СОДЕРЖАНИЕ

ВВЕДЕНИЕ

С каждым днем мы все чаще сталкиваемся с устройствами Интернета вещей (Internet of Things, IoT), которые призваны упростить нашу повседневную жизнь. Концепция IoT подразумевает подключение различных устройств между собой, которые обмениваются данными как в рамках внутренней сети, так и внешней. Классическим примером данной концепции является ”умный” дом, когда, например, мы можем регулировать уровень освещенности в комнате с помощью голосового управления на телефоне.

Однако технологии Интернета вещей внедряются не только в нашу обыденную жизнь, но и в промышленности. Так называемый промышленный Интернет вещей (Industrial Internet of Things, IIoT) уже сегодня позволяет повысить эффективность процессов производства. Типичный производственный объект может быть источником большого количества данных, собранных с датчиков, анализ которых позволит обнаружить и устранить проблему на ранних этапах ее появления. В качестве примера внедрения IIoT можно вспомнить компанию Amazon, которая активно применят роботов и различные датчики на своих складах для более быстрого поиска нужных товаров [1].

На сегодняшний день устройства IoT и IIoT чаще всего применяются в связке с облачной инфраструктурой. То есть данные, собираемые устройствами отправляются в ”облако”, где дальше уже сам анализ происходит на машинах с большими вычислительными ресурсами. Такой подход больше применим для долговременной аналитики, но если результат обработки нужен ”здесь и сейчас”, то задержки при передачи данных от устройств до центров обработки данных (ЦОД) могут быть критичны. В связи с этим начали набирать популярность граничные вычисления (edge computing), когда обработка данных происходит рядом с конечными устройствами. Так как количество данных с каждым годом будет только расти, то можно утверждать,

что граничные вычисления будут находить все более широкое применение. Согласно исследованиям компании Gather, к 2025 году около 75% данных будет обрабатываться на границе, не доходя до ЦОД [2].

Часто для анализа данных собранных с устройств IoT и IIoT применяются методы глубокого обучения, которые позволяют установить более сложные зависимости во входных данных. Глубокое обучение подразумевает построение искусственных нейронных сетей для решения прикладных задач. Методы глубокого обучения появились еще в 80-х годах, но начали стремительно набирать популярность около 2010 года. Обусловлено это несколькими причинами. Во-первых, с ростом Интернета увеличивалось количество данных, которые можно было использовать для обучения. Во-вторых, с развитием графических процессоров (Graphics Processing Unit, GPU) процесс обучения нейронных сетей сократился в десятки раз. Впоследствии предобученные модели могут запускаться на устройствах и без GPU, но при этом демонстрировать приемлемое качество работы.

Исторически задача распознавания объектов на изображениях являлась тяжелой, но с развитием методов глубокого обучения удалось повысить эффективность решения данной задачи. Сегодня предобученные модели для распознавания объектов могут быть запущены даже на компьютерах с ограниченными вычислительными ресурсами (КОВР). В данной работе под КОВР подразумевается одноплатный компьютер без каких-либо специальных модулей для аппаратного ускорения методов машинного обучения. Классическим примером КОВР является одноплатный компьютер Raspberry Pi. Чаще всего КОВР собирают данные с различных устройств, проводят их первичную предобработку и далее отправляют их для полного анализа в ”облако”. Однако при текущем развитии КОВР, их вычислительные ресурсы могут использоваться для куда более сложных вещей, таких как распознавания объектов на видеоизображениях. Вполне ожидаемо, что одному экземпляру КОВР решить такую задачу будет сложно, но при объединении нескольких

экземпляров КОВР в кластер можно добиться приемлемых результатов. Основными преимуществами использования кластера из КОВР являются:

1) снижение затрат на анализ видеоизображений. Во-первых, если на объекте уже имеются экземпляры КОВР, то для решения этой задачи может использоваться существующая инфраструктура. Во-вторых, развертывание кластера из КОВР требует меньше финансовых ресурсов по сравнению с долговременным использованием облачной инфраструктуры;
2) уменьшение задержек при передаче видеоизображений, что способствует увеличению отзывчивости систем, которые опираются на анализ этих данных;
3) повышение конфиденциальности собранных данных за счет их локальной обработки.

Приведенные выше утверждения говорят об **актуальности** данной работы. Главной **проблемой** на сегодняшний день является отсутствие в открытом доступе каких-либо полноценных решений для задачи распределенного распознавания объектов на видеоизображениях на кластере из КОВР. В связи с этим, **целью** данной работы является организация распределенного распознавания объектов на видеоизображениях на кластере из КОВР. Для достижения поставленной цели требуется решить следующие **задачи**:

1) провести обзор методов для распознавания объектов на изображениях;
2) провести анализ особенностей архитектуры КОВР;
3) разработать алгоритм для распределения вычислений, необходимых для распознавания объектов на видеоизображениях, между узлами кластера;
4) провести тестирование разработанного алгоритма на кластере из КОВР.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Распознавание объектов на изображениях

1.1.1 Исторический обзор

Задачи распознавания объектов на изображениях являются основными в области компьютерного зрения. Разработанные методы могут применяться в различных сферах, начиная от идентификации пользователя по фотографии, заканчивая сложными системами видеонаблюдения и робототехникой. В целом, все существующие методы можно разделить на две категории: методы ручного выделения признаков (handcrafted features) и методы, основанные на нейронных сетях. При использовании методов ручного выделения, какие-то признаки изображений выделяются по заранее определенному алгоритму с учетом специфики данных. Примером таких признаков могут служить границы объектов. Так как с учетом специфики изображений мы знаем, что если значения соседних пикселей сильно различаются, то это может свидетельствовать о том, что в этом месте проходит граница между двумя объектами. В свою очередь, методы с применением нейронных сетей устроены по другому. Нейронная сеть сама определяет набор признаков на основе данных, полученных в процессе обучения.

До 2012 года наибольшей точностью обладали методы с ручным выделением признаков. Тогда пользовались популярностью методы основанные на гистограммах направленных градиентов и признаках Хаара. Однако в 2012 году эта тенденция изменилась после того, как на соревновании ImageNet Large Scale Visual Recognition Challenge (ILSVRC) методы с применением искусственных нейронных сетей показали лучшие результаты.

Участникам соревнования ILSVRC предлагается разработать алгоритмы для задач классификации изображений и обнаружения объектов [3]. Сами соревнования начали проводиться с 2010 года и использовали ImageNet, в качестве базы данных изображений. ImageNet - это база данных, которая

содержит более 14 миллионов изображений, объекты на которых могут относиться к одному из 1000 классов [4].

На рисунке 1.1 представлена диаграмма с лучшими решениями задачи классификации на ILSVRC с 2011 по 2016 год. По горизонтальной оси откладывается время, по вертикальной - процент ошибок. Для каждого из годов в скобках указано название предложенного метода. Как можно заметить, в 2012 году произошел достаточно резкий скачок в точности классификации изображений, почти на 10%. Этот прирост обусловлен тем, что авторы решили отойти от классических подходов и разработали метод, в основе которого лежали сверточные нейронные сети (СНС). В последующие года победителями также становились методы, основанные на СНС, а в 2015 году предложенное решение стало классифицировать изображения даже точнее, чем это может выполнить человек.

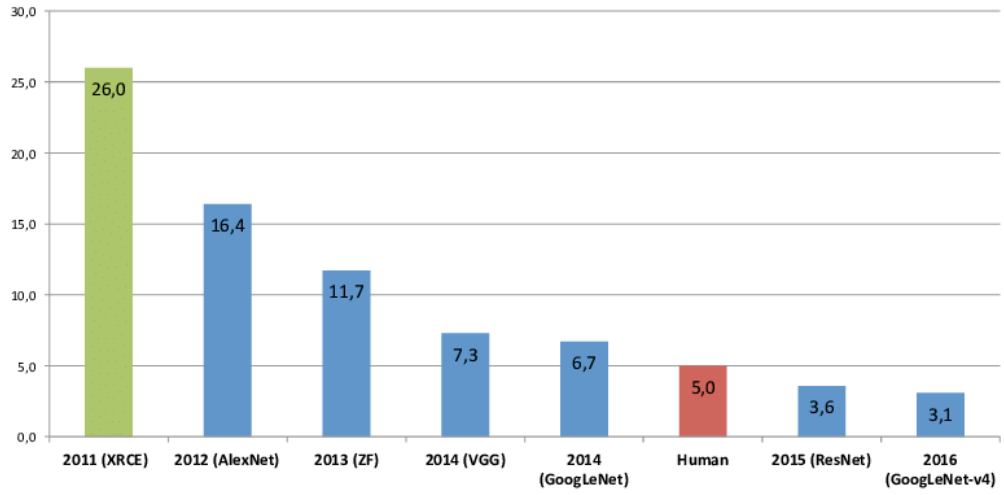

Рисунок 1.1 – Лучшие решения ILSVRC с 2011 по 2016

1.1.2 Сверточные нейронные сети

Идея сверточных нейронных сетей (СНС) появилась достаточно давно, примерно в начале 80-х годов [5]. По принципу своей работы СНС схожи с тем, как функционирует зрительная кора головного мозга человека. Именно из-за этого сходства СНС удается достигать наилучших результатов в различных задачах компьютерного зрения.

