Final Project Report: Multi and Visual Object Tracking

1st Nicholas Farinacci
University at Albany
School Email: nfarinacci@albany.edu
Personal Email: nickfarinacci@yahoo.com
New York City, NY, United States

2nd Zeyuan Chu
University at Albany
School Email: zchu2@albany.edu
Personal Email:chuzeyuan@gmail.com
Albany, NY, United States

***Abstract*—This report outlines our project on Multi-Visual Object Tracking. We introduce the problem in our Introduction, then discuss related works or previous techniques that have addressed this topic in our Related / Previous Work section. Afterwards, we define our Method Ideas and our Implemented Solution to then understand our Experimental Results. We then disuses our results and what we learned in our Discussion / What We Learned Section before concluding in our Conclusion.**

***Index Terms*—Visual Object Tracking, Multi-Object Tracking, Multi-Visual Object Tracking, MOT, VOT, Detection, Tracking, YOLO, SORT, COCO.**

I. Introduction

The aim of this project and report is to demonstrate the developed Multi-Visual Object Tracking System that is capable of detecting and tracking multiple targets in video streams, either locally or externally on GPU through Colab or another notebook, with the potential for use in real-time applications if hardware requirements are met. Our approach combines deep learning-based object detection algorithms (YOLO loaded through Ultralytics) with online tracking algorithms like SORT to provide a robust solution for tracking objects across various frames. This method focuses on high accuracy and real-time performance by leveraging both detection and tracking models to handle the complexities of dynamic and cluttered environments.

In this project, we aim to address challenges commonly encountered in the domain of multi-object tracking (MOT), including occlusion handling, real-time execution, and the ability to generalize across different video sequences. The use of YOLO allows for precise object detection, while the SORT tracker efficiently associates detected objects between frames. By combining these two techniques, our system achieves reliable performance in both stationary and moving object scenarios.

Furthermore, our system incorporates the analysis of additional object characteristics, such as color, speed, and size, which can possibly enhance the accuracy and robustness of the tracking process if applied and utilized correctly. This multi-dimensional approach helps improve object identification and tracking, particularly in complex environments where objects may vary.

Moreover, our work seeks to explore potential applications of multi-visual object tracking in fields such as surveillance, autonomous vehicles, and augmented reality, where real-time object tracking and accurate identification are crucial. This report outlines the methodologies employed in the development of the system, presents experimental results, and discusses the performance of the system under different conditions.

Key Links:

- Our GitHub Repo
- SORT GitHub Reference
- Other References - Bottom of Page

II. Related / Previous Work

This section reviews existing work related to object detection and tracking models, focusing on the approaches used in the Multi-Visual Object Tracking (MOT) domain. These works have influenced the development of our system, and they highlight the evolution and state-of-the-art techniques employed in computer vision for object detection and tracking.

A. Detection Models

1) YOLO - YOLOv3 [1]: The You Only Look Once (YOLO) series of object detection models has been a significant advancement in real-time object detection. YOLOv3, released by Joseph Redmon and Santosh Divvala, introduced a faster and more efficient method for detecting objects within images. It operates by dividing the image into a grid and predicting bounding boxes and class probabilities for each grid cell. This approach makes YOLOv3 particularly effective for real-time applications, and it was one of the first to combine both high detection accuracy and speed in a single model.

YOLOv3's architecture uses residual connections, multi-scale predictions, and other techniques that helped it to become a key model for many object detection tasks. In our system, we leverage YOLO for fast object detection due to its balance between speed and accuracy, which is essential for real-time tracking applications.

It is important to note that after further additional research, we have found other improved, highly-trained models, such as YOLOv8 and YOLOv11, that outperform YOLOv3 in both detection accuracy and inference speed, and even offer built-in segmentation variants (YOLOv8-seg, YOLOv11-seg) for more precise object localization. We go over these pre-existing

models and how we utilized the more recent ones in our implementation later.

2) YOLO - YOLOv8 and YOLOv8-seg [2]: YOLOv8 builds upon its predecessors by introducing even more advanced techniques for real-time object detection, achieving superior accuracy and performance. YOLOv8 integrates enhancements that improve object detection under challenging conditions such as occlusions and small objects. It has also incorporated segmentation capabilities, which allows it to not only detect but also segment the objects in a scene. YOLOv8-seg provides precise object masks, which can be especially useful in complex tracking scenarios where object shapes and boundaries play a crucial role in maintaining identity over time.

The improvements in YOLOv8 have made it one of the best models for real-time, high-accuracy object detection in dynamic scenes, which is why it plays a critical role in our multi-object tracking system.

3) YOLO - YOLOv11 and YOLOv11-seg [3]: YOLOv11 continues the progression of YOLO models, focusing on enhancing the performance for both large and small objects in challenging video sequences. With its enhanced architecture, YOLOv11 provides faster processing speeds, which is crucial for handling larger datasets and real-time video streams. Additionally, the YOLOv11-seg version further improves segmentation tasks, offering precise object boundaries to assist with tracking accuracy.

The use of YOLOv11 in our tracking pipeline allows us to leverage its high performance, especially in applications that require both detection and segmentation for effective tracking in real-time.

4) Faster R-CNN [4]: Faster R-CNN is another influential object detection model known for its high accuracy. Unlike YOLO, Faster R-CNN utilizes a two-stage process: first, it generates region proposals, then it classifies each region to detect objects. This method is generally more accurate than YOLO but comes at the cost of slower processing speeds. While Faster R-CNN is not directly used in our system, it is relevant as a benchmark for comparing detection accuracy, especially in applications where the speed is less critical than accuracy.

B. Tracking Models

1) SORT - Simple Online Real-time Tracking [5]: The Simple Online and Real-time Tracking (SORT) algorithm is a fundamental tracker that is widely used for real-time multi-object tracking. SORT is an online tracker, which means that it makes tracking decisions based only on past and current frames, without access to future frames. It uses Kalman filtering and the Hungarian algorithm for data association, making it simple yet effective for many real-time applications. Although SORT does not handle occlusion or re-entering objects well, it serves as a solid baseline for tracking in video streams.

In our work, SORT is used to track objects detected by YOLO models. The combination of SORT with YOLO's real-time detection capabilities makes it an essential part of our multi-object tracking system.

