Object Detection in Live YouTube Streams

Report

William Acuna Jonathan Agustin Alec Anderson

University of San Diego

1 Problem Definition

The problem we address in this project is the detection and classification of objects in live streaming video content from YouTube. With the growing popularity of live streaming, there's a significant need to analyze these streams effectively. Our focus is on processing these streams to detect various objects they contain. This capability is crucial for applications such as monitoring public spaces for safety, managing traffic flow, and moderating online content.

The necessity of solving this problem arises from the challenges posed by the vast and growing volume of live streaming content. Manual monitoring of these streams is impractical due to their sheer number and the real-time nature of the content. Automated object detection in these streams can provide valuable insights for decision-making in public safety, urban planning, and digital content management.

A computer vision algorithm, specifically the You Only Look Once (YOLO) algorithm, will address the core aspect of this problem: accurately identifying and classifying objects within the video frames of these live streams. The application of YOLO will enable us to process the video data efficiently, identify relevant objects, and categorize them, making the streams more actionable and informative. This project aims to demonstrate how computer vision can transform the way we interact with and manage live video data, making it a subject of both technical interest and practical significance.

2 Dataset

Our model used the COCO dataset, which is known for its broad application in object detection, segmentation, and captioning. This dataset is widely used in computer vision research due to its extensive collection of images and annotations across various object categories. COCO is structured into three subsets: Train2017 with 118K images, Val2017 with 5K images, and Test2017 with 20K images. These subsets are important for training, validating, and testing models, respectively.

The COCO dataset includes 80 object categories, ranging from everyday items to more specific objects. Each image in the dataset comes with annotations like bounding boxes and segmentation masks. For our project, we primarily used the Train2017 and Val2017 subsets to train and validate the YOLOv8 model. The COCO dataset's diverse range of annotated images was essential for developing an accurate and efficient object detection model. We managed dataset configurations through a YAML file, outlining essential details like dataset paths and class information, helping with efficient training and validation processes.

3 Exploratory Data Analysis (EDA)

We analyzed the COCO dataset to understand its structure. This dataset contains over 330,000 images and 1.5 million object instances. It includes 80 object categories and 91 stuff categories. We examined the distribution of these categories and the annotations for each image. This analysis aimed to identify the variety in the dataset and its implications for our object detection and segmentation goals.

4 Preprocessing

The preprocessing primarily involved resizing pictures to 640x640 images. Most of the preprocessing was already done through the dataset maintainers. To confirm, we ran terminal commands on the COCO dataset and searched for corrupted data. We created a `ModelManager` class for managing the dataset during model training. The class sets up TensorBoard for logging, managed model directories, and oversaw the training process. This overall approach transformed the dataset to be used effectively in our project.

5 Modeling Methods

Our object detection system was built around the YOLOv8 algorithm, the latest iteration in the YOLO series known for its object detection capabilities. YOLOv8's architecture, a convolutional neural network, is split into two primary parts: the backbone and the head. The backbone is based on a modified CSPDarknet53 architecture, featuring 53 convolutional layers enhanced with cross-stage partial connections for improved information flow. This structure supports the intricate processing required for correct object detection. The head of YOLOv8 comprises multiple convolutional layers followed by connected layers. These are pivotal in predicting bounding boxes, objectness scores, and class probabilities. A significant feature of YOLOv8's head is the integration of a self-attention mechanism. This addition lets the model selectively focus on different parts of an image, adjusting the importance of features based on their relevance to the detection task.

YOLOv8 excels in multi-scaled object detection. It uses a feature pyramid network to detect objects at various scales within an image. This capability is essential for accurately identifying both large and small objects, making YOLOv8 versatile for diverse object detection scenarios. YOLOv8's architecture supports multiple backbones like EfficientNet, ResNet, and CSPDarknet, offering flexibility in model selection based on specific needs. The customizability of YOLOv8's architecture is another strength, enabling changes to the model's structure and parameters to tailor it for various applications ranging from autonomous vehicles and surveillance to retail and medical imaging.

5.1 Training

The training code includes a `ModelManager` class, which encapsulates the functionality needed to manage the model's lifecycle, including downloading configuration files, setting up TensorBoard for logging, and training the model.

The `train_model` method within the `ModelManager` class is responsible for initiating the training process. It checks if a pre-trained model exists and, based on the `resume` flag, either resumes training or starts afresh using the specified model configuration. The training uses the COCO dataset, as indicated by the `data` argument pointing to "coco.yaml". The model is trained for a small number of epochs (3) with a batch size of 1 and an image size of 640 pixels. The `save_period` argument ensures that the model is saved after every epoch. If a CUDA-capable GPU is available, the model utilizes it to accelerate the training process.

Several important arguments in the script control various aspects of the training:

- `epochs`: The number of complete passes through the dataset.
- `batch`: The number of samples processed before the model is updated.
- `imgsz`: The size of the images that the model will process.
- `optimizer`: The optimization algorithm used for training.
- `device`: The device on which the model will be trained, e.g., a CUDA-enabled GPU.
- `project` and `name`: The directory and name for saving the trained model.
- `resume`: A flag indicating whether to resume training from the last checkpoint.
- `amp`: Automatic Mixed Precision for faster training on compatible hardware.
- `iou`: The Intersection Over Union threshold used for evaluating object detection performance.
- `max_det`: The maximum number of detections allowed per image.
- `plots`: Whether to generate plots during training, such as precision-recall curves.

