

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

ИГРОВОЕ ПРИЛОЖЕНИЕ НА ОСНОВЕ
КОМПЬЮТЕРНОГО ЗРЕНИЯ

Студент Тру 30.05.2025 Трошкин Владимир Константинович
(подпись, дата) (ФИО полностью)

Руководитель [подпись] 30.05.2025 Синев М. П.
(подпись, дата) (фамилия, инициалы)

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

Рецензент зед. каф. Программирование Пенз ГТУ, АИИ, 04.06.2025 Мартышкин А.И.
(подпись, дата) (фамилия, инициалы) (должность, место работы)

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

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

Работа защищена с отметкой отлично (протокол заседания ГЭК от 5.06.25 № 9)

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

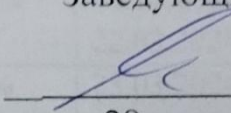
Пенза 2025

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 Трошкин Владимир Константинович гр. 23ВВИм1 факультета вычислительной техники, направления 09.04.01.

2. Тема работы «Игровое приложение на основе компьютерного зрения». Тема утверждена приказом ПГУ № 986/о от "29" сентября 2022 г.

3. Руководитель магистерской диссертации к.т.н., доцент кафедры ВТ ПГУ Синев М.П.

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

Назначение разработки: создание игрового приложения на основе компьютерного зрения.

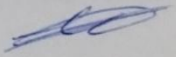
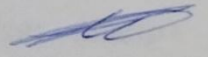
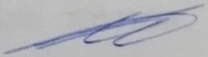
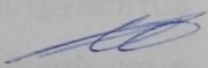
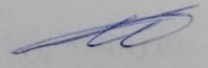
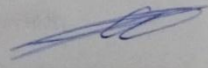
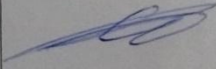
Разработать игровое приложение на основе компьютерного зрения для игры «камень, ножницы бумага» с автоматическим распознаванием жестов.

5. Перечень подлежащих разработке вопросов:

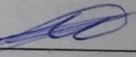
- 1) Анализ существующих способов захвата жестов.
- 2) Проектирование и обучение нейронной сети.
- 3) Разработка игрового приложения для интеграции модели нейронной сети.

4) Тестирование согласно ранее описанному техническому заданию на разработку.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1.	Изучение теоретических основ	10%	29.09.2023- 15.11.2024	
2.	Анализ действующих решений из сферы захвата жестов и движения	10%	16.11.2024- 22.03.2024	
3.	Выбор инструментов для разработки и утверждение функционала системы	5%	23.03.2024- 14.06.2024	
4.	Проектирование и обучение нейронной сети	30%	15.06.2024- 10.10.2024	
5.	Разработка игрового приложения	20%	11.10.2024- 05.01.2025	
6.	Интегрирование модели в разработанное приложение	15%	06.01.2025- 17.04.2025	
7.	Апробация разработанного игрового приложения в форме тестирования	5%	18.04.2025- 9.05.2025	
8.	Оформление выпускной работы	5%	10.05.2025- 28.05.2025	

Дата выдачи задания "29" сентября 2023 г.

Руководитель магистерской диссертации  29.09.2023 Синев М.П.
(подпись, дата)

Задание к исполнению принял студент  29.09.2023 Трошкин В.К.
(подпись, дата)

Работу к защите допустить
Декан факультета  05.06.2023 Фионова Л.Р.
(подпись, дата)

РЕФЕРАТ

Пояснительная записка содержит 81 лист, 32 рисунка, 10 источников и 1 приложение.

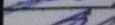
НЕЙРОННЫЕ СЕТИ, PYTORCH, PYTHON, TKINTER, MEDIAPIPE

Цель работы – проектирование и обучение модели нейронной сети для распознавания и классификации жестов и игрового приложения «камень, ножницы, бумага», в которое она будет интегрирована.

Необходимыми задачами для достижения цели являются: анализ актуальности проекта; анализ известных методов распознавания жестов; анализ типов нейронных сетей; изучение современных инструментов разработки нейронных сетей; проектирование и обучение нейронной сети; разработка кодовой базы приложения, которое будет использовать обученную модель нейронной сети; тестирование работы созданного приложения.

Объектами разработки являются: нейронная сеть для распознавания и классификации жестов и игровое приложение «камень, ножницы, бумага», в которое она интегрирована.

В процессе работы были выполнены все поставленные задачи. Были проанализированы: актуальность проекта, известные методы распознавания жестов, типы нейронных сетей. Были изучены современные инструменты разработки нейронных сетей. В результате на языке Python с использованием фреймворков PyTorch и MediaPipe и была разработана и обучена нейронная сеть, которая способна классифицировать жесты с точностью в 90%. Данная нейросеть была интегрирована в игровое приложение, написанное с помощью фреймворка Tkinter.

					ПГУ 1.090401.17.001 ПЗ					
Изм.	Лист	№ докум.	Подпись	Дата						
Разраб.		Трошкин В.К.		30.05.25	Игровое приложение на основе компьютерного зрения Пояснительная записка					
Провер.		Синев М.П.		30.05.25						
Н. Контр		Бычков А.С.		30.05.25						
				30.05.25						
Утверд.		Митрохин М.А.			Лит.		Лист		Листов	
								2	80	
					ФВТ, каф. ВТ					
					гр. 23ВВИМ1					

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ	
.3	
ВВЕДЕНИЕ	
4	
1 Анализ предметной области	
9	
1.1 Анализ актуальности работы	
.9	
1.2 Анализ существующих подходов	
.11	
1.2.1 Маркерный подход.	
11	
1.2.2 Безмаркерный подход	
12	
1.3 Анализ правил игры	
13	
1.4 Анализ и сравнение технологий	
.14	
1.4.1 Языки программирования	
14	
1.4.2 Фреймворки для машинного обучения	
18	
1.4.3 Средства создания графических интерфейсов	
19	
1.4.4 Выводы	
21	
2 Реализация	
23	

2.1. Проектирование и обучение нейронной сети	.23
2.1.1 Двухслойная свёрточная нейросеть	.23
2.1.2 Mediapipe со слоем классификации	38
2.2. Разработка игрового приложения	43
2.2.1.Постановка задачи	43
2.2.2.Проектирование приложения	44
3 Тестирование	.65
ЗАКЛЮЧЕНИЕ	.62
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	64
ПРИЛОЖЕНИЕ А Игровое приложение Листинг программы	.66
ПРИЛОЖЕНИЕ А.1 Файл «GUI.py»	69
ПРИЛОЖЕНИЕ А.2 Файл «detector.py»	76
ПРИЛОЖЕНИЕ А.3 Файл «game.py»	.80

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В данной работе применяют следующие сокращения и обозначения
ИИ – искусственный интеллект. Способность вычислительной машины моделировать процесс мышления за счет выполнения функций, связанных с человеческим интеллектом

API (Application Programming Interface) – описание способов, которыми одна компьютерная программа может взаимодействовать с другой программой

CNN (Convolutional Neural Network) – сверточная нейронная сеть

GPU (Graphic Processing Unit) – устройство обработки графики

GUI (Graphical User Interface) – графический интерфейс пользователя

JIT-компиляция (Just In Time) – компиляция во время выполнения программы

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

ВВЕДЕНИЕ

Современные технологии значительно расширили возможности взаимодействия человека и вычислительной машины. Одно из передовых направлений в этой области – компьютерное зрение. Оно широко используется в медицинской диагностике, системах контроля качества на производстве, игровой индустрии, системах виртуальной реальности и других областях. Исследования в области компьютерного зрения направлены на разработку систем, способных анализировать и понимать зрительную информацию подобно человеку. Для распознавания на изображениях и видео объектов, форм, движения сейчас используются такие технологии, как искусственные нейронные сети, классическое и глубокое машинное обучение. В развлекательной индустрии использование компьютерного зрения открывает возможность создания свежих захватывающих игровых механик. Распознавание жестов, захват движения, взаимодействие с виртуальным пространством – всё это становится возможным благодаря прогрессу в области компьютерного зрения.

Разрабатываемая система обнаружения и распознавания жестов в реальном времени будет использоваться в качестве системы ввода для игрового приложения. Назначение системы определяет предъявляемые к ней требования: высокая точность обнаружения, низкая задержка, устойчивость к изменениям условий, простота интеграции.

Целью работы является рассмотрение и анализ существующих способов реализации систем обнаружения и распознавания жестов в реальном времени.

Предметом исследования являются технологии, позволяющие осуществить процесс поиска и распознавания кисти руки с максимальной производительностью.

1 Анализ предметной области

1.1 Анализ актуальности работы

Системы распознавания жестов приобретают всё большую актуальность для игровой индустрии и прочих областей, поскольку позволяют взаимодействовать с машиной более интуитивно. В играх эти системы заменяют или дополняют традиционные устройства ввода, такие как клавиатуры, мыши или контроллеры, позволяя игрокам взаимодействовать с виртуальными средами посредством движений тела, жестов рук или выражения лица. Игровой процесс становится более естественным, благодаря чему граница между реальностью и виртуальным миром размывается, усиливается погружение игрока, что особенно ценно для игр в виртуальной и дополненной реальности. Вследствие этого распознавание жестов способствует более насыщенному повествованию, появлению более динамичных игровых механик и повышению доступности для пользователей, испытывающих трудности с традиционными элементами управления [1].

Системы обнаружения жестов находят широкое применение не только в игровой индустрии, но и в медицине, образовании и других сферах, что подчеркивает их универсальность и актуальность в контексте технологического развития. Например, они могут использоваться для комфортного использования технологий людьми с инвалидностью, для которых недоступны прочие методы ввода данных [2]. Ещё одной областью, где широко применяется захват жестов, является индустрия кино, где эта технология используется для спецэффектов и комбинированной съёмки.

Системы обнаружения и распознавания жестов – это актуальное и динамично развивающееся инженерное направление, а внедрение их в игровые приложения позволяет отработать технологии и создать базу для использования компьютерного зрения во многих других сферах. На основании этого можно сделать вывод, что данная диссертация является актуальной.

1.2 Анализ существующих подходов

Существует два подхода к захвату жестов: маркерный и безмаркерный.

1.2.1 Маркерный подход

Маркерный подход заключается в использовании специальных датчиков – маркеров, закрепляемых на теле или одежде. При необходимости захвата мимики они могут закрепляться и на лице.

Существует несколько видов маркерных систем:

– оптические пассивные системы. На костюме, входящем в состав такой системы, установлены датчики-маркеры, отличающиеся тем, что они только отражают свет, поступающий на них, но сами не светятся. В этих системах инфракрасное излучение направляется на маркеры с высокочастотных стробоскопов, установленных на камерах. Отраженный свет попадает обратно в объектив камеры, передавая информацию о позиции маркера [3]. Минусы оптических пассивных систем:

- 1) длительная подготовка;
- 2) при быстрых движениях или близком расположении маркеров они могут быть перепутаны;

– оптические активные системы. В этих системах вместо светоотражающих маркеров используются светодиодные датчики со встроенными микроконтроллерами и радио-синхронизацией. Каждому светодиоду присваивается идентификатор, что предотвращает ошибки в определении маркеров и позволяет системе распознавать их после временного перекрытия [4]. Минусы оптических активных систем:

- 1) необходимость подзарядки маркеров;
- 2) сравнительно большой размер маркеров;
- 3) невозможность захвата мимики лица;
- 4) хрупкость и относительно высокая стоимость маркеров;

– магнитные системы. В этих системах маркерами являются магниты, а камеры действуют в качестве ресиверов, определяя их позиции на основе искажений магнитного потока [3]. Минусы магнитных систем:

- 1) ограничения в захвате движений и мимики лица;
- 2) влияние магнитных и электрических помех от окружающих объектов;
- 3) меньшая рабочая зона по сравнению с оптическими системами;
- 4) необходимость дополнительного контроллера и проводов, соединяющих пользователя с компьютером;

– механические системы. Эти системы отслеживают движения, следя за сгибами суставов через специальный механический экзоскелет. Минусы механических систем:

- 1) ограничение движений механическим скелетом с дополнительным контроллером;
- 2) отсутствие возможности захвата движений и мимики лица;
- 3) риск поломки механических элементов при неаккуратном использовании;

– инерциальные системы. Такие системы используют инерциальные датчики, такие как акселерометры и гироскопы, прикрепленные к телу человека [5]. Преимущества:

- 1) переносимость и невозможность перекрытия;
- 2) подходят для использования вне лабораторных условий.

Недостатки:

- 3) возможно накопление ошибок из-за дрейфа гироскопов;
- 4) могут быть ограничены в точности, особенно при сложных движениях.

1.2.2 Безмаркерный подход

Безмаркерный подход полагается на методы компьютерного зрения, а потому не требует дополнительного оборудования в виде маркеров,

специальных костюмов или перчаток. В качестве камеры может использоваться одна цифровая камера, несколько камер или камера со встроенным радаром, формирующая карту глубины. Преимуществами такого подхода являются низкая стоимость, быстрота развёртывания, удобство в использовании. В качестве минусов можно назвать несколько меньшую точность и стабильность метода, а также более высокие требования к вычислительной производительности.

Для игрового приложения более важными критериями являются низкая стоимость и удобство, так что остановимся на безмаркерном подходе в его простейшей реализации с одной камерой. Требования к производительности при этом всё ещё будут оставаться в рамках обычного персонального компьютера.

1.3 Анализ правил игры

«Камень, ножницы, бумага» – это классическая игра, в которой два игрока одновременно выбирают один из трех возможных жестов: камень, ножницы или бумагу. Правила игры просты: камень побеждает ножницы, ножницы побеждают бумагу, бумага побеждает камень.

Жесты выглядят следующим образом:

- «камень»: рука сжата в кулак,
- «ножницы»: указательный и средний пальцы выпрямлены, остальные согнуты,
- «бумага»: пальцы выпрямлены и прижаты друг к другу.

На рисунке 1 приведён внешний вид этих жестов:

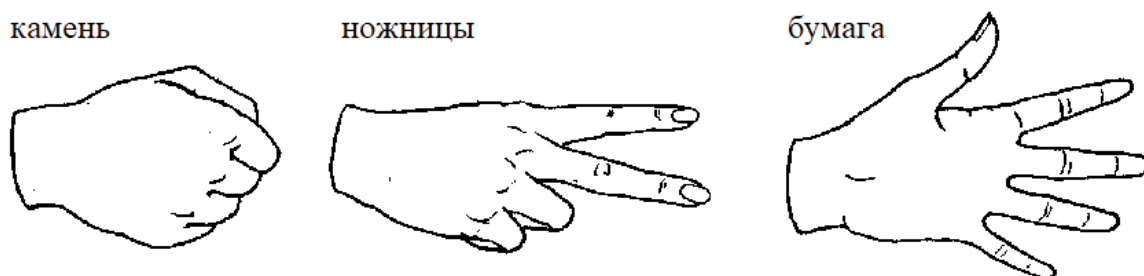


Рисунок 1 – Внешний вид жестов

В игровом приложении на основе компьютерного зрения игрок будет использовать жесты рук перед камерой для выбора одного из трех символов. Приложение должно распознавать жесты игрока и генерировать жесты компьютера, после чего определять победителя раунда.

1.4 Анализ и сравнение технологий

Для создания игрового приложения «камень, ножницы, бумага» на основе компьютерного зрения необходимо провести анализ и сравнение различных технологий: языков программирования, фреймворков для компьютерного зрения и средств создания графических интерфейсов. Рассмотрим несколько популярных вариантов и выберем наиболее подходящие.

1.4.1 Языки программирования

При создании игры «камень, ножницы, бумага» с функцией распознавания жестов на роль подходящих кандидатов претендуют несколько языков программирования, каждый из которых обладает своими преимуществами и недостатками в контексте реализации нейронных сетей и обработки изображений в реальном времени.

Python обычно является основным выбором для задач машинного обучения благодаря своей обширной экосистеме специализированных библиотек. TensorFlow и PyTorch являются комплексными фреймворками для разработки нейронных сетей, OpenCV предлагает возможности для реализации надёжных систем компьютерного зрения. Интерпретируемая природа языка способствует удобству эксперимента и быстрому прототипированию, позволяя разработчикам оперативно перебирать различные архитектуры моделей и методы предварительной обработки данных. Но в приложениях реального времени ограничения Python в скорости выполнения иногда становятся критичны, ведь задержки при обработке кадров могут существенно омрачить пользовательский опыт.

JavaScript представляет собой все более привлекательную альтернативу, особенно с появлением TensorFlow.js и ускорения графики на базе WebGL. Библиотека TensorFlow.js позволит использовать машинное обучение в программах на JavaScript. Также понадобится WebRTC – технология, которая обеспечивает беспрепятственный доступ к камере. Развертывание на стороне браузера устраняет необходимость в установке, благодаря чему приложение будет являться мгновенно доступным для пользователей на разных платформах. Современные движки JavaScript демонстрируют впечатляющий рост производительности, а ускорение на GPU через WebGL может конкурировать с нативными реализациями для определенных операций с нейронными сетями. Тем не менее, ограничения безопасности браузера могут сужать доступ к расширенным функциям камеры, а вычислительные издержки при запуске сложных моделей в браузерной среде могут создавать нагрузку на системные ресурсы.

C++ предлагает непревзойденную производительность для вычислительно интенсивных задач, что делает его привлекательным для приложений, где критична минимальная задержка. Библиотеки, такие как OpenCV, предоставляют нативные реализации на C++, оптимизированные для скорости, а фреймворки, такие как PyTorch C++ и TensorFlow Lite, позволяют эффективно развертывать нейронные сети. Детальное управление памятью, присущее языку, обеспечивает оптимальное использование ресурсов, что критически важно при обработке видеопотоков высокого разрешения в реальном времени. Однако такая работа с памятью требует внимательности и высокой квалификации программиста, увеличивает вероятность возникновения ошибок, значительно увеличивает время разработки и отладки. Отсутствие высокоуровневых абстракций для машинного обучения требует более глубоких знаний в области низкоуровневого программирования.

Java предлагает сбалансированный подход, сочетая приемлемую производительность с независимостью от аппаратной платформы. JIT-компиляция в виртуальной машине Java позволяет достигать скоростей, близких к нативным. Для машинного обучения на Java существуют такие библиотеки, как Deeplearning4j [6]. Строгая система типизации и обширная поддержка инструментов в Java облегчают разработку и сопровождение крупномасштабных приложений. Механизм сборки мусора, хотя в целом и эффективен, может вносить непредсказуемые паузы, которые могут негативно сказаться на производительности в реальном времени. Кроме того, экосистема Java для компьютерного зрения и машинного обучения не настолько широка и отлажена, как у Python, хотя и функциональна.

Анализ производительности выявляет значительные различия между языками и подходами к реализации. Реализации на Python, как правило, достигают достаточной частоты кадров для приложений по распознаванию жестов, обрабатывая 15–30 кадров в секунду в зависимости от сложности модели и характеристик оборудования. Реализации на JavaScript могут соответствовать этим показателям при использовании ускорения WebGL, хотя выполнение только на центральном процессоре, скорее всего приведёт к недостаточной производительности. Реализации на C++ стабильно обеспечивают самую высокую частоту кадров, часто превышая 60 кадров в секунду даже при использовании сложных архитектур нейронных сетей.

Сложность разработки существенно различается между языками. Высокоуровневые абстракции и обширная документация Python обеспечивают быстрый цикл разработки, позволяют создать базовый прототип программы для распознавания жестов в течение нескольких дней. Плюсом разработки на JavaScript являются привычные веб-технологии и обширные онлайн-ресурсы, хотя отладка реализаций нейронных сетей может оказаться сложной задачей. Разработка на C++ требует значительно больших

временных затрат: на создание эквивалентной функциональности могут уйти недели из-за необходимости тщательной отладки работы с памятью.

Кратко всё вышеописанное можно свести в таблицу 1.

