Project Phase 1

Ehsan Merrikhi 400101967
github repository

Deep Learning

Dr. Fatemizadeh

January 28, 2025

Deep Learning

Project Phase 1

Ehsan Merrikhi 400101967
github repository

Contents

Contents **1**

1. Overview of Algorithms **2**
In-depth Analysis of Algorithm Techniques 2
Performance Comparison 4

2. Application of Algorithms in Sports Videos **6**
Review and Analysis 6
Appropriate Tracking Approach 7

3. Metrics for Comparing Tracking Methods **8**
Explanation of Metrics 8
Analysis of Metric Limitations 9

4. Review of Datasets Specific to Tracking **10**
Tracking Datasets in Sports 10
Suggested Dataset 10

5. Challenges Related to Tracking **11**

6. Coding and Initial Implementation **12**

7. Conclusion and Recommendations **19**

1. Overview of Algorithms

In-depth Analysis of Algorithm Techniques

A comprehensive analysis of each of the four mentioned algorithms and the core techniques applied within them.

Soloution

SORT (Simple Online and Real-time Tracking)
SORT is a straightforward multi-object tracking approach that relies on:

- *Kalman Filters for Predictive Tracking*: Each detected object is represented by a separate Kalman Filter which predicts its future position (bounding box) in the next frame based on motion cues such as object velocity and location.
- *Hungarian Algorithm for Data Association*: After new detections are generated by the object detector at each frame, the algorithm associates these detections to existing tracked objects by maximizing the Intersection over Union (IoU). The Hungarian algorithm solves this assignment problem efficiently.
- *Simplicity and Speed*: SORT is lightweight and fast, making it suitable for real-time tracking applications. However, it struggles in crowded scenes or under significant occlusion since it relies predominantly on motion and IoU cues.

DeepSORT
DeepSORT extends the fundamental ideas of SORT but enhances it in terms of robustness to occlusions and appearance-based identity switches. It introduces:

- *Deep Appearance Embeddings*: Each detection is passed through a deep neural network (often a CNN) to extract an appearance descriptor. This descriptor helps to distinguish objects with similar motion but different visual cues.
- *Combined Motion and Appearance Cost*: The data association step combines the motion model (Kalman Filter predictions) with the appearance distance (e.g., cosine distance of embeddings). This increases robustness in crowded scenes.
- *Better ID Switching Performance*: By leveraging appearance features, DeepSORT significantly reduces identity switches compared to standard SORT, albeit at a higher computational cost due to the additional deep feature extraction.

Soloution

ByteTrack
ByteTrack is a more recent approach that aims to capture both high-confidence and low-confidence detections to improve track completeness:

- *Two-stage Association*: First, ByteTrack performs association using **high-confidence** detections with existing tracks. Next, it associates the remaining **low-confidence** detections to potentially recover missed associations from the first step.
- *Robust Tracking in Low Score Range*: By not discarding low-scoring detections prematurely, ByteTrack can maintain continuous tracks even when objects are partially occluded or the detector fails to produce high scores momentarily.
- *State-of-the-Art Performance*: Through careful thresholding and optimization, ByteTrack has demonstrated leading results on multiple MOT benchmarks while keeping computational overhead in check.

FairMOT
FairMOT unifies the detection and re-identification tasks into a single network:

- *Multi-task Network*: FairMOT adopts a dual-head architecture—one for object detection (bounding box regression) and one for re-identification features. This single-shot design improves efficiency by jointly learning detection and ReID.
- *Anchor-free Detection*: By using an anchor-free framework, FairMOT attempts to be more robust to varied object sizes and aspect ratios. This often translates into better generalization across different scenarios.
- *Balanced Detection and ReID Performance*: Unlike previous methods that sacrifice one task for the other, FairMOT focuses on maintaining high-quality detection and re-identification simultaneously, leading to strong tracking performance with fewer identity switches.

Performance Comparison

A detailed comparison of the algorithms focusing on accuracy, speed, and implementation complexity.