6 Validation

In validating our model with Ultralytics YOLOv8's Val mode, we used metrics such as Precision, Recall, mAP50, and mAP50-95. This approach allowed us to assess the model's object detection accuracy and speed across different classes in the COCO dataset. The Val mode's automatic retention of training settings simplified the process, ensuring consistent testing conditions.

The validation provided class-wise performance data, visual charts like F1 Score and Precision-Recall Curves, and confusion matrices. These outputs helped us understand the model's strengths and weaknesses, guiding adjustments in hyperparameters. We stored all results for ongoing analysis and model refinement.

7 Performance Metrics Analysis

We evaluated the accuracy and efficiency of our object detection model using several metrics. These metrics provide insights into the model's ability to identify and localize objects and its handling of false positives and negatives.

7.1 Key Metrics

- **Intersection over Union (IoU)**: Measures the overlap between predicted and ground truth bounding boxes, indicating localization accuracy.
- **Average Precision (AP)**: Reflects the model's precision and recall, calculated as the area under the precision-recall curve.

- **Mean Average Precision (mAP)**: Averages the AP values across multiple object classes, important for models detecting various classes.
- **Precision and Recall**: Precision assesses the model's ability to minimize false positives, while recall measures its ability to detect all instances of a class.
- **F1 Score**: Balances precision and recall, important for models where both false positives and negatives are critical.

7.2 Calculating and Interpreting Metrics for YOLOv8

To compute these metrics, we used YOLOv8's validation mode. This process involves processing a validation dataset and returning various performance metrics.

7.2.1 Model Validation Outputs

- **Class-wise Metrics**: Includes precision, recall, and mAP for each class, providing a detailed view of model performance per category.
- **Speed Metrics**: Analyzes the time taken for different stages of validation, crucial in real-time detection scenarios.
- **COCO Metrics**: Additional precision and recall metrics for the COCO dataset, giving insights into performance across different object sizes and IoU thresholds.
- **Visual Outputs**: Includes F1 score curve, precision-recall curve, precision and recall curves, confusion matrices, and validation batch images. These visuals aid in understanding the model's detection and classification accuracy.

7.2.2 Results Interpretation

- **Low mAP** might suggest a need for model refinement.
- **Low IoU** indicates poor object localization, possibly requiring different bounding box methods.
- **Low Precision** implies excessive false detections, which can be reduced by adjusting confidence thresholds.
- **Low Recall** indicates missed real objects, potentially improvable with better feature extraction or more data.
- **Class-specific AP scores** can highlight specific classes where the model underperforms.

8 Model Results

During validation, our model's capabilities were put to the test across three distinct environments, each presenting unique challenges for object detection.

8.1 Diverse Environments and Object Interactions

The first environment captured a varied urban setting with objects like trains, handbags, and even animals like elephants and zebras. Despite the complexity, the model showed robust detection capabilities. However, the precision varied among objects with simpler shapes and distinct colors being identified more accurately than others, as shown in Figure 1.

Figure 1: Object detection in a diverse urban setting with a range of static and dynamic objects.

The second environment displayed a mix of natural and artificial elements, from beaches with surfboards to trains and airplanes. The model effectively distinguished between stationary and moving objects, showing versatility. Notably, it was able to detect smaller objects like kites, suggesting good performance on objects with less area, as depicted in Figure 2.

Figure 2: Object detection capturing a mix of natural and artificial elements.

The third environment was particularly challenging, featuring objects in motion such as airplanes, cars, and various animals. The model demonstrated a strong ability to track movement and offer clear detections, which is crucial for potential applications like traffic monitoring and wildlife surveillance, as illustrated in Figure 3.

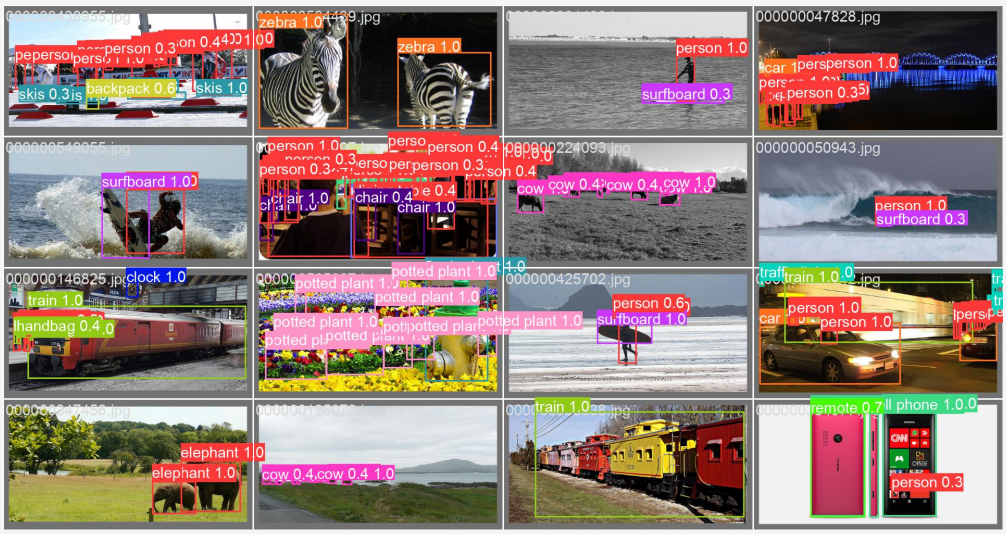

Figure 3: Object detection in dynamic scenes with moving objects and animals.