СНС получили свое название из-за операции свертки (convolution), которая как раз и выделяет интересующие признаки на изображениях. При операциях свертки используется так называемые фильтры (filters) или ядра (kernels). В случаях двумерной свертки это двумерная матрица весов нечетного размера (обычно 3×3 или 5×5). Операцию двумерной свертки для изображения заданного в оттенках серого (то есть каждый пиксель занимает один байт) можно описать следующим образом:

1) переместиться в левый-верхний угол входного изображения, таким образом, чтобы каждому весу фильтра соответствовал пиксель изображения;
2) выполнить произведение соответствующих пикселей изображения и весов фильтра;
3) просуммировать полученные произведения. Эта сумма и будет результатом свертки области изображения под фильтром;
4) сдвинуть фильтр вправо на величину шага (stride). При достижении правой границы изображения спуститься на строку ниже;
5) вернуться на шаг два, пока полностью не будет обработано все изображение.

Как можно понять из приведенного описания, операция свертки уменьшает размер результирующего изображения, то есть происходит потеря информации на краях изображения. Для решения этой проблемы стали использовать дополнение (padding). Чаще всего применяется нулевое дополнение, когда изображение по краям дополняется нулями. Пример нулевого дополнения для картинки размером 7×7 и размером фильтра 5×5 представлен на рисунке 1.2.

В настоящее время большинство изображений используют цветовую модель RGB, когда каждый пиксель кодируется тремя числами – интенсивностью соответственно красного, зеленого и синего цветов. Поэтому фильтрам

в операции свертки добавляют третью компоненту, чтобы покрывались все каналы входного изображения.

Рисунок 1.2 – Пример нулевого дополнения

Основными составляющими СНС являются сверточные слои, где обычно происходит порядка 90-95% всех вычислительных операций [6]. Каждый из слоев может содержать в себе некоторое количество фильтров, размер и количество которых определяются архитектурой СНС. После применения фильтра к результату предыдущего слоя формируется так называемая карта признаков (feature map). То есть каждый фильтр в сверточном слое выделяет только те признаки, которые он выучил в процессе обучения. В итоге фильтры первого сверточного слоя будут выделять какие-то простые признаки (прямые, наклонные), далее фильтры второго слоя смогут выделять уже полноценные фигуры (квадраты, треугольники) и чем глубже будет находиться слой, тем все более сложные конструкции он сможет выделять.

Так как для всей операции свертки используются одинаковые веса фильтров, то это приводит к более быстрому обучению сверточных слоев по сравнению с теми же полносвязными слоями. Как раз идея разделения весов фильтров является уместной для задач обработки изображений, так как основным свойством всех изображений является их пространственная инвариантность [7]. Иными словами любая форма на изображении должна обрабатываться аналогичным образом, в не зависимости от ее расположения.

За сверточным слоем в СНС следует операция активации. Основной задачей функции активации в нейронных сетях является добавление нелинейности, иначе при отсутствии функции активации (либо при использовании линейной функции активации) увеличение количества слоев не давало бы никакого выигрыша [7]. Иначе говоря при использовании функции активации нейронная сеть может выучить более сложные зависимости во входных данных.

Классическими примерами функций активации являются сигмоида и гиперболический тангенс. На рисунке 1.3 приведены графики данных функций активации. Однако в последнее время все чаще используется выпрямленная линейная функция активации (ReLU). Причин у этого несколько:

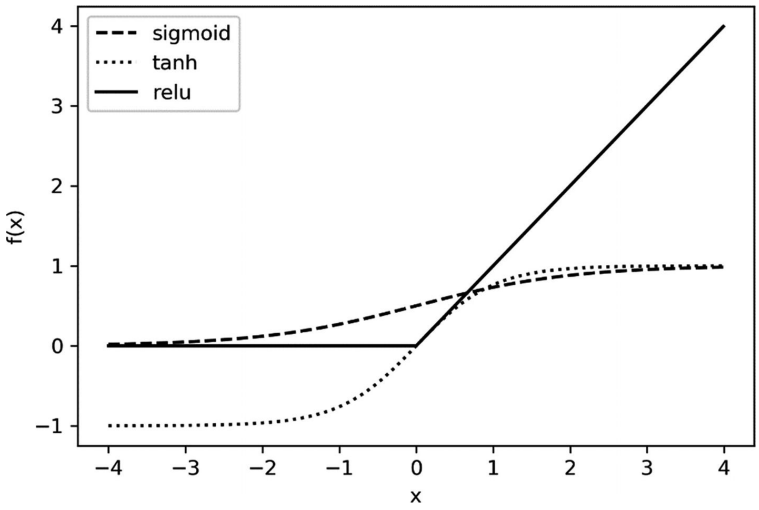

Рисунок 1.3 – Примеры графиков функций активации

1) значения функций сигмоиды и гиперболического тангенса изменяются незначительно при стремлении аргумента к $+\infty$ или $-\infty$. Это становится причиной проблемы затухающего градиента (vanishing gradient), что приводит к большему времени обучения для более глубоких слоев. Функция активации ReLU лишена этой проблемы. Именно благодаря этой особенности авторам известной СНС AlexNet удалось в шесть раз ускорить процесс обучения при использовании ReLU по сравнению с гиперболическим тангенсом [8];

2) расчет значений функций сигмоиды и гиперболического тангенса более ресурсоемкий с точки зрения центрального процессора, так как необходимо выполнять операции возведения в степень и деления.

После применения функции активации чаще следует операция субдискретизации (pooling). Существует несколько типов данной операции, но на данный момент широко используемой является операция взятия максимума (max-pooling). Сама операция состоит из разбиения результатов работы предыдущего слоя на локальные области размером $w \times h$ и взятия максимума из этой области.

Основной целью операции субдискретизации является уменьшение размеров карты признаков. Это приводит к тому, что последующие операции свертки будут оперировать над все большей областью исходного изображения. Также очевидным следствием данной операции будет уменьшение количества вычислений в последующих слоях. Помимо этого, в результате операции субдискретизации СНС становится более устойчивой к небольшим трансформациям изображения вроде сдвига или поворота [5].

После чередования нескольких сверточных слоев с операциями активации и субдискретизации, завершающим обычно служит полносвязный (fully connected) слой, который уже непосредственно соединяется с выходным слоем СНС. В действительности количество полносвязных слоев может быть больше одного, этот вопрос остается на усмотрение авторам СНС. Главной задачей полносвязного слоя является установления соответствия, какие из высокоуровневых признаков принадлежат к тому или иному классу. Чаще всего полносвязные слои из-за своей структуры содержат порядка 95% всех параметров СНС, но при этом в этих слоях происходит порядка 5-10% от всех вычислительных операций [6].

Обобщая все вышесказанное, типичная архитектура СНС состоит из чередующихся слоев свертки с операциями активации и субдискретизации и завершающим это полносвязным слоем. Структура выходного слоя опреде-

ляется задачей, решаемой СНС. Например, для задач классификации размер выходного слоя будет равен количеству классов. На рисунке 1.4 представлена архитектура известной СНС AlexNet без частей, связанных с разделением слоев между двумя GPU. Современные СНС могут содержать десятки сверточных слоев, поэтому часто операции активации даже не приводят на рисунках, подразумевая, что после каждого сверточного слоя следует ReLU активация. Операция субдискретизации обозначается как MP (сокращение от max-pooling) и можно заметить, что операция субдискретизации может отсутствовать на выходе некоторых сверточных слоев.

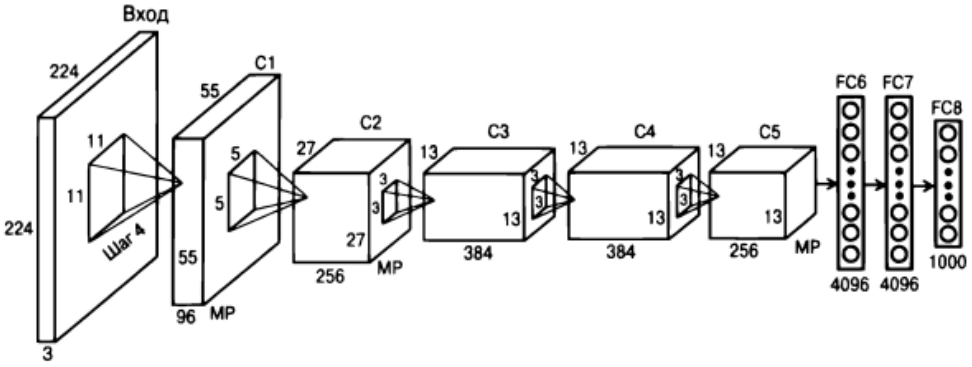

Рисунок 1.4 – Архитектура AlexNet без GPU-разделения [7]

1.1.3 Методы для распознавания объектов

После большого успеха СНС в решении задач классификации, данный тип нейронных сетей также начал широко применяться в методах для распознавания объектов. Под распознаванием объектов будем понимать обнаружение объектов на изображениях, то есть определение координат ограничивающей рамки (bounding box), и определение их класса.

