

ABSTRACT

SUNG, HSIN-HSUAN. Runtime Optimization of Multi-Tenant DNN Executions on Heterogeneous Mobile Devices. (Under the direction of Xipeng Shen).

In the burgeoning field of Edge AI, this study delves into the critical aspect of runtime optimization of multi-tenant Deep Neural Networks (DNNs) on heterogeneous mobile devices. Edge AI systems, which are increasingly integral to advanced driver assistance, smart home technologies, and mobile AI applications, rely heavily on DNNs for a wide array of services. However, the efficient management and optimization of DNN executions present unique challenges due to the limitations of CPUs and the need for specialized processing units like GPUs, DSPs, TPUs, NPU, and DLAs.

This research tackles the complex task of optimally optimizing and scheduling multi-tenant DNNs on heterogeneous mobile devices, essential for maximizing efficiency in diverse scenarios. The study unfolds in three pivotal directions: First, it conducts a systematic analysis of open-source advanced autonomous driving systems to identify and resolve critical scheduling deficits, facilitating their transition from high-end accelerators to lower-end edge devices. Second, it introduces the first decentralized adaptive scheduling mechanism for multi-tenant DNNs on open mobile devices, utilizing Deep Reinforcement Learning for enhanced scheduling effectiveness, adaptability, and rapid convergence. Third, the study pioneers COSMA, a novel solution for the systematic co-optimization of multi-tenant DNNs and scheduling, effectively addressing the elastic DNN optimization-scheduling problem (EDOSP) and demonstrating superiority over existing methods in terms of robustness and portability.

By offering comprehensive solutions to the primary bottlenecks in AI systems, namely the optimization and scheduling of concurrently running multi-tenant DNNs, this research sets new performance benchmarks in the field and opens up new avenues for future research in the architecture and design of autonomous driving systems, mobile AI applications and runtime optimization.

© Copyright 2024 by Hsin-Hsuan Sung

All Rights Reserved

Runtime Optimization of Multi-Tenant DNN Executions on Heterogeneous Mobile Devices

by
Hsin-Hsuan Sung

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2024

APPROVED BY:

Huiyang Zhou

Man-Ki Yoon

Min Chi

Xipeng Shen
Chair of Advisory Committee

DEDICATION

This dissertation is dedicated to my beloved parents in Taiwan. Your love and support have been my constant inspiration. Through your sacrifices and belief in me, you have instilled values of perseverance and hard work. This achievement is as much yours as it is mine. Thank you for everything.

BIOGRAPHY

Hsin-Hsuan Sung was born in New Taipei City, Taiwan, in 1992. He received his Bachelor's degree from National Taiwan Normal University in 2015 and his Master's degree from National Cheng Kung University in 2017. Beginning his Ph.D. journey in 2019, he started his studies in Computer Science at North Carolina State University under the supervision of Dr. Xipeng Shen. His research primarily focuses on Real-time Edge AI and High-Performance Computing, exploring innovative approaches and technologies in these dynamic fields.

ACKNOWLEDGEMENTS

I am deeply grateful to my advisor, Dr. Xipeng Shen, whose rigorous standards and insightful guidance have profoundly shaped my work and personal growth. His dedication to excellence and meticulous attention have not only refined my dissertation but also taught me invaluable lessons in persistence and precision. Dr. Shen's mentorship has been instrumental in my development as a researcher and professional, and I am immensely thankful for his unwavering support.

I extend my thanks to my committee members: Dr. Huiyang Zhou, Dr. Man-Ki Yoon, and Dr. Min Chi, whose expert insights and detailed feedback have significantly enhanced this dissertation.

My appreciation also goes to my collaborators: Dr. Wei Niu, and Dr. Bin Ren, for their invaluable contributions.

I owe a special thanks to my wife, Jou-An, who encouraged me to pursue my PhD studies in the U.S. and seek a better life. Without her support, I cannot imagine completing this journey; her influence has transformed what could have been an ordinary life into an extraordinary one.

Lastly, I would like to thank my parents and my in-laws for their unwavering support and love throughout my academic journey. Their encouragement has been a cornerstone of my success.