8.2 Graphs

During the evaluation of our model, we relied on various performance curves to understand its predictive capabilities and to identify areas for improvement. Below we discuss the significance of the Precision-Recall, Precision-Confidence, F1-Confidence, and Recall-Confidence curves generated during our testing phase. These curves collectively provide a comprehensive view of the model's strengths and weaknesses in terms of its prediction capabilities. They are instrumental in fine-tuning the model to achieve the best performance possible on unseen data.

8.3 Precision-Recall Curve

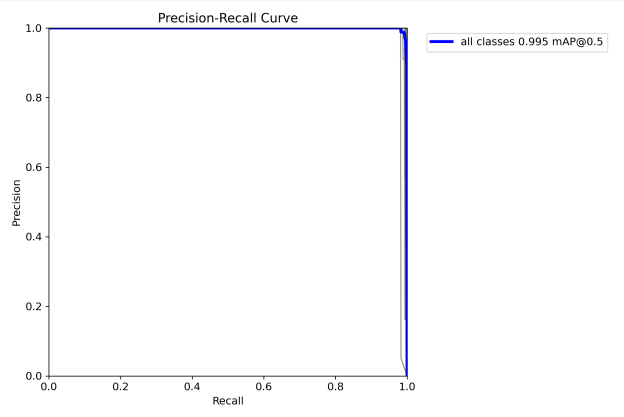

Figure 4: Precision-Recall Curve showing model performance at different thresholds.

The Precision-Recall (PR) Curve in Figure 4 is a plot that shows the trade-off between precision and recall for different probability thresholds. A high area under the curve (AUC) represents both high recall and high precision, where high precision relates to a low false positive rate, and high recall relates to a low false negative rate. Our model achieved an impressive mAP@0.5, indicating that it can detect objects with high accuracy while maintaining a low rate of false positives.

8.4 Precision-Confidence Curve

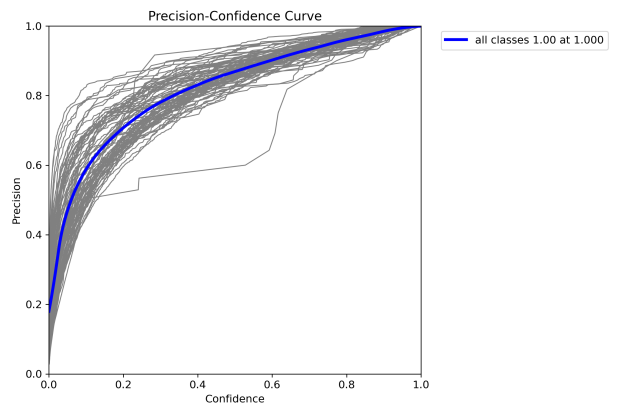

Figure 5: Precision-Confidence Curve illustrating the relationship between precision and model confidence.

The Precision-Confidence Curve in Figure 5 indicates the model's precision at various confidence levels. A high precision across confidence levels suggests that when the model predicts an object class, it does so with a high degree of certainty. This curve is particularly useful for setting a confidence threshold that balances the number of detections with their quality.

8.5 F1-Confidence Curve

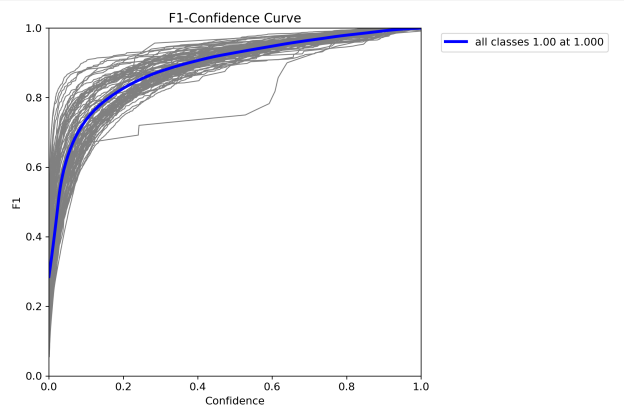

Figure 6: F1-Confidence Curve showing the F1 Score across varying levels of confidence.

The F1-Confidence Curve in Figure 6 provides insights into the balance between precision and recall across different confidence thresholds. The F1 score is the harmonic mean of precision and recall, and a higher score indicates a better balance. The model demonstrates a robust performance with an optimal balance at a specific confidence threshold, as indicated by the peak of the curve.

