UNIVERSITY OF BIRMINGHAM

SCHOOL OF COMPUTER SCIENCE

Spatio-temporal Understanding of Agents in Multiple Object Tracking

MSC DISSERTATION
MASTER OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

By Shoaib Ahmed

ID: 2336882

Supervisor: Dr. Leandro Minku

Table of Contents

Abstract

Multiple Object Tracking is from one of the most popular commercial applications of modern Deep Learning based Computer Vision algorithms. It is heavily used in Security, Crowd Management, Smart Cities and Autonomous driving. Since most of the applications of Multiple Object Tracking (MOT) require real-time results, speed is a crucial factor in determining the usability of MOT algorithms. With this in mind, we propose a new Tracking algorithm that uses global spatio-temporal movement patterns of people in a fixed-frame setting that aims to reduce inference time.

MOT is usually comprised of two sub-problems: Detection and Re-Identification. The detector is responsible for detecting people in each frame of a video. The Re-Identification refers to matching each of these detections to their corresponding tracks from previous frames. The proposed algorithm works by reducing the search space of the detections to be matched with tracks. We implement this Algorithm in Python along with a baseline algorithm to compare the two algorithms on several factors such as CPU usage, Memory usage, Throughput and Performance Accuracy. We find that the proposed algorithm roughly uses the same amount of CPU and RAM, and is qualitatively similar to the baseline algorithm, but the throughput of the proposed algorithm is found to be better than the baseline algorithm.

Keywords: Artificial Intelligence, Computer Vision, Multi Object Tracking, Detection, Tracking, Python, Yolo, Graphs

Acknowledgements

I would like to thank my Supervisor Dr. Leandro L Minku for the continuous guidance, advice, and suggestions throughout the Project and all the lecturers and colleagues who have taught me the concepts of Computer Science, Software Development, Artificial Intelligence & Machine Learning.

Introduction

Tracking object movements in a scene is a popular problem in Computer Vision and the recent boom of Deep Learning techniques has made the Multiple Object Tracking a useful commercial application with improvements being made continuously. We find applications of Multiple Object Tracking (MOT) in various fields especially Security, Crowd Management, Autonomous Driving, and Human Computer Interface. The applications of Security and Crowd management usually work on human tracking from a fixed frame i.e., the camera being used to capture video is fixed. CCTV feeds are an example of fixed frame inputs. Autonomous driving and Human Computer Interaction are examples of applications where the camera is moving. They are also much more likely to use 3D inputs along with Video streams or some sort of a fusion input like RGBD cameras. Due to this, the applications using moving frame inputs require more complicated tracking algorithms as compared to fixed frame applications.

Keeping in mind the duration of the project, we focus on working with the first category of input type: Fixed frame. These applications are used in highly scaled deployment scenarios like Indoor Crowd management in high security environments like Airports as well as Outdoor people management like Smart Cities. These applications usually have hundreds, if not thousands, of cameras running 24 hours a day, every day. They usually run on Server-grade GPU-enabled workstations on-site in a private network. This is due to 2 reasons: First is Privacy – People do not want to send their private CCTV feeds to the Internet and second is Cost- sending hundreds or thousands of Full-HD video streams 24/7 to the cloud for processing is simply not economic. Since offline computing cannot be scaled as readily as the cloud, application developers need to be very efficient and precise with the Throughput computations of their software to be able to quote the quantity and specifications of Hardware required to run analytics on a given number of Camera feeds. Thus, little improvements in the throughput, usually represented in Frames per Second (FPS), mean that the same machine can process a few extra cameras than it usually would or incorporate more advanced or additional processing on the same number camera feeds. Thus, we aim to improve the throughput of Multiple Object Tracking algorithms in a fixed frame setup.

The solution for the problem of Multiple Object tracking (MOT) is usually split into a two-stage solution: Detection and Reidentification. Deep Learning methods traditionally use two different Models to solve each of these problems independently, but there has been a recent rise in single-stage solutions [6,7] to this problem where detection and Reidentification are done by the same network. Two-stage solutions still make up most of the State-of-the-art methods [5, 11,12]. Commercial establishments still prefer to use Two-stage trackers because of the modularity they offer. This allows us to train and re-train both the nets separately according to their individual performance, allows us to run these networks in different numbers, and allows us to run the two stages on different devices altogether.

The two stages in a two-stage model are: **Detection** and **ReIdentification**. We first detect objects in each frame using a DL-based Detection Model. We then try and match each of these detections to a track that represents a unique object from previous frames or create a new track if the detection does not match any existing track. This matching or Reidentification is mainly done today using a DL network that generates Feature Vectors from Detection crops. We then compare feature vectors from detections and from tracks to assign a similarity score. We use Yolov5[13] as our detection model and OSNet[3] as a Feature Extractor in our experiments. We explore potential areas of improvement in this part of the Tracking pipeline and propose a novel ReIdentification algorithm – GTracker that addresses these issues and gives better throughput as compared to the baseline algorithm. GTracker works by learning the Global spatio-temporal patterns of a given camera setup and uses this information to restrict the number of potential matching tracks thereby increasing the throughput.

The rest of the report is divided into six sections. We first talk about the Related work that is being done in the field of Multiple Object Tracking, an in-depth look at the detection and encoder models we are using to solve this problem, and finally revisit some graph theory concepts that are used to build our proposed GTracker. We then look at the kind of data available at our disposal to solve this problem. We then discuss the Methodology used to implement GTracker and take a closer look at the finer implementation details. We then talk about the Experimental setup and then metrics used to compare the baseline with the proposed GTracker. We also talk about the popular metrics used to measure the performance of MOT algorithms. We then present the results of the experiments we have performed in terms of the metrics we had previously defined. Finally, we discuss these results and talk about the drawbacks of this approach along with possible ways to improve the proposed algorithm.

Related Work

Extensive research has been done on Multiple Object Tracking. The typical solution to the problem consists of detecting objects and then re-identifying them from previous instances. This is being done in two ways: Single stage and Two stage methods. We choose to solve MOT using the two-stage or two-stage method where Detection and ReID (Re Identification) are two separate components. Throughput of an algorithm in video analytics refers to the rate of successful processing of frames per second. This is usually denoted by a term called FPS or Frames Per Second. Video Analytics works reasonably well at 12 FPS and thus 12 FPS is usually referred to as real-time in this setting.

Single-stage Trackers

There has been a recent rise in single-stage solutions being proposed for MOT. These methods solve the problem of Detection and ReID using a single network that is jointly optimized. They have been shown to increase computational efficiency but do not perform as well as the other trackers qualitatively. Some work [6] has been done to try and improve this joint learning task but the performance of single-stage methods is still lower than most state-of-the-art two-stage methods. FairMOT [6] paper lists a few reasons for this drop in performance of single-stage trackers and one of the main reasons is that the sub-tasks of detector and ReID usually learn the same features while the two tasks require different kinds of features to be learnt and hence end up competing.

Two Stage Trackers

Two-stage Object trackers use two separate models to perform the tasks of Detection and ReIdentification. This allows them to be highly specialized at their individual tasks without having compete for feature representation. It also allows users to be able to independently fine-tune each of their performance as required. The drawback of two-stage trackers is that it requires more resources to deploy two DL models as compared to single-stage trackers that only use one model. However, the accuracy and modularity offered by two-stage trackers make them favourable when one is looking to implement a tracker in a commercial setup. This kind of modularity allows for flexible deployment use-cases such as running the detector on Edge devices and then sending just the detected crops to the cloud to be able to perform track matching, re-identification, and any application dependant post-processing thereby decreasing bandwidth consumption while taking advantage of the flexibility offered by the Cloud as well as the portability offered by edge and on-camera processing. We now talk about the two stages in detail.

