

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

НЕЙРОСЕТЕВАЯ ОБРАБОТКА МНОГОМЕРНОГО

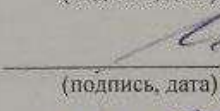
СИГНАЛА

Студент


(подпись, дата)

Федяшов Марк Сергеевич
(ФИО полностью)

Руководитель


(подпись, дата)

Митрохин М.А.
(фамилия, инициалы)

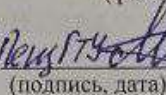
Нормоконтролёр


(подпись, дата)

Бычков А.С.
(фамилия, инициалы)

Рецензент

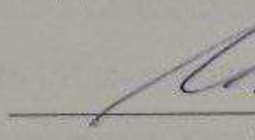
зав. каф. Программирования ИИ
(должность, место работы)


(подпись, дата)

Маритский А.Н.
(фамилия, инициалы)

Работа допущена к защите (протокол зас. кафедры от 27.05.24 № 11)

Заведующий кафедрой


(подпись)

М.А. Митрохин
(фамилия, инициалы)

Работа защищена с отметкой отлично (протокол заседания ГЭК от 6.06.24 № 3)

Секретарь ГЭК


(подпись)

Е.И. Гурин
(фамилия, инициалы)

Пенза 2024

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

Факультет

Вычислительной техники

Кафедра

Вычислительная техника

«Утверждаю»

Заведующий кафедрой

М.А. Митрохин

«15» 09 2022 г.

ЗАДАНИЕ НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ

1. Студент Федяшов Марк Сергеевич, гр. 22ВВиМ1 факультета вычислительной техники, направления 09.04.01.

2. Тема работы: «Нейросетевая обработка многомерного сигнала»

Тема утверждена приказом ПГУ № 765/о от «13» сентября 2022 г.

3. Руководитель магистерской диссертации д.т.н., профессор кафедры Митрохин М.А.

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

Назначение разработки: прогнозирования фазового угла источника колебаний на основании цифрового сигнала с нескольких сейсмографов.

Моделирование и обучение нейросетевого алгоритма для решения задачи регрессии в области цифровой обработки сигналов.

5. Перечень подлежащих разработке вопросов

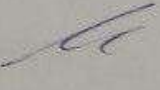
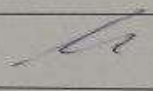
1. Исследовать существующие архитектуры нейросетевых алгоритмов и современные инструменты для работы с нейронными сетями, отобрать наиболее подходящие.

2. Провести анализ исходных данных, на их основе сформировать обучающий набор.

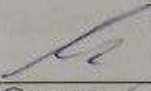
3. Обучить подобранные модели нейронных сетей и провести сравнительный анализ результатов обучения.

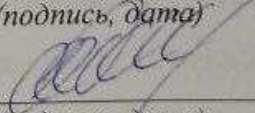
4. Спроектировать программную систему для функционирования нейронной сети.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1.	Исследование существующих архитектур нейросетевых алгоритмов и современных инструментов для работы с нейронными сетями	15%	13.09.2022 – 18.12.2022	
2.	Проведение анализа исходных данных и формирование обучающего набора	35%	19.12.2022 – 11.06.2023	
3.	Обучение подобранных моделей нейронных сетей, проведение сравнительного анализа результатов обучения	20%	12.06.2023 – 15.10.2023	
4.	Проектирование программной системы для функционирования нейронной сети	20%	16.10.2023 – 17.03.2024	
5.	Оформление пояснительной записки	10%	18.03.2024 – 19.05.2024	

Дата выдачи задания «13» сентября 2022 г.

Руководитель магистерской диссертации  Митрохин М.А.
(подпись, дата)

Задание к исполнению принял студент  Федяшов М.С.
(подпись, дата)

Работу к защите допустить
Декан факультета  03.06.2024 Фионова Л.Р.
(подпись, дата)

РЕФЕРАТ

Магистерская работа содержит 78 листов, 36 рисунков, 3 таблицы, 10 источников, 2 приложения

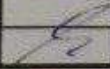
МАШИННОЕ ОБУЧЕНИЕ, НЕЙРОННЫЕ СЕТИ, СВЁРТОЧНЫЕ НЕЙРОННЫЕ СЕТИ, РЕКУРРЕНТНЫЕ НЕЙРОННЫЕ СЕТИ, PYTHON, PYTORCH

Цель работы – исследование алгоритмов машинного обучения применительно к цифровой обработке сигналов, выявление наиболее подходящей модели для решения задачи пеленгации. Объектом исследования является комплекс алгоритмов машинного обучения, основанных на нейронных сетях.

В результате проведенной работы были исследованы различные нейросетевые алгоритмы, выявлена наиболее успешная модель, обучен конкретный экземпляр сети, а также спроектирована архитектура ПО для мониторинга и контроля качества нейросетевого алгоритма.

Для исследования и реализации было применено большое количество современных технологий для разработки нейронных сетей, такие как язык программирования Python, а также ряд библиотек: PyTorch, numpy, matplotlib.

ПГУ 1.090401.25.001 ПЗ

					ПГУ 1.090401.25.001 ПЗ			
Изм.	Лист	№ докум.	Подпись	Дата	Нейросетевая обработка многомерного сигнала Пояснительная записка	Лит.	Лист	Листов
Разраб.		Федяшов М.С.		03.06.24				
Провер.		Митрохин М.А.		03.06.24			2	78
Н. Контр		Бычков А.С.		03.06.24				
Утверд.		Митрохин М.А.		03.06.24		ФВТ, каф. ВТ гр. 228ВuM1		

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ	4
ВВЕДЕНИЕ	5
1 Анализ современных методологий и решений в областях машинного обучения и нейронных сетей	7
1.1 Анализ архитектур нейронных сетей.....	7
1.2 Анализ языков программирования для реализации и обучения нейронных сетей.....	12
1.3 Анализ библиотек машинного обучения.....	13
1.4 Анализ языков программирования и систем управления базами данных для реализации прикладного API.....	14
2 Подготовка данных и обучение нейросетевых алгоритмов	16
2.1 Анализ и предобработка данных.....	16
2.2 Подготовка данных и обучение моделей.....	21
2.3 Обучение CNN	29
2.3 Анализ результатов обучения CNN	32
2.4 Обучение RNN	35
2.5 Анализ результатов обучения сетей	43
3 Проектирование программной системы	44
3.1 Требования, предъявляемые к программной системе.....	45
3.2 Проектирование	45
ЗАКЛЮЧЕНИЕ	53
ПРИЛОЖЕНИЕ А	55
ПРИЛОЖЕНИЕ Б.....	72

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящей работе применяют следующие сокращения и обозначения

ОС – операционная система

СУБД – система управления базами данных

ЦОС – цифровая обработка сигналов

ЦПУ – центральное процессорное устройство

ЭВМ – электронная вычислительная машина

API (application programming interface) – прикладной программный интерфейс

ACID (atomicity, concurrency, isolation, durability) – атомарность, параллелизм, изолированность и устойчивость; основные требования, предъявляемые к СУБД

CNN (convolutional neural network) – сверточная нейронная сеть

GPS (global positioning system) – система глобального позиционирования

FNN (fully-connected neural network) – полносвязная нейронная сеть

GRU (gated recurrent unit) – управляемый рекуррентный блок

HTTP (hypertext transfer protocol) – протокол передачи гипертекста

JSON (JavaScript object notation) – схема передачи объектов для JavaScript

JWT (java web token) – маркер доступа

LSTM (long short term memory) – сеть с долгосрочной и краткосрочной памятью

MVCC (multiversion concurrency control) – многоверсионный контроль за параллельным доступом

RBA (role based authorization) – авторизация, основанная на ролях

RNN (recurrent neural network) – рекуррентная нейронная сеть

SQL (structured query language) – язык структурированных запросов

SLA (service level agreement) – соглашение об уровне услуг

ВВЕДЕНИЕ

Обработка сигналов является одной из наиболее сложных, но вместе с тем перспективным и востребованным направлением для исследований. Сигналы – распространяющиеся в среде колебания – встречаются во многих сферах, начиная от любой электрической цепи и заканчивая медицинскими анализами. Поскольку среда распространения колебаний может быть разнообразной, возрастает и количество способов обнаружения, изучения и принятия решений на их основе.

Изначально алгоритмы, работающие с сигналами, являлись совокупностью математических методов и эвристического анализа, основанного на наблюдении за параметрами сигнала, такими как амплитудно-частотная характеристика, зашумленность и т.д. За последние 20 лет методы анализа и обработки цифровых сигналов претерпели значительные изменения: в настоящее время применяются методы машинного обучения и нейронные сети [1].

Нейронная сеть – это математическая модель, состоящая из множества искусственных нейронов. Данная модель появилась как попытка смоделировать процессы, протекающих в живом мозге. В результате совокупности факторов, таких как вычислительные мощности, объем собранных данных и достижений в области алгоритмов, модель стала доминирующей для решения задач в области искусственного интеллекта и других наукоёмких сферах: применение нейронных сетей различной архитектуры позволяет решать задачи регрессии и классификации, генерировать новые данные, основанные на артефактах.

Отдельным случаем является анализ сигналов сразу из нескольких источников: такие задачи зачастую носят пространственный характер. В данной работе рассмотрена одна из таких задач: имеется набор сигналов, собранный с четырех сейсмографов, по которому необходимо найти координаты объекта, вызывающего возмущения на поверхности земли.

Исследования в области применения нейросетевых алгоритмов для решения задач, связанных с цифровой обработкой сигналов, могут привести к созданию наиболее передовых решений в различных областях. Решаемая в данной работе задача носит исследовательский характер и вместе с тем может найти применение в реальных сферах, таких как охранные системы, системы локации и программные комплексы по анализу цифровых последовательностей. Цель работы заключается в анализе и преобразовании исходных данных в набор данных для обучения, а также в применении различных типов нейронных сетей для решения описанной задачи.

Результаты и методы, исследованные в данной работе, могут быть использованы в прикладных программных системах, работающих с цифровыми сигналами. Решаемую в данной работе проблему можно отнести к задаче пеленгации: имеются наборы из четырёх сигналов, снятых с сейсмографов. Набор сигналов характеризует колебания поверхности земли в процессе перемещения человека в окрестности датчиков. Дополнительно к этому имеются данные трекинга человека (набор пар «метка времени – *GPS* координата»).

Имеется несколько вариантов перемещения человека: перемещение по кругу и по прямой линии, в обоих случаях датчики расположены на фиксированном расстоянии друг от друга в вершинах квадрата.

Задача заключается в создании системы искусственного интеллекта, которая самостоятельно может как минимум определить угол относительно геометрического центра квадрата датчиков по сигналам. Описанная задача относится к классу машинного обучения с учителем и носит регрессионный характер, поскольку у нас имеются реальные данные, на основе которых необходимо обучить алгоритм делать прогноз координаты, основываясь лишь на сигнале.

1 Анализ современных методологий и решений в областях машинного обучения и нейронных сетей

1.1 Анализ архитектур нейронных сетей

В настоящее время существует огромное количество методов искусственного интеллекта и машинного обучения для решения задач регрессии и классификации, но в большинстве случаев применяют нейронные сети различной архитектуры и степени сложности [2, 3]. Каждая из архитектур определённым образом связана с формой представления данных, с которой она наиболее успешно справляется, но, тем не менее, не ограничивается применением конкретно к «своей» форме.

Первой архитектурой нейронной сети была нейронная сеть прямого распространения – *FNN* (рисунок 1). Она представляет из себя последовательность операций матричного умножения и нелинейных дифференцируемых функций, называемых обычно функциями активации, различных типов, таких как *ReLU*, сигмоидальная функция и функция гиперболического тангенса. Данная сеть хорошо решает задачу классификации и регрессии, но имеет ряд недостатков:

- большое количество параметров, поскольку каждый искусственный нейрон предыдущего слоя связан с каждым искусственным нейроном следующего слоя,
- затухание градиента, при котором ошибка от конечных слоев к начальным передается с огромными потерями. Данная ошибка возникает при глубокой вложенности,
- отсутствие «памяти» – события в нейронной сети являются по сути точечными наблюдениями, ввиду чего крайне трудно обучить *FNN* на последовательности.

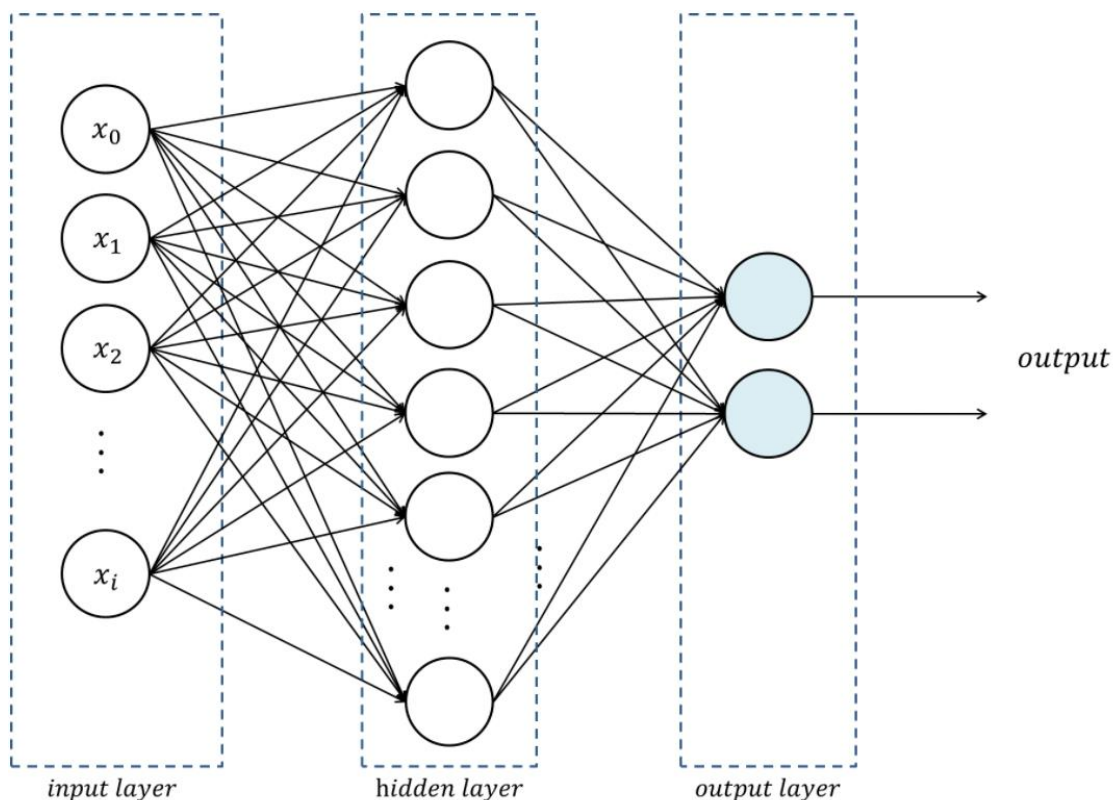


Рисунок 1 – Нейронная сеть прямого распространения

Сверточные нейронные сети – второе поколение архитектур, предложена Яном Лекунем в 1988 году. Наиболее часто данный вид архитектуры применяют для работы с изображениями разной природы. Данную архитектуру применяют для задач классификации, детекции и сегментации. По сравнению с сетью прямого распространения данная архитектура имеет меньшее количество параметров за счёт использования сверточных слоёв и пулинг слоёв. Схема расположения слоёв сверточной нейронной сети представлена на рисунке 2.

Сверточная нейронная сеть похожа на сеть прямого распространения, вместе с тем она имеет в своём составе новые типы слоёв – свёртка и пулинг. Свёртка – это оконная функция, которая применяется к входному тензору, N-мерному вектору, значением которой является тензор размерности, отличной или такой же от исходной. Обычно размерность тензора уменьшается, а количество каналов увеличивается. Для каждого канала существует собственный матричный оператор.

Пулинг – операция по сокращению размерности тензора. Существует две основные разновидности:

- *max_pooling* – функция вычисляет наибольшее значение в окне,
- *avg_pooling* – функция вычисляет среднее значение в окне.

Свёрточная нейронная сеть может анализировать как непосредственно сам сигнал, так и его производные представления – спектрограмму, которая по сути является картой частот.

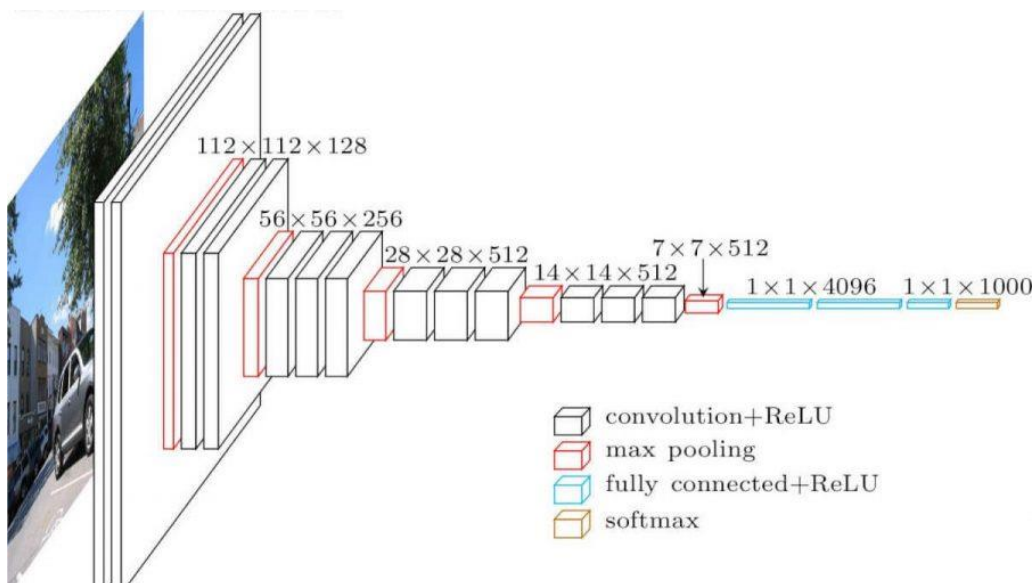


Рисунок 2 – Свёрточная нейронная сеть

Свёрточная нейронная сеть имеет большую способность к обобщению и выявлению паттернов, но также не имеет запоминающего механизма. Для решения этой проблемы были сконструированы рекуррентные нейронные сети

Рекуррентная нейронная сеть (*RNN*) – очередное поколение архитектур; не смотря на то, что впервые была описана в 1982 году, она начала активно использоваться в прошлом десятилетии в связи с ростом вычислительных мощностей различных типов ЭВМ. Основное отличие от *FNN* архитектуры является наличие рекуррентной связи: более того, данный вид сети Тьюринг полные (т.е. можно реализовать любую вычислимую функцию), а также данная архитектура хорошо подходит для работы с

последовательностями, моделированием контекста и временными зависимостями.

Основные принципы работы были заложены в 1980 году. Нейронная сеть устроена таким образом, что каждый узел соединен с остальными. Нейроны разделяются на несколько типов: выходные, скрытые и входные (рисунок 3).

Сложность обучения такой нейронной сети заключается в нескольких проблемах:

- необходимы большие вычислительные мощности (в частности, память), поскольку нужно создавать копию слоя для каждого временного отсчёта,
- вычислительная неустойчивость, что проявляется в большом отклонении весов от итерации к итерации (веса могут как затухать, так и становиться слишком большими).