8.6 Recall-Confidence Curve

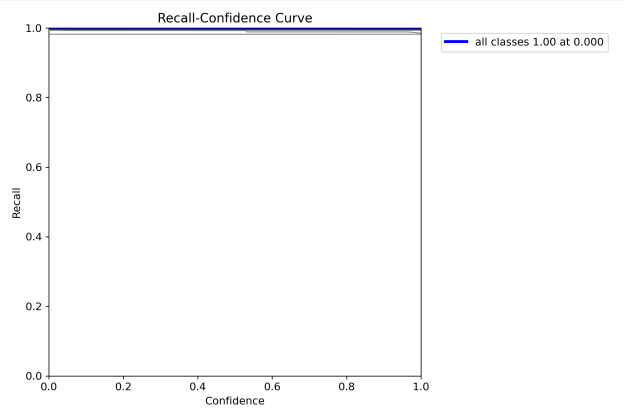

Figure 7: Recall-Confidence Curve depicting how recall changes with varying confidence thresholds.

The Recall-Confidence Curve in Figure 7 reveals the model's ability to find all the relevant cases (recall) across different confidence levels. A high recall at a certain threshold indicates that the model can detect the majority of objects without missing many.

9 Findings

9.1 Deployed Examples

Example 1: California Beach at Sunset We tested our model on a live stream from a California beach at sunset. The model effectively identified bicycles and people, demonstrating good performance even in challenging lighting. Accurate bounding boxes were generated, with a color-coded key for clarity. However, it misclassified two lifeguard towers as boats, indicating a need for improvement in object differentiation.

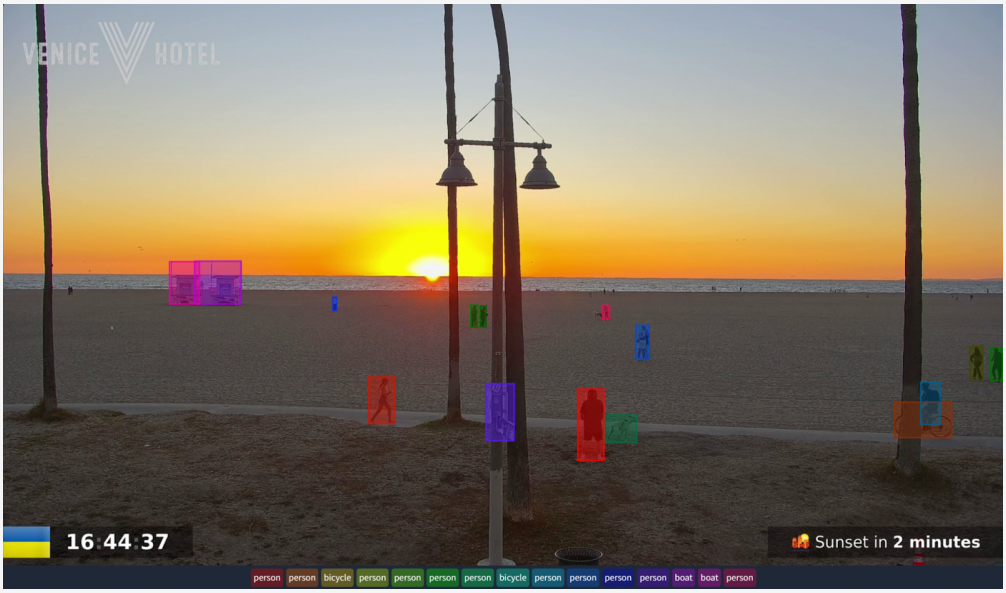

Figure 8: Object detection on a California beach at sunset.

Example 2: Overcast Conditions in Brazil In Brazil, under overcast conditions, the model showed proficiency in classifying chairs and umbrellas, even with overlapping objects. It detected almost every person present, missing only one individual in the background. This test underscores the model's capability in handling complex scenes with multiple objects.

Figure 9: Object detection on an overcast beach in Brazil.

Example 3: Crowded Bar Scene The crowded bar scene test was critical for assessing the model in dense areas. The model successfully identified people and objects like handbags and bottles. However, it faced challenges with overlapping bounding boxes and false positives, highlighting areas for refinement in

crowded environments.

Figure 10: Object detection in a crowded bar scene.

9.2 Challenges and Insights

During development, we shifted from Streamlit to Gradio for a more interactive interface. The tests, particularly in the crowded bar scene, revealed precision issues. Despite these challenges, the model's resilience to varying lighting conditions was a positive outcome. Inaccuracies in object localization in dense scenes indicate the need for improved distinction between closely situated objects.

9.3 Potential Applications

The system shows promise for surveillance security and traffic management. In security, it can detect threats or unauthorized activities, such as unattended luggage in airports. For urban planning, it can analyze traffic patterns and pedestrian flows, aiding in smarter traffic control system designs. Additionally, the project is beneficial for creating labeled datasets for diverse environments, contributing to the research community.

9.4 Future Improvements

Future enhancements will focus on increasing accuracy and real-time processing. This involves refining bounding box precision, enhancing object classification, and reducing false positives in dense environments. Improving real-time processing is crucial for timely decision-making and responsiveness in deployment.

10 Conclusion

Our tests across different scenarios demonstrated the adaptability and practicality of our object detection system, particularly with the 'yolov8x' model. The system's ability to handle diverse lighting conditions was a significant achievement. Future improvements will concentrate on object localization and classification accuracy in crowded scenes and enhancing real-time processing efficiency. These enhancements are pivotal for advancing the system's utility across various sectors.

