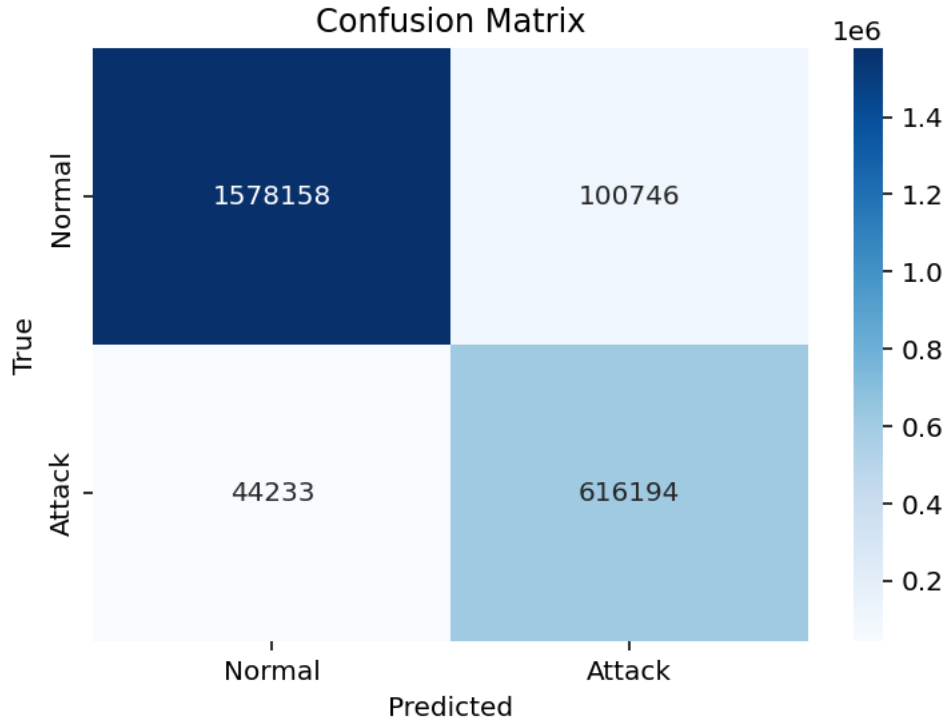


Рисунок 8 – Матрица ошибок (Confusion Matrix)

Дополнительно была рассчитана интегральная метрика ROC-AUC, равная 0.9867. Это значение близко к 1, что указывает на практически идеальную способность модели разделять классы при варьировании порога принятия решения. Построенная ROC-кривая также продемонстрировала высокую площадь под кривой, что подтверждает стабильность модели при различных значениях порога, она показана на рисунке 9.

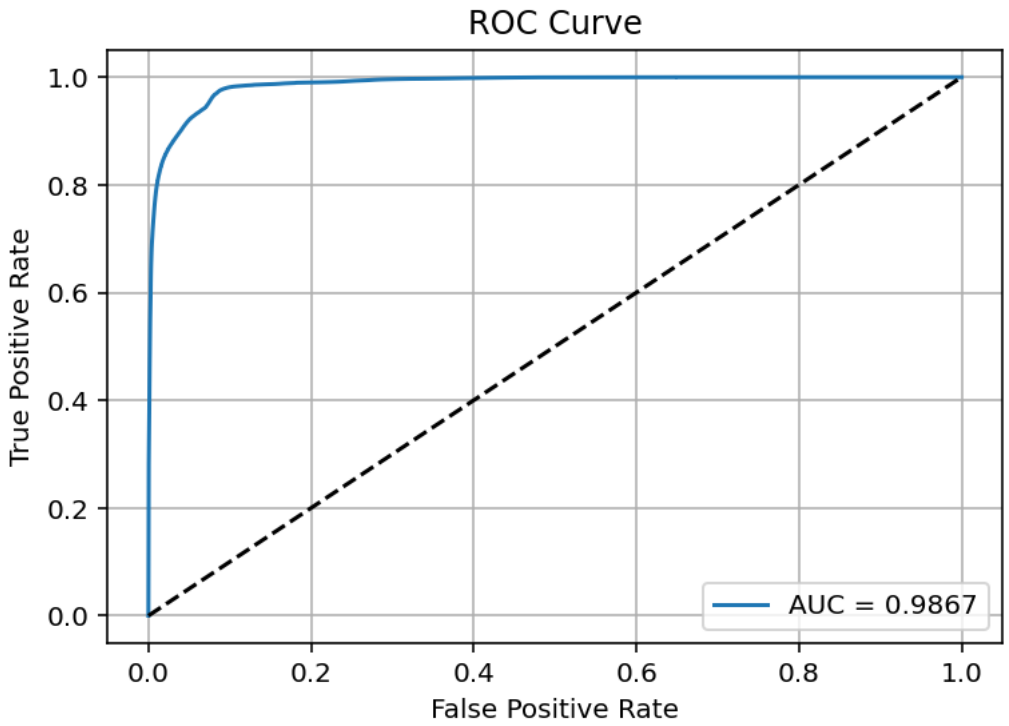


Рисунок 9 – ROC- кривая модели классификации

Проведённый эксперимент позволяет сделать вывод о том, что разработанная нейросетевая модель хорошо справляется с задачей обнаружения сетевых атак. Она демонстрирует высокую точность и устойчивость к дисбалансу классов. Использование SMOTE и методов регуляризации позволило улучшить обобщающую способность модели, а архитектура с несколькими скрытыми слоями эффективно извлекает как линейные, так и нелинейные закономерности в сетевом трафике. Модель может быть рекомендована в качестве основы для более сложных систем обнаружения вторжений (IDS), особенно в условиях большого объёма данных и ограниченного времени обработки.

3.2 Многоклассовая классификация

В рамках эксперимента было проведено сравнение двух нейросетевых архитектур - многослойного перцептрона (MLP) и сверточной нейронной сети (1D-CNN) - на задаче многоклассовой классификации сетевого трафика. В качестве основы использован Kitsune Network Attack Dataset, содержащий как нормальные, так и вредоносные пакеты, распределённые по семи классам. Целью эксперимента являлась оценка точности, устойчивости к

дисбалансу, поведения на редких классах, времени обучения и общей пригодности моделей для задач обнаружения различных типов атак.

Для обеих моделей использовались одни и те же входные данные: 115 числовых признаков, стандартизированных заранее. Для борьбы с дисбалансом применялась техника SMOTE, позволяющая синтезировать записи по миноритарным классам. Выходной слой содержал 7 нейронов, по числу классов, с функцией активации softmax, обеспечивающей вероятностное распределение по всем категориям.

Архитектура MLP включала пять полносвязных слоёв с постепенным уменьшением числа нейронов: от 256 до 32. Регуляризация реализована через слои Dropout и BatchNormalization, что обеспечивало устойчивость и ускоренную сходимость. Обучение проводилось с использованием функции потерь Categorical Crossentropy и оптимизатора Adam.

Модель 1D-CNN включала два сверточных слоя с 64 и 128 фильтрами, ядро - 3×1. Такая конфигурация позволяла извлекать локальные зависимости между признаками во временном разрезе. После каждого сверточного слоя применялись нормализация и Dropout, а на выходе использовался полносвязный классификатор. Данные подавались в трёхмерном виде (samples, 115, 1), что соответствовало требованиям 1D-свёртки.

Обе модели обучались с ранней остановкой, контролируя валидационные метрики. Процесс мониторился по значениям точности и функции потерь. Графики точности и потерь для моделей MLP и CNN показаны на рисунках 10 и 11.