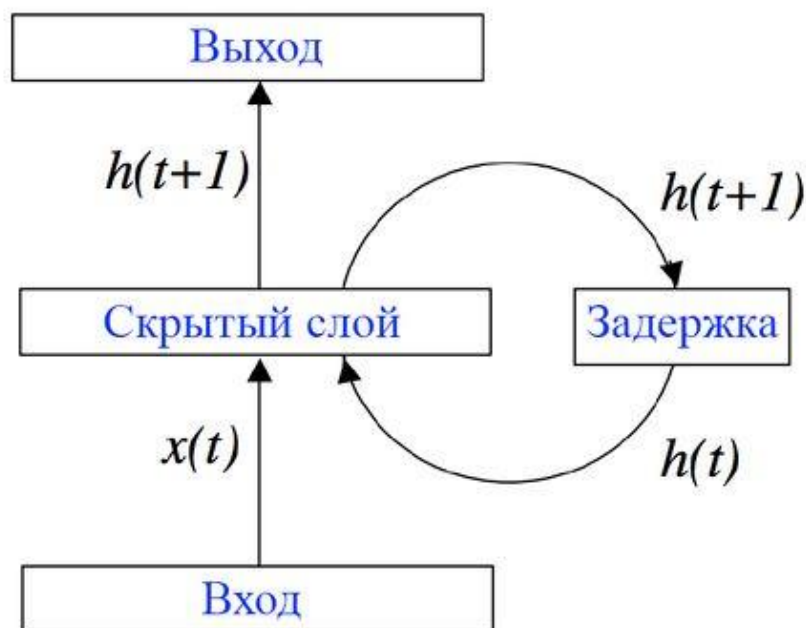


Рисунок 3 – Рекуррентная нейронная сеть

Данная архитектура так же как и сеть прямого распространения при увеличении количества слоёв подвержена затуханию градиента[4]. В связи с этим была разработана модификация данной архитектуры – *LSTM* (*Long Short-*

Term Memory) и *GRU (Gated Recurrent Unit)*. Данные блоки сложнее чем обычные искусственные нейроны, они выполняют роль ячеек памяти.

Проблему резко изменяющихся весов удалось решить с помощью архитектуры *LSTM*. Сеть вносит новый механизм, называемый вентилями забывания (рисунок 4). Распространение ошибок может происходить через неограниченное количество виртуальных слоёв, благодаря чему сетью *LSTM* могут учитываться даже далёкие от конечного события.

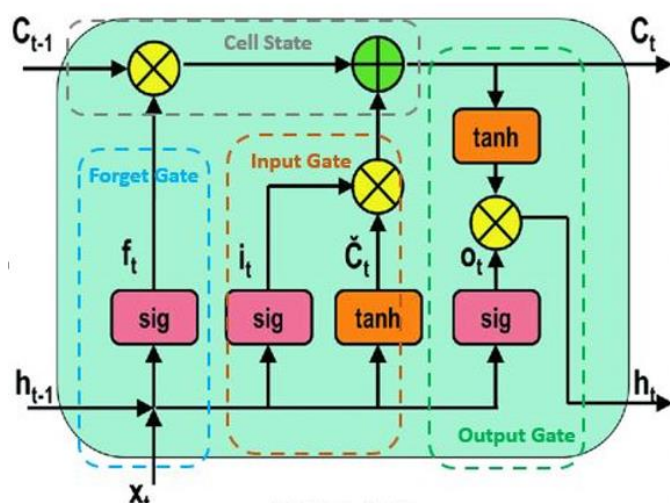


Рисунок 4 – LSTM ячейка

LSTM также сильно требовательна к вычислительным мощностям, что может затруднить её применение в ряде случаев [5]. С проблемой значительных требований к вычислительным ресурсам успешно справился управляемый рекуррентный блок (*GRU*) – в моделях речевого сигнала его производительность схожа с *LSTM*, вместе с тем, она требует меньшее количество параметров (рисунок 5).

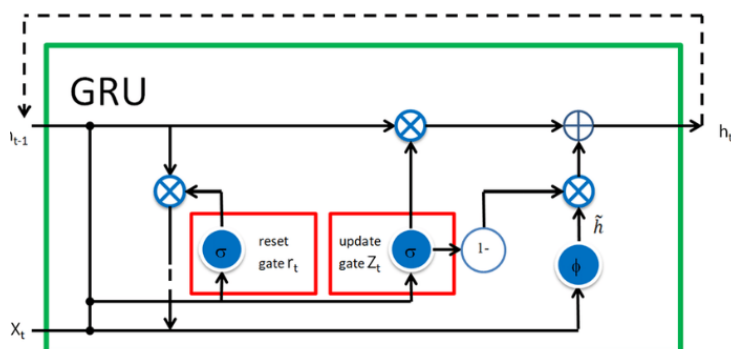


Рисунок 5 – GRU ячейка

1.2 Анализ языков программирования для реализации и обучения нейронных сетей

Для решения задач в области искусственного интеллекта и машинного обучения в настоящее время используется в основном два языка программирования: *Python* и *C++*. Оба языка обладают как преимуществами и недостатками [6]. Язык *Python* является интерпретируемым языком, который позволяет увеличить скорость разработки, но на этапе вывода нейронной сети (*inference*) является нежелательным, так как задача машинного обучения зачастую относится к задачам, ограниченным по ЦПУ. Сравнение двух языков представлено в таблице 1.

Таблица 1 – Сравнительный анализ языков программирования

Свойство	<i>Python</i>	<i>C++</i>
Кроссплатформенность	Да	Нет
Скорость разработки	Высокая	Низкая
Скорость вычислений	Низкая	Высокая
Библиотеки для машинного обучения	Большое количество библиотек различной сложности и функциональности, простота установки и начала работы	Количество библиотек меньше, степень поддержки различна; сложности при начале работы
Цели использования	Исследование и обучение моделей, возможность интерактивного исполнения написанного кода (при использовании <i>Jupyter</i>) — оптимально для <i>compile-time</i>	Реализация уже исследованной и хорошо зарекомендовавшей себя модели машинного обучения; оптимально для <i>inference-time</i>

В силу того, наша задача в настоящий момент носит более исследовательский характер, нежели практический, и мы не имеем готового решения, нам следует выбрать в качестве инструмента разработки язык программирования *Python*.

1.3 Анализ библиотек машинного обучения

Язык программирования *Python* в силу своей простоты и выразительности смог обзавестись огромным количеством библиотек различной популярности; некоторые из них стали стандартом для разработки проектов с применением искусственного интеллекта и машинного обучения. Эти библиотеки имеют поддержку либо от огромного сообщества *Open source* разработчиков, либо от больших *IT* корпораций, таких как *Google* и т.д.

Некоторые библиотеки стали безальтернативны: для математических вычислений используется *numpy*, для работы с визуализацией данных, построения графиков, диаграмм, гистограмм и иных визуальных методов представления информации применяется *matplotlib* и *seaborn*, для работы с табличными данными – *pandas*.

Для библиотек машинного обучения нейросетевых моделей дело обстоит иначе. В свободном доступе имеется большое количество библиотек, и основную долю рынка делят между собой *tensorflow* и *pytorch*. Данные библиотеки отличаются прежде всего способом программирования модели нейронной сети. Каждая из указанных библиотек также, как и *numpy*, использует технологию *BLAS* (*basic linear algebra subprograms*), которая позволяет значительно ускорить выполнение векторных и матричных операций.

PyTorch был впервые показан в 2016 году, он был основан на *Torch* – *matlab* подобной библиотеки для *Lua*. *PyTorch* написан на *Python* и *C/C++*, что позволяет ему достаточно быстро исполнять вычисления на как на *CPU*, так и на *GPU*. Создание модели в данном фреймворке происходит посредством создания класса наследника. Данная библиотека имеет отличную документацию и большое количество литературы с примерами использования

той или иной функциональности. Библиотека обладает средствами оптимизации модели для переноса её на процессоры с иной архитектурой, встроенные системы и мобильные устройства.

Tensorflow – библиотека для машинного обучения, разработанная компанией *Google* в 2015 году. Данная библиотека имеет две версии; вторая версия библиотеки появилась в результате того, что *Tensorflow* была практически вытеснена библиотекой *PyTorch* с рынка исследовательских работ. В настоящее время *Tensorflow* содержит в себе библиотеку *Keras*, в прошлом отдельный проект, основанный на *Tensorflow*. *Tensorflow* также имеет хорошую документацию, но количество литературы и обучающих примеров с использованием библиотеки несколько меньше.

В связи со всеми преимуществами и недостатками, перечисленными выше, мы склоняемся к использованию библиотеки *PyTorch* в данном проекте.

1.4 Анализ языков программирования и систем управления базами данных для реализации прикладного API

К данному моменту реализована непосредственно нейронная сеть – часть системы, отвечающая за обработку данных от сенсорной системы. На данном этапе необходимо выбрать

- язык реализации приложения, реализующего *REST API*
- СУБД, используемая для хранения данных

Существует огромное количество языков программирования для реализации веб-сервисов. Наиболее популярными являются языки с использованием виртуальных машин и управляемой средой, которые позволяют быстро реализовать проект, а также содержат большое количество вспомогательных библиотек. Эти языки: *Java*, *Python*, *C#*, *Ruby*.

С другой стороны, в области проектирования и построения микросервисной архитектуры наблюдается тенденция к сокращению требуемых вычислительных мощностей, требуемых для работы веб-сервисов. Таким языком является *Go*, с одной стороны, имеющий сборщик мусора и

высокоуровневый конструкции для применения асинхронности, и с другой стороны являющийся компилируемым.

Go является современным и технологическим средством для создания веб-сервисов, поэтому при имплементации веб-сервиса мы склоняемся к выбору данного языка.

Существует большое количество СУБД, так или иначе обеспечивающих гарантии по атомарности, согласованности, изолированности и устойчивости (*ACID*). Данные гарантии необходимы для обеспечения общей отказоустойчивости системы. Наиболее популярными решениями являются *MySQL*, *PostgreSQL* и *SQL Server*.

MySQL является высокопроизводительной базой данных, оптимизированной под большое количество операций чтения. Вместе с тем она обладает слабым параллелизмом во время операций записи ввиду использования блокировок строк и страниц внутри файлов таблицы базы данных. Так же *MySQL* имеет меньшую согласованность в сравнении с другими рассматриваемыми решениями.

PostgreSQL ставила целью быть базой данных общего назначения, справляясь с нагрузками как при записи, так и при чтении. Она имеет более сильные гарантии *ACID*, поскольку использует технологию *MVCC* для хранения строк таблицы.

Разработанная *Microsoft SQL Server* является коммерческим решением, имеющим богатый набор дополнительных функций по подключению дополнительных источников данных и обладающим высокой производительностью. Вместе с тем, *SQL Server* также использует большое количество блокировок при исполнении пользовательских запросов.

Основываясь на вышеизложенном, мы склоняемся к использованию *PostgreSQL*, поскольку именно эта система полностью соответствует требованиям описываемой системы.

2 Подготовка данных и обучение нейросетевых алгоритмов

2.1 Анализ и предобработка данных

Анализ и предобработка – важная часть работы в машинном обучении, от которой во многом зависит результат обучения модели.

Анализ и предобработку данных следует разделить на следующие этапы:

- чтение данных из файлов,
- визуализация данных,
- приведение данных к форме, доступной в качестве обучающих примеров для нейронной сети.

У нас в наличие имеется два файла. Первый файл – файл *grx*, содержит временные метки и координаты (широта и долгота) в соответствующий момент времени. Временные метки не равноудалены друг от друга, а имеют различное временное расстояние между друг другом. Второй файл – файл с расширением *mat*, применяемый в программном обеспечении *Matlab*. Он содержит 4 сигнала (4 массива амплитуд), приведённые к декартовым координаты (в точке (0, 0) находится центр квадрата датчиков).

Для того, чтобы прочесть файлы с расширениями *grx* и *mat*, нам потребуется использовать две библиотеки: *grxpy* и *scipy* версий *latest* (на момент написания – 1.6.2 и 1.12.0 соответственно). С использованием данных библиотек напомним функции-утилиты для чтения файлов. Данные утилиты мы можем вынести в отдельный файл и папку для построения пакетной структуры проекта, которая значительно упростит разработку.

```

def read_gpx(file_name):
    with open(file_name, 'r') as f:
        data = gpxpy.parse(f)
        return data

def get_points(gpx):
    return gpx.tracks[0].segments[0].points

def as_unix(tm):
    t = tm.replace(tzinfo=None)
    return (t - datetime.datetime.fromtimestamp(datetime.UTC)).total_seconds()

def get_track_data(file):
    data = read_gpx(file)
    points = get_points(data)

    long_arr = []
    lat_arr = []
    time_arr = []

    for r in points:
        long_arr.append(r.longitude)
        lat_arr.append(r.latitude)
        time_arr.append(as_unix(r.time))

    long_arr = np.array(long_arr)
    lat_arr = np.array(lat_arr)
    time_arr = np.array(time_arr)

    time_arr = time_arr - np.min(time_arr)

    return long_arr, lat_arr, time_arr

```

Рисунок 6 – Функция для чтения gpx файла

Затем, прочитав данные, мы можем визуализировать их. Для визуализации будем использовать библиотеку *matplotlib*, имеющую интеграцию с *numpy*.

Прежде всего попробуем визуализировать траекторию движения человека вокруг датчиков. Поскольку у нас уже имеются приведённые координаты, нам остаётся лишь вывести данные точки на координатную плоскость.

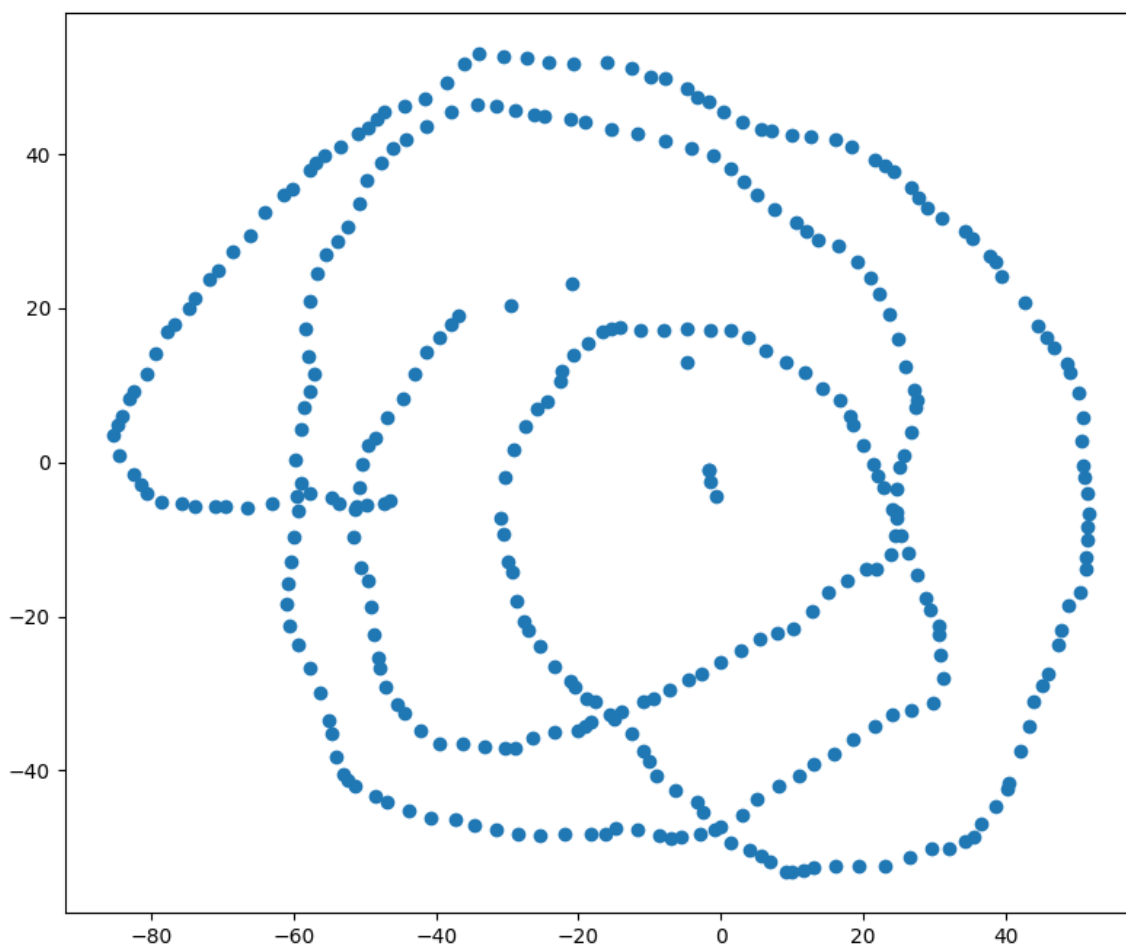


Рисунок 7 – *GPS*-трекинг

Указанные координаты являются декартовыми. Основной задачей для нас является определение угла относительно центра квадрата, в вершинах которого располагаются датчики. Таким образом, дополнительно ко всему нам потребуется переход от декартовой системы координат к полярной.

Затем мы можем отобразить график сигнала с датчиков. Поскольку у нас отсутствуют временные метки для каждой амплитуды, мы будем считать, что длительность сигнала – это разность времени между начальной и конечными точками времени в файле *grx*.

Узнав длительность сигнала, мы можем построить не только график амплитуд колебаний, но и спектрограммы сигналов.

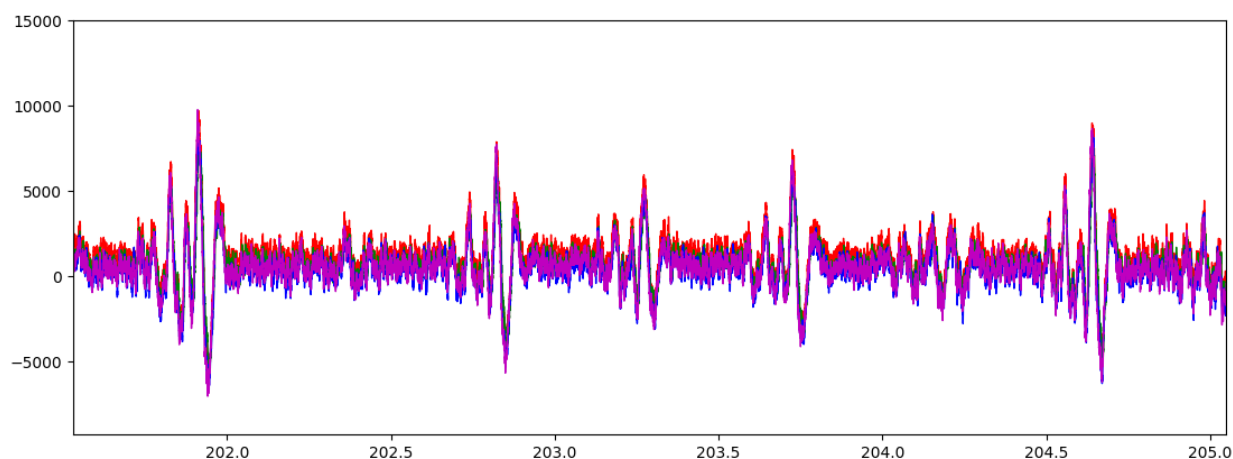


Рисунок 8 – График колебаний, записанный с датчиков

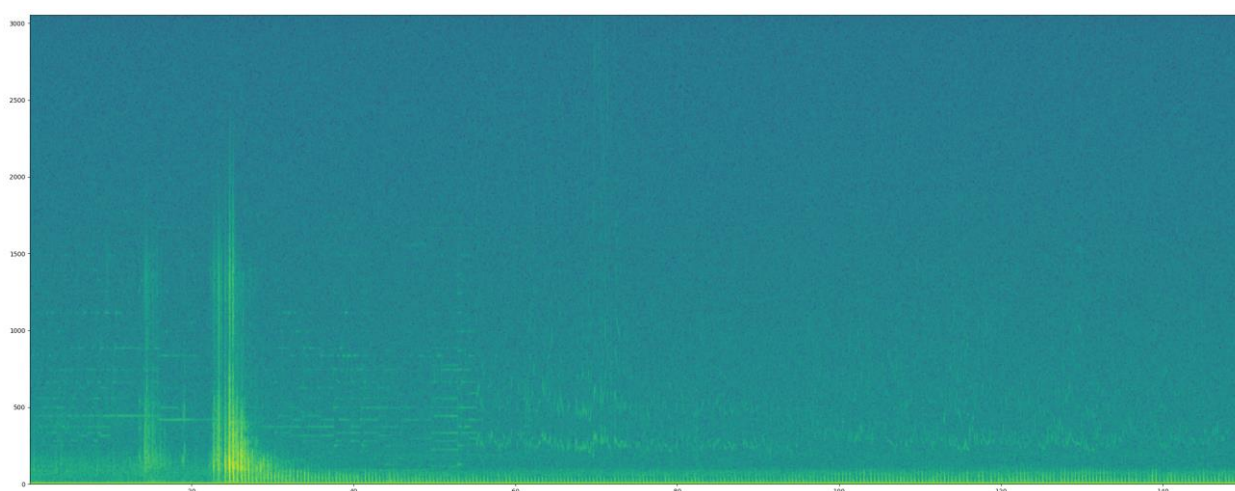


Рисунок 9 – Спектрограмма сигнала, полученная с первого датчика

Визуально можно заметить отдельные отрезки возмущений среди шумов, которые возникают с определённой частотой; данные возмущения – отдельные шаги.