Таблица 1 – Сравнение языков программирования

Язык	Скорость разработки	Производительность	Библиотеки машинного обучения
Python	Высокая	Средняя	TensorFlow и PyTorch
JavaScript	Средняя	Низкая	TensorFlow.js
C++	Низкая	Высокая	PyTorch C++ и TensorFlow Lite
Java	Средняя	Средняя	Deeplearning4j

В конечном итоге, выбор языка программирования для разработки игры «камень, ножницы, бумага» с распознаванием жестов на основе нейронных сетей зависит от конкретных требований и ограничений проекта. Python остается оптимальным выбором для проектов, ориентированных на исследования, где в приоритете быстрота экспериментов и разработки моделей. JavaScript превосходит в сценариях, требующих широкой доступности и кроссплатформенного развертывания без необходимости установки. C++ становится незаменимым, когда максимальная производительность имеет первостепенное значение, особенно это важно для компьютерных игр, использующих максимум вычислительных возможностей; для коммерческих приложений или сред с ограниченными ресурсами. Java предлагает прагматичный компромисс для корпоративных приложений, требующих платформенной независимости и долгосрочной поддержки кодовых баз.

Поскольку перед нами стоит задача создания экспериментального приложения, ключевым свойством языка для нас будет являться скорость разработки, удобство эксперимента. Таким образом, в качестве языка программирования целесообразно выбрать язык Python, позволяющий

удобно и быстро проверять на практике свои гипотезы, обеспечивая при этом достаточный уровень производительности.

1.4.2 Фреймворки для машинного обучения

Рассмотрим несколько фреймворков для машинного обучения, доступных для использования с помощью языка Python, выбранного в предыдущем пункте.

TensorFlow, благодаря интеграции с Keras, является практичным выбором для быстрого прототипирования. Он имеет обширную документацию и предобученные модели (например, MobileNet) что ускоряет разработку. TensorFlow Lite обеспечивает беспрепятственное развертывание на мобильных устройствах. Однако в TensorFlow используются статические вычислительные графы, что иногда усложняет отладку.

PyTorch предлагает большую гибкость за счет динамических вычислительных графов, что упрощает внесение изменений в модель в процессе экспериментов. Это может стать преимуществом при доработке архитектур свёрточных нейронных сетей для распознавания положения пальцев условиях недостаточного освещения. Интуитивно понятный синтаксис фреймворка позволяет сократить шаблонный код для загрузки и преобразования данных. На граничных случаях задержка инференса несколько выше, чем у TensorFlow, что может повлиять на производительность.

Scikit-learn дополняет эти фреймворки для вспомогательных задач. Он содержит ряд функций для обработки обучающей последовательности перед подачей в нейронную сеть. Однако он не имеет нативной поддержки глубоких архитектур, необходимых для распознавания изображений.

Бенчмарки производительности показывают превосходство TensorFlow в скорости работы обученной сети при использовании квантованных моделей, что может быть важно для плавности игрового процесса. PyTorch же демонстрирует лучшую эффективность обучения за счет JIT-компиляции.

Поддержка сообщества и ресурсы для устранения неполадок одинаково хорошо развиты как для TensorFlow, так и для PyTorch.

Поскольку для стоящих перед нами исследовательских задач важна скорость прототипирования, удобство эксперимента, воспользуемся PyTorch, который позволяет более быстро обучать нейросети и менять при необходимости архитектуру.

1.4.3 Средства создания графических интерфейсов

При разработке графических интерфейсов в Python выбор библиотеки определяет интерактивность, визуальный стиль и сложность разработки проекта. Три ведущих варианта – Tkinter, Pygame и PyQt – служат для различных сценариев использования, обусловленных задачами, набором функций и кривой обучения.

Tkinter входит в пакет стандартной поставки Python и не требует дополнительной установки [7]. Его сильная сторона – доступность: готовые виджеты (кнопки, надписи, текстовые поля) и простая событийно-ориентированная модель делают его идеальным для быстрой разработки простых приложений для настольных компьютеров – например, форм, служебных панелей или приложений, где функциональность важнее, чем красота. Простота фреймворка позволяет даже новичкам в разработке графических интерфейсов связывать кнопки с функциями, отображать данные в реальном времени в строках или читать пользовательский ввод. Однако создание быстрых игровых интерфейсов со стилизацией и анимациями потребует применения обходных решений, так как архитектура Tkinter отдает приоритет простоте использования, а не стилю и гибкости. Для проектов, где критичны скорость развертывания и базовая интерактивность, Tkinter ценен благодаря готовности к работе сразу, без необходимости отдельной установки, настройки и длительного изучения.

Pygame был создан для разработки игр. Он фокусируется на рендеринге в реальном времени, мультимедийной обработке и высокой

скорости отклика. В отличие от виджет-ориентированного подхода Tkinter, в Pygame весь экран – это динамический холст: спрайты отрисовываются и обновляются кадр за кадром, звуковые эффекты и музыка синхронизируются с действиями пользователя, а программа молниеносно реагирует на ввод с клавиатуры, мыши или джойстика. Это делает его идеальным для создания интерактивных приложений – аркадных игр, физических симуляций или проектов, где визуальная обратная связь является ключевым элементом взаимодействия. Однако Pygame не имеет готовых виджетов для многих распространённых задач (например, диалогов выбора файлов, выпадающих меню со вложенными опциями или календарей). В результате его использование для бизнес-приложений или инструментов ввода данных кажется неестественным, его истинная сила проявляется в проектах, где интерактивность основана на динамичной визуализации, действиях пользователя в реальном времени и иммерсивной аудиовизуальной обратной связи.

PyQt – это Python-обертка для фреймворка Qt, предназначенная для создания сложных, функционально насыщенных приложений. Он предлагает обширную библиотеку настраиваемых виджетов (от простых кнопок до сложных таблиц данных и 3D-видов), поддержку гибкой стилизации (через Qt Designer с drag-and-drop интерфейсом или CSS-подобный синтаксис) и надежную кроссплатформенную совместимость – приложения выглядят и работают одинаково на Windows, macOS и Linux. Для проектов, требующих изысканных пользовательских интерфейсов (например, таких как корпоративные инструменты, мультимедийные редакторы, финансовые панели или ПО со сложной навигацией и многооконными рабочими процессами), модульная архитектура и встроенные утилиты PyQt упрощают разработку. Кривая обучения круче, чем у Tkinter, так как освоение концепций, таких как сигналы и слоты (система обработки событий PyQt), иерархии виджетов и управление ресурсами требует времени. Однако

результат – экстремальная гибкость: разработчики могут создавать все, от гладких современных панелей с интерактивными диаграммами до визуализаторов научных данных, используя высокопроизводительные функции Qt. PyQt хорошо подходит в тех случаях, когда проекты требуют как эстетической изысканности, так и глубокой функциональной сложности, позволяя сочетать дружелюбный дизайн с промышленной надежностью.

В таблице 2 представлены важные нам свойства графических библиотек.

Таблица 2 – Сравнение графических библиотек

Библиотека	Скорость разработки	Гибкость
Tkinter	Высокая	Низкая
Pygame	Средняя	Средняя
PyQt	Низкая	Высокая

Решающими особенностями в нашей работе будут являться простота развёртывания и скорость разработки, поскольку задача не требует высокой интерактивности графического интерфейса (основной игровой процесс будет осуществляться жестами, а не экранными элементами управления). Целесообразнее всего выбрать Tkinter в качестве библиотеки для реализации экранного графического интерфейса.

1.4.4 Выводы

На основании проведенного анализа, наиболее подходящими технологиями для создания игрового приложения "камень, ножницы, бумага" на основе компьютерного зрения будут следующие:

– язык программирования: Python, так как он обеспечивает простоту разработки и имеет отличную поддержку библиотек для машинного обучения и компьютерного зрения,

– фреймворк для компьютерного зрения: PyTorch, так как позволяет гибко менять архитектуру модели, быстро её обучать и поэтому хорошо подходит для экспериментирования,

– графическая библиотека: Tkinter, так как он прост в освоении, позволяет быстро разрабатывать графические интерфейсы приложений и входит в стандартную поставку Python.

2 Реализация

Реализация приложения будет происходить в два этапа:

- 1) проектирование и обучение нейронной сети,
- 2) разработка игрового приложения на основе этой нейронной сети.

2.1. Проектирование и обучение нейронной сети

Алгоритм должен решать задачу классификации изображений на 4 класса: «камень», «ножницы», «бумага», «ничего».

Классам должны соответствовать следующие виды изображений:

- «камень»: рука сжата в кулак,
- «ножницы»: указательный и средний пальцы выпрямлены, остальные согнуты,
- «бумага»: пальцы выпрямлены и прижаты друг к другу,
- «ничего»: любые изображения, которые невозможно отнести ни к одному из вышеперечисленных классов.

Система должна обеспечивать распознавание жеста в условиях комнатного освещения и без подготовленного фона.

Схожей задачей является распознавание рукописного текста, одним из этапов которого является классификация. С одной стороны, требования к нашему алгоритму более жёсткие, так как текст обычно располагается на сравнительно однородном фоне и контрастирует с ним. С другой стороны, требования к нашему алгоритму более мягкие, так как классов жестов намного меньше, чем букв в латинском или кириллическом алфавите.

2.1.1 Двухслойная свёрточная нейросеть

Попробуем применить архитектуру, использующуюся для распознавания рукописного текста – свёрточную нейросеть с двумя слоями с ReLU активацией, подвыборкой максимального значения (maxpooling) и третьим классификационным слоем.

Входное изображение масштабируется до размера 256 на 256 пикселей и подаётся на первый свёрточный слой нейросети. Ядро свёртки имеет

размер 5 на 5, количество карт признаков – 16, функция активации – ReLU (график её изображён на рисунке 2). Размер пакета – 4.

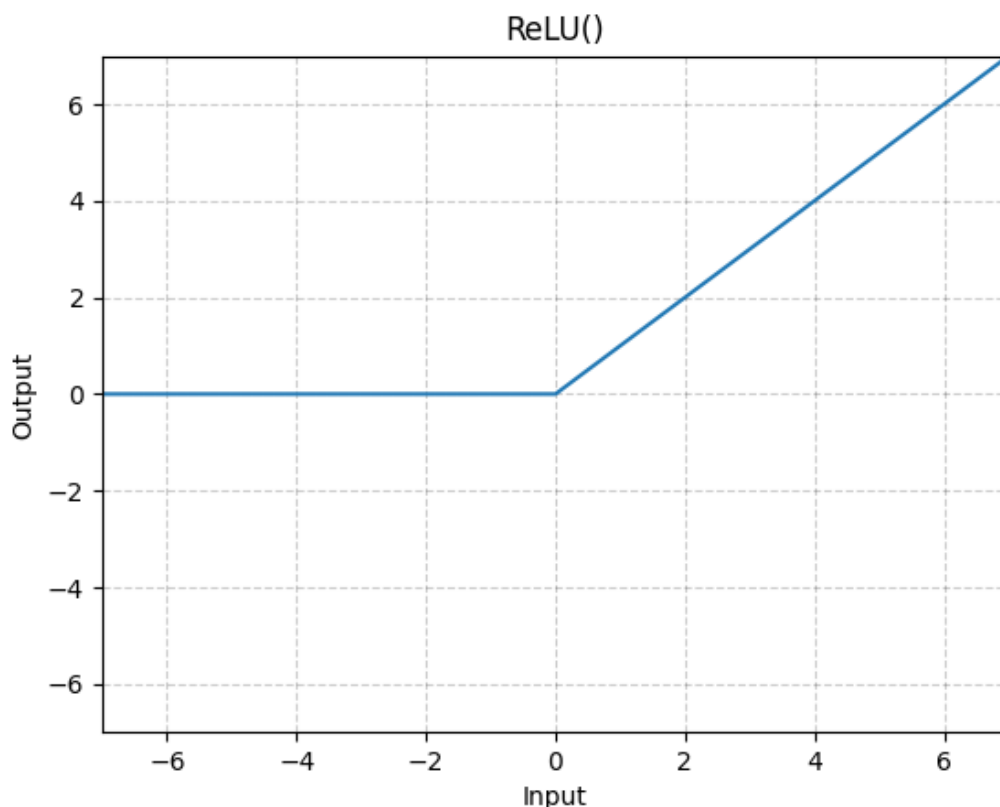


Рисунок 2 – График функции ReLU

Затем размер карт признаков уменьшается вдвое, для чего формируется подвыборка, где в каждом квадрате 2 на 2 выбирается максимальное значение. На выходе первого слоя получается тензор размерностью 16 на 128 на 128. В следующем слое свёртка с ядром 5 на 5 формирует 32 карты признаков, размер которых снова уменьшается вдвое при помощи подвыборки. Итоговый тензор 32 на 64 на 64 разворачивается в вектор и подаётся на полносвязный слой, а с него на 4 выходных нейрона, соответствующих искомым классам. Каждый из этих нейронов будет выдавать нецелое число от нуля до единицы, которое будет соответствовать степени схожести поданной на вход картинки с классом, который соответствует этому нейрону. Будем выбирать из всех этих классов тот, число на выходе нейрона которого максимально.

Общий вид архитектуры представлен на рисунке 3.

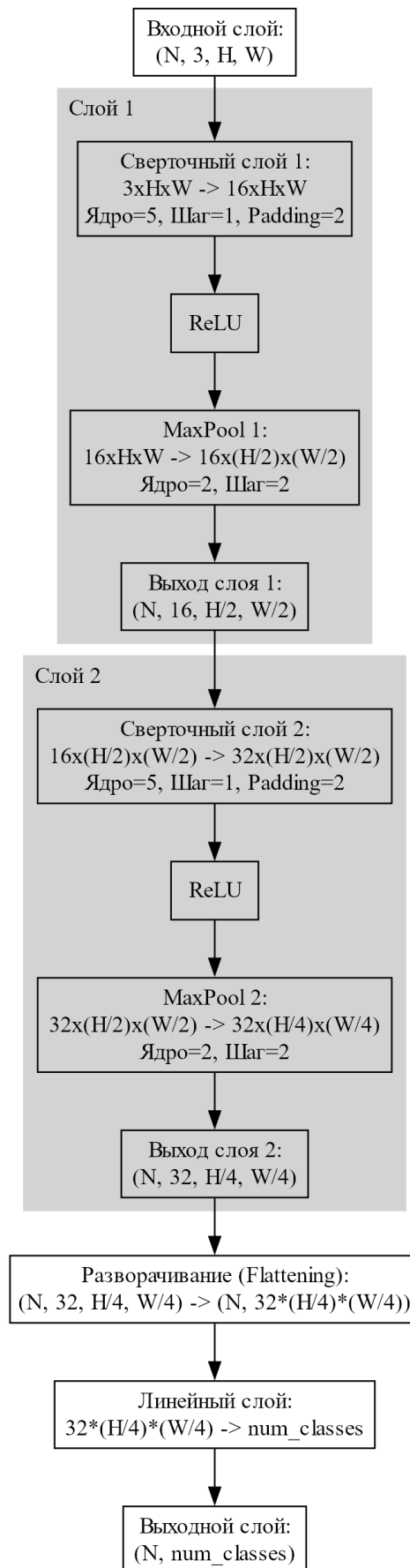


Рисунок 3 – Архитектура сети

Для обучения нейросети воспользуемся набором данных, воспользуемся набором обучающих изображений rock-paper-scissors за авторством Yash Mohan (<https://www.kaggle.com/datasets/yash811/rockpaperscissors>), поскольку он содержит фотографии рук на разном фоне, что соответствует реальным условиям, а также имеет класс «none», когда рука в кадре отсутствует.

Будем обучать нейросеть эпоха за эпохой и измерять её точность на тестовой выборке, пока точность не начнёт падать.

Получили наилучшую точность 85,3% на 2 эпохе, считая с нуля. На рисунке 4 изображён график изменения точности нейросети, на рисунке 5 – график функции потерь.

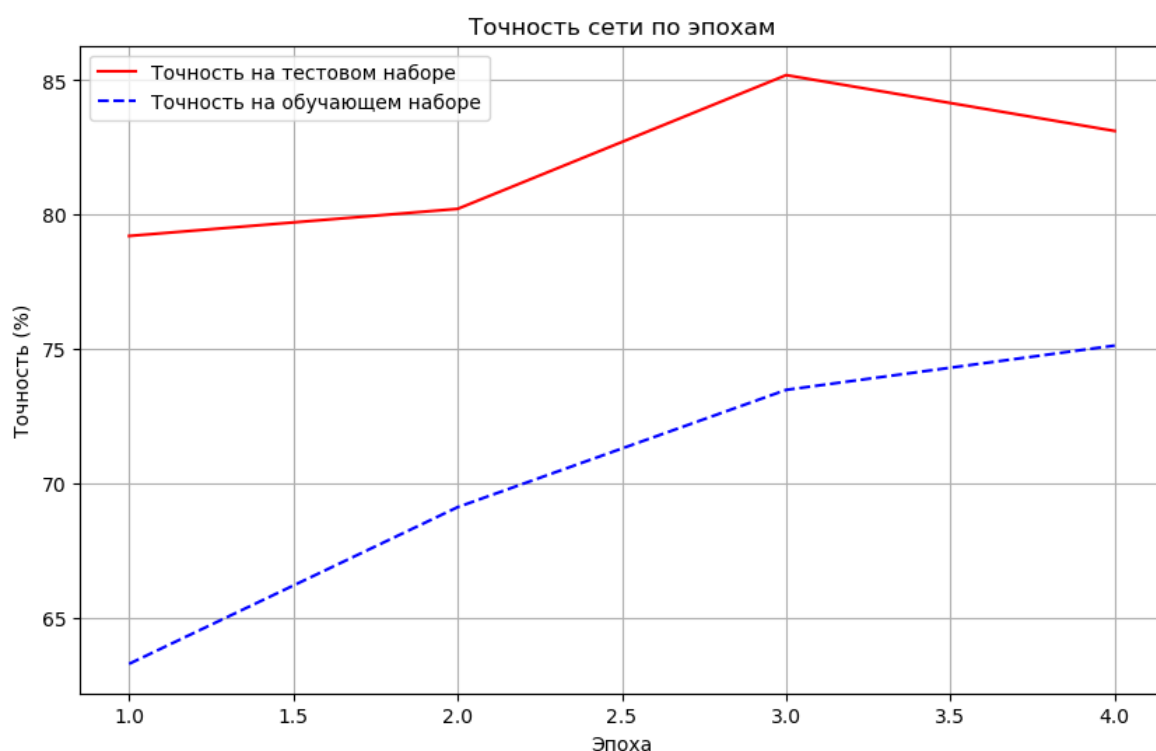


Рисунок 4 – График изменения точности нейросети

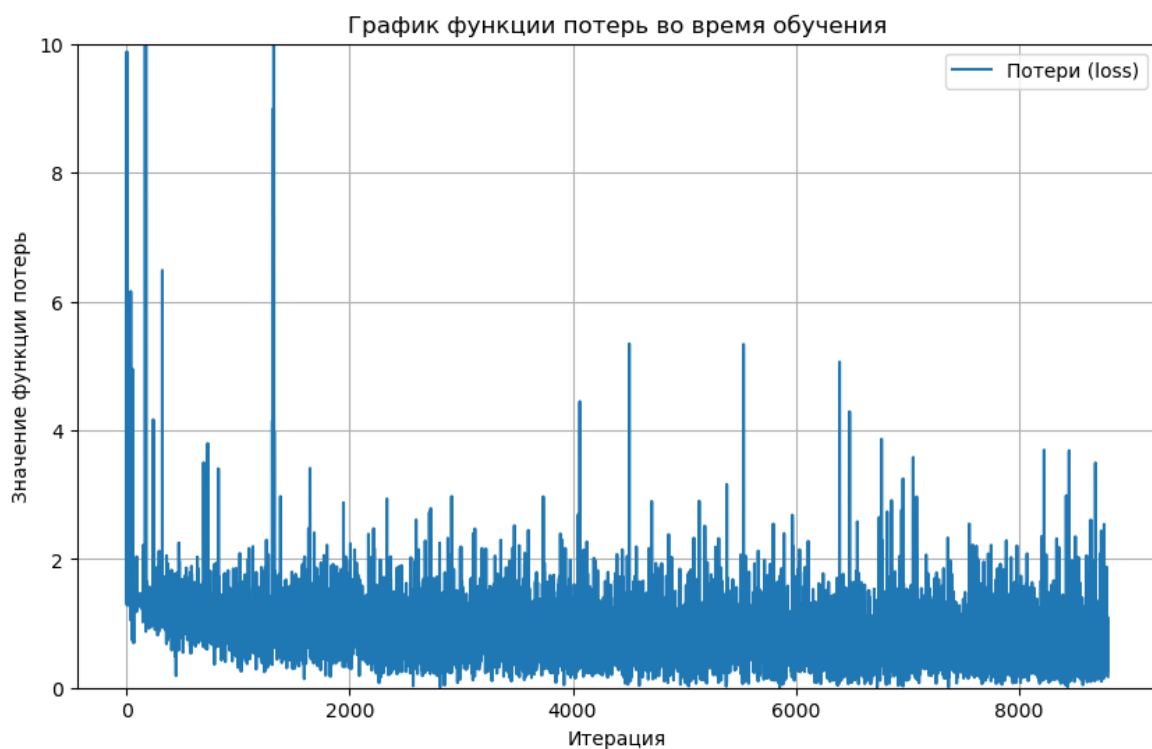


Рисунок 5 – График функции потерь

Построение графиков точности при обучении нейронных сетей и настройке их гиперпараметров позволяет получить непосредственное и интуитивное представление о траектории обучения модели на протяжении последовательных эпох обучения.

