UNIVERSITY OF CALIFORNIA, MERCED

Adaptive Edge Systems for Smart IoT Applications

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Electrical Engineering and Computer Science

by

Miaomiao Liu

Committee in charge:

Professor Wan Du, UC Merced, Chair
Professor Shawn Newsam, UC Merced
Professor Dong Li, UC Merced

Spring 2023

The dissertation of Miaomiao Liu is approved, and
it is acceptable in quality and form for publication
on microfilm and electronically:

Professor Wan Du, Chair

Professor Shawn Newsam

Professor Dong Li

University of California, Merced

Spring 2023

DEDICATION

I dedicate this dissertation to all of the professors and mentors who have shaped my life and my career. Their wisdom, guidance, and encouragement have been instrumental in helping me to achieve my goals, and I am forever grateful for their influence. I am also grateful to my friends and colleagues, who have supported me throughout this journey and who have made this work a truly collaborative effort. Finally, I dedicate this dissertation to my family, who have always been my greatest source of inspiration and my biggest supporters. Without their love and encouragement, I would not be where I am today. Thank you all for your unwavering support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to express my profound gratitude for the incredible journey that has led me to this point. When I began my Ph.D study in Fall 2018, I could not have imagined the depth and breadth of experiences that awaited me.

I would like to express my sincere gratitude to my advisor, Professor Wan Du, who has played an instrumental role in the completion of my Ph.D. program. His guidance, support, and expertise have been the cornerstones of my academic journey. I will always be grateful for the countless hours he devoted to discussing my research ideas, providing insightful feedback on my work, and encouraging me to strive for excellence. His commitment to academic rigor and his passion for research have been constant sources of inspiration for me.

I am also deeply grateful to the members of my dissertation committee, Professor Shawn Newsam and Professor Dong Li, for their invaluable feedback and for their willingness to share their knowledge and expertise. Their insights and comments have been essential to the refinement of my dissertation.

I would like to express my appreciation to my colleagues and friends at UC Merced, Xianzhong Ding, Kang Yang, Sikai Yang, Wyssanie Chomsin, Yuning Chen, Zhiyu An, Bohao Xu, Weifeng Gao, Haiqiao Wu, and Di An. I am grateful for the many discussions, debates, and brainstorming sessions that have helped me to develop my ideas and to improve my arguments. I am also grateful for the encouragement during times of struggle.

I sincerely thank my collaborators outside UC Merced, Professor Yanjie Fu and Professor Dapeng Wu, for their guidance and suggestions on my work.

I would like to thank the authors of the papers I have read, who I have never met, but who have generously responded to my emails and provided valuable advice. Their willingness to help and their kindness have saved me a significant amount of time and prevented me from making mistakes in my research.

Finally, I want to extend my deepest appreciation to my family and friends, whose unwavering love and support have been the pillars of my study. Their belief in me has given me the strength to overcome challenges.

Thank you all for everything. It is you who have made me who I am today.

VITA

2010 - 2014	B. S. in Software Engineering, Northeastern University (China)
2015 - 2018	M. S. in Software Engineering, Northeastern University (China)
2018 - Now	Ph. D. in Electrical Engineering and Computer Science, University of California, Merced

PUBLICATIONS

Miaomiao Liu, Sikai Yang, Wyssanie Chomsin, and Wan Du. *Demo Abstract: Real-Time Tracking of Smartwatch Orientation and Location by Multitask Learning*, in Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems **(SenSys)**, 2022.

Miaomiao Liu, Sikai Yang, Wyssanie Chomsin, and Wan Du. *Real-Time Tracking of Smartwatch Orientation and Location by Multitask Learning*, in Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems **(SenSys)**, 2022.

Miaomiao Liu, Kang Yang, Yanjie Fu, Dapeng Oliver Wu, and Wan Du. *Driving Maneuver Anomaly Detection based on Deep Auto-Encoder and Geographical Partitioning*, ACM Transactions on Sensor Networks **(TOSN)**, 2022.

Miaomiao Liu, Xianzhon Ding, and Wan Du. *Continuous, Real-Time Object Detection on Mobile Devices without Offloading*, in Proceedings of the 40th IEEE International Conference on Distributed Computing Systems **(ICDCS)**, 2020.

Miaomiao Liu and Wan Du. *Poster Abstract: Geo-Distributed Driving Maneuver Anomaly Detection*, in proceedings of the 7th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation **(BuildSys)** 2020.

Miaomiao Liu, Sikai Yang, Arya Rathee, and Wan Du. *RLPilot: Deep Reinforcement Learning Piloted Classical Orientation Estimation*, under submission.

Kang Yang, **Miaomiao Liu**, and Wan Du. *RALoRa: Rateless-Enabled Data Rate Adaption for LoRa Networks*, under submission.

Sikai Yang, **Miaomiao Liu**, and Wan Du. *Magnetic Distortion Resilient Orientation Estimation*, under submission.

ABSTRACT OF THE DISSERTATION

Adaptive Edge Systems for Smart IoT Applications

by

Miaomiao Liu
Doctor of Philosophy in Electrical Engineering and Computer Science
University of California Merced, Spring 2023
Professor Wan Du, UC Merced, Chair

The proliferation of the Internet of Things (IoT) and cloud services has given rise to the edge computing paradigm, where data is processed partly or entirely at the edge of the network, rather than solely in the cloud. Edge computing can address problems such as latency, limited battery life of mobile devices, bandwidth costs, security, and privacy [1, 2, 3]. Typical applicable scenarios based on edge computing include video analytics, smart home, smart city, and collaborative edge.

With the development of deep learning techniques, research on employing deep learning to develop intelligent edge systems is emerging. In this dissertation, we aim to investigate how deep learning can process data on source-constrained individual edge devices in real time and how deep learning can process data by utilizing collaborative edge devices to provide better services.

We build several critical systems, including video analytics, driving anomaly detection, arm posture tracking, and device orientation tracking. In the video analytics system, we combine deep learning with traditional image processing techniques to achieve real-time object detection on mobile devices without offloading. In the driving anomaly detection system, we train deep learning models for driving anomaly detection by leveraging the information from collaborative peer devices to provide better accuracy. In the arm posture tracking system, we employ multitask learning to track the orientation and location of the wrist simultaneously, which significantly improves the latency compared to the conventional methods. In the

device orientation tracking system, we develop a deep reinforcement learning framework to train an agent that adjusts the parameters of a conventional orientation tracking method in response to changing environments.

As IoT systems continue to grow in complexity and size, preserving training data has become an increasingly important challenge. In our future work, we plan to investigate the use of representation learning to address this issue. By leveraging representation learning techniques, we aim to develop a more robust and efficient method for saving and utilizing training data in IoT systems. This could enable better performance of IoT systems, which in turn could lead to significant improvements in a variety of fields, such as healthcare, transportation, and manufacturing.

Chapter 1

Introduction

The widespread adoption of wireless networks and IoT devices has led to a significant increase in the number of edge devices, which are located at the periphery of the network. These devices generate a vast amount of data every day, which traditionally has been offloaded to data centers for processing. This centralized processing approach is known as cloud computing, but it has several limitations. First, it cannot guarantee real-time responses for applications with time-sensitive requirements. Second, transferring large amounts of data to data centers can strain network bandwidth. Additionally, cloud computing raises concerns about data security and privacy.

In response to these challenges, edge computing was proposed as a distributed computing technology that enables data to be stored and processed closer to the devices where it is generated. This approach can significantly reduce application response times and conserve network bandwidth. Furthermore, since the data does not need to be transferred to data centers, there are fewer concerns about data security and privacy [1, 2, 3]. Research in this area has shown promising results, with edge computing demonstrating its potential to enable new use cases and improve the performance of existing applications. Edge computing is becoming increasingly popular in various fields, such as healthcare, transportation, and manufacturing, where real-time response and data privacy are critical.

IoT devices, with their network connectivity, enable the automatic collection of data from the environment, which in turn, allowing us to develop more intelligent

edge systems. Deep learning has been particularly effective in bringing intelligence to these systems and improving their performance in various applications.

In this dissertation, we present methods for developing smart IoT applications using deep learning techniques on resource-limited edge devices. We introduce three applications in this direction: the first uses a full DNN model to detect objects in real-time videos on edge devices. The second utilizes deep learning techniques to track arm postures on mobile devices using the IMU readings of smartwatches. The third combines deep reinforcement learning with classical algorithms to track the orientation of a mobile device. Our approaches significantly reduce resource usage, reduce latency, and improve accuracy compared to conventional methods. We also extend our approach to multiple edge devices by demonstrating a driving anomaly detection system that utilizes collaboration among multiple devices to achieve better system performance. Our approach highlights the potential benefits of edge computing for distributed systems, particularly in scenarios where real-time response and resource efficiency are critical factors.

In Chapter 2, we introduce *AdaVP*, the real-time video processing system which adapts to the change of video content on resource-limited mobile devices. Automatic object detection and recognition in videos become a fundamental component in many mobile applications, such as traffic monitoring and control, autonomous driving, and augmented reality. However, executing large object detection deep learning on mobile devices in real-time remains a challenge. Additionally, urban traffic monitoring systems generate a vast number of videos daily, and streaming all of these videos to the cloud consumes excessive bandwidth and can induce network congestion. To address those challenges, we develop *AdaVP*, which employs object detection and object tracking for video object detection tasks on mobile devices. The object detector is heavyweight, while the object tracker is lightweight, and the combination of the two speeds up computation. However, due to changes in video content over time, this approach may not always achieve high accuracy. To overcome this challenge, *AdaVP* proposes a model setting adaptation method which aims at dynamically switching the settings of the system at runtime, depending on the changes of video content. The material in this Chapter appears in the 40th

IEEE International Conference on Distributed Computing Systems. The authors are Miaomiao Liu, Xianzhong Ding, and Wan Du. © 2020 IEEE.

In Chapter 3, we present *GeoDMA*, the driving maneuver anomaly detection system that collaboratively utilizes multiple mobile devices. Abnormal driving maneuvers are responsible for numerous fatal accidents every year, making it essential to detect them and alert nearby pedestrians or drivers in a smart city. The GPS data of vehicles contains a wealth of information about reckless driving, which can be potentially utilized to detect abnormal driving maneuvers. First, the system processes the GPS data from multiple vehicles by using a deep learning model to do the anomaly detection. Second, driver maneuvers can vary based on the location in the city. For example, in the downtown, the vehicles usually have more stop-and-go due to the more traffic lights and the speed of them are usually slower. We further develop a geographical partitioning algorithm to divide a city into several sub-regions to adapt to the road and traffic conditions in each sub-region. This allows for more precise anomaly detection in specific areas. The material in this Chapter appears in ACM Transactions on Sensor Networks. The authors are Miaomiao Liu, Kang Yang, Yanjie Fu, Dapeng Wu, and Wan Du. © 2023 Copyright held by the authors.

In Chapter 4, we present *RTAT*, the first 3D human wrist tracking system that utilizes inertial measurement unit (IMU) data from smartwatches. This system tracks both the orientation and location of the wrist simultaneously using a multitask learning neural network. Since not all IMU sensors are always important for orientation estimation, we design an attention mechanism and a smooth loss in the multitask neural network to further improve its performance. The attention mechanism focuses on the most important IMU sensors for orientation estimation, while the smooth loss ensures a smooth output trajectory. *RTAT* is lightweight and supports real-time tracking on smartphones with high sampling frequency. Compared to conventional methods, *RTAT* significantly reduces resource usage and improves latency. To support *RTAT*, we also develop a ground truth data collection system based on a VR system to collect orientation and location labels. This enables us to train and evaluate the system with high-quality labeled data.

In Chapter 5, we briefly introduce an adaptive device orientation estimation system piloted by deep reinforcement learning. We then present the future directions of what we plan to study based on common problems - saving labeled training data for smart IoT applications.

Chapter 2

Continuous, Real-Time Object Detection on Mobile Devices without Offloading

This chapter presents *AdaVP*, a continuous and real-time video processing system for mobile devices without offloading. *AdaVP* uses Deep Neural Network (DNN) based tools like YOLOv3 [4] for object detection. Since DNN computation is time-consuming, multiple frames may be captured by the camera during the processing of one frame. To support real-time video processing, we develop a mobile parallel detection and tracking (MPDT) pipeline that executes object detection and tracking in parallel. When the object detector is processing a new frame, a light-weight object tracker is used to track the objects in the accumulated frames. As the tracking accuracy decreases gradually, due to the accumulation of tracking error and the appearance of new objects, new object detection results are used to calibrate the tracking accuracy periodically. In addition, a large DNN model produces high accuracy, but requires long processing latency, resulting in a great degradation for tracking accuracy. Based on our experiments, we find that the tracking accuracy degradation is also related to the variation of video content, e.g., for a dynamically changing video, the tracking accuracy degrades fast. A model adaptation algorithm is thus developed to adapt the DNN models according to the change rate of video content. We implement *AdaVP* on Jetson TX2 and conduct

a variety of experiments on a large video dataset. The experiments reveal that *AdaVP* improves the accuracy of the state-of-the-art solution by up to 43.9%.

2.1 Introduction

Continuous and real-time object detection is essential for many mobile applications, like traffic monitoring [5] and augmented reality (AR) [6]. For example, real-time warnings can be sent to road users automatically by a camera installed on top of a highway road if any reckless driving maneuvers are detected. AR-based videos are promising in many applications, such as tourism, navigation and entertainment [7], which require the detection and tracking of objects in videos on mobile devices continuously and in real time, like 30 or 60 Frames Per Second (FPS). Deep learning has shown superior performance in object detection and many deep neural networks have been developed, like YOLO [8] and SSD [9]. Although they achieve high accuracy, they normally involve intensive computation that cannot be fully supported by the constrained hardware resource of mobile devices, i.e., they cannot accomplish the processing of one frame before the next frame is captured, i.e., 33 ms for 30 FPS. Some solutions offload a part of computation from mobile devices to the cloud [10, 11, 12]. However, offloading suffers from privacy concerns and unpredictable network latency [13].

Recently, some compressed DNN models have been developed to do object detection on mobile devices without offloading, e.g., YOLOv3-tiny[8] and Faster R-CNN based on MobileNets [14]. Our experiments show that YOLOv3-tiny can process a frame on an Nvidia Jetson TX2 within 60 milliseconds, but its detection accuracy is low. At the same time, some light-weight deep learning frameworks have been developed, such as DeepMon [13] and NestDNN [15]; whereas they cannot meet the real-time requirement of video processing. For example, DeepMon achieves continuous video processing at 1-2 frames per second [13].

In this chapter, we develop *AdaVP*, an accurate and real-time video processing system on mobile devices. *AdaVP* is based on a novel parallel object detection and tracking pipeline, named as Mobile Parallel Detection and Tracking (MPDT).

Nowadays, many mobile devices have a GPU, like the Samsung Galaxy S10 and the Apple iPhone 11, which allows us to implement DNN-based object detection on the GPU and object tracking on the CPU. The two types of operations are executed independently on two different hardware resources.

At the beginning, we use a general DNN-based object detector (i.e., YOLOv3 [4] in our current implementation) to process one frame (frame 0). After the processing of that frame (e.g., 330 ms), the camera may have already captured several frames in the buffer (e.g., 11 frames at a capture rate 30 FPS). The object detector will start processing the newest frame in the buffer (e.g., the 12th frame). At the same time, based on the objects identified by the object detector, an object tracking algorithm will localize these objects in the accumulated frames (the 1st-11th frames). The tracking accuracy degrades gradually due to the accumulation of tracking error and the appearance of new objects. Before the tracking accuracy drops to too low, the object detector will provide the objects in a new frame (e.g., the 12th frame) with a high detection accuracy. By parallel detection and tracking, we can obtain the object detection results at the maximum frequency and use them to calibrate the object tracking.

To further improve the accuracy of the proposed parallel detection and tracking pipeline, we adjust the settings of the DNN object detection model at runtime, based on the tradeoff of the initial object detection accuracy and the accuracy degradation of object tracking. On the one hand, a high object detection accuracy normally requires intensive computation (i.e., a heavy-weight DNN model) and long processing time, which means more frames will be accumulated in the buffer. One the other hand, the object tracking accuracy degrades sharply if the video content changes fast, since the tracking error accumulates fast and many new objects may appear in the accumulated frames. In this case, we need to run a light-weight object detector in order to calibrate the object tracker more frequently, although the detection accuracy is relatively low. On the contrary, if the video content changes slowly, we prefer to use a heavy object detector that provides high detection accuracy. Although the detection latency is long, it does not cause the tracking accuracy to degrade much.

In this work, we adapt the DNN model settings by changing the frame size of YOLOv3 at runtime. YOLOv3 allows us to change the frame size at runtime without reloading the model. A large frame size indicates long computation latency and high detection accuracy. In *AdaVP*, we measure the video content change rate based on the intermediate results of object tracking. We learn the quantified relationship between the best frame size and the video content changing rate, based on a large amount of training data. We then develop a DNN model setting adaptation algorithm that decides whether to switch to a different frame size after each object detection.

To perform object tracking, we use the standard *good features to track* [16] method to extract good features in the last DNN detected frame. And then we track these features in the following frames by the *Lucas-Kanade* optical flow method [17]. Due to the tracking and rendering latency of one frame (from 57 to 70 ms) is larger than the frame interval of a video (e.g., 33 ms), we also design a scheme to skip some frames from tracking without impacting the synchronization with the operations of object detection.

We implement *AdaVP* on an open mobile platform, the Nvidia Jetson TX2, based on the Pytorch deep learning framework. We use both a standard video dataset [18] and some videos downloaded from public websites [19, 20] to do experiments. The data we used to train our DNN model adaptation module contains 105205 frames, and the videos for validation contain 141213 frames. The evaluation results show that *AdaVP* improves the accuracy of the state-of-the-art solution up to 43.9%. In particular, the parallel detection and tracking pipeline (MPDT) improves accuracy up to 21.95%, and the model setting adaptation algorithm in *AdaVP* further improves accuracy up to 34.1% on top of MPDT.

In summary, the contributions of this work are as follows:

- We develop *AdaVP*, a mobile video processing system that achieves high detection accuracy in real time on mobile devices without offloading.
- We develop a parallel detection and tracking pipeline to fully utilize the computation resources on current mobile devices for high detection accuracy.

Table 2.1: Comparison of *AdaVP* and existing work

	Glimpse [21]	Tiny YOLO [8]	Deep Mon [13]	Deep Cache [22]	Deep Decision [11]	Liu et al. [6]	MARLIN [23]	*AdaVP*
Real-time updates	✓	✓				✓	✓	✓
No offloading		✓	✓	✓			✓	✓
Localizes multiple objects		✓	✓	✓	✓	✓	✓	✓
Using DNN		✓	✓	✓	✓	✓	✓	✓
Model adaptation								✓
High accuracy			✓	✓		✓		✓

- We further increase the detection accuracy by adjusting the DNN model settings at runtime according to the variation of video content.
- We implement the system on an open mobile platform, the Nvidia Jetson TX2 and extensively evaluate the system using different types of videos.

2.2 Related work

We compare AdaVP with some prior works in terms of some common features of video processing in Table 2.1.

Real-Time Mobile Vision without Offloading. MARLIN [23] is the latest work about real-time mobile vision without offloading. It runs the object detector and object tracker in a sequential order. When the object tracker starts tracking the objects in the accumulated frames, the object detector stops its detecting task. MARLIN is inefficient when the video content becomes complex and varies fast. And it always uses a fixed DNN model setting during video processing. Different

from MARLIN, AdaVP runs the object detector and object tracker in parallel and switches among different input size settings according to the change of video content at runtime.

Real-Time Mobile Vision with Offloading. Due to the limited computation, storage and power of mobile devices, offloading intensive tasks to the cloud to process is popular [24, 25, 11, 26, 12, 21]. However, mobile vision offloading approaches are easily affected by network conditions and usually receive stale results which degrade the processing accuracy of mobile continuous vision. MCDNN [12] and DeepDecision [11] design a framework which decides to offload tasks to the cloud or execute locally according to the network conditions. Glimpse [21] and Liu et al. [6] offload some key frames to the cloud to do detection and track the detected objects on mobile devices. However, they all suffer from long transmission latency and privacy concerns. In contrast, AdaVP focuses on executing object detection on mobile devices without offloading.

Mobile Deep Learning. With the development of deep learning and deep reinforcement learning [27, 28], many works seek to reduce the latency and computation time of deep learning algorithms on mobile devices. DeepX [29] designs a pair of resource control algorithms for deep learning inference. DeepMon [13] designs a suite of optimization techniques to accelerate the processing of DNNs on mobile devices. DeepEye [30] proposes a novel inference software pipeline that enables multiple CNN models to execute locally without offloading. DeepCache [22] presents a principled cache design for deep learning inference in continuous mobile vision. NestDNN [15] takes the dynamic runtime resources into account to enable resource-aware on-device deep learning for mobile vision systems. Naderiparizi et al. design [31] a novel architecture that gates wearable vision using low-power vision modalities to reduce mobile power and data usage. AdaDeep [32] develops a usage-driven selection framework to automatically select a combination of compression techniques for a given DNN. Our work is orthogonal and complementary to these prior works, it can work on the top of the above works to make the camera-based mobile applications more efficiently.

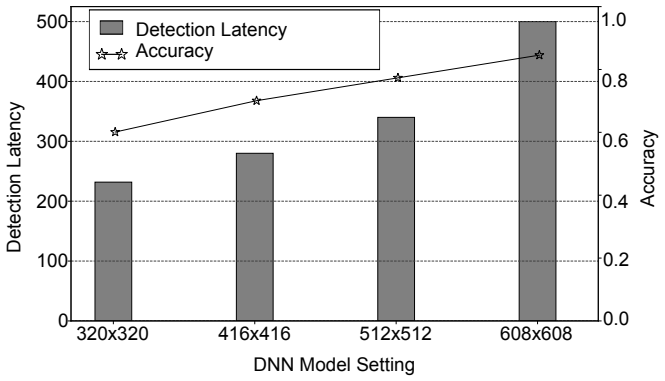


Figure 2.1: Detection latency and accuracy per frame for different frame sizes. The latency is presented by the bars, and the accuracy is shown by the lined stars.

2.3 Motivation

In this section, we conduct experiments on real mobile devices to evaluate the performance of state-of-the-art object detection and tracking models. From the experimental results, we derive four observations that motivate our design.

2.3.1 Experimental Setting

DNN-based Object Detection. Two DNN-based object detection pipelines are widely used. The first type detects and classifies objects by a single DNN object detection model, e.g., Faster RCNN [33], SSD [9] and YOLO [8]. The second type first detects the regions of interest using background subtraction and then classifies each small region using a DNN classification model, e.g., ResNet [34] and InceptionV4 [35]. The latter sometimes is insufficient if there are many regions of interests to detect [36]. In this work, we thus adopt a single DNN model for both detection and classification.

In particular, we use YOLOv3 [4] based on the following considerations. 1) YOLOv3 has an optimal overall performance (i.e., accuracy and latency) among all the object detection architectures [4]. On the same hardware, YOLOv3 runs significantly faster than other detection models such as SSD and R-FCN, and still provides comparable accuracy [37]. 2) YOLOv3 scales with a set of input frame sizes, which further determines the processing time of one frame. We can

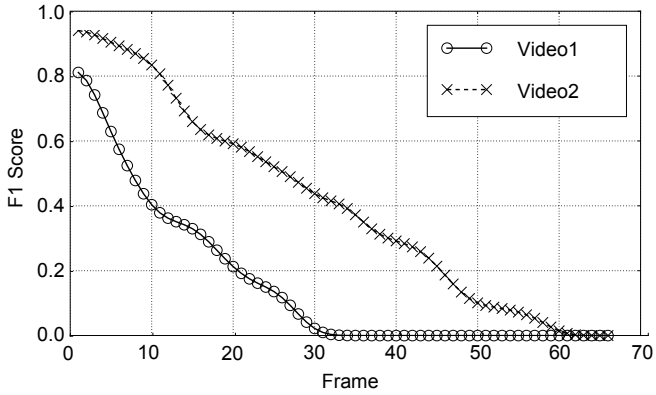


Figure 2.2: Tracking accuracy of two different videos. The content of Video 1 changes faster than that of Video 2. YOLOv3-608 is used to detect the objects in the first frame.

change the input frame size of YOLOv3 during the processing of one video without changing the weights of the model. This feature allows us to adjust its accuracy and detection latency tradeoff without reloading a new model.

Object Tracking Algorithms. When the DNN model is detecting the objects in one frame, *good features to track* [16] algorithm is used to extract good feature points from the detected frame. And then we implement an object tracking algorithm based on the standard *Lucas-Kanade* method [17]. The details of object tracking can be found in 2.4. The *Lucas-Kanade* method is widely used in some prior works [21, 23]. It can estimate the positions of feature points in next frame by a local image flow (velocity) vector (V_x ,V_y).

Besides tracking objects, it is unique in our work to use the intermediate results of the *Lucas-Kanade* method to adapt the DNN model settings. We need to detect how fast the video content changes in time. The average moving velocity of key points between frames is a lightweight and good indicator to describe the change rate of video content.

Hardware. We use the Nvidia Jetson TX2 as the mobile device in this work. TX2 is a representative open-source mobile platform that can easily establish a mobile development environment with the Jetson TX2 development kit, like configuring cuDNN, CUDA Toolkit. It includes 6 CPU cores, 8-GB DRAM and an integrated 256-core Pascal GPU. In our implementation, we run Ubuntu 16.04

OS on the Nvidia Jetson TX2 platform. We implement our system based on Nvidia JetPack[38], Nvidia TensorRT [39] and Nvidia Video Codec SDK [40].

Performance Metric. We use the F1 score to measure the detection or tracking accuracy of a single frame. F1 score is the harmonic mean of precision and recall, calculated as:

$$F1\ Score = 2\ (\frac{1}{Precision} + \frac{1}{Recall}), \tag{2.1}$$

where precision is the ratio of the number of true positives to the total number of objects detected by the scheme, and recall is the ratio of the number of true positives to the total number of objects in the ground truth. When the detected bounding box has the same label and sufficient spatial overlap with the ground truth box, this object will be identified as a true positive. We use intersection over union (IoU) to measure the spatial overlap.

$$IoU = \frac{area(Y \cap G)}{area(Y \cup G)}, \tag{2.2}$$

where Y is the detected area of objects from the scheme under evaluation and G is the ground truth area of objects. The value of IoU is set to 0.5 in this work.

Generally, a high F1 score indicates better detection and tracking accuracy, e.g., an object detector is considered to be perfect when its F1 score is 1. To calculate the false positives and false negatives, we need to know the ground truth, i.e., labels and accurate locations of objects in frame. In our experiment, we use the detection results of YOLOv3-704 as the ground truth. Since the large frame size 704x704 provides a high detection accuracy.

2.3.2 Experimental Results and Observations

YOLOv3 has two versions, i.e., a full version [4] and a lightweight version YOLOv3-tiny. YOLOv3-tiny is tailored for mobile devices by trading detection accuracy for short processing latency [8]. Based on our experiments on 13 video clips with 141213 frames, YOLOv3-tiny still cannot provide real-time (30 fps) video processing on mobile device. What is worse, its average F1 score per frame is as low as 0.3. Only 0.7% of frames achieve an F1 score higher than 0.7. The

Table 2.2: The latency of detection and tracking for one frame.

Component	Time (ms)
YOLOv3 detection latency	230-500
Good feature extraction	40
Tracking latency	7-20
Overlay latency	50

official website of YOLO [8] also shows that the full YOLOv3 can provide more than 55.6% higher accuracy than YOLOv3-tiny. Therefore, we use YOLOv3 not YOLOv3-tiny as the object detector in this work.

Detection Accuracy and Latency. Figure 2.1 depicts the detection latency and accuracy of YOLOv3 under different settings of frame sizes. In this experiment, we use YOLOv3 to process 4000 frames one by one. We record the F1 score and the processing latency of each frame. Both the average processing time and detection accuracy per frame increases as the frame size of DNN model increases. The processing time changes from 230 ms to 500 ms. The F1 score per frame augments from 0.62 to 0.88, if the largest frame size 608x608 among our settings is used.

Observation 1: Even with the lightest model setting (i.e., YOLOv3-320 in our implementation), the DNN-based object detector cannot process a video in real time. To capture the speed of mobile cameras (like 30 or 60 FPS), after processing one frame, the object detector must process the newest frame in the frame buffer that was captured by the camera. The frames between two processed frames will be skipped by the object detector. The objects detected by the first processed frame will be used as the reference by the tracking algorithm to track these objects until the object detector processes the next frame.

Observation 2: For one frame, a larger YOLOv3 frame size contributes to a higher detection accuracy, but suffers from a longer processing latency, and vice versa. If a large frame size is used, the detection accuracy is high, which provides the tracking algorithm with a high initial accuracy; but it also needs a

long processing time, which means the number of frames between the two YOLOv3 detected frames is large.