TABLE OF CONTENTS

List of Tables	vii
List of Figures	viii
Chapter 1 Introduction	1
Chapter 2 Enabling Level-4 Autonomous Driving on a Single \$1k Off-the-Shelf Card	5
2.1 Introduction	5
2.2 Autonomous Driving Application and Device	7
2.2.1 High-level Workflow	9
2.2.2 ADApp: A Set of Level-4 Autonomous Driving Applications	9
2.2.3 Device	12
2.3 Observations and Analysis of the Default Executions	13
2.3.1 Observations	13
2.3.2 Analysis	14
2.4 Just-In-Time Affinity and Priority Adjustment	16
2.4.1 Background on Linux Schedule	16
2.4.2 Just-In-Time Adjustment	16
2.4.3 Performance and Analysis	18
2.5 Migration to Take Advantage of All Accelerators	18
2.5.1 Changes	18
2.5.2 Performance and Analysis	20
2.6 Hardware-Aware Model Customization	22
2.6.1 Changes	22
2.6.2 Performance and Analysis	22
2.7 Other Experimented Optimizations	23
2.8 Power, Synergy, Insights and SCAD Kit	24
2.9 Implications	25
2.10 Related Work	26
2.11 Conclusion	27
Chapter 3 Decentralized Application-Level Adaptive Scheduling for Multi-Instance DNNs on Open Mobile Devices	31
3.1 Introduction	31
3.2 Background	34
3.3 Problem Statement and Research Questions	35
3.3.1 Problem Definition	35
3.3.2 Design Considerations and Principles	36
3.3.3 Research Questions	37
3.4 Decentralized DQN Scheduler	38
3.5 Convergence Discussion	40
3.6 Evaluation	42
3.6.1 Evaluation Methodology	42

3.6.2	Overall DQN Scheduling Performance	45
3.6.3	Reward Convergence Analysis	48
3.6.4	DQN Performance w/ Uncontrollable Apps	49
3.7	Related Work	51
3.8	Conclusion	52
Chapter 4 COSMA: Contention-conscious Co-optimization of DNNs and Scheduling on Mobile Systems		53
4.1	Introduction	53
4.2	Background	56
4.2.1	Reinforcement Learning and Value-Based Methods	56
4.2.2	Deep Reinforcement Learning	57
4.3	EDOSP Problem Statement	58
4.4	Challenges and Design Principles	59
4.5	COSMA: Design and Implementation	60
4.5.1	Overview	60
4.5.2	Introducing CTDE	60
4.5.3	Formulation of EDOSP	62
4.5.4	Components for CTDE Learning	64
4.5.5	Learning Algorithm	65
4.5.6	COSMA Software Architecture and Workflow	66
4.6	On-Device Learning v.s. Server Cooperative Learning	68
4.7	Evaluation	70
4.7.1	Methodology	71
4.7.2	Scenario I: Static Models Set	73
4.7.3	Scenario II: Random Models Set	73
4.7.4	Scenario III: Dynamic Workloads with Random Models and Additional Random Interference	75
4.7.5	Scenario IV: Random Models set with Real-world Uncontrolled Apps	76
4.7.6	Cross-Hardware Portability Test	77
4.7.7	Multi-exit DNN models	78
4.7.8	Transfer Learning	79
4.8	Related Work	82
4.9	Conclusion	84
Chapter 5 Conclusions and Future Work		85
5.1	Conclusions	85
5.2	Future Work	86
References		88

LIST OF TABLES

Table 2.1	Differences among the ADApp applications	10
Table 2.2	Configurations of the Target Device (Jetson AGX Xavier)	12
Table 2.3	Execution time (mean± std) of each module in the ADApp applications on Jetson AGX Xavier and the miss rates. The ∞ represents timeout. The miss rate of a module is how often the module misses its expected latency (shown in the parentheses in the table header)—up to 10% over is allowed to tolerate system noises. The column Miss Rate shows the miss rates of the most sluggish modules (whose times are prefixed with an *), that is, the modules with the largest miss rate in the application.	29
Table 2.4	Power consumption for the ADApp applications on Jetson AGX Xavier by INA3221 power monitors (Xav21) (unit: milliWatts)	30
Table 2.5	Accuracy before and after the hardware-aware model customization	30
Table 3.1	Nine DNN Models Used in Our Evaluation. They form three groups. DNN models in each group are executed simultaneously.	43
Table 3.2	Absolute Latency of DQN. This table reports the absolute latency of DQN in Figure 3.3, Figure 4.6, and Figure 3.7.	43
Table 3.3	Absolute Energy Factor of DQN. This table reports the absolute energy of DQN in Figure 3.3, Figure 4.6, Figure 3.7.	44
Table 3.4	Selection Rates of Actions of Last 100 Runs of Figure 3.5.	46
Table 3.5	Comparisons with simple Heuristic-based adaptation. This table reports the average latency and energy factor for G2 in three co-run settings described in Figures 3.3, 4.6, and 3.7	50
Table 4.1	DNN Models and Variants and their Standalone Latencies in different Setting of TFLite Runtime Interpreter (unit: million-seconds). The Android Neural Networks API (NNAPI) (Dev23) is an Android C API designed for running computationally intensive operations for machine learning on Android devices. K threads means use K threads to run on CPU	67
Table 4.2	The comparison of two structure designs: Server Cooperative vs. On-device learning	69