Изучив данные, мы можем перейти к алгоритму построения набора данных:

- разделение сигнала на отрезки времени, в которые попадает как минимум один шаг (0.7 – 0.9 секунд),
- аппроксимация координаты к середине изучаемого диапазона,
- переход от полученных декартовых координат к полярным.



Рисунок 10 – GPS трекинг с аппроксимацией по времени

Отдельно отметим, что сигнал имеет большое количество шумов. С одной стороны, задачи в области цифровой обработки сигналов зачастую применяют фильтры, например, фильтр Гаусса, чтобы понизить уровень шума, а также используют и другие методы предобработки, например, линейное преобразование, выравнивание амплитуд и т.д.

С другой стороны, поскольку задержка между фронтами сигналов крайне мала, математические и иные операции над вектором, представляющим сигнал, могут исказить и даже снизить результаты обучения, поскольку есть вероятность удаления важных признаков.

2.2 Подготовка данных и обучение моделей

Для формирования итогового набора данных имеется три пары файлов, один из которых – трекинг человека, другой – сигнал снятый с сейсмографов за указанный диапазон времени).

Характеристики файлов с начальными данными указаны в таблице 2.

Таблица 2 – Характерные черты файлов с необработанными данными

Номер файла	Тип движения	Длительность	Количество отсчётов / сек.
1	Линейный, с центром в «квадрате датчиков»	≈ 1200 с.	5125
2	Круговой, с центром в «квадрате датчиков»	≈ 670 с.	5125
3	Круговой, с центром в «квадрате датчиков»	≈ 520 с.	5125

Отметим, что файлы с трекингом и с сигналом имеют немного различную длительность, в связи с чем нам потребуется либо аппроксимировать координаты трекинга, либо удалить конечную часть сигнала, для которой не имеется достоверного трекинга. Ввиду ограниченного набора данных, нам следует использовать именно аппроксимацию.

В связи с тем, что аппроксимировать требуется отрезки длиной в 5-10 секунд, а также тем, что каждый файл имеет свой собственный паттерн движения, аппроксимация даст результат, достаточно близкий к реальным данным.

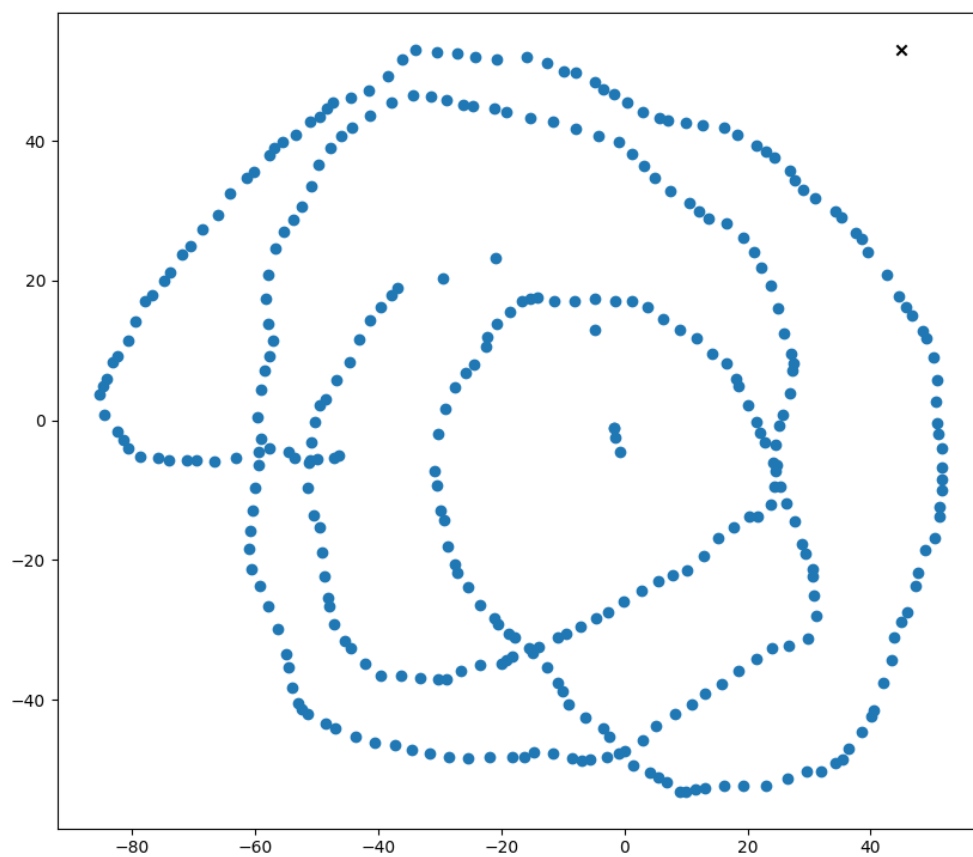


Рисунок 11 – Трекинг координат для второго файла

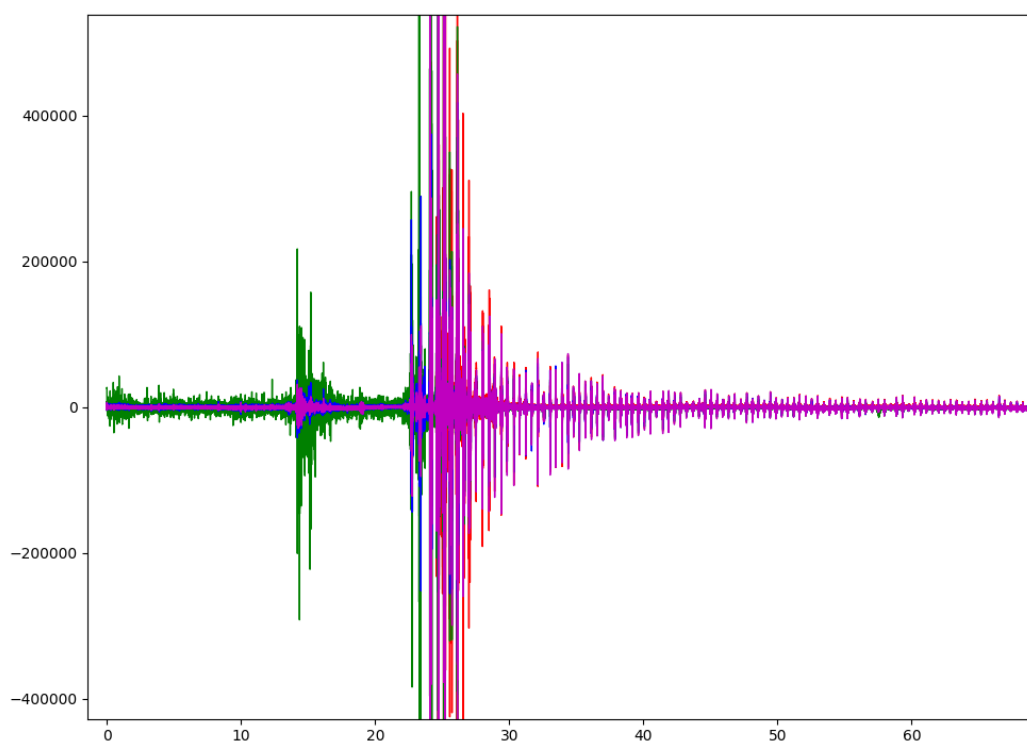


Рисунок 12 – График зависимости амплитуды сигнала от времени

Исследуем сигнал амплитуд более подробно. На графике чётко просматриваются синусоидальные «всплески» длиной 0.3-0.4 секунды с периодичностью 0.5 секунд – отдельные шаги человека. Они имеют сильную низкочастотную составляющую, в то время как шум в основном состоит из набора высокочастотных сигналов.

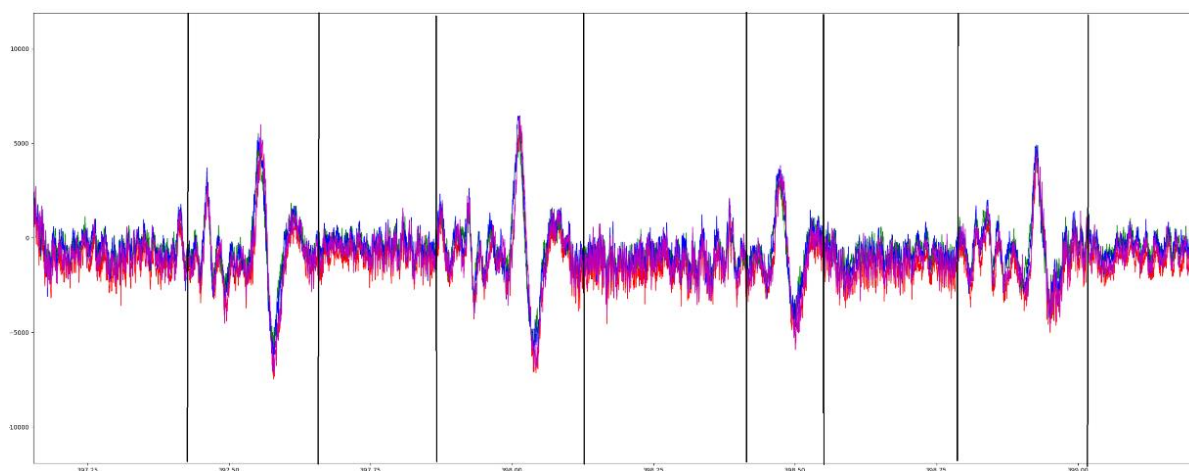


Рисунок 13 – Выявление области интересов из сигнала

Сигнал можно разделить на обучающие примеры при помощи функции скользящего окна $window(S, delta, duration)$, которая извлекает из сигнала S множество семплов с длительностью $duration$ и шагом окна $delta$. В связи с тем, что сигнал не синхронизирован по времени с трекингом, разделение сигнала на обучающие примеры должно происходить по номеру (индексу) амплитуды. Данный метод позволит нам извлечь большее количество обучающих примеров из «сырых» данных, не вызывая при этом переобучения. Отметим, что $delta$ и $duration$ должны быть подобраны таким образом, чтобы их сумма давала длительность как минимум одного полного шага. Это нужно для того, чтобы в обучающих примерах не было экземпляров, содержащих исключительно шум.

Затем, нужно аппроксимировать координаты из трекинга относительно середины окна. Ниже представлен алгоритм разбиения исходных данных

```

def split(track, time, signal, rate, sample_size=7000, sample_delta=600, polar=True):
    l, r = 0, sample_size

    total_samples = int((signal.shape[0] - sample_size - sample_delta) / sample_delta)

    X = np.zeros(shape=(total_samples, sample_size, 4))
    Y = np.ones(shape=(total_samples, 2))

    for i in range(total_samples - 1):
        try:
            x = signal[l:r, :]

            target_time = (l + sample_size / 2) / rate
            y = approximate_coord_at_time(track, time, target_time)
            if polar:
                y = cartesian2polar(*y)

            X[i] = x
            Y[i] = y

            l += sample_delta
            r += sample_delta
        except:
            print('Exxception on split dataset')

    return X, Y

```

Рисунок 14 – Функция разделения исходных данных

После разбивки данных на обучающие примеры нам необходимо сохранить их на жёсткий диск для экономии времени: выгрузить уже разбитый набор данных будет кратно быстрее, чем загружать отдельные файлы, снова разбивать их на обучающие примеры.

```

import os
import re

import gpx_read as gpx
import signal_reader as matlab
import numpy as np
import utils
import torch

```

```

def etl_dataset():
    all_track_files = ['resources\\18_11_2014 17_05 Ман1.gpx',
                       'resources\\18_11_2014 17_24 Ман2.gpx',
                       'resources\\18_11_2014 17_47 ман3.gpx'
                       ]

    all_signal_files = ['resources\\18_11_2014 17_05
Ман1.gpx.mat', 'resources\\18_11_2014 17_24 Ман2.gpx.mat',
'resources\\18_11_2014 17_47 Ман3.gpx.mat']

    adc = [('man1ADC1', 'man1ADC2'), ('man2ADC1', 'man2ADC2'),
('man3ADC1', 'man3ADC2')]

    i = 0

    for t_file, s_file in zip(all_track_files,
all_signal_files):
        long, lat, time_points = gpx.get_track_data(t_file)
        zero, track, sensor1_signal, sensor2_signal =
matlab.read_mat(s_file, k_adc1=adc[i][0], k_adc2=adc[i][1])
        track = track[:, 1:]
        sample_rate = 5120

        X, Y = utils.split(track, time_points, sensor1_signal,
sample_rate)

        # [семпл, канал, амплитуда]
        X = torch.tensor(X).float().permute((0, 2, 1))
        Y = torch.tensor(Y).float()

        torch.save(X, f'dataset/X_{i}.pt')
        torch.save(Y, f'dataset/Y_{i}.pt')

    i += 1

```

Помимо сохранения, нам нужно иметь функцию-загрузчик, которая может извлечь данные из разных частей набора в один общий набор. Её код представлен ниже.

```
def get_dataset():
    files = os.listdir('dataset')
    dct = {}
    for d in files:
        part, index =
re.compile('(\w)_(\d)\.pt').match(d).groups()
        index = int(index)
        if dct.get(index) is None:
            dct[index] = {part: torch.load(f'dataset/{d}')}
        else:
            dct[index][part] = torch.load(f'dataset/{d}')

    x = []
    y = []
    for d in dct:
        x.append(dct[d]['X'])
        y.append(dct[d]['Y'][:, 1])

    x = torch.cat(x)
    y = torch.cat(y)

    return x, y.
```

Полученные данные хранятся в следующем порядке:

- семпл,
- порядковый номер амплитуды,
- канал.

Указанный выше порядок подойдёт для использования рекуррентных моделей, *RNN*, *LSTM* и *GRU*. *CNN* слои в *pytorch* работают с отличным от данного порядком, поэтому перед сохранением набора данных нам

необходимо изменить порядок хранения на требуемый при помощи функции *torch.permute*, а перед обучением и тестированием модели уже приводить к нужному порядку измерений. Данные внутри файлов будут храниться в следующем порядке:

- семпл,
- канал,
- порядковый номер амплитуды.

Примеры, сформированные из исходных данных, необходимо сохранить на жёсткий диск с целью оптимизации процесса разработки модели нейросетевого алгоритма.

Как итог, нам удалось сформировать 6 файлов, 3 файла с обучающими примерами и 3 файла с соответствующими им целевым значениям. Обучающий набор имеет размер около 2 Гб.

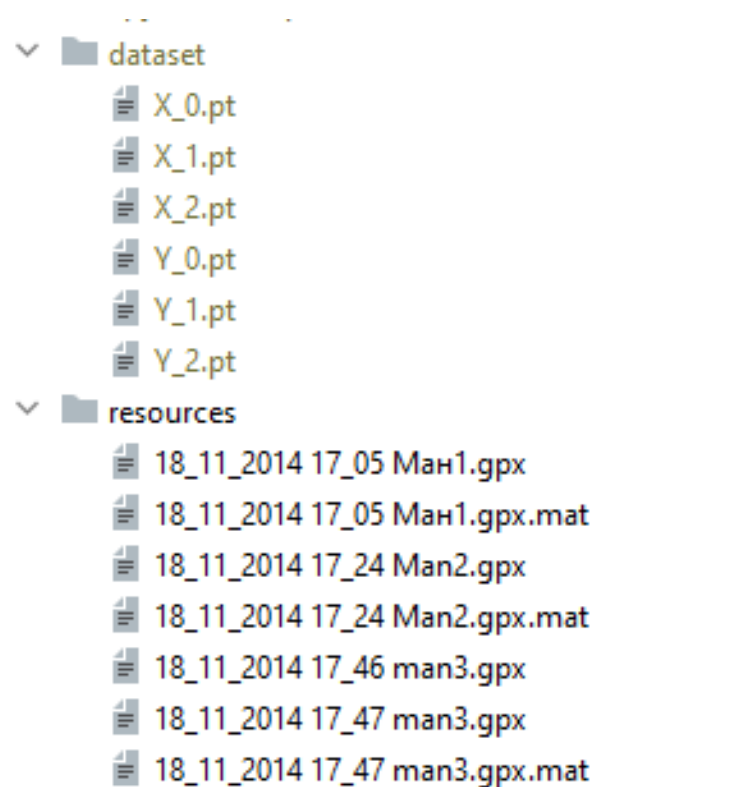


Рисунок 15 – Папка со сформированным набором данных

Современные подходы к обучению нейросетевых алгоритмов говорят нам о необходимости разделения всего множества обучающих примеров на три когорты:

1) тренировочный набор, на котором происходит подбор параметров нейросетевого алгоритма, содержит от 60 до 80% от общего набора данных

2) тестовый набор для подтверждения результатов обучения. Тестовый набор применяется после обучения и составляет от 10% до 30% данных

3) валидационный набор, применение которого каждую N итерацию в процессе обучения позволяет выявить недообучение или переобучение нейросетевого алгоритма. Составляет от 10% до 20% от общего набора.

Исходные данные были разбиты на 19 000 обучающих примеров, и, исходя из многочисленных опытов, мы имеем следующее разбиение наборов данных: *train* – 14000, *test* – 3200, *validation* – 2500.

Поскольку данные загружались из трёх разных файлов, их необходимо смешать между собой для формирования наиболее репрезентативных наборов. Проблема может заключаться в том, что в зависимости от многих параметров перестановка обучающих примеров может быть разной, и (к примеру) ранее сохранённая и обученная модель получит обучающий пример из тренировочного, а не из тестового набора. Во избежание данной проблемы этого нам нужно создать генератор с предустановленным начальным значением (*seed*), которое не будет меняться[7]. Таким образом мы можем добиться формирования одинаковых наборов данных от сессии к сессии; для формирования другого разделения данных мы должны будем лишь изменить параметр *seed*.

