UCLA
UCLA Electronic Theses and Dissertations

Title
Learning-enabled Cyber-Physical Systems: Challenges and Strategies

Permalink
https://escholarship.org/uc/item/4mb7w0pn

Author
Sandha, Sandeep Singh

Publication Date
2022

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA

Los Angeles

Learning-enabled Cyber-Physical Systems: Challenges and Strategies

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

by

Sandeep Singh Sandha

2022

© Copyright by

Sandeep Singh Sandha

2022

ABSTRACT OF THE DISSERTATION

Learning-enabled Cyber-Physical Systems: Challenges and Strategies

by

Sandeep Singh Sandha
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2022
Professor Mani B. Srivastava, Chair

Cyber-physical systems (CPS) are increasingly adopting learning-enabled components having deep neural networks in their decision-making pipelines. Deep neural networks show the promise to simplify the CPS pipelines for high-dimensional sensors as they require little pre-processing of data and are shown to be more accurate than their traditional counterparts. However, integrating neural networks into the sense-infer-actuate pipeline of CPS faces several challenges. In this dissertation, we study the following challenges in the context of learning-enabled CPS and propose new algorithms and system design strategies to address them.

First, we study the challenge of characterizing uncertainty in sensor data timestamps and its impact on multimodal fusion applications. Motivated by smartphones' integration in several CPS applications, we quantify the data timestamp uncertainty across modern smartphone devices. To our surprise, we find drastic timestamping errors ranging up to multiple seconds in Android devices. Then, we explore if these timing errors are significant enough to impact the neural network's performance. Our evaluation shows that the observed timing errors can cripple the deep neural networks doing multimodal fusion due to data misalignments. Our finding signifies the need to rethink the shared notion of time on smartphones. To mitigate timestamp errors, we introduce approaches

to improve time across smartphones having up to 200 microseconds of timing accuracy. We also propose a novel time-shift data augmentation technique to train time-resilient neural networks robust to the inevitability of timing errors and, as such, degrade gracefully in the face of timing errors.

As a second challenge, we explore the impact of variable delays on the emerging deep reinforcement learning (RL) controllers, which are preferred due to their capability to handle high-dimensional data. Conventional controllers can model and account for delay variations in their design. However, handling variable delays in deep-RL is challenging as a black-box neural network represents the controller policy. Researchers currently use domain randomization and worst-case delay modeling to train deep-RL policies on a spread of expected delay variations. We demonstrate a significant performance degradation in applications even when using the state-of-the-art domain randomization approach. To address this, we propose Time-in-State RL, a delay-aware deep RL approach that augments the agent's state with temporal properties (sampling interval and execution latency). Time-in-State RL trains policies that show superior performance by adapting to the variable timing characteristics at runtime. We further show the superior performance of Time-in-State to the worst-case delay controllers when worst-case delays are significant. We demonstrate the efficacy of Time-in-State RL on HalfCheetah, Ant, and car in simulation and on a real scaled car robot.

Thirdly, we study the challenge of modeling the CPS environment to train end-to-end controllers using deep-RL for closed-loop systems. We specifically consider the example of autonomous pan-tilt-zoom (PTZ) controllers. Existing autonomous PTZ controllers have multiple stages: detecting objects of interest, short-term tracking, and control of pan, tilt and zoom parameters to keep objects in the field of view. The multiple stages suffer from performance bottlenecks as it is difficult to optimize each step. Further, these multiple stages are computationally intensive to be realized in real-time on embedded camera platforms. Despite these shortcomings, developers adopt existing multi-stage solutions due to the lack of simulators needed to develop end-to-end controller policies. We propose *Eagle*, an end-to-end deep-RL approach using raw images to control

a PTZ camera. To enable successful training of Eagle, we also introduce *EagleSim*, a simulation framework to study PTZ cameras in photo-realistic virtual worlds. Our evaluation across a suite of PTZ tracking scenarios shows that Eagle outperforms current multi-stage approaches by providing superior tracking performance. Further, we also show that Eagle policies are transferable to real-scene videos and are lightweight to enable real-time deployment on Raspberry PI and Jetson Nano class devices.

Finally, we study the challenge of developing machine learning classifiers having optimal accuracy within the desired resource budget of CPS applications. Selecting an optimal classifier is becoming increasingly complex, with many choices for classifiers and their rich hyperparameter parameter spaces. Although several hyperparameter tuning frameworks exist, their practical adoption is hindered due to inferior search algorithms, inflexible architecture, software dependencies, or closed source nature. As a solution, we propose designing a lightweight library with a flexible architecture and state-of-the-art parallel optimization algorithms. We present *Mango*, a parallel hyperparameter tuning library, to realize the proposed design. Mango is currently used in production at Arm for more than 30 months and is available open-source. We evaluate Mango on several benchmarks to highlight its superior performance. We discuss production use cases of Mango in an AutoML framework and commercial CPU design pipeline. We also showcase another advantage of Mango in enabling hardware-aware neural architecture search to transfer deep neural networks to TinyML platforms (microcontroller class devices) used by CPS/IoT applications.

The dissertation of Sandeep Singh Sandha is approved.

Anthony John Nowatzki

Omid Abari

Puneet Gupta

Mani B. Srivastava, Committee Chair

University of California, Los Angeles

2022

To my parents, Ramandeep Kaur and Sulakhan Singh, who were with me all the way,

and

my brother, Jashan Meet, who supported me on this journey.

TABLE OF CONTENTS

1 Introduction **1**

1.1 Challenge 1: Data Misalignment due to Timestamp Uncertainty 2

1.1.1 Improving Time and Training Resilient Classifiers 3

1.2 Challenge 2: Delay Awareness in Deep Reinforcement Learning 3

1.2.1 Handling Variable Delays 4

1.3 Challenge 3: Realizing Training Environments to enable End-to-end Control 5

1.3.1 Deep Reinforcement Learning-based Controllers for PTZ Cameras 6

1.4 Challenge 4: Training Machine Learning Models with Optimal Hyperparameters in Production 6

1.4.1 Enabling Hyperparameter Tuning in Production 7

1.5 Dissertation outline 7

2 Data Timestamp Uncertainty and its Impact on Multimodal Fusion **9**

2.1 Distributed Sensing using Smartphones 9

2.1.1 Data Timestamping in Smartphones 9

2.1.2 Deep Learning-Based Multimodal Fusion 10

2.2 Background and Related Work 11

2.3 Smartphone System Clock Study 13

2.3.1 Understanding Android System Time 17

2.3.2 Forcing a Sync Event 17

2.3.3 NITZ vs NTP 18

2.4 Impact of Timing Errors on Deep Learning-Based Multimodal Fusion 19

2.4.1 Multimodal Deep Learning for Human Activity Recognition 19
2.5 Strategies to Mitigate Timing Errors 24
2.5.1 System Clock Replacement 24
2.5.2 Time-Shift Data Augmentation 25
2.6 Discussion 27
3 Synchronizing Time across Smartphones 28
3.1 Shared Notion of Time across Smartphones 28
3.1.1 Challenges and Tradeoffs 28
3.2 Background and Related Work 30
3.3 Smartphone Time Synchronization 31
3.3.1 Time Synchronization Approaches 31
3.3.2 Time Synchronization Comparison 33
3.4 Evaluation 35
3.4.1 Experimental Setup 36
3.4.2 Variability Evaluation 37
3.4.3 Cross-Peripheral Evaluation 40
3.5 Discussion 41
3.5.1 Tradeoffs 41
3.5.2 Recommended Sync Solution 42
4 Variable End-to-end Delays in Deep Reinforcement Learning 43
4.1 State Transition Delay in Deep-RL 43
4.1.1 Time-in-State RL 44

4.2 Background 45
4.2.1 Temporal Variability in Deep-RL 45
4.2.2 Variability in Execution Latency and Sampling Interval 47
4.2.3 Impact of Temporal Variability on Deep-RL Policy 48
4.3 Related Work 49
4.3.1 Control System Approaches 49
4.3.2 Handling Delays in Reinforcement Learning 51
4.3.3 Accuracy of Delay Measurements at Runtime 53
4.4 Training Deep-RL Policies with Temporal Variations 55
4.4.1 Low Dimensional Use Cases: HalfCheetah and Ant 56
4.4.2 High-Dimensional Use Case: Autonomous Vehicle 59
4.5 Evaluation of Time-in-State RL 60
4.5.1 HalfCheetah and Ant Tasks 60
4.5.2 Experiments with Variable Delays within an Episode and Timing Noises . . 61
4.5.3 DeepRacer Robotic Car 62
4.6 Experiments with Recurrent Policies 68
4.7 Vanilla Deep Reinforcement Learning Policy without Varying Timing Characteristics 69
4.8 Comparison of Time-in-state with Worst Case Delay Controller 70
4.9 Conclusion 72

5 End-to-end Deep Reinforcement Learning for Autonomous Control of PTZ Cameras 73

5.1 Introduction 74
5.2 Background and Related Works 79
5.2.1 Autonomous Control of PTZ Cameras 79

5.2.2 Frameworks for Pan-Tilt-Zoom Cameras 81
5.3 Eagle: End-to-end Deep-RL for PTZ 82
5.3.1 State Space, Policy Network and Actions 82
5.3.2 Reward Function: Single Object 83
5.3.3 Generalizable PTZ Tracking 84
5.3.4 Dynamic Tasking of Eagle Policies 86
5.4 Design of EagleSim 88
5.4.1 Photo-Realistic Virtual Worlds 88
5.4.2 PTZ Abstractions 89
5.5 Evaluation 90
5.5.1 Performance Metrics for PTZ Tracking 91
5.5.2 Implementation of Eagle 94
5.5.3 Tracking Scenarios 95
5.5.4 Eagle vs Other Approaches 104
5.5.5 PTZ Tracking using Lightweight Object Detectors 107
5.5.6 Transfer of Eagle to the Real Scene Videos 110
5.5.7 Runtime of Eagle on Embedded Cameras 111
5.6 Discussion 114
5.7 Conclusion 115

6 Enabling Hyperparameter Tuning of Machine Learning Classifiers in Production . 116
6.1 Introduction 117
6.2 Background and Related Work 120
6.2.1 Hyperparameter Tuning Frameworks 120

6.2.2 Hyperparameter Tuning Algorithms . . . 121
6.2.3 Algorithms Implemented in Mango . . . 122
6.3 Mango . . . 123
6.3.1 Mango Abstractions . . . 123
6.3.2 Optimization Algorithms in Mango . . . 127
6.4 Evaluation and Case Studies . . . 131
6.4.1 Optimization Performance Evaluation . . . 131
6.4.2 Case Study: Bug Hunting in Design Verification of Integrated Circuits . . . 138
6.4.3 Case Study: AutoML Framework . . . 140
6.4.4 Case Study: Network Architecture Search for TinyML Platforms . . . 141
6.5 Discussion . . . 144

7 Discussion and Future Work . . . 145
7.1 Extending Timing Analysis . . . 145
7.2 A Vision of Timing Stack for Deep Reinforcement Learning . . . 146
7.3 Future Training Environments for CPS Applications . . . 146
7.4 Limitations of End-to-end Control . . . 147
7.5 Possible Extension in Mango . . . 147

8 Conclusion . . . 149

9 Appendix . . . 151
9.1 Delay Measurements on Different Hardware Platforms . . . 151
9.2 Additional Details on HalfCheetah and Ant Tasks . . . 154
9.3 Additional Details on Autonomous Vehicle Task . . . 154

9.4 Learning Curves of Worst-Case Delay Policy 156

References **157**

LIST OF FIGURES

1.1 Different components in a typical Cyber-physical systems application. 2

2.1 Observed system clock errors across all devices when compared to an NTP baseline during the undisturbed 5-day study. Three significant time adjustments (>100ms) occur: A6 at minute 1396, A8 at minute 6166, and A2 at minute 6457. 13

2.2 Observed iOS system clock errors during a 6-hour snapshot of the 5-day study. iOS errors are an order-of-magnitude less compared to Android due to the aggressive clock correction that occurs at semi-periodic intervals. 15

2.3 System clock error of an Android Nexus 5X device after triggering 100 independent NITZ timing updates. 18

2.4 System clock error of an Android Nexus 5X device after triggering 100 independent NTP timing updates. 19

2.5 The architecture of Multimodal Audio-IMU Network with separate branches fused in cross sensing layers. *Conv* and *fc* refer to convolutional modules and fully connected layers, respectively. 20

2.6 Variation in the accuracy of the Multimodal Audio-IMU network with increased timing errors between smartphones collecting audio and IMU data. *10-Sec, 20-Sec*, and *60-Sec* periods represent varying duration between activity changes. 1000ms of timing error results in an accuracy drop of 6% for *10-Sec Period*. 23

2.7 Comparison of Multimodal Audio-IMU Network test accuracy with different augmented training datasets. The augmented models can preserve classifier accuracy despite 250ms and 600ms of timing error. Without augmentation, 600ms of error results in model accuracy drop by 3%, from 96.5% to 93.5%. 26

3.1 Timestamping events for audio, Wi-Fi and BLE peripherals in Android. Timestamp delays are not drawn to scale. 34

3.2 Relative clock offset for audio-based sync over time, with respect to a fourth phone (Pixel) serving as a reference clock. Results are normalized to the initial computed offset for each device. Due to clock drift, offsets change as a function of the relative drift between the synchronizing device and the reference device. A regression line for each device indicates the overall relative drift trend. 38

3.3 Sync offset variability with respect to the fourth (Pixel) reference device for (a) audio, (b) BLE, and (c) Wi-Fi implementations. 86% of audio sync attempts fall within $\pm 200\mu s$. 85% of BLE sync attempts fall within $\pm 3000\mu s$. 95% of Wi-Fi sync attempts fall within $\pm 1000\mu s$. 39

4.1 Delays for a typical sensing to actuation pipeline. TSRL augments the observed state with sampling interval and inferencing latency. 47

4.2 The DeepRacer car on a real track and the simulated car in the Gazebo environment. The OptiTrack motion capture system is used to quantify the performance of policies on the real track. 56

4.3 The learning curves for HalfCheetah, Ant and DeepRacer car for TS and DR policies. . 57

4.4 Comparison of time-in-state (TS) and domain randomization (DR) policies for HalfCheetah and Ant tasks across different execution latencies ($\Delta\tau_\eta$). The sampling intervals ($\Delta\tau_\sigma$) is selected to be maximum of (4.12 ms , $\Delta\tau_\eta$), so that agent can act for each sensed state. The mean is shown in green, the black 'x' marker shows the median of IQR. For both tasks, TS policies achieve higher mean reward than DR policies. 58

4.5 Comparison of time-in-state (TS) and domain randomization (DR) policies for HalfCheetah and Ant tasks across different multitenancy settings. The mean is shown in green. The back 'x' marker shows the median of IQR. 62

4.6 A comparison of a single instance of two deep reinforcement learning-based controllers on a $1/18^{th}$ scale real autonomous car in the presence of 60 ms execution time. The proposed Time-in-State (TS) based controller performs better than the domain randomization (DR) based controller. 63

4.7 Evaluation of time-in-state (TS) and domain randomization (DR) policies using DeepRacer car. (a) The distance of the real car from the centerline captured using OptiTrack cameras. The number of points plotted is 2400, except the DR ($\Delta\tau_\eta$=60ms), which has 1657 points. The onboard camera of car was running at 30Hz. (b) Analysis of TS and DR policies across different execution latencies ($\Delta\tau_\eta$) in DeepRacer simulator. The sampling intervals ($\Delta\tau_\sigma$) is selected to be maximum of (33 ms , $\Delta\tau_\eta$). The green color shows mean, the black 'x' marker shows the median of IQR. 64

4.8 Learning curves of time-in-state recurrent (TS-Recurrent), time-in-state fully connected (TS-FF), domain randomization recurrent (DR-Recurrent), and domain randomization fully connected (DR-FF) policies for HalfCheetah task. The fully connected policies are trained for $\sim$ 2400 iterations whereas the recurrent policies are trained for $\sim$ 10000 iterations. TS policies achieve higher training reward than the DR policies. 66

4.9 Sim2sim comparison of time-in-state recurrent (TS-Recurrent), time-in-state fully connected (TS-FF), domain randomization recurrent (DR-Recurrent), and domain randomization fully connected (DR-FF) policies for HalfCheetah task. The comparison is done across different execution latencies ($\Delta\tau_\eta$). The sampling intervals ($\Delta\tau_\sigma$) is selected to be maximum of (4.12 ms , $\Delta\tau_\eta$), so that agent can act for each sensed state. The mean is shown in green, the black 'x' marker shows the median of IQR. TS policies achieve higher mean reward than DR policies. 67

4.10 (a) The learning curves for vanilla policy training for the HalfCheetah task. (b) The evaluation of vanilla policy across different execution latencies ($\Delta\tau_\eta$). The sampling intervals ($\Delta\tau_\sigma$) is selected to be maximum of (4.12ms , $\Delta\tau_\eta$). The mean is shown in green. The back 'x' marker shows the median of IQR. 69

4.11 The performance comparison of Time-in-state (TS), Time-in-state with noisy measurements (TS-Noise), Domain Randomization, Worst-case delay of 3X (Worst-Case3x), and Worst-case delay of 5X (Worst-Case5x) for HalfCheetah task. The Worst-Case3x (3x4.12 ms) and Worst-Case5x (5x4.12 ms) have stable performance up to 3X (12.3 ms) and 5X (20.6 ms) latencies. However, worst-case policies show immediate performance loss beyond their worst-case limits. When the delay variations are large or worst-case delays are significant; TS policies are a clear winner. 71

5.1 Eagle trains end-to-end deep-RL controllers for PTZ cameras. Sample scenes for vehicle and human tracking from the EagleSim simulator are shown. The direct transfer of Eagle policies to real scene videos is also demonstrated. 76

5.2 Different approaches for autonomous control of PTZ cameras illustrated using a vehicle tracking scenario. A PTZ camera is controlled to keep a car in the field-of-view (FoV). The horizontal FoV (FoV_h) and vertical FoV (FoV_v) control zoom parameter. Approach-1 (*Object-detection+tracking+control*): Represented by 1,2,3,4,9 is the widely used multi-stage technique of identifying objects (using object detectors), followed by a short term tracker and a controller. Approach-2 *Object-detection+RL*: Given by 1,2,5,9 shows a setting where the bounding boxes are used to train a RL policy. Approach-3 *Relative-location+control*: Steps 1,6,7,9 shows an alternative to bounding boxes where a neural network predicts the relative location of objects that the controller uses. Approach-4 *Eagle: End-to-end deep-RL*: Steps 1,8,9 show the proposed Eagle approach to directly control the pan, tilt, and zoom parameters using the raw input images. 78

5.3 A sample bounding box for the object of interest (car). The center of the image is the origin (0,0). (x,y) is the center of the bounding box ($Xmin, Ymin, Xmax, Ymax$). 85

5.4 The architecture of EagleSim and its integration with Eagle. Step-1 shows placement of a PTZ camera for vehicle tracking. Step-2 shows an image captured by camera. Step-3 shows a bounding box for the object of interest (car). 87

5.5 Visualization of scenarios shown in Table 5.2. Sc-1 has a fixed background. In Sc-2, fixed background is extended with random tree placements and image augmentations. Sc-3 shows background variations with image augmentations and trees. Sc-4 extends the Sc-3 scenes with humans and same vehicle with different colors. In, Sc-5, we add vehicles of different types as well. DT shows the scenes for dynamic tasking of policy to track human characters. 93

5.6 Average training reward of Eagle policies for scenarios shown in Table 5.2. We calculate average reward by training three policies for each scenario and show its min-max spread. Each policy is trained for 69 hours (2000 iterations). 97

5.7 Learning curves of 6 different networks trained to predict bounding boxes of a car from images. The validation loss is shown for 1000 epochs. *240_150k* refers to the model trained using 240×240 image input on 150k image dataset. The mean and min-max spread of checkpoints for each network are shown. 108

5.8 Eagle policies on real videos. The arrows show the vehicle to track in the video scene. The PTZ view is maintained by Eagle while tracking the vehicle. The top images show the starting point where the PTZ view is not focused. 112

5.9 A sample scene with multiple objects of interest. 114

6.1 A *Bug Hunting Workflow* is illustrated which is part of the design verification of integrated circuits at Arm. A machine learning pipeline replaced the default pipeline to predict the preferred input candidates. Mango is deployed on the Dask distributed cluster to automate the hyperparameter tuning of ML models used for design verification. 119

6.2 An example of Mango to tune the hyperparameters of XGBClassifier from the Xgboost library using a parallel scheduler on the local machine. Parameter space consists of distribution, range, and categorical variables. 124

6.3 Skeleton code of Mango on Kubernetes cluster that is deployed as part of the AutoML framework at Arm. Partial *results* are returned by the *objective* function based on timeout. The *conf* data structure modifies the default behavior of *Tuner*. 126

6.4 Skeleton code deploying *MetaTuner* algorithm on the Dask cluster, which is part of the bug hunting application used for design verification of Arm integrated circuits designs. 128

6.5 Comparison of Mango to optimize functions. 134

6.6 Comparison of Mango to tune hyperparameters. 135

6.7 The comparison of Mango's *MetaTuner* for combined classifier selection and hyperparameter optimization problem with other libraries. The evaluation uses five different classifiers (Xgboost, k-nearest neighbor, Support Vector Machines, decision tree, and neural network). Sub-figure (a) is for the Breast cancer dataset, sub-figure (b) the Iris plants dataset, and sub-figure (c) the Wine recognition dataset. Mango performs better than Hyperopt and SMAC and is competitive with Optuna. 137

6.8 Workflow of the AutoML framework using Mango for hyperparameter tuning on the Kubernetes (K8s) cluster. 140

6.9 Performance of Mango for hardware-aware NAS for OxIOD and RoNIN datasets. Subgraphs (a) and (b) illustrate how Mango maximizes resource usage with looser compute and memory constraints to improve error metrics for three different hardware models. Subgraph (c) shows the difference in model size and error metric with and without hardware-in-the-loop (HIL) for the RoNIN dataset on three different hardware models. Subgraph (d) shows the relation between FLOPS and latency for the RoNIN dataset and the difference in error metric with and without HIL. 142

9.1 Delay measurements on deepracer and Intel neural compute stick. The mean is shown in green. The back 'x' marker shows the median of IQR. 152

9.2 Image augmentations applied to enable successful Sim2Real transfer. (a) Original Image, (b) Random Shadows, (c) Random Shadow + Sharpen, (d) Random Shadow + Sharpen + Random noise 153

9.3 The learning curves of Time-in-state and Fixed latency (worst case = 5x4.12 ms) for HalfCheetah task. Fixed latency converges faster. Time-in-state is trained across a vast range of delay variations, and as seen in Figure 4.11 it outperforms worst case policies across a range of delay variations. 156

LIST OF TABLES

2.1 The list of devices that participated in the chirp system clock study: five iOS devices and eight Android devices. 14

2.2 Test accuracy of baseline models 21

3.1 Clock synchronization accuracy across peripherals for three smartphones with respect to a reference Pixel phone. Results are 95% confidence intervals. 40

4.1 Prediction error in neural network inference latency using different approaches and inference latency variations studied by researchers. Across a suite of devices, neural network inference latency can be predicted within 10% error on an average using different proposed approaches. 54

4.2 (a) Comparison of time-in-state (TS) and domain randomization (DR) policies in completing laps on the real track out of 24 trials at different execution latencies. (b) The average speed used by TS and DR. TS adapts its speed with increase in execution latency. 63

5.1 Policy architecture used by Eagle. 91

5.2 Different tracking scenarios in the increasing order of tracking complexity to evaluate Eagle. The goal of Sc-1 to Sc-2 is to track vehicles. Dynamic tasking (DT) trains a policy to track either a vehicle or human characters. 92

5.3 Evaluation of Eagle policies trained for different scenarios in *Fixed background* scene. Sc-1 to Sc-5 are the vehicle tracking scenarios shown in Table 5.2. 95

5.4 Evaluation of Eagle policies trained for different scenarios in *Variable backgrounds+Trees* scenes. Sc-1 to Sc-5 are the vehicle tracking scenarios shown in Table 5.2. 96

5.5 Evaluation of Eagle policies trained for different scenarios in *Variable backgrounds+Trees+Humans* scenes. Sc-1 to Sc-5 are the vehicle tracking scenarios shown in Table 5.2. 96

5.6 Generalization of Eagle policies trained for Sc-4 and Sc-5 (see Table 5.2) scenarios to track $HatchBack_{green}$ and $Truck_{blue}$ vehicles which were not present during training. . 98

5.7 The dynamic tasking (DT) performance of Eagle policies to track either a humans character (DT_h) or a vehicle (DT_v). 99

5.8 %Tracking of Eagle policies at different heights (meters) of the PTZ camera. 100

5.9 Comparison of Eagle with the current state-of-the art approaches for different scene complexities. 103

5.10 Performance of multi-stage approaches when perfect bounding boxes are available from EagleSim simulator. 103

5.11 Performance of PTZ tracking using lightweight object detectors having an architecture similar to the Eagle's policy network. Six different networks are trained to predict the bounding boxes of a car from images. Two image sizes (240×240) and three training datasets (50k, 100k, 150k) are used. *240_150k* refers to the model trained using 240×240 image input on 150k image dataset. In simple tracking scenes having *Fixed background*, all approaches have a very high tracking duration (around 98% Tracking) and maintain other parameters also very well. In complex scenes (*Variable backgrounds + Trees + Humans*), Eagle outperforms the next best approach (*240_150k*) by 16% tracking duration and maintain other metrics also similar. 109

5.12 Inference time in milliseconds (ms) of Eagle and optimized Yolo5s on embedded camera platforms. 111

6.1 Wall clock time (sec) taken by optimizers to sample next evaluation in sequential, parallel, and CASH settings. 138

9.1 Execution latency ($\Delta\tau_{\eta}$) on GAP8 increases with the increase in the number of CNN layers in the neural network. 151

ACKNOWLEDGMENTS

I want to thank all the fantastic people I interacted with at UCLA. My advisor, Professor Mani Srivastava, whose guidance throughout this journey was inspiring and a learning experience that I will cherish forever. I want to thank my thesis committee, Professor Anthony John Nowatzki, Professor Omid Abari, and Professor Puneet Gupta, for their time and suggestions that helped me to improve my research directions.

I want to thank my parents (Sulakhan Singh and Ramandeep Kaur) and younger brother (Jashan Meet) for supporting me. I appreciate the support of NESL members and Industry collaborators who became close friends in this long journey. Special thanks to the computer science department graduate office for their help. I want to thank the past and current members of the Computer Science Graduate Student Association (CS-GSA) who made UCLA a home away from home.

Finally, I want to thank the Almighty, Waheguru Ji, for giving me the strength and wisdom to enjoy this learning adventure.

The research presented in this dissertation is supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, by the IoBT REIGN Collaborative Research Alliance funded by the Army Research Laboratory (ARL) under Cooperative Agreement W911NF-17-2-0196, by the National Science Foundation (NSF) under awards # CNS-1329755 and IIS-1636879. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL, DARPA, NSF, SRC, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

VITA

2010-2014	Bachelor of Technology, Computer Science, IIT Roorkee.
2014	Member of Technical Staff, Oracle.
2014-2016	User Experience Professional, IBM Research.
2016-2018	M.S., Computer Science, UCLA.
2017-2022	President, Computer Science Graduate Student Association (CS-GSA), UCLA.
2021	Outstanding Mentorship Award, Computer Science Department, UCLA.
2022	Student Commencement Speaker, Henry Samueli School of Engineering, UCLA.

PUBLICATIONS

Sandha, S.S., Balaji, B., Garcia, L. and Srivastava, M., 2022, Eagle: End-to-end Deep Reinforcement Learning based Autonomous Control of PTZ Cameras, (Under Review)

Sandha, S.S., Aggarwal, M., Saha, S.S. and Srivastava, M., 2021, December. Enabling hyper-parameter tuning of machine learning classifiers in production. In 2021 IEEE Third International Conference on Cognitive Machine Intelligence (CogMI) (pp. 262-271). IEEE.

Sandha, S.S., Garcia, L., Balaji, B., Anwar, F. and Srivastava, M., 2021, October. Sim2Real

Transfer for Deep Reinforcement Learning with Stochastic State Transition Delays. In Conference on Robot Learning (pp. 1066-1083). PMLR.

Saha, S.S., **Sandha, S.S.** and Srivastava, M., 2021. Deep Convolutional Bidirectional LSTM for Complex Activity Recognition with Missing Data. In Human Activity Recognition Challenge (pp. 39-53). Springer, Singapore.

Sandha, S.S., Aggarwal, M., Fedorov, I. and Srivastava, M., 2020, May. Mango: A python library for parallel hyperparameter tuning. In ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (pp. 3987-3991). IEEE.

Sandha, S.S., Noor, J., Anwar, F.M. and Srivastava, M., 2020, April. Time awareness in deep learning-based multimodal fusion across smartphone platforms. In 2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI) (pp. 149-156). IEEE.

Sandha, S.S., Noor, J., Anwar, F.M. and Srivastava, M., 2019, November. Exploiting smartphone peripherals for precise time synchronization. In 2019 IEEE Global Conference on Signal and Information Processing (GlobalSIP) (pp. 1-6). IEEE.

Sandha, S.S., Cabrera, W., Al-Kateb, M., Nair, S. and Srivastava, M., 2019. In-database distributed machine learning: demonstration using Teradata SQL engine. Proceedings of the VLDB Endowment, 12(12).

Xing, T., **Sandha, S.S.**, Balaji, B., Chakraborty, S. and Srivastava, M., 2018, June. Enabling edge devices that learn from each other: Cross modal training for activity recognition. In Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking (pp. 37-42).

CHAPTER 1

Introduction

Cyber-physical systems, commonly termed CPS, combine sensors and computing with physical or software elements to monitor and control an environment. The first generation of CPS has been using heuristic or first principle algorithms to make decisions. Recently, with the advent of machine learning, the new generation of CPS is increasingly adopting learning-enabled components such as deep neural networks in their processing pipelines. This trend is further exacerbated by the rise of computing platforms providing rich sensing modalities allowing complex inferences directly at the edge.

A typical pipeline of a learning-enabled CPS is shown in Figure 1.1. The different components include sensing of the *Environment* by multimodal *Sensors* such as vision, audio, lidar, and radar. Machine learning models (neural networks) use the sensed data to make an *Inference* for a specific application. The inference is communicated to the *Actuators*. The *Actuators* apply the action that may or not impact the sensed *Environment*. Several CPS applications can be mapped to the pipeline of Figure 1.1. Due to the promise of superior performance, it is desirable to replace traditional decision-making algorithms with neural networks in the decision pipeline of CPS. But adopting neural network models faces several challenges at each stage of CPS. In this dissertation, we specifically focus on the following challenges:

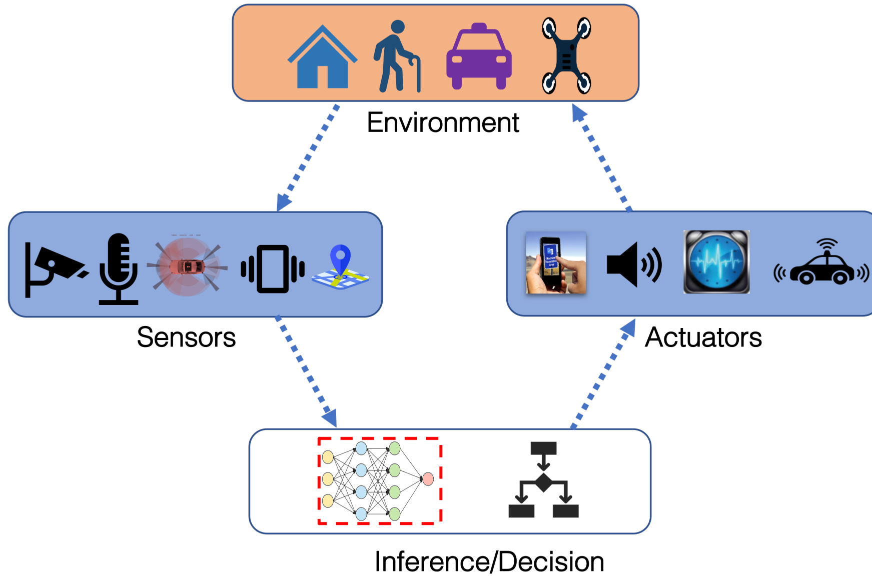


Figure 1.1: Different components in a typical Cyber-physical systems application.

1.1 Challenge 1: Data Misalignment due to Timestamp Uncertainty

Common to many CPS applications is their reliance on the rich suite of sensors present on or accessible by edge devices such as smartphones. With each modality capturing an alternate perspective, its fusion can often improve application performance [NKK11, MHM16]. A key assumption in multimodal fusion is that underlying inputs are synchronized [FBA16, KBM15]; however, this may not be the case when inputs are captured across multiple devices. For example, sensing modalities timestamped using the system clock can be misaligned due to clock errors between smartphones. We focus on the data timestamp uncertainty[1] and its impact on deep learning-based multimodal fusion classifier accuracy.

We present a systematic study to quantify the system clock's accuracy and precision across modern smartphones. The observed clock discrepancies range from tens of milliseconds across iOS devices to as much as 5+ seconds across Android devices. To understand Android's errors, we

[1]By uncertainty [DP12], we refer to the quantity that is not known by the application at runtime. It might be hard to measure, resulting in a tradeoff between lack of information and data acquisition cost.

present an analysis of the procedure by which the Android kernel maintains system time. We analyze the impact of data timestamp uncertainty on the accuracy of deep learning-based multimodal fusion classifiers showing it can cripple the classifier accuracy.

1.1.1 Improving Time and Training Resilient Classifiers

To mitigate the data timestamp uncertainty, we study several approaches to improve time synchronization across smartphones by exploiting their rich peripherals. We introduce the notion of cross-peripheral evaluation to enable applications using multiple sensors across smartphones in deciding the optimal synchronization methodology. For example, our cross-peripheral evaluation results suggest that a single peripheral can synchronize time for multiple sensing modalities for smartphones with similar software stack and hardware. However, in practice, timing synchronization errors are inevitable. To account for that, we propose a novel *Time-Shift* data augmentation technique to develop deep learning classifiers that are resilient to residual timing errors. Our evaluation shows that Time-Shift data augmentation can improve the deep learning-based multimodal classifier's resilience to data timestamp errors of up to 600ms. In essence, Time-Shift data augmentation prevents the classifier from *overfitting on timing characteristics* of data.

1.2 Challenge 2: Delay Awareness in Deep Reinforcement Learning

It is desirable to replace conventional first principle controllers in CPS applications with emerging deep reinforcement learning (RL) policies that enable end-to-end control directly using rich multimodal observations and are shown to have better performance [BMG20, TFR17]. Several applications, such as navigation [BMG19], manipulation [TFR17], and locomotion [HLD19] operate in a closed-loop system where delays from sensing to the actuation significantly impact the application performance. We show that handling variable delay is crucial to ensuring the success of deep reinforcement learning (RL) policies.

The inference latency is a leading contributor to the end-to-end delay from sensing to actu-

ation for neural network policies trained using deep-RL. Various factors can result in variable inferencing latency at runtime, such as power management, computational resources, and complex operating system (OS) environments [WWD94, TSS17]. The sensors' sampling rate can also vary widely depending on both software and hardware characteristics [SBB15]. Traditional controllers can model and incorporate the variable delays in their design [Nil98, LR90]; however, in the context of deep reinforcement learning, black-box neural networks don't allow explicit delay modeling. We study the impact of variable[2] sampling rate and end-to-end delays on the neural network policies trained using deep-RL. Currently, researchers widely use the approach of domain randomization [PAZ18, ABC20] and worst-case delay modeling [KE03, SBB10] to train deep-RL policies in the presence of variable delays. We demonstrate that deep reinforcement learning-based controller performance degrades due to variability in delays even when using the state-of-the-art domain randomization technique.

1.2.1 Handling Variable Delays

We propose Time-in-State RL, a delay-aware Deep-RL approach. The Time-in-State approach extends the system's observed state by explicitly including temporal properties (such as sensor sampling rate and execution latency) during training. During deployment, the Time-in-State approach monitors the temporal properties to adapt its actions. Our evaluation of Time-in-State in simulation and on a real robot using a scale car shows its superior performance compared to domain randomization. We also compare Time-in-State with the worst-case delay controllers showing its superior performance when worst-case delays are significant. Time-in-State showcases a practical approach to train delay adaptive deep neural networks for control pipelines utilizing complex modalities.

---

[2] By variability, we mean a quantity that can be measured directly/indirectly at runtime by the application; thus, an intelligent application can learn to adapt.

1.3 Challenge 3: Realizing Training Environments to enable End-to-end Control

The realization of end-to-end control, such as Deep-RL policies for CPS applications, is preferred over the traditional multi-stage approaches as it is hard to optimize multiple stages jointly. However, deep-RL policies are extremely data-hungry and thus are very difficult to train in the real world. Thus, researchers train these policies in simulators. Training in simulators opens up a challenging question of how to realize accurate training environments to transfer policies from simulation to real-world deployments.

To study end-to-end control, we focus on the application of pan-tilt-zoom (PTZ) cameras to track objects of interest. Existing autonomous PTZ controllers have multiple stages, namely detection of objects, tracking of their trajectories, and control of PTZ parameters to keep objects in the field-of-view [BVS07, CSB15, MRF10]. It is common to use neural object detectors to identify objects of interest, first-principles-based algorithms such as Kalman filters for trajectory prediction using model-based state estimation on bounding boxes, and a separate controller [BGO16, UNC19, LMC21].

The multiple stages suffer from performance bottlenecks as it is non-trivial to tune each step. For example, tuning the parameters of the Kalman filter requires expert domain knowledge and can incur many trial-and-errors [Kyr21,LSZ19]. Further, it is often infeasible to run multi-stage control algorithms in real-time on platforms [Ard22, Kyr21] having memory and computation constraints. For example, even the lightweight object detectors (such as YOLO [RF17, Kyr21] with several millions of network parameters are too complex for embedded camera platforms. Even though multiple stages have lower performance and are computationally demanding, developers often have no choice as it is very challenging to develop and evaluate controllers in the real world [PAZ18, BMG20].

1.3.1 Deep Reinforcement Learning-based Controllers for PTZ Cameras

We propose *Eagle*, an end-to-end deep-RL approach using raw images to control a PTZ camera. Eagle trains a neural network policy directly mapping raw images to pan-tilt-zoom actions removing the multiple stages of object detection, localization, and control. To our knowledge, there has not yet been any attempt to develop an end-to-end Deep-RL policy for PTZ cameras. This is partly due to the challenges of training deep-RL in the real world as it requires a large number of environment interactions, expensive experimental setups, and labeling efforts [LSZ19, BMG20, ABC20, PAZ18] and the difficulty of creating PTZ tracking scenarios in the real world. To enable successful training of Eagle, we also introduce *EagleSim*, a simulation framework for placement and control of PTZ cameras in photo-realistic virtual worlds.

Addressing simulation-to-reality gap: Although training in simulators is extensively explored in deep-RL as it is cheaper and safer, transferring simulation policies to the real world is still a challenge [BMG20, PAZ18, LSZ19]. EagleSim includes a significant engineering effort to provide scene variations with multiple objects (vehicles and human characters) and different surroundings (background materials/patterns and trees). We highlight that these rich scene variations are necessary to bridge the simulation and the real-world gap.

1.4 Challenge 4: Training Machine Learning Models with Optimal Hyperparameters in Production

Enabling the training of machine learning models at a production scale is crucial in providing optimal classifiers for emerging CPS/IoT applications. However, a typical machine learning training pipeline in production can be too specialized and complex, demanding a trained team of human experts with specific domain knowledge for classifier selection with optimal hyperparameters. Automating hyperparameter tuning in production is becoming increasingly difficult, with many choices for classifiers and their rich parameter spaces. Consequently, to speed up the search, *intel-*

ligent parallel algorithms utilizing parallel computing are needed.

1.4.1 Enabling Hyperparameter Tuning in Production

Although several hyperparameter tuning frameworks exist, they have limitations in their search algorithms, include software dependencies, or are closed source. To enable state-of-the-art hyperparameter tuning in production, we propose the design of a lightweight library (1) having a flexible architecture and (2) parallel optimization algorithms allowing combined classifier selection and hyperparameter tuning. We present *Mango*, a parallel hyperparameter tuning library, to realize the proposed design. Mango is available open-source and is currently used in production at Arm for more than 30 months. Our evaluation shows that Mango outperforms existing libraries in tuning hyperparameters of ML classifiers when complex search spaces are explored. We discuss two use cases of Mango deployed in production at Arm, highlighting its flexible architecture and ease of adoption. The first use case trains ML classifiers using Mango to design Arm's commercial integrated circuits. We introduce an AutoML framework using Mango to train optimal classifiers as a second use case. Finally, we present the third use-case of Mango in enabling neural architecture search (NAS) to transfer deep neural networks to TinyML platforms (microcontroller class devices) used by CPS/IoT applications.

1.5 Dissertation outline

The above challenges and our solution approaches are organized in different chapters as follows:

- Chapter 2 characterizes the data timestamp uncertainty across modern smartphone devices. We evaluate the impact of data misalignment on a machine learning classifier's accuracy using a representative human activity recognition application. Finally, we present the vision of making deep learning-based multimodal fusion time-aware.

- Chapter 3 presents different approaches to improve the shared notion of time across smart-

phones. We evaluate the accuracy of time synchronization approaches realized using audio, Wi-Fi, and BLE peripherals across smartphone devices. Based on our findings, we discuss the limitation of each synchronization approach and provide recommended application-specific solutions.

- Chapter 4 studies the impact of delay variations on the performance of deep-RL policies. We present a new Time-in-State RL approach to bring delay-awareness to the RL-based end-to-end control.

- Chapter 5 introduces the challenges in the existing multi-stage pipeline of autonomous controllers for PTZ cameras. We discuss our approach to developing training environments to enable end-to-end control policies. We present Eagle, an end-to-end deep-RL approach mapping input raw images directly to the pan, tilt, and zoom actions. To enable the development of Eagle, we also introduce a new software simulator environment and show the direct transfer of Eagle policies from simulation to real scene videos.

- Chapter 6 focuses on training machine learning classifiers with optimal hyperparameters in production. We discuss existing frameworks' shortcomings and introduce a new open-source library called Mango. Mango is designed to be production scalable with state-of-the-art parallel search algorithms. We discuss several use cases of Mango, highlighting its superior performance and flexible deployment architecture.

- Finally, we present the possible future directions in Chapter 7 and the conclusion in Chapter 8.

CHAPTER 2

Data Timestamp Uncertainty and its Impact on Multimodal Fusion

2.1 Distributed Sensing using Smartphones

Modern smartphones are used by a wide variety of emerging applications, such as those in crowd-sensing, smart homes, mobile health, transportation, and public safety [Kan11, JLM15, JPG13, KB13, LPR13, WTX14]. Common to many of these applications is their reliance on the rich suite of sensors present on or accessible by smartphones. Given the broad outreach of smartphones, we study the timestamp uncertainty in data captured across modern smartphone devices.

2.1.1 Data Timestamping in Smartphones

Developers often rely on the smartphone system clock to timestamp multimodal sensor data across devices (e.g., [HHS17]). The system clock is (i) universally available despite the heterogeneous and fragmented smartphone ecosystem, (ii) easily accessible to geo-distributed applications (even without network connectivity), and (iii) does not require access to external hardware peripherals. Given the universal applicability of the system clock it is critical for developers to be aware of the possibile range of system clock errors across smartphones.

Researchers have previously noted that the system time across smartphones is unreliable, but have yet to quantify and characterize these errors [LRS15b, MDB16]. Although there have been previous studies conducted to understand the variable characteristics of the Network Time Proto-

col (NTP) [Mil91] for various wired, wireless and embedded devices [MDB16], smartphones have remained excluded from these analyses. Due to the protected system clock update mechanisms on smartphone platforms (barring a rooted device), minimal access to the timing stack has restricted the ability to thoroughly characterize these systems. In this chapter, we present a systematic study to quantify the system clock's accuracy and precision across a suite of modern smartphones. Observed clock discrepancies range from tens of milliseconds across iOS devices to as much as 5+ seconds across Android devices. To understand these drastic errors, we present an analysis of the procedure by which the Android kernel maintains system time.

2.1.2 Deep Learning-Based Multimodal Fusion

With each data modality capturing an alternate perspective, its fusion can often result in improved application performance [NKK11, MHM16]. Given the wide success of deep learning, emerging applications are transitioning towards approaches of using deep learning for multimodal fusion [NKK11], which can achieve state-of-the-art accuracy and provide robustness to noise in sensors [RRV19, ESS15]. A fundamental assumption in multimodal fusion is that underlying inputs are synchronized [FBA16, KBM15]; however, this may not be the case when inputs are captured across multiple devices. For example, sensing modalities timestamped using the system clock can be misaligned due to clock errors between smartphones [XSB18]. Although researchers have previously designed traditional machine learning approaches to handle asynchronous modalities [DL00, Ben03], deep learning-based approaches have yet to be evaluated. We provide the quantification of the impact of timing errors on the accuracy of a deep learning classifier fusing modalities. Specifically, we evaluate an architecture fusing modalities at an intermediate layer for human activity recognition, which is shown to outperform the traditional techniques of feature concatenation and model ensemble [RTB18, OSG19, ESS15, KZX11]. We show that timing errors of 600ms can result in a 3% drop in the classifier accuracy, and an error of 5 seconds can degrade accuracy by up to 25%. To the best of our knowledge, this is the first analysis of the impact of timing errors on deep learning-based sensor fusion.

The significant impact of timing errors on deep learning classifiers over multi-device input could be theoretically avoided with perfect synchronization. However, in practice, data timestamp errors are unavoidable. To address asynchronous data, we propose an extensible two-fold solution; we suggest (1) improving time synchronization across smartphones by replacing the system clock with an application-level timing library based on NTP, and (2) making deep learning models resilient to residual timing errors using *Time-Shift* data augmentation. Our evaluation shows Time-Shift data augmentation can improve the deep learning-based multimodal classifier resilience to timing errors of up to 600ms. Another way to look at the Time-Shift data augmentation approach is to prevent the classifier from *overfitting on timing characteristics* of data.

2.2 Background and Related Work

Smartphone System Clock: The two methods currently used by the smartphone platforms for clock synchronization are the Network Identity and Time Zone (NITZ) [Sca18] and Network Time Protocol (NTP) [Mil91]. Android employs both NITZ and NTP for maintaining system time. To guarantee precise timing across Apple Watches, Apple built and deployed their own set of Stratum One NTP timeservers [Ula15]. They claim their watches are accurate to within ± 50 milliseconds of UTC as measured by atomic clocks [Wil15].

Time Synchronization Approaches: The synchronization approaches proposed for wireless sensor networks, including reference broadcasts [EGE02], TPSN [GKS03], FTSP [MKS04], and PulseSync [LSW15], while theoretically possible, are currently not achievable in practice across smartphones due to the requirements of a shared wireless connection, specialized MAC-layer timestamping, and timing stack control. Although smartphones have a GPS module, vendor-specific implementations [MDB16] and poor indoor signal prohibit GPS-based time synchronization. Several synchronization works that exploit ambient signals, including skin electric potential [YLT17], powerline radiation [RGR09], and radio data system [LXS11] require specialized external hardware, thereby limiting their adoption.