LIST OF FIGURES

Figure 2.1	High-level workflow of a level-4 autonomous vehicle.	8
Figure 2.2	Detailed tasks within an example autonomous driving application (ADApp) derived from Autoware. We categorize the tasks into the modules in Figure 2.1: Sensing (blue), Perception (red), Localization (gray), Tracking (purple), Prediction (green), Planning (yellow), and final control output (orange)	8
Figure 2.3	The TensorRT Workflow. (ten21)	19
Figure 2.4	Default execution	21
Figure 2.5	After changes to use DLA	21
Figure 2.6	After hardware-aware model customization	21
Figure 2.7	The timeline of the execution of one inference by the YOLOv3-608 × 608 DNN, collected via NVIDIA Nsight.	21
Figure 2.8	The cyclic dependence in the scheduling problem	23
Figure 3.1	Standalone Profiling of Three DNN Networks. We profile the average power and inference time for different DNNs. They have their own best option for delegation. Thread = i means the model is executed on the CPU with i threads. NNAPI <i>low power</i> and <i>fast single answer</i> are two options that consider using GPU and accelerators.	32
Figure 3.2	The Structure of Decentralized DQN Scheduler. The figure shows three controllable DNN applications that have their RL agent that selects actions (different code generation of the model). All of them collect system status as their RL agent input state. Also, we include the co-running uncontrollable apps that do not contain RL agents inside.	37
Figure 3.3	Overall Comparison between DQN and Three Baselines on Three DNN Groups. Compare (a) normalized average inference time and (b) rewards on Samsung Galaxy Tab S8+ and the same metrics (c) and (d) on Samsung Galaxy S21 FE when each group co-running three DNN apps (* denotes this DNN runs in the foreground and the others run in the background).	44
Figure 3.4	Reward Convergence Trend for the Three Apps Co-Run in G2. It reports DQN’s normalized reward trend to prove DQN converges in different situations. It uses DNNs in G2 and places each DNN in the foreground.	45
Figure 3.5	Selected Action Convergence Trend. This figure shows the action selection through runs for co-running results in G1 (all of them are in the background). The action selection will be converged into one or two options.	46

Figure 3.6	Co-Run Three DNN Groups with Uncontrollable App TikTok. Compare (a) normalized average inference time and (b) rewards on Samsung Galaxy Tab S8+ and the same metrics (c) and (d) on Samsung Galaxy S21 FE when each DNNs group co-running with uncontrollable app TikTok that plays video posts on the foreground.	47
Figure 3.7	Co-Run Three DNN Groups with Uncontrollable App Web Browser (Google Chrome). Compare (a) normalized average inference time and (b) rewards on Samsung Galaxy Tab S8+ and the same metrics (c) and (d) on Samsung Galaxy S21 FE when each DNNs group co-running with the uncontrollable app Google Chrome randomly browsing pages on the foreground.	48
Figure 4.1	A multi-exit DNN with three potential outputs determined by thresholds like latency or accuracy. Deeper exits offer higher accuracy but greater latency. Blue layers are for feature extraction, circles indicate classifiers, and the cat image is the input.	56
Figure 4.2	The COSMA framework orchestrates the co-scheduling of multiple DNNs. The execution workflow, depicted with blue arrows, involves: (1) gathering local states $\tau_t^0, \dots, \tau_t^i$ from the participating Apps; (2) obtaining the selected actions a_t^0, \dots, a_t^i for each agent from the Policy Network; (3) compiling the subsequent local states $\tau_{t+1}^{0:i}$, action set $a_t^{0:i}$, global states s_{t+1} , and computing the total reward r_{t+1} for each agent in the transition from time slot t to $t + 1$, subsequently storing these into the replay memory. The training workflow, indicated by purple arrows, includes: (1) selecting a batch from the replay memory to feed into the Policy Networks and Target Networks; (2) calculating the loss value using Q_{tot} and y_{tot} ; (3) updating the weights of the policy networks. Following K updates, the policy networks transfer their weights to the target networks.	61
Figure 4.3	At the top is the Server Cooperative Learning Structure. The red message encompasses the current and next states, along with additional details such as hit/miss counts, aiding the COSMA controller in reward calculation and action generation for subsequent steps. The blue message denotes the chosen actions for each controllable app, along with the signal to initiate the next step. At the bottom is the On-Device Learning Structure, featuring the COSMA controller.	68
Figure 4.4	Static Selected Three models in each Episode for co-scheduling multiple DNNs without Co-running Apps.	70
Figure 4.5	Three distributions of different model combinations in each episode. (0-0-1-1-2), for instance, means that two 0:EfficientNet, two 1:MobileNet, and one 2:YoloX in Table 4.1 will be co-scheduled together.	70