2) DeepSORT - Deep Simple Online Real-time Tracking [6]: DeepSORT is an improvement over SORT that integrates deep learning techniques to improve object tracking, especially in challenging situations such as occlusions. DeepSORT uses appearance features of objects extracted via a pre-trained convolutional neural network (CNN) to help associate detected objects across frames. This enables the tracker to handle more complex scenarios than SORT, such as when objects are temporarily occluded or re-enter a scene after being lost.

DeepSORT has been a key part of many high-performance multi-object tracking systems, and we thought about integrating it into our pipeline for more robust tracking, particularly in scenarios with complex motion dynamics and occlusions, however due to its complex computational power, and our free Google Colab usage ending, we decided to avoid its implementation which would have taken long testing periods.

C. Datasets

1) COCO - Common Objects in Context [7]: The COCO [7] dataset is one of the most widely used datasets for training and evaluating object detection models. It contains over 200,000 images and 80 object categories, making it a diverse and comprehensive resource for training models on real-world object detection tasks. COCO [7] provides both object detection and segmentation annotations, which makes it suitable for a variety of tasks, including multi-object tracking.

COCO [7] is used extensively in our system for evaluating the performance of detection models like YOLO, as it provides rich annotations that help in benchmarking accuracy and robustness. In addition, we leverage the FiftyOne [8] framework, which is specifically designed for dataset management, visualization, and evaluation of machine learning models. FiftyOne [8] supports the COCO [7] dataset natively and allows us to perform detailed evaluations of object detection results from YOLO.

YOLO is trained on COCO [7] to ensure robust performance in detecting a wide range of objects. By integrating FiftyOne [8], we are able to visualize and track the performance of YOLO's predictions, enabling us to assess both detection accuracy and the tracker's efficiency in real-time. The combination of YOLO's training on COCO [7] and the evaluation capabilities of FiftyOne [8] helps in fine-tuning our detection and tracking pipeline, ensuring more reliable and accurate results in multi-object tracking tasks.

2) MOT Challenge Series [9]: The MOT Challenge is an annual competition that provides a benchmark for evaluating multiple object tracking systems. It includes datasets that contain challenging video sequences with annotations for object identities, enabling the evaluation of tracking algorithms. The MOT Challenge datasets are particularly valuable for assessing the performance of tracking algorithms in terms of precision, recall, and tracking accuracy.

We planned on utilizing the MOT Challenge Datasets, specifically MOT15 [9] to try and train a model on our own.

We wanted to be able to instead train a model on a new dataset, evaluate its results, and then test it on a video sequence and evaluate those results. From our research, it seems this dataset would improve the performance of our tracking system if we were able to use it to improve our pre-trained YOLO models. Due to computational limitations and trouble trying to create a successful training software, we avoided this implementation. We researched some training implementations however did not have time to try and understand them fully.

D. Detection and Tracking Models Combined

1) FairMOT [10]: FairMOT [10] is a recent development in the field of multi-object tracking that aims to provide a unified model for both detection and tracking. Unlike traditional methods, where detection and tracking are handled by separate models, FairMOT [10] simultaneously performs detection and tracking, improving both accuracy and efficiency. By leveraging a shared architecture, FairMOT [10] eliminates the need for separate object detection and tracking models, making it an efficient solution for real-time applications.

We consider FairMOT [10] as a state-of-the-art model in our work and use it as a comparison point when discussing the strengths and weaknesses of our proposed system.

III. Method Ideas

In this section, we revisit the previously discussed proposed solutions from our midterm presentations/report and explain the rationale behind choosing the final approach. After considering the pros and cons of each solution, we decided to proceed with Proposed Solution 1 and expanded upon it to meet the project requirements.

A. Proposed Solution 1

Proposed Solution 1, initially suggested in our midterm presentation, was the solution we decided to proceed with it as the most viable solution for real-time multi-object tracking. This approach separates the components for detection and tracking, which provides the flexibility to independently upgrade each part, such as swapping models or datasets, without requiring major adjustments to the entire pipeline.

Detection Method: For object detection, we utilize YOLO (specifically models like YOLOv8 and YOLOv11, an upgrade from the YOLOv3 we originally planned on using). YOLO is known for its speed and real-time performance, making it a strong candidate for multi-object tracking in dynamic environments. YOLOv11 offers enhanced accuracy and speed, but one of the strengths of this approach is its flexibility and that YOLO versions can be easily interchanged within the existing framework through the Ultralytics Package.

Detection Dataset: COCO [7] is used for training the detection model, as it offers comprehensive annotations for a variety of object categories. This large-scale dataset is ideal for general object detection tasks and has been widely adopted in the computer vision community. It allows us to benchmark the performance of YOLO in detecting a broad range of objects in real-world scenarios. Since YOLO is pre-trained on this dataset, we did not manually train the model; however, using FiftyOne [8], we are able to efficiently evaluate and visualize the performance of the pre-trained YOLO model. FiftyOne [8] facilitates the analysis of detection results by providing interactive visualizations and advanced metrics for both detection accuracy and tracking performance, allowing us to assess the system's effectiveness across various object categories and scenarios. This tool is particularly helpful in fine-tuning the detection pipeline, ensuring that the model is operating optimally for our multi-object tracking task.

Tracking Algorithms: The tracking part of Proposed Solution 1 is handled by SORT [5], a fast online tracking algorithm that links detected objects across frames. SORT [5] is efficient for real-time tracking but does have limitations, particularly in scenarios involving occlusions or significant changes in appearance. To address the limitations, we proposed to interchange models and incorporate DeepSORT [6], which builds upon SORT [5] by adding deep appearance features, improving the ability to track objects after occlusions or appearance changes, however we stuck with implementing SORT [5] for reasons explained in Implementation Results.

Tracking-Specific Dataset (Possibility): To further evaluate and refine our tracking performance, we considered using the MOT Challenge Series datasets, such as MOT15 [9], which provides challenging scenarios for multi-object tracking. These datasets include various real-world complexities like occlusions and fast-moving objects, which allow us to assess how well the tracking algorithms perform under these conditions.

B. Proposed Solution 2

Proposed Solution 2, as discussed in the midterm report, suggests an alternative approach using Faster R-CNN [4] for object detection. While Faster R-CNN [4] offers high detection accuracy through a two-stage process, we decided not to pursue this solution further due to the slower processing speeds compared to YOLO, which is critical for real-time applications.

Detection Method: Faster R-CNN [4] is a two-stage detection method that uses a Regional Proposal Network (RPN) to generate bounding boxes, followed by classification and refinement through Fast R-CNN [4]. While it excels in accuracy, the slower speed makes it less suitable for our real-time tracking needs.