Deep Learning-Based Multimodal Fusion: The traditional approaches of multimodal fusion include feature concatenation and ensemble classifiers [OSG19,NKK11]. Recently, researchers have proposed deep learning architectures [NKK11] doing mid-layer fusion on the extracted features from individual modalities, which achieves state-of-the-art accuracy across domains including human activity recognition [RTB18], object detection [ESS15], and emotion recognition [OSG19].

Classifier Resilience to Timing Errors: Handling asynchrony across audio-visual modalities [DL00, Ben03] has been studied in the context of the traditional classifiers. Dupont et al. [DL00] learn asynchrony between audio-visual modalities for a Multistream Hidden Markov Model (HMM). Asynchronous HMM is presented by Bengio et al. [Ben03] to handle audio-visual modalities. Wollmer et al. [WAE09] present a dynamic time warping algorithm to fuse asynchronous data streams. To the best of our knowledge, the impact of asynchronous data streams due to the timing errors across smartphones have not been evaluated for deep learning-based multimodal fusion.

The approach of Model refinement [RNG18] uses new labeled data to account for variations in the deployment setting. In contrast, we introduce a novel time-shift augmentation technique to improve the classifier resilience to timing errors without requiring the collection of new deployment-specific labeled data. Time-shift augmentation adds artificial misalignments between modalities to account for variations in timing characteristics during deployment. The introduced augmentation also compliments application-specific approaches that either observe or add selective events (like chirps) in data to align the data streams [FBA16, PCH12].

The notion of learning models over labeling uncertainty has been addressed by Adams and Marlin [AM17, AM18]. They developed techniques to account for errors in human labeling of time-series data, proposing frameworks for learning time-series detection and segmentation models from temporally imprecise labels. The underlying sensor data is assumed to be time-synchronized; in contrast, our work focuses on the imprecision of the input data streams themselves.

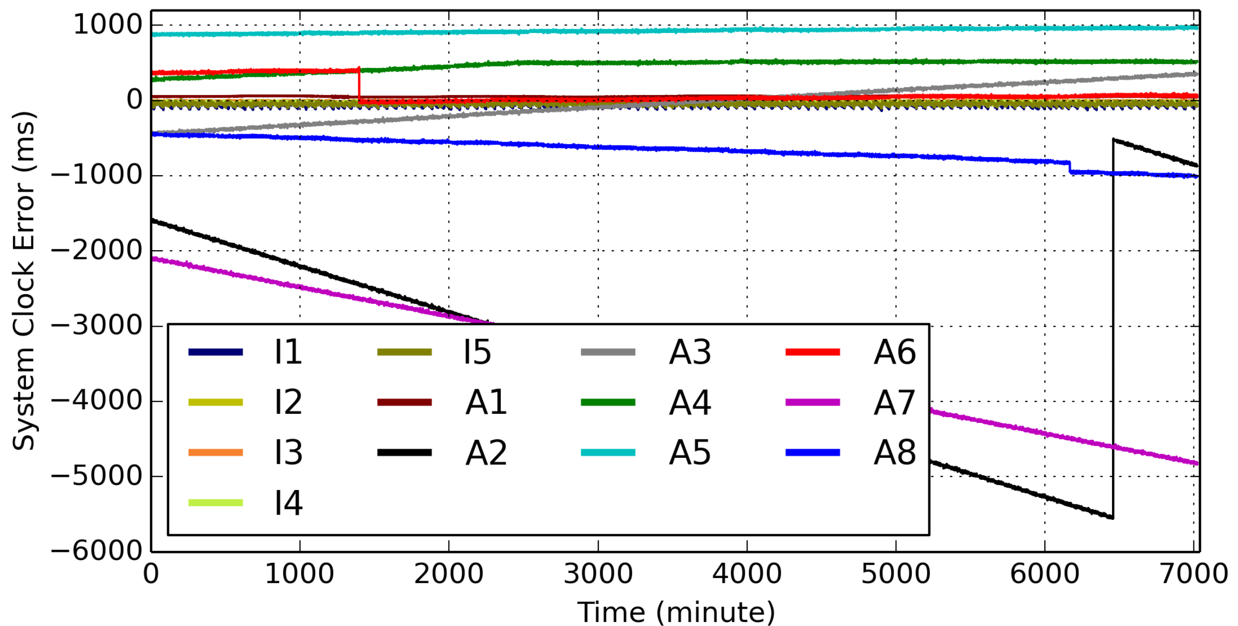


Figure 2.1: Observed system clock errors across all devices when compared to an NTP baseline during the undisturbed 5-day study. Three significant time adjustments (>100ms) occur: A6 at minute 1396, A8 at minute 6166, and A2 at minute 6457.

2.3 Smartphone System Clock Study

To provide a quantitative analysis of the timing errors present on smartphones, we conduct an observational study to reveal system clock errors. Table 2.1 describes the participating device specifications. These 13 devices were placed in an isolated room within 50cm of a speaker that played a periodic square wave *chirp* approximately every twenty seconds. Each device ran an application in the foreground that monitored the microphone. When the chirp event's low-to-high transition is observed by the application, the device system time is recorded. During the study, the background system tasks were running to allow typical device behavior. We used the audio subsystem to measure the system clock due to its high sampling rate. The per-device delay in the audio pipeline is typically around 10-60ms [Inc18] for smartphones, making it a suitable candidate to compare system clocks at a coarser granularity.

Table 2.1: The list of devices that participated in the chirp system clock study: five iOS devices and eight Android devices.

ID	Device	OS	Year	SIM?
I1	iPhone 6	iOS 12.1.4	2014	N
I2	iPad Pro 9”	iOS 12.1.4	2016	N
I3	iPhone 7+	iOS 12.1.4	2016	N
I4	iPhone 6S	iOS 12.1.4	2015	N
I5	iPhone 6	iOS 12.1.4	2014	Y
A1	Nexus 5X	Android 8.1.0	2015	Y
A2	Nexux 7 Tab	Android 6.0.1	2012	N
A3	Huawei P9	Android 7.0	2016	N
A4	OnePlus A1	Android 5.1.1	2014	N
A5	Samsung GTS2	Android 7.0	2015	N
A6	Nexus 5X	Android 8.1.0	2015	Y
A7	Nexus 7 Tab	Android 6.0.1	2012	N
A8	Pixel 3	Android 9.0	2018	Y

All devices had the “Set Time Automatically” setting enabled and were connected to campus Wi-Fi. A T-Mobile unlimited data plan was used for devices with SIM cards due to its NITZ support. To ensure a warm start, devices were on and connected to the internet for at least 24 hours before data collection. As such, system clock errors observed at time 0 refer to devices that have been online for some time.

2.3.0.1 5-day Study

Over a span of five days, periodic audio chirps were independently observed and timestamped by each device. Figure 2.1 presents the approximate system clock error with respect to a baseline

over the course of five days. The baseline was generated by an application-level NTP clock that frequently synchronized to a fixed pool of timeservers. Given that typical NTP error is on the order of tens of milliseconds [LRS15b], and an audio latency variability with similar magnitude [Inc18], the reported system clock errors are accurate on the order of tens of milliseconds. The overall spread of error ranged from 3-6 seconds. All major system clock errors can be attributed to Android devices; iOS devices always timestamped within 110ms of the baseline NTP clock.

Three Android devices corrected their system clocks once during the five days; the other five never adjusted their clock. Two of the three Android corrections, A8 at minute 6166 and A2 at minute 6457, still resulted in significantly erroneous system clocks. Of the three Android devices with an active SIM card, two performed a system clock adjustment, one increasing clock error, and the other decreasing clock error. The older devices, tablets A2 and A7, presented the largest clock errors. Newer Android devices kept a smaller error spread of around 2 seconds.

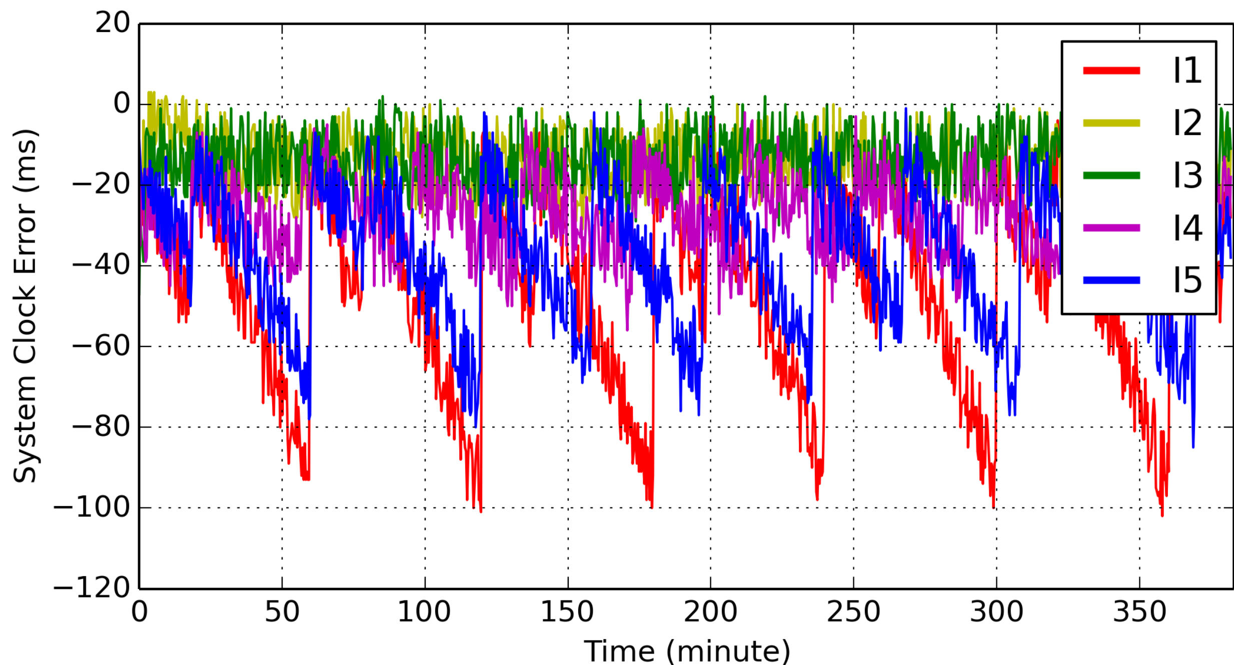


Figure 2.2: Observed iOS system clock errors during a 6-hour snapshot of the 5-day study. iOS errors are an order-of-magnitude less compared to Android due to the aggressive clock correction that occurs at semi-periodic intervals.

In contrast to Android, iOS devices maintained significantly more accurate system time. Figure 2.2 presents iOS system clock errors over a 6-hour window of the 5-day study. The spread of error is an order-of-magnitude less than the Android devices; iOS appears to correct the system clock aggressively. This results in a semi-periodic trend in which the clock drifts away and is adjusted approximately every hour. No iOS system clock ever reported a chirp time that was more than 110ms different from the baseline NTP time. The presence of a cellular connection seemed to have no notable effect on the system clock time of iOS devices.

2.3.0.2 Drift

Due to manufacturing variations, the device's timekeeping oscillators deviate from their nominal frequency. As a result, system clocks derived from these oscillators drift in time with respect to each other. Older Android devices, such as A2 and A7, drifted 37 and 24 ms per hour, respectively, amounting to a total change in the system clock time of 3-4 seconds. Newer phones such as A1 and A5 drifted less than 1ms per hour. Similarly, the older iPhones I1 and I5 drift much more rapidly compared to newer iOS devices.

2.3.0.3 Jitter

Each device presented a variability in recorded system clock offset between two consecutive chirps. The contributing factors include variance in audio latency, audio data processing, system clock read latency, NTP baseline, and application/system management. The distribution of jitter was approximately gaussian across devices with an average near zero. The standard deviation of jitter ranged from 10-15 ms for all devices except for A2 and A7, which presented a standard deviation of around 60ms.

2.3.1 Understanding Android System Time

The errors encountered across Android devices demanded further investigation. To understand the mechanisms by which Android performs time synchronization, we studied the Android operating system kernel. A detailed description of this process, as of Android 10.0, is as follows[1]:

The Android system clock time is set via two protocols, NITZ and NTP, in which NITZ receives priority. The Radio Interface Layer (RIL) socket is monitored for a NITZ packet arrival, upon which the system clock is updated. If a NITZ timing update has not occurred within a specified window specified by the configuration (set to 24 hours) `config_ntpPollingInterval`, NTP time is used. NTP updates are triggered at 4 key action points: (1) when the phone boots up, (2) when the "set time automatically" option is enabled, (3) when the default network connection changes, and (4) when a timer fires (set and always reset to 24 hours since the last NTP trigger).

Independently of whether an NTP sync is performed, the system clock is only corrected if the following conditions are met: (1) no NITZ update has occurred, (2) a successful NTP sync has occurred within the previous 24 hours, and (3) the system clock differs from NTP time by at least `config_ntpThreshold` (5 seconds). The system clock is updated using `settimeofday`, a system call that directly overwrites the current system time, resulting in a direct jump that affects clock monotonicity (i.e., time may go backward).

2.3.2 Forcing a Sync Event

With the knowledge of how the Android kernel is updating its system clock, we performed a restart event to forcefully trigger an update to the system clock in all devices simultaneously. The observed timing errors after this event represents a best-case scenario given the current system design.

After the restart event, iOS devices showed the least notable change. All five Android devices

[1]To absolve these issues, we submitted a simple patch to Android [Noo19]. Our patch was reviewed, but ignored, and instead, refactoring changes were made that reduce (but still do not fix) these core issues.

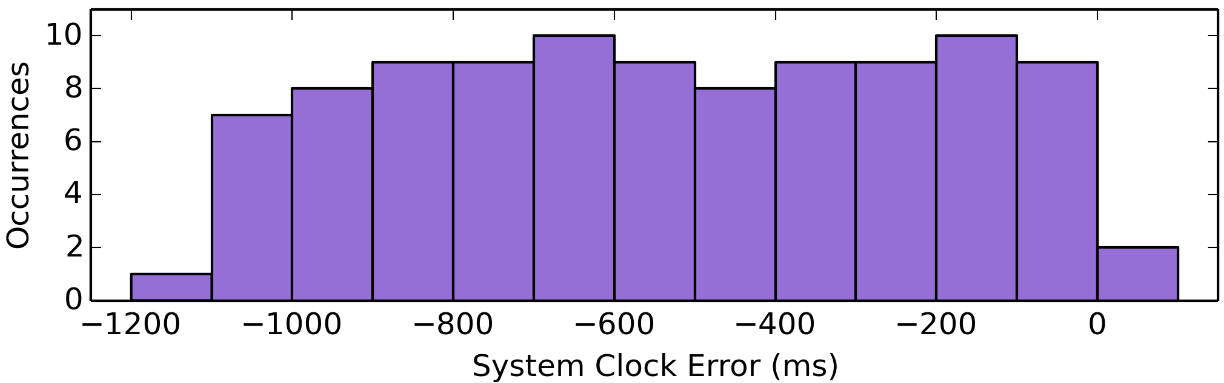


Figure 2.3: System clock error of an Android Nexus 5X device after triggering 100 independent NITZ timing updates.

without a SIM card performed an NTP sync and update, resulting in a system time error within ±40ms of an NTP baseline. The three NITZ-enabled devices, A1, A6, and A8, updated their system clock times to an error between -295 and -830 ms. Despite all devices updating their system clock time, the total spread of error after the restart was still over 800ms.

2.3.3 NITZ vs NTP

Given the Android prioritization of NITZ over NTP, we performed an analysis of the accuracy and variability of these two protocols. To generate a NITZ sync, a phone with a SIM card was restarted. To generate an NTP sync, the SIM was removed, and the phone was restarted. In order to verify the true cause of each update, we performed this study on a rooted phone that logged the cause for each update. This process was repeated 100 times to generate a distribution of errors with respect to a baseline.

The total system clock error due to NITZ updates, as shown in Figure 2.3, ranged from -1190ms to 70ms. NITZ time notably lags behind the NTP baseline. In contrast, NTP updates as shown in Figure 2.4 are significantly more accurate in comparison to NITZ; 75% of all updates were within 10ms of the NTP baseline, and 90% within 20ms. This shows that the application level solution

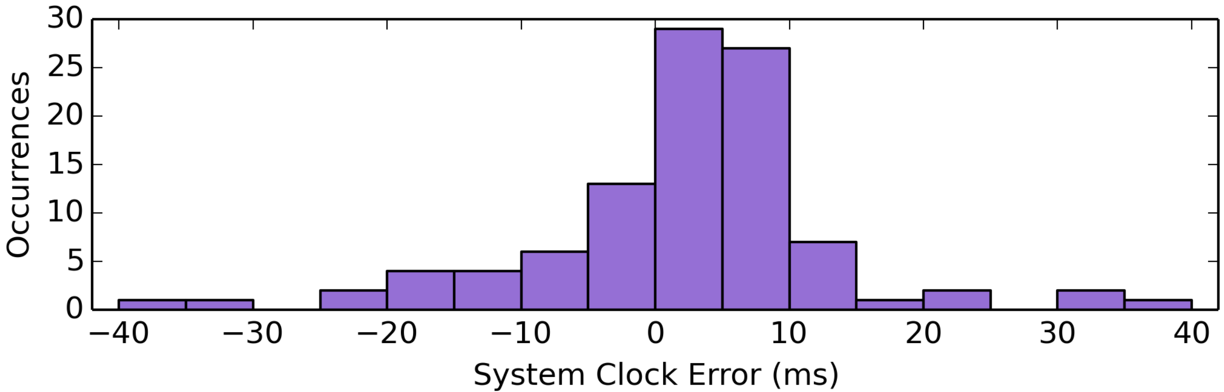


Figure 2.4: System clock error of an Android Nexus 5X device after triggering 100 independent NTP timing updates.

based on NTP can provide an accurate time in the order of tens of milliseconds.

2.4 Impact of Timing Errors on Deep Learning-Based Multimodal Fusion

System clock discrepancies range from tens of ms across iOS devices to as much as 5+ seconds across Android devices. For distributed applications that combine data across smartphones, tablets, wearables, or other Android devices, the reliance on system time can lead to timing errors on the order of seconds, with potential jumps of 5 seconds or more. Next, we analyze the impact of timing errors of this magnitude on a popular application leveraging multimodal data fusion to enhance the accuracy of a deep learning classifier.

2.4.1 Multimodal Deep Learning for Human Activity Recognition

We leverage a multi-device dataset captured across smartphones to train a classifier for human activity recognition. Our multimodal neural network architecture fuses modalities in an intermediate layer [RTB18, OSG19, ESS15], which is shown to outperform traditional approaches of feature concatenation and ensemble classifiers.

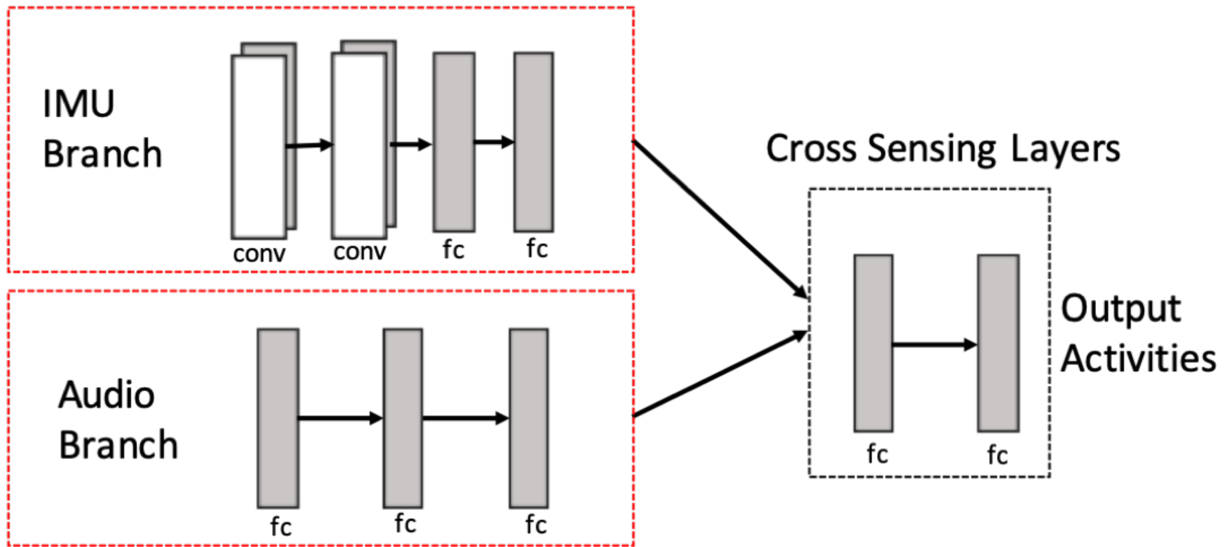


Figure 2.5: The architecture of Multimodal Audio-IMU Network with separate branches fused in cross sensing layers. *Conv* and *fc* refer to convolutional modules and fully connected layers, respectively.

2.4.1.1 Dataset

We use the CMActivities dataset [SX19] containing video, audio, and IMU modalities collected using two smartphones from users performing activities. We refer to the user performing activities as performer hereafter. Here we explain, the data collection setting for one performer. An observer is holding the first smartphone, which is used to record and timestamp the video (along with audio) of the performer. In this way, the first smartphone is acting as an ambient sensor that is recording (video and audio) the performer. The second smartphone was in the trouser's front pocket of the performer. The second smartphone is used to timestamp the IMU data captured from the left and right wrist sensors worn by the performer. Both smartphones were synchronized using NTP.

The dataset is collected by repeating the above procedure with another performer. The data is obtained for seven activities (upstairs, downstairs, walk, run, jump, wash hand, and jumping jack). Every data collection session roughly lasted for 10 seconds, where the performer performed a singular activity. Due to the presence of multiple smartphones used in data collection, this dataset

illustrates a representative scenario for evaluating the impact of time synchronization errors when fusing modalities across smartphones. The dataset was split into 624 training sessions and 71 testing sessions, where each split contains data from both performers.

2.4.1.2 Baseline Models

Three models were built: (i) IMU data only, (ii) audio data only, and (iii) combined audio+IMU data. To show the benefits of data fusion, we compare individual classifiers with a multimodal classifier. We use the method of Xing et al. [XSB18] to train classifiers by using extracted audio features and raw IMU samples. For IMU data, the device sampling rate of 25Hz was fluctuating. We downsample it to 20Hz.

(a) Audio Network is motivated by the acoustic event classifier used by Gencoglu et al. [GVH14] and has five fully connected layers with a total of 157K parameters.

(b) IMU Network has two convolutional modules (convolution + maxpooling layers) followed by four fully connected layers. It has 130K parameters and is based on the classifier proposed by Yang et al. [YNS15] for human activity recognition.

(c) Multimodal Audio-IMU Network has separate branches for audio and IMU modalities. These different branches are joined in the unifying cross sensor layers, as shown in Figure 2.5. It has a total of 287K parameters.

Table 2.2: Test accuracy of baseline models

Networks	Audio	IMU	Multimodal Audio-IMU
Test Accuracy	91.34%	90.10%	96.12%

The test accuracies of Audio Network, IMU Network, and Audio-IMU Network over an average of five training experiments are shown in Table 2.2. Multimodal Audio-IMU Network has around 5% better accuracy than the individual networks. This is due to the superior feature representations with access to observations from alternate perspectives [NKK11, RTB18]. As such,

multimodal networks can be an ideal choice, but sensor fusion requires synchronized modalities. To this end, we evaluate the impact of synchronization errors on the performance of the Multimodal Audio-IMU network.

2.4.1.3 Inducing Time Synchronization Errors

To imitate a real-world scenario where timing errors exist across smartphones, we induce errors by shifting the IMU data with respect to audio data. With a 20 Hz IMU sampling rate, one sample shift corresponds to 50ms of sync error. We incrementally introduce these shifts in the test data only and evaluate the accuracy of the Multimodal Audio-IMU Network. A range of 0 to 100 samples shift corresponds to 0 to 5 seconds of effective time sync errors. As shown in our timing study in Section 2.3, the errors of 5 seconds can occur between two smartphones. When shifting IMU samples, activity labels are aligned with the audio data.

Figure 2.6 shows the degradation in the accuracy of Multimodal Audio-IMU Network with timing errors. The accuracy drops due to a combination of the following two reasons: first, timing errors cause modalities to capture different windows over the same overall activity. Second, the shift in IMU data samples across activity transition boundaries results in incorrect classification. The impact of transitions is more pronounced for frequently changing activities. To evaluate the contribution of activity transitions, we replicate testing the model with varying activity periods. **(i) 10-Sec Period**: Each activity spans 10 seconds. For example, a user performs *walking* for 10 seconds, then *upstairs* for 10 seconds, and so-on. **(ii) 20-Sec Period**: In this setting, we consider each activity spans for 20 seconds. **(iii) 60-Sec Period**: Finally, we consider a case where users perform each activity for 60 seconds.

With no time sync errors, accuracy is the same for all periods. As shown in Table 2.2, fusion improves accuracy by 5% over individual classifiers. Although deep learning can handle slight misalignment, significant timing errors can cause accuracy to drop by up to 25%. For *10-Sec Period*, 1000ms of time errors result in the Multimodal Audio-IMU Network accuracy to drop to

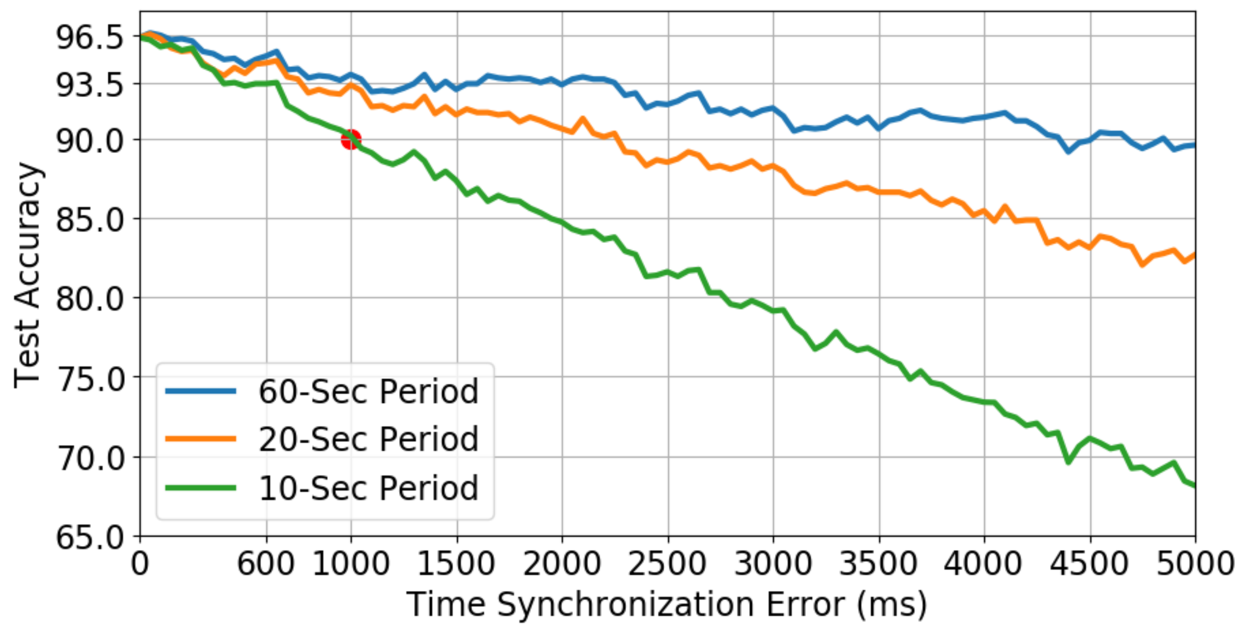


Figure 2.6: Variation in the accuracy of the Multimodal Audio-IMU network with increased timing errors between smartphones collecting audio and IMU data. *10-Sec, 20-Sec*, and *60-Sec* periods represent varying duration between activity changes. 1000ms of timing error results in an accuracy drop of 6% for *10-Sec Period*.

90%. This shows that time errors of one second are enough to negate the benefits of fusion. The larger periods (*20-Sec* & *60-Sec*) also suffer degradation in accuracy within 2-3% with 1000ms errors. To avoid the need for time sync across smartphones, a central hub can be used to timestamp the data, and classifiers can use the most recent window to make the inference. However, this approach may suffer from asymmetric transmission delays resulting in misaligned modalities. Future work is needed to study the magnitude of this misalignment.

2.5 Strategies to Mitigate Timing Errors

Through the study conducted in Section 2.3, the Android system clock has been determined to have timing errors on the order of seconds. We now discuss mitigation strategies to handle the timing errors in smartphones.

2.5.1 System Clock Replacement

In the cases where intermittent network connectivity is available, developers can leverage application-level NTP solutions. To this end, we offer GoodClock [SN19], an easy-to-use library, to enable high-quality time synchronization over wide-area networks. GoodClock is available for iOS, Android and also has a python implementation to extend usage to IoT devices. GoodClock includes drift correction and performs multiple successive NTP requests, selecting responses with the lowest round-trip time. Our evaluation of the library, available at [SN19], reduces clock errors from seconds to a few milliseconds. Smartphones in the future can improve system clock by adopting the Precision Time Protocol (PTP) [EL02]. Since PTP is designed mostly for local area networks, its extension to wirelessly networked smartphones needs future exploration.

2.5.2 Time-Shift Data Augmentation

Even with the best synchronization techniques, timing discrepancies are unavoidable. We advocate the design of applications, which are robust to the expected timing errors. To train models that are resilient to timing errors, we present *Time-Shift* data augmentation. Time-Shift data augmentation exploits the observation that timing characteristics are variable for the devices capturing data during training and deployment.

We modify the training data by artificially shifting IMU data samples with respect to audio. This results in an overall time shift between input data sources. For the purposes of our evaluation, we generate two different augmented datasets. Note that the validation and test datasets are not augmented.

Augmentation 50-100ms: Two datasets are generated by shifting IMU data (20Hz) by 50ms (1 sample) and 100ms (2 samples) with respect to the audio data. We add these two augmented datasets to the original training dataset.

Augmentation 50-1000ms: In this augmentation, we add four augmented datasets to the original training data by shifting the IMU modality by 50ms (1 sample), 100ms (2 samples), 500ms (10 samples), and 1000ms (20 samples) with respect to audio.

Figure 2.7 shows the variation in accuracy of Multimodal Audio-IMU Network trained using augmented datasets due to time errors in the test dataset. Data augmentation enables input data to cover the unexplored space of modality misalignment. With no artificially induced timing errors, Time-Shift data augmentation provides a boost in accuracy, thus confirming the fact that timing errors are inherently present in the training data and more generally in any deployment setting.

The test data is considered with the *10-Sec Period*, as discussed in Section 4.1.4. Figure 2.7 shows that the augmented models perform significantly better in comparison to the baseline. With time synchronization errors of 1000ms between modalities in test data, the accuracy of the Multimodal Audio-IMU Network trained on original training dataset drops to 90.1% (6% drop), whereas the accuracy of Multimodal Audio-IMU Network trained using augmented datasets only drops

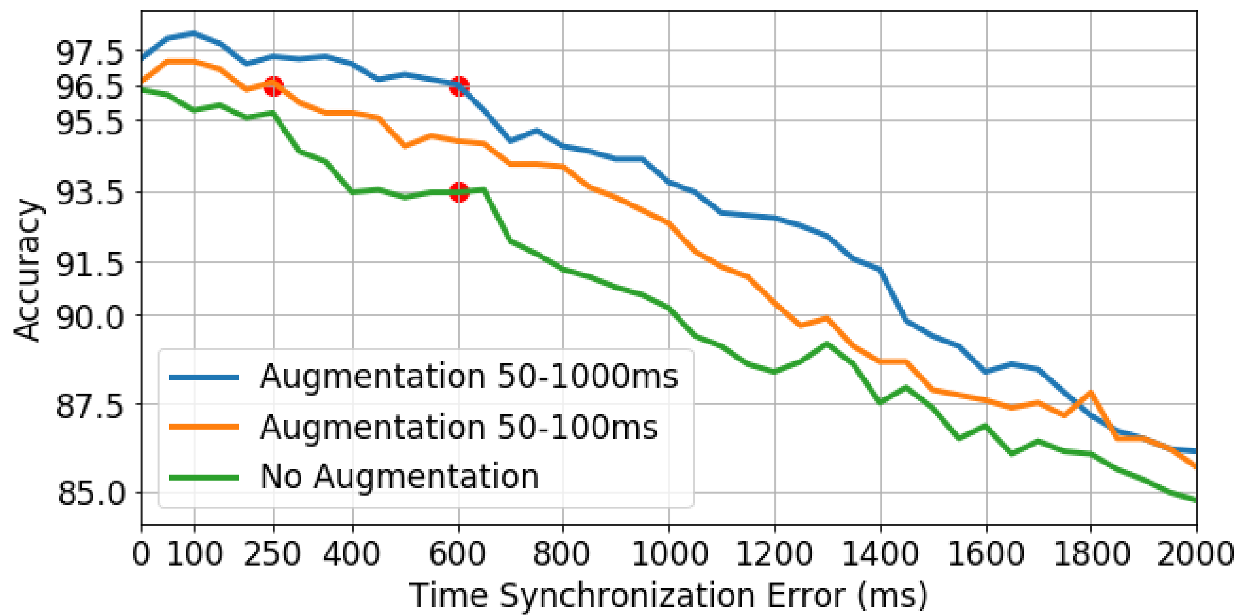


Figure 2.7: Comparison of Multimodal Audio-IMU Network test accuracy with different augmented training datasets. The augmented models can preserve classifier accuracy despite 250ms and 600ms of timing error. Without augmentation, 600ms of error results in model accuracy drop by 3%, from 96.5% to 93.5%.

to 93.75% and 92.5% for *Augmentation 50-1000ms* and *Augmentation 50-100ms*, respectively. This shows that data augmentation can achieve notably improved classifier resilience, even with 1000ms of timing errors. As seen in Figure 2.7, *Augmentation 50-100ms* maintains the accuracy at 96.5% even with an error of 250ms, and *Augmentation 50-1000ms* maintains classifier accuracy of >96.5% even for errors of up to 600ms. In contrast, without augmentation, accuracy drops by 3% from 96.5% to 93.5%. By artificially time-shifting to enrich training data, data augmentation improves overall model performance and resilience to timing errors.

Once significant timing errors are introduced, all trained models degrade and suffer from notable declines in the accuracy. This suggests that Time-Shift data augmentation helps mitigate timing errors to a reasonable extent, after which improvements are lost. Therefore, an accurate shared notion of time across devices for deep learning-based multimodal fusion is a necessity; the inclusion of a system clock replacement like GoodClock with the Time-Shift data augmentation approach compliment each other. Prior knowledge of the expected time errors may guide the augmentation strategy in deciding the number of shifts necessary in training data.

2.6 Discussion

Due to the wide applicability of the smartphone system clock, we quantified the accuracy of the system clock, which highlighted the troublesome degree of errors in Android. We use an example of human activity recognition to show that timing errors across smartphones profoundly impact deep learning-based multimodal fusion classifier. Several strategies are discussed to improve the system clock errors. Our system clock characterization empowers future developers with the knowledge of expected timing errors in smartphones, which can help them to develop application-specific mitigation strategies. To improve the robustness of deep learning-based multimodal applications to timing errors, we introduce a data augmentation approach to handle the inevitability of timing errors.

CHAPTER 3

Synchronizing Time across Smartphones

3.1 Shared Notion of Time across Smartphones

A shared notion of time is an essential requirement for applications intending to fuse data and/or coordinate concerted actions across multiple devices [DR17]. Applications on smartphones including beamforming and localization require precise time synchronization on the order of microseconds to maintain correctness [LRS15b]. Popular apps such as AmpMe, which converts a collection of smartphones into a distributed music player, requires sync error of less than 10 milliseconds to prevent ear fatigue [Kha18]. Other IoT systems where smartphones serve an integral role, including crowdsensing, robotics, and cross-modal deep learning at the edge, demand best-effort synchronization on the order of milliseconds [DR17, XSB18].

3.1.1 Challenges and Tradeoffs

Achieving precise time synchronization across a suite of smartphones pose unique challenges. First, hardware support for protocols such as PTP [EL02] is unavailable. Second, these devices operate exclusively via wireless networking, which has been shown to exacerbate the variability of achievable accuracy [MDB16]. Third, given the overall difficulty in achieving precise time, evaluating the accuracy of a particular synchronization mechanism is in-and-of-itself a troublesome task; that is, a ground truth baseline is impossible to attain. Finally, and most importantly, the smartphone platforms (e.g., Android, iOS) restrict timing stack control solely to the underlying operating system, with poor clock maintenance [MDB16].

Given the rich suite of capabilities on smartphones, there is a myriad of alternative techniques that can be employed to achieve clock synchronization. For local area relative synchronization, Lazik et al. [LRS15b] introduced an audio-based clock synchronization method that relies on external beacons to achieve sub-millisecond precision. For wide-area synchronization, Yan et al. [YLT17] leverage skin electric potentials measured from external wearables, achieving accuracy on the order of milliseconds. For general-purpose timing, wireless NTP client libraries are available that can typically provide accuracy in the order of tens of milliseconds [Mil12].

Each of the available mechanisms for smartphone synchronization possesses unique tradeoffs that result in *one size does not fit all* policy. Selecting the appropriate technique then becomes a function of the requirements and characteristics of a particular application over distributed devices. Some synchronization requires external infrastructure or tightly integrated peripheral sensors [LRS15a]. Others are restrictive to local area networks [LRS15b]. In considering these tradeoffs, the questions we pose and strive to answer in this chapter are: (1) *which peripherals can be used to synchronize distributed smartphones*? (2) *Which synchronization technique best suits a particular peripheral*? Also, (3) *how do different peripheral synchronizations compare with each other?*

To answer these questions, this work exploits various peripherals on smartphones to provide a comprehensive implementation and evaluation of precise clock synchronization solutions for modern distributed applications spanning smartphones. We begin by describing various smartphone peripherals over which clock synchronization can be achieved. Provided implementations are based on either receiver-receiver [EGE02] or sender-receiver [GKS03] synchronization protocols. Due to fundamental hardware limitations of specific smartphone peripherals and their reliance on external infrastructure for synchronization, we focus on audio subsystem, Bluetooth Low Energy, and Wi-Fi in particular. Given the difficulty in evaluating time sync approaches on smartphones, we show how to probe the accuracy of a particular technique effectively.

We present the first work that provides a detailed comparison of cross-peripheral synchronization on a smartphone. Traditionally, reliance on specialized hardware for low-level timestamping

and precise synchronization of peripheral clocks has restricted the use of one peripheral sync for the other. In this work, we argue that cross-peripheral sync is viable without low level timestamping or specialized hardware support by using the shared monotonic clock. Given symmetric stack delays of a particular peripheral across smartphones, we provide cross-peripheral sync that is good for timestamping and generating synchronous events across different peripherals. We provide an offering of open source library implementations for specific techniques to reduce the developer overhead [San19].

3.2 Background and Related Work

Peripherals are key to synchronizing distributed devices with each other. For example, audio peripherals form synchronized acoustic sensing arrays for range finding and target localization [GE01]. The availability of Bluetooth Low Energy (BLE) in consumer electronics and its low power architecture has made it a viable choice for sensor fusion and time sync [SMG16]. Packet-based synchronization protocols over IEEE 1588 [EL02], 802.11 [MGT11], 802.14.5 [AS17] are used extensively for high precision depending upon the quality of transmission and reception timestamps. Note that most of these peripherals expose IO capabilities on embedded platforms for precise synchronization. Although smartphones possess a rich suite of peripherals, they do not provide IO capabilities that can be exploited for high quality timestamping and synchronization.

Synchronizing smartphone clocks in the context of localization has been discussed in recent literature. Lazik et al. localize a smartphone by synchronizing to a network of beacons producing ultrasonic chirps [LRS15b]. Beacons are also fed to a smartphone audio peripheral to synchronize a phone with a speaker [LRS15a]. The audio peripheral is widely used in smartphone-based localization, either with infrastructure beacons or speakers. In this chapter, however, we seek to go beyond acoustic capabilities and compare the synchronization capabilities of various peripherals on a smartphone.

There are many applications in distributed sensing that rely on time synchronized smartphones

without any infrastructure support. The only available system clock – maintained and controlled by the operating system – is disciplined via NTP and NITZ protocols[1] [MDB16, SNA20]. Timestamping variability and poor disciplining mechanisms of system clock exacerbate timing precision in smartphones [MDB16, SNA20]. In contrast, this chapter provides various alternatives to smartphone synchronization with a comprehensive guideline to which synchronization technique best meets the needs of a particular distributed application.

3.3 Smartphone Time Synchronization

Given the diverse set of hardware attachments and sensing modalities on modern smartphones, a plethora of techniques can achieve clock synchronization. However, due to the differences in the underlying hardware characteristics (e.g. sampling rate, variance, latency), some approaches are more fundamentally limited in achievable performance.

3.3.1 Time Synchronization Approaches

There are two general approaches to synchronizing a collection of devices, commonly referred to as receiver-to-receiver (R2R) and sender-to-receiver (S2R). R2R relies on a common event observed by all devices, thus providing a unified reference point for each device to independently establish and maintain its own clock relative to the reference signal [EGE02]. In recent literature, RF [EGE02], acoustic [GE01], and power line signals [RGR09] provide reference signals for R2R. S2R relies on a server-client model, with a designated device serving as a reference clock by which other devices attempt to relatively synchronize their clocks [GKS03] [Mil12]. Both R2R and S2R techniques have their pros and cons; while R2R eliminates sender side non-determinism, S2R compensates for propagation delays in the network.

Across distributed smartphone devices, R2R and S2R synchronization can be achieved through

[1] To the best of our knowledge, GPS is not used to adjust the Android system clock.

a variety of alternative peripherals. An R2R broadcast reference signal may be either opportunistically observed or intentionally generated, and can potentially span any of the available sensing modalities available (e.g. audio, ambient light, proximity, IMU, camera, bluetooth, network). Similarly, S2R synchronization may occur over any of the available wireless networking mediums (e.g. Wi-Fi, Bluetooth, cellular). We select and implement a subset of possible sync techniques across different peripherals; open-source implementations are available at [San19]. Our goal is to provide a set of plausible, real-world solutions that can be reasonably integrated without requiring a fundamental re-working of the overall system.

3.3.1.1 Receiver-to-Receiver

Many of the sensing modalities on smartphones that can be leveraged for R2R synchronization possess fundamental hardware, infrastructure, or compute limitations restricting its practicality in deployment. The IMU, ambient light, and proximity sensors operate at frequencies of 100Hz or less, resulting in a theoretical clock sync limitation of 10ms or more [Goo19]. The camera can offer a slightly higher sampling rate (up to 240Hz), but requires significant compute overhead to inference and extract reference signals, making it generally impractical. Furthermore, generating a reference signal in many of these domains require external infrastructure.

We select the audio subsystem for our R2R implementation due to its high sampling rate. A notable benefit of the audio subsystem is its support for system timestamping, which results in comparatively less jitter in timestamp delays. As such, audio presents the best potential for achieving good performance in precise clock synchronization. Finally, generating a broadcast signal in the audio domain is relatively straightforward; smartphones themselves have speakers capable of generating these reference events.

To achieve audio clock synchronization, all smartphones listen for a broadcasted audio event, which may be opportunistically observed or intentionally generated. Each smartphone independently timestamps the event upon observation using the reported system timestamping. One smart-

phone is designated as the reference clock and shares its observed audio event timestamp with all other smartphones. Each device then computes its relative clock offset with respect to the reference device. Previous work by Lazik et al. [LRS15a] noted high audio sampling variability in Android, which potentially leads to degraded clock sync performance. In our provided implementation, we account for this variability to enable precise audio clock sync across Android smartphones.

3.3.1.2 Sender-to-Receiver

S2R synchronization can be accomplished through various peripherals, including cellular, wifi, and bluetooth. Cellular in particular often incurs large latencies and asymmetry; as such we collapse its usage into a Wi-Fi based implementation. In contrast, bluetooth offers a close proximity carrier signal with efficient energy consumption, making it a valuable alternative to standard networking. Given that many embedded sensing platforms support Bluetooth Low Energy (BLE), oftentimes without Wi-Fi (e.g. wearables), a bluetooth sync solution improves compatibility with a wider application domain. Devices periodically ping a reference device, either an external server or one of the smartphones, and compute a relative clock offset. Due to the lack of smartphone support for system timestamping of bluetooth or Wi-Fi events, application-level timestamping must be used.

In summary, some peripherals serve as sub optimal mediums for precise clock synchronization. Therefore, we select and implement three synchronization techniques across the audio, bluetooth, and Wi-Fi peripherals.

3.3.2 Time Synchronization Comparison

When two smartphones are synchronized using a particular peripheral, the clock difference fails to capture the true offset, as it is affected by the peripheral timestamping delays. Figure 3.1 illustrates the timestamp delays in audio (t_{audio}), Wi-Fi (t_{wifi}) and BLE (t_{ble}) peripherals in Android to capture the occurrence of an event observed by the hardware. Audio, Wi-Fi and BLE peripherals have access to timestamping capabilities at different stack layers. Kernel level timestamping is available

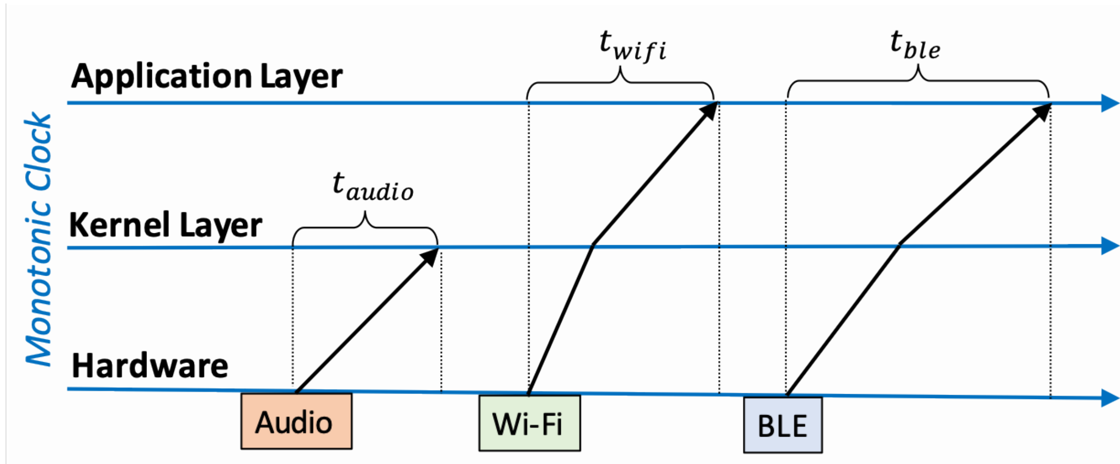


Figure 3.1: Timestamping events for audio, Wi-Fi and BLE peripherals in Android. Timestamp delays are not drawn to scale.

for audio, whereas for Wi-Fi and BLE peripherals timestamping is only available in the application layer. If smartphone A and B are synchronized using a particular peripheral p, the computed $offset$ between two clocks in timestamping p's events is, $offset_p = offset_{true} + (t_p^A - t_p^B)$, where $p \in \{audio,\ ble,\ wifi, ...\}$, $offset_{true}$ is the difference between smartphones monotonic clocks in the absence of the timestamp delays, and t_p^A and t_p^B correspond to the timestamp delays for a peripheral p on smartphone A and B, respectively.