Detection

Object Detection is perhaps the most extensively researched topic in Computer Vision and Deep Learning has delivered some highly performant Object Detectors [16]. This task consists of two sub-tasks: Localization and Classification. Localization refers to the problem of finding the pixel coordinates of the bounding boxes around objects of interest and classification refers to the problem of finding out what class the object is an instance of. Detectors are mainly of two types: Single shot and two shot. Single Shot detectors perform localization and classification in a single shot I.e., using the same DL network while Two-shot detectors run two passes on the image to accomplish the same. Two-shot detectors perform qualitatively better than single-shot detectors while single-shot detectors have the advantage of speed, which is a crucial factor when building real-time applications.

We use a single-shot detector, Yolov5 [13], because of the good balance of speed and accuracy it offers. The Yolo [4] family of detectors work by splitting the input image into a grid and each cell in this grid is responsible for predicting the location and type of the object whose centre lies in the cell. It then performs Non-maximal Suppression (NMS) [4] to merge duplicate detections and present a final list of objects present in the Scene. The model we use is pre-trained on the COCO dataset [17] which is the most popular large-scale object detection, segmentation, and captioning dataset.

ReIdentification

For ReIdentification we use a DL model called OSNet [3] to extract feature vectors from detection crops of people. This feature vector represents this person in a latent space and the idea is to have a latent space where the more similar the detection crops look alike, the closer the feature vectors will be in this space. To find the distance between two feature vectors, we use the measure of Cosine similarity, S_c. The OSNet is pre-trained on a person re-identification dataset called Market 1501 [2] using Contrastive Learning [18]. Other related works [8,19] include using Kalman Filters to predict the future pixel co-ordinates of an object but we choose not to use the Kalman filter or any version of it to strictly understand the performance of the proposed method using global spatio-temporal patterns instead of local track-based estimates and patterns.

$$S_c(A,B) = \cos(\theta) = \frac{A.B}{\|A\|\|B\|}$$

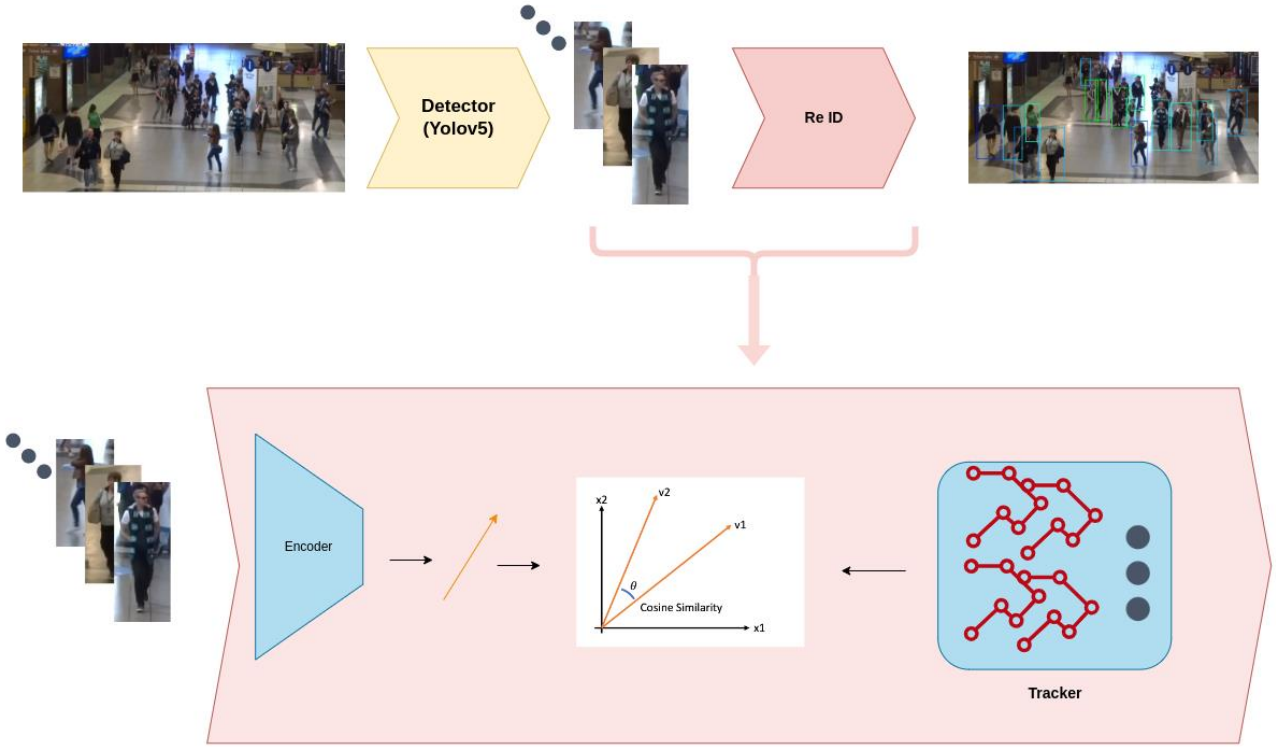

Figure 1: An overview of 2 stage Multi-Object Tracking Algorithms

Graph Basics

Nodes and Edges make up Graph structures and we use graphs to represent the pairwise relation between the abstract objects its nodes represent. These Edges can have different properties such as weight and directionality. We use both these properties to implement GTracker. A Directional graph is a graph whose edges have orientations I.e., they point from one node to another. The in-degree of a node is the number of edges directed towards the node. The out-degree is the number of edges pointing out of the node, towards other nodes. Weights can be used to denote abstract quantitative properties of Edges and I.e., the more weight an edge has, the more important that edge is. Neighbourhood of a node N represents the sub-graph of the node where each node of the subgraph is connected to the node N.

The IoU is an evaluation metric used to find how much a predicted bounding box and the corresponding ground truth bounding box overlap. It is defined as

$$IoU = \frac{Area\ of\ Overlap}{Area\ of\ Union}$$

Data

Since the problem of MOT is popular in Computer Vision, we have a good amount of freely available datasets that are challenging as well as realistic. One of the most used datasets is the MOT Challenge [9,10]. The MOT challenge has a variety of datasets for Multi Object Tracking including some 3D data. Since our problem is confined to person-based fixed frame Tracking, we manually pick out all matching videos from MOT17 [20], MOT20 [9], MOT15 [10], and MOT16 [21]. Since we also need to measure the performance accuracy of our algorithm on these videos and compare it with the baseline, we can only use the videos provided in the training set. We cannot use the test set since it does not have the ground truth annotations available.

MOT generally provides short duration videos in the form of individual frames as input images and a ground truth text file that has, for each frame in the video, a list of all detections, their corresponding class, their bounding box coordinates, their unique track ID, and their confidence/occlusion score. We use these annotations to calculate the performance metrics of our algorithms.