На данный момент существующие методы для распознавания объектов можно разделить на два класса: одношаговые (one-stage) и двухшаговые (two-stage). В двухшаговых методах входное изображение разбивается на регионы (regions of interest, RoI) и далее каждый из RoI обрабатывается отдельно. Также определением координат ограничивающих рамок и классов объектов занимаются разные компоненты. При использовании одношаговых методов

входное изображение полностью обрабатывается нейронной сетью, которая одновременно предсказывает, и координаты ограничивающих рамок, и классы объектов. В общем случае двухшаговые методы требуют на обработку больше времени по сравнению с одношаговыми, но при этом обладают большей точностью распознавания [9].

R-CNN

Первым представителем двухшаговых методов стал метод Regions with CNN features (R-CNN). Тогда авторам удалось повысить среднюю точность (mean average precision) распознавания объектов на 30% по сравнению с предыдущими методами на датасете PASCAL VOC2012 [10]. На рисунке 1.5 представлено описание метода R-CNN. Рассмотрим каждый из шагов отдельно:

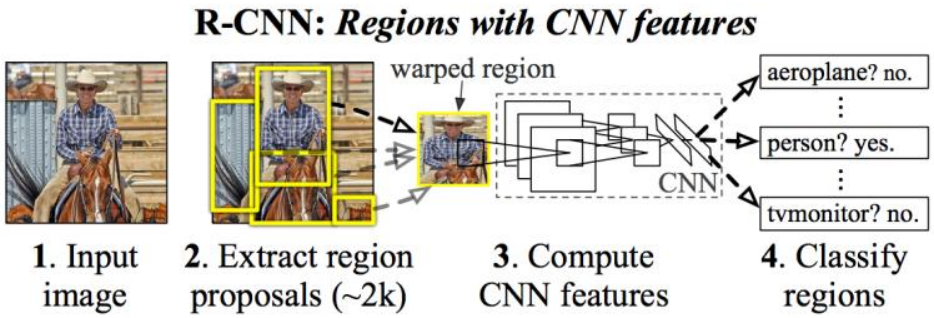

Рисунок 1.5 – Описание метода R-CNN [10]

1) изначально из входного изображения выделяются примерно 2000 RoI, на которых могут находиться объекты. Для поиска RoI авторы использовали метод селективного поиска (selective search). Упрощая, можно сказать, что метод селективного поиска определяет границы объектов по изменению цветов и интенсивностей соседних пикселей;
2) так как выделенные RoI могут быть разных размеров, то каждый из них необходимо приводить к входным размерам СНС – 227×227;
3) после каждый из RoI передается на вход СНС. В качестве СНС авторы используют архитектуру AlexNet, у которой удален последний полно-

связный слой. В связи с этим на выходе получается вектор признаков размерностью 4096;

4) сама классификация объектов осуществлялась с помощью метода опорных векторов (support-vector machine, SVM). Для каждого из классов был свой бинарный SVM-классификатор;
5) координаты объектов, полученные на первом шаге могут быть неточными, поэтому каждый из RoI в конце еще обрабатывается моделью линейной регрессии, отдельной для каждого из классов. По утверждениям авторов данный шаг увеличил среднюю точность на 3-4% [10].

Fast R-CNN

Метод R-CNN обладал хорошей точностью, однако общее время обработки изображения было крайне высоким. Например, при использовании в качестве CHC архитектуру VGG16 обработка одного изображения методом R-CNN занимала 47 секунд (измерения проводились при использовании одной GPU NVIDIA Tesla K40) [11]. У этого было несколько причин. Во-первых, проход 2000 RoI через CHC требовал много вычислительных и временных ресурсов, особенно ситуация ухудшалась для более глубоких CHC, таких как, VGG16. Во-вторых, модель линейной регрессии зависела от результатов классификации, что не давало выполнять эти операции параллельно. Также наличие 3 отдельных моделей требовало много времени и дискового пространства в процессе обучения [11].

Для решения вышеперечисленных проблем авторами R-CNN вскоре был представлен улучшенный метод – Fast R-CNN. На рисунке 1.6 приведено описание данного метода. Рассмотрим обновленный метод подробнее:

1) теперь входное изображение сразу поступает на вход CHC. Параллельно с этим из входного изображения выделяются RoI с помощью того же метода селективного поиска;
2) далее из карты признаков выделяется вектор признаков для каждого из RoI. Решением этой задачи занимается RoI pooling layer;

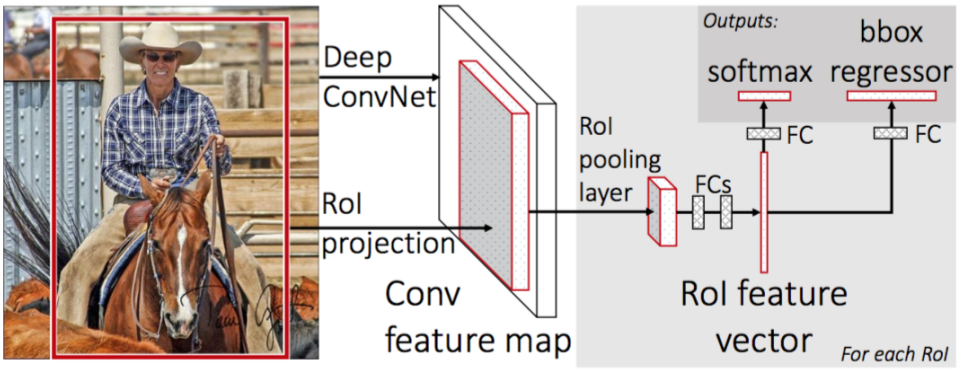

Рисунок 1.6 – Описание метода Fast R-CNN [11]

3) после вектор признаков передается в полносвязный слой, а затем следуют два параллельных слоя - Softmax и линейная регрессия. Слой Softmax определяет класс объекта, а слой линейной регрессии уточняет координаты ограничивающей рамки.

Faster R-CNN

Метод Fast R-CNN устранил основные проблемы первоначального метода R-CNN, но все равно еще плохо подходил для тех же задач обработки видео. Например, на обработку одного изображения уходило порядка двух секунд и большая часть этого времени тратилась на поиск RoI с помощью селективного поиска [12]. Для решения этой проблемы авторы метода Faster R-CNN заменили селективный поиск отдельной СНС, которую назвали Region Proposal Network (RPN). Все остальные компоненты метода Fast R-CNN остались без изменений.

На рисунке 1.7 иллюстрируется принцип работы RPN. На вход RPN передается карта признаков, ранее выделенных с помощью СНС. Далее по карте признаков проходятся ”скользящим окном” (sliding window) размером 3×3. Для каждого положения окна выделяется k RoI, где k это количество якорей (anchors). Якоря используются далее для определения абсолютных координат RoI. После наложения якорей на текущее положение скользящего окна, из карты признаков выделяется вектор признаков размерностью 256.

Затем полученный вектор передается в два параллельных сверточных слоя с размером фильтра 1×1. Слой *cls* имеет *2k* выходов для определения вероятности наличия/отсутствия объекта внутри RoI. В свою очередь слой *reg* имеет *4k* выходов, каждый из которых соответствует поправкам к координатам якорей.

Проведенные авторами тесты показывают, что предложенная ими RPN тратит порядка 10 мс на выделение RoI [12]. Благодаря этому удалось достичь скорости обработки порядка пяти кадров в секунду, что соответствует 200 мс на кадр. При всем при этом предложенный метод оказался более точным. На датасете PASCAL VOC2007 прирост средней точности составил порядка 9% по сравнению с методом Fast R-CNN.

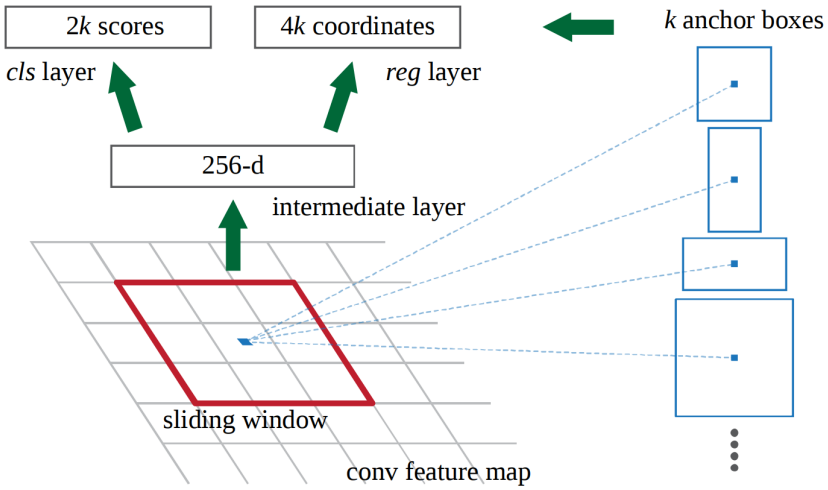

Рисунок 1.7 – Принцип работы RPN [12]

YOLO

Метод You Only Look Once (YOLO) является одним из первых представителей одношаговых методов. По утверждению авторов последняя версия метода YOLOv4 обладает лучшей эффективностью среди существующих методов [13]. Под эффективностью понимается отношение точности к времени обработки одного изображения. На текущий момент представлено четыре версии метода: YOLO [14], YOLOv2 [15], YOLOv3 [16] и YOLOv4 [13]. Рассмотрим обобщенно принцип работы данных методов.