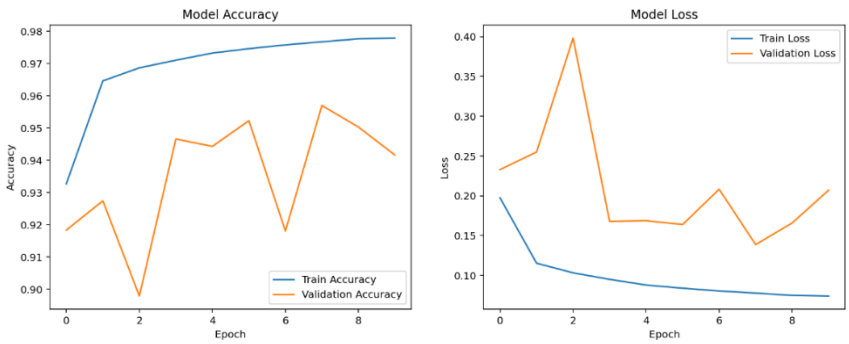


Рисунок 10 – Графики accuracy и loss по эпохам для модели MLP

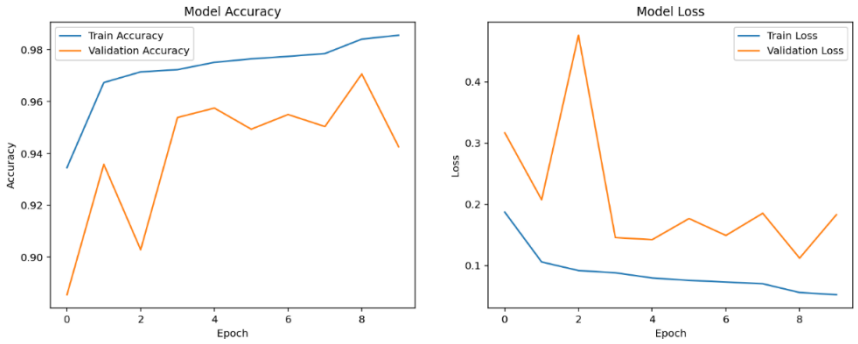


Рисунок 11 - Графики accuracy и loss по эпохам для модели CNN

MLP достиг максимальной валидационной точности (97.06%) уже на 9-й эпохе, продемонстрировав быстрое обучение. CNN показывал более постепенную сходимость, без резких скачков, с устойчивым снижением val_loss и плавным ростом точности. Это свидетельствует о лучшей устойчивости CNN к переобучению - за счёт свёрточной природы архитектура "фокусировалась" на значимых паттернах, фильтруя шумовые признаки.

По метрике accuracy модели показали практически идентичные результаты, однако различия стали очевидны при анализе precision, recall и F1 по отдельным классам. CNN обеспечила более стабильное качество на миноритарных классах. Особенно это проявилось при классификации атак, которые встречаются редко, но являются критичными для безопасности. Результат сравнения моделей показан на таблице 3

Таблица 3 – Сравнение результатов моделей MLP и CNN

Модель	Accuracy	Precision	Recall	F1-score	Время обучения (с)
MLP	94.26%	94.65%	94.16%	94.28%	173.06
CNN	94.16%	94.45%	94.08%	94.26%	288.34

Анализ матриц ошибок выявил, что у MLP наблюдаются трудности при классификации некоторых атак. В частности, 2067 нормальных пакетов были ошибочно отнесены к ARP MitM, а также произошли смешения между классами Active Wiretap и ARP MitM - 3344 случая. У CNN эти ошибки проявлялись в меньшей степени. Количество ложных срабатываний

уменьшилось, а точность по классу “1” (Active Wiretap) достигла recall = 0.9978 против 0.9972 у MLP.

Наиболее редкий класс – “6” (SSL Renegotiation) - оказался хорошо различим обеими моделями. При этом CNN достигла recall на уровне 99.6%, уступив MLP по precision. У MLP наблюдалась чуть меньшая чувствительность, но более стабильные предсказания по метке. Это указывает на разницу в стратегии моделей: одна более склонна к агрессивному выявлению, другая - к консервативной точности. Матрицы ошибок моделей MLP и CNN показаны на рисунках 12 и 13.

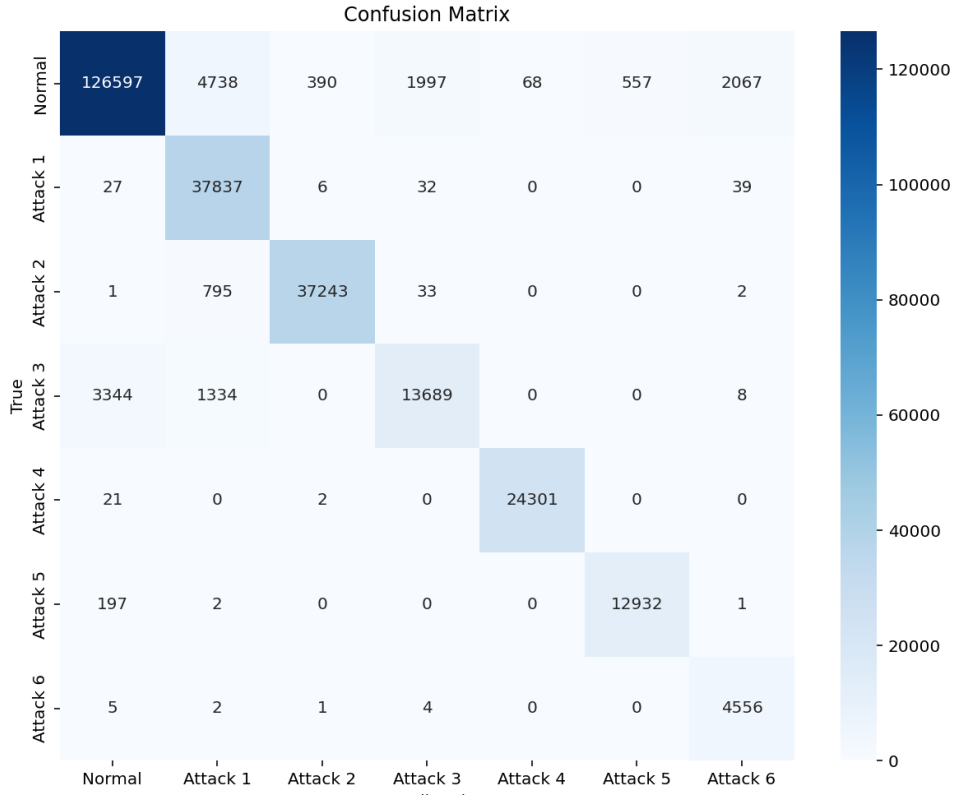


Рисунок 12 – Матрица ошибок для модели MLP

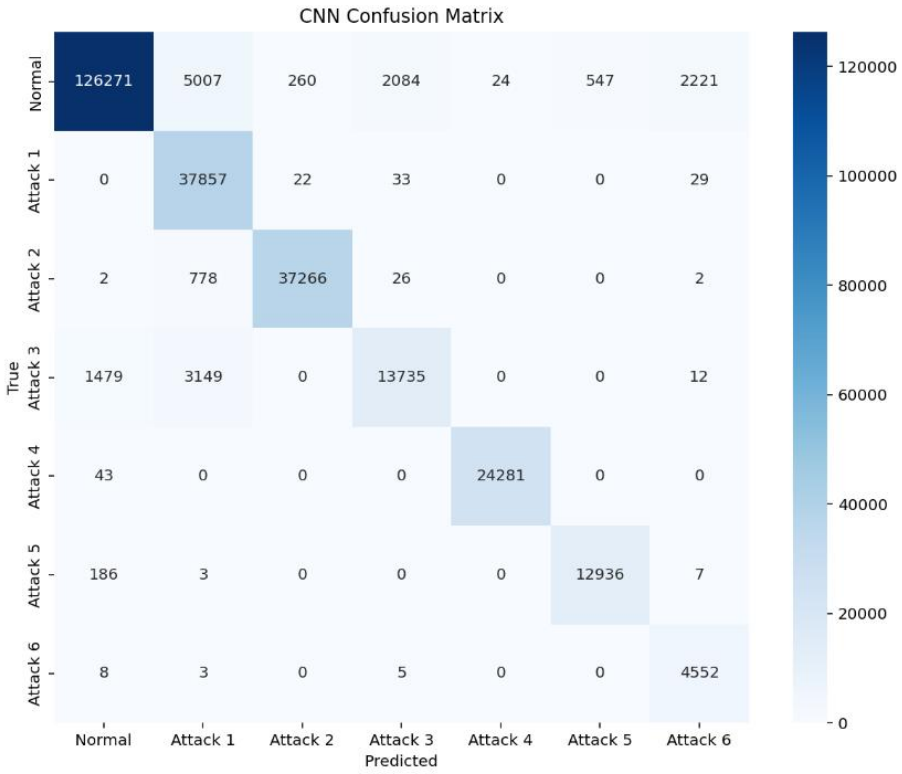


Рисунок 13 – Матрица ошибок для модели CNN

Также оценивалось время обучения. MLP показала высокую скорость сходимости и завершила обучение значительно быстрее, чем CNN. Последняя потребовала больше времени из-за свёрточных операций и работы с тензорами большего размера, но обеспечила лучшую устойчивость к переобучению и более качественную обработку сложных временных структур.

