# 1 Введение и общая архитектура

## *1.1 Постановка задачи*

***Актуальность*** – повышение безопасности дорожного движения является одной из приоритетных задач в транспортной сфере. Разработка интеллектуальных систем, способных в реальном времени анализировать дорожную обстановку, открывает возможности для создания продвинутых систем помощи водителю (ADAS), автоматизации мониторинга состояния дорожной инфраструктуры и повышения общей осведомленности участников движения.

***Цель проекта*** – разработка и реализация программного комплекса (веб-приложения), способного в режиме реального времени принимать видеопоток с камеры, детектировать на нем дорожные знаки и разметку типа "зебра" и визуализировать результаты анализа для пользователя

***Задачи проекта***

1. Подобрать и подготовить наборы данных (RTSD, CeyMo) для обучения моделей распознавания.
2. Обучить две модели на базе архитектуры YOLO для решения задач:
   - детекции и классификации 155 типов дорожных знаков;
   - семантической сегментации разметки "пешеходный переход".
3. Реализовать асинхронный сервер на Python с использованием фреймворка FastAPI, способный управлять WebSocket-соединениями и обрабатывать ML-задачи в параллельных потоках.
4. Создать многостраничный веб-интерфейс на чистом JavaScript, который захватывает видео с камеры, отправляет его на сервер и отрисовывает полученные результаты на элементе <canvas>.
5. Объединить все компоненты в единую систему и провести тестирование функциональности и производительности.

## *1.2 Обзор архитектуры системы*

Проект «Road Vision App» построен на основе классической клиент-серверной архитектуры, адаптированной для задач обработки данных в реальном времени. Система состоит из трех логических компонентов:

1. Веб-приложение, работающее в браузере пользователя. Его основная задача – захватить видеопоток с камеры устройства, отправить его на сервер для анализа и отобразить полученные результаты поверх исходного видео.
2. Ядро системы, реализованное на FastAPI. Сервер управляет логикой приложения: принимает WebSocket-соединения от клиентов, получает от них видеокадры, передает их на обработку ML-модулям и отправляет результаты обратно клиенту.

3. Две независимые, предварительно обученные нейронные сети YOLO, которые выполняют основную вычислительную работу: поиск знаков и разметки на изображении.

Процесс обработки данных в системе может идти по двум сценариям:

Таблица 1. Сценарии обработки данных

| **Обработка real-time видеопотока** (основной сценарий) | **Обработка одиночного изображения** (дополнительный сценарий) |
|---|---|
| 1. Пользователь нажимает кнопку "Start Streaming".<br>2. Клиент (JavaScript) устанавливает постоянное WebSocket-соединение с сервером.<br>3. С определенной частотой клиент захватывает кадр, кодирует его в Base64 (JPEG) и отправляет по WebSocket.<br>4. Сервер (FastAPI) запускает параллельную обработку кадра двумя моделями.<br>5. После завершения сервер объединяет результаты в JSON-объект и отправляет его обратно клиенту.<br>6. Клиент отрисовывает результаты на элементе <canvas>. | 1. Пользователь выбирает файл изображения и нажимает "Отправить".<br>2. Клиент отправляет изображение на сервер с помощью стандартного POST-запроса на REST эндпоинт /api/image.<br>3. Сервер использует ту же логику параллельной обработки, что и для WebSocket, и возвращает JSON-объект с результатами в ответе на POST-запрос.<br>4. Клиент отображает полученные результаты. |

Такой двойной подход делает приложение более гибким, позволяя анализировать как живое видео, так и статичные изображения.

Этот цикл повторяется до тех пор, пока пользователь не остановит трансляцию. Выбор WebSocket в качестве транспортного протокола является ключевым, так как он обеспечивает постоянное двунаправленное соединение с низкой задержкой, что идеально подходит для задач реального времени.

### *1.3 Технологический стек*

Для реализации проекта был выбран следующий набор технологий, обеспечивающий высокую производительность, масштабируемость и простоту разработки.

Таблица 2. Технологический стек

| Компонент | Технология | Обоснование выбора |
|---|---|---|
| **Машинное обучение** | Python, PyTorch, Ultralytics (YOLO) | YOLO – state-of-the-art архитектура для задач детекции в реальном времени. Фреймворк Ultralytics предоставляет удобный API для обучения и инференса. |
| **Серверная часть** | Python, FastAPI, Uvicorn | FastAPI – современный, высокопроизводительный веб-фреймворк с нативной поддержкой асинхронности и WebSockets, что критично для данного проекта. |
| **Клиентская часть** | HTML5, CSS3, JavaScript (ES6+) | Универсальный стандарт для веб-клиентов. Использование нативного JS (без фреймворков) дает полный контроль над API браузера (getUserMedia, Canvas). |
| **Обмен данными** | WebSocket | Протокол, обеспечивающий постоянное соединение с низкой задержкой, что является стандартом для real-time приложений. |
| **Управление версиями** | Git, Git LFS | Git – стандарт для контроля версий кода. Git LFS (Large File Storage) необходим для хранения больших файлов моделей нейронных сетей в репозитории. |

### 1.4 Репозиторий

Репозиторий доступен для чтения и создания форков (ответвлений) по ссылке: https://github.com/sackvoich/road-vision-app.