After synchronizing smartphones clocks using one peripheral, the same peripheral can be used to listen and generate synchronous events. More precisely, if A and B are synchronized using audio peripheral, the expected error in capturing synchronous audio events on both smartphones is the variability of the term $(t_{audio}^A - t_{audio}^B)$ i.e, the sync error corresponds to timestamp delay jitter of the phones audio pipeline after accounting for clock drift. For minimum variability, it is preferred that timestamp delays are minimized and taken as close to the hardware as possible. Similarly, the accuracy of the Wi-Fi peripheral to observe and generate synchronous network events is the variability of the term $(t_{wifi}^A - t_{wifi}^B)$. The same applies to the BLE peripheral.

Under certain conditions, one synchronized peripheral can be used to timestamp and generate distributed synchronous events across other peripherals. For example, if smartphone A and B are

synchronized using audio peripheral, this synchronization can be used to generate synchronous network events on the Wi-Fi peripheral. This is possible only if timestamp delays for a particular clock are symmetric across different devices for the same peripherals. We capture this condition in the following equations:

$$offset_{p1} = offset_{true} + (t^A_{p1} - t^B_{p1}) \tag{3.1}$$

$$offset_{p2} = offset_{true} + (t^A_{p2} - t^B_{p2}) \tag{3.2}$$

Subtracting (1) and (2),

$$offset_{p1} - offset_{p2} = (t^A_{p1} - t^B_{p1}) - (t^A_{p2} - t^B_{p2}) \tag{3.3}$$

In these equations, note that both peripherals $p1$ and $p2$ timestamp events relative to the same monotonic clock as shown in Figure 3.1. If $p1$ is audio and $p2$ is Wi-Fi, this error comparison captures the difference between audio and Wi-Fi offset calculations, which directly corresponds to the mismatch in timestamp delays across these peripherals. According to (3.3), if the timestamping delays of the audio peripheral in devices A and B are the same, and the timestamping delays of the Wi-Fi peripheral in A and B are the same, then their difference in clock offsets are the same. This implies that synchronization across one peripheral can be used for other peripheral events if there is no mismatch in their timestamping delays. In summary, the above equations capture the heterogeneity in timestamp delays for different peripherals stacks. An important observation is that timestamping delays are symmetric when the devices possess the same hardware, kernel, and application load.

3.4 Evaluation

Given a particular sync approach and implementation, a challenge arises in how to determine its accuracy. As a ground truth baseline is impossible to attain on smartphones, alternative techniques must be used to approximate overall accuracy and precision of a given time synchronization solution.

First and foremost, a sync solution performed via a particular peripheral is best suited to measure events that occur across that same peripheral. For example, audio subsystem sync is able to most precisely record audio events. Previous work by [LRS15b] introduced a methodology of observing variability in sync offsets (Equation 3.1) after repeated audio sync attempts as a means of quantifying the accuracy of the audio sync solution. More generally, a broadcasted reference event (e.g. audio chirp) can be repeatedly observed and timestamped across multiple smartphones to evaluate sync precision. Hence, variability evaluations capture sync precision for the same peripheral.

An alternative evaluation methodology is to compare two sync solutions that are performed in parallel. This cross-peripheral comparison better quantifies the ability of a particular sync technique using one peripheral to accurately capture events over other peripherals. For example, one might evaluate the accuracy of a Wi-Fi sync implementation with respect to an audio sync implementation. The observed results are fundamentally limited by the combined precision of the two sync solutions. For these results to be meaningful, the baseline must be more accurate than the evaluated approach. As such, this methodology offers an upper bound on precision, indicating how well a particular sync solution maps to other forms of events observable by the smartphone device.

3.4.1 Experimental Setup

Two Pixel 3 and two Nexus 5X smartphones comprised the set of devices used for the evaluation. One Pixel 3 was designated as the reference device onto which other smartphones attempt to relatively synchronize their clock. Note that all peripherals are synchronizing the monotonic clock maintained by the operating system. We disable other system level sync attempts to this monotonic clock. The experimental setup consisted of repeated sync attempts for each of the three peripherals. After every successful sync, the computed relative clock offset is recorded. For Wi-Fi and BLE sync, our S2R implementations are based on NTP. Evaluations based on NTP over Wi-Fi best captures performance both at local and global scale for smartphones. Other S2R protocols

such as PTP [EL02] and TPSN [GKS03] require specialized hardware support and as such are not optimized for smartphone sync.

3.4.2 Variability Evaluation

For each of the synchronization techniques introduced, we begin with a reflective evaluation; that is, the precision of each of audio, BLE, and Wi-Fi sync in capturing audio, bluetooth, and wifi events, respectively. These results indicate the ability of a particular sync solution when solely capturing events generated by the same peripheral used to perform synchronization.

3.4.2.1 Drift Correction

Figure 3.2 indicates the normalized relative offset for three of the smartphones over the course of an hour, specifically for audio sync. The fourth phone, serving as the reference, is excluded as it would by definition maintain an offset of exactly zero. Due to the differences in clock drift between each smartphone and the reference, each smartphone experienced a different rate of overall change in the total clock offset. A least-squares regression line is plotted for each phone, with the slope indicating the rate of relative drift between each smartphone clock with respect to the reference. The error of each sync attempt can then be estimated as the difference between recorded offsets and this regression line.

3.4.2.2 Results

Figure 3.3 presents three histograms detailing the estimated sync error for the audio (3.3a), bluetooth (3.3b), and Wi-Fi (3.3c) implementations in capturing their own respective peripherals. Audio achieves the best accuracy, with 86% of sync attempts falling within $200\mu s$ of the estimated true offset, and a total spread of just over one millisecond. Bluetooth sync had an order of magnitude more variability, with 85% of sync attempts falling within $3000\mu s$, and a spread of approximately $20ms$. Wi-Fi sync variability fell in between the two, with 95% of sync attempts falling within

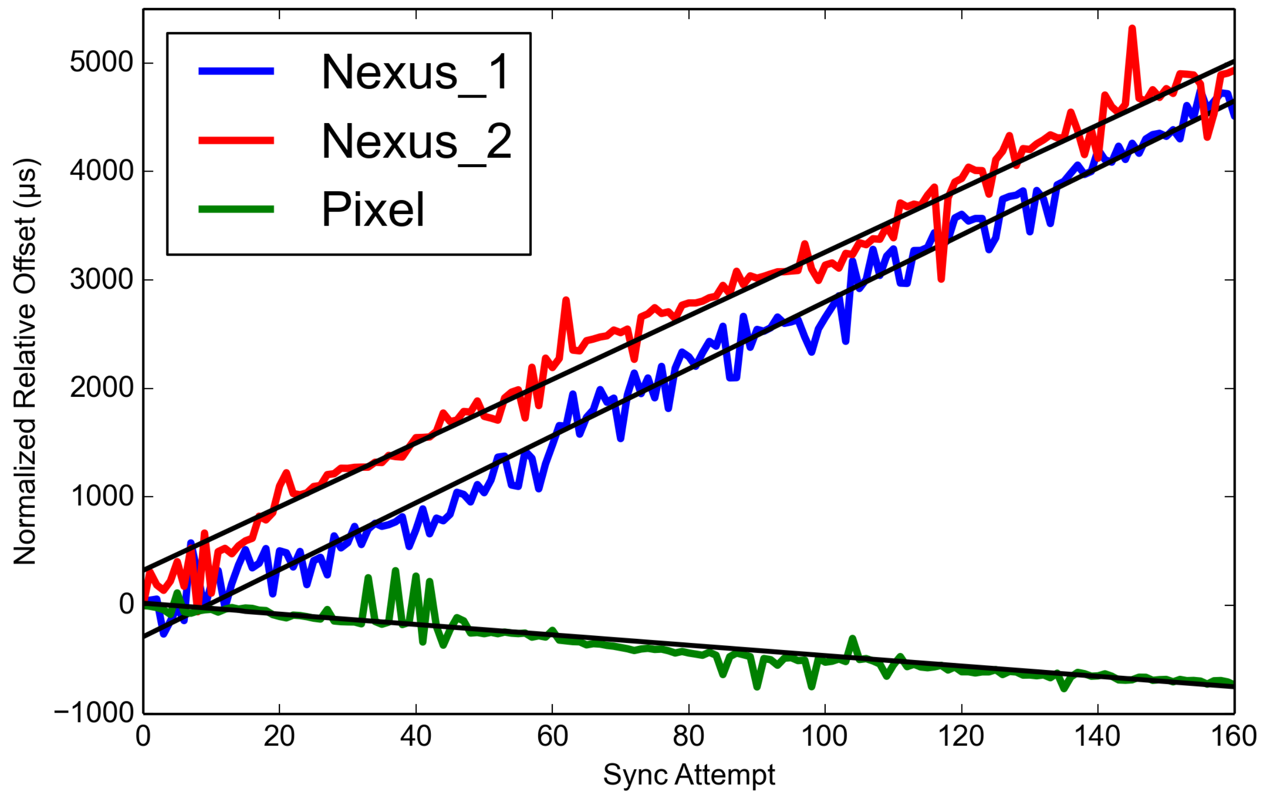
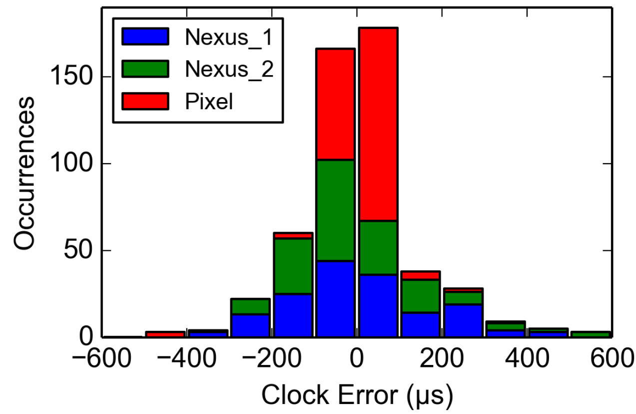
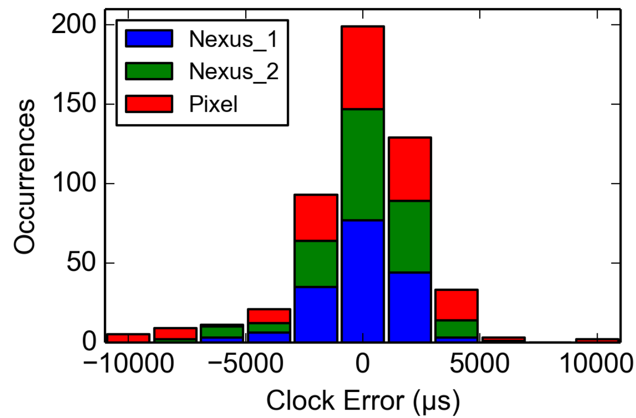


Figure 3.2: Relative clock offset for audio-based sync over time, with respect to a fourth phone (Pixel) serving as a reference clock. Results are normalized to the initial computed offset for each device. Due to clock drift, offsets change as a function of the relative drift between the synchronizing device and the reference device. A regression line for each device indicates the overall relative drift trend.

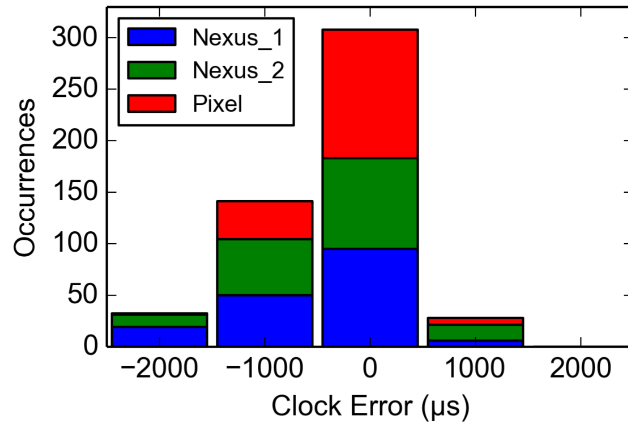
$1000\mu s$ and a spread of $3ms$. Despite the fact that wireless NTP has been previously shown to present errors on the order of tens of milliseconds [MDB16], our controlled experiment with a collection of local smartphones synchronizing with the same server (17.253.26.253) over campus Wi-Fi was able to achieve a precision of only three milliseconds. Wi-Fi NTP round trip time was commonly observed between 4 to 8ms, whereas BLE NTP round trip time was typically between 35 and 50ms.



(a) Audio sync variability



(b) BLE sync variability



(c) Wi-Fi sync variability

Figure 3.3: Sync offset variability with respect to the fourth (Pixel) reference device for (a) audio, (b) BLE, and (c) Wi-Fi implementations. 86% of audio sync attempts fall within $\pm 200\mu s$. 85% of BLE sync attempts fall within $\pm 3000\mu s$. 95% of Wi-Fi sync attempts fall within $\pm 1000\mu s$.

Table 3.1: Clock synchronization accuracy across peripherals for three smartphones with respect to a reference Pixel phone. Results are 95% confidence intervals.

	Audio vs Wi-Fi	Audio vs BLE	Wi-Fi vs BLE
Pixel 3	1.51 ± 1.50 ms	2.20 ± 5.06 ms	0.51 ± 5.06 ms
Nexus $5X_1$	13.12 ± 1.50 ms	13.85 ± 4.00 ms	2.15 ± 5.46 ms
Nexus $5X_2$	12.39 ± 1.50 ms	12.93 ± 3.74 ms	-0.86 ± 4.5 ms

3.4.3 Cross-Peripheral Evaluation

A variability evaluation characterizes the precision of a particular sync implementation in capturing events generated across its own peripheral. However, in practical deployment synchronized clocks are often used to timestamp events across a wide suite of peripherals. Given the diverse set of capabilities on smartphones (e.g. ambient light, audio, bluetooth, camera, IMU, proximity, network, proximity), exclusively reporting a sync method's precision with respect to its own peripheral is insufficient. A complete evaluation captures the differences across peripherals; this can be accomplished by comparing two parallel synchronization methods using alternate peripherals. Table 3.1 presents the results from comparing each of the three clock sync techniques with respect to the other two synchronized clocks using Equation 3.3. One of the Pixel phones was reserved as the reference device to which other phones are attempting to synchronize. Each sync technique was performed and compared at least 100 times; 95% confidence intervals are reported. Confidence interval sizes are a function of the combined variability of the two peripherals; as such, the comparisons using BLE present the highest error bounds. One interesting observation is the significant bias induced when comparing clock sync solutions between varying hardware. While the Audio-WiFi and Audio-BLE overall comparison are near zero when comparing two Pixel devices, a bias of ˜13ms are induced when comparing Nexus to Pixel devices. This is a direct result of the varying audio, Wi-Fi, and BLE stack latencies between Pixel and Nexus smartphones. These results indicate that an audio sync solution in particular fails to extrapolate to other peripherals; in

contrast, WiFi-BLE comparisons maintain reasonably comparable results independent of the hardware devices evaluated. On the other hand, when comparing peripherals across similar hardware, audio sync can be used to synchronously generate Wi-Fi and BLE events; this is shown by the minimal differences in stack latencies across both Pixel devices.

3.5 Discussion

The choice in selecting a sync solution for a smartphone incurs varying tradeoffs. No technique is objectively optimal; the appropriate selection is dependent on a number of application factors, most notably the range of participating devices and the active peripherals required.

3.5.1 Tradeoffs

Bluetooth presents the most straightforward tradeoff; it has high variability and limited range, and as such should be reserved for usage where the gain in energy efficiency is worth the sacrifice in precision. Averaging offsets from repeated bluetooth sync attempts can help alleviate the impact of this variability. The exception to this tradeoff is when interfacing with devices that only support bluetooth; when capturing bluetooth events, bluetooth sync will most accurately account for differing BLE stack latencies across varying hardware.

The audio subsystem tradeoffs are more complex. While system timestamping provides audio sync with the lowest variability, processing broadcasted audio events demand low relative ambient noise in the generated audio frequency band (i.e. sufficient signal-to-noise ratio). This restricts the environments where audio sync can be deployed, and places a dynamic range limitation. Finally, despite the fact that audio has the lowest variability, Table 3.1 indicates that clocks synchronized with audio events are limited in their accuracy of cross-peripheral timestamping; this is likely due to notable differences in audio subsystem latencies across varying hardware platforms.

Wi-Fi synchronization is the most general-purpose and flexible solution. While other tech-

niques rely on local-area relative synchronization, NTP over Wi-Fi can support wide-area deployments. However, as network characteristics differ, so does the achievable clock sync precision. Nevertheless, NTP over a local network can achieve precision on the order of a few milliseconds with minimal effort and relatively low-overhead (i.e. does not require microphone, speaker, or bluetooth to be active).

3.5.2 Recommended Sync Solution

Given the order of magnitude reduction in precision when comparing same peripheral variability to cross-peripheral clock comparisons, an overwhelmingly clear message arises: in order to precisely timestamp events arriving across a particular peripheral, that same peripheral should be used to perform clock synchronization. For applications combining information across multiple peripherals and multiple smartphones, one phone should be selected as a reference device onto which all other smartphones synchronize each peripheral independently. An exception to this rule is when peripherals incurs similar latencies across different devices; in this case, one peripheral synchronization can be used to synchronously sense and/or generate events across the other peripheral.

CHAPTER 4

Variable End-to-end Delays in Deep Reinforcement Learning

Deep Reinforcement Learning (RL) has shown promising results for a range of robotics applications, such as navigation [BMG19], manipulation [TFR17], and locomotion [HLD19]. Deep-RL policies are often trained with simulations due to cost, time to train, and safety concerns that arise when training on real robots [TFR17]. Simulations are imperfect and difficult to calibrate. The resulting modeling discrepancies cause a *reality gap*, which makes the transfer of RL policies from simulation to the real-world (Sim2Real) a challenge [KMD10]. Prior works have proposed domain randomization and adaptation techniques to address the reality gap in dynamics [TFR17, ABC20] and image observations [SL16]. We study the reality gap introduced due to uncertainty in time between state transitions and the resultant impact on dynamics in agile robotic tasks such as locomotion and navigation.

4.1 State Transition Delay in Deep-RL

RL agents make sequential decisions in a Markov Decision Process (MDP) in *discrete time steps*, where the input to the agent is the current state s_t of the environment, where t is the current time step, and output is the action a_t. The environment transitions to the next state s_{t+1} once the action is executed, and in turn used as the input for the next action a_{t+1}. If the states s_t and s_{t+1} were captured at world clock time τ and τ' respectively, the timing delay between state transitions is defined as $\Delta\tau = \tau - \tau'$. Note that t is a discrete time step in simulation while $\Delta\tau$ represents the actual passage of time on a robot. A common trend is to assume $\Delta\tau$ is fixed for Sim2Real transfer [BMG19, MCH19]. However, $\Delta\tau$ varies in real robots due to variations in RL policy

execution time, sensor sampling interval and communication delays. In deep-RL, policy execution time of neural networks dominates $\Delta\tau$ and shows significant variation due to various factors – uncertainties due to locally shared compute resources, asymmetric communication latencies with use of shared cloud resources [CSH19], and delay variations due to processor throttling for thermal and energy constraints [WWD94].

As the real robot operates in continuous world clock time, the state transitions observed by the agent will change with variations in $\Delta\tau$. If these variations are not captured by the simulator during training, it leads to poor Sim2Real transfer [MKK18, XBC18]. Prior works randomized the state transition delays $\Delta\tau$ during training for a successful Sim2Real transfer [ABC20]. We demonstrate that the policy performance degrades with variation in $\Delta\tau$ even with domain randomization. Another approach is to artificially extrapolate varying delays to the worst case $\Delta\tau$. For example, Molchanov et al. [MCH19] use a fixed $\Delta\tau$ of 2ms for controlling a quadrotor while the neural network inference latency was only 0.8ms. With this approach, one is forced to pick a conservative $\Delta\tau$ that accounts for the *worst case delays* in the system or pick a small neural network to keep inference latency to a minimum. Large $\Delta\tau$ limits the applicability of deep-RL in agile tasks where fast response times are required [WNL09], and small neural networks limit scalability for complex tasks with large state-action space.

4.1.1 Time-in-State RL

We introduce Time-in-State RL (TSRL), a deep-RL approach that extends the observed state of the system by explicitly including time-delays, i.e., incorporating $\Delta\tau$ introduced by the sensor sampling interval and execution latency at training time. Even though the inferencing latency and sampling interval can vary for various reasons, they can be accurately measured by deep-RL agents at runtime. We test the following hypothesis: if the agent observes the factors that impact the $\Delta\tau$, and hence the state transitions, it helps the agent to learn a better policy compared to a policy that partially observes the impact of changing $\Delta\tau$ using domain randomization. We evaluate our approach on simulation-to-simulation (Sim2Sim) transfer on PyBullet HalfCheetah, PyBullet

Ant [CB19] and DeepRacer [BMG19]. We evaluate our approach on Sim2Real transfer using a $\frac{1}{18}^{th}$ scale car. We compare the TSRL policies with the policies trained using domain randomization (DR) of timing characteristics. Our results demonstrate that the TSRL policies are robust to the varying state transition delays and, as a result, transfer better across simulations and to real-world environments than the DR policies.

4.2 Background

4.2.1 Temporal Variability in Deep-RL

RL agents learn to make sequential decisions in the environment to maximize the expected cumulative discounted reward. At each discrete time step t of an MDP, the agent takes action a_t based on state s_t, and the environment returns with scalar reward r_t and next state s_{t+1}. We consider episodic MDPs, where the environment is initialized with state s_0 and the interaction continues until the environment reaches the terminal state s_T. $p(s_{t+1}|s_t, a_t)$ is the probability of transition to state s_{t+1} given current state s_t and action a_t, and $\pi(a|s)$ represents the probability of taking action a given state s by policy π. Any changes to the real state transition time $\Delta\tau$ – as opposed to fixed discrete time steps in t – directly impacts the state transition probability $p(s_{t+1}|s_t, a_t)$.

The objective of the agent is to learn a policy π that maximizes:

$$J = \mathop{\mathbb{E}}_{\substack{\pi(a|s) \\ p(s_{t+1}|s_t, a_t)}} \left[\sum_{t=0}^{T} \gamma^{t-1} r_t \right] \tag{4.1}$$

where $\gamma \in [0,1]$ discounts future rewards, T is the episode length, $p(s_{t+1}|s_t, a_t)$ is the probability of transition to state s_{t+1} given current state s_t and action a_t, and $\pi(a|s)$ represents the probability of taking action a given state s by policy π. Any changes to the real state transition time $\Delta\tau$ – as opposed to fixed discrete time steps in t – directly impacts the state transition probability $p(s_{t+1}|s_t, a_t)$.

We focus on model free RL algorithms, where the state transition probabilities $p(s_{t+1}|s_t, a_t)$ are unknown to the agent and are inferred indirectly through environment interactions. One of the simplest deep-RL algorithms is REINFORCE, where the policy is represented by a neural network with parameters θ. The policy is learned using gradient ascent on the objective function.

$$\nabla_\theta J(\theta) = \sum_{n=0}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t=0}^{T} \gamma^{t-1} R(s_t, a_t) \tag{4.2}$$

where $R(s_t, a_t)$ is the reward function and data from N episodes is used for the update. REINFORCE has high variance as the gradient update depends on the total discounted reward collected during each episode. To reduce variance, the advantage function $A(s_t, a_t)$ is used for the gradient update [Bai93, WSH15]:

$$\nabla_\theta J(\theta) = \sum_{n=0}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t) \tag{4.3}$$

$$A(s_t, a_t) = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t) \tag{4.4}$$

where $V_\phi(s_t)$ estimates the cumulative discounted reward from state s_t using a separate value network with parameters ϕ. Intuitively, advantage estimates the relative benefit of taking action a_t compared to other possible actions in state s_t. The value network is trained with a mean squared error loss function:

$$L_\phi = \frac{1}{2} \left\| \sum_t V_\phi(s_t) - \left(r_t + \gamma V_\phi(s_{t+1})\right) \right\|^2 \tag{4.5}$$

Due to variable state transition delay $\Delta\tau$, the agent observes stochasticity in state transitions $p(s_{t+1}|s_t, a_t)$ for the same state and action. Ignoring the variations in $\Delta\tau$ during training results in a distribution mismatch from simulations to the real world, leading to poor transfer. On the other hand, if we introduce domain randomization in time by considering variable $\Delta\tau$ during training, the additional state transition stochasticity introduces noise in the value function estimates $V_\phi(s)$ estimates in Equation 4.5 and makes it difficult to converge to a good policy.

To address variations in the state transition delay, we propose augmenting the agent state with execution time $\Delta\tau_\eta$ and sampling interval $\Delta\tau_\sigma$ measurements: $\tilde{s} = [s, \Delta\tau_\eta, \Delta\tau_\sigma]$ where $\tilde{s}$ represents the augmented state. This simple trick enables the agent to distinguish between state transitions

introduced by variations in delays. We train the agent with the augmented state and introduce delay variations in the simulator. As the delays are explicitly represented in state, it becomes much easier to estimate the value function $V_\phi(s)$ using Equation 4.5. Since the delay measurements are directly fed as input to both policy and value networks, the agent learns to generalize beyond the exact numbers seen during training.

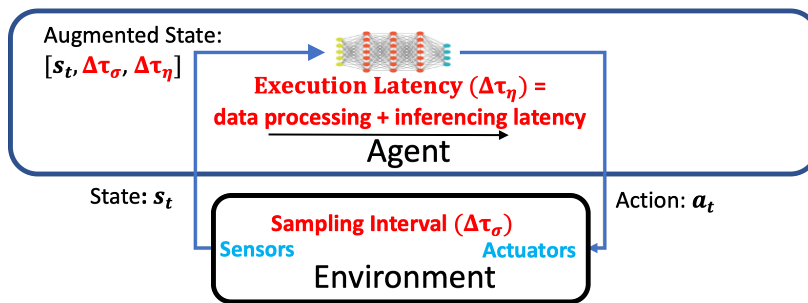


Figure 4.1: Delays for a typical sensing to actuation pipeline. TSRL augments the observed state with sampling interval and inferencing latency.

4.2.2 Variability in Execution Latency and Sampling Interval

The delays in a typical sensing to actuation pipeline for a deep-RL agent are shown in Figure 4.1. A typical state transition begins with sensing the current state s_t of the environment, executing the agent action a_t on the environment, and again sensing the updated state s_{t+1}. Each of these steps can have variability in the real world as determined by the sampling interval of sensors $\Delta\tau_\sigma$, the execution latency $\Delta\tau_\eta$, and communication delays $\Delta\tau_m$. When multiple sensors are present, $\Delta\tau_\sigma$ is the maximum of individual sensor sampling intervals. Similar arguments are extended to $\Delta\tau_\eta$. We assume communication delays $\Delta\tau_m$ are small, and subsume them into execution latency $\Delta\tau_\eta$.

Execution Latency: Various factors can affect execution latency $\Delta\tau_\eta$ such as power management, computational resources, and complex operating system (OS) environments. Dynamic frequency scaling [WWD94] is a commonly used technique to manage power dissipation [TSS17]. Frequency over-clocking/under-clocking changes computation speed and lead to variable latencies. The OS

scheduling processes result in scheduling noise that varies with system load and affects process latencies.

We analyzed the inference latencies of commonly used neural network architectures in deep-RL policies on several hardware platforms. The runtime latency depends on the complexity of neural network, hardware device, and multi-tenancy. On the GAP8 [FRC18] microcontroller, the execution latency of a simple neural network increases from ~8ms to ~60ms when the number of CNN layers increases from 2 to 4. For deep learning accelerators like the Intel Neural Compute Stick 2 [Int], the execution latency of a 2 layer CNN network is increased from ~3ms to ~20ms in presence of multiple inference tasks. We characterized the execution latency of the default deep-RL policy in DeepRacer [BMG19], a $1/18^{th}$ scale autonomous car that comes with an Intel Atom processor and an integrated GPU. The execution latency varies from 15-20 ms on the GPU and goes up to 34 ms on the CPU. We include additional analysis on execution latency in Section 9.1.

Sampling Interval: Stisen et al. [SBB15] demonstrate that sampling interval of accelerometers can vary widely in smartphones depending on both software and hardware characteristics. We characterized the variation of the frame rate in the DeepRacer front facing camera. The variation in sampling interval was 20-45 ms in the 30Hz frame rate setting and 62-71ms in the 15Hz setting respectively.

4.2.3 Impact of Temporal Variability on Deep-RL Policy

The impact of state transition time variations on the RL policy depends on the relative value of sampling interval $\Delta\tau_\sigma$ to execution latency $\Delta\tau_\eta$.

(a) $\Delta\tau_\sigma \ll \Delta\tau_\eta$: When the sampling interval $\Delta\tau_\sigma$ is very small, the variations in execution latency $\Delta\tau_\eta$ dominate the impact on the policy. As $\Delta\tau_\eta$ varies, the *observed evolution of the environment state* in world clock time τ also varies correspondingly. Hence, the agent will observe stochasticity in state transitions $p(s_{t+1}|s_t, a_t)$ for the same state and action. If we ignore the variation in $\Delta\tau_\eta$ during training, there will be distribution mismatch from simulations to the real world, leading to poor

transfer. Mahmood et al. [MKK18] and Xie et al. [XBC18] demonstrate that policies can break down with small changes ($<$100 ms) in latency for manipulation and locomotion respectively. On the other hand, if we introduce $\Delta\tau_\eta$ variations during training [TFR17, ABC20], the additional state transition stochasticity introduces noise in the value function $V_\phi(s)$ estimates in Equation 4.5 and makes it difficult to converge to a good policy. We demonstrate this in both simulation and a real robot in Section 4.5.

(b) $\Delta\tau_\sigma \gg \Delta\tau_\eta$: In this case, variation in the sampling interval $\Delta\tau_\sigma$ dominates the impact on the RL policy. The agent needs to observe the effect of its action on the environment. When sampling interval is large or if changes in state are minor with a single action, it is common practice to repeat the agent action a fixed number of times [MKS15, LHP15]. With variations in $\Delta\tau_\sigma$, the number of repeated actions need to be varied and the impact on the RL policy follows the same argument as above.

(c) $\Delta\tau_\sigma \sim \Delta\tau_\eta$: In this case, we want the agent to act for every sensed state [Hal13]. When $\Delta\tau_\sigma$ and $\Delta\tau_\eta$ vary, there is a *phase shift* in each state transition depending on when the state gets sampled and when the action is executed. These phase shifts introduce noise in the state transition probabilities $p(s_{t+1}|s_t, a_t)$, and impact the performance of the RL policy. While phase shifts do occur in the other two cases, the impact on the policy is dominated by overall variation in either $\Delta\tau_\sigma$ or $\Delta\tau_\eta$.

4.3 Related Work

4.3.1 Control System Approaches

Multiple researchers have investigated the design of classical controllers in the presence of delays. The presented experiments in this chapter could be reformulated as classical control problems. Control systems approaches typically model finite-dimensional systems that may require linearization. Traditionally, the delay is modeled and incorporated in the design of optimal controllers to

compensate for it. The goal in these contexts is to develop robust controllers by approximating the worst case time-delays [Beq03, LR90] and sampling variation [WRG14], or by compensating for delays using damping components. Wittenmark et al. [WBN98] models the delays showing complicated patterns in nested communication loops suggesting that it is important to consider delays in the design of the controller. Hespanha et al. [HNX07] and Lian et al. [LMT01] discuss the presence of variable delays in networked controlled systems. Luck et al. [LR90] propose the design of a worst-case delay controller by using buffers in the closed-loop system. It is shown that if the buffers are chosen larger than the expected worst-case delay, then the runtime delays can be made deterministic. The approach of worst-case delay is popular for uncertain delays with known upper bounds.

Nilsson et al. [Nil98] proposes the design of an event-driven controller, where the controller actions are applied as soon as possible based on the delays in the pipeline. Nilsson et al. [Nil98] compare an optimal event-driven controller with the worst-case delay controller proposed by Luck et al. [LR90]. The results show that an optimal event-driven controller can outperform the worst-case delay controller for linear control systems.

Our assumptions in designing Time-in-State (shown in Figure 4.1) are similar to the Nilsson et al. [Nil98]. We assume that the sensing is clock-driven according to a sampling interval. The sampling interval is not fixed and can vary at runtime, as discussed in Section 4.2.2. The action application from the neural network is event-driven, where we apply action as soon as possible according to the delays in the pipeline. As discussed, prior work [Nil98] has shown that optimal event-driven controllers can outperform the worst-case delay controller for linear control systems. The assumption by Nilsson et al. [Nil98] is that the past delays can be used to predict future expected delays. In Time-in-state, we also assume that the future expected delays could be predicted. We evaluate Time-in-state assuming a measurement error of 20% in Section 4.5.2 and also compare Time-in-state with worst-case delay (constant delay) controllers in Section 4.8. As discussed in Section 4.3.3, the latency of neural networks at runtime across a diverse suite of devices can be predicted within 10% measurement errors on an average.

Unlike a controller designed via analytical means, the DNN-based controller trained via RL is a black box. There are no known mechanisms to compensate for delays. The introduced strategy of Time-in-State shows a practical approach to modify the training of neural networks to enable variable delay adaptation based on the runtime expected delay measurements. Because Time-in-State is augmenting state space with measured time-delays, one may hypothesize that the adaptive policies are mimicking the classical control approaches in adapting the actions based on the expected delays.

4.3.2 Handling Delays in Reinforcement Learning

Handling of delay variations in a robot has been identified as crucial for successful Sim2Real transfer by many prior works [XBC18, MKK18, TFR17, ABC20]. Variable delays are studied in the literature using different delay models. One approach is to apply action as fast as possible and either train robust RL policies [MKK18, ABC20, XBC18] or learn system dynamics [WNL09, CXL20] to account for delay variations. We call this approach in RL as the *event-driven controller* as defined by Nilsson et al. [Nil98]. Another approach is to adopt the conventional control systems method [LR90] to convert the variable delay to constant delays by adding extra delay buffers [SBB10, RP19, KE03]. We call this approach in RL as the *worst-case delay controller* [LR90].

Event-driven controller: By default, in several robotic simulators, action is applied instantaneously, assuming no execution latency. For example, in HalfCheetah and Ant task in PyBullet simulator [CB16], the simulation is paused during inference of the neural network, and the most recent action is applied when advancing simulation. Thus no execution latency is assumed. Several researchers [MKK18, ABC20, XBC18] propose the design of robust RL policies using *domain randomization* during training to account for runtime delay variations. Mahmood et al. [MKK18] examine the design decisions for the deployment of deep-RL policies for manipulation. They present a computation model on a UR5 reacher robot running an RL controller where the action is sent to the actuator as soon as possible. They highlight the lack of guidelines for picking state

transition times or implementing asynchronous mechanisms to reduce the state transition delays. It is also empirically shown that for the UR5 reacher robot, there exists an optimal delay at which the performance is best and beyond which performance degrades significantly. Andrychowicz et al. [ABC20] studies dexterous hand robots where RL policy sends an action to the low-level controller as soon as possible after an average inference latency of 25ms. They show the presence of delay variations on real robots and use domain randomization by varying delays during training to train a robust policy. Peng et al. [PAZ18] also observe the variable delays for a fetch robotic arm. They use domain randomization to randomize the state transition delay each time step according to an exponential distribution. We compare Time-in-state with the domain randomization approach showing superior performance of Time-in-state policies.

Walsh et al. [WNL09] and Chen at al. [CXL20] studies action application in discrete steps. They propose learning dynamics of the system using model-based RL that can work in the presence of variable discrete delays. Walsh et al. [WNL09] compare their controller that acts in every discrete step with a constant delay controller (constant step delays), showing the superior performance of their design. A limitation of the method proposed by Walsh et al. [WNL09] is that they use model-based RL to study systems where state-space is simple/finite and has deterministic dynamics, and delay variations are in discrete steps only. Chen et al. [CXL20] shows that the constant step delay controller doesn't work when step delays are varied at runtime. In a way, this controller is a discrete event-based controller which can act in every discrete step and outperforms a worst-case controller that waits for an integer number of step delays. In contrast, in Time-in-state RL, we propose an event-driven controller that can adapt to the continuous changes in the runtime delays. Our proposed Time-in-state trains adaptive controllers without explicitly modeling the physics dynamics of the system and works for complex modalities. We show applications of Time-in-state to both low-dimensional and high-dimensional state spaces and its direct transfer to the real robotic car.

Worst-case delay controller: Worst-case delay or constant delay controller is an alternative approach to make the run-time delays deterministic. Several researchers [RP19, KE03, SBB10]

have proposed constant delay MDP as an extension to the vanilla MDP. These works assume that action is applied either at a fixed delay or at the end of a few time-steps. Ramstedt et al. [RP19], and Katsikopoulos et al. [KE03] convert MDP with constant delays to MDP without delays by modifying the state space and state transition structure. Ramstedt et al. [RP19] proposes a new actor-critic algorithm for a single-step delay and shows it speeds up the training as compared to the vanilla soft actor-critic method. Katsikopoulos et al. [KE03] present theoretical results on the equivalence of the modified MDP representing constant delays. We compare our proposed Time-in-state with worst-case delay (constant delay) controllers in Section 4.8.

4.3.3 Accuracy of Delay Measurements at Runtime

Our assumption in Time-in-State RL (TSRL) is that expected delays can be measured at runtime. A significant part of the execution time is the inference latency of the neural network itself. At runtime the variations in inference latency can happen due heterogeneity of devices (CPU, GPU, mobile-CPU, embedded-GPU, TPU), neural network complexity, software stack optimizations, shared resources and several other factors discussed in Section 4.2.2. We summarize a short survey on the current state-of-the-art showing error in the inference latency prediction at runtime in Table 4.1. Table 4.1 also includes the reported inference latency variations due to factors such as heterogeneity of devices and varying complexity of different neural networks. As seen, in the survey results at runtime across a diverse suite of devices, neural network inference latency can be predicted within 10% measurement errors on an average. We evaluate the TSRL policies with measurement noises of up to 20% in Section 4.5.2 which is well within the prediction errors.

The approaches to predict latency can be divided into two categories:

(i) Using a look-up table: A look-up table of individual blocks or operations [CZH18, PMO21, ZHW21] is developed using input/output feature map, kernel size, stride, etc. During prediction latency of each operation is added to get the final expected inference latency.

(ii) Training a separate model to predict latency: Another widely used approach is to train a

Table 4.1: Prediction error in neural network inference latency using different approaches and inference latency variations studied by researchers. Across a suite of devices, neural network inference latency can be predicted within 10% error on an average using different proposed approaches.

Method	Prediction Error	Latency Variations	Tested Devices
Look-up table [CZH18] for individual blocks	1%	80ms - 130 ms	Pixel1
Look-up table and trained models [PMO21]	12%	50ms - 1800 ms	HUAWEI mobile devices
Modified Look-up table using kernels [ZHW21]	10% on CPU/GPU 10% on VPU	1ms -1000 ms	Pixel4, Intel NCS2 Xiaomi Mi9
Trains a GCN model [DCA20] on network graph	10%	1ms - 100 ms	Desktop CPU/GPU Jetson Nano, TPU
Trains a DNN model [AWS22] using performance counters	10%	1ms - 100 ms	Desktop CPU/GPU
Trains a DNN model [LLC21] using hardware embeddings	10%	1ms - 100 ms	Desktop CPU/GPU Raspi4, FPGA

separate neural network [DCA20, AWS22, LLC21] to predict the inference latency of the desired network across different hardware devices. This approach is shown to be generalizable across unseen devices and networks.

Reason for prediction errors: The look-up table approach doesn't account for the data movements [PMO21] between block/operations, which results in prediction errors. Training a separate model suffers from errors when generalizing to unseen devices and networks. The whole expected search space needs to be carefully covered [PMO21].

Runtime latency variations: As seen in Table 4.1, the runtime latency varies from a few ms to 1000's ms across devices with different network complexities. This is because a particular

network's runtime latency depends on the device's choice and the specific implementation framework [PMO21] as discussed in Section 4.2.2.

Due to the increased hardware compute capability of modern GPUs, often, a single inference job cannot fully utilize the GPU resources. Enabling GPU sharing across applications can result in inference latency overhead [GHY22]. Several scheduling approaches are proposed to either maximize the utilization, fairness, or timely response of critical applications [GHY22]. The commercial Nvidia GPUs [Nvi22] support partitioning of a single GPU into as many as seven instances, each fully isolated with its high-bandwidth memory, cache, and compute cores. Mendoza et al. [MW22] have shown extra latency overhead of up to 130ms when sharing GPU across applications. We include additional analysis on execution latency variations in Section 9.1.

4.4 Training Deep-RL Policies with Temporal Variations

A common technique in literature is to do domain randomization during training to account for uncertain and unmodeled environment dynamics [TFR17]. Domain randomization makes sense for physical quantities such as friction and contact forces, as they are difficult to measure and calibrate across real robots. However, state transition delay related variables such as execution time $\Delta\tau_\eta$ and sampling interval $\Delta\tau_\sigma$ are measurable both in simulation and real robots. We propose augmenting the agent state with these measurements: $\tilde{s} = [s, \Delta\tau_\eta, \Delta\tau_\sigma]$ where $\tilde{s}$ represents the augmented state. This simple trick enables the agent to distinguish between state transitions introduced by variations in delays. We refer to the augmented state policies as *Time in State (TS)* policies.

For low-dimensional state spaces, the execution time and sampling interval can be directly added as another state. For augmenting high-dimensional state spaces, such as images, the delay observations are fused in an intermediate layer of both policy and value networks [NKK11]. To evaluate TSRL, we instrumented *HalfCheetahBulletEnv-v0* and *AntBulletEnv-v0* environments in the PyBullet simulator [CB16] to demonstrate Sim2Sim transfer on a low dimensional use case, as well as DeepRacer [BMG19] to test both Sim2Sim and Sim2Real transfer on a high dimensional

use case. We train our policies using the Proximal Policy Optimization (PPO) [SWD17] algorithm as implemented in the OpenAI Baselines[1]. We use PPO as it has been widely used in robotics applications [BMG19, ABC20, MCH19].

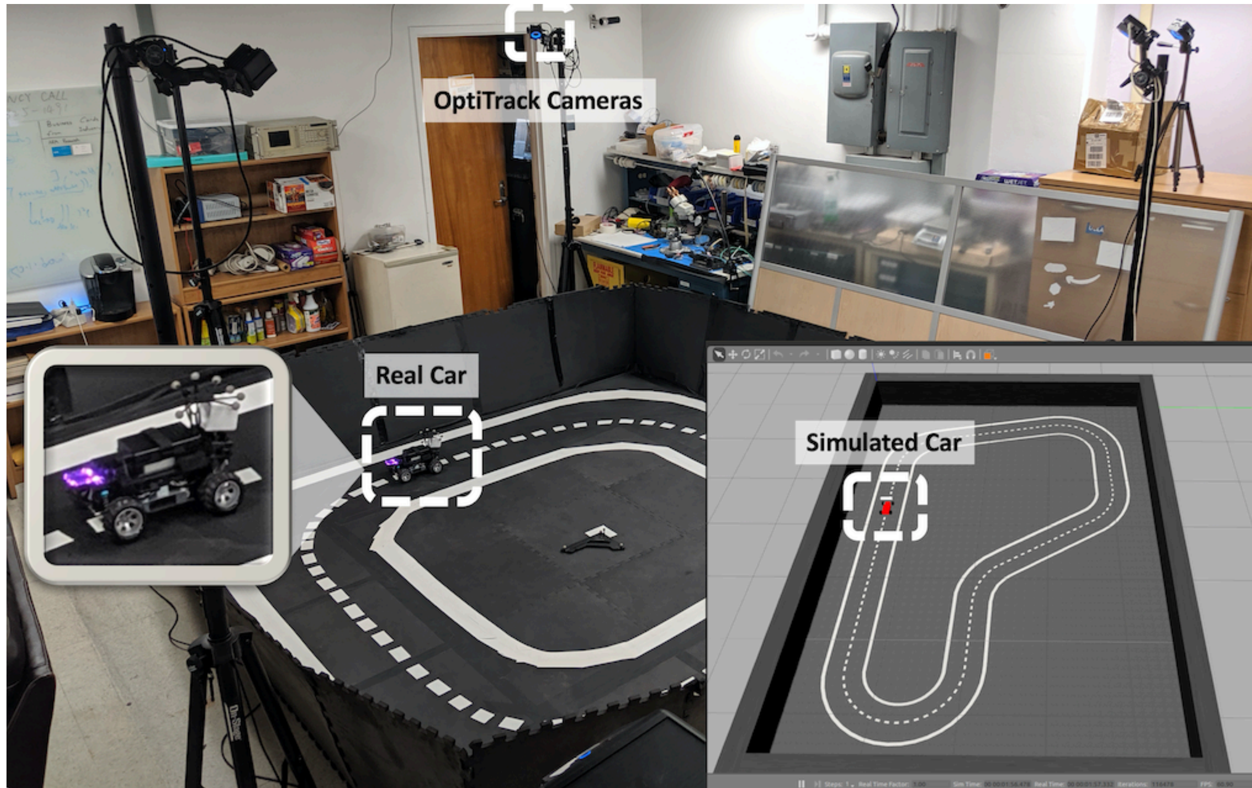
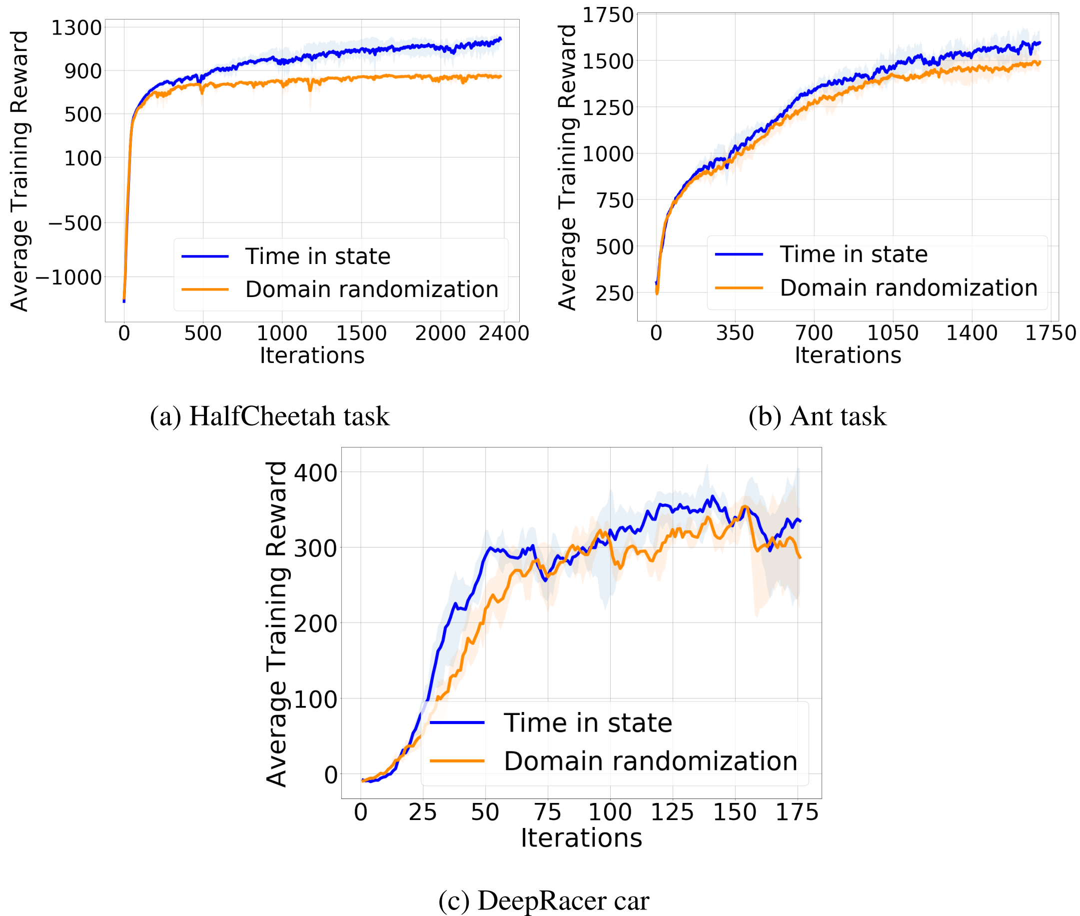


Figure 4.2: The DeepRacer car on a real track and the simulated car in the Gazebo environment. The OptiTrack motion capture system is used to quantify the performance of policies on the real track.