Изначально авторы первой версии YOLO решили рассмотреть проблему распознавания объектов как проблему регрессии, то есть одновременное предсказание координат ограничивающей рамки и класса объекта внутри нее. Сам метод разделяет входное изображение на сетку размером $S \times S$. Каждому из объектов присваивается клетка (grid cell) из этой сетки. Если объект занимает более одной клетки, то ему присваивается та клетка, внутри которой расположен центр этого объекта. Каждая клетка предсказывает B ограничивающих рамок, разных форм и размеров.

Само предсказание для ограничивающих рамок состоит из пяти чисел: (x, y, w, h, a). Числа (x, y) представляют собой координаты центра ограничивающей рамки, относительно клетки объекта, а числа (w, h) соответственно являются шириной и высотой рамки. Число a представляет собой точность предсказания. Также для каждой из клеток сетки предсказывается вероятность принадлежности к тому или иному классу. Если предположить, что общее количество классов равно C, то на выходе из нейронной сети будет получаться тензор размером $S \times S \times (5B + C)$.

Так как для одного объекта может быть предсказано больше одной ограничивающей рамки, то для выбора наилучшей используют метод Non-Max Suppression (NMS). На рисунке 1.8 представлен результат работы NMS. Рассмотрим этот метод подробнее:

Рисунок 1.8 – Пример работы метода NMS

1) выбрать охватывающую рамку, имеющую самую большую точность предсказания;
2) отбросить все рамки, которые имеют площадь перекрытия с выбранной рамкой больше определенной границы. Для этого используется значение Intersection over Union, которое рассчитывается как отношение площади пересечения двух рамок к площади их объединения;
3) перейти к первому шагу, пока есть лишние рамки.

Также большим преимуществом метода YOLO является тот факт, что авторы предоставляют версии предобученных моделей для запуска в разных окружениях. Например, в последней работе авторы представили легковесную модель YOLOv4-tiny, которая очень хорошо подходит для запуска на устройствах с ограниченными вычислительными ресурсами, при этом средняя точность составляет 42% [17].

1.2 Архитектура процессоров ARM

1.2.1 Особенности

В настоящее время большинство устройств Интернета вещей находятся под управлением именно процессоров архитектуры ARM. Прямым конкурентом ARM является не менее известная архитектура процессоров – x86. Данная архитектура изначально разработана компанией Intel и большинство процессоров настольных компьютеров и серверов реализуют эту архитектуру. Однако со временем повсеместное использование процессоров x86 в перечисленных областях может склониться в сторону ARM, для этого достаточно вспомнить, как в 2020 году компания Apple перешла на использование в своих ноутбуках процессоров архитектуры ARM [18].

Архитектура ARM является типичным представителем архитектуры Reduced Instruction Set Computer (RISC), в то время, как x86 относится к архитектуре Complex Instruction Set Computer (CISC). Рассмотрим отличительные особенности архитектуры RISC и сравним их с CISC:

1) меньшее количество инструкций. Так как чаще всего используется только ограниченный набор популярных инструкций, то аппаратные блоки для менее популярных инструкций в архитектуре CISC не задействованы и только занимают место на кристалле. Недостатком этой особенности может быть увеличение размера программы, но при текущем количестве памяти у компьютеров это не является большой проблемой;
2) фиксированный размер и простой формат инструкций. Это приводит к тому, что операция декодирования инструкции занимает меньше времени и сама аппаратная структура декодировщика остается простой. В случае CISC операция декодирования будет занимать больше времени, даже для самых простых инструкций [19];
3) инструкции выполняется за один такт. Современные процессоры часто используют конвейеризацию инструкций и данная особенность RISC позволяет повысить производительность конвейеров;
4) большое количество регистров. Например, процессоры архитектуры ARMv8-a имеют 31 регистр общего назначения [20], в то время как x86_64 предоставляет только 16 регистров общего назначения [21]. В большей степени это связано с тем, что операции обработки данных в RISC в качестве операндов могут использовать только регистры, поэтому перед обработкой данных их нужно загрузить из памяти, а после сохранить результат, используя соответствующие инструкции.

Стоит также упомянуть, что для архитектуры x86 некоторые описанные выше проблемы CISC со временем начали усугубляться из-за необходимости сохранять поддержку не самых популярных инструкций, которые были добавлены десятки лет назад. Поэтому начиная с середины 90-х годов процессоры архитектуры x86 разбивают сложную инструкцию на несколько микроопераций (micro-operations), а после эти инструкции уже выполняются RISC ядром [19].

Также отличительной особенностью процессоров архитектуры ARM является пониженное энергопотребление, по сравнению с процессорами на x86 архитектуре. Например, тот же ARM процессор Apple M1 на данный момент обладает одной из наилучших производительностей на один ватт [18]. Такое достигается за счет использования гетерогенной архитектуры процессоров big.LITTLE, когда используется два типа ядер: высокопроизводительные (big) и энергоэффективные (LITTLE). Высокопроизводительные используются для требовательных задач, в то время как энергоэффективные активизируются для менее ресурсоемких задач, но при этом обладают пониженным энергопотреблением.

1.2.2 Наборы команд

Со временем разброс устройств, для которых применялись процессоры ARM стал увеличиваться и тогда компания решила ввести понятие профиля архитектуры, который характеризует класс решаемых задач. На данный момент определены три профиля [20]:

1) профиль приложений (application profile). К данному профилю относятся устройства, требующие высокой производительности, такие как: персональные компьютеры, планшеты, смартфоны. Процессоры данного профиля имеют поддержку виртуальной памяти за счет наличия устройства управления памятью (memory managment unit, MMU) [20];
2) профиль реального-времени (real-time profile). Процессоры данного профиля применяются для решения задач жесткого реального времени, то есть таких задач, где требуется гарантия, что обработка события займет не более определенного количества времени. Примером таких систем может служить антиблокировочная система тормозов на автомобилях и самолетах. Особенностью данного профиля является отсутствие MMU и наличие тесно связанной памяти (Tightly Coupled Memory, TCM). TCM - это область памяти, которая по своей струк-

туре похожа на кэш процессора, но содержимым этой памяти строго управляет программист и доступ к этой части памяти занимает одни такт процессора;

3) профиль микроконтроллеров (microcontroller profile). Подобные процессоры обладают пониженным энергопотреблением и позволяют быстрее обрабатывать внешние прерывания [20]. Может использоваться в качестве процессоров для микроконтроллеров или как составная часть системы на кристалле, например, может выступать контроллером ввода-вывода.

После выхода архитектуры ARMv8 у процессоров появилось два режима работы (execution state): aarch64 и aarch32. В режиме работы aarch32 используются 32-битные адреса, есть доступ к 13 регистрам общего назначения размером 32 бита, имеется 32 64-битных регистра для SIMD инструкций и в качестве набора инструкций могут использоваться A32 или T32 [20]. В свою очередь в режиме aarch64 используются 64-битные адреса, количество регистров общего назначения увеличилось до 31, размерность регистров для SIMD инструкций выросла до 128 бит, а в качестве набора команд может использоваться A64 [20]. Процессоры, начиная с архитектуры ARMv8, могут исполнять программы в двух режимах, тем самым сохраняется обратная совместимость с ранее разработанным ПО, для которого использовался набор инструкций A32 или T32.

Как ранее упоминалось, процессоры архитектуры ARM поддерживают SIMD (Single Instruction Multiple Data) инструкции, для этого используется расширение набора команд – NEON. Данный тип инструкций позволяет за одну инструкцию процессора выполнить несколько операций, например, перемножить между собой векторы чисел, каждый из которых содержит по 4 числа. Подобные инструкции могут значительно ускорить операции кодирования/декодирования видео, потому что при решении таких задач часто оперируют векторами чисел. Этими инструкциями можно воспользоваться

напрямую при написании на ассемблере, либо использовать интринсики компиляторов на языке C/C++. Также некоторые библиотеки могут использовать при своей работе SIMD инструкции, например, такую поддержку имеет популярная библиотека OpenCV, разработанная для решения задач компьютерного зрения.

2 РАЗРАБОТКА АЛГОРИТМА

2.1 Постановка задач

На сегодняшний день двумя популярными методами для распределения нейронных сетей являются распределение модели и данных. При распределении модели слои нейронной сети располагаются на нескольких компьютерах и входные данные сначала проходят через все слои на первом компьютере, далее результат отправляется на обработку последующим слоям на втором компьютере и так происходит, пока данные не пройдут через всю сеть. В методе распределения данных весь набор входных данных разбивается на части, и после каждая из этих частей отправляется на обработку определенному компьютеру, где уже запущена полная модель. Метод распределения модели чаще применяется в случаях, когда размер памяти, занимаемой моделью превышает количество памяти на одной машине. Отрицательным следствием этого метода является увеличение нагрузки на сеть, так как один элемент из входного набор данных будет приводить к нескольким обменам результатов между слоями нейронной сети. Метод распределения данных оказывает меньшую нагрузку на сеть, так как каждая часть единожды отправляется на обработку.

