datature.io

An Introduction to BYTETrack: Multi-Object Tracking by Associating Every Detection Box

Ben Le

21–27 minutes

Introduction

Computer vision as a field has grown tremendously both in academia and industry in the past few decades. With the advent of deep learning and improved computational capability of processors such as GPUs and TPUs, researchers have made significant progress in solving a growing range of tasks.

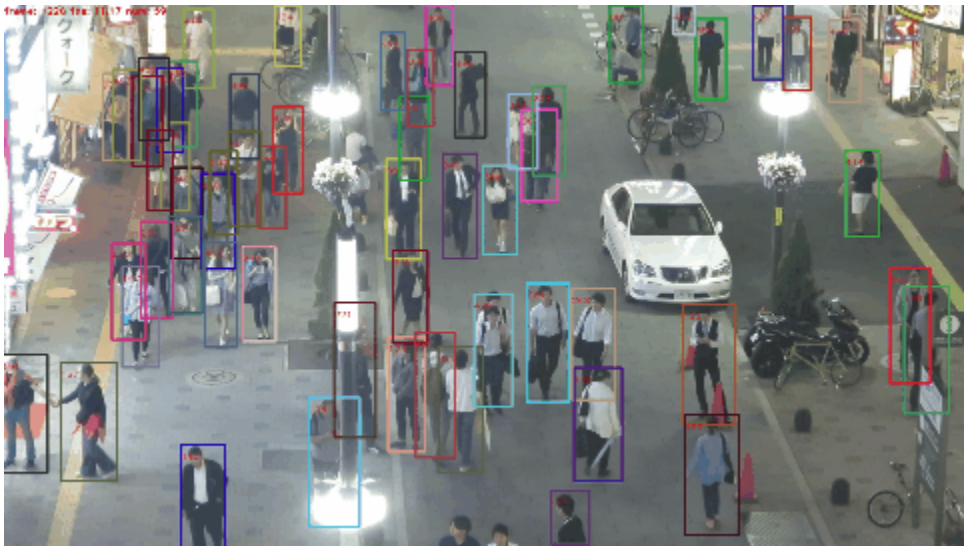


Figure 1: Example of Pedestrian Tracking

Traditional computer vision methods using rule-based and feature-based techniques for object detection have since been overshadowed by deep learning approaches based on the revolutionary Convolutional Neural Networks (CNNs). Deep neural networks have achieved state-of-the-art performance for tasks like classification, object detection, and image/instance segmentation. As machine learning computer vision's capability has rapidly progressed, researchers are now shifting their attention towards more complex computer vision tasks such as multiple object tracking.

What is Multiple Object Tracking?

Multiple object tracking (MOT) is a computer vision task that involves tracking the movements of multiple objects over time in a video sequence. The goal is to determine the identity, location, and trajectory of each object in the video, even in cases where objects are partially or fully occluded by other objects in the scene. MOT is an important problem in computer vision, as it has numerous practical applications. For example, in surveillance, MOT can be used to detect and track suspicious behaviour in a crowd or monitor the movement of vehicles in a parking lot. In robotics, it can be used to guide autonomous vehicles, identify obstacles, and plan safe routes through a dynamic environment. Other applications of MOT are in sports analytics, medical imaging, and many more.

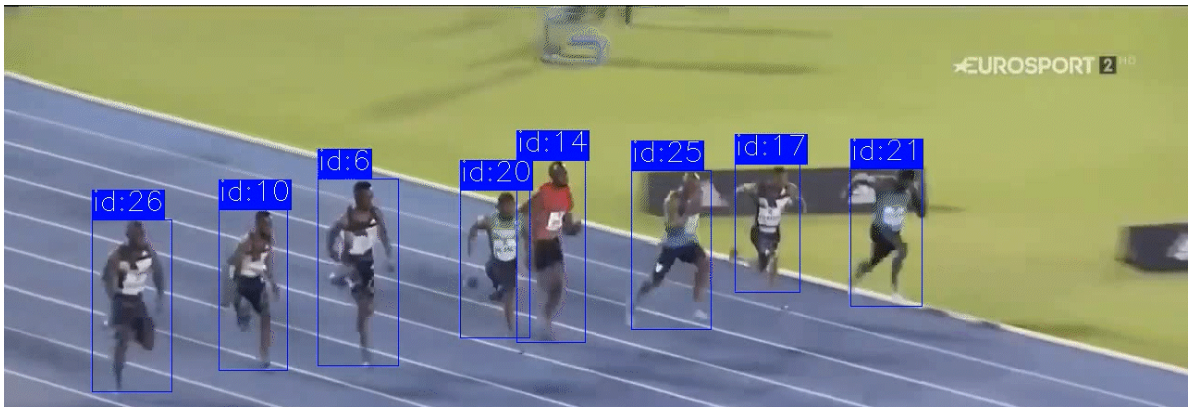


Figure 2: MOT applications in sports analytics

Generally, multiple object tracking happens in two stages: object detection and object association. Object detection is the process of identifying all potential objects of interest in the current frame using object detectors such as Faster-RCNN or YOLO. Object association is the process of linking objects detected in the current frame with its corresponding objects from previous frames, referred to as tracklets. Object or instance association is usually done by predicting the object's location at the current frame based on previous frames' tracklets using the Kalman Filter followed by one-to-one linear assignment typically using the Hungarian Algorithm to minimise the total differences between the matching results.

