МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ
Ордена Трудового Красного Знамени федеральное государственное
бюджетное образовательное учреждение
высшего образования

«Московский технический университет связи и информатики»
(МТУСИ)

ОТЧЕТ

по производственной практике
(вид и тип практики)

студента **Чекалова Павла Владиславовича**

группы **БВТ2202**

(ФИО, подпись)

31.01.2026 (дата)

Москва
2026

Оглавление

Введение

В современном мире все чаще возникает задача автоматизированного обнаружения и отслеживания объектов и событий. Например, для контроля популяции животных в заповеднике, отслеживания нарушений правил парковки и оценки загруженности автодороги. Подобные задачи решаются с помощью нейросетевых алгоритмов, количество которых растет с каждым годом. И если для разработки и обучения собственной нейросети требуется множество машинного времени на обучение и человеческих часов на подготовку данных, то для небольших (локальных) проектов можно воспользоваться уже предобученными моделями. Однако, помимо разработки или подбора самой модели, необходимо также интегрировать данную модель в инфраструктуру организации. В рамках данной учебной практики перед нами была поставлена задача: разработать механизмы интеграции готовой нейросетевой модели в аппаратную инфраструктуру гипотетического предприятия.

Цель работы

Освоить полный цикл разработки системы искусственного интеллекта для обработки изображений: от выбора архитектуры нейронной сети до внедрения модели в веб-приложение с визуализацией результатов.

Этапы выполнения:

1) Ознакомление с вариантом (можно взять свой по согласованию с руководителем практики).
2) Подбор архитектуры нейронной сети.
3) Реализация веб-интерфейса. Создать простое веб-приложение. Реализовать: Загрузку изображений/видео/стрим с камеры, кнопку запуска обработки, вывод результатов.
4) Интеграция предобученной модели.
5) Вывод статистики в веб-интерфейсе.

Вариант индивидуального задания:

Контроль катания на скейтборде в запрещенных местах

Github: https://github.com/MasterOfUniverses/7sem_prac#

Основная часть

Постановка задачи

Ввиду немногословного описания индивидуального задания и невозможности разработать универсальное программное решение в короткие сроки была выработана следующая «легенда»:

1) *Заведомо известно, что камеры стоят в местах, где катание на скейтборде запрещено*. Правила, регулирующие катание, могут очень сильно различаться между городами, областями и странами. Поскольку географических данных у нас нет, введем такую гипотезу.
2) *Наше предприятие обладает собственной инфраструктурой камер наблюдения, а потому обладает минимальной серверной инфраструктурой*. Данное предположение обобщает логические выводы из самой постановки задачи – для задач машинного зрения совмещенных с выводом данных через веб-интерфейс требуется хотя бы один небольшой сервер, поскольку на самих камерах совмещать оба процесса затруднительно.
3) *Предприятие обладает возможностью арендовать удаленный сервер для обработки большого потока изображений.* Современные нейросетевые модели разработаны под вычисления на GPU или TPU, в то время как обычный CPU ПК средней ценовой категории при наличии всего 3 видео-потоков будет показывать скорость около 2 секунд на кадр. Потому для задач массовой нейросетевой обработки изображений крайне рекомендуется арендовать (или собрать) специальные серверные машины.
4) *Предприятие, как контролер общественного поведения, должно снизить число повторных штрафов.* В подавляющем большинстве законодательств нельзя оштрафовать дважды за одно и то же нарушение. Поскольку задача разработчика максимально автоматизировать процесс контроля общественного поведения, нам необходимо не только производить детекцию и анализ возможных нарушений, но и разделять нарушения по времени, а потому выполнять задачу отслеживания (треккинга) нарушителей.
5) *Мы являемся независимым разработчиком с жесткими ограничениями по времени*. Поэтому основной объем работ будет включать разработку по профильным направлением – интеграции нейросетевой модели в структуру простого веб-сервера. *Потому*

задача UI/UX-дизайна в данный момент не стоит. Добавить стили и поменять шаблоны веб-страниц можно в любое другое время.

6) Главная задача автоматизации – отслеживание факта и количества нарушений. Данные события генерируют относительно мало статистики, а потому вместо записи статистики в таблицу (которая включала бы в себя только поле количества нарушений) будет стоять задача фиксации нарушений – а *именно сохранение изображений предполагаемых нарушителей*. И возможность эти данные выгрузить через веб-интерфейс.