Soloution

Accuracy:

- *SORT*: While accurate in simple scenes, it suffers from more identity switches in crowded or long-occlusion scenarios due to the reliance on motion and IoU alone.
- *DeepSORT*: Improves accuracy considerably compared to SORT, especially in complex scenes, thanks to deep appearance features.
- *ByteTrack*: Achieves state-of-the-art accuracy on several benchmarks by not discarding low-confidence detections and conducting a refined two-stage association process.
- *FairMOT*: Also achieves high accuracy by jointly learning detection and ReID. It often shows a good balance of detection and ID tracking performance.

Speed:

- *SORT*: Extremely lightweight and fast. Suitable for real-time applications if the detection step is also efficient.
- *DeepSORT*: Slightly slower than SORT due to the additional forward pass for appearance embeddings.
- *ByteTrack*: Maintains relatively high speed, though it may require more careful thresholding and a two-stage association, adding some overhead.
- *FairMOT*: Speeds can vary depending on the backbone network. While it can reach real-time performance with certain backbones, the multi-task learning adds complexity.

Soloution

Implementation Complexity:

- *SORT*: Minimal complexity. Uses only a Kalman Filter and a standard assignment algorithm (Hungarian). Easy to implement and maintain.
- *DeepSORT*: More complex due to the requirement of a trained deep appearance model and the integration of motion + appearance cues.
- *ByteTrack*: Requires careful threshold tuning and a two-phase association mechanism but is relatively straightforward once thresholds are set.
- *FairMOT*: Involves training a single model for both detection and ReID. Can be more involved in terms of network design and parameter tuning but simplifies the runtime pipeline once trained.

Practical Considerations:

- If *real-time speed* and simplicity are critical, **SORT** remains a strong baseline.
- For *robust performance* in crowded scenarios with **reduced ID switches**, **DeepSORT** or **FairMOT** are excellent choices.
- **ByteTrack** stands out if you require *high accuracy* across a wide range of scenarios (including low-scoring detections) while retaining competitive speed.
- **FairMOT** is favored if a unified training pipeline for detection and ReID is desired, simplifying deployment once the model is trained.

2. Application of Algorithms in Sports Videos

Review and Analysis

Review of related articles and analysis of the application of algorithms in sports scenarios such as tracking players and balls.

Soloution

In sports video analysis, accurate and robust tracking can be challenging due to frequent player occlusions, fast movements, and sudden appearance changes (e.g., uniform numbers partially obscured or the ball moving at high speed).

SORT in Sports
SORT's simplicity and speed make it suitable for real-time tracking in less crowded scenes or scenarios with fewer occlusions. However, sports often involve complex interactions among multiple players, increasing the likelihood of ID switches. Thus, while SORT can provide a quick baseline, it may fail to maintain stable IDs for players during intense action.

DeepSORT in Sports
DeepSORT mitigates many of SORT's shortcomings by incorporating appearance embeddings. In sports, uniforms typically have distinctive colors or patterns, which a well-trained feature extractor can leverage to reduce identity switches. As a result, DeepSORT is widely used in scenarios like basketball or soccer, where players often get close to each other. The primary trade-off is higher computational cost due to deep feature extraction.

ByteTrack in Sports
ByteTrack's two-stage association strategy (high-confidence and then low-confidence detections) helps recover missed associations in highly dynamic sports settings—especially when the ball or players briefly have low detection confidence due to partial occlusion or fast movement. This is particularly beneficial in sports videos where the object detector's confidence can fluctuate depending on the camera angle and motion blur.

FairMOT in Sports
FairMOT unifies detection and re-identification within a single network, which can be advantageous in sports where training data (player bounding boxes and IDs) can be carefully curated. By doing detection and ReID simultaneously, FairMOT often shows strong performance in multi-player tracking, especially when consistent labeling is available (e.g., jersey numbers). The integrated approach can simplify the pipeline once the model is trained, though it may require more initial effort to collect and annotate training data.

Appropriate Tracking Approach

Determining which approach (tracking balls, players, and referees) is more suitable in each sports scenario, using Single Object Tracking (SOT) or Multiple Object Tracking (MOT).

Soloution

In many sports, both ball-tracking and player-tracking are crucial but present different challenges. Deciding on whether to use Single Object Tracking (SOT) or Multiple Object Tracking (MOT) often depends on the number of objects being tracked and the complexity of the scene.