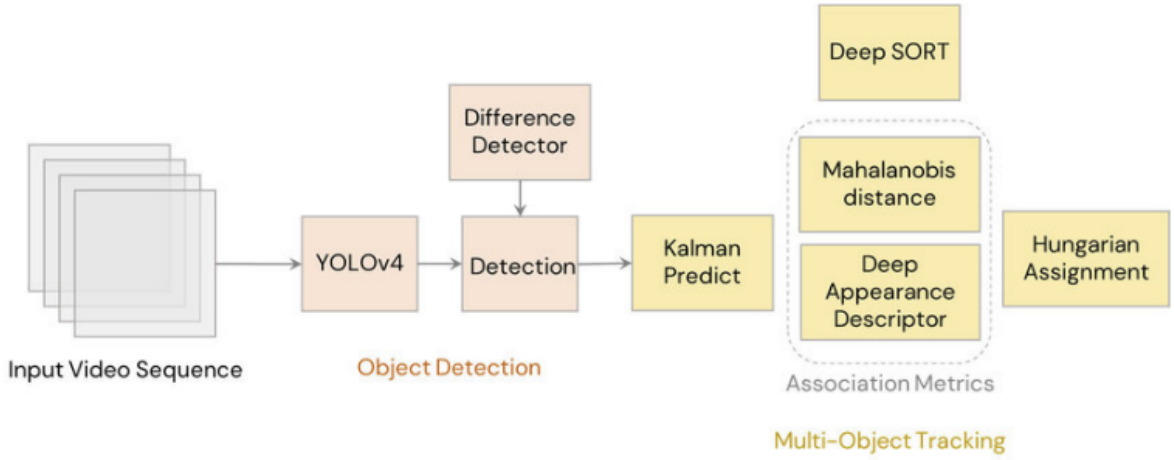


Figure 3: DeepSORT Algorithm

Challenges with MOT

Despite much progress over the years, MOT remains a challenging task.

Described below are some critical issues that have inhibited high quality performance, and have been the source of motivation for recent approaches.

Complications can originate from the actual visual data itself. For example, the motion and appearance of the same object can vary greatly throughout the video sequence. Objects can move at different speeds and directions, change in size or shape, and may be partially or fully occluded by other objects in the scene. These problems contribute to MOT tracking inaccuracies such as object ID switching or multiple tracklets being assigned to the same object.

At the architectural level, multi-class trackers may encounter class switching issues that exacerbate issues with object association between various frames if the detector fails to classify bounding boxes accurately because of classification issues such as similar classes which confuse the detector. Therefore, the tracking algorithm must be able to handle these variations and reliably associate each object in the current frame with its corresponding object in the previous frame.

For practical implementation, MOT architectures may be required to perform real-time predictions. Many MOT methods can struggle with video inference speed, especially for live predictions on videos with high frame rates, because the complexity of the models used requires more computation, and thus more compute time. In particular, multi-component or multi-stage video trackers with robust architectures will certainly encounter difficulties when performing inference on live video feeds.

MOT17-02 #Frame302 MOT17-02 #Frame310

Figure 4: ID Switching due to objects overlapping

Various existing methods have been used to tackle the MOT task in a broad range of complexity. From relatively simple algorithms using IoU for instance association to more complex algorithms such as SORT, DeepSORT, and Norfair, this large breadth of variation demonstrates the difficulties in balancing computational tradeoffs for inference speed and deeper extraction and utilization of visual features to produce more accurate predictions. Recent MOT approaches have used complex algorithms primarily based on trajectory prediction with the Kalman Filter followed by bounding box association with Hungarian Algorithm. Our latest blog explores one such example of a complex MOT tracker, Track Every Thing. In this article, we will explore an alternative approach, a simple yet powerful tracker - **BYTETrack**.

Figure 5: BYTETrack on Multiple Object Tracking of human surveillance

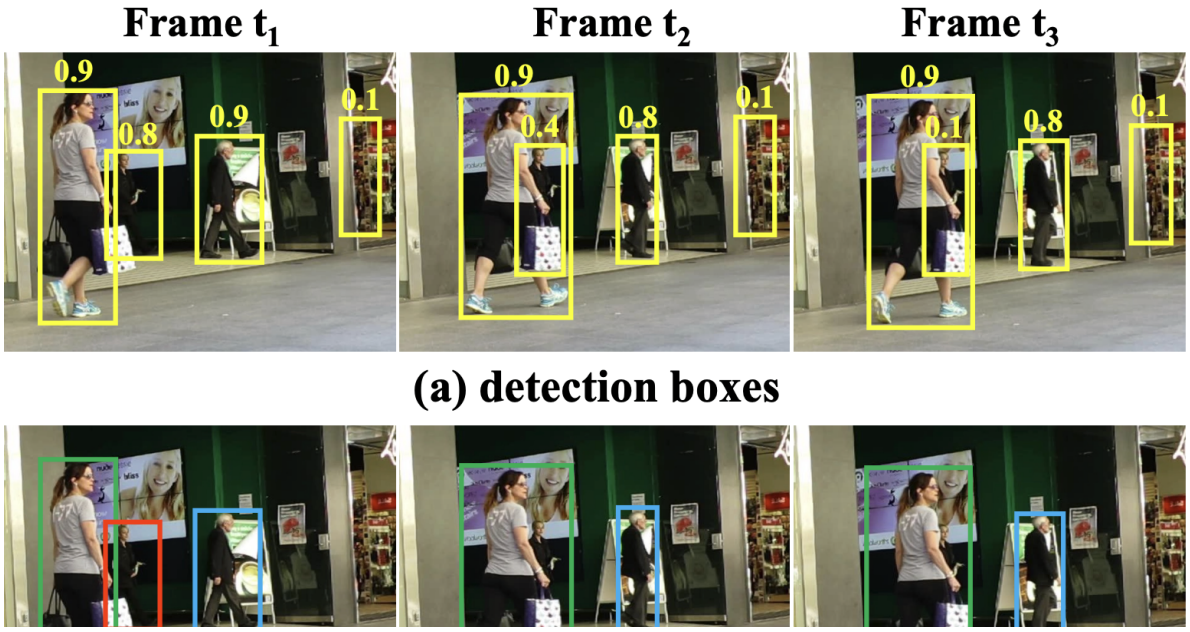
What is BYTETrack?

