Поиск объектов на видео с использованием Python и TensorFlow

Оглавление

[Введение 2](#_Toc171880070)

[Типы классификации изображений 3](#_Toc171880071)

[Обзор литературы 5](#_Toc171880072)

[Методология 6](#_Toc171880073)

[Шаги выполнения работы 7](#_Toc171880074)

[Результаты 17](#_Toc171880075)

[Заключение 18](#_Toc171880076)

[Список литературы 19](#_Toc171880077)

# **Введение**

В последние годы машинное обучение и компьютерное зрение стали важными направлениями в области искусственного интеллекта. Одной из ключевых задач компьютерного зрения является детекция объектов на видео, что имеет множество приложений, начиная от систем безопасности и заканчивая автономными транспортными средствами. Цель данной практической работы – изучение методов поиска объектов на видео с использованием Python и TensorFlow, а также их реализация и оценка эффективности.

Компьютерное зрение - это раздел искусственного интеллекта, который помогает компьютерам понимать и анализировать визуальные данные из цифровых изображений, видео и аналогичных визуальных входных данных. Используя цифровые визуальные данные, полученные из различных источников, мы можем научить компьютеры обнаруживать и интерпретировать визуальные объекты. Оно также играет важную роль в таких областях, как распознавание изображений и обнаружение объектов.

Компьютеры могут использовать изображения и видео для изучения и выполнения задач с использованием набора методов и алгоритмов. Эти методы и алгоритмы помогают им понимать визуальную информацию, выделяя важные детали из изображений и видео. Существует множество различных задач компьютерного зрения, и давайте подробно обсудим наиболее распространенные задачи компьютерного зрения и их применение в различных областях.

Одной из основных функций компьютерного зрения является классификация изображений. Основная цель - присвоить входному изображению предопределенную метку или категорию путем определения основного содержания конкретного изображения. Компьютерная система предсказывает, к какому классу или категории относится основное содержимое изображения. Классификация изображений в основном касается одного объекта. Например, модель классификации изображений может быть обучена распознавать и маркировать изображение, если оно содержит кошку, собаку, автомобиль, человека или определенный объект.

# **Типы классификации изображений**

Существует два основных типа классификации изображений для разделения их на предопределенные классы:

* **Классификация с одной меткой**: При классификации с одной меткой каждое изображение относят к одной категории, где цель состоит в том, чтобы предсказать одну метку для каждого изображения. Например, классифицировать изображение как содержащее кошку или собаку.
* **Классификация с несколькими метками**: Классификация с несколькими метками предполагает присвоение нескольких меток изображению, содержащему несколько объектов. Например, изображение может содержать кошку, собаку и дерево, и классификация изображений распознает все эти объекты и помечает их.

Одной из важных функций компьютерного зрения является обнаружение объектов. Основная цель обнаружения объектов - идентифицировать и локализовать определенные объекты в предоставляемых источниках ввода, таких как цифровые изображения или видео. Несколько примеров обнаружения объектов - определение местоположения пешехода на улице или автомобиля в потоке машин.

Процесс состоит из двух частей, а именно локализации объекта и классификации объектов, которые в совокупности составляют процесс обнаружения объекта:

* **Локализация объекта**: Локализация объекта означает определение местоположения объектов. Здесь мы обнаруживаем или идентифицируем объекты, точно определяя их конкретное местоположение на изображении или видео. Обнаружение объектов в задачах компьютерного зрения используйте ограничивающую рамку для обозначения местоположения объектов на изображении или отслеживания движущегося объекта на видео.
* **Классификация объектов**: Как только мы узнаем, где находятся объекты, мы переходим к классификации объектов. Это означает отнесение каждого объекта к заранее определенной категории, такой как "человек", "автомобиль" или "животное".

Сегментация из является важнейшей задачей в компьютерном зрении для разделения изображения на значимые сегменты или области. Разделенные сегменты могут соответствовать отдельным объектам, частям объектов или областям со схожими характеристиками. Этот процесс сегментации изображения может разбить изображение на значимые строительные блоки, чтобы помочь компьютеру идентифицировать и понять содержимое.

Основная цель сегментации изображения - разделить изображение на отдельные сегменты или области, которые связаны со значимыми объектами, областями или даже отдельными пикселями.

Существует 2 основных типа сегментации изображений:

* **Семантическая сегментация:** Семантическая сегментация в компьютерном зрении предполагает присвоение метки класса каждому отдельному пикселю изображения. Каждый пиксель изображения классифицируется и ему присваивается метка в зависимости от объекта, к которому он принадлежит. Когда семантическая сегментация выполняется на изображении, результатом является "карта сегментации", где цвет каждого пикселя представляет его класс.
* **Сегментация экземпляров:** Сегментация экземпляров позволяет изучить изображение на более детальном уровне путем идентификации и разграничения каждого отдельного экземпляра этих объектов. Это что-то вроде, например, наличия кошек разного цвета на изображении. Другим хорошим примером может быть, представьте групповую фотографию студентов. Семантическая сегментация помечает каждого как "человека", а сегментация экземпляра идентифицирует и выделяет каждого отдельного человека на групповой фотографии.