Tracking the Ball (SOT)

- *SOT Applicability*: If the primary focus is tracking a single ball (e.g., a basketball, football, or soccer ball) through the game, SOT methods can be employed. This is particularly useful when there is a need to follow the ball's trajectory for advanced analytics without necessarily tracking every player on the court or field.
- *Use of MOT Algorithms for Ball Tracking*: While SORT, DeepSORT, ByteTrack, or FairMOT are typically used for MOT, their core mechanisms (predictive Kalman Filter + data association) can be adapted to track a single object.

Tracking Players and Referees (MOT)

- *MOT Applicability*: In most sports (e.g., basketball, soccer, hockey), multiple players and referees move around simultaneously. MOT algorithms like DeepSORT or FairMOT are well-suited for handling multiple, visually similar targets in the same frame.
- *Choice of Algorithm*:
  - **SORT**: Good for real-time tracking with limited computational resources, but can fail under heavy occlusion.
  - **DeepSORT**: Balances speed and accuracy with re-identification features, reducing ID switches in crowded sports scenes.
  - **ByteTrack**: Helps maintain continuous tracks even when detections have lower confidence, beneficial in fast-paced sports with partial occlusions.
  - **FairMOT**: Ideal if a unified approach is desired for both detection and re-ID, and if a robust training set (including consistent labeling) is available.

In summary, SOT is typically more suitable for consistently following a single ball, while MOT-based approaches (SORT, DeepSORT, ByteTrack, FairMOT) are preferred for tracking multiple players and referees in high-traffic areas.

3. Metrics for Comparing Tracking Methods

Explanation of Metrics

Explanation of metrics such as MOTA, IDF1, Recall, and other performance evaluation criteria.

Soloution

MOTA (Multiple Object Tracking Accuracy)

- *Definition*: MOTA combines three error types—false positives (FP), missed detections (FN), and identity switches (IDS)—into a single score.
- *Formula*:

$$\mathrm{MOTA} = 1 - \frac{\sum(\mathrm{FN} + \mathrm{FP} + \mathrm{IDS})}{\sum \mathrm{GT}}$$

  where $\sum \mathrm{GT}$ is the total number of ground truth detections.
- *Interpretation*: Higher MOTA values indicate better overall tracking performance with fewer missed detections, fewer false positives, and fewer identity switches.

IDF1 (ID F1 Score)

- *Definition*: IDF1 measures the ratio of correctly identified detections over the average number of ground-truth and computed detections. It focuses on identity consistency.
- *Formula*:

$$\mathrm{IDF1} = \frac{2 \times \mathrm{IDTP}}{\mathrm{IDTP} + \mathrm{IDFP} + \mathrm{IDFN}}$$

  where IDTP (true positives), IDFP (false positives), and IDFN (false negatives) are computed in an identity-aware manner.
- *Interpretation*: A high IDF1 score indicates that an algorithm consistently maintains correct identities throughout the sequence.

Recall

- *Definition*: Recall reflects how many of the actual ground truth objects are successfully detected and tracked.

$$\mathrm{Recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

- *Interpretation*: High recall means fewer missed objects (FN). However, it does not directly consider identity consistency.

Other Performance Criteria

- *Precision*: Measures how many of the tracked objects are correct. $\frac{\mathrm{TP}}{\mathrm{TP+FP}}$
- *FPS*: Frames per second (FPS) indicates the runtime efficiency of the tracker—essential for real-time applications.

Analysis of Metric Limitations

Investigation of the limitations of each metric and analysis of how each metric addresses the shortcomings of others.

Soloution

Although each metric provides valuable insights, they each have inherent limitations:
MOTA Limitations

- *Overemphasis on Detection Errors*: MOTA aggregates false positives and false negatives into a single term, which can overshadow the impact of ID switches (IDS).
- *Identity Consistency Not Fully Captured*: A high MOTA does not necessarily mean the identities are well-preserved, as it pools errors rather than isolating ID-related mistakes.

IDF1 Limitations

- *Focus on Identity Consistency*: IDF1 heavily emphasizes how well a tracker retains correct identities over time, but it may not penalize false or missed detections as strongly as MOTA.
- *Complexity in Computation*: Calculating IDF1 requires a consistent matching of predicted tracks to ground truth identities over the sequence, which can be more involved than computing simpler detection metrics.