BYTETrack: Multi-Object Tracking by Associating Every Detection Box is a paper presented at ECCV2022 by Yifu Zhang et al. Thanks to its

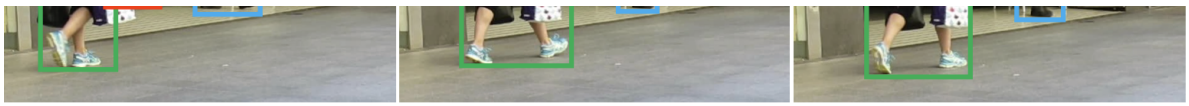
universal framework and relative simplicity, it has been adopted by many subsequent researchers for their MOT trackers (Bot-SORT, SMILEtrack).

The main idea of BYTETrack is simple - keep non-background low score boxes for a secondary association step between previous frame and next frame based on their similarities with tracklets. This helps to improve tracking consistency by keeping relevant bounding boxes which otherwise would have been discarded due to low confidence score (due to occlusion or appearance changes). The generic framework makes BYTETrack highly adaptable to any object detection (YOLO, RCNN) or instance association components (IoU, feature similarity).

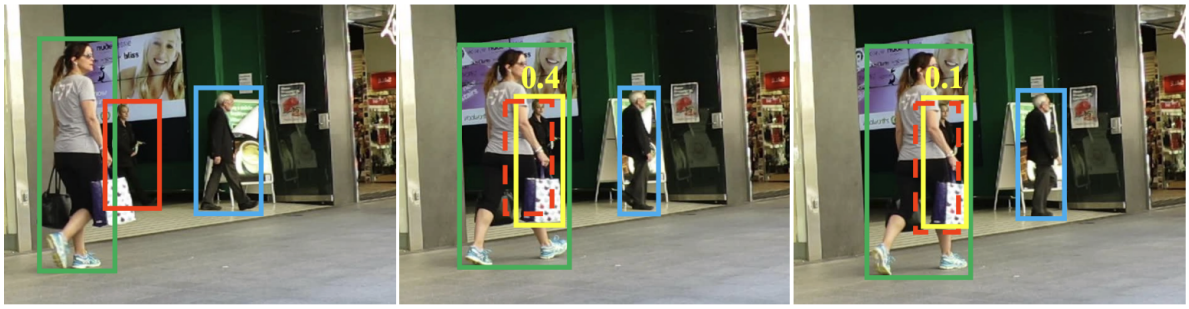
The primary innovation of BYTETrack is keeping non-background low confidence detection boxes which are typically discarded after the initial filtering of detections and use these low-score boxes for a secondary association step. Typically, occluded detection boxes have lower confidence scores than the threshold, but still contain some information about the objects which make their confidence score higher than purely background boxes. Hence, these low confidence boxes are still meaningful to keep track of during the association stage.



(a) detection boxes



(b) tracklets by associating high score detection boxes



(c) tracklets by associating every detection box

Figure 6: ByteTrack uses low-confidence bounding boxes in addition to high-confidence boxes

Methodology Breakdown

The overall idea of BYTETrack can be observed from Figure 7. After the detection stage, the detected bounding boxes are filtered into high confidence boxes, low confidence boxes and background boxes with fixed upper and lower thresholds. Background boxes are discarded after this process, but both low and high confidence detection boxes are kept for future association stages.

Similar to typical association steps from other algorithms, the high-score detection boxes of current frames are matched with predicted boxes from previous frames tracklets (using Kalman filter), which include both active tracklets and lost tracklets from recent frames. The matching is done using a simple IoU score or cosine similarity score between feature embeddings (using feature extractors such as DeepSORT, QDTrack, etc.) using the Hungarian algorithm or matching cascade on Nearest Neighbor Distance. Linear assignment between pairs of bounding boxes is only confirmed if the matching score is higher than a fixed match threshold. Unmatched high-score detection boxes in the actual

implementation are first matched with tracklets with updates from a single frame before being assigned to a new tracklet.

