Eye in the Sky: Tracking a Moving Animal Using Deep and Traditional Methods
A Comparison of Deep Learning and Traditional Object Detection in Simulated Environments

Paul Bliemegger
Justin Wenzel

Team Number: 8

May 1, 2025

Project Write Up for
HCI5750: Computational Perception

Iowa State University
Spring 2025

ABSTRACT

Object detection and tracking are popular topics in computer vision and are present in many applications, including surveillance, self-driving cars, and robots. With many approaches being proposed in these areas, the idea for more comparative studies of different situations opens up. This project compares and evaluates traditional and deep-learning-based approaches. For object detection, the model YOLO (deep learning) is compared against HOG+SVM (traditional), while DeepSORT (deep learning) is compared with a Kalman Filter (traditional) approach for tracking.

To perform this comparison, we utilize a controlled experimental setup using Microsoft Airsim, where a drone-mounted camera scans and tracks a virtual wolf that is walking. Performance is measured under different conditions, including a basic motion path, a path deviation, and occlusion. Multiple factors are monitored, including detection accuracy, tracking stability, and computational efficiency. This project provides a comparative analysis of existing techniques in different simulated scenarios, providing insight into each techniques' performance in different scenarios.

1 Introduction

1.1 Problem Statement

Object detection and tracking are a continuing problem in computer vision tasks with a growing interest in critical applications such as surveillance, self-driving cars, and robots. Researchers have proposed various techniques to solve this continuing problem, ranging from traditional machine learning-based methods such as Histogram of Oriented Gradients (HOG) + Support Vector Machines (SVM) to deep learning models such as You Only Look Once (YOLO) for object detection. Tracking methods also include various techniques, including mathematical methods such as the Kalman Filter and deep-learning-based models such as DeepSORT. Deep-learning-based approaches have been a major shift in computer vision performance compared to traditional approaches but still have limitations due to their high computational complexity and processing speeds [1]. A major challenge involved in assessing different object detection and tracking techniques is testing how they behave under different conditions. Common research approaches involve using real-world datasets, but simulation testing can also have benefits, which include controlled variations, reproducibility, and low cost. Microsoft Airsim is a high-fidelity simulation platform that offers a systematic and reproducible environment for testing detection and tracking approaches in a dynamic environment [2]. This is particularly useful for analyzing different object detection and tracking approaches on varying objects, such as animals whose motion is less predictable than a human or car. As the quality of detection plays a critical role in tracking results, it is important to study how different detection and tracking are influenced in diverse scenarios.

1.2 Problem Motivation

While deep-learning-based models have shown success in computer vision tasks, most existing work focuses on evaluating the performance of these proposed ideas against the weaknesses of other approaches in specific tasks. However, real-world accurate detection and tracking is a complex problem involving non-rigid objects such as animals, irregularly moving objects, and a dynamic environment, which are all essential nonspecific tasks for critical computer vision tasks. As a result there is no object detection or tracking approach that is superior in every circumstance, and often, performance is highly sensitive to environmental conditions, complexity of motion, and occlusions. Additionally, real-world testing can be costly and difficult to control specific scenarios, making simulation-based testing a suitable alternative. Microsoft AirSim allows researchers to simulate real-world environments and test different approaches with various conditions before real-world deployment. Testing detection and tracking approaches in a simulated environment provide valuable insights into the approach's performance against dynamic scenarios.

A side affect of our method is experimenting with using simulated data for machine learning models. As data is very important in the current world of machine learning, knowing how well simulated data can be used in these models can help in the process of finding good-quality data. Finding useful data is not trivial. [3]

1.3 Research Objectives

This projects objective is to compare and evaluate classical and deep-learning-based approaches for object detection and tracking in a simulated controlled environment. More precisely this research

aims to compare YOLO and HOG+SVM for object detection, and DeepSORT and Kalman Filter for object tracking. The operation of these techniques will be compared based on the accuracy of how well objects are identified, tracking stability of the identical object across frames, and computational efficiency regarding various patterns of motion, such as a straightforward trajectory of motion, a path deviation such as zig-zag, and occlusion of an object. This project can provide an overview of different object detection and tracking approaches to enhance the knowledge of performance against the same scenarios with different levels of complexity that could map to other potential real-world applications.

2 Related Work

Object detection and tracking have been a long-time problem in computer vision and, to this day, range in complexity from hand-crafted feature extraction to complex deep learning models. A popular classical approach HOG+SVM began because of original feature extraction work known as Histogram of Oriented Gradients (HOG) introduced by Dalal & Triggs [4], which began a foundation for object detection with its combination of involving State Vector Machines. While compared to this traditional method, deep-learning-based detectors such as YOLO found by Redmon, Divvala, Girshick, and Farhadi [5] have continued to evolve over time with researchers developing different variants of the model, including YOLO9000 [6] and YOLOV3 [7]. For object detection and tracking to work together, tracking algorithms must evolve over time also and research has continued with the same advancements as object detection, beginning with traditional methods such as the Kalman Filter introduced by Kalman [8]. The original Kalman filter has evolved over time and is now also used with DeepSORT in a deep learning approach for real-time object tracking that also evolved from SORT, a simple online real-time tracking approach proposed by Bewley, Paulus, and Wojke [9]. These deep learning methods are often found together and have been tailored for many applications, leading them to be insightful techniques to explore. A group of researchers also proposed a deep-learning-based object tracking model that they compared and analyzed trade-offs with other approaches, including DeepSORT [10]. These research papers all show the importance of evolving approaches in object tracking and detection, along with the importance of constantly evaluating and comparing the approaches with each other to help determine strengths and weaknesses for each approach in different use cases.

Microsoft AirSim has been a widely used simulator for testing deep-learning-based computer vision applications in vehicle and aerial robotics. Research conducted by Benoit, Xing, and Tsourdos used AirSim to generate synthetic data for a YOLO model to detect long distance airborne objects with great results [11]. While another group of researchers from ELAN Microelectronics Corporation and the National Taiwan University used AirSim for traffic object detection in virtual environments to simulate a real street view again using YOLO models for object detection [12]. While our work is not with synthetic data specifically or object detection from a vehicle, this research highlights Airsim's potential in our research to provide a controlled and consistent environment for analyzing real-time detection and tracking methods.