Существует также другой тип сегментации, называемый паноптической сегментацией, который сочетает в себе как семантическую, так и выборочную сегментацию для обеспечения полного понимания каждого пикселя изображения.

**Процесс сегментации изображений** используется в различных приложениях, таких как медицинская визуализация, для выявления опухолей или состояния органов, а также при автономном вождении, чтобы помочь различать дорогу, транспортные средства и пешеходов.

# **Обзор литературы**

Для выполнения практической работы были рассмотрены следующие исследования:

1. **"You Only Look Once: Unified, Real-Time Object Detection"** (Redmon et al., 2016) : В данной работе предложена модель YOLO (You Only Look Once), которая выполняет детекцию объектов в реальном времени. YOLO рассматривает детекцию объектов как задачу регрессии, что позволяет достигать высокой скорости и точности. Модель делит изображение на сетку и предсказывает границы объектов и их классы для каждой ячейки сетки.
2. **"SSD: Single Shot MultiBox Detector"** (Liu et al., 2016) : Авторы предложили архитектуру SSD, которая выполняет детекцию объектов с использованием одной сети и обеспечивает высокую точность и скорость. SSD использует несколько слоев с различными разрешениями для детекции объектов разного размера, что улучшает производительность модели.
3. **"Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks"** (Ren et al., 2015) : В этой работе представлена модель Faster R-CNN, которая улучшает производительность предыдущих моделей (R-CNN и Fast R-CNN) за счет введения сетей предложений регионов (RPN). Faster R-CNN генерирует предложения регионов и детектирует объекты в два этапа, обеспечивая высокую точность детекции.
4. **"EfficientDet: Scalable and Efficient Object Detection"** (Tan et al., 2020) : EfficientDet использует концепцию бифуркации уровней признаков (BiFPN) и масштабируемую архитектуру, что позволяет достичь высокой точности при меньшем количестве параметров. Модель масштабируется для различных потребностей, от мобильных устройств до серверных решений.

# **Методология**

Для выполнения задачи поиска объектов на видео были использованы следующие инструменты и библиотеки:

* **Python**: Основной язык программирования для написания кода.
* **TensorFlow**: Библиотека машинного обучения для построения и тренировки нейронных сетей.
* **Keras**: открытая библиотека, написанная на языке Python и обеспечивающая взаимодействие с искусственными нейронными сетями.

# **Шаги выполнения работы**

Алгоритм создания системы распознавания объектов будет состоять из 4 шагов:

1. Построение набора данных для обучения (Ручная разметка изображения)
2. Построение модели нейронной сети
3. Обучение нейронной сети
4. Тестирование сети

Разметка данных происходит в нескольких этапах. Первоначально, нужно подготовить изображения, на которых будет находиться объект, который, в дальнейшем, будет находить нейронная сеть. Далее, нужно произвести разметку изображений. Для решения задачи воспользуемся сервисом «**Supervisely**» (<https://app.supervisely.com/>). Сервис предоставляет инструмент, с помощью которого можно произвести разметку быстро и качественно. Полученные изображения, так называемые маски, являются монохромными, в которых пиксели объектов каждого класса имеют разные значения.

Перед началом работы необходимо через терминал установить следующие модули:

* tensorboard==2.17.0
* tensorboard-data-server==0.7.2
* tensorflow==2.17.0
* tensorflow-cpu==2.17.0
* tensorflow-intel==2.17.0
* tensorflow-io-gcs-filesystem==0.31.0
* tensorflow-object-detection-api==0.1.1
* Cython==3.0.10
* keras==3.4.1
* Keras-Applications==1.0.8
* keras-nightly==3.4.1.dev2024071303
* Keras-Preprocessing==1.1.2
* keras-unet==0.1.2
* numpy==1.26.3
* opencv-contrib-python==4.10.0.84
* opencv-python==4.6.0.66
* opencv-python-headless==4.10.0.84
* pillow==10.4.0
* protobuf==4.25.3
* pydot==2.0.0
* scikit-image==0.24.0
* tb-nightly==2.18.0a20240712
* tf-nightly==2.18.0.dev20240712
* tf-object-detection-util==0.1.7
* tf\_nightly\_intel==2.18.0.dev20240712
* utils==1.0.2
* urllib3==2.2.2

