МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

РАБОТА ДОПУЩЕНА К ЗАЩИТЕ

Руководитель
программы _____ Ш.Г. Магомедов

«30» мая 2025 г.

ИТОГОВАЯ АТТЕСТАЦИОННАЯ РАБОТА

по дополнительной программе профессиональной переподготовки
«Программные средства решения прикладных задач искусственного интеллекта»

На тему: «Модель распознавания вариант 510»

Обучающийся _____ Мусатов Иван Алексеевич
Подпись *Фамилия, имя, отчество*

группа ИКБО-11-23

Руководитель работы _____ Ш.Г. Магомедов
подпись

Москва 2025 г.

ФИО автора: Мусатов Иван Алексеевич

Учебная группа: ИКБО-11-23

Фактическая тема ВКР: «Разработка модели мультиклассовой детекции признаков возгорания на изображении»

АННОТАЦИЯ

В данной работе исследуются методы компьютерного зрения, направленные на автоматическое выявление признаков возгорания на изображении. Ставится задача автоматического обнаружения признаков возгорания на изображении с применением методов глубокого обучения.

Сперва проводится анализ предметной области, включающий выявление ее специфик и возможных аномалий данных. Затем производится поиск и выбор данных для обучения модели с последующим статистическим анализом и исправлением недостатков набора данных.

Задача машинного обучения ставится как многоклассовая детекция. Анализируются возможные архитектуры решения. На основе подробного обзора выбранной архитектуры производится программная реализация модели нейронной сети. Проведено обучение модели с последующей оценкой качества. Полученные результаты подтверждают среднюю обобщающую способность созданной модели.

Результаты исследования могут быть использованы в дальнейших работах, направленных на улучшение точности и надежности систем раннего обнаружения возгораний на основе методов компьютерного зрения.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Несмотря на активное развитие и внедрение противопожарных технологий, пожары остаются одной из существенных угроз безопасности людей и инфраструктуры. Согласно статистике [1], показатель смертности граждан в результате пожаров на территории России выше, чем в других развитых странах. А размеры экономического ущерба оцениваются в миллиардах рублей.

В связи с этим актуальной задачей является улучшение и модернизация противопожарных технологий. Наиболее востребованными являются превентивные меры, либо же методы, позволяющие обнаруживать возгорания на их ранней стадии. Физические решения в виде датчиков тепла и дыма не предоставляют такой возможности. Наиболее надежное решение – человеческий мониторинг – не может быть использовано массово. В таком случае применима автоматизация в виде использования систем видеонаблюдения с алгоритмами компьютерного зрения.

Таким образом, объектом данного исследования являются методы компьютерного зрения, направленные на автоматическое выявление признаков возгорания на изображении. Предметом исследования является архитектура модели детекции объектов пожара, а также особенности данных в предметной области.

Целью работы является исследование и разработка нейронной сети для решения задачи детекции признаков возгорания на изображении.

В процессе выполнения работы будут решаться следующие задачи:

- анализ предметной области;
- обзор существующих решений проблемы;
- сбор данных для обучения и их характеристика;

- постановка задачи обучения;
- исследование архитектуры модели и ее разработка;
- обучение модели;
- анализ результатов;

Работа состоит из двух частей: теоретической и практической. В теоретической части проводится полноценный анализ предметной области и подготавливается набор данных. В практической части ставится задача обучения, исследуется архитектура и производится разработка модели с последующим анализом результата. Ожидаемым результатом работы является модель компьютерного зрения, выполняющая поставленную задачу детекции.

1 ТЕОРЕТИЧЕСКИЙ АНАЛИЗ

1.1 Анализ предметной области

Тема распознавания пожаров и возгораний сопряжена с большим количеством потенциальных аномалий. Даже при ручном (человеческом) мониторинге не исключены ошибки неправильного восприятия объектов.

Огонь можно охарактеризовать, в первую очередь, как яркое световое явление с динамичной природой. Тем не менее, в природе (как естественной, так и искусственной) существует немало объектов с похожей структурой. Например, движущиеся источники света (скажем, фары), мерцающие лампы и фонари, недаром называющиеся «огнями». Блики на водной глади, металлах и стекле также при определенных условиях могут быть восприняты за возгорание.

Дым – верное следствие возгорания – также не уникален по своей природе. Его рассеянность и сероватый оттенок могут быть спутаны с туманом или выхлопными газами. Тем не менее, динамичность дыма позволяет отделять его от остальных привычных газообразных структур.

Но, пожалуй, более существенной и трудно устранимой проблемой является оценка вредоносности огня. Имеется в виду отличие огня в камине или мангале от горящего кресла рядом с камином или горящего куста рядом с мангалом. Человек-наблюдатель в большинстве случаев легко может определить, несет данный огонь пользу или вред. Но как это можно формализовать? Какими признаками можно разделить две ситуации? Возможно, различие как раз и достигается за счет окружающего контекста. Как в рассмотренных примерах – кресла и камина. Либо же подсказкой может стать динамика увеличения возгорания. В случае с камином огонь сохраняет свое положение. В случае же с возгоранием, область поражения будет распространяться, тем самым и обеспечивая бесконтрольность ситуации.

Проведенное рассмотрение позволяет нам заключить, что для создания обобщающей и корректной модели необходимо включить в набор данных помимо примеров изображений с возгоранием и дымом также и «аномальные» изображения с источниками света, бликами, туманом и «контролируемым» огнем.

1.2 Обзор существующих решений

Попытки определения характеристик огня на изображении осуществлялись и до широкого распространения сверточных нейронных сетей. Тогда методы компьютерного зрения заключались в разработке цветовой модели классификации пикселей пламени. В частности, в работе [2] была предложена цветовая модель, работающая в пространстве «YCbCr». Модель основывается на квантовании цветовых плоскостей на дискретные уровни. Помимо этого, осуществлялся анализ движения, т.е. изучение динамики огня. Выявлялись основные характеристики и проводилось моделирование мерцания пламени. Такие методы по-своему хороши и основаны на серьезной аналитической базе, но, все-таки, уязвимы к аномалиям данных по типу рассматривавшихся в предыдущем пункте настоящей работы. Так, предлагаемые цветовые решения показывали 31,5% ложных срабатываний.

Появление сверточных нейронных сетей позволило делегировать сложную задачу определения характеристик пламени глубокому обучению. Были адаптированы популярные архитектуры детекторов: Faster R-CNN и YOLO. В некоторых случаях используется оптимизация в виде подбора гиперпараметров обучения генетическими алгоритмами. Одностадийные модели типа YOLO работают довольно быстро, что позволяет использовать их в режиме реального времени.

Наиболее новыми и перспективными решениями на данный момент являются модели на основе архитектуры трансформеров. В частности, в

работе [3] используется Vision Transformer (RepViT) для улучшения глобального изучения признаков и снижения вычислительных затрат. Используется динамическая свёртка со змеевидными контурами (DSConv), позволяющая улавливать мелкие детали пламени и дыма, особенно на сложных изогнутых границах объектива. Механизм внимания позволяет более эффективно извлекать признаки из сложного фона. В работе утверждается, что модель трансформера по своим результатам обходит стандартные сверточные модели.

1.3 Поиск и сбор данных

Существует некоторое количество платформ и ресурсов, предоставляющих возможность поиска наборов данных. Воспользуемся наиболее популярными из них:

- Kaggle (платформа, организующая соревнования по ML и сбор датасетов);
- Roboflow (платформа для создания и развертывания моделей компьютерного зрения);
- Dataset Search (поисковый сервис для открытых наборов данных).

В результате поиска наборов данных по тегам fire, smoke, detection были выделены несколько релевантных экземпляров. Кратко опишем их преимущества и недостатки.

Датасеты с платформы Kaggle:

- Smoke Detection Dataset – набор данных содержит изображения дыма, но не огня;
- Fire and Smoke Dataset – маленький объем данных (200 изображений в открытом доступе). Для обучения не хватит;

- Forest_Fire_Smoke_and_Non_Fire_Image_Dataset – специализация на лесных пожарах. Локальные возгорания не предоставлены;

Поиск посредством Dataset Research приводил либо к лицензионным наборам данных, либо к наборам, объема которых для обучения недостаточно. Тем не менее, были даны ссылки на датасеты с платформы Roboflow.

Поиск данных на платформе Roboflow потребовал регистрации, но зато привел к продуктивным результатам. Были выделены два релевантных набора:

- Fire-Smoke-Detection-Yolov11 – набор с разнообразием локаций возгорания. Но отсутствует деление на огонь и дым. Объем – 4 тысячи изображений;

- FireSmokeDataset – также содержит разнообразные локации возгорания. Имеются «аномальные» данные, ведется классификация bbox по трем классам «огня», «дыма» и «прочего». Объем – 20 тысяч изображений;

В результате разбора недостатков всех вышеназванных наборов было принято решение об использовании датасета FireSmokeDataset с проведением дополнительной предобработки.

1.4 Характеристика данных

Датасет FireSmokeDataset представляет собой директорию с тремя разделенными выборками: train, test и validation. В каждой из папок с выборкой располагаются изображения в формате JPG, а также аннотационный файл.

Аннотации оформлены в нотации COCO (common objects in context). Такой формат предполагает структурирование файла на пять частей: описание датасета, список изображений (названий файлов), список категорий (классов), аннотации для задач обнаружения, аннотации для задач сегментации [4]. Структура позволяет организовывать данные для многоклассовой детекции. Из особенностей формата стоит также отметить способ указания рамок

объектов (bounding box). Они производятся в форме "bbox": [x, y, width, height], где x и y – минимальные координаты прямоугольника по осям x и y соответственно.

Проанализируем содержание датасета в количественной форме. Для этого напишем небольшой скрипт, собирающий статистику по данным (см. Приложение А).

Посмотрим на частоту сочетаемости различных классов друг с другом. Для этого визуализируем следующую pie-chart диаграмму (см. рис. 1.1).

Рисунок 1.1 – Диаграмма частоты сочетания классов на изображении

На основе диаграммы сочетания классов можем сделать вывод, что основную часть набора данных составляют изображения, содержащие как возгорание, так и задымление, т.е. соответствуют предметной области. Вместе с тем также отметим, что изображений, на которых нет ни возгораний, ни задымлений крайне мало, что говорит о потенциальной проблеме нехватки «отрицательных» данных. Количественно процент «отрицательных» данных в датасете можно оценить как отношение числа аннотаций «other» к общему числу аннотаций. Что составляет 3796 / 30297 ≈ 0,125 = 12,5%.

Для решения этой проблемы внедрим в наш набор данных изображения, на которых нет целевых объектов детекции, т.е. огня и дыма. Структура COCO аннотаций позволяет легко внедрить такие изображения, просто добавив названия файлов в раздел images.