As we are dealing with object detection in a simulated environment, the article "Development of a Novel Object Detection System Based on Synthetic Data Generated from Unreal Game Engine"[13] falls into a similar category. Here, they explored the idea of creating synthetic training data for deep learning models from scenes in Unreal Engine 4 and apply these models on real-world

data. This is supposed to simplify the model training process. While our goals differ, their testing of models trained on simulated data gives us some expectations for our results in a similar setting. They also provide possible improvements to their methodology. Another key takeaway from their research is how our approach of using simulated environments for object detection has various practical use cases. To further proof this point of practical use ases, comparisons of other approaches also seem viable as with agent based tracking which this article [14] used to compare the performance of Deep-Q-Networks and Proximity Policy Optimization object tracking agents.

3 Experimental Platform

3.1 Microsoft Airsim

For the control of the drone as well as acquiring images and footage for our further comparison processes we are using Microsoft AirSim. This open-source simulator provides an extensive API with a detailed documentation. Although further support has been deprecated, it was released in a stable state. Microsoft AirSim also provides implementations for Unity and Unreal Engine 4. We decided on the latter, as the Unity implementation is in an experimental state. Access to the API is easily possible via Python. This allows us to take advantage of all the drone's camera sensors (Lidar, Infrared and distance), as well as to its flight controller. While there is the option to define or use more powerful controllers, AirSim provides a "simple controller" which already provides a lot of funtionality out of the box. After retrieving footage from the sensors via the API we are then use OpenCV for image pre- and post-processing. With this we are aiming to achieve fast speeds that allow us to keep real-time object tracking.

3.2 Unreal Engine 4

As mentioned above we are using Unreal Engine 4 to setup the environment of our simulation. While there already exists a newer Unreal Engine 5 version, its predecessor is still a powerful simulation tool. It provides us with all the functionalities of a commercially viable real-time game engine, with high performance and the possibility of a somewhat painless environmental setup. As is it still actively being used for creating games and simulations to this day, many out of the box assets exist, helping us create complex and almost photo-realistic environments.

Figure 1: Using Microsoft AirSim to place a drone in a complex environment

Figure 2: Footage of the drone camera retrieved using Python and OpenCV

3.3 OpenCV and Python

The actual implementation and training of our models is done using Python and OpenCV. The latter is a standard and widely used image processing library with various tools necessary to properly pre-process the images taken from the drone. This will ensure that our models will be as effective as possible. While there also exists an implementation of OpenCV in C++, for our purposes we believe using Python allows us more flexibility in the interaction with our Machine Learning models.

4 Methodology

All code referenced in this section is available publicly in our GitHub repository [15]. This includes dataset generation, model training, evaluation scripts, and the full implementation of the object detection and tracking pipeline.

4.1 Data Collection

Working in Microsoft AirSim, we had to construct our own dataset to train our models for the environment. Our object of interest is a gray wolf actor with multiple different animations that was created by PROTOFACTOR INC [16]. The collection process began by creating a new JSON file with defined camera locations set by us, arranged in a top-down view, and the wolf was then placed in the center of the cameras.

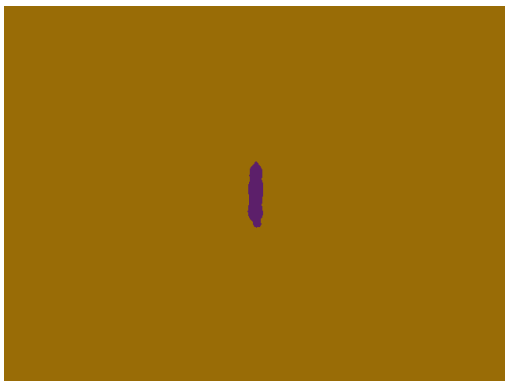
The entire image extraction process was completed by a single Python script available in our repository called ”Data_Creation.py”. The script would use Unreal Engine 4 to assign a unique semantic ID number to the wolf. Then AirSim API would perform an image request with a set delay between frames. It would request two 640x480 images, an RGB-based image, and a semantic image from the camera. The semantic image was then used to generate bounding box annotations for the wolf shown in Fig. 3. Pixel regions matching the wolf's set ID were extracted, and a bounding box was computed around the wolf and then stored in a YOLO answer format.

To create diversity in aspects of the data, the camera setup and the wolf were moved around to different locations. At each location, every camera would take twenty images, then the weather would be changed, and another twenty would be taken. The different weather conditions consisted

of sunny, rainy, and snowy, which were used to help create a more diverse dataset for training, offering changing in lighting, and wolf appearance. Along with different locations, some locations were picked as negative areas where images were taken with no visible wolves around.

This script provided an automated way to extract, annotate, and save images that were then used for model training in the future. Although the script was able to automate the process, locations were still hand-picked and set up randomly around the map.

((a)) Original RGB Image



((b)) Semantic Segmentation

((c)) Bounding Box Overlay

Figure 3: Examples of a captured frame.

4.2 Object Detection Approaches

4.2.1 YOLO (You Only Look Once)

YOLO is a one-pass object detection and classification deep learning model that is utilized for detecting and classifying objects from a single pass through an input image and is, therefore, a good choice for real-time processing applications [5]. We used the YOLOv11n model due to it being the lightweight version of the most recent released YOLO model, creating a balance between speed and accuracy. The model was trained using the Ultralytics implementation of YOLO for easy integration and usage. Our final training script is named "YOLO_Train.py" in the provided GitHub.

The model was trained for 100 epochs using a subset of random images from the custom dataset described in Section 4.1. This subset of the original dataset included a variety of different images, using images from each location, camera, wolf animation, and weather condition to make the

dataset. The model was trained on 640x640 input images, a batch size of 16, and used an initial learning rate of 0.001 and a learning rate fraction of 0.01 to adjust the learning rate during training.