Анализируя кривую точности на обучающей выборке, можно понять, насколько эффективно модель усваивает закономерности из тренировочных данных. Однако ещё более важной является кривая точности на валидационной выборке, которая, будучи построена рядом с первой, демонстрирует способность модели обобщать результаты на новые, ранее не виденные нейросетью данные — ключевой показатель её реальной эффективности. Взаимосвязь между этими двумя кривыми чрезвычайно информативна. Например, если точность на обучающей выборке постоянно растёт, а на валидационной — застывает или снижается, это явный визуальный признак переобучения. В таких случаях модель фактически запоминает обучающий набор данных вместе с шумом, что приводит к плохой работе на новых примерах. Напротив, постоянно низкие и плоские

кривые точности как на обучающей, так и на валидационной выборках указывают на недообучение, когда модель слишком проста или недостаточно обучена, чтобы уловить скрытую структуру данных.

При настройке гиперпараметров – таких как скорость обучения, размер батча или архитектура сети – графики точности становятся незаменимыми. Разные настройки дают различные кривые обучения. Сравнение этих графиков позволяет визуально оценить, какие конфигурации приводят к более быстрой сходимости, более высокой пиковой точности и лучшей обобщающей способности без чрезмерного переобучения. Такая визуальная обратная связь гораздо более показательна, чем просто финальные значения точности, так как она демонстрирует весь процесс обучения.

Более того, эти графики помогают принять обоснованное решение о ранней остановке обучения. Отслеживая, когда точность на валидационной выборке достигает плато или начинает ухудшаться, можно вовремя остановить обучение и откатиться до модели с наилучшими результатами. Это экономит вычислительные ресурсы и предотвращает ухудшение модели из-за переобучения. В результате становится возможным создание более устойчивой и хорошо обобщающей модели. Визуальные данные с графиков точности предоставляют четкую и интерпретируемую основу для принятия обоснованных решений на протяжении всего процесса разработки модели, обеспечивая эффективное обучение нейронной сети.

В нашем случае наблюдается повышение точности на обучающих данных и понижение на тестовых данных, то есть происходит переобучение. Также на тестовой выборке нейросеть даёт лучшие результаты, чем на обучающей, что свидетельствует о несбалансированности выборок.

Функция потерь по своей сути количественно оценивает расхождение между предсказаниями модели и истинными значениями. Визуализация этой метрики в процессе обучения по эпохам предоставляет прямое динамическое представление самого процесса обучения.

Наблюдение за кривой потерь на обучающей выборке показывает, действительно ли сеть обучается: устойчивая нисходящая тенденция указывает на то, что модель постепенно минимизирует ошибки на обучающем наборе данных. Крутизна этого спада также даёт представление о скорости обучения. Что ещё важнее, при построении графика потерь на валидационной выборке (ошибки на отдельном, не виденном нейросетью ранее наборе данных) эти графики становятся мощными диагностическими инструментами. Если потери на обучающей выборке продолжают снижаться, а на валидационной начинают расти, это классический визуальный признак переобучения. В таком случае модель становится слишком специализированной для обучающих данных, теряя способность к обобщению. Напротив, если потери как на обучающей, так и на валидационной выборках остаются высокими и стабильными, это сигнализирует о недообучении, что может означать, что модель слишком проста или недостаточно обучена для выявления скрытых закономерностей. Внезапные скачки или хаотичное поведение кривой потерь также могут указывать на проблемы, такие как чрезмерно высокая скорость обучения или числовая нестабильность, например, взрывающиеся градиенты, при которых потери могут даже стать равными «Not a Number» (NaN).

При настройке гиперпараметров, где исследуются такие параметры, как скорость обучения, размер батча или тип оптимизатора, графики функции потерь становятся незаменимыми для сравнительного анализа. Разные наборы гиперпараметров дают разные траектории потерь. Сравнивая эти графики, можно визуально оценить, какие конфигурации приводят к более быстрой сходимости к меньшему значению ошибки, а какие поддерживают хороший баланс между производительностью на обучающей и валидационной выборках без чрезмерных колебаний. Например, слишком высокая скорость обучения может проявляться в виде сильно колеблющихся или даже растущих потерь, а слишком низкая — в крайне медленном

снижении потерь. Такая визуальная обратная связь позволяет делать более обоснованные корректировки, чем если бы мы полагались только на финальные значения потерь, так как видна вся история обучения.

В конечном итоге визуализация функции потерь обеспечивает прозрачное понимание поведения модели на протяжении всего обучения. Это позволяет своевременно выявлять проблемы, помогает в выборе оптимальных гиперпараметров и способствует принятию решений о продолжительности обучения, например, путём внедрения ранней остановки, когда потери на валидационной выборке перестают улучшаться или начинают ухудшаться. В результате становится возможным создание более устойчивой и эффективной нейронной сети, которая не только хорошо работает на известных данных, но и эффективно обобщает результаты на новые, ранее не виданные примеры.

В нашем случае функция потерь сильно колеблется, что может свидетельствовать о зашумлённости данных, слишком высокой скорости обучения, слишком малом размере пакета.

Нейросеть показывает точность на тестовых данных выше, чем на обучающих. Это свидетельствует о несбалансированности выборок и о предвзятости тестовой выборки.

Взглянем на данные. Пример представлен на рисунке 6. Изображения в тестовой выборке очень похожи на изображения в тренировочной, хотя и не совпадают полностью. Кажется, они взяты из соседних кадров видео.



Тренировка

Тестирование

Рисунок 6 – Слишком малые различия в изображениях датасета

Кроме того, количество изображений отличается между классами.

Заменяем в тестовой выборке изображения рук на таковые из другого датасета (<https://www.kaggle.com/datasets/alexandredj/rock-paper-scissors-dataset>), по 400 для каждого класса. Класс «none» тестовый оставим прежний, но заменим его в обучающей выборке на 2500 изображений класса «none» из этого датасета (<https://www.kaggle.com/datasets/abhinowww/hand-cricket-symbols>), из класса «камень» обучающей выборки удалим 75 изображений для того, чтобы их стало 2500, ножницы и бумагу дополним до 2500 из этого же датасета жестами «2» и «5», то есть 2 и 5 разогнутыми пальцами соответственно.

При загрузке датасета будем случайным образом поворачивать изображения.

Результаты можно увидеть на рисунках 7 и 8.

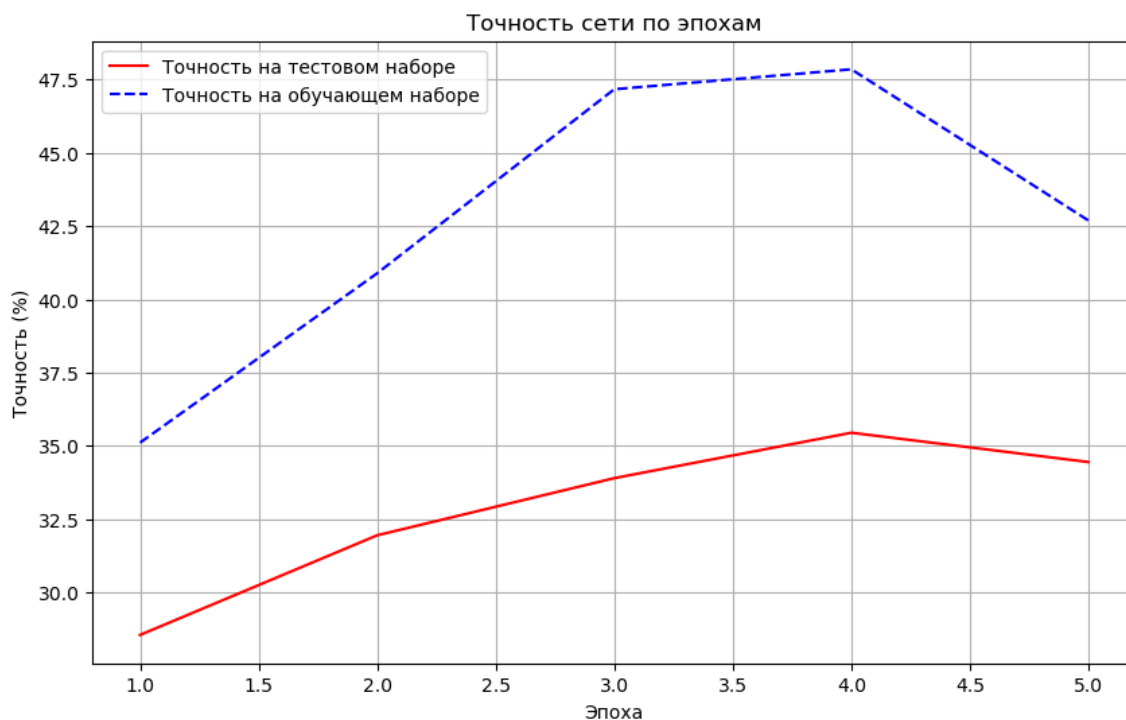


Рисунок 7 – Точность со сбалансированной выборкой

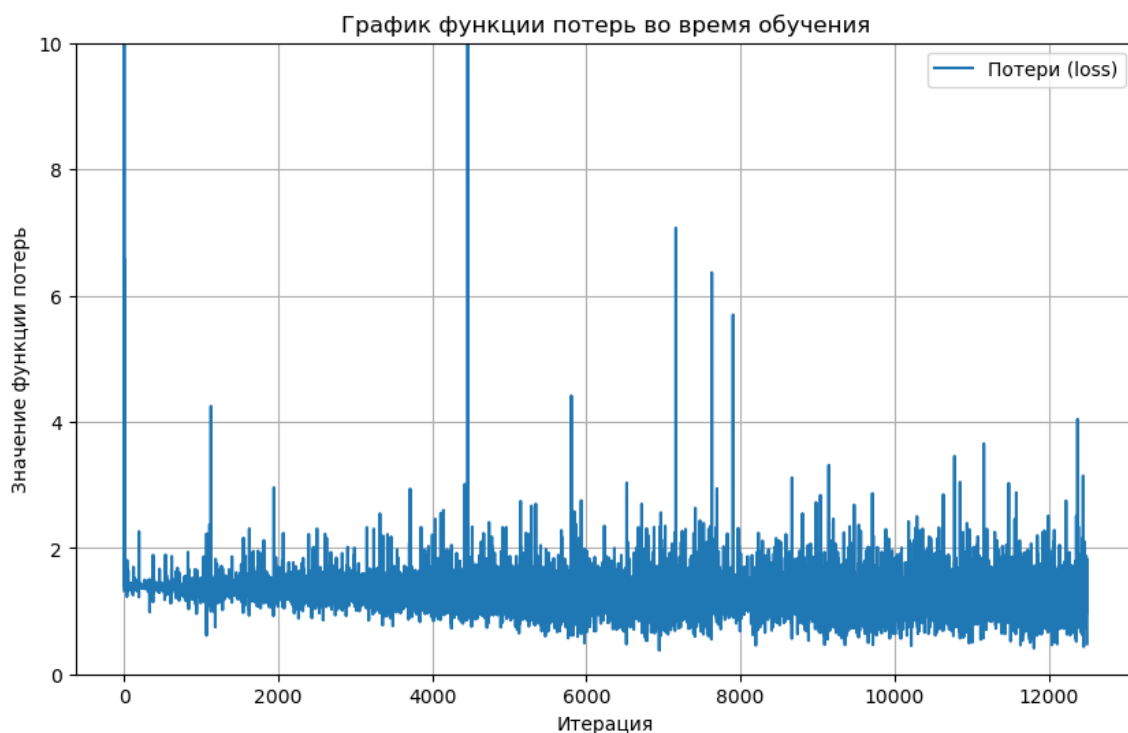


Рисунок 8 – Потери со сбалансированной выборкой

На графике функции потерь наблюдаются огромные выбросы. Точность сети при таких выбросах изменяется, по сути, случайным образом, обучения не происходит. Для борьбы со скачками функции потерь уменьшим скорость обучения в 10 раз – с 0,01 до 0,001.

Результат внесённых изменений – на рисунках 9 и 10.

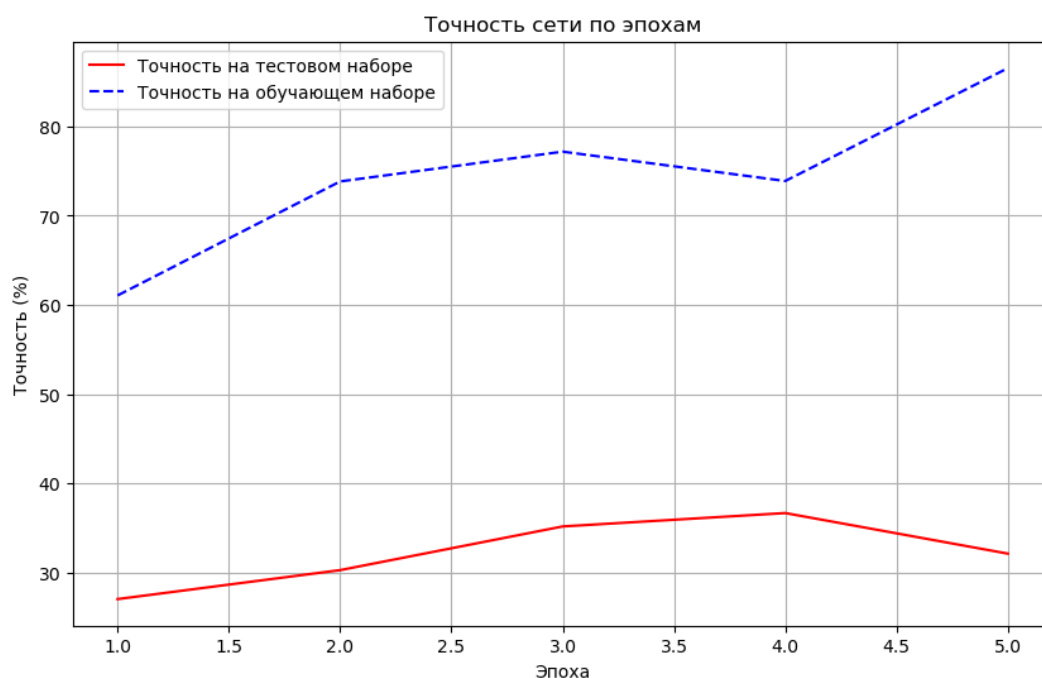


Рисунок 9 – Точность при скорости обучения 0,001

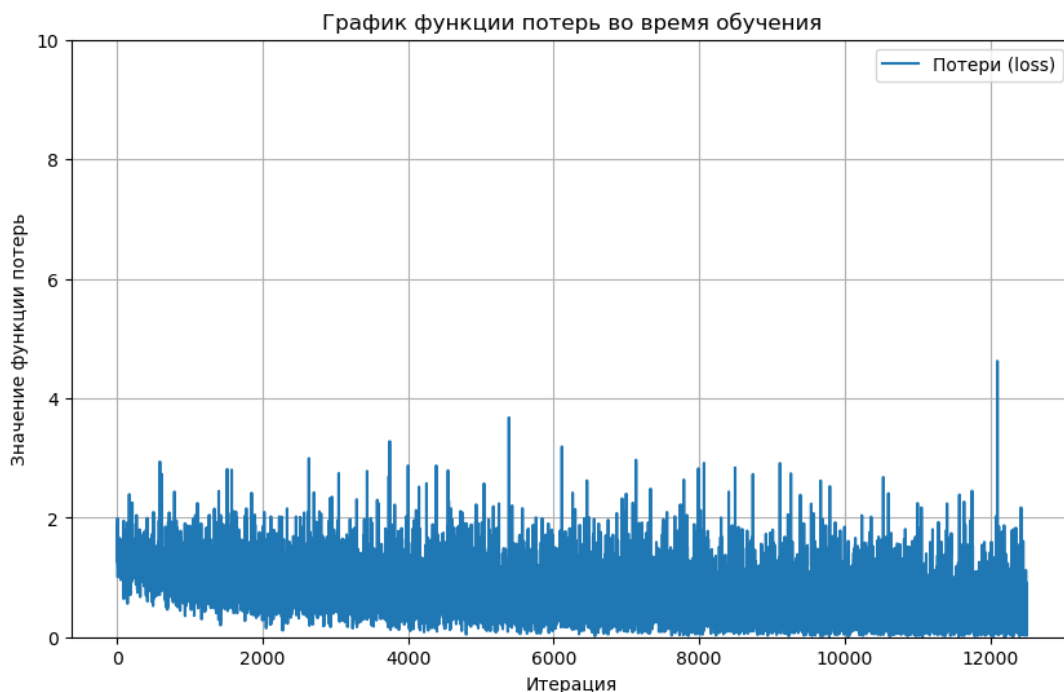


Рисунок 10 – Потери при скорости обучения 0,001

Выбросы несколько уменьшились, но всё равно остаются неприемлемо высокими. Также видно переобучение – точность на обучающей выборке растёт, на тестовой – падает.

Уменьшим скорость обучения до 0,00001, а функцию активации заменим на LeakyReLU (рисунок 11). Результаты – на рисунках 12 и 13.