Предметная область определяет диапазон допустимых «отрицательных» значений. Это должны быть преимущественно изображения помещений, пространств рядом со зданиями, либо же съемок камер. При этом изображения не должны содержать возгораний. Для этих целей был найден отличный датасет – Indoor Scenes от MIT, содержащий изображения помещений разного рода. Была выделена выборка в 5000 изображений. Такое количество взято в целях балансировки предполагаемого качества распознавания будущей модели. Объем полезных данных из нашего датасета составляет 21 тысячу изображений. А «отрицательных» примеров будет 5 / 26 ≈ 0,19 = 19%. Такой сбалансированный объем позволит значительно снизить процент ложных срабатываний, но при этом не помешает модели выполнять свою целевую задачу.

Приведем статистику нашего обновленного датасета (см. таблицу 1).

Таблица 1 – Статистика объемов расширенных данных по выборкам

Характеристика	**train**	**test**	**validation**
Кол-во изображений	16398	2749	7092
Аннотаций Smoke	12348	2466	4869
Аннотаций Fire	14153	2161	5986
Аннотаций Other	3796	804	2135
Аннотаций всего	30297	5431	12990
Изображений без аннотаций	4027	558	1206

Датасет имеет значительный объем данных, а именно изображений. Задача чтения и загрузки изображений в оперативную память не самая быстрая, поэтому она может очень сильно сказаться на времени обучения.

Предвкушая эту проблему, сериализуем наши выборки в формат TFRecord. Это специальный бинарный формат экосистемы Tensorflow, который обеспечивает быстрое чтение данных для обучения модели [5]. Для сериализации напишем специальный скрипт, который будет читать изображения и конвертировать их в байты памяти (см. рис. 1.2).

```python
import tensorflow as tf
import os
from tqdm import tqdm

def create_tfrecord(output_path, image_dir, images_dict, anns_dict, max_boxes, image_size=640):
    with tf.io.TFRecordWriter(output_path) as writer:
        for img_id in tqdm(images_dict):
            img_info = images_dict[img_id]
            img_path = os.path.join(image_dir, img_info["file_name"])
            image_raw = tf.io.read_file(img_path)
            image = tf.image.decode_jpeg(image_raw, channels=3)
            image = tf.image.resize(image, (image_size, image_size))
            image = tf.cast(image, tf.uint8).numpy()
            bboxes = []
            class_ids = []
            for ann in anns_dict.get(img_id, []):
                x, y, w, h = ann["bbox"]
                x1 = x / image_size
                y1 = y / image_size
                x2 = (x + w) / image_size
                y2 = (y + h) / image_size
                bboxes.extend([y1, x1, y2, x2])
                class_ids.append(ann["category_id"])

            num_boxes = len(class_ids)
            if num_boxes > max_boxes:
                bboxes = bboxes[:max_boxes * 4]
                class_ids = class_ids[:max_boxes]

            pad_bboxes = [0.0] * ((max_boxes - num_boxes) * 4)
            pad_class_ids = [0] * (max_boxes - num_boxes)
            feature = {
                "image": tf.train.Feature(bytes_list=tf.train.BytesList(value=[tf.io.encode_jpeg(image).numpy()])),
                "bboxes": tf.train.Feature(float_list=tf.train.FloatList(value=bboxes + pad_bboxes)),
                "labels": tf.train.Feature(int64_list=tf.train.Int64List(value=class_ids + pad_class_ids))
            }
            example = tf.train.Example(features=tf.train.Features(feature=feature))
            writer.write(example.SerializeToString())
```

Рисунок 1.2 – Скрипт сериализации данных в TFRecord

Для каждой из трех выборок (train, valid, test) прочитаем их аннотации, подгрузим изображения по названию файлов, а также соответствующие им bbox и метки классов. Процесс сериализации представлен на рисунке 1.3.

```
for sample in ['train', 'valid', 'test']:
    sample_json = f"Fire-smoke-dataset/{sample}/_annotations.coco.json"
    sample_dir = f"Fire-smoke-dataset/{sample}/"
    output_tfrecord = f"{sample}.tfrecord"

    images_dict, anns_dict = load_coco_annotations(sample_json)
    create_tfrecord(output_tfrecord, sample_dir, images_dict, anns_dict, max_boxes=14)
```

```
[4]  ✓ 3m 46.3s

...  100%|██████████| 16398/16398 [02:31<00:00, 108.00it/s]
     100%|██████████| 7092/7092 [00:46<00:00, 152.25it/s]
     100%|██████████| 2749/2749 [00:27<00:00, 101.38it/s]
```

Рисунок 1.3 – Процесс сериализации наборов данных в формат TFRecord

Теперь у нас имеется три сериализованных файла с выборками датасета (см. рис. 1.4), которые в дальнейшем будет несложно подгрузить при помощи встроенной в Tensorflow функции tf.data.TFRecordDataset.

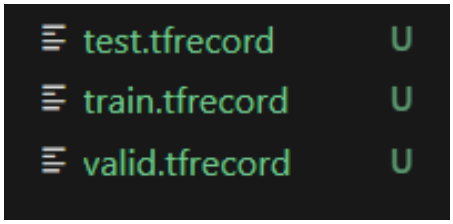

Рисунок 1.4 – Сериализованные в TFRrecord выборки датасета

2 РАЗРАБОТКА И ОБУЧЕНИЕ МОДЕЛИ

2.1 Постановка задачи машинного обучения

Так как рассматриваемая предметная область помимо определения факта возгорания запрашивает указание месторасположения на изображении, задача анализа данных формулируется как многоклассовая детекция объектов (multiclass object detection). Это означает, что модель должна научиться локализовать объекты возгорания/задымления при помощи ограничивающего прямоугольника – bbox, а также классифицировать их.

На языке данных задачу можно описать следующим образом:

Вход модели – изображение фиксированного размера (скажем, 640x640).

Выход модели – список найденных объектов, каждый из которых представлен координатами bbox ([x, y, width, height]) и вероятностью принадлежности к одному из классов fire, smoke, other (учитывая специфику датасета, класс other будет означать детекцию объекта, похожего на вредоносный огонь, но таковым не являющийся).

2.2 Выбор и обоснование архитектуры

На основании анализа предметной области и проведенной предобработки датасета можно выделить несколько возможных решений. Для задач реального развертывания больше всего подходят one-stage (одностадийные) детекторы. Рассмотрим две наиболее подходящих архитектуры.

Архитектура YOLO (You Only Look Once) известна своей высокой производительностью, компактностью и пригодностью для решения задач в режиме реального времени, что поощряет ее развертывание на реальном оборудовании. Тем не менее, она чувствительна к несбалансированности классов и детекции малых объектов. В нашей задаче часты ситуации, где огонь

занимает незначительную часть изображения, поэтому этот недостаток довольно критичен.

Архитектура RetinaNet традиционно устойчива к классовому дисбалансу данных (благодаря использованию функции потерь Focal Loss). Это способствует различению целевых объектов детекции и фона, что очень важно для снижения ложных срабатываний. Благодаря использованию FPN (Feature Pyramid Network) RetinaNet может эффективно обрабатывать объекты разного масштаба, в том числе и малые.

В отличие от моделей YOLO, RetinaNet гораздо реже используется в прикладных проектах по обнаружению возгораний, особенно в контексте многоклассовой детекции. Это оставляет пространство для исследований и возможность получения новых результатов в данной работе.

Стоит отметить, что RetinaNet легко масштабируется и может быть реализована при помощи библиотеки Keras с использованием функционального API, что предоставляет гибкость в построении и модификации модели (а также соответствует предпочтениям автора работы).

2.3 Исследование архитектуры модели

Перед тем, как приступить к программной реализации, подробно рассмотрим все составные компоненты архитектуры RetinaNet. Основными частями являются: backbone, FPN, Classification Subnet, Regression Subnet.

Backbone – это основная (базовая) сеть, которая служит для извлечения признаков из входного изображения. Обычно используется уже готовая глубокая сверточная нейросеть. На выходе backbone формируются активации с разными уровнями абстракции, обозначаемые как C3, C4, C5. Эти карты признаков используются для дальнейшей построения иерархии признаков.

Следующим уровнем архитектуры является FPN (Feature Pyramid Network). Это CNN, построенная в виде пирамиды и служащая для

объединения достоинств карт признаков нижних и верхних уровней сети. От нижних уровней она берет высокое разрешение изображения, а от верхних – высокую семантическую, обобщающую способность. FPN состоит из трех основных частей: восходящего пути (bottom-up), нисходящего пути (top-down) и боковых соединений (lateral connections). Восходящей сетью является уже рассмотренная backbone сеть. Она постепенно выделяет признаки из входного изображения, уменьшая разрешение. Нисходящий же путь также имеет вид пирамиды, каждые следующие нижние слои которой увеличивают карты признаков вдвое методом ближайшего соседа. Боковые соединения обеспечивают поэлементное сложение соответствующих слоев bottom-up и top-down пирамид (см. рис. 2.1). [6]

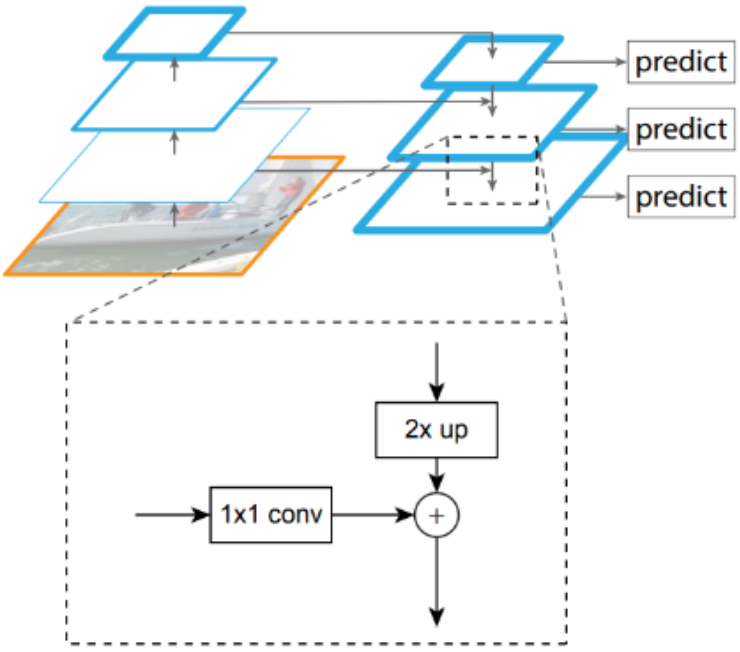

Рисунок 2.1 – Схема пирамиды признаков (FPN)

Для решения задачи многоклассовой детекции требуется решить две подзадачи: классификацию для класса объекта и регрессию для определения координат его рамок. Для этого в RetinaNet используется две подсети: классификации и регрессии.