Figure 4.6	Randomly Selected Five models under three distributions under Figure 4.5 in each Episode for co-scheduling multiple DNNs without Co-running Apps.	71
Figure 4.7	Selected Five models under Uniform distributions in each Episode for co-scheduling multiple DNNs with CPU Usage Co-running Apps.	74
Figure 4.8	Selected Five models under Uniform distributions in each Episode for co-scheduling multiple DNNs with CPU/GPU Usage Co-running Apps.	75
Figure 4.9	Selected Five models under Uniform distributions in each Episode for co-scheduling multiple DNNs with Real-world Apps including TikTok, YouTube, and WebBrowser . All results adopt COSMA(QMix)	76
Figure 4.10	Used Samsung Galaxy S10e and Randomly Selected Five models under Normal distributions under Figure 4.5 in each Episode for co-scheduling multiple DNNs without Co-running Apps.	77
Figure 4.11	Concurrently co-scheduling five multi-exit models, each with 5 exits, and 8 placements (1-8 threads) per model, presents significant challenges. The vast action space in the Centralized structure (resulting in 40^5 possible outputs) demands extensive memory resources, totaling 12.64 GB, leading to out-of-memory issues on the RTX 4060.	79
Figure 4.12	Co-scheduling Five multi-exit models with Real-world Apps including TikTok, YouTube, and WebBrowser . All results adopt COSMA(QMix)	80
Figure 4.13	The dotted lines represent the training of five selected models with varying patterns in each episode, starting from scratch. Conversely, the solid lines depict transfer learning from models with opposite patterns.	81
Figure 4.14	The dashed lines depict the training of five models, each representing one of the three types detailed in Table 4.1, starting anew in every episode. Conversely, the solid lines demonstrate transfer learning, utilizing trained weights from two distinct model types.	82
Figure 4.15	The dashed lines represent the selection of five models under Uniform distributions in each episode for co-scheduling multiple DNNs with the Real-world App, TikTok, starting from scratch. Conversely, the solid lines illustrate transfer learning from trained weights derived from the same setup but with the Real-world App, YouTube.	83

CHAPTER

1

INTRODUCTION

The integration of Edge AI systems into various aspects of daily life, including advanced driver assistance (Tes18; Fou18; Aut18; Way18), smart home technologies (KCK⁺21; SDR20; ST17), and mobile AI applications (Sam18; Qua18; Goo18a), is a testament to the evolving landscape of technology. At the core of these systems are Deep Neural Networks (DNNs), offering a plethora of services such as generative models (Goo18b), super-resolution (Sam18), obstacle recognition (HH20), and drowsiness detection (VDW19). These applications highlight the critical need for efficiently managing and optimizing DNN executions on mobile devices.

Addressing the inherent challenges posed by CPUs' unsuitability for DNN executions, a range of specialized processing units have emerged. This includes Graph Processing Units (GPUs) (amp23; Cor23), Digital Signal Processors (DSPs) (Quab), Tensor Processing Units (TPUs) (Goo), Neural Processing Units (NPU) (CR20), and Deep Learning Accelerators (DLAs) (NVI). These developments culminate in the trend of integrating such DNN-specialized hardware with traditional GPUs and CPUs, forming heterogeneous mobile device architectures.

Multi-tenant DNN Executions on a heterogeneous mobile device refers to the scenario where multiple DNN-based applications simultaneously share the resources of a single heterogeneous mobile device. This approach is particularly favored for its cost-effectiveness, making it a desirable choice for a broad spectrum of users who need to run multiple DNN-based applications concurrently. A classical example of this is found in autopilot systems, where

data from several cameras and a lidar are processed simultaneously for obstacle detection using DNN execution on a heterogeneous vehicular platform device (Quaa). Consequently, the foremost challenge in this domain lies in the efficient optimization and scheduling of multi-tenant DNNs to ensure peak performance across diverse scenarios.

In the context of heterogeneous mobile devices, scheduling multi-tenant DNNs for AI applications becomes a pivotal task (LHL⁺21). This challenge is compounded by the constrained resources and the diversity of processing units available. Effective scheduling is therefore essential to optimize hardware resource utilization, and enhance power and latency efficiencies. A prime example of this is in autonomous driving systems, which require simultaneous data processing from a range of sources, including cameras and radar/lidar sensors, to execute multiple DNNs for traffic sign recognition, lane detection, pedestrian detection, and 3D object recognition (Fou18).