В контексте практического применения обе модели продемонстрировали сильные стороны. MLP может использоваться в условиях ограниченных вычислительных ресурсов, например, в edge-устройствах или встраиваемых системах мониторинга. CNN, в свою очередь, подходит для сценариев, где важна высокая точность при работе с потоковыми или сложно структурированными данными, включая атаки типа Mirai или фаззинг API.

В обеих архитектурах успешно сработали механизмы регуляризации и балансировки. SMOTE позволил выровнять представление всех классов и повысить полноту, особенно по редким категориям. Dropout и

BatchNormalization обеспечили устойчивость обучения и предотвратили переобучение, особенно в глубоких слоях MLP и сверточных блоках CNN.

Сравнительный анализ показал, что выбор модели зависит от конкретных условий: MLP обеспечивает высокую скорость и стабильность, тогда как CNN лучше справляется с задачами, где важны временные зависимости и сложные паттерны поведения. Это открывает возможности для гибридных подходов, комбинирующих сильные стороны обеих архитектур.

ЗАКЛЮЧЕНИЕ

В ходе выполнения диссертации была решена прикладная задача автоматического обнаружения сетевых атак с использованием нейросетевых архитектур. Основной акцент был сделан на сравнительном анализе двух подходов – многослойного перцептрона (MLP) и сверточной нейронной сети (1D-CNN). В качестве источника данных использован Kitsune Network Attack Dataset, содержащий разнообразные типы трафика и атак в условиях приближённых к реальным.

На начальном этапе была проведена обработка и анализ структуры данных. Определены числовые признаки, выполнена стандартизация и устранение выбросов. Особое внимание уделено решению проблемы дисбаланса классов, которая представляет серьёзную трудность при построении моделей в задачах информационной безопасности. Для устранения перекоса применялись техники случайной выборки и синтетического дополнения выборки.

В рамках бинарной классификации была построена модель MLP, способная отличать нормальный трафик от аномального. Проведён эксперимент по её обучению и валидации. Модель продемонстрировала устойчивые результаты: точность составила более 93%, при этом достигнуто высокое значение полноты по классу атак, что особенно важно в системах раннего реагирования и предотвращения угроз.

Следующим этапом стала многоклассовая классификация. Целью этого эксперимента являлось не только зафиксировать факт атаки, но и определить её конкретный тип. Реализованы и обучены две модели: MLP и 1D-CNN. Обе архитектуры продемонстрировали высокие значения точности, однако CNN оказалась более устойчивой при работе с миноритарными классами, лучше справляясь с временными зависимостями в данных. Анализ метрик, а также матриц ошибок показал, что 1D-CNN снижает количество ложных срабатываний по сравнению с MLP, особенно в отношении атак, имеющих выраженную структуру во времени.

Построенные модели прошли количественную и визуальную валидацию. Были рассчитаны метрики точности, полноты, F1-мера и AUC, построены ROC-кривые и тепловые карты ошибок. Все расчёты и визуализации позволили сделать объективные выводы о достоинствах и слабых местах каждой из моделей.

Полученные результаты подтвердили применимость нейросетевых методов для задач интеллектуального анализа сетевого трафика. Выявлено, что даже сравнительно простые архитектуры способны достигать высоких показателей при надлежащей подготовке данных и настройке модели. Использование сверточных нейронных сетей показало дополнительный потенциал для обнаружения атак, связанных с временными закономерностями.

Практическая значимость работы заключается в том, что построенные модели и описанный пайплайн обработки могут быть адаптированы под различные типы сетевых данных и использованы в составе систем мониторинга безопасности. Перспективами дальнейшего развития могут стать: внедрение рекуррентных компонентов (LSTM, GRU), применение attention-механизмов для повышения интерпретируемости, а также построение гибридных моделей с возможностью online-обучения на новых данных.

В результате проведённого исследования подтверждена эффективность нейросетевых решений для обнаружения сетевых атак, а также выявлены архитектурные особенности, влияющие на устойчивость и точность классификации в зависимости от сложности задачи.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Чио, К. Машинное обучение и безопасность / К. Чио, Д. Фримен. – Москва :ДМК, 2020. – 388 с. – ISBN 978-5-97060-713-8.

2. Машинное обучение / Б. Хенрик, Д. Уиттон, Т. Хасти, Р. Тибширани. – Санкт-Петербург : Питер, 2017. – 336 с. – ISBN 978-5-496-02989-6

3. Таненбаум Э. "Компьютерные сети" 5-е изд. (2016) / Э. Таненбаум, Д. Уэзеролл. – Санкт Питербург : Питер, 2016. – 323 с.

4. Классификация DDos-атак на основе нейросетевой модели -М. В. Тумбинской, Р. М.Мухаматханова

5. Атака сетей на уровне протоколов - Джеймс Форшоу 2021

6. Machine Learning in Python [Электронный ресурс] – URL: https://scikit-learn.org/stable/.

7. Spyder [Электронный ресурс] – URL: https://www.spyder-ide.org.

8. NumPy [Электронный ресурс] – URL: https://numpy.org/.

9. Pandas [Электронный ресурс] – URL: https://pandas.pydata.org/.

10. Kaggle [Электронный ресурс] – URL: https://www.kaggle.com/.

ПРИЛОЖЕНИЕ А

(Обязательное)

Детектирование сетевых атак при помощи нейросети

Листинг программы

ПРИЛОЖЕНИЕ А.1

Файл mlp.py

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import os
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix,
classification_report
from sklearn.utils.class_weight import compute_class_weight
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline as ImbPipeline
from collections import Counter
from sklearn.preprocessing import StandardScaler,
MinMaxScaler
from tensorflow.keras.callbacks import EarlyStopping,
ReduceLROnPlateau
from tensorflow.keras.metrics import Precision, Recall
import time
import json
import numpy as np
# Параметры
BATCH_SIZE = 1024
EPOCHS = 10
CHUNK_SIZE = 1500000  # уменьшил для работы с SMOTE в памяти
INPUT_DIM = 115     # число признаков

# Карта файлов и соответствующих классов
FILES_AND_CLASSES = {
```

```
        'ARP_MitM_dataset.csv': ('ARP_MitM_labels.csv', 1),
        'Active_Wiretap_dataset.csv':
('Active_Wiretap_labels.csv', 2),
        'Fuzzing_dataset.csv': ('Fuzzing_labels.csv', 3),
        'Mirai_dataset.csv': ('mirai_labels.csv', 4),
        'OS_Scan_dataset.csv': ('OS_Scan_labels.csv', 5),
        'SSL_Renegotiation_dataset.csv':
('SSL_Renegotiation_labels.csv', 6),
    }

    DATA_DIR = 'C:\\diplom'

    def load_and_balance_data(file_class_mapping,
max_samples_per_class=500000):
        """Загрузка и балансировка данных"""
        all_data = []
        all_labels = []

        for data_file, (label_file, attack_class) in
file_class_mapping.items():
            data_path = os.path.join(DATA_DIR, data_file)
            label_path = os.path.join(DATA_DIR, label_file)

            if not os.path.exists(data_path) or not
os.path.exists(label_path):
                print(f" Файлы не найдены: {data_path} или
{label_path}")
                continue

            try:
                # Чтение данных
                data_chunk = pd.read_csv(data_path, header=None,
nrows=CHUNK_SIZE)
                labels = pd.read_csv(label_path,
header=None).iloc[:, -1].values  # берем последний столбец
```

```
                # Обрезаем метки по размеру данных
                labels = labels[:len(data_chunk)]

                # Создаем метки: 1 → attack_class, 0 →
нормальный трафик
                y = np.where(labels == 1, attack_class, 0)
                X = data_chunk.iloc[:, :INPUT_DIM].values

                # Балансировка классов
                attack_idx = np.where(y == attack_class)[0]
                normal_idx = np.where(y == 0)[0]

                # Берем одинаковое количество примеров
                n_samples = min(len(attack_idx),
len(normal_idx), max_samples_per_class)

                if n_samples == 0:
                    print(f" Нет данных для класса
{attack_class} в файле {data_file}")
                    continue

                selected_idx = np.concatenate([
                    np.random.choice(attack_idx, n_samples,
replace=False),
                    np.random.choice(normal_idx, n_samples,
replace=False)
                ])

                all_data.append(X[selected_idx])
                all_labels.append(y[selected_idx])

            except Exception as e:
                print(f"Ошибка при обработке {data_file}:
{str(e)}")
                continue
```

```
        if not all_data:
            raise ValueError("Нет данных для обработки!")