В данной работе будет использоваться метод распределения данных по нескольким причинам. Во-первых, при использовании метода распределения модели намного сложнее применять уже разработанные методы. Во-вторых, видеоизображения хорошо ложатся на метод распределения данных, так как видео представляет из себя набор кадров (изображений), каждый из которых может обрабатываться независимо от других. На рисунке 2.1 представлен принцип работы алгоритма в общем виде. Как можно понять из рисунка, алгоритм будет состоять из двух основных компонентов: планировщик и рабочий. Планировщик будет получать на вход видеофайл, разбивать его на кадры и далее отправлять кадры рабочим по определенному алгоритму пла-

нирования. Рабочий будет принимать кадр, применять метод распознавания объектов и отправлять список результатов обратно планировщику. Каждый результат будет представлять собой координаты ограничивающей рамки и идентификатор класса объекта. Затем планировщик будет отображать результаты и сохранять кадры в правильном порядке в выходное видео.

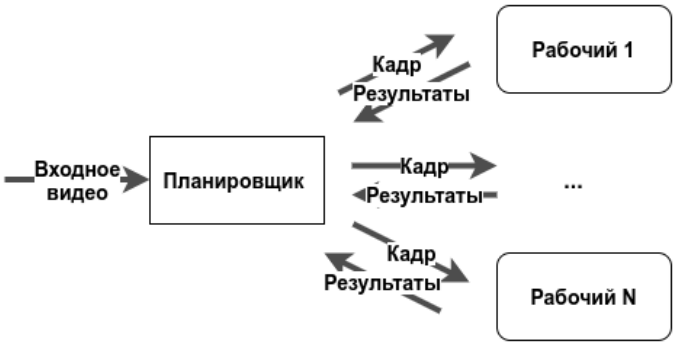

Рисунок 2.1 – Принцип работы алгоритма в общем виде

После того, как определились с общим принципом работы алгоритма, можно переходить к его реализации. Для этого поставим следующие задачи, которые нужно решить:

1) выбрать библиотеку, которая позволяет запускать разработанные методы для распознавания объектов на изображениях;
2) определить алгоритм планирования, согласно которому планировщик будет отправлять кадры рабочим;
3) разработать алгоритм для сохранения кадров в правильном порядке и учесть возможные потери кадров;
4) выбрать протокол транспортного уровня, посредством которого планировщик и рабочие будут обмениваться сообщениями;
5) определить формат сообщений для взаимодействия между планировщиком и рабочими.

2.2 Основные компоненты

2.2.1 Рабочий

В первой главе было установлено, что на данный момент самыми эффективными методами для распознавания объектов на изображениях являются одношаговые методы YOLO. При реализации данных методов авторы используют фреймворк Darknet, который реализован с поддержкой платформы CUDA [22]. Платформа CUDA разработана компанией Nvidia, которая позволяет выполнять отдельные части программы на GPU, произведенных этой же компанией. Так как в данной работе рассматриваются КОВР, которые чаще всего не имеют GPU, то использование фреймворка Darknet выглядит нецелесообразным. Помимо этого, по заверению самих разработчиков, они больше стремились оптимизировать свой фреймворк под работу с GPU, нежели с CPU. Данный тезис также подтверждается тестами, которые были проведены автором этой работы, что даже при сборки Darknet с библиотекой OpenMP, время работы при использовании только CPU оказывается больше, чем у альтернативных реализаций.

Структура разработанных моделей YOLO хранится в текстовом файле, формат которого заранее определен. Это позволяет реализовать модели, используя другие библиотеки. При изучении репозитория с реализацией метода YOLOv4, были найдены комментарии авторов, где утверждалось, что на данный момент самой лучшей реализацией их метода на CPU является реализация из библиотеки OpenCV [23]. Open Source Computer Vision Library (OpenCV) библиотека изначально разработанная компанией Intel для обработки изображений и решения различных задач компьютерного зрения. Использование библиотеки OpenCV дает следующие преимущества:

1) возможность загружать модели из разных форматов. На данный момент поддерживаются форматы всех популярных библиотек: PyTorch, Tensorflow, Caffe и Darknet. Также можно загружать модель из файлов

формата ONNX, данный формат призван унифицировать представление моделей между разными библиотеками для машинного обучения;

2) при выполнении вычислений на CPU, будут использоваться поддерживаемые процессором SIMD инструкции. То есть для процессоров архитектуры x86 это будет набор инструкций SSE и AVX, для процессоров архитектуры ARM соответственно NEON;
3) поддержка интеграции с библиотекой Tengine Lite. Данная библиотека разрабатывалась специально под КОВР и оптимизирует часто используемые операции (например, операция свертки) за счет использования SIMD инструкций, что приводит к уменьшению времени работы модели;
4) при наличии GPU можно полностью перенести на нее все вычисления, указав это при конфигурации. Также есть возможность указать фреймворк OpenCL, который позволяет задействовать CPU и GPU на гетерогенных платформах.

Учитывая все приведенные выше преимущества был сделан выбор именно в сторону библиотеки OpenCV. Исходный код детектора для распознавания объектов на изображениях на языке Python приведен в приложении А.

2.2.2 Планировщик

Алгоритм планирования

Обычно узлы кластера имеют одинаковые вычислительные ресурсы, поэтому должны быть нагружены одинаково. В нашем случае используется гомогенный кластер, в связи с этим был выбран алгоритм планирования Round-Robin. В данном алгоритме рабочие упорядочиваются в циклический список и после планировщик начинает проходить по этому списку, пока есть задачи на выполнение.

Алгоритм сохранения кадров

Поскольку время распознавания объектов на кадре в общем случае неопределенно, то могут возникать ситуации, когда последующий кадр будет обработан раньше предыдущего. Также нужно учесть, что в процессе обработки видео несколько рабочих могут выйти из строя и не завершить обработку отправленного им кадра. При возникновении таких ситуаций можно поступить двумя способами: отправить данный кадр другому рабочему или отбросить его. В нашем случае будем учитывать, что выход из строя рабочего это исключительная ситуация, которая происходит в редких случаях. А так как типичное видео имеет частоту кадров порядка 30 кадров в секунду и более, то потери единичных кадров не критичны, поэтому при отказе рабочего была выбрана стратегия отброса последнего отправленного ему кадра.

На рисунке 2.2 изображена блок-схема разработанного алгоритма, полная реализация на языке Python представлена в приложении Б. Рассмотрим алгоритм подробнее:

1) буфер полученных кадров *rcvd_buf* основан на структуре данных очередь с приоритетом. В качестве приоритета используется порядковый номер кадра, поэтому в голове очереди всегда находится самый ранний из полученных кадров. Размер очереди ограничен параметром *max_size*, который задается при старте планировщика. С учетом этого получаем, что операции вставки и удаления минимального элемента из очереди будут иметь асимптотическую сложность *O(log[max_size])*;
2) переменная *last_written* содержит в себе порядковый номер кадра, который был записан последним;
3) таймаут на получение кадра указывается при старте планировщика и будет зависеть от среднего времени обработки кадра одним рабочим;
4) планировщик старается сохранять кадры в правильном порядке, но если размер буфера *rcvd_buf* превышает максимальное значение, тогда следующим будет записан самый ранний кадр, который есть на данный

момент в буфере. Даже если позже от рабочего вернется недостающий кадр – он будет отброшен.