# 2 Подготовка и запуск проекта

## *2.1 Требования к окружению*

Для корректной работы проекта необходимо наличие следующего программного обеспечения:

1. Python (версия 3.11 или выше): Проект использует современные возможности языка Python и его асинхронной библиотеки asyncio, поэтому требуется версия не ниже 3.11.

2. Git: Система контроля версий, необходимая для получения исходного кода проекта из репозитория. Официальный сайт для скачивания: https://git-scm.com/downloads.

3. Git LFS (Large File Storage): Обязательное расширение для Git. Файлы обученных моделей (.pt) имеют большой размер и хранятся в репозитории с помощью Git LFS. Без этого расширения вместо реальных файлов моделей будут загружены лишь текстовые указатели, что приведет к ошибке при запуске приложения.

Перед началом установки необходимо убедиться, что Git LFS инициализирован в системе. Это делается один раз с помощью команды

```
git lfs install
```

## *2.2 Пошаговая инструкция по развёртыванию*

Процесс установки и запуска проекта состоит из шести последовательных шагов

**1. Клонирование репозитория**

Первым делом необходимо скачать исходный код проекта. Откройте терминал или командную строку, перейдите в директорию, где будет храниться проект, и выполните команду

```
git clone https://github.com/sackvoich/road-vision-app.git
cd road-vision-app
```

Эта команда создаст папку road-vision-app и загрузит в нее все файлы проекта, за исключением файлов, отслеживаемых LFS.

**2. Загрузка файлов моделей**

После клонирования репозитория необходимо явно загрузить файлы моделей из хранилища LFS. Выполните следующую команду

```
git lfs pull
```

Эта команда проверит все LFS-указатели в проекте и скачает соответствующие им реальные файлы. После успешного выполнения в папке models/ должны появиться два файла: traffic_signs_detection_model.pt и zebra_segmentation_model.pt.

### 3. Создание виртуального окружения

Для изоляции зависимостей проекта от глобальной системы рекомендуется использовать виртуальное окружение.

- **Создание окружения** (выполняется в корневой папке проекта):

```
python -m venv venv
```

Эта команда создаст папку venv, содержащую копию интерпретатора Python и менеджер пакетов pip.

- **Активация окружения**
  - Для Windows (в Command Prompt или PowerShell):

```
.\venv\Scripts\activate
```

  - Для macOS/Linux:

```
source venv/bin/activate
```

После активации в начале строки терминала появится префикс (venv), указывающий на то, что вы работаете внутри виртуального окружения.

### 4. Установка зависимостей

Все необходимые для работы проекта библиотеки перечислены в файле pyproject.toml. Для их быстрой установки используется менеджер пакетов uv.

- Если uv еще не установлен, его можно установить командой:

```
pip install uv.
```

- Для установки всех зависимостей проекта выполните:

```
uv pip sync
```

Эта команда прочитает файл uv.lock, который содержит точные версии всех библиотек, и установит их в ваше виртуальное окружение.

### 5. Запуск сервера

Когда все зависимости установлены, можно запустить веб-сервер FastAPI. Для этого используется uvicorn. Выполните команду:

```
uvicorn main:app --host 0.0.0.0 --port 8000 –reload
```

- main:app: указывает uvicorn найти в файле main.py объект app, который является экземпляром FastAPI.

- --host 0.0.0.0: делает сервер доступным не только с локальной машины, но и с других устройств в той же сети.
- --port 8000: запускает сервер на порту 8000.
- --reload: включает режим автоматической перезагрузки. Сервер будет перезапускаться при каждом изменении в исходном коде, что удобно для разработки.

**6. Доступ к приложению**

После успешного запуска сервера откройте веб-браузер и перейдите по адресу: http://127.0.0.1:8000. В браузере должен открыться главный интерфейс приложения, готовый к работе.

# 3 Модуль машинного обучения

Модуль машинного обучения является ядром всей системы «Road Vision App». Он отвечает за анализ изображений и извлечение из них полезной информации. Модуль состоит из двух независимых нейронных сетей, каждая из которых обучена для решения своей специфической задачи: одна – для обнаружения дорожных знаков, вторая – для сегментации пешеходных переходов.

## *3.1 Модель детекции дорожных знаков*

Эта модель решает задачу обнаружения объектов (Object Detection). Ее цель – найти на изображении все присутствующие дорожные знаки, определить их границы с помощью прямоугольной рамки (bounding box) и классифицировать каждый найденный знак.