References

Jocher, G., Chaurasia, A., & Qiu, J. (2023). *YOLO by Ultralytics* (Version 8.0.0). https://github.com/ultralytics/ultralytics

Lin, T.-Y., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C. L., & Dollár, P. (2015). Microsoft coco: Common objects in context.

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection.

11 Appendix

11.1 Project Artifacts

- Deployed Model Application:
  - https://huggingface.co/spaces/aai521-group6/youtube-object-detection
- Hugging Face Model
  - https://huggingface.co/aai521-group6/yolov8x-coco
- GitHub Repository
  - https://github.com/aai521-group6/project

11.2 Project Code

```
"""
This module integrates real-time object detection into live YouTube streams
using the YOLO (You Only Look Once) model, and provides an interactive user
interface through Gradio. It is designed to allow users to search for live
YouTube streams and apply object detection to these streams in real time.

Main Features:
- Search for live YouTube streams using specific queries.
- Retrieve live stream URLs using the Streamlink library.
- Perform real-time object detection on live streams using the YOLO model.
- Display the live stream and object detection results through a Gradio
interface.

The module comprises several key components:
- `SearchFilter`: An enumeration for YouTube search filters.
- `SearchService`: A service class to search for YouTube videos and retrieve
live stream URLs.
- `LiveYouTubeObjectDetector`: The main class integrating the YOLO model and
Gradio UI, handling the entire workflow of searching, streaming, and object
detection.

Dependencies:
- cv2 (OpenCV): Used for image processing tasks.
- Gradio: Provides the interactive web-based user interface.
- innertube, streamlink: Used for interacting with YouTube and retrieving
live stream data.
- numpy: Utilized for numerical operations on image data.
- PIL (Pillow): A Python Imaging Library for opening, manipulating, and
saving images.
- ultralytics YOLO: The YOLO model implementation for object detection.

Usage:
Run this file to launch the Gradio interface, which allows users to input
search queries for YouTube live streams, select a stream, and perform object
detection on the selected live stream.

"""
import logging
import os
import subprocess
import sys
from enum import Enum
from typing import Any, Dict, List, Optional, Tuple

import requests
```

```
def install_requirements():
    requirements_url =
"https://raw.githubusercontent.com/aai521-group6/project/main/requirements.txt"
    response = requests.get(requirements_url)
    if response.status_code == 200:
        with open("requirements.txt", "wb") as file:
            file.write(response.content)
        subprocess.check_call([sys.executable, "-m", "pip", "install", "-r",
"requirements.txt"])
    else:
        raise Exception("Failed to download requirements.txt")


try:
    import cv2
    import gradio as gr
    import innertube
    import numpy as np
    import streamlink
    from PIL import Image
    from ultralytics import YOLO
except ImportError:
    install_requirements()
    import cv2
    import gradio as gr
    import innertube
    import numpy as np
    import streamlink
    from PIL import Image
    from ultralytics import YOLO


logging.basicConfig(stream=sys.stderr, level=logging.DEBUG)


class SearchFilter(Enum):
    """
    An enumeration for specifying different types of YouTube search filters.

    This Enum class is used to define filters for categorizing YouTube
search
    results into either live or regular video content. It is utilized in
    conjunction with the `SearchService` class to refine YouTube searches
    based on the type of content being sought.
```

```
    Attributes:
        LIVE (str): Represents the filter code for live video content on
YouTube.
        VIDEO (str): Represents the filter code for regular, non-live video
content on YouTube.

    Each attribute consists of a tuple where the first element is the filter
code
    used in YouTube search queries, and the second element is a
human-readable
    string describing the filter.
    """

    LIVE = ("EgJAAQ%3D%3D", "Live")
    VIDEO = ("EgIQAQ%3D%3D", "Video")

    def __init__(self, code, human_readable):
        """Initializes the SearchFilter with a code and a human-readable
        string.

        :param code: The filter code used in YouTube search queries.
        :type code: str
        :param human_readable: A human-readable representation of the
filter.
        :type human_readable: str
        """
        self.code = code
        self.human_readable = human_readable

    def __str__(self):
        """Returns the human-readable representation of the filter.

        :return: The human-readable representation of the filter.
        :rtype: str
        """
        return self.human_readable


class SearchService:
    """
    SearchService provides functionality to search for YouTube videos using
the
    InnerTube API and retrieve live stream URLs using the Streamlink
library.
```

```
    This service allows filtering search results to either live or regular
video
    content and parsing the search response to extract relevant video
information.
    It also constructs YouTube URLs for given video IDs and retrieves the
best
    available stream URL for live YouTube videos.

    Methods:
        search: Searches YouTube for videos matching a query and filter.
        parse: Parses raw search response data into a list of video details.
        _search: Performs a YouTube search with the given query and filter.
        get_youtube_url: Constructs a YouTube URL for a given video ID.
        get_stream: Retrieves the stream URL for a given YouTube video URL.
    """

    @staticmethod
    def search(query: Optional[str], filter: SearchFilter =
    SearchFilter.VIDEO):
        """Searches YouTube for videos matching the given query and filter.

        :param query: The search query.