Обучение нейронной сети будет происходить на видеоадаптере *Nvidia 1080 Super*, 6 GB VRAM. Для оптимизации процесса обучения мы будем подавать на вход не отдельные примеры, а пакеты определённого размера (*mini-batch*).

```

n_train, n_val, n_test = 14000, 2500, 3200
X = X[:19700]
Y = Y[:19700]
batch_size_train = 100
batch_size_test = 1

# нужно для того чтобы была одинаковая разбивка раз за разом
gen = torch.Generator()
gen.manual_seed(4731511)

perm = torch.randperm(X.shape[0], generator=torch.Generator())
X = X[perm]
Y = Y[perm]

if X.shape[0] != Y.shape[0]:
    raise Exception('diff length')
n = X.shape[0]
iters = 0

X_train, X_val, X_test = torch.split(X, [n_train, n_val, n_test])
Y_train, Y_val, Y_test = torch.split(Y, [n_train, n_val, n_test])

X_train = X_train.reshape((-1, batch_size_train, 4, 7000))
Y_train = Y_train.reshape((-1, batch_size_train, 1))

X_val = X_val.reshape((-1, batch_size_train, 4, 7000))
Y_val = Y_val.reshape((-1, batch_size_train, 1))

X_test = X_test.reshape((-1, batch_size_test, 4, 7000))
Y_test = Y_test.reshape((-1, batch_size_test, 1))

```

Рисунок 16 – Разбиение обучающих примеров на наборы данных

2.3 Обучение CNN

Первая разработанная модель нейронной сети относится к классу сверточных нейронных сетей. Её особенностью является параметрическое пространство для входных данных – одномерный массив с 4 каналами. Модель организована с помощью слоя обёртки *Sequential*, которая передаёт выход слоя $n-1$ на вход слоя n . Слои последовательно увеличивают количество каналов, извлекая дополнительные признаки, после чего сверточные слои передают их в линейный слой (слой, аналогичный *FNN*). Линейные слои уменьшают количество признаков, агрегируя их в другие признаки, до тех пор,

пока не останется единственный признак, значение в котором – требуемый прогноз.

```
seq_model = nn.Sequential(  
    nn.Conv1d(4, 80, kernel_size=24, stride=8),  
    nn.ReLU(),  
    nn.MaxPool1d(2, 2),  
    nn.Conv1d(80, 160, kernel_size=16, stride=4),  
    nn.ReLU(),  
    nn.MaxPool1d(2, 2),  
    nn.Conv1d(160, 240, kernel_size=8, stride=2),  
    nn.ReLU(),  
    nn.MaxPool1d(2, 2),  
    nn.Flatten(),  
    nn.Linear(720),  
    nn.ReLU(),  
    nn.Linear(480),  
    nn.ReLU(),  
    nn.Linear(360),  
    nn.ReLU(),  
    nn.Linear(1)  
)
```

Рисунок 17 – Описание нейронной сети на *Pytorch*

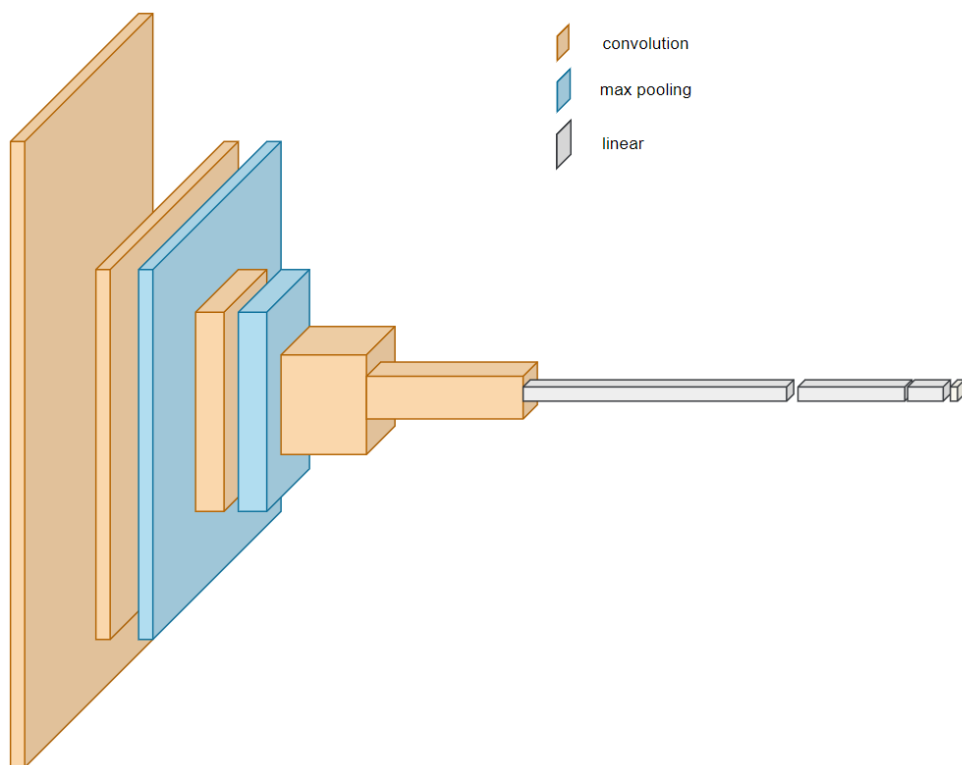


Рисунок 18 – Схема нейронной сети

Процесс обучения происходит на видеоадаптере, применяется мини-пакетное обучение, в роли оптимизатора выступает алгоритм *Adam* со стандартными параметрами, функция потерь – сглаженная *L1* метрика (*SmoothL1Loss*). Каждые три итерации происходит проверка модели на валидационном наборе данных.

На этапе валидации модели используется специфическая для набора данных функция точности. Выходом модели является значение угла в радианах в области $[0, 2\pi)$. При расчёте *L1* напрямую могут возникнуть ошибки; предположим, что прогнозируемое значение находится в точке А, в то время как целевое значение – точка В.

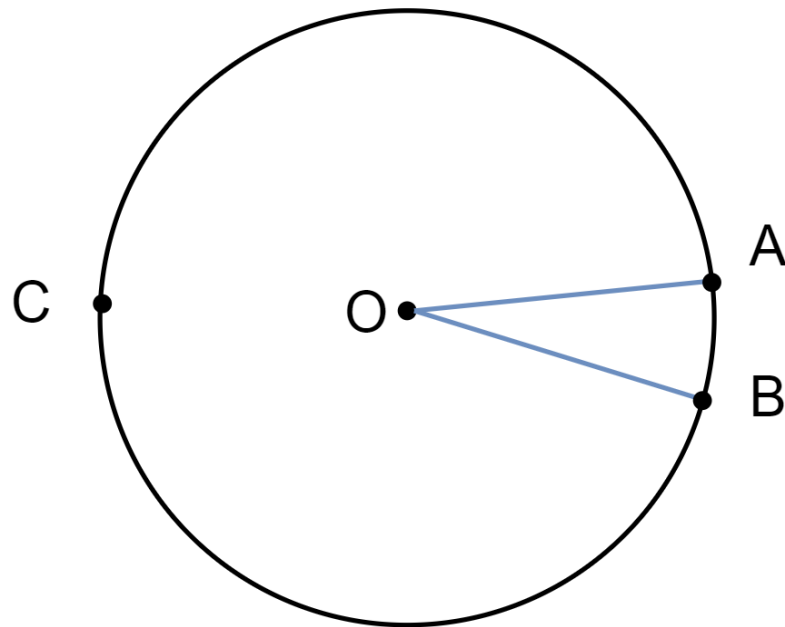


Рисунок 19 – Пример ошибки при вычислении разности прогноза и фактического значения

Истинная ошибка составляет $\angle AOB$, но при расчёте *L1* напрямую ошибка составит $\angle ACB$, что превышает фактическую в несколько раз. Во избежание данной ошибки мы должны определить функцию потерь:

$$loss(p, t) = \begin{cases} |p - t|, & |p - t| < \pi \\ |p - t| - \pi, & |p - t| \geq \pi \end{cases}$$

где:

p – ответ от нейронной сети (*prediction*),

t – реальная величина из тестового набора (*target*).

Оценка успешности предсказания вычисляется следующим образом:

$$success(p, t) = 1 - \frac{loss(p, t)}{\pi}$$

При превышении данной величины 0.935 процесс обучения приостанавливается, после чего обученная модель сохраняется в файл.

2.3 Анализ результатов обучения CNN

Результаты обучения описанной модели представлены на рисунке 10.

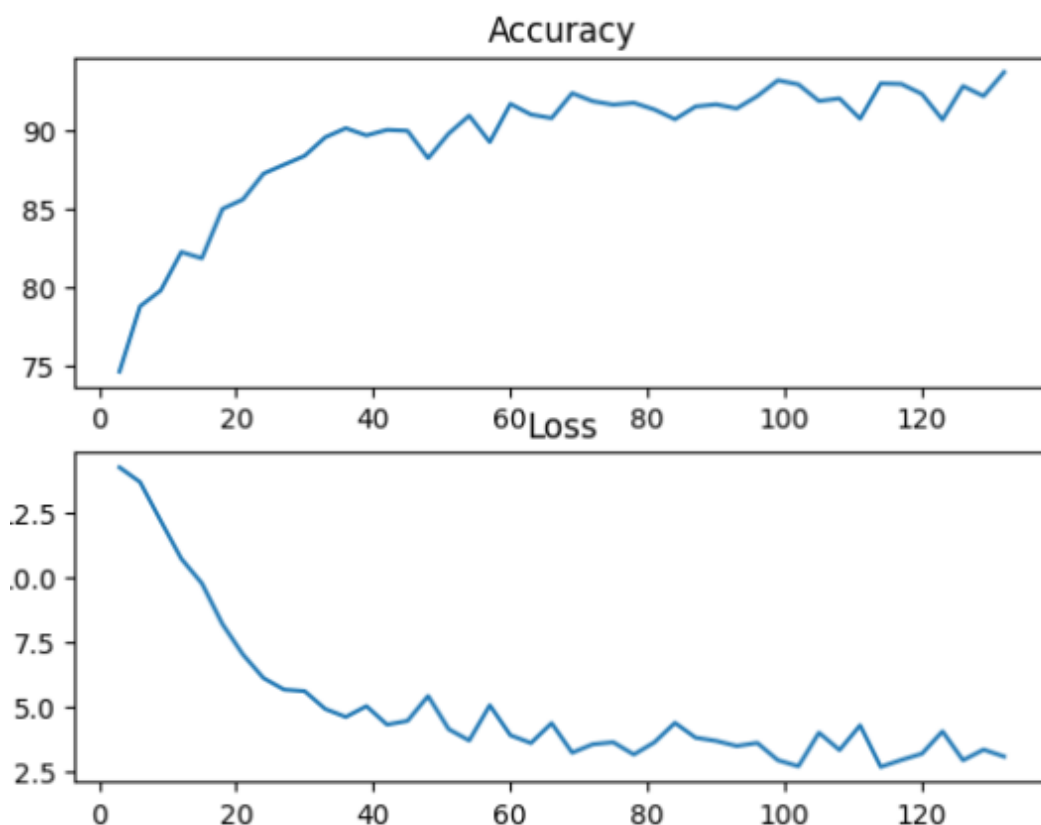


Рисунок 20 – Графики зависимости точности и функции потерь в зависимости от итерации

На начальном этапе в качестве метрики для определения результатов модели была выбрана RMSE. Данная метрика показала неудовлетворительные результаты – 0.32 *rad* (18.335°).

Исследуем потери более подробно: на рисунках 21 – 23 представлены гистограммы потерь для тестового набора данных в разных диапазонах

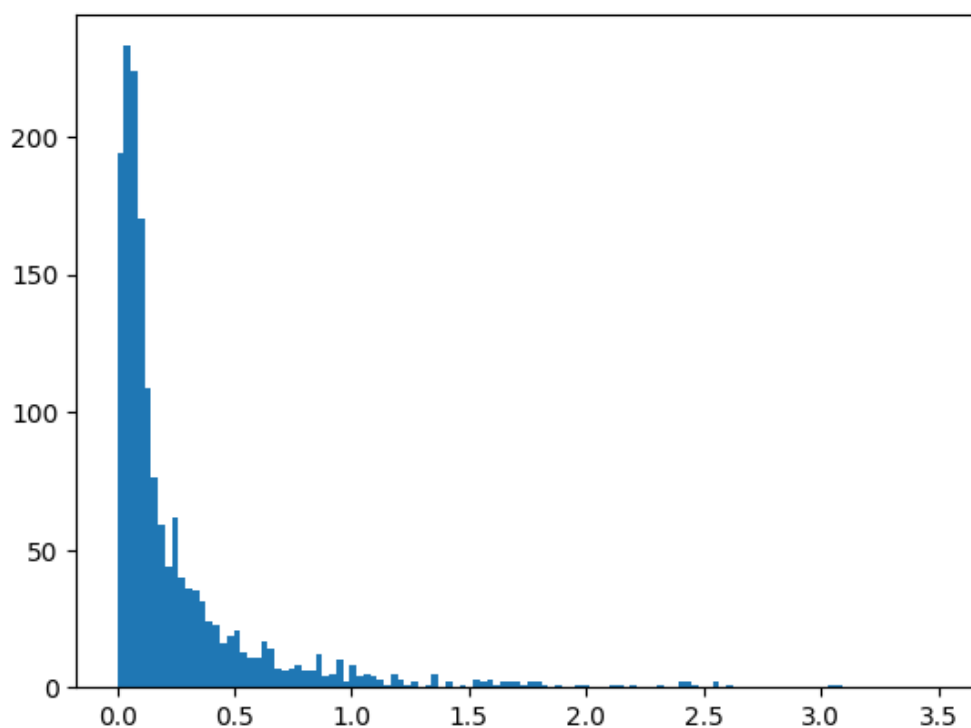


Рисунок 21 – Потери в диапазоне [0; 3.5]

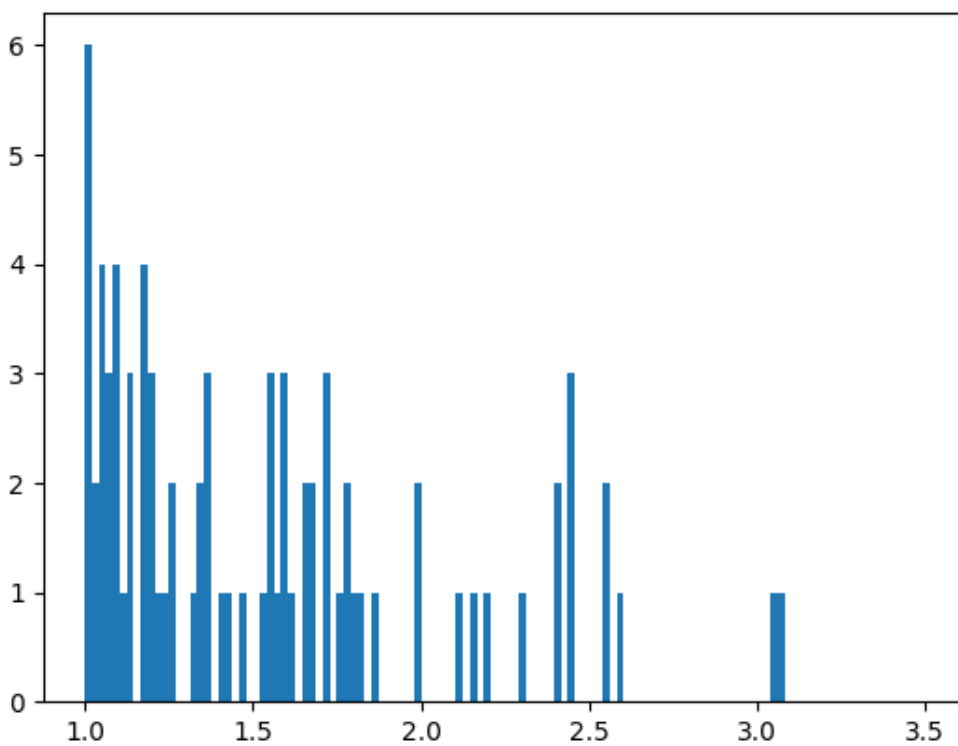


Рисунок 22 – Потери в диапазоне [1; 3.5]

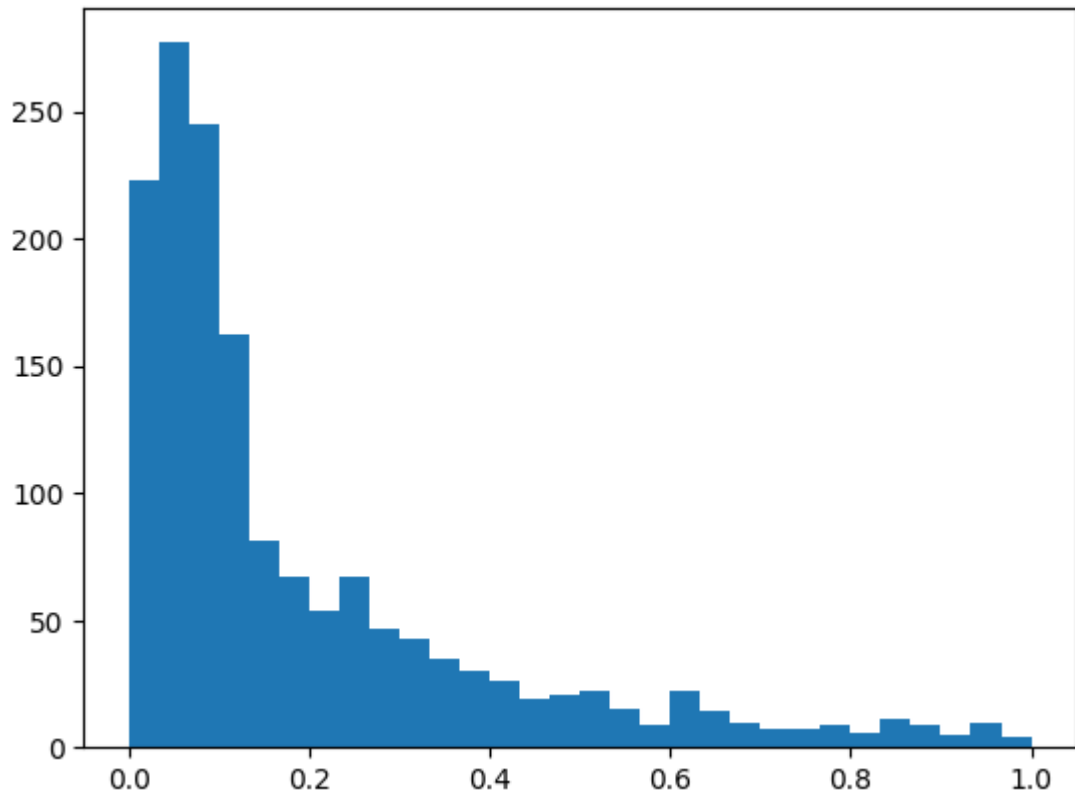


Рисунок 23 – Потери в диапазоне [0; 1]

На графиках мы видим, что основная масса ошибок сосредоточена в диапазоне [0; 0.8]. Но ввиду того, что распределение ошибки имеет некоторое количество значительных выбросов с отклонением, кратно превышающим математическое ожидание, мы получаем завышенную метрику *RMSE*.

В то же время метрика *MAE* (которая в сущности своей является мат. ожиданием ошибок) показывает низкие результаты – 0.146 *rad* (8.335°). Также отметим:

– $std(err) = 0.295$,

– 95% доверительный интервал – (0.136, 0.157).