До последнего слоя данные подсети работают по одному принципу. Каждая из них состоит из четырех слоев сверточных сетей. В каждом слое формируется 256 карт признаков. На пятом же слое количество признаков меняется: в классификационной сети K*A карт признаков, а в регрессионной – 4*A (где A – количество якорных рамок, а K – количество классов объектов). В шестом слое каждая карта признаков преобразуется в набор векторов. Регрессионная модель на выходе имеет для каждой якорной рамки вектор из четырех значений, указывающих смещение целевой рамки (ground-truth box) относительно якорной (anchor box). Классификационная модель на выходе имеет one-hot вектор длиной K, в котором индекс со значением 1 соответствует номеру определенного класса. [6]

Общая схема архитектуры RetinaNet со всеми указанными компонентами представлена на рис. 2.2.

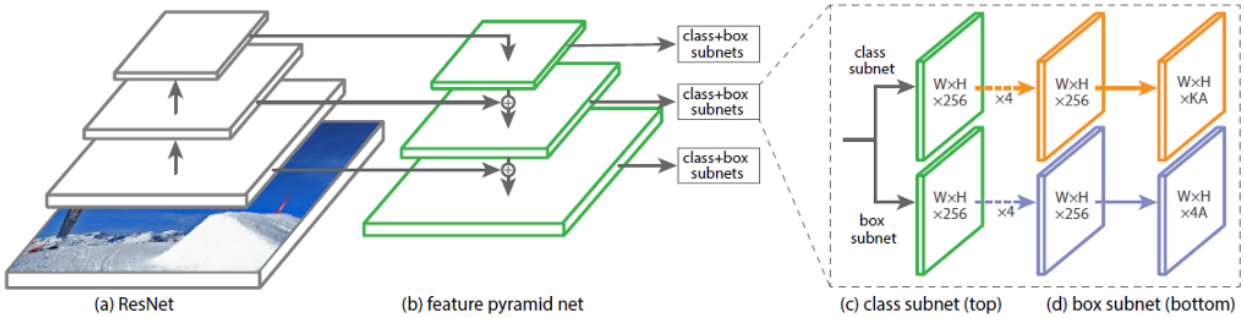

Рисунок 2.2 – Общая схема архитектуры RetinaNet

Потери в RetinaNet являются составными: ошибка регрессии (локализации) и ошибка классификации.

Каждой целевой рамке назначается якорная рамка. Можно обозначить такие пары как: $(A_i, G_i), i = \overline{1..N}$, где A – якорь, G – целевая рамка, а N – число сопоставленных пар. Для каждого якоря регрессионная сеть предсказывает четыре числа, которые можно обозначить как: $P_i = (P_{ix}, P_{iy}, P_{iw}, P_{ih})$. Эти значения показывают предсказанную разницу между координатами центров рамок и между их шириной и высотой. Для каждой целевой рамки вычисляется T_i как разница между якорной и целевой рамкой.

В результате, потеря регрессии представляется формулой 1:

$$RLoss = \sum_{j\in\{x,y,w,h\}} smoothL1(P_{ij} - T_{ij}) \quad (1)$$

Где $smoothL1(x)$ определяется формулой 2:

$$smooth_{L1}(x) = \begin{cases} 0.5x^2, & |x| < 0 \\ |x| - 0.5, & |x| \geq 0 \end{cases} \quad (2)$$

Потери задачи классификации определяются с помощью функции Focal Loss. По сути, это усовершенствованная функция кросс-энтропии. Функция зависит от следующих значений: p_{t_i}- вероятность правильного класса для i-го примера, α_{t_i} – вес класса для i-го примера, а N – число примеров в батче. Гамма – это параметр фокуса, решающий проблему несбалансированности классов. Вид функции потерь классификации представлен формулой 3. [7]

$$CLoss = -\frac{1}{N}\sum_{i=1}^{N} \alpha_{t_i}(1 - p_{t_i})^{\gamma} \log\left(p_{t_i}\right) \quad (3)$$

Общая функция потерь может быть записана как (см. формулу 4). Лямбда – это гиперпараметр, контролирующий баланс между потерями.

$$Loss = \lambda \cdot RLoss + CLoss \quad (4)$$

2.4 Разработка модели

Приступим к программной реализации архитектуры RetinaNet. Ввиду объемности реализации полный код приведем в приложении Б. А в данном разделе рассмотрим отдельные важные моменты.

В качестве базовой сети (backbone) будем использовать уже готовую сверточную сеть ResNet50. Архитектура ResNet удобна тем, что обеспечивает хорошую глубину сети при сравнительно небольшом числе параметров за счет использования остаточных связей (residual connections). В качестве

абстракций C3, C4, C5 будем использовать сверточные слои с выходами 80x80, 40x40 и 20x20 соответственно.

FPN получает на вход карты абстракций C3-5 и строит пирамиду уровней P3-5. Затем добавляются дополнительные уровни P6 и P7, получаемые при помощи сверток с шагом 2. Таким образом обеспечивается анализ объектов в широком масштабе: от малых до крупных.

Классификационную и регрессионную подсети оформим в виде функций, которые будут возвращать готовую подсеть tf.keras.Model с входами и выходами. Каждой из подсетей добавим по четыре одинаковых уровня layers.Conv2D согласно разобранной архитектуре.

На основе построенных backbone, FPN и подсетей классификации и регрессии соберем базовую модель RetinaNet, которая станет основой нашей кастомной модели.

Для подсчета ошибок во время обучения создадим кастомные метрики FocalLoss (см. рис. 2.4) и SmoothL1Loss (см. рис. 2.5) согласно представленным ранее формулам. Для удобства совместимости будем наследовать классы реализаций от базового класса tf.keras.losses.Loss.

```python
class FocalLoss(tf.keras.losses.Loss):
    def __init__(self, alpha=0.25, gamma=2.0, **kwargs):
        super().__init__(**kwargs)
        self.alpha = alpha
        self.gamma = gamma

    def call(self, y_true, y_pred):
        ce_loss = tf.nn.softmax_cross_entropy_with_logits(labels=y_true, logits=y_pred)
        probs = tf.nn.softmax(y_pred)
        probs_true = tf.reduce_sum(probs * y_true, axis=-1)
        alpha_factor = y_true * self.alpha + (1 - y_true) * (1 - self.alpha)
        alpha_factor = tf.reduce_sum(alpha_factor, axis=-1)
        modulating_factor = tf.pow(1.0 - probs_true, self.gamma)
        loss = alpha_factor * modulating_factor * ce_loss
        return tf.reduce_mean(loss)
```

Рисунок 2.3 – Кастомная реализация метрики FocalLoss

```python
class SmoothL1Loss(tf.keras.losses.Loss):
    def __init__(self, delta=1.0, **kwargs):
        super().__init__(**kwargs)
        self.delta = delta

    def call(self, y_true, y_pred):
        diff = tf.abs(y_true - y_pred)
        less = tf.less(diff, self.delta)
        loss = tf.where(less, 0.5 * tf.square(diff), diff - 0.5)
        return tf.reduce_mean(loss)
```

Рисунок 2.4 – Кастомная реализация метрики SmoothL1Loss

На каждом уровне пирамиды RetinaNet создает anchor-боксы – заранее определенные прямоугольные рамки с разными размерами и соотношениями сторон. Во время обучения каждый anchor сопоставляется с реальным объектом (bbox) на основе метрики IoU (Intersection over Union). На рисунке 2.5. представлена кастомная функция данной метрики.

```python
def compute_iou(boxes1, boxes2):
    boxes1 = tf.expand_dims(boxes1, 1)
    boxes2 = tf.expand_dims(boxes2, 0)

    y1 = tf.maximum(boxes1[..., 0], boxes2[..., 0])
    x1 = tf.maximum(boxes1[..., 1], boxes2[..., 1])
    y2 = tf.minimum(boxes1[..., 2], boxes2[..., 2])
    x2 = tf.minimum(boxes1[..., 3], boxes2[..., 3])

    inter = tf.maximum(0.0, y2 - y1) * tf.maximum(0.0, x2 - x1)
    area1 = (boxes1[..., 2] - boxes1[..., 0]) * (boxes1[..., 3] - boxes1[..., 1])
    area2 = (boxes2[..., 2] - boxes2[..., 0]) * (boxes2[..., 3] - boxes2[..., 1])
    union = area1 + area2 - inter
    return inter / tf.maximum(union, 1e-8)
```

Рисунок 2.5 – Реализация метрики IoU

После разработки базовой архитектуры RetinaNet, пирамиды FPN и функции вычисления anchor-рамок создадим класс-обертку для нашей модели. Унаследуем кастомную модель от tf.keras.Model. Добавим методы построения и компиляции модели. Опишем методы train_step и test_step, реализующие последовательность действий в рамках одного шага обучения и шага валидации соответственно.

Полный код реализации всех компонентов модели представлен в приложении Б.

Скомпилируем нашу модель и посмотрим на суммарную архитектуру методом .summary() библиотеки keras. Выведенная таблица с описанием слоев, их типов и количества параметров представлена на рис. 2.6 и 2.7.