Выбор архитектуры

Исходя из вышеописанной «легенды» была разработана следующая архитектура интеграционного решения:

1) Сбор изображений с камер производится на стороне предприятия с помощью простого скрипта. На обработку могут быть поданы как дескрипторы подключенных камер (в рамках тестирования – камера ноутбука), так и фрагменты видео-файлов. Разработка варианта подключения камер через веб-интерфейс в данный момент не производилась (параметры подключения камеры через такие интерфейсы могут сильно различаться).
2) Скрипт отправляет данные на удаленный сервера для дальнейшей обработки нейросетевой моделью. Отправка данных производится через интерфейс сокетов, ввиду легковесности и простоты подобного решения. На данном этапе данные перемещаются из локальной инфраструктуру в удаленную («арендованный сервер с GPU»). Потому можно использовать низкоуровневые интерфейсы.
3) На сервере данные кэшируются до и после обработки нейросетью. Тесты на доступном оборудовании показали, что прием и отправка данных происходят в разы быстрее их обработки, а потому данные необходимо кэшировать. Кэширование производится с помощью двух потоко-безопасных очередей для каждого из клиентов.
4) Поскольку конечной целью стоит представление данных через веб-интерфейс, каждому клиенту нужно имя. Имя должно быть уникальным для всего сервера – данное решение помогает как в архитектуре веб-представления, так и в визуальном представлении. Кодирование клиентов по id или адресам очень неинформативно.
5) Отсюда следует такая последовательность действий клиента: отправка на сервер своего имени (и размера данных с

изображением), получение ответа об уникальности имени. Если имя неуникально – попытка оперативно поменять имя (путем добавления дополнительного числового идентификатора), если имя уникально – захват локального видео-потока и начало передачи изображений на сервер.

6) Клиент состоит из двух классов: класса инкапсулирующего всю логику обработки данных Sender и класса отвечающего за считывание аргументов запуска, запуск (и перезапуск) обработки данных Runner.
7) Также клиент включает в себя 3 вспомогательных класса исключений и класс обработки прерываний MyInterrupter.
8) Для сервера на этапе приема данных необходимо подтвердить уникальность имени, получить и преобразовать данные в объект изображения и передать данное изображение нейросети.
9) Сервер состоит из 4 частей – сервер приема данных (ServerReciever), сервер нейросетевой обработки (Server), сервер веб-представления с использованием фреймворка ***Flask*** (ServerSender) и запускающий класс Runner. Для инкапсуляции данных были разработаны классы DataSource, объединяющий все данные по клиенту и StatStruct – поле DataSource для хранения статистики.
10) Ввиду подхода с использованием преимущественно ООП парадигмы, и невозможностью работать с Flask-приложением как полем класса (напрямую), был создан вспомогательный класс EndpointAction. Для рендера ссылок на видео-потоки на главной странице был создан вспомогательный класс element. Для мониторинга времени обработки кадра был создан вспомогательный класс Timer. Для обработки прерываний - MyInterrupter.

Разработка программного решения

Клиентская часть

Класс клиента выполняет следующие функции: создает сокет для передачи данных на сервер, рассчитывает размер данных, отправляет запрос на утверждение своего имени, читает ответ, проводит сериализацию изображений и отправку их по сети. Также нужно запустить все эти функции в правильном порядке и иногда проверять, открыт ли сокет.

В создании сокета и процессе подключения к серверу нет ничего необычного – функции работы с сокетами примерно одинаковые по всех языках. А вот процесс автоматической кодировки (сериализации) данных можно рассмотреть подробнее.

Для байтовой сериализации данных использовались два подхода – для запроса разрешения от сервера и получения ответа использовалась функция struct.pack(format, data…) и соответственно struct.unpack(). Данная функция преобразовывает несколько простых объектов данных в байтовый массив согласно указанному формату. В строке запроса хранились две величины – размер предстоящего кадра и текстовое имя клиента. Размер кодировался литералом “Q”, а для кодирования строки было 2 решения – кодирование автоматического размера (до 255 байт) и кодирование фиксированного размера. Первый вариант не работал в тестируемых условиях, поскольку требовал понимания длины строки. Поэтому был использован второй вариант, с ограничением по длине имени в 250 символов (литерал “250s” – 250 символов единым объектом строки). Для логического значения ответа – литерал “?”.

Для сериализации самого изображения использовался модуль pickle, который производил полную сериализацию сложного объекта (метод dumps() для упаковки данных и метод load() для извлечения).

Однако, для экономии времени на передаче и обработке сетью, клиент также изменяет размеры изображения так, чтобы ширина изображения стала 640 px (величина эмпирическая, данный этап нуждается в дальнейшей замене на ограничение размеров только по средствам итогового веб-интерфейса). Также изображение необходимо перевести в другой цветовой формат. Эти задачи, как и чтение кадра из видеопотока, выполнялись с помощью библиотеки open-cv (модуль cv2), изученной в рамках нескольких академических дисциплин. Также для тестирования и демонстрации клиентской части решения с помощью данной библиотеки производится зацикливание видеофайлов. Для дескрипторов видеокамер вводится ограничение по частоте кадров (5 кадров в секунду – примерное частота обработки кадров на текущем устройстве).