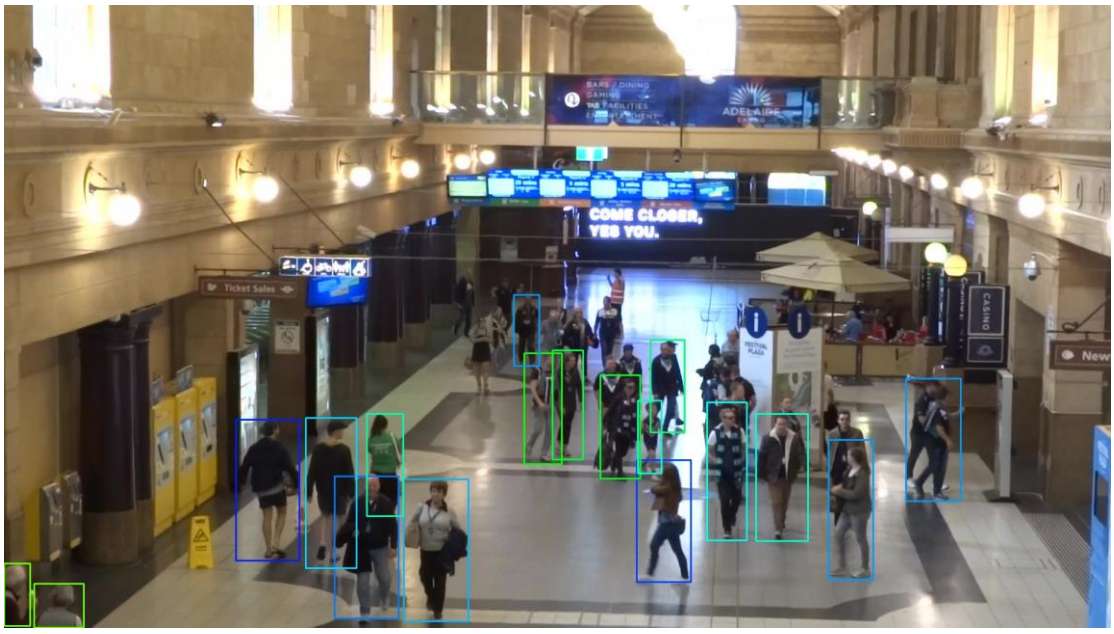

Figure 2: A sample frame from MOT20 Challenge with visual Ground Truth annotations

Method

This section lists the details of the implementation of both the baseline as well as the proposed algorithm. The proposed method differs to the baseline in just specific sub-section of Re-Identification - Tracking. The application is written in Python. The MOT application we develop is a multi-processing inference pipeline since we want to be able to run the DL models for Detection and Feature Vector extraction in parallel. This means that when the ReID process is processing a frame, the detection processing should be processing the next frame and not sit idly. Thus, the main application spawns three crucial child processes namely: det, reid , and eval for detection, re-identification, and evaluation, respectively. The processes communicate with each other using Multi-processing Queues. We also use a process called monitor that is a wrapper around the three child processes and is responsible for continuously and periodically collecting the hardware Resource consumption of each of the 3 spawned processes and displaying the results for comparison. We use PyTorch as the Deep Learning framework. The processes in the Infer pipeline are:

Main and monitor

The main process is the start point of the application. This is the parent process of all sub-processes spawned during execution. It handles all the user-definable parameters like the location of input data folder and location of the Yolov5 [13] and OSNet [22] models to be used for Detection and ReID. Its job is to send the location of each video present in the input data folder to the detection process for processing and collect the performance metrics received from the evaluation process for the same video. It spawns just one child process named monitor. Monitor is a wrapper process whose main function is to start the actual processes in the inference pipeline: det, reid, and eval. It starts these child processes and then goes into an infinite loop that periodically collects RAM and CPU usage of each of these child processes until all videos are done processing and the child processes terminate. It uses the psutils library in Python to get resource usage data for each process. At the end of processing, it saves all this data in the form of a graph to a .png file using the matplotlib library in Python.

Detection

The Detection Process is responsible for loading and running the Detection model- Yolov5 and passing on these detections from each frame to the next process in the pipeline- ReID. It first loads the Yolov5 model on the GPU (if available), and then customizes the model parameters such as the Confidence thresholds, IoU thresholds, and classes to consider. We talk more about IoU in the next section and the class to consider is 'Person' as we are only working on People Tracking. Once the model is loaded, we are now ready to start processing videos. We thus start an infinite loop that listens to the queue from the main process for new video paths. Once it receives a video path, it goes to the specified directory and processes all frames of the given video one-by-one. It first reads the frames using a Python library named OpenCV and runs the detection model on the frame. The model outputs the bounding box locations of each of the persons detected in the given frame, along with their confidence values. We collect the locations, confidence scores, and the actual crop of each of these objects along with the frame number in an array (list in Python) and pass this list to the next process in the pipeline: ReID using another queue and start processing the next frame. Once all frames for this video are done, we ask the main process to send the next video and start processing that in the same manner.

ReID

The ReID process has two main tasks: Feature Vector Extraction and Track matching. Feature Vector Extraction is done by the encoder network OSNet [3, 22]. For each of the detections in the frame, OSNet takes the cropped images of the detected persons as input and outputs a corresponding 512-dimensional Feature Vector. The goal is to place similar looking objects as close as possible in this 512-dimensional latent space. Once we have the feature vectors of all the detections of a frame, we send them to the Tracker along with other metadata received from the detector such as location and confidence. The Tracker is where we propose changes to the baseline to get faster track matching performance. The Tracker is implemented as an OOP Class (Object Oriented Programming). We instantiate a new Tracker object for each new video. The baseline tracker is a class named 'Tracker' and the proposed tracker is class named GTracker. We now go into the details of the baseline Tracker as well as the proposed GTracker.

Baseline Tracker

The tracker class is responsible for maintaining a list of recent tracks. It also implements class functions for finding the best matching track for a given detection,

adding a detection to the track, creating a new track, and deleting old tracks. A Track contains a history of all associated detections of a particular object, their location history, last seen location, last frame ID, and an average Feature Vector representative of the appearance of the object. We use this average feature vector when comparing new detections with existing tracks. For each frame matching, we build a distance matrix between all detections and all existing tracks using the Cosine distance metric. We then perform a linear sum assignment [23] where each detection is assigned a corresponding track. We go through these assignments and make sure that the matching is above threshold. If not, we create a new track for the detection. We mark the tracks that do not have any assignments for this frame using a counter and once the counter reaches a certain number of successive frames with the track unassigned, we delete the track assuming the object has left the frame. Once the assignment is done, we send the newly assigned Track IDs to the eval process for accumulation and final accuracy computations.

GTracker

In this method, we use graphs to represent and encode global spatio-temporal patterns of a video. The frame is initially divided into 4 equal regions and each region is represented by a node in the graph. The directional edges between these nodes represent how likely an object is to transition from one region to another or stay in the same region. We call this the **Transition Graph** and this is the data structure that essentially captures the global spatio-temporal patterns of the video. We initially assign all edges, including the self-loop, a weight of 0.25 as each node has three neighbours and we do not have any information of how objects move between them to begin with. We also use a duplicate graph named Running Graph that has the same nodes as the Transition graph but does not have any edges at the beginning. This graph helps keep a running history of object movements.

The matching is done using this graph as follows:

1. To find the matching track of a detection, we first find the node/region name in which the detection lies. We do this using a special tree data structure whose only job is to find the region names from pixel coordinates.
2. We then query the graph for all its neighbours (including itself because of the self-loop) and get a list of all the tracks last seen in this list of neighbours.
3. We then do the feature vector comparison between this detection and all the tracks from the neighbourhood.
4. If there is a match above a threshold, we assign the detection to the best matching track. If none of the tracks match, we make a new track.
5. When a detection is assigned a track, we increment the edge weight in the running graph by one. This edge weight is for the edge from the last region of the track to the new region of this detection (even if it is a self-loop).