Tracking Accuracy. Between the two frames that are processed by the object detector, we track the objects in these frames based on the *Lucas-Kanade* method. To study the tracking accuracy, we use YOLOv3-608 to detect the objects in one frame, and then run the tracking algorithm to track these objects in the following frames. We do such an experiment to detect and track objects 10 times on two different videos respectively. Figure 2.2 shows the average tracking performance of these two videos. The content of Video1 changes faster than Video2. For both videos, the initial tracking accuracy is high, as the input size used by the YOLO model is relatively larger (608x608). However, the tracking accuracy drops below 0.5 after 9 frames for video1 and 27 frames for video2. The tracking accuracy of video1 degrades faster. Because it is hard to accurately estimate the positions of the detected objects in the following fast-changing frames. In addition, many new objects also appear in these frames.

Observation 3: The tracking accuracy drops quickly for the videos in which the content varies fast. Before the tracking accuracy drops to a low level (e.g., F1 score is below 0.5), we need to perform another object detection to calibrate the tracking performance to the initial tracking accuracy level. If the video content varies fast, a small frame size may be used in YOLOv3 detection, so that its processing time of one frame is short. Therefore, the YOLOv3 frame size determines both the initial tracking accuracy and the number of frames the tracking algorithm needs to track before the next calibration. It needs to be carefully set according to the online change rate of video content.

Tracking Latency. Table 2.2 shows the processing time of tracking one frame. It takes 40 ms on average to extract good feature points for tracking. We do not need to extract good features for each frame, but only the DNN detected frames. It takes 7 ms to 20 ms on average to track all the feature points from one frame to another. The latency depends on the number of objects and good features in the frame. Finally, for each frame, it takes 50 ms to find a good feature for each object and overlay the bounding boxes on top of all objects.

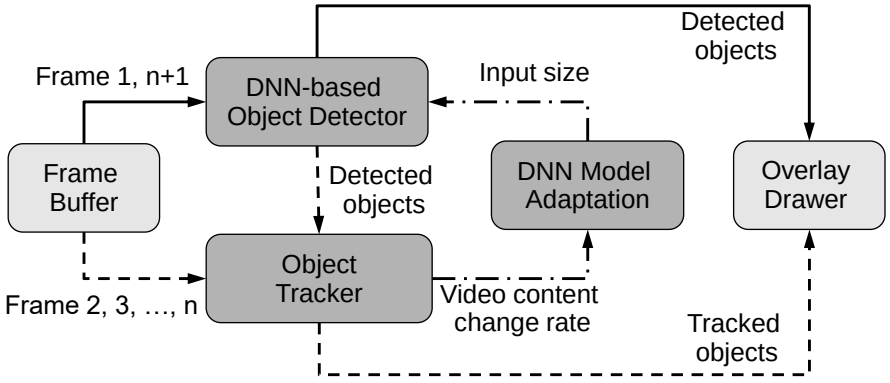


Figure 2.3: Architecture of *AdaVP*. Each frame is either processed by the object detector or by the object tracker. The object tracker takes the objects detected by the object detector as input. The object detector uses the results of the object tracker to calculate the video content change rate and further adapt its DNN model settings. Finally, the processed frame will be passed to the overlay drawer module to draw the bounding boxes and display the frame on screen.

Observation 4: The tracking latency of one frame is larger than the frame interval of normal camera video streams. To provide real-time video processing, we have to skip tracking in some frames to catch up with the frame capture speed of mobile cameras.

2.4 Design of *AdaVP*

In this section, we describe the design of *AdaVP*. After a brief overview of *AdaVP*'s architecture, we will introduce three key components of *AdaVP*.

2.4.1 System Overview

Figure 2.3 shows the system architecture of *AdaVP*. The frames taken by a mobile camera are first stored in a frame buffer. The task of *AdaVP* is to process the frames in the buffer one by one in real time, so that there is no frame accumulated in the frame buffer. After *AdaVP*'s processing, the objects of each frame will be identified. The results will be passed to the overlay drawer module

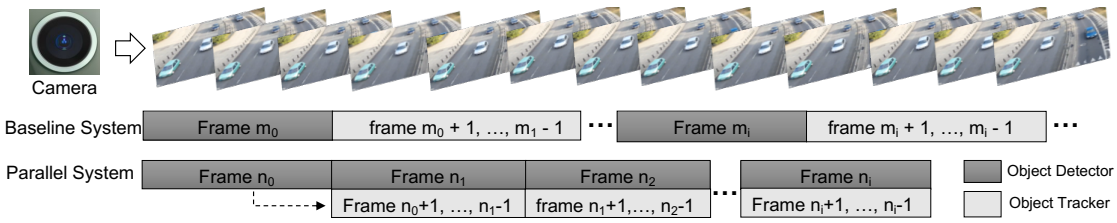


Figure 2.4: Two different video processing systems, i.e., a baseline system and the pipeline of parallel detection and tracking.

to draw the bounding boxes. The overlaid frames are the views with overlaid augmented objects, which will be finally displayed on the mobile screen.

AdaVP is mainly composed of three components, i.e., DNN-based object detector, object tracker and DNN model adaptation module. The object detector and the object tracker form a *parallel detection and tracking pipeline*. A frame in the frame buffer is either processed by the object detector or by the object tracker. When the object detector is processing a new frame, the object tracker handles all the accumulated frames before that frame in the buffer. In addition, the object detector uses the results of the object tracker to *calculate the video content change rate and adapts its DNN model settings*.

With the parallel detection and tracking pipeline, when the object detector accomplishes the processing of one frame, the object tracker takes the objects detected by detector as input to track these objects in the following frames. At the same time, the object detector fetches the newest frame from the buffer and starts detecting the objects in that frame. To support real-time video processing, when the object detector finishes the processing of the newly fetched frame, the object tracker needs to accomplish the processing of all frames before that frame (details can be found in 2.4.2).

To enable an efficient parallel detection and tracking, we adapt the DNN model settings to the change rate of video content. The DNN model adaptation module takes the intermediate results of the object tracker to calculate the change rate of video content. Based on *Observation 3*, if the video content change rate is high, the tracking accuracy degrades sharply. A model adaptation algorithm is developed to adapt the DNN model setting to the video content change rate.

2.4.2 Parallel detection and tracking pipeline

Figure 2.4 illustrates the workflow of our proposed *parallel detection and tracking pipeline* and a baseline system. The baseline system in Figure 2.4 is a simple implementation of the latest mobile video processing work, MARLIN [23]. To avoid offloading, the baseline system executes the object detector and object tracker on mobile devices sequentially. At the beginning, the object detector fetches $frame\ m_0$ from frame buffer and detects the objects in this frame. It takes hundreds of milliseconds for the DNN model to process one frame. During this process, k frames have been accumulated in the buffer. When the object detector completes its detection, it delivers the detection results to the object tracker. The latter will track the objects in the following j frames (from $frame\ m_0+1$ to $frame\ m_1-1$) to catch up. When the system detects significant scene changes, the object detector will be triggered to detect a new frame, the $frame\ m_1$. In this system, to catch up with the camera feed, j must be larger than k. If the system is required to achieve real-time processing, the object tracking time should be larger than the object detection time. However, if the system detects significant scene changes before the object tracker catches up, it will trigger the DNN object detector immediately and the system latency will be accumulated. The accumulated latency will further hurt overall accuracy.

To reduce the accumulated latency and improve processing accuracy, we propose MPDT (Mobile Parallel Detection and Tracking). It is a component in *AdaVP* that executes the object detector and object tracker in parallel. For MPDT, after the object detector delivers the detection result of $frame\ n_0$ to the object tracker, k frames have been accumulated in the buffer. The object detector fetches the newest frame from the buffer to do object detection, which is $frame\ n_1$. At the same time, the object tracker starts tracking the objects in the frames from $frame\ n_0+1$ to $frame\ n_1-1$ using the detection results (object locations, object labels) of $frame\ n_0$ received from the object detector. While the object detector is detecting $frame\ n_1$, the object tracker is tracking $frame\ n_0+1$ to $frame\ n_1-1$. In this way, the object detector and object tracker keep working in parallel, so the object detection time and object tracking time are basically same.

To implement MPDT, we use `multithreading` techniques. There are two technical problems. The first one is to prevent multiple threads to read/write the shared data at the same time. The second is the communication among multiple threads. We use three threads in our system, an object detector thread, an object tracker thread, and a main thread. The main thread is responsible for scheduling the other two threads and displaying images. The shared data among these threads are the frame buffer, detected results from the object detector and the display image. The shared data cannot be accessed by multiple threads at the same time, so we use `lock` to prevent data from being accessed at the same time. The threads also need to be notified when they can access the shared data. We use `event` to do the communication among different threads. To synchronize the object detector and object tracker threads, once the object detector fetches a new frame, the object tracker will cancel its tracking tasks after finishing the current task. But it does not display the current task. This is because the current task the object tracker is doing is on the prior frame to the frame is fetched by object detector, so if displays it, the displayed results will go backwards.

2.4.3 Object Tracker

MPDT needs to continuously track objects (detected by the object detector) across the frames in between two DNN executions. The number of frames to be handled could be large, e.g., 20 frames for YOLOv3 with a frame size of 608x608 (YOLOv3-608). To maintain the detection accuracy and achieve real-time performance, the object tracker needs to be accurate and lightweight. Typically, there are two steps in the object tracker, *Feature Extraction* and *Object Tracking*.

Feature Extraction. We first extract some features in the last DNN detected frame and then track these features in the following accumulated frames. By tracking these features, we can estimate the moving speed of the objects in the frame. For a frame, its features can be extracted using a feature detector and descriptor such as SIFT (Scale-invariant feature transform), SURF (Speeded-Up Robust Features), good features to track, FAST (Features from Accelerated Segment Test) and ORB (Oriented FAST and Rotated BRIEF) [41, 42, 16]. Generally,

the more accurate a feature descriptor is, the longer processing latency it needs. After evaluating the overall performance of all the above feature descriptors, we use the standard *good features to track* [16] method to extract the good feature points in the DNN detected frames.

(a) Frame 0, by detector (acc:0.79)

(b) Frame 8, by tracker(acc:0.62)

(c) Frame 14, by detector (acc:0.81)

(d) Frame 23, by tracker (acc:0.75)

(e) Frame 0, by detector (acc:1)

(f) Frame 8, by tracker (acc:0.83)

(g) Frame 14, by tracker (acc:0.73)

(h) Frame 23, by detector (acc:1)

Figure 2.5: Frame accuracy of MPDT using two different model settings (First row: MPDT-YOLOv3-320; second row: MPDT-YOLOv3-608).

Object Tracking. We then track the extracted good features in the following frames between two DNN detected frames using the Optical Flow Based Object Tracking method. Optical flow captures the pattern of apparent motion of objects, surfaces, and edges among frames. We use the well-known Lucas-Kanade [17] optical flow method to track good features in the following frames. Because only the feature points inside the bounding boxes detected by YOLOv3 are useful for the system to track objects. We only detect and extract feature points inside the bounding boxes. The feature points extracted from the same object should have similar motion vectors (distance and direction) between frames, so the motion vector of these feature points can describe the motion vector of the object.

There are usually multiple objects in one video frame. Different objects may have different motion vectors. To achieve high tracking performance, instead of calculating an average motion vector of all objects, we calculate the motion vector for each object. As a result, the tracking time per frame is related to the numbers

of objects in the frame, i.e., the more objects a frame has, the longer time it takes to find good features and calculate the motion vector for each object.

Tracking Frame Selection: The intermediate frames between two DNN detected frames will be fetched from the frame buffer into a temporary buffer. In this work, we call the time of one DNN detection execution as a detection or tracking cycle. From the motivation experiments and *observation 4*, we know that it is not practical to track all the frames in the temporary buffer, as the feature tracking and overlay drawing of one frame take more than 33 ms (if the frame rate of the video is 30 FPS). We can leverage the temporal correlation between adjacent frames in a video, i.e., adjacent frames usually contain similar content [22, 21] to select a certain number of frames at regular intervals in the temporary buffer to do object tracking.

To decide the number of frames to track, we need to know the processing time of the feature tracking per frame. However, as explained above, the tracking time per frame varies according to the number of objects in one frame. MPDT uses the prior tracking experience to find this number. We assume the number of objects in two adjacent tracking cycles does not change much. MPDT counts the number of frames that h_{t-1} were tracked and the total number of frames f_{t-1} in the buffer during the last cycle and calculates the tracking frame fraction $p = \frac{h_{t-1}}{f_{t-1}}$. Then it gets the total number of frames f_t in the buffer during the current cycle and estimates how many frames can be tracked during this cycle $h_t = p * f_t$. After getting the predicted h_t, the system knows how to select frames at regular intervals. The frames that are not selected by the tracker use the location and label of objects from the previous tracked or detected frame [36].

The object tracker uses the last DNN detected frame as the reference frame, including the labels and bounding box positions (locations) of all the objects in the reference frame. It will process the frames selected by the tracking frame selection method. It outputs the labels and locations of the objects in these tracked frames. A label is a class as which the DNN identified the object (e.g., person or dog). A bounding box represents the position of an object in the frame, which is represented by a 4-tuple vector (left, top, width, height).

Workflow of the Object Tracker. Putting all of these components together, the workflow of our object tracker is as follows: 1) Receive the detection results (labels and bounding box positions) of $frame\ n_0$ from the object detector and fetch the $frame\ n_0 + i$ selected by tracking frame selection method. 2) Extract all the good feature points inside all the bounding boxes in $frame\ n_0$ using the good features to track method. 3) Find one feature point for each bounding box. 4) Use the Lucas-Kanade method to estimate the optical flow from $frame\ n_0$ to $frame\ n_0 + i$. 5) Calculate a motion vector for each good feature between $frame\ n_0$ to $frame\ n_0 + i$ and use it to shift the old bounding box positions to the current positions in $frame\ n_0 + i$. 6) Select a new frame in the frame buffer to track.

2.4.4 Model Adaptation

Observations 1-3 in Section 2.3 indicate that different frame sizes of YOLOv3 provide different detection accuracy and tracking performance. In this section, we adjust the frame size to achieve better accuracy of the proposed parallel object detection and tracking (MPDT). After showing some preliminary results, we introduce the change rate detection of video content and the adaptation algorithm.

Preliminary experiment results

Figure 2.5 depicts an example of the detection accuracy of MPDT under two different DNN model settings (MPDT-YOLOv3-608 and MPDT-YOLOv3-320) on the same video clip. We show the results of 4 frames each.

- **Frame 0.** Both settings perform object detection for Frame 0. The detection accuracy of MPDT-YOLOv3-608 is 1, and the accuracy of MPDT-YOLOv3-320 is 0.79. Because the latter has 3 false positive cases, i.e., it identifies 2 cars as trucks and 1 truck as car.

- **Frame 8.** Both settings execute object tracking by taking the detection results of Frame 0 as reference. The tracking accuracy of MPDT-YOLOv3-320 drops to 0.62; whereas the accuracy of MPDT-YOLOv3-608 is still 0.83, as the latter has an initial detection accuracy of 1.

- **Frame 14.** MPDT-YOLOv3-320 fetched this frame to do detection and its accuracy improves to 0.81. MPDT-YOLOv3-608 is still doing tracking using the detection results from frame 0, and its accuracy drops to 0.73, because new vehicles appear.

- **Frame 23.** MPDT-YOLOv3-320 is performing tracking and its accuracy drops to 0.75. MPDT-YOLOv3-608 fetched this frame to do detection and its accuracy is calibrated to 1.

From the above example, we see that MPDT-YOLOv3-608 has a high initial detection accuracy, but its long detection latency results in a large number of frames to be tracked (i.e., low tracking accuracy for the last few frames). On the other hand, MPDT-YOLOv3-320 has a relatively lower initial detection accuracy, but it calibrates its tracking accuracy more frequently by performing light-weight object detection. For some frames, MPDT-YOLOv3-320 has a higher accuracy; but for the others, MPDT-YOLOv3-608's performance is better.

From the experiment results in Figure 2.2, we know that when the video content changes slowly, the tracking accuracy degrades slowly. In this situation, MPDT-YOLOv3-608 should be used to have a high initial detection accuracy. On the other hand, when the video scenes change fast, MPDT-YOLOv3-320 should be used, as the tracking accuracy drops fast and needs to be calibrated more frequently. Therefore, we propose to dynamically switch the frame size of YOLOv3 at runtime according to the video content change rate to achieve the best performance all the time. We first design a metric to measure the change rate of video content. Based on that, we develop an adaptation algorithm to adjust the frame size at runtime.

Video Content Change Rate

The metric to evaluate the video content change rate must be lightweight so that its computation will not impact the tracking and detection operation of the real-time system. We propose to leverage the intermediate result from tracking to measure the change rate of video content. By doing so, it almost adds no extra computation. We use the average motion velocity of all good features extracted

from two adjacent frames (i and $i+1$) as the change rate of video content. It is calculated as follows.

$$v_{i,i+1} = \frac{\left|\sum_{k=1}^{M} f_i^k(x,y) - f_j^k(x,y)\right|}{M * (j-i)} \tag{2.3}$$

where $f_i^k(x,y)$ and $f_j^k(x,y)$ are the pixel positions of the kth feature in the ith frame and the jth frame respectively. We have M features extracted from these frames. Since we skip some frames during tracking, i.e., $j - i \neq 1$, we normalize the motion velocity of features to the velocity between two adjacent frames by dividing the results by the number of frames between the ith and jth frame.

We use the pixel coordinates of features to calculate the motion velocity. For different cameras with different capture distance and angles, our motion velocity metric can measure how fast the objects move in the pixel coordinates of a frame. A high motion velocity means the video content is changing fast, i.e., the existing objects moves out of the frame fast and new objects may appear frequently.

DNN Model Setting Adaptation

We design a lightweight DNN model adaptation module to find the relationship between the motion velocity and different frame sizes (4 settings in our current implementation, i.e., 320x320, 416x416, 512x512 and 608x608). At runtime, the model adaptation module decides whether to switch to another frame size (model setting). Our adaptation scheme also works for selecting the right model not just model settings at runtime, as long as these DNN models have complementary detection accuracy and latency [36]. However, in order to use multiple DNN models simultaneously, we must pre-load these models, which requires large amounts of memory and cannot be supported by mobile devices. Therefore, we focus on switching to different model settings in this work. Different model settings have similar performance as different DNN models.

Generally speaking, a high motion velocity indicates high video content change rate and in turn sharp degradation of object tracking; as a result, a small frame size is necessary to keep the detection latency small and calibrate the object tracking more frequently. We assume the relationship between the motion velocity and the

4 frame sizes is linear, i.e., high velocity requires small frame size. To quantify the relationship, we need to find 3 velocity thresholds, i.e., v_1, v_2 and v_3. If $v \leq v_1$, the frame size 608x608 will be used. If $v_1 < v \leq v_2$, $v_2 < v \leq v_3$ or $v_3 \geq v$, the frame size 512x512, 416x416 or 320x320 will be used respectively.

It is a typical classification problem to find the three velocity thresholds. We first generate a large amount of training data and then use the training data to find the thresholds. In our current implementation, 32 videos, corresponding to 105205 frames, are used for finding the thresholds. The videos include 14 scenarios, including surveillance videos at highways, intersections, city streets, train stations, bus stations, and residential areas; car-mounted videos driving on the highway or around downtown; mobile camera videos about airplanes, boat, animals in the wild, racetrack, meeting room and skating rink.

To collect training data, we divide each video into a sequence of chunks. Each chunk is 1 second. We run MPDT to process each video with 4 frame sizes independently. We calculate an average detection accuracy and an average motion velocity every second. For each video, we obtain 4 sequences of pairs (detection accuracy and motion velocity). For each chunk, by comparing the detection accuracy from 4 frame sizes, we can find the frame size that provides the highest detection accuracy. Finally, we generate a training dataset that is composed of a large number of vectors (motion velocity and the best frame size). The best frame size is the label of the corresponding motion velocity. We then use the motion velocities and their labels to train a classification model to find the three thresholds under a certain frame size.

To use the adaptation module, we use the motion velocity measured in the current detection cycle to decide which frame size of YOLOv3 will be used for the next cycle. The motion velocity measured in the current detection cycle is measured based on the current frame size. In our experiments, we find that for the same chunk of the video, the motion velocity measured under different frame size settings are similar, but not exactly the same. It may be because the object bounding boxes detected by 4 frame sizes are not exactly the same. The feature points are extracted within the bounding boxes, thus the extracted feature points

are not exactly the same. To solve this problem, we find the three thresholds for each frame size. For online adaptation, we use the correct thresholds based on the frame size of current detection cycle. After training, the DNN model setting adaptation module can be used to guide the system to adapt to the change of video content. At runtime, this module takes the motion velocity and current DNN model setting as input and outputs the next DNN model setting.

It only takes 8.49×10^{-2} ms to extract the motion features from the object tracker and 1.89×10^{-2} ms to switch to a different DNN model setting. Compared to other components, our motion feature extraction time and DNN setting switching time is negligible, but we can improve the system performance significantly by using these two components. Since we change the setting of DNN model at runtime, the latency of *AdaVP* is not fixed. It varies from 200 ms to 470 ms (one DNN detection time subtract one frame time). This latency is inevitable in DNN-based video processing system.

2.5 Implementation

Framework: We use PyTorch [43] as the deep learning framework, because it supports dynamic computational graph building. As we know, a computational graph is normally built to represent some complex computation in DNN models. PyTorch can build and compute the computational graph at the same time, which is different from Tensorflow or Darknet. Tensorflow or Darknet builds the computational graph statically before computations start. Our system intends to be adaptive to the video content, and switches the DNN model settings dynamically during video processing according to the change of video content. The computational graph will be built according to the DNN model setting of the model. When the setting changes, the computational graph changes as well. So static computation graph building cannot meet the system requirements.

System implementation: We use multithreaded programming to implement our system. The *object detector* and The *object tracker* are implemented as two threads. `CUDA` is set up for the object detector thread, so it is able to use GPU

resources at runtime. The *Frame Buffer* is implemented by using `Queue` data structure. Both the object detector and object tracker have access to it. For the object detector, we get the `weights` file and `cfg` file from official darknet [8] website. For the object tracker, we use the `good features to track` function provided in OpenCV [44] to detect and extract good feature points. Because only the feature points inside the bounding boxes detected by YOLOv3 are useful to track objects, we use `mask` for the detected bounding boxes and only detect and extract feature points inside the masks. Compared to extracting the features across the whole image, only extracting features within masks saves computation and energy. We use the `calcOpticalFlowPyrLK` function to track these feature points in the following frames. To reduce latency, for each bounding box, we find one point inside it, calculate the motion vector of this point, and then use this vector to shift the bounding box.

Energy consumption: We use the shell file `Power_Monitor.sh` to get the power of the GPU, CPU, DDR and SoC of TX2. We record the power when the TX2 is running *AdaVP* or other baseline systems and the power when it does not run anything. The difference between these two records is the power of *AdaVP* or other baseline systems. Then we can calculate the energy consumption by multiplying power and running time.

Data storage: We save some data at runtime, including frame numbers, object class labels and object locations, motions of video from the object detector and object tracker. We use these data to train our DNN model setting adaptation module and compute the evaluation accuracy offline. Saving data at runtime adds extra computational overhead to our video processing system, which impacts the system performance slightly. The real performance of *AdaVP* should be better than that in our evaluation.

2.6 Evaluation

In this section, we conduct a variety of experiments to evaluate the performance of *AdaVP* comprehensively.

2.6.1 Experimental Setting

Dataset. To evaluate our system completely, we use the ImageNet and Videezy video datasets [18, 19], we also find some real-world public videos from YouTube [20]. Our dataset includes 45 indoor or outdoor videos that are recorded by static, moving or car-mounted cameras. These videos contain various scenarios with multiple objects (e.g., cars, trucks, trains, persons, airplanes, animals). Compared to live camera feed, these videos are much more challenging. Most of the videos are 30 FPS at a resolution of 1280 x 720 pixels. The length of each video ranges from 15 seconds to 34 minutes. These videos contain 246418 frames in total. We use 105205 frames to train the model adaptation module which explores the relationship between the DNN model settings and the motion of video content and 141213 frames to evaluate the system performance. We conduct all the evaluation experiments on the entire test dataset.

Baselines. We compare the performance of *AdaVP* with three baselines.

- **MPDT.** MPDT uses fixed DNN model settings for all types of videos all the time. We use 4 settings (320x320, 416x416, 512x512 and 608x608) to do the experiments. These four settings refer to YOLOv3-320, YOLOv3-416, YOLOv3-512 and YOLOv3-608. *AdaVP* switches the model setting of DNN model at runtime according to the changing rate of video content.

- **MARLIN.** MARLIN [23] is the latest work that executes object detection and tracking sequentially, without parallel computing. The object detector will be triggered when significant changes are detected by video content change detector. We implement the idea of MARLIN in our framework by the same DNN detector, object tracker and video content change detector as *AdaVP*. For the video content change detector, we conduct a set of experiments to find a motion velocity threshold that provides the best detection accuracy for MARLIN.

- **Without Tracking.** In this scheme, there is no object tracker to do tracking. We only use the DNN model to do detection. The DNN model is always

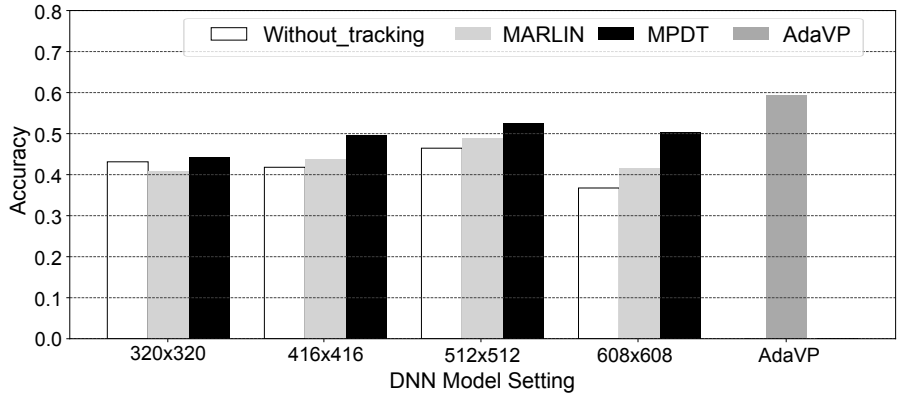


Figure 2.6: Performance comparison of *AdaVP* and baseline systems.

going to fetch the current video frame. For the skipped frames between two DNN executions, we use the detection result from the previous frame [36].

Detection Accuracy. We use the F1 score to measure the detection or tracking accuracy of a single frame. We use the percentage of frames with above a certain F1 score threshold to measure the accuracy of a video [21]. The F1 score threshold is set as 0.7 as default. For example, if the accuracy of a video is 0.6, it means there are 60% frames with F1 score higher than 0.7. For the video set, we use the average percentage per video as accuracy to demonstrate the evaluation.

2.6.2 Overall Performance

Figure 2.6 depicts the performance of *AdaVP* and the baseline systems on the whole testing dataset. From the experiments, we find *AdaVP* increases 20.4% to 43.9% accuracy compared to MARLIN and increases 13.4% to 34.1% accuracy compared to MPDT under different DNN model settings. The experimental results show that YOLOv3-512-based system achieves the best performance compared to other model settings for both MPDT and MARLIN.

Number of cycles per DNN model setting switching. Figure 2.7 presents the cumulative probability of number of cycles per DNN model setting switching. It is near 50% the system switches model setting after one cycle. Since *AdaVP* switches the DNN model setting at runtime, the duration of one cycle is not fixed, the number of frames within one cycle is not fixed (e.g., 10 to 25 frames per cycle).

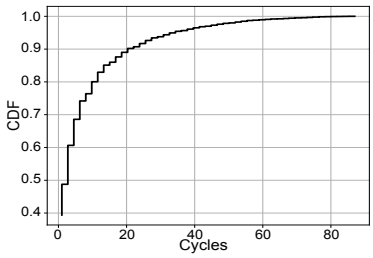


Figure 2.7: Cumulative probability of number of cycles per DNN model setting switching

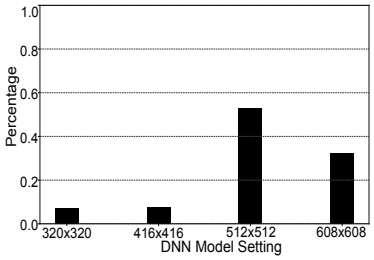


Figure 2.8: Trigger percentage of every DNN model setting from *AdaVP*

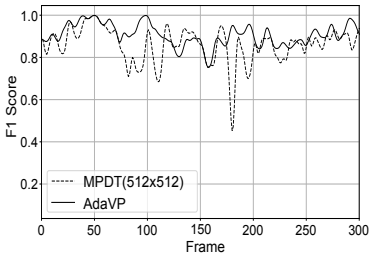


Figure 2.9: Frame accuracy comparison of *AdaVP* and MPDT-YOLOv3-512 (the best baseline)

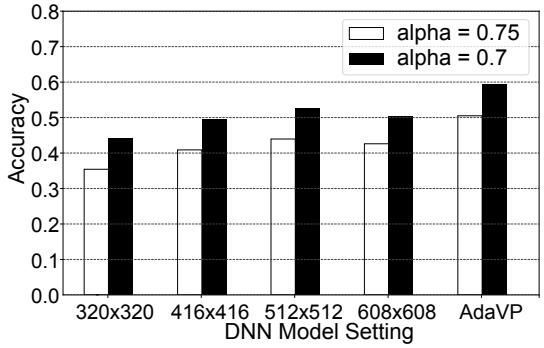


Figure 2.10: Performance comparison under different thresholds of F1 score

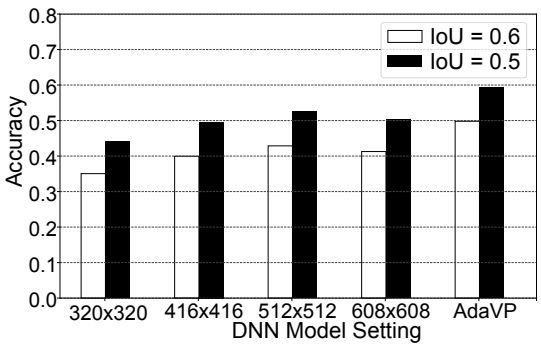


Figure 2.11: Performance comparison under different IoU value

For 90% of cases, the number of cycles per switching is below 20. There are 5% of cases that *AdaVP* switches to another model setting after 40 cycles. For these cases, the video content change detector does not detect significant change of the video content. Thus, *AdaVP* keeps using the same DNN model setting.