Раз в 1000 кадров (эмпирическое число) производится проверка состояния сокета, методом попытки неблокирующего чтения из него. Если данная операция заканчивается ошибкой ConnectionResetError, сокет признается закрытым и освобождается вместе с захватом видеопотока.

Вторая составляющая клиента – класс запуска Runner. Он выполняет чтение аргументов командной строки (модуль argparse) и попытку запуска клиента с обработкой исключений. Некоторые исключения были созданы специально для передачи информации о цикле работы между классами – например, класс SenderWrongNameException сообщает о необходимости перезапуска под другим именем. В случае сетевых ошибок между перезапусками делается перерыв в 10 секунд.

Серверная часть

Описание модуля чтения данных с клиента здесь сокращено, поскольку во многом повторяет стандарты клиент-сервера на сокетах. Сервер создает по потоку на каждого клиента, в которых ведется блокирующее чтение данных. Каждые 10 кадров вычисляется среднее время чтения одного кадра, каждые 1000 – проверка состояния сокета.

Модель данных

Отдельного внимания заслуживает модель данных. Данные клиента инкапсулированы в классе DataSource и содержат следующие поля:

- Имя клиента
- Объект сокета клиента
- Состояние сокета клиента
- Размер изображения
- Наличие потока обработки нейросетью
- Очередь queue.Queue для входящих изображений
- Очередь с отрисованными результатами
- Среднее время (на 10 кадров) чтение от клиента
- Среднее время (на 10 кадров) обработки нейросетью
- Поле для отслеживания траектории
- Поле со статистикой

Статистика также инкапсулирована в класс StatStruct и содержит кортеж для данных с последнего кадра (раньше здесь была очередь как для изображений, но, ввиду иного подхода к подготовке html-документа, необходимость в очереди исчезла) и 3 множества (set) под id отслеживаемых объектов (человек, скейт, скейтбордист). Об этих полях подробнее в разделе про нейросеть.

Разработка серверной части начиналась с модуля чтения данных. Потом к структуре данных клиента добавилось поле отслеживания потока

(чтобы периодически подключать на обработку новых клиентов – их надо было отделить от старых). Потом возникла необходимость отслеживать траекторию. Далее появилась структура для статистики (решение сделать все на множествах пришло сразу – исходя из парадигмы «минимизации штрафов»). Последними были добавлены таймеры – как дополнительная статистика и задел на будущую оптимизацию хранения кадров (к сожалению, сроки ограничены).

Работа с нейросетью

Следующим этапом стала разработка алгоритма обработки данных нейросетью. Была выбрана нейросеть Yolo (версия 26-nano – с минимальным количеством параметров). Одной из причин подобного выбора стал ее родной датасет – COCO, который содержал необходимый нам класс скейтборда. Вторым важным фактором является способность модели к отслеживанию объектов. А третий фактор – это модель детекции в один проход (в отличие от R-CNN), что должно немного ускорить обработку потокового ввода.

Изначально сервер с нейросетью планировался как основной класс всей серверной части, однако данная архитектура не была оптимальной и была заменена на вызывающих класс Runner, что уменьшило связность кода между собой.Runner содержит словарь (name : DataSource), на который ссылаются остальные классы.

Как и класс чтения клиентов, нейросетевой сервер содержит один общий поток, в котором проверяет каждый из источников на наличие у того потока, и по 1 потоку на источник, которые создаются в результате этой проверки. (Напоминает цикл accept сокет-сервера.) Для задачи отслеживания на каждый поток данных требуется собственный экземпляр нейросети. Также у каждого потока свой таймер.

Внутри цикла обработки клиентских данных на каждом этапе из DataSource выбирается самый старый кадр (из соотв. очереди), данный кадр отправляется на вход нейросети, а далее – на анализ результатов.

В процессе анализа необходимо сначала рассортировать полученные bounding box («окна»). В рамках одного кадра запоминаются все окна со скейтами и только те окна с людьми, которые стабильно отслеживаются моделью (box.is_track). Опытным путем установлено, что модель хорошо отслеживает людей, но плохо отслеживает скейты. На этапе фильтрации мы сохраняем полный объект «окна». Однако, после фильтрации мы проходим

по всем обнаруженным окнам и добавляем в глобальную память (множества в статистике DataSource) id всех **отслеживаемых** окон (у окон без отслеживания id = None). Таким образом мы запоминаем только новые продолжительные по времени детекции. Для эффективной работы с множествами на кадр создается еще одно множество для id отслеживаемых людей.