Model: "RetinaNet"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 640, 640, 3)	0	-
functional (Functional)	[(None, 80, 80, 512), (None, 40, 40, 1024), (None, 20, 20, 2048)]	23,587,712	input_layer[0][0]
conv2d (Conv2D)	(None, 20, 20, 256)	524,544	functional[0][2]
up_sampling2d (UpSampling2D)	(None, 40, 40, 256)	0	conv2d[0][0]
conv2d_1 (Conv2D)	(None, 40, 40, 256)	262,400	functional[0][1]
add (Add)	(None, 40, 40, 256)	0	up_sampling2d[0]… conv2d_1[0][0]
up_sampling2d_1 (UpSampling2D)	(None, 80, 80, 256)	0	add[0][0]
conv2d_2 (Conv2D)	(None, 80, 80, 256)	131,328	functional[0][0]
conv2d_6 (Conv2D)	(None, 10, 10, 256)	4,718,848	functional[0][2]
add_1 (Add)	(None, 80, 80, 256)	0	up_sampling2d_1[… conv2d_2[0][0]
activation (Activation)	(None, 10, 10, 256)	0	conv2d_6[0][0]
conv2d_3 (Conv2D)	(None, 80, 80, 256)	590,080	add_1[0][0]
conv2d_4 (Conv2D)	(None, 40, 40, 256)	590,080	add[0][0]
conv2d_5 (Conv2D)	(None, 20, 20, 256)	590,080	conv2d[0][0]
conv2d_7 (Conv2D)	(None, 5, 5, 256)	590,080	activation[0][0]

Рисунок 2.6 – Описание архитектуры модели. Часть первая

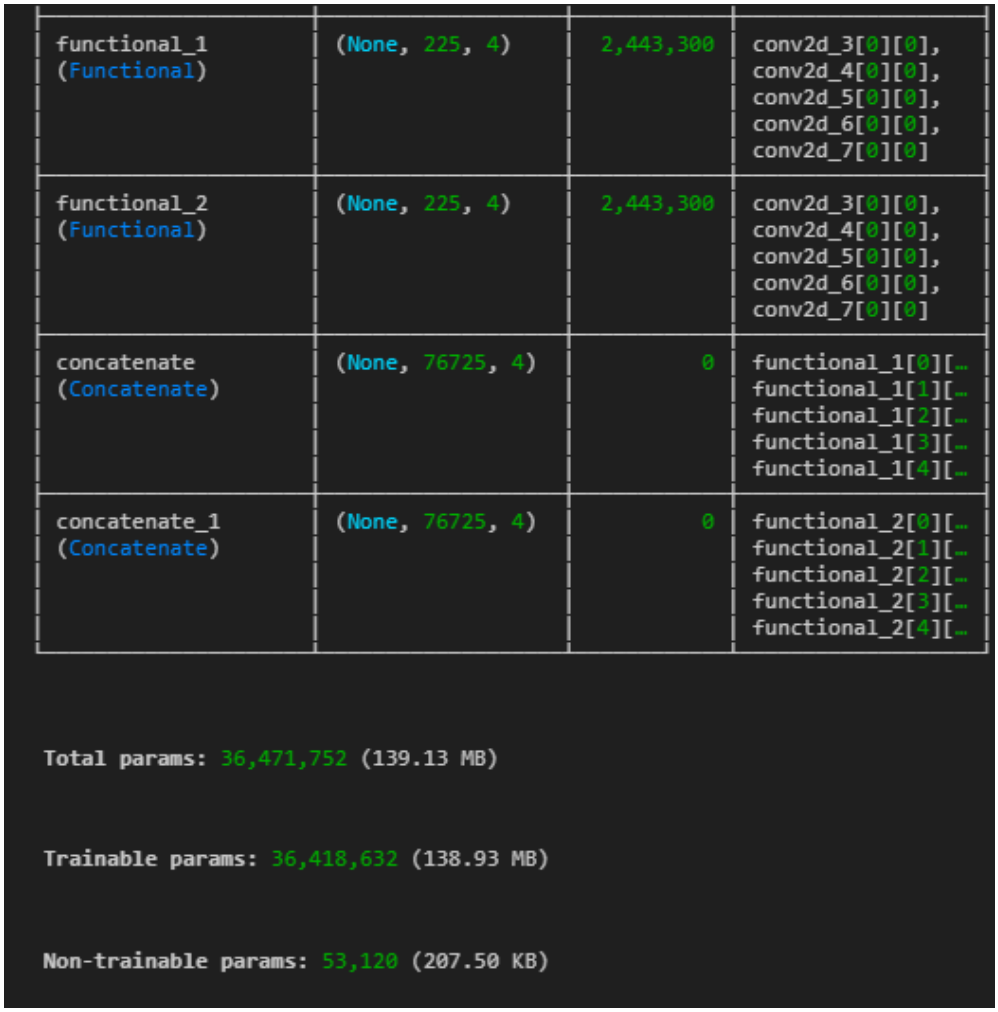

Рисунок 2.7 – Описание архитектуры модели. Часть вторая

2.5 Обучение модели

После программной реализации архитектуры приступим к обучению модели. Подгрузим файлы сериализованного формата TFRecord для обеспечения более быстрой обработки данных. Для обучения понадобятся выборки train и valid.

Произведем настройку параметров обучения. С учетом объемов данных по выборкам определим число шагов за одну эпоху и число эпох.

После нескольких неудачных попыток обучения протестируем вариант с батчами размером в 4 изображения. Число шагов на эпоху установим равным 256, а самих эпох – 16.

К сожалению, где-то после часа обучения, на третьей эпохе, среда выполнения выдала ошибку о падении вычислительного ядра. Возможно, была превышена нагрузка на процессор. К тому же время обучения оставляло желать лучшего. После испытания разных вариантов было решено запустить обучение с 8 изображениями на батч и надеждой на многоядерность процессора. Сначала был произведен пробный запуск на 10 эпох по 20 шагов в каждой (см. рис. 2.8).

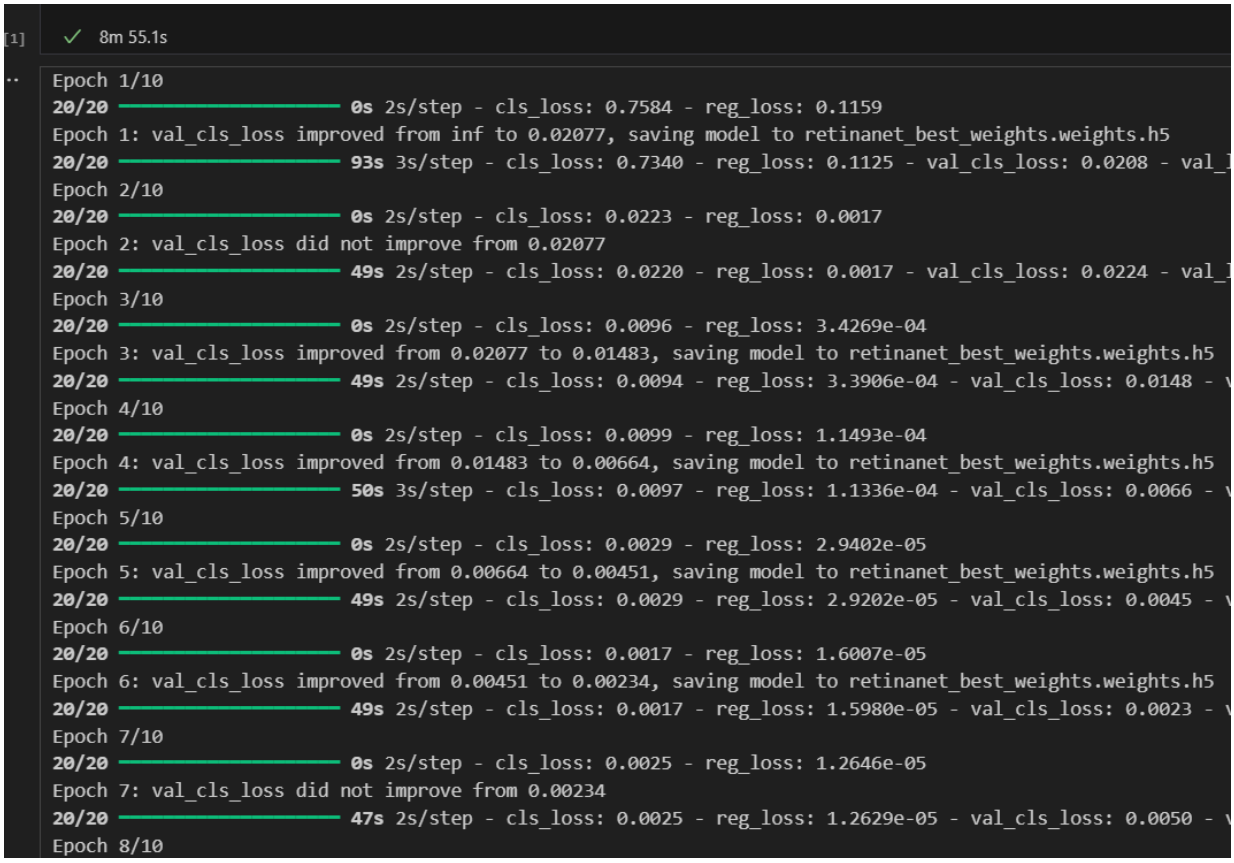

Рисунок 2.8 – Пробный прогон обучения

Пробный запуск завершился успешно, при этом сохранение весов модели также оказалось работоспособным. Поэтому было решено перейти к полномасштабному обучению модели (см. рис. 2.9 и 2.10). Было определено 12 эпох по 256 шагов в каждой. Число шагов на валидационной выборке – 220. Размер батча оставлен равным 8.