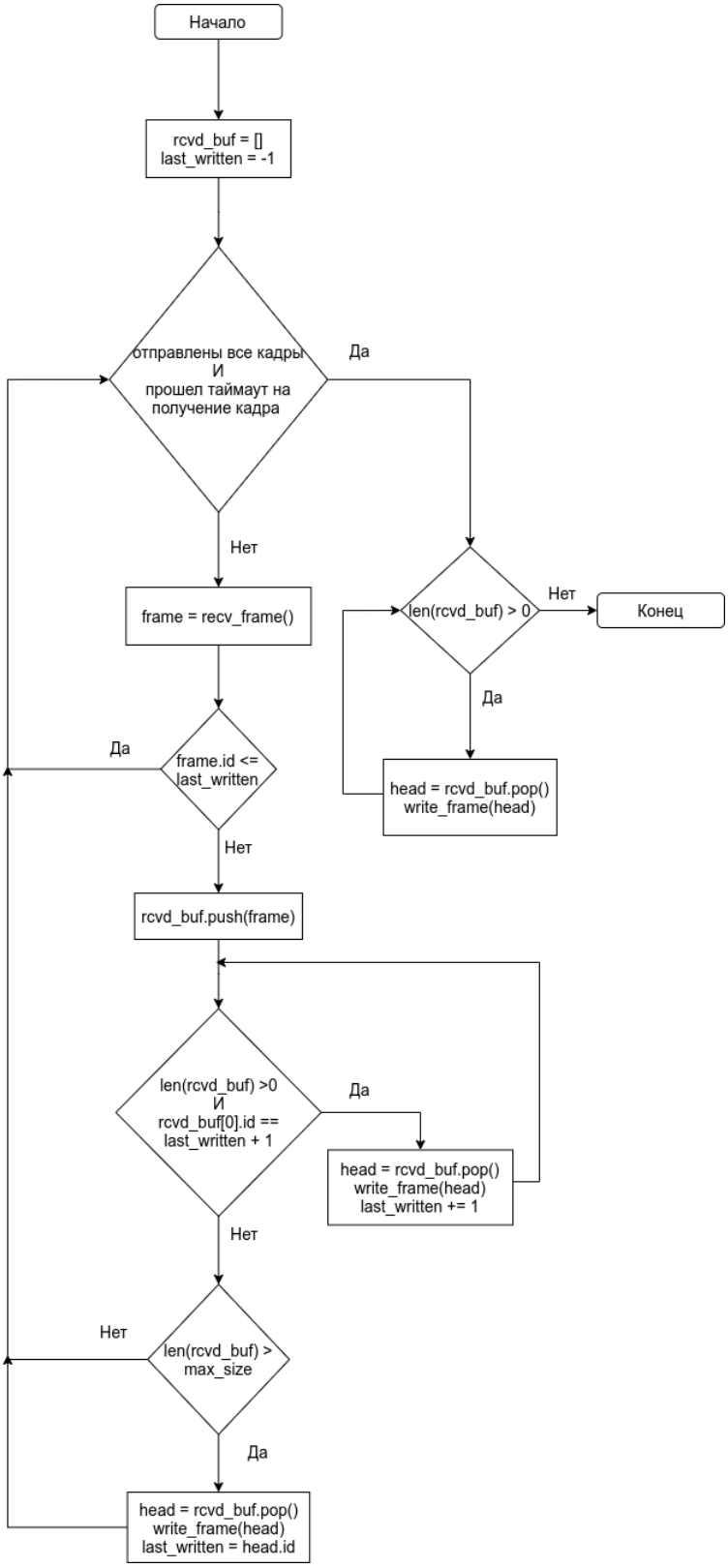

Рисунок 2.2 – Алгоритм сохранения кадров

2.3 Взаимодействие между компонентами

2.3.1 Транспорт

При выборе транспортного протокола главным требованием было возможность обмениваться атомарными сообщениями между узлами кластера. Это можно реализовать несколькими способами. Во-первых, можно взять протокол транспортного уровня (TCP, UDP) и поверх него реализовать протокол обмена сообщениями. Преимущество такого подхода, что мы можем гибче настраивать разработанный протокол под свои требования. Вполне очевидным недостатком данного подхода являются большие временные издержки необходимые для разработки, поэтому такой подход был отклонен. Во-вторых, можно использовать один из известных брокеров сообщений, которые уже имеют свой внутренний протокол. Недостатком такого подхода являются усложнение процесса развертывания приложения и необходимости дополнительных ресурсов для брокера сообщений. В-третьих, можно использовать уже готовую библиотеку для обмена сообщениями такую, как ZeroMQ и именно этот вариант был выбран.

ZeroMQ – это легковесная библиотека для обмена атомарными сообщениями, посредством использования предоставляемых сокетов [24]. Данная библиотека имеет более 10 типов сокетов, используя которые можно реализовывать разные топологии. Рассмотрим подробнее разработанную топологию на рисунке 2.3:

1) через *PUSH* сокет планировщика рабочим отправляются прочитанные кадры. При подключении нескольких клиентов к *PUSH* сокету сообщения им отправляются согласно алгоритму Round-Robin;
2) рабочий при старте начинает слушать сообщения со своего *PULL* сокета, который подключен к *PUSH* сокету планировщика;
3) результаты распознавания рабочий отправляет через свой *PUSH* сокет, который подключен к *PULL* сокету планировщика;

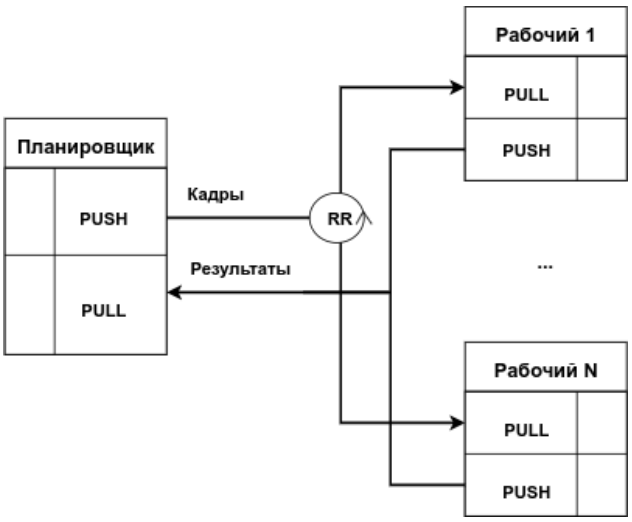

Рисунок 2.3 – Топология подключения планировщика и рабочих

4) планировщик слушает свой *PULL* сокет и при поступлении результата помещает его буфер, откуда после результаты обрабатываются согласно алгоритму сохранения кадров.

Также стоит упомянуть, что преимуществом разработанной топологии является возможность добавлять рабочих прямо в процессе обработки видео. Такое происходит из-за встроенного механизма обнаружения подключенных клиентов в ZeroMQ, то есть в протокол не нужно добавлять служебные сообщения для обнаружения новых рабочих.

2.3.2 Формат сообщений

Для взаимодействия между планировщиком и рабочими были определены два вида сообщений: запрос на распознавание и результат распознавания. Сам формат сообщений бинарный, порядок байт – big-endian.

На рисунке 2.4 представлена структура сообщения с запросом на распознавание. Поле *ID* хранит в себе порядковый номер отправляемого кадра, которое после добавляется в сообщение с результатом. В поле *Frame Data* содержится непосредственно передаваемый кадр с размерами $(Height \times Width \times Channels)$.

На рисунке 2.5 представлена структура сообщения с результатом распознавания. Как ранее упоминалось, поле *ID* содержит в себе порядковый

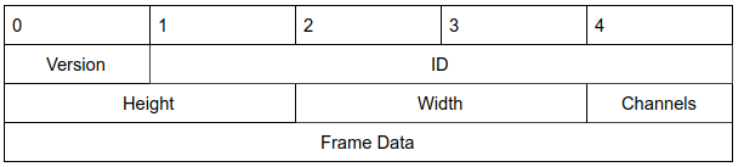

Рисунок 2.4 – Структура сообщения с запросом на распознавание

номер кадра. Поле *Detection Time* представляет собой количество секунд, потраченных на обработку данного кадра, это значение далее используется планировщиком для сбора статистики. Так как на одном кадре может быть более одного объекта, то поле *Length* хранит в себе количество распознанных объектов. *Detection Results* представляет собой массив результатов.

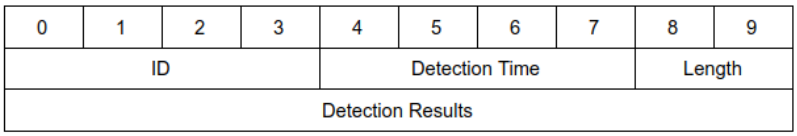

Рисунок 2.5 – Структура сообщения с результатом распознавания

На рисунке 2.6 представлена структура одного элемента из массива *Detection Results*. Координаты *(X1, Y1)* представляют собой координаты левого-верхнего угла ограничивающей рамки, а координаты *(X2, Y2)* – правого-нижнего угла. Значение *Score* является вероятностью того, что внутри ограничивающей рамки есть объект, и он относится к классу с идентификатором *Class ID*.