Модель основана на архитектуре YOLOv11. Семейство моделей YOLO (You Only Look Once) является отраслевым стандартом для задач, требующих высокой скорости обработки в реальном времени, так как оно обрабатывает все изображение за один проход, обеспечивая оптимальный баланс между скоростью и точностью.

В качестве обучающих данных использовался публичный датасет RTSD (Russian Traffic Sign Dataset). Этот набор данных содержит десятки тысяч изображений, снятых с видеорегистраторов в различных погодных условиях и в разное время суток. Датасет включает разметку для 155 различных классов дорожных знаков, что обеспечивает широкое покрытие знаков, используемых в том числе и на территории Беларуси.

Процесс обучения:

1. Исходный датасет в формате COCO был конвертирован в формат YOLO. Была применена фильтрация для удаления аннотаций со слишком маленькой площадью (<900 пикселей), которые могли бы внести шум в обучение.

2. Для повышения устойчивости (робастности) модели к реальным условиям применялся широкий спектр аугментаций в реальном времени, включая случайные повороты, сдвиги, изменения масштаба, а также модификации цветового пространства (оттенок, насыщенность, яркость). Использовались продвинутые техники, такие как Mosaic (объединение 4 изображений в одно) и MixUp.

3. Модель обучалась на протяжении 100 эпох с размером изображения 640x640 пикселей.

## *3.2 Модель сегментации дорожных знаков*

Эта модель решает задачу семантической сегментации (Semantic Segmentation). В отличие от детекции, ее цель – не просто найти объект, а классифицировать каждый пиксель изображения, принадлежащий этому

объекту. В данном случае модель находит точный контур ("маску") пешеходного перехода.

Модель основана на архитектуре YOLOv11-Seg, являющейся модификацией YOLO, специально адаптированной для задач сегментации. Она не только предсказывает ограничивающую рамку, но и генерирует полигональную маску для каждого обнаруженного объекта.

Модель обучалась на датасете CeyMo. Этот набор данных специализируется на дорожной разметке и содержит изображения с высокоточной пиксельной разметкой для различных ее типов, включая пешеходные переходы. Использование такого специализированного датасета критически важно для получения точных контуров.

Процесс обучения был аналогичен модели детекции, однако целевой метрикой являлось не только качество ограничивающей рамки (mAP), но и точность маски сегментации (Intersection over Union, IoU).

### *3.3 Назначение и использование скрипта конвертации*

Скрипт convert.py не участвует в основном цикле работы веб-приложения, а выполняет вспомогательную, но важную функцию – экспорт моделей.

Основной целью скрипта является конвертация моделей из внутреннего формата PyTorch (.pt) в формат Core ML (.mlmodel). Core ML – это фреймворк от Apple, предназначенный для высокопроизводительного запуска моделей машинного обучения непосредственно на устройствах iOS, iPadOS и macOS.

Скрипт загружает обе обученные модели (.pt) и с помощью встроенной в библиотеку ultralytics функции .export(format="coreml") производит их конвертацию.

Наличие этого скрипта демонстрирует, что архитектура проекта предусматривает возможность дальнейшего развития. Например, на основе этих же обученных моделей можно создать нативное мобильное приложение для iPhone. Эта возможность напрямую отражена в коде серверной части (main.py), где предусмотрена логика для загрузки моделей в скомпилированном формате .mlpackage при запуске на платформе macOS.

# 4 Серверная часть

Серверная часть является центральным компонентом системы, отвечающим за всю логику обработки данных. В данном разделе подробно анализируется исходный код файла main.py, объясняются ключевые архитектурные решения и принципы работы приложения.

## *4.1 Обзор FasiAPI и асинхронный подход*

В качестве основы для бэкенда был выбран веб-фреймворк FastAPI. Этот выбор обусловлен двумя его ключевыми преимуществами:

1. FastAPI является одним из самых быстрых Python-фреймворков, что критически важно для приложений, обрабатывающих данные в реальном времени.

2. FastAPI построен на базе стандарта ASGI (Asynchronous Server Gateway Interface), что позволяет ему эффективно обрабатывать большое количество одновременных подключений (например, множество клиентов, отправляющих видеопотоки) без блокировок. Асинхронный подход означает, что сервер может переключаться между задачами, не дожидаясь завершения каждой из них, что идеально подходит для работы с долгоживущими соединениями, такими как WebSockets.

При запуске сервер инициализирует приложение FastAPI, настраивает CORS (Cross-Origin Resource Sharing) для разрешения запросов из браузера и загружает в память обе модели машинного обучения, подготавливая их к работе.

## *4.2 Обработка WebSocket-соединений*

Взаимодействие между клиентом и сервером в реальном времени осуществляется через WebSocket. За эту логику отвечает эндпоинт, определенный декоратором @app.websocket("/ws/video").