Usage of different model settings. Figure 2.8 shows the trigger percentage of different DNN model settings of *AdaVP*. From the experimental results, we know that all of the model settings have been triggered at runtime according to the video content change rate. The frame sizes of 512x512 and 608x608 are mostly being used. The usage of the other two model settings is around 10%.

Frame Accuracy Comparison. Figure 2.9 demonstrates the accuracy of *AdaVP* at the frame level. We use MPDT by YOLOv3-512 as a comparison since it is better than other model settings we used. Most of time, *AdaVP* achieves higher accuracy than MPDT-YOLOv3-512. Around frame 180, the detection accuracy of MPDT-YOLOv3-512 drops heavily. But the accuracy of *AdaVP* is still high, this is because the DNN model adaptation module decides not to use YOLOv3-512 for this cycle according to detected change of video content. In the long run, *AdaVP* combines the benefits of different settings and achieves higher accuracy.

2.6.3 Performance Gain of Parallel Detection and Tracking

From Figure 2.6, we also know MPDT outperforms MARLIN and without tracking scheme under each model setting. MPDT achieves 7.1% to 21.95% higher accuracy than MARLIN, MPDT achieves 2.3% to 37.3% higher accuracy than without tracking scheme under different model settings. This is because MPDT keeps doing object detection and tracking concurrently and calibrates the object tracking by running object detector at the maximum frequency. However, MARLIN does object detection and tracking sequentially, which is inefficient for complex and challenging video scenes.

Table 2.3: Comparison of energy consumption and accuracy

	AdaVP	MPDT-YOLOv3-320	MARLIN-YOLOv3-320	YOLOv3-tiny-320	YOLOv3-320	MPDT-YOLOv3-512	MARLIN-YOLOv3-512	YOLOv3-608
GPU								
$(w \cdot h)$	3.65	2.85	2.22	4.09	36.25	3.53	3.03	68.84
CPU								
$(w \cdot h)$	1.88	2.08	1.25	3.14	6.64	2.14	1.84	6.24
SoC								
$(w \cdot h)$	0.39	0.34	0.24	0.53	3.60	0.40	0.32	6.62
DDR								
$(w \cdot h)$	1.34	1.18	0.82	1.66	11.25	1.36	1.13	20.17
Total								
$(w \cdot h)$	7.26	6.45	4.53	9.42	57.74	7.43	6.32	101.87
Accuracy	0.59	0.44	0.41	0.07	0.57	0.52	0.48	0.89

2.6.4 Parameter Setting in *AdaVP*

F1 Score Threshold. Figure 2.10 presents the performance under different accuracy thresholds α at 30 FPS. As introduced in the experimental setting, we use the percentage of frames with certain F1 score threshold as the accuracy metric for a video. When we change the threshold α from 0.7 to 0.75, the accuracy is stricter. But *AdaVP* still outperforms the baseline system MPDT. When α is set as 0.75, *AdaVP* increases the accuracy of MPDT by 14.9% to 42.6%. The performance gain is even larger than the case when α is 0.7. From these two accuracy thresholds, we know *AdaVP* has more frames with higher accuracy than the baseline system.

IoU Threshold. We also compare the accuracy with different IoU values under 4 different model settings at 30 FPS. The widely-used IoU is 0.5 in the computer vision community. Here, we use a stricter IoU threshold, which is 0.6 for comparison. Higher IoU value means true positives are identified stricter. So, the F1 score per frame decreases and the overall accuracy decreases. Figure 2.11 reveals that *AdaVP* consistently outperforms the baseline when IoU is 0.6. It increases the accuracy by 16.1% to 41.8% compared to MPDT. The performance gain is even higher when IoU is 0.6, compared with the default 0.5.

2.6.5 Energy Consumption and Accuracy

Table 2.3 shows the energy consumption of different hardware components (GPU, CPU, SoC and DDR) from different video processing methods. We choose MPDT and MARLIN based on both YOLOV3-512 and YOLOv3-320, since they have the best real-time performance under the setting of 512x512, and they are most energy-efficient under 320x320. We choose YOLOv3-tiny-320 because it is almost real-time on TX2 without tracking or skipping any frames. For comparison, we also execute YOLOv3-320 and YOLOv3-608 continuously without frame skipping. If we do not consider latency and execute DNN for every frame, YOLOv3-320 is the most energy-efficient, and YOLOv3-608 can provide the highest accuracy for each frame among the model settings we used.

From Table 2.3, we found *AdaVP* increases by 20.4% accuracy compared to MARLIN-YOLOv3-512 at the cost of 14.9% more energy. This is because *AdaVP* targets at improving accuracy, but MARLIN focuses on energy efficiency. They have different design goals. We also found compared to the best baseline MPDT-YOLOv3-512, *AdaVP* increases the accuracy by 13.4% with 2.3% less energy consumption. *AdaVP* even achieves 3.5% more accuracy with 7.95x less energy compared to YOLOv3-320. We do not consider the 7x latency from YOLOv3-320 into the accuracy calculation. If we take the latency into consideration, the accuracy of YOLOv3-320 even much worse. Though YOLOv3-608 without frame skipping achieves the highest accuracy, it has 10.3x latency and consumes 14x more energy than *AdaVP*.

2.7 Conclusion

This chapter presents a continuous and real-time video processing system that incorporates object detection and object tracking on mobile devices without offloading. We develop MPDT, a parallel detection and tracking pipeline that executes object detection and tracking concurrently. On top of that, we design a DNN model setting adaptation module. This module switches the DNN model settings at runtime according to the detected video content changes.

Chapter 3

Driving Maneuver Anomaly Detection Based on Deep Auto-Encoder and Geographical Partitioning

This chapter presents *GeoDMA*, which processes the GPS data from multiple vehicles to detect anomalous driving maneuvers, such as rapid acceleration, sudden braking, and rapid swerving. First, an unsupervised deep auto-encoder is designed to learn a set of unique features from the normal historical GPS data of all drivers. We consider the temporal dependency of the driving data for individual drivers and the spatial correlation among different drivers. Second, to incorporate the peer dependency of drivers in local regions, we develop a geographical partitioning algorithm to partition a city into several sub-regions to do the driving anomaly detection. Specifically, we extend the vehicle-vehicle dependency to road-road dependency and formulate the geographical partitioning problem into an optimization problem. The objective of the optimization problem is to maximize the dependency of roads within each sub-region and minimize the dependency of roads between any two different sub-regions. Finally, we train a specific driving anomaly detection model for each sub-region and perform in-situ updating of these models

by incremental training. We implement *GeoDMA* in Pytorch and evaluate its performance using a large real-world GPS trajectories. The experiment results demonstrate that *GeoDMA* achieves up to 8.5% higher detection accuracy than the baseline methods.

3.1 Introduction

Although annual traffic fatalities have decreased from 42,702 in 2006 to 36,560 in 2018 in the United States, it is far from the "Toward Zero Deaths National Strategy" goal [45]. Most of the fatal crashes are the results of human error or negligence, such as speeding, distracted driving, or driving under the influence of drugs and alcohol [46].

Smartphone apps [47, 48] have been developed by companies like Google and Uber to detect anomalous driving maneuvers. They read instant driving sensing data (e.g., vehicle velocity and orientation) from smartphone sensors for driving anomaly detection. A recent work, pBEAM [49], applies conditional adversarial recurrent neural networks to develop a personalized model for each driver. It detects anomalies from instant sensor readings and captures the temporal dependency of driving data. However, these existing solutions are focused on individual drivers, the peer dependency of drivers is not considered, i.e., the correlation of driving maneuvers across vehicles, which is normally caused by local road structures, traffic conditions, and driving habits. Peer dependency depicts the similarity of driving maneuvers across vehicles, which is an important factor that needs to be considered in driving anomaly detection. For example, if most of the drivers in proximity show similar driving maneuvers during the same period, they should not be related to anomalous drivings.

In this chapter, we develop *GeoDMA*, which detects driving maneuver anomalies in real time by analyzing the instant GPS samples of drivers and taking the peer dependency of drivers into account. Driving maneuver anomalies, like aggressive driving, distracted driving, drowsy driving and driving under the influence, can be analyzed from vehicle-based features including the pressure exerted on the

brake, the fluctuation of vehicle speed, the angle of wheels, and the steering wheel movement [50, 51, 52, 53, 54]. We summarize the anomalous driving maneuvers that can be detected by *GeoDMA* in Table 3.1, including rapid acceleration, sudden braking, rapid swerving, frequent speed and direction changing. To effectively detect anomalous driving maneuvers in real time, *GeoDMA* takes instant GPS samples as input, and processes them by a deep auto-encoder model for driving maneuver anomaly detection. The deep auto-encoder model is designed by considering both the temporal dependency of each individual vehicle and peer dependency across different vehicles. In addition, to capture stronger vehicle-vehicle peer dependencies, a geographical region partitioning algorithm is developed to divide a city into different sub-regions, in each of which we train an auto-encoder-based driving anomaly detector, further improving the detection accuracy.

Our deep auto-encoder model is composed of an encoder and a decoder. The encoder takes the driving state transition feature vector calculated from the instant GPS samples of vehicles as input. A driving state is a combination of a speed-related operation and a direction-related operation. The driving speed and direction of a vehicle can be calculated from its GPS samples. The encoder is designed as a fully-connect layer and a Recurrent Neural Network (RNN) layer, which transforms an original driving state transition feature into a representation feature in a lower-dimensional latent space. The decoder is a fully-connected neural network with two fully-connected layers, which recovers the lower-dimensional representation feature to a reconstruction feature. It takes the representation feature as input and outputs the reconstructed feature. The reconstructed feature has the same dimension as the input of the encoder (the original driving state transition feature). The temporal dependency can be captured by the RNN, and peer dependency is incorporated as a regularizer of the deep auto-encoder model. *GeoDMA* uses the reconstruction error between the original feature vector and the reconstructed feature vector of the auto-encoder for anomaly detection, since anomalous driving data cannot be represented and reconstructed well by a model that was trained only by normal driving data. With unsupervised learning, the labeling of anomalous driving data is not needed when training the model. The

deep auto-encoder is trained to learn what the normal driving maneuvers look like. Any driving maneuvers that do not follow the distribution of normal driving maneuvers will be considered as anomalous driving maneuvers.

To maximize the effectiveness of spatial peer dependency in driving maneuver anomaly detection, *GeoDMA* incorporates the First Law of Geography [55], *"Everything is related to everything else, but near things are more related than distant things"*. Research in [56] also proposes that locality preservation of spatial data leads to better service. We consider the locality of peer dependency in our system. Since the road layouts and traffic conditions vary across a city, the driving behaviors exist common patterns in small regions with similar contextual features (i.e., traffic conditions and road structures). The peer dependency in a local area is stronger than that in the entire city. We develop a geographical partitioning scheme to perform driving anomaly detection in a geo-distributed way. Specially, we divide a city into several sub-regions. The data of the vehicles from the same sub-region will be collected together to develop the anomaly detection model for this sub-region. The detection accuracy of the model in each sub-region is expected to achieve higher accuracy than the centralized model that is trained by the data from the whole city.

Table 3.1: Anomalous Driving Maneuvers that can be Detected by *GeoDMA*.

Anomalous Driving Maneuvers	**Corresponding Anomaly in Driving Feature**
Rapid Acceleration	The transition duration from a driving state to a acceleration-related driving state is small
Sudden Braking	The transition duration from a driving state to a deceleration-related driving state is small
Rapid Swerving	The transition duration from a driving state to a bearing-related driving state is small
Frequent Acceleration or Deceleration	The transition probability from a driving state to a acceleration-related or deceleration-related state is high
Frequent Turns	The transition probability from a driving state to a bearing-related driving state is high

The objective of geographical region partitioning is to maximize the spatial dependency within the same sub-region and minimize the spatial dependency

among different sub-regions. The road segments have stronger dependency are supposed to be divided into the same sub-region and road segments have weaker dependency are supposed to be divided into different sub-regions. We first construct the road network of a city as an undirected and weighted graph. We treat the road segments of the city as vertices of the graph. There is an edge between two vertices if the roads they represent are geographically connected. We then formulate the geographical region partitioning problem as an optimization problem and solve the optimization problem by Normalized Cut (NCut) algorithm. In NCut algorithm, we extend the vehicle-vehicle spatial dependency to calculate the road-road spatial dependency. The weight of an edge is specifically designed to depict the spatial dependency between two vertices (road segments), which is measured by the similarity of the driving maneuvers that have happened on these road segments and similarity of the representation features of those driving maneuvers learned from the auto-encoder.

After obtaining the partitioning result, we train a specific anomaly detection model for each sub-region and perform in-situ updating by incremental training to further improve the detection accuracy. We implement *GeoDMA* in Pytorch platform [57] and conduct extensive experiments using the T-Drive dataset [58, 59], which contains vehicle GPS trajectories collected from a big city. Results from extensive experiments demonstrate that *GeoDMA* achieves up to 8.5% and 2.2% higher accuracy than the single-user approach and the centralized approach.

3.2 Related Work

Autoencoder-based Anomaly Detection. With the development of deep learning, some recent solutions apply deep autoencoders for anomaly detection [60, 61, 62, 63]. RDA [60] demonstrates the effectiveness of deep autoencoder-based anomaly detection on image datasets. MemAE [62] uses deep autoencoder for both image and video anomaly detection. [63] leverages a deep autoencoder for video anomaly detection. DAGMM [61] shows the effectiveness of deep autoencoder-based anomaly detection model on network intrusion detection, thyroid cancer

analysis and arrhythmia analysis. In this chapter, we apply a deep autoencoder for anomaly driving maneuver detection.

Driving Anomaly Detection. Smartphone-based methods have been developed by commercial companies, such as Google and Uber, to detect anomalous driving maneuvers [47, 48, 64, 65, 66]. CarSafe [47] detects drowsy and distracted drivers using cameras on smartphones. DriveSafe [48] uses the rear camera, the microphone, the inertial sensors, and the GPS of a smartphones to assess if a driver is drowsy or distracted. The camera-based methods have difficulty in achieving real-time performance. SenSpeed [66] utilizes the accelerometer and gyroscope of smartphones to acquire the instant vehicle speed. However, this solution assumes the alignment between the vehicle's coordinate system and the smartphone's coordinate system is fixed, which is hard to guarantee when the vehicle is driving. [64, 65] use GPS data, and SafeDrive [67] leverages the data from On-Board Diagnostics to identify driving anomalies. However, they rely on the sensor readings of a single driver, and the temporal and spatial correlations of the driving maneuvers from multiple drivers are ignored. Deep learning shows its efficiency in anomaly detection [68, 69, 70, 71]. In pBEAM [49], it leverages un unsupervised conditional adversarial RNN to train a single-user driving anomaly detection model. However, it ignores the peer dependency across vehicles. In this chapter, we consider the temporal correlation of driving maneuvers and peer dependency of driving maneuvers in our driving anomaly detection model.

Driving Behavior Modeling. In the literature, there are also some crowdsensing works on collecting data from multiple users for driving behavior modeling [54, 72, 73, 74]. Some of these studies process the data of individual vehicles independently, but the statistical correlations across vehicles in those studies were ignored. Some studies group behaviors of drivers, but do not detect the driving maneuvers of each vehicle. Most of the driving behavior modeling methods are offline analysis and assessment. PTARL [75] learns the representation of driving state transitions and uses the learned representation features to assess the historical driving score of each driver. It focuses on representation learning but not driving maneuver anomaly detection. However, *GeoDMA* is focused on real-time driving

anomaly detection, which takes both individual driving maneuvers and vehicle-vehicle peer dependency into account. Besides, the contextual information like weather, traffic, and road conditions in a small region tends to be similar.

Geographic Region Partition. The First Law of Geography proposed in 1970 [55] has been applied in many applications, like city management, urban traffic control, and transportation simulations [76, 77, 78, 79, 80, 81, 82, 83]. Graph-based road network partition methods are commonly used for region partitioning [76, 77, 84]. Following those solutions, we also leverage the road network graph to perform geographical partitioning. However, the graph we define is closely related to the driving maneuver anomaly detection, e.g., we define the weight of edges based on the spatial correlation of road segments. Our optimization objective is to maximize the spatial correlation in a sub-region and minimize the spatial correlation between any two sub-regions.

3.3 Motivation

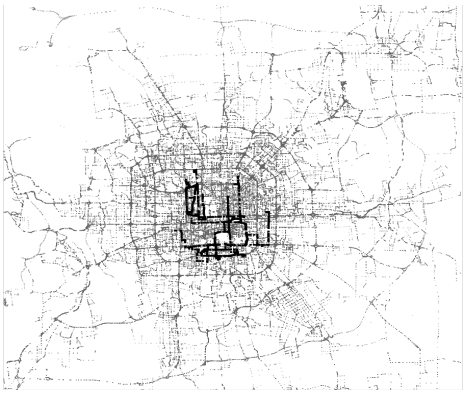


Figure 3.1: The trajectories of vehicles that receive low detection accuracy with singer-user models.

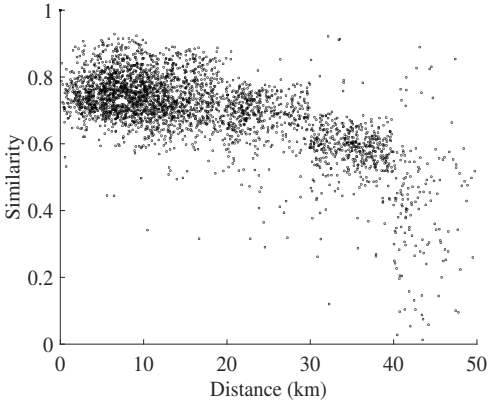


Figure 3.2: The relationship between vehicles' distance and vehicles' driving maneuver similarity.

In this section, we process the T-Drive dataset and demonstrate the motivation of two key components in *GeoDMA*. The detailed descriptions of the T-Drive dataset is in Section 3.6.2.

Peer dependency of multiple drivers. Peer dependency is considered in driving behavior analysis [75], which processes the historical GPS trajectories of vehicles to evaluate their driving history. We believe peer dependency is also useful for online driving anomaly detection. For example, if a driver exhibits stop-and-go (deceleration and acceleration) frequently, it is difficult to validate whether this kind of behavior is caused by the driver's anomalous driving maneuver, the road structure or the local traffic conditions. If a driver is observed to consistently show different driving maneuvers from the other drivers around him/her, it's reasonable to suspect that the driving maneuvers of that driver are abnormal. Here we do experiments to investigate the importance of peer dependency in anomaly detection. We use an auto-encoder (details in Section 3.4.2) to detect anomalous driving maneuvers and test the performance of single-user models. We use the driving data of each driver to train an auto-encoder detector independently and analyze the detection result of each single-user model. We find that the accuracy of some models can be lower than 0.7. Figure 3.1 highlights the trajectories of those vehicles that receive inaccurate driving anomaly detection at the same time slot. They are geographically close and driving in the downtown of the city. The driving maneuvers of these vehicles can be the reference to one another when developing the driving anomaly detection models. Based on this observation, we propose to develop a centralized model by incorporating peer dependency in the single-user model to improve its accuracy.

Locality of peer dependency. The centralized model is able to consider the vehicle-vehicle peer dependency across a city. However, the peer dependency should be considered in a fine-grained way. If a vehicle is driving far away from the other vehicle, they will has less peer dependency than some vehicles that are close to one another. It is hard for a centralized model developed for a big city to take all local features of the city into consideration. Figure 3.2 depicts the relationship between the similarity of driving features and their distances between any two drivers. The driving feature similarity is measured by Equation (3.5), which will be introduced in details in Section 3.4.2. We use 504 drivers to do the experiments and show the relationship of the driving feature similarities and the distances of

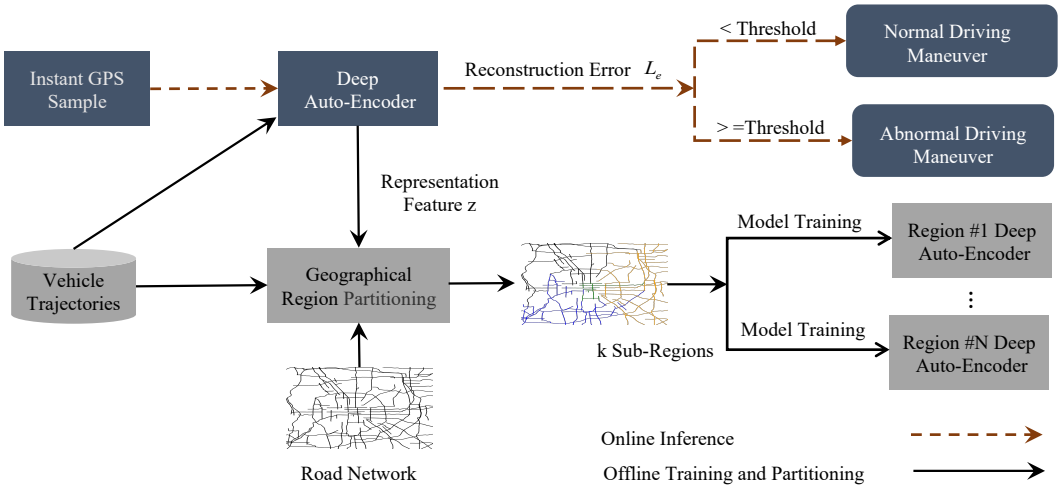


Figure 3.3: System architecture of *GeoDMA*.

all driver pairs. As shown in Figure 3.2, as the distance increases, the similarity of driving features decreases. The experiment result confirms with the First Law of Geography, which motivates us to partition the road segments in proximity into the same region for driving maneuver anomaly detection.

3.4 System Design of *GeoDMA*

In this section, we introduce the design of GeoDMA, including a brief overview, deep auto-encoder for driving anomaly detection, geographical partitioning, sub-region model and in-situ updating.

3.4.1 System Overview of *GeoDMA*

Figure 3.3 depicts the architecture of *GeoDMA*, which is mainly composed of two modules, i.e., a deep auto-encoder for driving anomaly detection and a geographical region partitioning algorithm.

The driving data from each driver will be first converted into the driving state transition vectors (Section 3.4.2). They are the input of our deep auto-encoder network. In the deep auto-encoder, we consider both individual vehicle temporal dependency and the vehicle-vehicle spatial dependency (Section 3.4.2). The deep

auto-encoder model is trained by normal driving maneuvers, it cannot reconstruct the anomalous driving data accurately. We use the reconstruction error of the model to perform driving maneuver anomaly detection (Section 3.4.2).

To further improve detection accuracy, we partition a city into multiple sub-regions by a geographical region partitioning algorithm. We formulate the region partitioning problem as a graph partitioning problem (Section 3.4.3). As shown in Figure 3.3, the inputs of region partitioning include the road network of a city, the vehicle trajectories and the representation features generated by the centralized auto-encoder. We develop an NCut algorithm to solve the optimization problem (Section 3.4.3). Finally, we train a specific anomaly detection model for each sub-region and update these models in-situ by incremental training (Section 3.4.4).

3.4.2 Auto-Encoder for Driving Maneuver Anomaly Detection

Figure 3.4 presents the inference workflow of our driving anomaly detection model. During each time window, the vehicle GPS data is fed into the vector calculator to generate the original driving feature vector x (state transition vector). Then the auto-encoder verifies x is normal or anomalous. Next, we will introduce the vector calculator, the deep auto-encoder, and the driving anomaly detection model in detail.

Vector Calculator

In the context of driving maneuver anomaly detection, the driving state of a vehicle can be described by its moving speed and direction. The speed-related driving operations include acceleration, deceleration, and driving in a constant speed. The direction-related operations include turning left, turning right, and going straight. A speed-related operation and a direction-related operation constitute a driving state. There can be nine driving states: S_1 (acceleration, turning left), S_2 (acceleration, turning right), S_3 (acceleration, going straight), S_4 (deceleration, turning left), S_5 (deceleration, turning right), S_6 (deceleration, going straight), S_7

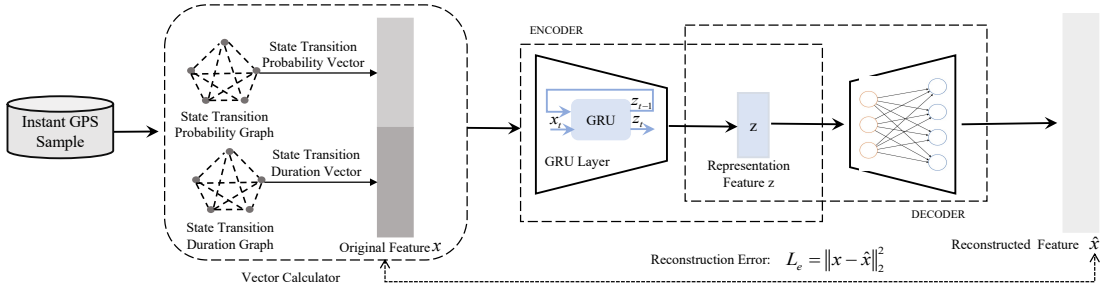


Figure 3.4: The workflow of deep auto-encoder. The vehicle GPS data is preprocessed by vector calculator. It generates the original feature vector x. The vector x is represented by combining the state transition probability vector and the state transition duration vector. This feature x is mapped to a representation feature z by the encoder. Then z is reconstructed as $\hat{x}$ by the decoder. Finally, we leverage the reconstruction error L_e as the criterion to detect anomaly.

(constant speed, turning left), S_8 (constant speed, turning right) and S_9 (constant speed, going straight). The details about speed and direction calculation from GPS data are introduced by Equation (3.14) and Equation (3.15).

When a vehicle is moving, the driving state of a vehicle usually changes over time. A sequence of driving states of a vehicle during a time window can be obtained. For instance, [S_2, S_2, S_5, ..., S_8]. Such a time-varying sequence can be summarized as a state transition graph. As shown in the vector calculator of Figure 3.4, the nodes of the graph are driving states, there are nine nodes in the graph and they are fully connected, we use five nodes in the figure for simplicity. The weights of edges depict the relations between any two states [67]. In this chapter, the weights are constructed from two aspects, the state transition probability and state transition duration between two driving states [75]. The value of the weights are normalized between 0 and 1. The driving state transition probability is the frequency a driver drives from a state to another state during a time window. The driving state transition duration is how long a driver takes to response from one driving state to another. For example, if the transition from <constant speed, going straight> to <acceleration, going straight> is small, it indicates this is a rapid acceleration anomaly.

During each time window, we can get a state transition probability graph and a state transition duration graph. The feature of the driving maneuvers are captured from these two graphs. The adjacent matrices of these two graphs can be flattened into two vectors, the state transition probability vector and the state transition duration vector. As shown in Figure 3.4, the combination of these two vectors is the original feature x, which is the output of the vector calculator and the input of the deep auto-encoder. Since there are nine states, there can be 81 state transitions (including self to self) for both probability graph and duration graph, the dimension of x is 162. The state transition vectors that differ significantly from the normal state transition vectors will be detected as anomalies.

Deep Auto-Encoder

As shown in Figure 3.4, the encoder projects the original feature vector x into a lower-dimensional feature, *i.e.*, representation feature z. The decoder reconstructs z to $\hat{x}$. The more similar the x and the $\hat{x}$, the more accurate the model is. In this work, the encoder is a fully-connected layer and a Gated Recurrent Unit (GRU) [85] layer. The GRU can capture the temporal dependency of the input features. The decoder is a fully-connected neural network with two fully-connected layers. The auto-encoder is updated as:

$$\mathbf{z}_i^t = GRU(\mathbf{z}_i^{t-1}, \mathbf{x}_i^t) \tag{3.1}$$

$$\hat{\mathbf{x}}_i^t = \mathbf{D}_\theta(\mathbf{z}_i^t) \tag{3.2}$$

where the $\mathbf{D}_\theta$ in Equation (3.2) represents the decoder network and θ is its weight.