Таким образом, мы можем утверждать, что данная модель успешна с точки зрения прогнозирования; тем не менее, в процессе работы есть небольшая вероятность получить результат, сильно отличающийся от реальности. Для получения наиболее точного прогноза можно в качестве аргумента на вход нейронной сети подавать несколько смежных обучающих примеров и применять некий агрегирующий результат к выходам для получения оптимального ответа.

После успешного применения сверточной нейронной сети попробуем обучить другие архитектуры нейронных сетей.

2.4 Обучение RNN

Реализация *RNN* в *Pytorch* требует создания отдельного класса. На рисунке 24 представлен исходный код для рекуррентной нейронной сети.

```
from torch.autograd import Variable
class RNNModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(RNNModel, self).__init__()

        # Number of hidden dimensions
        self.hidden_dim = hidden_dim

        # Number of hidden layers
        self.layer_dim = layer_dim

        # RNN
        self.rnn = nn.RNN(input_dim, hidden_dim, layer_dim, batch_first=True, nonlinearity='relu')

        # Readout layer
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        device = torch.cuda.current_device()
        # Initialize hidden state with zeros
        h0 = Variable(torch.zeros(self.layer_dim, x.size(0), self.hidden_dim)).to(device)

        # One time step
        out, hn = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])
        return out
```

Рисунок 24 – Исходный код реализации *RNN* модели

В данном случае *PyTorch* позволяет применять разновидность *stacked RNN* (сложенная рекуррентная сеть). Благодаря нескольким слоям *layer_dims* алгоритм машинного обучения способен запоминать более длительные последовательности и делать вывод на основе большего числа предшествующих событий. Остальные параметры стандартны для нейронной сети: *input_dim* отвечает за размерность входного вектора (единичного события), *hidden_dim* указывает необходимое количество скрытых слоёв, что также влияет на способность к более долговременному запоминанию событий,

а также *output_dim* – аргумент, отвечающий за размерность выходного вектора.

Обучение происходит с применением стандартного разбиения набора данных на тренировочный, валидационный и тестовый. На рисунке 25 представлен код для инициализации и обучений алгоритма.

```
ep = []  
l = []  
a = []  
rnn_model = RNNModel(4, 1, 1, 1)  
ml.train_rnn(ep, l, a, X_train, Y_train, X_val, Y_val, rnn_model, 230)
```

Рисунок 25 – Обучение *RNN* модели

Результаты работы *RNN* с данными параметрами несколько ниже чем при использовании *CNN*. На рисунках 26 и 27 представлены график обучения и график распределения ошибок на тестовом наборе данных.

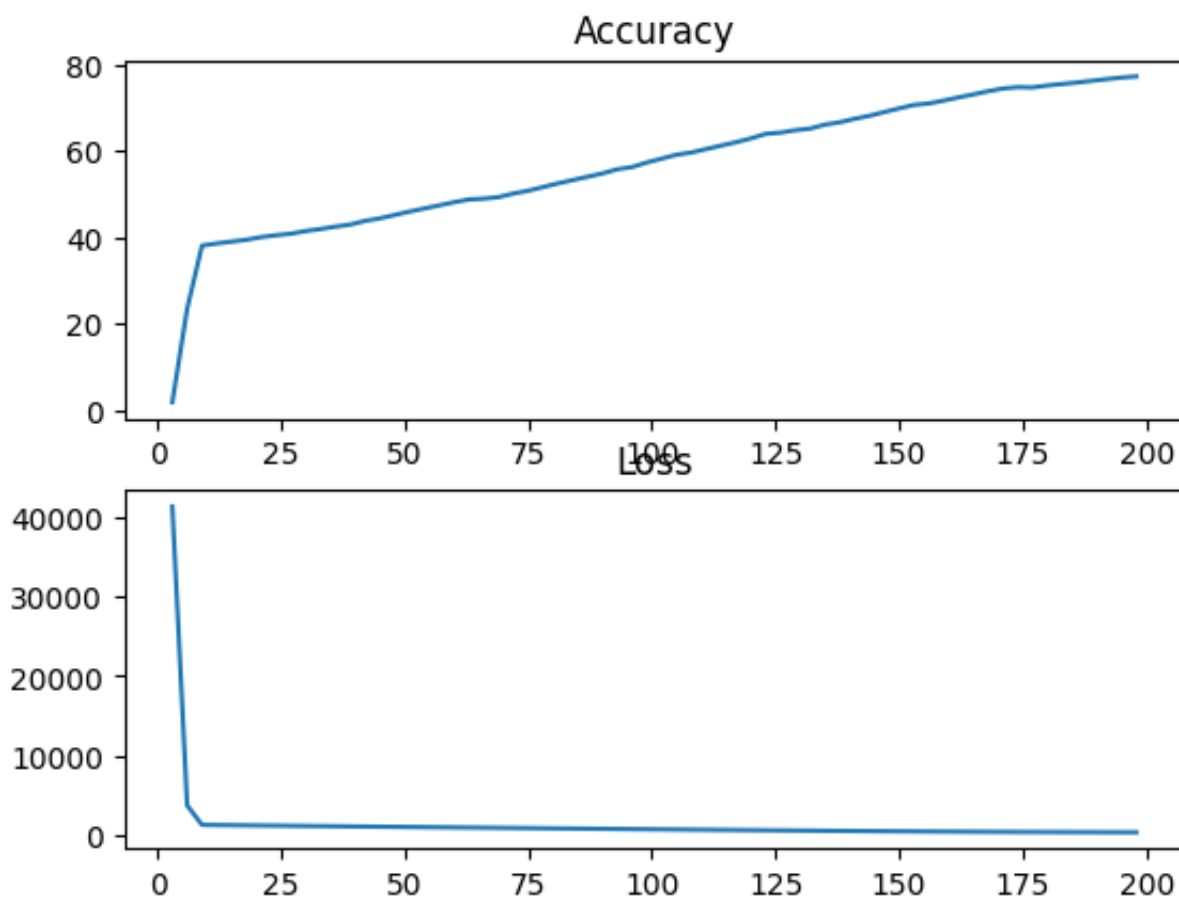


Рисунок 26 – График обучения *RNN*

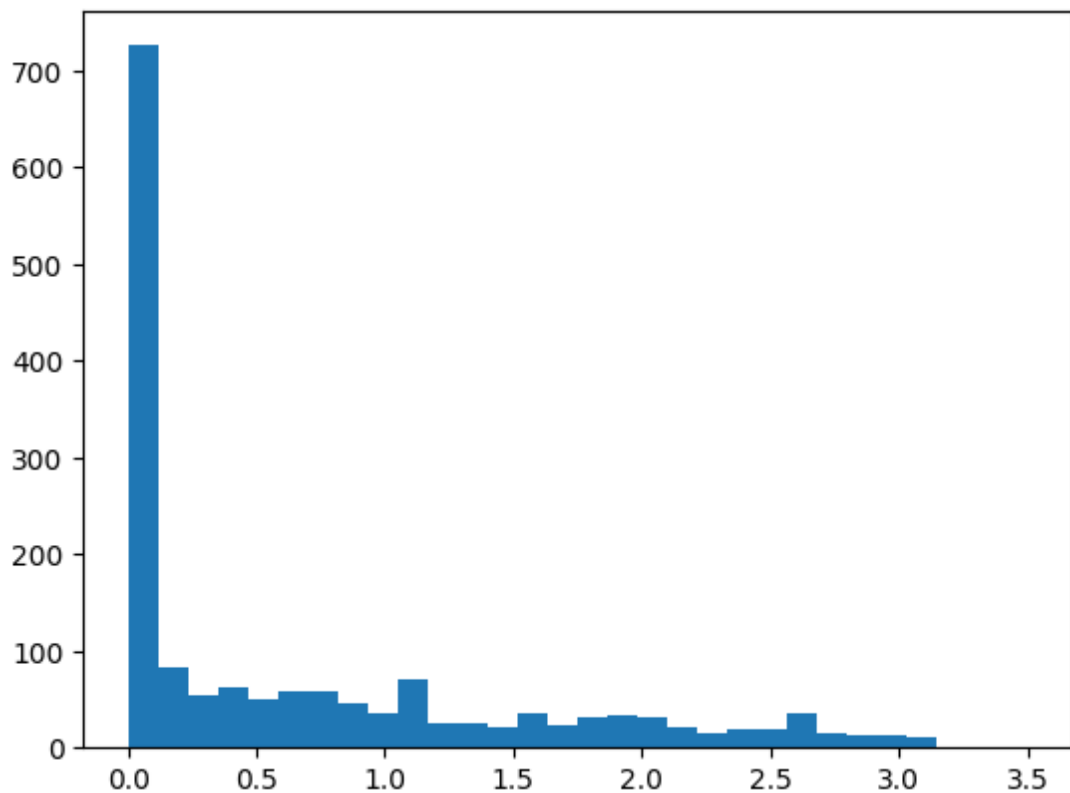


Рисунок 27 – Гистограмма ошибок для *RNN*

Параметры распределения:

- *RMSE*: 1.434
- *mean*: 0.731
- *std*: 1.234

Средняя ошибка достаточно велика за счет длинного хвоста распределения, как и в случае с *CNN*.

Исправить ситуацию можно попытаться при использовании *LSTM* и *GRU*, а также с дополнительной настройкой метапараметров нейронной сети (числа внутренних состояний и количества внутренних слоёв).

Попробуем применить *LSTM*, её реализация несколько проще, поскольку не требуется дополнительная начальная инициализация внутреннего состояния, но, тем не менее, требует регистрации отдельного класса, поскольку *LSTM*-слой, как и *RNN*-слой, имеет два возвращаемых параметра. Метапараметры оставим те же самые, что и в случае обучения *RNN*.

```

from torch.autograd import Variable
class LSTMModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(LSTMModel, self).__init__()

        self.hidden_dim = hidden_dim

        self.layer_dim = layer_dim

        self.lstm = nn.LSTM(input_dim, hidden_dim, layer_dim, batch_first=True)

        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        out, _ = self.lstm(x)
        out = self.fc(out[:, -1, :])
        return out

# обучение LSTM
ep = []
l = []
a = []
rnn_model = LSTMModel(4, 2, 10, 1)
ml.train_rnn(ep, l, a, X_train, Y_train, X_val, Y_val, rnn_model, 200)

```

Рисунок 28 – Пример реализации *LSTM* модели

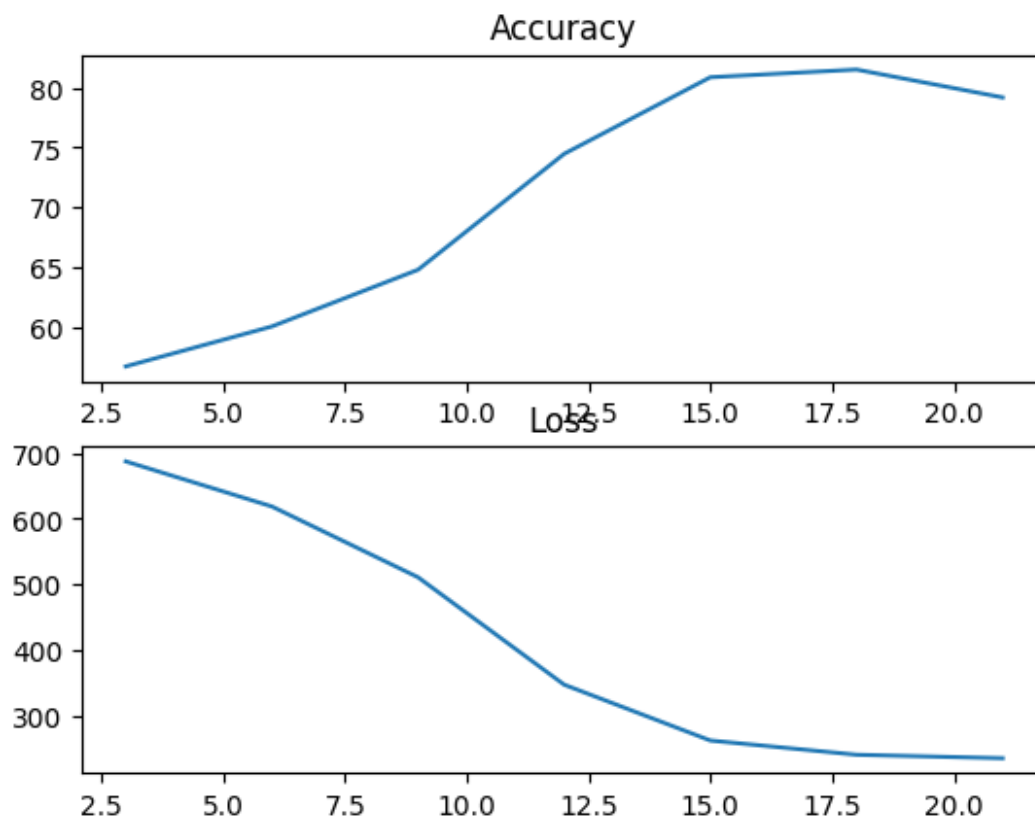


Рисунок 29 – Графики обучения *LSTM* модели

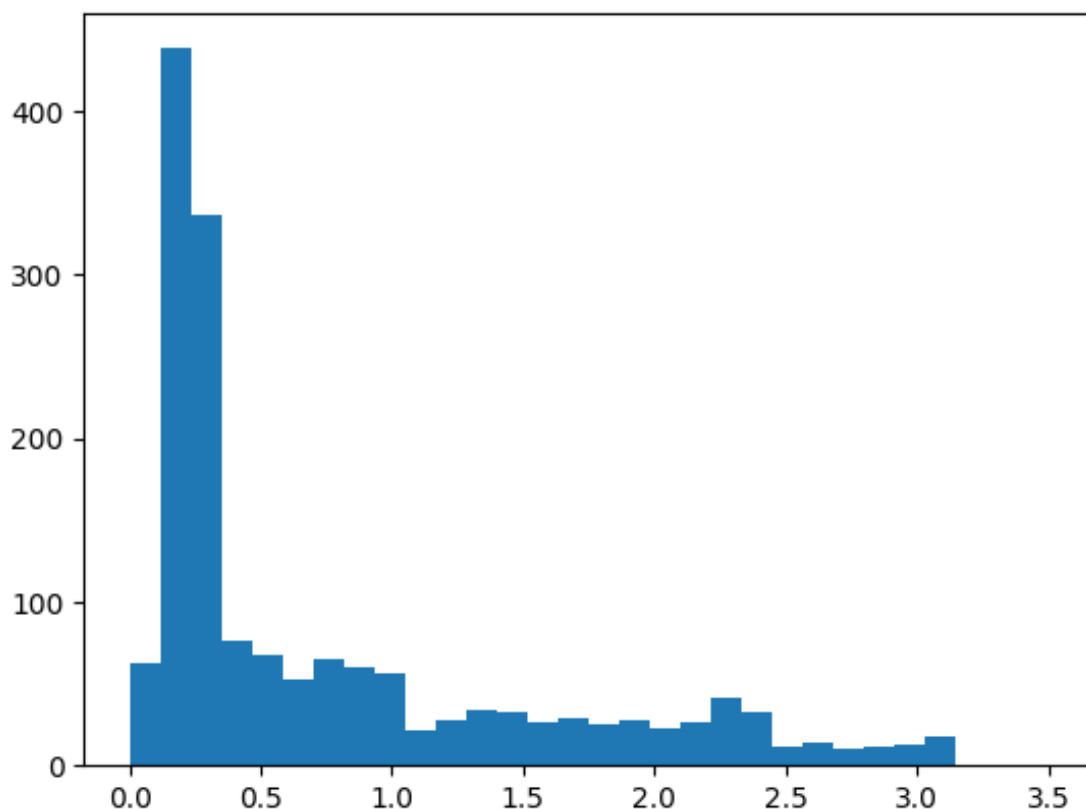


Рисунок 30 – Гистограмма распределения ошибок (*LSTM*)

Параметры распределения:

– *RMSE*: 1.17,

– *mean*: 0.779,

– *std*: 0.787.

Можно заметить, что мы достаточно рано смогли добиться некоторого переобучения модели, улучшили параметры среднеквадратичного отклонения, а также математического ожидания и стандартного отклонения ошибки.

В качестве эксперимента исследуем алгоритм, который будет иметь 2 скрытых состояния и 10 вложенных слоёв.

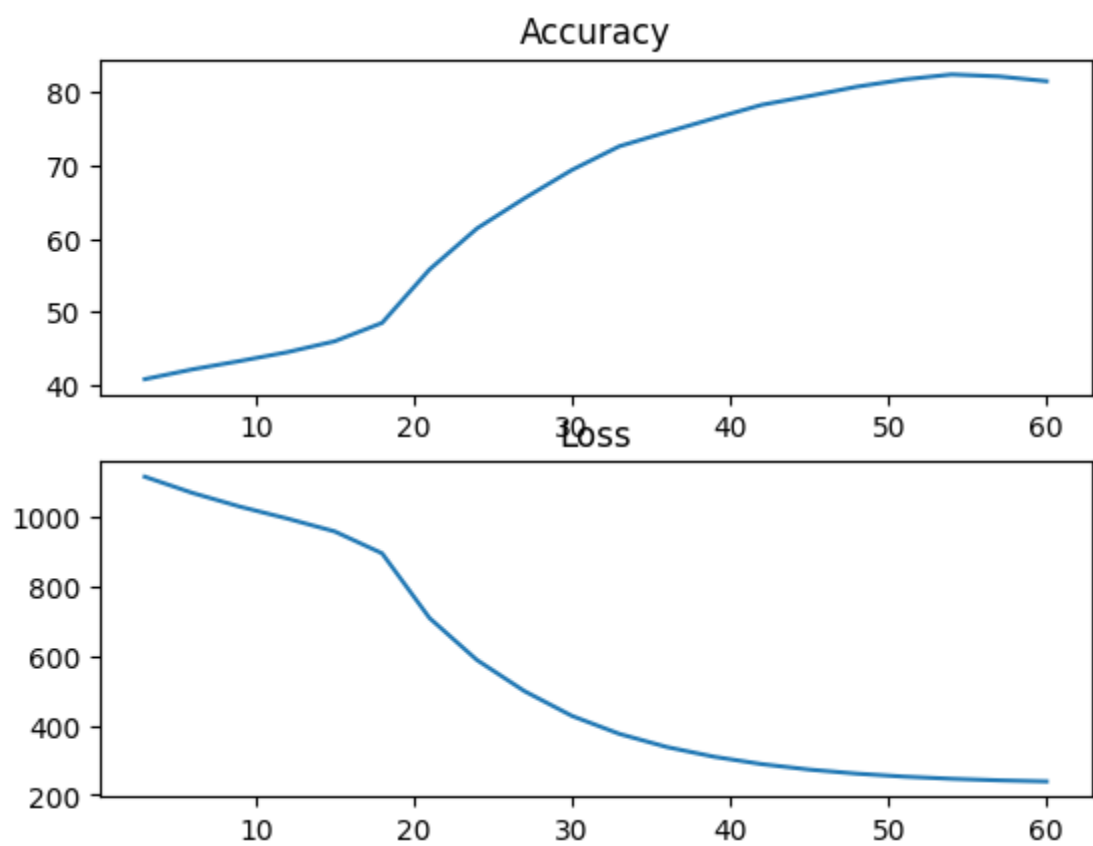


Рисунок 31 – График обучения $LSTM - 2$

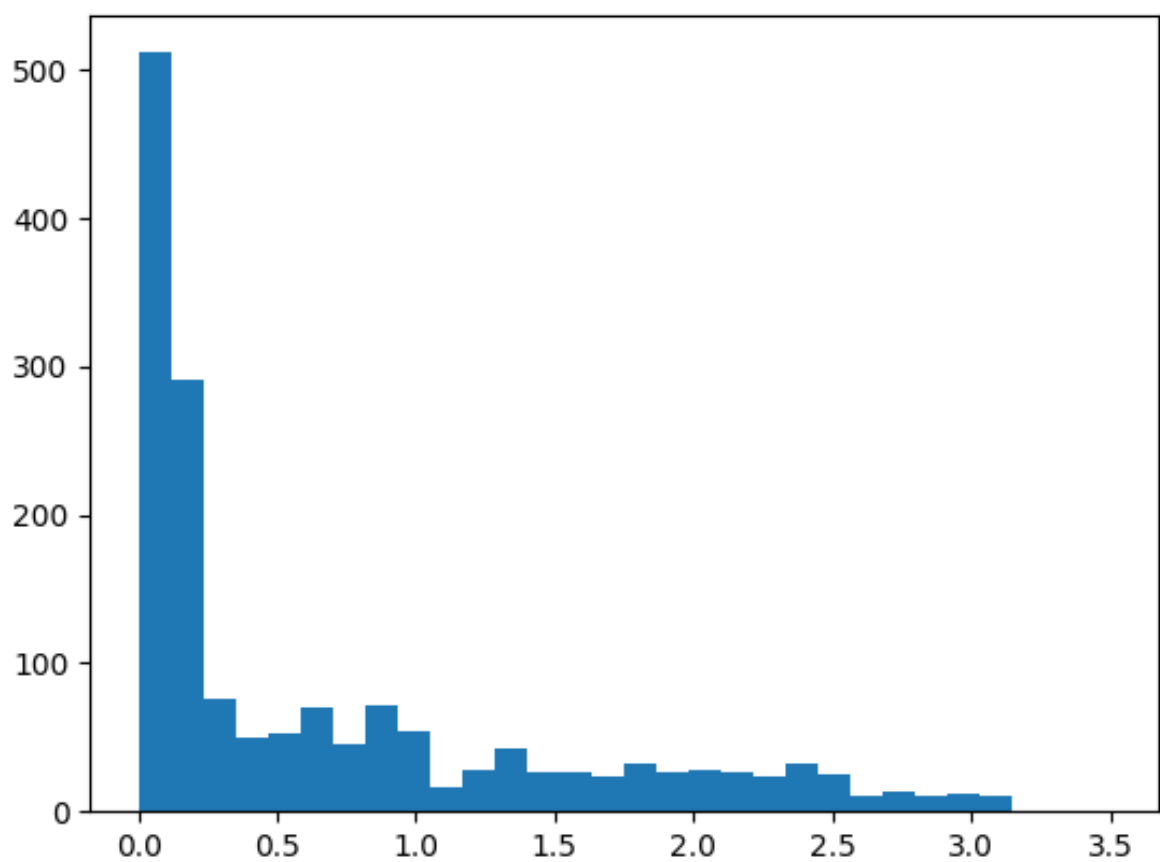


Рисунок 32 – Гистограмма распределения ошибок ($LSTM - 2$)

