Deep Learning
Final Project

Iman Alizadeh Fakouri

Student Number: 401102134

Professor: Dr. Emad Fatemizadeh

Date: February 5, 2025

Contents

Introduction

This project focuses on object tracking in football videos using the SportsMOT dataset. Our approach consists of two main stages: object detection and object tracking. For object detection, we employ the YOLO model. Once objects are detected, we implement two tracking methods: the Channel and Spatial Reliability Tracker (CSRT) for single-object tracking and ByteTrack for multi-object tracking. CSRT is a discriminative correlation filter-based tracker that provides robust single-object tracking, while ByteTrack is an advanced multiple-object tracking algorithm that effectively handles occlusions and identity switches.

By integrating these techniques, we aim to develop a reliable sports video object tracking framework that can enhance automated analysis in football games.

Data Processing and Augmentation

Data augmentation is a crucial step in training deep learning models, particularly for object detection and tracking. Augmenting data helps improve the generalization of the model by introducing variations in the training data, making it more robust to real-world scenarios. Common augmentation techniques include flipping, adding noise, and adjusting brightness, all of which help the model learn to recognize objects under different conditions.

For training YOLO on the SportsMOT dataset, we need to preprocess the dataset to ensure compatibility. The SportsMOT dataset consists of multiple sports, but since our project focuses on football, we must extract the relevant football-related videos. Additionally, YOLO requires images and labels in a specific format, so we must convert the annotations accordingly.

```
def read_scripts_from_file(file_path):
    with open(file_path, 'r') as file:
        scripts = set(line.strip() for line in file.readlines())
    return scripts

football_file = '/content/SportsMOT/sportsmot_publish/splits_txt/football.txt'
train_file = '/content/SportsMOT/sportsmot_publish/splits_txt/train.txt'
val_file = '/content/SportsMOT/sportsmot_publish/splits_txt/val.txt'
test_file = '/content/SportsMOT/sportsmot_publish/splits_txt/test.txt'

football_scripts = read_scripts_from_file(football_file)
train_scripts = read_scripts_from_file(train_file)
val_scripts = read_scripts_from_file(val_file)
test_scripts = read_scripts_from_file(test_file)

train_football = football_scripts & train_scripts
val_football = football_scripts & val_scripts
test_football = football_scripts & test_scripts

print("Football scripts in train:", train_football)
print("Football scripts in val:", val_football)
print("Football scripts in test:", test_football)
```

The script above extracts football-related videos from the SportsMOT dataset. It cross-references the list of football videos with the dataset's train, validation, and test splits to ensure only relevant videos are selected for training YOLO.

```
DATA_DIRS = {
    'train': "/content/SportsMOT/sportsmot_publish/dataset/train",
    'val': "/content/SportsMOT/sportsmot_publish/dataset/val"
}

PREPROCESSED_DIRS = {
    'train': "/content/SportsMOT/preprocessed/train",
    'val': "/content/SportsMOT/preprocessed/val"
}

for split in PREPROCESSED_DIRS.values():
    os.makedirs(os.path.join(split, 'images'), exist_ok=True)
    os.makedirs(os.path.join(split, 'labels'), exist_ok=True)
```

This snippet creates directories for preprocessed images and labels, ensuring the dataset is organized correctly for YOLO training.

```
def gpu_augmentations(img, boxes):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    tensor_img = torch.tensor(img).permute(2, 0, 1).float().to(device) / 255.0

    aug_images = [tensor_img]
    aug_boxes = [boxes]

    flipped_img = TF.hflip(tensor_img)
    flipped_boxes = [(1 - xc, yc, w, h) for (xc, yc, w, h) in boxes]
    aug_images.append(flipped_img)
    aug_boxes.append(flipped_boxes)

    noise = torch.randn_like(tensor_img) * 0.1
    noisy_img = torch.clamp(tensor_img + noise, 0, 1)
    aug_images.append(noisy_img)
    aug_boxes.append(boxes)

    return aug_images, aug_boxes
```

This function performs GPU-based augmentations, including horizontal flipping and adding Gaussian noise, which enhances the dataset diversity for better model generalization.

```
def process_video(split, video_dir):
    if video_dir not in FOOTBALL_VIDEOS[split]:
        return

    video_path = os.path.join(DATA_DIRS[split], video_dir)
    gt_file = os.path.join(video_path, 'gt/gt.txt')
    img_dir = os.path.join(video_path, 'img1')

    image_output_dir = os.path.join(PREPROCESSED_DIRS[split], 'images')
    label_output_dir = os.path.join(PREPROCESSED_DIRS[split], 'labels')
    os.makedirs(image_output_dir, exist_ok=True)
    os.makedirs(label_output_dir, exist_ok=True)

    det_df = pd.read_csv(gt_file, header=None, names=['frame', 'id', 'x', 'y', 'w', 'h', 'f1', '
```

```
    for frame_num, frame_data in det_df.groupby('frame'):
        img_file = os.path.join(img_dir, f"{frame_num:06}.jpg")
        if not os.path.exists(img_file):
            continue

        img = cv2.imread(img_file)
        img_h, img_w = img.shape[:2]
        valid_boxes = []

        for _, row in frame_data.iterrows():
            if row['w'] > 0 and row['h'] > 0:
                xc = (row['x'] + row['w'] / 2) / img_w
                yc = (row['y'] + row['h'] / 2) / img_h
                w = row['w'] / img_w
                h = row['h'] / img_h
                valid_boxes.append((xc, yc, w, h))

        with torch.no_grad():
            aug_images, aug_boxes = gpu_augmentations(img, valid_boxes)
```

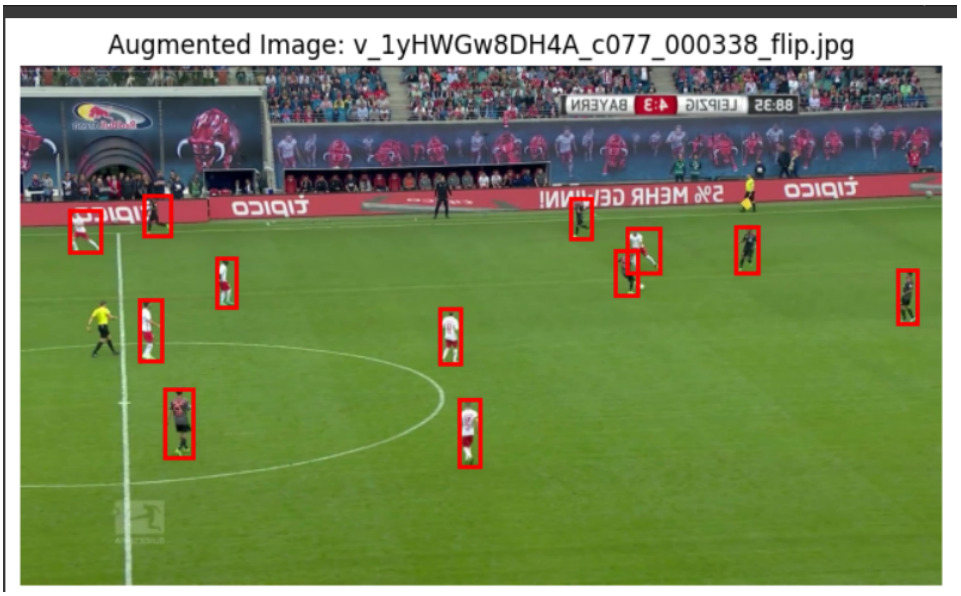
This function processes each video, reads the ground truth annotations, and applies GPU-based augmentations before saving the images and labels in a format suitable for YOLO training.

```
def load_image_and_labels(image_path, label_path):
    image = cv2.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    labels = []
    with open(label_path, 'r') as f:
        for line in f:
            parts = line.strip().split()
            xc, yc, w, h = map(float, parts[1:])
            labels.append((xc, yc, w, h))
    return image, labels
```

This function loads an image and its corresponding labels, converting the image from BGR to RGB for visualization.

```
def plot_image_with_labels(image, labels, title=""):
    plt.figure(figsize=(8, 8))
    plt.imshow(image)
    for (xc, yc, w, h) in labels:
        img_h, img_w, _ = image.shape
        x1 = int((xc - w / 2) * img_w)
        y1 = int((yc - h / 2) * img_h)
        x2 = int((xc + w / 2) * img_w)
        y2 = int((yc + h / 2) * img_h)
        plt.gca().add_patch(plt.Rectangle((x1, y1), x2 - x1, y2 - y1, linewidth=2, edgecolor='r'
    plt.title(title)
    plt.axis('off')
    plt.show()
```

This function visualizes an image with bounding box annotations, helping verify the preprocessing pipeline's correctness.



Object Detection

3.1 Why YOLO Over Faster R-CNN?

YOLO is one of the most advanced object detection algorithms, offering a balance between accuracy and speed. Unlike traditional object detection methods like Faster R-CNN, which use a two-stage pipeline (region proposal followed by classification), YOLO applies a single-stage detection approach. This makes it significantly faster and suitable for real-time applications such as sports analytics.

The key innovations in YOLO include:

- **Grid-based Detection:** YOLO divides an image into an $S \times S$ grid. Each grid cell predicts a fixed number of bounding boxes and class probabilities, making the process highly parallelizable.
- **Anchor Boxes:** The use of predefined anchor boxes allows YOLO to detect objects of different shapes and sizes effectively.
- **Regression-based Detection:** YOLO treats object detection as a regression problem, predicting bounding box coordinates, confidence scores, and class probabilities in a single pass through the network.
- **Non-Maximum Suppression (NMS):** To reduce duplicate detections, YOLO applies NMS, keeping only the highest-confidence predictions for each object.
- **Multi-scale Detection:** YOLOv3 and later versions employ a feature pyramid network (FPN) that enables detection at multiple scales, improving performance on small and large objects alike.

In object detection, we chose YOLO over Faster R-CNN due to several key advantages:

- **Speed:** YOLO is significantly faster since it processes an image in a single forward pass, whereas Faster R-CNN uses a two-stage approach (region proposal and classification).

- **Real-time Capability:** YOLO can run in real-time, making it ideal for sports analytics.
- **Unified Architecture:** Unlike Faster R-CNN, YOLO treats object detection as a regression problem, making it more efficient.
- **Improved Accuracy:** YOLO's anchor-based detection and better spatial generalization improve detection performance in complex scenes.