The standard method to train an auto-encoder is to minimize the reconstruction error L_e between x_i^t and $\hat{\mathbf{x}}_i^t$.

$$L_e = \|\mathbf{x}_i^t - \hat{\mathbf{x}}_i^t\|_2^2 \tag{3.3}$$

If the trajectories of two drivers exhibit similar original driving features x, the representation features z of them that are generated from the encoder should be similar. Therefore, the vehicle-vehicle peer dependency can be integrated into the loss function. The peer dependency can be modeled as a regularizer L_r.

$$L_r = s_{i,j}^t \cdot \|\mathbf{z}_i^t - \mathbf{z}_j^t\|_2^2 \tag{3.4}$$

$$s_{i,j}^t = cos(\mathbf{x}_i^t, \mathbf{x}_j^t) \tag{3.5}$$

where $s_{i,j}^t$ in Equation (3.4) is the cosine similarity between the original driving feature vectors x_i^t and x_j^t that are generated from driver d_i and driver d_j during time window t, which is calculated by Equation (3.5). The z_i^t and z_j^t are the representation features of x_i^t and x_j^t during time window t. To incorporate the peer dependency, the loss function L is defined as:

$$L = \arg\min \sum_{t \in \mathcal{T}} (\sum_{d_i \in \mathcal{D}} L_e + \alpha \cdot \sum_{d_j \in \mathcal{D}, d_i \neq d_j} L_r) \tag{3.6}$$

where α is the hyperparameter to control the regularizer L_r. The d_i and d_j are any two drivers in the driver set D. The regularizer L_r is used to guide the training of the auto-encoder.

Driving Maneuver Anomaly Detection

Instead of training a multicalss classifier using labeled normal and anomalous data by supervised learning, we leverage deep auto-encoder to train a one class classifier. One class classifier is an unsupervised learning process and has been widely used in the state-of-the-arts for anomaly detection [86, 87]. We use one class classifier is because it is not easy to collect, predefine and manually labelling various types of anomalies to train a supervised muilticlass classifier to detect different types of anomalies. More importantly, if any new anomalies occur in the future, the anomalies need to be redefined and the classifier needs to be retained. By using unsupervised one class classifier, we address the above problems. During training phase, the deep auto-encoder model is trained only on the normal driving data. During inference phase, the anomalies can be detected as long as the model memorized the feature of normal data well. That is because anomalies cannot be reconstructed accurately by the model trained only on normal data [60, 88, 89]. As shown in Figure 3.4, we leverage the reconstruction error L_e as the criterion to detect the anomalies.

During inference phase, the model is expected to produce higher reconstruction error for the driving anomalies than the normal ones. The detection principle is

shown as follows:

$$x = \begin{cases} Normal\ Input, & if\ L_e(x, \hat{x}) < \tau \\ Anomalies, & Otherwise \end{cases} \tag{3.7}$$

where τ is the largest reconstruction error of normal data. It is predefined for online inference [86, 87]. During inference, given a sample input x, if the reconstruction error L_e between input vector x and its reconstructed vector $\hat{x}$ is lower than the threshold, it indicates that the well-trained model can reconstruct the input accurately. The input x will be considered as a normal driving maneuver. Otherwise, the input x will be detected as an anomalous driving maneuver.

3.4.3 Geographical Partitioning for Anomaly Detection

In this work, we propose to develop the driving anomaly detection model in a geo-distributed way. We first partition a city into several sub-regions to get stronger spatial peer dependency, and then develop a specific model for each sub-region. The challenge here is how to partition the region to the full extent of spatial peer dependency. To tackle the challenge, we develop a geographical region partition algorithm . The design objective is to improve the accuracy of driving maneuver anomaly detection in each sub-region. We extend the vehicle-vehicle dependency to the road-road dependency. Three principles are considered in the partition algorithm. 1) The road segments with stronger correlations should be partitioned into the same sub-region. 2) The road segments with weaker correlations should be partitioned into different sub-regions. 3) The road segments that are partitioned into the same sub-region should be geographically connected.

Problem Formulation

We represent the road network of a city as a graph $G = (V, E)$, where $V = (v_1, v_2, ..., v_n)$ represents all road segments, and $E = \{e(v_m, v_n)\}_{v_m, v_n \in V}$ describes the relationships among these road segments. If two road segments are geographically connected, there will be an edge connecting these two vertices. The weight of an edge, $w(v_m, v_n)$ of $e(v_m, v_n)$, qualifies the correlation between two vertices

v_m and v_n. We extend the vehicle-vehicle dependency to road-road dependency to get the correlation between two road segments. The similarity of driving maneuvers that happened on two road segments is used to approximate the correlation between them. We formulate this similarity by Equation (3.8).

$$w(v_m, v_n) = \sum_{t \in \mathcal{T}} \frac{1}{K_m^t K_n^t} \sum_{d_i \in \mathcal{D}_m, d_j \in \mathcal{D}_n} \frac{s_{d_i,d_j}^t}{\|\mathbf{z}_{d_i}^t - \mathbf{z}_{d_j}^t\|_2^2} \tag{3.8}$$

where $\mathcal{T}$ represents all time windows, v_m and v_n denote two road segments, K_m^t and K_n^t represent the number of drivers on these two road segments at time window t, $\mathcal{D}_m$ and $\mathcal{D}_n$ are two sets of drivers on these two road segments at time t, s_{d_i,d_j}^t is the similarity of original driving maneuvers between driver d_i and driver d_j at time t. s_{d_i,d_j}^t is acquired by Equation (3.5), its value range is between 0 to 1. The higher the similarity, the higher the s_{d_i,d_j}^t. The d_i is the driver on road segment v_m and the d_j is the driver on road segment v_n. The $z_{d_i}^t$ and $z_{d_j}^t$ are the representation features of the driving maneuvers that are generated from the encoder for driver d_i and d_j at time t. If the original driving maneuvers (x) of two drivers are similar, the s_{d_i,d_j}^t is higher, and the $\|z_{d_i}^t - z_{d_j}^t\|_2^2$ is lower, the ratio of them is higher. At each time window, we calculate this ratio from all of driver pairs on the two road segments v_m and v_n and get the average ratio of all pairs. We use multiple time windows to do the experiments and use the average value from these time windows as the correlation of these two road segments. The underlying rationale for the Equation (3.8) is that two road segments have strong correlation when the original driving maneuvers (x) on these two road segments are similar and their representation features (z) are also similar. A higher $w(v_m, v_n)$ means a higher correlation between the road segments v_m and v_n.

Based on Equation (3.8), we can obtain the weight of each edge in G. The objective of graph partitioning is to divide vertices $v_x \in V$ into k subsets V_1, V_2, ... , V_k. Each subset V_i corresponds to a set of the road segments that are partitioned into the same sub-region. According to the above partition principles, the spatial correlation within one subset V_i should be strong, and the spatial correlation among different sub-regions (V_i and V_k) should be weak. We formulate

it as an optimization problem. Its objective is shown as follows:

$$max \ \frac{1}{M}\sum_{v_m,v_n\in V_i} w(v_m,v_n) - \frac{1}{N}\sum_{v_m\in V_i, v_l\in V_k} w(v_m,v_l), \quad v_m \neq v_n \neq v_l, V_i \neq V_k \tag{3.9}$$

where V_i and V_k are two vertex subsets in the graph $G = (V, E)$, vertex v_m and v_n belong to the same subset V_i, v_l is a vertex in subset V_k. $w(v_m, v_n)$ is the weight of edge $e(v_m, v_n)$, $w(v_m, v_l)$ is the weight of edge $e(v_m, v_l)$. The edge $e(v_m, v_l)$ connects subset V_i and V_k. M is the total number of edge within the same subset V_i and N is the total number of edges that connect subset V_i and V_k. The first part of the objective is the average weight of edges within a subset V_i, the second part is the average weight of edges that connect two subsets V_i and V_k. The objective is to maximize the average weight difference across different subsets.

Normalized Cut Algorithm

Graph partitioning has been proven to be an NP-hard problem [77, 90, 91]. Partitioning the graph into $k = 2$ parts is already an NP-hard problem. Solutions are generally derived using heuristic algorithms. Spectral clustering is one of the most common graph partitioning methods by grouping graph vertices. MinCut, RatioCut and Ncut are three common spectral clustering algorithms. MinCut tries to find the minimum weight of edges that connect different sub-graphs. But in many cases, MinCut simply separates a vertex in the graph from the rest of the vertices, which is not what we want. A reasonable solution should consider that there are as many vertices as possible in each sub-graph. RatioCut and Ncut were proposed to solve the limitation of MinCut. In RatioCut, the size of a sub-graph is measured by the number of vertices in this sub-graph, while in Ncut the size of a sub-graph is measured by the weights of edges in this sub-graph. Since maximizing the number of vertices in each sub-graph does not necessarily mean the total weights of this sub-graph is large, cutting the graph based on the weights is more in line with our goal. Ncut is a normalized spectral clustering algorithm, while RatioCut is an unnormalized one. In general, Ncut is better than RatioCut [92]. Hence, we leverage Ncut [93] to solve our optimization problem.

The normalized cut criterion measures the total similarity within each subset of the graph and the total dissimilarity among different subsets of the graph.

It is important to construct a similarity function among different vertices when using Ncut. The vertices which are defined as "similar" by the similarity function should be closely related in the application. In our scenario, the correlation between any two vertices does not only depend on how similar these two vertices are. It first depends on the physical geographical connection, because vertices are the road segments. The two road segments that are supposed to be divided into the same sub-region should be at first connected with each other. We use the correlation function (Equation (3.8)) as the similarity function of vertices. Suppose the vertex set V and edge set E in the graph $G = (V, E)$ can be partitioned into two subsets V_i and V_k, $V_i \cup V_k = V$, $V_i \cap V_k = \varnothing$ by removing the edges that connect these two subsets. The degree of similarity between these two parts is defined as $cut(V_i, V_k)$. It is the total weights of the edges that connect these two subsets.

$$cut(V_i, V_k) = \sum_{v_i \in V_i, v_k \in V_k} w(v_m, v_l) \tag{3.10}$$

where v_m is the vertex in subset V_i, v_l is the vertex in subset V_k. Finding the minimum cuts is the objective of graph partitioning. It is designed by considering both the total disassociation (Ncut) between two subsets and the total association (Nassoc) within each subset. The Ncut and Nassoc are defined as follows:

$$Ncut(V_i, V_k) = \frac{cut(V_i, V_k)}{cut(V_i, V)} + \frac{cut(V_i, V_k)}{cut(V_k, V)} \tag{3.11}$$

$$Nassoc(V_i, V_k) = \frac{cut(V_i, V_i)}{cut(V_i, V)} + \frac{cut(V_k, V_k)}{cut(V_k, V)} \tag{3.12}$$

Importantly, the Ncut and Nassoc are closely related to each other, it can be inferred that:

$$\begin{aligned} Ncut(V_i, V_k) &= \frac{cut(V_i, V_k)}{cut(V_i, V)} + \frac{cut(V_i, V_k)}{cut(V_k, V)} \\ &= \frac{cut(V_i, V) - cut(V_i, V_i)}{cut(V_i, V)} + \frac{cut(V_k, V) - cut(V_k, V_k)}{cut(V_i, V)} \\ &= 2 - \frac{cut(V_i, V_i)}{cut(V_i, V)} + \frac{cut(V_k, V_k)}{cut(V_k, V)} \\ &= 2 - Nassoc(V_i, V_k) \end{aligned} \tag{3.13}$$

So minimizing the dissociation (Ncut) between the sub-graphs and maximizing the association (Nassoc) within each sub-graph can be reached simultaneously.

The solution can be approximated as follows:

1. Build a weighted graph $G = (V, E, w)$ from the road network, compute the weight of each edge according to Equation (3.8).

2. Compute the diagonal matrix D of the graph. The diagonal value $d_i = \sum_j w(i, j)$. Then get the symmetrical matrix W of the graph with $W(i, j) = w_{ij}$.

3. Solve the equivalent eigenvalue system $(D - W)y = \lambda Dy$ for eigenvectors, obtain the second smallest eigenvalue.

4. Use the eigenvector with the second smallest eigenvalue to bipartition the graph.

5. Recursively partition the subgraphs until the desired number of clusters is reached.

3.4.4 Sub-Region Model and In-Situ Updating

The subsets partitioned by our Ncut are the sub-regions in the road network. We then train a specific detection model for each sub-region by using the vehicle trajectories that are just collected from this sub-region and perform in-situ updating of these models in each sub-region periodically.

Model Training and Inference for Sub-Regions

Since the models for the sub-regions need to be trained by using the driving data collected from each sub-region, we first map the data in the dataset into different sub-regions. To implement this, we add the <RoadID> attribute to every sample of the GPS data. By grouping the GPS data with the same RoadID value, the data will be mapped into different sub-regions. Besides, we want to find the drivers with driving behaviors in a set of consecutive time windows to train the model, so

we filter the drivers that do not meet the requirements. The number of sub-regions k to be partitioned is a parameter of the Ncut algorithm. We try different values of k and find the one with the best performance. Since we divide the whole city into several sub-regions, the driving maneuvers that happen within each sub-region become the peer dependencies when training the models. The models trained using the driving data within each sub-region learn the distribution of normal driving maneuvers in corresponding sub-regions. The workflow of the models for the sub-regions is the same as the base model we introduced in 3.4.2. After training, we use the threshold of reconstruction error of these models to classify the driving maneuvers. The sub-region models are trained with different subsets of the data, the reconstruction error threshold for these models are not exactly the same.

In-Situ Updating of the Sub-Region Models

To improve the trustworthiness of the geo-distributed driving maneuver anomaly detection system, we will perform incremental learning to update all the driving maneuver detection models in-situ. The models will be updated offline after each server collects a reasonable amount of data or simply after a fixed time period T. In our current implementation, T is set to 25 time windows, corresponding to 54 minutes. We update the models twice in our implementation. Each server uses the data which detects as the normal driving data by itself to update its driving maneuver detection model.

3.5 Implementation

Deep Learning Models. *GeoDMA* is implemented in Pytorch [57]. The encoder contains one fully-connected (FC) layer and one GRU layer. The FC layer has 40 neurons. The GRU has 20 neurons. The decoder consists of two FC layers, which has 40 and 162 neurons in the first and second layer. The activation function is a sigmoid function. The performance comparison of different representation sizes can be found in Section 3.6.5. Besides, we use an SGD optimizer to optimize our models, as it has shown good performance in modeling sequential data.

The learning rate is set to be 0.1. The batch size is 128. The number of epochs is 500. The α in Equation (3.6) is 0.01. The models are trained on an Alianware Aurora R7, which contains one Intel Core i5 8400 CPU (6-core) and one NVIDIA GeForce GTX 1070 GPU (8GB memory). Sklearn python package is used to implement the evaluation metrics.

Graph Partitioning. It is non-trivial to construct a graph based on the road network for a big city. To implement it, the road network information is needed. We get the *GEOJSON* file of road network from OpenStreetMap [94], which is a kind of file format for representing geodata. After parsing the file, there are 43 types of roads, including 138,155 road segments. We visualize the road information and filter the road segments that do not contain any driving data, such as pedestrian ways, cycle ways and so on. There are many road segments that only contain several pair of coordinates, which indicates these roads are short. We first combine these short road segments into longer ones and then construct the graph using the longer road segments. We assign a *RoadID* to each road and find out the connected roads by comparing coordinates. After getting the geographical connection of each road, the graph $G = (V, E)$ can be constructed and the weight of each edge can be computed. Finally, we implement Ncut by using Sklearn package to partition the graph.

3.6 Evaluation

We conduct a variety of experiments to evaluate the performance of GeoDMA, including overall performance, performance under different scenarios, effectiveness of region partitioning, in-situ model updating, and execution efficiency.

3.6.1 Experimental Setting

We compare the performance of *GeoDMA* with two baselines over a large vehicle GPS trajectories dataset. We use 80% of the data to do training, and 20% of the data to do testing [27, 95, 96]. We set the number of sub-regions to 4, the size of the representation feature to 20. We use these settings by default in the following

experiments. In Section 3.6.5, we present experimental results demonstrating how we fine-tuned these parameters.

Performance Metrics.

We use the F1 Score [97] and AUC to do the performance evaluation.

Baselines.

We compare the performance of *GeoDMA* with two baselines.

- **Centralized Model.** The centralized model is a simple version of *GeoDMA*. It uses all training data to train a general model, without geographical partitioning. The general model is based on an auto-encoder that takes both temporal dependency and peer dependency into account.
- **Single-User Model.** We implement a similar version of the latest personalized driving anomaly system pBEAM [49] based on deep auto-encoder architecture. The single-user model is trained using the driving data that is collected from each individual driver without considering peer dependency and geographical partitioning.

3.6.2 Dataset and Data Processing

We conduct data-driven evaluation on a real-world dataset, T-Drive [58, 59]. The data format in the dataset is (Driver ID d, Date Time t, Longitude λ, Latitude φ). We first use a map matching algorithm [98] to map the GPS locations to corresponding locations on the road network. We then interpolate the missing data after map matching. We next pre-process the data, calculate the driving speed and direction of the vehicles and simulate anomalous data for evaluation.

Map Matching. The T-Drive dataset includes the GPS trajectories collected from 10,357 drivers. In the dataset, its sampling rate of trajectories is uneven, the average sampling interval is about 177 seconds with a distance of about 623 meters. When the sampling interval is large, there are only a few data samples during their corresponding time windows. That is not good to derive the driving

features. Therefore, it may not fully show the driving maneuvers of drivers during these time windows. In this chapter, we leverage a map matching algorithm to solve this drawback. The map matching algorithm maps the GPS trajectories to the road network [98, 99]. After map matching, the data can be augmented and is much more denser than the original data. The average sampling rate is about 9 seconds. Practically, we apply a standard map matching algorithm based on Hidden Markov Model [98] to map the driving trajectories in T-Drive dataset onto the road network. The map matching algorithm is implemented in Java.

Data Interpolation. After map matching, we find some pieces of data do not have the timestamp attribute. To solve this problem, we use an interpolation method to fill up the lost timestamps. It assumes that all the drivers are driving with a constant speed between two consecutive timestamps. This assumption is in line with normal driving maneuvers. For example, point A, B, C, D, E are five samples on a route, only A and E have time values. We first calculate the average speed v_{AE} of the vehicle from A to E, it is $v_{AE} = \frac{d_{AE}}{t_E - t_A}$, where d_{AE} is the driving distance from A to E. We assume the driver drives in a constant speed, so $t_B = t_A + \frac{d_{AB}}{v_{AE}}$. Thus, the time information of all the samples can be obtained. And all the drivers are driving in a constant speed most of the time. Therefore, we assume all the driving maneuvers derived from the dataset after map matching and data interpolation are normal.

Data Pre-processing. To make the data in the dataset closer to the normal driving data, we filter the bad data from the dataset. After filtering, there are 504 drivers left in the dataset. The driving time of each driver is 404 minutes. We then use a sliding time window to construct the two driving state transition graphs. The default size of sliding time window is 30 minutes. For example, the first time window is 0-30 minutes, the second one is 1-31 minutes, the third one is 2-32 minutes and so on. The driving state transition graph of drivers during each time window are the inputs to anomaly detection models. Since the total driving time of each driver is 404 minutes, we divide it into 375 sliding time windows. Each driver generates one driving feature vector during each window. The total sample size of normal data is 189, 000.

Table 3.2: Data distribution of normal data and anomalous data

	Value	Normal	Anomaly 1 acc (1, 1)	Anomaly 2 angle (1, 1)	Anomaly 3 acc (0.2, 0.04) angle (0.2, 0.04)
Acceleration (m/s^2)	< -0.5	21.5%	17.0%	21.5%	18.8%
	$[-0.5, 0.5]$	57.3%	19.5%	57.3%	52.3%
	> 0.5	21.2%	63.5%	21.2%	28.9%
Bearing Angle (radian)	< -1	8.1%	8.1%	7.1%	13.5%
	$[-1, 1]$	84.3%	84.3%	43.4%	68.9%
	> 1	7.6%	7.6%	49.5%	17.6%

Driving Speed and Direction. In the T-Drive dataset, given two data samples ($d_1,\ t_1,\ \lambda_1,\ \varphi_1$) and ($d_1,\ t_2,\ \lambda_2,\ \varphi_2$), the driving distance $D_{1,2}$ of driver d_1 from t_1 to t_2 can be calculated by [100]:

$$D_{1,2} = atan2(\sqrt{sin^2(\frac{\Delta\varphi}{2}) + cos\varphi_1 \cdot cos\varphi_2 \cdot sin^2(\frac{\Delta\lambda}{2})}, \sqrt{1 - sin^2(\frac{\Delta\lambda}{2}) - cos\lambda_1 \cdot cos\lambda_2 \cdot sin^2(\frac{\Delta\lambda}{2})}) \cdot 2R \tag{3.14}$$

where $\Delta\varphi = \varphi_2 - \varphi_1$ is the difference between latitudes, $\Delta\lambda = \lambda_2 - \lambda_1$ is the difference between longitudes, and R is the radius of the earth. Then we can get the average driving speed from t_1 to t_2 by $v_{1,2} = \frac{D_{1,2}}{t_2 - t_1}$. Similarly, the driving speed $v_{2,3}$ from t_2 to t_3 can be acquired. By comparing the value of $v_{1,2}$ and $v_{2,3}$, we can know the driver is accelerating, decelerating or driving in a constant speed. The bearing radian $\theta_{1,2}$ of driver d_1 from t_1 to t_2 can be calculated by [100]:

$$\theta_{1,2} = atan2(sin\Delta\lambda \cdot cos\varphi_2, cos\varphi_1 \cdot sin\varphi_2 - sin\varphi_1 \cdot cos\varphi_2 \cdot cos\Delta\lambda) \tag{3.15}$$

By comparing $\theta_{1,2}$ and $\theta_{2,3}$ between two consecutive timestamps, we can obtain the bearing angle of the driver, i.e., bearing towards left, bearing towards right, or driving straight.

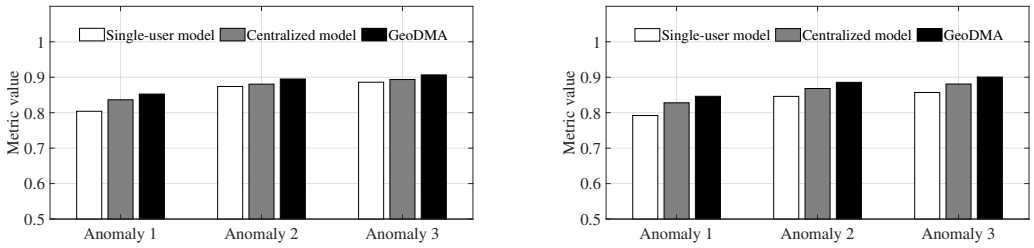
Driving Maneuver Anomaly Data. We need anomalous driving maneuvers to test the performance of our system. Since it is dangerous to collect anomalous driving data from real world, we borrow the idea from pBEAM [49] to simulate the anomalies based on the real-world data to evaluate the system. Three kinds of anomalies are simulated as follows:

- Anomaly 1: For aggressive drivers, they usually have irregular acceleration or deceleration [51]. This is one of the typical anomalies in real world. To simulate this scenario, we add Gaussian noise (mean 1.0, variance 1.0) to the acceleration of our normal test data. The unit of acceleration is m/s^2.

- Anomaly 2: When the driver is sleepy or distracted, the driving trajectories may show unexpected bearing angles [50]. We add Gaussian noise (mean 1.0, variance 1.0) to the bearing angle of our normal test data to simulate this scenario. The unit of bearing angle is radian.

- Anomaly 3: For DUI (driving under the influence) or DWI (driving while intoxicated/driving while impaired), both the acceleration and the bearing angle can become abnormal [101]. To simulate this scenario, we add Gaussian noise (mean 0.2, variance 0.04) to both acceleration and bearing angle. The mean and variance are set to a smaller value than the previous two cases is because there are changes in both acceleration and bearing angle.

Table 3.2 shows the data distribution of normal data and the generated anomalous data. We use 0.5 m/s^2 as the threshold to define the acceleration of a vehicle. If the acceleration of the vehicle is larger than 0.5 m/s^2, we define the vehicle is accelerating. If it is smaller than $-$ 0.5 m/s^2, we define it is decelerating. Otherwise, it is going at a constant speed. Similarly, we use 1 radian and -1 radian as the threshold to define if a vehicle is turning right, turning left, or going straight. In most of the experiments, we set the ratio of normal and abnormal test data as 1:1. Table 3.4 shows the experimental result of changing this ratio.



(a) AUC (area under the ROC curve) (b) F1 Score

Figure 3.5: Overall performance comparison.

3.6.3 Overall Performance

For the centralized model, we use the driving data of all 504 drivers to train it. In *GeoDMA*, we divide drivers into four sub-regions based on their GPS trajectories. But most of the drivers do not drive in the same sub-region all the time. To be consistent with the centralized model, we filter the drivers in each sub-region to find the drivers that driving in the same sub-region during all the 375 sliding windows. After filtering, there are 33, 24, 10, and 9 drivers left in each sub-region, respectively. We train a model for each sub-region by using the filtered data. We calculate the average AUC and F1 score of four region-based models. For the single-user model, we train 76 different models for these 76 drivers and calculate the average AUC and F1 score of these models. Figure 3.5 depicts the performance of *GeoDMA* and two baselines.

The experiment results show that the AUC of the centralized model achieves 0.836, 0.881 and 0.894 for the three generated anomalies. The corresponding F1 score is 0.828, 0.868 and 0.881. It achieves up to 8.5% higher accuracy than the single-user model. The reason for such an improvement is that we consider the vehicle-vehicle peer dependency in the centralized model. Moreover, the AUC of *GeoDMA* achieves 0.853, 0.895 and 0.907 for the three generated anomalies respectively. The corresponding F1 score is 0.846, 0.886 and 0.901. It further improves the accuracy of the centralized model by up to 2.2%.

In all anomalous scenarios, *GeoDMA* outperforms the centralized model. This is because *GeoDMA* considers the region partitioning. The driving features exist

common pattern in a small region across the vehicles due to similar contextual environments. Whereas, the centralized model is a general model and is trained for the whole city, it cannot take the local contextual features into account. We also found basically the performance of all the systems on anomaly 2 is better than anomaly 1. The performance on anomaly 3 is almost the same as anomaly 2. This demonstrates the model is more sensitive to anomalous bearing angles than anomalous speeds.

3.6.4 Performance under Different Scenarios

We evaluate the performance of *GeoDMA* in different sub-regions, with different amount of anomalous data. We also analyze the performance of *GeoDMA* along with the time and among different drivers.

Table 3.3 depicts the performance comparison of *GeoDMA* with two baselines. Among these four regions, the model for region 4 performs the best, the performance of the other three region models under the the same accuracy metric do not show much difference. The model for region 4 achieves up to 10.2% and 6.0% higher accuracy than the single-user model and the centralized model on average under F1 score. It also achieves up to 11.5% and 6.6% higher accuracy than the single-user model and centralized model on average under AUC. More importantly, almost all of these four models achieve better performance than the singer-user model and centralized model under the same evaluation metric on the three anomalies except the model for region 3. But the average performance of these models is still better than the singer-user model and the centralized model as we get from Figure 3.5. The experiment results demonstrate that it is reasonable to divide a big city into multiple sub-regions and develop multiple region models.

Different ratio of normal data to the anomalous data

Table 3.4 shows the performance of *GeoDMA* when we change the ratio of normal and anomalous test data. For all of the three anomalies under AUC, the performance is basically the same when we change the ratio. The performance under F1 score on anomaly 2 and anomaly 3 do not change much. The performance

Table 3.3: Performance comparison of different models on real-world data with simulated anomalies

	Model	Anomaly 1				Anomaly 2				Anomaly 3			
AUC	Single-User	0.80				0.87				0.89			
	Centralized	0.84				0.88				0.89			
	GeoDMA	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
		0.84	0.84	0.84	**0.89**	0.9	0.9	0.86	**0.92**	0.91	0.9	0.89	**0.93**
F1	Single-User	0.79				0.85				0.86			
	Centralized	0.83				0.87				0.88			
	GeoDMA	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
		0.84	0.84	0.83	**0.89**	0.89	0.89	0.84	**0.92**	0.91	0.89	0.87	**0.93**

under F1 score on anomaly 1 becomes a little higher when the anomalous test data becomes smaller. This experiment shows that the performance of our system is robust and suitable to the real-world scenarios.

Performance analysis from time perspective

We further investigate the performance of *GeoDMA* from time perspective. For the construction of state transition vectors, we set the length of a sliding window as 30 minutes. The driving maneuvers of the drivers may change over different time windows. We evaluate the performance of drivers over different time windows. In test data, there are 75 time windows. Figure 3.6(a) presents the anomaly detection performance of all drivers from time perspective under four sub-regions. As shown in the figure, the F1 score of anomaly detection is higher than 0.8 during most of the time windows. And we can see that the F1 score of the model fluctuates slightly but is still stable during different time windows. For the third region, from time window 35 to time window 60, its F1 score is lower than 0.8 but is still higher than 0.7. The experimental result demonstrates that *GeoDMA* has good performance along with time.

Performance analysis from driver perspective

We analyze the performance of drivers in each sub-region from space perspective. Each sub-region has a different driver set. Figure 3.6(b) depicts the cumulative probability of F1 score for the drivers in the four sub-regions. The cumulative probability of drivers achieving a certain F1 score in the four sub-regions has a similar trend. On average, only 15.2% of the drivers has a F1 score lower than 0.73. The F1 score of 66.7% drivers is higher than 0.79, the F1 score of 44.4% drivers is higher than 0.90. It proves that *GeoDMA* can provide high accuracy for almost all drivers. Moreover, the region models are robust because they show similar trend in all sub-regions.

