Visual Object Tracking 2024

Hugo Deplagne

January 2024

Contents

1 Introduction

In this report, we explore the intricacies of a sophisticated Visual Object Tracking system. Our focus lies in delving into the technical aspects of the tracking algorithm, discussing improvements and challenges encountered, and analyzing the quality metrics to assess performance enhancements.

2 Description of the Tracking Algorithm

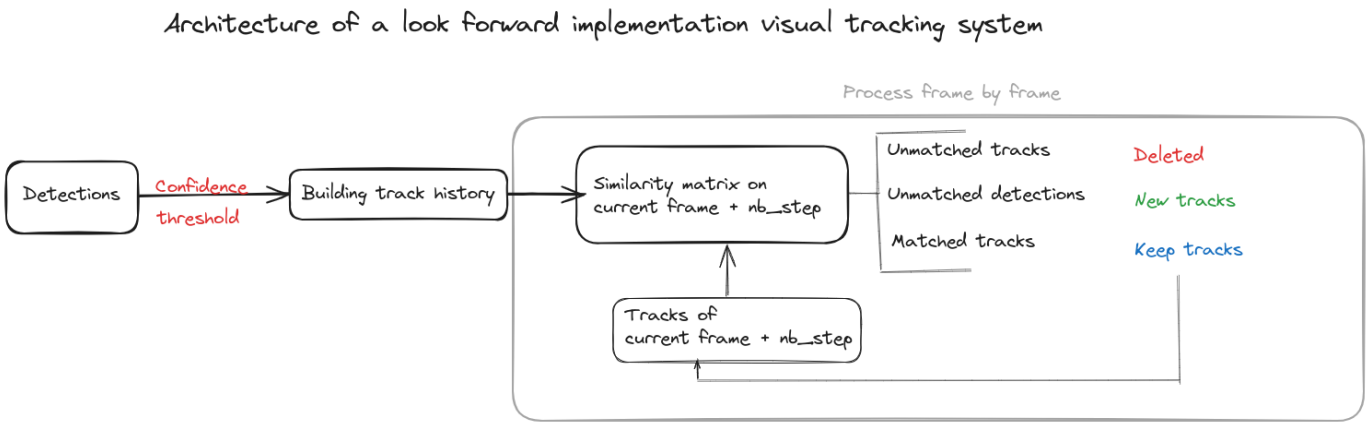

Figure 1: Final Architecture of the Tracking Algorithm

2.1 Building the Track History

At the core of our tracking algorithm lies the `track_history` construct, a crucial component that encapsulates the temporal evolution of tracked objects. The initialization of `track_history` is particularly vital and occurs as follows:

- In the first frame, we loop through a predetermined number of initial images (`nb_step`), extracting and processing each frame.
- For each frame, we identify detections that surpass a confidence threshold (`conf_threshold`), extracting their bounding boxes and associated confidence scores.
- We initialize new tracks for each detection in the first frame, storing them in `tracks_history`.
- For subsequent frames within the `nb_step` window, we employ a Kalman Filter (`kf`) to predict the state of each track.

- We implement a process to ensure the uniqueness of each track, thereby avoiding duplicate tracking.
- The similarity matrix is then created for the current detections against the existing tracks, facilitating the association of detections to tracks.

```python
if frame_number == 1:
    # Loop over the first nb_step images
    for i, frame_filename in enumerate(sorted(os.listdir(frames_path))[:nb_step], start=1):
        frame_path = os.path.join(frames_path, frame_filename)
        frame = cv2.imread(frame_path)

        # Get the detections for the current frame
        current_detections = det_df[det_df['frame'] == i]
        current_detections = current_detections[current_detections['conf'] >= conf_threshold]
        current_boxes = current_detections[['bb_left', 'bb_top', 'bb_width', 'bb_height']].values
        current_confidences = current_detections['conf'].values

        if i == frame_number:
            tracks_history[i] = []
            for det, conf in zip(current_boxes, current_confidences):
                new_track = initialize_new_track(det, conf, i, frame)
                tracks_history[i].append(new_track)
        else:
            for tracks_idx in range(frame_number, i):
                for track in tracks_history[tracks_idx]:
                    track['kf'].predict()

            unique_tracks = {}
            for j in range(i, frame_number, -1):
                for track in tracks_history.get(j, []):
                    if track['id'] not in unique_tracks:
                        unique_tracks[track['id']] = track

            tracks = list(unique_tracks.values())
            (parameter) tracks_history: Any
            tracks_history: dictionnary with as key the frame number and as value the list of tracks
                                                                                    frame)
                                                                                ons_to_tracks(similarity_matrix, sigma)
            tracks_history[i] = update_tracks(similarity_matrix,
                                              matches,
                                              unmatched_tracks,
                                              unmatched_detections,
                                              current_boxes,
                                              current_confidences,
                                              tracks,
                                              i,
                                              frame)
```

Figure 2: Building the first tracks history

2.2 Creating the Similarity Matrix

The similarity matrix is a pivotal element in associating current detections with existing tracks. It employs a blend of techniques:

- **Deep Feature Extraction**: We use a ResNet model (ResNet18), customized to output a specific feature size (`feature_size`). This model extracts deep features from detections, we are then able to make cosine_similarities between features.
- **IOU Matching**: The original Intersection over Union (IOU) metric provides a fundamental geometric comparison between detections and predicted track positions. It gives us a good indication between close bounding boxes but is useless for distinct ones.
- **Direction Similarity**: We compute a direction similarity score based on the predicted movement of a track and its alignment with the current detection's position. We then check if two detections have the same direction to add a bonus for the similarity.
- **Overall Similarity Computation**: The final similarity score is a weighted amalgamation of IOU scores, deep feature similarities, and directional alignment.

```python
def compute_similarity(iou_score, deep_features1, deep_features2, velocity_similarity):
    """
        Compute the similarity between two detections.
    """
    sim = 0.1 if velocity_similarity > 0.8 else -0.1 if velocity_similarity < -0.7 else 0
    if use_resnet:
        deep_similarity = torch.nn.functional.cosine_similarity(deep_features1, deep_features2).item()
        return max(0, min(1, (iou_score + deep_similarity * 0.5 + sim)))
    else:
        return max(0, min(1, iou_score + sim))

def create_similarity_matrix(current_detections, previous_tracks, frame):
    """
        Create the similarity matrix between the current detections and the previous tracks.
        Parameters:
            current_detections: list of current detections
            previous_tracks: list of previous tracks
    """
    similarity_matrix = np.zeros((len(current_detections), len(previous_tracks)))
    for i, current_det in enumerate(current_detections):
        # Extract features for the current detection
        deep_features1 = extract_deep_features(frame, current_det)

        for j, previous_track in enumerate(previous_tracks):
            predicted_box = get_predicted_box(previous_track['kf'], current_det[2], current_det[3])
            iou_score = track_iou(current_det, predicted_box)

            # Use the stored features from the track
            deep_features2 = previous_track['deep_features']

            # Calculate direction similarity
            track_velocity = previous_track['kf'].x[2:4]
            velocity_similarity = direction_similarity(predicted_box[:2], current_det[:2], track_velocity)

            similarity_score = compute_similarity(iou_score, deep_features1, deep_features2, velocity_similarity)
            similarity_matrix[i][j] = similarity_score
    return similarity_matrix
```

Figure 3: System building the similarity matrix

3 Analysis of Results

Our analysis focuses on contrasting the system's performance with and without the application of the ResNet model:

Configuration	FPS	Number of IDs
Without ResNet (TP4)	2453.73	240
With ResNet (TP5)	5.38	188
Looking Forward without ResNet	365.74	122
Looking Forward with ResNet	6.24	106

Table 1: Performance Analysis

This analysis uses the ADL-Rundle-6 detections dataset from the MOT challenge as its primary data source. The core of the model is built around ResNet18, which is employed to extract and compare deep features for object tracking. This report chronologically presents the development progress, starting from the initial implementation using a Kalman Filter (referred to as TP4) and culminating in the final implementation.

It should be noted that this analysis does not include evaluations from TrackEval. Despite efforts to integrate this tool, its use was ultimately hindered by prolonged and unresolved dependency conflicts. As such, the evaluation presented here is conducted independently of TrackEval metrics.

3.1 Discussing results

The integration of the ResNet model for deep feature extraction significantly impacts the system's performance. While the use of deep features enhances tracking accuracy and reduces ID switches, it substantially lowers the frames per second (FPS), indicating a trade-off between accuracy and computational efficiency.

In scenarios where high accuracy is paramount and computational resources are abundant, employing deep features from ResNet proves advantageous. Conversely, in applications where real-time processing is critical, a lighter approach without ResNet yields a much higher FPS, albeit at the cost of tracking precision.

4 Challenges and Hardships

4.1 Balancing Aspects of Similarity

One of the significant challenges faced in the development of our tracking algorithm was the intricate task of balancing various results of techniques, deep feature extraction, IOU matching, and directional similarity.
Each aspect plays more or less their role in enhancing the accuracy of the system, this poses a complex problem. Deciding the relative weight and impact of each similarity measure required meticulous experimentation and fine-tuning to achieve an optimal balance that accurately represents the dynamics of object tracking.

4.2 Forward-Looking Frame Processing

Another complex aspect was implementing a forward-looking mechanism in the frame processing. This approach necessitated computing detections for not only the current frame but also for frames a specific number of steps ahead (‘nb_step‘).
The challenge lay in predicting future states accurately while ensuring computational efficiency. Furthermore, retaining relevant old detections to maintain unique and continuous tracks added an additional layer of complexity. This process required careful management of data and resources, ensuring that the system remains both accurate in prediction and efficient in performance.

5 Conclusion

Setting aside the reduction in FPS, we successfully managed to reduce the number of identified tracks to a mere 106, approaching closer to the ground truth count of 23. This outcome signifies a substantial advancement in the accuracy and reliability of our tracking system, underscoring the effectiveness of integrating deep learning techniques, particularly the use of ResNet for feature extraction.

Throughout this project, I encountered and overcame numerous challenges, notably in the realm of computational efficiency and in resolving dependency conflicts, particularly with the TrackEval tool. These experiences have provided valuable insights into balancing accuracy with processing speed and underscore the importance of continual adaptation and problem-solving in the field of computer vision.

Looking forward, there is an exciting avenue for further optimization and refinement. Enhancements in computational efficiency, perhaps through using a YOLO model or preprocessing first the detections before tracking. This project lays a solid foundation for these future endeavors, leading the way for more advanced and efficient visual object tracking solutions.