Recall Limitations

- *Ignores False Positives*: A high recall can be achieved at the expense of precision, i.e., detecting many non-existent objects without penalty to the recall metric itself.
- *No ID Tracking Insight*: Recall reflects detection ability more than tracking stability. It does not account for identity switches or re-identification performance.

How Metrics Address Each Other's Shortcomings

- *MOTA and IDF1 Together*: Combining MOTA with IDF1 helps paint a fuller picture. MOTA provides an overview of detection and tracking performance, while IDF1 highlights identity consistency specifically.
- *Precision and Recall*: Used together, they clarify the trade-off between detecting more objects (recall) and introducing fewer false positives (precision).
- *Additional Metrics (MT, ML)*: Provide insight into how consistently individual targets are tracked over their entire lifespans, complementing aggregate metrics like MOTA and IDF1.

In summary, no single metric can comprehensively evaluate all aspects of multi-object tracking performance. The best practice is to use a combination of metrics to gain a complete understanding of detection accuracy, identity consistency, and overall track stability.

4. Review of Datasets Specific to Tracking

Tracking Datasets in Sports

Examination of tracking datasets related to sports scenarios.

Soloution

1. **SoccerNet**
   - *Scope*: Focuses on soccer games, providing video data, annotations for events (e.g., goals, fouls), and some player bounding boxes.
   - *Usage*: Researchers use it for tasks like activity recognition, event detection, and multi-player tracking in a real-world soccer environment.
   - *Challenges*: Large field of view and frequent player occlusions
2. **NBA Dataset**
   - *Scope*: Broadcast basketball footage with annotations for players, ball positions, and sometimes additional event markers (e.g., rebounds, shots).
   - *Usage*: Often utilized for developing multi-player tracking algorithms that cope with close-contact scenarios and complex motion in confined courts.
   - *Challenges*: Fast player movements and near-constant occlusions demand strong re-identification strategies.
3. **Hockey and Other Sports**
   - *Scope*: Various hockey-specific datasets exist, focusing on challenging conditions such as quick puck movement and visually similar uniforms.
   - *Usage*: Tracking players, the puck, and officiating staff in a fast-paced environment.
   - *Challenges*: Low contrast between uniforms and rink ice, high-speed puck trajectories, and frequent partial visibility.

Suggested Dataset

Suggested link for the SportsMOT dataset: https://github.com/MCG-NJU/SportsMOT.

Soloution

SportsMOT

- *Scope*: A new dataset that covers multiple sports scenarios, aiming to provide a comprehensive resource for multi-object tracking in diverse conditions.
- *Usage*: Intended for benchmarking and developing advanced MOT algorithms that handle varied sports contexts with complex motion and crowded scenes.
- *Challenges*: Each sport poses unique difficulties—ranging from rapid ball trajectories to visually similar player jerseys and unpredictable motion patterns.

5. Challenges Related to Tracking

Explanation of challenges such as occlusion, scale variation, and illumination change, and analysis of how each affects the output metrics.

Soloution

- **Occlusion:**
  - *Description:* Objects are partially or fully hidden by other objects or scene elements (e.g., players in crowded sports environments).
  - *Impact on Tracking:*
    - Missed detections (FN) and identity switches (IDS) increase.
  - *Effect on Metrics:*
    - **MOTA:** Decreases due to added errors.
    - **IDF1:** Drops when identities are inconsistently maintained.
- **Scale Variation:**
  - *Description:* Objects appear at different sizes due to camera zoom or perspective (e.g., players moving between foreground and background).
  - *Impact on Tracking:*
    - Detection inconsistencies for small or far-away objects.
    - Inaccurate bounding boxes during rapid size changes.
  - *Effect on Metrics:*
    - **Recall:** Decreases if smaller objects are missed.
    - **Precision:** Falls with poorly scaled bounding boxes.
- **Illumination Change:**
  - *Description:* Lighting variations such as shadows, glares, or transitions (e.g., during evening outdoor sports).
  - *Impact on Tracking:*
    - Appearance drift leads to ID mismatches.
    - Detection failures increase due to unreliable visual features.
  - *Effect on Metrics:*
    - **MOTA:** Drops due to more FP and FN errors.
    - **IDF1:** Decreases with less reliable re-identification.