В самом начале объявим переменные, которые задают число классов. Это все объекты, присутствующие на сцене и класс для обозначения заднего фона. Задаем цвет, с помощью которого будет подсвечиваться объект на изображениях, задаем фиксированный размер входного изображения, которое будет использоваться в нейронной сети и размер выходного изображения, которое будет формироваться после финальной обработки данных.

|  |
| --- |
| images **=** **sorted(**glob**.**glob**(**'dataset/img/\*.jpg'**))**  masks **=** **sorted(**glob**.**glob**(**'dataset/masks\_machine/\*.png'**))**  CLASSES **=** 2  COLORS **=** **[**'black'**,** 'red'**]**  SAMPLE\_SIZE **=** **(**256**,** 256**)**  OUTPUT\_SIZE **=** **(**1280**,** 960**)** |

Далее, определим две функции. Первая из них - это load\_images. Она принимает на вход путь к изображению и путь к маске. С помощью специальных функций tensorflow загружаем этот файл и преобразуем его в необходимый формат, изменяем размер, нормализуем значения и аналогичным образом загружаем маску.

|  |
| --- |
| **def** load\_images**(**image**,** mask**):**  image **=** tf**.**io**.**read\_file**(**image**)**  image **=** tf**.**io**.**decode\_jpeg**(**image**)**  image **=** tf**.**image**.**resize**(**image**,** OUTPUT\_SIZE**)**  image **=** tf**.**image**.**convert\_image\_dtype**(**image**,** tf**.**float32**)**  image **=** image **/** 255.0  mask **=** tf**.**io**.**read\_file**(**mask**)**  mask **=** tf**.**io**.**decode\_png**(**mask**)**  mask **=** tf**.**image**.**resize**(**mask**,** OUTPUT\_SIZE**)**  mask **=** tf**.**image**.**convert\_image\_dtype**(**mask**,** tf**.**float32**)**    masks **=** **[]**    **for** i **in** **range(**CLASSES**):**  masks**.**append**(**tf**.**where**(**tf**.**equal**(**mask**,** **float(**i**)),** 1.0**,** 0.0**))**    masks **=** tf**.**stack**(**masks**,** axis**=**2**)**  masks **=** tf**.**reshape**(**masks**,** OUTPUT\_SIZE **+** **(**CLASSES**,))**  **return** image**,** masks |

Следующая функция называется augmentate\_images. Она принимает на вход трехканальные RGB-изображение и многоканальную маску, которую сформировали на предыдущем этапе и проводит с ними различные преобразования. Этот процесс называется аугментация данных или искусственным увеличением набора данных. Эти преобразования способствует увеличению вариативности выборки, что в конечном итоге положительно сказывается на качестве работы нейросетевой модели. Применяется функция central\_crop, которая извлекает центральный фрагмент со случайным значением масштаба и применяется случайное отражение изображение по горизонтали. В конце этого конвейера устанавливается выходной размер изображений и масок.

|  |
| --- |
| **def** augmentate\_images**(**image**,** masks**):**  random\_crop **=** tf**.**random**.**uniform**((),** 0.3**,** 1**)**  image **=** tf**.**image**.**central\_crop**(**image**,** random\_crop**)**  masks **=** tf**.**image**.**central\_crop**(**masks**,** random\_crop**)**    random\_flip **=** tf**.**random**.**uniform**((),** 0**,** 1**)**  **if** random\_flip **>=** 0.5**:**  image **=** tf**.**image**.**flip\_left\_right**(**image**)**  masks **=** tf**.**image**.**flip\_left\_right**(**masks**)**    image **=** tf**.**image**.**resize**(**image**,** SAMPLE\_SIZE**)**  masks **=** tf**.**image**.**resize**(**masks**,** SAMPLE\_SIZE**)**    **return** image**,** masks |

После того как функции обработки определены, загружаем данные. Главное - убедиться в том, чтобы изображению соответствовало нужная маска. С помощью функции tensorflow формируем наборы данных из изображений и масок, после чего объединяем их для параллельной обработки. Пользуясь функцией load\_images и загружая данные в память с помощью функции repeat, увеличиваем объем данных в 50 раз простым копированием, после чего ко всему объему данных применяем функцию augmentate\_images. Теперь каждое изображение в этом наборе будет уникально. На этом подготовка входного конвейера данных tensorflow завершена.