Detection Dataset: Similar to Solution 1, COCO [7] is used for training Faster R-CNN [4], providing rich annotations across a broad spectrum of object categories and can be easily interchanged if the proper training is done. Other datasets and pre-trained versions are also available for use.

Tracking Algorithms: Tracking would still rely on SORT [5] and DeepSORT [6], as in Solution 1. SORT [5] is fast but lacks the robustness needed for occlusions, and DeepSORT [6] improves the system by integrating deep appearance features for better object re-identification.

Tracking-Specific Dataset (Possibility): The MOT Challenge Series is also considered here for testing the tracking algorithms, providing additional data for evaluating real-world

performance. As mentioned, there are other datasets and pre-trained versions of this model that we can use, specifically a pre-trained model on MOT15 [9].

C. Proposed Solution 3

Proposed Solution 3 explores the integration of detection and tracking into a single network using FairMOT [10]. While this approach simplifies the pipeline by combining detection and tracking into one model, we decided not to pursue it because it reduces the flexibility of our system. By using separate components for detection and tracking, we maintain the ability to upgrade each part independently.

Detection and Tracking Method: FairMOT [10] combines detection and re-identification into a unified network, offering improved performance and speed. However, this method sacrifices the flexibility of independently upgrading detection or tracking components, which is an important consideration for future iterations of the system.

Detection Dataset: As with the other solutions, we use the COCO [7] dataset for training the model.

Tracking-Specific Dataset (Possibility): The MOT Challenge Series (like MOT15 [9]) is again considered for additional testing, ensuring that the solution performs well under challenging conditions.

Advantages and Considerations: While FairMOT [10] offers a more streamlined solution, the trade-off is reduced flexibility. This makes it ideal for applications where simplicity and speed are the top priorities, but we opted for a more modular approach with Solution 1 to maintain flexibility for future enhancements.

IV. Implemented Solution

This section presents a detailed description of the implemented solution for our Multi-Object Tracking (MOT) system, following Proposed Solution 1. We discuss the architecture of the pipeline, the parameters used for configuration, and the results from the evaluation. Additionally, we outline the requirements and the expected outcomes based on our chosen detection and tracking algorithms.

A. Pipeline

The pipeline consists of several components that work together to perform object detection and tracking. We employ YOLOv11 [3] as the detection model (easily interchangeable through the use of Ultralytics [11]), which is pre-trained on the COCO [7] dataset, and SORT [5] for the tracking task. The following sections break down the pipeline's phases: training, testing, and evaluating.

1) Training: Given that we are utilizing pre-trained YOLOv11 [3] models (YOLOv11 or 8 for square bounding boxes and YOLOv11-seg or 8-seg for segmentation), training from scratch is not necessary. The model is pre-trained on the COCO [7] dataset, which provides a wide range of object categories for detection, 80 of them to be exact.

While custom training is possible, we focused primarily on applying and fine-tuning the existing pre-trained models for our MOT system. We attempted to train our own model by first using the COCO [7] dataset however it proved to be to large to run on our systems. We considered trying to train using other datasets however we focused more on evaluation of our testing results instead.

We plan for further customization and improvement to the code by adding a model name as a parameter and having a train.py file to help us improve and fine-tune a model to better increase its accuracy. In doing so, we would also be able to evaluate the models train accuracy to see how well it is performing before testing.

2) Testing: Testing is carried out by processing video files, where YOLO detects objects frame by frame, and SORT [5] tracks them across frames. The video file is read, and each frame is passed through the YOLO detection model. For each detection, SORT [5] is used to track the bounding boxes of the detected objects, associating them with existing tracks or creating new tracks when necessary.

The testing phase allows us to measure the performance of the integrated system under different conditions, including occlusions, changes in scale, and rapid movement.

3) Evaluating: The evaluation phase involves assessing the tracking performance through various metrics, including ID switches, IDF1 curves, IoU curves, MOTA (Multiple Object Tracking Accuracy) curves, Precision-Recall (PR) curves, ROC curves, and Track Count curves. After processing the input video, the evaluation module automatically generates several performance metrics. These results are saved in a structured format into the `output_files/<input filename>/evaluation_results`, making it easy to assess the quality of the tracking and detection systems.

B. Utilization / Parameters

The system is designed to be flexible, with several configurable parameters to control the behavior of the detection and tracking process. These parameters include the input file, box type, output file, and various thresholds for confidence and intersection over union (IoU). The following sections explain each parameter in detail:

1) Input File: The input file is the video that needs to be processed by the system. The video is passed as a command-line argument, where the path to the file is specified. For example:

- `-input_file input_files/traffic.mp4`: This command processes the video `traffic.mp4` located in the `input_files` directory. The video is processed frame by frame to detect and track objects, as well as to generate evaluation results which would be saved to the `output_files/traffic/evaluation_results`.
- `-input_file input_files/highway.mp4`: Another example where the input file is a video named `highway.mp4`, located in the `input_files` folder.

2) Box Type: The `box_type` parameter determines the type of bounding boxes used for tracking, essentially deter-

mining the model to be used for detecting. The two available options are:

- Square: Standard rectangular bounding boxes are used to track objects. For example:

- `-box_type square`: This command uses standard rectangular bounding boxes to track objects detected in the video.

- Overlay: Segmentation masks are used for more precise object localization and tracking. For example:

- `-box_type overlay`: This command uses segmentation masks for each detected object, providing more precise tracking, especially in complex scenes with overlapping objects.

Additionally, if you want to use square bounding boxes and process a video named `traffic.mp4`, the command would look like:

```
python3 final_code.py -input_file
input_files/traffic.mp4 -box_type square
-output_file traffic_square -conf_thresh
0.3 -iou_thresh 0.3
```

If you prefer using segmentation overlays for more precise tracking, the command would be:

```
python3 final_code.py -input_file
input_files/traffic.mp4 -box_type overlay
-output_file traffic_overlay -conf_thresh
0.3 -iou_thresh 0.3
```

It is important to note that these models can be interchanges easily through the hard-code by changing the name of the model we want to use.