To accommodate the novel changes in the tracker, we also introduce some new data structures in the track histories as well as in the GTracker class. To increase the accuracy of the method, we can take the graph neighbours of the current node up to n levels instead of just 1. Where each level denotes the distance of the node from the current node I.e., n=2 denotes neighbours of neighbours. The lower n is, the smaller the search space and thus faster the track matching becomes. Feature vector comparison is done using the same metric as the baseline I.e., Cosine Similarity. We use the Python library networkx to build the graphs.

After every couple hundred frames, once we have enough knowledge of the track transitions from the running graph, we rebuild the Transition Graph. Rebuilding is done to learn granular patterns and we do this by recursively splitting busy regions into 4 equal sub-regions so as to learn the transition patterns between these sub-regions better. The rebuilding is done as follows:

1. Loop through each of the nodes in the running graph and mark the nodes whose majority of transitions are to itself I.e.; self-loop has weight more than half of total outgoing weight.
2. These marked nodes denote that tracks in this region are more likely to stay in the same region than transition out of them. We thus divide this region into 4 equal sub-regions. Each of the new sub-regions is assigned a new node in the Transition Graph and the old parent node is removed
3. Once all the marked nodes are popped, we now need to assign edges between them. To do this, we loop through all existing tracks and find the normalized transitions between all the nodes in the new Transition Graph.
4. Once the Transition Graph is ready, we delete the previous Running and build a new running graph that is a duplicate of the new Transition Graph with no edges. The Running graph is then assigned Edge weights during matching and the Transition graph remains the same until the next re-build

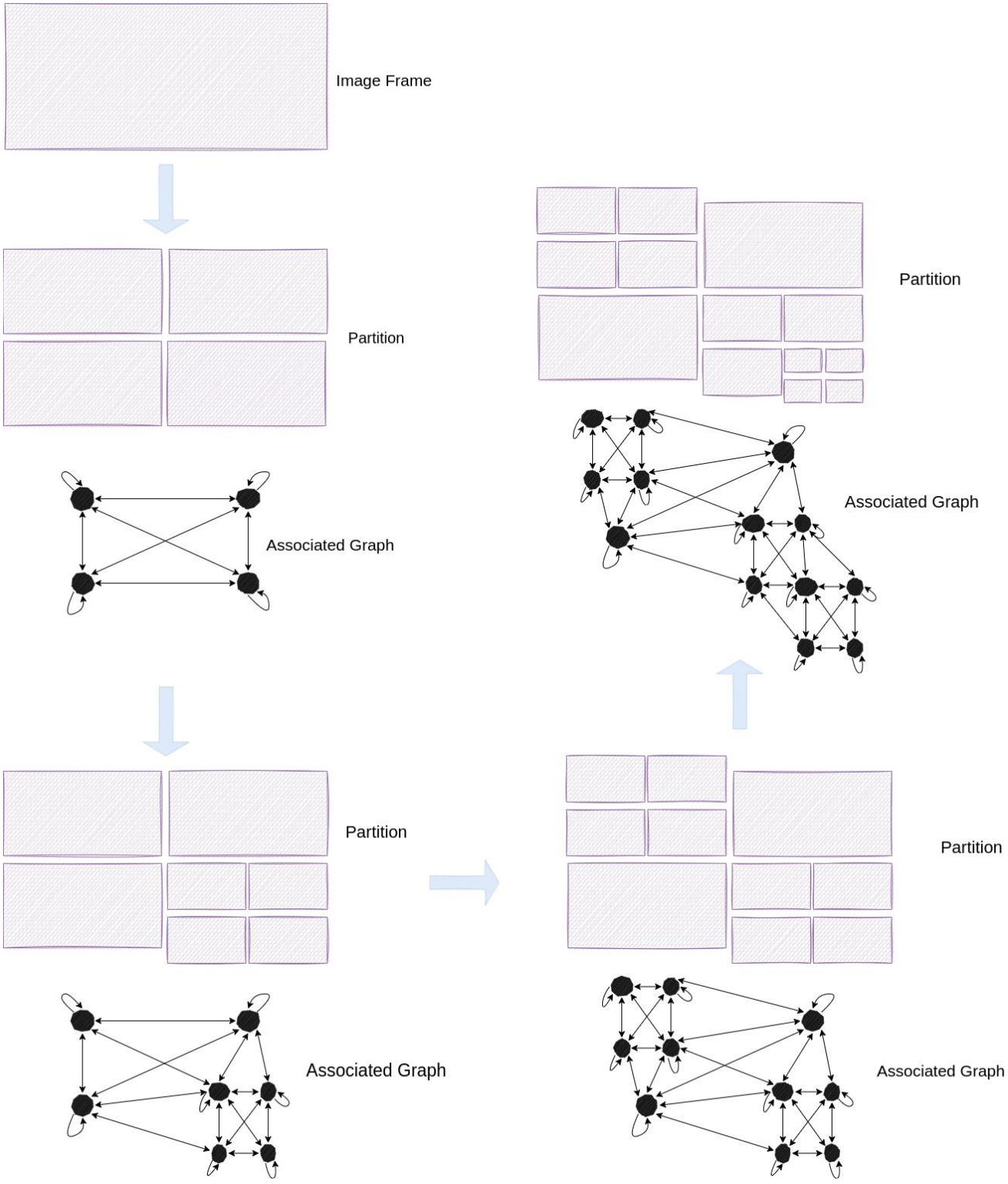

Figure 3: A representation of Graph Rebuilding

We assume that the video patterns are usually fixed, and the Transition Graph should be able to reach a steady state after some iterations of Rebuilding. We confirm this assumption in the Experiments section. Figure 3 shows the process of rebuilding. The Running Graph acts like a running memory of transitions between two rebuilds that determines which region needs to be partitioned further. AS the number of partitions increase, the time required to find a matching track decreases.

Together, the Running Graph and the Transition Graph learn from each other in a Bayesian manner. We start off with a prior of equal transition weights (0.25) and as the frames come in, the running graph collects information of transition based on the current weights. When rebuilding, the Transition Graph updates the prior information by learning from the object movements captured in the running graph between two consecutive rebuilds. After a rebuild, we start collecting latest information again in the running graph.

The Special Tree Data structure used to find the node names from pixel location. We use the python library anytree [24] to build and search this tree data structure. We use this special tree structure to represent the recursive sub-partitioning to avoid the rigidity that comes with using specialized encoding in case we want to change how the sub-partitions works or we want to increase the number of sub partitions from 4 to 6 or 8. Each node of this tree represents the abstract idea of a partition. Each node then has 4 potential children I.e., pointers to sub-partitions of each of its partitions. This can be better understood by Fig 4. Each node contains the pixel limits of that partition from x1 to x3 and y1 to y3. X2 and y2 are simply the mid points where the partitions separate from each other. Thus, the root node has x1=0 and x3=width of the original frame and y1=0 and y3=height. We name the partitions as p1, p2, p3, and p4 in clockwise order. Each time we split a partition, we extend the parent node's name by appending p1, p2, p3, and p4 to generate the names of the new sub-partitions. Thus, the name of the original partition is root and the first level of sub-partitions generated during initialization are rootp1, rootp2, rootp3, and rootp4. The next level of sub-partition of a node, say rootp4, would be rootp4p1, rootp4p2, rootp4p3, rootp4p4. This naming convention continues as we keep splitting the regions. The name of the node in which a pixel location lies can be found by searching for the leaf node which satisfies the condition of $x1<x<x3$ and $y1<y<y3$.