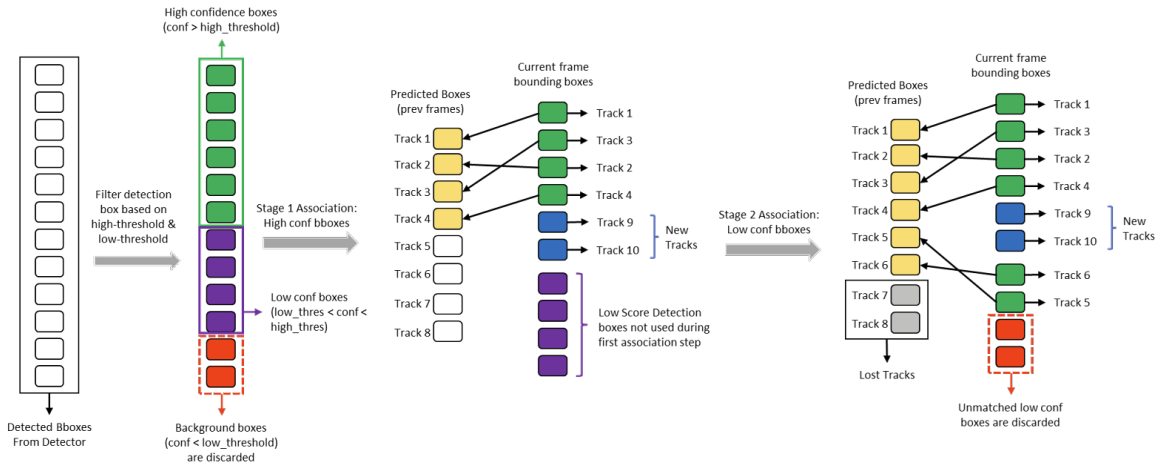


Figure 7: BYTETrack Algorithm

In the second association stage, low-score detection boxes are matched against the remaining unmatched predicted boxes from previous frames. The matching algorithm is identical to the first association stage, but the matching threshold is set lower than the first association stage because of the intuition that occluded boxes should be more poorly matched to boxes from previous frames. Unmatched predicted boxes are assigned as lost tracklets, while unmatched detection boxes are discarded. The lost tracklets are kept for a certain number of frame duration and added back to the active tracklets before Kalman filter prediction. This allows the trackers to recover some tracklets which have been lost due to objects completely disappearing for a short number of frames.

Implementation Details

In the paper, the author uses YOLOX as the base Detector. However, the Detector choice is easily customizable to FasterRCNN or other YOLO versions as BYTETrack simply takes in detection inputs with bounding boxes coordinates and class probability. Depending on the nature of datasets, users can choose different types of matching metrics (IoU or

any ReID). Based on the authors, both IoU and ReID are good options for first stage association of high-score detections. ReID performs better on low-frame rate videos or videos with significant frame-to-frame movements; while IoU is more robust for cases of severe occlusion in which ReID features are not reliable. However, second stage association should always use IoU as the matching criteria, as we can expect low-score detection boxes to contain occluded objects whose ReID features may not be good representations of the objects.

Tutorial Example

In this tutorial, I have taken video footage of the 2022 NBA finals between the Warriors and the Celtics. Our objective is to train a BYTETracker to track the movement of basketball players. I am interested in the movement of 3 classes: Warriors players, Celtics players and the referees. I will train a custom YOLOv8 Detector and use ByteTrack on top of the Detector outputs to keep track of objects.

Package and Dependencies Installation

First, we clone the BYTETrack repository from https://github.com/ifzhang/ByteTrack.git. There are a few steps required for installation of BYTETrack, which can be found below. It is recommended to install the dependencies in an isolated virtual environment. The dependencies in this section cover the necessary libraries for ByteTrack specifically.

```
!git clone https://github.com/ifzhang/ByteTrack.git
%cd ByteTrack
!pip3 install -r requirements.txt
!python3 setup.py develop
```

```
!pip3 install cython
!pip3 install 'git+https://github.com/cocodataset/
cocoapi.git#subdirectory=PythonAPI'
!pip3 install cython_bbox
```

Data Preparation

I then prepare a custom dataset by downloading a short footage from the game and sample frames from it. My dataset contains approximately 90 side-view frames. For data annotation, I used Datature Nexus to label the objects and export the annotations for training our YOLOv8 model. With Datature's Nexus, it was a simple task to upload the frame images, make bounding box annotations for multiple classes, and export the annotations. Nexus makes it even simpler to annotate videos with its support for video annotation. One can simply upload their .mp4 video file onto Nexus and scrub between frames to easily annotate the video. For more details, please refer to this blog post on how to work with the newly released YOLOv8 model.