Also, to improve the model's generalization ability, data augmentation was applied to input images during training. This included color augmentation, geometric augmentation, and flipping. This allowed for slight changes to be made in the wolf's appearance involving orientation and color, mostly.

Lastly, it should be noted that the final experiment only involves the walking animation. But the training dataset for YOLO did include other wolf animations, such as running and biting. These different animations were collected using the same dataset collection pipeline, only we changed the animation that was set for the wolf. This was an intentional choice to try and benefit YOLO, because of its depth and parameter count, it needs a more diverse training distribution to limit overfitting and improve generalization during inference.

4.2.2 HOG + SVM (Histogram of Oriented Gradients with Support Vector Machine)

HOG+SVM is a traditional computer vision approach that uses extracted gradient-based features to capture edge structures, then is followed by classification using an SVM [17].In our scenario, the detector was explored as a non-deep-learning baseline approach for object detection. This is because it relies on hand-crafted gradient-based features extracted from fixed patch sizes instead of learning its own representations of the data.

To explore the feasibility of HOG+SVM in our approach, we used a training pipeline to train multiple models using different hyperparameter configurations. This training adjusted different parameters, which included different block structures, cell sizes, and patch sizes to capture different features from the HOG outputs. Additionally, different regularization strengths were applied to a linear SVM.

The dataset used for training was constructed by taking positive and negative patches from the dataset discussed in section 4.1. There were fewer examples in this dataset compared to YOLO due to HOG+SVM's inability to handle complex datasets. Therefore, it was only trained on images of different weather conditions and the wolf walk animation. This allowed for variations in the photos but also kept the models dataset simple and tailored towards our object of test.

The final model that we used in our tests was selected through a more observational-based and small metric-based testing. We conducted a simple test that adjusted step sizes and detection thresholds for each model over some images. We then identified the model that performed the best on the small sample of images and used that as our main model in our traditional object detection and tracking pipeline.

4.3 Object Tracking Approaches

4.3.1 DeepSORT

DeepSORT is a deep-learning-based tracker that uses motion and appearance information by feature extraction and using a convolutional neural network (CNN) to maintain object identity consistently, which is extended from the Simple Online and Realtime Tracking (SORT) method [18]. In our pipeline, we used the official DeepSORT implementation directly from one of the original

authors, Wojke's GitHub [19].

In our pipeline, we would pass the resulting bounding box information from object detection into the DeepSORT tracker along with confidence scores for each box and the original frame. The Wojke implementation used these inputs to create different tracks for each object, as long as the detected object qualified for a track to be made. To qualify, an object had to appear consistently multiple times. This limited false positive tracks. Once the track of the object was initialized, it would continue to be updated whenever detected, and a consisted object ID would be assigned to it based on its features. If the object was missing in a few frames, DeepSORT would use its internal Kalman filter to try and continue to predict where the object was to help reduce potential ID-switching and would update when the object was detected again.

4.3.2 Kalman Filter

Kalman Filter is a traditional lightweight object tracker that predicts object location over time based on past estimates and updates its predictions when the object is detected again [20]. In our project, we implemented a custom tracker using OpenCV's KalmanFilter class. Each object track begins by holding the object in an 8-dimensional vector that holds the center X and Y, width, height, and their initial velocities.

Tracks are maintained in our setup using a wrapper class, KalmanBoxTracker, which provides functions to maintain and update a single tracked object's lifecycle. While the SimpleKalmanTracker class maintains all objects being tracked, when an update is called, it compares all inputted detections to current tracks. In the update function in SimpleKalmanTracker, a request is made to increment predictions. Then all current predictions of the tracks are compared with the inputted detections, and if any align with the prediction box for the track to some IoU threshold (0.3), it is then considered a valid match. Once matched, the predictions will be updated to match the detection.

If a track does not receive a match, its age is increased. If a track's age exceeds the age limit, which in our case is 5 frames, it will be erased. While any new detections that can not be matched with a current track will begin to initialize a new track. But to ensure many false-positive tracks are not made before the track can begin returning results to the output, it must meet a hit streak of 2, where it is detected and matched at least twice back to back; this will suppress false positives and potential noise.

4.4 Performance Evaluation Metrics

The robustness of these object detection and tracking approaches is being assessed based on three primary evaluation metrics: detection accuracy, tracking stability, and computation efficiency. The implementation we used to record these metrics during testing is available in the "Metrics.py" script in the project repository.

Detection accuracy was computed using three main metrics, which include precision, recall, and F1 score. Precision measures the positive prediction accuracy as a ratio of the true positive detections to all predictions made. Recall measures the model's ability to identify all instances that are relevant to what is trying to be detected, calculated as the ratio of true positives to the total number of ground truth instances. The F1 Score then serves as a harmonic mean between precision and

recall, offering a balance performance perspective. Ground truth labels are extracted in real time using AirSim's semantic segmentation images and OpenCV, then applies the same technique used in creating the truth labels for the model training dataset, as discussed earlier.

Tracking stability was measured by monitoring if the tracker was able to maintain the same ID across each frame for the wolf. During each frame, the output IDs from the tracker would be compared to the previous track ID that was given to the wolf from the previous frame's track output. Then, based on which output track from the current frame aligned with the ground truth location for the wolf extracted from the semantic image would decide if an ID switch occurred. The metric only considered switched to the wolf object and not other potentially false-positive objects that could influence the ID switch count.

Computation efficiency was measured by recording the average time it took to process a frame through the entire detection and tracking pipeline. This was represented as the frames-per-second (FPS) of the system. The timer was started when the image first entered the pipeline and ended when the tracking loop ended to ensure end-to-end performance was being monitored. This metric allowed for the evaluation of the real-time applicability of each method.

4.5 Modular Pipeline Architecture

Earlier, we introduced the different object detection and tracking techniques conceptually and or their training processes used; this section will now outline how each method was implemented in a modular pipeline for flexibility in our experimentation, and the hyperparameters we used for each technique. This consisted of designing multiple object classes via Python that could be imported as needed, making testing easier. The setup was done by multiple different components that were isolated from each other, including detection, tracking, evaluation, and drone control. Using this approach, we were able to implement multiple different main scripts for testing with very minimal changes to the overall system.