В процессе фильтрации также устанавливались флаги наличия человека и наличия скейта (хотя эти данные можно получить напрямую из размера соотв. множеств окон). Если на кадре обнаружены оба типа объектов, то запускается цикл прохода по всем обнаруженным парам человека и скейта. Если для человека есть подходящий скейт и если его не было в множестве скейтеров, то он туда записывается, а участок изображения с ним вырезается (и сохраняется в виде .png файла, вместе с полным изображением), после чего работа с этим человеком прекращается. Для того, чтобы скейт можно было назвать подходяшим были выделены следующие условия:

1) Центр человека по горизонтали находится в пределах скейта
2) Верхняя грань окна скейта ниже 70% окна человека.

Первое условие отсекает тени и скамейки, которые модель ошибочно принимает за скейты. Второе условие также помогает отсекать тени, а еще способствует более корректному соотнесению человека с ближайшим к нему скейтом.

Хотя данный анализ далеко не оптимален, он порождает минимум повторных срабатываний, что важно для нашей задачи. Для полного устранения повторных срабатываний можно обработать полученный фрагменты с помощью карты признаков и найти совпадающие детекции (однако выполнять это лучше в качестве отдельного процесса, уже никак не зависящего от текущей задачи).

Последним этапом анализа является подсчет объектов на экране – если для людей и скейтов все понятно – там достаточно посмотреть на размер множества, то чтобы корректно определить скейтера необходимо проверить человека на то, был ли он на каком-либо из прошлых кадров скейтером. Для этого мы ищем пересечение множества id людей на экране с множеством всех скейтеров (в глобальных данных хранятся только id, потому при переборе и добавлении данных создавалось дополнительное множество).

Структура файловой системы: ./{name}/{name}_{id}(_full).png

Такая структура позволяет записывать уникальные детекции для каждого потока, чтобы в дальнейшем все детекции для потока одной папкой записать в архив.

После этапа анализа идет этап отрисовки – из окон выбираем только отслеживаемые, рисуем рамки, получаем их центры и id. После чего добавляем центры в словарь DataSource.track_history по соотв. id. Если длина словаря больше определенного порога (в нашем случае – 60), то удаляем самую первую запись. После – преобразуем словарь в список координат точек и строим незамкнутый контур через cv2.polylines(). Если отслеживаемых окон нет, метод отрисовки отобразит все известные окна.

После отрисовки следов изображение кодируется в jpeg и в виде массива байт записывается в очередь на отображение.

Веб-сервер

За основу веб-сервера был взят фреймворк Flask. У него есть два существенных недостатка – трудно его использовать внутри класса – для этого методы конечных точек надо преобразовать в Action-объекты (для чего создан вспомогательный класс EndpointAction). А еще поток с Flask-приложением должен быть основным в процессе. Потому при запуске в Runner сервера чтения и обработки запускаются во вторичных потоках.

Однако данный веб-сервер поддерживает базовую валидацию параметризированных адресов, потоковый ответ, рендер шаблонов Jinja и удобную отправку файлов.

Порядок создания путей был примерно следующий – сначала была создана главная страница, на которой предполагался список ссылок на страницы со стримами. Для этого совершается проход по всем ключам словаря источников и для каждого источника создается соответствующая ему ссылка. Ссылка и имя инкапсулируются в класс element и добавляются в список. В шаблоне выполняется проход по данному списку и для каждого элемента создается новый пункт (ненумерованного списка) содержащий ссылку под именем источника.

Для корректной работы внутри класса потребовалось также определить методы add_endpoint() – оборачивает метод в Action и привязывает к адресу и prepare_routes() – объявляет все пути сервера.

Следующим этапом стала потоковая отправка видео. Для этого потребовалось разделить отправку на 3 части – отправку шаблона с тегом-запросом, отправку ответа на данный запрос с заголовком и потоковую отправку кадров. В заголовке необходимо было указать специальный тип медиа – multipart/x-mixed-replace. Данный тип сообщает браузеру, что далее последует потоковая передача изображений (согласно всем изученным источникам – браузеры не воспринимают потоки этого типа, состоящие из других типов данных – для текста пришлось изменять подход к потоковой передаче). В самой же отправке кадров добавлялся заголовок, содержащий разделитель (обязательный параметр boundary типа multipart), тип кадра – image/jpeg – и сам кадр. Кадр забирался из готовой очереди на веб. Ответ должен быть ввиде байтовой строки.