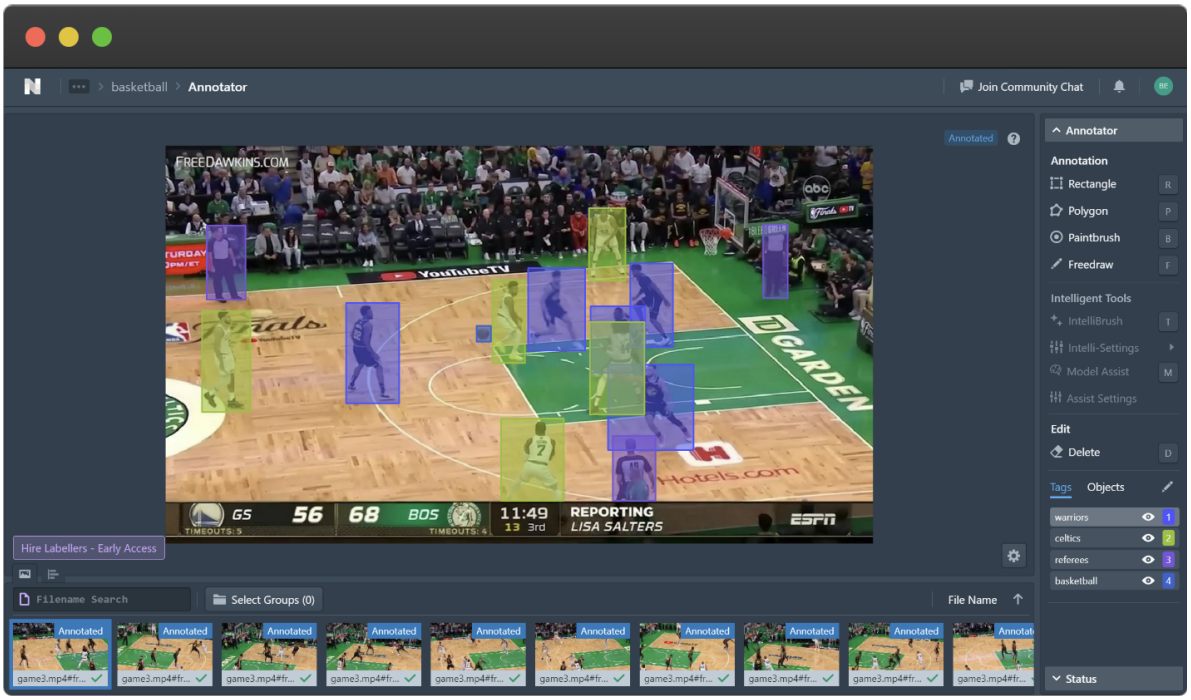


Figure 8: Annotation of Custom Dataset on Nexus Platform

Training the Object Detection Model (YOLOv8)

To train the model, we should ensure that our training image data and annotation files are ready, by ensuring that our frames are split into separate images, and having the annotation files associated correctly with the images. Additionally, we should ensure that our locally stored data is organized accordingly for our training. As an example, I've arranged the images and annotation files into the following format for YOLOv8 training.

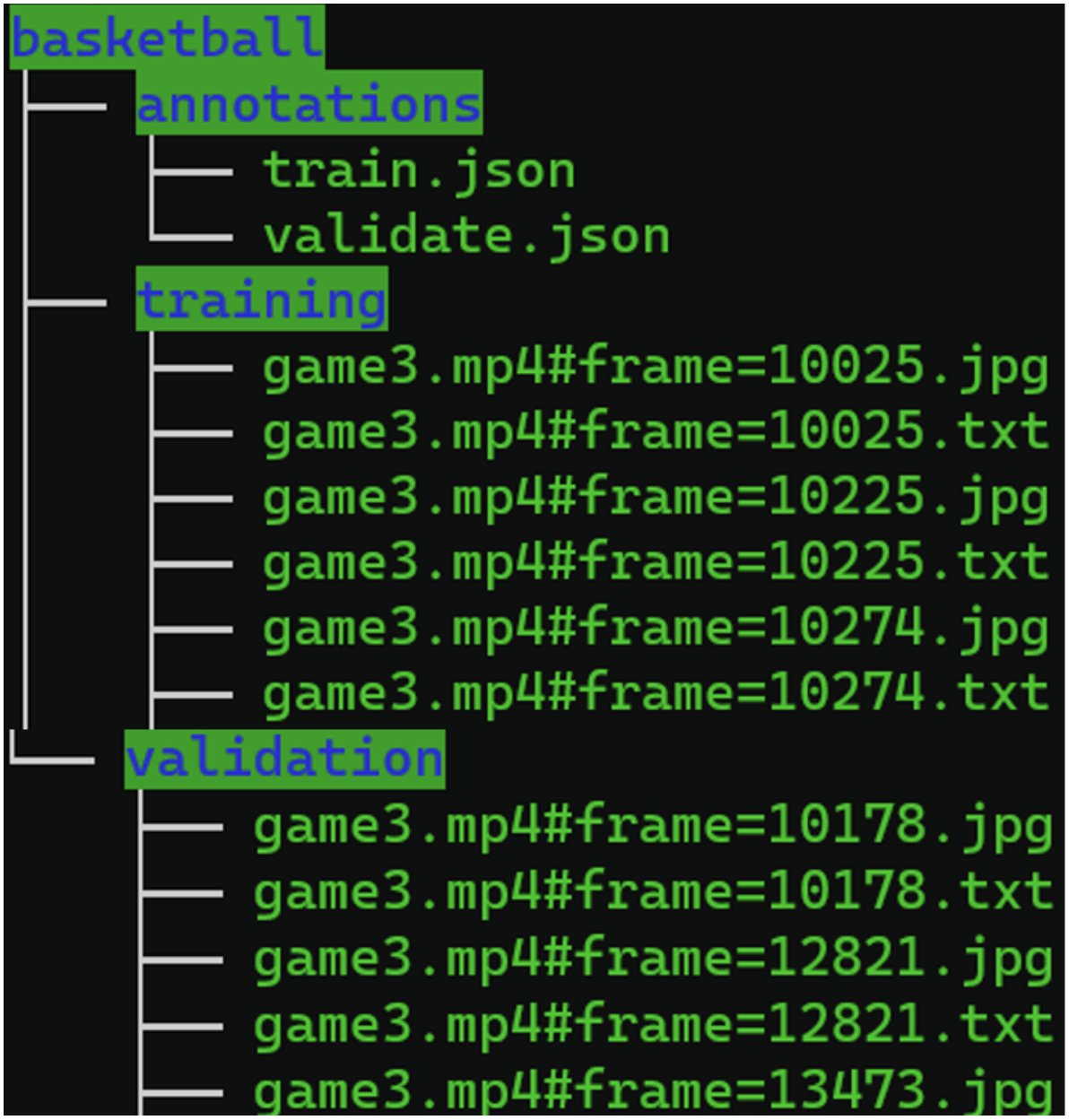


Figure 9: Image and annotation directory for YOLOv8 Custom Training

YOLOv8 model training can be performed using CLI command or Ultralytics Python SDK. In this tutorial, I used Ultralytics Python SDK to load the model checkpoint and train the model. The code below outlines the steps needed to get through training the whole process.