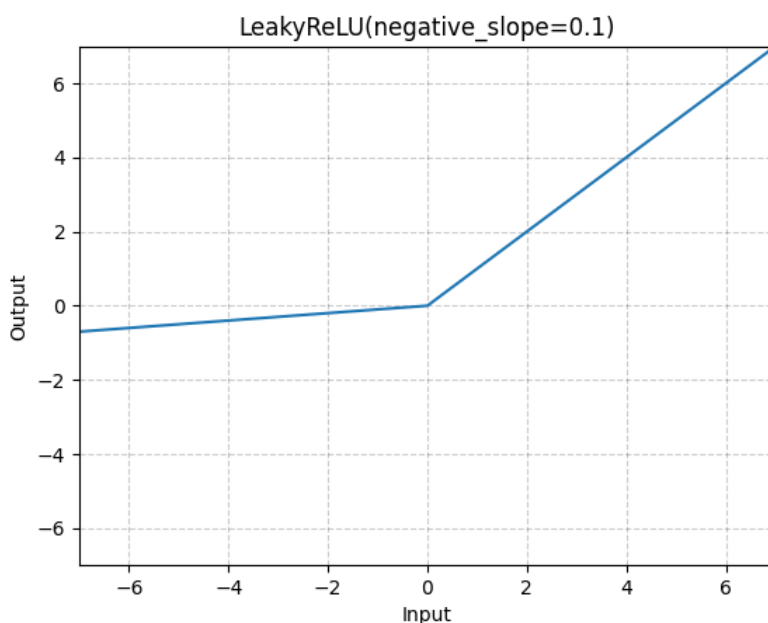


Рисунок 11 – График функции LeakyReLU

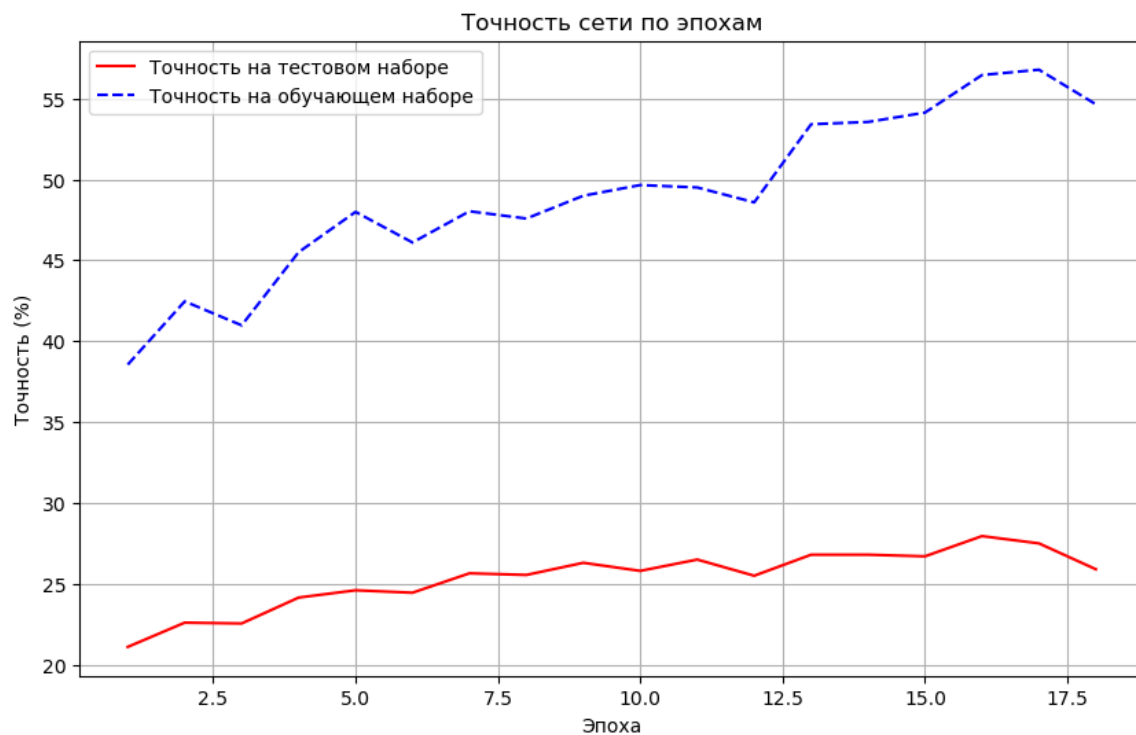


Рисунок 12 – Точность при скорости обучения 0,00001

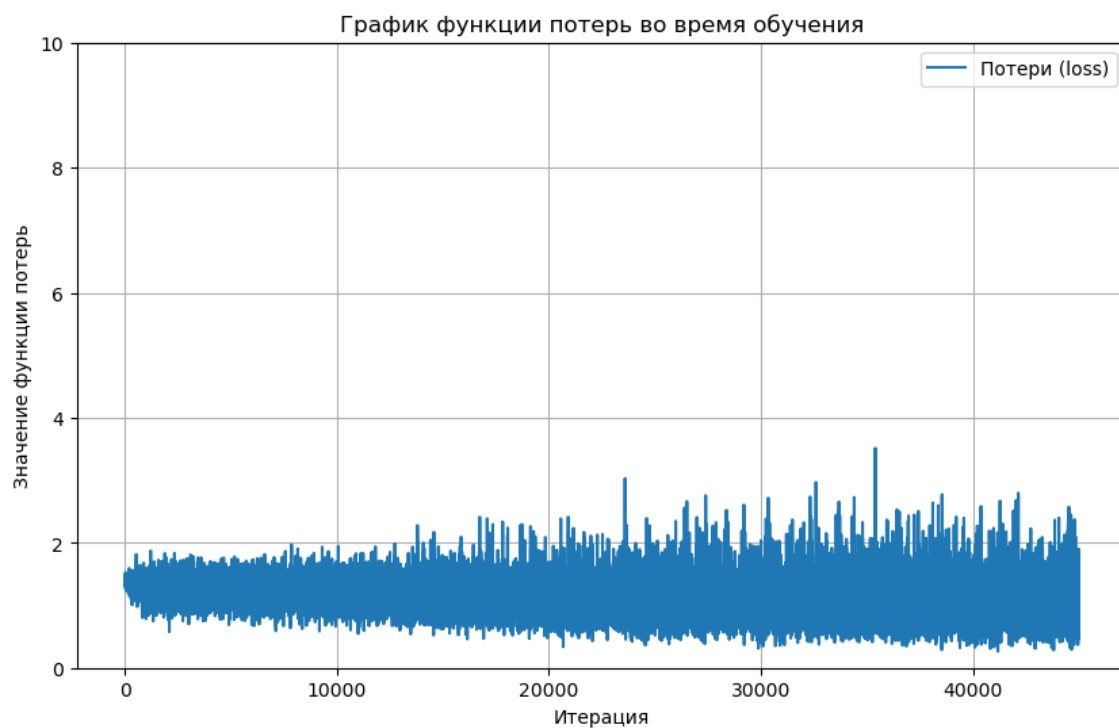


Рисунок 13 – Потери при скорости обучения 0,00001

Модель обучается крайне медленно, ошибка со временем увеличивается. Вернём скорость обучения 0,001, добавим на внутренние слои нормализацию. Результаты – на рисунках 14 и 15.

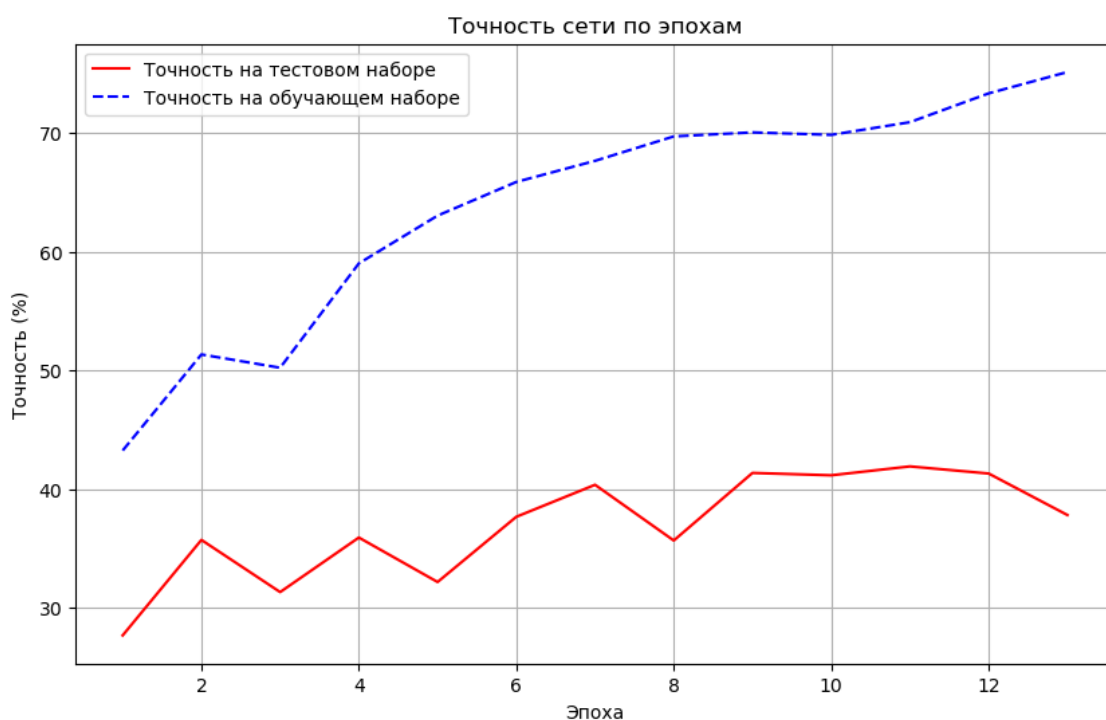


Рисунок 14 – Точность с нормализацией

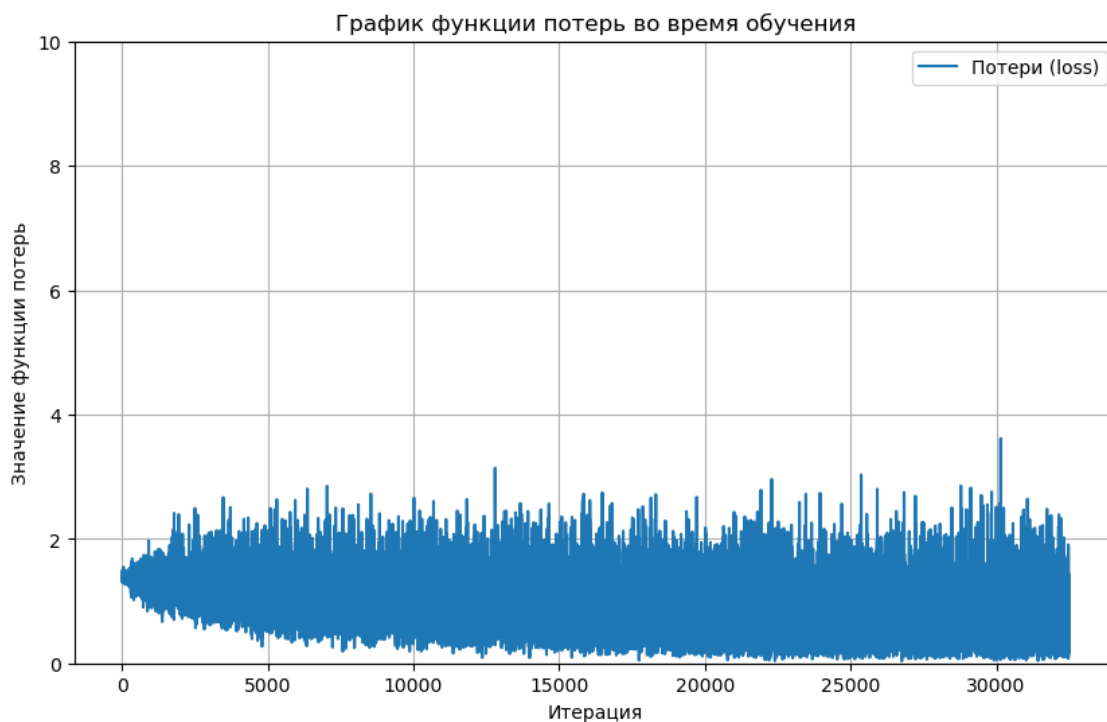


Рисунок 15 – Потери с нормализацией

Картина практически не меняется. Попробуем изменить оптимизатор на Adam (адаптивная оценка импульса) [8]. Результаты – на рисунках 16 и 17.

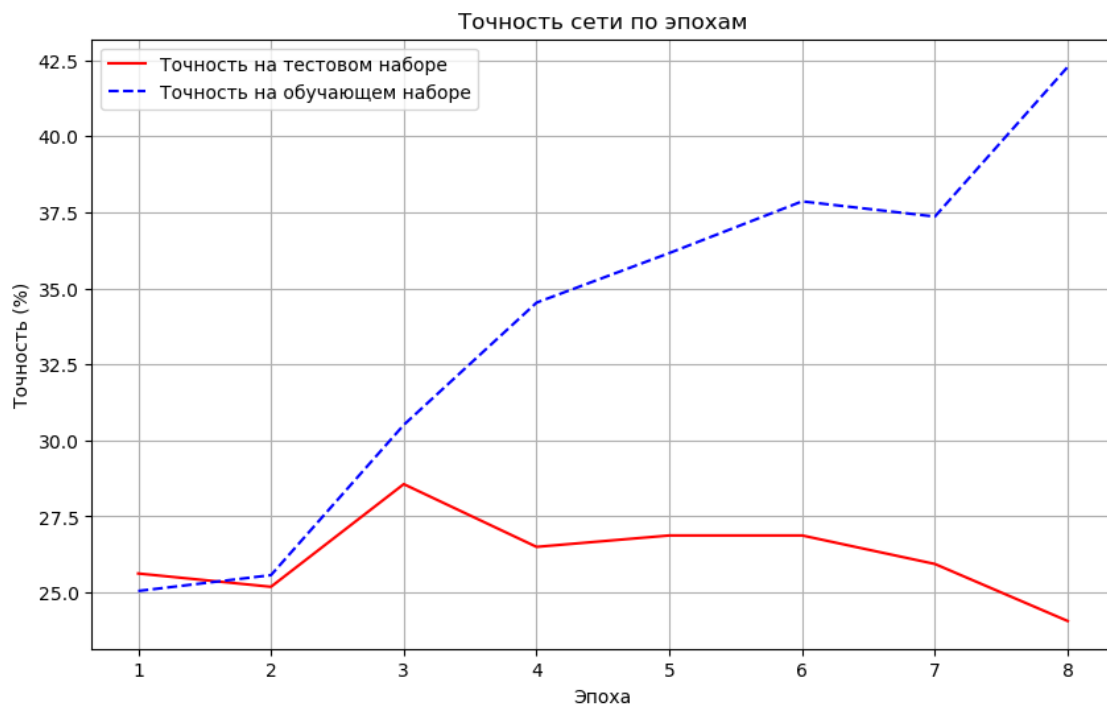


Рисунок 16 – Точность с нормализацией Adam

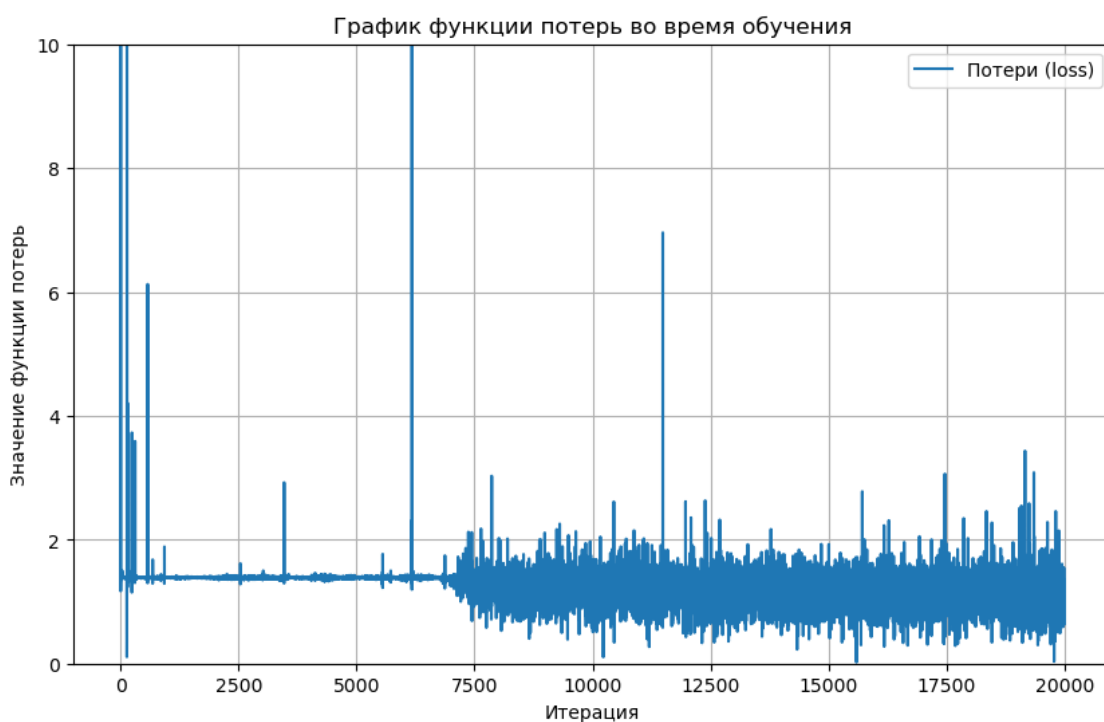


Рисунок 17 – Выбросы функции потерь при нормализации Adam

Очень быстро наступает ярко выраженное переобучение. Для борьбы с ним уменьшим разрешение изображения на входе до 64x64, соответственно уменьшим и количество нейронов во всех слоях.

На графике функции потерь наблюдаются выбросы, затем начинаются скачки. Для борьбы с выбросами функции потерь увеличим размер пакета до 40 [9]. Зададим l2-регуляризацию со штрафом в 0,001 для борьбы с переобучением и шумом.

Нейросеть показала результат в 59% на тестовой выборке при 63% при обучающей. Кривая обучения представлена на рисунке 18, функция потерь – на рисунке 19.

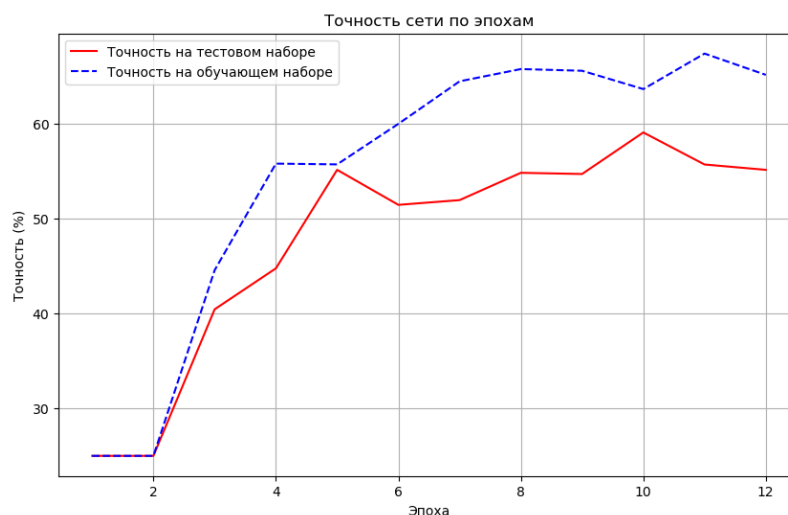


Рисунок 18 – Точность после внесённых изменений

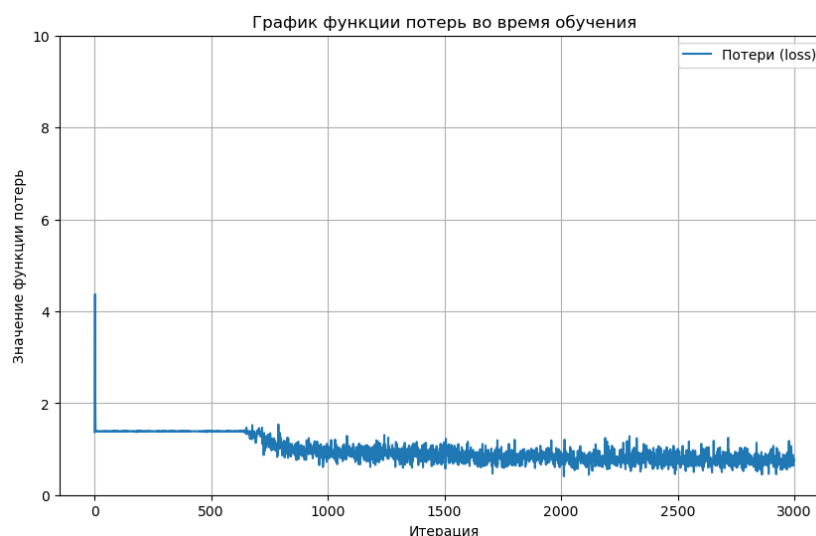


Рисунок 19 – Функция потерь после внесённых изменений

Точность в 59% неудовлетворительна. Архитектура, применяемая для распознавания символов, плохо подходит для распознавания жестов.