Параметры распределения:

- *RMSE*: 1.082,
- *mean*: 0.713,
- *std*: 0.814.

RMSE и математическое ожидание несколько улучшились, но также увеличилась плотность хвоста распределения.

Перейдём к алгоритму машинного обучения, основанного на *GRU*. Его инициализация идентична коду алгоритма, использующего *RNN*.

```
from torch.autograd import Variable
class GRUModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(GRUModel, self).__init__()

        self.hidden_dim = hidden_dim

        self.layer_dim = layer_dim
        self.gru = nn.GRU(input_dim, hidden_dim, layer_dim, batch_first=True)

        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        device = torch.cuda.current_device()
        h0 = Variable(torch.zeros(self.layer_dim, x.size(0), self.hidden_dim)).to(device)

        # One time step
        out, hn = self.gru(x, h0)
        out = self.fc(out[:, -1, :])
        return out

# обучение RNN
ep = []
l = []
a = []
rnn_model = GRUModel(4, 2, 2, 1)
ml.train_rnn(ep, l, a, X_train, Y_train, X_val, Y_val, rnn_model, 200)
```

Рисунок 33 – Инициализация *GRU* модели

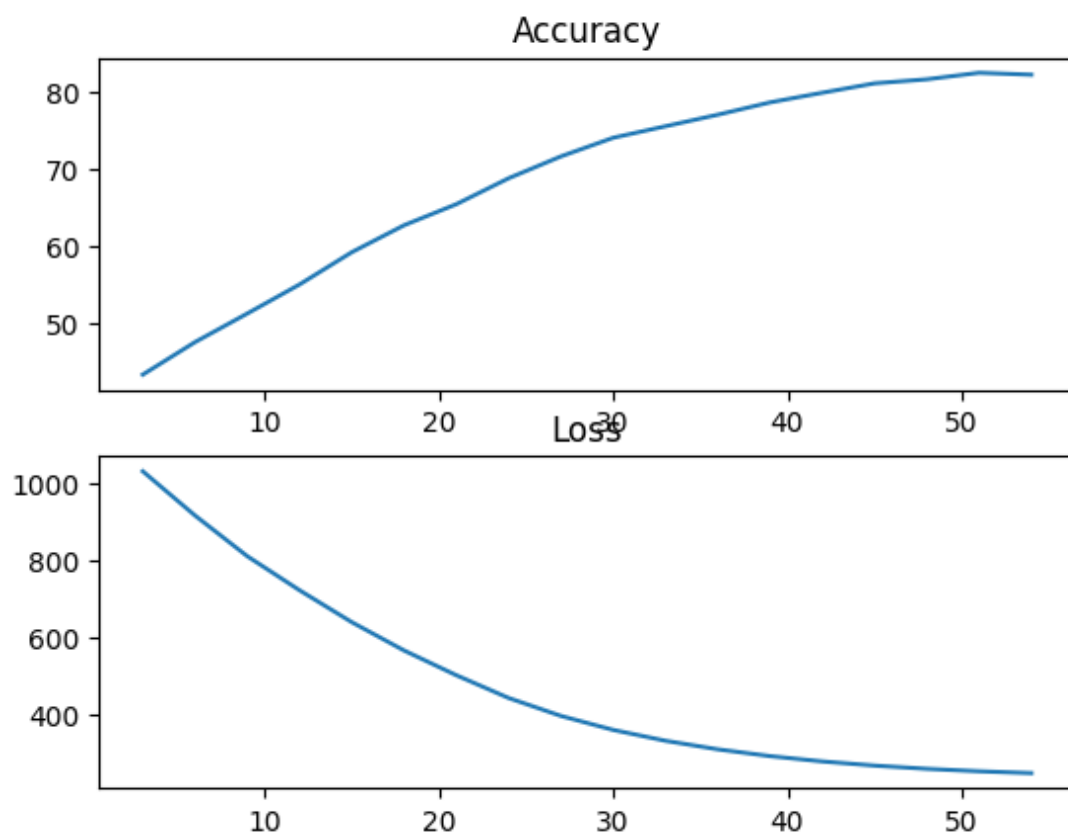


Рисунок 34 – График обучения *GRU* модели

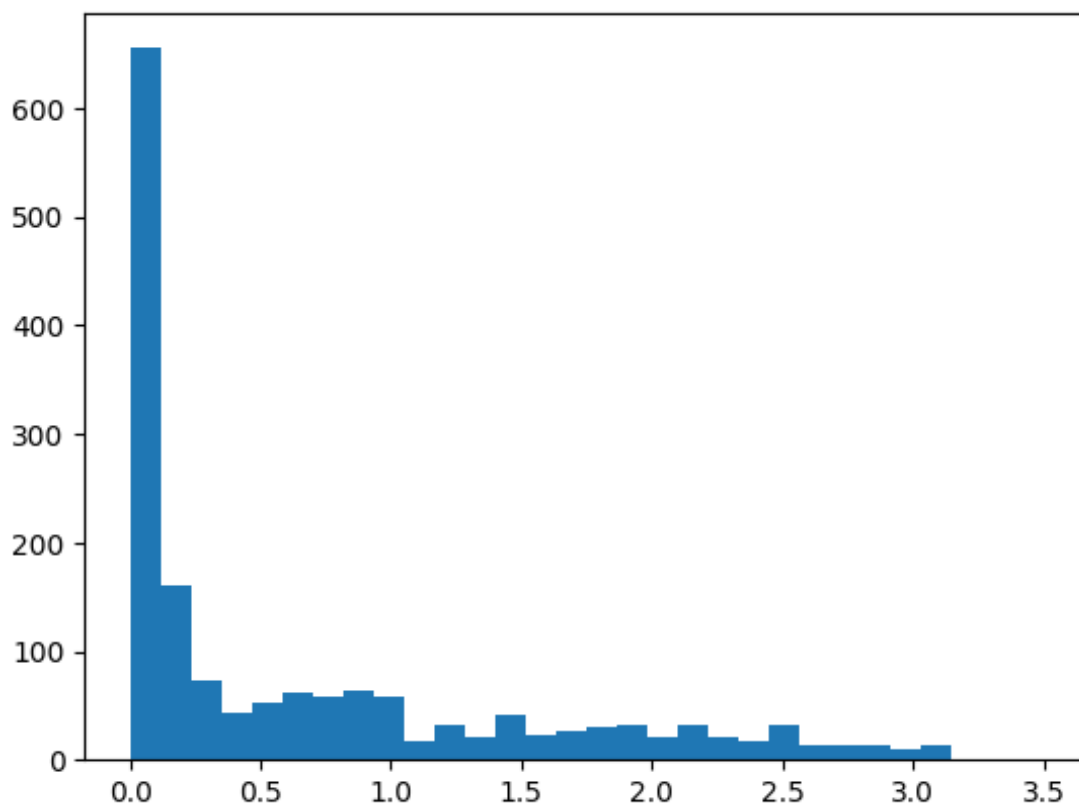


Рисунок 34 – Гистограмма распределения ошибок *GRU* модели

Параметры распределения:

– *RMSE*: 1.077,

– *mean*: 0.699,

– *std*: 0.82.

GRU модель обучается несколько быстрее, а также демонстрирует лучшие результаты *RMSE* и математического ожидания. Стандартное отклонение незначительно увеличилось.

2.5 Анализ результатов обучения сетей

В данной работе были проанализированы различные архитектуры рекуррентных нейронных сетей. В таблице 2 предоставлен сводный результат обучения различных архитектур.

Таблица 2 – Результаты обучения различных алгоритмов

Архитектура	<i>RMSE</i>	<i>m(err)</i>	<i>std(err)</i>	Итераций
<i>CNN</i>	0.32	0.261	0.375	135
<i>RNN</i>	1.434	0.731	1.234	200
<i>LSTM-1</i>	1.17	0.779	0.787	20
<i>LSTM-2</i>	1.082	0.713	0.814	60
<i>GRU</i>	1.077	0.699	0.82	55

Исходя из полученных результатов, мы можем сделать вывод, что наиболее подходящей из рекуррентных сетей для решения текущей задачи является *GRU* архитектура, а наиболее удачной и точной моделью.

3 Проектирование программной системы

Построение среды для функционирования программного обеспечения – комплексная задача, нацеленная на решение большого количества вопросов, основные из которых:

- достаточность вычислительной мощности для решения программным обеспечением задачи в условиях различных ограничений (в англоязычной литературе часто применяется термин *SLA*),
- простота и прозрачность обслуживания инфраструктуры,
- минимизация финансовых затрат при условии соблюдения *SLA*.

При более гибких *SLA* мы можем добиться более простого обслуживания аппаратно-программного комплекса с минимальными финансовыми затратами благодаря паттерну *Web*-сервисов (клиент-серверная архитектура), основные вычисления которого приходятся на одну машину или группу машин (возможно, виртуальных), а сбор и передача данных поручается встраиваемым системам, имеющим кратно меньшую вычислительную мощность.

Однако, такая архитектура при большом количестве клиентов и сетевых задержках может привести к возникновению очереди на обработку данных, длительный разбор которой приводит к неактуальности ответа от сервера, поскольку реальная картина изменилась. Более жесткий *SLA* делает данное решение неподходящим.

Эту проблему можно решить использованием независимых клиентов – это встраиваемые системы, достаточно мощные для работы полной или усечённой версии нейронной сети. Данные клиенты решают задачу самостоятельно, а по сети передают уже готовый результат. Данное решение устойчиво к сбоям, предоставляет более репрезентативную информацию, но вместе с тем и более дорогостоящее, поскольку каждый отдельный участок, подлежащий анализу, должен иметь свой вычислительный центр.

3.1 Требования, предъявляемые к программной системе

В рамках разработки полноценного программного продукта наличие модели машинного обучения недостаточно. Остается большое количество задач, основные из которых:

- обеспечение доступа к функциональности обученной нейронной сети,
- обеспечение аутентификации и авторизации клиентов,
- сбор и хранение записей журнала,
- развёртывание программной системы в определённом окружении.

Перечисленные выше задачи решаются с помощью проектирования ПО и выбора соответствующих инструментов. Должны быть спроектированы требуемые интерфейсы взаимодействия, способы хранения и доступа к информации, причем основное внимание необходимо сфокусировать на применение современных и уже зарекомендовавших себя решений и методологий с открытыми спецификациями исходным кодом.

3.2 Проектирование

Описываемая программная система состоит из трёх основных компонентов

- клиентская система (клиент-источник), генерирующая непрерывный поток данных и отправляющая их в мастер-систему для обработки;
- мастер-система (сервер), содержащая в себе несколько подсистем (в том числе модель машинного обучения);
- клиентская система (клиент-инспектор), опрашивающая мастер-систему для извлечения определённой информации;

В данном случае мы имеем дело с несколькими клиентскими системами, которым должны быть предоставлены разные услуги. Клиенту источнику информации должны быть предоставлены функции для:

- проверки работоспособности сервера,
- отправки набора сигналов для последующего анализа мастер-системой.

В свою очередь, клиент-инспектор должен иметь доступ к услугам по:

- определению источника шума, основанной на данных из клиента источника, в текущий и прошедшие моменты времени,
- контролю качеством обработки данных нейронной сетью (поскольку мы имеем дело с системой машинного обучения с учителем, данный клиент должен уметь в определённых рамках влиять на процесс принятия решения нейронной сетью).

Существует множество способов взаимодействия между процессами, которые так или иначе основаны на сетевом взаимодействии [7, 8]. Выбор заключается между выборе протоколов, основанных на дейтаграммах, либо на протоколах, основанных на каналах (т.е. протоколы без установки соединения и с установкой соединения).

В проектируемой система важна гарантированная доставка как управляющих сообщений, так и сообщений с данными, поэтому мы склоняемся к выбору протоколов, основанных на соединениях, наиболее распространенным в данное время является протокол *HTTP(S)*, который работает поверх протокола *TCP*, гарантирующего как доставку, так и целостность.

Использование протокола *HTTP* в настоящее время подразумевает использование *REST* методологии проектирования интерфейсов, в котором используется *JSON* нотация при сериализации объектов.

Однако применение протокола *HTTP* будет нести большое количество накладных расходов при использовании его в случае отправки сырых данных клиентом, поскольку данные генерируются непрерывно, а *HTTP* протокол подразумевает модель взаимодействия «запрос – ответ». В данном случае мы можем использовать либо протокол *TCP* непосредственно, либо его легковесную версию на прикладном уровне, *WebSocket*. Применение одной из указанных технологий позволит значительно сократить издержки при передаче сырых данных от клиента к серверу.

Взаимодействие клиента-инспектора должно быть авторизованным. Для этого можно применить стандартный механизм *JWT* с использованием одной из моделей управления доступом. Наиболее простой моделью является *RBA* (авторизация, основанная на ролевой модели). Данная модель по управлению доступом открывает для использования ресурсы на основе роли субъекта, что позволит нам быстро и просто вносить функционал с учётом политик безопасности. В настоящее время существуют и иные модели доступа, но, поскольку у нас нет большого количества методов, то их применение не влечет какой-либо пользы.

Существует два подхода к реализации *JWT* аутентификации – отзывной и неотзывной *access* токен. В первом случае нам необходимо дополнительно иметь хранилище с выданными токенами, а при попытке доступа к защищаемому ресурсу проверять этот токен не только на правильность подписи, но и на существование его во внешнем хранилище. При выходе пользователя необходимо удалять токен из хранилища. В данном этапе будем ориентироваться на неотзывной вариант реализации, поскольку он быстрее и проще, но теоритически чуть более уязвим, особенно при условии долгоживущих *access* токенах. С другой стороны, не будет никаких технических препятствий для реорганизации к отзывному варианту.

На данном этапе можно выделить две роли:

- *auditor* – роль открывает доступ к услуге по определению источника шума,

- *administrator* – роль открывает доступ ко всем возможным услугам.

Первичными данными для аутентификации могут выступать логин и пароль.

В таблице 3 представлены методы для взаимодействия с сервером. Методы для получения токена и проверки работоспособности сервера выполняются без авторизации.

Таблица 3 – REST методы взаимодействия клиента и сервера

URL	Метод	Модели	Описание
<i>/auth/login</i>	<i>POST</i>	<i>AuthReq</i> , <i>AuthResp</i>	Метод для получения пары <i>access/refresh</i> токенов
<i>/auth/logout</i>	<i>POST</i>	-	Метод для отзыва refresh токена
<i>/ping</i>	<i>GET</i>	-	Метод для проверки работоспособности сервера
<i>/audit/locate</i>	<i>GET</i>	<i>LocateReq</i> , <i>LocateResp</i>	Метод для локализации источника шума
<i>/nn/correct</i>	<i>POST</i>	<i>CorrectNnReq</i> , -	Метод для дообучения нейронной сети

Далее описаны используемые объекты передачи данных (*data transfer object, DTO*):

AuthReq:

- *login: string*
- *passwordHash: string*

AuthResp:

- *accessToken: string*
- *refreshToken: string*
- *exp: integer*

LocateReq:

- *time: datetime*

LocateResp:

- *lat: float64*
- *long: float64*

CorrectNnReq:

- *data: []byte*
- *lat: float64*
- *long: float64*

Ведение журнала является неотъемлемой частью работы сервера, поскольку именно записи журнала позволяют узнать информацию в прошлом, обнаружить ошибки и восстановить последовательность действий системы. Ввиду важности подсистемы ведения журнала рассмотрим несколько методов.

Наиболее популярным методом логгирования является запись данных в один из потоков вывода: *stdout* либо определённый файл в файловой системе. Данный подход быстро реализуется, но он ненадёжен ввиду того, что при неожиданном сбое в работе ОС буферизованные данные могут не успеть записаться на диск, доступ к журналу не стандартизирован (файл может находиться в любой директории) и т.д.

Другим подходом является запись журнала в специальную систему (в Linux она называется *syslog* и *rsyslog*) [9]. Взаимодействие с данной подсистемой происходит посредством использования сокетов (*Unix* либо стандартных). Это позволяет запускать процесс на одной машине, в то время как хранение журнала будет вестись на другой. Подсистема *rsyslog* является более гибкой и надёжной, но для автоматического анализа журналов записей будет требоваться дополнительная работа по разбору этих данных.

Наиболее надёжный и расширяемый вариант – запись сообщений журнала в отдельно выделенную для этого базу данных. В этом случае мы можем:

- убедиться, что запись сохранена (посредством гарантий СУБД),
- структурировать записи для последующего анализа.

Отметим, что запись в СУБД может потенциально стать узким местом ввиду того, что для выполнения данной операции требуется сетевое взаимодействие (хотя СУБД может располагаться на той же вычислительной машине, что и сервер, тем не менее требуется межпроцессное взаимодействие,

которое зачастую задействует сетевой стек операционной системы для обеспечения переносимости между машинами).