4.4.1 Low Dimensional Use Cases: HalfCheetah and Ant

The implementation for *HalfCheetahBulletEnv-v0* and *AntBulletEnv-v0* in PyBullet simulator evolve physics at a fixed time (Sim_{Time}) of 4.12 ms for each action. We modified the default environments and advance the simulation for multiple simulation steps per action to vary the execution latency

[1]OpenAI Baselines: `https://github.com/openai/baselines`



(a) HalfCheetah task

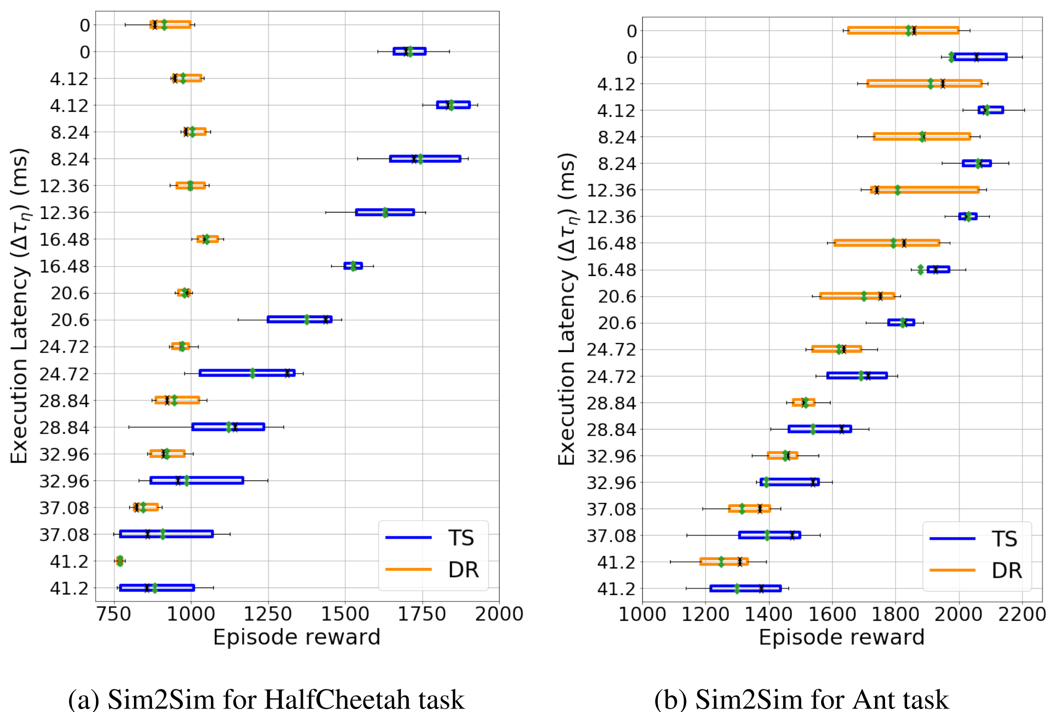
(b) Ant task

(c) DeepRacer car

Figure 4.3: The learning curves for HalfCheetah, Ant and DeepRacer car for TS and DR policies.

and sampling interval. Thus, the granularity of the variation in the execution latency and sampling interval in our PyBullet experiments is Sim_{Time}.

Variation of Timing Characteristics. We vary the state transition delays in the simulator when training the policies and consider the setting where execution latency $\Delta\tau_\eta \leq$ sampling interval $\Delta\tau_\sigma$. For reactive systems, this setting is desired so that the agent can act for every sensed state [Hal13]. The actuation of recent action is delayed by the execution latency $\Delta\tau_\eta$, and the next sensor sample is available after the sampling interval $\Delta\tau_\sigma$. We select the range of execution latencies $\Delta\tau_\eta$ between [0 - $10 * Sim_{Time}$] = [0 - 41.2 ms] and sampling interval $\Delta\tau_\sigma$ values between [Sim_{Time} - $10 * Sim_{Time}$] = [4.12 ms - 41.2 ms]. We varying $\Delta\tau_\eta$ and $\Delta\tau_\sigma$ for HalfCheetah and Ant within their respective ranges. Before the starting of episode, we fix $\Delta\tau_\eta$ and then decide $\Delta\tau_\sigma$ = max(4.12 ms , $\Delta\tau_\eta$).



(a) Sim2Sim for HalfCheetah task

(b) Sim2Sim for Ant task

Figure 4.4: Comparison of time-in-state (TS) and domain randomization (DR) policies for HalfCheetah and Ant tasks across different execution latencies ($\Delta\tau_\eta$). The sampling intervals ($\Delta\tau_\sigma$) is selected to be maximum of (4.12 ms , $\Delta\tau_\eta$), so that agent can act for each sensed state. The mean is shown in green, the black 'x' marker shows the median of IQR. For both tasks, TS policies achieve higher mean reward than DR policies.

Between consecutive steps, we introduce random jitter of $\pm Sim_{Time}$ in both $\Delta\tau_\eta$ and $\Delta\tau_\sigma$.

Policy Training. The vanilla policy trained without varying state transition delays fails to work in presence of variable ($\Delta\tau_\eta$ and $\Delta\tau_\sigma$). We present analysis of the vanilla policy in Section 4.7. We use domain randomization (DR) as our baseline algorithm, where the policy is trained by varying the state transition delays during training. We train the DR and TS policies by varying the $\Delta\tau_\eta$ and $\Delta\tau_\sigma$ as described above. When training DR policies, the default state from *HalfCheetahBulletEnv-v0* and *AntBulletEnv-v0* is used. We augment the state with $\Delta\tau_\eta$ and $\Delta\tau_\sigma$ for TS policies. We use 2-layer fully connected neural network with each layer having 64 nodes for policy and value function. We include additional details for reproducibility in Section 9.2.

4.4.2 High-Dimensional Use Case: Autonomous Vehicle

We use Gazebo simulator[2] to train navigation policies for the DeepRacer car [BMG19]. The simulator includes a robot model that is matched to the properties of the real car. The simulation advances in real-time. Images from the camera are used to navigate the car on the track. We modify the sampling rate of camera and runtime execution latency by adding controlled timing delays.

Variation of Timing Characteristics. During training in simulator, the execution latency $\Delta\tau_\eta$ is varied between the discrete set of values from 10 ms to 120 ms. The sampling interval $\Delta\tau_\sigma$ of 33 ms (30 Hz) is used when $\Delta\tau_\eta \leq \Delta\tau_\sigma$, otherwise $\Delta\tau_\sigma$ is matched to $\Delta\tau_\eta$. During policy training for each episode, we fix the value of $\Delta\tau_\eta$ and $\Delta\tau_\sigma$. The execution latency to do the policy network inference and image processing on the server machine is $\sim$10ms. The execution latency on the real car using integrated GPU is $\sim$20ms in the absence of other tasks. The sampling interval of 33 ms (30 Hz) corresponds to the supported camera sampling rate on the real car. The range of variations in $\Delta\tau_\eta$ and $\Delta\tau_\sigma$ are selected to benchmark the policy behavior of navigational policy in the presence of deployment variations of hardware, multi-tenancy, and communication delays.

Policy Training. The DR and TS policies are trained by varying the state transition delays as described above. In addition, we use the recommended image augmentations [BMG19] to enable the successful transfer of policy to the real car. Due to the image augmentation processing times and variations in simulation advancement, a jitter of 5 ms is present in both $\Delta\tau_\eta$ and $\Delta\tau_\sigma$. The simulation setting, along with the track and simulated car, is shown in Figure 4.2. The simulated track has a centerline of the length of 17 meters and a track width of 0.44 meters. The policy's goal is to follow the centerline of the track by controlling the steering angle and speed. The highest reward of 1.1 is given when the center of the car matches the centerline, and the reward is scaled to zero as the car moves away from the centerline to offtrack. Each episode consists of 500 steps. The neural network is represented by 2 CNN layers followed by a 2 fully connected layers and an output layer. The DR policy uses only the images from the camera as input. For TS policy, we fuse

[2] http://gazebosim.org/

images with the execution latency and sampling interval in the first fully connected layer after the CNN layers. We provide the details for reproducibility in Section 9.3.

Real-world environment. We created a real track as shown in Figure 4.2, with a center line distance of 7.3 meters and the track width of 0.52 meters. We compared the performance of policies by utilizing an OptiTrack motion capture system[3] shown in Figure 4.2. We localized the location of the car with respect to the center line of the real track.

4.5 Evaluation of Time-in-State RL

We compare the time-in-state (TS) based policies and the domain randomization (DR) policies in simulation and on the real robot. We evaluate their robustness across varying sampling intervals and execution latencies. This section compares the performance of fully connected policies. The experiments with recurrent policies are discussed in Section 4.6.

4.5.1 HalfCheetah and Ant Tasks

TS and DR policies are trained for HalfCheetah and Ant tasks, as explained in Section 4.4.1. We train three models for each task, both for TS and DR, with different seeds. The learning curves for the policies are shown in Figure 4.3. For both tasks, TS achieves a better mean training reward than the DR policies. The evaluation of policies for both HalfCheetah and Ant tasks across the three trained models is shown in Figure 4.4. The spread of test reward is captured across 10 episodes for each model at a particular sampling interval ($\Delta\tau_\sigma$) and execution latency ($\Delta\tau_\eta$). The analysis shows that the TS polices perform better than the DR policies across state transition delay variations and maintains a higher mean reward. Figure 4.4 also shows that, in general, the performance of deep-RL policies degrades when exposed to higher sampling intervals and execution latencies. The degradation in performance is task-specific.

[3] https://optitrack.com/

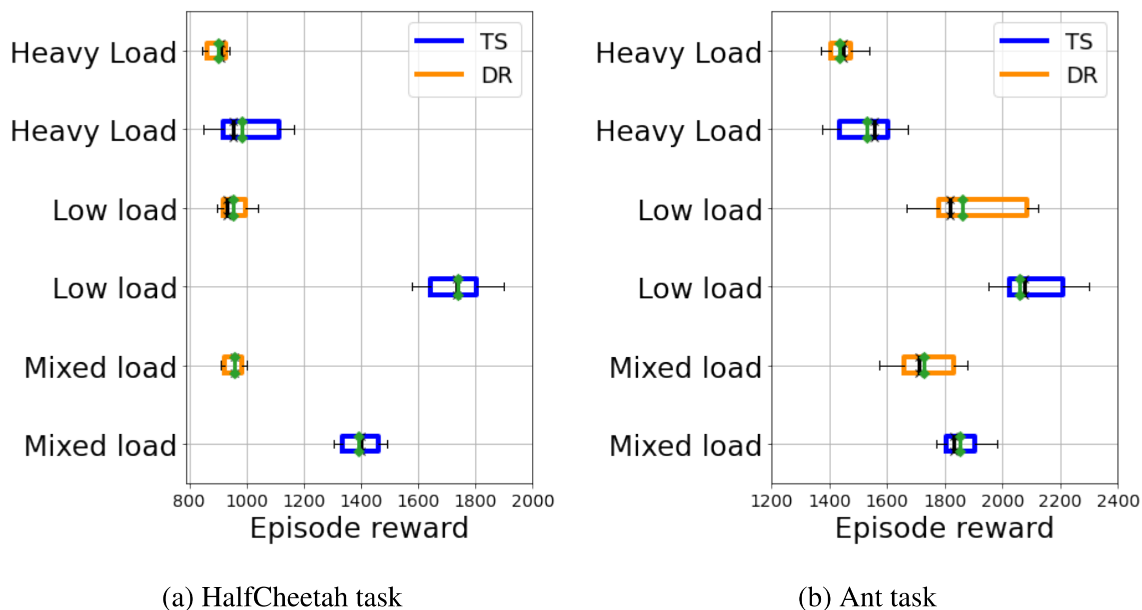
Next, we evaluates the policies with variable delays within an episode and with 20% timing noises in delays for HalfCheetah.

4.5.2 Experiments with Variable Delays within an Episode and Timing Noises

The experiments at different delays shown in Figure 4.4 and Figure 4.7 highlight Time-in-state policy's superior performance across a wide range of delay variations, along with quantifying the drop in performance as the delay magnitude is increased across episodes. Next, we expose the trained fully connected policies of Halfcheeth and Ant tasks to variable delay in a single episode.

We consider three different multitenancy settings: (i) low load, (ii) heavy load, and (iii) mixed load, exposing policies to a range of delay variations. These experiments simulate the arrival of multiple parallel tasks on the same hardware running the deep RL policy. In the low load setting, the execution latency is selected between [0-$2 * Sim_{Time}$] randomly for each step during the episode. For both HalfCheetah and Ant Task, the $Sim_{Time} = 4.12$ms, and each episode consists of 1000 steps. For heavy load, the execution latency is varied randomly between [$6 * Sim_{Time}$-$10 * Sim_{Time}$]. The execution latency is varied randomly between [0-$2 * Sim_{Time}$] for the first 333 steps, [$3 * Sim_{Time}$-$5 * Sim_{Time}$] for the next 333 steps, and [$6 * Sim_{Time}$-$10 * Sim_{Time}$] for the remaining 334 steps in the case of mixed load setting. The sampling interval is selected to be a maximum of (Sim_{Time}, execution latency). A random jitter of $\pm Sim_{Time}$ is added to the execution latency and the sampling interval during each step to account for the measurement noises. This can result in measurement noise of $2Sim_{Time}$ or 20%.

As shown in Figure 4.5, Time-in-State policies have superior performance as compared to the domain randomization policies. The spread of reward is captured across 10 episodes for each model at a particular multitenancy setting. The variation in Figure 4.5 follows the same behavior shown in Figure 4.4. For example, at a low execution latency (in Figure 4.4) for the HalfCheetah task, the difference in the performance of Time-in-state policy and domain randomization policy is significant. A similar significant performance difference is observed in the low load setting for



(a) HalfCheetah task

(b) Ant task

Figure 4.5: Comparison of time-in-state (TS) and domain randomization (DR) policies for HalfCheetah and Ant tasks across different multitenancy settings. The mean is shown in green. The back 'x' marker shows the median of IQR.

the HalfCheetah task in Figure 4.5. The heavy load and mixed load also follow the performance difference observed in Figure 4.4.

4.5.3 DeepRacer Robotic Car

Figure 4.3c shows the learning curve of TS and DR policies trained using DeepRacer simulator. We train 3 models for each policy. DeepRacer simulator advances simulation in real-time, and we train each model for 44 hours. Figure 4.7b shows the Sim2Sim of the policies across 3 set of models using the DeepRacer simulator. We evaluate each model for 16 episodes (500 steps on track for each episode). We test the policies across a spread of different sampling intervals ($\Delta\tau_{\sigma}$) and the execution latencies ($\Delta\tau_{\eta}$). The results also show that as $\Delta\tau_{\eta}$ and $\Delta\tau_{\sigma}$ are increased, the performance of the deep-RL policy degrades in general. However, in comparison to DR, the TS policies have better performance.

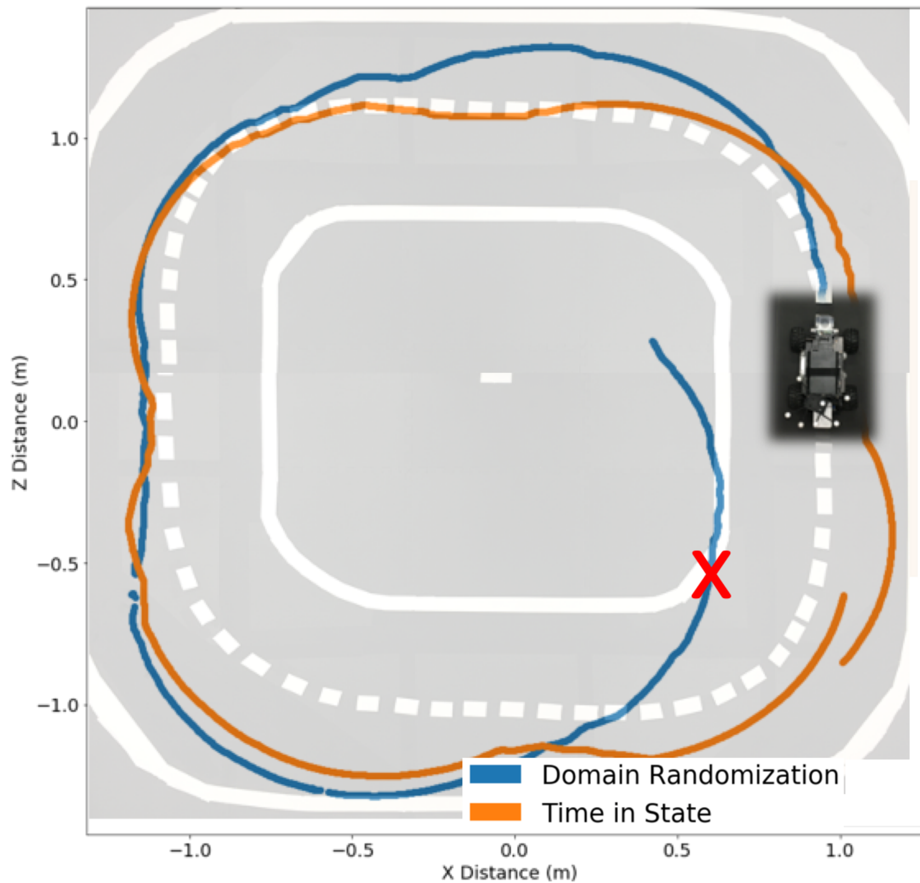


Figure 4.6: A comparison of a single instance of two deep reinforcement learning-based controllers on a $1/18^{th}$ scale real autonomous car in the presence of 60 ms execution time. The proposed Time-in-State (TS) based controller performs better than the domain randomization (DR) based controller.

Table 4.2: (a) Comparison of time-in-state (TS) and domain randomization (DR) policies in completing laps on the real track out of 24 trials at different execution latencies. (b) The average speed used by TS and DR. TS adapts its speed with increase in execution latency.

Latency	20 ms	60 ms	100 ms
TS	20	17	13
DR	20	11	7

(a)

Latency	20 ms	60 ms	100 ms
TS	1.50 m/s	1.45 m/s	1.40 m/s
DR	1.45 m/s	1.44 m/s	1.45 m/s

(b)

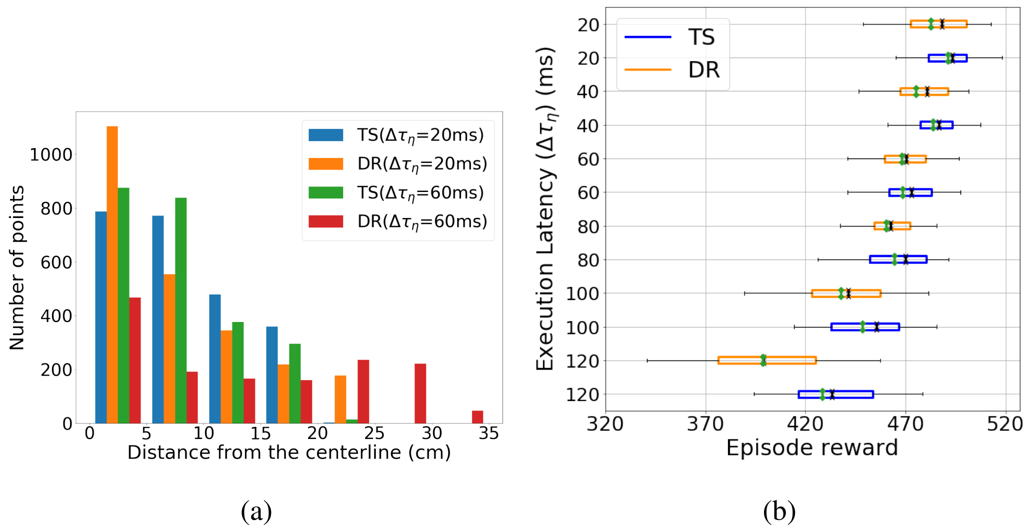


Figure 4.7: Evaluation of time-in-state (TS) and domain randomization (DR) policies using DeepRacer car. (a) The distance of the real car from the centerline captured using OptiTrack cameras. The number of points plotted is 2400, except the DR ($\Delta\tau_\eta$=60ms), which has 1657 points. The onboard camera of car was running at 30Hz. (b) Analysis of TS and DR policies across different execution latencies ($\Delta\tau_\eta$) in DeepRacer simulator. The sampling intervals ($\Delta\tau_\sigma$) is selected to be maximum of (33 ms , $\Delta\tau_\eta$). The green color shows mean, the black 'x' marker shows the median of IQR.

Sim2Real transfer. We compare the performance of the TS and DR policies on the DeepRacer robot using the real track. The results for 24 trials in both the directions for TS and DR policies across the 3 trained models are shown in Table 4.2a. The results shows the performance gain of the TS policies is transferred to real robot. DR and TS policies work very well on the real track by successfully completing higher number of laps for $\Delta\tau_\eta$ = 20ms. As the $\Delta\tau_\eta$ is increased to 60ms and 100ms, the performance of DR policies significantly degrade in comparison to the TS policies. Table 4.2b shows the average action speed of policies in the simulator track. TS adapts its speed with increasing execution latency by taking slower actions whereas DR does not change its action speed. In our supplementary video, we show that the TS policy speed adaptation occurs primarily on the curved regions of the track, and helps it achieve robust navigation. The $\Delta\tau_\eta$ of

neural network policy using GPU of Car is within 15-20ms. We introduce extra delay and fix the $\Delta\tau_\eta$ to 20ms, 60ms and 100ms respectively to generate the comparison of TS and DR policies. The measured sampling interval $\Delta\tau_\sigma$ was directly given as input to the TS policies. The camera was running at the sampling rate of 30 Hz, which was measured to have variable sampling interval from 25-45ms at runtime. Figure 4.6 shows an instance of the real run captured using OptiTrack setup comparing TS and DR based policies for the sampling rate of 30 Hz and $\Delta\tau_\eta$ of 60ms. The TS policies have more stable performance on the real track as compared to the DR policies. We analyzed the distance from the centerline maintained by both TS and DR policies on the real track. The distance is captured using OptiTrack setup. The distance maintained is shown in Figure 4.7a. TS policies maintain a closer distance to the centerline. The DR policies have more oscillating behavior around the centerline. We believe the oscillating behavior is the reason for higher number of points within [0-5 cm] for DR at $\Delta\tau_\eta$ to 20ms.

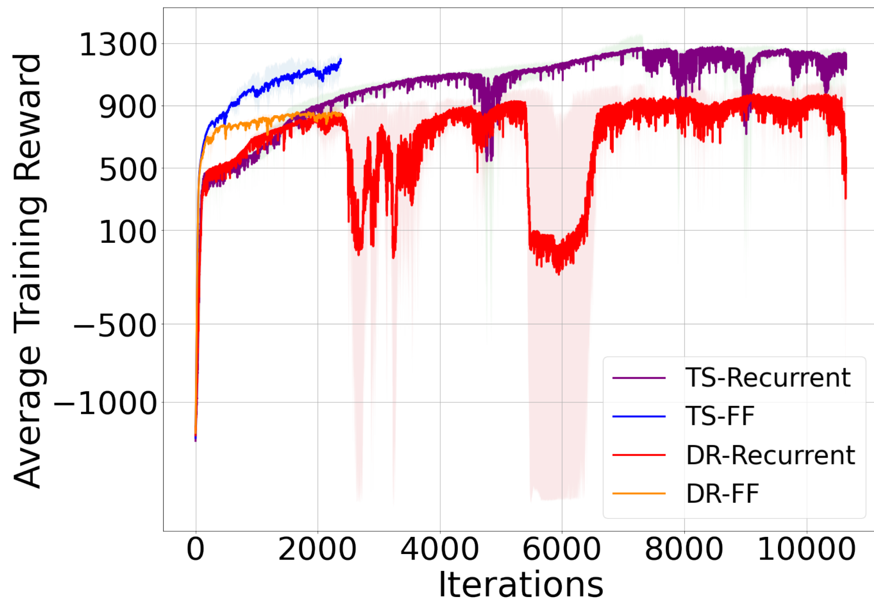


Figure 4.8: Learning curves of time-in-state recurrent (TS-Recurrent), time-in-state fully connected (TS-FF), domain randomization recurrent (DR-Recurrent), and domain randomization fully connected (DR-FF) policies for HalfCheetah task. The fully connected policies are trained for $\sim$ 2400 iterations whereas the recurrent policies are trained for $\sim$ 10000 iterations. TS policies achieve higher training reward than the DR policies.

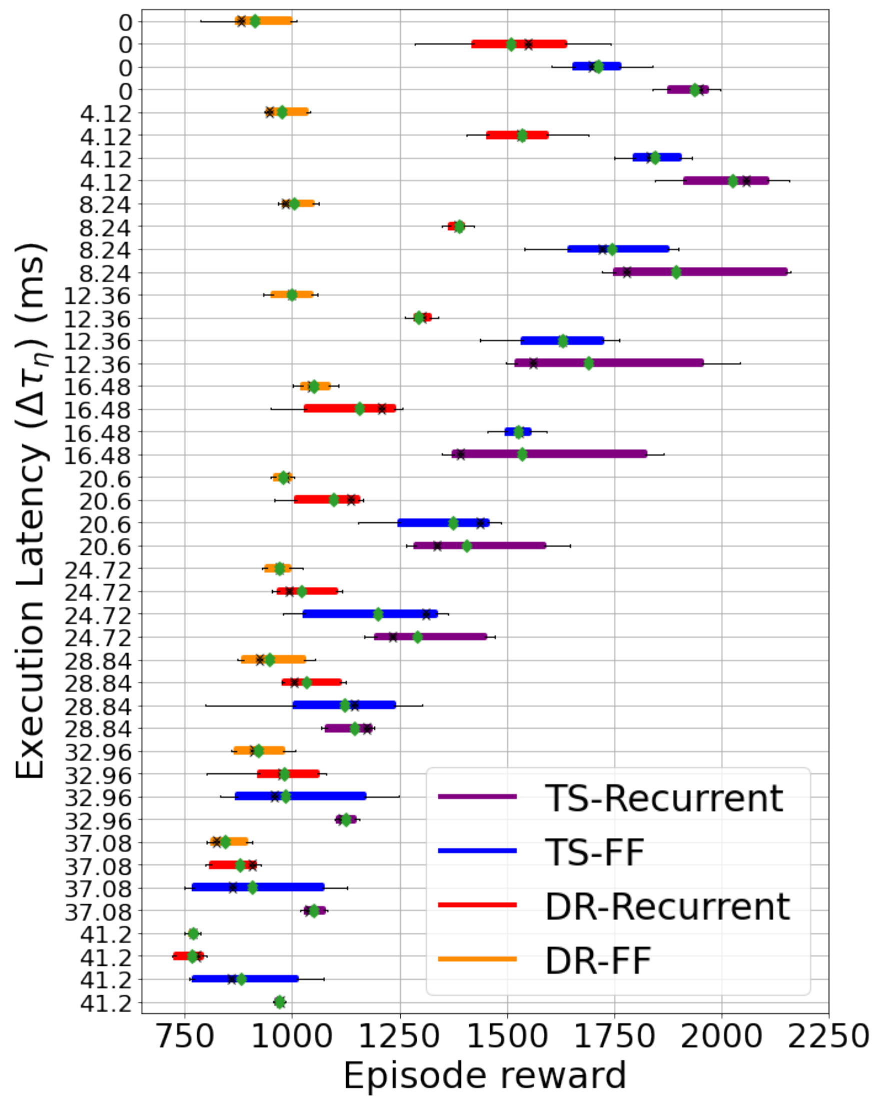


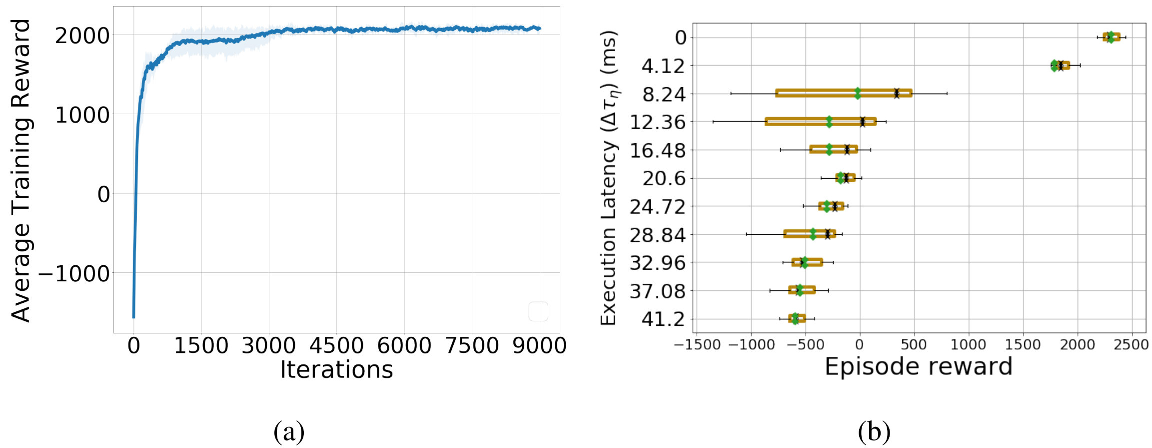
Figure 4.9: Sim2sim comparison of time-in-state recurrent (TS-Recurrent), time-in-state fully connected (TS-FF), domain randomization recurrent (DR-Recurrent), and domain randomization fully connected (DR-FF) policies for HalfCheetah task. The comparison is done across different execution latencies ($\Delta\tau_\eta$). The sampling intervals ($\Delta\tau_\sigma$) is selected to be maximum of (4.12 ms , $\Delta\tau_\eta$), so that agent can act for each sensed state. The mean is shown in green, the black 'x' marker shows the median of IQR. TS policies achieve higher mean reward than DR policies.

4.6 Experiments with Recurrent Policies

The recurrent policies are known to perform better than the fully connected policies for tasks where the partial state is observed. We evaluate TS and DR policies with recurrent architectures for the HalfCheetah task. The state space, actions, and reward function used are the same as the one discussed in Section 9.2 for fully connected policies. We modified the open-source code available from Hafner et al. [HDV17] to train recurrent policies. The variation of timing characteristics during the training was done as discussed in Section 4.4.1. The network architecture consists of a fully connected layer with 64 nodes, followed by a GRU layer with 64 units, and an output layer.

We train 3 models each for recurrent TS and recurrent DR, with different seeds. Figure 4.8 compares the learning curves of recurrent and fully connected policies. The learning curves for fully connected policies, also shown in Figure 4.3a, are added here for comparison. We stopped the training of fully connected policies after $\sim$ 2400 iterations. The recurrent policies require a significantly larger number of iterations ($\sim$ 10000) to train. The recurrent TS and recurrent DR achieve higher training rewards as compared to the fully connected TS and fully connected DR respectively. The max average training reward achieved by fully connected TS and fully connected DR is 1199 and 857 respectively. The recurrent TS and recurrent DR achieves max average training reward of 1278 and 974 respectively. We observe that the fully connected TS achieves significantly higher training reward than the recurrent DR, suggesting that adding time to the state is a better approach than training a recurrent DR policy.

Figure 4.9 shows the comparison of recurrent policies and fully connected across three trained models. The Sim2sim comparison of fully connected policies is added from Figure 4.4a for comparison. The spread of test reward is captured across 10 episodes for each model at a particular sampling interval ($\Delta\tau_{\sigma}$) and execution latency ($\Delta\tau_{\eta}$). The TS policies perform better than the DR policies across a range of state transition delay variations, maintaining a higher mean test reward.



(a) (b)

Figure 4.10: (a) The learning curves for vanilla policy training for the HalfCheetah task. (b) The evaluation of vanilla policy across different execution latencies ($\Delta\tau_\eta$). The sampling intervals ($\Delta\tau_\sigma$) is selected to be maximum of (4.12ms , $\Delta\tau_\eta$). The mean is shown in green. The back 'x' marker shows the median of IQR.

4.7 Vanilla Deep Reinforcement Learning Policy without Varying Timing Characteristics

We train a vanilla policy for the HalfCheetah task without varying the $\Delta\tau_\sigma$ and $\Delta\tau_\eta$. This policy is trained by using the default *HalfCheetahBulletEnv-v0* environment in which the simulation is advanced for a fixed time (Sim_{Time}) of 4.12 ms for each action. By default, the execution latency ($\Delta\tau_\eta$) of 0 is present, as the simulation is paused when deciding action (by doing neural network inference and other processing). Since updated state is available every Sim_{Time}, it has a fixed sampling interval ($\Delta\tau_\sigma = Sim_{Time}$). We train a set of three models for the vanilla policy with different seeds. The learning curve of the vanilla policy is shown in Figure 4.10a. Figure 4.10b shows that vanilla policy fails to work for execution latencies ($\Delta\tau_\eta$) $\geq$ 8.24 ms, which is twice the Sim_{Time}. This motivates the need to have variable state transition delays ($\Delta\tau_\sigma$ and $\Delta\tau_\eta$) during training to have policies robust to variable timing characteristics.

4.8 Comparison of Time-in-state with Worst Case Delay Controller

Here, we compare the TSRL with worst-case delay controllers for HalfCheetah task. The TSRL policy is trained using the Section 4.4.1. We train 2 worst-case delay controller assuming 3x4.12 ms and 5x4.12 ms delays (3X and 5X) delays. The worst-case delay controller are trained by assuming the fixed latency. The action is delayed by the desired execution latencies $\Delta\tau_\eta$ (3x4.12 ms or 5x4.12 ms delays) during training. The learning curves for Time-in-state and 5x4.12 ms worst-case delay (Fixed latency) are shown in Figure 9.3. Both Time-in-state policies and worst-case delay policies are trained for 3000 iterations.

Performance comparisons Figure 4.11 compare the performance of Time-in-state (TS) with several other approaches. The TS-Noise (20%) shows the performance when 20% noise is added to the measured sampling interval and execution latency. The reward shown in Figure 4.11 is the mean across 10 episodes for each model at a particular sampling interval ($\Delta\tau_\sigma$) and execution latency ($\Delta\tau_\eta$). The sampling intervals ($\Delta\tau_\sigma$) is selected to be maximum of (4.12 ms , $\Delta\tau_\eta$), so that agent can act for each sensed state. We introduce a random noise of $\pm Sim_{Time}$ in both $\Delta\tau_\eta$ and $\Delta\tau_\sigma$ for TS-Noise (20%).

As seen in Figure 4.11 TS policies and TS-Noise (20%) policies work across all latency variations. The Worst-Case3x (3x4.12 ms) and Worst-Case5x (5x4.12 ms) have stable performance up to 3X (12.3 ms) and 5X (20.6 ms) latencies. Both of these policies show immediate performance loss beyond these limits. Worst-Case5x (5x4.12 ms) policies are also inferior to TS when latencies are less than 3X (12.3ms). We see worst-case delays are a viable solution when the upper bounds on delays are small, as in the case of Worst-Case3x, which outperforms all policies for delays $<$3x4.12 ms. However, when the delay variations are larger or worst-case delays are significant, TS policies are a clear winner.

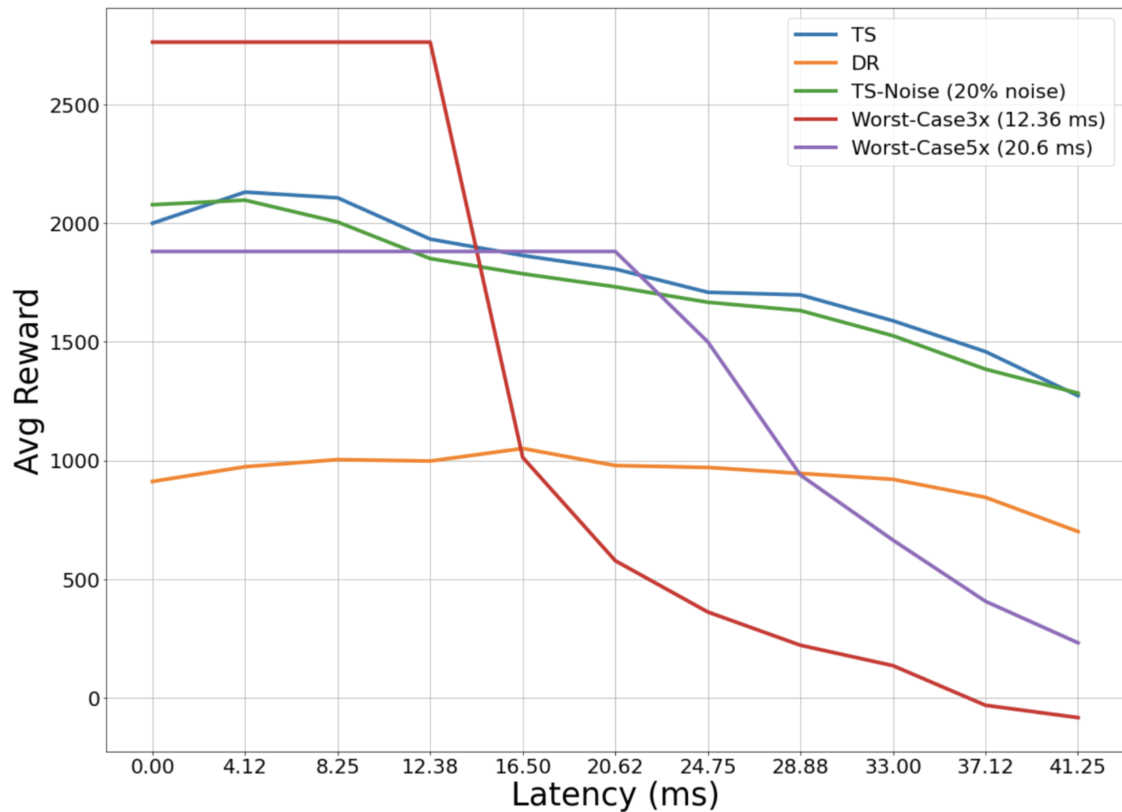


Figure 4.11: The performance comparison of Time-in-state (TS), Time-in-state with noisy measurements (TS-Noise), Domain Randomization, Worst-case delay of 3X (Worst-Case3x), and Worst-case delay of 5X (Worst-Case5x) for HalfCheetah task. The Worst-Case3x (3x4.12 ms) and Worst-Case5x (5x4.12 ms) have stable performance up to 3X (12.3 ms) and 5X (20.6 ms) latencies. However, worst-case policies show immediate performance loss beyond their worst-case limits. When the delay variations are large or worst-case delays are significant; TS policies are a clear winner.

Discussion We empirically see in the analysis of the HalfCheetah task in Figure 4.11 that when the worst-case latency is 3X (12.3 ms), the policy performs very optimally. However, when the delay is increased beyond the expected worst-case limit, the worst-case policy is significantly impacted and fails to work at higher delays. Further, when the worst-case delay is significant, as seen for 5X (20.6 ms), the Time-in-state policies can achieve superior performance even for smaller delays

(< 3X). Our introduced Time-in-state approach presents a way to train adaptive policy across significant delay variations and maintains superior performance when worst-case delays notably impact the performance. Time-in-state approach also trains a single policy instead of multiple worst-case delay policies that can transfer to different deployment hardware (e.g., Mobile CPU, embedded GPU, etc.) having variable delays at runtime.

4.9 Conclusion

We introduced Time-in-State RL (TSRL), a delay-aware deep reinforcement learning approach that incorporates sampling interval and execution latency into its state space. By utilizing domain randomization with time in the state, TSRL's policies are robust against varying execution latencies and sampling rates for both Sim2Sim and Sim2Real transfer. The application performance characterization of TSRL can be exploited by policies to conserve platform resources by acting slowly along with staying within the desired reward budget. The performance characterization can also help developers make informed decisions in selecting appropriate compute hardware and deployment settings. The evaluation of time in state policies show that the policies are able to maintain higher rewards across a range of timing characteristics and, thus, can be used in presence of deployment uncertainties impacting the timing characteristics at runtime. Through a range of choices of latencies and sampling intervals, our study also shows different tasks can work reasonably only up to to a certain latency and after that suffers significant degradation in performance. We hope the concepts and results introduced in this chapter will motivate the development of deep-RL policies that are robust to runtime uncertainties.

CHAPTER 5

End-to-end Deep Reinforcement Learning for Autonomous Control of PTZ Cameras

Enabling end-to-end data-driven control for CPS applications is preferred to avoid the dependence on domain expertise to realize optimal multiple stages of conventional control pipelines. However, end-to-end data-driven control approaches such as deep reinforcement learning are extremely data-hungry and are almost impossible to train in the real world. In this chapter, we study the design of end-to-end control in rich simulations using a representative application of a pan-tilt-zoom camera.

The conventional approaches for autonomous control of pan-tilt-zoom (PTZ) cameras use multiple stages where the object detection and localization are performed separately from the control of the PTZ mechanisms. However, these approaches suffer from performance bottlenecks due to multiple stages of information flow that are difficult to optimize jointly. The complex neural networks widely adopted for the object detection stage also make them infeasible for real-time deployment in resource-constrained environments. In contrast, we propose an end-to-end deep reinforcement learning (RL) approach called *Eagle* to train a neural network policy that directly takes an image as input to control the PTZ parameters.

Training reinforcement learning is cumbersome in the real world due to labeling effort, runtime environment stochasticity, and fragile experimental setups. To enable successful training of Eagle, we also introduce *EagleSim*, a photo-realistic simulation framework for PTZ cameras that automatically captures ground truth annotations. EagleSim addresses the simulation-to-reality gap, a significant challenge in deep-RL, by supporting rich scene variations with different background materials, trees, human characters, and multiple types of vehicles. Another advantage of Eagle

policies is that they are lightweight (90x fewer parameters than Yolo5s) and can run on embedded camera platforms such as Raspberry PI (33 FPS) and Jetson Nano (38 FPS), facilitating real-time PTZ tracking for resource-constrained environments.

We also compare the performance of Eagle with PTZ trackers using custom-developed lightweight object detectors. We train the lightweight object detectors using large datasets having rich scene variations captured using the EagleSim simulator. We choose lightweight detectors with network architecture similar to the Eagle policy, thus having the same inference latency as Eagle. Our evaluation shows that Eagle achieves superior camera control performance by maintaining the object of interest close to the center of captured images at high resolution and has up to 16% more tracking duration than the next best approach. We test the generalizability of Eagle on unseen objects/scenes and highlight the direct transfer of policies trained purely in the simulator to the real scene videos.

5.1 Introduction

Active vision endows applications with the ability to decide 'where to look' at runtime. Autonomous control of pan-tilt-zoom (PTZ) cameras can provide superior monitoring for active vision systems by tracking objects of interest in real-time [MRF10, CLK11]. Active vision systems are increasing deployment in resource-constrained environments such as remote surveillance [MRF10, HZL17, BXD20] and mobile robotics [CLK11, UNC19] demanding lightweight algorithms for real-time PTZ control.

Existing autonomous PTZ controllers have multiple stages, namely detection of objects of interest, tracking of their trajectories, and control of PTZ parameters to keep objects in the field-of-view [BVS07, CSB15, HZL17, WDH16, MRF10]. It is common to use neural object detectors to identify objects of interest, first-principles-based algorithms such as Kalman filters for trajectory prediction using model-based state estimation on bounding boxes, and a separate controller [BGO16, UNC19, LMC21]. Although autonomous PTZ control algorithms have been stud-

ied for many years, the current multi-stage pipeline faces the following challenges:

1. Expensive fine-tuning efforts: The multiple stages suffer from performance bottlenecks as it is non-trivial to tune each step. For example, tuning the parameters of the Kalman filter requires expert domain knowledge and can incur many trial-and-errors [Kyr21, LSZ19]. Fine-tuning neural object detectors for a specific deployment may require human efforts to label bounding boxes [LSZ19, CSB15].

2. Real-time deployment on resource-constrained platforms: Even the lightweight object detectors (such as YOLO [RF17, Kyr21] with several millions of network parameters are too complex for embedded camera platforms. This make it infeasible to run multi-stage PTZ control algorithms in real-time on platforms [Ard22, Kyr21] having memory and computation constraints.

Along with the above challenges, it is also difficult to evaluate the performance of existing multi-stage approaches and to explore design of new control algorithms due to the complexity of creating PTZ tracking scenarios in the real world. The dynamic nature of PTZ tracking involves both sensing the objects of interest and control of the camera's pan-tilt-zoom parameters that can only be performed online [CSB15, SCD11]. The online nature makes it difficult to set up and reproduce real-world experiments due to runtime stochasticity in object/camera movements, illumination changes, and the requirement of ground truth annotations [CSB15].

To address these challenges, we propose Eagle, an end-to-end deep reinforcement learning (RL) approach using raw images to control a PTZ camera. Eagle trains a neural network policy directly mapping raw images to pan-tilt-zoom actions removing the multiple stages of object detection, localization, and control. Recently, deep-RL has been shown to outperform conventional control for several robotic applications [LSZ19, BMG20, ABC20, PAZ18]; to the best of our knowledge, there has not yet been any attempt to develop an end-to-end Deep-RL policy for PTZ cameras. This is partly due to the challenges of training deep-RL in the real world as it requires a large number of environment interactions, expensive experimental setups, and labeling efforts [LSZ19, BMG20, ABC20, PAZ18]. This is further exacerbated due to the difficulty of creat-

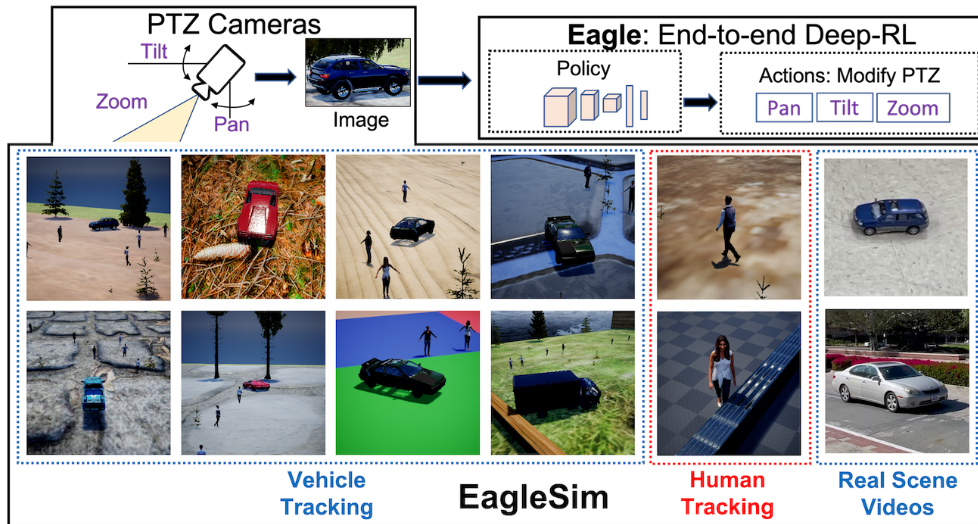


Figure 5.1: Eagle trains end-to-end deep-RL controllers for PTZ cameras. Sample scenes for vehicle and human tracking from the EagleSim simulator are shown. The direct transfer of Eagle policies to real scene videos is also demonstrated.

ing PTZ tracking scenarios in the real world. Although several PTZ frameworks [SCD11, CSB15, HZL17] exist, they either require human labeling effort or demand specialized equipment.