We need to install the Ultralytics Python SDK with the Python package installer, PIP.

```
### Install YOLOv8 ###
!pip install ultralytics
```

Secondly, we need to prepare our dataset as described earlier.

We can use the following commands to download the dataset to your local workspace.

```
### Load and Train YOLOv8 Model ###

# Load a model
model = YOLO("yolov8m.pt")  # load a pretrained model
(recommended for training)

# Train the model
results = model.train(data=f"{HOME}/datasets/
basketball.yaml", imgsz=TEST_SIZE, batch=BATCH_SIZE,
```

```
epochs=EPOCHS, plots=True)
```

Thirdly, we can use the Ultralytics Python SDK API to easily run the training for our YOLOv8 model.

Below, the graphs outline the training and validation loss over time after 50 epochs on the YOLOv8m model. The hyperparameters used were a batch size of 4, and a test size of 640. The metrics we tracked as good indicators of accuracy for the model were mAP50 on validation images, which was approximately 0.9 while mAP50-95 was approximately 0.65.

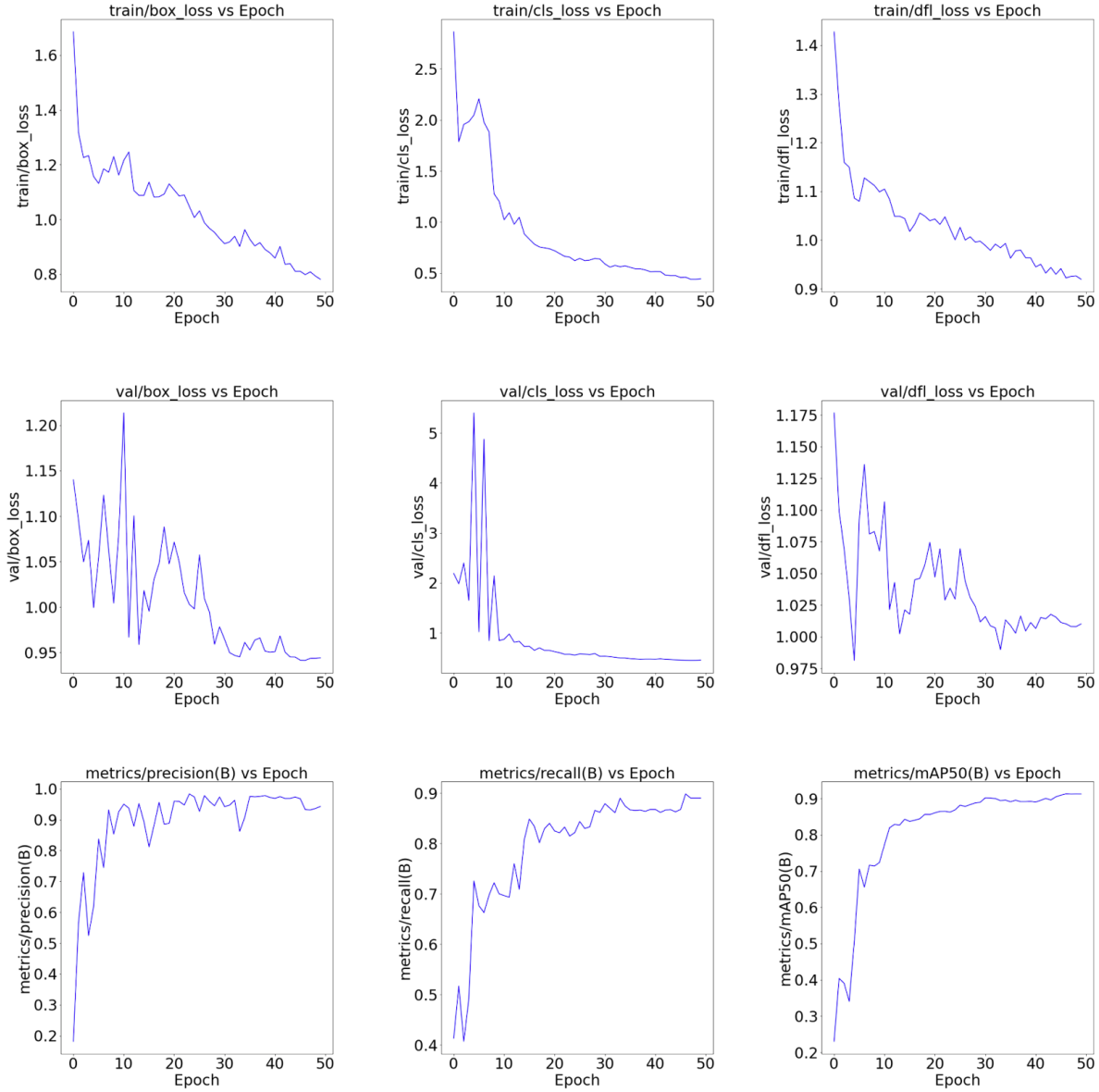


Figure 10: Training performance of YOLOv8m model on 50 epochs

We can further improve the object detector's performance by using a larger YOLO architecture, training more epochs, and adding more training images by collecting more data or data augmentation. For demonstration purposes, this provides sufficient accuracy to proceed to the next phase to perform the object tracking. Below is the result of performing object detection on each frame in the video. However, as you might be able to observe, there is no preserved state between frames that can recognize that the objects are the same, and thus cannot track object velocity and trajectory.