3.6.5 Parameters Settings of *GeoDMA*

We further conduct experiments to set two important parameters in *GeoDMA*, *i.e.*, the number of sub-regions, the dimension size of hidden representation feature.

The number of sub-regions

As introduced in Section 3.4, we use our customized Ncut algorithm to partition a city into multiple sub-regions. However, the Ncut algorithm cannot decide the optimal number of sub-regions automatically. The number needs to be tuned based on different applications. We explore this number from 2 to 5 to find the optimal one. For each number, we conduct experiments over the whole dataset to evaluate driving anomaly detection accuracy.

Figure 3.7 presents the performance comparison under different numbers of sub-regions. For each case, we use the average AUC and F1 score of all sub-regions as its performance. Figure 3.7 shows that both AUC and F1 score increase first and then degrade on all the three anomalies. The number of sub-region parameter has a great influence on the system. For example, if the number is 2, the partition result has a little difference with original road network. If *GeoDMA* applies such a partition, it may have similar performance with the centralized model, *i.e.*, 0.831 *v.s.* 0.836 in F1 score.

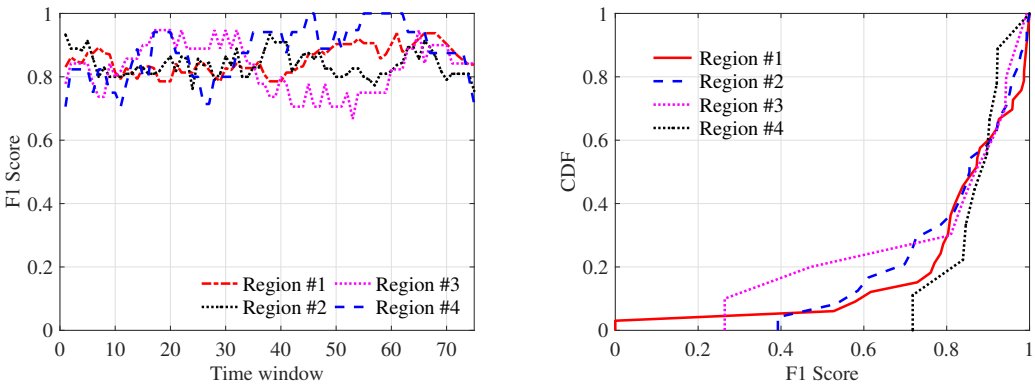
Table 3.4: Performance Comparison of different ratios of the normal data to the anomalous data in test data

	Ratio	Anomaly 1	Anomaly 2	Anomaly 3
AUC	1:1	0.853	0.895	0.907
	3:2	0.854	0.894	0.908
	3:1	0.852	0.89	0.908
F1 Score	1:1	0.846	0.886	0.901
	3:2	0.865	0.89	0.899
	3:1	0.881	0.893	0.899

When the number changes from 2 to 4, the performance increases. The reason is that the more sub-regions, the better the sub-region models extract peer dependency. However, based on our current dataset, if the number of sub-regions is too big, there will be few data belonging to the same sub-region, which may cause the model overfitting because of insufficient data. When the number is 5, we notice that the performance degrades much. We dig into this case and find that two of these five region models do not perform very well and influence the average performance of the system. Under the setting of our current dataset, *GeoDMA* performs the best when we partition the city into four regions. However, if we can get enough effective training data from this city, the optimal number of divisions may be different. In addition, if the dataset is collected from another city, the road network is different, the optimal partition result will be different. The larger the dataset, the more parts the system will divide a city into.

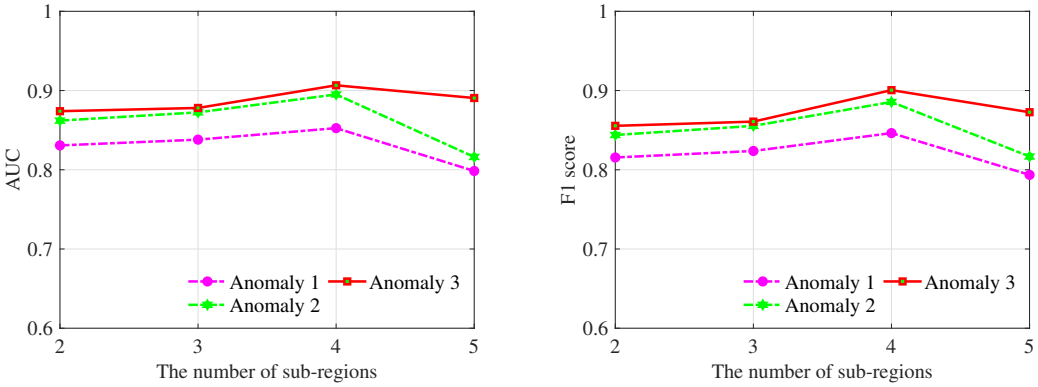
The size of the representation feature vector

We also do experiments to find out the optimal size of the representation feature. We set the representation feature size to 10, 20, 30, and 40 to train the region models. For each feature size, we show the average performance of 4 region models under AUC and F1 score. Here we use the Anomaly 1 to do the



(a) Accuracy along with time. (b) Accuracy of all drivers.

Figure 3.6: The performance of *GeoDMA* from time and driver perspective.



(a) AUC of different numbers of sub-regions (b) F1 score of different numbers of sub-regions

Figure 3.7: Performance of the different number of sub-regions.

experiments. Figure 3.8 shows that when the feature size is 20, the models perform the best in terms of both evaluation metrics. Therefore, we set the representation feature size to 20 to train all the driving anomaly detection models, including the singer-user model and the centralized model.

3.6.6 In-Situ Model Updating of *GeoDMA*

The prior experiments of *GeoDMA* are done without model updating. *GeoDMA* updates the model for each sub-region periodically in order to improve its performance. Each region model is updated by using the data collected from the vehicles within its coverage. Due to the limited available data, we update the models twice. Each time we use the data from 25 sliding windows to do incremental training. The data for model updating is different from the initial training data in the dataset. Figure 3.9 depicts the performance of *GeoDMA* under the settings without model updating ($U0$), updating once ($U1$) and updating twice ($U2$). It presents the average performance of the four sub-region models. We find that after each update, the performance of the updated models is improved on all anomalies under both AUC and F1 score.

3.6.7 Execution Efficiency of *GeoDMA*

A model for driving maneuver anomaly detection is required to be lightweight since the detected anomalies should be sent to the nearby drivers or passengers as fast as possible to avoid possible traffic. *GeoDMA* is lightweight and efficient. The size of *GeoDMA* auto-encoder is only 72.2 KB. The model provides an inference result in each sliding window (1-minute step). It takes about 8.37 ms to execute one inference in one time window for one driver. The inference time is measured on Alianware Aurora R7 (one Intel Core i5 8400 CPU (6-core) and one NVIDIA GTX 1070 GPU) without using its GPU. The driving data from other vehicles is no longer needed when doing anomaly detection for a specific vehicle with a well-trained model.

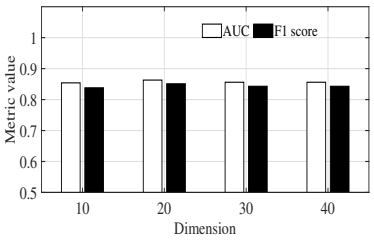
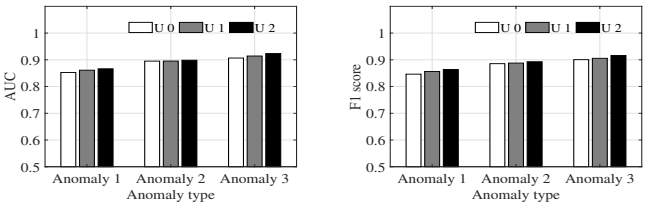


Figure 3.8: Different representation sizes.



(a) AUC (b) F1 score

Figure 3.9: *GeoDMA* with or without model updating.

3.7 Discussion

Privacy Issues. During online driving anomaly detection, *GeoDMA* collects the GPS locations from all the vehicles and uploads the locations to the corresponding edge servers. To protect user privacy, we anonymize the user identification information by a code. We will not collect any private and sensitive information from the users.

Static Geographical Partitioning. Due to the limited size of our current dataset, we perform the geographical partition statically. To provide better services for users, it is more reasonable to adjust the geographical partition dynamically as more new data is collected from the city. If the geographical partition is formed dynamically, new models for new local regions will be trained based on the new partition, using the data collected under new local regions. All historical data can be used for training, as long as it follows the new geographical partitioning. The new models will be trained offline. We still use the previous models to do online detection, while we are training the new models. Once the new models are well trained, they will be deployed for detection. Transfer learning that leverages the old models to train the new models will be explored to shorten the training time of new models in future work.

Model Generalization. Our models can be used to do anomaly detection for different scenarios (i.e., sparse, or dense traffic flows) if the models are trained with sufficient data collected from those scenarios. Moreover, we believe the detection accuracy can be further improved if the models are developed depending

on different scenarios. As more and more GPS locations are collected from the users, the models in the same area can be trained in a fine-grained way. We can train different models for workdays and weekends. To be more specific, during the same day, we can train different models for different time periods, e.g., a model for peak hours, a model for off-peak hours in the daytime, and a model for the night.

3.8 Conclusion

This chapter presents *GeoDMA*, which leverages unsupervised deep auto-encoders and a geo-distributed partitioning algorithm to develop a driving maneuver anomaly detection system. The auto-encoder learns normal driving features from historical data by considering both temporal and peer dependency. Our geo-distributed partitioning algorithm further divides a city into several sub-regions. We then train a specific model for each sub-region to improve the detection accuracy in each sub-region. Extensive experiments in a large-scale real-world vehicle GPS trajectory dataset show that *GeoDMA* outperforms the baseline methods.

Chapter 4

Real-Time Tracking of Smartwatch Orientation and Location by Multitask Learning

Arm posture tracking is essential for many applications, such as gesture recognition, fitness training, and motion-based controls. Smartwatches with Inertial Measurement Unit (IMU) sensors (i.e., accelerometer, gyroscope, and magnetometer) provide a convenient way to track the orientation and location of the wrist. Existing orientation estimations are based on predefined data fusion methods that do not consider the variations in the data quality of different IMU sensors. Existing location estimations rely on the estimated orientation results. A small orientation estimation error may cause high inaccuracy in location estimation. Moreover, these location estimation algorithms, e.g., Hidden Markov Model and Particle Filters, cannot provide real-time tracking on commercial mobile devices due to high computation overhead. This chapter presents *RTAT*, a Real-Time Arm Tracking system that tackles the above limitations in a data-driven way. *RTAT* estimates both orientation and location simultaneously using a multitask learning neural network. It also incorporates a unique attention layer and a dedicated loss function to learn the dynamic relationship among IMU sensors. *RTAT* supports real-time tracking by performing model inference on smartphones. Finally, to train *RTAT*'s neural network, we develop an easy-to-use labeled data collection system that uses

a low-cost virtual reality system to provide orientation and location labels for the smartwatch. Extensive experiments show *RTAT* significantly outperforms existing state-of-the-art solutions in both accuracy and latency.

4.1 Introduction

Real-time and accurate arm posture tracking is essential to the performance of many applications, e.g., gesture recognition [102, 103], gym exercise assessment [104], and motion-based control [105]. Once we know the a user's forearm length and orientation and location of wrist, we can estimate the user's elbow location and track the arm movements [106, 104]. As smartwatches are more pervasively adopted, they provide a more easily accessible arm posture-tracking alternative to other infrastructure-based systems, such as wireless sensing [107, 108, 109], visible light [110, 111] and customized wearable sensors [112]. The Inertial Measurement Unit (IMU) inside a smartwatch, can be used to track arm motions [104, 106, 113, 114].

Arm posture tracking requires continuous knowledge of a smartwatch's three-dimensional (3D) orientation and location in a desired reference frame, e.g., the Global Reference Frame (GRF), typically ¡North, East, Up¿. However, all three IMU sensors report sensing readings in the Watch's Reference Frame (WRF). We must find the transformation between these two reference frames. The WRF-to-GRF rotation is the orientation of the smartwatch in GRF. This rotation is needed to calculate the watch's location in GRF.

Gyroscopes measure the angular velocity around the three axes of a device. Intuitively, with a known initial orientation, subsequent orientations can be estimated by integrating gyroscope readings over time. However, estimation error accumulates with the noise and bias of the gyroscope [115, 116, 117, 118]. A^3 calibrates the orientation results using the direction anchors measured by the accelerometer and magnetometer when the smartphone is static or moving at a constant speed [119]. Only in these moments can gravity be accurately decomposed from accelerometer readings, since the accelerometer measures a mixture of gravity

and linear acceleration. The magnetometer measures geomagnetic North, which can be leveraged to estimate the heading angle of a device. Once the directions of both gravity and magnetic North are known in GRF, the device's orientation in GRF can be uniquely determined. However, such calibrations can only be done opportunistically because a device may pause infrequently. MUSE [113] adopts a complementary filter to do calibration continuously using magnetic North because the magnetic North is unaffected by the motion of a device. However, such fixed calibration methods prove insufficient because the magnetic readings are sensitive to nearby environments. For example, when the magnetic readings are skewed by nearby ferromagnetic materials, as is often true indoors [119, 113], orientation estimation should not strongly rely on the magnetic readings. Therefore, an adaptive orientation estimation method is necessary for incorporating the readings of three IMU sensors according to their varying quality.

State-of-the-art approaches for wrist location estimation rely on orientation estimation. ArmTrak [106] adopts a Hidden Markov Model (HMM) and MUSE [113] uses a Particle Filter. ArmTroi [104] reduces the computational latency of the HMM proposed in ArmTrak. These methods use estimated orientation to project accelerometer readings onto the desired reference frame. However, locations derived from inaccurate orientations can fall outside the possible space. Furthermore, they cannot provide real-time wrist location tracking on smartphones if the sampling frequency is higher than 10 Hz. A high sampling frequency (e.g., 50 Hz) is desired for fine-grained arm tracking applications, e.g., motion-based control.

In this chapter, we develop a Real-Time Arm Tracking (*RTAT*) system for smartwatches. *RTAT* uses a multitask neural network for simultaneous prediction of both orientation and location. It leverages Bidirectional Long Short-Term Memory (BiLSTM) as its backbone, considering its effectiveness for time-series data processing [49, 118]. Our proposed multitask learning scheme offers the following benefits over conventional arm tracking systems. First, *RTAT* estimates orientation and location simultaneously [120]. Orientation and location estimations are two related tasks. Solving them together with multitask learning improves accuracy and avoids additional overhead for training two separate models. Second,

as a supervised learning method, *RTAT* learns the best fusion scheme of three IMU sensor data streams from the labeled data, which is more immune to the noise of IMU sensor data [118, 117, 115, 116]. Unlike conventional sensor fusion methods with predefined calibrations, *RTAT* adapts to complex temporal variations in the quality of the three IMU sensors' readings. Third, *RTAT* is faster than conventional location estimation methods (i.e., 0.1633 ms vs. 2337.50 ms for processing 50 samples).

Building the aforementioned learning system involves the following three challenges. 1) How do we teach the neural network to adapt to temporal variations of IMU sensor data quality? 2) If we utilize a loss function that minimizes the difference between inferred results and the labels, the neural network's outputs are independent at different timestamps. However, arm movements are not just sequences of independent wrist orientations and locations. How can we incorporate the temporal correlation of consecutive arm movements in our neural network? 3) A large-scale labeled dataset is necessary to develop the system. It is challenging to collect accurate orientations and locations of a smartwatch at a low cost.

We tackle the above three challenges with a set of novel techniques developed for *RTAT*. 1) We develop a BiLSTM-based multitask neural network for processing three IMU data streams. We also design an attention mechanism in our neural network architecture to dynamically learn the importance of different IMU sensor streams. 2) We incorporate a smooth loss into the loss function of the neural network. The smooth loss ensures the change rate of the orientation and location are sufficiently similar to those of the labels. 3) We use the Meta Quest 2 VR system to collect labeled data to train and validate the model offline. Meta Quest 2 includes one headset and two touch controllers. To collect data, the users wear a smartwatch and hold a VR touch controller simultaneously while they are moving their arms. Meta Quest 2 is able to accurately measure the orientation and location of its touch controller, but not those of the smartwatch. To address this challenge, we develop a labeled data collection system that converts the VR measurements into the orientation and location of the smartwatch. *RTAT* only relies on the VR system for training data collection.

We collect data from nine volunteers (four females and five males) at two places. This research study has been approved and exempted by the IRB committee of UC Merced. We do not pre-define any gestures for the volunteer users. We ask them to move their arms freely at their natural speed. They perform random arm gestures or daily gestures, including driving, drinking, writing, exercising, push and pull, drawing and so on. Each user performs gestures in their own way. The arm motion style and motion speed of different users are different. To the best of our knowledge, this is the first dataset of fine-grained orientation and location labels for smartwatch tracking. The dataset is available at https://github.com/mmmmliu/RTAT.

We use data from five volunteers to train and evaluate a general neural network model. Extensive experiments show that *RTAT* reduces the orientation estimation error by up to 51% and 31.63% at two places, respectively, compared to state-of-the-art orientation estimation methods. *RTAT* also reduces the location error up to 45% and 46.9% these two places, respectively, compared with state-of-the-art location estimation methods. Additionally, we test *RTAT* with four new users whose data is not used for training. The experiment results show *RTAT* has no significant performance degradation on new users. Furthermore, *RTAT* can easily support real-time arm tracking with the maximum data sampling frequency on commercial smartphones.

In summary, this chapter makes the following contributions.

- To the best of our knowledge, this is the first work to leverage end-to-end deep learning for IMU-based arm tracking.
- This is the first work to track the orientation and location of a smartwatch simultaneously, rather than sequentially.
- We consider adjusting the importance of the three IMU channels according to their temporal variations.
- We develop a labeled data collection system to collect the orientation and location labels of a smartwatch.
- Extensive experiments are conducted to evaluate *RTAT*.

4.2 Related Work

Deep Learning for Device Orientation Estimation. Deep learning has been used in many applications, such as image processing [121, 97, 122], wireless networking [123], smart buildings [95, 27], smart driving [124], smart irrigation [96], and map matching [99]. Recent literature has begun utilizing deep neural networks for IMU measurements processing [115, 125, 126, 127] and orientation estimation [116, 117, 118]. OriNet [116] uses an LSTM-based architecture to estimate the 3D orientation of flying robots from gyroscope readings. Brossard et al. [117] estimate device orientation by correcting gyroscope readings with a CNN, then integrating the corrected readings. These methods do not explore learning from multi-modality sensors. The accelerometer and magnetometer can help estimate orientation when the gravity error or the deviation of magnetic field density is small. IDOL [118] proposes an Extended Kalman Filter (EKF) architecture to estimate device orientation. The prediction model of its EKF utilizes an LSTM-based neural network. This neural network estimates orientation using all three IMU sensors. The measurement model of its EKF is based on gyroscope readings integration. While Kalman Filter and its variants are dependent on the system noise parameters, the noise of IMU sensors is environment-dependent. IDOL uses a static diagonal propagation noise matrix for the gyroscope readings integration, which is not able to depict the system noise.

RTAT greatly differs from IDOL in the following ways. First, IDOL outputs its orientation result as a unit quaternion, which cannot be accurately predicted by a neural network without a dedicated loss design. Second, we propose a novel network architecture from IDOL. We design a multitask learning network to output orientation and location simultaneously rather than with separate neural networks. Third, we do not treat readings from the three IMU channels equally. Instead, we incorporate an attention layer to adjust the feature importance from the three IMU channels. Finally, we develop a labeled data collection system for smartwatch-based arm tracking.

Conventional Device Orientation Estimation. A^3 [119] intelligently selects the moments that gravity and magnetic North are reliable and calibrates the

orientation estimation from gyroscope integration. However, the assumption that device motions have frequent pauses for resetting the orientation is inapplicable for wearables. MUSE [113] proposes that magnetic North is more trustworthy than gravity because it is unpolluted by device motion. It designs a magnetometer-centric sensor fusion algorithm based on the complementary filter for orientation tracking. However, magnetic North can only calibrate 2 of 3-DoF (Degrees of Freedom) of orientation, and magnetic fields can vary significantly within the same space due to the local ferromagnetic disturbances. The complementary filter's performance is highly dependent on the appropriate selection of its parameters. For example, when magnetic interference is high, we should adjust our confidence in the magnetometer readings. The complementary filter proposed by MUSE cannot adapt to different environments with fixed magnetometer calibrations.

IMU-based Location Tracking. Prior studies [128, 129, 130, 131, 132, 133] leverage multiple sensors to track the human body or upper limb movement. ArmTrak [106] proposes to recover and track the 3D arm posture using only a smartwatch. It leverages a Hidden Markov Model (HMM) to continuously estimate elbow and wrist locations. However, its computation latency is high, and it cannot support real-time performance on smartphones. ArmTroi [104] optimizes ArmTrak with HMM state reorganization and hierarchical search. It is faster than ArmTrak but still suffers from high latency with sampling rates above 10 Hz. MUSE [113] uses Particle Filters to estimate the smartwatch location. It achieves a higher location estimation accuracy than ArmTrak and ArmTroi, but with higher computational latency than ArmTroi Its real-time computation still cannot be afforded by a smartphone.

Human Skeleton Tracking. Various sensing modalities have been used for estimating the posture of the human skeleton, including vision [134], light [110, 111], wireless signals [135, 136], and mm-wave [109]. However, these systems require infrastructure support. They also have limited service coverage and performance will decrease when tracking multiple people simultaneously.

4.3 Background & Motivation

Table 4.1: Average orientation error of complementary filter at different places for ten-minute data traces.

	Speed (m/s)	Gravity Error(°)	Magnet Deviation(°)	Gravity Opportunity(%)	Sensors_to_use	Orientation Error(°)
					Mag. + Accl.	58.93
S1:Hallway	0.39	7.15	45.41	3.34 %	Mag.	72.53
					Accl.	27.10
					Mag. + Accl.	19.17
S2: Room	0.30	5.37	5.10	4.4%	Mag.	46.93
					Accl.	29.88
					Mag. + Accl.	53.03
S3: Room	0.98	20.47	9.0	1.94%	Mag.	52.59
					Accl.	77.04

In this section, we introduce orientation representations and analyze the existing orientation estimation solutions.

4.3.1 Orientation Representation

The 3D orientation of an object can be represented in different ways, i.e., quaternion, rotation vector (axis/angle) and 3x3 rotation matrix. Each representation can be converted to another.

A unit quaternion is a 4D complex vector $q = (q_0, q_1, q_2, q_3)$.

$$q_0 = cos(\frac{\theta}{2}), q_1 = x \cdot sin(\frac{\theta}{2}), q_2 = y \cdot sin(\frac{\theta}{2}), q_3 = z \cdot sin(\frac{\theta}{2}). \tag{4.1}$$

It represents a rotation of degree θ along a vector (x, y, z) from the default orientation. All four items in a unit quaternion must satisfy the following constraint.

$$\sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} = 1 \tag{4.2}$$

A unit quaternion can be uniquely transformed into a rotation matrix using the Rodrigues' rotation formula as follows.

$$\begin{bmatrix} 1-2(q_2^2+q_3^2) & 2q_1q_2-2q_3q_0 & 2q_1q_3+2q_2q_0 \\ 2q_1q_2+2q_3q_0 & 1-2(q_1^2+q_3^2) & 2q_2q_3-2q_1q_0 \\ 2q_1q_3-2q_2q_0 & 2q_2q_3+2q_1q_0 & 1-2(q_1^2+q_2^2) \end{bmatrix} \tag{4.3}$$

A unit quaternion can also be uniquely transformed into a 3D rotation vector by Equation (4.4).

$$\begin{aligned} &\vec{v} = (v_1, v_2, v_3) = (\theta \cdot x, \theta \cdot y, \theta \cdot z) = \theta \cdot (x, y, z) \\ &\theta = 2 \cdot arccos(q_0) \ \ (\theta \neq 0) \end{aligned} \tag{4.4}$$

where $\|\vec{v}\| = \theta$, and its direction is the rotation axis. Uniquely, when $\theta = 0$, $\vec{v} = \mathbf{0}$. Any rotation vector can be uniquely transformed into a unit quaternion by Equation (4.5).

$$\begin{aligned} \mathbf{q} = (&cos(\frac{\theta}{2}), \frac{v_1}{\theta} \cdot sin(\frac{\theta}{2}), \frac{v_2}{\theta} \cdot sin(\frac{\theta}{2}), \frac{v_3}{\theta} \cdot sin(\frac{\theta}{2})) \\ &(\theta = \|\vec{v}\|, \|\vec{v}\| \neq 0) \end{aligned} \tag{4.5}$$

Uniquely, when $\|\vec{v}\| = 0$, $\mathbf{q} = (1, 0, 0, 0)$.

4.3.2 Conventional Orientation Tracking

We implement a complementary filter, one of the representative conventional orientation estimation approaches. We conduct a set of experiments under different scenarios to investigate its performance. The device we use is a Fossil Gen 5 smartwatch. A VR touch controller of Meta Quest2 is used to provide ground truth orientations for evaluation. Details about our data processing are introduced in Section 4.4.5.

We find two places to do experiments: a room and a hallway. From our measurements, the magnetic field in the room is stable, whereas the magnetic field in the hallway is unstable. We ask one volunteer to wear the smartwatch and hold the VR controller by his/her left hand, then perform free gestures as

introduced in Section 4.1. The experiments are conducted under three scenarios. For each, the user is asked to move his/her arm for ten minutes.

We implement three kinds of complementary filters, all primarily dependent on gyroscope readings integration. The first is an implementation of MUSE [113]. It fuses magnetometer and accelerometer readings to calibrate orientation drift caused by gyroscope readings integration. The magnetometer readings are used continuously, while the accelerometer readings are used opportunistically (when the accelerometer roughly measures 9.8 m/s^2). The second fuses just magnetometer readings continuously, and the third fuses just accelerometer readings opportunistically to perform the calibration.

Table 4.1 shows the statistical analysis of the three scenarios: S1, S2 and S3. In Table 4.1, speed is calculated as the average moving speed of the device for the ten-minute data. Gravity error is the average angular difference between every measured gravity direction and the true gravity direction ('down'). It is highly affected by the motion of a device. Magnet deviation represents the angular deviation of the measured magnetic direction in GRF. It measures the stability of the magnetic field. The gravity opportunities refer to the percentage of data samples that are considered to be linear-acceleration-free over the ten-minutes data. They are considered as such moments when the accelerometer readings are 9.8 ± 0.3 m/s^2 during consecutive 500 ms. Sensors_to_use denotes which sensors are used to calibrate the gyroscope readings integration. S1 stands for the scenario where the magnetic field is unstable but the motions of the device are slow (North is inaccurate, gravity is accurate). S2 is the scenario where the magnetic field is stable and the motions of the device are slow (both North and gravity are accurate). S3 represents the scenario where the magnetic field is stable but the motions of the device are fast (North is accurate, gravity is inaccurate). By analyzing the results in Table 4.1, we get two observations.

Observation 1: *Incorporating magnetometer readings will hurt the orientation estimation when the magnetic field is distorted; and vice versa.* S1 and S2 are conducted at different places with similar motion speeds. They have similar gravity errors. S1 has 45.41° magnetic direction deviation, whereas S2 has 5.10°.

This indicates the magnetic field in the hallway is unstable, but stable in the room. The results of S1 in Table 4.1 show a 58.93° error when using two sensors to calibrate. The error when using either magnetometer or accelerometer is 72.53° and 27.10°, respectively. Calibrating with two sensors results in a larger error than calibrating with solely accelerometer, but a smaller error than calibrating just with the magnetometer. This indicates incorporating magnetometer hurts the performance. In comparison with S2, calibrating with two sensors produces a smaller error than calibrating with one of them, which indicates the magnetometer improves the orientation estimation under this scenario.

Observation 2: *Incorporating gravity does not improve the orientation estimation when the device moves fast.* We further investigate the effect of gravity on orientation estimation with S3 in Table 4.1. In this scenario, the motions of the smartwatch are fast. The average moving speed is 0.98 m/s and gravity calibration opportunities occur only 1.94% of the time. The gravity error is larger than in S1 and S2. In this scenario, calibrating with two sensors gets almost the same error as calibrating just with the magnetometer, demonstrating that gravity does not improve the orientation estimation.

Summary: *A better sensor fusion scheme is needed to tolerate the noise from IMU sensors for accurate orientation tracking.* Each IMU sensor has its own limitations. Integration of gyroscope readings drifts over time. Accelerometer readings are highly motion-dependent. Magnetometer readings are highly environment-dependent. Due to these inherent hardware limitations, we need a more flexible data fusion method that can adapt to different scenarios automatically. Data-driven methods [118, 117, 115, 116] have shown great potential for handling data noises and adapting to the variation of data distribution in many computer vision and natural language processing applications [137, 138, 139].