2.1.2 Mediapipe со слоем классификации

Попробуем использовать библиотеку mediapipe. Её модуль hands содержит предобученную нейросеть, позволяющую получать по изображению 21 точку, описывающую кисть руки в экранных или мировых координатах. Эти точки будем подавать на полносвязный классификационный слой нейронов, который будем обучать. Результатом работы этого слоя будет один из 4 искомых классов: «камень», «ножницы», «бумага», «ничего».

Разработчики mediapipe так описывают свой алгоритм: «Для обнаружения исходных положений рук мы разработали однопроходную модель детектора, оптимизированную для мобильного использования в реальном времени, по аналогии с моделью обнаружения лиц в MediaPipe Face Mesh. Обнаружение рук – это несомненно сложная задача: наша облегченная модель и полная модель должны работать с различными размерами рук в большом диапазоне масштабов (~20x) относительно кадра изображения и быть способны обнаруживать перекрытые и самоперекрытые руки. В то время как лица имеют высококонтрастные особенности, например, в области глаз и рта, отсутствие таких особенностей у рук делает их надежное обнаружение только по их визуальным особенностям сравнительно трудным. Вместо этого, предоставление дополнительного контекста, такого как особенности руки, тела или человека, способствует точной локализации рук.

Наш метод решает вышеупомянутые проблемы, используя различные стратегии. Во-первых, мы обучаем детектор ладоней вместо детектора рук, поскольку оценка ограничивающих рамок жестких объектов, таких как ладони и кулаки, значительно проще, чем обнаружение рук с сочлененными пальцами. Кроме того, поскольку ладони являются более мелкими

объектами, алгоритм подавления немаксимумов хорошо работает даже для случаев самоперекрытия двух рук, таких как рукопожатия. Более того, ладони могут быть смоделированы с использованием квадратных ограничивающих рамок (якорей в терминологии машинного обучения), игнорируя другие соотношения сторон, и, следовательно, сокращая количество якорей в 3-5 раз. Во-вторых, используется экстрактор признаков типа «кодировщик-декодировщик» для учета более широкого контекста сцены даже для мелких объектов (аналогично подходу RetinaNet). Наконец, мы минимизируем фокусную функцию потерь во время обучения для поддержки большого количества якорей, возникающих из-за высокой вариативности масштаба.

Используя вышеупомянутые методы, мы достигаем средней точности 95,7% при обнаружении ладоней. Использование обычной кросс-энтропийной функции потерь и отсутствие декодировщика дает базовый результат всего 86,22%» [10].

Результаты работы модуля hands представлены на рисунках 20 и 21.

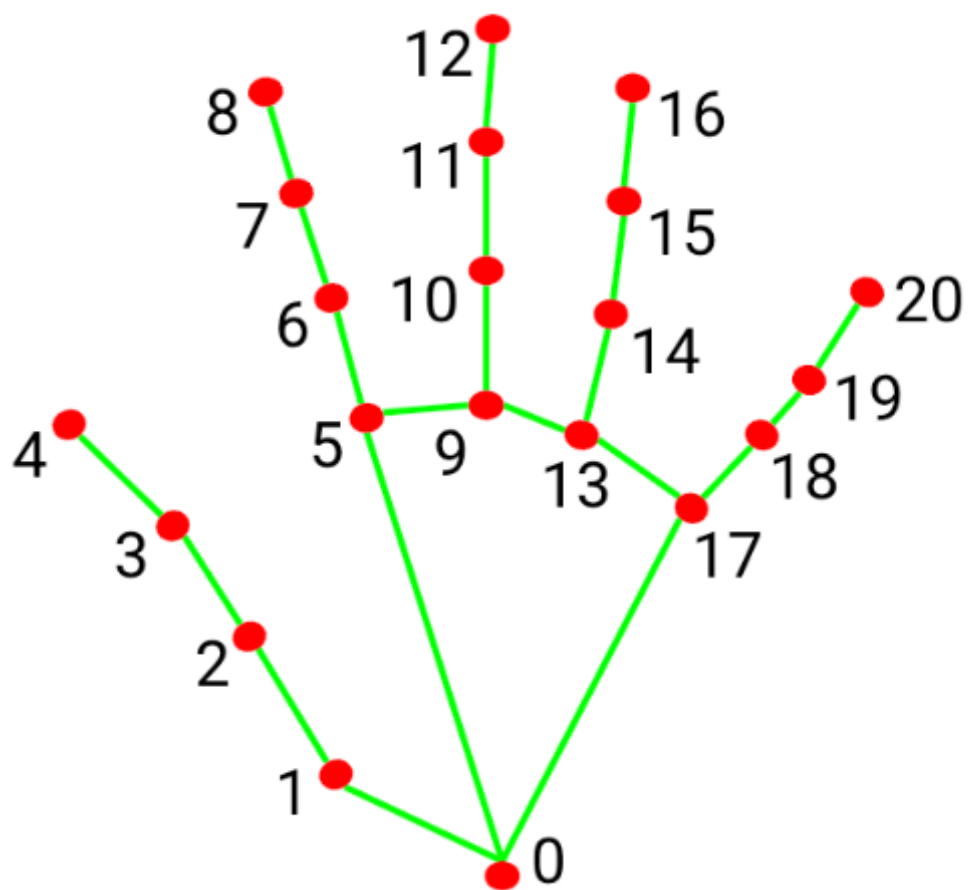


Рисунок 20 – Результат работы модуля hands

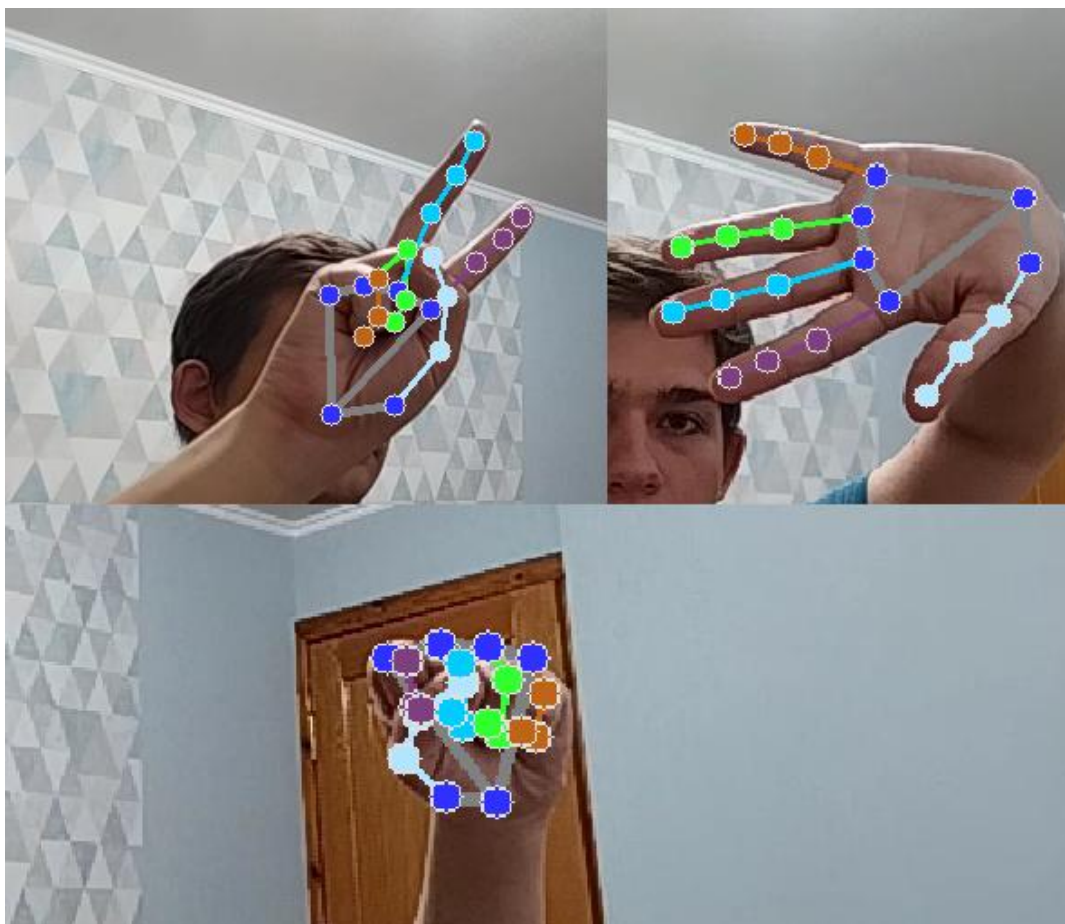


Рисунок 21 – Результат работы модуля hands, наложенный на исходные изображения

Будем использовать так называемые мировые координаты – координаты относительно центра ладони, поскольку они не зависят от положения руки в кадре, так как для нас эта информация не важна. Также будем разворачивать полученную модель руки так, чтобы отрезок между запястьем и основанием среднего пальца был направлен вдоль оси Y . Итоговая архитектура представлена на рисунке 22.

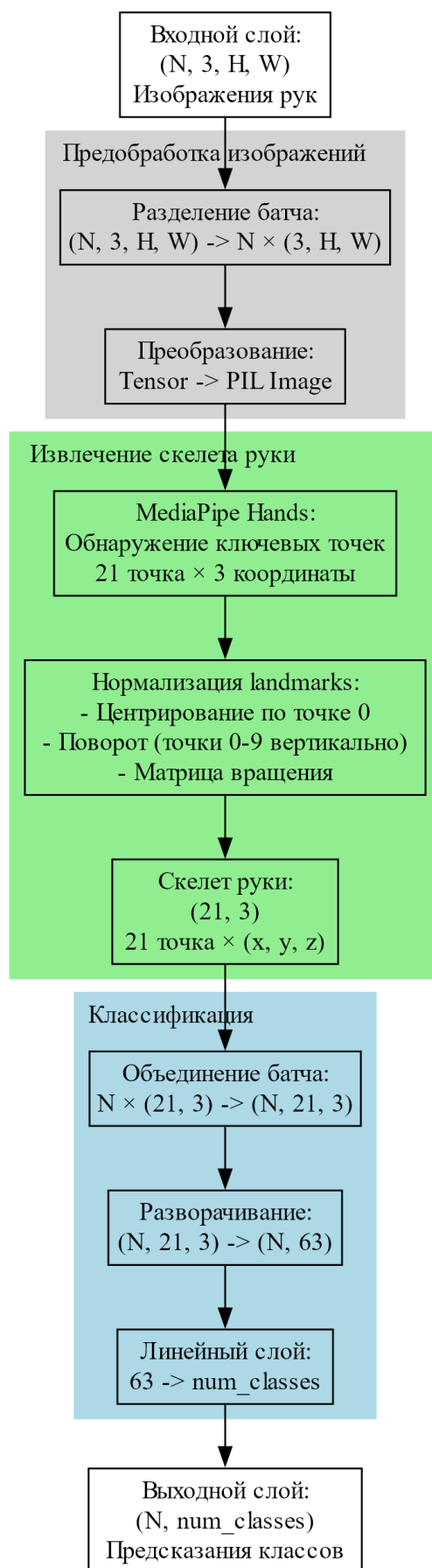


Рисунок 22 – Архитектура нейросети с использованием mediapipe

Будем обучать нейросеть эпоха за эпохой и измерять её точность на тестовой выборке, пока точность не начнёт падать.

Получили наилучшую точность 90,25% на второй эпохе, считая с нуля. К третьей эпохе точность предсказания на тестовой выборке начала падать. На рисунке 23 изображён график изменения точности нейросети.

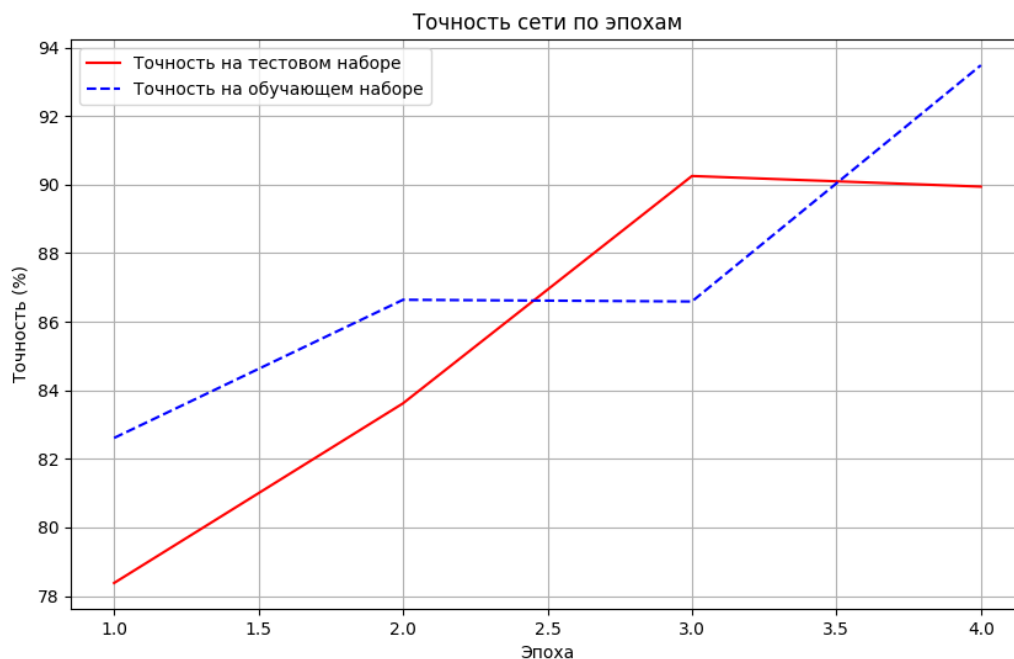


Рисунок 23 – График изменения точности нейросети с mediapipe

Такая точность является достаточной для игрового приложения. Сохраним веса данной нейросети для дальнейшего использования в игровом приложении.

2.2. Разработка игрового приложения

После того, как нейронная сеть для определения жеста игрока была спроектирована и обучена, необходимо разработать игровое приложение, с которым игрок будет при помощи неё взаимодействовать.

2.2.1. Постановка задачи

Задача заключается в разработке и реализации игры «камень-ножницы-бумага», где жест игрока распознается нейронной сетью в реальном времени, что позволяет ему соревноваться с компьютерным оппонентом. Предполагается следующий интерфейс и сценарий взаимодействия: экран

приложения будет разделен вертикально. Левая половина предназначена для вывода видео с веб-камеры игрока. Под видеопотоком расположится кнопка «старт», инициирующая игровую последовательность. После нажатия этой кнопки над ней начнется обратный отсчет. По его окончании под видеоизображением в левой части появится надпись «камень», «ножницы» или «бумага», указывающая распознанный жест игрока. Одновременно на правой части экрана отобразится изображение жеста, выбранного компьютером, сопровождаемое соответствующей надписью. Над этими панелями появится надпись «победил игрок» или «победил компьютер», объявляющая результат раунда. Повторное нажатие кнопки сбросит игру и начнёт отсчёт для следующего раунда.

После разработки нейросети следующим этапом будет интеграция модели в игровое приложение – реализация прямого взаимодействия и оценка производительности в реальных условиях.

Результатом проекта должно стать полностью функциональное игровое приложение «камень-ножницы-бумага», соответствующее описанному интерфейсу и механике игры. Помимо самой игры, целью является критическая оценка практической эффективности нейронной сети для распознавания жестов в данном контексте. Будут проанализированы ключевые показатели:

- устойчивость системы распознавания жестов в условиях недостаточного освещения,
- задержка обработки между истечением таймера и отображением результата.

2.2.2. Проектирование приложения

Приложение будет состоять из трёх модулей: GUI.py, game.py и detector.py. В модуле GUI.py будет содержаться та часть программы, которая отвечает за отрисовку пользовательского интерфейса; в модуле game.py –

игровая логика, в модуле detector.py – код для работы с камерой и обученной на предыдущих этапах работы нейросетью.

Диаграмма классов, их распределение по модулям показано на рисунке 24.

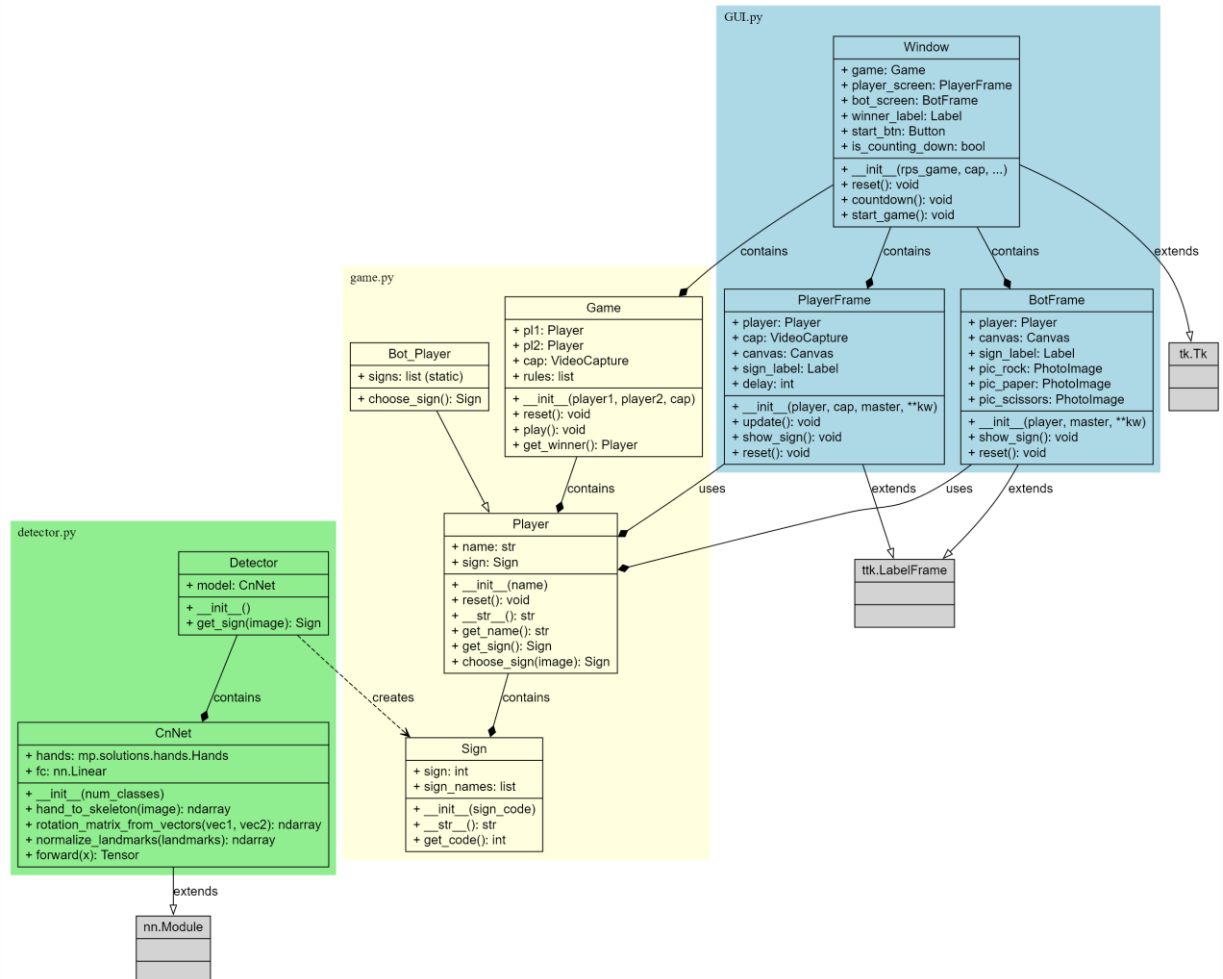


Рисунок 24 – Диаграмма классов игрового приложения

Разработанная структура модуля game.py включает четыре основных класса: Sign, Player, Bot_Player и Game, каждый из которых выполняет специфические функции в системе.