6. Coding and Initial Implementation

Implementation of a simple data loader for a dataset like SportsMOT.

Soloution

```python
import os
import torch
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms
from PIL import Image
import numpy as np
import configparser
import matplotlib.pyplot as plt
import random

class SportsMOTDataset(Dataset):
    def __init__(self, data_root, splits_txt_dir, split, transform=None):
        """
        Args:
            data_root (str): Path to the dataset root directory.
            splits_txt_dir (str): Path to the splits_txt directory.
            split (str): One of 'train', 'val', or 'test'.
            transform (callable, optional): Optional transform to be applied on a sample.
        """
        self.data_root = os.path.join(data_root, split)
        self.splits_txt_dir = splits_txt_dir
        self.split = split
        self.transform = transform

        # Get list of video names for the split
        self.video_names = self.parse_split_file(split)

        # Collect all image paths and corresponding annotation paths (if available)
        self.image_paths = []
        self.annotation_paths = []

        for video in self.video_names:
            video_path = os.path.join(self.data_root, video)
            img_dir = os.path.join(video_path, "img1")
            gt_dir = os.path.join(video_path, "gt")
            gt_file = os.path.join(gt_dir, "gt.txt")

            if not os.path.exists(img_dir):
                print(f"Image directory {img_dir} does not exist. Skipping video {video}.")
                continue

            images = sorted([img for img in os.listdir(img_dir) if img.endswith(('.jpg', '.png'))])
            self.image_paths.extend([os.path.join(img_dir, img) for img in images])

```

Soloution

```
            if self.split in ['train', 'val']:
                if os.path.exists(gt_file):
                    self.annotation_paths.extend([gt_file] * len(images))
                else:
                    print(f"Annotation file {gt_file} does not exist for
    video {video}.")
                    self.annotation_paths.extend([None] * len(images))
            else:
                # For test split, annotations may not be available
                self.annotation_paths.extend([None] * len(images))

        print(f"{split.capitalize()} split: {len(self.image_paths)}
    images found.")

    def parse_split_file(self, split_name):
        """
        Parses the split text file to get the list of video names for the
     given split.

        Args:
            split_name (str): One of 'train', 'val', or 'test'.

        Returns:
            List[str]: List of video names for the split.
        """
        split_file = os.path.join(self.splits_txt_dir, f"{split_name}.txt
    ")
        if not os.path.exists(split_file):
            raise FileNotFoundError(f"Split file {split_file} does not
    exist.")

        with open(split_file, 'r') as f:
            video_names = [line.strip() for line in f.readlines() if line
    .strip()]
        return video_names

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        # Load image
        img_path = self.image_paths[idx]
        try:
            image = Image.open(img_path).convert("RGB")
        except Exception as e:
            print(f"Error loading image {img_path}: {e}")
            # Return a black image and empty annotations
            image = Image.new("RGB", (1, 1), (0, 0, 0))
            annotations = torch.empty((0, 9), dtype=torch.float32)
            return image, annotations


```

Soloution

```
        # Load annotations if available
        ann_path = self.annotation_paths[idx]
        if ann_path:
            annotations = self._load_annotations(ann_path, idx)
        else:
            annotations = np.array([], dtype=np.float32).reshape(0, 9)  # Empty annotations for test
        # Apply transformations
        if self.transform:
            image = self.transform(image)

        # Convert annotations to tensor if not empty
        if len(annotations) > 0:
            annotations = torch.tensor(annotations, dtype=torch.float32)
        else:
            annotations = torch.empty((0, 9), dtype=torch.float32)

        return image, annotations

    def _load_annotations(self, ann_path, idx):
        """
        Load annotations for the given image index.

        Args:
            ann_path (str): Path to the annotation file.
            idx (int): Index of the image in the dataset.

        Returns:
            np.ndarray: Array of annotations for the image.
        """
        annotations = self.parse_gt_file(ann_path)

        # Extract frame number from image path
        # Assuming image filenames are zero-padded frame numbers, e.g., 000001.jpg
        frame_number = int(os.path.splitext(os.path.basename(self.image_paths[idx]))[0])

        # Filter annotations for the current frame
        frame_annotations = annotations[annotations[:, 0] == frame_number]

        return frame_annotations