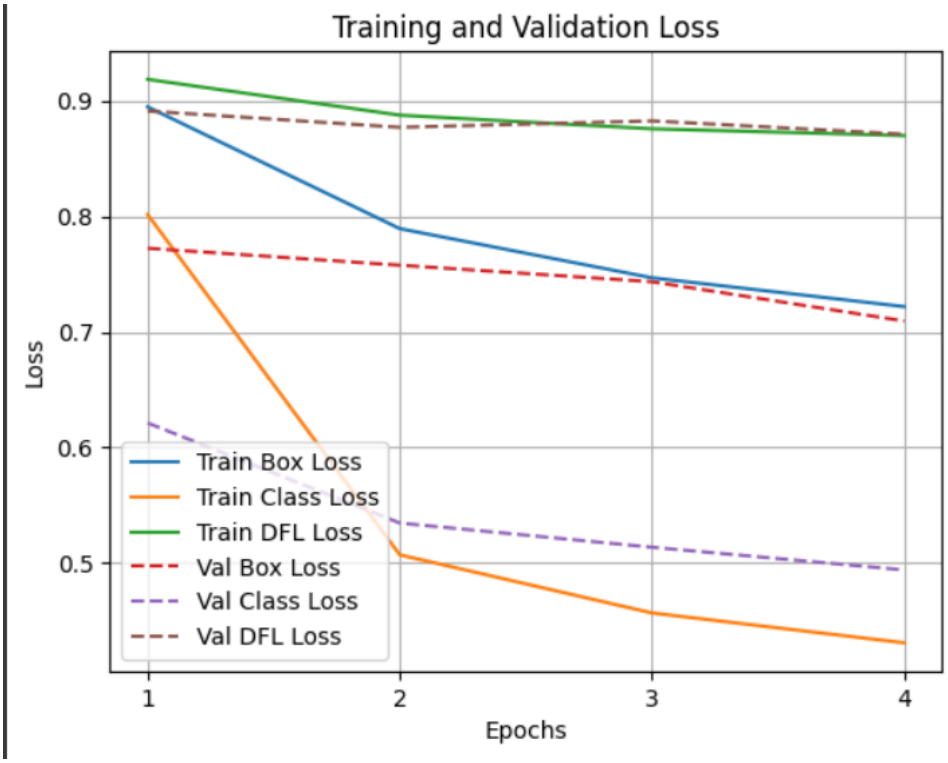
3.2 Fine Tuning the Model

To ensure our data pipeline works properly and that our model fine-tunes correctly, we first trained on a small dataset.

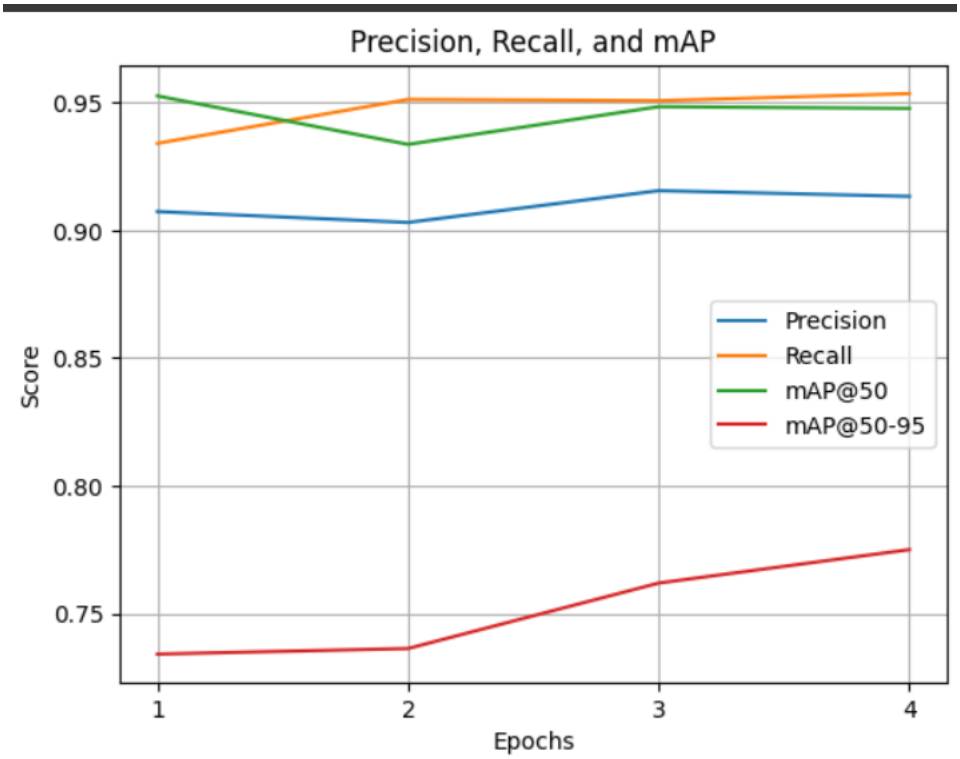
```
DATASET_YAML = f"""
    path: /content/SportsMOT/preprocessed
    train: {"train_2000/images"}
    val: {"val_200/images"}
    nc: 1  # Number of classes
    names: ['athlete']  # Class name
    """

with open("sportsmot.yaml", "w") as f:
    f.write(DATASET_YAML)

print("starting training...")
model = YOLO("yolov8n.pt")

results = model.train(
    data="sportsmot.yaml",
    epochs=4,
    imgsz=(1280, 720),
    batch=8,
    device="cuda" if torch.cuda.is_available() else "cpu",
    optimizer="AdamW",
    lr0=1e-3,
    name="sportsmot_v1",
    exist_ok=True
)

model.export(format="onnx")
print("model saved")

def create_small_dataset():
    os.makedirs("/content/SportsMOT/preprocessed/train_2000/images", exist_ok=True)
    os.makedirs("/content/SportsMOT/preprocessed/train_2000/labels", exist_ok=True)
    os.makedirs("/content/SportsMOT/preprocessed/val_200/images", exist_ok=True)
    os.makedirs("/content/SportsMOT/preprocessed/val_200/labels", exist_ok=True)

    all_train_images = glob(os.path.join("/content/SportsMOT/preprocessed/train/images", "*.jpg"
    all_val_images = glob(os.path.join("/content/SportsMOT/preprocessed/val/images", "*.jpg"))

    small_train_sample = random.sample(all_train_images, 2000)
    small_val_sample = random.sample(all_val_images, 200)

```

```
    for img_path in small_train_sample:
        shutil.copy(img_path, "/content/SportsMOT/preprocessed/train_2000/images")
        label_path = img_path.replace("images", "labels").replace(".jpg", ".txt")
        if os.path.exists(label_path):
            shutil.copy(label_path, "/content/SportsMOT/preprocessed/train_2000/labels")

    for img_path in small_val_sample:
        shutil.copy(img_path, "/content/SportsMOT/preprocessed/val_200/images")
        label_path = img_path.replace("images", "labels").replace(".jpg", ".txt")
        if os.path.exists(label_path):
            shutil.copy(label_path, "/content/SportsMOT/preprocessed/val_200/labels")

create_small_dataset()
```

We analyzed training loss curves to ensure convergence:




```
df = pd.read_csv("runs/detect/sportsmot_v1/results.csv")

plt.plot(df['epoch'], df['metrics/precision(B)'], label='Precision')
plt.plot(df['epoch'], df['metrics/recall(B)'], label='Recall')
plt.plot(df['epoch'], df['metrics/mAP50(B)'], label='mAP@50')
plt.plot(df['epoch'], df['metrics/mAP50-95(B)'], label='mAP@50-95')

plt.xlabel('Epochs')
plt.ylabel('Score')
plt.title('Precision, Recall, and mAP')
plt.xticks(df['epoch'].astype(int))
plt.legend()
plt.grid(True)
plt.show()
```



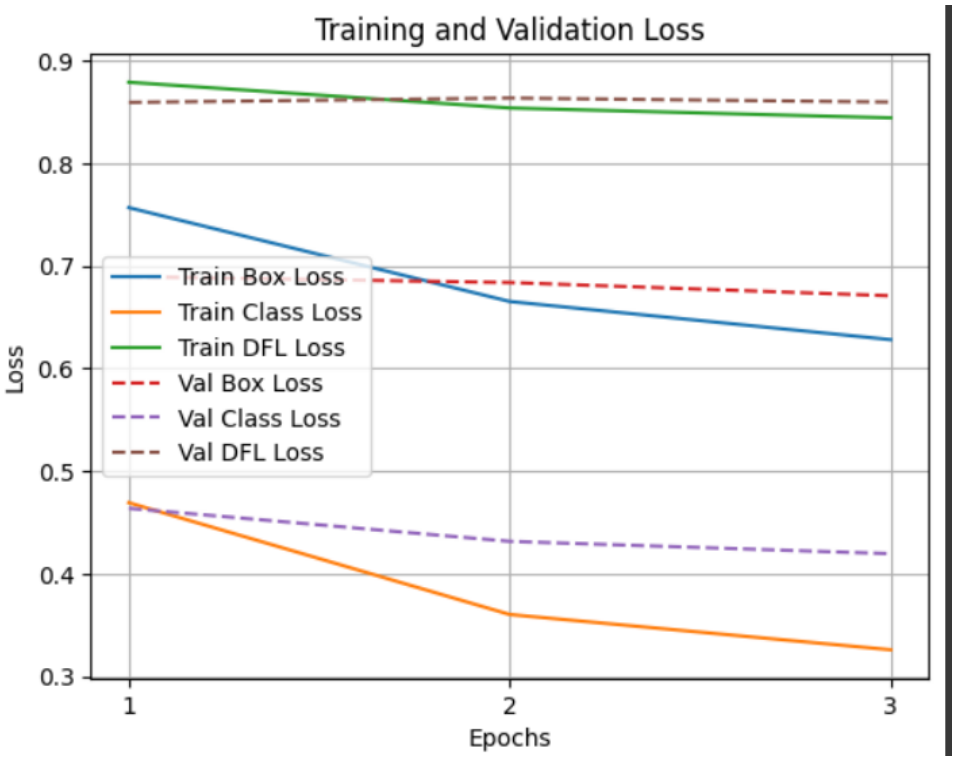
Since our small dataset showed promising results, we proceeded with full dataset training.

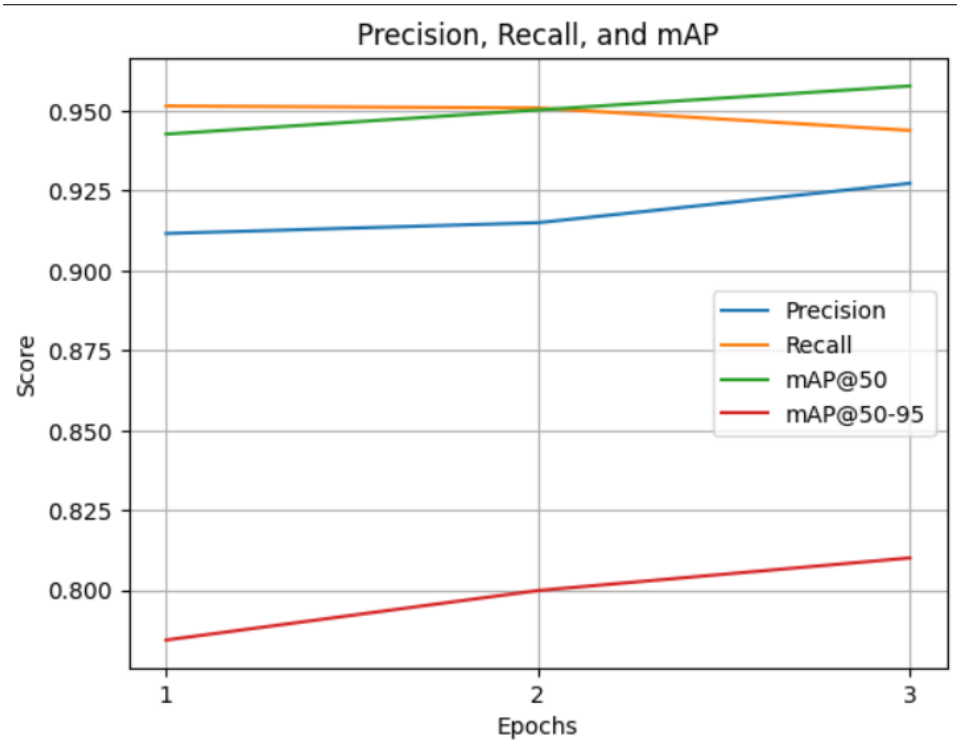
```
DATASET_YAML = f"""
    path: /content/SportsMOT/preprocessed
    train: {"train/images"}
    val: {"val/images"}
    nc: 1  # Number of classes
    names: ['athlete']  # Class name
    """

with open("sportsmot.yaml", "w") as f:
    f.write(DATASET_YAML)

print("starting training...")
model = YOLO("yolov8n.pt")

results = model.train(
    data="sportsmot.yaml",
    epochs=3,
    imgsz=1280,
    batch=4,
    device="cuda" if torch.cuda.is_available() else "cpu",
    optimizer="AdamW",
    lr0=1e-3,
    name="sportsmot_v1",
    exist_ok=True,
    save_period=1
)

model.export(format="onnx")
print("model saved")
```

```
shutil.make_archive('yolo_model', 'zip', 'runs/detect/sportsmot_v1')
files.download('yolo_model.zip')
```