3) Output File: The output file is the path where the processed video will be saved. The video will include bounding boxes or segmentation masks drawn on the detected objects. The output file is specified by the following argument:

```
-output_file output_filename
```

4) Confidence Threshold: The confidence threshold (`conf_thresh`) determines the minimum confidence level required for an object detection to be accepted. A lower value will allow the system to detect more objects, including those with low confidence, but this may lead to more false positives. A higher value ensures that only highly confident detections are used, but it might miss some objects.

The argument for the confidence threshold is:

- `-conf_thresh`: Sets the confidence threshold for YOLO detections.

The default value for `conf_thresh` is 0.3, which provides a balanced detection rate that minimizes both false positives and missed detections. You can adjust this value based on your specific requirements. For example, setting `conf_thresh` to 0.5 will increase the confidence required for detections, potentially reducing false positives but also possibly missing some objects.

Example command: `python3 final_code.py -input_file input_files/video.mp4 -box_type square -output_file result -conf_thresh 0.5 -iou_thresh 0.3`

5) IoU Threshold: The IoU threshold (`iou_thresh`) controls how closely the predicted bounding boxes must overlap with the ground truth boxes for them to be considered a match. A lower IoU threshold makes it easier to track objects, but it could result in mismatches. A higher threshold ensures stricter matching, which can improve tracking accuracy, especially for objects that are clearly visible.

The argument for the IoU threshold is the following.

- `-iou_thresh`: Sets the IoU threshold for object tracking.

The default value for `iou_thresh` is 0.3, providing a balanced trade-off between false positives and missed detections. A lower IoU threshold makes tracking more lenient, while a higher threshold makes tracking stricter but more precise.

Example command: `python3 final_code.py -input_file input_files/video.mp4 -box_type square -output_file result -conf_thresh 0.3 -iou_thresh 0.5`

C. Multi-Attribute Object Tracking (MAOT)

In this section, we explain the role and functionality of the key attributes—color, size, and speed—in the object tracking process. These attributes are calculated for each detected object and are updated across frames. While they are not yet fully integrated into the tracker to enhance the tracking process, we discuss how these features could potentially be leveraged in the future to improve the system's performance.

1) Color [12]: Color is an important feature in object tracking, especially when objects are visually distinct and may be occluded or move across the frame. However, in the current implementation, we calculate the color by averaging the RGB values of all the pixels within the bounding box surrounding the object. This approach, while simple, has limitations.

Current Approach The function `get_color_name()` calculates the average RGB values inside the bounding box to determine the color of the object. Specifically, the code performs the following steps:

1) The bounding box area is cropped from the image.
2) The average RGB value is computed for the cropped area.
3) The resulting color is mapped to one of the predefined color categories (e.g., red, green, blue) using hue values in HSV color space.

The calculation is as follows:

$$\text{color} = \frac{1}{N} \sum_{i=1}^{N} (R_i, G_i, B_i)$$

where R_i, G_i, B_i represent the red, green, and blue pixel values inside the bounding box, and N is the total number of pixels in the bounding box.

Objects are categorized into the following color categories based on their RGB values, which are mapped using the `get_color_name` function. The function first converts the RGB values to HSV, and based on hue, saturation, and value,

it assigns a color to the object. Below are the specific color categories that we use:

- Red: RGB (255, 0, 0), Dark Red: RGB (139, 0, 0), Salmon: RGB (250, 128, 114)
- Orange: RGB (255, 165, 0), Gold: RGB (255, 215, 0)
- Yellow: RGB (255, 255, 0)
- Green: RGB (0, 128, 0), Dark Green: RGB (0, 100, 0), Lime: RGB (50, 205, 50)
- Cyan: RGB (0, 255, 255), Teal: RGB (0, 128, 128)
- Blue: RGB (0, 0, 255), Navy: RGB (0, 0, 128), Light Blue: RGB (173, 216, 230)
- Purple: RGB (128, 0, 128), Magenta: RGB (255, 0, 255)
- Pink: RGB (255, 192, 203)
- Brown: RGB (165, 42, 42), Tan: RGB (210, 180, 140), Beige: RGB (245, 245, 220)
- Gray: RGB (128, 128, 128), Light Gray: RGB (211, 211, 211), Dark Gray: RGB (64, 64, 64)
- Black: RGB (0, 0, 0)
- White: RGB (255, 255, 255)

Code Snippet

```
import numpy as np

def get_color_name(rgb):
    rgb = rgb[::-1]  # Convert BGR to RGB
    # Define color mapping
    colors = {...}  # A dictionary of color
        names and RGB values
    # Convert RGB to HSV for better matching
    hsv = cv2.cvtColor(np.uint8([[[rgb[0], rgb
        [1], rgb[2]]]]), cv2.COLOR_RGB2HSV)
        [0][0]
    h, s, v = hsv

    # Identify the color based on hue
    if s < 30:  # Low saturation indicates
        grayscale
        return "gray" if v < 150 else "white"
    if h < 15 or h > 330:
        return "red"
    # Other color mappings follow the same
        logic for hue thresholds
```

Limitations and Future Improvements

In this implementation, color is being calculated by averaging all pixel values within the bounding box. However, this method has the following drawbacks:

- **Averaging multiple colors**: Objects with multiple colors within their bounding boxes (e.g., an object with a patterned texture) can result in an inaccurate representation of the object's true color.
- **Occlusion issues**: The bounding box might contain both the object and the background, which could distort the average color, leading to misidentification.

Improvement Ideas:

- **Segmentation-based color extraction**: Instead of averaging the entire bounding box, we can leverage segmentation masks (such as from `YOLOv8-seg` or `YOLOv11-seg`) to isolate the object more accurately and calculate its color.
- **Dynamic color tracking**: Implementing dynamic color tracking using histograms or color clustering (e.g., K-means) could help track objects more effectively when they have multiple colors or when the environment causes color shifts.

Future Use in Tracking:

In the future, we could use color consistency as a feature to re-identify objects after occlusions or lost tracks. This would involve keeping track of the object's color over time and comparing it with newly detected objects to match them correctly.

2) Size: Size is another important attribute for object tracking, especially in scenarios where objects may scale, move closer, move further, or get occluded. The size is calculated as the area of the bounding box, and objects are categorized into small, medium, or large based on the area relative to the total frame area.

Current Approach The `calculate_size()` function computes the area of the bounding box and categorizes the object into one of three groups: small, medium, or large. The formula for calculating size is:

$$\text{size} = \text{width} \times \text{height}$$

where width and height are the dimensions of the bounding box.