        X = np.vstack(all_data)
        y = np.hstack(all_labels)

        # Перемешиваем данные
        indices = np.random.permutation(len(X))
        X, y = X[indices], y[indices]

        return X, y

    def create_cnn_model(input_shape, n_classes):
        model = keras.Sequential([
            layers.Input(shape=input_shape),  # (1, 115)

            # 1D Convolutional Layers
            layers.Conv1D(64, kernel_size=3, activation='relu',
padding='same'),
            layers.BatchNormalization(),
            layers.Dropout(0.3),

            layers.Conv1D(128, kernel_size=3, activation='relu',
padding='same'),
            layers.BatchNormalization(),
            layers.Dropout(0.4),

            layers.Flatten(),

            # Fully Connected Layers
            layers.Dense(128, activation='relu'),
            layers.BatchNormalization(),
            layers.Dropout(0.5),

            layers.Dense(n_classes, activation='softmax')
        ])
```

```
        model.compile(
            optimizer='adam',
            loss='categorical_crossentropy',
            metrics=['accuracy',
                     Precision(name='precision'),  # Используем
класс Precision
                     Recall(name='recall')]       # Используем
класс Recall
        )
        return model



    def plot_history(history):
        """Визуализация истории обучения"""
        plt.figure(figsize=(14, 5))

        # Accuracy
        plt.subplot(1, 2, 1)
        plt.plot(history.history['accuracy'], label='Train
Accuracy')
        plt.plot(history.history['val_accuracy'],
label='Validation Accuracy')
        plt.title('Model Accuracy')
        plt.ylabel('Accuracy')
        plt.xlabel('Epoch')
        plt.legend()

        # Loss
        plt.subplot(1, 2, 2)
        plt.plot(history.history['loss'], label='Train Loss')
        plt.plot(history.history['val_loss'], label='Validation
Loss')
        plt.title('Model Loss')
        plt.ylabel('Loss')
```

```
        plt.xlabel('Epoch')
        plt.legend()

        plt.show()

    def evaluate_cnn_model(model, X_test, y_test):
        """Оценка CNN модели и сохранение метрик"""
        X_test_cnn = X_test.reshape(-1, 1, INPUT_DIM)
        y_test_cat = keras.utils.to_categorical(y_test,
num_classes=7)

        # Оценка модели
        test_metrics = model.evaluate(X_test_cnn, y_test_cat,
verbose=0)
        test_loss, test_acc, test_precision, test_recall =
test_metrics

        # Получение предсказаний
        y_pred = model.predict(X_test_cnn, verbose=0)
        y_pred_classes = np.argmax(y_pred, axis=1)

        # Confusion Matrix
        cm = confusion_matrix(y_test, y_pred_classes)

        # Classification Report
        cl_report = classification_report(y_test,
y_pred_classes, digits=4, output_dict=True)

        # Сохранение результатов
        np.save('cnn_confusion_matrix.npy', cm)
        with open('cnn_classification_report.json', 'w') as f:
            json.dump(cl_report, f)

        # Визуализация
        plt.figure(figsize=(10, 8))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
```

```
                    xticklabels=['Normal'] + [f'Attack {i}' for
i in range(1, 7)],
                    yticklabels=['Normal'] + [f'Attack {i}' for
i in range(1, 7)])
        plt.title('CNN Confusion Matrix')
        plt.xlabel('Predicted')
        plt.ylabel('True')
        plt.savefig('cnn_confusion_matrix.png')
        plt.close()

        print("\n CNN Test Metrics:")
        print(f"Accuracy: {test_acc:.4f}")
        print(f"Precision: {test_precision:.4f}")
        print(f"Recall: {test_recall:.4f}")
        print(f"Loss: {test_loss:.4f}")

        return {
            'test_metrics': {
                'loss': float(test_loss),
                'accuracy': float(test_acc),
                'precision': float(test_precision),
                'recall': float(test_recall)
            },
            'confusion_matrix': cm.tolist(),
            'classification_report': cl_report
        }

    def main():
        # 1. Загрузка и подготовка данных
        print("Loading and balancing data...")
        X, y = load_and_balance_data(FILES_AND_CLASSES)

        # 2. Разделение данных (до SMOTE!)
        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=0.2, random_state=42, stratify=y
        )
```

```
        # 3. Нормализация данных
        print(" Normalizing data...")
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)
        X_train_cnn = np.expand_dims(X_train, axis=1)  #
(samples, 1, 115)
        X_test_cnn = np.expand_dims(X_test, axis=1)    #
(samples, 1, 115)

        # 4. Применение SMOTE только к обучающим данным
        print("Applying SMOTE to training data...")
        # Перед применением SMOTE преобразуем данные в 2D
        X_train_2d = X_train.reshape(X_train.shape[0], -1)  #
(n_samples, n_features)

        # Применяем SMOTE
        smote = SMOTE(random_state=42)
        X_resampled_2d, y_resampled =
smote.fit_resample(X_train_2d, y_train)

        # Преобразуем обратно в 3D для CNN
        X_resampled = X_resampled_2d.reshape(-1, 1, INPUT_DIM)
        # Проверка баланса классов
        print("\n Class distribution:")
        print("Before SMOTE:", Counter(y_train))
        print("After SMOTE:", Counter(y_resampled))

        # 5. Преобразование меток в one-hot
        y_resampled_cat =
keras.utils.to_categorical(y_resampled, num_classes=7)
        y_test_cat = keras.utils.to_categorical(y_test,
num_classes=7)
        print(f"X_resampled shape: {X_resampled.shape}")  #
Должно быть (N, 1, 115)
```

```
        print(f"y_resampled_cat shape: {y_resampled_cat.shape}")
# Должно быть (N, 7)
        print(f"X_test_cnn shape: {X_test_cnn.shape}")  # Должно
быть (M, 1, 115)
        print(f"y_test_cat shape: {y_test_cat.shape}")  # Должно
быть (M, 7)
        # 6. Расчет весов классов
        class_weights = compute_class_weight(
            class_weight='balanced',
            classes=np.unique(y_resampled),
            y=y_resampled
        )
        class_weights_dict = dict(enumerate(class_weights))

        # 7. Создание и обучение модели
        print("\n Creating model...")
        model = create_cnn_model((1, INPUT_DIM), 7)
        model.summary()

        # Callbacks
        callbacks = [
            EarlyStopping(patience=5, restore_best_weights=True,
monitor='val_loss'),
            ReduceLROnPlateau(factor=0.1, patience=3, min_lr=1e-
6)
        ]

        print("\n Training CNN model...")
        start_time = time.time()

        history = model.fit(
            X_resampled,
            y_resampled_cat,
            validation_data=(X_test_cnn, y_test_cat),
            batch_size=BATCH_SIZE,
            epochs=EPOCHS,
```

```
            callbacks=callbacks,
            verbose=1
        )

        training_time = time.time() - start_time
        print(f"\n CNN Training time: {training_time:.2f}
seconds")

        # Сохранение истории обучения
        history_data = {
            'accuracy': history.history['accuracy'],
            'val_accuracy': history.history['val_accuracy'],
            'loss': history.history['loss'],
            'val_loss': history.history['val_loss'],
            'precision': history.history.get('precision', []),
            'val_precision':
history.history.get('val_precision', []),
            'recall': history.history.get('recall', []),
            'val_recall': history.history.get('val_recall', [])
        }

        with open('cnn_history.json', 'w') as f:
            json.dump(history_data, f)

        # Сохранение времени обучения
        with open('cnn_training_time.json', 'w') as f:
            json.dump({'training_time': training_time}, f)

        # Оценка модели
        evaluation_results = evaluate_cnn_model(model, X_test,
y_test)

        # Сохранение метрик
        with open('cnn_test_metrics.json', 'w') as f:
            json.dump(evaluation_results['test_metrics'], f)
```