This study aims to develop systematic solutions for scheduling multi-tenant DNNs in concurrent execution on heterogeneous SoC architectures. We focus on three key areas:

- Analyzing the state-of-the-art open-source autonomous driving systems from a systematic perspective, identifying critical deficits and proposing resolutions for each, thereby facilitating the transition of ADAS from high-end accelerator boxes to lower-end edge devices. Additionally, we provide a reusable benchmark for easy analysis of ADAS on lower-end edge devices.
- Introducing the first decentralized adaptive scheduling for multi-tenant DNNs on *open* mobile devices. Here, *open* mobile devices refer to mobile devices on which users can install or uninstall arbitrary apps anytime. This approach provides key insights into effective scheduling without the need for knowledge about other apps, adaptability of DRL-based scheduling to various execution factors, and rapid convergence of the decentralized algorithm. Empirical evaluations confirm significant speedups and energy savings across diverse DNN workloads and running scenarios.
- Presenting the first systematic co-optimization of multi-tenant DNNs and scheduling for heterogeneous mobile devices, addressing the *elastic DNN optimization-scheduling problem* (EDOSP). The *elastic* means that each DNN may be configured dynamically and show different speed-accuracy tradeoffs. Recent AI research (FZZ18; CYS⁺24; WLL⁺19; MHY⁺21; GZL⁺22) has shown that elastic DNNs can provide much better speed-accuracy quality, especially when the environment changes. However, none of the prior DNN scheduling studies have considered elastic DNNs. Therefore we develop COSMA, the first solution to EDOSP, which outperforms existing centralized and decentralized meth-

ods by 1.4x to 2x, proving its robustness and portability across different hardware in real-world application scenarios.

Our research offers an in-depth exploration of real-world AI systems on heterogeneous mobile devices, pinpointing and addressing key limitations in the current scheduling systems. We delve into the primary bottleneck in AI systems – the scheduling of concurrently running multi-tenant DNNs – and provide comprehensive solutions to simplify and focus on these challenges. The study introduces a decentralized scheduling approach for multi-tenant DNNs, enhancing adaptiveness and privacy by incorporating a reinforcement learning agent within each app. A novel framework is also proposed, employing a publicly centralized controller app, effectively addressing non-stationarity issues and surpassing the performance of the second study.

The structure of the subsequent chapters is outlined as follows:

Chapter 2 presents the research of enabling ADAS on a Single \$1k Off-the-Shelf Card. We demonstrate that ADAS software can operate on a single Jetson AGX Xavier card for under \$1K, significantly reducing costs while meeting latency requirements. This breakthrough, achieved through innovative solutions to existing challenges, challenges current perceptions about necessary computing resources and suggests new avenues for research in the architecture and design of autonomous driving systems.

Chapter 3 addresses the challenge of scheduling multiple DNN-powered apps running simultaneously on open mobile systems like smartphones and tablets. In contrast to closed systems with predetermined apps, open systems face dynamic app installations and require adaptable solutions. The study introduces a novel decentralized application-level scheduling mechanism using Deep Reinforcement Learning. This mechanism helps achieve a Nash equilibrium in app scheduling, balancing performance gains efficiently. It also adapts automatically to the running environment, OS, and hardware. Experimental results demonstrate consistent speedups and energy savings across various DNN workloads, hardware setups, and usage scenarios.

Chapter 4 investigates the elastic DNN optimization-scheduling problem (EDOSP), which allows real-time adjustments of DNN configurations and schedules on mobile devices hosting multiple applications. Addressing the challenges of runtime optimization and scheduling, the study introduces COSMA, a solution integrating Centralized Training with Distributed Execution (CTDE) and cooperative Deep Reinforcement Learning (DRL). COSMA overcomes key EDOSP complexities and ensures convergence, offering substantial performance improvements. It achieves a 1.4-2x reward increase in throughput and accuracy over alternative solutions in different device and workload scenarios.

Chapter 5 concludes innovative approaches to runtime optimization that can profoundly

impact the efficiency and effectiveness of multi-tenant DNN executions on heterogeneous mobile devices. By integrating advanced scheduling techniques and optimization frameworks like COSMA and decentralized adaptive scheduling, we pave the way for more resilient, adaptable, and cost-effective mobile AI applications. These developments not only fulfill the dissertation's objective but also significantly contribute to the broader field of mobile computing, enhancing the performance and scalability of DNNs across diverse hardware platforms.

CHAPTER

2

ENABLING LEVEL-4 AUTONOMOUS DRIVING ON A SINGLE \$1K OFF-THE-SHELF CARD

2.1 Introduction

Autonomous driving is drawing great interest. The high cost has been one of the major roadblocks that slow down the development and adoption of autonomous driving in practice (LG20). One of the most costly components is the hardware to execute the autonomous driving software.

At present, even partially autonomous (e.g., level-2) driving systems already require either a high-end accelerator box (e.g., NVIDIA Drive (nvi21) or some kind of custom hardware, and the cost of either is no lower than \$10k. A device that can run fully autonomous (level-4) driving software costs several times more for the much more computing resources it relies on for real-time response of the software of a larger scale and greater complexity. The Baidu autonomous driving platform, for instance, consists of two compute boxes, costs as much as \$30K, and draws 3000W power (Liu20).