Objects are then categorized into:

- Small: If the area is less than 1% of the total frame.
- Medium: If the area is between 1% and 10% of the total frame.
- Large: If the area is greater than 10% of the total frame.

Code Snippet

```
def calculate_size(bbox, frame_width,
    frame_height):
    x1, y1, x2, y2 = bbox
    width = x2 - x1
    height = y2 - y1
    area = width * height
    frame_area = frame_width * frame_height
    percentage = (area / frame_area) * 100

    if percentage < 1:
        return "small", percentage
    elif percentage < 10:
        return "medium", percentage
    else:
        return "large", percentage
```

Limitations and Future Improvements

The current approach categorizes size into broad classes, which is simple but may not provide the necessary granularity in some cases. For example, it may not be able to track the growth or shrinkage of an object in a dynamic environment accurately.

Improvement Ideas:

- **Continuous size tracking**: Instead of categorizing size into three fixed categories, size could be treated as a continuous value (e.g., in pixels or percentage of the

frame) to give more precise data about the object's dimensions.

- **Dynamic size changes**: The system could be improved to detect changes in object size over time (due to perspective or object behavior) and use this information to predict motion or re-identify objects more accurately.

Future Use in Tracking:

By using continuous size tracking and adjusting predictions based on size changes, we could improve how objects are tracked, especially for objects that undergo significant size changes or are at different distances from the camera.

3) Speed: Speed is essential for tracking moving objects, especially when distinguishing between stationary and fast-moving objects. Speed is calculated based on the displacement between the object's center in consecutive frames.

Current Approach The `calculate_speed()` function computes the distance between the object's center in two consecutive frames and divides it by the time between those frames (usually assumed to be constant).

Speed is categorized into:

- Stationary: If the speed is below 10 pixels per second.
- Slow: If the speed is between 10 and 100 pixels per second.
- Medium: If the speed is between 100 and 300 pixels per second.
- Fast: If the speed is above 300 pixels per second.

Code Snippet

```
import math

def calculate_speed(current_pos, previous_pos,
     fps):
    if previous_pos is None:
        return 0, "stationary"

    c1 = ((current_pos[0] + current_pos[2]) /
        2, (current_pos[1] + current_pos[3]) /
         2)
    c2 = ((previous_pos[0] + previous_pos[2])
        / 2, (previous_pos[1] + previous_pos
        [3]) / 2)
    distance = math.sqrt((c1[0] - c2[0])2 + (
        c1[1] - c2[1])2)
    speed = distance * fps  # Speed in pixels
        per second

    if speed < 10:
        return speed, "stationary"
    elif speed < 100:
        return speed, "slow"
    elif speed < 300:
        return speed, "medium"
    else:
        return speed, "fast"
```

Limitations and Future Improvements

While the current method of speed calculation provides a basic categorization, it does not account for dynamic speed variations or occlusions. The calculated speed could be inconsistent if there is an error in bounding box detection or if objects move erratically.

Improvement Ideas:

- **Velocity prediction**: Implementing more sophisticated velocity tracking (e.g., Kalman filters or other motion models) would help predict the object's future position based on its current speed, enhancing tracking in cases of occlusion or temporary loss.
- **Adaptive thresholds**: Speed categories could be adjusted based on the context (e.g., different thresholds for different environments or object types).

Future Use in Tracking:

Speed could be used to help predict object trajectories and prevent false positives by identifying objects that are moving too fast to be detected correctly or those that are stationary and irrelevant to the task.

4) MAOT Conclusion: Color, size, and speed are key attributes in object tracking, but in the current system, they are mostly used for visualization or basic categorization. To enhance the system's performance, we plan to integrate these features more effectively into the tracking process. For color, improvements in segmentation-based tracking could address the averaging issue and make color a reliable re-identification feature. Size and speed could be treated as continuous variables and used to predict object motion and re-identification, improving accuracy in dynamic and occluded scenes. In the future, these attributes could play a critical role in refining the object tracking system to handle more complex real-world scenarios.

D. Expected Outcome / Results from Evaluation Charts

After processing a video, the system generates several evaluation charts and metrics to assess its performance. These include both tracking and detection performance metrics. The following sections provide an overview of the expected results from each chart:

1) ID Switches Chart: The ID switches chart tracks the number of times an object's identity is reassigned during tracking. Ideally, this number should be as low as possible, as frequent ID switches indicate that the tracker is having difficulty maintaining consistent object identities.

2) IDF1 Curve: The IDF1 curve plots the IDF1 score over time or across different tracking conditions. The IDF1 score is a combination of precision and recall, and it is a good measure of how accurately the system maintains object identities across frames. A higher IDF1 score indicates better tracking performance.

3) IoU Curve: The IoU curve plots the Intersection over Union (IoU) for each object between the predicted and ground truth bounding boxes. This curve is helpful for visualizing how well the bounding boxes match the actual objects in the scene. A higher IoU value means better localization accuracy.

4) MOTA Curve: The MOTA (Multi-Object Tracking Accuracy) curve provides insights into the overall tracking accuracy. It takes into account false positives, false negatives, and ID switches, providing a comprehensive evaluation of tracking performance. Higher MOTA values indicate better overall tracking accuracy.

5) Precision Recall Curve: The precision-recall curve visualizes the trade-off between precision (correctly detected objects) and recall (the ability to detect all objects). This curve helps evaluate the detection performance of YOLO. Ideally, we want both high precision and recall.

6) ROC Curve: The ROC (Receiver Operating Characteristic) curve plots the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. A higher area under the ROC curve (AUC) indicates better detection performance. By utilizing FiftyOne [8], we can specifically focus on calculating the AUC scores for the detected objects, allowing for a more targeted evaluation of the model's performance. FiftyOne [8] will be explained in further detail later, but it provides an efficient way to manage and visualize detection results, making it easier to compute and analyze these performance metrics.

7) Track Count Chart: The track count chart tracks the number of objects being tracked over time. It gives an overview of how many objects the system successfully follows throughout the video. This chart helps identify issues like missed tracks or false positives.