4.5.1 Detection Modules

Both YOLO and HOG+SVM were implementation modularly to be interchangeable. Each detection method was implemented as its own Python class that matched a common interface between one another. Both modules are available on GitHub named "YOLO_Tracker.py" and "HOGSVM_Tracker.py"

- **YOLO Integration:**
  - Integrated via the Ultralytics Python API for predictions.
  - Offers CUDA support for GPU or defaults to a CPU
  - Inference-time parameters included: confidence threshold = 0.3, input resolution = 640×640.
- **HOG+SVM Integration:**
  - Custom sliding window and image pyramid used for detection, implemented using OpenCV and uses skimage for HOG feature extraction.
  - Applies non-max suppression (NMS) to help reduce false positives.

- Inference-time parameters included: window size = 64×144, step size = 32 px, score threshold = 2.0, scale factor = 1.25.

Both detector modules were modular and consistent with one another, ensuring a consistent interface to interact with the tracking modules in the next step.

4.5.2 Tracking Modules

We implemented a DeepSORT wrapper module and the Kalman Filter in two separate interchangeable modules. Both followed the same input and output format for simplicity and utilized the same class setup for simplicity. Both modules are available on GitHub, named "DeepSORT_Tracker.py" and "Kalman_Filter.py".

- **DeepSORT Integration:**
  - Wrapped in a custom class that internally calls the original DeepSORT implementation [19].
  - Accepts detections with confidence scores and the full RGB frame to extract appearance features.
  - Inference-time parameters included: maximum cosine distance = 0.8, all other parameters were left as DeepSORT's original values.
  - Used the original pre-trained CNN for feature extraction due to time constraints; increased max_dist to 0.8 to compensate for potentially lower feature specificity.
- **Kalman Filter Integration:**
  - Implemented using OpenCV's KalmanFilter class for simpler interaction and implementation.
  - Tracks undetected objects' state over time using motion-based predictions.
  - Inference-time parameters included: IoU threshold = 0.3, maximum age = 5 frames before termination, minimum hit streak = 2 before starting a track.

Both tracking modules are consistent in setup and usage compared to one another. They both provided outputs in the same format, making updating drone location and metrics recording easier.

4.5.3 Evaluation and Metrics Module

To monitor performance, a separate metrics module was made to assist in logging and tracking each performance metric during testing. This module is available on GitHub as "Metrics.py".

- **Detection Accuracy:**
  - Compared predicted bounding boxes with a ground truth mask extracted from a semantic image from AirSim

  - Used an IoU threshold of 0.5 to determine if a detection was considered valid with the ground truth.
  - Tracks common object detection metrics, including precision, recall, and F1 Score, also offers other metrics, including total frames, and number of false positives.

- **ID Switches:**
  - Used the same idea as detection accuracy using the semantic image.
  - An ID switch was recorded whenever the tracking module assigned a new ID to the ground truth location compared to the previous frame, with an IoU of 0.5 threshold.

- **Computational Efficiency:**
  - Measured processing time for detection and tracking for each frame using a simple timer.
  - FPS was computed by computing the average for processing each frame

The metrics module for evaluating performance made it easy to monitor performance during each test and ensured consistency. Also provided a simple print function that would display and save final metrics.

4.5.4 Drone Controller Module

The drone controller module was implemented to interact with the AirSim drone in Unreal by using a simple update function. The update function would move the drone to a desired location depending on the error of prediction from the center of the wolf to the center of the drone's camera image. While trying to center the wolf in the camera's view, the controller would also ensure the drone maintains its preset altitude. In this implementation, the drone controller has no obstacle avoidance.

4.5.5 Main Script Design

Using all the modules outlined previously, a main script was constructed for different tests, allowing for a combination of different pipelines to be made. Each script would follow a consistent structure:

1. **Initialization** All required components would be instantiated, and the connection to AirSim would be established.
2. **Acquire Image** Using the AirSim API, a request for an RGB and semantic image would be made.
3. **Detection and Tracking** The RGB image was processed by the detection module, and the output detections were sent to the tracking module to provide an ID and update tracks.
4. **Control and Visualization** Drone position was updated using the tracking outputs, then OpenCV was used to display annotated results for the frame.

5. **Evaluation** Throughout the loop, evaluation metrics would be called and updated throughout the frame process, recording each step.

An example of the modular pipeline architecture can be found in Figure 4, showing our object detection and tracking pipeline during testing.

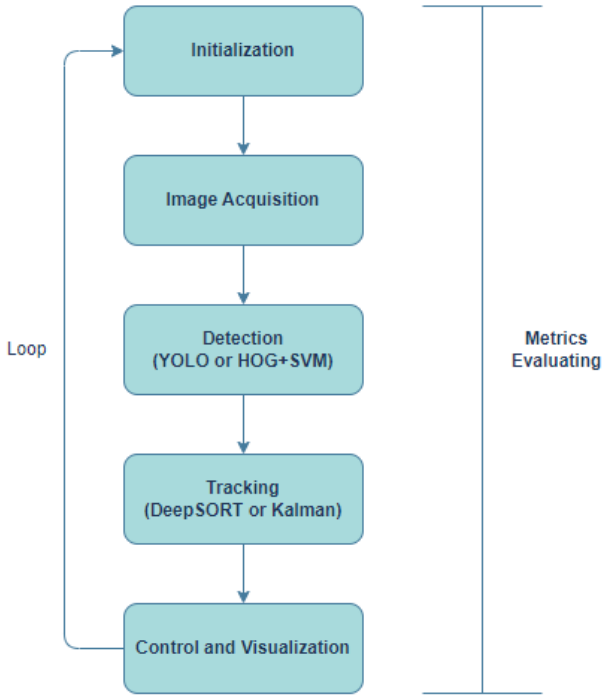


Figure 4: Flow diagram of the modular pipeline architecture.