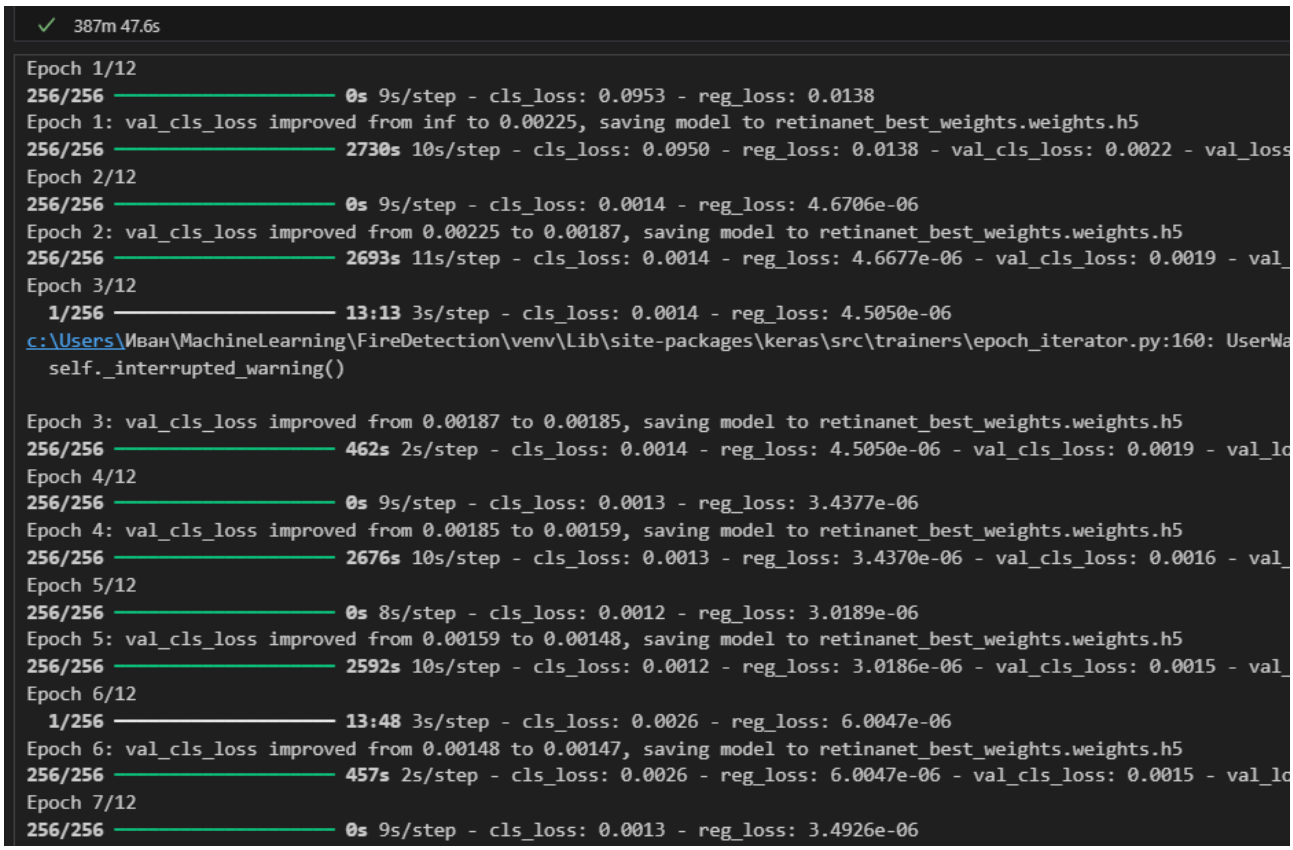

Рисунок 2.9 – Успешное обучение модели. Часть первая

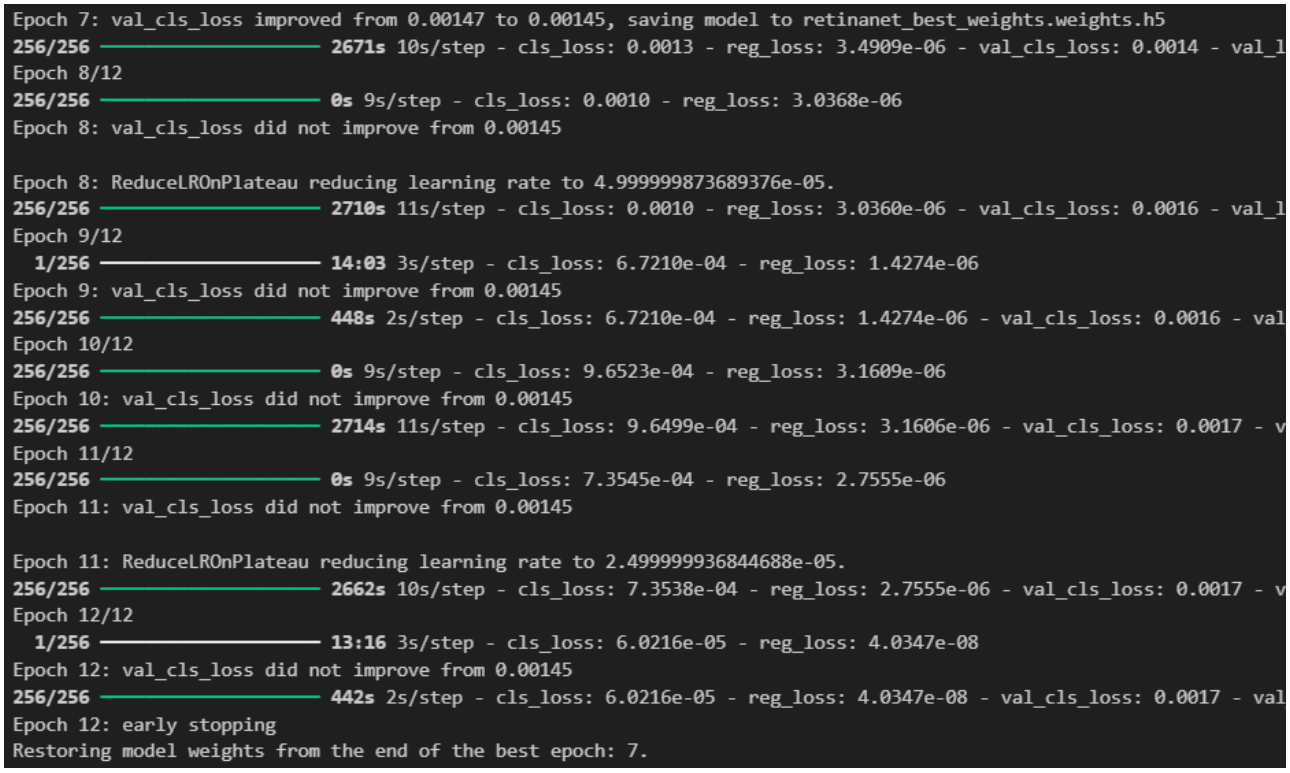

Рисунок 2.10 – Успешное обучение модели. Часть вторая

Обучение длилось более 6 часов (387 минут) и показывало успешную динамику с корректным сохранением весов и редуцированием параметра обучения согласно написанным колбэкам.

Рассмотрим динамику обучения по графикам, построенным на основе данных автоматического логирования метода .fit().

На рис. 2.11 и 2.12 показана динамика ошибок на данных обучения.

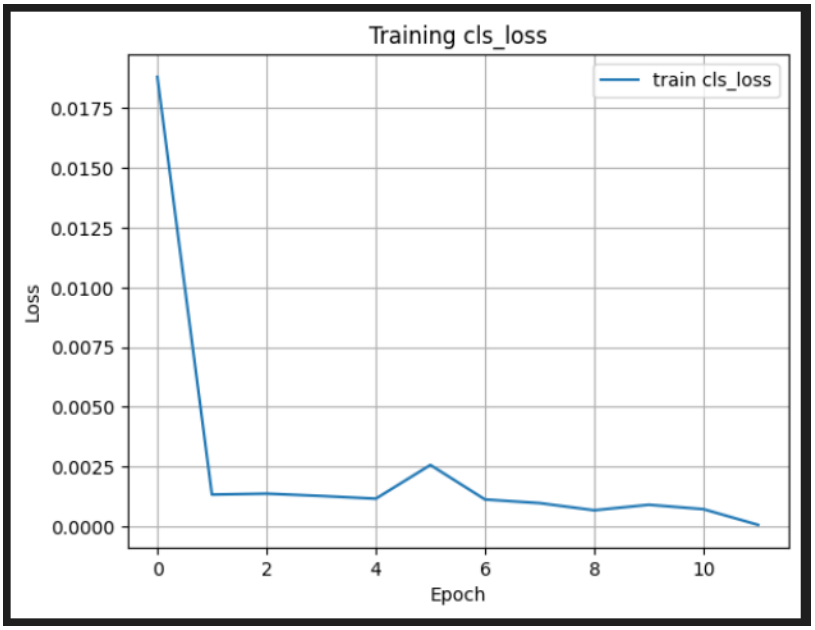

Рисунок 2.11 – Динамика ошибки классификации на train выборке

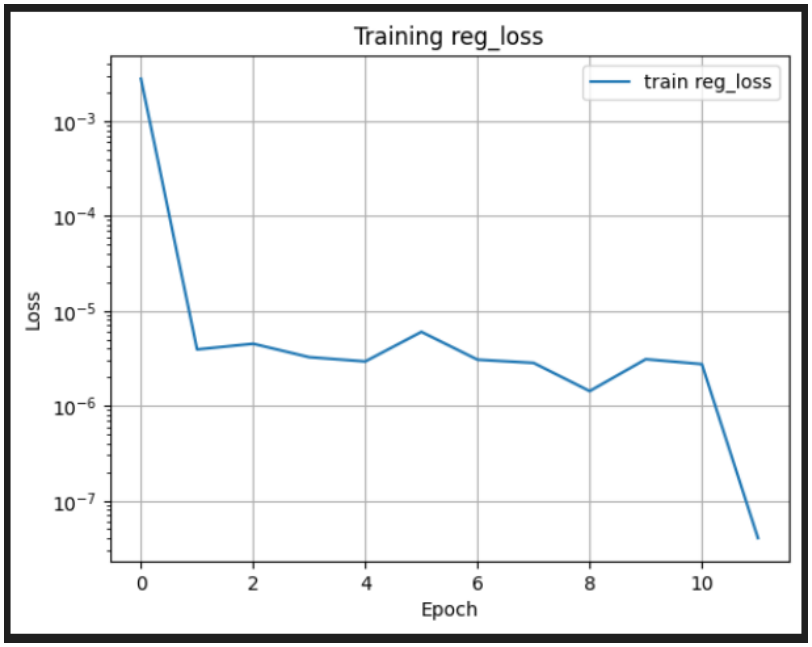

Рисунок 2.12 – Динамика ошибки регрессии на train выборке

А на рис. 2.13 и 2.14 показана динамика ошибок на валидационной выборке после окончания каждой эпохи обучения.