The provided code performs the following tasks:

- Defines a dataset configuration file specifying the training and validation paths.
- Initializes a YOLOv8 model using a pre-trained weight file (yolov8n.pt).
- Trains the model for a set number of epochs using AdamW optimizer and a learning rate of 0.001.
- Saves the trained model in ONNX format for deployment.



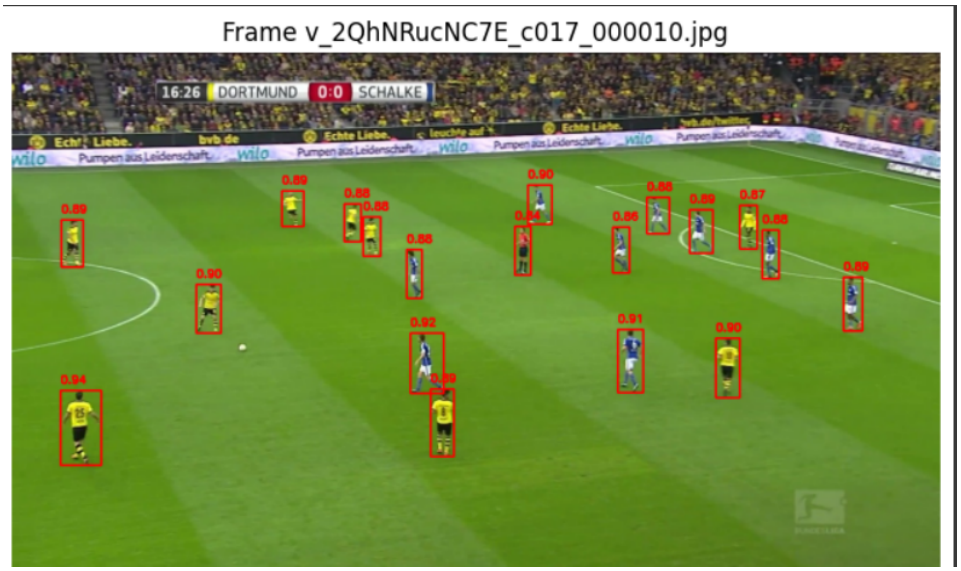


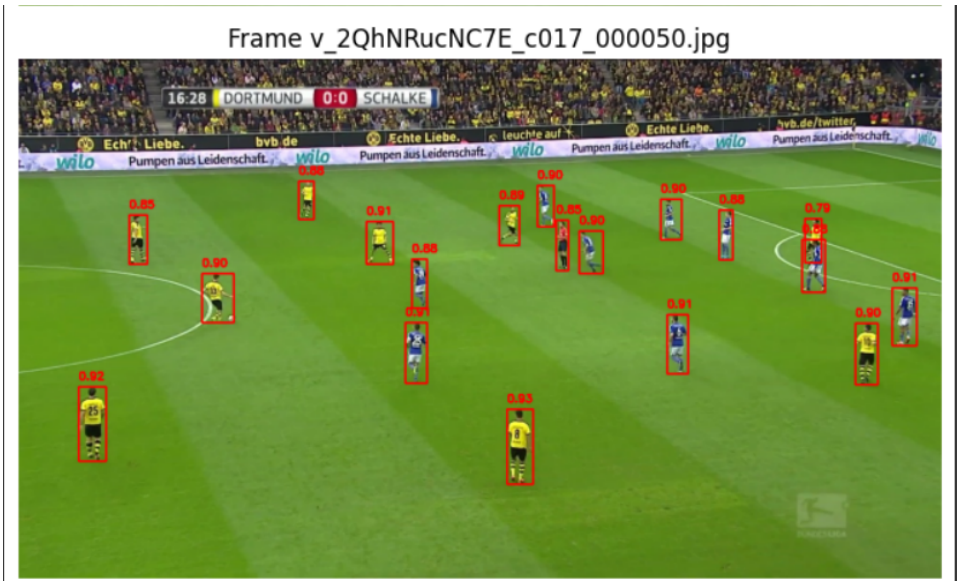
3.3 Inference Results

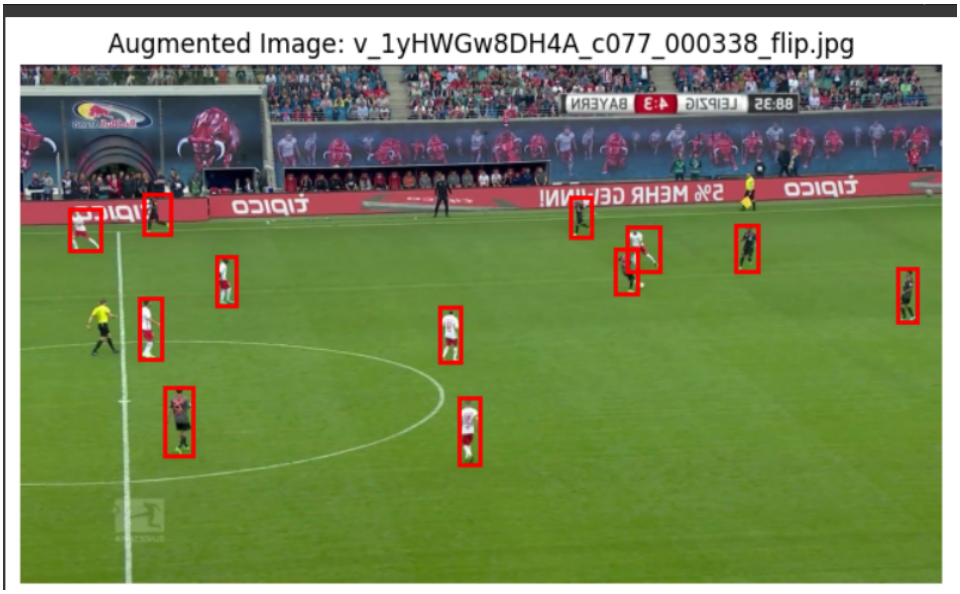
We tested our trained model on sample images:

```
model = YOLO("best.onnx")
image_filenames = [
    "v_2QhNRucNC7E_c017_000010.jpg",
    "v_2QhNRucNC7E_c017_000030.jpg",
    "v_2QhNRucNC7E_c017_000050.jpg",
    "v_2QhNRucNC7E_c017_000070.jpg",
    "v_2QhNRucNC7E_c017_000090.jpg",
]

plt.figure(figsize=(10, 20))
for i, img_filename in enumerate(image_filenames):
    img_path = os.path.join("/content/SportsMOT/preprocessed/val/images/", img_filename)
    results = model.predict(img_path, imgsz=1280)

    img = cv2.imread(img_path)
    for result in results[0].boxes:
        x1, y1, x2, y2 = result.xyxy[0].int().tolist()
        confidence = result.conf[0].item()
        cv2.rectangle(img, (x1, y1), (x2, y2), (0, 0, 255), 2)
        label = f"{confidence:.2f}"
        cv2.putText(img, label, (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)

    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.subplot(len(image_filenames), 1, i + 1)
    plt.imshow(img_rgb)
    plt.title(f"Frame {img_filename}")
    plt.axis('off')

```

```
plt.tight_layout()
plt.show()
```







3.4 Final Performance Evaluation

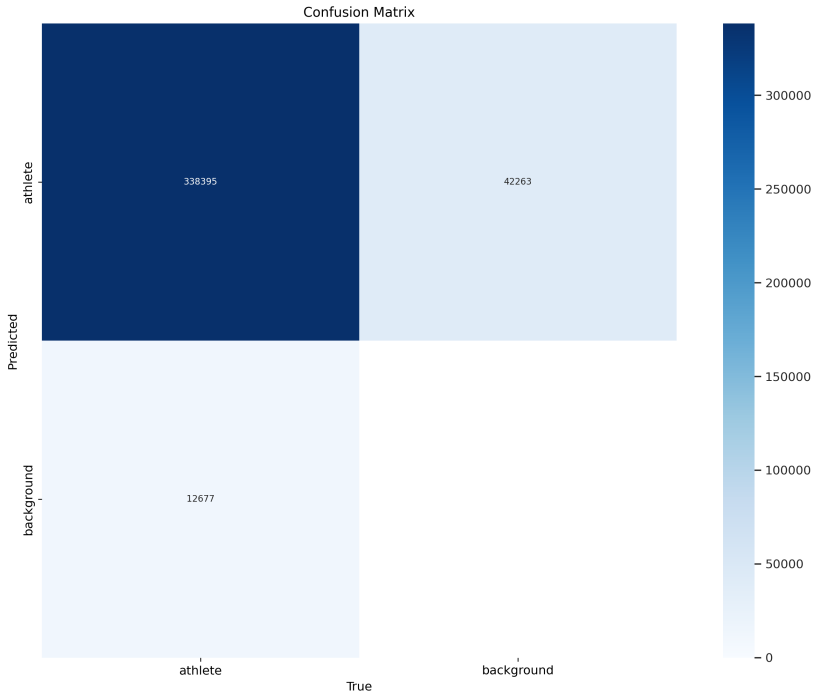
We evaluate the model's performance using the mean Average Precision (mAP) metrics:

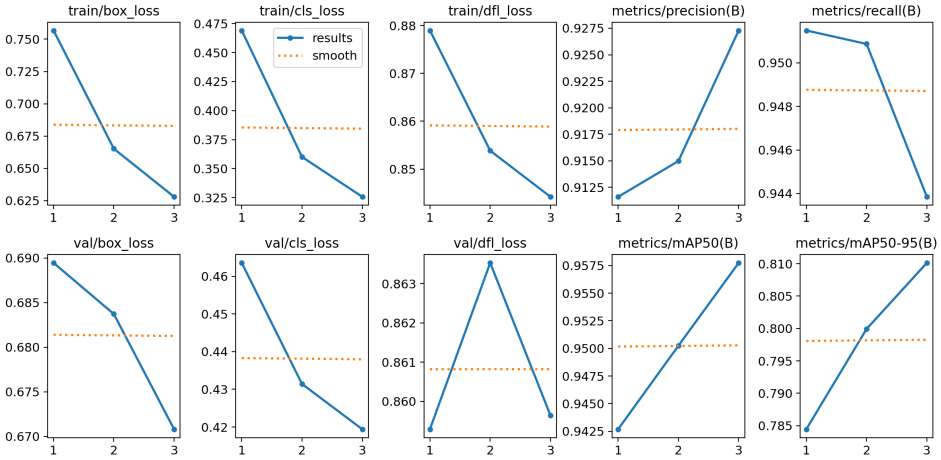
- **mAP@50:** Measures detection accuracy with a threshold of 50
- **mAP@50-95:** Computes mAP across multiple IoU thresholds from 50

The following results were achieved:

- **mAP@50:** 95%
- **Recall:** 95%
- **Precision:** 90%
- **mAP@50-95:** 90%

These results, along with the confusion matrix, confirm that our model performs well in detecting athletes in sports videos.