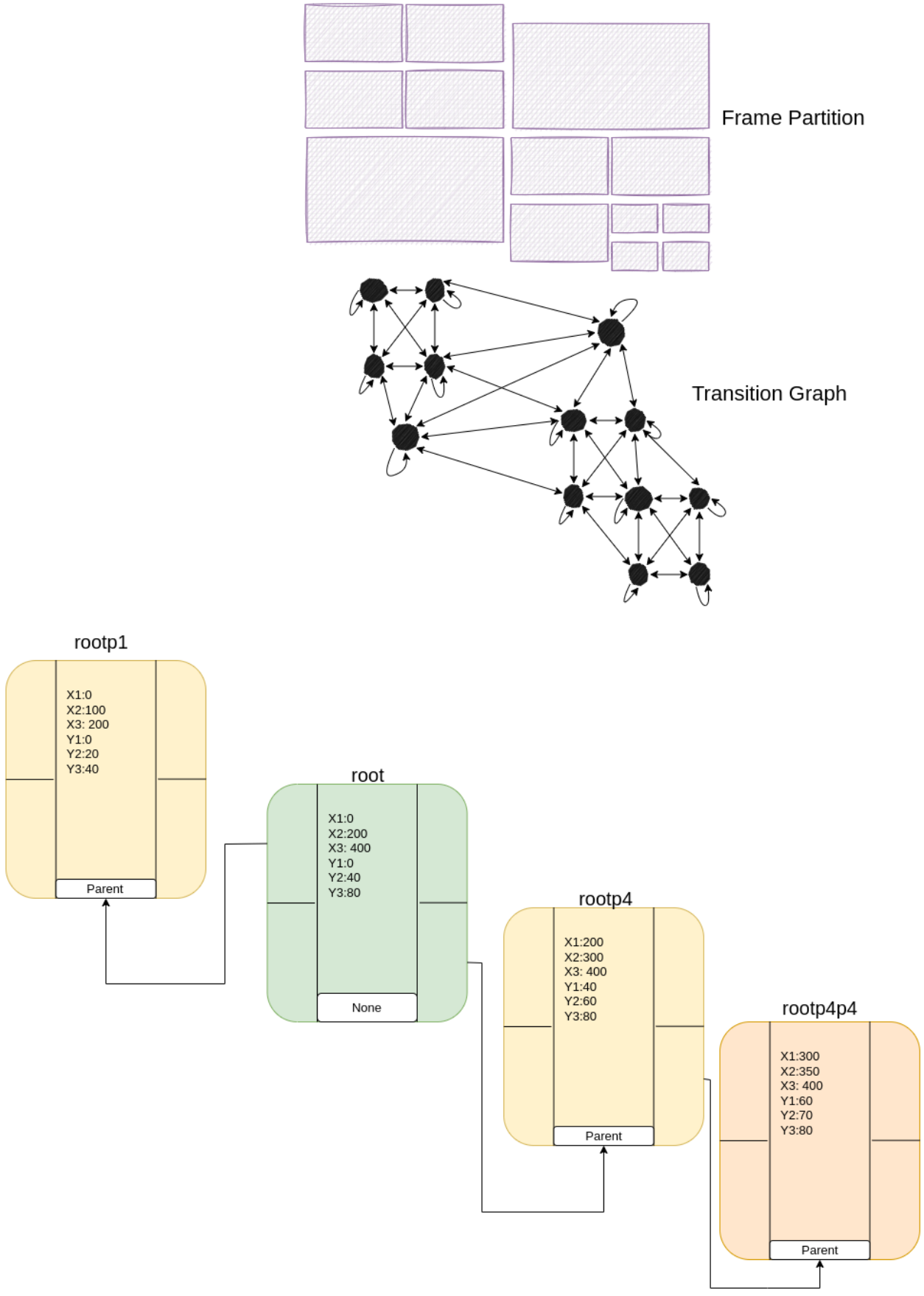

Figure 4: A representation of Special Tree Structure for node name storage

Eval

This process is responsible for generating the Qualitative performance metrics for each video. It uses a Python library called **motmetrics** [14]**.** This library accumulates ID assignments for each of the detections in each frame and produces final MOT metrics [1,15,21] by comparing the tracker detections and assignments with the Ground Truth detections and assignments present in the dataset. It accounts for False Positives, False Negatives, ID Switches, Precision and recall of Detections etc to produce overall Mulit-Object Tracking Accuracy (MOTA) and Multi-Object Tracking Precision (MOTP) described in the next section. As soon as a video is done, it sends this performance summary to the main process and clears and starts accumulating for the next video.

Experimental Design

We setup the MOT application to run both the Trackers: baseline as well as GTracker one after the other and compare their hardware resource consumption, Inference time, and Qualitative performance metrics. The machine used is an Intel core i7-8700, with 16 GB RAM, and an NVIDIA RTX 2060 GPU. The resource consumption is recorded by the monitor process described above. The only difference between the two methods is in the implementation of the tracker. The baseline does track matching of all detections with all tracks present whereas GTracker only performs matching of detections with the tracks last seen in the neighbourhood using the Transition Graph. While we periodically measure the CPU and RAM consumption of ReID processes in both cases, we also measure the inference time of the videos in two ways. One is the cumulative time I.e., the time period from the start of the video to time of sampling and the other is a periodic sampling where we sample the time taken to process every 200 frames. This periodic measurement gives us a detailed insight into the trend of inference time as the video is processed.

We record a variety of Qualitative performance metrics popularly used to measure MOT performance. These include Multi-Object Tracking Accuracy (MOTA) [1], Multi-Object Tracking Precision (MOTP) [1], ID Switches (IDs). Since the goal is to present an alternative to the baseline approach rather than present a new State-of-the-art method, we only present some of these metrics as a means of comparison between the two methods. More detailed metrics can be found in the logs present with the code on GitLab.

```
     num_frames      mota      motp
acc        1049  0.576374  0.171452
Stats for video:  MOT17-04-SDP
      IDF1   IDP   IDR  Rcll  Prcn GT MT PT ML   FP   FN IDs   FM  MOTA  MOTP IDt IDa IDm
full 62.6% 73.1% 54.8% 66.4% 88.5% 65 31 18 16 2490 9713  33  453 57.6% 0.171  40   7  25
part 75.3% 81.4% 70.0% 70.0% 81.4% 25 17  1  7    8   15   0    0 54.0% 0.160   0   0   0
End time: 84.40090489387512
```

Figure 5: An example of MOT metrics produced by motmetrics library

MOTA [1]: This is a highly expressive evaluation metric that combine three error terms: false positives, missed targets (false negatives) and ID switches. ID switch is when the tracker loses old identities or incorrectly swaps actual agent identities.

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

Hyperparameter Tuning

We choose 3 videos of varying lengths to present the results. The first is a short video with 400 frames, the next has about 1000 frames and the third is the longest with 2800 frames. We run both the trackers on all 3 videos one after the other multiple times to make sure the results are consistent. We manually tune the hyperparameters and thresholds by running some independent iterations on the feature being tuned. The main process has just one hyperparameter: the time it waits for the detector and ReID network to load before starting to send Video paths for processing.

The detector, Yolov5 [13], has 2 hyperparameters: Model IoU and Model Confidence threshold. The IoU is used for performing NMS when refining duplicate bounding boxes and lowering this leads to increased probability of duplicate detections. We carefully select this value through manual experimentation to be high enough to avoid duplicate detections of the same object but also low enough to be able to detect occlusions when one person passes in front of another. The confidence threshold is used to set the quality of detections and classifications and lowering this below a certain threshold tends to give bad detections. This threshold varies from video to video depending on the real-world Area covered by the frame and the range of object sizes seen in the video feed. A camera at the entrance of a building with a small region of interest usually tends to give confident detections whereas a CCTV monitoring a busy street, covering a few hundred square feet, does not detect people with the same confidence. Thus, we need to lower the confidence threshold in these scenarios to avoid missing detections at the cost of some bad detections. We set this threshold by experimenting with a few videos from our dataset and setting an average threshold that works for all the selected videos and gives minimal number of bad detections.