|  |
| --- |
| images\_dataset **=** tf**.**data**.**Dataset**.**from\_tensor\_slices**(**images**)**  masks\_dataset **=** tf**.**data**.**Dataset**.**from\_tensor\_slices**(**masks**)**  dataset **=** tf**.**data**.**Dataset**.zip((**images\_dataset**,** masks\_dataset**))**  dataset **=** dataset**.map(**load\_images**,** num\_parallel\_calls**=**tf**.**data**.**AUTOTUNE**)**  dataset **=** dataset**.**repeat**(**60**)**  dataset **=** dataset**.map(**augmentate\_images**,** num\_parallel\_calls**=**tf**.**data**.**AUTOTUNE**)**  # Разделим набор данных на обучающий и проверочный  train\_dataset **=** dataset**.**take**(**2000**).**cache**()**  test\_dataset **=** dataset**.**skip**(**2000**).**take**(**100**).**cache**()**    train\_dataset **=** train\_dataset**.**batch**(**16**)**  test\_dataset **=** test\_dataset**.**batch**(**16**)** |

Перейдем к реализации нейросетевой модели. Библиотека Keras, которая входит в состав tensorflow, содержит все необходимые для этого модули. Функция input\_layer задает входной слой нейронной сети и устанавливает размер входных данных. Функция downsample\_block описывает блоки, которые формируют энкодр. Она задает метод инициализации весовых коэффициентов, включает свёрточный слой, добавляет слой пакетной нормализации и устанавливает активационную функцию.

|  |
| --- |
| # Обозначим основные блоки модели  **def** input\_layer**():**  **return** tf**.**keras**.**layers**.**Input**(**shape**=**SAMPLE\_SIZE **+** **(**3**,))**  **def** downsample\_block**(**filters**,** size**,** batch\_norm**=True):**  initializer **=** tf**.**keras**.**initializers**.**GlorotNormal**()**  result **=** tf**.**keras**.**Sequential**()**    result**.**add**(**  tf**.**keras**.**layers**.**Conv2D**(**filters**,** size**,** strides**=**2**,** padding**=**'same'**,**  kernel\_initializer**=**initializer**,** use\_bias**=False))**  **if** batch\_norm**:**  result**.**add**(**tf**.**keras**.**layers**.**BatchNormalization**())**    result**.**add**(**tf**.**keras**.**layers**.**LeakyReLU**())**  **return** result |

Функция upsample\_block помогает формировать декодер нейронной сети, только вместо функции свёртки используется противоположное ей функция Transpose. Также в этом блоке предусмотрена возможность добавления dropout-слоев. Последняя функция output\_layer задает выходной слой, размерность которого соответствует количеству классов объектов на изображении и использует сигмоидную активационную функцию.

|  |
| --- |
| **def** upsample\_block**(**filters**,** size**,** dropout**=False):**  initializer **=** tf**.**keras**.**initializers**.**GlorotNormal**()**  result **=** tf**.**keras**.**Sequential**()**    result**.**add**(**  tf**.**keras**.**layers**.**Conv2DTranspose**(**filters**,** size**,** strides**=**2**,** padding**=**'same'**,**  kernel\_initializer**=**initializer**,** use\_bias**=False))**  result**.**add**(**tf**.**keras**.**layers**.**BatchNormalization**())**    **if** dropout**:**  result**.**add**(**tf**.**keras**.**layers**.**Dropout**(**0.25**))**    result**.**add**(**tf**.**keras**.**layers**.**ReLU**())**  **return** result  **def** output\_layer**(**size**):**  initializer **=** tf**.**keras**.**initializers**.**GlorotNormal**()**  **return** tf**.**keras**.**layers**.**Conv2DTranspose**(**CLASSES**,** size**,** strides**=**2**,** padding**=**'same'**,**  kernel\_initializer**=**initializer**,** activation**=**'sigmoid'**)** |

С помощью этих функций, создаем массив downsample\_stack, который представляет энкодер и upsample\_stack, который представляет декодер. Такая структура данных позволяет достаточно просто реализовать skip connections. С помощью этих двух циклов соединим блоки энкодера и декодера друг с другом и реализуем межслоевые соединения с помощью операции конкатенация. В результате создадим модель, в которой укажем входные и выходные слои.

|  |
| --- |
| inp\_layer **=** input\_layer**()**  downsample\_stack **=** **[**  downsample\_block**(**64**,** 4**,** batch\_norm**=False),**  downsample\_block**(**128**,** 4**),**  downsample\_block**(**256**,** 4**),**  downsample\_block**(**512**,** 4**),**  downsample\_block**(**512**,** 4**),**  downsample\_block**(**512**,** 4**),**  downsample\_block**(**512**,** 4**),**  **]**  upsample\_stack **=** **[**  upsample\_block**(**512**,** 4**,** dropout**=True),**  upsample\_block**(**512**,** 4**,** dropout**=True),**  upsample\_block**(**512**,** 4**,** dropout**=True),**  upsample\_block**(**256**,** 4**),**  upsample\_block**(**128**,** 4**),**  upsample\_block**(**64**,** 4**)**  **]**  out\_layer **=** output\_layer**(**4**)**  # Реализуем skip connections  x **=** inp\_layer  downsample\_skips **=** **[]**  **for** block **in** downsample\_stack**:**  x **=** block**(**x**)**  downsample\_skips**.**append**(**x**)**    downsample\_skips **=** **reversed(**downsample\_skips**[:-**1**])**  **for** up\_block**,** down\_block **in** **zip(**upsample\_stack**,** downsample\_skips**):**  x **=** up\_block**(**x**)**  x **=** tf**.**keras**.**layers**.**Concatenate**()([**x**,** down\_block**])**  out\_layer **=** out\_layer**(**x**)**  unet\_like **=** tf**.**keras**.**Model**(**inputs**=**inp\_layer**,** outputs**=**out\_layer**)** |