4.6 Experimental Scenarios

To evaluate each detection and tracking pipeline, we designed three controlled scenarios within a city environment demo map provided in an assets package by PurePolygons [21]. Using the original demo map, we made slight changes to each environment along the downtown path to support different experimental conditions for each scenario. All tests used the same gray wolf actor as defined earlier, using its walking animation and moving along a predefined path via a spline component in Unreal Engine. Both the drone and the wolf would start at the same position and begin testing from there.

- **Basic Path**: A straight line path along the downtown in an open environment with minimal background objects, no obstacles, and minimal lighting changes. This is used as a baseline, simple scenario (see Fig. 5(a)).
- **Zig-Zag Path**: Object moves along a path that involves more background objects, turns to pass around objects, and more changes in lighting caused by shadows from trees (see Fig. 5(b)).

- **Occlusion Path**: Follows a simpler path with very minimal turns and focuses more on object occlusion using tent awings to obscure the object for different periods (see Fig. 5(c)).

Each scenario was designed to simulate different tracking challenges that can be experienced in the real world. These different setups were used to evaluate the performance of each method under a range of conditions.

((a)) Basic Path

((b)) Zig-Zag Path

((c)) Occlusion Path

Figure 5: Example of each testing path area.

4.7 Challenges and Limitations

While using a simulated platform such as Microsoft AirSim to assist in experiments provides a controlled and reproducible environment for evaluating object detection and tracking approaches, there are a few limitations to be considered when interpreting the results. A primary concern is the inconsistency between synthetic and real performance, as simulated worlds do not represent true lighting conditions, texture variations, or sensor noise, which all can affect the ability for deep learning models to generalize and traditional approaches to interpret raw data. While deep learning models like YOLO and DeepSORT are available and widely used, they are computationally more expensive, which can limit real-time applicability. In comparison, traditional methods like HOG+SVM and Kalman Filter might be more computationally lighter, but can break down in complex visual situations. The simulation, even with potential drawbacks, allows for a direct comparison between detection and tracking algorithms and gives a hint about their potential strengths and weaknesses in different conditions. The following section presents our expectations of results before we performed the tests.

5 Expected Trends and Performance Results

5.1 YOLO vs. HOG+SVM

As per existing research by Kaplan & Şaykol (2018) [22], YOLO is expected to outperform HOG+SVM when it comes to detection accuracy. This is an expected trend based on how each method processes visual information. Since YOLO uses a deep neural network that allows it to learn a hierarchical representation of features, it can better allow it to recognize objects in various situations, including size, orientation, and complicated background. HOG+SVM, on the other hand, relies on pre-computed edge-based features extracted via the Histogram of Oriented Gradients method, then is followed by a Support Vector Machine classification. While effective in simple cases, HOG+SVM could struggle with changes in an object's appearance and background clutter.

However, a surprising finding of Kaplan & Şaykol (2018) [22] is that YOLO also ran faster during processing when it came to FPS compared to HOG+SVM. Contrary to normal expectations that a traditional method would be less computationally complex, this suggests that optimally tuned models can be superior to more traditional methods depending on the scenario and resources available.

5.2 DeepSORT vs. Kalman Filter

Comparing existing research to determine how DeepSORT might compare to the Kalman Filter in accuracy can be inferred by DeepSORTs strong pairing with YOLO. One research that used YOLO and DeepSORT to detect wrong-way vehicles managed to have near 100% accuracy [23]. Using this common pairing of DeepSORT and YOLO leads to the belief that DeepSORT will outperform the Kalman Filter in tracking stability.

Previously, with YOLO and HOG+SVM, the paper noted that the deep learning model was faster than the traditional approach. It is believed that the outcome might not happen in this case with DeepSORT and the Kalman Filter as a strong reason for HOG+SVM being slower was due to a sliding window for detection. The Kalman Filter is more simple operation that should not require a lot of computational cost and prove itself to be a strong candidate in real-time performance. While DeepSORT is a deep neural network built on top of SORT and the Kalman Filter itself, which will draw drawbacks in performance speed.

6 Results

Table 1: Basic Path (1) – Metrics Comparison

Method	FPS	Precision (%)	Recall (%)	F1 Score (%)	FP	ID Sw.	Frames	Correct	Total Det.	GT
YOLO+DeepSORT	10.07	73.61	66.76	70.02	210	23	1079	717	974	1074
HOG_SVM+Kalman	0.73	0.00	0.00	0.00	5	0	18	0	5	14
YOLO+Kalman	10.38	82.21	81.54	81.87	140	4	1094	892	1085	1094

Table 2: Zig-Zag Path (2) – Metrics Comparison

Method	FPS	Precision (%)	Recall (%)	F1 Score (%)	FP	ID Sw.	Frames	Correct	Total Det.	GT
YOLO+DeepSORT	9.05	81.03	75.27	78.04	145	18	1028	773	954	1027
HOG_SVM+Kalman	0.70	0.00	0.00	0.00	1	0	14	0	1	11
YOLO+Kalman	9.43	82.12	76.68	79.31	142	9	1042	799	973	1042

Table 3: Occlusion Path (3) – Metrics Comparison

Method	FPS	Precision (%)	Recall (%)	F1 Score (%)	FP	ID Sw.	Frames	Correct	Total Det.	GT
YOLO+DeepSORT	8.40	93.29	85.11	89.01	20	1	392	320	343	376
HOG_SVM+Kalman	0.70	0.00	0.00	0.00	1	0	15	0	1	10
YOLO+Kalman	8.50	95.21	88.54	91.75	25	3	675	556	584	628

For our experimental setup, HOG_SVM+Kalman did not perform as expected, and we still wanted to ensure a proper comparison between the Kalman Filter and DeepSORT. We decided to create two YOLO pipelines, with each one containing a different tracking method.

6.1 Basic Path Results

Our first test scenario, also known as the Basic Path, provided a clear baseline for comparing each pipeline under a simple, consistent, unobstructed scenario. YOLO+Kalman achieved the highest F1 Score at 81.87%, which outperformed YOLO+DeepSORT's 70.02% F1 Score. For FPS, YOLO+Kalman has a slightly higher FPS of 10.38, while YOLO+DeepSORT has an FPS of 10.07, closely following each other. In terms of the ID switches, YOLO+Kalman had the lowest switches with only 4, while YOLO+DeepSORT had 23 ID switches during testing. The HOG_SVM+Kalman failed in all metrics with 0 detection accuracy, a low FPS of 0.73, and 0 ID switches.