The Eval process has one Hyperparameter: IoU threshold. This is used by the evaluation process to associate the detection from the pipeline to the ground truth detections. It assigns these detections irrespective of the Track IDs generated as the track naming conventions could be different from Ground Truth ID naming.

The Trackers also have a few hyperparameters: Distance threshold, Feature Vector Weightage, and Track delete threshold. Distance Threshold is the threshold used to determine if a detection belongs to a track I.e., once the track assignment is done, we use the distance threshold to check if the assigned track is appropriate or if we need to create a new track for this detection. Increasing this distance threshold allows bad track matches while too low a threshold could create new IDs for existing tracks. Feature vector Weightage is used to produce an average Feature Vector for a track that is then used for track matching. It can be represented as follows:

$$Avg.\ FV = \alpha \times Avg.\ FV + \beta \times Det.\ FV$$

This allows us to control the influence of the new detection on the Average Feature Vector of the track. $\alpha = \beta = 0.5$ seems like a reasonable value but a lower β helps us avoid rapid altering of the average Feature Vector if a few new detections are of substandard quality. The track delete threshold is the number of contiguous frames with no detections of a particular object after which we delete its track.

Results

When we run the two Trackers, baseline and GTracker, we find that the Resources used by the two methods in terms of CPU and RAM are almost the same as seen in Figure 6. In the case of CPU usage, we clearly see a first short but sharp spike that represents Model loading for detection and ReID. We then see three distinct and increasing periods of activity. These are the areas where the three videos of increasing length are processed. As we can see, the Graphs mostly overlap except for some areas where the proposed approach takes slightly higher CPU. The Memory Usage in figure 7 shows that the two approaches use the same amount of RAM throughout the run.

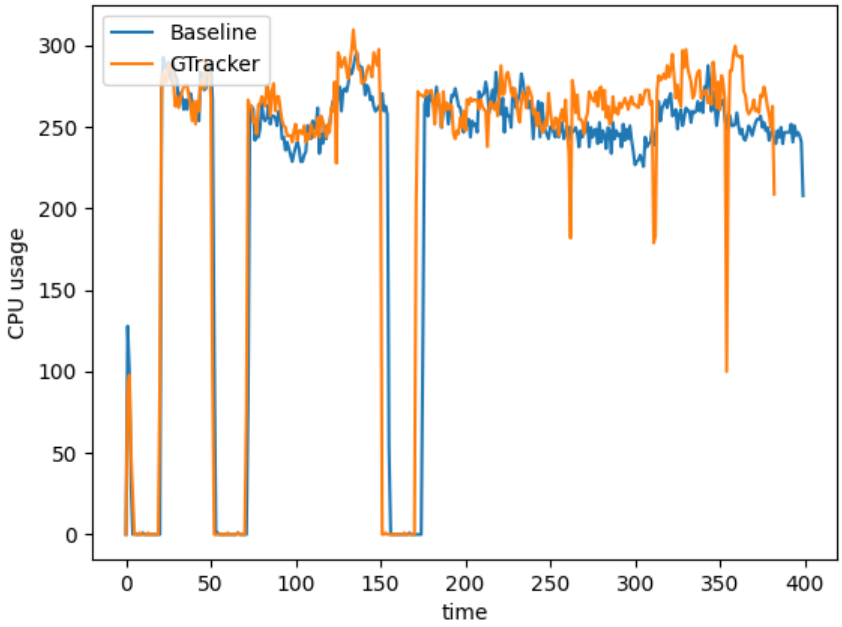

Figure 6: A Baseline v/s GTracker comparison of CPU usage by ReID process

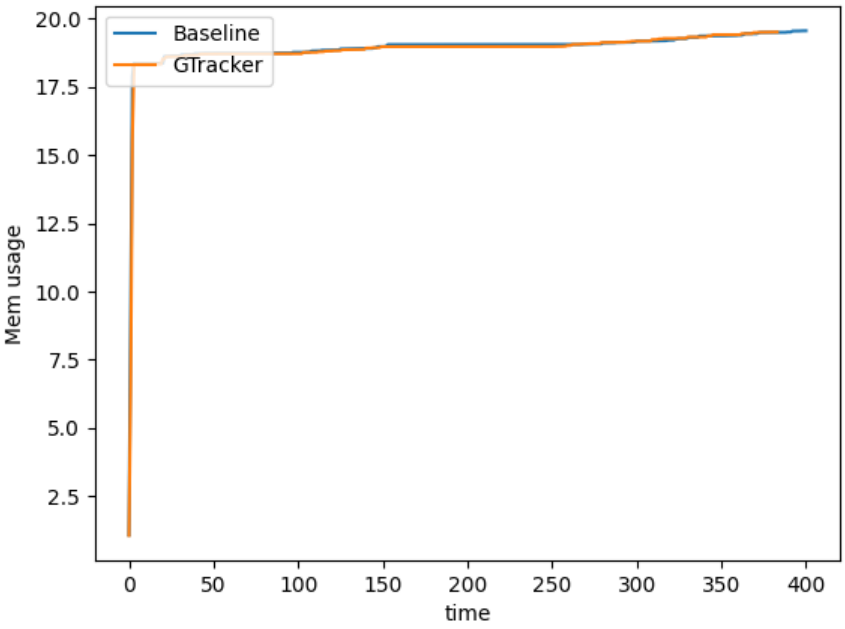

Figure 7: A Baseline v/s GTracker comparison of Memory usage by ReID process

We now look at the Inference Time and therefore throughput. Table 1 shows the Inference time and the average throughput comparison between the two methods. We find that the proposed GTracker takes less time to process the data and therefore gives a better throughput as compared to the baseline method. To better understand the difference between the throughputs of the two approaches, we sample the inference time of the longest video we have in our dataset measured in a cumulative as well as a periodic sense as described in the previous section. Figures 8 and 9 show the inference time measured for every 200 frames.