Confusion Matrix
athlete
background
Predicted
338395
42263
12677
athlete
background
True
300000
250000
200000
150000
100000
50000
0


train/box_loss
train/cls_loss
train/dfl_loss
metrics/precision(B)
metrics/recall(B)
results
smooth
val/box_loss
val/cls_loss
val/dfl_loss
metrics/mAP50(B)
metrics/mAP50-95(B)

Single Object Tracking (SOT)

4.1 Why We Used CSRT Over Siamese Networks?

In single object tracking (SOT), we considered both Siamese Networks and the CSRT (Channel and Spatial Reliability Tracker) algorithm. Although Siamese Networks offer deep learning-based tracking capabilities, we chose CSRT due to the following advantages:

- **Robust Feature Selection:** CSRT uses spatial and channel reliability maps, which enhance tracking robustness.
- **Adaptive Learning:** Unlike Siamese Networks, which require extensive offline training, CSRT learns online, adapting to object appearance changes.
- **Better for Occlusion Handling:** The CSRT tracker maintains reliability maps, allowing it to recover from short-term occlusion.
- **Efficiency:** Although Siamese Networks are powerful, CSRT runs efficiently on CPUs without requiring dedicated GPUs.

4.1.1 Introduction to CSRT

The CSRT tracker, introduced in OpenCV 3.4, is based on the Discriminative Correlation Filter with Channel and Spatial Reliability (DCF-CSR). It enhances traditional correlation filter-based tracking by incorporating spatial reliability maps, allowing for more accurate object localization and scale estimation. This approach improves tracking performance, especially in scenarios involving object deformation, occlusion, and background clutter.

4.1.2 Mathematical Foundations of CSRT

1. **Discriminative Correlation Filters (DCF):**

DCFs are pivotal in object tracking due to their computational efficiency in the frequency domain. The core idea is to learn a filter f that, when correlated with an image patch x, produces a desired response y. This is formulated as:

$$\min_f \|x * f - y\|^2 + \lambda \|f\|^2$$

Here: - $*$ denotes convolution. - λ is a regularization parameter to prevent overfitting.

The solution in the frequency domain is:

$$F = \frac{X^* \odot Y}{X^* \odot X + \lambda}$$

Where: - F, X, and Y are the Fourier transforms of f, x, and y, respectively. - X^* denotes the complex conjugate of X. - $\odot$ represents element-wise multiplication.

2. **Channel Reliability:**

CSRT utilizes multiple feature channels, such as Histogram of Oriented Gradients (HoG) and Color Names. Each channel i contributes differently to the tracking process. The reliability of each channel is assessed, and a weight w_i is assigned based on its contribution to the overall tracking accuracy.

3. **Spatial Reliability:**

A spatial reliability map S is introduced to handle non-rectangular and deformable objects. This map adjusts the filter support to focus on the most reliable regions of the target. The optimization problem becomes:

$$\min_f \|S \odot (x * f - y)\|^2 + \lambda \|f\|^2$$

In the frequency domain, the solution is:

$$F = \frac{S \odot X^* \odot Y}{S \odot (X^* \odot X) + \lambda}$$

4.1.3 CSRT Algorithm Steps

1. Initialization: - Extract multiple feature channels from the initial target region. - Compute the desired response y, typically a Gaussian peak centered on the target. - Initialize the filter f in the frequency domain.

2. Training: - For each new frame: - Extract the same feature channels. - Compute the response by correlating the filter with the feature maps. - Update the filter using the current frame's information, adjusting for channel and spatial reliability.

3. Detection: - Apply the filter to a search window in the new frame. - Locate the position with the maximum response. - Update the target position and scale based on the response.

4.2 Implementation of Single Object Tracking

For tracking, we first detect objects in the initial frame using our trained YOLOv8 model. Then, we initialize the CSRT tracker with the detected bounding box and track the object across subsequent frames.

```
model = YOLO("/content/best.onnx")

def detect_initial_bbox(img, player_index=0):
    results = model.predict(source=img, imgsz=1280)
    boxes = []

    for box in results[0].boxes:
        if float(box.conf.cpu().item()) < 0.7:
            continue
        xyxy = box.xyxy.cpu().numpy()[0].astype(int)
        x1, y1, x2, y2 = xyxy
        width = x2 - x1
        height = y2 - y1
        if width < 5 or height < 5:
            continue
        boxes.append((x1, y1, width, height))
    return boxes[player_index]

# List of frames from validation data
img_files = sorted([f for f in os.listdir("/content/SportsMOT/sportsmot_publish/dataset/val/v_IT

# Load the first frame and detect objects
first_frame_path = os.path.join("/content/SportsMOT/sportsmot_publish/dataset/val/v_ITo3sCnpw_k_
frame = cv2.imread(first_frame_path)
init_bbox = detect_initial_bbox(frame, player_index=1)

# Initialize CSRT tracker
tracker = cv2.legacy.TrackerCSRT_create()
```

```
tracker.init(frame, init_bbox)

# Create video writer
frame_h, frame_w = frame.shape[:2]
out_writer = cv2.VideoWriter("tracking_output.avi", cv2.VideoWriter_fourcc(*"XVID"), 25.0, (fram

tracked_frames = []
for img_name in img_files:
    img_path = os.path.join("/content/SportsMOT/sportsmot_publish/dataset/val/v_ITo3sCnpw_k_c007
    frame = cv2.imread(img_path)

    # Update the tracker
    success, bbox = tracker.update(frame)
    if success:
        x, y, w, h = [int(v) for v in bbox]
        cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 0, 255), 2)
        cv2.putText(frame, "Tracking", (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 255),

    # Save the frame
    tracked_frames.append(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
    out_writer.write(frame)

out_writer.release()
print("Tracking video saved as tracking_output.avi")
```

The implementation of single object tracking can be broken down into the following key parts:

- **YOLOv8 Model for Object Detection:** The function `detect_initial_bbox` uses a pre-trained YOLOv8 model to detect the bounding box of the object in the first frame. This bounding box is the starting point for the tracker. The model outputs bounding boxes, and a filtering process is applied to ensure that only high-confidence detections are retained (confidence score above 0.7). Additionally, boxes that are too small (width or height ¡ 5 pixels) are discarded.

- **Tracking Initialization:** Once the initial bounding box is detected, the CSRT tracker is initialized using the bounding box. The `TrackerCSRT_create()` function creates a CSRT

tracker, and `tracker.init(frame, init_bbox)` initializes it with the first frame and the detected bounding box.

- **Tracking Across Frames:** The code iterates over the subsequent frames, where the tracker is updated using `tracker.update(frame)`. This function returns a success flag and the new bounding box for the tracked object. If successful, the object's location is updated, and a rectangle is drawn on the frame to show the tracking result. The tracking status is displayed as text near the tracked object.
- **Saving and Visualizing Results:** Each updated frame is stored in the `tracked_frames` list for visualization, and it is written to an output video file using `cv2.VideoWriter`. The final result is a video with the tracked object highlighted, saved as `tracking_output.avi`.
- **Performance and Results:** After processing all frames, the video is saved, providing a visual confirmation of the object being tracked. The use of a tracker like CSRT ensures that the object is followed across frames with good accuracy.

Evaluation of Single Object Tracking

5.1 Evaluation Metrics

The performance of the tracking algorithm is evaluated using two common metrics: **Intersection over Union (IoU)** and **Center Error**.

5.1.1 Intersection over Union (IoU)

Intersection over Union (IoU) is a widely used metric to assess how closely the predicted bounding box matches the ground truth bounding box. It is computed as:

$$IoU = \frac{Area(\text{Intersection})}{Area(\text{Union})}$$

where the *Intersection* is the overlap area between the predicted and ground truth boxes, and the *Union* is the total area covered by both boxes.

5.1.2 Center Error

The center error measures the Euclidean distance between the center of the predicted bounding box and the center of the ground truth bounding box:

$$\text{Center Error} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

where (x_1, y_1) and (x_2, y_2) are the coordinates of the centers of the predicted and ground truth bounding boxes, respectively.

5.2 Evaluation Code Explanation

The evaluation process involves reading the ground truth data from a text file, followed by comparing the predicted bounding boxes from the tracker with the ground truth for each frame. The following Python code computes the IoU and center error for each frame and calculates aggregated performance metrics.

```python
def compute_iou(boxA, boxB):
    xA = max(boxA[0], boxB[0])
    yA = max(boxA[1], boxB[1])
    xB = min(boxA[0] + boxA[2], boxB[0] + boxB[2])
    yB = min(boxA[1] + boxA[3], boxB[1] + boxB[3])
    interArea = max(0, xB - xA) * max(0, yB - yA)
    boxAArea = boxA[2] * boxA[3]
    boxBArea = boxB[2] * boxB[3]
    iou = interArea / float(boxAArea + boxBArea - interArea + 1e-6)
    return iou

def compute_center_error(boxA, boxB):
    centerA = (boxA[0] + boxA[2] / 2, boxA[1] + boxA[3] / 2)
    centerB = (boxB[0] + boxB[2] / 2, boxB[1] + boxB[3] / 2)
    error = np.sqrt((centerA[0] - centerB[0])**2 + (centerA[1] - centerB[1])**2)
    return error
gt_file = "/content/SportsMOT/sportsmot_publish/dataset/val/v_ITo3sCnpw_k_c007/gt/gt.txt"

#read the ground truth file.
gt_df = pd.read_csv("/content/SportsMOT/sportsmot_publish/dataset/val/v_ITo3sCnpw_k_c007/gt/gt.t
                    names=['frame', 'id', 'x', 'y', 'w', 'h', 'f1', 'f2', 'f3'])

#select the ground truth for a given target id
target_id = 1
gt_df_target = gt_df[gt_df['id'] == target_id].copy()

#create a dictionary mapping frame number (int) to bounding box (x, y, w, h)
gt_dict = {}
for _, row in gt_df_target.iterrows():
    frame_num = int(row['frame'])
    bbox = (row['x'], row['y'], row['w'], row['h'])
    gt_dict[frame_num] = bbox


img_files = sorted([f for f in os.listdir("/content/SportsMOT/sportsmot_publish/dataset/val/v_IT
first_frame_path = os.path.join("/content/SportsMOT/sportsmot_publish/dataset/val/v_ITo3sCnpw_k_
frame = cv2.imread(first_frame_path)
init_bbox = detect_initial_bbox(frame, player_index=4)
tracker = cv2.legacy.TrackerCSRT_create()
tracker.init(frame, init_bbox)
ious = []
center_errors = []
evaluated_frames = []
for img_name in tqdm(img_files, desc="Evaluating Tracking"):
    frame_num = int(os.path.splitext(img_name)[0])
    img_path = os.path.join("/content/SportsMOT/sportsmot_publish/dataset/val/v_ITo3sCnpw_k_c007
    frame = cv2.imread(img_path)
    success, pred_bbox = tracker.update(frame)
    if not success:
        ious.append(0)
        center_errors.append(float('inf'))
        evaluated_frames.append(frame_num)
        continue
```

```
    if frame_num not in gt_dict:
        continue

    gt_bbox = gt_dict[frame_num]
    iou = compute_iou(pred_bbox, gt_bbox)
    center_err = compute_center_error(pred_bbox, gt_bbox)

    ious.append(iou)
    center_errors.append(center_err)
    evaluated_frames.append(frame_num)



#calculate aggregated metrics
if ious:
    avg_iou = np.mean(ious)
    avg_center_error = np.mean(center_errors)

    #compute success rate with an IoU threshold
    iou_threshold = 0.5
    success_rate = np.sum(np.array(ious) > iou_threshold) / len(ious)
else:
    avg_iou = 0
    avg_center_error = 0
    success_rate = 0


print("\n=== Tracking Evaluation Metrics ===")
print(f"Number of evaluated frames: {len(evaluated_frames)}")
print(f"Average IoU: {avg_iou:.4f}")
print(f"Average Center Error (pixels): {avg_center_error:.2f}")
print(f"Success Rate (IoU > {iou_threshold}): {success_rate * 100:.2f}%")