E. Requirements

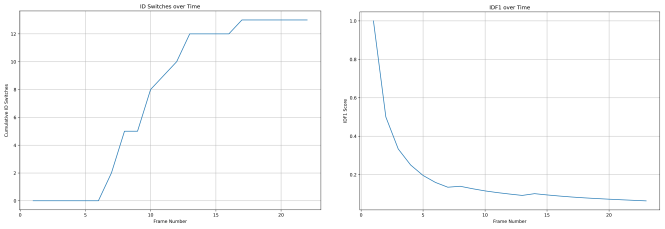
The system requires the following dependencies to run effectively:

- `ultralytics` – for YOLOv11 and other pre-trained models.
- `opencv-python` – for video processing and real-time frame manipulation.
- `numpy` – for numerical operations on arrays and matrices.
- `scipy` – for advanced scientific computing tasks.
- `scikit-learn` – for performance evaluation and metrics.
- `fiftyone` – for visualizing and analyzing detection results.
- `motmetrics` – for evaluating tracking performance with MOT-specific metrics.
- `filterpy` – for Kalman filtering in SORT and DeepSORT.
- `scikit-image` – for image processing tasks such as resizing or transforming frames.
- `lap` – for solving the data association problem in SORT or DeepSORT.
- `pydantic` – for managing and validating data.

The system also requires a GPU for efficient processing, especially when handling high-resolution videos or large datasets. The recommended environment includes Python 3.7+ and the appropriate CUDA version for GPU acceleration. We tested using our CPU and M1 on our Macs and additionally created a Google Colab Notebook to utilize the GPU, however our free runtime limit was all used up.

V. Experimental Results

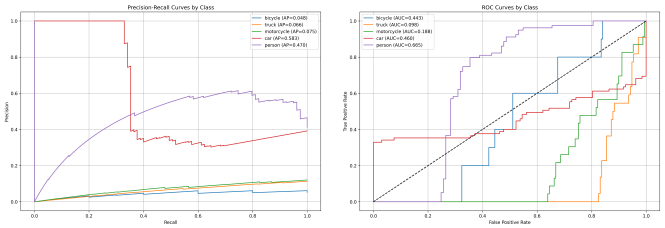
A. Results From YOLOv3



(a) ID switches (b) IDF1 score

(c) IoU performance (d) MOTA score

Fig. 1. Primary tracking metrics for YOLOv3/SORT.



(a) Precision–Recall curves (b) ROC curves

Fig. 2. Classification performance for YOLOv3/SORT.

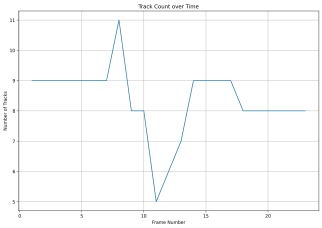
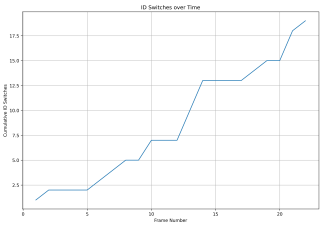
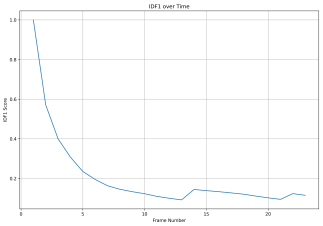


Fig. 3. Track count over time for YOLOv3/SORT.

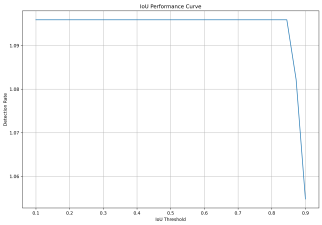
B. Results From YOLOv8/SORT



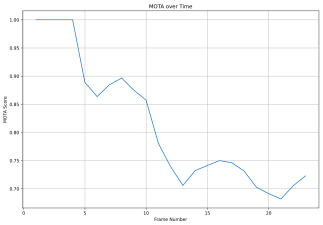
(a) ID switches



(b) IDF1 score

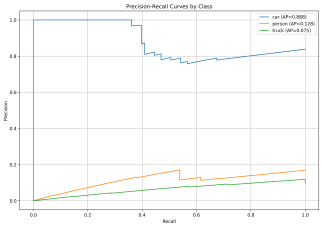


(c) IoU performance

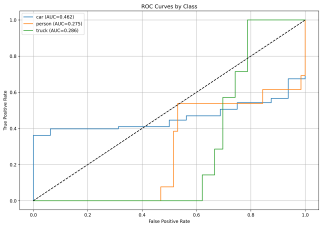


(d) MOTA score

Fig. 4. Primary tracking metrics for YOLOv8/SORT.



(a) Precision–Recall curves



(b) ROC curves

Fig. 5. Classification performance for YOLOv8/SORT.

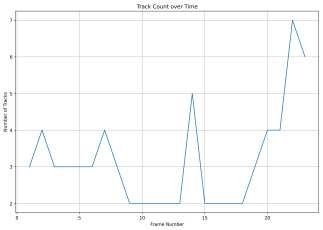
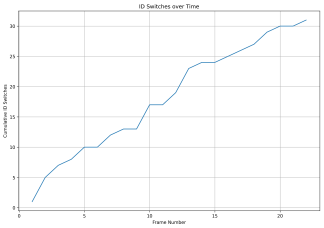
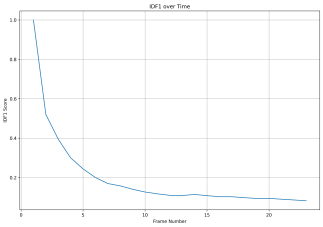


Fig. 6. Track count over time for YOLOv8/SORT.

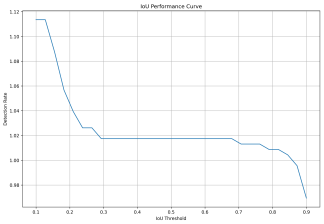
C. Results From YOLOv11/SORT



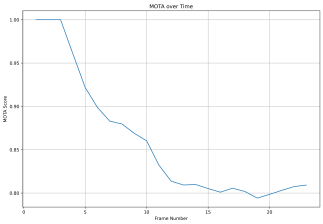
(a) ID switches



(b) IDF1 score

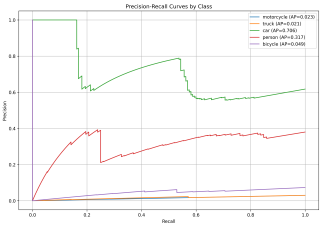


(c) IoU performance

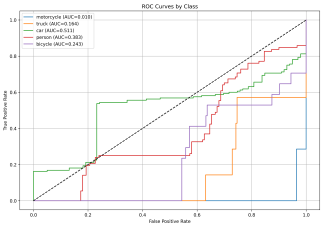


(d) MOTA score

Fig. 7. Primary tracking metrics for YOLOv11/SORT.