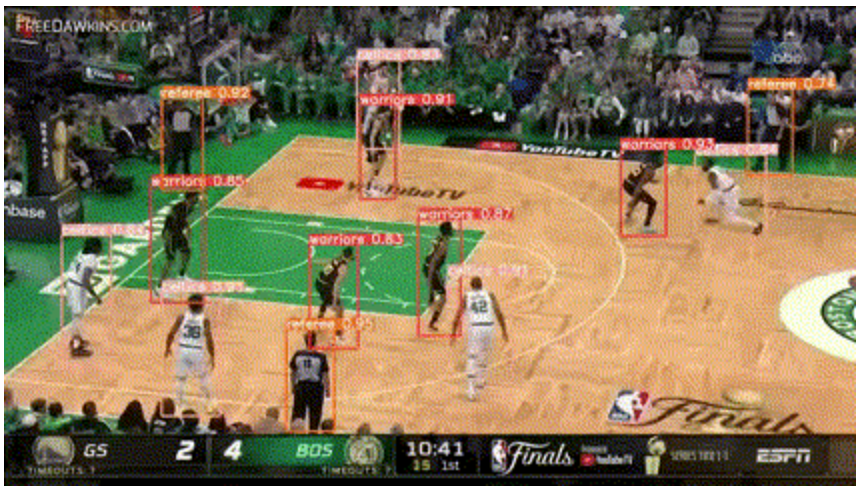


Figure 11: Video Inference on YOLOv8 Object Detection model

Tracking and Inferencing

With a trained YOLOv8 Detection model, I am ready to utilize the ByteTrack implementation. In the original implementation of the author, a tracker instance is initiated and for each image in the video sequence, the detections are used to update the tracklets of the tracker instance. The detection can be inputted in two types of format **<x1, y1, x2, y2, score>** or **<x1, y1, x2, y2, obj_score, class_score, class_id>**. The output **online_targets** is a list of active tracklets containing attributes **track_id, current frame bounding box and confidence score**. The

original implementation only tracks humans, and hence is class agnostic.

The following code can be used to track the objects using our trained detector:

```
tracker = BYTETracker(args)
for image in images:
   dets = detector(image)
   online_targets = tracker.update(dets, info_imgs,
img_size)
```

In our case, where we seek to perform multi-class, multi-object tracking, we can make some changes to our custom implementation to track the multiple classes in our dataset: Warriors players, Celtics players and referees. To get around the limitations of class agnostic tracking, I created multiple BYTETracker instances to keep track of individual classes and only updated a tracker using detections from its designated class.

```
trackers = [BYTETracker(ByteTrackArgument),
BYTETracker(ByteTrackArgument),
BYTETracker(ByteTrackArgument)]
frame_id = 0
results = []
history = deque()

while True:
```

```
    if frame_id % 20 == 0:
        print(f'Processing frame {frame_id}.')
    ret_val, online_im = cap.read()
    if ret_val:
        outputs = model.predict(source=online_im,
conf=MIN_THRESHOLD)
        img_height, img_width =
outputs[0].boxes.orig_shape
        outputs = outputs[0].boxes.boxes
        all_tlwhs = []
        all_ids = []
        all_classes = []
        for i, tracker in enumerate(trackers):
            class_outputs = outputs[outputs[:, 5] == i]
[:,:5]
            if class_outputs is not None:
                online_targets =
tracker.update(class_outputs.cpu(), [img_height,
img_width], [img_height.item(), img_width.item()])
                online_tlwhs = []
                online_ids = []
                online_scores = []
                online_classes = [i] * len(online_targets)
                for t in online_targets:
                    tlwh = t.tlwh
                    tid = t.track_id
                    vertical = tlwh[2] / tlwh[3] >
ByteTrackArgument.aspect_ratio_thresh
                    if tlwh[2] * tlwh[3] >
ByteTrackArgument.min_box_area and not vertical:
```

```
                    online_tlwhs.append(tlwh)
                    online_ids.append(tid)
                    online_scores.append(t.score)
                    box = (tlwh[0], tlwh[1], tlwh[2],
tlwh[3])

            all_tlwhs += online_tlwhs
            all_ids += online_ids
            all_classes += online_classes
```

Optimization and Customization

The above demo is designed to display the potential of the ByteTrack library as an object tracker. As such, I've outlined a few ways in which you can improve your own implementation to achieve your desired results. Below are some hyperparameters which we can tune to improve the tracker performance. Do note that for MIN_THRESHOLD, I set the threshold at **0.001** to retrieve almost all detections. The author has hardcoded a background threshold at **0.1** to further filter bounding boxes which are considered "background" boxes. If we want to be more selective with the detections, MIN_THRESHOLD can be increased to values higher than 0.1; however, this may potentially filter out meaningful detections of occluded objects. Thus, one should qualitatively examine whether the threshold used provides the appropriate quality.

```
class ByteTrackArgument:
    track_thresh = 0.5 # High_threshold
    track_buffer = 50 # Number of frame lost tracklets are
kept
```

```
    match_thresh = 0.8 # Matching threshold for first
stage linear assignment
    aspect_ratio_thresh = 10.0 # Minimum bounding box
aspect ratio
    min_box_area = 1.0 # Minimum bounding box area
    mot20 = False # If used, bounding boxes are not
clipped.

MIN_THRESHOLD = 0.001
```

ByteTracker initiates a new tracklet only if a detection is not matched with any previous tracklet and the bounding box score is higher than a threshold.