To enable successful training of Eagle, we also introduce EagleSim, a simulation framework for placement and control of PTZ cameras in photo-realistic virtual worlds. EagleSim enables the creation of reproducible tracking scenarios and automatically captures fine-grained ground truth annotations. Although training in simulators is extensively explored in deep-RL, transferring simulation policies to the real world is still a challenge [BMG20, PAZ18, LSZ19, SGB20]. EagleSim includes a significant engineering effort to provide scene variations with multiple objects (vehicles and human characters) and different surroundings (background materials/patterns and trees) as shown in Figure 5.1. We demonstrate that these rich scene variations are necessary to train a generalizable policy and to bridge the gap between simulation and the real world. EagleSim allows the creation of multiple parallel scenes that reduces the training times for complex vehicle tracking scenarios from 17 days to 2.9 days on a GPU machine (GeForce RTX 3090 Ti [Bal21]).

Using EagleSim simulator, we compare Eagle with existing state-of-the-art approaches for

autonomous PTZ control in three categories: (i) *Object_detection+tracking+control*, (ii) *Object_detection+reinforcement learning*, and (iii) *Relative_location+control*. Our results show that Eagle outperforms current approaches across a suite of vehicle tracking scenarios by achieving more tracking duration. Eagle train lightweight neural network policies (79k model parameters and 320KB model size) that are real-time deployable on resource-constrained embedded camera platforms having computation limitations of Raspberry PI (30 FPS) and Jetson Nano (40 FPS) class devices.

We further evaluate the performance of Eagle as the tracking complexity is increased and its generalizability to unseen objects/surroundings. To enable flexible tracking goals, we introduce an extra contextual input along with images during training. Depending on the application's needs, the contextual input can modify the policy behavior at deployments, such as either tracking vehicles or humans.

We also investigate the design of custom PTZ trackers using lightweight object detectors. We design lightweight object detectors for vehicles by removing the complexity of widely used object detectors such as Yolo. The light object detectors have network complexity same as the Eagle policy network. We train six different detectors on datasets captured using the EagleSim simulator. Our results show that Eagle also outperforms the custom PTZ trackers utilizing lightweight object detectors and maintains superior PTZ control performance having up to 16% more tracking duration across scenarios. Finally, we show that Eagle policies trained purely in simulation transfer directly to the real videos. To allow testing on real videos, EagleSim includes the capability to simulate pan, tilt, and zoom on the pre-recorded videos.

In summary, we make the following contributions in this chapter:

- We present Eagle, an end-to-end deep-RL approach to control a PTZ camera directly using raw images. Eagle is lightweight to enable deployment on real-time embedded camera platforms and doesn't require multi-stage fine-tuning.
- We introduce a new simulator called EagleSim to study PTZ cameras in photo-realistic virtual worlds. EagleSim is designed to create reproducible PTZ tracking scenarios with rich

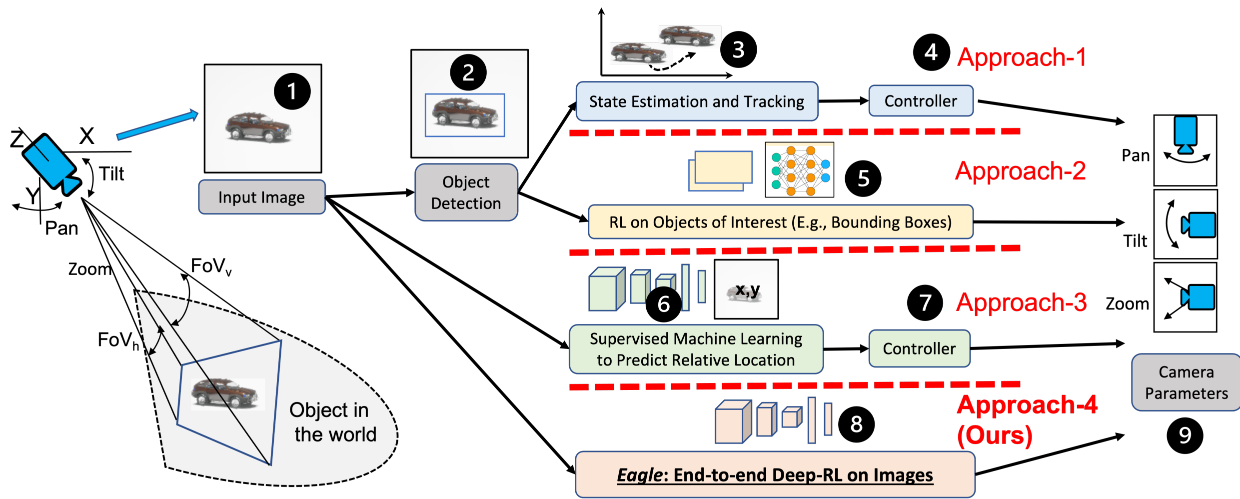


Figure 5.2: Different approaches for autonomous control of PTZ cameras illustrated using a vehicle tracking scenario. A PTZ camera is controlled to keep a car in the field-of-view (FoV). The horizontal FoV (FoV_h) and vertical FoV (FoV_v) control zoom parameter. Approach-1 (*Object-detection+tracking+control*): Represented by 1,2,3,4,9 is the widely used multi-stage technique of identifying objects (using object detectors), followed by a short term tracker and a controller. Approach-2 *Object-detection+RL*: Given by 1,2,5,9 shows a setting where the bounding boxes are used to train a RL policy. Approach-3 *Relative-location+control*: Steps 1,6,7,9 shows an alternative to bounding boxes where a neural network predicts the relative location of objects that the controller uses. Approach-4 *Eagle: End-to-end deep-RL*: Steps 1,8,9 show the proposed Eagle approach to directly control the pan, tilt, and zoom parameters using the raw input images.

scene variations and includes software abstractions to enable the training of the proposed Eagle approach in an end-to-end fashion.

- We compare Eagle with existing state-of-the-art methods to show its superior performance. We evaluate the generalization of Eagle across different object/surrounding variations. We further show that it is possible to modify the tracking goal of Eagle policies at runtime based on application needs.
- Finally, we show the direct transfer of Eagle trained purely in EagleSim simulator to the real videos.

5.2 Background and Related Works

5.2.1 Autonomous Control of PTZ Cameras

Our objective is to keep one or more objects of interest in field-of-view (FoV) of the PTZ camera at high resolution. An autonomous controller achieves this by controlling the pan and tilt to keep the desired object in the center of the captured image and zooming without clipping the object. It is desired to keep an object in the center of the image to avoid target loss during sudden movement/direction changes. Figure 5.2 shows a sample object of interest (car) and a PTZ camera to track the car. The configuration of pan, tilt and zoom decide if the car will be captured in the image.

Figure 5.2 represents the different steps in autonomous PTZ control for four classes of approaches. We compare the recently proposed approaches that use learning-based components for active tracking. Luo et al. [LSZ19] have shown that neural network-based active trackers outperform the traditional tracker like MIL [BYB09], Meanshift [CRM00] and KCF [HCM14]. Each approach is represented from the input image ❶ to the control of camera parameters ❾.

Object_detection+tracking+control [BVS07, UNC19, LMC21]: Steps ❶❷❸❹❾. The objects of interest are detected in the image, followed by a short-term tracking algorithm to predict their location in the future frame. The controller adjusts the pan-tilt-zoom to track the objects of interest.

Bernardin et al. [BVS07] focus on human targets. They use a face detector followed by a mean-shift tracker and a fuzzy controller in combination with expert knowledge to control the PTZ of the camera. Unlu et al. [UNC19] present the control of a PTZ camera for UAV tracking where a neural network (ResNet) based object detector is used for UAV identification, followed by a short-term tracker to estimate the location of the bounding box in the future frame and three PID-controllers for adjusting the PTZ parameters. Lopez et al. [LMC21] track less frequent objects using a PTZ camera with a faster R-CNN object detector. The object occurrence probability identifies the less frequent objects of interest, and a rule-based controller modifies the PTZ parameters to focus on the object of interest. The usage of neural network object detectors makes it infeasible to

deploy this multi-stage pipeline on embedded camera platforms [Kyr21]. Multi-stage information flow necessitates fine-tuning of each stage, and its performance is impacted by errors in each stage [Kyr21, LSZ19]. E.g., the object detector's errors impact the tracking performance [Kyr21].

Object_detection+reinforcement learning [BXD20, RET14, KKP19]: Steps ❶❷❺❾. These methods combine the tracking and control stages with a neural network policy trained using deep-RL on the detected object locations [KKP19]. Another variation of this approach is that instead of using object detectors' bounding boxes, researchers [RET14] assumes the availability of the region of interest in the image. Bisagno et al. [BXD20], Rudolph et al. [RET14], and Kim et al. [KKP19] show the control of PTZ camera using deep-RL where the inputs to the neural networks are the information about the object of interest (e.g., bounding-boxes, location of pedestrians). An actual deployment may need to use external object detectors to measure these inputs where the performance suffers from object detector errors and demands more compute resources due to the complexity of detectors.

Relative_location+control [Kyr21, PL19, HG19]: Steps ❶❻❼❾. A neural network is trained using supervised machine learning to output the relative location of the object of interest in the captured image. This relative location is used to control the pan-tilt-zoom parameters without requiring an explicit object detector. However, training the relative locations from images demands the availability of labeled bounding boxes in videos which are difficult to get for arbitrary deployments. Further, a separate controller that uses the relative locations needs to be fine-tuned for the specific scenario and camera parameters to make an end-to-end system.

Eagle: Steps ❶❽❾. We propose Eagle, an end-to-end deep-RL approach that directly uses the raw input images to control the PTZ parameters of a camera. Luo et al. [LSZ19] propose end-to-end deep-RL for first-person tracking, where the first-person observer moves along the object to track. In contrast, we study end-to-end deep-RL to control PTZ cameras where the location of the camera is fixed, but its PTZ parameters are modified to keep an object of interest in the field-of-view. Eagle removes the need to develop and tune the multiple stages and provides superior PTZ tracking performance. Further, the policies trained by Eagle are very lightweight, enabling their

real-time deployment on embedded devices.

5.2.2 Frameworks for Pan-Tilt-Zoom Cameras

Due to the difficulty of creating PTZ tracking scenarios in the real world, researchers have proposed several PTZ frameworks. Chen et al. [CSB15] propose a framework where a virtual PTZ camera is controlled to generate images from panoramic videos. However, this framework depends on the human annotation of videos for ground truth. Further, creating new scenarios requires manual video capture using specialized spherical cameras. Hanoun et al. [HZL17] propose a framework to study PTZ camera placement using a CAD environment. However, they assume that the objects of interest are available (similar to the object detectors), and it is not a photo-realistic environment. Salvagnini et al. [SCD11] propose a framework by placing a real PTZ camera and a calibrated projector screen. However, this requires specialized equipment, and the spherical screen limits the PTZ camera motion. Hamesse et al. [HPL21] propose a PTZ tracker for air traffic control using Unreal Engine. The simulator doesn't keep track of ground truth annotation; instead, it uses external object detectors and achieves only 3 FPS on a GPU server used by authors. The proposed simulator [HPL21] lacks rich scene variations or control of objects in the scenes. It is also unclear how to use existing simulation frameworks for end-to-end deep-RL due to a lack of capability to learn from trial and error, requiring control of objects in the scenes and automatic ground truth annotations for reward calculations.

EagleSim: We introduce the EagleSim simulator to train end-to-end deep-RL policies for PTZ control. EagleSim provides software abstractions, built using Unreal Engine [unr22] and AirSim [SDL18], for PTZ camera placement and control in photo-realistic virtual worlds. EagleSim relieves the need to create real-world tracking scenarios and automatically provides ground truth annotations of objects of interest. These ground truth annotations are perfect, unlike human labels or pre-trained object detectors, which may be noisy. We address the challenging simulation-to-reality gap in EagleSim by including virtual words with a wide variety of scenarios. EagleSim includes packaged virtual worlds with multiple objects (30 types of vehicles and 9 types of human

characters), surroundings (25 types of background materials/patterns and 10 types of trees), and image augmentations to support rich variations in tracking scenarios. These scene variations and control of movements of other objects (trees, background materials, and human characters) are not supported in the virtual worlds available from AirSim developers. Airsim API allows control of specialized vehicles (a blue car and a drone) only and offer no abstractions to modify surrounding scene objects (such as trees, human characters, other vehicles, and background patterns). EagleSim provides the capability to control scene objects which are needed for trial and error learning of PTZ tracker using deep-RL. We show that included scene variations are essential to train a generalizable policy that can work with variations in objects/surroundings and is directly transferable from simulation to real videos. EagleSim include capability to simulate pan, tilt, and zoom on the real videos. EagleSim also enables parallel scenes supporting >200 FPS to speed up the learning process from 17 days to 2.9 days on a GPU machine (GeForce RTX 3090 Ti).

5.3 Eagle: End-to-end Deep-RL for PTZ

Eagle trains lightweight neural network policies that map input images directly to pan, tilt, and zoom actions to track an object of interest in a scene. We consider a standard Markov decision process where an agent learns from trial and error by interacting with an environment over many discrete time steps. At each step, the agent receives a scalar reward defining its performance on the task. The agent uses the current state of the environment to decide the action. The reward measures the tracking performance.

5.3.1 State Space, Policy Network and Actions

The current state is represented by the most recent image captured from the PTZ camera. The Agent uses the image at every step to decide the next action, where the step length is determined by the video frame rate. We use lightweight network architecture for our policy to enable real-time inference on resource-constrained devices. We use a discrete action space that modifies the current

values of pan, tilt, and zoom parameters. The implementation-specific details of the input size, state transition delays, network architecture, and action space are discussed in Section 5.5.2.

5.3.2 Reward Function: Single Object

First, we formalize the reward function to track a single object of interest in a scene when no other objects are present. This is our simplest setting. Later, we generalize to handle the presence of other objects.

Consider an object O^a present in the scene. For active tracking, we formulate a reward function to keep O^a near the center of the image with maximum possible resolution. Consider a scenario in Figure 5.2, where a PTZ camera tracks a car as the object of interest O^a. A sample image of O^a captured by the PTZ camera is shown in Figure 5.3. The bounding box of the car represented by $[Xmin, Ymin, Xmax, Ymax]$, image $Height$, and image $Width$. The center coordinate (x,y) of the bounding box is defined as $[x,y] = [(Xmin+Xmax)/2, (Ymin+Ymax)/2]$.

Knowing the bounding box and its center coordinate for O^a, we define the reward (r_i^a) for step i in Equation 5.1.

$$r_i^a = \begin{cases} Center_x^a \times Center_y^a \times Obj_{size}^a \times Clip^a \text{ - P} & Condition \\ -L & Otherwise \end{cases} \quad (5.1)$$

Where *Condition* is a binary value. For a single object setting, the *Condition* is given by l^a. Here, $l^a = 1$, if O^a is captured in the image, else $l^a = 0$, if O^a is not captured. When *Condition* is True, the reward is a multiplication of four terms $Center_x^a$, $Center_y^a$, Obj_{size}^a, and $Clip^a$ after penalty P is subtracted. $P > 0$ is a hyper-parameter which slightly penalizes the agent on modifying the camera's PTZ parameters to avoid the jittery behavior. When the object O^a is not present in the captured images, the agent receives a negative reward L, where $L > 0$ is a hyper-parameter.

$$Center_x^a = \frac{abs\left(\frac{Width}{2} - x\right)}{\frac{Width}{2}} \tag{5.2}$$

$$Center_y^a = \frac{abs\left(\frac{Height}{2} - y\right)}{\frac{Height}{2}} \tag{5.3}$$

$$Obj_{size}^a = \frac{(Xmax - Xmin) \times (Ymax - Ymin)}{Width \times Height} \tag{5.4}$$

$$Clip^a = \begin{cases} M & \text{if } Xmin = -(Height/2) \text{ or } Ymin = -(Width/2) \\ M & \text{if } Xmax = (Height/2) \text{ or } Ymax = (Width/2) \\ 1 & \text{otherwise} \end{cases} \tag{5.5}$$

$$R^a = \sum_{i=0}^{N} (r_i^a) \tag{5.6}$$

$Center_x^a$, $Center_y^a$ and Obj_{size}^a are defined in Equations 5.2, 5.3, 5.4. $Center_x^a$ measures the accuracy with which the agent can ensure the object O^a is centered on the X-axis of the image. Similarly, $Center_y$ measures accuracy along the Y-axis. If the target is not close to the center of the image, its probability of leaving FoV is high on sudden movements or its direction changes.

Obj_{size}^a measures the relative size of O^a in the image. $Center_x^a, Center_y^a, Obj_{size}^a \in [0,1]$. $Clip^a$ is defined in Equation 5.5, where $M \in (0,1)$ is a hyper-parameter. $Clip^a$ penalizes the reward r_i^a when O^a is clipped in the captured image. The total reward R^a for each episode of N steps is the summation of step rewards r_i^a as shown in Equation 5.6.

5.3.3 Generalizable PTZ Tracking

It is common to have multiple objects in a tracking scene. For example, a vehicle tracking scenario can have other objects like trees, humans, and background patterns/buildings. Here, we extend the reward function to complex scenes by modifying the *Condition* definition in Equation 5.1.

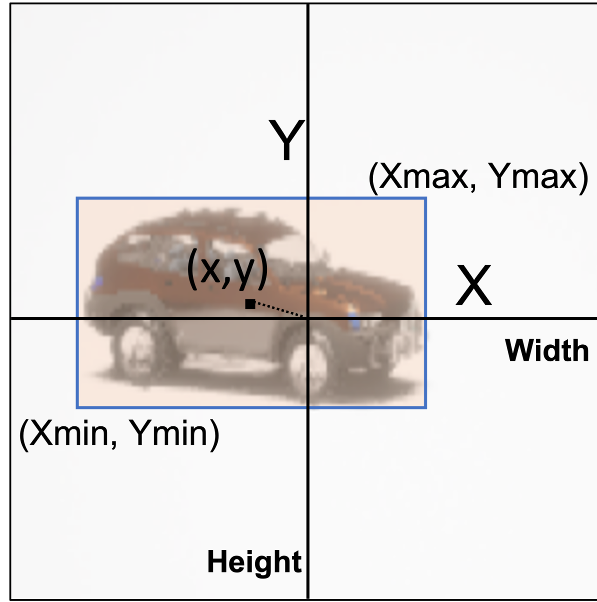


Figure 5.3: A sample bounding box for the object of interest (car). The center of the image is the origin (0,0). (x,y) is the center of the bounding box $(Xmin, Ymin, Xmax, Ymax)$.

Consider 2 classes of objects in a scene. The first class $A = \{O^{a1}, ..., O^{aT}\}$ is a collection of objects that we are interested to track. For example, class $A = \{SUV_{blue}, SUV_{red}, Sports_{red}, Sports_{grey}, Pickup_{red}, Pickup_{grey}\}$ is representing different vehicles that we are interested to track in a vehicle tracking scenario. The second class $B = \{O^{b1}, ..., O^{bU}\}$ is the collection of objects to be ignored. For example, the objects in the class B may refer to background buildings, trees, and human characters for a vehicle tracking scenario. We assume that only one of the objects from class A is present at a given time in the scene, while the same policy generalizes to all objects of class A. For example, the same policy can track SUV_{blue}, $Sports_{red}$ or a $Pickup_{grey}$ vehicle, but only one of them is present in the scene. The agent is given a reward only when the objects of class A are captured in the image. We expressed this by modifying the *Condition*, which is True when $(l^a = 1) \wedge$ (a $\in$ A).

We update the calculation of reward (r_i^a) (Equation 5.1) using this new *Condition*. This incentivizes an agent to track objects from the class A only. The approach to generalize is called domain randomization (or environment augmentations) [LSZ19, BMG20]. The discussion also apply to

other scenarios such as tracking human characters. Here, the variations of human characters are added to class A, and class B contain objects to be ignored in the scene.

When multiple objects of class A are present in the same scene, we observe that the policy trained using Equation 5.1 is incentivized to track an object giving better future rewards. Future rewards depend on the relative size of objects in the initial image and their center locations, as discussed in Section 5.6. We introduce the idea of dynamic tasking next to enable selective tracking of objects.

5.3.4 Dynamic Tasking of Eagle Policies

We define dynamic tasking as the capability to change the tracking goal at deployment. More specifically, the same policy can be tasked to track a specific object from class A during deployment. This is particularly important when multiple objects belonging to the class A are present in the same scene.

Dynamic tasking in existing multi-stage approaches is enabled by changing the object detection stage. The identified objects from the object detectors are filtered during deployment to change the tracking behavior [MRF10]. However, in Eagle, there is no separate object detector. To allow dynamic tasking, we modify the training of Eagle by including an extra *contextual input* along with the current PTZ camera image and do the reward shaping [Grz17].

We use structured contextual input in the integer space to specify different sub-class of objects in class A. To simplify notation, we explain dynamic tasking with an example of two sub-classes in class $A = \{SUV_{blue}, SUV_{red}, Human_1, Human_2\}$. The first sub-class of vehicles $A_v = \{SUV_{blue}, SUV_{red}\}$ and the second sub-class of human characters $A_h = \{Human_1, Human_2\}$. The contextual input $CI \in \{0, 1\}$ represents two integer values. The formulation generalizes to more complex contextual inputs at the expense of increased training time. Our goal is to task policy at runtime to either track sub-class A_v when $CI = 0$ or sub-class A_h when $CI = 1$. We define a new *Condition* in the Equation 5.7 to allow this.

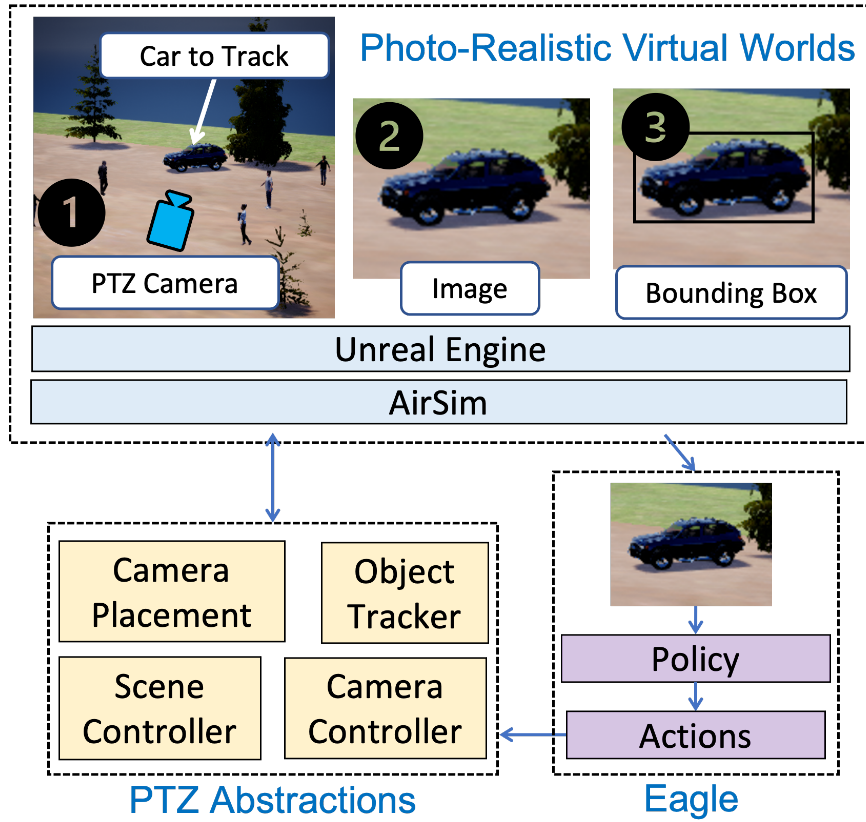


Figure 5.4: The architecture of EagleSim and its integration with Eagle. Step-1 shows placement of a PTZ camera for vehicle tracking. Step-2 shows an image captured by camera. Step-3 shows a bounding box for the object of interest (car).

$$Condition = \begin{cases} True & C_v \vee C_h \\ False & \text{otherwise} \end{cases} \tag{5.7}$$

Where C_v is True when $(l^a = 1) \wedge (\text{a} \in A_v) \wedge (CI = 0)$ and C_h is True when $(l^a = 1) \wedge (\text{a} \in A_h) \wedge (CI = 1)$. The *Condition* in the Equation 5.7 brings in the domain knowledge to ensure that the agent can learn to associate a specific contextual input with a particular sub-class of objects of interest.

5.4 Design of EagleSim

To train Eagle in an end-to-end fashion, we need to set up online tracking scenarios where an agent can learn from trial and error. Deep-RL training in the real world is difficult due to large number of environment interactions, labeling effort for ground truth annotations, and fragile experimental setups [BMG20, PAZ18, ABC20]. Further, the creation of PTZ tracking scenarios with vehicles, humans, and background variations is not trivial in the real world. To enable training of Eagle, we introduce a new simulator called EagleSim. Next, we discuss the architecture of EagleSim shown in Figure 5.4.

5.4.1 Photo-Realistic Virtual Worlds

EagleSim can be integrated with arbitrary virtual worlds created in the Unreal engine and precompiled virtual world binaries available for Airsim. With these integrations, developers can simulate PTZ cameras in static worlds. Although AirSim enables control of a drone and a car in virtual worlds, it doesn't support different types of vehicles, control over human characters, background trees, boundaries, and surroundings. These scene variations are needed to create a rich tracking scenario and to generalize Eagle policies. Creating rich scene variations is non-trivial, and requires a significant engineering effort. To address this, we design new virtual worlds (4300 lines of c++ code in Unreal engine) supporting rich variations and include them with EagleSim.

Packaged virtual worlds in EagleSim: We package virtual worlds for vehicle tracking and human tracking scenarios. The vehicle tracking virtual world keeps track of ground truth annotations of vehicles, whereas the human tracking world keeps track of human characters. Each virtual world supports 6 types of vehicles (SUV1, Pickup, Sports, HatchBack, SUV2, Truck) in five colors (30 different vehicles), 6 human characters, 25 background materials/patterns, and 10 types of trees. EagleSim also supports image augmentations to add random shadows, salt-pepper noises, random contrast and brightness changes to the PTZ images.

Each virtual world represents an open space (70 meters $\times$ 70 meters) with boundaries. The

boundaries can be made invisible, creating a simple scene with a blue skyline (see Sc-1 setting in Figure 5.5). The background patterns can be applied to the floor and the boundaries to create variations of urban, forest, and rural areas, as shown in Figure 5.1. Vehicles and humans can move freely within the open space. The different placement of objects, their movements, and scene variations allows the creation of endless tracking scenarios. A sample vehicle and human tracking scenes are shown in Figure 5.1 and Figure 5.5. The speed and steering of vehicles are controlled using AirSim API. We implemented abstractions directly using the Unreal engine to control human characters, background patterns, and tree placements. The image augmentations are implemented using python functions.

Addressing simulation-to-reality gap: A major challenge with policies trained in a simulator is to make them transferable to the real world. The research community represents this as the simulation-to-reality gap [PAZ18, BMG20] due to sensing and dynamics differences between the simulators and the real world. For example, the images in the real world can have observations (such as object variations, backgrounds, shadows, occlusions, and illuminations) never observed in the static/simple simulations.

The virtual worlds packaged with EagleSim are carefully designed to address the simulation-to-reality gap. We use domain randomization to train a policy that can generalize to unseen variations of objects and surroundings. The idea is to train policies on a combination of different scene variations. The hypothesis is that the real world lies in one of the training variations.

5.4.2 PTZ Abstractions

The PTZ abstractions in EagleSim provide four modular components. Figure 5.4 shows a sample vehicle tracking scene. A PTZ camera in step ❶ is placed using *Camera Placement* component. The camera is controlled using the *Camera Controller*. ❷ shows a sample camera image. The car may or may not get captured in the image depending on the camera location, car's location, and PTZ parameters. The bounding box (❸) is captured by the *Object Tracker*.

Camera Placement places the PTZ camera at any arbitrary location in the scene. We use AirSim API to anchor the camera to a vehicle. For each camera in the virtual world, a new vehicle is spawned. The relative location of the camera and vehicle is controllable, allowing the vehicle to be placed in the far area of the scene without interfering with the control of the PTZ camera.

Camera Controller exposes control of pan, tilt, and zoom parameters of the cameras at runtime. The resolution supported for pan and tilt is one degree. The zoom is controlled by modifying the horizontal field-of-view (FoV) at runtime with a resolution of 1 degree. The vertical FoV is based on the aspect ratio and the horizontal FoV: $vertical_{FoV} = horizontal_{FoV} * height/width$.

Object Tracker provides bounding boxes of objects of interest captured in the PTZ image. A bounding box is calculated by transforming the 3-D outer coordinate mesh of the object available from the Unreal engine to the 2-D image coordinates using a pinhole camera model. We implemented this using the object detector abstractions from AirSim. The bounding boxes of EagleSim are perfect, unlike the bounding boxes predicted by a neural network-based object detector, which may have prediction errors. The bounding boxes are used to calculate rewards (Section 5.3) to train Eagle.

Scene Controller selects scene variations to enable the evolution of the training environment by integrating the capturing of images from PTZ cameras, controlling the pan, tilt, and zoom parameters and exposing rewards (bounding boxes) of PTZ tracking.

5.5 Evaluation

First, we discuss the performance metrics for PTZ tracking. Then, we evaluate the performance and generalizability of Eagle policies as the complexity of the PTZ tracking task is increased gradually. Later, we compare Eagle with the current state-of-the-art, showcase its transfer to real-scene videos, and measure its deployment delays on embedded platforms.

Table 5.1: Policy architecture used by Eagle.

Layer	1	2	3	4	5	6	7	8
Config	C5x5-64	C3x3-32	C3x3-32	C3x3-16	64	64	64	3,3,3

5.5.1 Performance Metrics for PTZ Tracking

The end-to-end approach doesn't produce intermediate bounding boxes, which are generally available in multi-stage approaches. So it is not possible to compare the detection performance with standard object detection metrics. We adopt metrics (% Tracking, $Center_x$, $Center_y$, Obj_{size}) to evaluate the camera control performance directly. The metric of tracking duration (referred to as % Tracking) [Kyr21] measures the duration for which the controller successfully keeps an object of interest in the FOV of the camera. When % Tracking is 100, the object was kept in the camera's FoV for the entire duration. Tracking duration is also equivalent to the episode length adopted by Luo et al. [LSZ19]. Instead of the number of steps, we report the percentage. Like the tracking duration, Chen et al. [CSB15] uses *track fragmentation* which is the number of steps as a fraction between 0 and 1.

One of the goals of the PTZ controller is to capture an object in the center of the image to avoid its loss on sudden movements or direction changes. To measure center location error, we directly use the average $Center_x$ (Equation 5.2) and average $Center_y$ (Equation 5.3) maintained by controller for the entire trajectory. The metrics of $Center_x$ and $Center_y$ are equivalent to the center location error used by Chen et al. [CSB15] to evaluate PTZ controller performance. To compare the resolution of the object in the captured image, we adopt Obj_{size} (Equation 5.4), which measures the relative size of an object in the captured image.

Table 5.2: Different tracking scenarios in the increasing order of tracking complexity to evaluate Eagle. The goal of Sc-1 to Sc-2 is to track vehicles. Dynamic tasking (DT) trains a policy to track either a vehicle or human characters.

Scenario	**Tracking Goal**	**Scene Variations**
Sc-1	$SUV1_{blue}$	Fixed background
Sc-2	$SUV1_{blue}$	Fixed background+Trees+ Image augmentations
Sc-3	$SUV1_{blue}$	Variable backgrounds+Trees+ Image augmentations
Sc-4	$SUV1_{blue} + SUV1_{red}$ $+SUV1_{grey}$	Variable backgrounds+Trees+ Image augmentations+Humans
Sc-5	$SUV1_{blue} + SUV1_{red}+$ $Pickup_{grey} + Pickup_{red}+$ $Sports_{blue} + Sports_{grey}$	Variable backgrounds+Trees+ Image augmentations+Humans
Dynamic Tasking	$SUV1_{blue}/Humans$	Variable backgrounds+Trees+ Image augmentations

Figure 5.5: Visualization of scenarios shown in Table 5.2. Sc-1 has a fixed background. In Sc-2, fixed background is extended with random tree placements and image augmentations. Sc-3 shows background variations with image augmentations and trees. Sc-4 extends the Sc-3 scenes with humans and same vehicle with different colors. In, Sc-5, we add vehicles of different types as well. DT shows the scenes for dynamic tasking of policy to track human characters.

5.5.2 Implementation of Eagle

State Space, Policy Network and Actions: We downsample the PTZ camera images to 120×120 and convert them to grayscale. This reduces the input dimensionality as working directly with large color images is computationally demanding [MKS15, Kyr21]. The policy network architecture is shown in Table 5.1, which consists of 7 hidden layers and an output layer. This network architecture is motivated from neural networks trained for Atari games by Mnih et al. [MKS15] The first five layers are convolution layers, each with a stride 2 and a rectifier nonlinearity. The first layer denoted by C5x5-64 convolves 64 filters of 5 × 5 each. The fifth, sixth, and seventh layers are fully connected and consist of 64 rectifier units each. The last layer produces 3 discrete outputs to control each of the pan, tilt, and zoom parameters. The policy network is designed to be lightweight to enable real-time inference of Eagle on Raspberry PI 4B and Jetson Nano devices and has a total of 79k model parameters.

The 3 outputs modify the current parameter values as follows: [-2 degrees, 0 degrees, 2 degrees] for pan and tilt. We control the FoV with three possible outputs for the zoom configuration: [-1 degree, 0 degree, 1 degree]. The horizontal FoV and vertical FoV are equal in our setting due to the same aspect ratio of input images. We use the ground truth bounding boxes from EagleSim to calculate the episodic reward (Equation 5.6) and use $L = 10$, $M = 0.3$, and $Penalty = 0.01$ as the hyperparameters.

State Transitions: During training, we maintain an end-to-end delay from sensing a state (image) to action at 30 milliseconds. This delay is matched to the embedded PTZ camera platform [Ard22] when deploying Eagle policy using Raspberry PI 4B. We use an episode length of 2000 steps during training which translates to one minute of continuous tracking.

Distributed Training: EagleSim supports the creation of multiple scenes in parallel, giving control of each scene to the developer. The reward calculations from each scene are exposed using Python wrappers in the OpenAI Gym [BCP16] format. This allows the integration of EagleSim with state-of-the-art reinforcement learning libraries. We use 6 parallel scenes to train a single policy in 69

Table 5.3: Evaluation of Eagle policies trained for different scenarios in *Fixed background* scene. Sc-1 to Sc-5 are the vehicle tracking scenarios shown in Table 5.2.

	Fixed background			
	%Tracking	$Center_x$	$Center_y$	Obj_{size}
Sc-1	99.6 ± 4.3	0.87 ± 0.1	0.85 ± 0.1	0.31 ± 0.1
Sc-2	99.9 ± 0.1	0.87 ± 0.1	0.86 ± 0.1	0.30 ± 0.1
Sc-3	99.1 ± 8.5	0.87 ± 0.1	0.86 ± 0.1	0.30 ± 0.1
Sc-4	99.7 ± 4.3	0.87 ± 0.1	0.86 ± 0.1	0.29 ± 0.1
Sc-5	99.7 ± 5.4	0.86 ± 0.1	0.85 ± 0.1	0.27 ± 0.1

hours (2.9 days) on a GPU server machine (GeForce RTX 3090 Ti) [Bal21] to track vehicles (see Figure 5.6). Without 6 parallel scenes, it would take 17 days (2.9*6) for an agent to collect the same amount of data from a sequential environment.

Training Algorithm: We train Eagle using the distributed implementation of Proximal Policy Optimization (PPO) [SWD17] algorithm from the stable-baselines3 [RHE19] library. The default hyperparameters of PPO present in the library are used, except the updates to the network are performed every 24576 (4096*6) steps collected from 6 parallel environments and a batch size of 256.

5.5.3 Tracking Scenarios

We consider five vehicle tracking scenarios (Sc-1 to Sc-5) with increasing tracking complexity and a dynamic tasking (DT) scenario, as shown in Table 5.2. The scenes from tracking scenarios are shown in Figure 5.5. Sc-1 tracks a single $SUV1_{blue}$ car in a *Fixed background*. In Sc-5, the agent tracks any of the 6 vehicles ($SUV1_{blue}$, $SUV1_{red}$, $Pickup_{grey}$, $Pickup_{red}$, $Sports_{blue}$, $Sports_{grey}$) in the presence of *variable backgrounds*, *tree*, *image augmentations* and *human characters*. *Variable backgrounds* refers to the random selection of background materials (any one of 25 materials

Table 5.4: Evaluation of Eagle policies trained for different scenarios in *Variable backgrounds+Trees* scenes. Sc-1 to Sc-5 are the vehicle tracking scenarios shown in Table 5.2.

	Variable backgrounds+Trees			
	%Tracking	$Center_x$	$Center_y$	Obj_{size}
Sc-1	9.7 ± 6.3	0.65 ± 0.2	0.63 ± 0.2	0.13 ± 0.2
Sc-2	18.8 ± 16.2	0.81 ± 0.2	0.79 ± 0.2	0.29 ± 0.2
Sc-3	98.9 ± 6.9	0.87 ± 0.1	0.86 ± 0.1	0.30 ± 0.1
Sc-4	98.9 ± 7.4	0.86 ± 0.1	0.86 ± 0.1	0.29 ± 0.1
Sc-5	99.3 ± 6.9	0.86 ± 0.1	0.86 ± 0.1	0.27 ± 0.1

Table 5.5: Evaluation of Eagle policies trained for different scenarios in *Variable backgrounds+Trees+Humans* scenes. Sc-1 to Sc-5 are the vehicle tracking scenarios shown in Table 5.2.

	Variable backgrounds+Trees+Humans			
	%Tracking	$Center_x$	$Center_y$	Obj_{size}
Sc-1	8.5 ± 4.4	0.64 ± 0.2	0.62 ± 0.2	0.08 ± 0.1
Sc-2	17.2 ± 14.8	0.81 ± 0.2	0.77 ± 0.2	0.27 ± 0.2
Sc-3	90.0 ± 22.1	0.86 ± 0.1	0.85 ± 0.1	0.31 ± 0.1
Sc-4	99.1 ± 6.3	0.86 ± 0.1	0.86 ± 0.1	0.29 ± 0.1
Sc-5	99.2 ± 4.6	0.86 ± 0.1	0.85 ± 0.1	0.27 ± 0.1

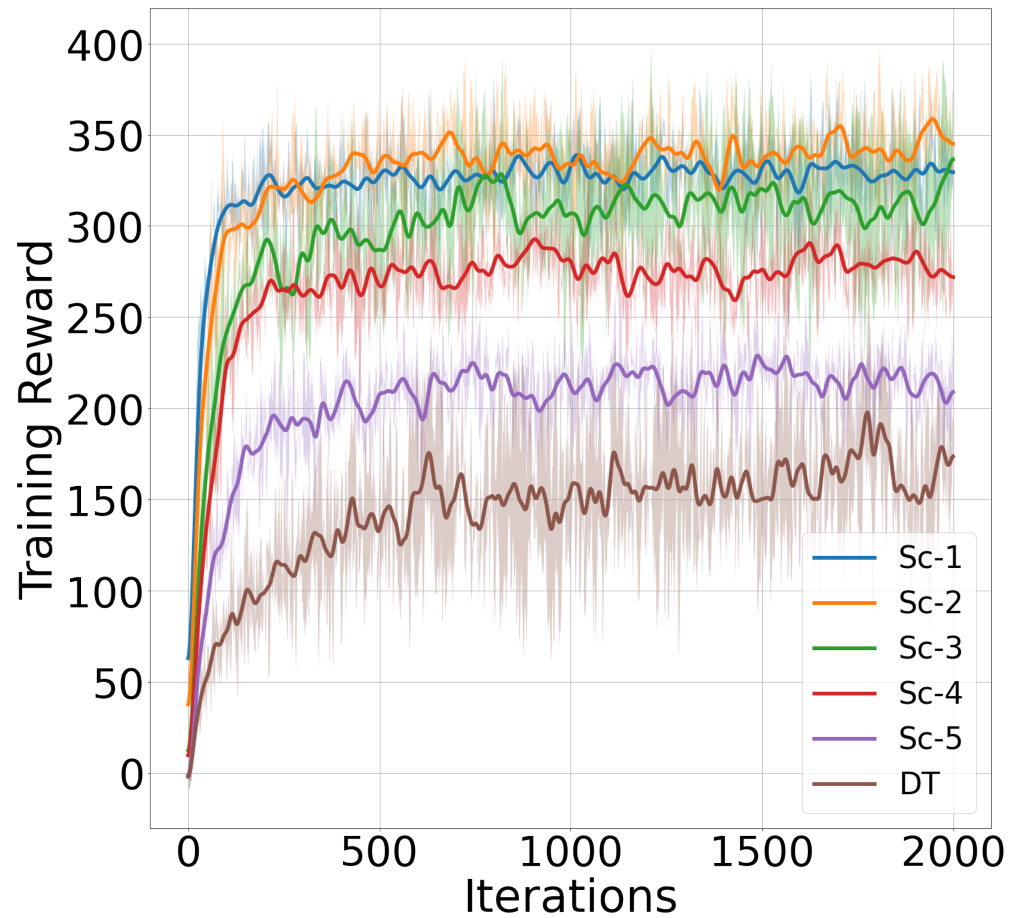


Figure 5.6: Average training reward of Eagle policies for scenarios shown in Table 5.2. We calculate average reward by training three policies for each scenario and show its min-max spread. Each policy is trained for 69 hours (2000 iterations).

Table 5.6: Generalization of Eagle policies trained for Sc-4 and Sc-5 (see Table 5.2) scenarios to track $HatchBack_{green}$ and $Truck_{blue}$ vehicles which were not present during training.

	$HatchBack_{green}$ **Vehicle**			
	%Tracking	$Center_x$	$Center_y$	Obj_{size}
Sc-4	83.0 ± 28.2	0.85 ± 0.1	0.86 ± 0.1	0.21 ± 0.1
Sc-5	88.1 ± 25.4	0.84 ± 0.1	0.85 ± 0.1	0.22 ± 0.1
	$Truck_{blue}$ **Vehicle**			
	%Tracking	$Center_x$	$Center_y$	Obj_{size}
Sc-4	92.8 ± 19.4	0.87 ± 0.1	0.82 ± 0.1	0.27 ± 0.1
Sc-5	95.1 ± 16.3	0.84 ± 0.1	0.83 ± 0.1	0.34 ± 0.1

included in EagleSim) during different training episodes. Various trees and human characters are randomly placed in the scene when enabled. Dynamic tasking (DT) trains a policy to track objects from one of the two sub-classes based on the contextual input. The first sub-class contains a single vehicle ($SUV1_{blue}$), and the second sub-class contains 4 different human characters.

Tracking setup: During training and evaluation, the vehicles are given random trajectories by controlling steering and throttle in the virtual world in an open space of 70 meters $\times$ 70 meters. A vehicle has an average speed of 6 m/s. The vehicle comes to a standstill when reaching the boundary and randomly changes its direction, and has a max speed of 16 m/s. A PTZ camera is placed at the mid-point of the south (or bottom) boundary at the height of 8 meters as shown in Figure 5.4 (step ❶). The camera initially looks at the wider scene, capturing a zoomed-out image with a vehicle in it. As the vehicle moves, the goal is to track and focus on it. This tracking setup is shown for different scenes in Figure 5.5 (Sc-1 to Sc-4), where the initial image shows a wider view of the scene, and subsequently, the PTZ camera tracks the car. With the tracking progress, the PTZ camera focuses more on the object of interest, as seen in the right images of Figure 5.5. While tracking the vehicle, variations are enabled in specific scenarios, as seen in Figure 5.5. For

Table 5.7: The dynamic tasking (DT) performance of Eagle policies to track either a humans character (DT_h) or a vehicle (DT_v).

	Fixed background			
	%Tracking	$Center_x$	$Center_y$	Obj_{size}
DT_v	98.8 ± 7.6	0.86 ± 0.1	0.85 ± 0.1	0.30 ± 0.1
DT_h	95.8 ± 15.2	0.90 ± 0.1	0.90 ± 0.1	0.27 ± 0.1
	Variable backgrounds+Trees			
	%Tracking	$Center_x$	$Center_y$	Obj_{size}
DT_v	92.0 ± 18.7	0.85 ± 0.1	0.84 ± 0.1	0.30 ± 0.1
DT_h	86.3 ± 22.7	0.89 ± 0.1	0.90 ± 0.1	0.28 ± 0.1
	Variable backgrounds+Trees+Humans			
	%Tracking	$Center_x$	$Center_y$	Obj_{size}
DT_v	88.7 ± 21.4	0.85 ± 0.1	0.84 ± 0.1	0.29 ± 0.1
DT_h	83.6 ± 23.7	0.89 ± 0.1	0.90 ± 0.1	0.28 ± 0.1

the dynamic tasking (DT) scenario, a human character is placed in the scene along with the vehicle ($SUV1_{blue}$), where both are initially visible to the wider view of the PTZ camera as shown in Figure 5.5: DT. Both human and vehicle move on random trajectories. A random human character (out of 4) is selected during each training episode.

Training reward: The training reward for different scenarios is shown in Figure 5.6. The policies are trained using 6 parallel environments for 69 hours (2000 iterations). The training reward is calculated as an average of three policies for each scenario. As seen, the simpler tracking scenario converges faster. We also see with the increasing complexity of scenarios, Eagle policies converge to a lower training reward. Next, we analyze the performance of policies trained for different scenarios to understand this behavior.

Table 5.8: %Tracking of Eagle policies at different heights (meters) of the PTZ camera.

Height	20m	15m	10m	8m	5m	4m
%Tracking	96.5 ± 15	99.7 ± 2.4	99.7 ± 3.1	99.7 ± 5.4	99.3 ± 6.6	90.2 ± 23

5.5.3.1 Performance of Eagle Policies

We evaluate the policies using the checkpoint with the highest training reward. For each scenario, three policies with different random seeds are trained, and a checkpoint is evaluated from each policy. Each checkpoint is evaluated for 100 episodes (each episode is of 2000 steps or 1 minute of tracking). We report the mean performance metrics (Tracking duration (% Tracking), $Center_x$, $Center_y$, Obj_{size}) and their std.