Класс Sign:

```

class Sign:
    def __init__(self, sign_code):
        self.sign_names = ["камень", "бумага", "ножницы",
                           "нет жеста"]
        self.sign = sign_code
    def __str__(self):

```

```

        return self.sign_names[self.sign]
    def get_code(self):
        return self.sign

```

Класс Sign инкапсулирует концепцию жеста. Он инициализируется числовым кодом, соответствующим определенному жесту. Также в этом классе есть список, содержащий названия жестов: "камень", "бумага", "ножницы" и "нет жеста". Текстовое представление жеста формируется методом `__str__`, возвращающим соответствующую числовому коду строку из списка названий. Этот метод вызывается при попытке приведения типа экземпляра этого класса к строковому, а также при форматированном выводе экземпляра. Также есть вспомогательная функция `get_code`, которая извлекает числовой код, облегчая дальнейшую обработку в конвейере данных.

Класс Player:

```

class Player:
    def __init__(self, name):
        self.name = str(name)
        self.sign = Sign(NO_SIGN)
    def reset(self):
        self.sign = Sign(NO_SIGN)
    def __str__(self):
        return self.name + ": " + str(self.sign)
    def get_name(self):
        return self.name
    def get_sign(self):
        return self.sign
    def choose_sign(self, image):
        self.sign = detector.Detector().get_sign(image)
        return self.sign

```

Класс Player моделирует участника игры. При инициализации сохраняется имя игрока, а жест устанавливается по умолчанию как «нет жеста» через экземпляр класса Sign. Функциональность сброса состояния

жеста обеспечивается специальным методом `reset`. При выполнении хода метод `choose_sign` принимает входное изображение, захваченное камерой и посылает его в модуль детектора. Внешний модуль используется для определения жеста, после чего обновляется атрибут жеста игрока. Предоставлены методы доступа: `get_name` возвращает имя игрока, `get_sign` – текущий жест, а дополнительный метод преобразования в строку `__str__` конкатенирует эти данные в читаемый формат.

Класс `Bot_Player`:

```
class Bot_Player(Player):
    signs = [ROCK, PAPER, SCISSORS]
    def choose_sign(self):
        self.sign = Sign(choice(self.signs))
        return self.sign
```

Класс `Bot_Player` расширяет класс `Player`, добавляя автоматизированное поведение, характерное для компьютерного противника. В этом подклассе определен массив допустимых кодов жестов (камень, бумага, ножницы) как переменная уровня класса. Метод выбора жеста переопределен для игнорирования любых входных данных с видеокamеры, вместо этого он осуществляет случайный выбор из доступных кодов жестов. После выбора случайного кода создается экземпляр `Sign`, обеспечивая соответствие решения компьютера правилам игры. Таким образом, противник играет честно и не пытается подстраиваться под ход игрока.

Класс `Game`:

```
class Game:
    def __init__(self, player1, player2, cap):
        self.pl1 = player1
        self.pl2 = player2
        self.cap = cap

        #rock          paper          scissors
no_sign
        self.rules = [[None,          self.pl2,          self.pl1,
self.pl1], #rock
```

```

                                [self.pl1,      None,      self.pl2,
self.pl1], #paper
                                [self.pl2,      self.pl1,      None,
self.pl1], #scissors
                                [self.pl2,      self.pl2,      self.pl2,
None]]      #no_sign
    def reset(self):
        self.pl1.reset()
        self.pl2.reset()
    def play(self):
        success = False
        while not success:
            success, image = self.cap.read()
            image = cv2.flip(image, 1)
            self.pl1.choose_sign(image)
            self.pl2.choose_sign()
    def get_winner(self):
        sign1 = self.pl1.get_sign().get_code()
        sign2 = self.pl2.get_sign().get_code()
        winner = self.rules[sign1][sign2]
        return winner

```

Центральным в приложении является класс Game, который координирует общий игровой процесс между человеком-игроком и компьютерным противником. При конфигурации этот класс принимает два экземпляра игроков (человек и бот) и объект захвата видео, отвечающий за получение изображения с камеры. Внутри класса реализована двумерная матрица, определяющая правила игры. Эта матрица указывает, какой игрок объявляется победителем на основе комбинаций жестов, выбранных участниками. Метод сброса инициализирует жесты обоих игроков для последующих раундов.

Метод игрового процесса play в классе Game включает цикл, совершающий попытки получить кадры с камеры до подтверждения успешного захвата. После получения изображения функция преобразования

изображения переворачивает кадр горизонтально, выравнивая его с типичными зеркальными представлениями, используемыми в системах распознавания жестов. После предварительной обработки изображения выполняются методы выбора жестов для обоих игроков (человека и бота), обновляя их соответствующие жесты. Наконец, функция определения победителя `get_winner` извлекает числовые коды из экземпляров жестов игроков и обращается к предопределённой матрице правил. Результат, возвращаемый этой функцией, идентифицирует победившего игрока на основе установленных правил взаимодействия.

Код в модуле `detector.py` предназначен для захвата видео и распознавания на нём жестов руки. Он использует несколько ключевых библиотек: `cv2` (OpenCV) для захвата и предварительной обработки изображений, `mediapipe` для сложного детектирования ключевых точек руки, `torchvision.transforms` для манипуляций с данными изображения, `numpy` для эффективных числовых операций, а также `torch` и `torch.nn` для построения и запуска нейронной сети. Также импортируется модуль игры `game`, содержащий определения игровых состояний и представлений жестов. Определены глобальные константы, такие как `NUM_CLASSES` (установлено в 4, обозначая камень, бумагу, ножницы и состояние "нет жеста") и `PATH` (имя файла с предобученными весами нейронной сети). Переменная `device` – дескриптор устройства, на котором выполняется инференс нейросети. Она настраивается на использование GPU с поддержкой CUDA при наличии такового, иначе откатывается на CPU, обеспечивая оптимальную производительность для тензорных вычислений.

Архитектура представлена в рамках двух основных классов: `CnNet` и `Detector`.

Класс `CnNet`:

```
class CnNet(nn.Module):  
    def __init__(self, num_classes=10):  
        nn.Module.__init__(self)
```

```

        self.hands =
mp.solutions.hands.Hands(static_image_mode=True,
                           max_num_hands=1,
                           model_complexity=1)
        self.fc = nn.Linear(21*3, num_classes)
        def hand_to_skeleton(self, image):
            open_cv_image = cv2.cvtColor(np.array(image),
cv2.COLOR_RGB2BGR)
            points = np.array(np.zeros((21, 3)),
dtype=np.float32)
            results =
self.hands.process(cv2.cvtColor(open_cv_image,
cv2.COLOR_BGR2RGB))
            if results.multi_hand_world_landmarks:
                for hand_world_landmarks in
results.multi_hand_world_landmarks:
                    points =
self.normalize_landmarks(hand_world_landmarks)
            return points
        def rotation_matrix_from_vectors(self, vec1, vec2):
            a = vec1 / np.linalg.norm(vec1)
            b = vec2 / np.linalg.norm(vec2)
            v = np.cross(a, b)
            c = np.dot(a, b)
            s = np.linalg.norm(v)
            if s == 0:
                return np.eye(3)
            kmat = np.array([[0, -v[2], v[1]],
                           [v[2], 0, -v[0]],
                           [-v[1], v[0], 0]])
            rotation = np.eye(3) + kmat + (kmat @ kmat) * ((1 -
c) / (s ** 2))
            return rotation
        def normalize_landmarks(self, landmarks):
            inner_landmarks = []

```

```

        for landmark in landmarks.landmark:
            inner_landmarks.append([landmark.x, landmark.y,
landmark.z])

        points = np.array(inner_landmarks, dtype=np.float32)
        v = points[9] - points[0]
        v /= np.linalg.norm(v)
        up = np.array([0, -1, 0])
        rot_mat = self.rotation_matrix_from_vectors(v, up)
        centered_points = points - points[0]
        rotated_points = centered_points @ rot_mat.T
        return rotated_points

    def forward(self, x):
        batch_size = x.size(0)
        outputs = []
        for i in range(batch_size):
            img = x[i]
            img = img.cpu().numpy()
            img = img.transpose(1, 2, 0)
            img = transforms.ToPILImage()(img)    #
преобразуем в PIL Image
            skeleton = self.hand_to_skeleton(img)    #
получаем скелет руки
            skeleton = torch.tensor(skeleton,
dtype=torch.float32).to(device)
            outputs.append(skeleton)
        outputs = torch.stack(outputs, dim=0)
        outputs = outputs.view(outputs.size(0), -1)
        outputs = self.fc(outputs)
        return outputs

```

Класс CnNet, наследуемый от `torch.nn.Module`, определяет структуру нейронной сети. Его конструктор `__init__` инициализирует базовый модуль и настраивает ключевые компоненты. Создается экземпляр `mediapipe.solutions.hands.Hands`, сохраняемый как `self.hands`, сконфигурированный для статического анализа изображений,

детектирования одной руки и с заданной сложностью модели, равной 1; это движок для локализации ключевых точек руки. Критически важной частью сети является полносвязный слой `self.fc`, предназначенный для приема сжатого вектора из 21 трехмерных координат ключевых точек руки (21 точка по 3 координаты - всего 63 признака) и отображения их на `NUM_CLASSES` выходных значений, представляющих вероятность каждого жеста.

Несколько методов внутри `CnNet` обрабатывают преобразование необработанных изображений в формат, пригодный для нейронной сети. Метод `hand_to_skeleton` принимает изображение `PIL` на вход. Сначала оно конвертируется в цветовой формат `BGR`, ожидаемый `OpenCV`. Затем вызывается `self.hands.process` для детектирования ключевых точек руки на изображении. Если точки успешно детектированы (результаты `results.multi_hand_world_landmarks` не пусты), запускается утилита `normalize_landmarks`. Метод `normalize_landmarks` – сложный инструмент. Он получает необработанные данные ключевых точек, преобразует их в массив `NumPy` и выполняет ряд геометрических преобразований. Ключевая операция включает вычисление вектора между определенными точками (от запястья до основания среднего пальца) и затем определение матрицы вращения с помощью `rotation_matrix_from_vectors`, чтобы выровнять этот вектор с вертикальным направлением (вдоль оси `Y`). Этот шаг нормализации, также центрирующий точки относительно запястья, призван сделать представление жеста инвариантным к ориентации и положению руки, обеспечивая более стабильный вход для нейронной сети. Вспомогательная функция `rotation_matrix_from_vectors` вычисляет необходимую 3D-матрицу вращения для двух векторов. Наконец, метод `forward` управляет процессом предсказания для пакета входных изображений `x`. Он итерирует по каждому изображению в пакете, конвертирует тензор `PyTorch` в изображение `PIL` и вызывает `hand_to_skeleton` для извлечения нормализованных 3D-координат ключевых точек. Эти координаты возвращаются в тензор `PyTorch`, и все

обработанные скелеты из пакета объединяются. Затем полученный тензор разворачивается (линеаризуется) и пропускается через слой `self.fc`, формируя окончательные оценки классификации.

Класс `Detector`:

```
class Detector:
    def __init__(self):
        self.model = CnNet(NUM_CLASSES).to(device)
        self.model.load_state_dict(torch.load(PATH,
weights_only=True, map_location=device))
        self.model.eval()

    def get_sign(self, image: np.ndarray) -> int:
        sign_list = [game.NO_SIGN, game.PAPER, game.ROCK,
game.SCISSORS]
        image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        image_tensor =
torch.from_numpy(image_rgb.astype(np.float32) / 255.0)
        image_tensor = image_tensor.permute(2, 0, 1)
        image_batch = image_tensor.unsqueeze(0).to(device)
        with torch.no_grad():
            outputs = self.model(image_batch)
            _, predicted_class = torch.max(outputs.data, 1)
        return game.Sign(sign_list[predicted_class.item()])
```

Класс `Detector` служит высокоуровневым интерфейсом для использования модели `CnNet`. Его конструктор `__init__` создает экземпляр модели `CnNet`, переносит ее на соответствующее устройство и загружает предобученные веса из файла, указанного в `PATH`, с помощью `torch.load_state_dict`. Критически важно, что `self.model.eval()` устанавливает сеть в режим оценки, отключая механизмы, такие как `dropout`, активные только во время обучения. Основным публичным методом этого класса – `get_sign`. Он принимает необработанное изображение (в виде массива `NumPy` в формате `BGR` из `OpenCV`). Определяется `sign_list`, сопоставляющая целочисленные индексы классов с объектами `game.Sign` (например,

game.PAPER, game.ROCK). Входное изображение предварительно обрабатывается: конвертируется из BGR в RGB, нормализуются значения пикселей, переставляются измерения из формата HWC (высота, ширина, каналы) в CHW (каналы, высота, ширина), добавляется размерность пакета и переносится на устройство. С отключенными градиентами через `torch.no_grad()` для эффективности, пакет предобработанных изображений пропускается через `self.model` (экземпляр CnNet) для получения оценок вероятности классов. `Torch.max` используется для выявления индекса класса с максимальным значением, представляющего наиболее вероятный жест. В результате метод возвращает соответствующий объект `game.Sign`, найденный по предсказанному индексу класса в `sign_list`. Это позволяет остальной логике игры легко интерпретировать выход нейронной сети.

Модуль `GUI.py` содержит классы, относящиеся к графическому интерфейсу пользователя. Также в нём содержится точка входа в основной цикл программы.

Класс `PlayerFrame.py`

```
class PlayerFrame(ttk.LabelFrame)
    def __init__(self, player, cap, master=None, **kw):
        self.s = ttk.Style()
        self.cap = cap
        self.s.configure('Red.TLabelframe.Label',
font=FONT_SIZE)
        super().__init__(master, text = player.get_name(),
style = "Red.TLabelframe", **kw)
        self.player = player
        self.delay =
int(1000/self.cap.get(cv2.CAP_PROP_FPS))
        self.f_top = tk.Frame(self)
        self.canvas = tk.Canvas(self.f_top, width=640,
height=360, bg="black")
        self.sign_label = ttk.Label(self, text="", font =
FONT_SIZE)
```

```

        self.f_top.pack(side=tk.TOP,                        expand=True,
fill="both")

        self.canvas.pack(anchor=tk.CENTER, expand=True, padx
= "0.5m", pady = "0.5m")
        self.sign_label.pack(side=tk.TOP)
        self.update()
def update(self):
    ret, image = self.cap.read()
    if ret:
        with mp_hands.Hands(
            static_image_mode=False,
            max_num_hands=1,
            model_complexity=0,
            min_detection_confidence=0.5) as hands:
            image = cv2.cvtColor(image,
cv2.COLOR_BGR2RGB)

            results = hands.process(image)
            image.flags.writeable = True
            if results.multi_hand_landmarks:
                for hand_landmarks in
results.multi_hand_landmarks:
                    mp_drawing.draw_landmarks(
                        image,
                        hand_landmarks,
                        mp_hands.HAND_CONNECTIONS,

mp_drawing_styles.get_default_hand_landmarks_style(),
mp_drawing_styles.get_default_hand_connections_style())
                    self.photo =
ImageTk.PhotoImage(image=Image.fromarray(image))
                    self.canvas.create_image(0, 0, image=self.photo,
anchor=tk.NW)

                    self.master.after(self.delay, self.update)
    else:
        self.cap.release()

```

```

def show_sign(self):
    sign = self.player.get_sign()
    self.sign_label['text'] = str(sign)
    return
def reset(self):
    self.sign_label['text'] = ""
    self.canvas.delete("all")

```

Класс `PlayerFrame` представляет собой специализированный компонент графического пользовательского интерфейса, разработанный для визуального представления игрока в приложении игры "камень-ножницы-бумага". Он наследуется от `tk.LabelFrame` – тематизированного виджета из набора инструментов Tkinter, что позволяет ему представлять свое содержимое в рамке с границами и заголовком, содержащим имя игрока. Данный класс играет ключевую роль в отображении видеопотока в реальном времени с веб-камеры игрока, наложении обнаруженных ключевых точек руки в режиме реального времени и последующем показе распознанного жеста.

При создании экземпляра через метод `init` объект `PlayerFrame` настраивается соответствующим образом. Он получает объект игрока, объект `cap` (экземпляр захвата видео OpenCV для доступа к веб-камере) и главный виджет (его родительский элемент в иерархии GUI). Для настройки внешнего вида используется объект `tk.Style`, в частности шрифта заголовка `LabelFrame`, который динамически устанавливается на имя игрока, получаемое через `player.get_name()`. Фрейм также вычисляет значение задержки, выводимое из частоты кадров веб-камеры, получаемой из `self.cap.get(cv2.CAP_PROP_FPS)`; эта задержка управляет частотой обновления видеотображения. Внутри этого основного фрейма создается вложенный `tk.Frame` с именем `self.f_top` для целей компоновки. Внутри `self.f_top` устанавливается виджет `tk.Canvas` с именем `self.canvas` с предопределенными размерами и черным фоном. Этот холст служит основной областью для отрисовки видеок кадров. Под холстом

инициализируется виджет `ttk.Label` с именем `self.sign_label` для отображения текстового представления выбранного игроком знака, изначально пустого. Затем эти виджеты размещаются внутри `PlayerFrame` с использованием менеджера геометрии упаковки `Tkinter`. Инициализация завершается вызовом метода `update`, который запускает процесс непрерывного отображения видеопотока.

Метод `update` является сердцем механизма визуальной обратной связи. Он пытается прочесть новый кадр из объекта захвата видео `self.cap`. Если кадр успешно получен (`ret` равно `true`), происходит обработка изображения и отрисовка. Объект `mp_hands.Hands` из библиотеки `MediaPipe` инициализируется внутри контекстного менеджера. Этот объект настроен для динамического режима изображения (подходящего для видеопотоков), для обнаружения максимум одной руки, с заданной сложностью модели и минимальной достоверностью обнаружения. Захваченное BGR-изображение из `OpenCV` преобразуется в формат `RGB`, который является стандартным для `MediaPipe`. Затем на этом `RGB`-изображении вызывается метод `hands.process` для обнаружения ключевых точек руки. Если ключевые точки найдены (`results.multi_hand_landmarks` не пуст), утилита `mp_drawing.draw_landmarks` используется для рисования обнаруженных ключевых точек и их соединений непосредственно на изображении. Это дополненное изображение, теперь представляющее собой массив `NumPy`, преобразуется в объект `PIL.Image`, а затем в объект `ImageTk.PhotoImage` с именем `self.photo`. Затем это `PhotoImage` рисуется на `self.canvas` в позиции `(0,0)` с использованием `self.canvas.create_image`. Для создания эффекта непрерывного видео вызывается `self.master.after(self.delay, self.update)`, планируя выполнение метода `update` снова после вычисленной задержки. Если чтение кадра не удастся (`ret` равно `false`), устройство захвата видео освобождается через `self.cap.release()`.

Два других метода поддерживают интерактивное состояние игры. Метод `show_sign` отвечает за отображение распознанного жеста игрока. Он получает текущий знак из связанного объекта `self.player`, используя `self.player.get_sign()`, и обновляет текст `self.sign_label` для отражения этого жеста. Метод `reset` предназначен для очистки визуального отображения игрока, обычно между раундами игры. Он устанавливает текст `self.sign_label` в пустую строку и вызывает `self.canvas.delete("all")` для удаления любых рисунков (таких как последний видеокادر или ключевые точки руки) с холста.