Для отслеживания активных клиентов используется простой класс ConnectionManager. Он хранит список всех подключенных WebSocket-объектов, что позволяет чисто управлять их подключением и отключением.

Внутри эндпоинта запускается бесконечный цикл (while True), который ожидает поступления данных (видеокадров в формате Base64) от клиента.

Для защиты от перегрузки введена серверная система ограничения частоты кадров. Глобальная переменная last_processed_time хранит время обработки последнего кадра. Если новый кадр поступает раньше, чем истечет минимальный интервал (MIN_INTERVAL), он игнорируется. Это гарантирует, что сервер не будет пытаться обрабатывать кадры чаще, чем он физически способен, даже если клиент отправляет их со слишком высокой частотой.

### *4.3 Параллельная обработка кадров двумя моделями*

Функции инференса моделей машинного обучения (model(frame)) являются синхронными (блокирующими) и ресурсоемкими операциями. Прямой вызов такой функции внутри асинхронного цикла FastAPI "заморозил" бы весь сервер на время ее выполнения, делая невозможной обработку запросов от других клиентов.

Для решения этой проблемы используется механизм asyncio.to_thread. Эта функция позволяет запустить блокирующую (синхронную) функцию в отдельном потоке, не блокируя основной асинхронный цикл событий:

1. Сервер получает кадр и декодирует его из Base64

2. Создаются две "задачи" – по одной для каждой модели – с помощью asyncio.to_thread:

```
detection_task = asyncio.to_thread(run_detection, frame)
segmentation_task = asyncio.to_thread(run_segmentation, frame)
```

3. Далее используется asyncio.gather, который позволяет асинхронно ожидать завершения обеих задач. Это означает, что обе модели будут обрабатывать один и тот же кадр параллельно в разных потоках.

4. Как только обе задачи завершаются, gather возвращает их результаты, которые объединяются в один JSON-объект и отправляются клиенту.

Такой подход позволяет максимально эффективно использовать ресурсы процессора (или GPU), значительно сокращая общее время обработки одного кадра.

### *4.4 Форматирование результатов моделей*

Функции run_detection и run_segmentation служат "обертками" (wrappers) для моделей YOLO. Их основная задача – преобразовать "сырой" вывод нейронной сети в структурированный и понятный для клиента JSON-формат.

Логика работы:

1. Функция принимает на вход кадр изображения в виде NumPy-массива.

2. Выполняется вызов model(frame, verbose=False, conf=0.1), который запускает инференс.

3. Результат работы модели итерируется, и из него извлекаются ключевые данные:

3.1 Для детекции: координаты ограничивающей рамки (box.xyxy), уверенность (box.conf) и ID класса (box.cls). ID класса преобразуется в читаемое имя знака (например, 2_1) с помощью словаря model.names.

3.2 Для сегментации: контурные точки полигона (masks.xy), а также уверенность и класс объекта.

4. Эти данные упаковываются в список словарей, формируя чистый JSON-ответ.

Обе функции обернуты в декоратор @ThreadingLocked() из библиотеки ultralytics. Этот декоратор обеспечивает потокобезопасность, гарантируя, что если несколько потоков попытаются одновременно использовать один и тот же экземпляр модели, они будут делать это последовательно, избегая конфликтов и ошибок.

### 4.5 Обработка одиночных изображений (REST API)

Помимо обработки видеопотока, сервер предоставляет стандартный REST API эндпоинт, определенный декоратором @app.post("/api/image"). Он предназначен для анализа одиночных изображений, загружаемых пользователем.

Эндпоинт принимает файл (UploadFile), проверяет, что это изображение, и считывает его содержимое.

Вместо дублирования кода, полученные байты изображения кодируются в формат Base64 и передаются в уже существующую асинхронную функцию process_frame_parallel. Это элегантное решение позволяет использовать одну и ту же высокопроизводительную логику параллельной обработки как для WebSockets, так и для REST.

Результат обработки возвращается клиенту в виде стандартного JSON-ответа на POST-запрос.

### 4.6 Маршрутизация и отдача статических файлов

Для обслуживания многостраничного фронтенда в main.py реализована маршрутизация с помощью FastAPI.

Эндпоинты, определенные декораторами @app.get("/"), @app.get("/info") и @app.get("/people"), отвечают за отдачу соответствующих HTML-файлов (index.html, info.html, people.html). Они используют FileResponse для прямой отправки файла из папки static.

Команда app.mount("/static", ...) "монтирует" всю папку static (включая подпапки css и js), делая все находящиеся в ней файлы доступными для браузера по пути /static/.... Это позволяет HTML-страницам корректно загружать свои стили и скрипты.