This dissertation strives to answer three research questions (RQ) crucial for the cost and

hence the future development of the autonomous driving industry:

- RQ-1: Does fully autonomous driving really need that much computing resource?
- RQ-2: What causes the currently observed performance deficiency of low-end devices for fully autonomous driving?
- RQ-3: Is it possible for level-4 autonomous driving to achieve real-time performance on a single off-the-shelf card for as little as \$1K, an order of magnitude less than state-of-the-art? How to achieve that?

To answer these questions, we conduct a focused study, trying to optimize the deployment of six level-4 autonomous driving applications (derived from Autoware (KTM⁺18)) on a single off-the-shelf low-end card, Jetson AGX Xavier (jet21) from NVIDIA, as part of the core module of the computing system of the second-generation level-4 autonomous driving system of our company. The result overturns some common perceptions held by the industry. For the first time, we show that it is possible to run industrial-level level-4 autonomous driving on a single off-the-shelf card (Jetson) for as little as \$1k while meeting all latency requirements. Meanwhile, this study produces a set of key insights on the important pitfalls or technical deficits in the current autonomous driving industry practice and contributes several practical solutions:

- Deficit I: Starvation happens when prior scheduling schemes are applied to autonomous driving applications that are deployed to a single low-end device.
 - Resolution: We propose a simple solution, *just-in-time priority adjustment*, which resolves the starvation by adjusting the affinity and priorities of tasks in a just-in-time manner.
- Deficit II: Some types of accelerators are left substantially under-utilized due to hardware-oblivious model designs and implementations.
 - Resolution: We employ *hardware-aware model customization*, an approach that significantly increases the accelerators' utilization by bridging the gap between DNN models and multiple types of accelerators.
- Deficit III: Current scheduling algorithms for autonomous driving cannot deal with hybrid workloads that can employ multiple types of accelerators.
 - Resolution: We propose *DAG instantiation-based scheduling*, an approach that extends the scheduling of autonomous driving to meet the needs via accelerator-based DAG instantiation.

The explorations together led to the success of making all of the six level-4 autonomous driving applications achieve real-time performance on a single Jetson card. The success has multi-fold implications. It entails the need for the industry and the research community to reexamine some assumptions (on architecture, power budget, cost, the impact of interference, etc.) commonly held on autonomous driving systems, which in turn may lead to a series of new research opportunities, such as (i) reexamining the entire system design in the backdrop of the completely different power budget and space budget, (ii) adding redundancy and reliability to low-end devices in a cost-effective manner for level-4 autonomous driving, (iii) reconsidering the research on fine-grained scheduling optimizations under the new deployment settings, (iv) reexamining the design, optimization, and deployment of other kinds of autonomous driving applications (e.g., those based on strongly-integrated multi-task DNNs).

This work, in addition, contributes an open-source research kit named *Single-Card Autonomous Driving Research Kit (SCAD)*. It allows the reproduction of the results in this study, easy deployment of autonomous driving on a Jetson card, as well as the generation of various autonomous driving Directed Acyclic Graphs (DAGs) for experiments and benchmarking, offering a vehicle for the community to more quickly advance the research in this field.

Research in autonomous driving is an active field, with many papers published on important research points, scheduling layers of DNNs (BL20; BL18; BZZL18), scheduling memory allocation of DNNs (BL19), applying real-time scheduling (SSA⁺18), supported with micro-services (TYLG20), heterogeneity study (TLL⁺20; LTL⁺21), and so on. This study builds on the many prior research efforts but has a very different objective and level of focus. Rather than coming up with a better solution to a research point, this work aims to understand the whole application's performance deficits, the reasons, and practical solutions.

It is worth noting that besides performance, accuracy, and energy efficiency, there are other aspects (e.g., redundancy and security) in our product to meet the full industry standard. This support increases the cost but only modestly, as Section 2.9 discusses.

2.2 Autonomous Driving Application and Device

In industry, level-4 is considered to be fully autonomous driving, which is of the primary interest of the current industrial research. This section explains the high-level workflow and presents an example level-4 autonomous driving application, which will be used in the following discussions.