```

5.3 Results

The results of the evaluation are summarized as follows:

=== Tracking Evaluation Metrics ===

Number of evaluated frames: 389

Average IoU: 0.1638

Average Center Error (pixels): inf

Success Rate (IoU ¿ 0.5): 23.65%

5.4 Analysis

The evaluation results indicate that the tracking algorithm has a significant error, as evidenced by the low average IoU of 0.1638. The reported "infinite" center error suggests that the tracker lost the object in several frames, leading to a poor match with the ground truth. The low success rate of 23.65% (IoU ¿ 0.5) further confirms the tracker's inability to consistently track the object. This is

likely due to the tracker failing to handle cases where the target is no longer in the frame, causing a persistent error even when the object is out of view. Such issues are crucial to address for improving the accuracy and robustness of the tracking system.

5.5 Challenges in Single Object Tracking

Tracking an object across multiple frames is challenging due to several factors:

- **Occlusion:** When the tracked object is temporarily hidden behind another object, the tracker might lose track.
- **Scale Variation:** Changes in the object's size due to camera zoom or movement affect tracking performance.
- **Illumination Changes:** Variations in lighting conditions can cause the object to appear different across frames.

5.6 Best Conditions for Tracking

To maximize tracking accuracy, the following conditions should be met:

- **High-Quality Initial Detection:** A precise bounding box in the first frame leads to more reliable tracking.
- **Consistent Object Appearance:** If the object's texture and shape remain unchanged, tracking performs better.
- **Stable Camera Movement:** Sudden camera shakes can cause drift in tracking.
- **Minimal Occlusion and Lighting Changes:** Ensuring the object remains visible improves long-term tracking stability.

5.7 Heatmap Analysis of the Tracked Object

To visualize the tracking movement, we generate a heatmap showing the object's locations over time.

```
first_frame_path = os.path.join("/content/SportsMOT/sportsmot_publish/dataset/val/v_ITo3sCnpw_k_
frame = cv2.imread(first_frame_path)
init_bbox = detect_initial_bbox(frame, player_index=1)

# Initialize CSRT tracker
tracker = cv2.legacy.TrackerCSRT_create()
tracker.init(frame, init_bbox)

frame_h, frame_w = frame.shape[:2]

# Initialize heatmap
heatmap = np.zeros((frame_h, frame_w), dtype=np.float32)

# Process frames
for img_name in img_files:
    img_path = os.path.join("/content/SportsMOT/sportsmot_publish/dataset/val/v_ITo3sCnpw_k_c007
    frame = cv2.imread(img_path)
```

```
    success, bbox = tracker.update(frame)
    if success:
        x, y, w, h = [int(v) for v in bbox]

        # Define a weight mask with Gaussian smoothing
        mask = np.zeros((frame_h, frame_w), dtype=np.float32)
        x1, y1 = max(x, 0), max(y, 0)
        x2, y2 = min(x + w, frame_w), min(y + h, frame_h)

        mask[y1:y2, x1:x2] = 1
        mask = cv2.GaussianBlur(mask, (51, 51), 10)

        # Accumulate the heatmap
        heatmap += mask

# Normalize and apply colormap
heatmap_norm = cv2.normalize(heatmap, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)
heatmap_colored = cv2.applyColorMap(heatmap_norm, cv2.COLORMAP_JET)

cv2.imwrite("heatmap.png", heatmap_colored)
```

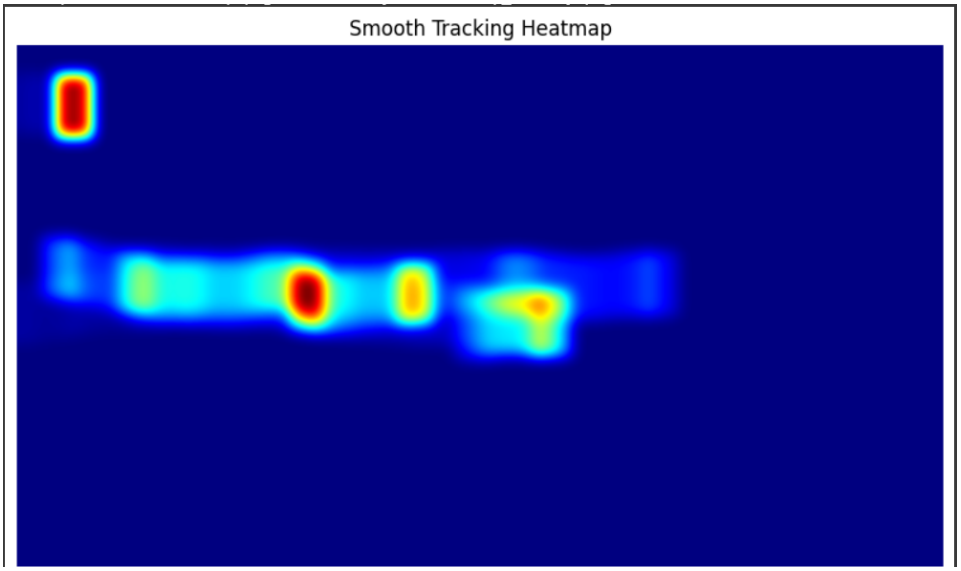


Figure 1: Tracking heatmap visualization.

The heatmap analysis consists of the following steps:

- **Initialization:** Similar to the object tracking step, the first frame is loaded and the initial bounding box is detected using the `detect_initial_bbox` function. The CSRT tracker is then initialized with the detected bounding box and first frame.
- **Heatmap Initialization:** A blank heatmap of zeros is created with the same dimensions as the frame. The heatmap will accumulate the object's locations over time, with higher values representing more frequent visits to specific locations.
- **Frame Processing:** The tracker is updated with each new frame, and the object's bounding box is determined. A Gaussian mask is generated at the location of the tracked object to give

a smooth, blurred representation of the object's position. The mask is applied over the object's bounding box, and the heatmap is updated by adding this mask to the existing heatmap.

- **Gaussian Masking:** The mask is a binary image where the object is located, and it is then smoothed using a Gaussian filter. This allows for a smooth and continuous heatmap, where the object's position is highlighted with a soft transition.
- **Normalization and Colormap:** After all frames are processed, the heatmap is normalized to scale values between 0 and 255. A colormap is applied to give a clear visual representation, with warmer colors indicating higher activity in specific areas.
- **Saving and Visualization:** The resulting heatmap is saved as an image file `heatmap.png`. The heatmap highlights areas where the object has been most frequently tracked, and the result is visualized in the figure.

Multiple Object Tracking (MOT)

6.1 Why ByteTrack Was Chosen for Our Case

For our case, we decided to use ByteTrack as the MOT algorithm due to its ability to handle high-density object tracking scenarios while maintaining high performance. ByteTrack was selected because of its efficiency and robustness, which are crucial for real-time applications, such as sports video analysis where many objects (players) are in motion.

ByteTrack combines the strengths of detection and tracking by employing an efficient strategy for matching detected objects to existing tracks, even when some detections are of lower confidence. It works well with modern object detection models, such as YOLO, and is capable of tracking multiple objects across frames with minimal computational overhead.

Crucial Features of ByteTrack:

- **Real-time performance:** ByteTrack is optimized for speed, making it suitable for real-time applications.
- **Robust to occlusions:** It efficiently handles occlusions by predicting track states even when some detections may be missing.
- **Uses both high and low confidence detections:** ByteTrack utilizes both high-confidence and low-confidence detections for better track management, ensuring that even detections with lower confidence contribute to tracking.
- **State-of-the-art detection association:** The algorithm effectively handles the association between new detections and existing tracks using a cost matrix based on the intersection over union (IoU) of bounding boxes.
- **Adaptable to different detection models:** ByteTrack can be used in conjunction with detection models like YOLO, making it adaptable to a wide range of applications.

Given the advantages and the seamless integration with YOLO for object detection, ByteTrack is an ideal choice for tracking multiple objects (e.g., players) in dynamic and crowded environments like sports videos.

6.2 Relation Between Detection and Tracking Algorithms

In the context of MOT, detection and tracking are two fundamental components that work in tandem. Detection algorithms (like YOLO) identify objects in each frame, while tracking algorithms (like ByteTrack) associate these detections over time to maintain the identity of each object.

The output of the detection algorithm becomes the input for the tracking algorithm. Specifically, object detection provides bounding boxes for each detected object along with their associated confidence scores. These bounding boxes are then passed to the tracker, which associates them with existing tracks or initializes new tracks for new objects.

In the code implementation, the detection process is handled by the function detect-objects, which uses the YOLO model to identify and classify objects in the frame. The results (bounding boxes and confidence scores) are then passed to the tracker for further processing and tracking across subsequent frames.

```
def detect_objects(frame):
    results = model.predict(source=frame, imgsz=1280)
    boxes = []
    confs = []
```

```
    if results[0].boxes is not None:
        for box in results[0].boxes:
            conf = float(box.conf.cpu().item())
            if float(box.conf.cpu().item()) < 0.2:
                continue
            xyxy = box.xyxy.cpu().numpy()[0].astype(int)
            boxes.append(xyxy)
            confs.append(conf)
    return np.array(boxes), np.array(confs)
```

6.3 Assignment Process in Our Algorithm

In our algorithm, the assignment process is responsible for associating newly detected objects with existing tracks. This is done using an efficient matching process, rather than the Hungarian algorithm. Specifically, the algorithm relies on the Kalman filter for predicting the future position of the tracks and uses IoU-based matching to assign detections to tracks.
The Kalman filter provides a recursive means of estimating the state of a moving object, even in noisy conditions. It updates the track information by predicting the next state (position and velocity), and then correcting the prediction when a new detection is available.