# 5 Клиентская часть

Клиентская часть проекта «Road Vision App» – это интерактивный веб-интерфейс, который служит точкой входа для пользователя. Он отвечает за захват видео, взаимодействие с сервером и наглядную визуализацию результатов анализа. Фронтенд реализован без использования сторонних фреймворков (на "чистых" HTML, CSS и JavaScript), что обеспечивает полный контроль над API браузера и высокую скорость загрузки.

## 5.1 Структура и стилизация интерфейса (HTML/CSS)

Интерфейс проекта является многостраничным, но логически простым.

Таблица 3. Структура файлов клиентской части

| Название файла | Описание |
|---|---|
| index.html | Главная страница, на которой происходит вся интерактивная работа с видео |
| info.html, people.html | Статические информационные страницы, описывающие проект и команду |
| css/styles.css | Единый файл стилей, обеспечивающий консистентный внешний вид всех страниц |

Ключевые элементы интерфейса (index.html):

1. Блок медиа (.media-wrap): Содержит два наложенных друг на друга элемента:

- <video id="camera">: Отображает "живое" видео с камеры пользователя.
- <canvas id="frameCanvas">: Прозрачный холст, расположенный поверх видео, на котором отрисовываются результаты обработки (рамки и маски).

2. Элементы управления (.controls, .upload): Набор кнопок для управления приложением: "Начать/Остановить стрим", "Сделать кадр" для отправки одиночного изображения с видео, а также поле для загрузки файла с диска.

3. Панель результатов (.results): Текстовое поле, в котором для отладки и наглядности отображается полученный от сервера JSON-ответ.

Стилизация выполнена с использованием современных практик CSS. Применяется адаптивная верстка (через @media-запросы), что делает интерфейс удобным для использования как на десктопных, так и на мобильных устройствах. Использование CSS Custom Properties (переменных) позволяет легко управлять цветовой схемой и основными параметрами дизайна.

### 5.2 Клиентская логика (app.js)

Вся интерактивность главной страницы реализована в файле app.js внутри класса VideoProcessor. Этот класс инкапсулирует всю логику работы с камерой, WebSocket-соединением и отрисовкой.

При загрузке страницы метод initializeCamera через navigator.mediaDevices.getUserMedia запрашивает у пользователя доступ к камере. Приоритет отдается задней камере (facingMode: 'environment'), что более удобно для сканирования дорожной обстановки на мобильных устройствах.

Методы startStreaming и stopStreaming управляют жизненным циклом WebSocket-соединения. При нажатии на кнопку "Start" создается новый экземпляр WebSocket, который подключается к серверному эндпоинту /ws/video. Настраиваются обработчики событий: onopen (запускает отправку кадров), onmessage (обрабатывает входящие данные от сервера), onerror и onclose.

Два независимых цикла: отправка и отрисовка:

Это ключевое архитектурное решение на клиенте, обеспечивающее плавность интерфейса.

1. Цикл отправки (sendFrames): Запускается после установки WebSocket-соединения. С помощью setTimeout он с фиксированной частотой (5 FPS) берет текущий кадр из тега <video>, рисует его на временном холсте, конвертирует в Base64 (JPEG) и отправляет на сервер. Ограничение FPS на клиенте снижает нагрузку на сеть и сервер.

2. Цикл отрисовки (renderLoop): Работает независимо и постоянно. С помощью requestAnimationFrame он с максимальной возможной для браузера частотой (обычно 60 FPS) копирует текущий кадр из <video> на видимый пользователю <canvas>. Если от сервера получены результаты, они отрисовываются поверх этого кадра. Такой подход гарантирует, что видео в браузере никогда не "тормозит", даже если сервер отвечает с задержкой.

Помимо стриминга, реализована логика для отправки одиночных изображений через REST API. Обработчики событий для кнопки "Сделать кадр" и поля загрузки файла формируют FormData, отправляют его POST-запросом на эндпоинт /api/image, получают JSON-ответ и вызывают те же функции отрисовки.

Методы drawDetections и drawSegmentations отвечают за визуализацию. Они получают на вход JSON-данные от сервера и с помощью Canvas API рисуют на холсте:

1. Зеленые прямоугольники (strokeRect) для дорожных знаков с подписями их классов и уверенности.

2. Полупрозрачные розовые полигоны (fill и stroke) для масок пешеходных переходов.

# 6 Схема разработки аналогичного проекта

Предыдущие разделы подробно описывали компоненты готового проекта «Road Vision App». Этот раздел представляет собой обобщенную пошаговую инструкцию, или "дорожную карту", для студентов, которые хотят разработать собственный проект компьютерного зрения на основе предложенного стека технологий.

## 6.1 Формулирование задачи и поиск данных

Прежде чем написать строчку кода, нужно четко понять, *что* мы делаем и *на чем* будем обучать модель:

1. *Определить задачу* – что вы хотите получить на выходе?