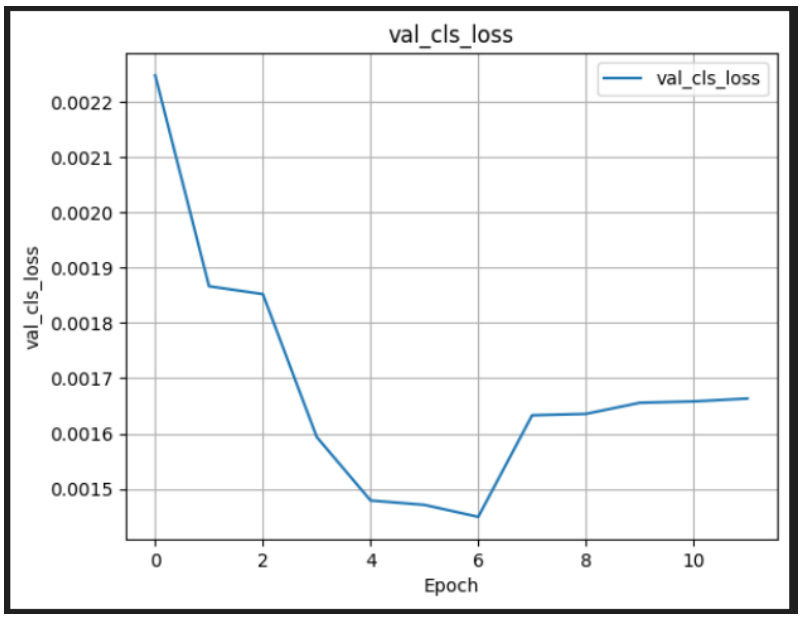

Рисунок 2.13 – Динамика ошибки классификации на valid выборке

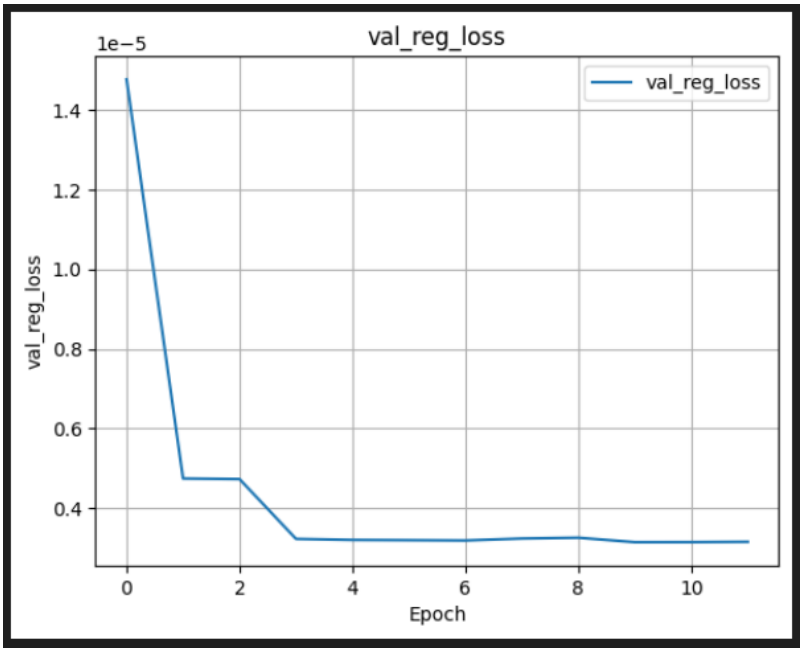

Рисунок 2.14 – Динамика ошибки регрессии на valid выборке

В целом, графики показывают неплохую тенденцию к снижению ошибок как классификации, так и регрессии. Тем не менее, есть подозрение на возможность переобучения модели.

2.6 Анализ результатов

Проведем анализ результатов обучения. Загрузим лучшие сохранившиеся веса модели и определим метрику для оценки.

В задачах обнаружения объектов общепринятым стандартом является метрика mAP (mean Average Precision). Она одновременно отражает точность (precision) и полноту (recall) модели, вычисляя площадь под кривой Precision-Recall (PR-curve) для каждого класса и усредняя ее. В отличие от простых бинарных метрик mAP учитывает локализацию объекта (по метрике IoU) и считает качество детекции каждого из классов. [9]

Напишем собственную реализацию данной метрики (см. рис. 2.15).

```
from mean_average_precision import MetricBuilder
import numpy as np

def evaluate_map(model, val_ds, num_classes=4, score_threshold=0.05, iou_threshold=0.5):
    metric_fn = MetricBuilder.build_evaluation_metric("map_2d", async_mode=False, num_cla
    for images, targets in val_ds:
        cls_pred, reg_pred = model.model(images, training=False)
        cls_pred_np = cls_pred.numpy()
        reg_pred_np = reg_pred.numpy()
        for i in range(images.shape[0]):
            scores = np.max(cls_pred_np[i], axis=-1)
            labels = np.argmax(cls_pred_np[i], axis=-1)
            boxes = reg_pred_np[i]
            keep = scores > score_threshold
            pred_boxes = boxes[keep]
            pred_scores = scores[keep]
            pred_labels = labels[keep]
            preds = np.concatenate([
                pred_boxes,
                pred_scores[:, None],
                pred_labels[:, None]
            ], axis=1)
            gt_boxes = targets[i].numpy()
            gt_boxes = gt_boxes[gt_boxes[:, 4] > 0]
            gts = np.concatenate([
                gt_boxes[:, :4],
                (gt_boxes[:, 4] - 1)[:, None],
                np.zeros((gt_boxes.shape[0], 2))
            ], axis=1)
            metric_fn.add(preds, gts)
    print('mAP:', metric_fn.value(iou_thresholds=iou_threshold)['mAP'])
```

Рисунок 2.15 – Реализация метрики mAP

Проведем оценку модели при помощи данной метрики. Будем использовать тестовую (test) выборку.

Показатель метрики mAP представлен на рис. 2.16.

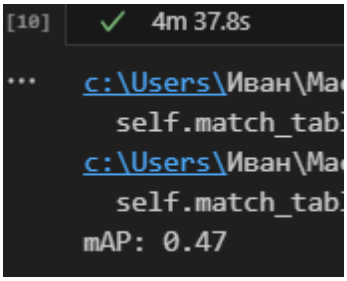

Рисунок 2.16 – Показатель метрики mAP

Проверим показатель визуализацией детекции объектов (см. рис. 2.17).

Рисунок 2.17 – Пример многоклассовой детекции объектов

Несмотря на то, что показатель mAP, равный 0,47, не является выдающимся, он представляет собой достойный средний результат с учетом сложности решаемой задачи многоклассовой детекции. Проанализируем возможные проблемы обучения модели.

Наличие перекрывающихся рамок объектов на изображениях часто усложняет классификацию детектированной области. В нашем случае это

особенно чувствительно, т.к. огонь и дым зачастую присутствуют в локации одновременно.

Дисбаланс классов в виде наличия небольшого количества примеров побочного класса Other для огнеподобных аномальных объектов также играет роль в снижении точности классификации.

Помимо этого, качество детекции во многом зависит от точности разметки. Поэтому при проведении повторных испытаний необходимо будет, по возможности, вручную проверить весь набор данных на соответствие рамок реальному расположению объектов. Данную задачу можно несколько ускорить, написав GUI утилиту, которая будет принимать на вход поток изображений и аннотаций и размечать рамки. А ревизору будет достаточно только одобрять или отклонять изображения.

Также стоит учитывать, что архитектура RetinaNet мало распространена в задачах детекции возгораний и задымлений, поэтому ее применение в контексте данной работы имело больше характер исследования.

ЗАКЛЮЧЕНИЕ

В данной выпускной квалификационной работе рассматривалась одна из актуальных и практически значимых задач компьютерного зрения – автоматическое обнаружение признаков возгорания на изображении.

В рамках выполненного исследования был проведен глубокий анализ предметной области: рассмотрены визуальные аномалии данных, создающие помехи распознавания, а также проанализированы существующие подходы и средства автоматизированного решения проблемы.

Первой частью работы над созданием собственной модели детекции стал поиск и выбор данных для обучения. По результату просмотра множества наборов данных с разных платформ был избран наиболее подходящий. Затем производился статистический анализ данных, выявивший проблемы набора. Проблема была решена путем добавления отрицательной выборки, взятой из других наборов данных. В целях удобства загрузки данных для обучения была произведена сериализация данных в бинарный формат.

Вторым этапом работы стала постановка задачи машинного обучения. Отдельно рассматривались возможные архитектуры реализации. По выбранной архитектуре RetinaNet был проведен подробный обзор составляющих компонентов, позволивший в дальнейшем осуществить собственную программную реализацию. Выполненная при помощи средств библиотеки Keras разработка модели была продолжена процессом обучения. В ходе многочисленных экспериментов с параметрами обучения было зафиксировано наиболее удачное обучение модели. На основе лучших сохраненных весов была произведена оценка качества модели при помощи метрики mAP. Несмотря на то, что показатель качества не оправдал ожиданий обучения, он, все же, свидетельствует о приемлемом среднем качестве модели в рамках исследования.

Таким образом, все поставленные задачи исследования выполнены успешно. В результате работы получена модель компьютерного зрения, выполняющая задачу детекции возгорания. Цель по разработке собственной модели нейронной сети была достигнута со степенью обобщающей способности среднего качества. Произведенный анализ предметной области оставляет также и пространство для дальнейших исследований. В частности, для улучшения качества модели существует возможность дополнительной обработки данных вроде аугментации. Говоря о поставленной проблеме предметной области, для исследования открыты также и другие подходы к реализации нейронных сетей и ансамблевых решений.

Задача автоматического обнаружения признаков возгорания остается одной из ключевых в области прикладного применения компьютерного зрения. На данный момент именно интеллектуальные алгоритмы остаются самым массово применимым и точным методом обеспечения своевременного мониторинга пожарной ситуации. Разработка моделей решения данной проблемы представляет не только инженерный вызов, но и вклад в формирование безопасной окружающей среды. И кто знает, быть может, данная работа станет основой дальнейших исследований, направленных на повышение надежности таких систем.

СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Савченко А.А. Комплексный статистический анализ причин пожаров и их экономических последствий для развития региона: магистерская диссертация [Электронный ресурс] / Тольятти: ТГУ, 2024. – 91 с. – URL: https://dspace.tltsu.ru/bitstream/123456789/29917/1/Савченко%20А.А._ТБмд-2104а.pdf (дата обращения: 27.05.2025).

2. Turgay Çelik, Hasan Demirel. Fire detection in video sequences using a generic color model [Электронный ресурс] // Fire Safety Journal. – 2009. – Vol. 44, № 2. – P. 147-158. – URL: https://doi.org/10.1016/j.firesaf.2008.05.005 (дата обращения: 15.06.25).

3. He Y, Sahma A, He X, Wu R, Zhang R. FireNet: A Lightweight and Efficient Multi-Scenario Fire Object Detector [Электронный ресурс] // Remote Sensing. – 2024. – Vol. 16(21):4112. – URL: https://doi.org/10.3390/rs16214112 (дата обращения: 15.06.25).

4. Lin T.-Y., Maire M., Belongie S. et al. Microsoft COCO: Common Objects in Context [Электронный ресурс] // arXiv. – 2014. URL: https://arxiv.org/pdf/1405.0312 (дата обращения: 29.05.2025).

5. TFRecord и tf.train.Example: [Электронный ресурс]. URL: https://www.tensorflow.org/tutorials/load_data/tfrecord?hl=ru (дата обращения: 15.06.25).

6. Архитектура нейронной сети RetinaNet: [Электронный ресурс]. URL: https://habr.com/ru/articles/510560/ (дата обращения: 16.06.25).

7. Lin T. Y. et al. Focal loss for dense object detection [Электронный ресурс] // Proceedings of the IEEE international conference on computer vision. – 2017. – С. 2980-2988. – URL: https://arxiv.org/pdf/1708.02002 (дата обращения: 17.06.25).

8. Keras Functional API: [Электронный ресурс]. URL: https://kirenz.github.io/deep-learning/docs/keras-functional.html (дата обращения: 17.06.25).

9. Mean Average Precision (mAP) in Computer Vision: [Электронный ресурс]. URL: https://www.geeksforgeeks.org/computer-vision/mean-average-precision-map-in-computer-vision/ (дата обращения: 20.06.25).