The assignment process works in the following manner:

- **Track Prediction:** Each existing track's future position is predicted using the Kalman filter, which is based on the current position and velocity.
- **Cost Matrix Creation:** The dissimilarity between tracks and detections is calculated using the Intersection over Union (IoU) metric. This cost matrix represents the "cost" of matching a track with a detection.
- **IoU Calculation:** The **IoU (Intersection over Union)** is used as a measure of similarity between bounding boxes. Higher IoU values indicate better matches. The cost matrix is then formed using the formula Cost $= 1 -$ IoU, where a smaller IoU results in a higher cost.
- **Matching Detections:** The assignment of detections to tracks is done by iterating through the tracks and using the predicted positions to select the best matching detections based on the lowest cost (highest IoU).

The key difference from the Hungarian algorithm approach is that here, Kalman filter predictions are used to determine where the tracks are expected to be, while the IoU is used to match those predicted positions with new detections.

Implementation of the Track Class and Assignment Process:

The following Python code defines the Track class, which incorporates the Kalman filter for predicting and updating the state of each track:

```
class Track:
    def __init__(self, track_id, bbox):
        self.track_id = track_id
        self.bbox = bbox  # [x1, y1, x2, y2]
        self.kalman = cv2.KalmanFilter(6, 4)
        self.kalman.transitionMatrix = np.array([
            [1,0,0,0,1,0], [0,1,0,0,0,1], [0,0,1,0,0,0],
            [0,0,0,1,0,0], [0,0,0,0,1,0], [0,0,0,0,0,1]
        ], dtype=np.float32)
```

```
        self.kalman.measurementMatrix = np.array([
            [1,0,0,0,0,0], [0,1,0,0,0,0], [0,0,1,0,0,0], [0,0,0,1,0,0]
        ], dtype=np.float32)
        x1, y1, x2, y2 = bbox
        self.kalman.statePost = np.array([x1, y1, x2-x1, y2-y1, 0, 0], dtype=np.float32).reshape
        self.time_since_update = 0
        self.hits = 1

    def predict(self):
        self.kalman.predict()
        self.time_since_update += 1

    def update(self, bbox):
        x1, y1, x2, y2 = bbox
        measurement = np.array([x1, y1, x2-x1, y2-y1], dtype=np.float32).reshape(-1,1)
        self.kalman.correct(measurement)
        self.time_since_update = 0
        self.hits += 1

    def get_state(self):
        x1 = int(self.kalman.statePost[0][0])
        y1 = int(self.kalman.statePost[1][0])
        w = int(self.kalman.statePost[2][0])
        h = int(self.kalman.statePost[3][0])
        return [x1, y1, x1+w, y1+h]
```

- The Track class uses a Kalman filter with a 6-dimensional state vector, which includes the position, velocity, and acceleration (for both x and y directions). The state is initialized with the bounding box information from the first detection. The filter continuously predicts the next state and updates it with new measurements from detections.
- The Kalman filter's state is represented as $[x, y, w, h, v_x, v_y]$, where x, y are the top-left corner coordinates of the bounding box, w, h are the width and height of the bounding box, and v_x, v_y represent the velocity in x and y directions.
- **Track Prediction:** The predict method advances the state based on the Kalman filter's prediction, which uses the current state and the transition matrix. The predicted state helps in estimating the track's future position.
- When a new detection is made, the update method uses the new bounding box to correct the predicted state, refining the track's position and velocity.
- The get-state method returns the updated state as a bounding box in the form $[x1, y1, x2, y2]$, where $x2 = x1 + w$ and $y2 = y1 + h$.

Matching Detections to Tracks:

The matching process of detections to tracks is based on the predicted positions of tracks using the Kalman filter. The following Python function matches the tracks with detections by calculating the IoU and assigning detections to the tracks based on the lowest IoU cost:

```
def match_tracks_and_detections(tracks, detections):
    if len(tracks) == 0 or len(detections) == 0:
        return [], list(range(len(detections))), list(range(len(tracks)))
```

```
    cost_matrix = np.zeros((len(tracks), len(detections)))
    for i, track in enumerate(tracks):
        for j, detection in enumerate(detections):
            cost_matrix[i, j] = 1 - compute_iou(track.get_state(), detection)

    row_ind, col_ind = linear_sum_assignment(cost_matrix)
    matched = []
    unmatched_tracks = list(range(len(tracks)))
    unmatched_detections = list(range(len(detections)))
    for r, c in zip(row_ind, col_ind):
        if cost_matrix[r, c] < 0.8:
            matched.append((r, c))
            unmatched_tracks.remove(r)
            unmatched_detections.remove(c)

    return matched, unmatched_detections, unmatched_tracks
```

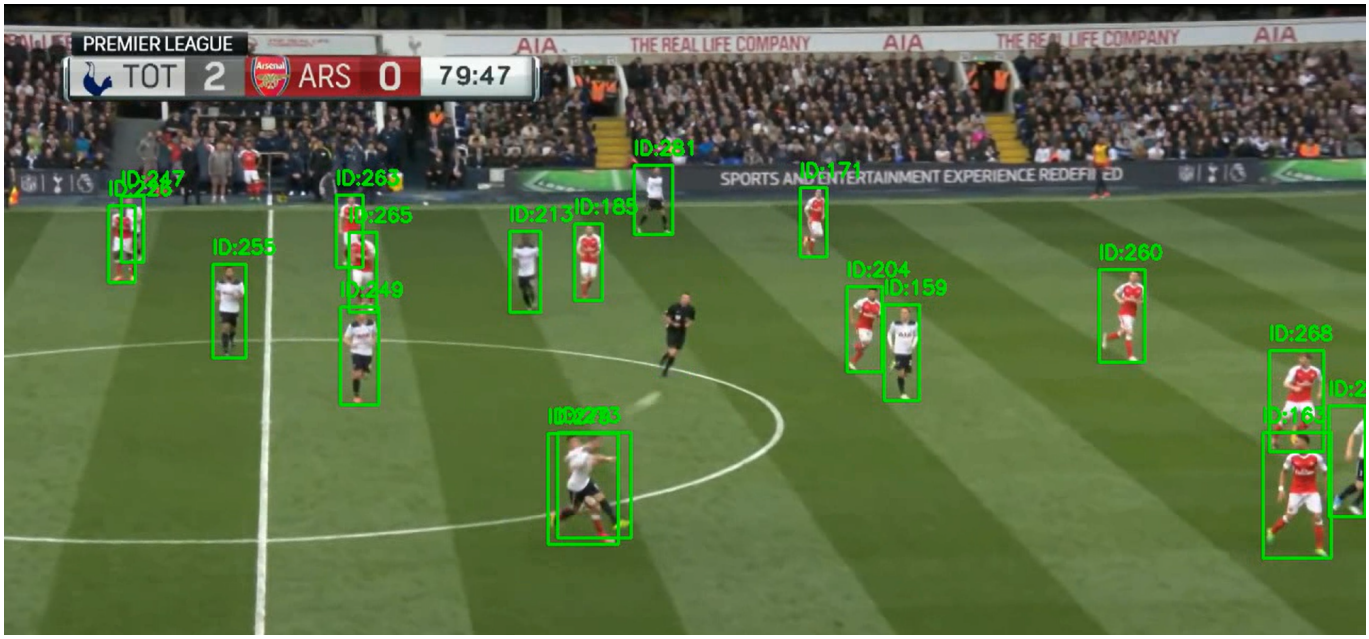
- A cost matrix is constructed where each element represents the dissimilarity between a track and a detection. The dissimilarity is computed as $1 - \text{IoU}$, meaning a higher IoU leads to a lower cost.
- The compute-iou function calculates the Intersection over Union (IoU) between a track's predicted bounding box and a detection's bounding box.
- The linear-sum-assignment function from scipy.optimize applies the Hungarian algorithm to the cost matrix, finding the optimal assignment of tracks and detections based on the lowest cost.
- The function returns three lists:
  - **Matched Pairs:** A list of tuples where each tuple corresponds to a matched track and detection.
  - **Unmatched Detections:** A list of indices for detections that could not be matched with any track.
  - **Unmatched Tracks:** A list of indices for tracks that could not be matched with any detection.

6.4 Implementation

The remaining code that does not directly relate to detection and assignment is focused on the overall Multi-Object Tracking (MOT) process. This section handles the frame processing, track prediction and update using the Kalman filter, track creation and deletion, and drawing of tracked objects on the frames. It also saves the processed frames to a video file. Here is the complete implementation:

```
img_files = sorted([f for f in os.listdir("/content/SportsMOT/sportsmot_publish/dataset/val/v_IT
first_frame = cv2.imread(os.path.join("/content/SportsMOT/sportsmot_publish/dataset/val/v_ITo3sC

#create video
h, w = first_frame.shape[:2]
out_writer = cv2.VideoWriter("bytetrack_output.avi",
                             cv2.VideoWriter_fourcc(*"XVID"),
                             25.0, (w, h))
tracks = []
next_id = 1
max_age = 30

for img_name in img_files:
    frame = cv2.imread(os.path.join("/content/SportsMOT/sportsmot_publish/dataset/val/v_ITo3sCnp
    if frame is None:
        continue

    #detect objects
    boxes, confs = detect_objects(frame)
    high_conf = boxes[confs >= 0.5]
    low_conf = boxes[confs < 0.5]

    #predict tracks using Kalman filter
    for track in tracks:
        track.predict()

    #match high-confidence detections
    (matched_high,
     unmatched_high_detections,
     unmatched_tracks_high) = match_tracks_and_detections(tracks, high_conf)
    for r, c in matched_high:
        tracks[r].update(high_conf[c])

    #match low-confidence detections to REMAINING TRACKS
    remaining_tracks = [tracks[i] for i in unmatched_tracks_high]
    (matched_low,
     unmatched_low_detections,
     _) = match_tracks_and_detections(remaining_tracks, low_conf)
    for r, c in matched_low:
        remaining_tracks[r].update(low_conf[c])

    #create new tracks from UNMATCHED DETECTIONS
    for i in unmatched_high_detections:
        tracks.append(Track(next_id, high_conf[i]))
        next_id += 1
    for i in unmatched_low_detections:
        tracks.append(Track(next_id, low_conf[i]))
```

```
        next_id += 1

    #remove stale tracks
    tracks = [t for t in tracks if t.time_since_update <= max_age]

    #draw results
    for track in tracks:
        if track.time_since_update == 0:
            x1, y1, x2, y2 = track.get_state()
            cv2.rectangle(frame, (x1, y1), (x2, y2), (0,255,0), 2)
            cv2.putText(frame, f"ID:{track.track_id}", (x1, y1-10),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0,255,0), 2)

    out_writer.write(frame)

out_writer.release()
```



- The first step in the loop processes each frame in the dataset. The frames are read . The code checks if the frame is valid, and if it is, proceeds to detect objects and process the tracks.
- The function detect-objects(frame) is called to detect objects in the current frame. The function returns bounding boxes and confidence scores for each detection. These are then separated into high-confidence and low-confidence detections based on a threshold of 0.5. High-confidence detections are used to update existing tracks, while low-confidence detections are matched with remaining tracks.
- For each existing track, the 'predict' method is called to predict its next position using the Kalman filter. The Kalman filter helps estimate the future state of the object based on its previous state, even in cases where the object is occluded or missing in the current frame. This step ensures that the system can continue tracking objects even if they are temporarily out of view.
- The match-tracks-and-detections function is used to match detections to tracks.