4.3.3 Conventional Location Tracking

Conventional location tracking approaches of smartwatches rely on the orientation of the smartwatches. Orientation is used to transform the accelerometer readings into a desired reference frame, e.g., GRF, to infer the location of the device

in GRF. As introduced in Section 4.1 and Section 4.2, three solutions have been proposed for smartwatch location tracking recently, ArmTrak [106], ArmTroi [104], and MUSE [113]. These solutions require pre-existing knowledge of user-specific information, including shoulder width and the lengths of the lower arm, upper arm and torso. They use user-specific information to generate 3D point clouds for each user and search the possible locations of the smartwatches from these point clouds. There are three main limitations to those solutions. First, the point cloud generation process is time-consuming. The more point clouds generated, the more time is needed. Second, point clouds are generated according to the user-specific information, which require each user to generate his/her personal point clouds and to run HMM or Particle Filters to estimate the possible locations of the smartwatch. Point clouds are not generalizable to different users. Third, the computation latency of such searching solutions is long and cannot be afforded by commodity mobile devices in real-time if the sampling rate is higher than 10 Hz.

Summary: *A new method that supports accurate real-time smartwatch location tracking on mobile devices is needed.* In this paper, we exploit the benefits of deep learning to track the device's orientation and location.

4.4 The Design of *RTAT*

In this section, we first introduce the overview of our system. We then show our design of multitask learning neural network, attention-based feature adjustment, and labeled data collection.

4.4.1 Overview

Figure 4.1 shows the two major parts of our system, i.e., real-time arm tracking and labeled data collection. During the offline training phase, we feed the collected IMU readings of the smartwatch and labels to *RTAT* to train a multitask model. During the online inference phase, a user can use the well-trained model to do inference by just wearing a smartwatch. The model can be executed on a smartphone. The smartwatch transmits the IMU sensor data stream to the smartphone

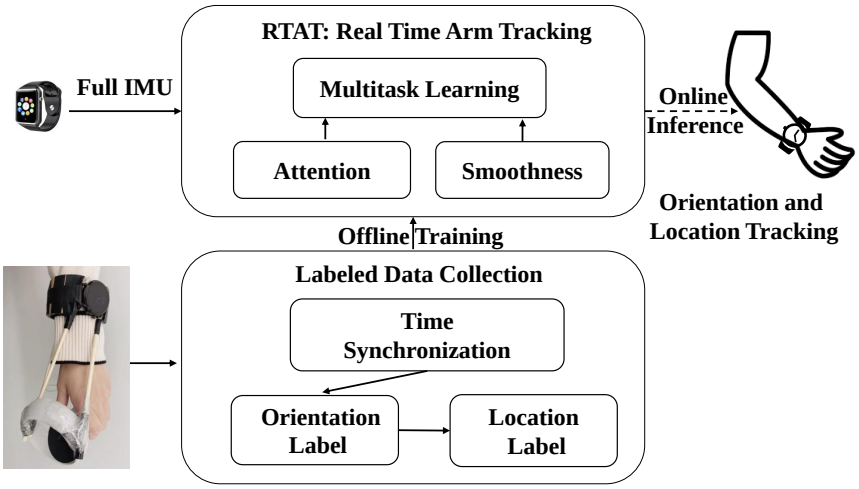


Figure 4.1: System Overview of *RTAT*

via Bluetooth. The smartphone receives and then forwards the IMU readings to the model deployed on the smartphone. The labeled data collection system is no longer needed during the inference phase.

***RTAT*'s Arm Tracking System.** We design a multitask neural network to jointly predict the orientation and location of the wrist (Section 4.4.2). The inputs of the neural network are the IMU sensor data stream from a smartwatch, and the outputs are the corresponding orientation and location series. Since the importance of different IMU sensors in orientation estimation is varying over time, we design an attention mechanism on top of our multitask neural network (Section 4.4.3). The attention mechanism adjusts the input focuses of the network automatically according to the varying sensor data quality. Furthermore, to guarantee the smoothness of the inferred arm motions, we employ smooth losses for both orientation and location tracking (Section 4.4.4).

***RTAT*'s Labeled Data Collection.** To train a model with supervised learning, we need to build a training dataset. The dataset should consist of a large amount of time-series IMU readings from the smartwatch, and the orientation and location labels of the smartwatch. However, acquiring accurate labels for the smartwatch is a non-trivial task. Thus we develop a labeled data collection part (Section 4.4.5) to derive training labels. As depicted in Figure 4.1, the training

data collection process requires volunteers to wear a smartwatch and hold a VR controller simultaneously. The VR system provides labels for the smartwatch. However, the readings we collect from the VR system describe the orientations and locations of the VR controller, not the smartwatch. Since the center of the VR controller and smartwatch are not aligned, training the model on unprocessed VR controller data would constitute training a model to predict the orientations and locations of the VR controller given IMU readings from the smartwatch. To fill in the gap between the acquired orientations and locations of the VR controller and the required ones of the smartwatch, we design a labeled data collection system. The system is built in three steps, i.e., time synchronization and alignment (Section 4.4.5 and Section 4.4.5), orientation label derivation (Section 4.4.5) and location label derivation (Section 4.4.5).

4.4.2 Multitask Learning Neural Network

Previous solutions usually estimate the orientation of devices first, and then the location. Multitask learning provides us the opportunity to solve multiple tasks simultaneously [140, 120]. We thus design a multitask neural network to jointly estimate the device orientation and location from the IMU readings. Our multitask neural network has two outputs, i.e., orientation and location.

Formally, we denote a posture P_w of a wrist as the union of an orientation ori_w and a location loc_w, where $P_w = \langle ori_w, loc_w \rangle$. Our multitask neural network is designed to learn a data-driven mapping from IMU measurements to orientations and locations as follows.

$$ori^t, loc^t = f(a^t, \omega^t, m^t) \tag{4.6}$$

where a^t, ω^t and m^t are the readings from accelerometer, gyroscope and magnetometer at timestamp t. Each of them is a 3D vector as depicted in Figure 4.2. f is the weight to be learned by the multitask network. ori^t and loc^t are the 3D orientation and 3D location predicted by the neural network at at timestamp t.

Figure 4.2 demonstrates our network architecture. The network consists of three BiLSTM layers [141] and two fully-connected (FC) layers. BiLSTM is one kind of Recurrent Neural Network (RNN). RNN is designed for processing sequen-

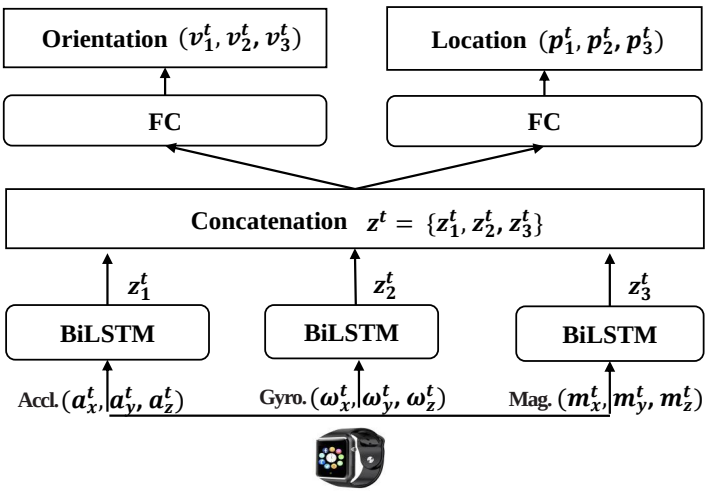


Figure 4.2: Multitask Network Structure.

tial data. We set the length of a sequence as 32 time steps (32 data samples) in our implementation. A BiLSTM consists of two LSTMs, a forward LSTM and a backward LSTM. The forward one takes an input sequence in a forward direction, and the backward one in a backward direction. The BiLSTM layer's output is a combination of the two LSTM layers' outputs. We set the sampling rate of IMU sensors to 50 HZ. Data for an input sequence (32 samples) can be collected in less than one second. From Section 4.6.5, our model takes on average 2˜3 ms to process one-second of data (50 samples) on smartphones.

Each BiLSTM layer takes the 3D vector sequences from one of the IMU sensors. The outputs of BiLSTM layers are the hidden states of the temporally dependent data. The hidden states from the three BiLSTMs are concatenated in the concatenation layer. The concatenated vectors are forwarded to each of the FC layers to predict the orientations and locations. A loss function plays an important role in the fast and accurate training of a neural network. The rest of this subsection focuses on our loss function design.

Orientation Output. A rotation is a process of 3 degrees-of-freedom (DoF). A quaternion we introduced in Section 4.3.1 is a 4D vector used to represent a 3D rotation. If we define the orientation output as a quaternion, we should normalize the square of the 4D vector as 1 to meet Equation (4.2). From our experiments,

the neural network cannot learn quaternions well following this design. Thus, we use a 3D rotation vector instead of a quaternion as the orientation output of our multitask neural network.

An orientation label we collect from the VR system at each timestamp is a unit quaternion. Therefore, we transform the 3D vector output of our neural network into a unit quaternion using Equation (4.5) when we calculate the loss. By comparing the inferred quaternions to the quaternion labels, the neural network is trained to learn from the labeled data. Since the quaternion label meets the constraint in Equation (4.2), we implicitly implant the constraint into our learning process to predict the 3D rotation vectors.

Loss Function for Orientation. Assume a quaternion label is q_t and the predicted quaternion is $\hat{q}_t$. The loss for orientation is:

$$L_{ori} = \frac{1}{T}\sum_{t=1}^{T} deg(\hat{q}_t \cdot q_t^{-1}) \tag{4.7}$$

$$deg(q) = 2 \cdot arccos((q'_0, q'_1, q'_2, q'_3)) = 2 \cdot arccos(q'_0) = \theta \tag{4.8}$$

where q_t^{-1} is the inverse of q_t, and $\hat{q}_t \cdot q_t^{-1}$ returns a quaternion, which is the rotation from q_t to $\hat{q}_t$. The operator $\cdot$ is the Hamilton product, which represents the quaternion product. The function $deg(q)$ is the quaternion difference of q_t and $\hat{q}_t$. The constant parameter T is the number of data samples in a training batch.

Loss Function for Location. Location can be represented as a 3D vector, with the form $p = (p_1, p_2, p_3)$. Mean squared error (MSE) between the predicted locations and labels is used for location loss.

$$L_{loc} = \frac{1}{T}\sum_{t=1}^{T}(p_t - \hat{p}_t)^2 \tag{4.9}$$

where the p_t and $\hat{p}_t$ are the position label and the predicted position at timestamp t, respectively.

Loss Function for Multitask Learning. With the definition of the orientation loss and location loss, the overall loss of the multitask learning is given.

$$L_1 = \alpha L_{ori} + \beta L_{loc} \tag{4.10}$$

twhere α and β are hyper-parameters to balance the two losses.

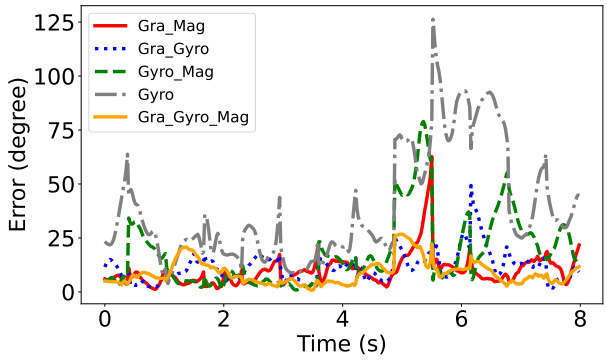


Figure 4.3: Orientation error changes along with time of the models with different combinations of sensor inputs.

4.4.3 Attention-based Feature Adjustment

As shown in the motivation experiments, three IMU sensors play varied roles in orientation estimation in the conventional filtering algorithms. We further investigate the importance of each sensor for orientation estimation in deep learning.

We build and train five models using data from the same user, each with different inputs. The inputs are different combinations of data from the three IMU sensors. Gra, Mag and Gyro stand for the readings of the accelerometer, magnetometer and gyroscope respectively. For example, the inputs for "Gra_Mag" are the data combinations from accelerometer and magnetometer.

Figure 4.3 demonstrates how the error of different models varies over time. None of them consistently outperform the other sensor combinations. Each of them achieves the lowest error at different points. The model that utilized all three sensors had the most stably low error. Table 4.2 shows a statistical analysis over ten minutes of data. The third row, "Be-The-Best," presents the percentage of time the corresponding data combination can achieve the lowest prediction error. "Gra_Gyro_Mag" performs the best the majority of the time, but not always. Other models sometimes take the place.

From Figure 4.3 and Table 4.2, we observe that we still need to use the data from all three sensors as input for our neural network. To fully exploit the neural network's capabilities and achieve higher accuracy, we must dynamically identify the importance of each sensor.

Table 4.2: The orientation error (degree) and Be-The-Best (BTW, %) for ten-minutes data from different sensor combinations.

Model	Gra_Mag	Gra_Gyro	Gyro_Mag	Gra_Gyro_Mag	Gyro
Error	9.38	17.14	16.92	8.12	33.98
BTW	30.16	11.22	12.82	44.18	1.62

We thus introduce an attention-based design on top of the multitask neural network to learn how important a sensor is at each timestamp. As shown in Figure 4.4, the network inputs include three parts; They are the data from accelerometer, gyroscope and magnetometer, respectively. The attention scheme is designed to automatically adapt the network to different parts of the inputs. Attention is an emerging technique. It is used for automatically adjusting the focuses of a DNN by multiplying a weighting vector, the value of each element in the vector can vary [138, 104]. In our case, the network should treat readings from different sensors differently. It intends to use the most effective portion (context) to derive outputs. We can exploit this ability to dynamically increase the weight of important sensor inputs and reduce the weight of unimportant sensor inputs. Hence, the attention scheme is suitable for learning the importance of different input channels.

As shown in Figure 4.4, we add an attention layer into the network, which learns to assign weights for different IMU input channels. Originally, as shown in Figure (4.2), z^t is the concatenation of the BiLSTMs' outputs, where $z^t = \{z_1^t, z_2^t, z_3^t\}$, they are from the accelerometer, gyroscope and magnetometer, respectively. This feature vector z^t serves as the input of the last two FC layers. Instead of equally fusing them into the last two layers, an adaptive context vector c_i^t is designed to weight z_1^t, z_2^t, z_3^t, and then concentrate the weighted value of them as shown in the Equation (4.11) below.

$$c_i^t = concat(\alpha_1^t z_1^t, \quad \alpha_2^t z_2^t, \quad \alpha_3^t z_3^t) \tag{4.11}$$

where z_r^t and α_r^t are the features from different inputs and their corresponding

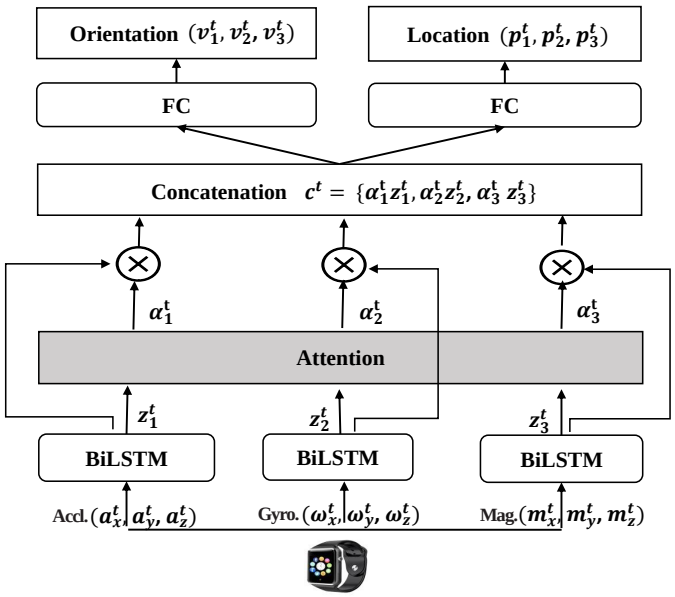


Figure 4.4: Attention Network Structure.

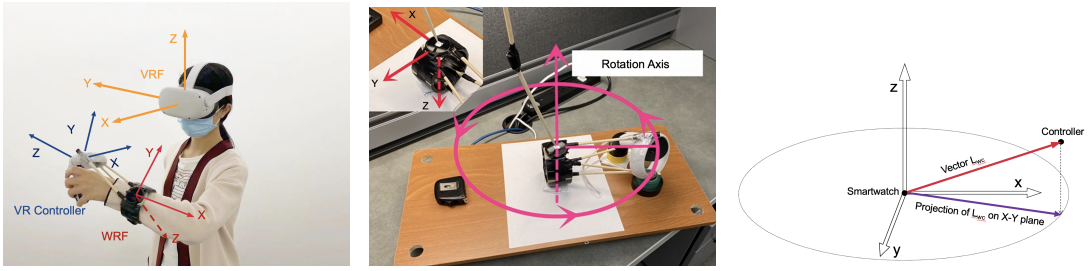
weights. α_r^t measures the importance of the feature from each part of the input data. It differentiates the contributions of each input part to the orientation prediction. The weight α_r^t for a highly contributed z_r^t will be greatly increased in Equation (4.11) by the attention function. Otherwise, α_r^t will be gradually decreased. The attention function is typically realized as a single-layer multiplayer perceptron, such as $tanh(\cdot)$ and $ReLu(\cdot)$. The calculation of α_r^t is:

$$\begin{aligned} att^t &= tanh(W \cdot z^t + b) \\ \alpha_r^t &= f_{softmax}(att^t) \end{aligned} \tag{4.12}$$

where att^t is an intermediate variable; $f_{softmax}(\cdot)$ scales the weights α_r^t to the range $[0, 1]$. W and b are the trainable parameters to be determined in the training phase.

4.4.4 Smooth Losses

We train our multitask learning neural network by minimizing the distance between the predicted postures ($P_w = \langle ori_w, loc_w \rangle$) and the posture labels. However, this loss design treats the posture at each time point independently. Sometimes, the inferred movements of the human wrist may not be continuous and smooth over time, resulting in an unrealistic posture estimation. To fully leverage the temporal



(a) Different reference frames (b) Rotation experiment (c) Projection of the vector

Figure 4.5: Labeled Data Collection System.

correlation between the wrist postures at two adjacent timestamps, we add a smoothness item in our loss function for both orientation and location predictions. The new smooth loss makes the difference between consecutive postures close to that of the labels:

$$L_{ori_s} = \frac{1}{T-1}\sum_{t=2}^{T} deg((q_t \cdot q_{t-1}^{-1}) \cdot (\hat{q}_t \cdot \hat{q}_{t-1}^{-1})) \tag{4.13}$$

$$L_{loc_s} = \frac{1}{T-1}\sum_{t=2}^{T}((p_t - p_{t-1}) - (\hat{p}_t - \hat{p}_t))^2 \tag{4.14}$$

The difference between consecutive orientations measures the changing rate of orientation, which is the angular velocity. The difference of consecutive locations measures the changing rate of location, which is the velocity. With these two smooth losses, we extend the multitask learning loss function in Equation (4.10) as follows:

$$L_2 = \alpha L_{ori} + \beta L_{loc} + \gamma L_{ori_s} + \eta L_{loc_s} \tag{4.15}$$

where α, β, γ, and η are the hyper-parameters to balance these four losses. In our current implementation, we set all of them to 1. We optimize the above loss through RMSprop optimizer. By this design, we minimize both the loss of the posture (orientation and location) and the loss of the posture's changing rate.

4.4.5 Labeled Data Collection

To train *RTAT*'s model, we need the orientation and location labels of the smartwatch for the IMU readings.

Why Meta Quest 2 is chosen for labeled data collection? To collect the labeled data, we may use Azure Kinect [142], VICON motion capture system [143], or a VR system.

Azure Kinect is not able to capture the pronation/supination (inward/outward rotation) of the forearm due to the limitation of its current algorithm, which means it can only capture 2 DoF orientation of the wrist [144].

VICON motion capture system can provide accurate arm posture tracking. However, VICON cameras are expensive (more than $3000 each). Multiple cameras from different points of view in a room are needed to provide high-precision tracking. For instance, WiPose [136] uses 21 VICON cameras to provide labeled locations for WiFi-based joint tracking. Moreover, once the VICON system is installed in one room, it is costly to move it to another room.

Meta Quest 2 is the most advanced all-in-one VR system that provides accurate orientation and location tracking of its controllers at a low cost ($299 for 64GB). It can be used in any indoor environment with ignorable setup efforts. Thus, we use a Meta Quest 2 to collect the labels.

Is Meta Quest 2 accurate enough for tracking? As shown in Figure 4.5 (a), the headset of Meta Quest 2 is embedded with four cameras on its four corners. It adopts Oculus Insight, the cutting-edge VR technology that leverages computer vision algorithms and visual-inertial simultaneous localization and mapping (SLAM) to compute 6-DoF postures (3-DoF orientation and 3-DoF location) [145]. Specifically, Oculus Insight combines information from multiple IMUs in its headset and controllers, i.e. ultra-wide-angle cameras in the headset and infrared LEDs in the controllers, to jointly track their orientation and location. Controlled experiments [146, 147, 148] have demonstrated that the orientation and location tracking errors of Meta Quest 2 are smaller than 0.85° and 0.7 cm respectively.

Labeled data in the VR Reference Frame (VRF). The orientation and location of the VR headset and controllers are tracked in VRF. Figure 4.5(a) shows the coordinates of WRF and VRF. The origin of VRF is the midpoint of the headset. The VRF can be established every time the VR system is used based on the initial position of the headset. We collect the orientation and location data

of one VR touch controller in VRF. Our goal is to use the measurements of the VR controller to calculate the orientation and location of the smartwatch. *RTAT* uses the smartwatch's IMU sensor readings measured in WRF to infer its orientation and location in VRF.

Once the VRF is established, it will not change until the next setup, even if the headset moves. As long as the user wears the headset properly, VRF can represent the body's reference frame. Therefore, *RTAT* tracks the arm movements relative to the user's body. VRF-based labeled data also offers two advantages. 1) We do not require users to stand in the exact same position when they collect labeled data or use *RTAT* for arm tracking since *RTAT* is focused on arm movements relative to the user's body. 2) We can combine the labeled data collected from the same user at different times. 3) We can also combine labeled data from multiple users to train a general *RTAT* model. Experiments in Section 4.6.2 show that our general model provides high performance across different users, including the users whose data has not been used for training.

Two challenges to collect labeled data with VR controllers.

- *Using the orientation and location of the VR controller to derive those of the smartwatch.* If we simply wear a smartwatch and hold one VR controller in the same hand to collect data, the relative orientation and location between these two devices may vary when a user moves her/his wrist. As shown in Figure 4.1, we bind the smartwatch and the VR controller onto a rigid frame. This is constructed with four sticks to ensure the two devices are static relative to each other. The VR system provides the orientations and locations of the controller, while we need those of the smartwatch.

- *Time Synchronization of VR system and smartwatch.* The smartwatch and VR system are based on two different time clocks. They need to be well synchronized to provide data at the same timestamps.

Orientation Label.

We use T_{wv}, a 3×3 orthogonal rotation matrix introduced in Section 4.3.1 to denote the smartwatch's orientation in VRF. T_{wv} will be used as the orientation label. Similarly, T_{cv} represents the orientation of the VR controller in VRF. T_{cv} is measured by the VR system. Our goal is to calculate T_{wv} from T_{cv}. To do so, we need the watch's orientation relative to the controller T_{wc}, i.e., $T_{wv} = T_{wc} \cdot T_{cv}$. T_{wc} is a constant matrix, as the controller and the watch are fixed to each other. However, it is impossible to physically measure T_{wc}.

Fortunately, the orientation of a smartwatch in GRF, T_{wg}, can be accurately calculated when the smartwatch is static using gravity and magnetic North as direction anchors. If the transformation from VRF to GRF T_{vg} is known, we can transform the orientation of the smartwatch in VRF T_{wv} to the orientation of the smartwatch in GRF T_{wg}, and then get T_{wc}. Leveraging this resource, we update the strategy into $T_{wg} = T_{wv} \cdot T_{vg} = T_{wc} \cdot T_{cv} \cdot T_{vg}$. All four matrices are 3×3 orthogonal matrices. Since VRF does not change each time it is established, the transformation from VRF to GRF T_{vg} is a constant but unknown matrix. We perform a set of static gestures to acquire multiple T_{wg}, and with the known T_{cv}, we jointly determine T_{wc} and T_{vg}. Finally, we apply T_{wc} to T_{cv} to get T_{wv}.

T_{wg} can be acquired via its inverse, $T_{wg} = T_{gw}^{-1}$. For T_{gw}, the three axes of GRF represented in WRF, $\overrightarrow{x_{gw}}$, $\overrightarrow{y_{gw}}$, $\overrightarrow{z_{gw}}$, can all be determined when the watch is static. $\overrightarrow{z_{gw}}$ is the opposite direction of gravity $\overrightarrow{Gr}$. $\vec{x}$ is the horizontal direction of Geo-North, which is perpendicular to $\overrightarrow{Gr}$. Thus $\overrightarrow{x_{gw}} = \frac{\vec{x'}}{\|\vec{x'}\|}$, and $\vec{x'} = \overrightarrow{Mr} - (\overrightarrow{Mr} \cdot \overrightarrow{z_{gw}})\overrightarrow{z_{gw}}$, where $\overrightarrow{Mr}$ is the magnetometer reading. The smartwatch we used applies the left-hand-rule, thus we also apply left-hand-rule to our reference frame, so that $\overrightarrow{y_{gw}} = \overrightarrow{x_{gw}} \times \overrightarrow{z_{gw}}$. The inverse of an orthogonal matrix is its transpose. Then,

$$T_{gw} = \begin{bmatrix} \overrightarrow{x_{gw}} \\ \overrightarrow{y_{gw}} \\ \overrightarrow{z_{gw}} \end{bmatrix}, \quad T_{wg} = T_{gw}^{-1} = \begin{bmatrix} \overrightarrow{x_{gw}} \\ \overrightarrow{y_{gw}} \\ \overrightarrow{z_{gw}} \end{bmatrix}^{-1} = \begin{bmatrix} \overrightarrow{x_{gw}} \\ \overrightarrow{y_{gw}} \\ \overrightarrow{z_{gw}} \end{bmatrix}^{T} \tag{4.16}$$

Recall that in $T_{wg} = T_{wc} \cdot T_{cv} \cdot T_{vg}$, both T_{wc} and T_{vg} do not change, and T_{wg} and T_{cv} are dynamic but can be acquired. Then we must determine the two constant

matrices T_{wc} and T_{vg}. If we collect N static periods, we have N pairs of ($T_{wg}(i)$, $T_{cv}(i)$). We define the calculation loss as:

$$Loss = \frac{\sum_{i=1}^{N} diff(T_{wg}(i), T_{wc} \cdot T_{cv}(i) \cdot T_{vg})}{N} \tag{4.17}$$

where $diff(T_x, T_y)$ returns the minimum degree of rotation between T_x and T_y.

A rotation of reference transformation is a process of 3-DoF. We have two unknown rotations, meaning there are six variables that need to be determined. Fortunately, we know the VRF and the GRF share the same **'up'** direction, which reduces the DoF of T_{vg} to 1. This leaves 3+1=4 variables to be determined. We jointly determine T_{wc} and T_{vg} by searching for the best pair of them that returns the minimum calculation loss in Equation (17). The minimum loss is 0.15°. We then apply T_{wc} we get to every T_{cv} to calculate T_{wv}.

Location Label

We can obtain the location of the VR controller represented in VRF L_{c-VRF} from the VR system. To derive the location of the smartwatch L_{w-VRF} from L_{c-VRF} in VRF, we need to obtain the vector points from the center of controller to the center of smartwatch in VRF L_{wc-VRF}. Then $L_{w-VRF} = L_{c-VRF} + L_{wc-VRF}$. Since the relative positions of smartwatch and the VR controller to each other are static, the vector L_{wc-VRF} is constant. However, it is impossible to manually measure this vector. As shown in Figure 4.5(b), we know that the center of the smartwatch is the center of its body, but we cannot exactly pinpoint the center of the VR controller.

Fortunately, we derived the transformation from WRF to VRF T_{wv} in Section 4.4.5, if we can acquire the vector in WRF, L_{wc-WRF}, then $L_{wc-VRF} = L_{wc-VRF} \cdot T_{wv}$. We thus design an experiment to derive the vector in WRF. If the locations of the smartwatch and the VR controller in WRF are known, the vector in WRF can be derived. The locations of the VR controller in WRF can be obtained by transforming the locations of VR controller in the VRF, because the transformation between the two references are known in Section 4.4.5. However, the smartwatch does not provide any information about its own location in WRF.

To eliminate the uncertainty of the location of the smartwatch, we do not move but rotate the center of the smartwatch along its axes, so that its location does not change. The location trace of the VR controller in VRF is a circle.