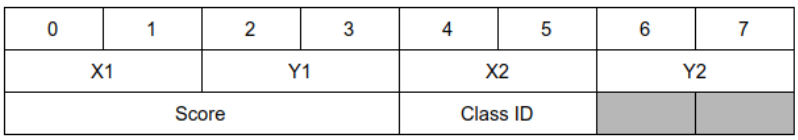

Рисунок 2.6 – Структура результата распознавания

2.4 Описание разработанного алгоритма

Поскольку основные задачи планировщика больше связаны с вводом-выводом, нежели с какими-то CPU интенсивными задачами, то было принято

решение при его реализации использовать пакет asyncio из стандартной библиотеки языка Python. На рисунке 2.7 представлена диаграмма последовательности обработки одного кадра. Планировщик при старте запускает три корутины *Reader*, *DetectionPoller* и *Writer*. Так как скорость чтения кадров будет явно больше скорости распознавания объектов, поэтому скорость чтения контролируется размером *ReadBuffer*, при заполнении которого корутина *Reader* будет заблокирована. Аналогичным образом устроен *WriteBuffer*, при заполнении которого корутина *DetectionPoller* заблокируется. Рассмотрим последовательность обработки кадра подробнее:

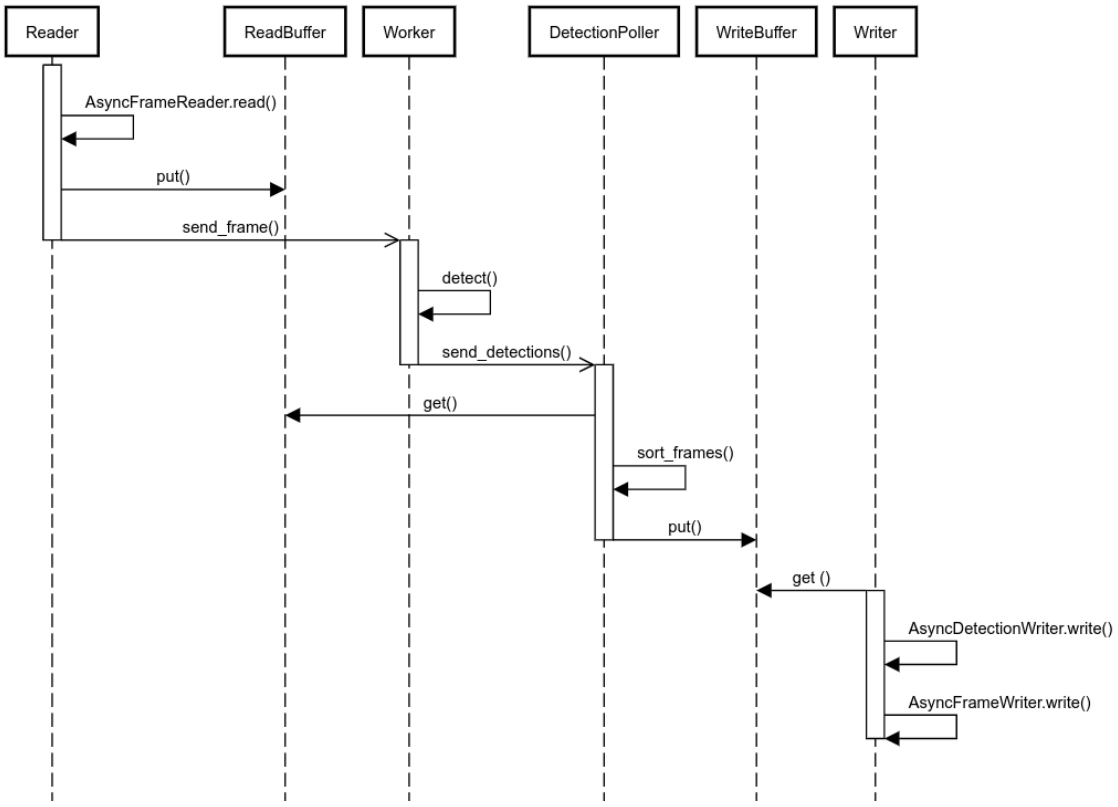

Рисунок 2.7 – Диаграмма последовательности обработки одного кадра

1) для разбиения входного видео на кадры также используется библиотека OpenCV. Поскольку данная библиотека не приспособлена для работы с asyncio, то был реализован класс *AsyncFrameReader*, который вызывает метод чтения кадра в другом потоке, тем самым не блокируя поток событийного цикла;

2) прочитанный кадр сохраняется в *ReadBuffer*, откуда позже будет извлечен корутиной *DetectionPoller*;
3) затем формируется сообщение с запросом на распознавание, и далее оно отправляется на обработку одному из рабочих. После отправки кадра *Reader* сразу переходит к чтению следующего кадра, не дожидаясь результатов распознавания;
4) получив кадр рабочий производит распознавание объектов и отправляет планировщику сообщение с результатами;
5) корутина *DetectionPoller* активируется в момент получения одного из результатов распознавания. После происходит упорядочивание уже полученных кадров *sort_frames*, согласно алгоритму сохранения кадров, описанному ранее. Затем кадры уже в правильном порядке записываются в *WriteBuffer*;
6) при получении кадра корутина *Writer* отображает на кадре ограничивающие рамки и название классов объектов, после кадр записывается на диск. Чтобы не блокировать поток событийного цикла также были реализованы два класса *AsyncDetectionWriter* и *AsyncFrameWriter*.

СПИСОК ЛИТЕРАТУРЫ

1. Inside the Amazon Warehouse Where Humans and Machines Become One [Электронный ресурс]. 2019. URL: https://www.wired.com/story/amazon-warehouse-robots/ (дата обращения: 22.04.2021).
2. What Edge Computing Means for Infrastructure and Operations Leaders [Электронный ресурс]. 2018. URL: https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders/ (дата обращения: 23.04.2021).
3. ImageNet Large Scale Visual Recognition Challenge [Электронный ресурс]. Официальная страница соревнований ILSVRC. 2021. URL: http://image-net.org/challenges/LSVRC/ (дата обращения: 13.04.2021).
4. ImageNet [Электронный ресурс]. Официальный сайт проекта ImageNet. 2021. URL: http://image-net.org/index (дата обращения: 13.04.2021).
5. Николенко С.И., Кадурин А.А., Архангельская Е.В. Глубокое обучение. Погружение в мир нейронных сетей. СПб: Питер, 2018. 480 с.
6. Krizhevsky A. One weird trick for parallelizing convolutional neural networks // arXiv:1404.5997. 2014.
7. Аггарвал Чару. Нейронные сети и глубокое обучение. Учебный курс. СПб: Диалектика, 2020. 752 с.
8. Krizhevsky A., Sutskever I., Hinton G. E. ImageNet Classification with Deep Convolutional Neural Networks // Advances in Neural Information Processing Systems / Ed. by F. Pereira, C. J. C. Burges, L. Bottou et al. Vol. 25. Curran Associates, Inc., 2012.

9. Deep Learning in Object Detection and Recognition / X. Jiang, A. Hadid, Y. Pang et al. Singapore: Springer, 2019. 240 p.

10. Rich feature hierarchies for accurate object detection and semantic segmentation / R. Girshick, J. Donahue, T. Darrell et al. // arXiv:1311.2524. 2014.

11. Girshick R. Fast R-CNN // arXiv:1504.08083. 2015.

12. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks / S. Ren, K. He, R. Girshick et al. // arXiv:1506.01497. 2016.

13. Bochkovskiy A., Wang C.-Y., Liao H.-Y. M. YOLOv4: Optimal Speed and Accuracy of Object Detection // arXiv:2004.10934. 2020.

14. You Only Look Once: Unified, Real-Time Object Detection / J. Redmon, S. Divvala, R. Girshick et al. // arXiv:1506.02640. 2016.

15. Redmon J., Farhadi A. YOLO9000: Better, Faster, Stronger // arXiv:1612.08242. 2016.

16. Redmon J., Farhadi A. YOLOv3: An Incremental Improvement // arXiv:1804.02767. 2018.

17. Wang C.-Y., Bochkovskiy A., Liao H.-Y. M. Scaled-YOLOv4: Scaling Cross Stage Partial Network // arXiv:2011.08036. 2021.

18. Apple unleashes M1 [Электронный ресурс]. Пресс-релиз компании Apple о процессоре M1. 2020. URL: https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/ (дата обращения: 09.05.2021).

19. Harris S. L., Harris D. M. Digital Design and Computer Architecture: ARM Edition. Waltham: Morgan Kaufmann, 2016. 584 p.

20. Arm, Cambridge, England. Arm Architecture Reference Manual. Armv8, for Armv8-A architecture profile. 2021.

21. Intel Corporation, Santa Clara, USA. Intel 64 and IA-32 Architectures Software Developer's Manual. 2021.

22. Darknet: Open Source Neural Networks in C [Электронный ресурс]. Официальный сайт фреймворка Darknet. 2021. URL: http://pjreddie.com/darknet/ (дата обращения: 04.05.2021).

23. Yolo v4, v3 and v2 for Windows and Linux [Электронный ресурс]. Репозиторий с реализацией метода YOLOv4. 2021. URL: https://github.com/AlexeyAB/darknet (дата обращения: 04.05.2021).

24. Zguide [Электронный ресурс]. Документация библиотеки ZeroMQ. 2021. URL: https://zguide.zeromq.org/docs (дата обращения: 06.05.2021).