6.2 Zig-Zag Path Results

Our second test scenario consisted of more turns and lighting changes, creating a more advanced scenario for testing. YOLO+Kalman had the highest F1 Score of 79.31% but YOLO+DeepSORT was not far behind with an F1 Score of 78.04%. For FPS, both pipelines with YOLO had a similar performance, achieving above 9 FPS, with YOLO+Kalman achieving 9.43 FPS as the highest. For ID switches YOLO+Kalman had the lowest number of switches with only 9, while YOLO+DeepSORT had the most at 18 switches. The HOG_SVM+Kalman failed in all metrics with 0 detection accuracy, an FPS of 0.7, and 0 ID switches.

6.3 Occlusion Path Results

The third and final test, involving simple occlusion cases, was created to test a challenge that object detection and tracking that can face. YOLO+Kalman once again produced the highest F1 Score at 91.75%, with YOLO+DeepSORT following behind at 89.01%. While the FPS between both YOLO pipelines achieved nearly identical performance, only differing by 0.1 FPS. Finally, YOLO+Kalman had 3 ID switches while YOLO+DeepSORT had only 1 ID switch. Lastly, HOG_SVM+Kalman was not able to perform again and had 0 in nearly all metrics and an FPS of

0.7 again.

6.4 Qualitative Detection and Tracking Examples

To support the discussion in Section 7 and our quantitative results, we have provided five frames extracted during experiments, where each demonstrates a different behaviour. These example frames show different forms of behaviour experienced during testing, including correct detections (see Fig. 6), duplicate bounding boxes (see Fig. 7), false positives (see Fig. 8), Wolf drift from frame misses (see Fig. 9), and an example of the start of an occlusion (see Fig. 10). These all provide a visual example for the performance metrics and assist in supporting the discussion points in the next sections, which analyze results and interpret different behaviours seen during the experiments.

Figure 6: Successful tracking example in the Basic Path scenario using YOLO+Kalman.

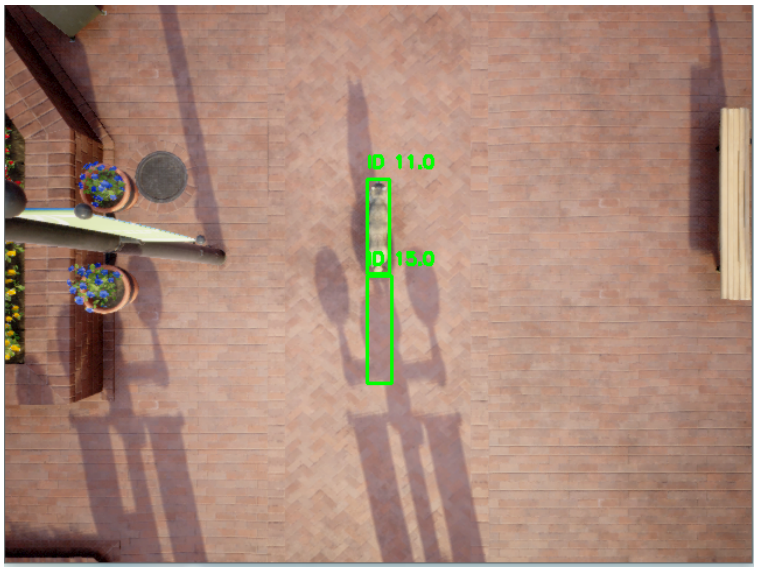


Figure 7: Double detection issue: YOLO detects two bounding boxes, and tracking applies separate IDs.

Figure 8: False positive detection in YOLO+DeepSORT pipeline.

Figure 9: Wolf drift example: DeepSORT loses detections from YOLO for a few frames.

Figure 10: Partial occlusion example, tracker should switch to prediction.

7 Discussion

7.1 Overview of Objectives

Stating the primary objectives of this project again, we aim to evaluate the performance of different object detection and tracking pipelines in a simulated drone-based tracking scenario. We want to compare a traditional and a deep-learning-based pipeline by comparing their accuracy, speed, and overall robustness to different scenarios. These metrics are measured through frame processing speed, detection precision, and recall, along with tracking stability through ID switches. These

metrics allow us to open a discussion about the potential usage of each pipeline in a real-time environment.

7.2 Baseline Limitations (HOG+SVM)

Before diving into an open discussion about performance, the limitations of HOG+SVM need to be addressed. We originally believed the detection method of HOG+SVM to be less computationally expensive, but it proved to be the opposite. The image pyramid scaling and sliding window that was applied to each image caused the HOG+SVM pipeline to not keep up. Maintaining an FPS below 0.75 in all tests. The pipeline failed right at the start, being able to make some detections, but nothing meaningful or fast enough to keep up with the simulation, resulting in early termination of the test.

This does not mean that HOG+SVM is entirely cut out as an option; in our instance, the HOG+SVM is not optimized, it could benefit from a multithreaded setup or a more tailored architecture to boost its processing speed. For example, researchers Wasal and Kryjak from AGH University of Science and Technology were able to achieve 60 FPS on 4K images with HOG+SVM on specialized hardware [4].

For our tests, HOG+SVM was not a viable pipeline due to its low precision and slow runtime performance. However, its inclusion in this report provides insight into different methods besides deep-learning-based implementations.

7.3 Detection and Tracking Performance

Reviewing results, it is obvious that YOLO-based pipelines outperformed the HOG+SVM-based pipeline in all metrics, including detection accuracy, tracking consistency, and computational efficiency. Performance differences discussed here will only pertain to the YOLO-based pipelines. Between the tests, it can be noted that tests 1 and 2 had a much more significant frame count compared to test 3, which could influence some of the higher resulting values recorded in test 3. This is because no method was able to pass all occlusion tests at this time.