Table 5.3, Table 5.4 and Table 5.5 show the performance of Eagle policies to track $SUV1_{blue}$ in scenes having *Fixed background*, *Variable backgrounds+Trees*, and *Variable backgrounds+Trees+Humans*. The $SUV1_{blue}$ vehicle is present during training of all scenarios (Sc-1 to Sc-5) (Table 5.2). The test settings are different from training due to the random placement of objects (backgrounds, trees, and human characters when enabled) and random vehicle/human trajectories.

Policy behavior in simple scenes and understanding training rewards: Looking at the *Fixed background* evaluation in Table 5.3, we see Eagle policies trained using all scenarios can successfully track >99% of the time and maintain similar $Center_x$ and $Center_y$ metrics. However, the policy zooms in conservatively on the object of interest when the training complexity of scenes is increased. The Obj_{size} maintained by policies decreases for Sc-1 to Sc-5 gradually. Our hypothesis is that the behavior to reduce zoom is learned so as to track different kinds of vehicles in the presence of other objects/occlusions and scene variations. Due to the reduction in Obj_{size}, the average training reward decreases from Sc-1 to Sc-5.

Generalization to unseen scene variations: Eagle policies for Sc-1 and Sc-2 are trained using a *Fixed background*. We see these policies doesn't work when tested under unseen scenes variations

(*Variable backgrounds+Trees* and *Variable backgrounds+Trees+Humans* as shown in Table 5.4) and Table 5.5). Adding *Image augmentations* in Sc-2 results in better performance in comparison to Sc-1 for unseen variations. During training, Sc-3 doesn't observe human characters; however, the policy can still maintain a tracking duration of 90% even in the presence of human characters. This shows variation in backgrounds is critical during training to have a generalizable tracking policy. We also see that Sc-4 and Sc-5 have similar behavior. This is because Sc-4 and Sc-5 are trained with variable backgrounds, trees, and human characters for $SUV1_{blue}$ vehicle. However, Sc-5 is also trained to track rich categories of vehicles, and next, we see how Sc-4 and Sc-5 generalize to unseen vehicle variations.

Generalization to unseen variations in object of interest: We analyze the performance of Sc-4 and Sc-5 policies to track vehicles not present during training. The performance is shown in Table 5.6 to track $HatchBack_{green}$ and $Truck_{blue}$ for a scene with *Variable backgrounds+Trees+Humans*. $HatchBack$ and $Truck$ vehicles were not present during training. Also, no green color vehicles were present during training. As seen, the policies trained for Sc-5 outperforms; this is because Sc-5 has different types of vehicles with multiple colors whereas Sc-4 has one type of vehicle with multiple colors. Hence, having variation in vehicle types generalizes better. Further, $HatchBack_{green}$ refers to a setting where neither the vehicle ($HatchBack$) nor the color (*green*) was present during training, which gives slightly worse performance than $Truck_{blue}$. *blue* color vehicles were present during the training of policies. Sc-5 maintains superior tracking duration (% Tracking) along with other metrics.

Dynamic Tasking Performance: Table 5.7 shows the performance of Eagle policies to track either $SUV1_{blue}$ (DT_v) or a $Human$ (DT_h) character. In *Fixed background* and *Variable backgrounds+Trees* only one of the object of interest ($SUV1_{blue}$ or a $Human$ character is present). In *Variable backgrounds+Trees+Object* both $SUV1_{blue}$ and $Humans$ are present in the scene and contextual input is use to decide the tracking goal. Dynamic tasking represents the most complex scenario in Table 5.2 as the agent needs to learn adaptation of its tracking goal across diverse objects (vehicle or humans). We also see this in the policy performance. In the absence of another

competing object, policy performs much better, whereas it suffers a performance degradation as measured by the tracking duration (% Tracking) in *Variable backgrounds+Trees+Object*. The dynamic tasking goal also has different complexity between the vehicle and human sub-class. The vehicle sub-class contains only a single vehicle ($SUV1_{blue}$), whereas the human sub-class contains 4 different characters. The average performance on 4 human characters is shown. We hypothesize that the imbalance in tracking complexity results in performance differences between DT_v and DT_h.

Camera height: Eagle policies are trained by placing a camera at 8 meters(m) height as discussed in Section 5.5.3 (Tracking setup). Table 5.8 shows the performance of Sc-5 policies trained at 8m by varying the height during evaluation. The %Tracking is captured across 100 episodes to track $SUV1_{blue}$ vehicle moving on random trajectories (avg speed of 6 m/s) in scenes having *Fixed Background*. The other metrics are similar across heights. The action space of Eagle modifies the current PTZ parameters (Section 5.5.2) and works well between 5m to 15m heights during deployment, even when trained at 8m. A significant performance drop happens when the height is below 5m or above 15m. This suggests with a substantial camera placement difference between training and deployment, policy retraining is needed for optimal performance.

Summary: As the PTZ tracking complexity is increased for vehicle tracking scenarios (Sc-1 to Sc-5), Eagle policies learn a more conservative behavior. The zoom-level maintained on the object of interest is reduced in complex scenes. Policies trained in simpler scenes don't transfer well to unseen backgrounds and cannot work in the presence of other objects like trees or humans. The trained policies for Sc-5 use complex scenes, which also perform better for unseen variations in the object of interest. This shows the complex scenes in EagleSim simulator are critical to developing generalizable policies. The evaluation of dynamic tasking showed that contextual input enables goal modification during the deployment; however, enabling dynamic tasking comes at a performance cost.

Table 5.9: Comparison of Eagle with the current state-of-the art approaches for different scene complexities.

Approach	**Fixed background**			
	%Tracking	$Center_x$	$Center_y$	Obj_{size}
Eagle	99.7 ± 5.4	0.86 ± 0.1	0.85 ± 0.1	0.27 ± 0.1
Yolo+Kalman+Controller	95.1 ± 18.2	0.81 ± 0.1	0.85 ± 0.1	0.24 ± 0.1
Yolo+Deep_RL	39.1 ± 18.9	0.81 ± 0.2	0.82 ± 0.2	0.19 ± 0.2
NN+Controller	81.2 ± 26.1	0.83 ± 0.1	0.82 ± 0.1	0.22 ± 0.1
Approach	**Variable backgrounds+Trees+Humans**			
	%Tracking	$Center_x$	$Center_y$	Obj_{size}
Eagle	99.2 ± 4.6	0.86 ± 0.1	0.85 ± 0.1	0.27 ± 0.1
Yolo+Kalman+Controller	75.4 ± 30.6	0.81 ± 0.1	0.84 ± 0.1	0.25 ± 0.1
Yolo+Deep_RL	36.8 ± 19.0	0.84 ± 0.2	0.82 ± 0.2	0.20 ± 0.2
NN+Controller	53.6 ± 31.5	0.86 ± 0.1	0.85 ± 0.1	0.23 ± 0.1

Table 5.10: Performance of multi-stage approaches when perfect bounding boxes are available from EagleSim simulator.

Approach	%Tracking	$Center_x$	$Center_y$	Obj_{size}
PerfectBB+ Deep_RL	98.5 ± 8.5	0.85 ± 0.1	0.81 ± 0.2	0.36 ± 0.1
PerfectBB+ Kalman+ Controller	98.3 ± 8.4	0.85 ± 0.1	0.83 ± 0.1	0.33 ± 0.1

5.5.4 Eagle vs Other Approaches

Here, we compare the performance of Eagle with the current state-of-the-art approaches. First, we discuss our realization of other approaches, and then present their control performance.

5.5.4.1 Object_detection+tracking+control

We use Yolo5s [yol22], a state-of-the-art lightweight object detector, followed by a Kalman filter for state estimation on bounding boxes. We use the open-source [BGO16] implementation of the SORT algorithm for the Kalman filter. Finally, the tracking outputs are used to control the PTZ parameters using a separate controller [LMC21]. The controller uses the centroid and the size of the target object to adjust the PTZ parameters. We fine-tune the thresholds of the controller using Mango [SAF20]. The input to the object detector is an image of size 240×240. We call this tracking setting, *Yolo+Kalman+Controller*. Yolo5s is trained to detect 80 different classes of objects [yol22]. To improve its performance on the specific $SUV1_{blue}$ vehicle used in the evaluation, we finetune Yolo5s by collecting a labeled dataset of 50k images from EagleSim. The dataset is collected by observing the $SUV1_{blue}$ vehicle in *Variable backgrounds+Trees+Humans* scenes and at varying zoom levels. The default hyperparameters recommended by Yolo5s developers [Ult22] were used to finetune the model for 100 epochs. We use the finetuned Yolo5s in our experiments. Finally, to completely remove object detector errors, we also implement another setting that uses the oracle bounding boxes provided by the EagleSim simulator as input to the Kalman filter. We call this setting, *PerfectBB+Kalman+Controller*, where PerfectBB signifies the perfect bounding boxes.

Kalman filter: We adopted the Kalman filter from the SORT algorithm [BGO16]. The state of the tracked object is modeled using seven variables. Four variables are used to represent the bounding box, two variables for the velocity (U, V) variables for the center of the bounding box, and one variable to track the changes in the size of the object. The current bounding box is given as input to the tracker. The assumption is that the aspect ratio of the object is constant. This

may not be true when a vehicle changes its orientation during motions. A constant velocity model is assumed during predicting objects in future frames. Bewley et al. [BGO16] mention that the constant velocity model is a poor predictor of true dynamics. Bewley et al. [BGO16] also observe that the object detection accuracy significantly affects the Kalman tracking performance.

Parameter tuning: We fine-tune the parameters of this pipeline using Mango [SAF20], a state-of-the-art hyperparameter tuning library. We use Equation 5.6 as the objective function for Mango to tune Kalman filter covariances for 1000 episodes (each of 2000 steps) in EagleSim. This amounts to a total tuning time of 16.6 hours.

5.5.4.2 Object_detection+reinforcement learning

We use the bounding boxes in this approach to train a deep-RL model. The input state of deep-RL consists of a vector of 4 variables (($[Xmin, Ymin, Xmax, Ymax]$) that represents the current bounding box as shown in Figure 5.3. The policy network is a 2-layer fully connected neural network, each having 64 hidden nodes, followed by an output layer with 3 discrete outputs for pan, tilt, and zoom parameters. The training is done using the PPO algorithm with the same hyperparameters as Eagle (Section 5.5.2).

We evaluate this approach at test time in two different ways: (i) Using the bounding boxes from the framework, which are error-free, called *PerfectBB+Deep_RL*. This is similar to the training setting. (ii) Using the bounding boxes from the finetuned Yolo5s object detector, *Yolo+Deep_RL*. The input to object detector is an image of size 240×240. This shows the usage of object detectors for a realistic setting when perfect bounding boxes are not available.

5.5.4.3 Relative_location+control

We build on the method proposed by Kyrkou et al. [Kyr21] using supervised machine learning for pan-tilt control. The training requires image datasets with annotated bounding boxes. We extend the proposed [Kyr21] approach by including a relative zoom variable. A neural network is trained

to predict three outputs defining a relative location: (i) $Rel_X = \frac{x}{Width/2}$, (ii) $Rel_Y = \frac{y}{Height/2}$, and (iii) $Rel_{Zoom} = \frac{(Xmax - Xmin)*(Ymax - Ymin)}{Width*Height}$. Where x, y, $Width$, $Height$, $Xmax$, $Xmin$, $Ymax$ and $Ymin$ are shown in Figure 5.3. The Rel_X, Rel_Y, and Rel_{Zoom} varies between -1 and 1. A separate controller uses the Rel_X and Rel_Y to control the pan and tilt of the camera and uses Rel_{Zoom} to modify the zoom.

The neural network has the same architecture as the policy network of Eagle (shown in Table 5.1), with a different output layer. The output layer consists of 3 nodes for each of Rel_X, Rel_Y, and Rel_{Zoom} with a linear activation. The input to the network is an image of size 240×240. This setting is called *NN+Controller*. We train this neural network by using the 50k labeled images capturing the relative location of $SUV1_{blue}$ vehicle in *Variable backgrounds+Trees+Humans* scenes from the EagleSim simulator.

5.5.4.4 Performance Comparisons

The comparison of Eagle with the current state-of-the-art approaches is shown in Table 5.9. The performance numbers for Eagle are added for policies trained for the scenario Sc-5 (from Table 5.3 and Table 5.5). For fairness, all approaches are evaluated for an end-to-end delay of 30ms from sensing image to applying PTZ actions. The evaluation in Table 5.9 is to track $SUV1_{blue}$ vehicle moving on random trajectories (with an average speed of 6 m/s) for 100 episodes (each episode is of 2000 steps or 1 minute in duration). The tracking setup of PTZ camera placement used for evaluation is the same as discussed in Section 5.5.3 (Tracking setup).

Removing the external object detector Yolo5s, we show the performance of multi-stage approaches using perfect bounding boxes in Table 5.10. The accuracy of perfect bounding boxes doesn't depend on the scene's complexity (or on the presence of backgrounds/other objects). But, these settings are not realistic, as, in practice, there will be errors in neural object detectors; however, this evaluation presents a valuable insight into the upper bound of performance as the object detectors improve in the future.

In simple scenes having a *Fixed background*, Eagle outperforms the next best approach of *Yolo+Kalman+Controller* by 4.6% in tracking duration (% Tracking). Eagle also maintains the metrics of $Center_x$, $Center_y$ and Obj_{size} better than others. With perfect bounding boxes (Table 5.10), controller performance improves as expected. However, when bounding boxes are imperfect (Table 5.9), there is a significant degradation. *Yolo+Deep_RL* suffers more degradation in comparison to *Yolo+Kalman+Controller*. This is because for *Yolo+Kalman+Controller*, Mango (hyperparameter tuner) selects lower Obj_{size} during tuning to maintain a higher tracking duration. However, in *Yolo+Deep_RL*, such fine-tuning is not possible and on replacing perfect bounding boxes with Yolo5s, object detector errors impact significantly. We see Eagle slightly outperforms even the *PerfectBB+Kalman+Controller* and *PerfectBB+Deep_RL* in tracking duration; we hypothesize that the raw images provide much richer information such as the orientation of the vehicle and the presence of other objects which is not available in the perfect bounding boxes.

In more complex scenes (*Variable backgrounds+Trees+Humans*), Eagle outperforms the next best approach (*Yolo+Kalman+Controller*) by 23.8% in tracking duration (% Tracking) and also maintains other metrics better in centering ($Center_x$, $Center_y$) the tracked object with superior resolution (Obj_{size}). This is because in more complex scenes, object detectors face more challenges in identifying the object of interest.

5.5.5 PTZ Tracking using Lightweight Object Detectors

We evaluate PTZ tracking using lightweight object detectors for vehicles having an architecture similar to the Eagle's policy network. We use the labeled dataset collected using EagleSim to train regression networks directly predicting bounding boxes of only vehicles in the images.

Training dataset: We collect the training data by driving 6 different vehicles ($SUV1_{blue}$, $SUV1_{red}$, $Pickup_{grey}$, $Pickup_{red}$, $Pickup_{grey}$, $Pickup_{red}$) in scenes having *Variable backgrounds+Trees+humans*. This data collection setup is same as the Sc-5 scenario (Table 5.2) used to train Eagle policies. We

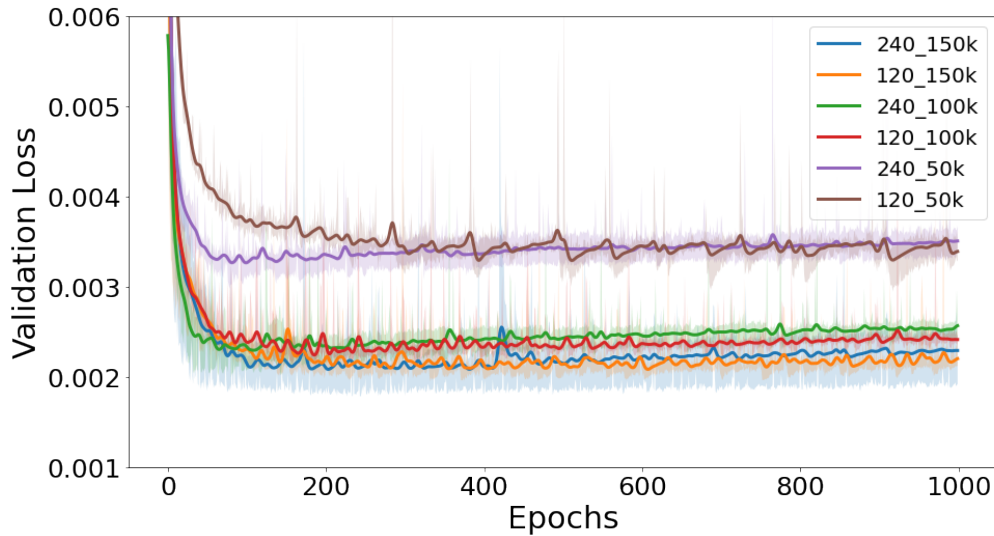


Figure 5.7: Learning curves of 6 different networks trained to predict bounding boxes of a car from images. The validation loss is shown for 1000 epochs. *240_150k* refers to the model trained using 240×240 image input on 150k image dataset. The mean and min-max spread of checkpoints for each network are shown.

use three different datasets (50k, 100k and 150k) of images captured by varying zoom levels of camera using the same tracking as discussed in Section 5.5.3 (Tracking setup).

Network architecture: We adopt the networks having same architecture as the Eagle' policy network (Table 5.1). We train the network using two different image inputs (120×120 and 240×240) in grayscale. The output layer of the network has four nodes predicting the bounding box of a vehicle in the image where the [Xmin, Ymin, Xmax, Ymax] are scaled between 0 and 1.

Learning curves: We train the networks using the mse loss function and adam optimizer for 1000 epochs. We train three checkpoints for each network with different random seeds and use 20% of the training data as the validation, and save the checkpoint with the lowest validation loss. The learning curves of 6 different networks (two different image sizes and three different dataset sizes) are shown in Figure 5.7.

PTZ tracking performance: We use these custom-trained object detectors to develop multi-stage

Table 5.11: Performance of PTZ tracking using lightweight object detectors having an architecture similar to the Eagle's policy network. Six different networks are trained to predict the bounding boxes of a car from images. Two image sizes (240×240) and three training datasets (50k, 100k, 150k) are used. *240_150k* refers to the model trained using 240×240 image input on 150k image dataset. In simple tracking scenes having *Fixed background*, all approaches have a very high tracking duration (around 98% Tracking) and maintain other parameters also very well. In complex scenes (*Variable backgrounds + Trees + Humans*), Eagle outperforms the next best approach (*240_150k*) by 16% tracking duration and maintain other metrics also similar.

Approach	**Fixed background**			
	%Tracking	$Center_x$	$Center_y$	Obj_{size}
Eagle	99.7 ± 5.4	0.86 ± 0.1	0.85 ± 0.1	0.27 ± 0.1
240_150k	97.8 ± 10.6	0.84 ± 0.1	0.84 ± 0.1	0.29 ± 0.1
240_100k	98.0 ± 11.1	0.84 ± 0.1	0.84 ± 0.1	0.30 ± 0.1
240_50k	97.6 ± 11.8	0.85 ± 0.2	0.84 ± 0.2	0.30 ± 0.1
120_150k	97.8 ± 10.8	0.84 ± 0.1	0.84 ± 0.1	0.29 ± 0.1
120_100k	98.4 ± 10.0	0.85 ± 0.1	0.84 ± 0.1	0.28 ± 0.1
120_50k	98.3 ± 10.2	0.85 ± 0.1	0.84 ± 0.1	0.28 ± 0.1
Approach	**Variable backgrounds+Trees+Humans**			
	%Tracking	$Center_x$	$Center_y$	Obj_{size}
Eagle	99.2 ± 4.6	0.86 ± 0.1	0.85 ± 0.1	0.27 ± 0.1
240_150k	82.5 ± 29.4	0.84 ± 0.1	0.84 ± 0.1	0.28 ± 0.1
240_100k	81.5 ± 29.1	0.85 ± 0.1	0.84 ± 0.1	0.28 ± 0.1
240_50k	77.7 ± 31.1	0.84 ± 0.1	0.85 ± 0.1	0.28 ± 0.1
120_150k	82.0 ± 29.6	0.85 ± 0.1	0.84 ± 0.1	0.28 ± 0.1
120_100k	81.8 ± 28.1	0.85 ± 0.1	0.84 ± 0.1	0.27 ± 0.1
120_50k	78.5 ± 31.2	0.84 ± 0.1	0.85 ± 0.1	0.27 ± 0.1

pipeline using Kalman filter [BGO16] and a rule-based controller [LMC21]. The Kalman filter and controller parameters are tuned using Mango similarly as discussed in Section 5.5.4. We evaluate these trackers for an end-to-end delay of 30ms from sensing image to applying PTZ actions as done for other evaluations in this chapter. The evaluation in Table 5.11 is to track $SUV1_{blue}$ vehicle moving on random trajectories (with an average speed of 6 m/s) for 100 episodes (each episode is of 2000 steps or 1 minute in duration). The tracking setup of PTZ camera placement used for evaluation is the same as discussed in Section 5.5.3 (Tracking setup).

As seen in the evaluation, for simple scenes having *Fixed background*, all approaches perform very well. This shows lightweight object detectors for vehicles work very well in predicting the bounding boxes of a car when no other objects are present in the scene and the background is a simple floor (Sc-1 shown in Figure 5.5). However, in complex scenes (*Variable backgrounds + Trees + Humans*), the tracking performance of lightweight object detectors degrades significantly. This is because detecting accurate bounding boxes becomes a harder task in the presence of other objects in the scenes. Eagle outperforms the next best approach (*240_150k*) by 16% tracking duration and maintains other metrics also similar to other approaches. We also see that the performance of object detectors using 240×240 images and 120×120 images is similar. The increase in the size of the dataset doesn't impact the performance in simple scenes, but in complex scenes, we observe a slight improvement in tracking performance from the 50k image dataset to the 150k image dataset.

5.5.6 Transfer of Eagle to the Real Scene Videos

Here, we show the direct transfer of Eagle policies trained purely in simulation to the real scene videos. EagleSim support capability to simulate pan-tilt-zoom actions on real videos. This is realized by simulating the effect of policy actions on an initial wider view containing an object of interest in the video. This evaluation presents a visual way to see the behavior of Eagle policies when transferred to real scenes. The simulation on real videos doesn't provide ground truth bounding boxes and cannot modify objects for trial and error learning. It thus doesn't scale to train

Table 5.12: Inference time in milliseconds (ms) of Eagle and optimized Yolo5s on embedded camera platforms.

Device	**Raspberry Pi 4B**	**Jetson Nano**
Eagle	9.2 ± 1.3 ms	6.2 ± 2 ms
Yolo5s	1817 ± 14.1 ms	86.2 ± 1.5 ms

deep-RL from real videos.

We simulate the pan-tilt-zoom actions from Eagle policies trained for Sc-5 on real-scene videos. The results are shown in Figure 5.8 for two different scenes. The left scene consists of a toy blue SUV vehicle moving on a concrete floor. The variations in this scene are much simpler. The tracking progress is shown from the top image to the bottom image in Figure 5.8:A. The initial bounding box of wide view (PTZ View) is manually selected, which is given as an input to the trained policy. The actions predicted by the policy are used to update the selected bounding box by moving it left-right for pan and up-down for tilt. The zoom action controls the size of the bounding box.

The scene in Figure 5.8:B shows a real vehicle of grey color moving in a background having trees/patterns. The tracking progress is again shown from top to bottom, where the first image of PTZ View shows the wider view given as an input to the policy. As the tracking progresses, the policy focuses on the object of interest and follows it. As seen, the PTZ view identifies the vehicle in the initial wide view and follows it in the video based on the policy actions. This shows that the Eagle is a very promising alternative to replace existing multi-stage approaches in the presence of richer scene variations of the real world.

5.5.7 Runtime of Eagle on Embedded Cameras

Next, we measure the runtime of Eagle policies on embedded camera platforms. We consider Raspberry Pi 4B and Jetson Nano devices which are also supported by embedded PTZ cameras [Ard22].

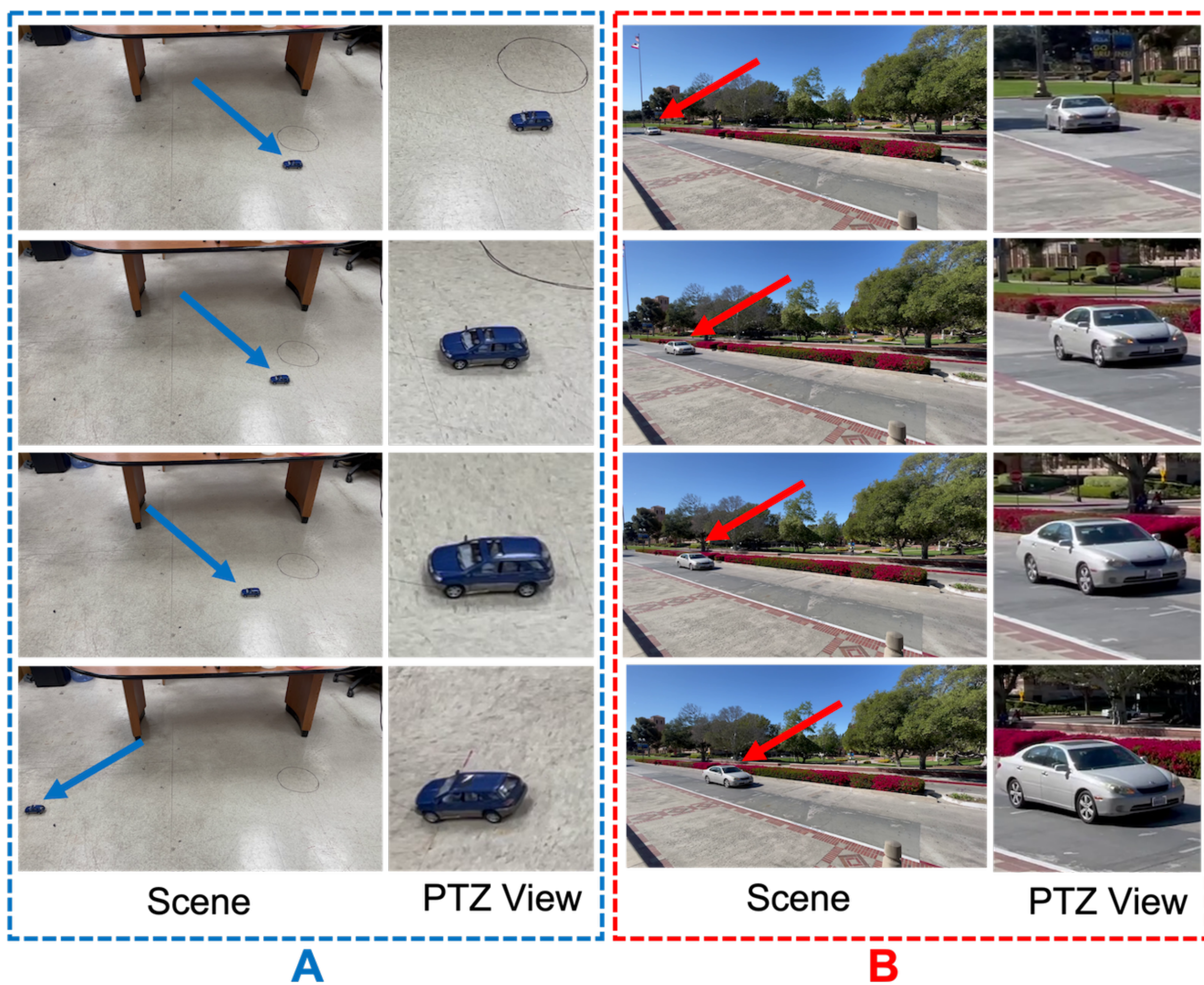


Figure 5.8: Eagle policies on real videos. The arrows show the vehicle to track in the video scene. The PTZ view is maintained by Eagle while tracking the vehicle. The top images show the starting point where the PTZ view is not focused.

We compare Eagle with the next best approach of *Yolo+Kalman+Controller* (Table 5.9), and with the PTZ tracker using the lightweight object detector designed in Section 5.5.5.

Eagle policies have only 79k network parameters compared to Yolo5s' 7.2 million parameters. The inference latency of Eagle and Yolo5s is reported in Table 5.12. On Raspberry Pi, we optimize the inference for Eagle and Yolo5s using TensorFlow Lite [Ten22a]. On Jetson Nano, the Yolo5s model has an inference latency of 218 milliseconds (ms). We optimize Yolo5s for Jetson Nano using TensorRT [Ten22b] and further quantize the model to float16 to reduce its inference latency. The inference latency of optimized Yolo5s on Jetson Nano is reported in Table 5.12.

The PTZ camera [Ard22] supports a frame rate of 120 Hz, which when tested with neural network inference reduces to 100 Hz. The camera supports PTZ actions with a resolution of 1 degree and has an actuation delay of 10 ms. When using Eagle for PTZ control on Raspberry PI, end-to-end delay is around 30 ms (10 ms for inference, 10 ms for sensing (100 Hz), and 10 ms for actuation), enabling a real-time deployment with 33 FPS. Yolo5s has an inference latency of 1817 ms on Raspberry PI, making it completely infeasible to run *Yolo+Kalman+Controller* on Raspberry PI.

On Jetson Nano, Eagle achieves an even higher FPS of 38 (inference latency of 6.2ms and similar sensing+actuation delay of 20ms). *Yolo+Kalman+Controller*, even when using optimized Yolo5s, has a significantly higher inference latency of 86.2ms on Jetson Nano. Considering the sensing and actuation delays, the total end-to-end delay for *Yolo+Kalman+Controller* is 106.2 ms, which results in a very low FPS of 9.

The PTZ tracker using a lightweight object detector in Section5.5.5 has similar inference latency as Eagle due to the same network architecture. This tracker also outperforms the Yolo5s pipeline; however, in complex scenes having background variations, trees, and human characters, Eagle policies have superior control performance, as shown in Table 5.11. Hence, Eagle policies represent a lightweight controller with superior control performance for real-time deployment.

Figure 5.9: A sample scene with multiple objects of interest.

5.6 Discussion

Policy behavior on multiple objects of interest: Eagle policies in Section 5.3.3 are trained by assuming a single object from class A is present at a given time in the scene, while the same policy generalizes across all objects of class A. Here, we test the policy behavior by adding multiple objects from class A to the same scene as shown in Figure 5.9. We see that the reward of Equation 5.1 incentivizes an agent to track the larger object lying closer to the center of the image to achieve a higher expected sum of rewards. In Figure 5.9, we test the policy of the Sc-5 scenario in a scene having two $SUV1_{blue}$ vehicles. As seen in the images from left to the right, the policy tracks the larger car and follows it as it moves in the scene.

Speed of objects: We evaluated controller performance on vehicles moving at an average of 6m/s and with a max speed of 16m/s. Vehicles come to a complete stop at the boundaries of the virtual worlds and randomly move in a different direction. The action space of Section 5.5.2 works across these speed variations and tracks slow-moving humans as well. To track even faster-moving objects, the action space can have more options, or the end-to-end delay can be reduced to modify the PTZ parameters faster.

Multiple cameras: EagleSim can capture bounding boxes of multiple objects of interests and also allows the control of multiple PTZ cameras in the same scenes. Multiple cameras present a problem of collaborative tracking. We leave the study of end-to-end controllers for collaborative tracking as future exploration.

Limitations of Eagle: Changing the tracking goal in multi-stage approaches is easier by filtering the outputs from the object detector. Modifying the tracking goal in Eagle to a new type of object may require retraining of a new policy from scratch. EagleSim simulator is designed to automate the retraining and alleviate the labeling efforts. We also saw that runtime tasking in Eagle to selectively track humans and vehicles comes at a performance cost. Runtime tasking in multi-stage approaches can be enabled by changing their tracking goal by adopting general-purpose object detectors. In light of these limitations, Eagle is more suitable for applications where the general categories of objects to track are known ahead, and lightweight controllers are required.

5.7 Conclusion

We introduced a new lightweight approach called Eagle for end-to-end PTZ control having superior performance. To realize the proposed solution, we also introduced an accompanying simulator called EagleSim, automating the training pipeline. The capability provided by EagleSim allows reproducible scenarios to ease the development and benchmark different classes of autonomous PTZ control algorithms. Further, the availability of Oracle bounding boxes in EagleSim enables us to study multi-stage approaches by removing errors in their object detection pipeline showing upper-performance bounds. We see that end-to-end control can slightly outperform these upper bounds. Finally, the action predicted by end-to-end Eagle policies trained purely in photo-realistic simulation can transfer to real-world videos, suggesting it is a promising alternative to conventional approaches.

CHAPTER 6

Enabling Hyperparameter Tuning of Machine Learning Classifiers in Production

Machine learning (ML) classifiers are widely adopted in the learning-enabled components of intelligent Cyber-physical Systems (CPS) and tools used in designing integrated circuits. Due to the impact of the choice of hyperparameters on an ML classifier performance, hyperparameter tuning is a crucial step for application success. However, the practical adoption of existing hyperparameter tuning frameworks in production is hindered due to several factors such as inflexible architecture, limitations of search algorithms, software dependencies, or closed source nature. To enable state-of-the-art hyperparameter tuning in production, we propose the design of a lightweight library (1) having a flexible architecture facilitating usage on arbitrary systems, and (2) providing parallel optimization algorithms supporting mixed parameters (continuous, integer, and categorical), handling runtime failures, and allowing combined classifier selection and hyperparameter tuning (CASH).

We present *Mango*, a black-box optimization library, to realize the proposed design. Mango is currently used in production at Arm for more than 30 months and is available open-source (`https://github.com/ARM-software/mango`). Mango outperforms other black-box optimization libraries in tuning hyperparameters of ML classifiers having mixed parameter search spaces. We discuss two use cases of Mango deployed in production at Arm, highlighting its flexible architecture and ease of adoption. The first use case trains ML classifiers on the Dask cluster using Mango to find bugs in Arm's integrated circuits designs. As a second use case, we introduce an AutoML framework deployed on the Kubernetes cluster using Mango. Finally, we present

the third use-case of Mango in enabling neural architecture search (NAS) to transfer deep neural networks to TinyML platforms (microcontroller class devices) used by CPS/IoT applications.

6.1 Introduction

Enabling Hyperparameter tuning at a production scale is crucial to designing better performing ML classifiers embedded in emerging CPS/IoT applications. However, a typical ML pipeline in production can be too specialized and complex, demanding a trained team of human experts with specific domain knowledge for classifier selection with optimal hyperparameters. The combined classifier selection and hyperparameter optimization in production face the following challenges:

1. Complex deployments: The production ML pipelines are complex and realized using a combination of arbitrary systems (e.g., custom on-premise software, cluster frameworks, cloud infrastructures) decided by several factors, including the nature of application and developer preferences. Therefore, flexible architecture with abstractions allowing *usage on arbitrary systems* is needed.

2. High complexity of the hyperparameter search: Search is becoming increasingly complex, with many choices for classifiers and their rich parameter spaces. It is further exacerbated in production pipelines due to the recurrent nature of tuning tasks triggered by data shifts or process changes. Consequently, searching the space of several classifiers demands *combined algorithm/classifier selection and hyperparameter optimization (CASH) [THH13].* Further, to speed up the search, *intelligent parallel algorithms* utilizing parallel computing with abstractions to handle runtime failures are needed.

Further, abstractions offering uniformity in local and cluster usage, including syntax compatibility with the widely used ML libraries like Scikit-learn [PVG11], can reduce the effort needed to integrate with existing deployments. While several hyperparameter tuning software exists, their adoption in an arbitrary production pipeline is hindered due to their dependence on particular compute scheduling extensions [BYC13, SLA12, HHL11, ASY19, FKE15], closed source nature [GSM17], search algorithm limitations [aut16a, aut16b] and significant overhead adopting

the entire software dependencies [LLN18,ZVP19,Roc15,Mou17]. For example, the parallel search in Hyperopt [BYC13] is dependent on the MongoWorker processes or Apache Spark [Hyp19].

To enable hyperparameters tuning in production, we present Mango, a black-box optimization library. Mango is a research project that provides hyperparameter tuning to ML pipelines at Arm with more than 30 months of production usage. Mango is open-sourced under Apache 2.0 license to contribute and learn from the community. Mango provides the following core features addressing the above challenges:

- Modular design that allows the user to schedule objective function evaluations on arbitrary infrastructure. Furthermore, API provides a unique capability to handle runtime failures crucial for production deployments.
- An efficient realization of Bayesian optimization using the Gaussian process (GP). We incorporate optimal handling of mixed parameters and intelligent batch sampling for parallel search for practical adoption of GP.
- An algorithm to directly solve CASH problem using multiple GP surrogates. To the best of our knowledge, existing GP libraries don't solve the CASH problem.

To highlight the flexible architecture enabling the adoption of Mango in complex ML pipelines, we discuss two production use cases (1) a bug hunting workflow deployed on the Dask cluster [Roc15] using ML classifiers to optimize the design verification of Arm's integrated circuits, (2) an AutoML framework deployed on the Kubernetes cluster[1]. Figure 6.1 shows the first use case doing design verification of integrated circuits at Arm. We evaluate the implemented optimization algorithms in Mango on a collection of benchmark functions and classifiers. Finally, we present a third use case of Mango enabling NAS for Cortex-M microcontrollers class devices found in resource constrained IoT and CPS applications.

[1] https://kubernetes.io/

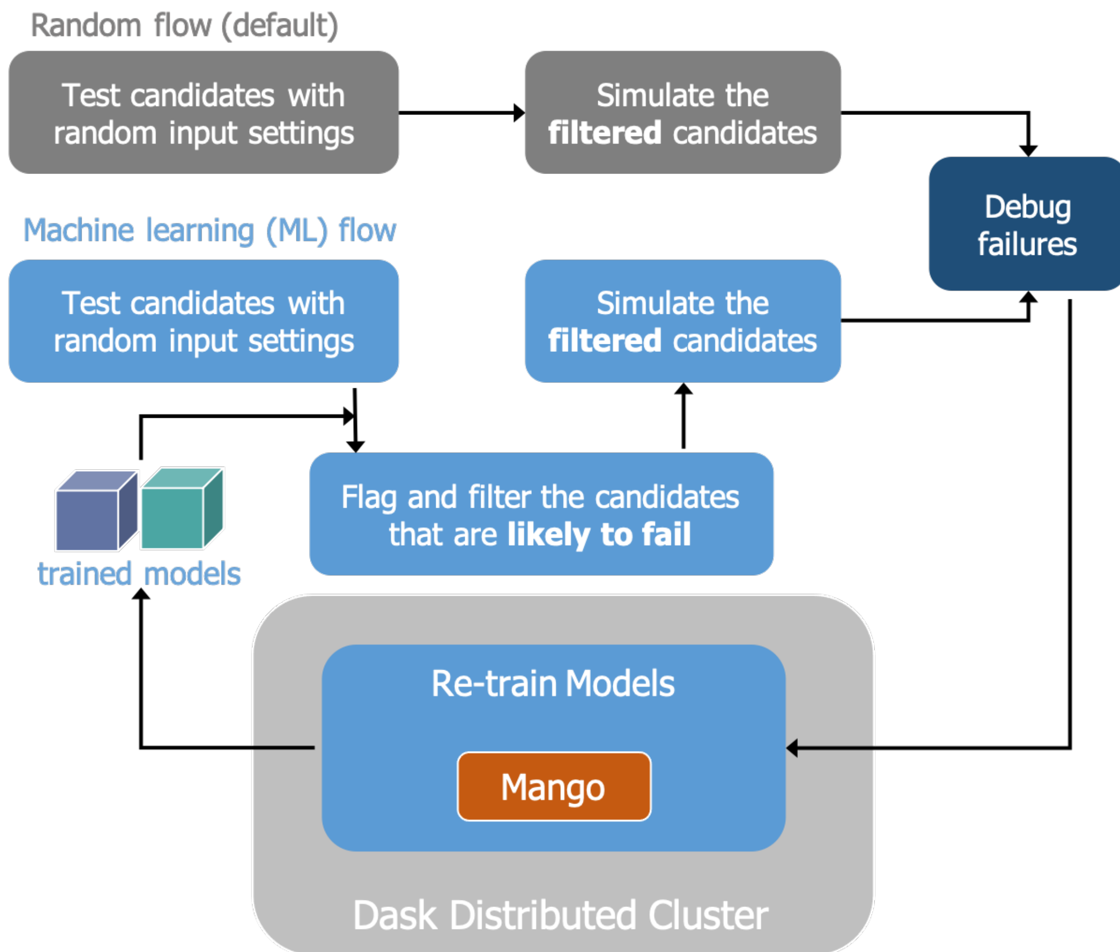


Figure 6.1: A *Bug Hunting Workflow* is illustrated which is part of the design verification of integrated circuits at Arm. A machine learning pipeline replaced the default pipeline to predict the preferred input candidates. Mango is deployed on the Dask distributed cluster to automate the hyperparameter tuning of ML models used for design verification.

6.2 Background and Related Work

The optimization algorithms and frameworks for automatic hyperparameter tuning are active research areas. Here, we discuss the hindrances in production deployments of widely used hyperparameter tuning frameworks and briefly review the black-box optimization algorithms.

6.2.1 Hyperparameter Tuning Frameworks

The parallel hyperparameter frameworks can be categorized into two groups (1) software built on distributed frameworks to provide hyperparameter tuning as a feature [LLN18,ZVP19,Aut,Mou17] and (2) optimization libraries with integrated scheduling extensions [ASY19, BYC13, HHL11, FKE15, THH13, aut16a]. However, the adoption of these frameworks in production ML pipelines deployed on arbitrary systems faces hindrances primarily due to *high overhead* in adoption for the former group and *dependence on custom-built parallel schedulers* for the latter.

For example, Katib [ZVP19] and Polyaxon [Mou17] are built on top of Kubernetes. Tune [LLN18] is a python library deployed using the Ray framework. Dask-ml [Aut] provides hyperparameter tuning using the Dask framework. These systems provide features like auto-scaling, failovers, and rich scheduling abstractions. However, their integration into an arbitrary deployment demands adopting the specific underlying framework and additional software dependencies, creating development and maintenance overhead. For example, using Katib needs the Kubernetes framework's adoption and additional database, API server, and controller processes. These systems can be a good fit if the application uses the underlying framework for all of its features.

The optimization libraries with integrated scheduling extensions have custom-built mechanisms to run parallel workers. However, their distributed extensions lack features like auto-scaling, failover, and architecture flexibility to deploy on any arbitrary underlying cluster computing framework essential in a production deployment. For example, Optuna [ASY19], Hyperopt [BYC13], SMAC [HHL11], Auto-sklearn [FKE15], GPyOpt [aut16a], and Auto-WEKA [THH13] lack scheduler abstractions to use arbitrary cluster computing frameworks. The parallel search in Hyperopt

is dependent on the Apache Spark or MongoDB [BYC13]. The parallel search in SMAC needs a shared file system for multiple sequential runs to collaborate. Auto-sklearn's parallel search is dependent on the Dask cluster framework [Roc15] or requires a shared file system to run SMAC parallelly. Further, Auto-sklearn and Auto-WEKA are dependent on the scikit-learn and WEKA, respectively.

Mango: In contrast to the existing frameworks, Mango is designed with flexible modular architecture providing local scheduler extensions, abstractions to enable integration with arbitrary compute infrastructures, and consideration for the failures to ease the adoption in production deployments.

6.2.2 Hyperparameter Tuning Algorithms

The simple methods include random search, in which each parameter is selected independent of the previous selections, and grid search, which selects parameters systematically along a grid. However, generally, the guided search methods discussed next outperforms them.

Bayesian optimization algorithms: Sequential model-based Bayesian optimization (SMBO) is a state-of-the-art approach to minimize the number of evaluations required to find optimal hyperparameters. SMBO uses a surrogate model to predict the performance of arbitrary parameter configuration [SLA12, SKK09]. A cheap acquisition function is used to select the next evaluated parameter using surrogate model predictions [SLA12]. Typical surrogate models widely used are Gaussian process (GP) [SLA12,SKK09], tree-structured Parzen estimators (TPE) [BBB11], neural network [SRS15], and random forest [HHL11].

The GP surrogate is shown to outperform TPE and random forest for functional benchmarks [ASY19] and is available in GPyOpt, Auto-sklearn, and Auto-WEKA, among others. However, GP demands special techniques when used for ML classifier to (1) handle categorical variables [GH20], (2) search conditional parameter spaces [LDG17], and (3) reduce search computational complexity [ASY19].

For parallel search, GP is extended to sample a batch of parameters. The proposed approaches

to sampling a batch use penalty [DKB14, GDH16, HHL11] in the acquisition function, select multiple peaks [NRG16, GP18] from the acquisition function, and Monte Carlo estimation [SLA12]. GPyOpt provides penalty and Monte Carlo estimation for parallel search. However, GPyOpt doesn't directly allow to solve the combined algorithm selection and hyperparameter optimization (CASH). Auto-WEKA and Auto-sklearn use multiple sequential runs with random seeds to simulate the parallel search.

TPE surrogate is available in Hyperopt library and Optuna framework. TPE is similar to kernel density estimators. It transforms the hyperparameters' generative process, replacing the distributions of the configuration prior with non-parametric densities. By construction, it can handle categorical and conditional parameters and scales linearly. However, TPE is designed to be sequential in nature [BYC13], thus suffers performance loss during the parallel search in comparison to the specialized GP approaches sampling a batch. SMAC handles categorical and conditional configurations using the random forest as the surrogate model. However, the intelligent parallel search is missing in SMAC as it uses multiple sequential runs with random seeds.

Population-based training (PBT) and multi-fidelity optimization algorithms: PBT [Sim13] methods such as genetic and evolutionary techniques maintain a population of parameters and improve this population by applying local perturbations. PBT methods are directly parallel, allowing the population of size N to be evaluated on N machines. However, training a large number of configurations is expensive. To speed up the search using PBT, multi-fidelity optimization algorithms like successive having [JT16] and Hyperband [LJD17] use partial training. Although these approaches are cheaper to evaluate, they may suffer from approximations errors in small budget evaluations, but often the speedup achieved is more significant.

6.2.3 Algorithms Implemented in Mango

The algorithms implemented in Mango use Bayesian optimization using GP. Mango addresses the limitations of the GP surrogate while maintaining its advantages over TPE and Random forest.

The implemented algorithms handle carefully the mixed numerical/categorical search spaces, allows sampling of a batch intelligently for parallel evaluations, and supports combined classifier selection and hyperparameter optimization while significantly reducing the computational complexity associated with GP regression.

We discuss the challenges in enabling hyperparameter tuning in production, Mango features addressing these challenges, hindrances in adopting existing frameworks, implemented algorithms, and share the learning experiences by including two deployed use cases.

6.3 Mango

Mango has a functional-based API to integrate with a model training pipeline. The four abstractions in Mango are (1) *Parameter Space Definer*, (2) *Objective Specifier*, (3) *Tuner*, and (4) *MetaTuner*. The modular architecture enables the integration of new functionality and the ease of production maintenance.