        :type query: Optional[str]
        :param filter: The search filter to apply.
        :type filter: SearchFilter
        :return: A list of search results, each a dictionary with video
details.
        :rtype: List[Dict[str, Any]]
        """
        client = innertube.InnerTube("WEB", "2.20230920.00.00")
        response = SearchService._search(query, filter)
        results = SearchService.parse(response)
        return results

    @staticmethod
    def parse(data: Dict[str, Any]) -> List[Dict[str, str]]:
        """Parses the raw search response data into a list of video details.

        :param data: The raw search response data from YouTube.
        :type data: Dict[str, Any]
        :return: A list of parsed video details.
        :rtype: List[Dict[str, str]]
        """
        results = []
        contents = data["contents"]["twoColumnSearchResultsRenderer"][
"primaryContents"]["sectionListRenderer"]["contents"]
```

```
        items = contents[0]["itemSectionRenderer"]["contents"] if contents
else []
        for item in items:
            if "videoRenderer" in item:
                renderer = item["videoRenderer"]
                results.append(
                    {
                        "video_id": renderer["videoId"],
                        "thumbnail_url":
                        renderer["thumbnail"]["thumbnails"][-1]["url"],
                        "title": "".join(run["text"] for run in
                        renderer["title"]["runs"]),
                    }
                )
        return results

    @staticmethod
    def _search(query: Optional[str] = None, filter: SearchFilter =
    SearchFilter.VIDEO) -> Dict[str, Any]:
        """Performs a YouTube search with the given query and filter.

        :param query: The search query.
        :type query: Optional[str]
        :param filter: The search filter to apply.
        :type filter: SearchFilter
        :return: The raw search response data from YouTube.
        :rtype: Dict[str, Any]
        """
        client = innertube.InnerTube("WEB", "2.20230920.00.00")
        response = client.search(query=query, params=filter.code if filter
else None)
        return response

    @staticmethod
    def get_youtube_url(video_id: str) -> str:
        """Constructs a YouTube URL for the given video ID.

        :param video_id: The ID of the YouTube video.
        :type video_id: str
        :return: The YouTube URL for the video.
        :rtype: str
        """
        return f"https://www.youtube.com/watch?v={video_id}"

    @staticmethod
    def get_stream(youtube_url: str) -> Optional[str]:
```

```
        """Retrieves the stream URL for a given YouTube video URL.

        :param youtube_url: The URL of the YouTube video.
        :type youtube_url: str
        :return: The stream URL if available, otherwise None.
        :rtype: Optional[str]
        """
        try:
            session = streamlink.Streamlink()
            streams = session.streams(youtube_url)
            if streams:
                best_stream = streams.get("best")
                return best_stream.url if best_stream else None
            else:
                gr.Warning(f"No streams found for: {youtube_url}")
                return None
        except Exception as e:
            gr.Error(f"An error occurred while getting stream: {e}")
            logging.warning(f"An error occurred: {e}")
            return None


INITIAL_STREAMS = SearchService.search("world live cams", SearchFilter.LIVE)


class LiveYouTubeObjectDetector:
    """
    LiveYouTubeObjectDetector is a class that integrates object detection
into live YouTube streams.
    It uses the YOLO (You Only Look Once) model to detect objects in video
frames captured from live streams.
    The class also provides a Gradio interface for users to interact with
the object detection system,
    allowing them to search for live streams, view them, and detect objects
in real-time.

    The class handles the retrieval of live stream URLs, frame capture from
the streams, object detection
    on the frames, and updating the Gradio interface with the results.

    Attributes:
        model (YOLO): The YOLO model used for object detection.
        streams (list): A list of dictionaries containing information about
the current live streams.
        gallery (gr.Gallery): A Gradio gallery widget to display live stream
thumbnails.
```

```
        search_input (gr.Textbox): A Gradio textbox for inputting search
queries.
        stream_input (gr.Textbox): A Gradio textbox for inputting a specific
live stream URL.
        annotated_image (gr.AnnotatedImage): A Gradio annotated image widget
to display detection results.
        search_button (gr.Button): A Gradio button to initiate a new search
for live streams.
        submit_button (gr.Button): A Gradio button to start object detection
on a specified live stream.
        page_title (gr.HTML): A Gradio HTML widget to display the page
title.

    Methods:
        detect_objects: Detects objects in a live YouTube stream given its
URL.
        get_frame: Captures a frame from a live stream URL.
        annotate: Annotates a frame with detected objects.
        create_black_image: Creates a black placeholder image.
        get_live_streams: Searches for live streams based on a query.
        render: Sets up and launches the Gradio interface.
    """

    def __init__(self):
        """Initializes the LiveYouTubeObjectDetector with YOLO model and UI
        components."""
        logging.getLogger().setLevel(logging.DEBUG)
        model_url =
"https://huggingface.co/aai521-group6/yolov8x-coco/resolve/main/yolov8x-coco.pt?download=tru
        local_model_path = "yolov8x-coco.pt"
        response = requests.get(model_url)
        if response.status_code == 200:
            with open(local_model_path, 'wb') as f:
                f.write(response.content)
            print("Model downloaded successfully.")