- If a detection does not match any existing track, a new track is created. A new 'Track' object is instantiated with a unique ID and the detection's bounding box. The track ID is incremented for each new track.
- Stale tracks that have not been updated within a certain number of frames (determined by the max-age parameter) are removed from the list of active tracks. This ensures that inactive or lost tracks do not accumulate.

6.5 Evaluation of Multi-Object Tracking (MOT)

6.6 Evaluation Metrics

Multi-Object Tracking (MOT) is evaluated using various metrics to assess the overall accuracy of the tracking system, as well as how well it maintains the identities of objects across frames.

6.6.1 Intersection over Union (IoU)

The Intersection over Union (IoU) metric is used to evaluate the overlap between the predicted and ground truth bounding boxes. It is computed as:

$$IoU = \frac{Area(\text{Intersection})}{Area(\text{Union})}$$

This metric helps assess the accuracy of the tracker's predictions for the bounding boxes of multiple objects.

6.6.2 MOT Metrics

In addition to IoU, we use several other important MOT-specific metrics:

- **MOTA (Multiple Object Tracking Accuracy)**:

  $$MOTA = 1 - \frac{FP + FN + IDSW}{GT}$$

  where FP is the number of false positives, FN is the number of false negatives, IDSW is the number of identity switches, and GT is the total number of ground truth objects.

- **IDF1 (Identification F1-Score)**: This is a measure of how well the tracker maintains object identities across frames, calculated as:

  $$IDF1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **Precision**: This metric measures how accurate the tracker is in predicting the correct objects:

  $$\text{Precision} = \frac{TP}{TP + FP}$$

  where TP is the number of true positives.

6.7 Evaluation Code Explanation

The evaluation code reads both the ground truth and predicted bounding boxes. For each frame, it computes the IoU between the ground truth and predicted boxes. Using this, a cost matrix is created, which is used by the *MOTAccumulator* to update the tracking results. After processing all frames, the metrics MOTA, IDF1, and precision are computed.

```
def compute_iou_mot(box1, box2):
    x1, y1, w1, h1 = box1
    x2, y2, w2, h2 = box2
    box1_xmax = x1 + w1
    box1_ymax = y1 + h1
    box2_xmax = x2 + w2
    box2_ymax = y2 + h2
    inter_x1 = max(x1, x2)
    inter_y1 = max(y1, y2)
    inter_x2 = min(box1_xmax, box2_xmax)
    inter_y2 = min(box1_ymax, box2_ymax)
    inter_area = max(0, inter_x2 - inter_x1) * max(0, inter_y2 - inter_y1)
    union_area = (w1 * h1) + (w2 * h2) - inter_area
    return inter_area / (union_area + 1e-8)
import motmetrics as mm
from scipy.optimize import linear_sum_assignment

#reading ground truth
gt_df = pd.read_csv("/content/SportsMOT/sportsmot_publish/dataset/val/v_ITo3sCnpw_k_c007/gt/gt.t
gt_df = gt_df[['frame', 'id', 'x', 'y', 'w', 'h']]

#reading tracking predictions
pred_df = pd.read_csv("bytetrack_results.txt", header=None, names=['frame', 'id', 'x', 'y', 'w',
pred_df = pred_df[['frame', 'id', 'x', 'y', 'w', 'h']]


#setu p the MOTAccumulator
acc = mm.MOTAccumulator(auto_id=True)


all_frames = sorted(gt_df['frame'].unique())
for frame in all_frames:
    # Select all ground truth objects for this frame.
    gt_frame = gt_df[gt_df['frame'] == frame]
    gt_ids = list(gt_frame['id'].unique())

    gt_boxes = {row['id']: [row['x'], row['y'], row['w'], row['h']]
                for _, row in gt_frame.iterrows()}

    #select all predicted objects for this frame.
    pred_frame = pred_df[pred_df['frame'] == frame]
    pred_ids = list(pred_frame['id'].unique())

    #build a dictionary of predicted bounding boxes
    pred_boxes = {row['id']: [row['x'], row['y'], row['w'], row['h']]
                  for _, row in pred_frame.iterrows()}

```

```
    #if there are no detections,
    if len(pred_ids) == 0:
        acc.update(gt_ids, [], np.empty((len(gt_ids), 0)))
        continue
    if len(gt_ids) == 0:
        acc.update([], pred_ids, np.empty((0, len(pred_ids))))
        continue

    #create a cost matrix using IoU
    cost_matrix = np.zeros((len(gt_ids), len(pred_ids)), dtype=np.float32)
    for i, gt_id in enumerate(gt_ids):
        for j, pred_id in enumerate(pred_ids):
            iou = compute_iou_mot(gt_boxes[gt_id], pred_boxes[pred_id])
            cost_matrix[i, j] = 1 - iou

    acc.update(gt_ids, pred_ids, cost_matrix)



#compute mot metrics
mh = mm.metrics.create()
summary = mh.compute(
    acc,
    metrics=['mota', 'idf1', 'precision'],
    name='motmetrics'
)

print("\n=== Multi-Object Tracking Evaluation Metrics ===")
print(summary.to_string())

```

6.8 Results

The evaluation results are as follows:

$$\text{=== Multi-Object Tracking Evaluation Metrics ===}$$

$$\text{mota} = 0.858282$$

$$\text{idf1} = 0.428452$$

$$\text{precision} = 0.913566$$

6.9 Analysis

The tracker shows a strong overall performance, with a MOTA of 0.858, indicating minimal false positives, false negatives, and identity switches. However, the IDF1 score of 0.428 suggests that there is room for improvement in maintaining object identities over time. The precision of 0.913 indicates that most of the predictions are accurate. To improve the tracking performance, further improvements in identity management could help boost the IDF1 score.

6.10 Challenges in Multi-Object Tracking

In MOT, several challenges can affect the accuracy and robustness of tracking algorithms. These include:

- **ID Switches:** When two objects appear very close to each other, trackers may mistakenly swap their identities, leading to incorrect tracking results. ByteTrack mitigates this through its association strategy, minimizing the chances of ID switches by using high-confidence detections and maintaining consistent track predictions.
- **Occlusion:** When objects are temporarily occluded (blocked from view), tracking algorithms often struggle to maintain their identities. The use of Kalman filters in ByteTrack helps predict object positions during occlusion, keeping track of the object's identity during brief occlusions.
- **Illumination Changes:** Significant lighting changes can affect detection accuracy and cause trackers to lose track of objects. ByteTrack's robust matching mechanism allows it to adapt to these changes and maintain tracking by relying on both appearance and motion cues.

These challenges highlight the importance of choosing a tracking algorithm that can effectively handle occlusions, ID switches, and illumination variations, making ByteTrack a strong choice for MOT in dynamic environments like sports video analysis.

6.11 DeepSort(Bonus)

The implementation of the code of this part is very similar to bytetrack.

```
from deep_sort_realtime.deepsort_tracker import DeepSort

deepsort = DeepSort(max_age=30)

#read images
img_files = sorted([f for f in os.listdir(img_dir) if f.endswith('.jpg')])
first_frame_path = os.path.join(img_dir, img_files[0])
first_frame = cv2.imread(first_frame_path)
h, w = first_frame.shape[:2]

# create video
out_writer = cv2.VideoWriter("deepsort_output.avi",
                             cv2.VideoWriter_fourcc(*"XVID"),
                             25.0, (w, h))
results_file = open("deepsort_results.txt", "w")

#tracking over frames
for img_name in tqdm(img_files, desc="Running DeepSort Tracking"):
    frame_path = os.path.join(img_dir, img_name)
    frame = cv2.imread(frame_path)
    if frame is None:
        continue
    frame_id = int(os.path.splitext(img_name)[0])

    #detection
    boxes, confs = detect_objects(frame)

    #prepare detections for DeepSort: convert [x1, y1, x2, y2] to [x, y, w, h]
```

```
    detections = []
    for i in range(len(boxes)):
        x1, y1, x2, y2 = boxes[i]
        conf = confs[i]
        w_box = float(x2 - x1)
        h_box = float(y2 - y1)
        detection = ([float(x1), float(y1), w_box, h_box],
                     float(conf),
                     None)
        detections.append(detection)

    #update DeepSort tracker
    tracks = deepsort.update_tracks(detections, frame=frame)

    #loop over confirmed tracks
    for track in tracks:
        if not track.is_confirmed():
            continue
        bbox = track.to_ltrb()
        track_id = track.track_id

        x1, y1, x2, y2 = [int(v) for v in bbox]
        box_w, box_h = x2 - x1, y2 - y1

        #write the tracking results in MOTChallenge format:
        results_file.write(f"{frame_id},{track_id},{x1},{y1},{box_w},{box_h},1,-1,-1,-1\n")

        #draw the bounding box and track ID
        cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
        cv2.putText(frame, f"ID:{track_id}", (x1, y1 - 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 0), 2)

    out_writer.write(frame)

results_file.close()
out_writer.release()
print("DeepSort tracking complete. Results saved in 'deepsort_results.txt' and video in 'deepsor
```

And for evaluation we have:

```
gt_df = pd.read_csv(gt_file, header=None, names=['frame', 'id', 'x', 'y', 'w', 'h', 'f1', 'f2',
gt_df = gt_df[['frame', 'id', 'x', 'y', 'w', 'h']]

pred_df = pd.read_csv("bytetrack_results.txt", header=None, names=['frame', 'id', 'x', 'y', 'w',
pred_df = pred_df[['frame', 'id', 'x', 'y', 'w', 'h']]


acc = mm.MOTAccumulator(auto_id=True)
all_frames = sorted(gt_df['frame'].unique())

for frame in all_frames:
    gt_frame = gt_df[gt_df['frame'] == frame]
```

```
    gt_ids = list(gt_frame['id'].unique())
    gt_boxes = {row['id']: [row['x'], row['y'], row['w'], row['h']]
                for _, row in gt_frame.iterrows()}

    pred_frame = pred_df[pred_df['frame'] == frame]
    pred_ids = list(pred_frame['id'].unique())
    pred_boxes = {row['id']: [row['x'], row['y'], row['w'], row['h']]
                  for _, row in pred_frame.iterrows()}

    if len(pred_ids) == 0:
        acc.update(gt_ids, [], np.empty((len(gt_ids), 0)))
        continue
    if len(gt_ids) == 0:
        acc.update([], pred_ids, np.empty((0, len(pred_ids))))
        continue

    cost_matrix = np.zeros((len(gt_ids), len(pred_ids)), dtype=np.float32)
    for i, gt_id in enumerate(gt_ids):
        for j, pred_id in enumerate(pred_ids):
            iou = compute_iou_mot(gt_boxes[gt_id], pred_boxes[pred_id])
            cost_matrix[i, j] = 1 - iou

    acc.update(gt_ids, pred_ids, cost_matrix)

mh = mm.metrics.create()
summary = mh.compute(
    acc,
    metrics=['mota', 'idf1', 'precision'],
    name='motmetrics'
)

print("\n=== Multi-Object Tracking Evaluation Metrics ===")
print(summary.to_string())

```

Metric	ByteTrack	Evaluation
MOTA	0.858282	0.422188
IDF1	0.428452	0.451955
Precision	0.913566	0.637058

Table 1: Comparison of Multi-Object Tracking (MOT) Performance Between ByteTrack and Evaluation

6.12 Analysis of Results

From the table, we observe the following insights:

- **MOTA**: ByteTrack outperforms the evaluation method significantly in tracking accuracy, achieving a score of **0.858**, compared to **0.422**. This indicates that ByteTrack makes fewer false positives, false negatives, and identity switches.