Parameter Space Definer provides python constructs to easily specify complex search spaces, including mixed numerical/categorical values. The design of *Objective Specifier* allows classifiers' training on local machines using an integrated scheduler and arbitrary systems (e.g., custom-local software, cluster frameworks) by exposing sampled batches. *Tuner* exposes implemented algorithms for serial and parallel search. *MetaTuner* solves a CASH problem. We show the skeleton codes from production use cases to highlight these features. Figure 6.2 shows an example of Mango for hyperparameter tuning of XGBClassifier on a local machine using the integrated parallel scheduler. The default optimization algorithm and configurations can be modified.

6.3.1 Mango Abstractions

Parameter Space Definer: Mango uses Python constructs (range and list) to define search spaces with mixed numerical/categorical values. As shown in Figure 6.2, *param_space* is defined as a

```
from mango import Tuner, scheduler
from scipy.stats import uniform
from xgboost import XGBClassifier
...
param_space = {'learning_rate': uniform(0, 1),
              'gamma': uniform(0, 5),
              'max_depth': range(1, 21),
              'n_estimators': range(1, 11),
              'booster':['gbtree','gblinear',
              'dart']}
@scheduler.parallel(n_jobs=4)
def objective(**params):
    ...
    clf = XGBClassifier(**params)
    accuracy = ...
    return accuracy
tuner = Tuner(param_dict, objective)
Study = tuner.maximize()
```

Figure 6.2: An example of Mango to tune the hyperparameters of XGBClassifier from the Xgboost library using a parallel scheduler on the local machine. Parameter space consists of distribution, range, and categorical variables.

python dictionary. Continuous variables use distributions from Scipy[2]. All the 60+ distributions from Scipy are supported, allowing the flexibility to specify preferred regions in the search space. Mango supports user-defined parameter distributions. The parameter space definitions are compatible with the Scikit-learn, allowing replacement for existing applications using Scikit-learn.

Objective Specifier: The objective specifications are available to train the classifier using a local machine or any arbitrary system. The objective function training classifier uses an integrated parallel scheduler on the local machine is shown in the Figure 6.2. Here, the input to the *objective()* function is a dictionary (*params*) with a single sampled point suggested for evaluation by Mango. The @*scheduler* decorator specifies the number of parallel jobs.

For *deployments on arbitrary systems*, a more general skeleton of *Objective Specifier* is available, as shown in Figure 6.3. It exposes the sampled batch directly to the user-defined objective function to evaluate an application-specific scheduler. This scheduler's nature is decided based on the deployment framework. We allows the user-defined objective function to discard the *failed evaluations* as shown in *objective* function in the Figure 6.3 to make progress even with runtime failures. The specific technique (e.g., timeout in Figure 6.3) to identify a failure is kept outside of the Mango, as it may depend on the underlying compute platform and tuning task.

The skeleton code shown in Figure 6.3 is part of an AutoML framework (see Section 6.4.3) deployed on the Kubernetes cluster at Arm. The objective function can return the result as a list of values specifying the successful evaluations and their respective hyperparameters without waiting for all the evaluations to complete. The batch objective function skeleton is kept independent of the underlying compute infrastructure, with no dependency on the additional databases or a shared file system, enabling its adoption across applications.

Tuner: A parameter space definition and the specified objective function are used by *Tuner* to search optimal hyperparameters. The *config* parameter (Figure 6.3) is optional. It controls the maximum number of iterations, the initial random iterations, the batch size for parallel search, and

[2]http://www.scipy.org/

```python
from kubernetes import client
...
param_space = ...
def objective(params_batch):
     # train on cluster using the sampled parameters
    jobs = [client.create_job(params, ...)
            for params in params_batch]
    # poll for job completion
    results = []
    while not timeout or not all_done:
        results = [job.result() for job in jobs
                   if job.complete()]
    return results
# control the max number of iterations, batch size,
conf = {'num_iteration':100, 'initial_random':5,
'batch_size':4, 'parallel':'clustering'}
tuner = Tuner(objective, param_space,conf)
Study = tuner.maximize()
```

Figure 6.3: Skeleton code of Mango on Kubernetes cluster that is deployed as part of the AutoML framework at Arm. Partial *results* are returned by the *objective* function based on timeout. The *conf* data structure modifies the default behavior of *Tuner*.

the optimization algorithm. *Tuner* exposes sequential and parallel search algorithms.

MetaTuner: *MetaTuner* is designed to solve a CASH problem. The skeleton code of *MetaTuner* deployed in production on a Dask distributed framework [Roc15] is shown in Figure 6.4. This code is part of the *Bug Hunting Workflow* shown in Figure 6.1. The *param_space* data structure is a list of search spaces for individual classifiers identified by their *type* during scheduling.

6.3.2 Optimization Algorithms in Mango

Mango algorithms are based on Bayesian optimization. Here, we summarize the sequential search, handling of the categorical variables, batch sampling to enable parallel search, and the CASH algorithm.

Sequential search: The sequential search uses Bayesian optimization with GP as the surrogate model. We use the Matern kernel function and the upper confidence bound (UCB) as the acquisition function [SKK09]. The next sampled hyperparameter is selected based on the predicted mean (exploitation) and the corresponding variance (exploration). The exploration factor is used to decide a trade-off between exploitation and exploration. The exploration factor in Mango is fixed by default to 2.0; however, for expert users, we allow adaptive exploration proposed by Srinivas et al. [SKK09], where the exploration factor is heuristically decided based on the complexity of the search space (domain size) and the current iteration count. The idea is to allow more exploration when the classifier's search space is huge. We do the Monte Carlo optimization of the acquisition function by sampling the parameter space and then selecting the next point to evaluate based on the acquisition function. The total number of samples drawn is decided based on the the complexity of the search space inferred using the definition.

Handling categorical values: The naive GP assumes continuous input variables. Thus, handling categorical and integer values requires careful consideration. We use one-hot encoding for the categorical values. However, naively rounding off the categories or integers during evaluations can result in poor performance as the actual point of objective evaluation may differ from the

```python
from dask.distributed import Client
...
dask_client = Client()

param_clf_nn = {'type': 'clf_nn',...}
param_clf_svm = {'type': 'clf_svm',...}
param_spaces = [param_clf_nn, param_clf_svm]

def objective(params_batch):
    futures = []
    # Submit Jobs to the Dask cluster
    for params in params_batch:
        #schedule classifier based on type
        clf = params.pop('type')
        future = dask_client.submit(fit_and_score,
                                    clf, **params)
        futures.append(future)
    # Job completion or wait for timeout
    results = [future.result(timeout) for future in
    futures]
    return results
metatuner = MetaTuner(objective, param_spaces)
Study = metatuner.maximize()
```

Figure 6.4: Skeleton code deploying *MetaTuner* algorithm on the Dask cluster, which is part of the bug hunting application used for design verification of Arm integrated circuits designs.

proposed point [GH20]. Our approach is motivated by the solution proposed by Garrido-Merchán et al. [GH20]. We optimize the acquisition function by sampling only the valid points from the search space; thus, there is no mismatch between the proposed and actual evaluation.

Parallel search: Conventionally, Bayesian optimization using GP is a sequential search since new information must update the acquisition function. The challenge in selecting a batch of values is to ensure exploration diversity in the batch. A simple technique to enforce diversity is that no choice is selected twice in the batch. It can be done by ranking the choices according to the UCB and then selecting top picks until new feedback is available. However, this naive approach has limited exploration [DKB14], demanding intelligent parallel strategies. We provide two algorithms to sample a batch of values in Mango.

The first algorithm, *Clustering search* used by default, is motivated by selecting peaks [GP18, NRG16] of acquisition function within a batch. It has the following two steps: (1) First, we select a set of promising domain samples (top 25% by default) based on the acquisition function. (2) Next, these domain samples are clustered based on their distance in the search domain space. We select the hyperparameter choice from each cluster with the highest acquisition function value and add it to the batch. We use K-Means clustering.

The second algorithm is *hallucination search* which is based on the idea of applying penalty [DKB14, GDH16] to sample a batch using the acquisition function.

MetaTuner algorithm to solve CASH: Direct addition of an extra algorithm selection parameter in GP assumes that information is shared between the hyperparameters of different classifiers. A regular GP would make an invalid credit assignment in these settings [LDG17,JAG17]. To address this, we train multiple GP surrogates for each classifier independently. Our approach is motivated by the idea of exploiting the structure of the optimization problem proposed by Bergstra et al. [BBB11] and Jenatton et al. [JAG17]. Algorithm 1 is the serial version of *MetaTuner* algorithm.

Some classifiers can have an exploration bias occurring from the evaluation of good accuracy regions early on. To avoid these issues, we use random exploration ($meta_{xpl}$) with a decay rate

Algorithm 1: *MetaTuner* algorithm.

input : list of parameter spaces P_{List}, objective function obj_fxn, and configuration $Conf$

output: type of classifier C_{type}, optimal parameters O_{par}

```
1  meta_xpl ← 1.0, min_xpl ← exp_value;
2  decay_rate ← decay_value, acc_max = 0;
3  C_type ← None, O_par ← None;
4  for i ← 1 to Conf[max_iterations] do
5  |   Curr_clf ← None, Curr_par ← None;
6  |   rand ← random();
7  |   if rand<meta_xpl then
8  |   |   Curr_clf ← randINT(1,no_of_clf)
9  |   |   Curr_par, _ ← get_gp_acq(P_List[Curr_clf],Conf);
10 |   |   meta_xpl ← max(meta_xpl * decay_rate, min_xpl);
11 |   else
12 |   |   for i ← 1 to Size(P_List) do
13 |   |   |   X[i],Y[i] ← get_gp_acq(P_List[i],Conf);
14 |   |   end
15 |   |   Curr_clf ← argmax(Y[i]);
16 |   |   Curr_par ← X[Curr_clf];
17 |   end
18 |   curr_evaluation ← obj_fxn ([Curr_clf,Curr_par]);
19 |   update_gp([Curr_clf,Curr_par,curr_evaluation]);
20 |   update_gp_exp([Curr_clf]);
21 |   if curr_evaluation>acc_max then
22 |   |   acc_max ← curr_evaluation;
23 |   |   C_type ← Curr_clf, O_par ← Curr_par;
24 |   end
25 end
26 return C_type, O_par
```

($decay_{rate}$) along with a minimum exploration (min_{xpl}) across classifier. Lines[7-10] do random exploration across classifiers using $meta_{xpl}$. Function *get_gp_acq* suggests the parameter and the respective acquisition function value using the parameter space definition and configuration of the used classifier. Lines[12-16] select a classifier and hyperparameter to evaluate based on its acquisition value. The objective function evaluation for the selected classifier and the Gaussian process surrogate update is done in Lines[18-19]. Line-20 updates the surrogate's exploration for the classifier that is evaluated. The idea is to favor other classifiers for future evaluations by using more exploration factors if they have high uncertainty due to their large search space. Finally the best performing classifier and optimal parameter is maintained in the Lines[21-23]. The default values ($decay_{rate}$=0.9, min_{xpl}=0.1) available in *MetaTuner* are the same that are used for experiments.

A batch version of this algorithm is implemented in Mango, where we initially select a batch $|B|$ of values from individual surrogates (*get_gp_acq*) using parallel search, and then rank these ($N * |B|$) values, where N is the number of classifiers, to select $|B|$ points to evaluate in parallel. Note that a mix of classifiers may be evaluated in batch based on their acquisition function.

6.4 Evaluation and Case Studies

6.4.1 Optimization Performance Evaluation

We compare Mango with several black-box optimization libraries using the multiple criteria methodology proposed by Dewancker et al. [DMC16], also used by Akiba et al. [ASY19]. Specifically, we measure performance by the solution's proximity to the optimal point (accuracy) and the number of iterations required to reach the optima (speed). We performed experiments across two classes of optimization tasks: (1) Synthetic test functions and (2) ML classifiers. Each optimization task uses 80 iterations and is repeated 30 times to account for the algorithm's stochastic nature [DMC16]. Results are statistically compared for accuracy and speed criteria using paired Mann-Whitney U test with $\alpha = 0.01$ [DMC16]. Libraries to compare against are chosen to represent different fla-

vors of Bayesian optimization: Hyperopt with TPE surrogate, Optuna with a mixture of TPE and CMA-ES, SMAC with random forest surrogate. GPyOpt with Gaussian process surrogate, and lastly, random search serves as the baseline.

6.4.1.1 Synthetic test functions

We used a collection of 53 functions having continuous search spaces from a benchmark suite of test functions [DMC16]. Figure 6.5a shows the results where the objective function is evaluated sequentially. Mango is worse than Optuna in 5/53 tests and Hyperopt in 2/53 tests. This is expected because the GP surrogate provides a more accurate representation of the objective function than TPE [ASY19]. Mango performs worse than SMAC in 12/53 tests. The performance of Mango is competitive when compared to the other GP-based optimizer GPyOpt (worse in 22/53, tied in 10/53 tests).

Figure 6.5b shows the results for parallel search where the objective function is evaluated using four workers. Mango's clustering parallel search performance is compared with GPyOpt's local penalization approach, Optuna's random sampling, and random search. Mango performs worse than Optuna in the 17/53 test and worse than GPyOpt in the 32/53 tests. Hyperopt and SMAC also provide distributed optimization using custom-built scheduling frameworks. However, we could not complete the experiments for them due to repeated failures of their custom scheduling framework.

GPyOpt internally uses a gradient-based method to optimize the acquisition function, while Mango uses Monte Carlo sampling. The gradient-based method provides a slight advantage to GPyOpt in continuous search spaces, which is the case for these test functions. However, Mango's sampling approach is more suitable for heterogeneous search spaces that include categorical and integer parameters, which is the case for hyperparameter tuning of ML classifiers, as discussed in the next section.

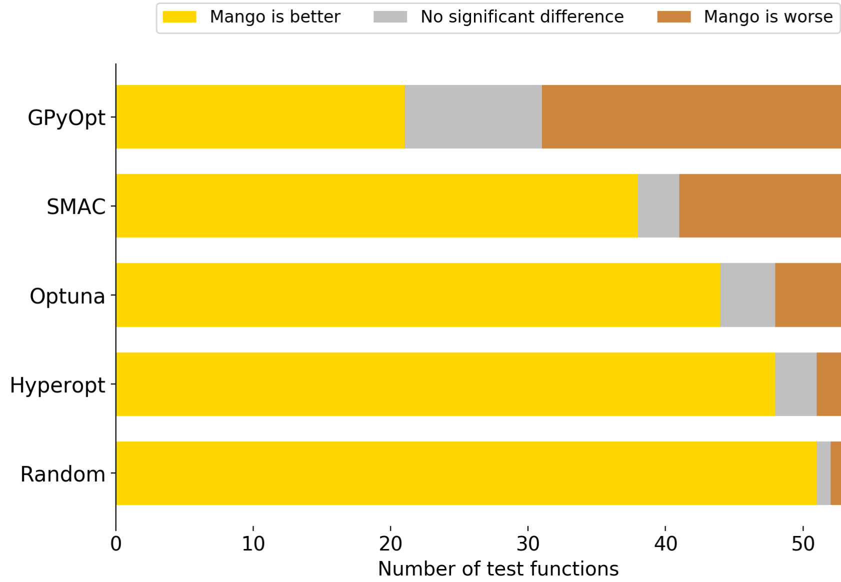
6.4.1.2 Tuning ML classifiers

We compared the performance for hyperparameter tuning of three ML classifiers: Xgboost, K-Nearest Neighbor (KNN), Support Vector Machines (SVM) to maximize the 3-way cross-validation accuracy for the iris plants dataset, wine recognition dataset, and breast cancer Wisconsin (diagnostic) dataset taken from Scikit-learn, i.e., a total of 9 tuning tasks (three classifiers trained using three datasets). The search space includes continuous, integer, and categorical parameters with the exact definitions available [Res21]. The experiment setup is the same as before, having 80 iterations and 30 repeated runs. Results are shown in Figure 6.6. As seen in Figure 6.6a, Mango performs better than all other libraries in 6 or more tasks out of 9. Figure 6.6b shows the results for parallel hyperparameter tuning with four workers.' As seen, the *clustering search algorithm* of Mango outperforms GPyOpt's local penalization approach and Optuna's random sampling.

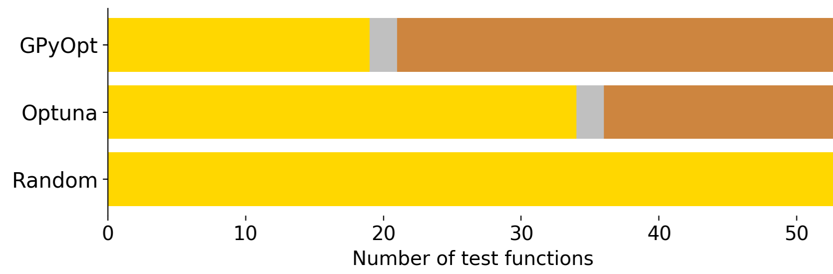
6.4.1.3 Hyperparameter Tuning across Classifiers

We compare the performance of *MetaTuner* to solve the CASH problem with Optuna's TPE+CMA-ES surrogate, Hyperopt's TPE surrogate, SMAC's random forest surrogate, and naive random search. GPyOpt is not included in this evaluation as it doesn't allow conditional search spaces. Sampling for the naive random search is done by uniformly choosing a classifier followed by randomly sampling a hyperparameter from the corresponding search space. The optimization objective is to find the best classifier and corresponding hyperparameters from the neural network, Xgboost, KNN, SVM, and decision tree. Experiments are done for three datasets taken from Scikit-learn: iris plants dataset, wine recognition dataset, and breast cancer Wisconsin (diagnostic) dataset. The exact parameter search spaces for all the classifiers are listed online [Res21].

Figure 6.7 shows the results for 150 serial iterations and an average of 30 runs. Optuna performs better than *MetaTuner* on iris dataset and breast cancer Wisconsin (diagnostic) dataset. *MetaTuner* performs better than Optuna on the wine recognition dataset. *MetaTuner* performs better than the Hyperopt and SMAC on all three datasets The results show that the *MetaTuner*

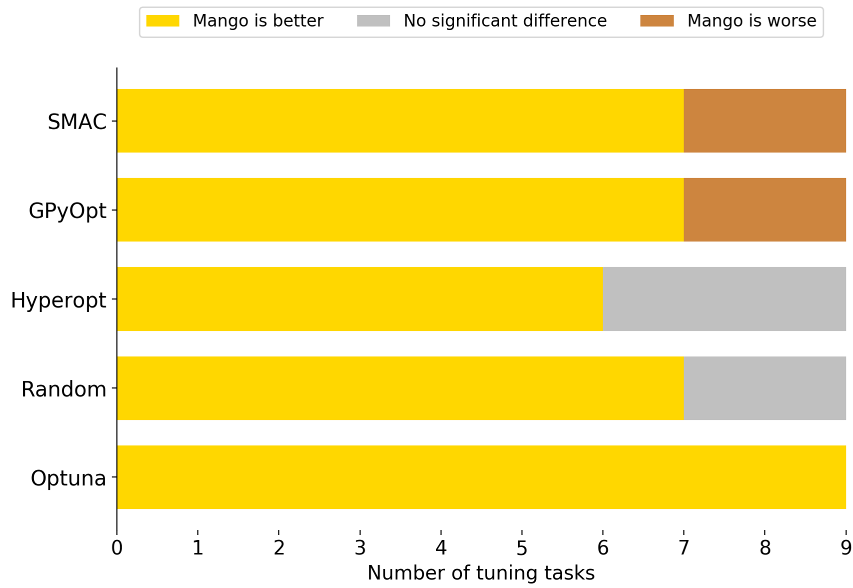


(a) Sequential optimization

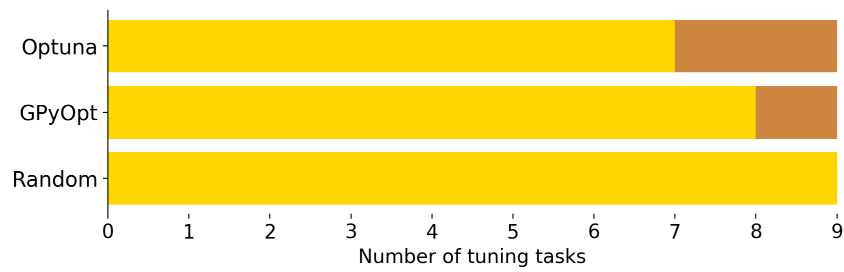


(b) Parallel optimization with 4 workers

Figure 6.5: Comparison of Mango to optimize functions.



(a) Sequential optimization



(b) Parallel optimization with 4 workers

Figure 6.6: Comparison of Mango to tune hyperparameters.

algorithm performs comparably with TPE and random forest surrogates that directly support conditional search spaces.

6.4.1.4 Optimizer sampling time

One disadvantage of GP surrogate is that it is computationally expensive due to the cubic complexity in the number of samples evaluated. Comparatively, TPE surrogate used in *Hyperopt* and *Optuna* is very inexpensive. In Mango, we have reduced the computational complexity by using Monte Carlo optimization of acquisition function instead of commonly used gradient-based methods like L-BFGS. We evaluate this feature by comparing various optimizers' sampling times in sequential, parallel, and CASH settings. Results are shown in Table 6.1. We did 30 runs of 80 iterations to calculate the average time taken per iteration. The sampling time depends on the complexity of the parameter space. For serial and parallel, we use the Xgboost's parameter space definition [Res21]. The CASH sampling time is shown for the Xgboost parameter space definition [Res21]. As expected TPE based optimizers are the fastest; however, Mango (GP) is significantly faster than the GPyOpt (GP) and SMAC (Random-forest). It is important to note that this comparison is inconsequential for hyperparameter tuning because the time taken to train ML models would dwarf the optimizer sampling time.

Summary: Mango outperforms other libraries in hyperparameter tuning for classifiers with mixed parameter (continuous, integer, and categorical) spaces. When evaluated for CASH problems, Mango's is competitive in performance to Optuna. In the case of functional benchmarks, Mango is competitive with the GpyOpt. However, Optuna performs poorly for functional benchmarks and tuning parameters for a single classifier. Further, GpyOpt performs poorly when tuning ML classifiers. Overall, Mango offers state-of-the-art algorithms having better or at par performance across settings.

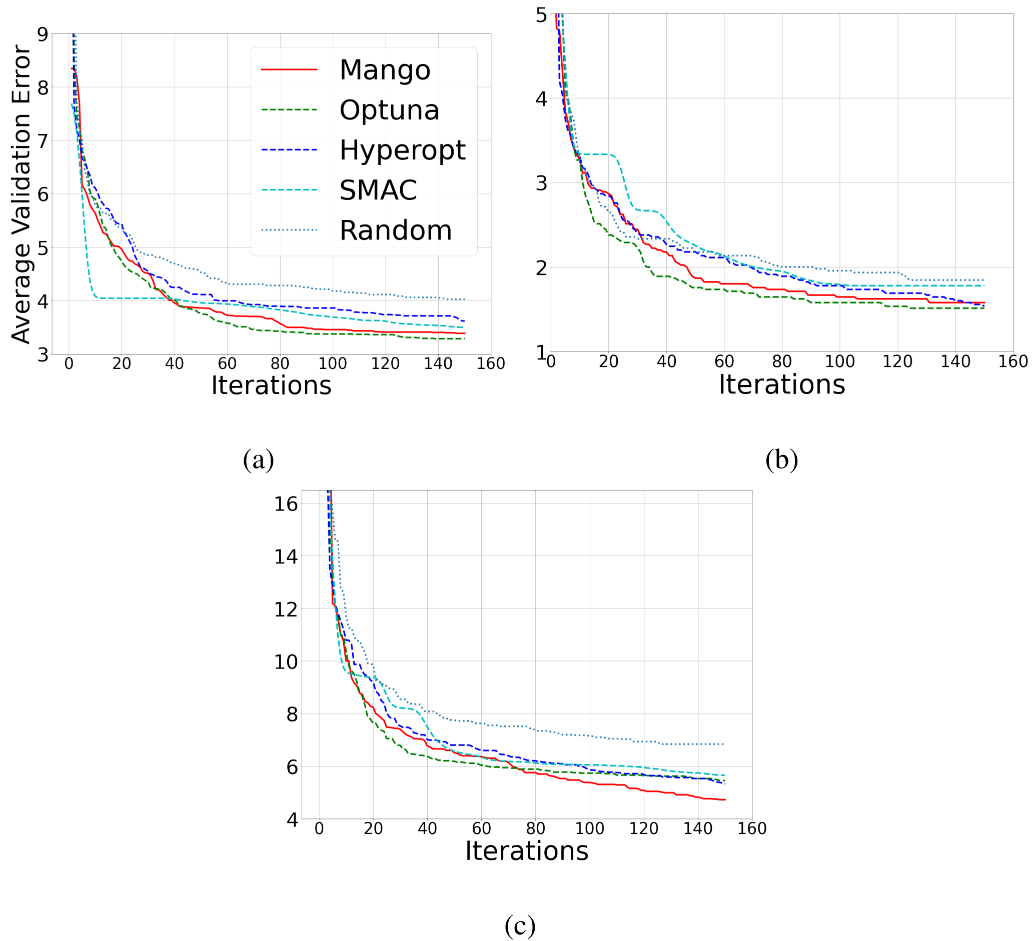


Figure 6.7: The comparison of Mango's *MetaTuner* for combined classifier selection and hyper-parameter optimization problem with other libraries. The evaluation uses five different classifiers (Xgboost, k-nearest neighbor, Support Vector Machines, decision tree, and neural network). Sub-figure (a) is for the Breast cancer dataset, sub-figure (b) the Iris plants dataset, and sub-figure (c) the Wine recognition dataset. Mango performs better than Hyperopt and SMAC and is competitive with Optuna.

Table 6.1: Wall clock time (sec) taken by optimizers to sample next evaluation in sequential, parallel, and CASH settings.

Optimizer (Surrogate)	**Sequential**	**Parallel**	**CASH**
Hyperopt (TPE)	0.001 ± 0.005	na	0.02 ± 0.001
Optuna (TPE)	0.07 ± 0.035	0.02 ± 0.006	0.02 ± 0.001
Mango (GP)	0.16 ± 0.008	0.12 ± 0.021	0.11 ± 0.002
GPyOpt (GP)	0.37 ± 0.051	1.76 ± 0.223	na
SMAC (Random forest)	0.70 ± 0.046	na	0.94 ± 0.037

6.4.2 Case Study: Bug Hunting in Design Verification of Integrated Circuits

The goal of design verification of integrated circuits (ICs) is to test the functionality correctness by generating input signals and evaluating the resulting output against the expected values. Modern ICs may contain billions of devices, so manual design verification is no longer feasible to verify all possible functionality. Standard practice in design verification is to generate the test signal candidates using constrained-random stimulus [Meh18]. The random input generation is controlled to allow a rich and diverse set covering the desired functionality. These inputs are simulated and monitored for bugs in the design. The bugs are then analyzed, fixed, and the entire process is repeated to verify the updated design. The input space for design verification is astronomically large, using a lot of computing using the random search. It is evident from the fact that the verification process accounts for a large fraction (50 %) of the total compute budget during development [Meh18].

At Arm, we are using ML to increase design verification efficiency. ML models are trained to classify test candidates likely to find bugs. The test candidates are then passed through the ML filter to select the tests with a high probability of failure. This process, called bug hunting ML flow, has been deployed in production and has been shown to increase the efficiency, measured as compute cycles used to find the same number of bugs by 40 %. The overall workflow is shown in Figure 6.1. The bug hunting workflow requires ML models to be frequently re-trained as the

design is updated or the test bench that generates the test candidates is modified. We also prefer training ML classifiers on the *entire dataset to avoid partial training errors* when comparing the hyperparameters. Overall our goal is to ensure that the compute budget for training ML models does not grow and eat into the gains made in the verification process. Hence our inclination for Bayesian optimization to reduce training iterations. Besides, we required the following features in the tuning library:

Deployment dependencies: The bug hunting workflow is implemented on a computing cluster using Dask distributed framework. Therefore, ideally, the hyperparameter tuning library should have the capability of being integrated with Dask without significant external dependencies. Furthermore, the library's compatibility with Scikit-learn's estimator interface would ease the integration due to the existing usage of a similar interface.

Runtime failures: Due to a cluster deployment, it is required that the library should expose abstractions to discarding the failed evaluations due to failed jobs, communication issues, or incorrect parameter values. This is critical to reducing the manual maintenance/debugging time in deployment.

CASH Problem Multiple ML classifiers are re-trained every time the training event is triggered, and the best model is chosen based on a custom metric.

The *RandomizedSearchCV* from Dask-ML partially supported these features and was used in the past ML production pipeline. However, the key missing features were efficient search and CASH. Mango provided all the required features with flexible, lightweight architecture, allowing scheduling on Dask without additional dependencies. Mango was integrated into the production ML pipeline to tune the ML models used for **Arm's ICs designs'** verification process. An extensive evaluation using Mango on six proprietary design verification benchmark datasets (3 test benches for 2 different designs) in comparison to the RandomizedSearchCV from Dask-ML showed that Mango reduces the model training iterations by an average of **45%** across all experiments, with the range being 23% - 69%.

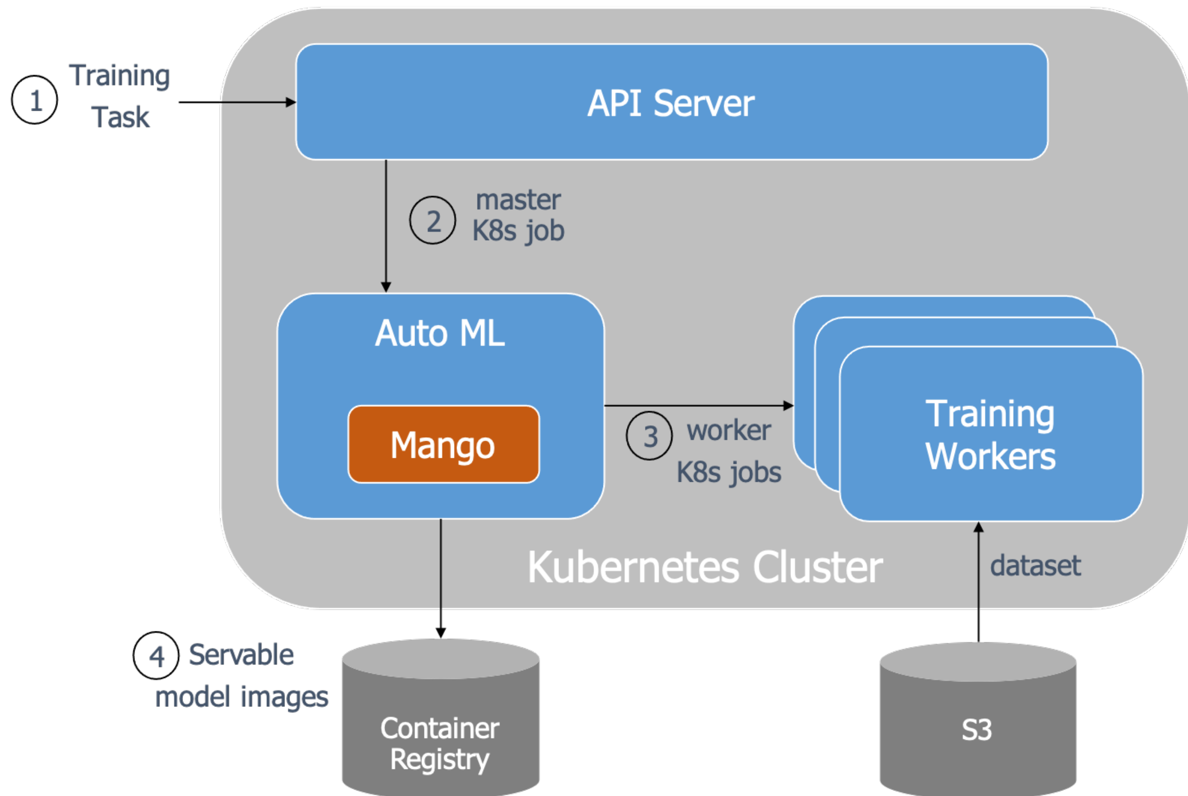


Figure 6.8: Workflow of the AutoML framework using Mango for hyperparameter tuning on the Kubernetes (K8s) cluster.

6.4.3 Case Study: AutoML Framework

At Arm, an AutoML platform was developed to provide a simple interface for non-data scientists to train and deploy ML models. The platform is deployed on a Kubernetes cluster. The AutoML framework uses the Kubernetes Jobs API to orchestrate distributed hyperparameter tuning. The hindrances in adopting Katib and Polyaxon hyperparameter tuning frameworks built on top of Kubernetes is their dependencies on components like API server, database, and persistent storage volumes increasing the maintenance and development overhead substantially. Mango provided a lightweight and robust alternative with efficient search algorithms.

Figure 6.8 shows the process flow of AutoML platform. The process is initiated by a POST request to the RESTful API server with the training task's configuration data. The configuration

data includes the dataset reference from S3, training type (classification, forecasting, regression), target column, performance metric, etc. The API server authenticates the request, fetches the relevant metadata from the database, and starts a master AutoML process using the Kubernetes Jobs API (Step 2). The master AutoML process is responsible for orchestrating the training task and invoking Mango for hyperparameter tuning. Mango's flexible scheduler interface is used to create parallel ML training tasks using the Kubernetes Jobs API (Step 3). Once tuning is complete, the master AutoML process saves the best model deployed as a Docker image in a container registry (Step 4). The pseudo-code of Mango used by the AutoML framework is shown in Figure 6.3. We use a timeout and return the partial results to make progress on the search.

6.4.4 Case Study: Network Architecture Search for TinyML Platforms

Modern CPS/IoT applications are bringing ML classifiers to microcontroller class devices. These devices, dubbed TinyML devices, have stringent hardware constraints. As a result, the neural architecture search (NAS) needs to be optimized by target hardware specifications [FAM19] to balance accuracy and efficiency via hardware-aware NAS.

We show the use case of Mango to model the search for limited flash and RAM requirements. The search space Ω consists of neural network weights w, hyperparameters θ, network structure denoted as a directed acyclic graph (DAG) g with edges E and vertices V representing activation maps and common ML operations v (e.g., convolution, batch normalization, pooling, etc.) respectively, which act on V. The goal is to find a neural network that maximizes the hardware SRAM and flash usage within the device capabilities while minimizing the error metric.

$$f_{\text{opt}} = \lambda_1 f_{\text{error}}(\Omega) + \lambda_2 f_{\text{flash}}(\Omega) + \lambda_3 f_{\text{SRAM}}(\Omega) \tag{6.1}$$

where

$$f_{\text{error}}(\Omega) = \mathscr{L}_{\text{test}}(\Omega), \Omega = \{\{V,E\}, w, \theta, v\} \tag{6.2}$$

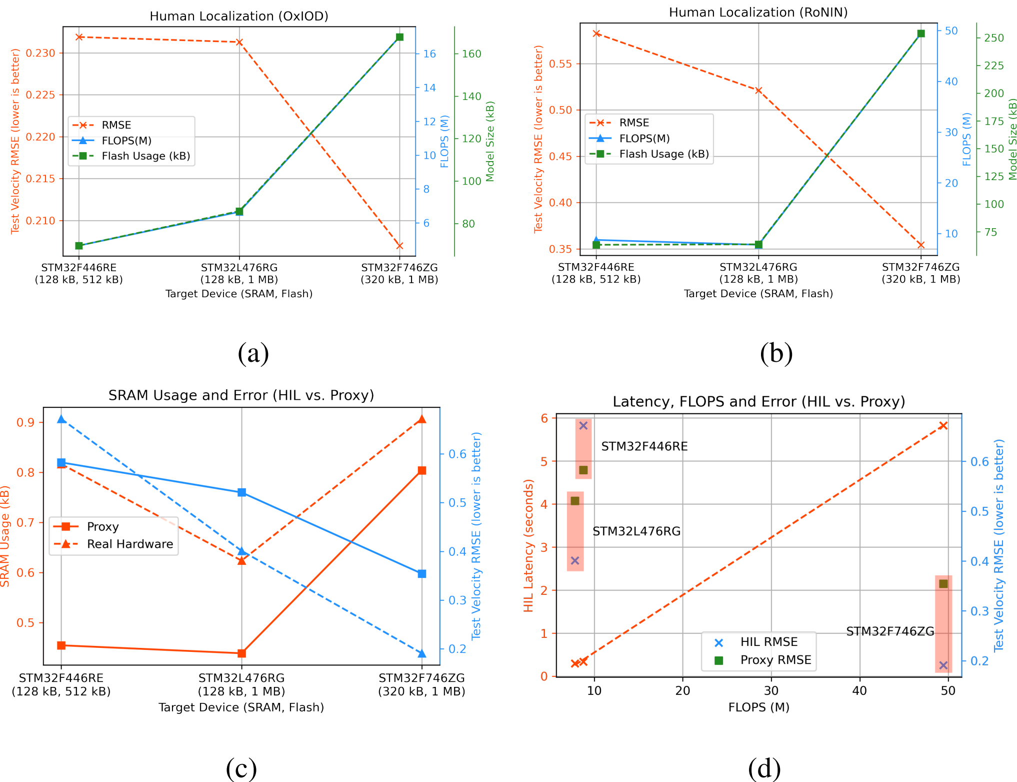


Figure 6.9: Performance of Mango for hardware-aware NAS for OxIOD and RoNIN datasets. Subgraphs (a) and (b) illustrate how Mango maximizes resource usage with looser compute and memory constraints to improve error metrics for three different hardware models. Subgraph (c) shows the difference in model size and error metric with and without hardware-in-the-loop (HIL) for the RoNIN dataset on three different hardware models. Subgraph (d) shows the relation between FLOPS and latency for the RoNIN dataset and the difference in error metric with and without HIL.

$$f_{\text{flash}}(\Omega) = \begin{cases} -\frac{||h_{\text{FB}}(w,\{V,E\})||_0}{\text{flash}_{\max}} \vee -\frac{\text{HIL information}}{\text{flash}_{\max}} \\ \infty, f_{\text{flash}}(\Omega) > \text{flash}_{\max} \end{cases} \tag{6.3}$$

$$f_{\text{SRAM}}(\Omega) = \begin{cases} -\frac{\max_{l\in[1,L]}\{||x_l||_0+||a_l||_0\}}{\text{SRAM}_{\max}} \vee -\frac{\text{HIL information}}{\text{SRAM}_{\max}} \\ \infty, f_{\text{SRAM}}(\Omega) > \text{SRAM}_{\max} \end{cases} \tag{6.4}$$

$$a = w \vee y, \quad y = \sum_{k=1}^{K} v_k g_k(x, w_k)$$

Error metric (e.g. RMSE or accuracy) serves as a proxy for the error characteristics $f_{\text{error}}(\Omega)$ of the model candidate. When real hardware is absent, we use the size of the flatbuffer model schema $h_{\text{FB}}(\cdot)$ [DDJ21] as a proxy for flash usage. Moreover, we use the standard RAM usage model as a proxy for SRAM usage $f_{\text{SRAM}}(\Omega)$, with intermediate layer-wise activation maps and tensors being stored in SRAM [FAM19]. When hardware-in-the-loop (HIL) is available, we obtain the SRAM and flash parameters directly from the target compiler and real-time operating system (RTOS). All hardware parameters are normalized by device capacity or target metrics.

Evaluation We evaluate our NAS formulation on three ARM Cortex-M microcontrollers with different compute and memory constraints. The task is to learn the velocity regression on the Oxford Inertial Odometry (OxIOD) [CZL20] and RoNIN datasets [HYF20] using a temporal convolutional network (TCN) having the parameter search space definition available here [Res21]. The performance of a classifier is measured by average test root-mean-square error (RMSE). Figure 6.9 illustrates the performance of Mango in finding optimal TCN networks on the two datasets. From Figure 6.9a and Figure 6.9b, it is evident that Mango attempts to exploit the full device capabilities within the resource constraints to minimize the error metric rather than choosing the smallest model every time. Thus, as compute capability improves, the network size for the target hardware also increases. In addition, we compare the performance between using HIL and using proxies to model device constraints and error metric in Figure 6.9c and Figure 6.9d. We observe that there is a constant offset between HIL and proxies in SRAM usage, stemming from model runtime interpreter and RTOS overhead on target hardware. However, the error metric can be opti-

mized further through HIL than proxies as compute constraints relax. The evaluation of latency in Figure 6.9d shows latency is proportional to FLOPS, thereby FLOPS servers a good latency proxy for microcontroller class devices.

6.5 Discussion

We presented the limitations of existing hyperparameter tuning frameworks hindering their adoption in production. Mango, a black-box optimization library with flexible architecture and state-of-the-art algorithms, was designed to address these limitations. Mango is evaluated on a set of functions and classifier tuning tasks to benchmark its superior performance. Finally, case studies are examined to highlight the adoption of Mango in production ML pipelines at Arm.

CHAPTER 7

Discussion and Future Work

The integration of machine learning components in CPS applications faces challenges at each stage of sense, infer/decide and actuate. We looked at the challenges of sensor data misalignment due to timestamp errors, training optimal model for inferences, and handling end-to-end delay variations during actuation. The final decision-making stage can have multiple steps, which are often difficult to optimize jointly. To address this, we explored the design of end-to-end controllers using deep reinforcement learning. To train optimal models, we introduced a new hyperparameter tuning library. Each of our presented works looks at bottlenecks at different stages of CPS pipelines, which can be extended further in many ways. Here, we discuss some of the future possible extensions.

7.1 Extending Timing Analysis

A unified timing system that provides synchronization solutions across cloud-to-edge via various complementary mechanisms is needed for modern distributed applications. We quantified the peripheral delays across smartphones. This can be extended across the different peripheral stacks of other edge devices of an entire distributed ecosystem of CPS, such as smartwatches, edge micro-controllers, etc. This will enable multiple devices and peripherals to synchronize, thus accurately providing a true shared notion of time across varying hardware platforms, vendors, smartphones, and other wireless and embedded devices.

An extended analysis of the introduced time-shifting data augmentation technique characterizing its limitations and providing recommended approaches for various time-series domains can

benefit immense CPS applications. Factors including dataset size and expected timing errors between training and deployment will likely play a key role in determining optimal augmentation.

7.2 A Vision of Timing Stack for Deep Reinforcement Learning

The introduced approach of including sensed timing state of the CPS system during deep-RL training showed a new way to rethink time-awareness in policies. These policies showed superior performance. But a major assumption in this analysis is that the policies can measure timing delays at runtime. Our research showed that the runtime timing measurements could account for jitters and noises. However, a proper timing stack is needed for CPS applications to convey to the application the expected sensing delays, execution latencies, and actuation delays. This is also important for CPS applications deployed across the network where the transfer of state and action may have to consider many delay variations depending on the network characteristics.

Most of the deep-RL training is done using a simulator. The timing delays in the simulator depend on the compute available and developer implementation, whereas in real-world deployments, time is a fundamental fabric of the system evolution. In a real deployment, the time also varies with hardware and deployment-specific characteristics like communication network, observation complexity, inference policy, etc. Bridging this gap between simulator time and real-world time is very important, given our analysis shows policy failures when time is not considered. We envision a very accurate timing stack to match different delays between training and deployment expectations for the future success of deep-RL.

7.3 Future Training Environments for CPS Applications

We consider the current success in deep-RL as the first generation of end-to-end controllers, which are easy to develop using past simulation efforts from the research community for Attari games and simplified robots (supported in Pybullet/Mujoco). The superior performance shown by these

end-to-end controllers offers a considerable promise for the complex control pipelines of CPS applications, as we showed for pan-tilt-zoom cameras. However, a significant hurdle in this direction is the lack of appropriate training environments in rich CPS settings to simulate multimodal sensing, real-world characters, their interactions, communication paradigms, and abstractions to integrate reinforcement learning algorithms.

Significant efforts are needed to develop rich training environments for broad adoption of the end-to-end controller with superior performance and low compute overheads. These training environments must be designed carefully to mimic the real deployment settings to bridge the gap between simulation and the real world.

7.4 Limitations of End-to-end Control

The end-to-end control has its limitations in explaining the output. Multi-stage control offers an output of intermediate steps which are often human-understandable and interpretable. For example, if a multi-stage pipeline in autonomous pan-tilt-zoom cameras is failing, we can analyze if the objects are detected, the tracking accuracy, and the controller separately. However, this intermediate analysis is not possible directly in an end-to-end control. We need new ways to explain the end-to-end control from the CPS perspective. What will explanation looks like is an essential topic of research itself.

Also, end-to-end control using deep-RL faces the challenge of modifying the tracking goal. This may require training a new policy from scratch. Finetuning or transfer learning can be explored as an alternative method to using pre-trained policies.

7.5 Possible Extension in Mango

There are a few features that can enhance the applicability of Mango. The implemented *MetaTuner* algorithm allows Mango to solve the CASH problem, which is an instance of the conditional

search. Naively handling general purpose conditional spaces using meta-variables in Gaussian Process is shown to under-perform [LDG17]. A new form of the conditional search can be realized by modifying the distance functions in the Gaussian Process kernel function [LDG17].

CHAPTER 8

Conclusion

Adopting learning-enabled components in CPS systems opens up venues to realize future applications working directly with rich multimodal datasets. We looked at the challenges of accurate timestamps of sensor data, variable delays during actuation, optimal classifier search for inference, and creation of training environments. When machine learning classifiers are part of a complex CPS application, the overall data flow from multiple stages impacts the classifier performance, which in turn decides the application's success or failure. Hence, when designing a new CPS application, we need to carefully analyze each stage and characterize the impact of the uncertainties and variations on the classifier's performance. The superior performance of machine learning classifiers cannot be seen in isolation from the sensing and actuation stages. The presence of timing uncertainties in sensor data can cripple the application's performance. The delay variations in actuation can result in complete task failure. We advocate that future CPS applications need to design approaches to characterize runtime uncertainties and variations carefully.

The different steps in conventional CPS control are being replaced by end-to-end control, among which deep-RL offers a promising alternative. However, deep-RL policies are very data-hungry. Thus to adopt such paradigms for real applications, rich training environments must be designed first. Often these training environments are created using simulators. Using simulators, we see the gap between simulation and reality decides the performance of end-to-end controllers in the real world. Finally, selecting an optimal classifier for the inference stage is challenging due to the scale of choices available and their hyperparameters. Accurate hyperparameter choice, along with the imposition of memory/compute/latency deployment constraints, is the key to selecting the

best performing classifiers for ubiquitous CPS devices such as microcontrollers.