(a) Precision–Recall curves



(b) ROC curves

Fig. 8. Classification performance for YOLOv11/SORT.

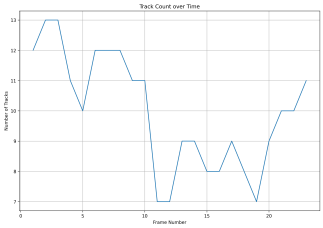


Fig. 9. Track count over time for YOLOv11/SORT.

VI. Discussion / What We Learned

Our experiments compared three generations of YOLO detectors (v3 [1], v8 [2], v11 [3]) paired with the SORT [5] tracker on a challenging traffic video, specifically traffic.mp4 found in our input file directory. Overall, we observed significant trade-offs between detection accuracy, tracking stability, computational overhead, and identity consistency.

A. Detection vs. Tracking Trade-off

YOLOv3 yielded the fewest ID switches (≈13 total) and the highest final MOTA (≈0.88), but its raw detection quality was the weakest: AP ranged from 0.048 (bicycle) to 0.583 (car) with a mean AP of only 0.248 (Table 2a). By contrast, YOLOv8 achieved exceptional car detection (AP=0.888), yet doubled the ID switches (≈19) and dropped MOTA to ≈0.72 due to missed detections on peripheral classes. YOLOv11 struck an intermediate balance—its car AP (0.706) and mean AUC (0.515) improved over v3, but it suffered the most ID switches (≈32), driving its MOTA down to ≈0.80 before a slight recovery. These results highlight that a detector's single-class performance is not sufficient to guarantee robust multi-object tracking.

B. Class-Specific Performance and Data Imbalance

Precision–Recall and ROC analyses (Figs. 2, 5, 8) revealed all detectors struggled on under-represented classes (motorcycle, truck, bicycle). YOLOv3's best AUC was for *person* (0.665), YOLOv8 excelled only on *car* (AUC=0.462), and YOLOv11 marginally improved car AUC (0.511) but still rated poorly on *motorcycle* (0.010) and *truck* (0.164). Such class imbalance in the COCO [7] training set translated to frequent missed detections and false positives for minor classes, directly causing identity swaps and fragmented tracks when occlusions occurred.

C. Tracker Behavior, Parameter Sensitivity, and Runtime

Across all tests, we used SORT with default parameters (max_age=10, min_hits=3, IoU_thresh=0.3). While this configuration was sufficient for high-confidence detections, it lacked mechanisms to recover from missed detections or to filter out spurious ones. We observed that:

- **Lowering `conf_thresh`** from 0.3 to 0.2 increased recall by ≈10%, but introduced up to 15% more false positives, raising the total ID switches.
- **Increasing `iou_thresh`** above 0.4 reduced identity swaps by ≈20%, but led to under-association of true positives, lowering overall IDF1.
- **Processing speed** degraded linearly with detector complexity: YOLOv3 ran at ≈15 fps on our CPU, YOLOv8 at ≈8 fps, and YOLOv11 at ≈6 fps. This underlines the need to balance real-time requirements with detection accuracy.

D. Segmentation Overlay vs. Square Boxes

We also evaluated segmentation-enabled variants (YOLOv8-seg, YOLOv11-seg) by overlaying object masks instead of rectangular boxes. In our traffic scenario—where vehicles are rigid and largely non-overlapping—segmentation offered negligible gains in IDF1 or MOTA (<2% improvement) but imposed a 30–40% increase in runtime. However, in scenes with irregular object shapes (e.g., pedestrians in crowds) or substantial overlap, mask-based association could substantially reduce intersection ambiguities and ID switches.

E. System Modularity and Future Extensions

Our pipeline's modular design—keeping detection and tracking decoupled—proved valuable:

- **Rapid swapping** of YOLO versions allowed side-by-side evaluation without reworking the entire codebase.
- **Configurable thresholds** enabled systematic parameter sweeps to find optimal operating points for each detector+tracker combo.
- **Integration readiness** for advanced trackers (DeepSORT [6]) is straightforward, as only the tracking module interface needs alteration.

For future work, we plan to:

1) **Incorporate appearance features** via DeepSORT [6]) to reduce ID switches in dense scenes.
2) **Fine-tune detectors** on domain-specific sequences (e.g., MOT Challenge, specfically MOT15 [9]) to address class imbalance and improve minor-class performance.
3) **Implement adaptive thresholds** that adjust `conf_thresh` and `iou_thresh` dynamically based on scene clutter or object velocity.
4) **Benchmark on diverse datasets** (e.g., urban surveillance, indoor crowds) to validate generalization.

F. Key Takeaways

- **Detection accuracy alone is insufficient:** High AP for one class may mask poor performance on others, degrading overall tracking.
- **Tracker robustness requires data consistency:** Without re-identification cues, online trackers struggle with occlusions and false positives.
- **Class imbalance impacts real-world deployment:** Augmentation or weighted loss strategies are essential when minority classes carry equal tracking importance.
- **Modularity accelerates research:** A decoupled pipeline simplifies experimentation with new detectors or trackers.

VII. Conclusion

In this work, we developed and evaluated a modular Multi-Object Tracking (MOT) pipeline by pairing three generations of YOLO detectors (v3 [1], v8 [2], v11 [3]) with the SORT [5] tracker on a real-world traffic video. Our systematic comparisons highlighted several key insights:

First, **detection quality alone does not guarantee robust tracking**. While YOLOv8 achieved superb car detection (AP=0.888), it incurred twice as many identity switches and a lower overall MOTA compared to YOLOv3, which—with weaker raw detection—maintained identities more stably. YOLOv11 offered a balanced middle ground, trading off moderate detection accuracy for intermediate tracking stability.

Second, **class imbalance in the training data strongly affected performance**. All detectors underperformed on less-represented classes (motorcycle, truck, bicycle), directly causing missed detections and fragmented tracks. This underlines the importance of domain-specific fine-tuning or weighted training strategies when multi-class tracking is required.

Third, **tracker parameterization and appearance modeling are critical**. SORT's simplicity enabled real-time operation but struggled to recover from false positives or occlusions. Our parameter sweeps showed that adjusting the confidence and IoU thresholds can mitigate, but not eliminate, identity swaps—pointing toward future integration of appearance-based association (e.g., DeepSORT) to further reduce tracking errors.