```

Soloution

```
    @staticmethod
    def parse_gt_file(gt_file_path):
        """
        Parses the ground truth annotation file.

        Args:
            gt_file_path (str): Path to the gt.txt file.

        Returns:
            np.ndarray: Array of annotations with shape (N, 9).
        """
        if not os.path.exists(gt_file_path):
            raise FileNotFoundError(f"Annotation file {gt_file_path} does not exist.")

        # Assuming the gt.txt has at least 6 columns: frame, id, x, y, w, h
        # Adjust dtype and delimiter as per your gt.txt format
        annotations = np.loadtxt(gt_file_path, delimiter=",", dtype=np.float32)
        if annotations.ndim == 1:
            annotations = annotations[np.newaxis, :]  # Handle single annotation case
        return annotations

# Visualization Function
def visualize_sample(image, annotations):
    """
    Visualizes an image and its corresponding annotations (bounding boxes).

    Args:
        image (torch.Tensor): Image tensor of shape (3, H, W).
        annotations (torch.Tensor): Tensor of annotations with shape (N, 9).
    """
    # Convert image tensor to numpy array and change shape to (H, W, 3)
    image = image.permute(1, 2, 0).numpy()
    image = (image * np.array([0.229, 0.224, 0.225]) +
             np.array([0.485, 0.456, 0.406]))  # Reverse normalization
    image = np.clip(image, 0, 1)  # Clip values to [0, 1] for visualization

    # Get original image size
    height, width, _ = image.shape

    # Plot the image
    plt.figure(figsize=(width / 100, height / 100))  # Adjust figure size based on image size
    plt.imshow(image)
    plt.axis("off")

```

Soloution

```
    # Draw bounding boxes from annotations
    for ann in annotations:
        if ann.numel() == 0:
            continue  # Skip if no annotations
        bbox = ann[2:6].numpy()  # Extract bounding box (x, y, w, h)
        x, y, w, h = bbox
        plt.gca().add_patch(plt.Rectangle(
            (x, y), w, h, fill=False, edgecolor="red", linewidth=2
        ))
    plt.show()

# Testing Function
def test_dataloaders(train_loader, val_loader, test_loader, num_samples=2):
    """
    Tests and visualizes samples from the dataloaders.

    Args:
        train_loader (DataLoader): Training dataloader.
        val_loader (DataLoader): Validation dataloader.
        test_loader (DataLoader): Test dataloader.
        num_samples (int): Number of samples to visualize from each loader.
    """
    def visualize_from_loader(loader, split_name, num_samples):
        print(f"\nVisualizing samples from the {split_name} loader...")
        for i, (images, annotations) in enumerate(loader):
            if i >= num_samples:
                break
            for j in range(2):
                print(f"\nSample {j + 1} in Batch {i + 1}:")
                print(f"  Image shape: {images[j].shape}")  # (3, H, W)
                print(f"  Annotations shape: {annotations[j].shape}")  # (N, 9)
                visualize_sample(images[j], annotations[j])

    visualize_from_loader(train_loader, "train", num_samples)
    visualize_from_loader(val_loader, "validation", num_samples)
    visualize_from_loader(test_loader, "test", num_samples)


def custom_collate_fn(batch):
    """
    Custom collate function to handle batches with variable number of annotations
    and potentially variable image sizes.

    Args:
        batch (list): List of tuples (image, annotations).

    Returns:
        images (list of torch.Tensor): List of image tensors.
        annotations (list of torch.Tensor): List of annotation tensors.
    """
    images, annotations = zip(*batch)
    return list(images), list(annotations)
```