ПРИЛОЖЕНИЕ А

Программная реализация статистического анализа датасета

```
import json
import matplotlib.pyplot as plt
from collections import defaultdict, Counter
import statistics

def load_coco_json(json_path):
    with open(json_path, 'r', encoding='utf-8') as f:
        return json.load(f)

def get_category_info(coco_data):
    return {cat["id"]: cat["name"] for cat in coco_data["categories"]}

def analyze_annotations(coco_data):
    annotations = coco_data["annotations"]
    images = coco_data["images"]
    category_names = get_category_info(coco_data)
    annotated_image_ids = {ann["image_id"] for ann in annotations}
    images_without_annotations = [
        img for img in images if img["id"] not in annotated_image_ids
    ]
    category_counter = Counter()
    for ann in annotations:
        category_counter[ann["category_id"]] += 1
    category_counter_named = {category_names[k]: v for k, v in
category_counter.items()}
    return {
        "num_images": len(images),
        "num_annotations": len(annotations),
        "num_images_without_annotations":
len(images_without_annotations),
        "category_counts": category_counter_named,
        "images_without_annotations": images_without_annotations
    }

def print_report(stats, sample):
    print(f"\nОбщая статистика по {sample}:")
    print(f"Изображений: {stats['num_images']}")
    print(f"Изображений без аннотаций:
{stats['num_images_without_annotations']}")
    print(f"Аннотаций: {stats['num_annotations']}")
    print("Кол-во аннотаций по категориям:")
    for category, count in stats['category_counts'].items():
        print(f"  - {category}: {count}")

for sample in ['train', 'test', 'valid']:
    coco_data = load_coco_json(f'./Fire-smoke-
dataset/{sample}/_annotations.coco.json')
```

```
    stats = analyze_annotations(coco_data)
    print_report(stats, sample)

with open("Fire-smoke-dataset/train/_annotations.coco.json", "r",
encoding="utf-8") as f:
    coco = json.load(f)

images = {img["id"]: img for img in coco["images"]}
categories = {cat["id"]: cat["name"] for cat in coco["categories"]}
annotations = coco["annotations"]

image_to_anns = defaultdict(list)
class_counter = Counter()
combo_counter = Counter()
bbox_areas = []

for ann in annotations:
    image_to_anns[ann["image_id"]].append(ann)
    class_counter[ann["category_id"]] += 1
    bbox = ann["bbox"]
    area = bbox[2] * bbox[3]
    bbox_areas.append(area)

for img_id, anns in image_to_anns.items():
    class_combo = tuple(sorted(set(ann["category_id"] for ann in anns)))
    combo_counter[class_combo] += 1

plt.figure(figsize=(6, 6))
labels = [", ".join(categories[c] for c in k) for k in
combo_counter.keys()]
sizes = list(combo_counter.values())
plt.pie(sizes, labels=labels, autopct="%1.1f%%", startangle=30)
plt.title("Сочетания классов на изображениях")
plt.axis("equal")
plt.show()

plt.figure(figsize=(8, 4))
plt.hist(bbox_areas, bins=40, log=True, color='skyblue')
plt.title("Площади bounding boxes")
plt.xlabel("Площадь (px²)")
plt.ylabel("Количество (логарифмическая шкала)")
plt.show()

median_area = statistics.median(bbox_areas)
image_area = 640 * 640
print(f"Медианная площадь bbox: {median_area}")
print(f"Медианное отношение площадей: {median_area / image_area}")
```