Класс BotFrame:

```
class BotFrame(ttk.LabelFrame):
    def __init__(self, player, master=None, **kw):
        self.s = ttk.Style()
        self.s.configure('Red.TLabelframe.Label',
font=FONT_SIZE)
        super().__init__(master, text = player.get_name(),
style = "Red.TLabelframe", **kw)
        self.player = player
        self.pic_rock = tk.PhotoImage(file='rock.png')
        self.pic_paper = tk.PhotoImage(file='paper.png')
        self.pic_scissors =
tk.PhotoImage(file='scissors.png')
        self.f_top = tk.Frame(self)
        self.canvas = tk.Canvas(self.f_top, width=640,
height=360, bg="black")
        self.sign_label = ttk.Label(self, text="", font =
FONT_SIZE)
        self.f_top.pack(side=tk.TOP, expand=True,
fill="both")
        self.canvas.pack(anchor=tk.CENTER, expand=True, padx
= "0.5m", pady = "0.5m")
        self.sign_label.pack(side=tk.TOP)
    def show_sign(self):
```

```

        sign = self.player.get_sign()
        self.sign_label['text'] = str(sign)
        match sign.get_code():
            case game.ROCK:
                self.canvas.create_image(0, 0, anchor=tk.NW,
image = self.pic_rock)
            case game.SCISSORS:
                self.canvas.create_image(0, 0, anchor=tk.NW,
image = self.pic_scissors)
            case game.PAPER:
                self.canvas.create_image(0, 0, anchor=tk.NW,
image = self.pic_paper)
        return
    def reset(self):
        self.sign_label['text'] = ""
        self.canvas.delete("all")

```

Класс BotFrame представляет собой комплексный компонент графического интерфейса, отображающий ход компьютерного оппонента. Он наследуется от ttk.LabelFrame и использует тематические виджеты Tkinter для размещения содержимого внутри рамки с четко обозначенным заголовком, содержащим имя бота. Данная реализация класса выходит за рамки простого заполнителя, интегрируя функциональность визуального отображения выбранного ботом жеста с помощью предзагруженных изображений и текстовых меток.

При создании экземпляра через метод `__init__` объект BotFrame конфигурируется. Он принимает объект игрока (представляющий собой бота), родительский виджет в интерфейсе (master) и дополнительные именованные аргументы (`**kwargs`). Объект `tk.Style (self.s)` применяется для настройки шрифта заголовка LabelFrame, который динамически устанавливается с использованием `player.get_name()`. Сам объект игрока сохраняется как `self.player`. Ключевым моментом является создание трех объектов `tk.PhotoImage` – `self.pic_rock`, `self.pic_paper` и `self.pic_scissors` –

путем загрузки изображений из соответствующих файлов (rock.png, paper.png, scissors.png). Эти изображения будут использоваться для визуализации выбора бота. По аналогии с PlayerFrame, создается вложенный `tk.Frame` (`self.f_top`), содержащий холст `tk.Canvas` (`self.canvas`) фиксированного размера с черным фоном. На этом холсте будут отображаться изображения жестов. Также инициализируется метка `ttk.Label` (`self.sign_label`) для вывода текстового названия выбранного ботом жеста. Эти виджеты затем размещаются внутри макета `BotFrame`.

Метод `show_sign` отвечает за обновление отображения бота при совершении хода. Сначала из `self.player` извлекается объект жеста (`sign`) с помощью `get_sign()`. Текст метки `self.sign_label` обновляется до строкового представления этого жеста. Затем используется оператор `match` (аналогичный `switch-case`), основанный на целочисленном коде жеста, полученном через `sign.get_code()`. В зависимости от того, соответствует ли код константам `game.ROCK`, `game.SCISSORS` или `game.PAPER`, определенным в модуле `game`, соответствующее предзагруженное изображение (`self.pic_rock`, `self.pic_scissors` или `self.pic_paper`) отрисовывается на холсте `self.canvas` в позиции (0,0) с помощью `self.canvas.create_image`.

Метод `reset` подготавливает `BotFrame` к новому раунду игры. Он очищает метку `self.sign_label`, устанавливая ее текст в пустую строку, и удаляет любое ранее отображенное изображение с холста `self.canvas` путем вызова `self.canvas.delete("all")`.

Класс `Window`:

```
class Window(tk.Tk):
    def __init__(self, rps_game, cap, screenName=None,
                  baseName=None, className='Tk',
                  useTk=True, sync=False, use=None):
        super().__init__(screenName, baseName, className,
                          useTk, sync, use)
        self.s = ttk.Style()
```

```

        self.s.configure('my.TButton', font=FONT_SIZE)
        self.game = rps_game
        self.f_top = ttk.Frame(self)
        self.player_screen = PlayerFrame(self.game.pl1, cap,
self.f_top)
        self.bot_screen = BotFrame(self.game.pl2,
self.f_top)
        self.f_bot = tk.Frame(self)
        self.winner_label = ttk.Label(self.f_bot, text="",
font = FONT_SIZE)
        self.start_btn = ttk.Button(self.f_bot,
text="Crapt", style='my.TButton', command = self.start_game)
        self.f_top.pack(side=tk.TOP, expand=True,
fill="both")
        self.f_bot.pack(side=tk.BOTTOM, fill="x")
        self.player_screen.pack(side=tk.LEFT, expand=True,
fill="both", padx = "0.5m", pady = "0.5m")
        self.bot_screen.pack(side=tk.RIGHT, expand=True,
fill="both", padx = "0.5m", pady = "0.5m")
        self.winner_label.pack(side=tk.TOP)
        self.start_btn.pack(side=tk.BOTTOM, padx = "0.5m",
pady = "0.5m")
        self.is_counting_down = False
    def reset(self):
        self.game.reset()
        self.bot_screen.reset()
        self.player_screen.reset()
        self.winner_label['text'] = ""
    def countdown(self):
        for i in range(3, 0, -1):
            self.winner_label.config(text=str(i))
            time.sleep(1)
        self.game.play()
        self.bot_screen.show_sign()
        self.player_screen.show_sign()

```

```

try:
    self.winner_label["text"] = (f'Победил
{self.game.get_winner().get_name()}')
except AttributeError:
    self.winner_label["text"] = "Ничья"
self.start_btn.config(state=tk.NORMAL)
self.is_counting_down = False
def start_game(self):
    self.reset()
    if not self.is_counting_down:
        self.is_counting_down = True
        self.start_btn.config(state=tk.DISABLED)
        self.countdown_thread =
threading.Thread(target=self.countdown)
        self.countdown_thread.start()

```

Класс Window, производный от tk.Tk, служит основой графического пользовательского интерфейса для игры «камень-ножницы-бумага». Он инициализирует игровое окно, организует различные компоненты и взаимодействует с логикой управления игрой. Основная цель этого класса заключается в создании интерактивной среды, где игроки могут участвовать в матче против бота и видеть результаты в режиме реального времени.

При инициализации настраиваются несколько ключевых элементов. Класс наследуется от tk.Tk, что предоставляет доступ к функциям управления окнами Tkinter. Настраивается стиль для кнопок, обеспечивая их соответствие единой визуальной теме. Игровая логика связывается с объектом `gps_game`, переданным во время создания экземпляра, и создаются два различных фрейма: `f_top` для экранов игрока и бота и `f_bot` для отображения победителя и элементов управления игрой. Состояния игры игрока и бота отображаются с помощью классов `PlayerFrame` и `BotFrame` соответственно, которые связаны с их соответствующими игровыми объектами. Эти фреймы размещаются в главном окне, обеспечивая постоянную видимость информации об игроке и боте бок о бок.

В нижнем фрейме расположена кнопка «Старт», предназначенная для начала игры. При нажатии эта кнопка запускает метод `start_game`, который сбрасывает состояние игры, отключает кнопку для предотвращения многократных нажатий и начинает обратный отсчет перед началом матча. Обратный отсчет обрабатывается в отдельном потоке для поддержания отзывчивости графического интерфейса. Этот обратный отсчет длится три секунды, причем число уменьшается каждую секунду, и визуально представляется обновлением метки `winner_label`.

После завершения обратного отсчета основная логика игры выполняется через метод `play` объекта `rps_game`. Отображаются жесты бота и игрока, и определяется победитель. Результат отображается на метке `winner_label`, причем имя победителя отображается, если есть победитель, или "Ничья", если победителя нет. После окончания матча кнопка "Старт" снова включается, чтобы пользователь мог начать новую игру.

Класс также включает метод `reset`, который сбрасывает состояние игры, очищая все предыдущие результаты и позволяя начать заново. Это гарантирует, что как игрок, так и бот начинают каждый матч с чистого листа, причем визуальные элементы отразят сброшенное состояние.

Дизайн класса `Window` ориентирован на простоту и ясность, он обеспечивает интуитивно понятный пользовательский интерфейс при сохранении основных функциональных возможностей, необходимых для игры.

Инициализация и основной цикл программы:

```
cap = cv2.VideoCapture(0)
rps_game = game.Game(game.Player("Игрок"),
game.Bot_Player("Компьютер"), cap)
root = Window(rps_game, cap)
root.title('Камень-ножницы-бумага')
root.mainloop()
```

В этом же модуле содержится точка входа, при входе в которую (то есть при запуске приложения) создаётся дескриптор видеокамеры, игра, игроки, окно пользовательского интерфейса. Затем запускается основной цикл.

На рисунке 25 представлен вид окна приложения с запущенным обратным отсчётом.

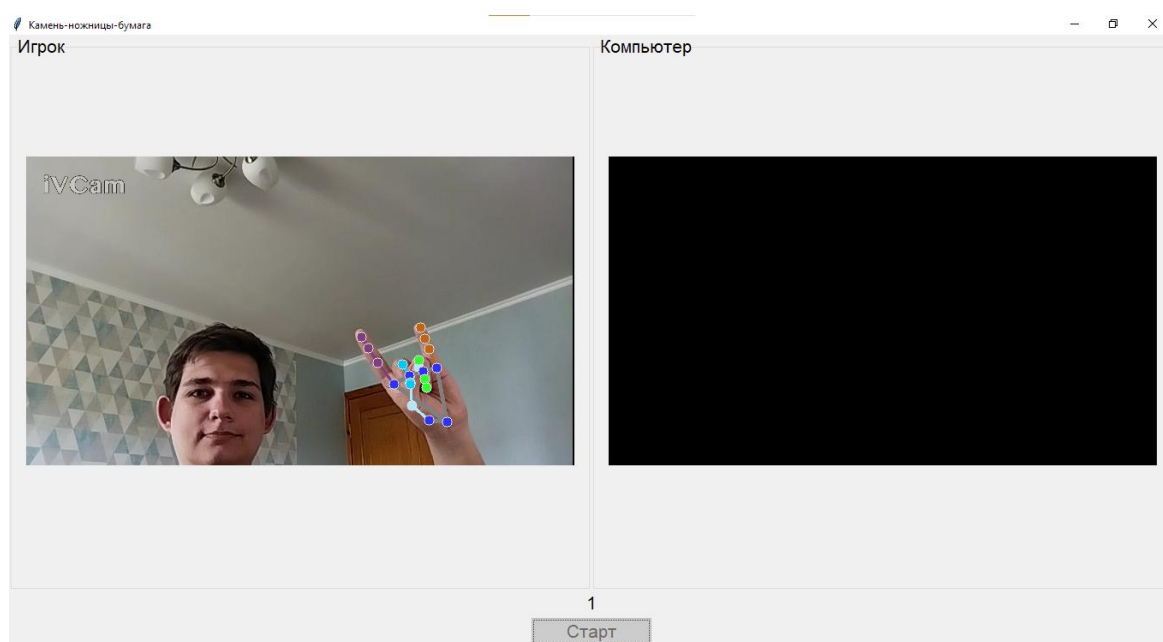


Рисунок 25 – Работа игрового приложения

3 Тестирование

Поскольку результаты работы программы могут сильно зависеть от оборудования, на котором она работает, укажем, что обработка видеопотока происходила на процессоре AMD FX-8350, в качестве веб-камеры использовалась Samsung S5KGM1, установленная на подключенный к компьютеру мобильный телефон.

Для того, чтобы проверить программу, будем демонстрировать жесты в условиях плохого освещения на неоднородном слабоконтрастном фоне. Демонстрировать жесты будем как правой, так и левой рукой.

На рисунках 26 – 32 изображён процесс тестирования приложения.