For detection accuracy, YOLO consistently achieved the highest recall and precision, confirming its ability to detect the wolf reliably across all the different test environments. With YOLO+Kalman achieving the highest F1 Score of 91.75% and YOLO+DeepSORT not far behind with a high of 89.01%. The YOLO+Kalman pipeline consistently had a higher precision, recall, and F1 Score. These higher results are most likely causes by the pipeline being able to consistently keep the wolf in the center of the frame, allowing the model to have easier visibility of the object, caused by the Kalman Filter filling in gaps between missed detections.

Tracking performance also differed between the two YOLO pipelines, with YOLO+Kalman still coming out on top with some of the lowest switches most of the time. It was able to have a lower number of ID switches in the Basic and Zig-Zag path tests. Normally, we would expect DeepSORT to have lower ID switches, but it's thought to be caused by the DeepSORT CNN model not being retrained for our scenario. We tried to compensate by increasing the max distance allowed in differences between features found by the CNN to try and help realign detections more easily, but this proved not to compensate as we thought. While the YOLO+DeepSORT pipeline did have the lowest ID switches in test 3, it should also be noted that it was not able to pass the first occlusion,

while YOLO+Kalman made it to the second occlusion.

Finally, with computational performance, both YOLO pipelines achieved very similar performance with consistent rates above 8 FPS. While the pipeline with the Kalman Filter was faster, this was expected as DeepSORT involves a Kalman Filter and a CNN model to perform tracking. While we expected the CNN to slow down the YOLO+DeepSORT pipeline more it did not, allowing for both YOLO pipelines to have near equal computational performance. It should be noted that the highest FPS achieved of only 10.38 FPS might not truly be considered as truly real-time as a 30+ FPS system, but it should be noted there are still ways to boost our system's performance, including a more optimized pipeline, removing metric recording in the pipeline and optimizing the YOLO model.

Overall, the results shown here initially point towards YOLO+Kalman Filter being a better pipeline for object detection in a tracking drone-based scenario. But both pipelines had a performance output that allowed for a good comparison.

7.4 Qualitative Behaviour Observations

Besides the quantitative performance metrics, we noticed several behavioral observations that should be noted that occurred during testing that also influenced performance.

One issue that would sometimes occur was false-positive tracks that would trigger because of multiple false-positive detections back to back from the detector. This often occurred in some cluttered scenarios, lighting changes, or sometimes randomly in the middle of the image. While these false-positives are outcomes from the detector, they would sometimes create tracks that would persist on the display for a short time until the lifetime of the track would end because of not having consistent updates.

Another observation was how differently both trackers handled detection failures. DeepSORT used a strict gating mechanism and would only return tracks that are both confirmed and have recently experienced an update from detection. While this might help prevent stale outputs or drifting predictions because of DeepSORT's high default track life of 70 frames, it was a problematic feature in our scenario that involved a moving camera performing the detections and updating drone movement. When YOLO would miss a detection due to occlusion or just poor detection abilities, DeepSORT would stop sending results for the track. This caused the drone to stop moving further or follow another track. This halt in movement raised other issues, such as causing the object to become uncentered (see Fig. 9), which could have caused the lower accuracy in detection, as the object was often not centered. This also caused the YOLO+DeepSORT pipeline to not pass the first occlusion in test 3 because the lack of detection would cause the drone to not pass over the occlusion and meet the wolf on the other side.

In contrast to DeepSORT, the Kalman Filter used these missed detections to update the drone controller by pushing out predictions to fill the gap until the next update or the track ages out. This would allow the drone to continue moving to predicted positions, but it also carried the risk of losing the track and following another track that might have been a false positive causing the drone to drift in another direction. Most often, this behaviour of using predictions to fill in gaps would help keep the wolf centered during missed predictions. It also allowed for the YOLO+Kalman pipeline to pass the first occlusion because, when not detected, the predictions pushed the drone

past the occlusion onto the other side to find the wolf again.

7.5 Future Work

While the results from these experiments were promising for YOLO-based pipelines, there are areas of improvement that can be expanded on in future work. First, the HOG+SVM module in our implementation should be optimized more thoroughly for it to continue to exist as a traditional method in experiments. The current implementation is limited by a single-threaded execution setup and an unoptimized image traversal function. Using different techniques to improve performance, such as hardware accelerators or a multithreaded window evaluation system, could help achieve a more usable performance for testing.

Secondly, DeepSORT should be further explored in future studies. Here we wanted to use as close to the original proposed DeepSORT, but this proved to be a bad decision, and with more time, we should have at least retrained the CNN. A future study with a more domain-adapted CNN model would likely improve ID consistency and performance for DeepSORT. DeepSORT should also be further explored and more tailored for a camera-moving tracking scenario to handle missed detections in another manner.

Additionally, the controller module we implemented is very limited as it only follows a simple 2D screen-center projection of the target. Future works expand this to include obstacle avoidance and 3D path planning to make the drone controller more robust for different environments. The current drone controller implementation is not sufficient for real-world navigation or cluttered environments that the drone could hit.

Lastly, further testing in difference simulated environments would be an interesting expansion. Expanding to more scenarios, such as forests, or different weather or time of day scenarios, could provide stronger insight into the performance and generalization ability of the detectors and the overall performance of the pipelines.

8 Conclusion

In this project, we aimed to compare and evaluate traditional and deep-learning-based object detection and tracking methods in a simulated drone-following scenario within Microsoft AirSim. We implemented, trained, and tested multiple methods, including YOLO, HOG+SVM detectors, with DeepSORT and Kalman Filter trackers.

Our findings showed that deep-learning-based mixed with a classical method, YOLO+Kalman, had the best performance in all metrics, such as precision, recall, F1 Score, and FPS. With the Kalman Filter's setup to fill in gaps with predictions on missed detections allowed for smoother tracking, and kept the object centered for detection. DeepSORT's more restrictive update strategy sometimes restricted its ability to maintain tracks during detection dropoffs, which sometimes affected the drone's ability to follow the wolf object and provide a stable centered environment for future detections.