ПРИЛОЖЕНИЕ А

Листинг А.1 – Детектор для распознавания объектов на изображениях

```
import dataclasses
import re
from typing import List, Tuple

import cv2
import numpy as np


@dataclasses.dataclass
class DetectionResult:
  x1: int
  y1: int
  x2: int
  y2: int
  score: float
  clazz: int


class DarknetObjectDetector:
  _SCALE_FACTOR = 0.00392
  _RGB_MEAN = (0, 0, 0)

  def __init__(self, cfg: str, weights: str, target: int, warm_up
     : bool = True):
    net = cv2.dnn.readNetFromDarknet(cfg, weights)
    net.setPreferableTarget(target)

    self._net = net
    self._size = self._parse_net_size(cfg)
    self._out_names = net.getUnconnectedOutLayersNames()

```

```
31      if warm_up:
32        shape = self._size + (3,)
33        dummy_img = np.random.randint(0, 255, shape, dtype=np.uint8
            )
34        self.detect(dummy_img)
35
36    def detect(
37      self, img: np.ndarray, conf_threshold: float = 0.5,
           nms_threshold: float = 0.4
38    ) -> List["DetectionResult"]:
39      blob = cv2.dnn.blobFromImage(
40        img,
41        self._SCALE_FACTOR,
42        self._size,
43        self._RGB_MEAN,
44        swapRB=True,
45        crop=False,
46      )
47      self._net.setInput(blob)
48      outs = self._net.forward(self._out_names)
49
50      layers = self._net.getLayerNames()
51      last_layer_id = self._net.getLayerId(layers[-1])
52      last_layer = self._net.getLayer(last_layer_id)
53
54      boxes = []
55      classes = []
56      confidences = []
57      img_h, img_w = img.shape[0], img.shape[1]
58      if last_layer.type == "Region":
59        for out in outs:
60          for detection in out:
61            scores = detection[5:]
62            clazz = np.argmax(scores)
63            confidence = scores[clazz]
```

```
64
65          if confidence > conf_threshold:
66            center_x, center_y = detection[0] * img_w, detection
                 [1] * img_h
67            w, h = detection[2] * img_w, detection[3] * img_h
68            left, top = center_x - w / 2, center_y - h / 2
69
70            classes.append(clazz)
71            # important: confidence must be float
72            confidences.append(float(confidence))
73            # important: boxes must be ints
74            boxes.append(list(map(int, [left, top, w, h])))
75      else:
76        raise ValueError(f"unknown last layer type: {last_layer.
             type}")
77
78      indices = [i[0] for i in cv2.dnn.NMSBoxes(boxes, confidences,
          conf_threshold, nms_threshold)]
79      results = []
80      for i in indices:
81        box = boxes[i]
82        left, top = box[0], box[1]
83        w, h = box[2], box[3]
84
85        res = DetectionResult(
86          x1=left,
87          y1=top,
88          x2=left + w,
89          y2=top + h,
90          score=confidences[i],
91          clazz=classes[i],
92        )
93        results.append(res)
94
95      return results
```

```
96
97    @staticmethod
98    def _parse_net_size(cfg_path: str) -> Tuple[int, int]:
99      with open(cfg_path, "r") as cfg:
100       raw_sections = []
101
102       section = {}
103       header_re = re.compile(r"^\[(?P<name>\w+)\]$")
104       for line in cfg:
105         if line.startswith("#") or line.isspace():
106           continue
107
108         match = header_re.match(line)
109         if match:
110           if section:
111             raw_sections.append(section)
112           section = {"type": match.group("name")}
113         else:
114           key, val = [s.strip() for s in line.split("=")]
115           if key in section:
116             raise ValueError(
117               f"duplicate␣section␣key:␣section={section['type']},
                    ␣key={key}"
118             )
119           section[key] = val
120       raw_sections.append(section)
121
122     for raw in raw_sections:
123       type_ = raw["type"]
124       if type_ == "net":
125         width = raw.get("width")
126         if width is None:
127           raise ValueError(f"missing␣width␣in␣net␣section:␣{
                cfg_path}")
128
```

```
129         height = raw.get("height")
130         if height is None:
131           raise ValueError(f"missing height in net section: {
                cfg_path}")
132 
133         return int(width), int(height)
134     raise ValueError(f"missing net section: {cfg_path}")
```

ПРИЛОЖЕНИЕ Б

Листинг Б.1 – Модуль для сохранения кадров в правильном порядке

```
import asyncio
import heapq
import logging
import time
from typing import List

import zmq

import proto
import stats
import video


class DetectionCollector:
  def __init__(
    self,
    sock: zmq.Socket,
    read_buf: "AsyncDict",
    read_all: asyncio.Event,
    recv_buf_size: int,
    write_buf_size: int,
    frame_timeout: int,
    classes: List[str],
    write_label: bool,
    frame_writer: video.AsyncFrameWriter,
    stats_ctl: stats.FrameEventsCollector,
  ):
    self._sock = sock
    self._read_buf = read_buf
    self._read_all = read_all
    self._recv_max_size = recv_buf_size
```

```
32      self._frame_timeout = frame_timeout
33
34      self._detection_writer = video.AsyncDetectionWriter(classes,
            write_label)
35      self._frame_writer = frame_writer
36      self._stats_ctl = stats_ctl
37
38      self._dropped = 0
39      self._write_queue = asyncio.Queue(maxsize=write_buf_size)
40      self._stopping = asyncio.Event()
41      self._log = logging.getLogger(__name__)
42
43    @property
44    def dropped_frames(self):
45      return self._dropped
46
47    def start(self) -> asyncio.Task:
48      write_task = asyncio.create_task(self._write_detections(),
            name="write_task")
49      return asyncio.create_task(self._poll_detections(write_task),
            name="poll_task")
50
51    async def _poll_detections(self, write_task: asyncio.Task):
52      name = asyncio.current_task().get_name()
53      self._log.info(f"{name}: started")
54      start_time = time.perf_counter()
55
56      pq = []
57      frames = {}
58
59      async def write_head():
60        head = heapq.heappop(pq)
61        frame = frames.pop(head.id)
62        self._log.debug(f"{name}: writing frame: id={head.id}")
63        await self._write_queue.put((frame, head.detections))
```

```
64        return head.id
65
66      async def check_drops(cur, prev):
67        if cur != prev + 1:
68          dropped = [i for i in range(prev + 1, cur)]
69          for i in dropped:
70            await self._read_buf.pop(i)
71
72          self._log.warning(f"{name}: dropping frame: ids={dropped}
              ")
73          self._dropped += len(dropped)
74
75      last_written = -1
76      while True:
77        try:
78          raw_resp = await asyncio.wait_for(
79            self._sock.recv(), self._frame_timeout
80          )
81          resp = proto.parse_detect_response(raw_resp)
82          self._log.debug(f"{name}: received frame: id={resp.id}")
83          await self._stats_ctl.send_detect_duration(resp.
              elapsed_sec)
84
85          if resp.id <= last_written:
86            # just drop received frame
87            continue
88
89          heapq.heappush(pq, resp)
90          frames[resp.id] = await self._read_buf.pop(resp.id)
91
92          while pq and pq[0].id == last_written + 1:
93            await write_head()
94            last_written += 1
95
96          if len(pq) > self._recv_max_size:
```

```
97            # write nearest frame which we already received
98            id_ = await write_head()
99            await check_drops(id_, last_written)
100           last_written = id_
101       except asyncio.TimeoutError:
102         if not self._read_all.is_set():
103           self._log.warning(f"{name}: got frame timeout")
104           continue
105
106         while pq:
107           id_ = await write_head()
108           await check_drops(id_, last_written)
109           last_written = id_
110
111         elapsed = time.perf_counter() - start_time
112         elapsed = round(elapsed - self._frame_timeout, 2)
113         self._log.info(
114           f"{name}: process all frames, stopping: processing time
                ={elapsed}s"
115         )
116
117         self._stopping.set()
118         await write_task
119         break
120       except asyncio.CancelledError:
121         self._log.info(f"{name}: got cancel signal, stopping")
122         break
123       except:
124         self._log.error(f"{name}: unhandled exception", exc_info=
              True)
125
126     self._log.info(f"{name}: finished")
127
128   async def _write_detections(self):
129     name = asyncio.current_task().get_name()
```

```
130     self._log.info(f"{name}:␣started")
131
132     async def write(res):
133       frame, detections = res
134       now = time.perf_counter()
135       for d in detections:
136         await self._detection_writer.write(frame, d)
137       await self._stats_ctl.send_write_detections_duration(
138         time.perf_counter() - now
139       )
140
141       now = time.perf_counter()
142       await self._frame_writer.write(frame)
143       await self._stats_ctl.send_encode_duration(time.
            perf_counter() - now)
144
145     async def write_until_empty(q):
146       while not q.empty():
147         res = await q.get()
148         await write(res)
149
150     wait_task = asyncio.create_task(self._stopping.wait())
151     async with self._frame_writer:
152       while True:
153         try:
154           write_task = asyncio.create_task(self._write_queue.get
                ())
155           tasks = {wait_task, write_task}
156
157           done, _ = await asyncio.wait(
158             tasks, return_when=asyncio.FIRST_COMPLETED
159           )
160           if write_task in done:
161             res = write_task.result()
162             await write(res)
```

```
163             elif wait_task in done:
164               self._log.info(f"{name}: got stopping signal")
165               await write_until_empty(self._write_queue)
166               break
167             else:
168               raise ValueError(f"unknown done tasks: {done}")
169           except asyncio.CancelledError:
170             self._log.info(
171               f"{name}: got cancel signal, writing all buffered
                      detections"
172             )
173             await write_until_empty(self._write_queue)
174             break
175           except:
176             self._log.error(f"{name}: unhandled exception",
                     exc_info=True)
177
178       self._log.info(f"{name}: finished")
```