        else:
            raise Exception(f"Failed to download model: Status code
            {response.status_code}")
        self.model = YOLO(local_model_path)
        self.streams = INITIAL_STREAMS

        # Gradio UI
        initial_gallery_items = [(stream["thumbnail_url"], stream["title"])
for stream in self.streams]
```

```
        self.gallery = gr.Gallery(label="Live YouTube Videos",
        value=initial_gallery_items, show_label=True, columns=[4], rows=[5],
        object_fit="contain", height="auto", allow_preview=False)
        self.search_input = gr.Textbox(label="Search Live YouTube Videos")
        self.stream_input = gr.Textbox(label="URL of Live YouTube Video")
        self.annotated_image = gr.AnnotatedImage(show_label=False)
        self.search_button = gr.Button("Search", size="lg")
        self.submit_button = gr.Button("Detect Objects", variant="primary",
        size="lg")
        self.page_title = gr.HTML("<center><h1><b>Object Detection in Live
        YouTube Streams</b></h1></center>")

    def detect_objects(self, url: str) -> Tuple[Image.Image,
    List[Tuple[Tuple[int, int, int, int], str]]]:
        """
        Detects objects in the given live YouTube stream URL.

        :param url: The URL of the live YouTube video.
        :type url: str
        :return: A tuple containing the annotated image and a list of
annotations.
        :rtype: Tuple[Image.Image, List[Tuple[Tuple[int, int, int, int],
str]]]
        """
        stream_url = SearchService.get_stream(url)
        if not stream_url:
            gr.Error(f"Unable to find a stream for: {stream_url}")
            return self.create_black_image(), []
        frame = self.get_frame(stream_url)
        if frame is None:
            gr.Error(f"Unable to capture frame for: {stream_url}")
            return self.create_black_image(), []
        return self.annotate(frame)

    def get_frame(self, stream_url: str) -> Optional[np.ndarray]:
        """
        Captures a frame from the given live stream URL.

        :param stream_url: The URL of the live stream.
        :type stream_url: str
        :return: The captured frame as a numpy array, or None if capture
fails.
        :rtype: Optional[np.ndarray]
        """
        if not stream_url:
            return None
```

```
        try:
            cap = cv2.VideoCapture(stream_url)
            ret, frame = cap.read()
            cap.release()
            if ret:
                return cv2.resize(frame, (1920, 1080))
            else:
                logging.warning("Unable to process the HLS stream with
cv2.VideoCapture.")
                return None
        except Exception as e:
            logging.warning(f"An error occurred while capturing the frame:
{e}")
            return None

    def annotate(self, frame: np.ndarray) -> Tuple[Image.Image,
    List[Tuple[Tuple[int, int, int, int], str]]]:
        """
        Annotates the given frame with detected objects and their bounding
boxes.

        :param frame: The frame to be annotated.
        :type frame: np.ndarray
        :return: A tuple of the annotated PIL image and list of annotations.
        :rtype: Tuple[Image.Image, List[Tuple[Tuple[int, int, int, int],
str]]]
        """
        frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        predictions = self.model.predict(frame_rgb)
        annotations = []

        for result in predictions._images_prediction_lst[0]:
            class_names = result.class_names
            for bbox, label in zip(result.prediction.bboxes_xyxy,
            result.prediction.labels):
                x1, y1, x2, y2 = bbox
                class_name = class_names[int(label)]
                bbox_coords = (int(x1), int(y1), int(x2), int(y2))
                annotations.append((bbox_coords, class_name))

        return Image.fromarray(frame_rgb), annotations

    @staticmethod
    def create_black_image():
        """
        Creates a black image of fixed dimensions.
```

```
        This method generates a black image that can be used as a
placeholder or background.
        It is particularly useful in cases where no valid frame is available
for processing.

        :return: A black image as a numpy array.
        :rtype: np.ndarray
        """
        black_image = np.zeros((1080, 1920, 3), dtype=np.uint8)
        pil_black_image = Image.fromarray(black_image)
        cv2_black_image = cv2.cvtColor(np.array(pil_black_image),
cv2.COLOR_RGB2BGR)
        return cv2_black_image

    @staticmethod
    def get_live_streams(query=""):
        """
        Searches for live streams on YouTube based on the given query.

        This method utilizes the SearchService to find live YouTube streams.
If no query is
        provided, it defaults to searching for 'world live cams'.

        :param query: The search query for live streams, defaults to an
empty string.
        :type query: str, optional
        :return: A list of dictionaries containing information about each
live stream.
        :rtype: List[Dict[str, str]]
        """
        return SearchService.search(query if query else "world live cams",
        SearchFilter.LIVE)

    def render(self):
        """
        Sets up and launches the Gradio interface for the application.

        This method creates the Gradio UI elements and defines the behavior
of the application.
        It includes the setup of interactive widgets like galleries,
textboxes, and buttons,
        and defines the actions triggered by user interactions with these
widgets.
```

```
        The Gradio interface allows users to search for live YouTube
streams, select a stream,
        and run object detection on the selected live stream.
        """
        with gr.Blocks(title="Object Detection in Live YouTube Streams",
        css="footer {visibility: hidden}", analytics_enabled=False) as app:
            self.page_title.render()
            with gr.Column():
                with gr.Group():