- Детекция объектов: найти объекты и нарисовать вокруг них рамки (каски, автомобили, игральные карты);
- Сегментация: найти объекты и выделить их точные контуры (опухоли, здания, дефекты на стали);
- Классификация: определить, к какому классу относится все изображение целиком (болезни растений по листу, поза йоги).

2. *Найти датасет* – без данных нет машинного обучения. Искать нужно на платформах вроде Kaggle и Roboflow. Введите в поиск ключевые слова вашей задачи (например, "face mask detection dataset").

3. *Проанализировать датасет* – скачайте и изучите его структуру. Если архитектура датасета не соответствует стандартной для модели (к примеру, для YOLO – это архитектуры YOLO (.txt) и COCO (.json)), необходимо будет привести структуру к стандартной или воспользоваться моделью, которая сможет обрабатывать такой порядок датасета.

4. *Сформировать список классов* – чётко определите, какие классы объектов вы будете распознавать.

## 6.2 Обучение и экспорт модели

На этом этапе мы создаем "мозг" нашего приложения. Удобнее всего это делать в среде вроде Jupyter Notebook или облачной Google Colab (если ресурсов вашего ПК недостаточно).

1. Напишите скрипт для конвертации формата данных (если это необходимо) и разделите датасет на обучающую и валидационную выборки.

2. Для большинства задач детекции и сегментации отлично подойдут предобученные модели YOLO (например, YOLO xS для быстрого старта или YOLO S или YOLO M для большей точности).

3. Используя библиотеку ultralytics, напишите код для запуска обучения, указав путь к вашему датасету, количество эпох и параметры аугментации.

4. Запустите обучение и дождитесь его завершения.

5. После обучения в папке с результатами (runs/detect/train/weights/ или подобной) найдите самый главный файл – best.pt. Это и есть ваша обученная

модель с наилучшими параметрами (если последняя эпоха показала себя хуже).

6. Создайте в корне вашего будущего проекта папку models и скопируйте туда файл best.pt, дав ему осмысленное имя (например, helmet_detection_model.pt).

### 6.3 Создание бэкенда на FastAPI

Теперь нужно обернуть нашу модель в веб-сервис, с которым сможет общаться браузер.

1. Создайте файл main.py, папки models, static. Настройте виртуальное окружение (см. п. 3 подраздела 2 раздела 2 данного пособия).

2. Инициализируйте приложение FastAPI, настройте CORS.

3. Напишите код, который при старте сервера загружает вашу модель из файла best.pt.

4. Напишите синхронную функцию (например, run_helmet_detection), которая принимает на вход кадр (NumPy-массив) и возвращает готовый JSON со списком обнаруженных объектов. Эта функция будет вызывать вашу модель.

5. Реализуйте WebSocket-эндпоинт. Можно скопировать и адаптировать логику из описываемого выше проекта. Эндпоинт должен принимать кадр в Base64, декодировать его и передавать в функцию параллельной обработки.

6. Реализовать REST-эндпоинт. Адаптируйте логику для приема одиночных изображений.

7. Обязательно используйте asyncio.to_thread для вызова вашей "обертки", чтобы не блокировать сервер.

### 6.4 Клиентская часть

1. В папке static создайте index.html. Добавьте на него теги <video>, <canvas>, кнопки управления и область для вывода результатов.

2. Создайте css/styles.css для оформления вашего приложения.

3. Напишите JS-логику:

- реализуйте доступ к камере через getUserMedia;

- напишите логику для установки WebSocket-соединения с вашим бэкендом;

- создайте два независимых цикла: sendFrames (для отправки кадров на сервер через setTimeout) и renderLoop (для плавной отрисовки видео и результатов через requestAnimationFrame);

- напишите функцию отрисовки (например, drawDetections), которая будет принимать JSON от сервера и рисовать рамки на <canvas>;

- добавьте логику для загрузки и отправки одиночного файла через fetch на ваш REST-эндпоинт.

### 6.5 Интеграция, тестирование и отладка

Самый творческий этап, на котором все компоненты собираются вместе.

1. Запустите сервер через uvicorn main:app –reload.
2. Откройте index.html в браузере.
3. Скорее всего, что-то сразу не заработает. Это нормально:

- *Не подключается WebSocket?* Проверьте URL в JS-коде, порт и убедитесь, что сервер запущен. Откройте вкладку "Network" в инструментах разработчика браузера.
- *Сервер падает с ошибкой?* Смотрите в консоль, где запущен uvicorn. Скорее всего, ошибка в путях к модели или в обработке данных.
- *Видео есть, а рамок нет?* Откройте консоль браузера. Проверьте, приходят ли данные от сервера. Если да, то какая у них структура? Возможно, вы неправильно обращаетесь к полям JSON в функции drawDetections. Используйте console.log() для отладки.