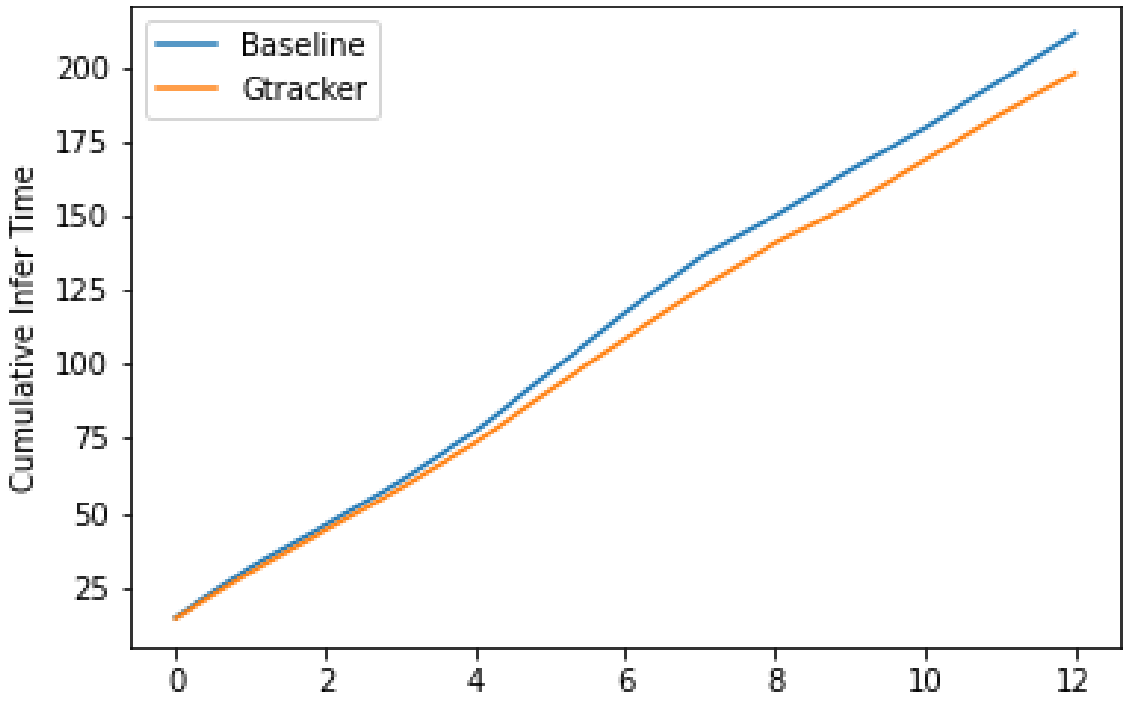

Figure 8: A Baseline v/s GTracker comparison of Cumulative Inference time

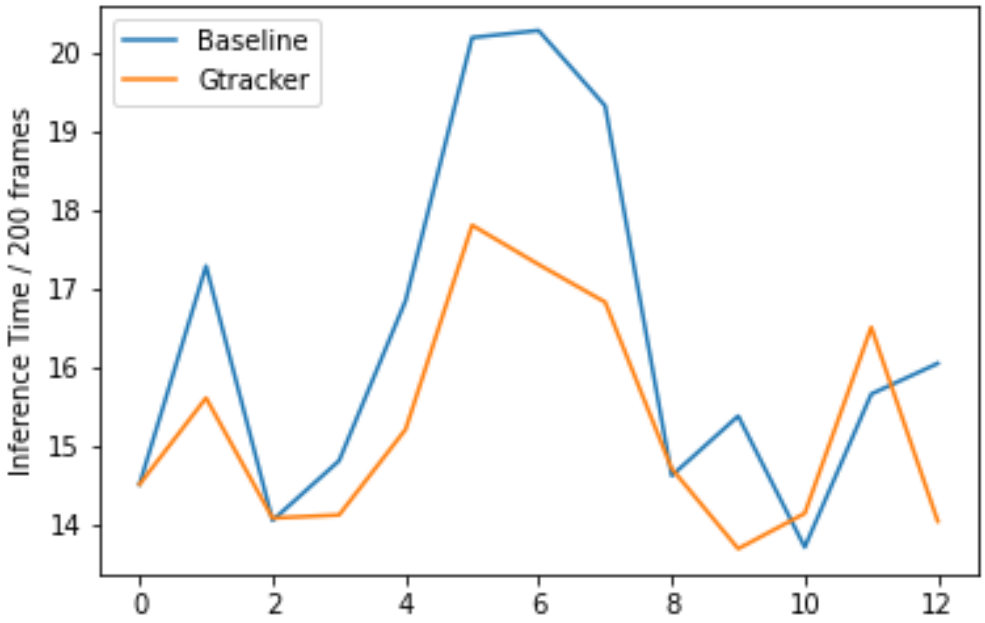

Figure 9: A Baseline v/s GTracker comparison of Periodic Inference time

		Baseline		**GTracker**	
Video Name	Frames	Inference Time in *s*	Throughput in *fps*	Inference time in *s*	Throughput in *fps*
MOT20-01	428	30.38	14.08	31.9	13.41
MOT17-04-SDP	1049	84.4	12.42	79.17	13.24
MOT20-02	2781	226	12.3	213.38	13.03
Total	4258	340.78	12.49	324.45	**13.123**

Table 1: Throughput comparison of Baseline and GTracker

We also compare the quality of Tracking done by the two methods as shown in fig 10 and find that the MOTA of both methods is very similar. We do find that the baseline performs negligibly better than the proposed approach.

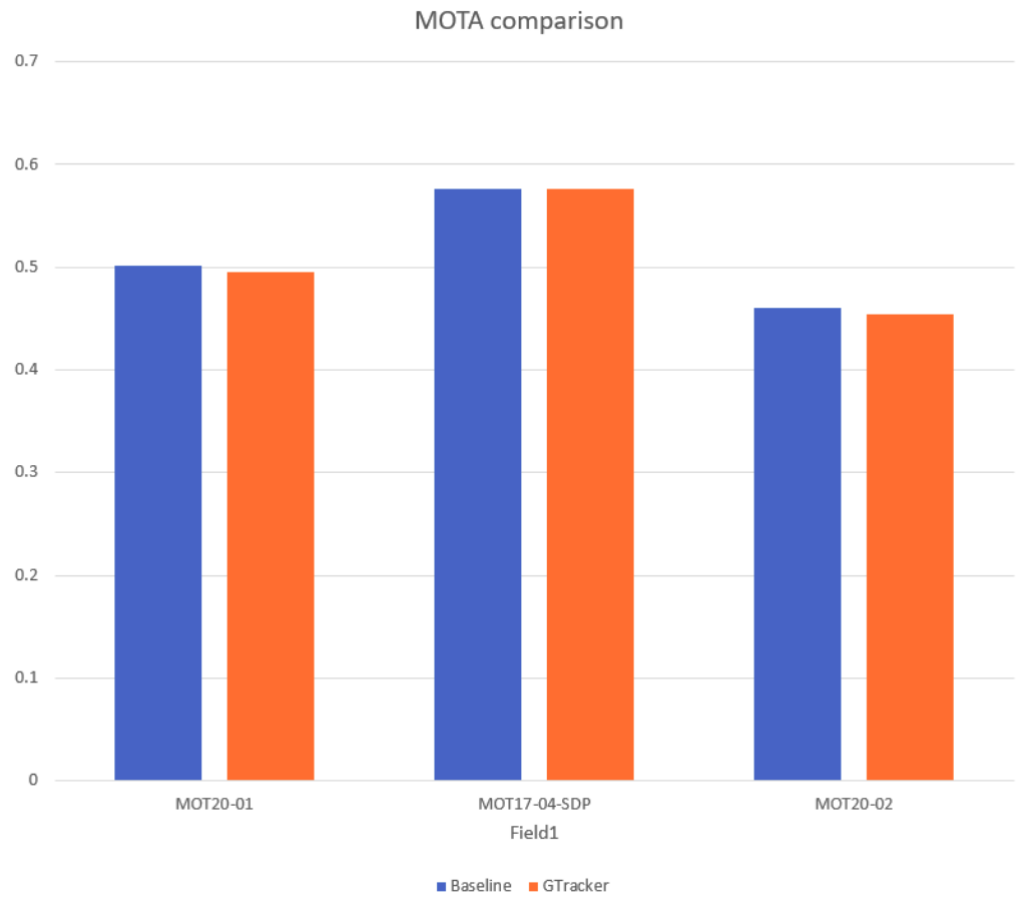

Figure 10: A Baseline v/s GTracker comparison of MOTA

Discussion and Future Work

From the Time comparison, we see that the baseline starts lagging around the 4th sample in figures 8,9. When we look at the logs, we find that the number of tracks being used at that time is 43. This is a high number, and it shows that as the scene gets busy, and the number of tracks/objects increases, our proposed GTracker starts giving better performance as compared to the baseline. This is because the baseline must search for a match for every detection in every track and whereas GTracker is not as affected by the increase in activity in the scene as it uses a much narrow search space when matching detections with Tracks. We also note an important implication of this due to our use case of CCTVs: CCTVs essentially run 24/7 and the application running Tracking must decode every single frame sent by the camera. This means that every frame must be processed and hence when the scene gets busy for some time and causes delays in the baseline method, the graph in fig(...) starts diverging. This Gap created due to this one period of rush is not closed since the throughput of GTracker does not decrease at any time to allow the baseline to catch up. Therefore, these periods of rush in the frame keep adding up the delays in the baseline over time and the time difference between the baseline and GTracker keep increasing and the gap in figure 8 keeps diverging increasingly. This means that in extended deployments, we expect to see a progressive increase in the throughput of GTracker. We also expect GTracker to have learnt the general patterns after reasonable time and expect that it does not have to keep rebuilding the Transition Graph, saving processing time and therefore increasing the throughput further.