Определим функцию dice\_mc\_metric. Эта функция принимает два параметра: один из них - это ответ нейронной сети, а второй - это результат, который должен получиться на самом деле, то, чему старается научиться нейронная сеть. Метрика дает оценку точности результатов работы нейронной сети. С помощью функции unstack распаковываем многоканальные изображения маски и с помощью цикла находим среднее значение коэффициента dice во всём классе объектов, которые присутствуют на сцене. Функция потерь реализована в виде dice\_mc\_loss.

|  |
| --- |
| # Определим метрики и функции потерь  **def** dice\_mc\_metric**(**a**,** b**):**  a **=** tf**.**unstack**(**a**,** axis**=**3**)**  b **=** tf**.**unstack**(**b**,** axis**=**3**)**    dice\_summ **=** 0    **for** i**,** **(**aa**,** bb**)** **in** **enumerate(zip(**a**,** b**)):**  numenator **=** 2 **\*** tf**.**math**.**reduce\_sum**(**aa **\*** bb**)** **+** 1  denomerator **=** tf**.**math**.**reduce\_sum**(**aa **+** bb**)** **+** 1  dice\_summ **+=** numenator **/** denomerator    avg\_dice **=** dice\_summ **/** CLASSES    **return** avg\_dice  **def** dice\_mc\_loss**(**a**,** b**):**  **return** 1 **-** dice\_mc\_metric**(**a**,** b**)**  **def** dice\_bce\_mc\_loss**(**a**,** b**):**  **return** 0.3 **\*** dice\_mc\_loss**(**a**,** b**)** **+** tf**.**keras**.**losses**.**binary\_crossentropy**(**a**,** b**)** |

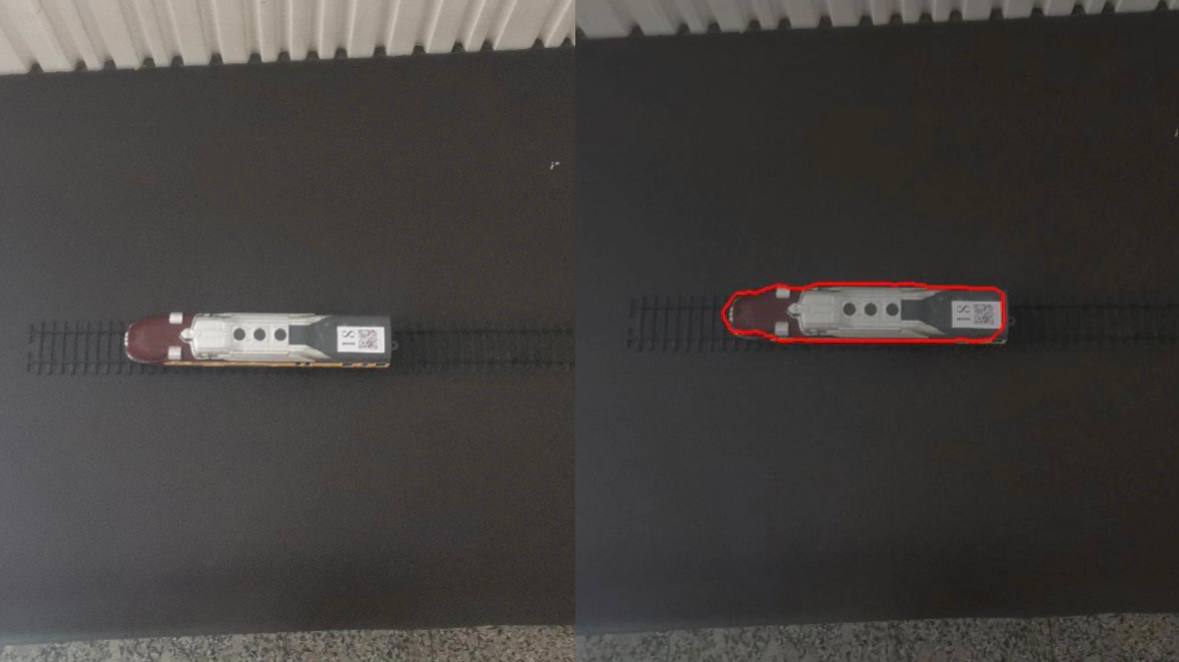
Когда все компоненты готовы, можно приступить к обучению нейронной сети. Компилируем модель с выбранными функциями потерь и метрикой. В качестве алгоритма будем использовать Adam. Затем запускаем обучение нейронной сети. 30 эпох обучения хватает для того, чтобы данная модель показывала приемлемый результат. После того, как обучение завершится, сохраняем модель.