Следуя этой "дорожной карте", можно систематически построить практически любое веб-приложение для задач компьютерного зрения, используя представленный в проекте стек.

# 6 Заключение

В ходе работы над проектом была успешно достигнута поставленная цель: разработан и реализован программный комплекс, способный в реальном времени выполнять задачи детекции дорожных знаков и сегментации разметки.

Разработано веб-приложение, демонстрирующее полный цикл обработки данных: от захвата видеопотока на клиенте до параллельной обработки двумя ML-моделями на сервере и визуализации результатов в браузере.

На основе современных архитектур семейства YOLO были успешно обучены две нейронные сети для решения узкоспециализированных задач. Модель детекции способна распознавать 155 классов дорожных знаков, а модель сегментации – точно определять контуры пешеходных переходов.

Выбор асинхронного фреймворка FastAPI и протокола WebSocket позволил создать производительный бэкенд, способный обрабатывать данные с низкой задержкой. Реализация параллельной обработки ML-задач в отдельных потоках является ключевым техническим достижением, которое позволяет максимально эффективно использовать вычислительные ресурсы.

Клиентская часть, написанная на чистом JavaScript, предоставляет пользователю два сценария работы (real-time видео и загрузка изображений) и корректно отображается на устройствах с разным разрешением экрана.

Вся работа была сопровождена созданием данного методического пособия, что обеспечивает воспроизводимость результатов и упрощает дальнейшую доработку проекта.

Таким образом, все задачи, поставленные в начале проекта, были успешно выполнены. Созданный прототип является прочной основой для дальнейших исследований и разработок в области интеллектуальных транспортных систем.

Текущая реализация является успешным прототипом, однако существует множество направлений для его улучшения и расширения функционала:

1. Оптимизация производительности – квантизация моделей, переход на TensorRT, оптимизация на клиенте.

2. Улучшение качества распознавания – дообучение на целевых данных, расширение классов, трекинг объектов.

3. Расширение функциональности – создание нативного мобильного приложения, система оповещений, интеграция с картами.

4. Улучшение пользовательского опыта – более информативная визуализация, история обнаружений.

## 7 Задания для самостоятельной работы

Номер задания соответствует порядковому номеру в журнале.

1. **Разработка системы детекции защитных касок на строительной площадке**. Создать веб-приложение, которое в реальном времени анализирует видеопоток с камеры и обнаруживает рабочих, не использующих защитные каски. Система должна выделять людей в касках и без них, что может быть использовано для автоматического контроля за соблюдением техники безопасности. Датасет: Hard Hat Detection (https://universe.roboflow.com/computer-vision-filqz/hard-hat-detection-62rrp).
2. **Создание системы распознавания товаров на полках магазина**. Разработать прототип системы для ритейла, которая по изображению или видеопотоку может идентифицировать и подсчитывать различные виды товаров на полках (например, бутылки, коробки, фрукты). Это может быть первым шагом к автоматизации инвентаризации или мониторинга цен. Датасет: shelves Dataset (https://universe.roboflow.com/tia-vwgz3/shelves-exlrc).
3. **Разработка системы сегментации опухолей головного мозга на МРТ-снимках**. Создать приложение, которое загружает МРТ-снимок головного мозга и с помощью модели семантической сегментации выделяет на нем область опухоли. Это задание направлено на создание инструмента для помощи в медицинской диагностике. Датасет: Brain MRI segmentation (https://www.kaggle.com/datasets/mateuszbuda/lgg-mri-segmentation).
4. **Классификатор заболеваний растений по фотографиям листьев**. Разработать систему, которая по фотографии листа растения определяет, здорово оно или поражено одним из заболеваний. В отличие от детекции, здесь основной задачей является классификация всего изображения. Датасет: New Plant Diseases Dataset (https://www.kaggle.com/datasets/vipoooool/new-plant-diseases-dataset).
5. **Детектор таблиц в отсканированных документах**. Создать сервис, который принимает на вход изображение отсканированного документа и обнаруживает на нем все табличные области. Такой инструмент является первым шагом в системах автоматического извлечения данных (OCR). Датасет: table detection (https://universe.roboflow.com/shubhayan-sarkar-kwzea/table-detection-q3pef).
6. **Сегментация зданий на аэрофотоснимках**. Разработать приложение, которое загружает спутниковый или аэрофотоснимок и выделяет на нем контуры всех зданий. Это задание применяется в картографии, урбанистике и оценке последствий стихийных бедствий. Датасет: Semantic segmentation of aerial imagery (https://www.kaggle.com/datasets/humansintheloop/semantic-segmentation-of-aerial-imagery).
7. **Распознавание игральных карт в реальном времени**. Создать приложение, которое с помощью видеокамеры в реальном времени распознает игральные карты на столе, определяя их масть и достоинство. Это может быть

основой для создания "умного" помощника для карточных игр. Датасет: Playing Cards (https://universe.roboflow.com/augmented-startups/playing-cards-ow27d).