```
        plot_history(history)

    if __name__ == "__main__":
        main()
```

ПРИЛОЖЕНИЕ А.2

Файл cnn.py

```
    import numpy as np
    import pandas as pd
    import tensorflow as tf
    from tensorflow import keras
    from tensorflow.keras import layers
    import os
    import matplotlib.pyplot as plt
    import seaborn as sns
    from sklearn.metrics import confusion_matrix,
classification_report
    from sklearn.utils.class_weight import compute_class_weight
    from sklearn.model_selection import train_test_split
    from imblearn.over_sampling import SMOTE
    from imblearn.pipeline import Pipeline as ImbPipeline
    from collections import Counter
    from sklearn.preprocessing import StandardScaler,
MinMaxScaler
    from tensorflow.keras.callbacks import EarlyStopping,
ReduceLROnPlateau
    import json
    import numpy as np
    import time
    # Параметры
    BATCH_SIZE = 1024
    EPOCHS = 10
    CHUNK_SIZE = 1500000  # уменьшил для работы с SMOTE в памяти
    INPUT_DIM = 115     # число признаков

    # Карта файлов и соответствующих классов
```

```
    FILES_AND_CLASSES = {
        'ARP_MitM_dataset.csv': ('ARP_MitM_labels.csv', 1),
        'Active_Wiretap_dataset.csv':
('Active_Wiretap_labels.csv', 2),
        'Fuzzing_dataset.csv': ('Fuzzing_labels.csv', 3),
        'Mirai_dataset.csv': ('mirai_labels.csv', 4),
        'OS_Scan_dataset.csv': ('OS_Scan_labels.csv', 5),
        'SSL_Renegotiation_dataset.csv':
('SSL_Renegotiation_labels.csv', 6),
    }

    DATA_DIR = 'C:\\diplom'

    def load_and_balance_data(file_class_mapping,
max_samples_per_class=1500000):
        """Загрузка и балансировка данных"""
        all_data = []
        all_labels = []

        for data_file, (label_file, attack_class) in
file_class_mapping.items():
            data_path = os.path.join(DATA_DIR, data_file)
            label_path = os.path.join(DATA_DIR, label_file)

            if not os.path.exists(data_path) or not
os.path.exists(label_path):
                print(f" Файлы не найдены: {data_path} или
{label_path}")
                continue

            try:
                # Чтение данных
                data_chunk = pd.read_csv(data_path, header=None,
nrows=CHUNK_SIZE)
                labels = pd.read_csv(label_path,
header=None).iloc[:, -1].values  # берем последний столбец
```

```
                # Обрезаем метки по размеру данных
                labels = labels[:len(data_chunk)]

                # Создаем метки: 1 → attack_class, 0 →
нормальный трафик
                y = np.where(labels == 1, attack_class, 0)
                X = data_chunk.iloc[:, :INPUT_DIM].values

                # Балансировка классов
                attack_idx = np.where(y == attack_class)[0]
                normal_idx = np.where(y == 0)[0]

                # Берем одинаковое количество примеров
                n_samples = min(len(attack_idx),
len(normal_idx), max_samples_per_class)

                if n_samples == 0:
                    print(f" Нет данных для класса
{attack_class} в файле {data_file}")
                    continue

                selected_idx = np.concatenate([
                    np.random.choice(attack_idx, n_samples,
replace=False),
                    np.random.choice(normal_idx, n_samples,
replace=False)
                ])

                all_data.append(X[selected_idx])
                all_labels.append(y[selected_idx])

            except Exception as e:
                print(f" Ошибка при обработке {data_file}:
{str(e)}")
                continue
```

```
    if not all_data:
        raise ValueError("Нет данных для обработки!")

    X = np.vstack(all_data)
    y = np.hstack(all_labels)

    # Перемешиваем данные
    indices = np.random.permutation(len(X))
    X, y = X[indices], y[indices]

    return X, y

def create_model(input_dim, n_classes):
    """Создание модели нейронной сети"""
    model = keras.Sequential([
        layers.Input(shape=(input_dim,)),
        layers.Dense(256, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.5),

        layers.Dense(128, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.4),

        layers.Dense(64, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.3),

        layers.Dense(32, activation='relu'),
        layers.BatchNormalization(),

        layers.Dense(n_classes, activation='softmax')
    ])

    model.compile(
```

```
optimizer=keras.optimizers.Adam(learning_rate=0.001),
            loss='categorical_crossentropy',
            metrics=['accuracy',
                     keras.metrics.Precision(name='precision'),
                     keras.metrics.Recall(name='recall')]
        )
        return model



    def plot_history(history):
        """Визуализация истории обучения"""
        plt.figure(figsize=(14, 5))

        # Accuracy
        plt.subplot(1, 2, 1)
        plt.plot(history.history['accuracy'], label='Train
Accuracy')
        plt.plot(history.history['val_accuracy'],
label='Validation Accuracy')
        plt.title('Model Accuracy')
        plt.ylabel('Accuracy')
        plt.xlabel('Epoch')
        plt.legend()

        # Loss
        plt.subplot(1, 2, 2)
        plt.plot(history.history['loss'], label='Train Loss')
        plt.plot(history.history['val_loss'], label='Validation
Loss')
        plt.title('Model Loss')
        plt.ylabel('Loss')
        plt.xlabel('Epoch')
        plt.legend()
```

```
        plt.show()

    def evaluate_model(model, X_test, y_test):
        """Оценка модели и вывод метрик"""
        y_test_cat = keras.utils.to_categorical(y_test,
num_classes=7)

        # Оценка модели
        test_metrics = model.evaluate(X_test, y_test_cat,
verbose=0)

        # Для моделей с несколькими метриками
        if len(test_metrics) == 4:  # loss, accuracy, precision,
recall
            test_loss, test_acc, test_precision, test_recall =
test_metrics
        else:  # Если метрик меньше, задаем значения по
умолчанию
            test_loss, test_acc = test_metrics[0],
test_metrics[1]
            test_precision, test_recall = 0.0, 0.0

        # Получаем предсказания для дополнительного анализа
        y_pred = model.predict(X_test, verbose=0)
        y_pred_classes = np.argmax(y_pred, axis=1)

        # Confusion Matrix
        cm = confusion_matrix(y_test, y_pred_classes)

        # Classification Report
        cl_report = classification_report(y_test,
y_pred_classes, digits=4, output_dict=True)

        # Сохраняем confusion matrix
        np.save('mlp_confusion_matrix.npy', cm)
```

```
        # Сохраняем classification report
        with open('mlp_classification_report.json', 'w') as f:
            json.dump(cl_report, f)

        # Вывод в консоль
        print(f"\nFinal Test Metrics:")
        print(f"Accuracy: {test_acc:.4f}")
        print(f"Precision: {test_precision:.4f}")
        print(f"Recall: {test_recall:.4f}")
        print(f"Loss: {test_loss:.4f}")

        # Визуализация confusion matrix
        plt.figure(figsize=(10, 8))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                    xticklabels=['Normal'] + [f'Attack {i}' for
i in range(1, 7)],
                    yticklabels=['Normal'] + [f'Attack {i}' for
i in range(1, 7)])
        plt.title('Confusion Matrix')
        plt.xlabel('Predicted')
        plt.ylabel('True')
        plt.savefig('mlp_confusion_matrix.png')
        plt.close()  # Закрываем график, чтобы не показывался в
консоли

        print("\nClassification Report:")
        print(classification_report(y_test, y_pred_classes,
digits=4))

        return {
            'test_metrics': {
                'loss': float(test_loss),
                'accuracy': float(test_acc),
                'precision': float(test_precision),
                'recall': float(test_recall)
            },
```

```
            'confusion_matrix': cm.tolist(),  # Преобразуем
numpy array в list для JSON
            'classification_report': cl_report
        }

    def main():
        # 1. Загрузка и подготовка данных
        print("Loading and balancing data...")
        X, y = load_and_balance_data(FILES_AND_CLASSES)

        # 2. Разделение данных (до SMOTE!)
        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=0.2, random_state=42, stratify=y
        )

        # 3. Нормализация данных
        print("Normalizing data...")
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)

        # 4. Применение SMOTE только к обучающим данным
        print(" Applying SMOTE to training data...")
        smote = SMOTE(random_state=42)
        X_resampled, y_resampled = smote.fit_resample(X_train,
y_train)

        # Проверка баланса классов
        print("\nClass distribution:")
        print("Before SMOTE:", Counter(y_train))
        print("After SMOTE:", Counter(y_resampled))