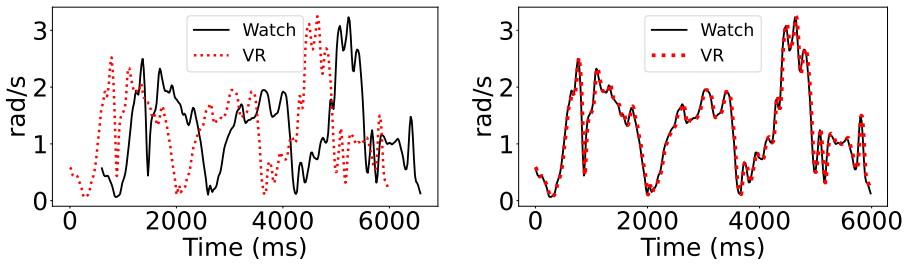
As shown in Figure 4.5(b), we set up a spinning platform, whose rotation axis is vertical. The platform rotates automatically with electric power. We carefully adjust the smartwatch to a certain posture so that the Z-axis of the smartwatch is exactly on the rotation axis of the spinning platform. In this situation, when we rotate the platform, the location of the smartwatch does not change, from the point of the watch (WRF), the location of the controller does not change, while the location trace of the controller in VRF is a circle. The center of the circle is on the Z-axis of the watch. As shown in Figure 4.5(c), the radius of the circle is the length of the projection of the vector L_{wc-WRF} on the X-Y plane in WRF. We use L_c^{X-Y} to denote the collected locations of the controller, and L_w^{X-Y} to denote the location of the smartwatch (center of the circle). Then in WRF, we acquire the vector from L_c^{X-Y} to L_w^{X-Y}, $L_{wc-WRF}^{X-Y} = L_w^{X-Y} - L_c^{X-Y}$, which is the projection of the vector on the X-Y in WRF.

With the same principle, we can get L_{wc-WRF}^{Y-Z} and L_{wc-WRF}^{X-Z}. Then we search for a vector L_{wc-WRF} in a small 3D space ($20cm^3$), whose projection on the Y-Z, X-Z and X-Y best matches L_{wc-WRF}^{Y-Z}, L_{wc-WRF}^{X-Z}, and L_{wc-WRF}^{X-Y}, respectively. L_{wc-WRF} will be the vector we want to derive in WRF. The best match loss is 0.14 cm. We then apply this vector to every data sample and acquire the true location of the watch in VRF:

$$L_{w-VRF} = L_{c-VRF} + L_{wc-VRF} = L_{c-VRF} + L_{wc-WRF} \cdot T_{wv} \tag{4.18}$$

Time Synchronization of VR system and Smartwatch

Since the VR controller and the smartwatch are static to each other. Their angular movements relative to the world should always be the same. To find the time bias between these two systems, we make use of the magnitude of angular velocity measured by the VR system and the smartwatch, $\|\omega_{vr}\|$ and $\|\omega_{watch}\|$. $\|\omega_{vr}\|$ and $\|\omega_{watch}\|$ should match the most when the two systems are well synchronized. As shown in Figure 4.6 (a), Given the data sequences from the VR system



(a) Before synchronization. (b) After synchronization.

Figure 4.6: Time synchronization of smartwatch and VR.

and the smartwatch, we add a time compensation t_c on the timestamp of VR readings and find the t_c that returns the minimum difference between $\|\omega_{vr}(t + t_c)\|$ and $\|\omega_{watch}(t)\|$. Finally, we synchronize the VR system with the smartwatch by adding t_c to all of the VR readings. Figure 4.6 (b) demonstrates the data traces after synchronization.

Alignment of Time Series Data From the Two Systems.

The smartwatch and the VR system output data at different timestamps with different frequencies. The smartwatch outputs data with an average interval of 20 ms. The three IMU sensors may output data at different timestamps. The VR system outputs data with an average interval of 15 ms. We need to align the four time series data from the two systems.

Assume we want a data sample from sensor S at time t, while the sensor does not have outputs at this specific timestamp. We search for the two nearest data samples t_i and t_{i+1} that enclose time t, t_i ¡ t ¡ t_{i+1}. Sensor S outputs $S(t_i)$ at t_i, and $S(t_{i+1})$ at t_{i+1}. We then perform a linear interpolation to acquire a virtual data sample $S(t)$ at time t , with t_i, t_{i+1}, $S(t_i)$ and $S(t_{i+1})$:

$$S(t) = \frac{S(t_{i+1}) \cdot (t - t_i) + S(t_i) \cdot (t_{i+1} - t)}{t_{i+1} - t_i} \tag{4.19}$$

We prepare a timeline with a fixed sampling frequency f, and perform linear interpolation at every wanted timestamp. We set f as 50Hz, which is close to the smartwatch's sampling frequency, and lower than VR system's frequency of 66 Hz. By this step, the data streams from the two systems are aligned.

4.5 Implementation

Data Collection. Data collection consists of IMU data from the smartwatch and ground truth data from the VR system.

To read the IMU data from the smartwatch, we develop and install an App into the watch. The *SensorManager* API in Android is used to read data from sensors. To collect the IMU readings, we establish a server based on Apache Tomcat and Eclipse. The application installed in the smartwatch capture the IMU sensor data of the smartwatch and sends the data automatically to the server via *http* by finding the IP address of the server when the application is running. The server connects to a MySQL database to store data. The application and data collection server are implemented in Java.

To read the orientation and location of the VR controller, we develop an application based on unity in C# and install it to the VR system. Since the VR device has sufficient disk space, we save the data locally. The orientation and location readings of the VR controller are saved into a *CSV* file automatically created by the application when it is running.

Multitask Model. Our multitask model is implemented and trained by Keras in Python. Each BiLSTM layer has 32 units, and each FC layer has three units. The optimizer is RMSprop, with a learning rate of 0.0001. The training epoch is 100, the batch size is 128, and the number of time steps considered by the BiLSTM is 32. The computer we use to collect data and develop our framework is an Alienware Aurora R7, with a Intel Core i5 8400 CPU (6-core) and NVIDIA GeForce GTX 1070 GPU (8GB memory).

TensorFlow Lite model. To run *RTAT* on smartphones, it is necessary to convert the well-trained TensorFlow model that was developed on a desktop computer into a TensorFlow Lite model that is capable to run and do inference on mobile devices. This can be achieved using the TensorFlow Lite Converter. First, the TensorFlow model is frozen into a concrete function. This process fixes the input size, which is specified as a tensor, and also specifies a *tf.function* that can be saved into a *.pb* file, which is the preferred input format for the TensorFlow Lite Converter. Next, the TensorFlow Lite Converter is used to convert the saved

model into a *FlatBuffer* file with a *.tflite* extension. This file can be imported and used on mobile devices running Android or iOS, as well as some embedded devices and microcontrollers.

4.6 Evaluation

In this section, we introduce our experiment settings. We demonstrate the performance and performance decomposition of *RTAT*, the performance of different applications, and the system overhead.

4.6.1 Experimental Settings

Platform and Devices. The platform we used to do evaluation experiments is introduced in Section 4.5. As introduced in Section 4.3, the smartwatch we use is Fossil Gen 5. It includes an LSM6DSO 3D accelerometer + 3D gyroscope and an AK0991X magnetometer. Similar chips are in many other commercial devices, including Samsung Galaxy A52, Samsung Galaxy S22 Ultra, Samsung Galaxy Note 10, Xiaomi Mi 10T Pro, Xiaomi POCO X3 Pro, Oneplus 7, MiWatch, TicWatch Pro 3, Sony Xperia 1 III, Motorola moto g fast, etc. We still use the Meta Quest 2 VR system to provide the ground truth for the evaluation.

Evaluation Metrics. We use the following metrics to quantify the performance of *RTAT* in estimating orientation and location.

- 3D orientation error. It is measured as the minimum degree of rotation required to align the estimated orientation to the ground truth orientation.
- 3D location error. It is the Euclidean distance between the estimated location and the ground truth location.

Baselines for Orientation Estimation. We compare the orientation estimation error of *RTAT* to two baselines.

- MUSE [113]: The state-of-the-art conventional sensor fusion approach for estimating device orientation.

- IDOL [118]: IDOL is based on Extended Kalman Filter (EKF). The prediction model of its EKF uses an RNN to predict orientation. The measurement model of its EKF is based on gyroscope readings integration.

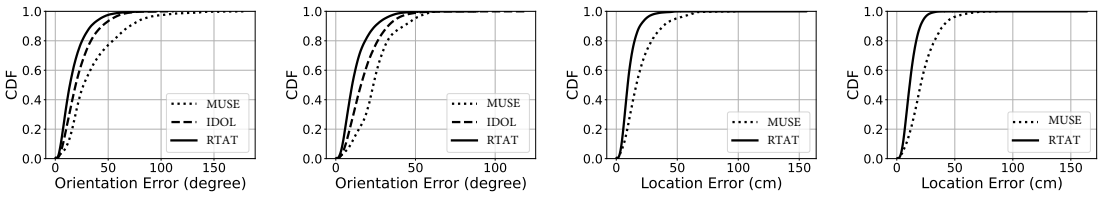
Baselines for Location Estimation. We compare the smartwatch's location estimation error of *RTAT* with two baselines.

- MUSE [113]: It estimates the location using Particle Filters.
- ArmTroi [104]: It estimates location using HMM. As stated in [104], ArmTroi provides real-time computation on smartphones when the sampling rate is 5 Hz, but it is less accurate than MUSE. We will compare the accuracy of *RTAT* with MUSE, and the latency of *RTAT* with ArmTroi.

Table 4.3: Statistical analysis on users' test data

Hallway / Room	Gravity Error(°)	Magnet Deviation(°)	Static Moments	Gravity Opportunity(%)	Speed (m/s)
User1	8.83 / 9.54	13.34 / 5.69	85 / 71	5.92 / 4.74	0.55 / 0.56
User2	8.05 / 6.23	21.50 / 6.82	141 / 124	8.97 / 10.63	0.59 / 0.48
User3	7.15 / 5.37	45.41 / 5.10	165 / 216	10.52 / 13.3	0.39 / 0.30
User4	10.57 / 20.47	26.13 / 9.0	34 / 40	3.37 / 3.49	0.66 / 0.98
User5	7.69 / 9.56	26.61 / 7.15	75 / 17	5.62 / 2.61	0.42 / 0.57
Mean(User1-5)	8.46 / 10.23	26.60 / 6.75	100 / 93.6	6.88 / 6.95	0.52 / 0.58
User6 (new)	7.21 / 4.54	24.71 / 5.91	110 / 351	8.22 / 30.77	0.61 / 0.29
User7 (new)	11.12 / 14.97	29.91 / 8.90	153 / 233	11.53 / 18.59	0.67 / 0.86
User8 (new)	12.61/10.25	26.15 /6.19	79/74	5.8/6.4	0.73 / 0.57
User9 (new)	23.92/16.94	36.08/9.10	17/5	3.0/4.5	1.27/0.78

Dataset. We collect data from five users (two females and three males) to train and test our model. We collect data at two places, the hallway and the room. At each place, we collect 50-minute data from each user. The dataset collected from the hallway includes 754,688 samples, and the dataset collected from the room



(a) Hallway (b) Room (c) Hallway (d) Room

Figure 4.7: Orientation error and location error at hallway and room.

includes 761,856 samples. We divide each dataset into train and test data at a ratio of 4:1. At each place, we use the data from all 5 users to train a general model and test its performance on all 5 users. The test data for *RTAT* and the baselines is the same. We also collect data from four new users (two females and two males) to test the model. The model has never seen their data during training. We collect 10-minute data (around 30000 samples) from each user at each place.

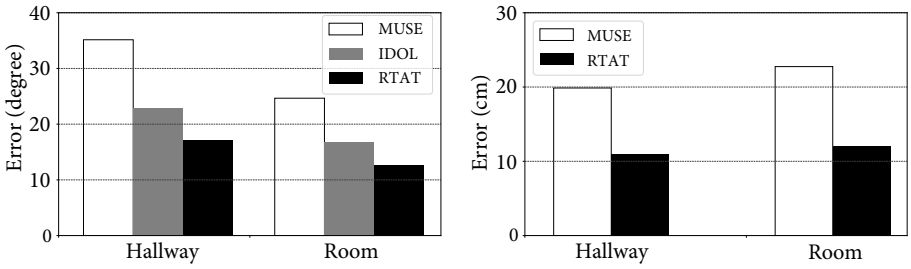
Dataset Collection Scenarios. The age of the nine users ranges from 21 to 32, i.e., 30, 28, 23, 26, 21, 32, 32, 22, and 22, respectively. Their height ranges from 158 cm to 182 cm, i.e., 168 cm, 170 cm, 182 cm, 177 cm, 158 cm, 175 cm, 165 cm, 179 cm, and 160 cm, respectively. We ask users to move their arms freely at their normal movement speed. They perform random arm gestures or daily gestures, including driving, drinking, writing, exercising, push and pull, drawing, and so on.

4.6.2 *RTAT* Performance

In this subsection, we compare the performance of *RTAT* to the baseline methods in orientation and location estimation. The performance of *RTAT* is evaluated from the overall performance, the performance over time, the performance of different users and the performance of new users.

Overall Performance

Figure 4.8 depicts the overall orientation and location estimation error at the two testing places. From Figure 4.8 (a), in the hallway, the orientation error of MUSE, IDOL and *RTAT* are 35.13°, 22.87° and 17.19°, respectively averaged over the whole test data. *RTAT* decreases the orientation error by 24.84% and 51.07%



(a) Orientation error (b) Location error

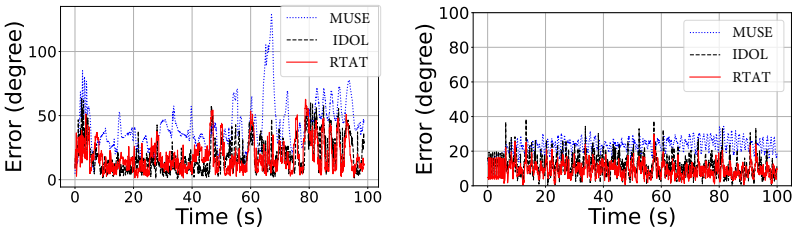
Figure 4.8: Overall orientation and location error.

compared to IDOL and MUSE, respectively. In the room, the orientation error of MUSE, IDOL and *RTAT* are 24.66°, 16.86° and 12.67°, respectively on the whole test data. *RTAT* decreases the orientation error by 24.85% and 31.63% compared to IDOL and MUSE, respectively. All systems perform better in the room than the hallway as magnet deviation in the hallway is larger.

The neural network of IDOL outputs quaternions directly. The orientation error defined as the norm quaternion difference between its predicted quaternions and the ground truth quaternions. Although its orientation estimation error is not high, we found its predicted quaternions do not meet Equation (4.2). Because its loss design ignores this constraint. The neural network finds a way to minimize the loss but it does not realize the constraint.

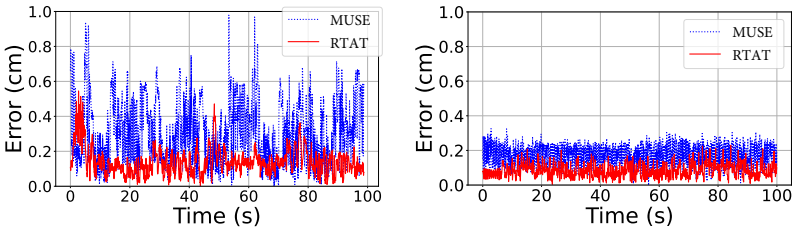
Figure 4.8 (b) shows the average location estimation error at the two different places. In the hallway, the location error of MUSE and *RTAT* are 19.87 cm and 10.93 cm, respectively. *RTAT* decreases the location error by 45% compared to MUSE. In the room, the location error of MUSE and *RTAT* are 22.75 cm and 12.09 cm, respectively. *RTAT* decreases the error by 46.9% compared to MUSE.

Figure 4.7 demonstrates the CDF of *RTAT*'s orientation and location error compared to the baselines at the two places. *RTAT* consistently outperforms the baselines for orientation and location estimation. From Figure 4.7 (a), for 80% of cases, the orientation error of *RTAT* is less than 25 ° in the hallway. Similarly, Figure 4.7 (b) shows the orientation error of *RTAT* is less than 25 ° for 90% of cases in the room. The CDF of location error at two places are similar, for 80% of cases, the location error is smaller than 25 cm.



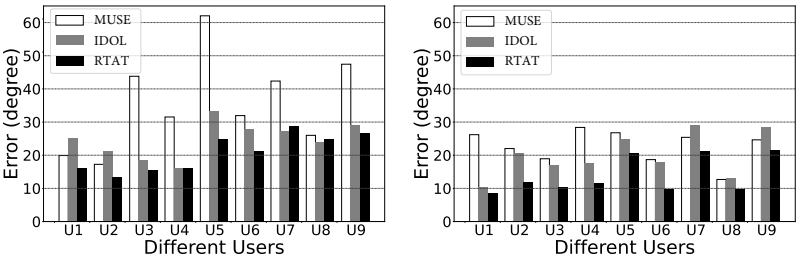
(a) Orientation at hallway (b) Orientation at room

Figure 4.9: Orientation error along with time in the hallway and room.



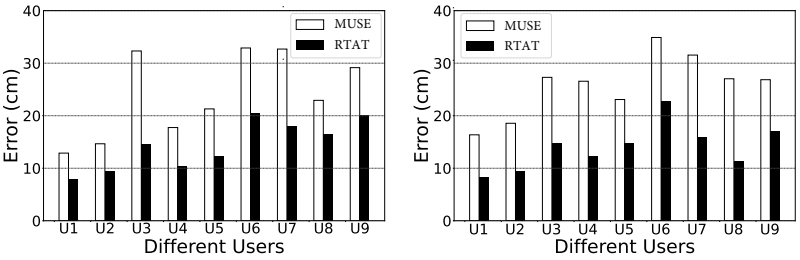
(a) Location at hallway (b) Location at room

Figure 4.10: Location error along with time in the hallway and room.



(a) Orientation at hallway (b) Orientation at room

Figure 4.11: Orientation of different users in the hallway and room.



(a) Location at hallway (b) Location at room

Figure 4.12: Location error of different users in the hallway and room.

Performance Along with Time

Figure 4.9 and Figure 4.10 show the orientation and location error of *RTAT* and baselines along with time at the two places. They are plotted based on 100-second data. From Figure 4.9 and Figure 4.10, we show *RTAT* performs better than MUSE and IDOL on orientation estimation and exhibits better location estimation than MUSE over time. *RTAT* is also more stable than the baseline orientation and location estimation methods.

Performance of Different Users

Figure 4.11 and Figure 4.12 plot the orientation and location error of *RTAT* and the baseline methods on different users at the two places. We collect both training and test data for *RTAT* from the U1-U5, while *RTAT* has never seen the data of the U6-U9 during training. For each user, *RTAT* exhibits better than MUSE and IDOL on orientation estimation and exhibits better than MUSE on location estimation. In comparison with MUSE, *RTAT* achieves more consistent performance across all users.

Performance of New Users

The U6-U9 in Figure 4.11 and Figure 4.12 depict the orientation and location of new users, whose data was omitted from the training phase. For the new users, the performance of *RTAT* does not degrade much, indicating its applicability to new users. Moreover, *RTAT* still outperforms baselines at both orientation and location estimation for the four new users. Table 4.3 shows the statistical analysis on the users' test data at the two places. The motion speed of users will influence the prediction accuracy.

4.6.3 Performance Decomposition of *RTAT*

We test the benefits provided by the three components of *RTAT* (i.e., multitask learning, attention and smooth loss) on the whole test data at the two places.

Table 4.4: Average orientation and location estimation error of different models at hallway and room.

Model	Hallway		Room	
	Orientation Error(°)	Location Error (cm)	Orientation Error(°)	Location Error(cm)
RTAT _Orientation_Only	21.00	\	13.23	\
RTAT _Location_Only	\	18.40	\	13.44
RTAT _Multitask	19.09	14.39	13.00	12.70
RTAT _Multitask_Att	18.20	11.85	12.86	12.50
RTAT _Multitask_Att_Smooth	17.19	10.93	12.67	12.09

Table 4.5: Analysis on smoothness of orientation and location at two places.

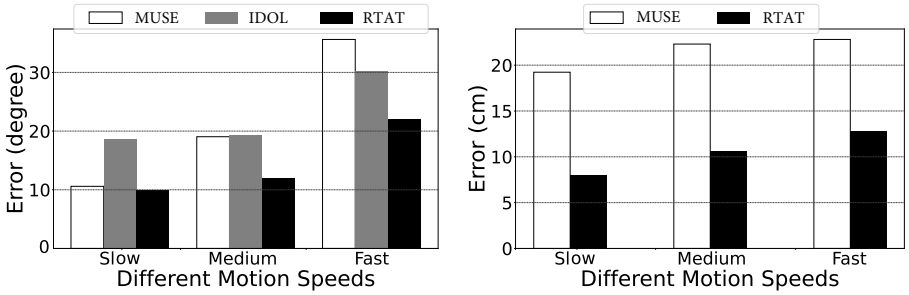
	Hallway				Room			
	Orientation		Location		Orientation		Location	
Model	Error (°)	Error_std	Error (°)	Error_std	Error (°)	Error_std	Error (°)	Error_std
RTAT w/o Smooth Loss	1.28	2.91	1.58	2.43	1.30	2.68	1.43	2.0
RTAT w/ Smooth Loss	1.05	2.71	1.27	1.84	0.92	1.91	1.39	2.0

Multitask Learning. We first compare a multitask learning model against two single-task models, an orientation-only model and a location-only model. Both of the models are trained with the data from all three IMU sensors. Table 4.4 shows that while multitask learning decreases the orientation and location error by 9.1% and 21.8% in the hallway, it decreases the orientation and the location error by 1.7% and 5.5% in the room. Multitask learning reduces the model training time and the model inference overhead by half in comparison to implementing two single-task learning methods.

Attention Mechanism. We then compare *RTAT* _Multitask_Att to *RTAT* _Multitask. Table 4.4 demonstrates that the attention mechanism decreases the orientation and location error by 4.7% and 17.7% in the hallway. Compared to the hallway, the performance gain from the attention mechanism in the room is marginal. It only decreases the orientation and location error by 1.1% and 1.6% in the room. This is because the gravity error at these two places is very similar, but the magnetic field in the room is more stable than in the hallway. The attention mechanism contributes less in the room.

Smooth Loss. Finally, we show the benefits of smooth loss. The primary purpose of smooth loss is not to reduce the orientation and location error but to improve the smoothness of orientation and location tracking and makes posture tracking more realistic. We evaluate the orientation/location smoothness error by comparing the orientation/location difference of predicted ones to the orientation/location difference of ground truth between two consecutive timestamps. We also calculate the standard deviation of the orientation/location smoothness error.

Table 4.4 shows that using smooth loss decreases the orientation and location error by 5.5% and 7.8% in the hallway, and 1.5% and 3.3% in the room. Table 4.5 provides the orientation/location smoothness error and the standard deviation of smoothness error at the two places. The *RTAT* _Multitask_Att model is denoted as *RTAT* w/o Smooth Loss. From Table 4.5, with smooth loss, *RTAT* improves the orientation smoothness and location smoothness by 18% and 19.6% in the hallway, and by 29.2% and 2.8% in the room. Additionally, the standard deviations of orientation/location smoothness error decrease under almost all scenarios.



(a) Orientation performance. (b) Location performance.

Figure 4.13: Performance under different motion speeds at room.

4.6.4 Performance of Different Applications

Table 4.6: Statistic on different motion speed

Room	Speed (m/s)	Gravity Error (degree)	Magnet Deviation (degree)	Gravity Opportunities (%)
Slow	0.35	4.43	5.59	25.06
Medium	0.51	8.14	5.86	7.21
Fast	1.01	20.78	9.0	2.36

For different application scenarios, user motion speed may vary. When a user performs gym gestures, such as front raise and chest fly, motion is slow. When a user performs some daily gestures like making a call, pushing and pulling, the motion speed is moderate. When a user performs some AR games, motion is fast. We ask one user to perform gestures with different motion speeds in the room. Table 4.6 shows the statistical analysis. Figure 4.13 plots the orientation and location error with different motion speeds. As motion speed increases, the error of all systems increases. However, *RTAT* always performs better than the baseline systems, demonstrating its stability.

4.6.5 System Overhead

Location latency on desktop

We evaluate the location latency of different systems on an Alienware Aurora R7 desktop. It takes MUSE, ArmTroi and *RTAT* 5427.05 ms, 2337.50 ms and 0.1633 ms respectively to process one-second data (50 samples). In MUSE [113] and ArmTroi [104], they use 10 Hz and 5 Hz respectively. *RTAT* is significantly faster than two conventional approaches, since they are based on searching algorithms. The deep learning architecture of *RTAT* is very lightweight.

Inference overhead on mobile devices

We also run *RTAT* on two commercial smartphones, Samsung S9 and Google Pixel3. We convert a well-trained TensorFlow model to a TensorFlow Lite model capable of inference on mobile devices by TensorFlow Lite Converter. Then we test the overhead of *RTAT* on the two smartphones. As shown in Table 4.7, the execution latency, memory usage and CPU usage of *RTAT* are low at both devices. It is even much faster than MUSE and ArmTroi running on the desktop.

Energy Consumption on Mobile Devices

We measure the energy consumption of *RTAT* on Samsung S9 by Monsoon monitor in Figure 4.14. The IMU data measured by the smartwatch is transmitted to the smartphone for processing via Bluetooth. We run the inference of *RTAT* for about 2 seconds, as indicated as the inference segment in Figure 4.14. The average working current (mA) in idle state with screen on is about 221 mA. The average working current on the *RTAT* inference state is about 357 mA. The average working current on playing music state is about 498 mA. *RTAT* only increases the working current by 136 mA, including both running the model and receiving IMU data via Bluetooth; whereas, playing music increases the current consumption by 277 mA. The power consumption of *RTAT* is less than half of playing music.

Similar to previous works [106, 104, 113], *RTAT* requires the smartwatch to continuously collect IMU sensor data and transmit the data to the smartphone

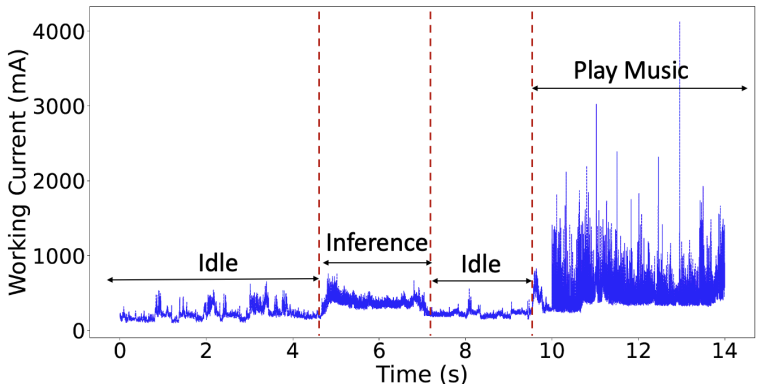


Figure 4.14: Energy Consumption on Samsung S9.

while the system is in use. This process is power-consuming. Based on our experiments, it drains the watch's battery in around 3.5 hours. However, this limitation is not unique to our system. All the applications that require smartwatches to transmit IMU readings suffer from the same problem due to the limited battery power of smartwatches. We expect smartwatches will have more powerful batteries in the future.

Table 4.7: Inference overhead of *RTAT* on smartphones

	Samsung S9	Google Pixel3
Latency(ms)	2.98	1.82
Memory Usage (MB)	4.8	5.1
CPU Usage(%)	17	13

4.7 Discussion

Data efficiency. We develop our system based on supervised learning. Collecting labeled data is labor-intensive. To train a generalized and robust model, the dataset should cover a wide variety of gestures, users, and environments. To reduce the labeled data, we may explore self-supervised training to capture temporal relations and feature distributions in IMU data in the future [127].

Smartwatch-wearing angles. We indirectly study the different smartwatch-wearing angles in this work. Each user collects the data several times. We do not require the users to wear the smartwatch at the same angle. Even the same user may wear the smartwatch slightly differently each time. Different users also wear smartwatches at different angles. Our system is tolerant to wearing the smartwatch with slightly different rotation angles. Our data collection scenario is to accurately reflect how people wear a smartwatch in daily life, thus we do not consider wearing the watch with significantly different rotation angles on the wrist when we build the dataset. Inference accuracy drops when we apply the model to data collected from a smartwatch worn at significantly different rotation angles on the wrist. To accommodate this limitation, we may collect training data by wearing the watch with different rotation angles on the wrist, summarizing the data features with different rotation angles, and exploring domain shift in future work.

Generalization. It refers to two perspectives: the generalization across different users and the generalization across different places. The accuracy of our system drops slightly for new users because the data from new users has not been seen by the model during the training process. To improve the model generalization across users, training data could be collected from users with broader age and height ranges. To improve the model generalization across different places, transfer learning may be used to transfer the DNN model learned by the data at one place to a new model for another place. We leave generalization as future work.

4.8 Conclusion

This chapter presents *RTAT*, the first 3D human wrist tracking system via a smartwatch based on multitask deep learning. We design an attention mechanism and a smooth loss on top of the multitask learning network to improve its performance. *RTAT* is lightweight and supports real-time tracking on smartphones with high sampling frequency. We collect a large-scale dataset using our customized labeled data measurement system. Extensive experimental results show *RTAT* achieves higher accuracy and lower latency when compared with baseline methods.