Soloution

```python
# DataLoader Function
def get_dataloaders(data_root, splits_txt_dir, batch_size=32, num_workers=4):
    """
    Prepares train, validation, and test dataloaders with a custom collate function.

    Args:
        data_root (str): Path to the dataset root directory.
        splits_txt_dir (str): Path to the splits_txt directory.
        batch_size (int): Batch size for dataloaders.
        num_workers (int): Number of workers for dataloaders.

    Returns:
        train_loader, val_loader, test_loader: DataLoader instances.
    """
    # Define transformations without resizing
    transform = transforms.Compose([
        transforms.ToTensor(),  # Convert PIL Image to Tensor
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std=[0.229, 0.224, 0.225])  # Normalize
    ])

    # Create datasets
    train_dataset = SportsMOTDataset(
        data_root=data_root,
        splits_txt_dir=splits_txt_dir,
        split="train",
        transform=transform
    )
    val_dataset = SportsMOTDataset(
        data_root=data_root,
        splits_txt_dir=splits_txt_dir,
        split="val",
        transform=transform
    )
    test_dataset = SportsMOTDataset(
        data_root=data_root,
        splits_txt_dir=splits_txt_dir,
        split="test",
        transform=transform
    )
```

Soloution

```python
    # Create dataloaders with the custom collate function
    train_loader = DataLoader(
        train_dataset,
        batch_size=batch_size,
        shuffle=False,
        num_workers=num_workers,
        collate_fn=custom_collate_fn
    )
    val_loader = DataLoader(
        val_dataset,
        batch_size=batch_size,
        shuffle=False,
        num_workers=num_workers,
        collate_fn=custom_collate_fn
    )
    test_loader = DataLoader(
        test_dataset,
        batch_size=batch_size,
        shuffle=False,
        num_workers=num_workers,
        collate_fn=custom_collate_fn
    )
    return train_loader, val_loader, test_loader
# Define the root directory of your dataset
ROOT = "/kaggle/input/sportsmot/sportsmot_publish"  # Replace with your
    actual path
# Define the directory containing split text files
SPLITS_TXT_DIR = os.path.join(ROOT, "splits_txt")
# Create dataloaders
train_loader, val_loader, test_loader = get_dataloaders(
    data_root=os.path.join(ROOT, "dataset"),
    splits_txt_dir=SPLITS_TXT_DIR,
    batch_size=32,
    num_workers=4
)
# Test and visualize samples from the dataloaders
test_dataloaders(train_loader, val_loader, test_loader, num_samples=2)

```

output:

```
Train split: 28574 images found.
Val split: 26970 images found.
Test split: 94835 images found.

Visualizing samples from the train loader...

Sample 1 in Batch 1:
Image shape: torch.Size([3, 720, 1280])
Annotations shape: torch.Size([6, 9])
```

(rest of the output is available in the jupyter notebook)

7. Conclusion and Recommendations

Summarize the challenges identified in the reviewed algorithms and provide suggestions for improving performance.

Soloution

Key challenges and potential solutions:

- **Robust Re-identification and Occlusion Handling**:
  - *Challenge*: SORT struggles with identity switches during occlusions, and while DeepSORT and FairMOT improve re-identification, heavy occlusion scenarios remain problematic.
  - *Recommendation*: Incorporate advanced appearance embeddings or graph-based matching to handle prolonged occlusions. Using larger and more diverse training data can help models learn robust features.
- **Maintaining Consistency Over Low-confidence Detections**:
  - *Challenge*: Traditional trackers often discard low-confidence detections, leading to track fragmentation. ByteTrack addresses this partially with a two-stage association strategy.
  - *Recommendation*: Fine-tune threshold strategies and employ multi-stage associations. Additionally, leveraging temporal information (e.g., optical flow or trajectory smoothing) can help maintain consistent tracks.
- **Balancing Speed and Accuracy**:
  - *Challenge*: High-accuracy approaches (DeepSORT, FairMOT) can be computationally expensive, whereas faster methods (SORT) have higher ID-switch rates in crowded scenes.
  - *Recommendation*: Optimize neural networks for efficiency (e.g., lightweight backbones for DeepSORT/FairMOT). Implement model pruning or quantization techniques to achieve near real-time performance without significant accuracy loss.
- **Domain-specific Adaptations and Model Generalization**:
  - *Challenge*: Sport-specific scenarios, varying lighting conditions, and unique occlusion patterns require specialized training and adaptation to maintain robust performance.
  - *Recommendation*: Apply domain adaptation or fine-tuning on sport-specific datasets. Augment training data with scenarios reflecting real-world conditions (e.g., different camera angles, lighting, and crowd densities).

Overall, the recommendations highlight the need for improved re-identification under challenging conditions, refined handling of low-confidence detections, and a thoughtful balance between speed and accuracy.