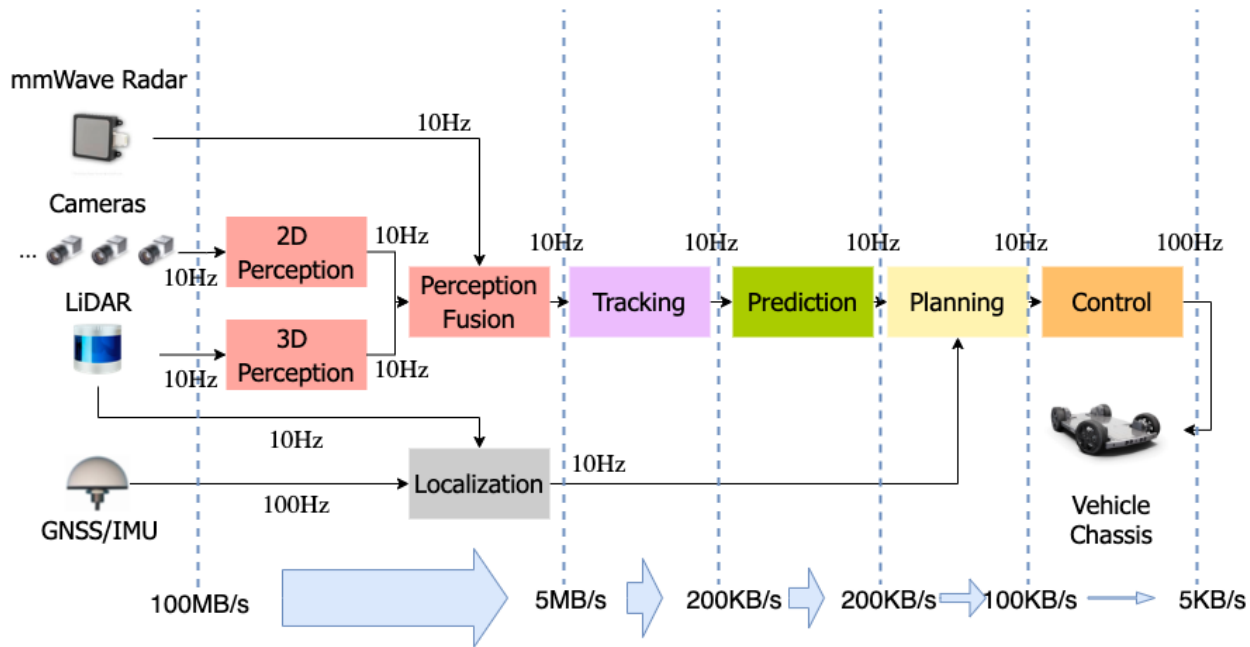


Figure 2.1: High-level workflow of a level-4 autonomous vehicle.



Figure 2.2: Detailed tasks within an example autonomous driving application (ADApp) derived from Autware. We categorize the tasks into the modules in Figure 2.1: Sensing (blue), Perception (red), Localization (gray), Tracking (purple), Prediction (green), Planning (yellow), and final control output (orange)

2.2.1 High-level Workflow

Figure 2.1 illustrates the high-level structure of a level-4 autonomous vehicle’s workflow. (i) Starting from the left side, the sensing system generates raw sensing data from mmWave radars, LiDARs, cameras, Global Navigation Satellite System (GNSS) receivers, and Inertial Measurement Units (IMUs), where each sensor produces raw data at its own frequency. For instance, the cameras capture images at 30 FPS, the LiDARs capture point clouds at 10 FPS, the GNSS/IMUs generate positional updates at 100 Hz. Certain temporal and spatial synchronizations (LYL⁺21) are used. To collect enough info, a vehicle is often equipped with multiple sensing devices (e.g., cameras) of a type. (ii) The perception components create a comprehensive perception list of all detected objects. The perception and localization system consumes 100 MB/s of raw sensing data and produces 5 MB/s of semantic data in real time. (iii) The perception list is then fed into the Tracking node at 10 Hz to create a tracking list of all detected objects. (iv) The tracking list is then fed into the Prediction node at 10 Hz to create a prediction list of all objects. The Tracking and Prediction system consumes 5 MB/s of perception inputs and further reduces the data size to 200 KB/s. (v) At last, both the prediction results and the localization results are fed into the Planning node at 10 Hz to generate a navigation plan (LLT⁺20). (vi) The navigation plan is then fed into the Control node at 10 Hz to generate control commands, which are finally sent to the autonomous machine for execution at 100 Hz.

For every 10 ms, the autonomous machine needs to generate a control command. If any upstream node, such as the Perception node, misses the deadline for generating output, the Control node must still generate a command, which would be based on out-of-date info and hence potentially lead to disastrous results as the autonomous machine would then be essentially driving blindly without timely participation from the perception unit. Real-time response is hence essential for all the main components of an autonomous driving application.

2.2.2 ADApp: A Set of Level-4 Autonomous Driving Applications

This study is conducted on ADApp, a set of six autonomous driving applications built on Autoware (the15), the most commonly used open-source level-4 autonomous driving software by the non-profit Autoware Foundation, created for synergies between corporate development and academic research to enable autonomous driving technology. It is worth noting that in our product, significant extensions have been made to Autoware to meet the industrial Level-4 requirements, including multiple input image sensing streams missing in the default Autoware workload, the added support of AI accelerators, and the use of real-time scheduling.