Исходя из вышеописанного, мы можем прийти к комбинированной схеме ведения журнала. Оперативные и малозначимые данные (результаты запросов, предупреждения о потенциальной ошибке) могут записываться при помощи записи в файл или *rsyslog*, в то время как записи журнала аудита (аутентификация и авторизация пользователей) должны быть синхронно записаны в СУБД.

Публикация программного обеспечения является комплексной задачей, решение которой позволяет эксплуатировать систему без существенных затрат и переносить программное обеспечение на различные дистрибутивы и вычислительные комплексы.

Наиболее популярным решением для разделения ресурсов и обеспечения переносимости ПО является контейнеризация[10]. Контейнеризация – это виртуализация, в которой виртуальные узлы представляют из себя контейнеры с общим ядром, но разными файловыми системами и пространствами (*namespaces*). Технология имеет наименьшие накладные расходы среди систем виртуализации, поэтому следует применить именно её. Основным инструментом для применения контейнеризации является Docker.

Отличительной особенностью является то, что *Docker* имеет встроенную легковесную систему управления контейнерами (оркестрация) – *Docker compose*. Оркестратор – это система, позволяющая автоматически управлять состоянием множества контейнеров, периодически проверяя их работоспособность, и, в случае обнаружения сбоев, перезапускать их, или же другим способом решать пытаться решить неисправности. *Docker compose* описывает систему декларативно через файл конфигурации в *yaml* формате.

Таким образом, применение *Docker* в данном проекте позволит решить задачу публикации программного обеспечения. На рисунке 35 представлена схема работы с контейнеризованными приложениями. Пользователь из

внешней среды по протоколу *ssh* подключается к вычислительной машине (на которой установлен один из дистрибутивов *Linux*), и с использованием удалённого терминала может управлять состоянием системы: перезапускать или добавлять экземпляры контейнеров и приложений, просматривать журнал активностей как системы, так и пользователей, обновлять образы программного обеспечения.

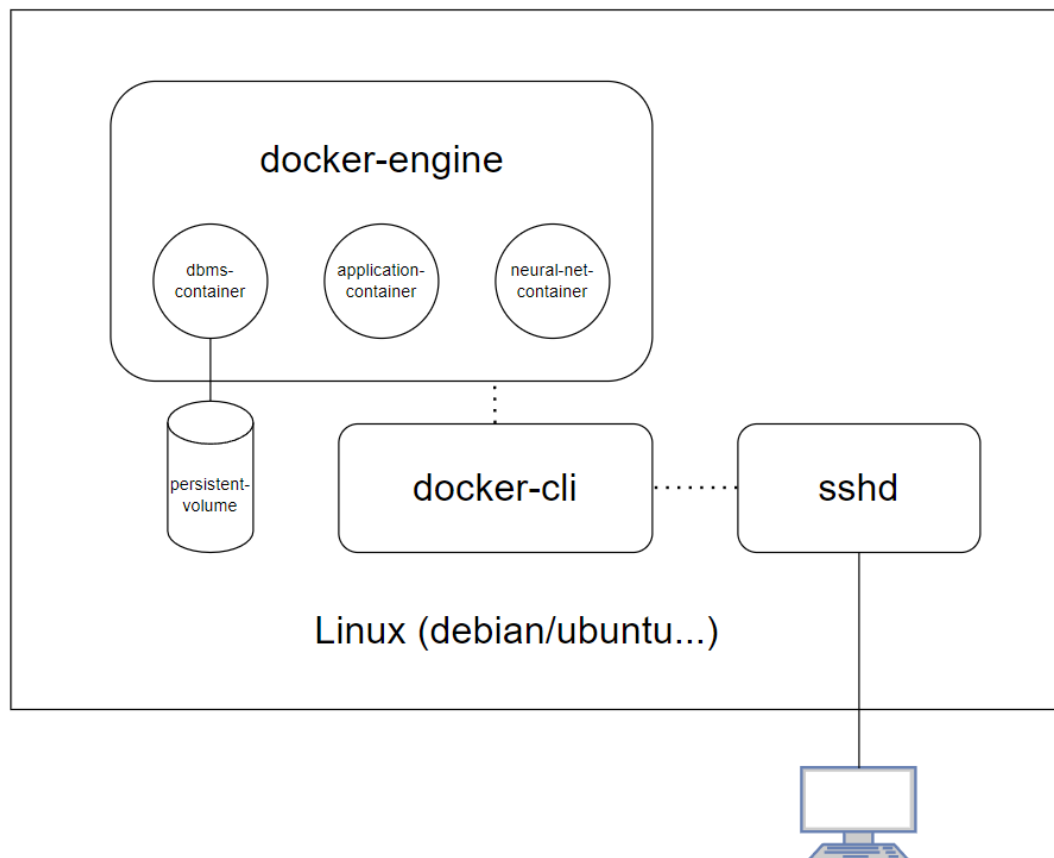


Рисунок 35 – Инфраструктурная схема программной системы

В итоге мы можем получить систему мониторинга объекта или участка, представленную на рисунке 36. Зона наблюдения оборудована микроЭВМ с сейсмографами и средствами доступа к сети Интернет (коммутаторы и маршрутизаторы).

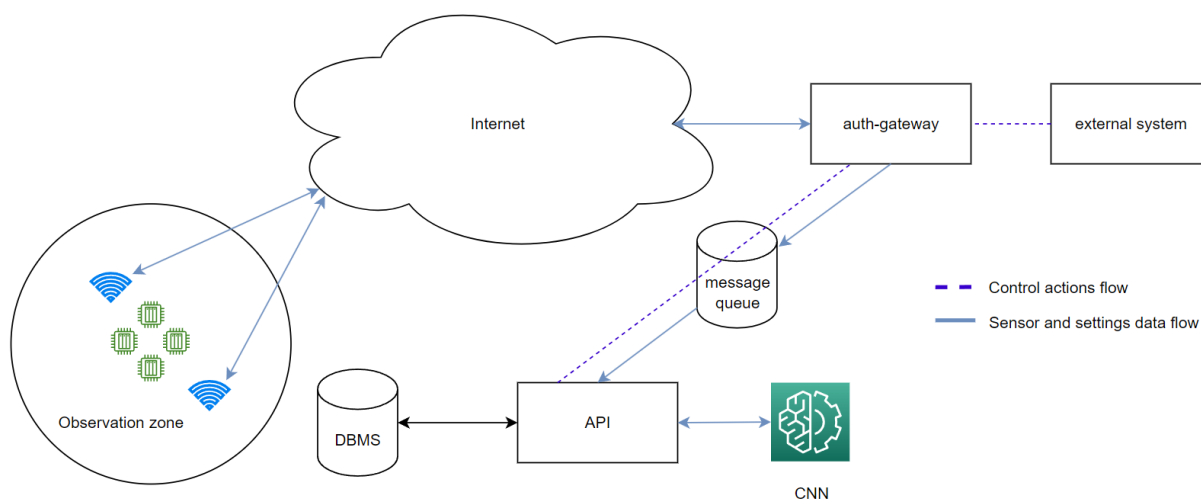


Рисунок 36 – Схема развертывания с применением архитектуры клиент-сервер

МикроЭВМ в режиме реального времени отправляет данные через сеть интернет в конечную точку серверной части системы. Проверив подлинность клиента, данные попадают в очередь сообщений. Очередь сообщений доставляет их до конечной точки (*API*), которая логирует результаты работы в СУБД. Внешняя система осуществляет управляющее воздействие (конфигурацию) на целевую систему и мониторинг зоны наблюдения. За обработку данных отвечает *CNN*.

Отметим, что зон наблюдения может быть большое количество, что может сильно нагрузить узлы *auth-gateway* и *API*. С целью распределения нагрузки и снижения затрат на аппаратные ресурсы мы можем внедрить систему непрерывной доставки сообщений, которая будет распределять нагрузку на несколько серверов *API*.

ЗАКЛЮЧЕНИЕ

В ходе выполнения магистерской диссертации были проанализированы современные архитектуры нейронных сетей, проведены исследования по применению различных архитектур для решения задачи в области обработки сигналов, подобрана и обучена наиболее подходящая архитектура для решения задачи пеленгации, а также было спроектировано типовое программное обеспечение для развёртывания и эксплуатации нейросетевого алгоритма.

За время исследования и проектирования были рассмотрены современные методы, языки программирования и библиотеки для решения задач в области машинного обучения, визуализации различных данных, а также разработке и реализации *WebAPI*, системы управления базами данных и методы виртуализации.

Результаты магистерской диссертации были представлены на конференции *IT OPEN 2023* с темой «Применение нейронных сетей в задачах обработки сигналов».

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Downey A. B. Think DSP. Digital Signal Processing in Python. — Green Tea Press, 2014. — 153 с.
2. Gulli A., Kapoor A., Pal S. Deep Learning with TensorFlow 2 and Keras. — 2 изд. — Packt Publishing, 2019. — 647 с.
3. Вейдман С. Глубокое обучение: легкая разработка проектов на Python. — СПб.: Питер, 2021. — 272 с.
4. Шолле Ф. Глубокое обучение на Python. — СПб.: Питер, 2018. — 400 с.
5. Shubham Saboo, GPT-3: Building Innovative NLP Products Using Large Language Models, O'Reilly, 2022 – 158 с.
6. Будума Н., Локашо Н. Основы глубокого обучения. Создание алгоритмов для искусственного интеллекта следующего поколения — М.: Манн, Иванов и Фербер, 2020. — 304 с.
7. Sam Newman, Building Microservices, 2nd Edition, O'Reilly, 2022 – 280 с.
8. Martin Kleppmann, Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems, O'Reilly, 2017 – 611 с.
9. Robert Love, Linux System Programming: Talking Directly to the Kernel and C Library, O'Reilly, 2013 – 454 с.
10. Harihara Subramanian, Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs, O'Reilly, 2019 – 378 с.

ПРИЛОЖЕНИЕ А

(Обязательное)

Нейросетевая обработка многомерного сигнала

Листинг программы

Файл gpx_read.py

```
import datetime

import gpxpy
import numpy as np

def read_gpx(file_name):
    with open(file_name, 'r') as f:
        data = gpxpy.parse(f)
        return data

def get_points(gpx):
    return gpx.tracks[0].segments[0].points

def as_unix(tm):
    t = tm.replace(tzinfo=None)
    return (t -
datetime.datetime.fromtimestamp(datetime.UTC)).total_seconds()

def get_track_data(file):
    data = read_gpx(file)
    points = get_points(data)

    long_arr = []
    lat_arr = []
    time_arr = []

    for r in points:
        long_arr.append(r.longitude)
        lat_arr.append(r.latitude)
        time_arr.append(as_unix(r.time))

    long_arr = np.array(long_arr)
```

```

lat_arr = np.array(lat_arr)
time_arr = np.array(time_arr)

time_arr = time_arr - np.min(time_arr)

return long_arr, lat_arr, time_arr

```

Файл main.py

```

import gpx_read as gpx
import matplotlib.pyplot as plt

import ml
import signal_reader as mat
import numpy as np
import utils
import persistance

show_track = False
show_signals = False
show_spectrum = False
etl = True

def view_spectrogram(x, samples, nfft):
    plt.specgram(x, Fs=samples, NFFT=nfft, noverlap=nfft//8)
    plt.show()

def explore():
    np.set_printoptions(precision=4, suppress=True)

    long, lat, time_points =
gpx.get_track_data('resources\\18_11_2014 17_24 Man2.gpx')

    zero, track, sensor1_signal, sensor2_signal =
mat.read_mat('resources\\18_11_2014 17_24 Man2.gpx.mat',

```

```

_adc1='man2ADC1', k_adc2='man2ADC2')
    track = track[:, 1:]
    n = sensor1_signal.shape[0]
    sample_rate = 5120

    if etl:
        persistance.etl_dataset()

    if show_track:
        plt.scatter(zero[0, 0][0][0, 0], zero[0, 0][1][0, 0],
c='black', marker='x')
        # plt.scatter(long, lat)
        plt.scatter(track[:, 0], track[:, 1])
        # plt.plot(long, lat)

        plt.show()

    # генерация пространства времен, по мощности соответствующая
    мощности множества амплитуд сигналов

    # т.е. количество точек времени должно быть таким же как кол
    во точек амплитуд

    time_space = utils.time_linspace(time_points, n)

    # _, axis = plt.subplots(4, 1)

    c = ['r', 'g', 'b', 'm']
    markers = ['-', '-', '-', '-']

    # sensor1_signal, time_space =
    utils.get_every(sensor1_signal, time_space, 2)

    # sample_rate = int(sensor1_signal.shape[0] /
    np.max(time_points))

    if show_signals:

```

```

        for i in range(0, 4):
            # axis[i].plot(time, sensor1_signal[:, i], c[i])
            plt.plot(time_space, utils.center(sensor1_signal[:,
i]), c[i] + markers[i], linewidth=1)

        # for t in range(1000):
        #     plt.axvline(x=t*0.5, c='g')

    plt.show()

    if show_spectrum:
        for i in range(0, 4):
            sample = utils.get_sample(sensor1_signal[:, i],
sample_rate, 150, 550)
            view_spectrogram(sample, sample_rate, nfft=512)

        # разделим на семплы/координаты
        _, Y = utils.split_per_samples(track, time_points,
sensor1_signal, sample_rate, duration=0.5, polar=False)

    if show_track:
        plt.scatter(track[:, 0], track[:, 1], s=8)
        plt.scatter(0, 0, c='black', marker='x')
        plt.scatter(Y[:, 0], Y[:, 1], c='r', s=8)

    plt.show()

    return

def main():
    explore()

if __name__ == '__main__':
    main()

```

Файл ml.py

```
import math

import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt

# 1 is same
# 0 otherwise
def metric(p, t):
    x = rad_diff(p, t)
    return 1 - np.tanh(x / torch.pi)

def rad_diff(p, t):
    if p is torch.Tensor:
        diff = torch.abs(p - t).item()
    else:
        diff = np.abs(p - t)
    if diff > torch.pi:
        diff -= torch.pi
    return diff

def acc(pred, yt):
    correct, total = 0, 0
    for p, t in zip(pred, yt):
        correct += metric(p, t)
        total += 1
    return correct, total

def check(i, ep, l, a, model, device, criterion, Xx, Yy,
log=True):
```

```

with torch.no_grad():
    loss, correct, total = 0, 0, 0

    for xt, yt in zip(Xx, Yy):
        xt = xt.to(device)
        yt = yt.to(device)
        pred = model(xt)
        pred = pred.to('cpu')
        yt = yt.to('cpu')
        loss += criterion(pred, yt).item()

        c, t = acc(pred, yt)
        correct += c
        total += t

    accuracy = 100 * correct / total
    if log:
        ep.append(i)
        l.append(loss)
        a.append(accuracy)
    print(f'Iteration: {i}. Loss: {loss}. Accuracy:
{accuracy}')

    return accuracy >= 93.5

def check_rmse(model, device, Xx, Yy):
    with torch.no_grad():
        loss, correct, total = 0, 0, 0
        losses = np.zeros(shape=len(Xx))
        index = 0
        abnormal = 0

        for xt, yt in zip(Xx, Yy):
            xt = xt.to(device)

```

```

        yt = yt.to(device)
        pred = model(xt)
        pred = pred.to('cpu')
        yt = yt.to('cpu')

        temp = rad_diff(pred, yt)
        losses[index] = temp
        loss += temp ** 2
        if temp > 0.8 :
            abnormal += 1
        index += 1

    return math.sqrt(loss / len(Xx)), abnormal, losses

def train(ep, l, a, X_train, Y_train, X_val, Y_val, nn,
iterations):
    nn = nn.cuda()
    iters = 0

    device = torch.cuda.current_device()
    optimizer = torch.optim.Adam(nn.parameters())
    criterion = torch.nn.SmoothL1Loss()

    for epoch in range(iterations):
        perm = torch.randperm(X_train.shape[0])
        X_train = X_train[perm]
        Y_train = Y_train[perm]

        for x, y in zip(X_train, Y_train):
            x = x.to(device)
            y = y.to(device)
            optimizer.zero_grad()

            outputs = nn(x)

```

```

        loss = criterion(outputs, y)
        loss.backward()

        optimizer.step()
        iters += 1

        if iters % 3 == 0:
            if check(iters, ep, l, a, nn, device, criterion,
X_val, Y_val):
                break

        check(-1, ep, l, a, nn, device, criterion, X_val, Y_val,
log=False)

    return

def train_rnn(ep, l, a, X_train, Y_train, X_val, Y_val, nn,
iterations):
    nn = nn.cuda()
    iters = 0

    device = torch.cuda.current_device()
    optimizer = torch.optim.Adam(nn.parameters())
    criterion = torch.nn.SmoothL1Loss()

    for epoch in range(iterations):
        perm = torch.randperm(X_train.shape[0])
        X_train = X_train[perm]
        Y_train = Y_train[perm]

        for x, y in zip(X_train, Y_train):
            x = x.to(device)
            y = y.to(device)
            optimizer.zero_grad()

```

```

        outputs = nn(x)

        loss = criterion(outputs, y)
        loss.backward()

        optimizer.step()
        iters += 1

        if iters % 3 == 0:
            if check(iters, ep, l, a, nn, device, criterion,
X_val, Y_val):
                break

        check(-1, ep, l, a, nn, device, criterion, X_val, Y_val,
log=False)
    return

```

Файл persistence.py

```

import os
import re

import gpx_read as gpx
import signal_reader as matlab
import numpy as np
import utils
import torch

def etl_dataset():
    all_track_files = ['resources\\18_11_2014 17_05 Ман1.gpx',
                        'resources\\18_11_2014 17_24 Ман2.gpx',
                        'resources\\18_11_2014 17_47 ман3.gpx'
                        ]

```

```

    all_signal_files = ['resources\\18_11_2014 17_05
Ман1.gpx.mat',
                        'resources\\18_11_2014 17_24
Ман2.gpx.mat',
                        'resources\\18_11_2014 17_47
Ман3.gpx.mat']

    adc = [('man1ADC1', 'man1ADC2'), ('man2ADC1', 'man2ADC2'),
            ('man3ADC1', 'man3ADC2')]

    i = 0

    for t_file, s_file in zip(all_track_files,
all_signal_files):
        long, lat, time_points = gpx.get_track_data(t_file)
        zero, track, sensor1_signal, sensor2_signal =
matlab.read_mat(s_file, k_adc1=adc[i][0], k_adc2=adc[i][1])
        track = track[:, 1:]
        sample_rate = 5120

        X, Y = utils.split(track, time_points, sensor1_signal,
sample_rate)

        # [смпл, канал, амплитуда]
        X = torch.tensor(X).float().permute((0, 2, 1))
        Y = torch.tensor(Y).float()

        torch.save(X, f'dataset/X_{i}.pt')
        torch.save(Y, f'dataset/Y_{i}.pt')

        i += 1

def get_dataset():
    files = os.listdir('dataset')
    dct = {}
    for d in files:

```

```

        part, index =
re.compile('(\w)_ (\d)\.pt').match(d).groups()
        index = int(index)
        if dct.get(index) is None:
            dct[index] = {part: torch.load(f'dataset/{d}')}
        else:
            dct[index][part] = torch.load(f'dataset/{d}')

x = []
y = []
for d in dct:
    x.append(dct[d]['X'])
    y.append(dct[d]['Y'][:, 1])

x = torch.cat(x)
y = torch.cat(y)

return x, y

```

Файл signal_read.py

```

import scipy

def read_mat(file, k_zero='point_zero10', k_coord='data_t_X_Y',
k_adc1='man1ADC1', k_adc2='man1ADC2'):
    mat_file = scipy.io.loadmat(file)
    if k_coord not in mat_file.keys():
        k_coord = 'data_t_X_Y_1'
    return mat_file.get(k_zero), mat_file.get(k_coord),
mat_file.get(k_adc1), mat_file.get(k_adc2)