|  |
| --- |
| # Компилируем модель  unet\_like**.compile(**optimizer**=**'adam'**,** loss**=[**dice\_bce\_mc\_loss**],** metrics**=[**dice\_mc\_metric**])**  # Обучаем нейронную сеть и сохраняем результат  history\_dice **=** unet\_like**.**fit**(**train\_dataset**,** validation\_data**=**test\_dataset**,** epochs**=**30**,** initial\_epoch**=**0**)**  unet\_like**.**save\_weights**(**'network/my\_model.weights.h5'**)**  # Загружаем обученную модель  unet\_like**.**load\_weights**(**'network/my\_model.weights.h5'**)** |

Полный код модели:

|  |
| --- |
| # Подключаем необходимые модули  **import** os  **import** glob  **import** numpy **as** np  **import** tensorflow **as** tf  **import** matplotlib**.**pyplot **as** plt  **import** sys  **from** skimage **import** measure  **from** skimage**.**io **import** imread**,** imsave  **from** skimage**.**transform **import** resize  **from** skimage**.**morphology **import** dilation**,** disk  **from** skimage**.**draw **import** polygon\_perimeter  **print(**f'Tensorflow version {tf**.**\_\_version\_\_}'**)**  **print(**f'GPU is {"ON" **if** tf**.**config**.**list\_physical\_devices**(**"GPU"**)** **else** "OFF" }'**)**  sys**.**setrecursionlimit**(**100000**)**  # Подготовим набор данных для обучения  images **=** **sorted(**glob**.**glob**(**'dataset/img/\*.jpg'**))**  masks **=** **sorted(**glob**.**glob**(**'dataset/masks\_machine/\*.png'**))**  CLASSES **=** 2  COLORS **=** **[**'black'**,** 'red'**]**  SAMPLE\_SIZE **=** **(**256**,** 256**)**  OUTPUT\_SIZE **=** **(**1280**,** 960**)**  **def** load\_images**(**image**,** mask**):**  image **=** tf**.**io**.**read\_file**(**image**)**  image **=** tf**.**io**.**decode\_jpeg**(**image**)**  image **=** tf**.**image**.**resize**(**image**,** OUTPUT\_SIZE**)**  image **=** tf**.**image**.**convert\_image\_dtype**(**image**,** tf**.**float32**)**  image **=** image **/** 255.0  mask **=** tf**.**io**.**read\_file**(**mask**)**  mask **=** tf**.**io**.**decode\_png**(**mask**)**  mask **=** tf**.**image**.**resize**(**mask**,** OUTPUT\_SIZE**)**  mask **=** tf**.**image**.**convert\_image\_dtype**(**mask**,** tf**.**float32**)**    masks **=** **[]**    **for** i **in** **range(**CLASSES**):**  masks**.**append**(**tf**.**where**(**tf**.**equal**(**mask**,** **float(**i**)),** 1.0**,** 0.0**))**    masks **=** tf**.**stack**(**masks**,** axis**=**2**)**  masks **=** tf**.**reshape**(**masks**,** OUTPUT\_SIZE **+** **(**CLASSES**,))**  **return** image**,** masks  **def** augmentate\_images**(**image**,** masks**):**  random\_crop **=** tf**.**random**.**uniform**((),** 0.3**,** 1**)**  image **=** tf**.**image**.**central\_crop**(**image**,** random\_crop**)**  masks **=** tf**.**image**.**central\_crop**(**masks**,** random\_crop**)**    random\_flip **=** tf**.**random**.**uniform**((),** 0**,** 1**)**  **if** random\_flip **>=** 0.5**:**  image **=** tf**.**image**.**flip\_left\_right**(**image**)**  masks **=** tf**.**image**.**flip\_left\_right**(**masks**)**    image **=** tf**.**image**.**resize**(**image**,** SAMPLE\_SIZE**)**  masks **=** tf**.**image**.**resize**(**masks**,** SAMPLE\_SIZE**)**    **return** image**,** masks  images\_dataset **=** tf**.**data**.**Dataset**.**from\_tensor\_slices**(**images**)**  masks\_dataset **=** tf**.**data**.**Dataset**.**from\_tensor\_slices**(**masks**)**  dataset **=** tf**.**data**.**Dataset**.zip((**images\_dataset**,** masks\_dataset**))**  dataset **=** dataset**.map(**load\_images**,** num\_parallel\_calls**=**tf**.**data**.**AUTOTUNE**)**  dataset **=** dataset**.**repeat**(**60**)**  dataset **=** dataset**.map(**augmentate\_images**,** num\_parallel\_calls**=**tf**.**data**.**AUTOTUNE**)**  # Разделим набор данных на обучающий и проверочный  train\_dataset **=** dataset**.