Chapter 5

Adaptive Orientation Estimation Piloted by Deep Reinforcement Learning and Envision

Adaptive Orientation Estimation. The orientation of a mobile device is a crucial input for many applications, including augmented reality, gaming, and navigation. However, accurately estimating the device's orientation over a longer period of time is challenging due to various factors like sensor noise, drift, and external magnetic interference. To address this problem, researchers have proposed classical sensor fusion and supervised learning methods for device orientation estimation. Sensor fusion combines data from multiple IMU sensors (i.e., accelerometer, gyroscope, magnetometer) to obtain more accurate estimates, while deep learning techniques can learn from large amounts of data to improve orientation estimation. However, the generalization ability of supervised learning is limited, and the adaptive ability of classical sensor fusion methods is limited. To overcome these limitations, we integrate a classical complementary filter with a deep reinforcement learning (DRL) framework to leverage the benefits from both methods. The DRL agent learns dynamic sensor characteristics to improve orientation estimation over time. It then continuously pilots the complementary filter to adjust its parameters based on the change sensor noise to improve the adaptability of the complementary filter and the generalization ability of the supervised learning.

Envision. In this thesis, we have investigated the data processing by deep learning on individual edge devices and collaborated edge devices. The systems we developed including video analytics, driving anomaly detection, arm posture tracking, and device orientation tracking. Based on the insights from those systems, in this chapter, we present the future directions of what we plan to study based on common problems - saving labeled training data for IoT applications.

Collecting labeled data for IoT systems can be challenging due to several reasons. Firstly, IoT devices generate vast amounts of data, and labeling this data manually can be time-consuming and expensive. In many cases, it may not be feasible to label all the data generated by IoT devices due to resource constraints. Secondly, labeling IoT data can also be challenging due to the nature of the data. IoT data can be complex, heterogeneous, and noisy, making it difficult to define labeling criteria and ensure labeling consistency. Thirdly, in some cases, labeling may require human expertise and domain knowledge. For example, in a healthcare application, labeling medical data requires specialized knowledge and expertise that may not be available to the labeling team.

Contrastive learning has recently been applied to IoT applications, including human activity recognition, 3D pose estimation and silhouette generation [149, 150]. Contrastive learning is a machine learning technique that learns representations of data by contrasting different views of the same data. The goal of it is to learn a representation of data that is invariant to various transformations, such as rotation, scaling, and translation. In contrastive learning, the model is trained to distinguish between positive pairs (two views of the same data) and negative pairs (two views of different data). This is done by mapping each view of the data to a low-dimensional embedding space, where views of the same data are close to each other, and views of different data are far apart.

Using contrastive learning, we can train a model to learn a representation of the IMU data that is invariant to various transformations, such as different orientations and movements. This involves contrasting different views of the same IMU data, such as different time windows or sensor configurations, to learn a low-dimensional embedding space that captures the essential features of the data. However, using

contrastive learning for IMU data processing requires careful consideration of the data preprocessing steps, the model architecture, and the choice of contrastive loss function. These are some of the key aspects that we will need to focus on to apply the technique effectively in our system.

In the future, we plan to investigate the optimal data preprocessing steps that can be taken to ensure that the IMU data is appropriately processed before being used in contrastive learning. We will also explore different model architectures and experiment with different types of contrastive loss functions to determine which ones are most effective in learning the essential features of the IMU data. By paying close attention to these factors, we hope to develop a more effective approach to processing IMU data using contrastive learning, which can help us to reduce the amount of labeled training data for our system and improve its overall performance.

Bibliography

[1] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[2] Weisong Shi and Schahram Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, 2016.

[3] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE access*, 8:85714–85728, 2020.

[4] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[5] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. Real-time video analytics: The killer app for edge computing. *Computer*, 2017.

[6] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *ACM MobiCom*, 2019.

[7] Dimitris Chatzopoulos, Carlos Bermejo, Zhanpeng Huang, and Pan Hui. Mobile augmented reality survey: From where we are to where we go. *Ieee Access*, 5:6917–6950, 2017.

[8] Joseph Redmon. Darknet: Open source neural networks in c. `http://pjreddie.com/darknet/`, 2013–2016.

[9] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *ECCV*, 2016.

[10] Jiajun Wu, Joshua B Tenenbaum, and Pushmeet Kohli. Neural scene derendering. In *IEEE CVPR*, 2017.

[11] Xukan Ran, Haolianz Chen, Xiaodan Zhu, Zhenming Liu, and Jiasi Chen. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE INFOCOM*, 2018.

[12] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *ACM MobiSys*, 2016.

[13] Loc N Huynh, Youngki Lee, and Rajesh Krishna Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *ACM MobiSys*, 2017.

[14] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[15] Biyi Fang, Xiao Zeng, and Mi Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *ACM MobiCom*, 2018.

[16] Jianbo Shi and Carlo Tomasi. Good features to track. Technical report, Cornell University, 1993.

[17] Bruce D Lucas, Takeo Kanade, et al. An iterative image registration technique with an application to stereo vision. In *IJCAI*, 1981.

[18] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, 2015.

[19] Videezy. `https://www.videezy.com/free-video/traffic//`.

[20] Youtube. `https://www.youtube.com//`.

[21] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *ACM SenSys*, 2015.

[22] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. Deepcache: Principled cache for mobile deep vision. In *ACM MobiCom*, 2018.

[23] Kittipat Apicharttrisorn, Xukan Ran, Jiasi Chen, Srikanth V Krishnamurthy, and Amit K Roy-Chowdhury. Frugal following: Power thrifty object detection and tracking for mobile augmented reality. In *ACM SenSys*, 2019.

[24] Jiaxi Gu, Jiliang Wang, Zhiwen Yu, and Kele Shen. Walls have ears: Traffic-based side-channel attack in video streaming. In *IEEE INFOCOM*, 2018.

[25] Linsong Cheng and Jiliang Wang. Vitrack: Efficient tracking on the edge for commodity video surveillance systems. In *IEEE INFOCOM*, 2018.

[26] Qiang Liu, Siqi Huang, Johnson Opadere, and Tao Han. An edge network orchestrator for mobile augmented reality. In *IEEE INFOCOM*, 2018.

[27] Xianzhong Ding, Wan Du, and Alberto Cerpa. Octopus: Deep reinforcement learning for holistic smart building control. In *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, pages 326–335, 2019.

[28] Zhihao Shen, Kang Yang, Wan Du, Xi Zhao, and Jianhua Zou. Deepapp: a deep reinforcement learning framework for mobile application usage prediction. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, pages 153–165, 2019.

[29] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *IEEE/ACM IPSN*, 2016.

[30] Akhil Mathur, Nicholas D Lane, Sourav Bhattacharya, Aidan Boran, Claudio Forlivesi, and Fahim Kawsar. Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. In *ACM MobiSys*, 2017.

[31] Saman Naderiparizi, Pengyu Zhang, Matthai Philipose, Bodhi Priyantha, Jie Liu, and Deepak Ganesan. Glimpse: A programmable early-discard camera architecture for continuous mobile vision. In *ACM SenSys*, 2017.

[32] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *ACM MobiSys*, 2018.

[33] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *NeurIPS*, 2015.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[35] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, 2017.

[36] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *ACM SIGCOMM*, 2018.

[37] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *ICCV*, 2017.

[38] Péter Baranyi. Nvidia jetpack. `https://developer.nvidia.com/embedded/jetpack`.

[39] Nvidia tensorrt. `https://developer.nvidia.com/tensorrt`.

[40] Péter Baranyi. Nvidia video codec sdk. `https://developer.nvidia.com/nvidia-video-codec-sdk`.

[41] Peizhen Guo and Wenjun Hu. Potluck: Cross-application approximate deduplication for computation-intensive mobile applications. In *ACM ASPLOS*, 2018.

[42] Ling-Yu Duan, Vijay Chandrasekhar, Shiqi Wang, Yihang Lou, Jie Lin, Yan Bai, Tiejun Huang, Alex Chichung Kot, and Wen Gao. Compact descriptors for video analysis: The emerging mpeg standard. *IEEE MultiMedia*, 2018.

[43] Pytorch. `https://pytorch.org/`.

[44] Opencv. `https://https://opencv.org//`.

[45] NHTSA. 2018 Fatal Motor Vehicle Crashes: Overview. `https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812826`.

[46] WHO. Global Status Report on Road Safety 2018. `https://www.who.int/violence_injury_prevention/road_safety_status/2018/en/`.

[47] Chuang-Wen You, Nicholas D Lane, Fanglin Chen, Rui Wang, Zhenyu Chen, Thomas J Bao, Martha Montes-de Oca, Yuting Cheng, Mu Lin, Lorenzo Torresani, et al. Carsafe app: Alerting drowsy and distracted drivers using dual cameras on smartphones. In *ACM MobiSys*, 2013.

[48] Bergasa and Roberto Arroyo. Drivesafe: An app for alerting inattentive drivers and scoring driving behaviors. In *IEEE IV*, 2014.

[49] Xingzhou Zhang, Mu Qiao, Liangkai Liu, Yunfei Xu, and Weisong Shi. Collaborative cloud-edge computation for personalized driving behavior modeling. In *ACM/IEEE SEC*, 2019.

[50] Sinan Kaplan, Mehmet Amac Guvensan, Ali Gokhan Yavuz, and Yasin Karalurt. Driver behavior analysis for safe driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 16(6):3017–3032, 2015.

[51] Cian Ryan, Finbarr Murphy, and Martin Mullins. End-to-end autonomous driving risk analysis: A behavioural anomaly detection approach. *IEEE Transactions on Intelligent Transportation Systems*, 2020.

[52] Zhongyang Chen, Jiadi Yu, Yanmin Zhu, Yingying Chen, and Minglu Li. D 3: Abnormal driving behaviors detection and identification using smartphone sensors. In *2015 12th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pages 524–532. IEEE, 2015.

[53] Lex Fridman, Daniel E Brown, Michael Glazer, William Angell, Spencer Dodd, Benedikt Jenik, Jack Terwilliger, Aleksandr Patsekin, Julia Kindelsberger, Li Ding, et al. Mit advanced vehicle technology study: Large-scale naturalistic driving study of driver behavior and interaction with automation. *IEEE Access*, 7:102021–102038, 2019.

[54] Jin-Hyuk Hong, Ben Margines, and Anind K Dey. A smartphone-based sensing platform to model aggressive driving behaviors. In *ACM SIGCHI*, 2014.

[55] Waldo R Tobler. A computer movie simulating urban growth in the detroit region. *Economic geography*, 46:234–240, 1970.

[56] Isam Mashhour Al Jawarneh, Paolo Bellavista, Antonio Corradi, Luca Foschini, and Rebecca Montanari. Locality-preserving spatial partitioning for geo big data analytics in main memory frameworks. In *GLOBECOM 2020-2020 IEEE Global Communications Conference*, pages 1–6. IEEE, 2020.

[57] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NeurIPS-W*, 2017.

[58] Jing Yuan, Yu Zheng, Chengyang Zhang, Wenlei Xie, Xing Xie, Guangzhong Sun, and Yan Huang. T-drive: driving directions based on taxi trajectories. In *ACM SIGSPATIAL*, 2010.

[59] Jing Yuan, Yu Zheng, Xing Xie, and Guangzhong Sun. Driving with knowledge from the physical world. In *ACM SIGKDD*, 2011.

[60] Chong Zhou and Randy C Paffenroth. Anomaly detection with robust deep autoencoders. In *ACM SIGKDD*, 2017.

[61] Bo Zong, Qi Song, Martin Renqiang Min, Wei Cheng, Cristian Lumezanu, Daeki Cho, and Haifeng Chen. Deep autoencoding gaussian mixture model for unsupervised anomaly detection. In *ICLR*, 2018.

[62] Dong Gong, Lingqiao Liu, Vuong Le, Budhaditya Saha, Moussa Reda Mansour, Svetha Venkatesh, and Anton van den Hengel. Memorizing normality to detect anomaly: Memory-augmented deep autoencoder for unsupervised anomaly detection. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1705–1714, 2019.

[63] Mahmudul Hasan, Jonghyun Choi, Jan Neumann, Amit K Roy-Chowdhury, and Larry S Davis. Learning temporal regularity in video sequences. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 733–742, 2016.

[64] Apichon Witayangkurn, Teerayut Horanont, Yoshihide Sekimoto, and Ryosuke Shibasaki. Anomalous event detection on large-scale gps data from mobile phones using hidden markov model and cloud platform. In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, pages 1219–1228, 2013.

[65] Siqian Yang, Cheng Wang, Hongzi Zhu, and Changjun Jiang. App: augmented proactive perception for driving hazards with sparse gps trace. In *Proceedings of the twentieth ACM international symposium on mobile Ad Hoc networking and computing*, pages 21–30, 2019.

[66] Jiadi Yu, Hongzi Zhu, Haofu Han, Yingying Jennifer Chen, Jie Yang, Yanmin Zhu, Zhongyang Chen, Guangtao Xue, and Minglu Li. Senspeed: Sensing driving conditions to estimate vehicle speed in urban environments. *IEEE Transactions on Mobile Computing*, 15(1):202–216, 2015.

[67] Mingming Zhang, Chao Chen, Tianyu Wo, Tao Xie, Md Zakirul Alam Bhuiyan, and Xuelian Lin. Safedrive: online driving anomaly detection from large-scale vehicle data. *IEEE Transactions on Industrial Informatics*, 13(4):2087–2096, 2017.

[68] Vidyasagar Sadhu, Teruhisa Misu, and Dario Pompili. Deep multi-task learning for anomalous driving detection using can bus scalar sensor data. *arXiv preprint arXiv:1907.00749*, 2019.

[69] Raghavendra Chalapathy and Sanjay Chawla. Deep learning for anomaly detection: A survey. *arXiv preprint arXiv:1901.03407*, 2019.

[70] Sarah M Erfani, Mahsa Baktashmotlagh, Masud Moshtaghi, Vinh Nguyen, Christopher Leckie, James Bailey, and Kotagiri Ramamohanarao. From shared subspaces to shared landmarks: A robust multi-source classification approach. In *AAAI*, 2017.

[71] Samet Akcay, Amir Atapour-Abarghouei, and Toby P Breckon. Ganomaly: Semi-supervised anomaly detection via adversarial training. In *ACCV*, 2018.

[72] Abd-Elhamid M Taha and Nidal Nasser. Utilizing can-bus and smartphones to enforce safe and responsible driving. In *IEEE ISCC*, 2015.

[73] Tanushree Banerjee, Arijit Chowdhury, and Tapas Chakravarty. Mydrive: Drive behavior analytics method and platform. In *ACM WPA*, 2016.

[74] Xiaoyu Zhu, Yifei Yuan, Xianbiao Hu, Yi-Chang Chiu, and Yu-Luen Ma. A bayesian network model for contextual versus non-contextual driving behavior assessment. *Transportation research part C: emerging technologies*, 81:172–187, 2017.

[75] Pengyang Wang, Yanjie Fu, Jiawei Zhang, Pengfei Wang, Yu Zheng, and Charu Aggarwal. You are how you drive: Peer and temporal-aware representation learning for driving behavior analysis. In *ACM SIGKDD*, 2018.

[76] Yan Xu and Gary Tan. An offline road network partitioning solution in distributed transportation simulation. In *IEEE/ACM DS-RT*, 2012.

[77] Ying-Ying Ma, Yi-Chang Chiu, and Xiao-Guang Yang. Urban traffic signal control network automatic partitioning using laplacian eigenvectors. In *IEEE ITSC*, 2009.

[78] Yuxuan Ji and Nikolas Geroliminis. On the spatial partitioning of urban transportation networks. *Transportation Research Part B: Methodological*, 46(10):1639–1656, 2012.

[79] Jing Yuan, Yu Zheng, and Xing Xie. Discovering regions of different functions in a city using human mobility and pois. In *ACM SIGKDD*, 2012.

[80] Nicholas Jing Yuan, Yu Zheng, and Xing Xie. Segmentation of urban areas using road networks. *MSR-TR-2012–65, Tech. Rep.*, 2012.

[81] Wei Liu, Yu Zheng, Sanjay Chawla, Jing Yuan, and Xie Xing. Discovering spatio-temporal causal interactions in traffic data streams. In *ACM SIGKDD*, 2011.

[82] Yanjie Fu, Pengyang Wang, Jiadi Du, Le Wu, and Xiaolin Li. Efficient region embedding with multi-view spatial networks: A perspective of locality-constrained spatial autocorrelations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 906–913, 2019.

[83] Yunchao Zhang, Yanjie Fu, Pengyang Wang, Xiaolin Li, and Yu Zheng. Unifying inter-region autocorrelation and intra-region structures for spatial embedding via collective adversarial learning. In *ACM SIGKDD*, 2019.

[84] Hector Gonzalez, Jiawei Han, Xiaolei Li, Margaret Myslinska, and John Paul Sondag. Adaptive fastest path computation on a road network: a traffic mining approach. In *VLDB*, 2007.

[85] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[86] Mohammad Sabokrou, Mohammad Khalooei, Mahmood Fathy, and Ehsan Adeli. Adversarially learned one-class classifier for novelty detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3379–3388, 2018.

[87] Muhammad Zaigham Zaheer, Jin-ha Lee, Marcella Astrid, and Seung-Ik Lee. Old is gold: Redefining the adversarially learned one-class classifier training paradigm. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14183–14193, 2020.

[88] Hongyong Wang, Xinjian Zhang, Su Yang, and Weishan Zhang. Video anomaly detection by the duality of normality-granted optical flow. *arXiv preprint arXiv:2105.04302*, 2021.

[89] Mujtaba Asad, Jie Yang, Enmei Tu, Liming Chen, and Xiangjian He. Anomaly3d: Video anomaly detection based on 3d-normality clusters. *Journal of Visual Communication and Image Representation*, 75:103047, 2021.

[90] Andreas Emil Feldmann and Luca Foschini. Balanced partitions of trees and applications. *Algorithmica*, 71(2):354–376, 2015.

[91] Michael R Garey and David S Johnson. *Computers and intractability: A guide to the theory of NP-completeness*, volume 174. freeman San Francisco, 1979.

[92] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

[93] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

[94] Openstreetmap. `https://www.openstreetmap.org`.

[95] Xianzhong Ding, Wan Du, and Alberto E Cerpa. Mb2c: Model-based deep reinforcement learning for multi-zone building control. In *Proceedings of the 7th ACM international conference on systems for energy-efficient buildings, cities, and transportation*, pages 50–59, 2020.

[96] Xianzhong Ding and Wan Du. Drlic: Deep reinforcement learning for irrigation control. In *ACM/IEEE IPSN*, 2022.

[97] Miaomiao Liu, Xianzhong Ding, and Wan Du. Continuous, real-time object detection on mobile devices without offloading. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 976–986. IEEE, 2020.

[98] Paul Newson and John Krumm. Hidden markov map matching through noise and sparseness. In *ACM SIGSPATIAL*, 2009.

[99] Zhihao Shen, Wan Du, Xi Zhao, and Jianhua Zou. Dmm: fast map matching for cellular data. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[100] Geosphere. `http://www.edwilliams.org/avform.htm#Dist`.

[101] Jiangpeng Dai, Jin Teng, Xiaole Bai, Zhaohui Shen, and Dong Xuan. Mobile phone based drunk driving detection. In *2010 4th International Conference on Pervasive Computing Technologies for Healthcare*, pages 1–8. IEEE, 2010.

[102] Yash Jain, Chi Ian Tang, Chulhong Min, Fahim Kawsar, and Akhil Mathur. Collossl: Collaborative self-supervised learning for human activity recognition. *ACM IMWUT*, 6(1):1–28, 2022.

[103] Linlin Tu, Xiaomin Ouyang, Jiayu Zhou, Yuze He, and Guoliang Xing. Feddl: Federated learning via dynamic layer sharing for human activity recognition. In *ACM SenSys*, 2021.

[104] Yang Liu, Zhenjiang Li, Zhidan Liu, and Kaishun Wu. Real-time arm skeleton tracking and gesture inference tolerant to missing wearable sensors. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 287–299, 2019.

[105] Wenguang Mao, Mei Wang, Wei Sun, Lili Qiu, Swadhin Pradhan, and Yi-Chao Chen. Rnn-based room scale hand motion tracking. In *ACM MobiCom*, 2019.

[106] Sheng Shen, He Wang, and Romit Roy Choudhury. I am a smartwatch and i can track my user's arm. In *Proceedings of the 14th annual international conference on Mobile systems, applications, and services*, pages 85–96, 2016.

[107] Peijun Zhao, Chris Xiaoxuan Lu, Bing Wang, Niki Trigoni, and Andrew Markham. Cubelearn: End-to-end learning for human motion recognition from raw mmwave radar signals. *arXiv preprint arXiv:2111.03976*, 2021.

[108] Chengkun Jiang, Yuan He, Songzhen Yang, Junchen Guo, and Yunhao Liu. 3d-omnitrack: 3d tracking with cots rfid systems. In *ACM/IEEE IPSN*, 2019.

[109] Hao Kong, Xiangyu Xu, Jiadi Yu, Qilin Chen, Chenguang Ma, Yingying Chen, Yi-Chao Chen, and Linghe Kong. m3track: mmwave-based multi-user 3d posture tracking. In *MobiSys*, 2022.

[110] Tianxing Li, Chuankai An, Zhao Tian, Andrew T Campbell, and Xia Zhou. Human sensing using visible light communication. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 331–344, 2015.

[111] Tianxing Li, Qiang Liu, and Xia Zhou. Practical human sensing in the light. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 71–84, 2016.

[112] Dongyao Chen, Mingke Wang, Chenxi He, Qing Luo, Yasha Iravantchi, Alanson Sample, Kang G Shin, and Xinbing Wang. Wearable, untethered hands tracking with passive magnets. In *ACM MobiCom*, 2021.

[113] Sheng Shen, Mahanth Gowda, and Romit Roy Choudhury. Closing the gaps in inertial motion tracking. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 429–444, 2018.

[114] Qiang Yang and Yuanqing Zheng. Model-based head orientation estimation for smart devices. *ACM IMWUT*, 5(3):1–24, 2021.

[115] Changhao Chen, Xiaoxuan Lu, Andrew Markham, and Niki Trigoni. Ionet: Learning to cure the curse of drift in inertial odometry. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[116] Mahdi Abolfazli Esfahani, Han Wang, Keyu Wu, and Shenghai Yuan. Orinet: Robust 3-d orientation estimation with a single particular imu. *IEEE Robotics and Automation Letters*, 5(2):399–406, 2019.

[117] Martin Brossard, Silvere Bonnabel, and Axel Barrau. Denoising imu gyroscopes with deep learning for open-loop attitude estimation. *IEEE Robotics and Automation Letters*, 5(3):4796–4803, 2020.

[118] Scott Sun, Dennis Melamed, and Kris Kitani. Idol: Inertial deep orientation-estimation and localization. *arXiv preprint arXiv:2102.04024*, 2021.

[119] Pengfei Zhou, Mo Li, and Guobin Shen. Use it free: Instantly knowing your phone attitude. In *ACM MOBICOM*, 2014.

[120] Hussein Hazimeh, Zhe Zhao, Aakanksha Chowdhery, Maheswaran Sathiamoorthy, Yihua Chen, Rahul Mazumder, Lichan Hong, and Ed Chi. Dselect-k: Differentiable selection in the mixture of experts with applications to multi-task learning. *Advances in Neural Information Processing Systems*, 34:29335–29347, 2021.

[121] Yuxin Tian, Xueqing Deng, Yi Zhu, and Shawn Newsam. Cross-time and orientation-invariant overhead image geolocalization using deep local features. In *IEEE/CVF Winter Conference on Applications of Computer Vision*, 2020.

[122] Yunzhong He, Yuxin Tian, Mengjiao Wang, Feier Chen, Licheng Yu, Maolong Tang, Congcong Chen, Ning Zhang, Bin Kuang, and Arul Prakash. Que2engage: Embedding-based retrieval for relevant and engaging products at facebook marketplace. *arXiv preprint arXiv:2302.11052*, 2023.

[123] Kang Yang and Wan Du. LLDPC: A Low-Density Parity-Check Coding Scheme for LoRa Networks. In *ACM SenSys*, 2022.

[124] Miaomiao Liu, Kang Yang, Yanjie Fu, Dapeng Oliver Wu, and Wan Du. Driving maneuver anomaly detection based on deep auto-encoder and geographical partitioning. *ACM Transactions on Sensor Networks (TOSN)*, 2022.

[125] Hang Yan, Qi Shan, and Yasutaka Furukawa. Ridi: Robust imu double integration. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 621–636, 2018.

[126] Sachini Herath, Hang Yan, and Yasutaka Furukawa. Ronin: Robust neural inertial navigation in the wild: Benchmark, evaluations, & new methods. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3146–3152. IEEE, 2020.

[127] Huatao Xu, Pengfei Zhou, Rui Tan, Mo Li, and Guobin Shen. Limu-bert: Unleashing the potential of unlabeled data for imu sensing applications. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, pages 220–233, 2021.

[128] Andrea Giovanni Cutti, Andrea Giovanardi, Laura Rocchi, Angelo Davalli, and Rinaldo Sacchetti. Ambulatory measurement of shoulder and elbow kinematics through inertial and magnetic sensors. *Medical & biological engineering & computing*, 46(2):169–178, 2008.

[129] Mahmoud El-Gohary and James McNames. Shoulder and elbow joint angle tracking with inertial sensors. *IEEE Transactions on Biomedical Engineering*, 59(9):2635–2641, 2012.

[130] Qaiser Riaz, Guanhong Tao, Björn Krüger, and Andreas Weber. Motion reconstruction using very few accelerometers and ground contacts. *Graphical Models*, 79:23–38, 2015.

[131] Jochen Tautges, Arno Zinke, Björn Krüger, Jan Baumann, Andreas Weber, Thomas Helten, Meinard Müller, Hans-Peter Seidel, and Bernd Eberhardt. Motion reconstruction using sparse accelerometer data. *ACM Transactions on Graphics (ToG)*, 30(3):1–12, 2011.

[132] Pengfei Zhou, Yuanqing Zheng, and Mo Li. How long to wait? predicting bus arrival time with mobile phone based participatory sensing. In *ACM MobiSys*, 2012.

[133] Wan Du, Panrong Tong, and Mo Li. Uniloc: A unified mobile localization framework exploiting scheme diversity. *IEEE Transactions on Mobile Computing*, 20(7):2505–2517, 2020.

[134] Zhengyou Zhang. Microsoft kinect sensor and its effect. *IEEE multimedia*, 19(2):4–10, 2012.

[135] Mingmin Zhao, Yonglong Tian, Hang Zhao, Mohammad Abu Alsheikh, Tianhong Li, Rumen Hristov, Zachary Kabelac, Dina Katabi, and Antonio Torralba. Rf-based 3d skeletons. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 267–281, 2018.

[136] Wenjun Jiang, Hongfei Xue, Chenglin Miao, Shiyang Wang, Sen Lin, Chong Tian, Srinivasan Murali, Haochen Hu, Zhi Sun, and Lu Su. Towards 3d human pose construction using wifi. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[137] Yuxin Tian, Shawn Newsam, and Kofi Boakye. Fashion image retrieval with text feedback by additive attention compositional learning. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 1011–1021, 2023.

[138] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057. PMLR, 2015.

[139] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[140] Huimin Ren, Sijie Ruan, Yanhua Li, Jie Bao, Chuishi Meng, Ruiyuan Li, and Yu Zheng. Mtrajrec: Map-constrained trajectory recovery via seq2seq multi-task learning. In *ACM SIGKDD*, 2021.

[141] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649, 2013.

[142] Azure kinect, 2022. https://azure.microsoft.com/en-us/services/kinect-dk/.

[143] Vicon motion system, 2022. https://www.vicon.com/.

[144] Orientation tracking of azure kinect, 2019. https://github.com/microsoft/Azure-Kinect-Sensor-SDK/issues/654.

[145] Oculus insight, 2019. https://ai.facebook.com/blog/powered-by-ai-oculus-insight/.

[146] Ana Rojo, Javier Cortina, Cristina Sánchez, Eloy Urendes, Rodrigo García-Carmona, and Rafael Raya. Accuracy study of the oculus touch v2 versus inertial sensor for a single-axis rotation simulating the elbow's range of motion. *Virtual Reality*, pages 1–12, 2022.

[147] Valentin Holzwarth, Joy Gisler, Christian Hirt, and Andreas Kunz. Comparing the accuracy and precision of steamvr tracking 2.0 and oculus quest 2 in a room scale setup. In *2021 the 5th International Conference on Virtual and Augmented Reality Simulations*, pages 42–46, 2021.

[148] Tyler A Jost, Bradley Nelson, and Jonathan Rylander. Quantitative analysis of the oculus rift s in controlled movement. *Disability and Rehabilitation: Assistive Technology*, 16(6):632–636, 2021.

[149] Xiaomin Ouyang, Xian Shuai, Jiayu Zhou, Ivy Wang Shi, Zhiyuan Xie, Guoliang Xing, and Jianwei Huang. Cosmo: contrastive fusion learning with small data for multimodal human activity recognition. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pages 324–337, 2022.

[150] Ruiyuan Song, Dongheng Zhang, Zhi Wu, Cong Yu, Chunyang Xie, Shuai Yang, Yang Hu, and Yan Chen. Rf-url: unsupervised representation learning for rf sensing. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pages 282–295, 2022.