                    with gr.Row():
                        self.stream_input.render()
                        self.submit_button.render()
                self.annotated_image.render()
            with gr.Group():
                with gr.Row():
                    self.search_input.render()
                    self.search_button.render()
            with gr.Row():
                self.gallery.render()

            @self.gallery.select(inputs=None, outputs=[self.annotated_image,
            self.stream_input], scroll_to_output=True)
            def detect_objects_from_gallery_item(evt: gr.SelectData):
                if evt.index is not None and evt.index < len(self.streams):
                    selected_stream = self.streams[evt.index]
                    stream_url =
SearchService.get_youtube_url(selected_stream["video_id"])
                    frame_output = self.detect_objects(stream_url)
                    return frame_output, stream_url
                return None, ""

            @self.search_button.click(inputs=[self.search_input],
            outputs=[self.gallery])
            def search_live_streams(query):
                self.streams = self.get_live_streams(query)
                gallery_items = [(stream["thumbnail_url"], stream["title"])
for stream in self.streams]
                return gallery_items

            @self.submit_button.click(inputs=[self.stream_input],
            outputs=[self.annotated_image])
            def detect_objects_from_url(url):
                return self.detect_objects(url)

        return app.queue().launch(show_api=False)
```

```
if __name__ == "__main__":
    LiveYouTubeObjectDetector().render()


%reload_ext tensorboard
%tensorboard --logdir ultralytics/runs


# TRAINING
import logging
import os
import subprocess
import sys

import requests
import torch
from torch.cuda import is_available as cuda_is_available
from torch.utils.tensorboard import SummaryWriter


def install_requirements():
    requirements_url =
"https://raw.githubusercontent.com/aai521-group6/project/main/requirements.txt"
    response = requests.get(requirements_url)
    if response.status_code == 200:
        with open("requirements.txt", "wb") as file:
            file.write(response.content)
        subprocess.check_call([sys.executable, "-m", "pip", "install", "-r",
"requirements.txt"])
    else:
        raise Exception("Failed to download requirements.txt")


try:
    from ultralytics import YOLO
except ImportError:
    install_requirements()
    from ultralytics import YOLO

logging.basicConfig(level=logging.DEBUG)


class ModelManager:
    def __init__(self, model_dir="models", model_config="yolov8x.yaml",
    tensorboard_port=6006):
        self.setup_tensorboard()
```

```
        self.model_dir = model_dir
        self.model_config = model_config
        self.model_name = "yolov8x-coco.pt"
        self.model_path = os.path.join(self.model_dir, self.model_name)
        self.tensorboard_port = tensorboard_port
        self.ensure_model_dir_exists()
        self.download_yaml()
        self.tensorboard_logger = None

    def ensure_model_dir_exists(self):
        if not os.path.exists(self.model_dir):
            os.makedirs(self.model_dir)

    def download_yaml(self):
        yaml_file ="coco.yaml"
        if not os.path.exists(yaml_file):
            try:
                response = requests.get(
"https://raw.githubusercontent.com/ultralytics/ultralytics/main/ultralytics/cfg/datasets/coc
)
                response.raise_for_status()
                with open(yaml_file, "wb") as file:
                    file.write(response.content)
            except Exception as e:
                raise Exception(f"Failed to download {yaml_file}: {e}")

    def setup_tensorboard(self):
        self.tensorboard_logger = SummaryWriter("ultralytics/runs")

    def train_model(self, resume=False):
        self.ensure_model_dir_exists()
        if resume and os.path.exists(self.model_path):
            model = YOLO(self.model_path)
        else:
            model = YOLO(self.model_config)

        if cuda_is_available():
            model.cuda()

        model.train(
            data="coco.yaml",
            epochs=3,
            batch=1,
            imgsz=640,
            save_period=1,
            project=self.model_dir,
```

```
            name=self.model_name.replace(".pt", ""),
            exist_ok=True,
            optimizer='auto',
            verbose=True,
            plots=True,
            val=True,
            resume=resume
        )
        torch.save(model.state_dict(), self.model_path)

    def get_model(self):
        if not os.path.exists(self.model_path):
            self.train_model()
        return self.model_path

    def stop_tensorboard(self):
        self.tensorboard_logger.close()


ModelManager().get_model()


# VALIDATION
%pip install ultralytics --quiet --progress-bar off
import os
import requests
from ultralytics import YOLO

yaml_file ="coco.yaml"
if not os.path.exists(yaml_file):
    try:
        response = requests.get(
"https://raw.githubusercontent.com/ultralytics/ultralytics/main/ultralytics/cfg/datasets/coc
)
        response.raise_for_status()
        with open(yaml_file, "wb") as file:
            file.write(response.content)
    except Exception as e:
        raise Exception(f"Failed to download {yaml_file}: {e}")

model_url =
"https://huggingface.co/aai521-group6/yolov8x-coco/resolve/main/yolov8x-coco.pt?download=tru
local_model_path = "yolov8x-coco.pt"
response = requests.get(model_url)
if response.status_code == 200:
    with open(local_model_path, 'wb') as f:
```

```
        f.write(response.content)
    print("Model downloaded successfully.")
else:
    raise Exception(f"Failed to download model: Status code
    {response.status_code}")
model = YOLO(local_model_path)
metrics = model.val(
    data="coco.yaml",
    save_json=True,
    save_hybrid=True,
    plots=True
)
metrics.box.map
```