From the results above we learn that the proposed GTracker is similar to the baseline approach in terms of resource consumption but gives has a better throughput or less processing time. This increase of just around 1 FPS in the throughput might seem small but in cases of Scaled Deployments such as smart cities that have thousands of cameras running, with each workstation/server responsible for a hundred camera feeds, this slight increase in throughput means that we can easily fit 8 extra cameras on the same machine as compared to earlier. This increase in throughput can also be utilized by other components of the software like the Application layer on top of the Tracker, Database, UI and Other DL-based Inference features.

The performance of GTracker can further be improved by implementing better DL Deployment methods such as batching, quantization, frame resizing, and processing only specific Regions of Interest rather than the whole frame. Detection can be run on a reduced frame size, but ReID can still be done on the crops from the original frame. We can also increase the number of partitions GTracker works on by making some simple changes to the code and this does not require any extensive design changes.

One main challenge of this method is that it may not perform very well when the area of activity is exactly in the middle of the frame. It may be difficult to model such movement patterns especially if we use the existing model of 4 partitions instead of a higher number like 6.

Future Work

- Ensemble – The Global Spatio-temporal patterns of a video frame might keep changing through the day or have different patterns for different days of the weeks. This information can be learnt if we are able to build some sort of Ensemble of stable-Transition graphs that can use previously learnt patterns if there is a concept drift.
- Dynamic – Introducing a merging mechanism in addition to the splitting mechanism while rebuilding the graph to allow for the graph to be more dynamic. We can also use this dynamic graph to understand how the global movement patterns change over time.
- Adding more parameters to distance measure calculation in addition to Cosine similarity

References

1. Keni Bernardin, Alexander Elbs, Rainer Stiefelhagen, ‘Multiple Object Tracking Performance Metrics and Evaluation in a Smart Room Environment,’ ECCV 2006.
2. Zheng, Liang, and Shen, Liyue and Tian, Lu and Wang, Shengjin and Wang, Jingdong and Tian, Qi, ‘Scalable Person Re-identification: A Benchmark,’ ICCV 2015
3. Kaiyang Zhou, Yongxin Yang, Andrea Cavallaro, Tao Xiang, ‘Omni-Scale Feature Learning for Person Re-Identification,’ ICCV 2019
4. Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi, ‘You Only Look Once: Unified, Real-Time Object Detection,’ CVPR 2016
5. Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Fucheng Weng, Zehuan Yuan, Ping Luo, Wenyu Liu, Xinggang Wang, ‘ByteTrack: Multi-Object Tracking by Associating Every Detection Box,’ ECCV 2022
6. Yifu Zhang, Chunyu Wang, Xinggang Wang, Wenjun Zeng, Wenyu Liu, ‘FairMOT: On the Fairness of Detection and Re-Identification in Multiple Object Tracking,’ International Journal of Computer Vision, 2021
7. Philipp Bergmann, Tim Meinhardt, Laura Leal-Taixe, ‘Tracking without bells and whistles,’ ICCV 2019
8. A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft, ‘Simple online and realtime tracking,’ ICIP 2016
9. Dendorfer, Patrick, Hamid Rezatofighi, Anton Milan, Javen Qinfeng Shi, Daniel Cremers, Ian D. Reid, Stefan Roth, Konrad Schindler and Laura Leal-Taix'e. “MOT20: A benchmark for multi object tracking in crowded scenes.” ArXiv abs/2003.09003 (2020)
10. Leal-Taixé, Laura, Anton Milan, Ian D. Reid, Stefan Roth, and Konrad Schindler. “MOTChallenge 2015: Towards a Benchmark for Multi-Target Tracking.” ArXiv abs/1504.01942 (2015)
11. Xingyi Zhou, Vladlen Koltun, and Philipp Krähenbühl, ‘Tracking Objects as Points,’ ECCV 2020
12. Caglayan Dicle, Octavia I. Camps, and Mario Sznaier, ‘The Way They Move: Tracking Multiple Targets with Similar Appearance,’ ICCV 2013
13. Glenn Jocher, YoloV5, https://github.com/ultralytics/yolov5
14. Cheind, Motmetrics, https://github.com/cheind/py-motmetrics
15. E. Ristani, F. Solera, R. S. Zou, R. Cucchiara and C. Tomasi, ‘Performance Measures and a Data Set for Multi-Target, Multi-Camera Tracking,’ ECCV 2016
16. Elahe Arani, Shruthi Gowda, Ratnajit Mukherjee, Omar Magdy, Senthilkumar Kathiresan, Bahram Zonooz, ‘A Comprehensive Study of Real-Time Object Detection Networks Across Multiple Domains: A Survey,’ Transactions on Machine Learning Research, 2022
17. Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, Piotr Dollár, ‘Microsoft COCO: Common Objects in Context,’ ECCV 2014

18. Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, Dilip Krishnan,' Supervised Contrastive Learning',NeurIPS 2020
19. N. Wojke, A. Bewley and D. Paulus, "Simple online and realtime tracking with a deep association metric," IEEE International Conference on Image Processing (ICIP), 2017
20. ShiJie Sun, Naveed Akhtar, HuanSheng Song, Ajmal Mian, Mubarak Shah, 'Deep Affinity Network for Multiple Object Tracking,' IEEE Transactions on Pattern Analysis and Machine Intelligence 2021
21. Milan, Anton and Leal-Taixe, Laura and Reid, Ian and Roth, Stefan and Schindler, Konrad, 'MOT16: A Benchmark for Multi-Object Tracking', arXiv 2016
22. Kaiyang Zhou, TorchReID, https://github.com/djidje/deep-person-reid-1
23. Scikit Learn, https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linear_sum_assignment.html
24. Anytree, https://anytree.readthedocs.io/en/latest/

Appendix

All Code can be found in the school's GitLab repository:

https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2021/sxa1507

The source code is found in the folder 'code/' and 'test_dev ' folder contains some test scripts used during development.

All the .py files in 'code' are part of the application that starts using main.py. The 'models/reid' contains the ReID OSNet model. And the .png files are outputs of the application listing the CPU and Mem usage of processes. The 'typescript' file is a log of the command line output of the application during runtime. Here we can see detailed logs for each video, for both baseline and GTracker method, along with their throughput and qualitative results.

The 'test_dev' folder also has a file called 'requirements.txt.' We can use this file to setup Python environment needed to run our application by using the command 'pip3 install -r requirements.txt'.

A code/data folder is expected to be created and have video folders in MOT Challenge format.

To run the main application, just go to the code folder and run 'python3 main.py'