**take**(**2000**).**cache**()**  test\_dataset **=** dataset**.**skip**(**2000**).**take**(**100**).**cache**()**    train\_dataset **=** train\_dataset**.**batch**(**16**)**  test\_dataset **=** test\_dataset**.**batch**(**16**)**  # Обозначим основные блоки модели  **def** input\_layer**():**  **return** tf**.**keras**.**layers**.**Input**(**shape**=**SAMPLE\_SIZE **+** **(**3**,))**  **def** downsample\_block**(**filters**,** size**,** batch\_norm**=True):**  initializer **=** tf**.**keras**.**initializers**.**GlorotNormal**()**  result **=** tf**.**keras**.**Sequential**()**    result**.**add**(**  tf**.**keras**.**layers**.**Conv2D**(**filters**,** size**,** strides**=**2**,** padding**=**'same'**,**  kernel\_initializer**=**initializer**,** use\_bias**=False))**  **if** batch\_norm**:**  result**.**add**(**tf**.**keras**.**layers**.**BatchNormalization**())**    result**.**add**(**tf**.**keras**.**layers**.**LeakyReLU**())**  **return** result  **def** upsample\_block**(**filters**,** size**,** dropout**=False):**  initializer **=** tf**.**keras**.**initializers**.**GlorotNormal**()**  result **=** tf**.**keras**.**Sequential**()**    result**.**add**(**  tf**.**keras**.**layers**.**Conv2DTranspose**(**filters**,** size**,** strides**=**2**,** padding**=**'same'**,**  kernel\_initializer**=**initializer**,** use\_bias**=False))**  result**.**add**(**tf**.**keras**.**layers**.**BatchNormalization**())**    **if** dropout**:**  result**.**add**(**tf**.**keras**.**layers**.**Dropout**(**0.25**))**    result**.**add**(**tf**.**keras**.**layers**.**ReLU**())**  **return** result  **def** output\_layer**(**size**):**  initializer **=** tf**.**keras**.**initializers**.**GlorotNormal**()**  **return** tf**.**keras**.**layers**.**Conv2DTranspose**(**CLASSES**,** size**,** strides**=**2**,** padding**=**'same'**,**  kernel\_initializer**=**initializer**,** activation**=**'sigmoid'**)**  # Построим U-NET подобную архитектуру  inp\_layer **=** input\_layer**()**  downsample\_stack **=** **[**  downsample\_block**(**64**,** 4**,** batch\_norm**=False),**  downsample\_block**(**128**,** 4**),**  downsample\_block**(**256**,** 4**),**  downsample\_block**(**512**,** 4**),**  downsample\_block**(**512**,** 4**),**  downsample\_block**(**512**,** 4**),**  downsample\_block**(**512**,** 4**),**  **]**  upsample\_stack **=** **[**  upsample\_block**(**512**,** 4**,** dropout**=True),**  upsample\_block**(**512**,** 4**,** dropout**=True),**  upsample\_block**(**512**,** 4**,** dropout**=True),**  upsample\_block**(**256**,** 4**),**  upsample\_block**(**128**,** 4**),**  upsample\_block**(**64**,** 4**)**  **]**  out\_layer **=** output\_layer**(**4**)**  # Реализуем skip connections  x **=** inp\_layer  downsample\_skips **=** **[]**  **for** block **in** downsample\_stack**:**  x **=** block**(**x**)**  downsample\_skips**.**append**(**x**)**    downsample\_skips **=** **reversed(**downsample\_skips**[:-**1**])**  **for** up\_block**,** down\_block **in** **zip(**upsample\_stack**,** downsample\_skips**):**  x **=** up\_block**(**x**)**  x **=** tf**.**keras**.**layers**.**Concatenate**()([**x**,** down\_block**])**  out\_layer **=** out\_layer**(**x**)**  unet\_like **=** tf**.**keras**.**Model**(**inputs**=**inp\_layer**,** outputs**=**out\_layer**)**  # Определим метрики и функции потерь  **def** dice\_mc\_metric**(**a**,** b**):**  a **=** tf**.**unstack**(**a**,** axis**=**3**)**  b **=** tf**.**unstack**(**b**,** axis**=**3**)**    dice\_summ **=** 0    **for** i**,** **(**aa**,** bb**)** **in** **enumerate(zip(**a**,** b**)):**  numenator **=** 2 **\*** tf**.**math**.**reduce\_sum**(**aa **\*** bb**)** **+** 1  denomerator **=** tf**.**math**.