        # 5. Преобразование меток в one-hot
        y_resampled_cat =
keras.utils.to_categorical(y_resampled, num_classes=7)
```

```
        y_test_cat = keras.utils.to_categorical(y_test,
num_classes=7)

        # 6. Расчет весов классов
        class_weights = compute_class_weight(
            class_weight='balanced',
            classes=np.unique(y_resampled),
            y=y_resampled
        )
        class_weights_dict = dict(enumerate(class_weights))

        # 7. Создание и обучение модели
        print("\nCreating model...")
        model = create_model(INPUT_DIM, 7)
        model.summary()

        # Callbacks
        callbacks = [
            EarlyStopping(patience=5, restore_best_weights=True,
monitor='val_loss'),
            ReduceLROnPlateau(factor=0.1, patience=3, min_lr=1e-
6)
        ]

        print("\nTraining model...")
        start_time = time.time()  # Добавляем замер времени
        history = model.fit(
            X_resampled,
            y_resampled_cat,
            validation_data=(X_test, y_test_cat),
            batch_size=BATCH_SIZE,
            epochs=EPOCHS,
            callbacks=callbacks,
            class_weight=class_weights_dict,
            verbose=1
        )
```

```
        training_time = time.time() - start_time
        print(f"\nTraining time: {training_time:.2f} seconds")

        # 8. Оценка результатов
        plot_history(history)

        # Получаем результаты оценки
        evaluation_results = evaluate_model(model, X_test,
y_test)

        # Сохраняем историю обучения
        history_data = {
            'accuracy': history.history['accuracy'],
            'val_accuracy': history.history['val_accuracy'],
            'loss': history.history['loss'],
            'val_loss': history.history['val_loss'],
            'precision': history.history.get('precision', []),
# Используем get для безопасного доступа
            'val_precision':
history.history.get('val_precision', []),
            'recall': history.history.get('recall', []),
            'val_recall': history.history.get('val_recall', [])
        }

        with open('mlp_history.json', 'w') as f:
            json.dump(history_data, f)

        # Сохраняем время обучения
        with open('mlp_training_time.json', 'w') as f:
            json.dump({'training_time': training_time}, f)