Данный ответ упаковывается в тег <image> и автоматически обновляется при поступлении новых данных. Для более равномерного обновления между отправками данных была добавлена задержка в 100ms.

Однако оставалась следующая проблема – при выходе со страницы видео переставало подгружаться, а потому (при работе с подключенной камерой) происходила рассинхронизация. Для решения это проблемы очередь очищалась при каждом новом запросе, а после добавлялась задержка в 100ms на генерацию новых кадров.

Однако данный поход не является оптимальным – лучше вместо очереди использовать циклический список ограниченной длины (но для подобного изменения в архитектуре не хватило времени разработки – требуется рассчет оптимального размера списка и алгоритм движения по списку).

Следующим этапом стало отображение статистики. Первой идеей была повторить уже существующий код с использованием multipart типа. Однако среди данного семейства не нашлось ни одного подходящего под эти цели типа, а с x-mixed-replace ни один тег не работал как с текстом (проводились эксперименты с тегами <embed>, <object>, <iframe>). Также проводилась попытка написания скрипта с XMLHttpRequest. Однако подобная технология обработки x-mixed-replace признана устаревшей. Проводилось еще несколько попыток работы через WebSockets и SSE.

Итоговым решением стало расширение HTMX. Встраиваемый тег отправляет запросы по триггеру и обновляет данные страницы. Триггером может быть не только нажатие на кнопку, но и простая периодическая

отправка (например, раз в 500ms). Триггеры могут быть составными, но под конкретную задачу они не пригодились. Метод запроса был определен как GET, а действие – замена текста внутри данного тега (тег div). Итоговая строка с учетом шаблонов Jinja была такой:

Отправка запроса раз в 500ms обусловлена скоростью работы нейросетевой модели (около 350ms на кадр).

Однако для работы с данной моделью запросов потребовалось немного переписать код. Вместо асинхронной отправки кадров, отдельно от главного ответа, в данной модели каждое срабатывание триггера – это новый запрос, а потому каждый запрос обрабатывается как обычный запрос с полноценным ответом. В данной модели синхронизация методом очистки очереди при новом запросе просто не сработает – очередь будет всегда пустая, а время отклика вырастет на величину задержки генерации. Поэтому для хранения статистики за текущий = последний кадр используется простой кортеж (3 поля соответственно 3 классам объектов).

Последним этапом стала выгрузка изображений-результатов в zip-архиве. Для этого потребовалось подключить модули os, io, zipfile. Все файлы сгруппированы по соответствующим папкам, а потому корень для архивации уже известен. Далее создается байтовый поток ввода-вывода, на котором в режиме записи открывается zip-файл. И начинаем исследовать предложенный корень. Для всех файлов получаем полный путь и записываем их в zip-файл, сохраняя относительный путь. Далее необходимо переместить указатель байтового потока на начало этого потока, и можно будет отправить этот поток сразу как файл (для него генерируется имя и указывается, что этот файл передается на скачивание).

Возможность дальнейшего развития проекта:

Данный проект еще далек от полного завершения – анализ данных здесь выполнен в лоб, а при наличии времени и ресурсов стоило бы переделать анализ на дообучение нейросети (возможно, используя текущий метод анализа для разметки данных). В ходе работы могут возникнуть ошибки, которые в данный момент никак не обрабатываются (например, ошибка десериализации или ошибка занятого сокета – несмотря на специальную обработку прерываний, слушающий сокет не всегда освобождается вовремя – если ускорить сборщик мусора нельзя, то можно

поискать возможности для работы на 2-3 сокетах). Веб-интерфейс может быть значительно улучшен (добавлены стили, классы и скрипты). Также полезным было бы распределить клиентскую нагрузку между разными копиями сервера, с учетом уже хранимых ключей – т.е. настроить контейнеризацию и оркестрацию.

Результат: была разработана архитектура интеграции системы машинного зрения в существующую инфраструктуру компании согласно данным условиям. Данное программное решение хоть и является минимальным жизнеспособным продуктом, однако многие архитектурные решения могут оставаться жизнеспособными на протяжении многих версий рефакторинга данной разработки. На следующую стадию разработки данного проекта необходимо хотя бы 3-4 человека и минимум 1 месяц времени. (Для большинства задач, за исключением нейросетевого анализа.)

Заключение

Полный цикл разработки интеграционного решения – это огромный спектр задач. Несмотря на довольно удачную попытку пройти этот цикл в одиночку за неделю, в реальных проектах для такой задачи потребуются месяцы и хотя бы небольшая команда специалистов. Тем не менее, было успешно разработано решения для задачи детекции и отслеживания объектов с камер наружного наблюдения.