While HOG+SVM was intended to be a traditional method that was going to be used as a simple baseline, it failed to operate at a level of performance that was measurable. Due to its high computational expense in our setup. Along with its simple structure, it struggled in complex scenes

and detecting the wolf. HOG+SVM remains open for discussion as an option for a detector in a real-time drone application such as our scenario.

Also, our modular setup allowed for easy scripting and created an environment that could easily be interchanged or expanded with other detection and tracking methods. As future work could use this pipeline to expand and try to provide better FPS results, or improve the generalization abilities of YOLO in our scene. Along with fine-tuning DeepSORT more for our use case by retraining the CNN model for the domain environment we defined. Along with further environment testing or expanding to multi-object tracking.

Lastly, this project demonstrates that system context matters when deciding what techniques to use in the construction of the object detection and tracking pipeline. As we saw here, while DeepSORT is presented as a strong alternative to the Kalman Filter, it may be more suitable for static camera scenarios and not designed for scenarios where the tracker drives the movement of the camera in the scenario. This highlights the importance of choosing your implementation techniques based on the whole system, and not on performance metrics.

REFERENCES

[1] N. O. Mahony, S. Campbell, A. Carvalho, S. Harapanahalli, G. Velasco-Hernandez, L. Krpalkova, D. Riordan, and J. Walsh, "Deep learning vs. traditional computer vision," in *Advances in Computer Vision: Proceedings of the 2019 Computer Vision Conference (CVC), Volume 1*. Springer Nature Switzerland AG, 2020, pp. 128–144. [Online]. Available: https://doi.org/10.48550/arXiv.1910.13796

[2] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," 2017. [Online]. Available: https://arxiv.org/abs/1705.05065

[3] O. Zendel, K. Honauer, M. Murschitz, M. Humenberger, and G. Fernandez Dominguez, "Analyzing computer vision data - the good, the bad and the ugly," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[4] M. Wasala and T. Kryjak, "Real-time hog+svm based object detection using soc fpga for a uhd video stream," in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, 2022, pp. 1–6.

[5] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," 2016. [Online]. Available: https://arxiv.org/abs/1506.02640

[6] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," 2016. [Online]. Available: https://arxiv.org/abs/1612.08242

[7] ——, "Yolov3: An incremental improvement," 2018. [Online]. Available: https://arxiv.org/abs/1804.02767

[8] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[9] N. Wojke, A. Bewley, and D. Paulus, "Simple online and realtime tracking with a deep association metric," 2017. [Online]. Available: https://arxiv.org/abs/1703.07402

[10] D. A. Balamurugan, T. T. Khoei, and A. Singh, "An efficient deep learning-based model for detecting and tracking multiple objects," in *2024 2nd International Conference on Artificial Intelligence, Blockchain, and Internet of Things (AIBThings)*, 2024, pp. 1–6.

[11] P. Benoit, Y. Xing, and A. Tsourdos, "Eyes-out airborne object detector for pilots situational awareness," in *2024 IEEE Aerospace Conference*, 2024, pp. 1–8.

[12] B.-Y. Lin, C.-S. Huang, J.-M. Lin, P.-H. Liu, and K.-T. Lai, "Traffic object detection in virtual environments," in *2023 International Conference on Consumer Electronics - Taiwan (ICCE-Taiwan)*, 2023, pp. 245–246.

[13] I. Rasmussen, S. Kvalsvik, P.-A. Andersen, T. N. Aune, and D. Hagen, "Development of a novel object detection system based on synthetic data generated from unreal game engine," *Applied Sciences*, vol. 12, no. 17, 2022, visited on 2025-03-15. [Online]. Available: https://www.mdpi.com/2076-3417/12/17/8534

[14] K. Farkhodov, S.-H. Lee, J. Platos, and K.-R. Kwon, "Deep reinforcement learning tf-agent-based object tracking with virtual autonomous drone in a game engine," *IEEE Access*, vol. 11, pp. 124 129–124 138, 2023, visited on 2025-03-15.

[15] J. Wenzel, "Cpre5750 final project modular vision pipeline," https://github.com/Jwenzel1001/CPRE5750-FinalProject, 2025, accessed: 2025-05-04.

[16] PROTOFACTOR INC, "Fantasy wolf 3d model - unreal engine asset," https://www.fab.com/listings/2dd7964c-a601-4264-a53d-465dcae1644c, 2024, accessed: 2025-05-04.

[17] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, 2005, pp. 886–893 vol. 1.

[18] N. Wojke, A. Bewley, and D. Paulus, "Simple online and realtime tracking with a deep association metric," 2017. [Online]. Available: https://arxiv.org/abs/1703.07402

[19] N. Wojke, "deep_sort," https://github.com/nwojke/deep_sort, 2019, accessed: 2025-05-04.

[20] P. R. Gunjal, B. R. Gunjal, H. A. Shinde, S. M. Vanam, and S. S. Aher, "Moving object tracking using kalman filter," in *2018 International Conference On Advances in Communication and Computing Technology (ICACCT)*, 2018, pp. 544–547.

[21] PurePolygon, "Environment asset pack - unreal engine assets," https://www.fab.com/listings/0faf8b5d-7a5f-4fee-a297-7a8efaba8896, 2024, accessed: 2025-05-04.

[22] Ö. Kaplan and E. Saykol, "Comparison of support vector machines and deep learning for vehicle detection." in *RTA-CSIT*, 2018, pp. 64–69.

[23] A. B. Siddique Mahi, F. S. Eshita, and T. Helaly, "An automated system for wrong-way vehicle detection using yolo and deepsort," in *2023 5th International Conference on Sustainable Technologies for Industry 5.0 (STI)*, 2023, pp. 1–6.

Appendix A Additional Resources

The following resources provide further information on the tools and frameworks used in this project:

- **Microsoft AirSim Documentation**: Official guide for understanding and using Microsoft AirSim for drone and autonomous vehicle simulation. Available at: `https://microsoft.github.io/AirSim/`
- **Unreal Engine Tutorials and Samples**: Epic Games' official documentation for working with Unreal Engine, which is used as the foundation for Microsoft AirSim. Available at: `https://dev.epicgames.com/documentation/en-us/unreal-engine/samples-and`