```python
"""
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
```