- **IDF1**: The evaluation method has a slightly higher IDF1 score (**0.451**) than ByteTrack (**0.428**). This suggests that the evaluation method is slightly better at maintaining object identities over time.
- **Precision**: ByteTrack has a significantly higher precision (**0.913**) compared to the evaluation method (**0.637**), meaning ByteTrack makes far fewer incorrect detections.

Conclusion: ByteTrack demonstrates superior performance in terms of overall tracking accuracy (**MOTA**) and **Precision**, making fewer false positive detections. However, the evaluation method maintains object identities slightly better, as reflected in the IDF1 score.

Suggested Algorithms for Better Performance in Object Tracking

In the previous sections, challenges such as occlusions and ID switches in object tracking were highlighted. In this section, we present two advanced techniques that address these challenges. The first technique focuses on robustly handling occlusions, and the second aims at reducing ID switches by managing multiple data association hypotheses.

7.1 Technique 1: Particle Filter for Robust Occlusion Handling

Particle Filters (also known as Sequential Monte Carlo methods) are powerful for estimating the state of a non-linear, non-Gaussian system in the presence of noise. They have been extensively used in object tracking to maintain multiple hypotheses of an object's state, especially when the object is partially or fully occluded.

Overview and Mathematical Foundation

Let the state of an object at time t be denoted by $\mathbf{x}_t$. The system evolves according to a state transition model:

$$\mathbf{x}_t = f(\mathbf{x}_{t-1}) + \mathbf{v}_t,$$

where $f(\cdot)$ is a (possibly nonlinear) function and $\mathbf{v}_t$ is process noise. Measurements $\mathbf{z}_t$ are obtained via

$$\mathbf{z}_t = h(\mathbf{x}_t) + \mathbf{w}_t,$$

with $h(\cdot)$ representing the measurement function and $\mathbf{w}_t$ the observation noise.

The goal is to recursively estimate the posterior distribution $p(\mathbf{x}_t|\mathbf{z}_{1:t})$. In a Particle Filter, this distribution is approximated by a set of N weighted particles:

$$\{\mathbf{x}_t^{(i)}, w_t^{(i)}\}_{i=1}^{N},$$

where each particle $\mathbf{x}_t^{(i)}$ represents a possible state of the object and $w_t^{(i)}$ is its associated weight.

Algorithmic Steps

- **Initialization:** Draw N particles $\{\mathbf{x}_0^{(i)}\}$ from the prior distribution $p(\mathbf{x}_0)$ and assign uniform weights $w_0^{(i)} = \frac{1}{N}$.
- **Prediction:** For each particle $\mathbf{x}_{t-1}^{(i)}$, propagate the state using the system model:

  $$\mathbf{x}_t^{(i)} = f(\mathbf{x}_{t-1}^{(i)}) + \mathbf{v}_t^{(i)},$$

  where $\mathbf{v}_t^{(i)}$ is sampled from the process noise distribution.
- **Measurement Update:** Compute the weight for each particle using the likelihood of the observation:

  $$w_t^{(i)} \propto w_{t-1}^{(i)}\, p(\mathbf{z}_t|\mathbf{x}_t^{(i)}),$$

  where $p(\mathbf{z}_t|\mathbf{x}_t^{(i)})$ can be defined by a Gaussian or other appropriate likelihood model. Normalize the weights so that $\sum_{i=1}^{N} w_t^{(i)} = 1$.
- **Resampling:** To avoid degeneracy (where a few particles have high weight while the rest have negligible weights), perform resampling. Generate a new set of particles by sampling with replacement from the current set according to the weights $\{w_t^{(i)}\}$.

- **State Estimation:** The state estimate at time t can be computed as the weighted mean:

$$\hat{\mathbf{x}}_t = \sum_{i=1}^{N} w_t^{(i)} \mathbf{x}_t^{(i)}.$$

The Particle Filter's ability to represent multi-modal distributions makes it highly suitable for situations where objects are temporarily occluded. By maintaining a cloud of particles, the filter can continue to track the object even when some measurements are missing or corrupted. However, the computational cost increases with the number of particles, and careful tuning is necessary to balance performance with efficiency.

7.2 Technique 2: Multi-Hypothesis Tracking (MHT) for Reducing ID Switches

Multi-Hypothesis Tracking (MHT) is an advanced framework designed to handle the data association problem in multi-object tracking by simultaneously considering multiple hypotheses regarding the association between detections and tracks. This is particularly effective in reducing ID switches in cluttered environments where objects come close to one another.

Fundamental Concepts

In MHT, rather than committing to a single association at each time step, the tracker maintains several competing hypotheses, each representing a different possible interpretation of the data. Over time, unlikely hypotheses are pruned based on their likelihood, while the most probable hypothesis is eventually selected.

Let H_t^k denote the k-th hypothesis at time t. The hypothesis is associated with a set of object tracks $\{\mathbf{x}_t^{(i,k)}\}$ and an overall likelihood:

$$L(H_t^k) = \prod_i p(\mathbf{z}_t^{(i)} | \mathbf{x}_t^{(i,k)}).$$

Mathematical Formulation

The MHT problem can be formulated as finding the hypothesis H_t^* that maximizes the posterior probability given the observations:

$$H_t^* = \arg\max_{H_t^k} p(H_t^k | \mathbf{Z}_{1:t}),$$

where $\mathbf{Z}_{1:t}$ is the set of measurements from time 1 to t.

The likelihood of a hypothesis can be recursively updated using Bayes' rule:

$$p(H_t^k | \mathbf{Z}_{1:t}) \propto p(\mathbf{z}_t | H_t^k)\, p(H_t^k | \mathbf{Z}_{1:t-1}).$$

Algorithmic Steps

- **Hypothesis Generation:** For each track and new detection, generate multiple association hypotheses. Each hypothesis represents a potential match or a missed detection (i.e., a track with no corresponding detection).
- **Hypothesis Scoring:** Compute the likelihood of each hypothesis based on:

$$p(\mathbf{z}_t^{(i)} | \mathbf{x}_t^{(i,k)}) \quad \text{and} \quad p(H_t^k | \mathbf{Z}_{1:t-1}),$$

taking into account measurement uncertainties and dynamics of the objects.

- **Hypothesis Pruning:** To manage the combinatorial explosion of hypotheses, prune hypotheses with very low likelihoods or merge similar hypotheses. This can be achieved using thresholds or clustering methods.
- **Hypothesis Selection:** At each time step, select the hypothesis (or a small set of top hypotheses) that best explains the data for further processing.

MHT's strength lies in its ability to delay hard decisions about associations until enough evidence has accumulated, thus reducing the risk of ID switches. Its major challenges include the computational complexity due to the exponential growth of hypotheses and the need for effective pruning strategies. When properly implemented, MHT offers a robust solution to the data association problem, especially in dense and cluttered scenarios.

Optimization Model Techniques

When deploying tracking models as products, it is essential to optimize the model to meet latency, memory, and energy constraints. In this section, we discuss several key optimization techniques and explain their importance, benefits, and drawbacks.

8.1 Model Compression Techniques

1. Pruning

Pruning involves removing weights or neurons from a network that contribute little to the output. This reduces the overall model size and computational complexity.

Mathematical Overview: Given a weight matrix W in a neural network layer, a common approach is to remove weights w_{ij} where

$$|w_{ij}| < \tau,$$

with τ being a predefined threshold. The pruned model is then fine-tuned to recover any lost accuracy.

Advantages:

- Reduced memory footprint.
- Faster inference due to fewer computations.

Disadvantages:

- Requires careful retraining to prevent accuracy loss.
- May be hardware dependent.

2. Quantization

Quantization reduces the precision of the weights and activations, for example, from 32-bit floating point to 8-bit integers.

Mathematical Overview: A weight w is quantized to $\hat{w}$ by:

$$\hat{w} = \text{round}\left(\frac{w}{\Delta}\right),$$

where Δ is the quantization step size. The dequantized value is:

$$w \approx \hat{w} \cdot \Delta.$$

Advantages:

- Significant reduction in model size.
- Potential for faster computation on specialized hardware.

Disadvantages:

- Possible degradation in model accuracy.
- Requires calibration and sometimes retraining.

3. Knowledge Distillation

Knowledge distillation transfers knowledge from a larger (teacher) model to a smaller (student) model by mimicking the teacher's outputs.

Mathematical Overview: The student model is trained to minimize a combined loss:

$$\mathcal{L} = \alpha \, \mathcal{L}_{\text{CE}}(y, \hat{y}) + (1 - \alpha) \, \mathcal{L}_{\text{KD}}(z^{(T)}, z^{(S)}),$$

where $\mathcal{L}_{\text{CE}}$ is the cross-entropy loss between the true labels y and the student's predictions $\hat{y}$, and $\mathcal{L}_{\text{KD}}$ is a distillation loss (often a Kullback-Leibler divergence) between the teacher's logits $z^{(T)}$ and the student's logits $z^{(S)}$. The parameter α balances the two losses.

Advantages:

- Allows a smaller model to achieve performance closer to a larger one.
- Often improves generalization.

Disadvantages:

- Requires a well-performing teacher model.
- The distillation process adds an extra training stage.

8.2 Hardware-Aware Optimization and Model Deployment

When delivering models as products, it is also crucial to optimize for the target hardware. This may involve:

- **Using Frameworks like TensorRT or ONNX:** These allow for further optimizations such as layer fusion and precision calibration.
- **Edge Optimization:** Techniques such as model partitioning, dynamic inference, and using specialized accelerators (e.g., NPUs) can greatly reduce latency.
- **Automated Machine Learning (AutoML):** Tools that optimize hyperparameters and network architectures to best fit deployment constraints.

8.3 Why Optimization is Important

- **Real-Time Performance:** Optimized models ensure low-latency inference, which is crucial in real-time applications like video tracking.
- **Resource Efficiency:** Reduced model size and computation mean lower power consumption and memory usage, enabling deployment on embedded systems.

- **Cost Reduction:** Efficient models reduce computational costs and allow scaling to more users or devices.
- **Robust Deployment:** Hardware-aware optimizations lead to models that perform reliably in production environments with varying conditions.

Optimizing models through compression, quantization, knowledge distillation, and hardware-aware deployment methods is vital for transitioning from research prototypes to production systems. Each method has its own trade-offs regarding accuracy, efficiency, and complexity, and the choice depends on the application requirements and the target deployment environment.