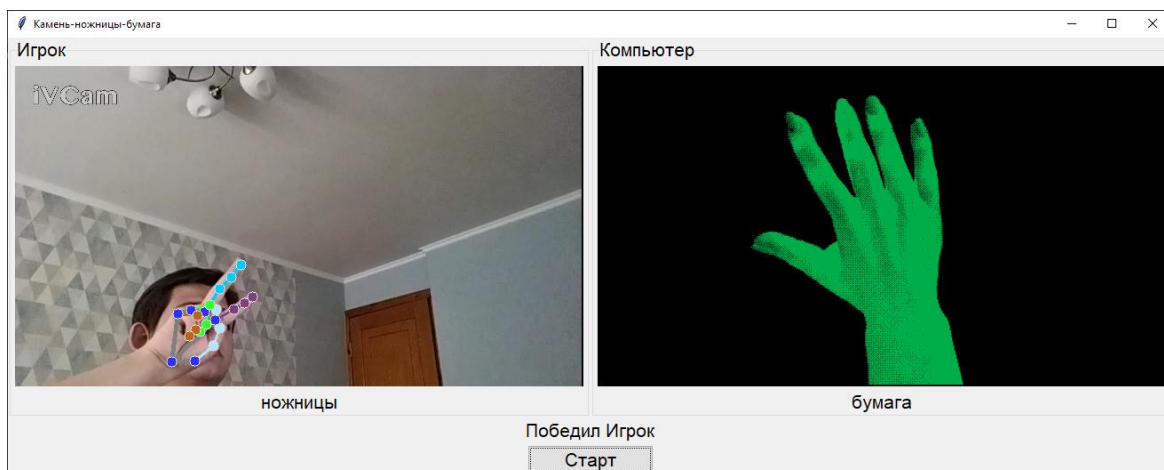


Рисунок 26 – «Ножницы» правой рукой

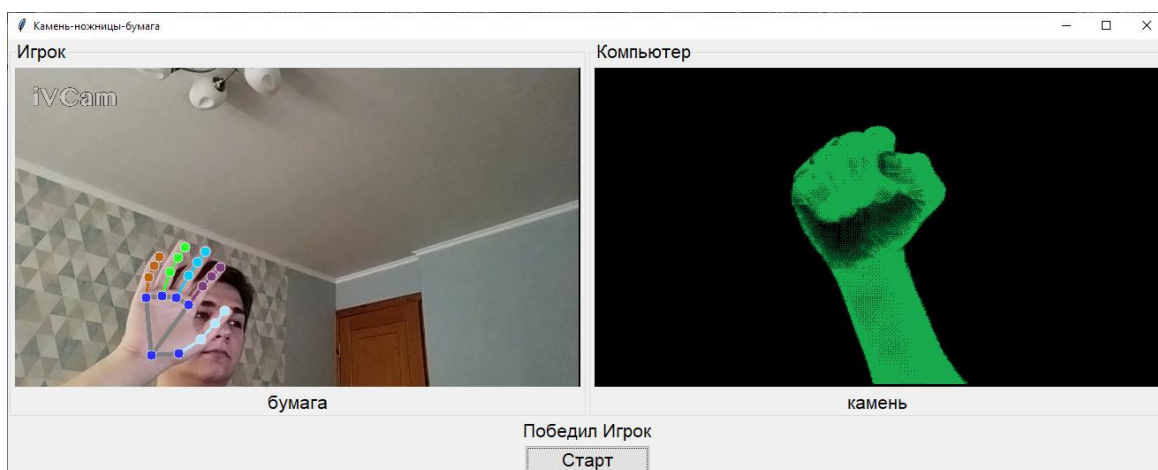


Рисунок 27 – «Бумага» правой рукой

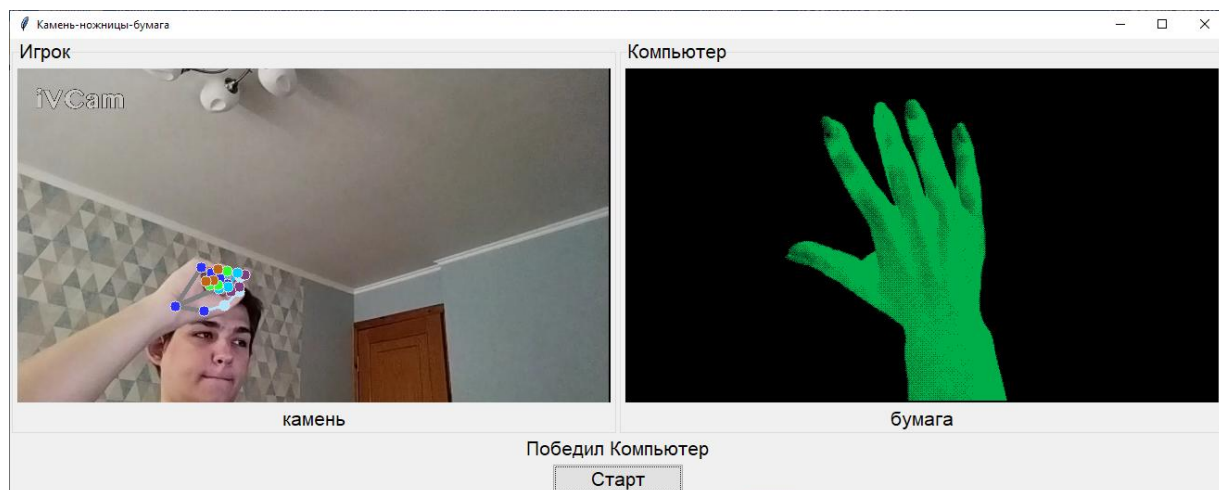


Рисунок 28 – «Камень» правой рукой

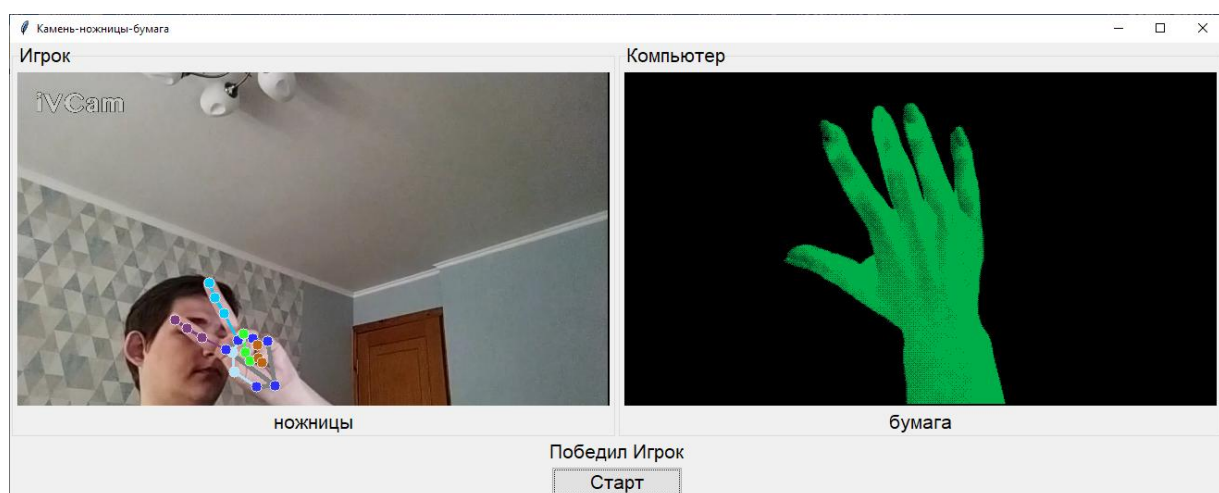


Рисунок 29 – «Ножницы» левой рукой

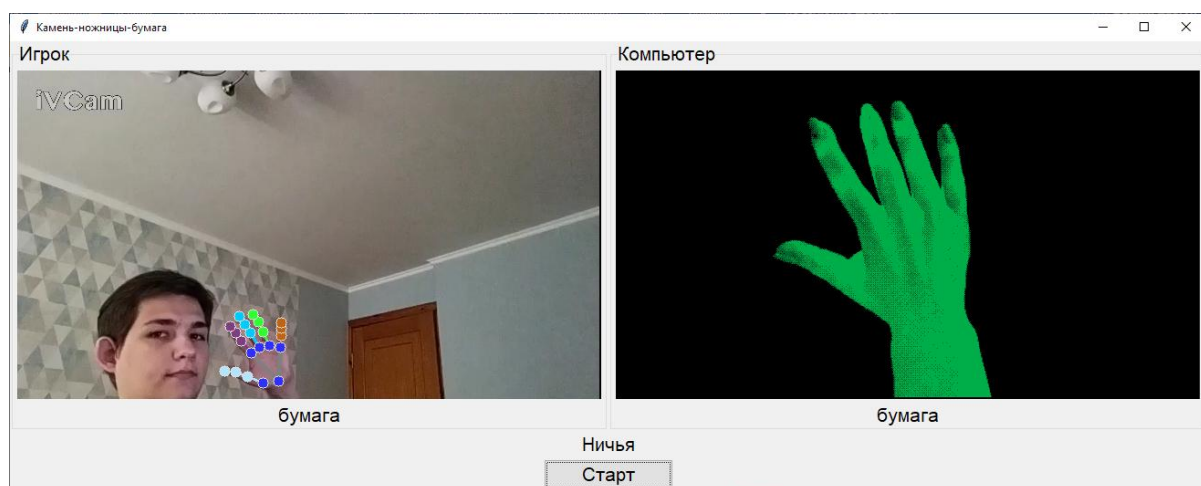


Рисунок 30 – «Бумага» левой рукой

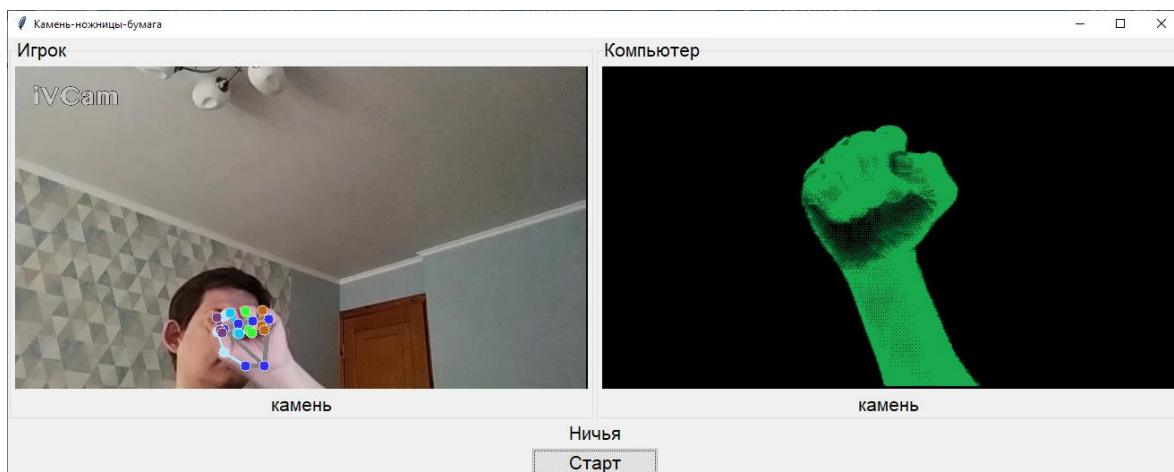


Рисунок 31 – «Камень» левой рукой

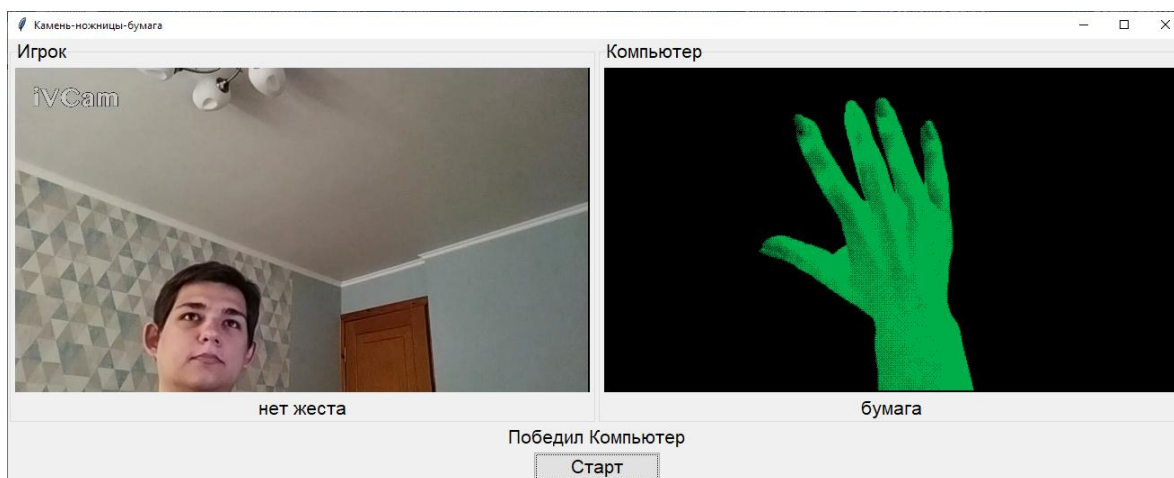


Рисунок 32 – Отсутствие жеста

Программа верно распознала все представленные жесты и правильно определила победителей во всех раундах. При этом заметной задержки после завершения обратного отсчёта не возникало.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы был осуществлен обзор предметной области, в процессе которого была рассмотрена актуальность создания системы распознавания жестов. В процессе проектирования приложения была проведена разработка требований к системе и сформулированы параметры ее тестирования. При разработке системы была обучена нейронная сеть позволяющая определять жесты в игре «камень, ножницы, бумага».

В результате выполнения работы разработана система распознавания жестов и игра «камень, ножницы, бумага» на её основе.

Полученные результаты обладают научной ценностью. Ценность заключается в перспективе дальнейших исследований в области применения систем распознавания жестов в медицине, в искусстве и на производстве.

Дальнейшее развитие разработанной системы может включать в себя разработку подсистемы добавления пользовательских жестов и работы в направлении облегчения интеграции этой системы с другими приложениями.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Авербух В.Л., Авербух Н.В., Стародубцев И.С., Тоболин. Д.Ю. Использование жестовых интерфейсов при взаимодействии с объектами // Научная перспектива. № 10 (56) / 2014. С.57-66.
2. Кривонос Ю. Г., Крак Ю. В. Моделирование движений рук, мимики и артикуляции лица человека для синтеза и визуализации жестовой информации //Кибернетика и системный анализ. – 2011.
3. Лутохин Артем Сергеевич, Тычков Александр Юрьевич, Сотников Александр Михайлович, Алимуратов Алан Казанферович АНАЛИЗ СИСТЕМ ЗАХВАТА ДВИЖЕНИЯ В СРЕДЕ ВИРТУАЛЬНОЙ РЕАЛЬНОСТИ // Вестник ПензГУ. 2021. №2 (34).
4. Moeslund T. B., Granum E. A survey of computer vision-based human motion capture //Computer vision and image understanding. – 2001. – Т. 81. – №.3.–С.231-268.
5. Mohammadzadeh F. F. et al. Feasibility of a wearable, sensor-based motion tracking system //Procedia Manufacturing. – 2015. – Т. 3. – С. 192-199
6. Хамидулин Т. Г. СРАВНЕНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ГЛУБОКОГО ОБУЧЕНИЯ // Экономика и социум. 2019. №2 (57).
7. Гурбансахедов М. Ч., Дурбаева Ф. Д., Атаев Э. PYTHON ПРОГРАММИРОВАНИЕ И ООП: УГЛУБЛЕННОЕ ИЗУЧЕНИЕ // Символ науки. 2024. №2-2-2.
8. Журавлева Н.А. Выбор оптимизатора для обучения сверточной нейронной сети: задача классификации лесных пожаров // StudNet. 2022. №5. С. 4722-4737.
9. Бадика Е. М., Марченков З.В. Проблема переобучения нейронных сетей. Способы устранения переобучения // Инновационные идеи молодых исследователей. 2021. С. 236-243.

10. MediaPipe Docs [Электронный ресурс] // URL: <https://mediapipe.readthedocs.io/en/latest/solutions/hands.html> (Дата обращения: 03.04.2025)

ПРИЛОЖЕНИЕ А

Игровое приложение

Листинг программы

ПРИЛОЖЕНИЕ А.1

Файл «GUI.py»

```
import tkinter as tk
from tkinter import ttk
from PIL import Image, ImageTk
import threading
import time
import game
import cv2
import mediapipe as mp
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles
mp_hands = mp.solutions.hands

try:
    from ctypes import windll
except ImportError:
    pass
else:
    windll.shcore.SetProcessDpiAwareness(1)

FONT_SIZE = ("TkTextFont", 16)

class PlayerFrame(ttk.LabelFrame):

    def __init__(self, player, cap, master=None, **kw):

        self.s = ttk.Style()
        self.cap = cap
        self.s.configure('Red.TLabelframe.Label',
font=FONT_SIZE)
        super().__init__(master, text = player.get_name(),
style = "Red.TLabelframe", **kw)
        self.player = player
```

```

        self.delay =
int(1000/self.cap.get(cv2.CAP_PROP_FPS))

        self.f_top = tk.Frame(self)
        self.canvas = tk.Canvas(self.f_top, width=640,
height=360, bg="black")
        self.sign_label = ttk.Label(self, text="", font =
FONT_SIZE)

        self.f_top.pack(side=tk.TOP, expand=True,
fill="both")
        self.canvas.pack(anchor=tk.CENTER, expand=True, padx
= "0.5m", pady = "0.5m")
        self.sign_label.pack(side=tk.TOP)
        self.update()

def update(self):
    ret, image = self.cap.read()
    if ret:

        with mp_hands.Hands(
            static_image_mode=False,
            max_num_hands=1,
            min_detection_confidence=0.5) as hands:
            image = cv2.cvtColor(image,
cv2.COLOR_BGR2RGB)

            results = hands.process(image)
            image.flags.writeable = True
            if results.multi_hand_landmarks:
                for hand_landmarks in
results.multi_hand_landmarks:
                    mp_drawing.draw_landmarks(
                        image,
                        hand_landmarks,

```

```

mp_hands.HAND_CONNECTIONS,

mp_drawing_styles.get_default_hand_landmarks_style(),

mp_drawing_styles.get_default_hand_connections_style())
        self.photo =
ImageTk.PhotoImage(image=Image.fromarray(image))
        self.canvas.create_image(0, 0, image=self.photo,
anchor=tk.NW)

        self.master.after(self.delay, self.update)
    else:
        self.cap.release()

    def show_sign(self):

        sign = self.player.get_sign()
        self.sign_label['text'] = str(sign)
        return

    def reset(self):

        self.sign_label['text'] = ""
        self.canvas.delete("all")

class BotFrame(ttk.LabelFrame):

    def __init__(self, player, master=None, **kw):

        self.s = ttk.Style()
        self.s.configure('Red.TLabelframe.Label',
font=FONT_SIZE)
        super().__init__(master, text = player.get_name(),
style = "Red.TLabelframe", **kw)
        self.player = player

```

```

        self.pic_rock = tk.PhotoImage(file='rock.png')
        self.pic_paper = tk.PhotoImage(file='paper.png')
        self.pic_scissors =
tk.PhotoImage(file='scissors.png')

        self.f_top = tk.Frame(self)
        self.canvas = tk.Canvas(self.f_top, width=640,
height=360, bg="black")
        self.sign_label = ttk.Label(self, text="", font =
FONT_SIZE)

        self.f_top.pack(side=tk.TOP, expand=True,
fill="both")
        self.canvas.pack(anchor=tk.CENTER, expand=True, padx
= "0.5m", pady = "0.5m")
        self.sign_label.pack(side=tk.TOP)

def show_sign(self):

    sign = self.player.get_sign()
    self.sign_label['text'] = str(sign)

    match sign.get_code():
        case game.ROCK:
            self.canvas.create_image(0, 0, anchor=tk.NW,
image = self.pic_rock)
        case game.SCISSORS:
            self.canvas.create_image(0, 0, anchor=tk.NW,
image = self.pic_scissors)
        case game.PAPER:
            self.canvas.create_image(0, 0, anchor=tk.NW,
image = self.pic_paper)
    return

```

```

def reset(self):

    self.sign_label['text'] = ""
    self.canvas.delete("all")


class Window(tk.Tk):

    def __init__(self, rps_game, cap, screenName=None,
baseName=None, className='Tk',
                useTk=True, sync=False, use=None):

        super().__init__(screenName, baseName, className,
useTk, sync, use)
        self.s = ttk.Style()
        self.s.configure('my.TButton', font=FONT_SIZE)
        self.game = rps_game
        self.f_top = ttk.Frame(self)
        self.player_screen = PlayerFrame(self.game.pl1, cap,
self.f_top)
        self.bot_screen = BotFrame(self.game.pl2,
self.f_top)
        self.f_bot = tk.Frame(self)
        self.winner_label = ttk.Label(self.f_bot, text="",
font = FONT_SIZE)
        self.start_btn = ttk.Button(self.f_bot,
text="Crapt", style='my.TButton', command = self.start_game)

        self.f_top.pack(side=tk.TOP, expand=True,
fill="both")
        self.f_bot.pack(side=tk.BOTTOM, fill="x")
        self.player_screen.pack(side=tk.LEFT, expand=True,
fill="both", padx = "0.5m", pady = "0.5m")

```

```

        self.bot_screen.pack(side=tk.RIGHT, expand=True,
fill="both", padx = "0.5m", pady = "0.5m")
        self.winner_label.pack(side=tk.TOP)
        self.start_btn.pack(side=tk.BOTTOM, padx = "0.5m",
pady = "0.5m")

        self.is_counting_down = False

def reset(self):

    self.game.reset()
    self.bot_screen.reset()
    self.player_screen.reset()
    self.winner_label['text'] = ""

def countdown(self):

    for i in range(3, 0, -1):
        self.winner_label.config(text=str(i))
        time.sleep(1)

    self.game.play()
    self.bot_screen.show_sign()
    self.player_screen.show_sign()
    try:
        self.winner_label["text"] = (f'Победил
{self.game.get_winner().get_name()}')
    except AttributeError:
        self.winner_label["text"] = "Ничья"
    self.start_btn.config(state=tk.NORMAL)
    self.is_counting_down = False

```

```

def start_game(self):

    self.reset()
    # Начало обратного отсчета
    if not self.is_counting_down:
        self.is_counting_down = True
        self.start_btn.config(state=tk.DISABLED)
        self.countdown_thread =
threading.Thread(target=self.countdown)
        self.countdown_thread.start()


cap = cv2.VideoCapture(0)
rps_game = game.Game(game.Player("Игрок"),
game.Bot_Player("Компьютер"), cap)
root = Window(rps_game, cap)
root.title("Камень-ножницы-бумага")
root.mainloop()

```

ПРИЛОЖЕНИЕ А.2

Файл «detector.py»

```
import cv2                                     # библиотека opencv
(получение и обработка изображения)

import mediapipe as mp                         # библиотека
mediapipe (распознавание рук)

import torchvision
import torchvision.transforms as transforms
import numpy as np
import torch
import torch.nn as nn

import game

NUM_CLASSES = 4
PATH = 'HandNet.ckpt'

device = torch.device('cuda:0' if torch.cuda.is_available()
else 'cpu')

class CnNet(nn.Module):

    def __init__(self, num_classes=10):
        nn.Module.__init__(self)
        self.hands =
mp.solutions.hands.Hands(static_image_mode=True,
                           max_num_hands=1,
                           model_complexity=1)

        self.fc = nn.Linear(21*3, num_classes)

    def hand_to_skeleton(self, image):
```

```

        open_cv_image = cv2.cvtColor(np.array(image),
cv2.COLOR_RGB2BGR)
        points = np.array(np.zeros((21, 3)),
dtype=np.float32)
        results =
self.hands.process(cv2.cvtColor(open_cv_image,
cv2.COLOR_BGR2RGB))
        if results.multi_hand_world_landmarks:
            for hand_world_landmarks in
results.multi_hand_world_landmarks:
                points =
self.normalize_landmarks(hand_world_landmarks)

        return points

def rotation_matrix_from_vectors(self, vec1, vec2):
    a = vec1 / np.linalg.norm(vec1)
    b = vec2 / np.linalg.norm(vec2)
    v = np.cross(a, b)
    c = np.dot(a, b)
    s = np.linalg.norm(v)
    if s == 0:
        return np.eye(3)
    kmat = np.array([[0, -v[2], v[1]],
                    [v[2], 0, -v[0]],
                    [-v[1], v[0], 0]])
    rotation = np.eye(3) + kmat + (kmat @ kmat) * ((1 -
c) / (s ** 2))
    return rotation

def normalize_landmarks(self, landmarks):

```

```

        inner_landmarks = []

        for landmark in landmarks.landmark:
            inner_landmarks.append([landmark.x, landmark.y,
landmark.z])

        points = np.array(inner_landmarks, dtype=np.float32)
        v = points[9] - points[0]
        v /= np.linalg.norm(v)
        up = np.array([0, -1, 0])
        rot_mat = self.rotation_matrix_from_vectors(v, up)
        centered_points = points - points[0]
        rotated_points = centered_points @ rot_mat.T

        return rotated_points

    def forward(self, x):
        batch_size = x.size(0)
        outputs = []

        for i in range(batch_size):
            img = x[i]
            img = img.cpu().numpy()
            img = img.transpose(1, 2, 0)
            img = transforms.ToPILImage()(img)
            skeleton = self.hand_to_skeleton(img)
            skeleton = torch.tensor(skeleton,
dtype=torch.float32).to(device)
            outputs.append(skeleton)

        outputs = torch.stack(outputs, dim=0)
        outputs = outputs.view(outputs.size(0), -1)
        outputs = self.fc(outputs)
        return outputs

class Detector:

```

```

def __init__(self):

    self.model = CnNet(NUM_CLASSES).to(device)
    self.model.load_state_dict(torch.load(PATH,
weights_only=True, map_location=device))
    self.model.eval()

    def get_sign(self, image: np.ndarray) -> int:
        sign_list = [game.NO_SIGN, game.PAPER, game.ROCK,
game.SCISSORS]
        image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        image_tensor =
torch.from_numpy(image_rgb.astype(np.float32) / 255.0)
        image_tensor = image_tensor.permute(2, 0, 1)
        image_batch = image_tensor.unsqueeze(0).to(device)
        with torch.no_grad():
            outputs = self.model(image_batch)
            _, predicted_class = torch.max(outputs.data, 1)
        return game.Sign(sign_list[predicted_class.item()])

```

ПРИЛОЖЕНИЕ А.3

Файл «game.py»

```
from random import choice
import detector
import cv2

ROCK = 0
PAPER = 1
SCISSORS = 2
NO_SIGN = 3

class Sign:

    def __init__(self, sign_code):

        self.sign_names = ["камень", "бумага", "ножницы",
"нет жеста"]
        self.sign = sign_code

    def __str__(self):

        return self.sign_names[self.sign]

    def get_code(self):

        return self.sign

class Player:

    def __init__(self, name):
```

```

        self.name = str(name)
        self.sign = Sign(NO_SIGN)

    def reset(self):

        self.sign = Sign(NO_SIGN)

    def __str__(self):

        return self.name + ": " + str(self.sign)

    def get_name(self):

        return self.name

    def get_sign(self):

        return self.sign

    def choose_sign(self, image):

        self.sign = detector.Detector().get_sign(image)

        return self.sign

class Bot_Player(Player):

    signs = [ROCK, PAPER, SCISSORS]

```

```

def choose_sign(self):

    self.sign = Sign(choice(self.signs))
    return self.sign


class Game:

    def __init__(self, player1, player2, cap):

        self.pl1 = player1
        self.pl2 = player2
        self.cap = cap

        #rock      paper      scissors
no_sign
        self.rules = [[None,      self.pl2,  self.pl1,
self.pl1], #rock
                     [self.pl1,  None,      self.pl2,
self.pl1], #paper
                     [self.pl2,  self.pl1,  None,
self.pl1], #scissors
                     [self.pl2,  self.pl2,  self.pl2,
None]]      #no_sign

    def reset(self):

        self.pl1.reset()
        self.pl2.reset()

    def play(self):

        success = False
        while not success:

```

```
        success, image = self.cap.read()
    image = cv2.flip(image, 1)

    self.pl1.choose_sign(image)
    self.pl2.choose_sign()

def get_winner(self):

    sign1 = self.pl1.get_sign().get_code()
    sign2 = self.pl2.get_sign().get_code()

    winner = self.rules[sign1][sign2]

    return winner
```