        # Сохраняем метрики из evaluation_results
        with open('mlp_test_metrics.json', 'w') as f:
            json.dump(evaluation_results['test_metrics'], f)
    if __name__ == "__main__":
        main()
```

ПРИЛОЖЕНИЕ В

(Обязательное)

Детектирование сетевых атак при помощи нейросети

Презентация

тектирование сетевых а
при помощи нейросети

аботы

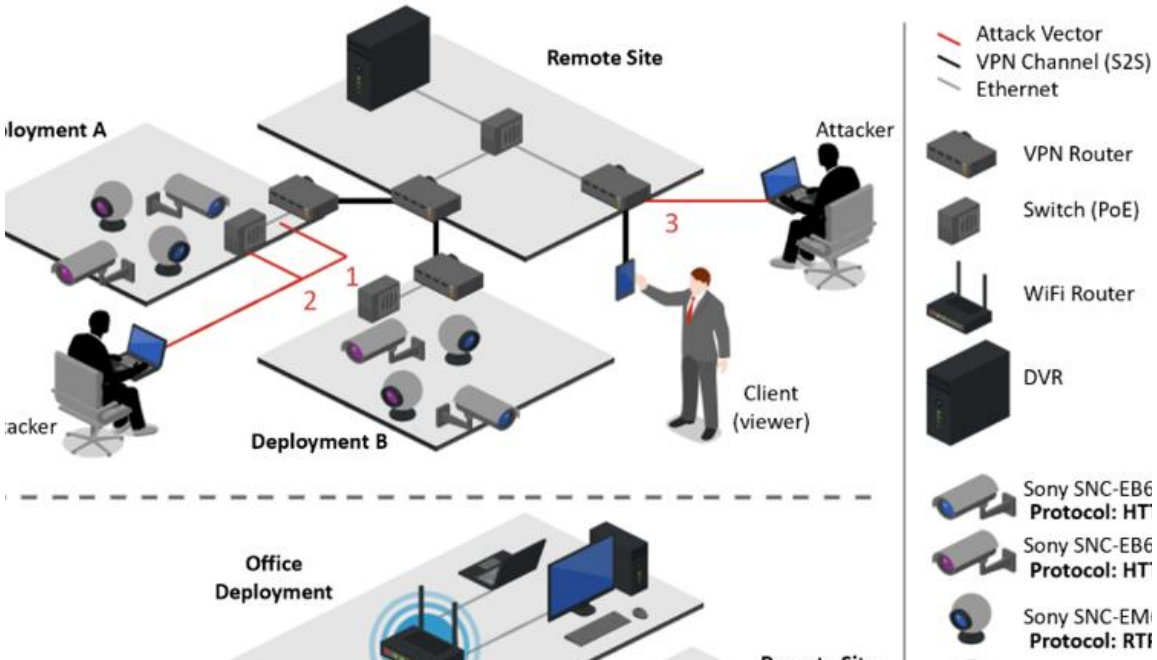
модели глубокого обучения для автоматич
ения и классификации сетевых атак на осн
сетевого трафика

ьзуемый датасет

	Краткое описание
	Вредоносное ПО, заражающее IoT-устройства. Используется для DDoS-атак, сканирования и перебора паролей.
ар	Атака типа "человек посередине" (Man-in-the-Middle), при которой злоумышленник перехватывает и изменяет сетевые пакеты между двумя узлами.
	Локальная MitM-атака через подмену ARP-таблицы, позволяющая перехватывать трафик в сети.
	Отправка случайных, некорректных или специально сформированных данных для поиска ошибок в

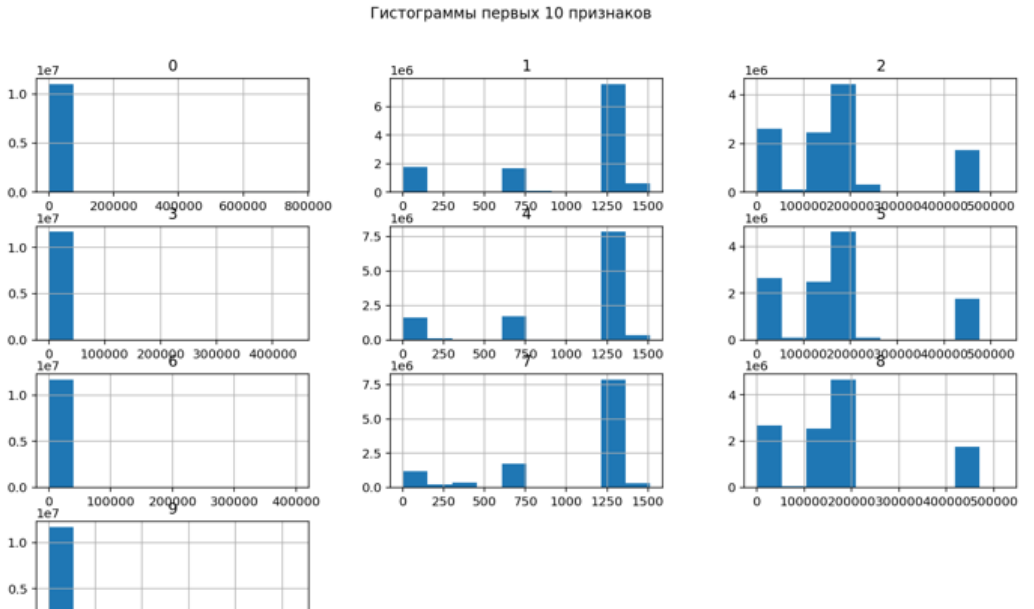
- 9 миллионо
- 115 числов

Структура сети

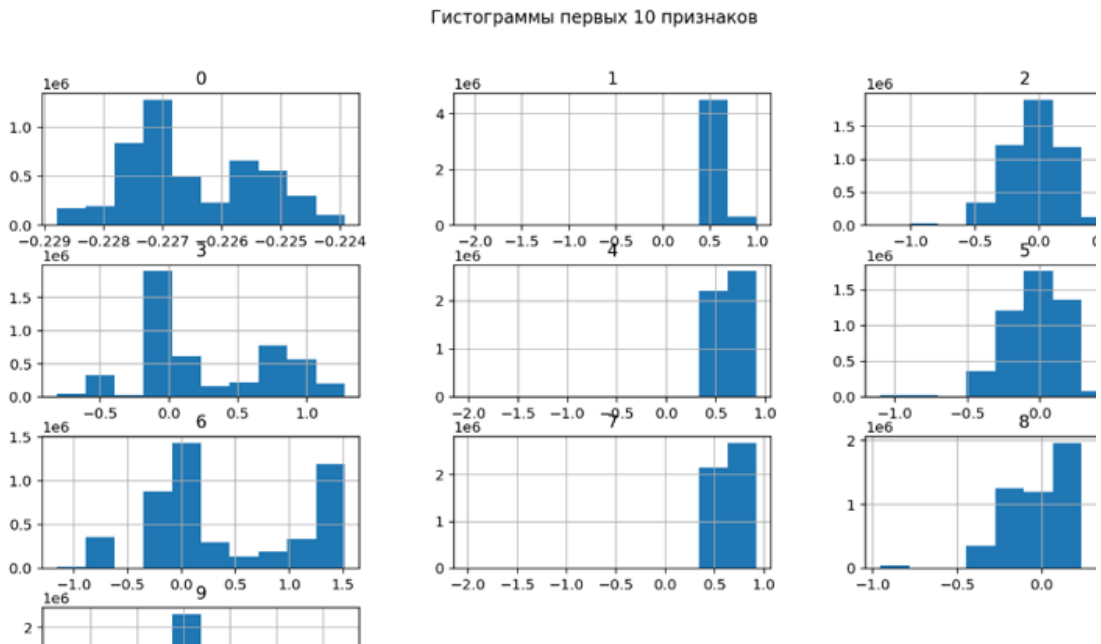


Обработка и подготовка Д

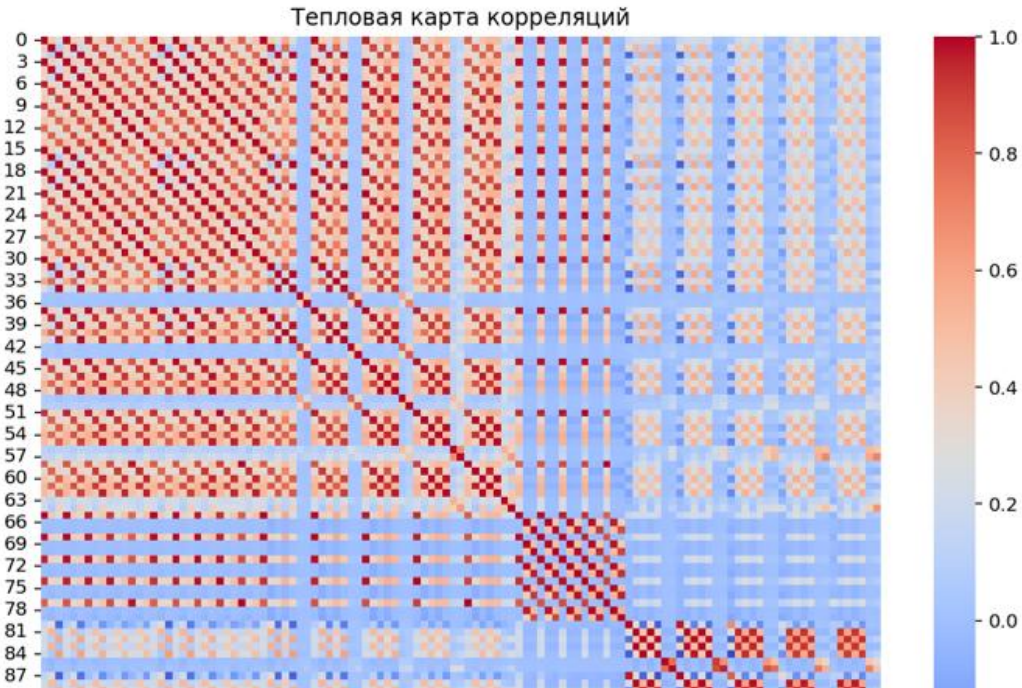
ртизация данных



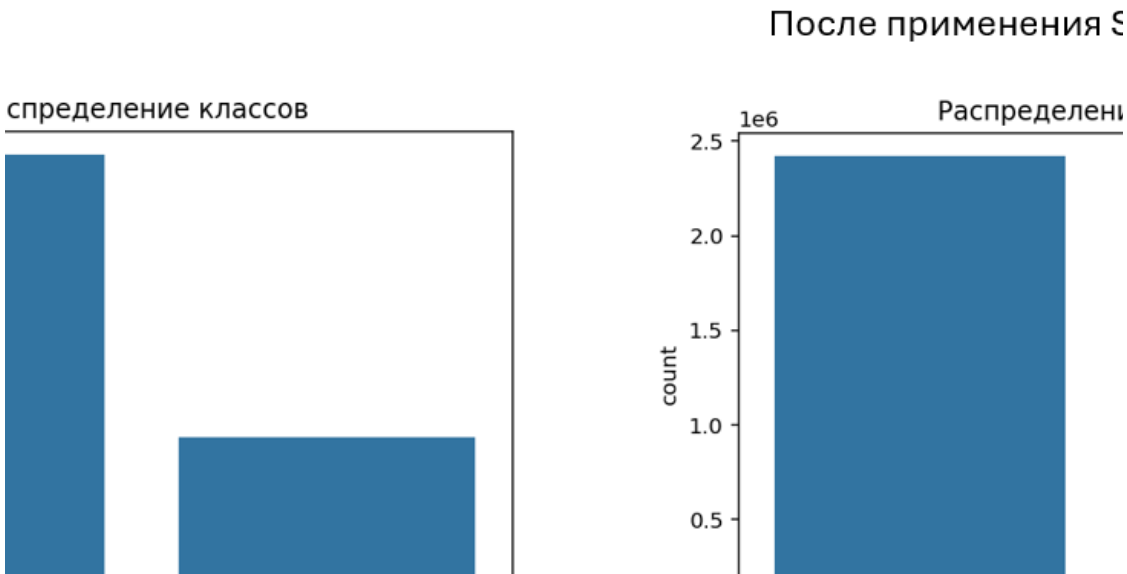