Листинг 1. Скрипт для статистического анализа данных

ПРИЛОЖЕНИЕ Б

Программная реализация модели RetinaNet

```
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np

# Конфигурация модели
INPUT_SIZE = 640
NUM_CLASSES = 4
BATCH_SIZE = 8

def build_retinanet(num_classes, input_shape=(INPUT_SIZE, INPUT_SIZE,
3), feature_size=256):
    # Backbone
    backbone_base = tf.keras.applications.ResNet50(
        include_top=False,
        weights='imagenet',
        input_shape=input_shape
    )
    c3 = backbone_base.get_layer("conv3_block4_out").output  # 80x80
    c4 = backbone_base.get_layer("conv4_block6_out").output  # 40x40
    c5 = backbone_base.get_layer("conv5_block3_out").output  # 20x20
    backbone = tf.keras.Model(inputs=backbone_base.input, outputs=[c3,
c4, c5])

    # FPN
    def build_fpn(c3, c4, c5):
        p5_1 = layers.Conv2D(feature_size, 1)(c5)
        p4_1 = layers.Conv2D(feature_size, 1)(c4)
        p3_1 = layers.Conv2D(feature_size, 1)(c3)

        p5_up = layers.UpSampling2D()(p5_1)
        p4_merge = layers.Add()([p5_up, p4_1])
        p4_up = layers.UpSampling2D()(p4_merge)
        p3_merge = layers.Add()([p4_up, p3_1])

        p3 = layers.Conv2D(feature_size, 3, padding='same')(p3_merge)
        p4 = layers.Conv2D(feature_size, 3, padding='same')(p4_merge)
        p5 = layers.Conv2D(feature_size, 3, padding='same')(p5_1)

        p6 = layers.Conv2D(feature_size, 3, strides=2,
padding='same')(c5)
        p6_relu = layers.Activation('relu')(p6)
        p7 = layers.Conv2D(feature_size, 3, strides=2,
padding='same')(p6_relu)

        return [p3, p4, p5, p6, p7]

    # Классификационная подсеть
```

```
    def classification_subnet(num_classes, num_anchors=9):
        inputs = layers.Input(shape=(None, None, feature_size))
        x = inputs
        for _ in range(4):
            x = layers.Conv2D(feature_size, 3, padding='same',
activation='relu')(x)
        x = layers.Conv2D(num_classes * num_anchors, 3,
padding='same')(x)
        x = layers.Reshape((-1, num_classes))(x)
        return tf.keras.Model(inputs=inputs, outputs=x)

    # Регрессионная подсеть
    def regression_subnet(num_anchors=9):
        inputs = layers.Input(shape=(None, None, feature_size))
        x = inputs
        for _ in range(4):
            x = layers.Conv2D(feature_size, 3, padding='same',
activation='relu')(x)
        x = layers.Conv2D(4 * num_anchors, 3, padding='same')(x)
        x = layers.Reshape((-1, 4))(x)
        return tf.keras.Model(inputs=inputs, outputs=x)

    inputs = backbone.input
    c3, c4, c5 = backbone(inputs)
    features = build_fpn(c3, c4, c5)
    cls_head = classification_subnet(num_classes)
    reg_head = regression_subnet()
    cls_outputs = [cls_head(f) for f in features]
    reg_outputs = [reg_head(f) for f in features]

    cls_output = layers.Concatenate(axis=1)(cls_outputs)
    reg_output = layers.Concatenate(axis=1)(reg_outputs)
    return tf.keras.Model(inputs=inputs, outputs=[cls_output,
reg_output], name="RetinaNet")

# Создание якорных рамок
def generate_anchors(base_size, scales, ratios):
    anchors = []
    for scale in scales:
        for ratio in ratios:
            w = base_size * scale * np.sqrt(1.0 / ratio)
            h = base_size * scale * np.sqrt(ratio)
            x1 = -w / 2
            y1 = -h / 2
            x2 = w / 2
            y2 = h / 2
            anchors.append([x1, y1, x2, y2])
    return np.array(anchors)

# Размещение рамок
def shift(feature_map_shape, stride, anchors):
    H, W = feature_map_shape
    shift_x = (np.arange(W) + 0.5) * stride
```

```
    shift_y = (np.arange(H) + 0.5) * stride
    shift_x, shift_y = np.meshgrid(shift_x, shift_y)
    shifts = np.stack([shift_x.ravel(), shift_y.ravel(),
                       shift_x.ravel(), shift_y.ravel()], axis=1)
    A = anchors.shape[0]
    K = shifts.shape[0]
    all_anchors = anchors.reshape((1, A, 4)) + shifts.reshape((K, 1, 4))
    return all_anchors.reshape((-1, 4))

# Генерация якорных рамок для всех уровней FRN
def generate_all_anchors(feature_map_shapes, strides, base_size=32):
    anchors_per_level = []
    scales = [2**0, 2**(1/3), 2**(2/3)]
    ratios = [1.0, 2.0, 0.5]
    for shape, stride in zip(feature_map_shapes, strides):
        anchors = generate_anchors(base_size, scales, ratios)
        shifted = shift(shape, stride, anchors)
        anchors_per_level.append(shifted)
    return np.concatenate(anchors_per_level, axis=0)

# Метрика Intersection Over Union
def compute_iou(boxes1, boxes2):
    boxes1 = tf.expand_dims(boxes1, 1)
    boxes2 = tf.expand_dims(boxes2, 0)

    y1 = tf.maximum(boxes1[..., 0], boxes2[..., 0])
    x1 = tf.maximum(boxes1[..., 1], boxes2[..., 1])
    y2 = tf.minimum(boxes1[..., 2], boxes2[..., 2])
    x2 = tf.minimum(boxes1[..., 3], boxes2[..., 3])

    inter = tf.maximum(0.0, y2 - y1) * tf.maximum(0.0, x2 - x1)
    area1 = (boxes1[..., 2] - boxes1[..., 0]) * (boxes1[..., 3] -
boxes1[..., 1])
    area2 = (boxes2[..., 2] - boxes2[..., 0]) * (boxes2[..., 3] -
boxes2[..., 1])
    union = area1 + area2 - inter
    return inter / tf.maximum(union, 1e-8)

# Вычисление смещения рамок от bbox
def box_transform(anchors, gt_boxes):
    ay = (anchors[:, 0] + anchors[:, 2]) / 2.0
    ax = (anchors[:, 1] + anchors[:, 3]) / 2.0
    ah = anchors[:, 2] - anchors[:, 0]
    aw = anchors[:, 3] - anchors[:, 1]

    by = (gt_boxes[:, 0] + gt_boxes[:, 2]) / 2.0
    bx = (gt_boxes[:, 1] + gt_boxes[:, 3]) / 2.0
    bh = gt_boxes[:, 2] - gt_boxes[:, 0]
    bw = gt_boxes[:, 3] - gt_boxes[:, 1]

    ty = (by - ay) / ah
    tx = (bx - ax) / aw
    th = tf.math.log(bh / ah)
```

```
    tw = tf.math.log(bw / aw)

    return tf.stack([ty, tx, th, tw], axis=1)

# Кодирование целевых значений
def encode_targets(anchors, gt_boxes, gt_labels, num_classes,
iou_threshold_pos=0.5, iou_threshold_neg=0.4):
    num_gt = tf.shape(gt_boxes)[0]

    def encode_with_gt():
        iou_matrix = compute_iou(anchors, gt_boxes)
        matched_iou = tf.reduce_max(iou_matrix, axis=1)
        matched_idx = tf.argmax(iou_matrix, axis=1,
output_type=tf.int32)
        cls_targets = tf.one_hot(tf.fill([tf.shape(anchors)[0]],
num_classes - 1), depth=num_classes)
        reg_targets = tf.zeros_like(anchors, dtype=tf.float32)

        pos_mask = matched_iou >= iou_threshold_pos
        pos_indices = tf.where(pos_mask)[:, 0]
        has_pos = tf.shape(pos_indices)[0] > 0

        def process_positive():
            matched_gt = tf.gather(gt_boxes, tf.gather(matched_idx,
pos_indices))
            pos_labels = tf.gather(gt_labels, tf.gather(matched_idx,
pos_indices))
            pos_one_hot = tf.one_hot(pos_labels - 1, depth=num_classes)
            cls_targets_updated = tf.tensor_scatter_nd_update(
                cls_targets,
                tf.expand_dims(pos_indices, axis=1),
                pos_one_hot
            )
            pos_anchors = tf.gather(anchors, pos_indices)
            reg_values = box_transform(pos_anchors, matched_gt)
            reg_targets_updated =
tf.tensor_scatter_nd_update(reg_targets, tf.expand_dims(pos_indices, 1),
reg_values)
            return cls_targets_updated, reg_targets_updated

        def no_positive():
            return cls_targets, reg_targets

        return tf.cond(has_pos, process_positive, no_positive)

    def encode_no_gt():
        cls_targets = tf.one_hot(tf.fill([tf.shape(anchors)[0]],
num_classes - 1), depth=num_classes)
        reg_targets = tf.zeros_like(anchors, dtype=tf.float32)
        return cls_targets, reg_targets

    return tf.cond(num_gt > 0, encode_with_gt, encode_no_gt)
```

```
# Метрика FocalLoss
class FocalLoss(tf.keras.losses.Loss):
    def __init__(self, alpha=0.25, gamma=2.0, **kwargs):
        super().__init__(**kwargs)
        self.alpha = alpha
        self.gamma = gamma

    def call(self, y_true, y_pred):
        ce_loss = tf.nn.softmax_cross_entropy_with_logits(labels=y_true,
logits=y_pred)
        probs = tf.nn.softmax(y_pred)
        probs_true = tf.reduce_sum(probs * y_true, axis=-1)
        alpha_factor = y_true * self.alpha + (1 - y_true) * (1 -
self.alpha)
        alpha_factor = tf.reduce_sum(alpha_factor, axis=-1)
        modulating_factor = tf.pow(1.0 - probs_true, self.gamma)
        loss = alpha_factor * modulating_factor * ce_loss
        return tf.reduce_mean(loss)

# Метрика SmoothL1Loss
class SmoothL1Loss(tf.keras.losses.Loss):
    def __init__(self, delta=1.0, **kwargs):
        super().__init__(**kwargs)
        self.delta = delta

    def call(self, y_true, y_pred):
        diff = tf.abs(y_true - y_pred)
        less = tf.less(diff, self.delta)
        loss = tf.where(less, 0.5 * tf.square(diff), diff - 0.5)
        return tf.reduce_mean(loss)

# Модель-обертка
class RetinaNetModel(tf.keras.Model):
    def __init__(self, base_model, anchors, num_classes, **kwargs):
        super().__init__(**kwargs)
        self.model = base_model
        self.anchors = tf.constant(anchors, dtype=tf.float32)
        self.num_classes = num_classes
        self.focal_loss = FocalLoss()
        self.reg_loss = SmoothL1Loss()
        self.cls_tracker = tf.keras.metrics.Mean(name="cls_loss")
        self.reg_tracker = tf.keras.metrics.Mean(name="reg_loss")

    def build(self, input_shape):
        self.model.build(input_shape)
        super().build(input_shape)

    def call(self, inputs, training=None, **kwargs):
        return self.model(inputs, training=training)

    def compile(self, optimizer, **kwargs):
        super().compile(**kwargs)
        self.optimizer = optimizer
```

```
    # Описание шага обучения
    def train_step(self, data):
        images, targets_raw = data

        def process_single_target(target):
            mask = tf.greater_equal(target[:, 4], 1)
            valid_targets = tf.boolean_mask(target, mask)

            def has_valid_targets():
                gt_boxes = valid_targets[:, :4]
                gt_labels = tf.cast(valid_targets[:, 4], tf.int32)
                return gt_boxes, gt_labels

            def no_valid_targets():
                gt_boxes = tf.zeros((0, 4), dtype=tf.float32)
                gt_labels = tf.zeros((0,), dtype=tf.int32)
                return gt_boxes, gt_labels

            gt_boxes, gt_labels = tf.cond(
                tf.shape(valid_targets)[0] > 0,
                has_valid_targets,
                no_valid_targets
            )
            cls_t, reg_t = encode_targets(self.anchors, gt_boxes,
gt_labels, self.num_classes)
            return cls_t, reg_t

        cls_targets, reg_targets = tf.map_fn(
            process_single_target,
            elems=targets_raw,
            fn_output_signature=(
                tf.TensorSpec(shape=(self.anchors.shape[0],
self.num_classes), dtype=tf.float32),
                tf.TensorSpec(shape=(self.anchors.shape[0], 4),
dtype=tf.float32),
            )
        )
        with tf.GradientTape() as tape:
            cls_pred, reg_pred = self.model(images, training=True)
            cls_loss = self.focal_loss(cls_targets, cls_pred)
            reg_loss = self.reg_loss(reg_targets, reg_pred)
            total_loss = cls_loss + reg_loss

        grads = tape.gradient(total_loss,
self.model.trainable_variables)
        self.optimizer.apply_gradients(zip(grads,
self.model.trainable_variables))
        self.cls_tracker.update_state(cls_loss)
        self.reg_tracker.update_state(reg_loss)
        return {"cls_loss": self.cls_tracker.result(), "reg_loss":
self.reg_tracker.result()}
```

```
    # Описание шага валидации после окончания эпохи
    def test_step(self, data):
        images, targets_raw = data

        def process_single_example(inputs):
            image, target = inputs
            mask = tf.greater_equal(target[:, 4], 1)
            valid_targets = tf.boolean_mask(target, mask)

            def has_valid_targets():
                gt_boxes = valid_targets[:, :4]
                gt_labels = tf.cast(valid_targets[:, 4], tf.int32)
                return gt_boxes, gt_labels

            def no_valid_targets():
                gt_boxes = tf.zeros((0, 4), dtype=tf.float32)
                gt_labels = tf.zeros((0,), dtype=tf.int32)
                return gt_boxes, gt_labels

            gt_boxes, gt_labels = tf.cond(
                tf.shape(valid_targets)[0] > 0,
                has_valid_targets,
                no_valid_targets
            )
            cls_t, reg_t = encode_targets(self.anchors, gt_boxes,
gt_labels, self.num_classes)
            return cls_t, reg_t

        cls_targets, reg_targets = tf.map_fn(
            process_single_example,
            (images, targets_raw),
            fn_output_signature=(
                tf.TensorSpec(shape=(self.anchors.shape[0],
self.num_classes), dtype=tf.float32),
                tf.TensorSpec(shape=(self.anchors.shape[0], 4),
dtype=tf.float32),
            )
        )
        cls_pred, reg_pred = self.model(images, training=False)
        cls_loss = self.focal_loss(cls_targets, cls_pred)
        reg_loss = self.reg_loss(reg_targets, reg_pred)
        total_loss = cls_loss + reg_loss
        self.cls_tracker.update_state(cls_loss)
        self.reg_tracker.update_state(reg_loss)
        return {"loss": total_loss, "cls_loss":
self.cls_tracker.result(), "reg_loss": self.reg_tracker.result()}

# Создание модели
num_classes = 4
base_model = build_retinanet(num_classes)

# Генерация якорных рамок
feature_shapes = [(80, 80), (40, 40), (20, 20), (10, 10), (5, 5)]
```

```
strides = [8, 16, 32, 64, 128]
anchors = generate_all_anchors(feature_shapes, strides)

# Оборачивание в кастомную модель
model = RetinaNetModel(base_model, anchors=anchors,
num_classes=num_classes)

# Построение модели
model.build((None, 640, 640, 3))

# Десериализация TFRecord
def parse_tfrecord_fn(example_proto, max_boxes=14):
    feature_description = {
        "image": tf.io.FixedLenFeature([], tf.string),
        "bboxes": tf.io.FixedLenFeature([max_boxes * 4], tf.float32),
        "labels": tf.io.FixedLenFeature([max_boxes], tf.int64)
    }
    example = tf.io.parse_single_example(example_proto,
feature_description)
    image = tf.image.decode_jpeg(example["image"], channels=3)
    image = tf.image.convert_image_dtype(image, tf.float32)
    image = tf.image.resize(image, (640, 640))
    bboxes = tf.reshape(example["bboxes"], (max_boxes, 4)) *
tf.constant([640.0, 640.0, 640.0, 640.0])
    labels = tf.reshape(example["labels"], (max_boxes, 1))
    targets = tf.concat([bboxes, tf.cast(labels, tf.float32)], axis=-1)
    return image, targets

# Загрузка датасета из TFRecord
def load_tfrecord_dataset(tfrecord_path, batch_size=8, shuffle=True):
    ds = tf.data.TFRecordDataset(tfrecord_path)
    ds = ds.map(lambda x: parse_tfrecord_fn(x),
num_parallel_calls=tf.data.AUTOTUNE)
    if shuffle:
        ds = ds.shuffle(1000)
    ds = ds.batch(batch_size).prefetch(tf.data.AUTOTUNE)
    return ds

train_ds = load_tfrecord_dataset("train.tfrecord", batch_size=8)
val_ds = load_tfrecord_dataset("valid.tfrecord", batch_size=8)

# Функции обратного вызова
callbacks = [
    tf.keras.callbacks.ModelCheckpoint(
        filepath="retinanet_best_weights.weights.h5",
        monitor="val_cls_loss",
        save_best_only=True,
        save_weights_only=True,
        verbose=1
    ),
    tf.keras.callbacks.EarlyStopping(
        monitor="val_cls_loss",
        mode='min',
```

```
        patience=5,
        restore_best_weights=True,
        verbose=1
    ),
    tf.keras.callbacks.ReduceLROnPlateau(
        monitor="val_cls_loss",
        mode='min',
        factor=0.5,
        patience=3,
        verbose=1
    )
]

# Компиляция модели
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4))

# Обучение модели
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=12,
    steps_per_epoch=256,
    validation_steps=220,
    callbacks=callbacks
)
```

Листинг 2. Реализация модели RetinaNet