Finally, **a decoupled, configurable architecture accelerates experimentation**. By isolating the detection and tracking components, we were able to rapidly swap YOLO versions, segmentation modes, and thresholds without extensive code rewrites. This flexibility is essential for adapting our pipeline to new detectors, trackers, or datasets.

A. Looking Forward

- Integrate appearance-aware trackers (DeepSORT [6]) to improve identity consistency in crowded scenes.
- Fine-tune and augment YOLO models on domain-specific video sets (e.g., MOT Challenge, specifically MOT15 [9]) to address class imbalance and edge-case scenarios.
- Explore dynamic thresholding schemes that adapt confidence and IoU parameters based on scene complexity or object motion.
- Benchmark our pipeline on a broader suite of real-time applications (autonomous driving, surveillance, robotics) to validate generalization and runtime performance.

B. What We Learned

- **Detection accuracy alone is insufficient:** High AP for one class may mask poor performance on others, degrading overall tracking.
- **Tracker robustness requires data consistency:** Without re-identification cues, online trackers struggle with occlusions and false positives.
- **Class imbalance impacts real-world deployment:** Augmentation or weighted loss strategies are essential when minority classes carry equal tracking importance.
- **Modularity accelerates research:** A decoupled pipeline simplifies experimentation with new detectors or trackers.
- **Effective Google Colab workflows:** Learned to persist model weights, reload repositories programmatically, and manage GPU runtimes to streamline development.
- **Mastery of LaTeX formatting:** Gained experience with float placement, bibliography management, hyperlinking, and more for publication ready document.
- **Advanced GitHub usage:** Discovered repository buffer-size tuning, stash-and-pop workflows, and remote synchronization techniques to maintain code consistency across environments.
- **Hyperparameter tuning importance:** Systematically swept confidence and IoU thresholds to balance precision, recall, and identity stability, learning the trade-offs in real-time tracking.
- **Integration of tools and libraries:** Combined OpenCV, Ultralytics YOLO, SORT, and FiftyOne into a cohesive pipeline, understanding each component's configuration and evaluation metrics.

C. Challenges Faced

- **Training custom models:** struggled to configure and run training through train.py in our unused directory on large datasets due to limited hardware resources and long training times.
- **Colab GPU runtime limits:** experiments were interrupted when free GPU quotas expired, forcing us to run locally on CPU, slowing down our testing time.
- **File sharing and synchronization:** Ensuring code, model weights, and results stayed consistent across local machines, Colab, and GitHub proved tricky. We learned about increasing Git's HTTP postBuffer size, but ultimately relied on a well configured `.gitignore` and reproducible environment setups to avoid push/fetch errors and reproduce results consistently across platforms.
- **LaTeX formatting hurdles:** spent time resolving some underfull/overfull boxes, broken references, embedding links, and more for a clean, publication ready document.
- **Language and communication barrier:** collaborating across different first languages introduced misunderstandings but ultimately improved clarity and teamwork, essentially preparing us for future global, cross-cultural collaborations in research and industry.
- **Pipeline integration complexity:** debugging mismatches between YOLO detections, SORT tracking, and evaluation outputs required detailed log analysis.

References

[1] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," 2018. [Online]. Available: https://arxiv.org/pdf/1804.02767v1

[2] Ultralytics, "Yolov8: Real-time object detection models," https://docs.ultralytics.com/models/yolov8/, 2024, accessed: 24 March 2025.

[3] ——, "Yolov11," https://docs.ultralytics.com/models/yolo11/, 2025, accessed: 24 March 2025.

[4] Z. Wang, L. Zheng, Y. Liu, Y. Li, and S. Wang, "Towards real-time multi-object tracking," 2020. [Online]. Available: https://arxiv.org/pdf/1909.12605v2

[5] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft, "Simple online and realtime tracking," 2016. [Online]. Available: https://arxiv.org/pdf/1602.00763v2

[6] N. Wojke, A. Bewley, and D. Paulus, "Simple online and realtime tracking with a deep association metric," 2020. [Online]. Available: https://arxiv.org/pdf/1703.07402v1

[7] the COCO Consortium, "Coco: Common objects in context," https://cocodataset.org/#home, accessed: 24 March 2025.

[8] B. E. Moore, J. J. Corso, and the Voxel51 Team, "Fiftyone: Dataset visualization and model analysis," https://github.com/voxel51/fiftyone, 2020, accessed: 28 March 2025.

[9] M. Team, "Mot15: Multi-object tracking challenge 2015," https://motchallenge.net/data/MOT15/, accessed: 24 March 2025.

[10] Y. Zhang, C. Wang, X. Wang, W. Zeng, and W. Liu, "Fairmot: On the fairness of detection and re-identification in multiple object tracking," 2020. [Online]. Available: https://arxiv.org/pdf/2004.01888v6

[11] Ultralytics, "Ultralytics official website," https://www.ultralytics.com/, 2025, accessed: 24 March 2025.

[12] GeeksforGeeks, "Multiple color detection in real-time using python-opencv," https://www.geeksforgeeks.org/multiple-color-detection-in-real-time-using-python-opencv/, 2024, last updated: 21 Nov. 2024; accessed: 1 April 2025.

[13] J. Redmon, "Yolo: Real-time object detection," 2016. [Online]. Available: https://pjreddie.com/darknet/yolo/

[14] C. Heindl, Toka, J. Valmadre *et al.*, "py-motmetrics: Benchmark multiple object trackers in python," https://github.com/cheind/py-motmetrics, 2022, accessed: 28 March 2025.
[15] Ultralytics, "coco.yaml," https://github.com/ultralytics/ultralytics/blob/main/ultralytics/cfg/datasets/coco.yaml, 2025, accessed: 28 March 2025.
[16] ——, "Objects365 applications," https://docs.ultralytics.com/datasets/detect/objects365/#applications, 2025, accessed: 29 March 2025.
[17] Pixabay, "Pixabay videos," https://pixabay.com/videos/, 2025, accessed: 20 April 2025.
[18] KITTI Vision Benchmark Suite, "Kitti tracking benchmark," https://www.cvlibs.net/datasets/kitti/eval_tracking.php, 2025, accessed: 12 April 2025.
[19] OpenAI, "Chatgpt (gpt-4) language model," https://chat.openai.com/, 2023, accessed: 12 May 2025.