ртизация данных



я карта


Тепловая карта корреляций
0
3
6
9
12
15
18
21
24
27
30
33
36
39
42
45
48
51
54
57
60
63
66
69
72
75
78
81
84
87
1.0
0.8
0.6
0.4
0.2
0.0

деление классов



нные модели

Модель 1D-CNN

- Вход: данные в фор
- Сверточные слои:
  - Conv1D (64 фильтр
  - Conv1D (128 филь
- Регуляризация: BatchNormalization (0.3 и 0.4)
- Скрытый полносвя 128 нейронов

лой: 115 признаков

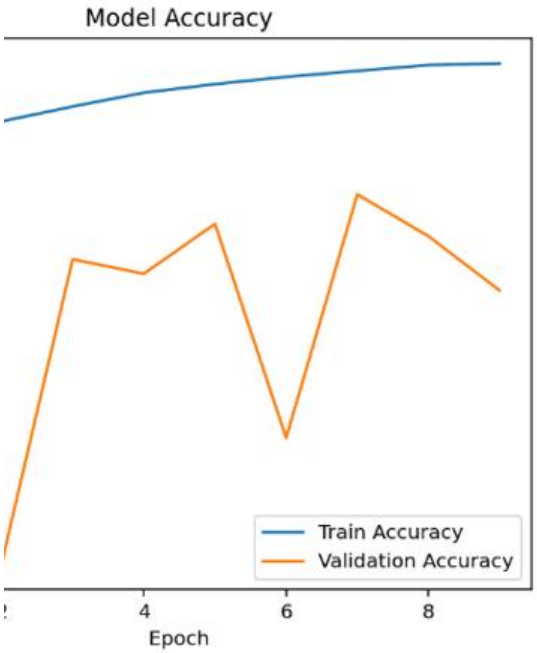
слои: 256 → 128 → 64 → а

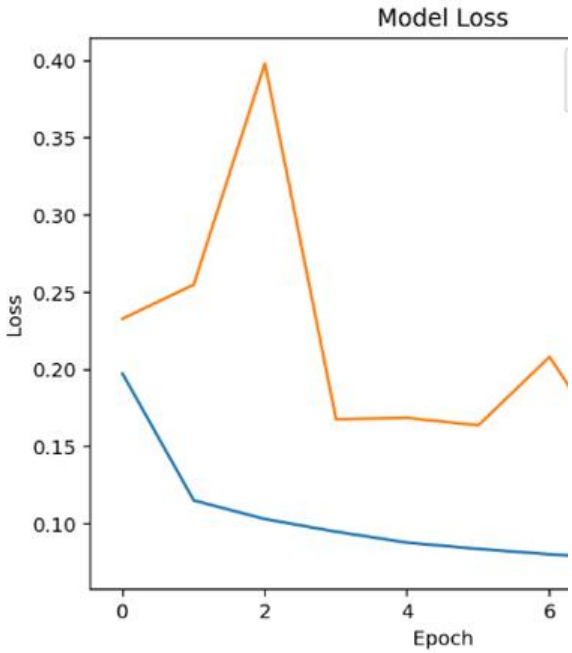
I: ReLU

ация: Dropout (от 0.5 atchNormalization

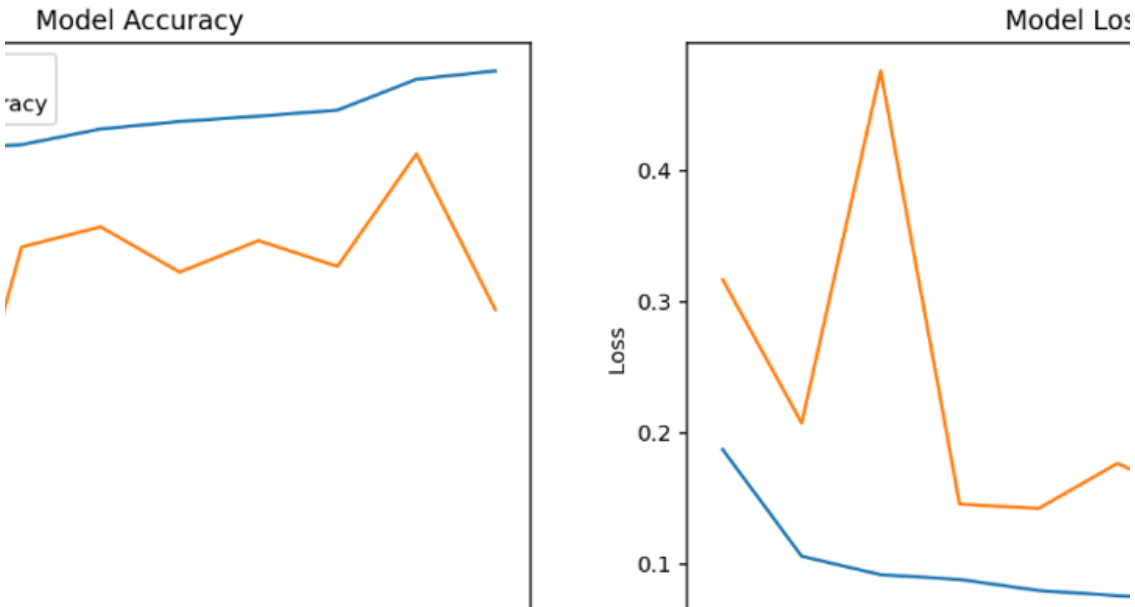
слой: softmax (для совой кации)

accuracy и loss по эпохам для модел





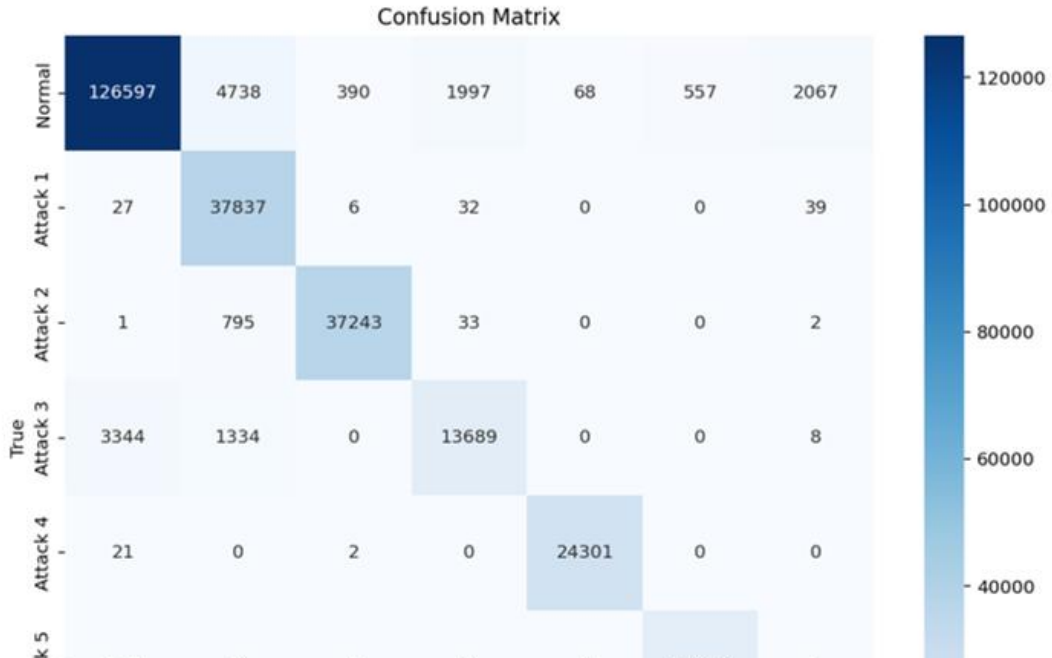
accuracy и loss по эпохам для моде



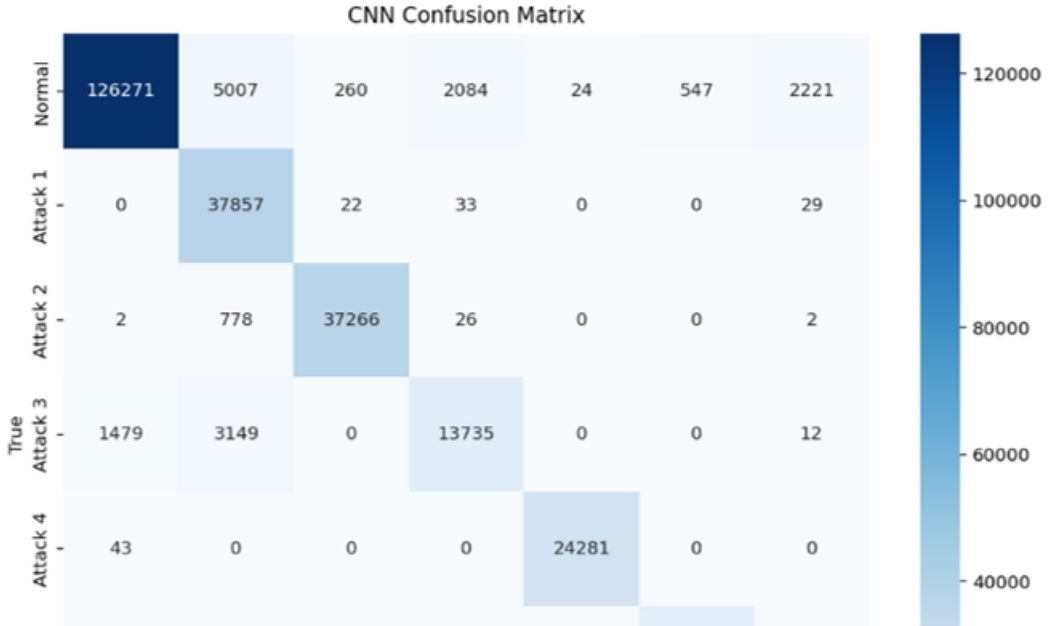
ие результатов моделей MLP и CNN

Accuracy	Precision	Recall	F1-sc
94.26%	94.65%	94.16%	94.28
94.16%	94.45%	94.08%	94.26

ца ошибок для модели MI



ца ошибок для модели CN



ение

ы и обучены две модели нейросетей, способ
о обнаруживать атаки

вляется лучше с редкими и сложными атака
ри наличии временных зависимостей

ечивает сопоставимую точность при мены
льных затратах

комплексная оценка качества: метрики, вк

асибо за вниман