Optimising this threshold as it can reduce the number of ID switches significantly

```
""" Step 4: Init new stracks"""
for inew in u_detection:
    track = detections[inew]
    if track.score < 0.7:
        continue
    else:
        track.activate(self.kalman_filter, self.frame_id)
# Initiate a new track
```

Demo

After running the tracking code and fine-tuning the discussed hyperparameters, you can achieve results as seen below. Here is the rendered video demo of our implementation. The tracker manages to quite consistently follow players movement even when there are some occasional occlusions. However, there are few instances of ID switching in crowded areas. This is understandable as there are multiple strongly matched candidates in close proximity making it difficult for the tracker to differentiate based on IoU. We can consider adding Re-Identification (ReID) as a further improvement that will add more complexity to our solution architecture, by matching features between current detection boxes and previous tracklet boxes.

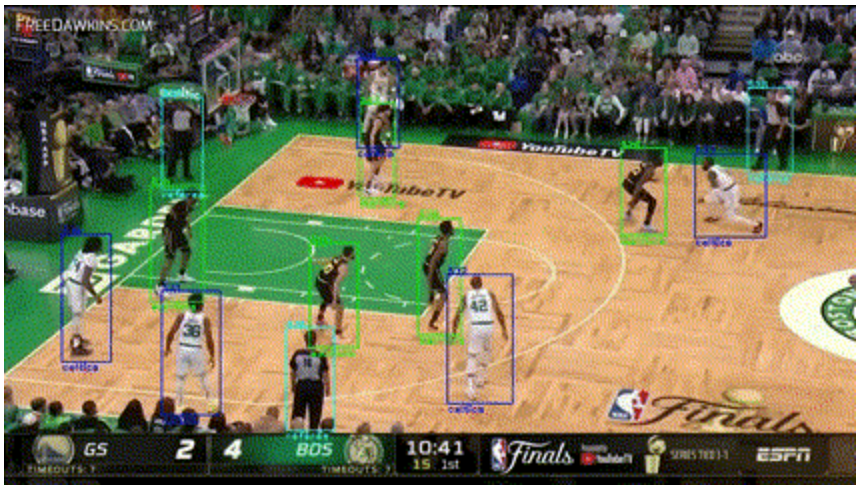


Figure 12: ByteTrack demo on basketball players tracking (multiclass)

Conclusion

From quantitative review and anecdotal experience working with the tracker, BYTETrack certainly demonstrates the effectiveness of its simple architectural design. The IoU-based matching, while simplistic, has demonstrated some effectiveness in reducing ID switching which can be caused by occlusion. Its lightweight and simple framework allows for real-time tracking of multiple objects, which makes it suitable for a broad range of applications such as autonomous vehicles and robotics.

It can also be easily adapted to improve existing multiple object tracking frameworks. However, because of its usage of relatively simple metrics, BYTETrack demonstrates limited accuracy in crowded scenes with many objects moving amongst each other in close proximity. This is likely due to the reliance on IoU matching and trajectory prediction, which means that it is unable to easily distinguish between objects when their tracklets heavily overlap which can occur frequently. Additionally, the strategy of maintaining the usage of boxes with low confidence scores increases noise that can induce the tracking of irrelevant objects. Finally, BYTETrack does not natively support multi-class tracking as well, which is often a point of interest in many real-world tracking use cases, and thus requires computationally expensive workarounds.

In conclusion, BYTETrack is a relatively simple but highly effective Multiple Object Tracker utilising low confidence boxes to account for object occlusion and appearance changes. It is especially useful as it can be adopted to many existing SOTA MOT algorithms to further boost their performance, whether these trackers are motion-based (IoU matching) or feature-based (ReID feature matching). ByteTrack's ability to track objects in real-time with high accuracy and efficiency makes it suitable for various applications such as autonomous driving, video surveillance, and augmented reality. Although ByteTrack is still in its early stages of development, it holds great potential and it will be exciting to see how ByteTrack evolves and impacts different industries.

Want to Get Started?

Our readers can use the sample tutorial notebook below to start building your own Multiple Object Tracking project. If you have questions, feel free to join our Community Slack to post your questions or contact us on how to apply SOTA tracking models like ByteTrack to your projects.

For more detailed information about the model functionality, customization options, or answers to any common questions you might have, read more on our Developer Portal. If you would like to view the entire notebook, you can access it here.

Developer Roadmap

With machine learning applications taking the world by storm, Datature is also pushing to help more users be able to begin their journey with computer vision or extend their current applications further. As such, Datature is pushing efforts to enable compatibility with external libraries, tools, and other resources such as a tracker like BYTETrack. With tools such as hosted dataloaders, users will be able to integrate their assets and models from Nexus to other applications smoothly and easily without having to transition assets off and on the platform.

References

- https://learnopencv.com/understanding-multiple-object-tracking-using-deepsort
- https://www.marketsandmarkets.com/Market-Reports/computer-vision-market-186494767.html
- https://www.datature.io/blog/implementing-object-tracking-for-computer-vision
- https://www.datature.io/blog/introduction-to-multiple-object-tracking-and-recent-developments
- https://arxiv.org/abs/2110.06864
- https://github.com/ifzhang/ByteTrack
- https://www.arxiv-vanity.com/papers/2012.15460/

- https://github.com/mikel-brostrom/yolov8_tracking