8. **Детекция признаков пневмонии на рентгеновских снимках грудной клетки**. Разработать систему поддержки принятия врачебных решений. Приложение должно принимать на вход рентгеновский снимок и выделять области, подозрительные на наличие пневмонии, помогая рентгенологам в постановке диагноза. Датасет: RSNA Pneumonia Detection Challenge (https://www.kaggle.com/competitions/rsna-pneumonia-detection-challenge/data, необходимо подтверждение участия в соревновании).

9. **Система обнаружения дефектов на металлической поверхности**. Создать прототип системы для промышленного контроля качества. Приложение должно анализировать изображение поверхности металлического листа и обнаруживать на нем различные типы дефектов (царапины, вмятины и т.д.). Датасет: Severstal: Steel Defect Detection (https://www.kaggle.com/competitions/severstal-steel-defect-detection/data, необходимо подтверждение участия в соревновании).

10. **Детектор пчел для отслеживания их поведения**. Нестандартная исследовательская задача. Создать систему, которая по видео из улья или специальной установки может обнаруживать пчел и отслеживать их. Датасет: bee tracking (https://universe.roboflow.com/beelink/bee-tracking-yalbo).

11. **Система распознавания автомобильных номеров (ANPR)***. Разработать систему, которая в два этапа решает задачу распознавания номеров. Этап 1: Модель детекции находит на изображении прямоугольную область с автомобильным номером. Этап 2: Вырезанная область с номером передается второй модели или библиотеке для оптического распознавания символов (OCR), которая "читает" текст номера. Это классическая задача, требующая построения конвейера из двух моделей. Требуется реализовать двухступенчатый пайплайн на бэкенде: выход одной модели становится входом для другой. Датасет для детекции: License-Plate-Recognition (https://universe.roboflow.com/lpr-ydttq/license-plate-recognition-bgpr1). Для OCR можно использовать библиотеку easyocr в Python.

12. **Распознавание букв языка жестов (дактильная азбука)***. Создать приложение, которое в реальном времени распознает статические жесты руки, соответствующие буквам американской дактильной азбуки ASL. Модель должна классифицировать положение пальцев на руке, определяя загаданную букву. Это задача мелкозернистой классификации (fine-grained classification). Множество классов (25-30 букв) очень похожи друг на друга, что требует от модели умения различать тонкие детали, а от разработчика – тщательного подхода к аугментации и обучению. Датасет: ASL Alphabet (https://www.kaggle.com/datasets/grassknoted/asl-alphabet).

13. **Система подсчета трафика на перекрестке***. Разработать приложение, которое анализирует видеопоток с камеры, направленной на дорогу, и подсчитывает количество транспортных средств, пересекающих виртуальную линию. Система должна уметь отличать разные типы транспорта

(легковой автомобиль, грузовик, автобус). Простая детекция приведет к многократному подсчету одного и того же автомобиля в разных кадрах. Необходимо внедрить алгоритм отслеживания (трекинга) объектов, например, SORT или DeepSORT. Каждому обнаруженному автомобилю должен присваиваться уникальный ID, а подсчет должен происходить только один раз при пересечении линии. Датасет для детекции: Vehicle detection (https://universe.roboflow.com/vaishak-shetty-sri7e/vehicle-detection-byizq).

14. **Анализатор поз йоги или фитнес-упражнений***. Создать "виртуального тренера", который с помощью камеры определяет позу человека и может дать обратную связь о правильности ее выполнения. Модель должна определять расположение ключевых точек тела (суставов). Требуется использовать не стандартную детекцию, а оценку поз (Pose Estimation). Для этого нужно обучить специальную модель (например, YOLO-Pose). Бэкенд должен не просто возвращать координаты точек, но и содержать логику для анализа этих поз (например, вычислять углы в суставах и сравнивать их с эталонными). Датасет: Yoga Pose Image classification dataset (https://www.kaggle.com/datasets/shrutisaxena/yoga-pose-image-classification-dataset).

15. **Система контроля ношения медицинских масок с тремя классами***. Разработать систему, которая не просто определяет наличие или отсутствие маски на лице, но и классифицирует три состояния: маска надета правильно, маска надета неправильно (например, не закрыт нос) и маска отсутствует. Как и в задаче с языком жестов, это мелкозернистая классификация. Классы "правильно" и "неправильно" очень похожи и требуют от модели высокой точности. Задача также подразумевает работу с большим количеством объектов в кадре и возможными перекрытиями. Датасет: face-mask-detection (https://universe.roboflow.com/facemaskdetection-zvlot/face-mask-detection-oo94u).

***** – задачи имеют повышенный уровень сложности.