CHAPTER 9

Appendix

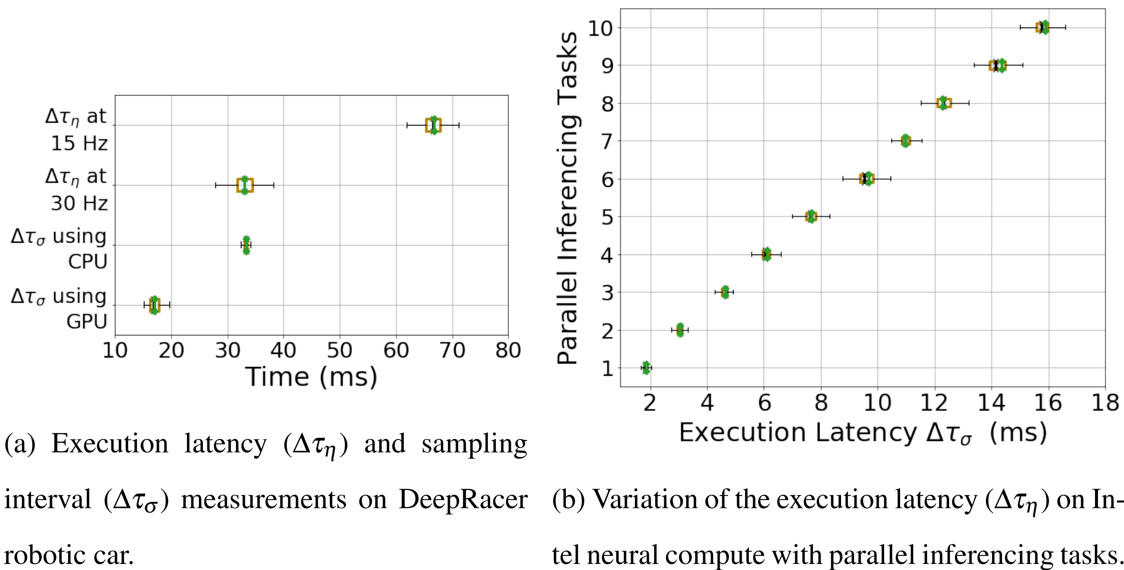
9.1 Delay Measurements on Different Hardware Platforms

Figure 9.1a shows the delay measurements of the navigational policy using the Deepracer robotic car. The execution latency ($\Delta\tau_\eta$) is dependent on the choice of the hardware resource at runtime (GPU vs. CPU). Deepracer camera supports a sampling rate of 15Hz and 30Hz. The sampling interval ($\Delta\tau_\sigma$) is 20-45 ms in the 30Hz frame rate setting and 62-71ms in the 15Hz setting, respectively. We also measure the execution latency when the complexity of the neural network is increased by adding more CNN layers and in the presence of other inference tasks. The complexity of the neural network and the required compute requirements vary with additional CNN layers that can happen when an application selects a more complex network to achieve better inference accuracy. The case for computing in the presence of other inference tasks can show up in two ways: (i) multiple models need to run for the same robotic application; (ii) an edge server acting like a machine learning model server for several applications.

We analyze the execution latency of shallow neural networks on a low power microcontroller

Table 9.1: Execution latency ($\Delta\tau_\eta$) on GAP8 increases with the increase in the number of CNN layers in the neural network.

Num. of CNN layers	2	3	4
Network parameters	54k	157k	267k
Execution Latency	7.5ms	19.75ms	55.85ms



(a) Execution latency ($\Delta\tau_\eta$) and sampling interval ($\Delta\tau_\sigma$) measurements on DeepRacer robotic car.

(b) Variation of the execution latency ($\Delta\tau_\eta$) on Intel neural compute with parallel inferencing tasks.

Figure 9.1: Delay measurements on deepracer and Intel neural compute stick. The mean is shown in green. The back 'x' marker shows the median of IQR.

device (GAP8) and edge accelerator (Intel neural compute stick 2). GAP8 is a milli-watt range microcontroller device having a dedicated CNN accelerator enabling battery-operated edge devices with rich analytics capabilities. We analyze the execution latency of the neural network trained using the MNIST dataset on GAP8 as the number of CNN layers is increased from 2 to 4. The network consists of the CNN layers followed by an output layer. Table 9.1 shows the increase in inferencing latency from 7.5ms to 55.90ms on increasing CNN layers from 2 to 4. Average results for 10 runs are reported. The variations across individual runs are very small (few microseconds).

Intel neural compute stick 2 (NCS2) is a deep learning processor on a USB stick that provides faster neural network inference capabilities to Raspberry Pi like edge devices. We analyze the variation in execution latency of a neural network with 2 CNN layers and an output layer trained using the MNIST dataset in the presence of other inference tasks on NCS2. We simulation parallel inference tasks by hosting the same neural network multiple times, all of which are running parallel. Figure 9.1b shows the results. As seen, when more parallel tasks are using the same hardware resource, in this case, the NCS2, the execution latency can increase up to 10x times.

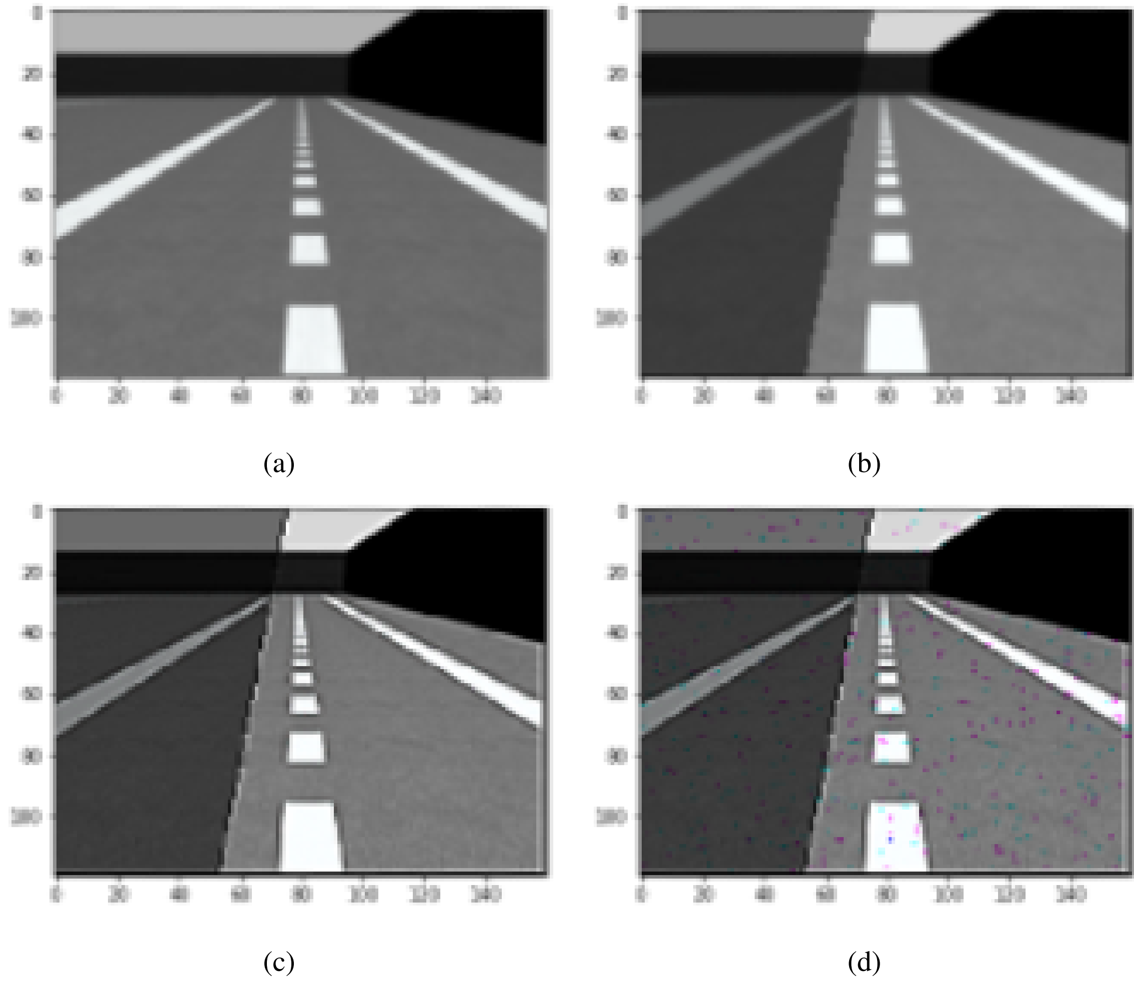


Figure 9.2: Image augmentations applied to enable successful Sim2Real transfer. (a) Original Image, (b) Random Shadows, (c) Random Shadow + Sharpen, (d) Random Shadow + Sharpen + Random noise

9.2 Additional Details on HalfCheetah and Ant Tasks

We vary the state transition delays ($\Delta\tau_\sigma$ and $\Delta\tau_\eta$) in the simulator when training the policies for HalfCheetah and Ant Tasks. For HalfCheetah and Ant Tasks, PyBullet simulator evolves physics at a fixed time (Sim_{Time}) of 4.12 ms for each action. When the agent acts, the simulation is advanced by applying the past action for $\frac{\Delta\tau_\eta}{Sim_{Time}}$ simulation steps, and the most recent action for $\frac{\Delta\tau_\sigma - \Delta\tau_\eta}{Sim_{Time}}$ simulation steps.

State space. The default state of *HalfCheetahBulletEnv-v0* and *AntBulletEnv-v0* is used for domain randomization policies. To train a policy with time in the state, $\Delta\tau_\sigma$ and $\Delta\tau_\eta$ are directly added as another state thereby increasing the input dimensions by 2.

Actions. The actions available in default environments for *HalfCheetahBulletEnv-v0* and *AntBulletEnv-v0* is used.

Reward function. For every agent's action, the simulation is advanced for multiple simulation steps depending on the $\Delta\tau_\eta$ and $\Delta\tau_\sigma$. This results in multiple reward calculation for each simulation step. For every agent's action, we take the average of the rewards from all simulation steps. This ensures the reward is on the same scale as the default environments.

Hyperparameters. For training, we use the default hyperparameters from OpenAI Baselines PPO implementation other than making the following changes. We used a learning rate of $3 * 10^{-4}$ and 10,000 steps between policy update.

9.3 Additional Details on Autonomous Vehicle Task

State space. The images (160x120) are directly used to train the domain randomization (DR) policy. For time in state (TS) policy, the sampling interval and execution latency are fused with images using the approach of multimodal fusion.

Image augmentations. We apply augmentations to the image by modifying its brightness randomly, adding random shadows, sharpen and random noises during the training of both DR and

TS policies so as to enable successful transfer to the real track in the presence of sensor noises. Without image augmentations, we observe an inferior Sim2Real transfer. Figure 9.2 visualizes the image augmentations.

Actions. The action space of the agent consists of speed and steering angle. The agent is given a choice of 15 actions which consists of 3 different speeds (1.2m/s,1.5m/s, 1.8m/s) and 5 steering angels which are (-30,-15,0,15,30) in degrees.

Reward function. The reward signal is calculated based on the distance of the car from the centerline of the track. The highest reward of 1.1 is given when the center of the car matches the centerline, which is scaled to zero when the distance from the centerline makes the car close to offtrack. The agent is rewarded more for high-speed actions. A negative reward of -30 is given to the agent when the car goes off the track.

Hyperparameters. We use the default hyperparameters from OpenAI Baselines PPO implementation other than making the following changes. We fixed 7,000 Steps between policy update and an entropy coefficient of 0.001.

9.4 Learning Curves of Worst-Case Delay Policy

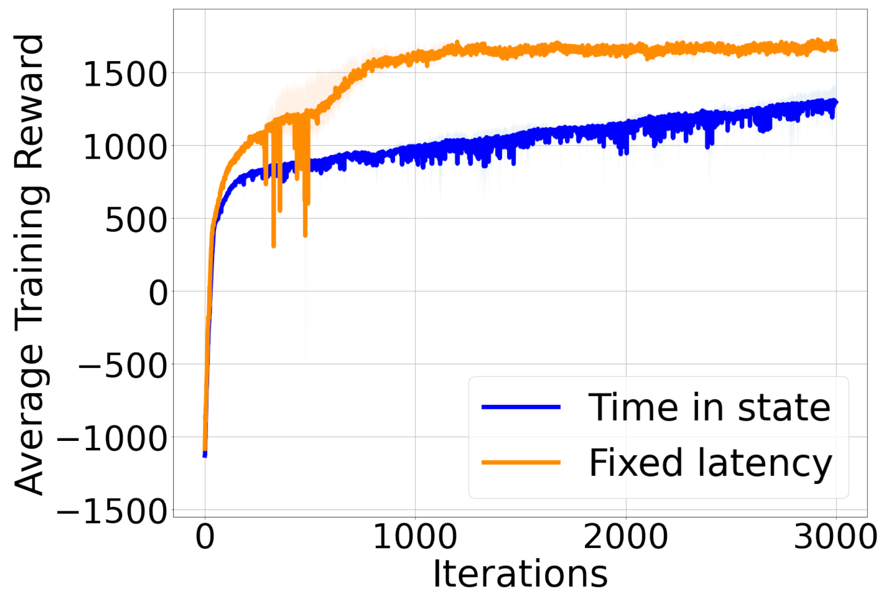


Figure 9.3: The learning curves of Time-in-state and Fixed latency (worst case = 5x4.12 ms) for HalfCheetah task. Fixed latency converges faster. Time-in-state is trained across a vast range of delay variations, and as seen in Figure 4.11 it outperforms worst case policies across a range of delay variations.

REFERENCES

[ABC20] OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. "Learning dexterous in-hand manipulation." *The International Journal of Robotics Research*, **39**(1):3–20, 2020.

[AM17] Roy J Adams and Benjamin M Marlin. "Learning time series detection models from temporally imprecise labels." *Proceedings of machine learning research*, **54**:157, 2017.

[AM18] Roy Adams and Benjamin M Marlin. "Learning Time Series Segmentation Models from Temporally Imprecise Labels." In *UAI*, pp. 135–144, 2018.

[Ard22] Arducam. "Embedded vision for raspberry pi, jetson, Arduino and more.", Apr 2022.

[AS17] Fatima M Anwar and Mani B Srivastava. "Precision time protocol over LR-WPAN and 6LoWPAN." In *2017 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS)*, pp. 1–6. IEEE, 2017.

[ASY19] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. "Optuna: A next-generation hyperparameter optimization framework." In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2623–2631, 2019.

[Aut] The Dask ML Authors. "Dask-ML." `url="https://ml.dask.org/",`. Accessed: 2021-1-29.

[aut16a] The GPyOpt authors. "GPyOpt: A Bayesian Optimization framework in python." `http://github.com/SheffieldML/GPyOpt`, 2016.

[aut16b] The Skopt authors. "Skopt: scikit-optimize." `https://scikit-optimize.github.io/`, 2016.

[AWS22] Saad Abbasi, Alexander Wong, and Mohammad Javad Shafiee. "Maple: Microprocessor a priori for latency estimation." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2747–2756, 2022.

[Bai93] Leemon C Baird III. "Advantage updating." Technical report, WRIGHT LAB WRIGHT-PATTERSON AFB OH, 1993.

[Bal21] Michael Balaban. "Deep Learning Hardware Deep Dive – RTX 3090, RTX 3080, and RTX 3070.", Aug 2021.

[BBB11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. "Algorithms for hyper-parameter optimization." *Advances in neural information processing systems*, **24**:2546–2554, 2011.

[BCP16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. "Openai gym." *arXiv preprint arXiv:1606.01540*, 2016.

[Ben03] Samy Bengio. "An asynchronous hidden markov model for audio-visual speech recognition." In *Advances in Neural Information Processing Systems*, pp. 1237–1244, 2003.

[Beq03] B Wayne Bequette. *Process control: modeling, design, and simulation*. Prentice Hall Professional, 2003.

[BGO16] Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos, and Ben Upcroft. "Simple online and realtime tracking." In *2016 IEEE International Conference on Image Processing (ICIP)*, pp. 3464–3468, 2016.

[BMG19] Bharathan Balaji, Sunil Mallya, Sahika Genc, Saurabh Gupta, Leo Dirac, Vineet Khare, Gourav Roy, Tao Sun, Yunzhe Tao, Brian Townsend, et al. "DeepRacer: Educational Autonomous Racing Platform for Experimentation with Sim2Real Reinforcement Learning." *arXiv preprint arXiv:1911.01562*, 2019.

[BMG20] Bharathan Balaji, Sunil Mallya, Sahika Genc, Saurabh Gupta, Leo Dirac, Vineet Khare, Gourav Roy, Tao Sun, Yunzhe Tao, Brian Townsend, et al. "Deepracer: Autonomous racing platform for experimentation with sim2real reinforcement learning." In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2746–2754. IEEE, 2020.

[BVS07] Keni Bernardin, Florian Van De Camp, and Rainer Stiefelhagen. "Automatic person detection and tracking using fuzzy controlled active cameras." In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–8. IEEE, 2007.

[BXD20] Niccolò Bisagno, Alberto Xamin, Francesco De Natale, Nicola Conci, and Bernhard Rinner. "Dynamic Camera Reconfiguration with Reinforcement Learning and Stochastic Methods for Crowd Surveillance." *Sensors*, **20**(17):4691, 2020.

[BYB09] Boris Babenko, Ming-Hsuan Yang, and Serge Belongie. "Visual tracking with online multiple instance learning." In *2009 IEEE Conference on computer vision and Pattern Recognition*, pp. 983–990. IEEE, 2009.

[BYC13] James Bergstra, Dan Yamins, and David D Cox. "Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms." In *Proceedings of the 12th Python in science conference*, pp. 13–20. Citeseer, 2013.

[CB16] Erwin Coumans and Yunfei Bai. "Pybullet, a python module for physics simulation for games, robotics and machine learning." 2016.

[CB19] Erwin Coumans and Yunfei Bai. "PyBullet, a Python module for physics simulation for games, robotics and machine learning." `http://pybullet.org`, 2016–2019.

[CLK11] Shengyong Chen, Youfu Li, and Ngai Ming Kwok. "Active vision in robotic systems: A survey of recent developments." *The International Journal of Robotics Research*, **30**(11):1343–1377, 2011.

[CRM00] Dorin Comaniciu, Visvanathan Ramesh, and Peter Meer. "Real-time tracking of non-rigid objects using mean shift." In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2000 (Cat. No. PR00662)*, volume 2, pp. 142–149. IEEE, 2000.

[CSB15] Gengjie Chen, Pierre-Luc St-Charles, Wassim Bouachir, Guillaume-Alexandre Bilodeau, and Robert Bergevin. "Reproducible evaluation of pan-tilt-zoom tracking." In *2015 IEEE International Conference on Image Processing (ICIP)*, pp. 2055–2059. IEEE, 2015.

[CSH19] Sandeep Chinchali, Apoorva Sharma, James Harrison, Amine Elhafsi, Daniel Kang, Evgenya Pergament, Eyal Cidon, Sachin Katti, and Marco Pavone. "Network Offloading Policies for Cloud Robotics: A Learning-Based Approach." In *Proceedings of Robotics: Science and Systems*, FreiburgimBreisgau, Germany, June 2019.

[CXL20] Baiming Chen, Mengdi Xu, Liang Li, and Ding Zhao. "Delay-Aware Model-Based Reinforcement Learning for Continuous Control." *arXiv preprint arXiv:2005.05440*, 2020.

[CZH18] Han Cai, Ligeng Zhu, and Song Han. "Proxylessnas: Direct neural architecture search on target task and hardware." *arXiv preprint arXiv:1812.00332*, 2018.

[CZL20] Changhao Chen, Peijun Zhao, Chris Xiaoxuan Lu, Wei Wang, Andrew Markham, and Niki Trigoni. "Deep-Learning-based Pedestrian Inertial Navigation: Methods, Data Set, and On-Device Inference." *IEEE Internet of Things Journal*, **7**(5):4431–4441, 2020.

[DCA20] Lukasz Dudziak, Thomas Chau, Mohamed Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas Lane. "Brp-nas: Prediction-based nas using gcns." *Advances in Neural Information Processing Systems*, **33**:10480–10490, 2020.

[DDJ21] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. "TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems." *Proceedings of Machine Learning and Systems*, **3**, 2021.

[DKB14] Thomas Desautels, Andreas Krause, and Joel W Burdick. "Parallelizing exploration-exploitation tradeoffs in Gaussian process bandit optimization." *The Journal of Machine Learning Research*, **15**(1):3873–3923, 2014.

[DL00] Stéphane Dupont and Juergen Luettin. "Audio-visual speech modeling for continuous speech recognition." *IEEE transactions on multimedia*, **2**(3):141–151, 2000.

[DMC16] Ian Dewancker, Michael McCourt, Scott Clark, Patrick Hayes, Alexandra Johnson, and George Ke. "A strategy for ranking optimization methods using multiple criteria." In *Workshop on Automatic Machine Learning*, pp. 11–20. PMLR, 2016.

[DP12] Didier Dubois and Henri Prade. *Possibility theory: an approach to computerized processing of uncertainty*. Springer Science & Business Media, 2012.

[DR17] Sandeep D'souza and Ragunathan Raj Rajkumar. "Time-based coordination in geo-distributed cyber-physical systems." In *9th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, 2017.

[EGE02] Jeremy Elson, Lewis Girod, and Deborah Estrin. "Fine-grained network time synchronization using reference broadcasts." *ACM SIGOPS Operating Systems Review*, **36**(SI):147–163, 2002.

[EL02] John Eidson and Kang Lee. "IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems." In *Sensors for Industry Conference, 2002. 2nd ISA/IEEE*, pp. 98–105. Ieee, 2002.

[ESS15] Andreas Eitel, Jost Tobias Springenberg, Luciano Spinello, Martin Riedmiller, and Wolfram Burgard. "Multimodal deep learning for robust RGB-D object recognition." In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 681–687. IEEE, 2015.

[FAM19] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul N Whatmough. "SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers." *Advances in Neural Information Processing Systems*, **32**, 2019.

[FBA16] Lex Fridman, Daniel E Brown, William Angell, Irman Abdić, Bryan Reimer, and Hae Young Noh. "Automated synchronization of driving data using vibration and steering events." *Pattern Recognition Letters*, **75**:9–15, 2016.

[FKE15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. "Efficient and robust automated machine learning." In *Advances in neural information processing systems*, pp. 2962–2970, 2015.

[FRC18] Eric Flamand, Davide Rossi, Francesco Conti, Igor Loi, Antonio Pullini, Florent Rotenberg, and Luca Benini. "GAP-8: A RISC-V SoC for AI at the Edge of the IoT." In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 1–4. IEEE, 2018.

[GDH16] Javier González, Zhenwen Dai, Philipp Hennig, and Neil Lawrence. "Batch bayesian optimization via local penalization." In *Artificial intelligence and statistics*, pp. 648–657, 2016.

[GE01] Lewis Girod and Deborah Estrin. "Robust range estimation using acoustic and multimodal sensing." In *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No. 01CH37180)*, volume 3, pp. 1312–1320. IEEE, 2001.

[GH20] Eduardo C Garrido-Merchán and Daniel Hernández-Lobato. "Dealing with categorical and integer-valued variables in bayesian optimization with gaussian processes." *Neurocomputing*, **380**:20–35, 2020.

[GHY22] Wei Gao, Qinghao Hu, Zhisheng Ye, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. "Deep Learning Workload Scheduling in GPU Datacenters: Taxonomy, Challenges and Vision." *arXiv preprint arXiv:2205.11913*, 2022.

[GKS03] Saurabh Ganeriwal, Ram Kumar, and Mani B Srivastava. "Timing-sync protocol for sensor networks." In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pp. 138–149. ACM, 2003.

[Goo19] Google. "Android Sensors." `https://source.android.com/devices/sensors/index.html`, 2019. Accessed: 2019-03-26.

[GP18] Matthew Groves and Edward O Pyzer-Knapp. "Efficient and Scalable Batch Bayesian Optimization Using K-Means." *arXiv preprint arXiv:1806.01159*, 2018.

[Grz17] Marek Grzes. "Reward shaping in episodic reinforcement learning." 2017.

[GSM17] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. "Google vizier: A service for black-box optimization." In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 1487–1495, 2017.

[GVH14] Oguzhan Gencoglu, Tuomas Virtanen, and Heikki Huttunen. "Recognition of acoustic events using deep neural networks." In *2014 22nd European Signal Processing Conference (EUSIPCO)*, pp. 506–510. IEEE, 2014.

[Hal13] Nicolas Halbwachs. *Synchronous programming of reactive systems*, volume 215. Springer Science & Business Media, 2013.

[HCM14] João F Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista. "High-speed tracking with kernelized correlation filters." *IEEE transactions on pattern analysis and machine intelligence*, **37**(3):583–596, 2014.

[HDV17] Danijar Hafner, James Davidson, and Vincent Vanhoucke. "Tensorflow agents: Efficient batched reinforcement learning in tensorflow." *arXiv preprint arXiv:1709.02878*, 2017.

[HG19] Tyler Highlander and John Gallagher. "Attention Neural Networks for Pan-Tilt-Zoom Control with Active Hand-Off." In *2019 7th International Conference on Robot Intelligence Technology and Applications (RiTA)*, pp. 130–135. IEEE, 2019.

[HHL11] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. "Sequential model-based optimization for general algorithm configuration." In *International conference on learning and intelligent optimization*, pp. 507–523. Springer, 2011.

[HHS17] Syed Monowar Hossain, Timothy Hnat, Nazir Saleheen, Nusrat Jahan Nasrin, Joseph Noor, Bo-Jhang Ho, Tyson Condie, Mani Srivastava, and Santosh Kumar. "mCerebrum: A Mobile Sensing Software Platform for Development and Validation of Digital Biomarkers and Interventions." In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, p. 7. ACM, 2017.

[HLD19] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. "Learning agile and dynamic motor skills for legged robots." *Science Robotics*, **4**(26):eaau5872, 2019.

[HNX07] Joo P Hespanha, Payam Naghshtabrizi, and Yonggang Xu. "A survey of recent results in networked control systems." *Proceedings of the IEEE*, **95**(1):138–162, 2007.

[HPL21] Charles Hamesse, Benoît Pairet, Rihab Lahouli, Timothée Fréville, and Rob Haelterman. "Simulation of Pan-Tilt-Zoom Tracking for Augmented Reality Air Traffic Control." In *2021 International Conference on 3D Immersion (IC3D)*, pp. 1–5. IEEE, 2021.

[HYF20] 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)*, pp. 3146–3152. IEEE, 2020.

[Hyp19] Hyperopt. "hyperopt-mongo-worker." `https://hyperopt.github.io/hyperopt/`, 2019. Accessed: 2021-1-29.

[HZL17] Samer Hanoun, James Zhang, Vu Le, Burhan Khan, Michael Johnstone, Michael Fielding, Asim Bhatti, Doug Creighton, and Saeid Nahavandi. "A framework for designing active Pan-Tilt-Zoom (PTZ) camera networks for surveillance applications." In *2017 Annual IEEE International Systems Conference (SysCon)*, pp. 1–6. IEEE, 2017.

[Inc18] Superpowered Inc. "Android Audio's 10 Millisecond Problem: The Android Audio Path Latency Explainer." `https://superpowered.com/androidaudiopathlatency`, 2018. Accessed: 2019-03-26.

[Int] Intel. "Intel Neural Compute Stick 2 Product Specifications.".

[JAG17] Rodolphe Jenatton, Cedric Archambeau, Javier González, and Matthias Seeger. "Bayesian optimization with tree-structured dependencies." In *International Conference on Machine Learning*, pp. 1655–1664. PMLR, 2017.

[JLM15] Kasthuri Jayarajah, Youngki Lee, Archan Misra, and Rajesh Krishna Balan. "Need accurate user behaviour?: pay attention to groups!" In *Proceedings of the 2015 ACM international joint conference on pervasive and ubiquitous computing*, pp. 855–866. ACM, 2015.

[JPG13] Prem Prakash Jayaraman, Charith Perera, Dimitrios Georgakopoulos, and Arkady Zaslavsky. "Efficient opportunistic sensing using mobile collaborative platform mosden." In *9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pp. 77–86. IEEE, 2013.

[JT16] Kevin Jamieson and Ameet Talwalkar. "Non-stochastic best arm identification and hyperparameter optimization." In *Artificial Intelligence and Statistics*, pp. 240–248, 2016.

[Kan11] Salil S Kanhere. "Participatory sensing: Crowdsourcing data from mobile smartphones in urban spaces." In *2011 IEEE 12th International Conference on Mobile Data Management*, volume 2, pp. 3–6. IEEE, 2011.

[KB13] Fahim Kawsar and AJ Brush. "Home computing unplugged: why, where and when people use different connected devices at home." In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*, pp. 627–636. ACM, 2013.

[KBM15] Aggelos K Katsaggelos, Sara Bahaadini, and Rafael Molina. "Audiovisual fusion: Challenges and new approaches." *Proceedings of the IEEE*, **103**(9):1635–1653, 2015.

[KE03] Konstantinos V Katsikopoulos and Sascha E Engelbrecht. "Markov decision processes with delays and asynchronous cost collection." *IEEE transactions on automatic control*, **48**(4):568–574, 2003.

[Kha18] J Khari. "AmpMe plans to kill Bluetooth speakers by syncing music between smartphones." `https://venturebeat.com/2018/07/03/ampme-plans-to-kill-bluetooth-speakers-by-syncing-music-between-sma` 2018. Accessed: 2019-06-28.

[KKP19] Dongchil Kim, Kyoungman Kim, and Sungjoo Park. "Automatic PTZ camera control based on deep-Q network in video surveillance system." In *2019 International Conference on Electronics, Information, and Communication (ICEIC)*, pp. 1–3. IEEE, 2019.

[KMD10] Sylvain Koos, Jean-Baptiste Mouret, and Stéphane Doncieux. "Crossing the reality gap in evolutionary robotics by promoting transferable controllers." In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pp. 119–126, 2010.

[Kyr21] Christos Kyrkou. "C3 Net: end-to-end deep learning for efficient real-time visual active camera control." *Journal of Real-Time Image Processing*, pp. 1–13, 2021.

[KZX11] Matthew Keally, Gang Zhou, Guoliang Xing, Jianxin Wu, and Andrew Pyles. "Pbn: towards practical activity recognition using smartphone-based body sensor networks." In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pp. 246–259. ACM, 2011.

[LDG17] Julien-Charles Lévesque, Audrey Durand, Christian Gagné, and Robert Sabourin. "Bayesian optimization for conditional hyperparameter spaces." In *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 286–293. IEEE, 2017.

[LHP15] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. "Continuous control with deep reinforcement learning." *arXiv preprint arXiv:1509.02971*, 2015.

[LJD17] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. "Hyperband: A novel bandit-based approach to hyperparameter optimization." *The Journal of Machine Learning Research*, **18**(1):6765–6816, 2017.

[LLC21] Hayeon Lee, Sewoong Lee, Song Chong, and Sung Ju Hwang. "Hardware-adaptive efficient latency prediction for nas via meta-learning." *Advances in Neural Information Processing Systems*, **34**:27016–27028, 2021.

[LLN18] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. "Tune: A research platform for distributed model selection and training." *arXiv preprint arXiv:1807.05118*, 2018.

[LMC21] Ezequiel López-Rubio, Miguel A Molina-Cabello, Francisco M Castro, Rafael M Luque-Baena, Manuel J Marín-Jiménez, and Nicolás Guil. "Anomalous object detection by active search with PTZ cameras." *Expert Systems with Applications*, **181**:115150, 2021.

[LMT01] Feng-Li Lian, James Moyne, and Dawn Tilbury. "Time delay modeling and sample time selection for networked control systems." In *Proceedings of ASME-DSC*, volume 20, pp. 11–16. DCS New York, 2001.

[LPR13] Neal Lathia, Veljko Pejovic, Kiran K Rachuri, Cecilia Mascolo, Mirco Musolesi, and Peter J Rentfrow. "Smartphones for large-scale behavior change interventions." *IEEE Pervasive Computing*, **12**(3):66–73, 2013.

[LR90] Rogelio Luck and Asok Ray. "An observer-based compensator for distributed delays." *Automatica*, **26**(5):903–908, 1990.

[LRS15a] Patrick Lazik, Niranjini Rajagopal, Oliver Shih, Bruno Sinopoli, and Anthony Rowe. "ALPS: A bluetooth and ultrasound platform for mapping and localization." In *Proceedings of the 13th ACM conference on embedded networked sensor systems*, pp. 73–84. ACM, 2015.

[LRS15b] Patrick Lazik, Niranjini Rajagopal, Bruno Sinopoli, and Anthony Rowe. "Ultrasonic time synchronization and ranging on smartphones." In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 108–118. IEEE, 2015.

[LSW15] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. "PulseSync: An efficient and scalable clock synchronization protocol." *IEEE/ACM Transactions on Networking (TON)*, **23**(3):717–727, 2015.

[LSZ19] Wenhan Luo, Peng Sun, Fangwei Zhong, Wei Liu, Tong Zhang, and Yizhou Wang. "End-to-end active object tracking and its real-world deployment via reinforcement learning." *IEEE transactions on pattern analysis and machine intelligence*, **42**(6):1317–1332, 2019.

[LXS11] Liqun Li, Guoliang Xing, Limin Sun, Wei Huangfu, Ruogu Zhou, and Hongsong Zhu. "Exploiting FM radio data system for adaptive clock calibration in sensor networks." In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pp. 169–182. ACM, 2011.

[MCH19] Artem Molchanov, Tao Chen, Wolfgang Hönig, James A Preiss, Nora Ayanian, and Gaurav S Sukhatme. "Sim-to-(multi)-real: Transfer of low-level robust control policies to multiple quadrotors." *arXiv preprint arXiv:1903.04628*, 2019.

[MDB16] Sathiya Kumaran Mani, Ramakrishnan Durairajan, Paul Barford, and Joel Sommers. "Mntp: Enhancing time synchronization for mobile devices." In *Proceedings of the 2016 Internet Measurement Conference*, pp. 335–348. ACM, 2016.

[Meh18] Ashok B Mehta. "Constrained Random Verification (CRV)." In *ASIC/SoC Functional Design Verification*, pp. 65–74. Springer, 2018.

[MGT11] Aneeq Mahmood, Georg Gaderer, Henning Trsek, Stefan Schwalowsky, and Nikolaus Kerö. "Towards high accuracy in IEEE 802.11 based clock synchronization using PTP." In *2011 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, pp. 13–18. IEEE, 2011.

[MHM16] Abhinav Mehrotra, Robert Hendley, and Mirco Musolesi. "Towards multi-modal anticipatory monitoring of depressive states through the analysis of human-smartphone interaction." In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, pp. 1132–1138. ACM, 2016.

[Mil91] David L Mills. "Internet time synchronization: the network time protocol." *IEEE Transactions on communications*, **39**(10):1482–1493, 1991.

[Mil12] David Mills. "Executive Summary: Computer Network Time Synchronization." `https://www.eecis.udel.edu/~mills/exec.html`, 2012. Accessed: 2019-03-26.

[MKK18] A Rupam Mahmood, Dmytro Korenkevych, Brent J Komer, and James Bergstra. "Setting up a reinforcement learning task with a real-world robot." In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 4635–4640. IEEE, 2018.

[MKS04] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. "The flooding time synchronization protocol." In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pp. 39–49. ACM, 2004.

[MKS15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. "Human-level control through deep reinforcement learning." *Nature*, **518**(7540):529–533, 2015.

[Mou17] Mourad Mourafiq. "Polyaxon: Cloud native machine learning automation platform." Web page, 2017.

[MRF10] Christian Micheloni, Bernhard Rinner, and Gian Luca Foresti. "Video analysis in pan-tilt-zoom camera networks." *IEEE Signal Processing Magazine*, **27**(5):78–90, 2010.

[MW22] Daniel Mendoza and Sijin Wang. "Predicting latency of neural network inference.", Jul 2022.

[Nil98] Johan Nilsson et al. "Real-time control systems with delays." 1998.

[NKK11] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. "Multimodal deep learning." In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pp. 689–696, 2011.

[Noo19] Joseph Noor. "improves system clock accuracy - Gerrit Code Review." `https://android-review.googlesource.com/c/platform/frameworks/base/+/1148834`, 2019. Accessed: 2019-01-31.

[NRG16] Vu Nguyen, Santu Rana, Sunil K Gupta, Cheng Li, and Svetha Venkatesh. "Budgeted batch Bayesian optimization." In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 1107–1112. IEEE, 2016.

[Nvi22] Nvidia Nvidia. "Nvidia multi-instance gpu(mig).", Jul 2022.

[OSG19] Juan DS Ortega, Mohammed Senoussaoui, Eric Granger, Marco Pedersoli, Patrick Cardinal, and Alessandro L Koerich. "Multimodal Fusion with Deep Neural Networks for Audio-Video Emotion Recognition." *arXiv preprint arXiv:1907.03196*, 2019.

[PAZ18] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. "Sim-to-real transfer of robotic control with dynamics randomization." In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1–8. IEEE, 2018.

[PCH12] Thomas Plotz, Chen Chen, Nils Y Hammerla, and Gregory D Abowd. "Automatic Synchronization of Wearable Sensors and Video-Cameras for Ground Truth Annotation–A Practical Approach." In *2012 16th International Symposium on Wearable Computers*, pp. 100–103. IEEE, 2012.

[PL19] Liliana Lo Presti and Marco La Cascia. "Deep Motion Model for Pedestrian Tracking in 360 Degrees Videos." In *International Conference on Image Analysis and Processing*, pp. 36–47. Springer, 2019.

[PMO21] Evgeny Ponomarev, Sergey Matveev, Ivan Oseledets, and Valery Glukhov. "Latency estimation tool and investigation of neural networks inference on mobile gpu." *Computers*, **10**(8):104, 2021.

[PVG11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python." *Journal of Machine Learning Research*, **12**:2825–2830, 2011.

[Res21] Arm Research. "Parameter search spaces use to evaluate Mango on classifiers." `https://github.com/ARM-software/mango/blob/master/benchmarking/Parameter_Spaces_Evaluated.ipynb`, 2021.

[RET14] Stefan Rudolph, Sarah Edenhofer, Sven Tomforde, and Jörg Hähner. "Reinforcement learning for coverage optimization through PTZ camera alignment in highly dynamic environments." In *Proceedings of the International Conference on Distributed Smart Cameras*, pp. 1–6, 2014.

[RF17] Joseph Redmon and Ali Farhadi. "YOLO9000: better, faster, stronger." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7263–7271, 2017.

[RGR09] Anthony Rowe, Vikram Gupta, and Ragunathan Raj Rajkumar. "Low-power clock synchronization using electromagnetic energy radiating from ac power lines." In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pp. 211–224. ACM, 2009.

[RHE19] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. "Stable Baselines3." `https://github.com/DLR-RM/stable-baselines3`, 2019.

[RNG18] Seyed Ali Rokni, Marjan Nourollahi, and Hassan Ghasemzadeh. "Personalized human activity recognition using convolutional neural networks." In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[Roc15] Matthew Rocklin. "Dask: Parallel computation with blocked algorithms and task scheduling." In *Proceedings of the 14th python in science conference*, volume 126. Citeseer, 2015.

[RP19] Simon Ramstedt and Chris Pal. "Real-time reinforcement learning." In *Advances in Neural Information Processing Systems*, pp. 3073–3082, 2019.

[RRV19] Hazem Rashed, Mohamed Ramzy, Victor Vaquero, Ahmad El Sallab, Ganesh Sistu, and Senthil Yogamani. "FuseMODNet: Real-Time Camera and LiDAR based Moving Object Detection for robust low-light Autonomous Driving." *arXiv preprint arXiv:1910.05395*, 2019.

[RTB18] Valentin Radu, Catherine Tong, Sourav Bhattacharya, Nicholas D Lane, Cecilia Mascolo, Mahesh K Marina, and Fahim Kawsar. "Multimodal deep learning for activity and context recognition." *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, **1**(4):157, 2018.

[SAF20] Sandeep Singh Sandha, Mohit Aggarwal, Igor Fedorov, and Mani Srivastava. "Mango: A Python Library for Parallel Hyperparameter Tuning." In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3987–3991. IEEE, 2020.

[San19] Sandeep Singh Sandha. "Github: Time Sync Across Smartphones." `https://github.com/nesl/Time-Sync-Across-Smartphones`, 2019. Accessed: 2019-05-05.

[SBB10] Erik Schuitema, Lucian Buşoniu, Robert Babuška, and Pieter Jonker. "Control delay in reinforcement learning for real-time dynamic systems: a memoryless approach." In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3226–3231. IEEE, 2010.

[SBB15] Allan Stisen, Henrik Blunck, Sourav Bhattacharya, Thor Siiger Prentow, Mikkel Baun Kjærgaard, Anind Dey, Tobias Sonne, and Mads Møller Jensen. "Smart devices are different: Assessing and mitigatingmobile sensing heterogeneities for activity recognition." In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pp. 127–140. ACM, 2015.

[Sca18] Enrico Scarrone. "3GPP Specification 22.042." https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=576, 2018. Accessed: 2019-03-26.

[SCD11] Pietro Salvagnini, Marco Cristani, Alessio Del Bue, and Vittorio Murino. "An experimental framework for evaluating PTZ tracking algorithms." In *International Conference on Computer Vision Systems*, pp. 81–90. Springer, 2011.

[SDL18] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. "Airsim: High-fidelity visual and physical simulation for autonomous vehicles." In *Field and service robotics*, pp. 621–635. Springer, 2018.

[SGB20] Sandeep Singh Sandha, Luis Garcia, Bharathan Balaji, Fatima Anwar, and Mani Srivastava. "Sim2Real Transfer for Deep Reinforcement Learning with Stochastic State Transition Delays." pp. 1066–1083, 2020.

[Sim13] Dan Simon. *Evolutionary optimization algorithms*. John Wiley & Sons, 2013.

[SKK09] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. "Gaussian process optimization in the bandit setting: No regret and experimental design." *arXiv preprint arXiv:0912.3995*, 2009.

[SL16] Fereshteh Sadeghi and Sergey Levine. "Cad2rl: Real single-image flight without a single real image." *arXiv preprint arXiv:1611.04201*, 2016.

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. "Practical bayesian optimization of machine learning algorithms." *Advances in neural information processing systems*, **25**:2951–2959, 2012.

[SMG16] Sabarish Sridhar, Prasant Misra, Gurinder Singh Gill, and Jay Warrior. "Cheepsync: a time synchronization service for resource constrained bluetooth le advertisers." *IEEE Communications Magazine*, **54**(1):136–143, 2016.

[SN19] Sandeep Singh Sandha and Joseph Noor. "Github: GoodClock Library." https://github.com/nesl/GoodClock, 2019. Accessed: 2020-01-31.

[SNA20] Sandeep Singh Sandha, Joseph Noor, Fatima M Anwar, and Mani Srivastava. "Time Awareness in Deep Learning-Based Multimodal Fusion Across Smartphone Platforms." In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pp. 149–156. IEEE, 2020.

[SRS15] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. "Scalable bayesian optimization using deep neural networks." In *International conference on machine learning*, pp. 2171–2180, 2015.

[SWD17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal policy optimization algorithms." *arXiv preprint arXiv:1707.06347*, 2017.

[SX19] Sandeep Singh Sandha and Tianwei Xing. "Github: CMActivities DataSet." `https://github.com/nesl/CMActivities-DataSet`, 2019. Accessed: 2020-01-31.

[Ten22a] Tensorflow Tensorflow. "Tensorflow Lite: ML for Mobile and edge devices.", Jun 2022.

[Ten22b] Tensorflow Tensorflow. "Tensorflow/TENSORRT: Tensorflow/TENSORRT integration.", Jun 2022.

[TFR17] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. "Domain randomization for transferring deep neural networks from simulation to the real world." In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pp. 23–30. IEEE, 2017.

[THH13] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. "Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms." In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 847–855. ACM, 2013.

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. "{CLKSCREW}: exposing the perils of security-oblivious energy management." In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pp. 1057–1074, 2017.

[Ula15] Lance Ulanoff. "Here's how Apple synchronized all your Apple Watches." `https://mashable.com/2015/12/30/apple-watch-synchronized/#DvsLrXLHeZq4`, 2015. Accessed: 2019-03-26.

[Ult22] Ultralytics Ultralytics. "Train Custom Data · ultralytics/yolov5 wiki.", Apr 2022.

[UNC19] Halil Utku Unlu, Phillip Stefan Niehaus, Daniel Chirita, Nikolaos Evangeliou, and Anthony Tzes. "Deep learning-based visual tracking of UAVs using a PTZ camera system." In *IECON 2019-45th Annual Conference of the IEEE Industrial Electronics Society*, volume 1, pp. 638–644. IEEE, 2019.

[unr22] "The most powerful realtime 3D creation tool.", Apr 2022.

[WAE09] Martin Wöllmer, Marc Al-Hames, Florian Eyben, Björn Schuller, and Gerhard Rigoll. "A multidimensional dynamic time warping algorithm for efficient multimodal fusion of asynchronous data streams." *Neurocomputing*, **73**(1-3):366–380, 2009.

[WBN98] Björn Wittenmark, Ben Bastian, and Johan Nilsson. "Analysis of time delays in synchronous and asynchronous control loops." In *Proceedings of the 37th IEEE Conference on Decision and Control (Cat. No. 98CH36171)*, volume 1, pp. 283–288. IEEE, 1998.

[WDH16] Rui Wang, Hao Dong, Tony X Han, and Lei Mei. "Robust tracking via monocular active vision for an intelligent teaching system." *The Visual Computer*, **32**(11):1379–1394, 2016.

[Wil15] Rhiannon Williams. "Why the Apple Watch will be the most accurate way to ring in the New Year." `https://www.telegraph.co.uk/technology/apple/watch/12074452/Why-the-Apple-Watch-will-be-the-most-accurate-way-to-ring-in-the-Ne html`, 2015. Accessed: 2019-03-26.

[WNL09] Thomas J Walsh, Ali Nouri, Lihong Li, and Michael L Littman. "Learning and planning in environments with delayed feedback." *Autonomous Agents and Multi-Agent Systems*, **18**(1):83, 2009.

[WRG14] Karl Worthmann, Marcus Reble, Lars Grune, and Frank Allgower. "The role of sampling for stability and performance in unconstrained nonlinear model predictive control." *SIAM Journal on Control and Optimization*, **52**(1):581–605, 2014.

[WSH15] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. "Dueling network architectures for deep reinforcement learning." *arXiv preprint arXiv:1511.06581*, 2015.

[WTX14] Yu Wang, Rui Tan, Guoliang Xing, Jianxun Wang, Xiaobo Tan, Xiaoming Liu, and Xiangmao Chang. "Aquatic debris monitoring using smartphone-based robotic sensors." In *Proceedings of the 13th international symposium on Information processing in sensor networks*, pp. 13–24. IEEE Press, 2014.

[WWD94] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. "Scheduling for reduced CPU energy." In *Mobile Computing*, pp. 449–471. Springer, 1994.

[XBC18] Zhaoming Xie, Glen Berseth, Patrick Clary, Jonathan Hurst, and Michiel van de Panne. "Feedback control for cassie with deep reinforcement learning." In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1241–1246. IEEE, 2018.

[XSB18] Tianwei Xing, Sandeep Singh Sandha, Bharathan Balaji, Supriyo Chakraborty, and Mani Srivastava. "Enabling Edge Devices that Learn from Each Other: Cross Modal Training for Activity Recognition." In *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking*, pp. 37–42. ACM, 2018.

[YLT17] Zhenyu Yan, Yang Li, Rui Tan, and Jun Huang. "Application-layer clock synchronization for wearables using skin electric potentials induced by powerline radiation." In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, p. 10. ACM, 2017.

[YNS15] Jianbo Yang, Minh Nhut Nguyen, Phyo Phyo San, Xiao Li Li, and Shonali Krishnaswamy. "Deep convolutional neural networks on multichannel time series for human activity recognition." In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[yol22] "Yolo neural object detector.", Apr 2022.

[ZHW21] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. "nn-Meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices." In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 81–93, 2021.

[ZVP19] Jinan Zhou, Andrey Velichkevich, Kirill Prosvirov, Anubhav Garg, Yuji Oshima, and Debo Dutta. "Katib: A distributed general automl platform on kubernetes." In *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)*, pp. 55–57, 2019.