```

Файл utils.py

```

import numpy as np

def cartesian2polar(x, y, degree=False):

```

```

    rho = np.sqrt(x ** 2 + y ** 2)
    phi = np.arctan(y / x)
    if not(np.isclose(x, rho * np.cos(phi)) and np.isclose(y,
rho * np.sin(phi))):
        phi += np.pi
    # приводим исключительно в положительную область
    if phi < 0:
        phi += 2 * np.pi
    if degree:
        phi = phi / (2 * np.pi) * 360
    return rho, phi

def polar2cartesian(rho, phi):
    return

def center(signal):
    signal = signal - np.average(signal)
    return signal

def time_linspace(time_arr, count):
    max_time = time_arr.max()
    return np.linspace(0, max_time, count)

def get_sample(x, sample_rate, start, end):
    return x[int(start * sample_rate):int(end * sample_rate)]

def l1_distance(p2, p1):
    return np.subtract(p2, p1)

def l2_distance(p1, p2):
    return np.sqrt(np.power(p1[0] - p2[0], 2) + np.power(p1[1] -
p2[1], 2))

def approximate_coord_at_time(points, time_points, target_time):

```

```

    left_index = 0
    print(target_time)

    # ищем, между какими точками времени попадаем
    for prev, nxt in zip(time_points[:len(time_points) - 1],
time_points[1:]):
        if prev < target_time < nxt:
            break
        left_index += 1

    right_index = left_index + 1

    # точки, между которыми требуется аппроксимация
    left = points[left_index]
    right = left
    if right_index < len(points):
        right = points[right_index]
    else:
        return None

    # расстояние между точками
    dist = l2_distance(left, right)

    # берём коэффициент процент заполнения временного отрезка
    # 0 - наша точка точно совпадает с left
    # 1 - наша точка точно совпадает с right
    time_prop = (target_time - time_points[left_index]) /
(time_points[right_index] - time_points[left_index])
    delta = time_prop * l1_distance(right, left)

    result = left + (delta * time_prop)

    return result

def get_every(signal, time, every):
    return signal[0:-1:every, :], time[0:-1:every]

```

```

def split_per_samples(track, time, signal, sample_rate=5120,
duration=2, delta=0.2, polar=True):

    l, r = 0, duration

    points_per_sample = int(duration * sample_rate)

    total_samples = int((np.max(time) - duration - delta) /
delta)

    X = np.zeros(shape=(total_samples, int(points_per_sample),
4))
    Y = np.ones(shape=(total_samples, 2))

    for i in range(total_samples - 1):
        try:
            # берём сэмпл
            s = int(sample_rate * (delta * i))
            e = s + points_per_sample
            x = signal[s:e, :]

            target_time = l + duration / 2
            y = approximate_coord_at_time(track, time,
target_time)
            if polar:
                y = cartesian2polar(*y)

            X[i] = x
            Y[i] = y

            l += delta
            r += delta
        except:
            print('hehe')

    return X, Y

```

```

def split(track, time, signal, rate, sample_size=5120,
sample_delta=1280, polar=True):
    l, r = 0, sample_size

    total_samples = int((signal.shape[0] - sample_size -
sample_delta) / sample_delta)

    X = np.zeros(shape=(total_samples, sample_size, 4))
    Y = np.ones(shape=(total_samples, 2))

    breakflag = False
    i = 0
    for i in range(total_samples - 1):
        try:
            x = signal[l:r, :]

            target_time = (l + sample_size / 2) / rate
            y = approximate_coord_at_time(track, time,
target_time)

            if y is None:
                print(f'Мы потеряли {total_samples - i} тк
дорожка закончилась раньше')
                breakflag = True
                break
            if polar:
                y = cartesian2polar(*y)

            X[i] = x
            Y[i] = y

            l += sample_delta
            r += sample_delta
        except:

```

```

        print('Exxception on split dataset')

    if breakflag:
        X = X[:i, :]
        Y = Y[:i, :]

    return X, Y

class CircleCohort:
    def __init__(self, parts):
        delta = 2 * 3.1415926535 / parts
        self.cohorts = {}
        for i in range(parts):
            self.cohorts[i] = (i * delta, (i + 1) * delta)

    def indexOf(self, angle):
        for k, v in self.cohorts:
            if v[0] <= angle < v[1]:
                return k
        return None

    def indexesOf(self, angels):
        return

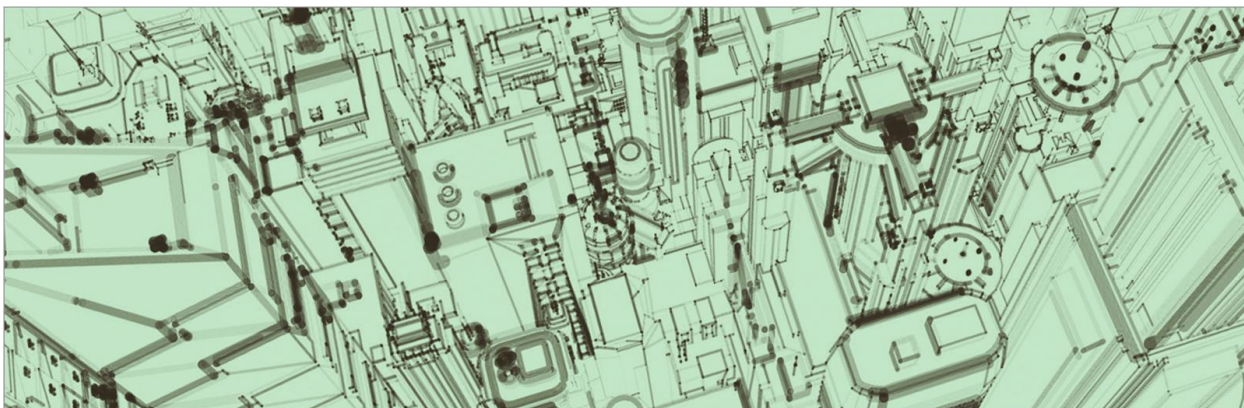
```

ПРИЛОЖЕНИЕ Б

(Обязательное)

Нейросетевая обработка многомерного сигнала

Презентация



Нейросетевая обработка многомерного сигнала

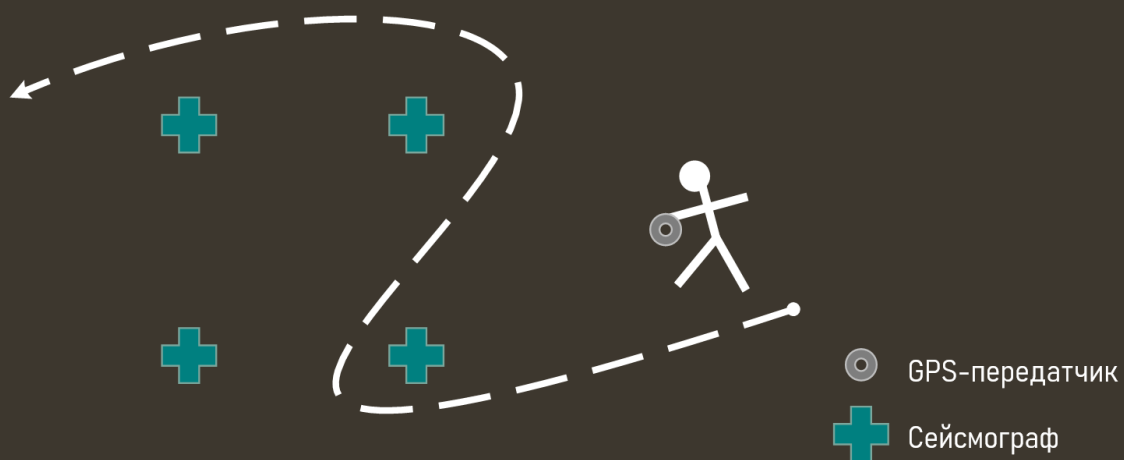
Выполнил студент гр. 22ВВиМ1 Федяшов М.С.

Научный руководитель: д.т.н., профессор кафедры ВТ ПГУ Митрохин М.А.

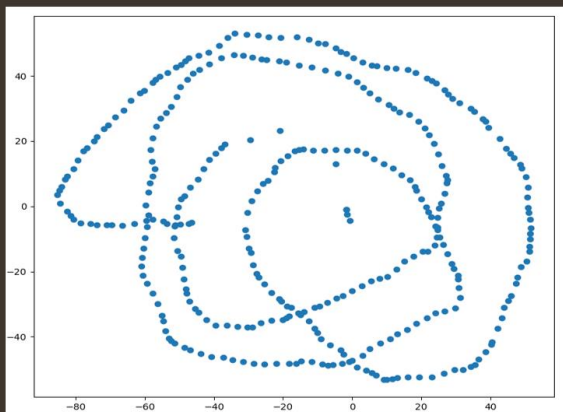
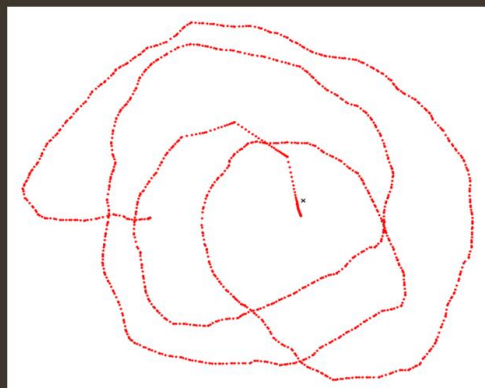
Пенза, 2024



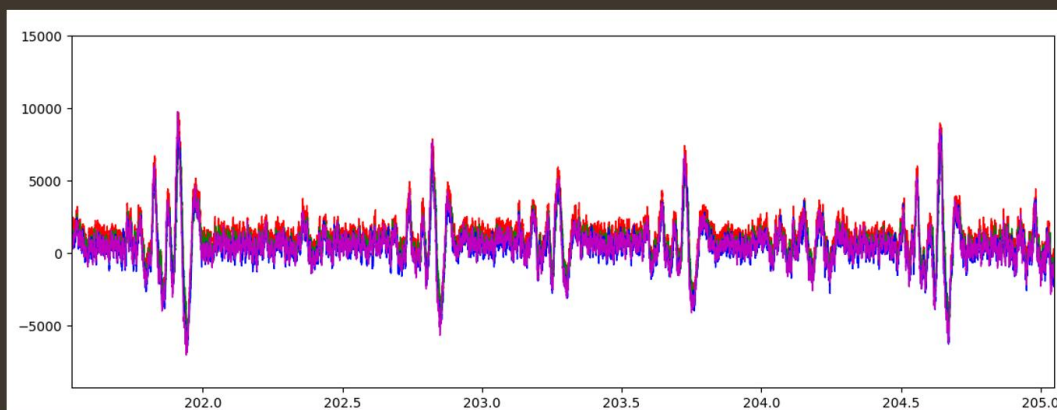
Постановка задачи



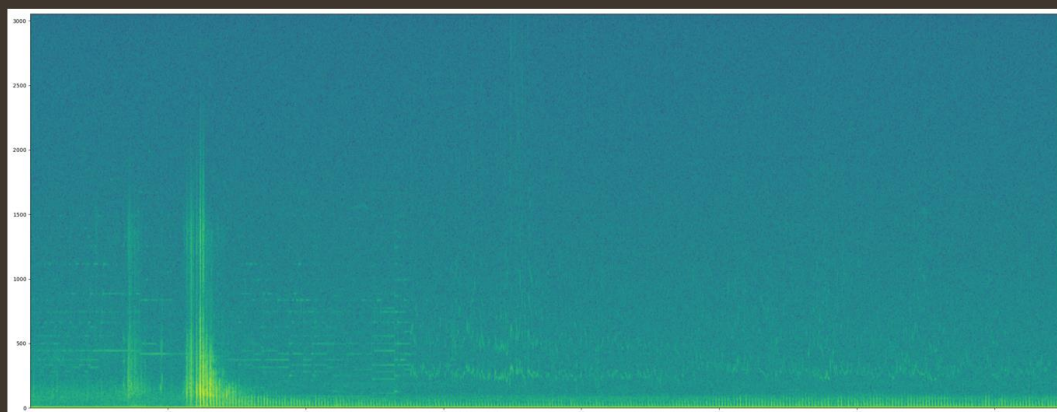
Визуализация данных. GPS-трекинг и его аппроксимация



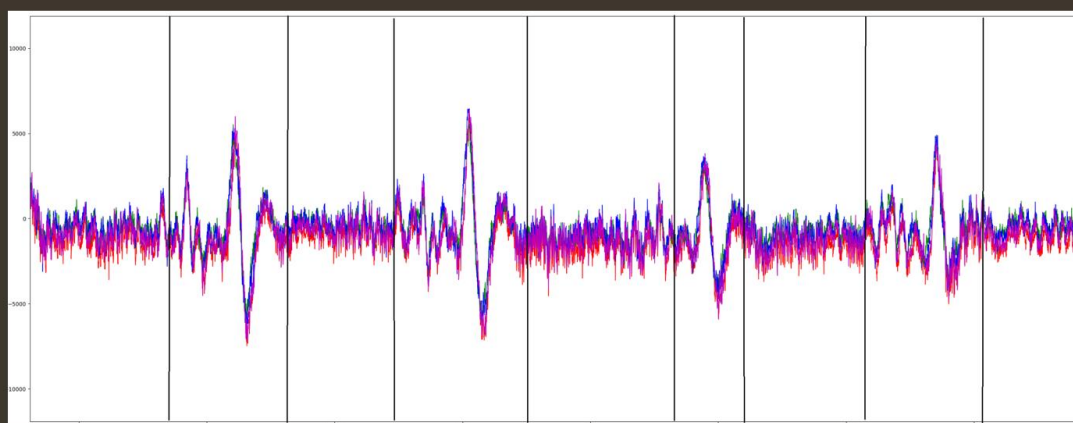
Визуализация данных. Сигнал с сейсмографов



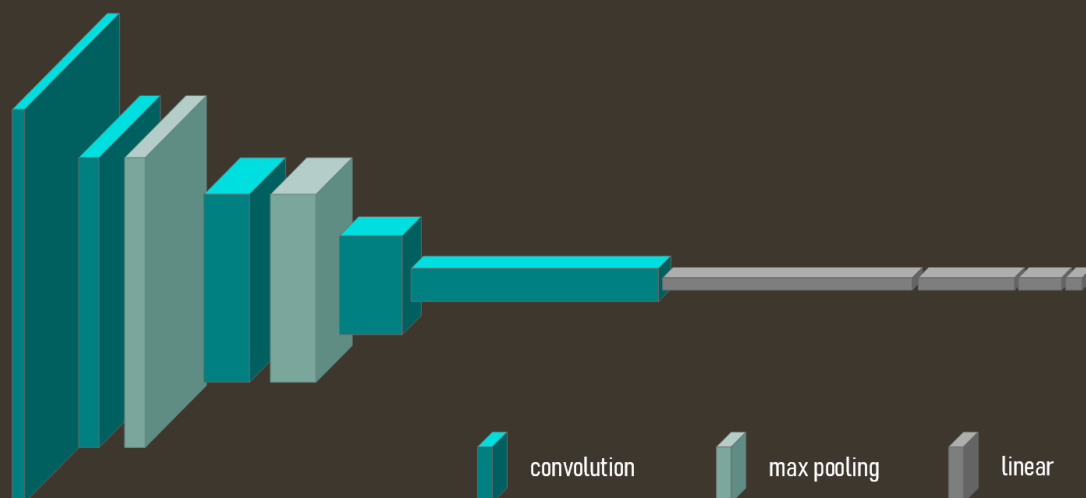
Визуализация данных. Спектрограмма сигнала



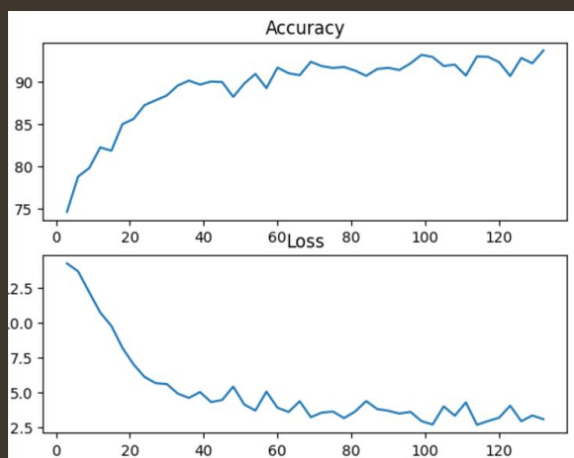
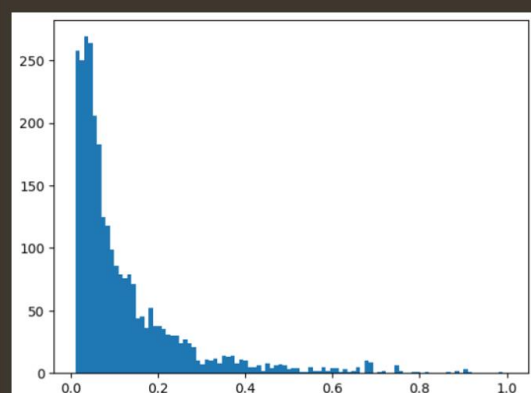
Разбиение и сохранение данных



Архитектура CNN



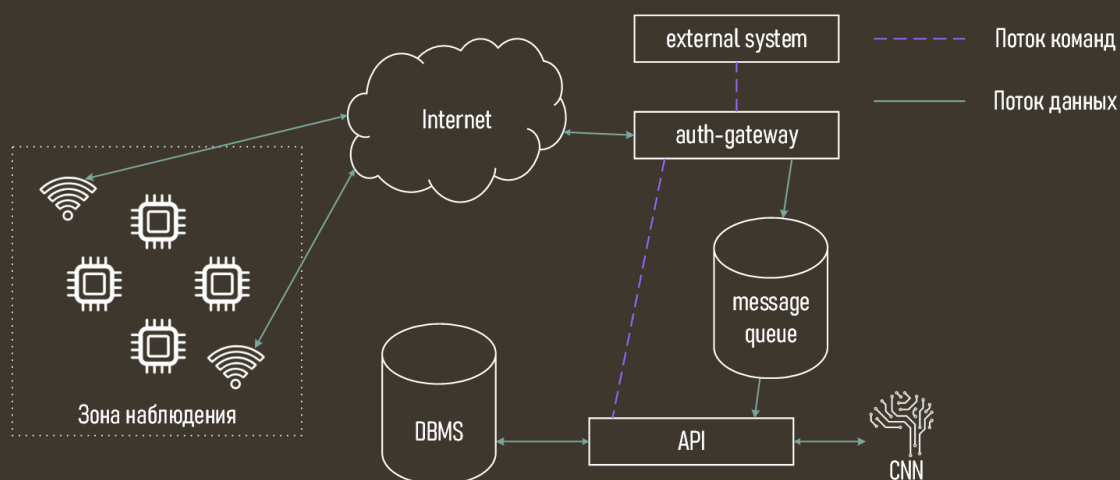
Результаты обучения CNN



Результаты обучения различных моделей

Архитектура	RMSE	m(err)	std(err)	Итераций
CNN	0.32	0.261	0.375	135
RNN	1.434	0.731	1.234	200
LSTM-1	1.17	0.779	0.787	20
LSTM-2	1.082	0.713	0.814	60
GRU	1.077	0.699	0.82	55

Модель веб-сервиса



Спасибо за внимание!