**reduce\_sum**(**aa **+** bb**)** **+** 1  dice\_summ **+=** numenator **/** denomerator    avg\_dice **=** dice\_summ **/** CLASSES    **return** avg\_dice  **def** dice\_mc\_loss**(**a**,** b**):**  **return** 1 **-** dice\_mc\_metric**(**a**,** b**)**  **def** dice\_bce\_mc\_loss**(**a**,** b**):**  **return** 0.3 **\*** dice\_mc\_loss**(**a**,** b**)** **+** tf**.**keras**.**losses**.**binary\_crossentropy**(**a**,** b**)**  # Компилируем модель  unet\_like**.compile(**optimizer**=**'adam'**,** loss**=[**dice\_bce\_mc\_loss**],** metrics**=[**dice\_mc\_metric**])**  # Обучаем нейронную сеть и сохраняем результат  history\_dice **=** unet\_like**.**fit**(**train\_dataset**,** validation\_data**=**test\_dataset**,** epochs**=**30**,** initial\_epoch**=**0**)**  unet\_like**.**save\_weights**(**'network/my\_model.weights.h5'**)**  # Загружаем обученную модель  unet\_like**.**load\_weights**(**'network/my\_model.weights.h5'**)**  # Работа обученной модели на изображениях  rgb\_colors **=** **[**  **(**0**,** 0**,** 0**),**  **(**255**,** 0**,** 0**),**  **(**0**,** 255**,** 0**),**  **(**0**,** 0**,** 255**),**  **(**255**,** 165**,** 0**),**  **(**255**,** 192**,** 203**),**  **(**0**,** 255**,** 255**),**  **(**255**,** 0**,** 255**)**  **]**  frames **=** **sorted(**glob**.**glob**(**'dataset/img/\*.jpg'**))**  **for** filename **in** frames**:**  frame **=** imread**(**filename**)**  sample **=** resize**(**frame**,** SAMPLE\_SIZE**)**    predict **=** unet\_like**.**predict**(**sample**.**reshape**((**1**,)** **+** SAMPLE\_SIZE **+** **(**3**,)))**  predict **=** predict**.**reshape**(**SAMPLE\_SIZE **+** **(**CLASSES**,))**    scale **=** frame**.**shape**[**0**]** **/** SAMPLE\_SIZE**[**0**],** frame**.**shape**[**1**]** **/** SAMPLE\_SIZE**[**1**]**    frame **=** **(**frame **/** 1.5**).**astype**(**np**.**uint8**)**    **for** channel **in** **range(**1**,** CLASSES**):**  contour\_overlay **=** np**.**zeros**((**frame**.**shape**[**0**],** frame**.**shape**[**1**]))**  contours **=** measure**.**find\_contours**(**np**.**array**(**predict**[:,:,**channel**]))**    **try:**  **for** contour **in** contours**:**  rr**,** cc **=** polygon\_perimeter**(**contour**[:,** 0**]** **\*** scale**[**0**],**  contour**[:,** 1**]** **\*** scale**[**1**],**  shape**=**contour\_overlay**.**shape**)**    contour\_overlay**[**rr**,** cc**]** **=** 1    contour\_overlay **=** dilation**(**contour\_overlay**,** disk**(**1**))**  frame**[**contour\_overlay **==** 1**]** **=** rgb\_colors**[**channel**]**  **except:**  **pass**  imsave**(**f'result/{os**.**path**.**basename**(**filename**)**}'**,** frame**)** |

# **Результаты**

В результате работы кода создается модель нейронной сети, которая способная находить и отображать на изображениях объект. Результаты работы нейронной сети представлены ниже:  
  

# **Заключение**

В данной практической работе был реализован и протестирован алгоритм поиска объектов на видео с использованием Python и TensorFlow. Модель продемонстрировала отличные результаты, обеспечивая баланс между скоростью и точностью детекции. Результаты исследования подтверждают, что современные методы машинного обучения и компьютерного зрения могут эффективно решать задачи детекции объектов в реальном времени.

# **Список литературы**

1. Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. arXiv preprint arXiv:1506.02640.
2. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. Y., & Berg, A. C. (2016). SSD: Single Shot MultiBox Detector. arXiv preprint arXiv:1512.02325.
3. Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. arXiv preprint arXiv:1506.01497.
4. Tan, M., Pang, R., & Le, Q. V. (2020). EfficientDet: Scalable and Efficient Object Detection. arXiv preprint arXiv:1911.09070.