The DAG in Figure 2.2 shows the backbone architecture of Autoware-based applications, a concretization of the main functions shown in Figure 2.1. The architecture is representative of

Table 2.1: Differences among the ADApp applications

Application name	2-D Model		
	model name	num	description
ADy288	Yolo-v3-288	10	ten Yolo-V3 models processing ten streams of videos of 288×288 resolution each frame
ADy416	Yolo-v3-416	5	five Yolo-V3 models processing five streams of videos of 416×416 resolution each frame
ADy608	Yolo-v3-608	3	three Yolo-V3 models processing three streams of videos of 608×608 resolution each frame
ADs288	SPP-v3-288	10	ten SPP-V3 models processing ten streams of videos of 288×288 resolution each frame
ADs416	SPP-v3-416	5	five SPP-V3 models processing five streams of videos of 416×416 resolution each frame
ADs608	SPP-v3-416	3	three SPP-V3 models processing three streams of videos of 608×608 resolution each frame

level-4 autonomous driving, similar to the DAGs in other industrial autonomous driving systems (e.g., DiDi¹). Each node in the DAG is a *task*, corresponding to a process in the underlying middleware Robot Operating System (ROS) on which the application executes. There are 28 tasks in the application, forming complicated producer-consumer relations, represented by the edges. The colors of the tasks indicate the corresponding function blocks in Figure 2.1.

The right half of Figure 2.2 shows that the application gets 2D images from a number of cameras and 3D points cloud from a LiDAR scanner. The *velodyne_driver* handles messages from the LiDAR device and down-samples them with *voxel_grid_filter*. The node *lidar_point_pillars* detects 3D objects with a PointPillars DNN model (LVC⁺19), and 2D objects with a number of YOLOv3² DNN models (RF18) (one for each image stream). Both are the most commonly used models in autonomous driving. The node *range_vision_fusion* collects the detected objects from multiple DNNs and outputs the fused results. The node *imm_ukf_tracker* uses the IMM-UKF-PDA tracking algorithm (Sch17), which utilizes three combined Bayesian filters to simultaneously tackle association uncertainties, motion uncertainties, and estimate non-linear stochastic motion model in real-time. The node *native_motion_predictor* generates predicted angles and velocities (KTM⁺18) from which the node *costmap_generator* creates an obstacle map to show the passable probability around the car.

The left half of Figure 2.2 shows that the application collects the location info (in nmea messages) from Global Navigation Satellite System (GNSS), processed by the node *nmea2tfpose* and then handled by the *ndt_matching* node to get the position of the vehicle on the map and its velocity with the Normal Distribution Transform (NDT) algorithm (BS03). The info is then forwarded to other nodes by the *pose_relay* and *vel_relay*, respectively. The application loads and processes the given global *waypoints* (from pre-loaded 3D map) (*waypoint_replanner*, *lane_rule*, *lane_stop*). Then, without considering surrounding obstacles, *lane_select* generates a selected lane based on the given map, current position, and current velocity. It then generates local feasible *waypoints* (i.e., trajectories) by *astar_avoid* with A^* (HNR68) and hybrid-state A^* (DTMD10) graph-searching algorithms. The node *velocity_set* then calculates the velocity to set, and the node *pure_pursuit* calculates the actuation commands with the Pure Pursuit algorithm (Cou92).

The architecture is the de facto architecture in level-4 autonomous driving systems. It is hence shared by the six applications in ADApp. The differences among the six applications are in the 2-D perception component: Perception is the performance bottleneck of autonomous driving software; variations in other components may affect the recommended actions to take

¹<https://blogs.nvidia.com/blog/2021/05/17/didi-nvidia-drive-self-driving-robotaxis/>

²YOLOv3 is used because the more recent model YOLOv4 is not supported by Autoware by default; our separate tests show similar observations when YOLOv4 is used.

for the vehicle but have no observable effect on the end-to-end performance of the software. Performance is the focus of this study. In perception, as 3-D perception devices (lidar) are much more expensive than 2-D perception devices (camera), common vehicles are equipped with only one lidar but differ in the number of cameras. The design of the ADApp set reflects the variations in the real world.

Table 2.1 lists the six applications in ADApp and their 2D perception components. The first three use Yolov3 (RF18) as the base for 2D perception; they differ in the input video resolutions. The lower the resolution is, the smaller the Yolov3 model is, and the lower the accuracy (and also time and device cost) is. To compensate for the lower accuracy, the vehicle is often equipped with more cameras, which explains the different numbers of 2D models used in the three applications. The bottom three applications use SSP (HWF⁺20). SPP is another commonly used 2D perception model. As a variation of Yolov3, it gets the best features in Max-Pooling layers. The settings of these three applications resemble the first three otherwise. These six applications are comparable to many level-4 autonomous driving systems in the industry (way21; bai21). To make the results easy to reproduce, the used Yolov3 and SPP are both in the standard form; no custom compression or other changes are made to them.

The inputs used for all the experiments are from Autoware-AI (AA20). Following the common practice, each experiment first loads the static information (e.g., car configuration, 3D map, and lane information) before execution and uses the recorded ROSBAG to playback the dynamic data (e.g., point clouds) to simulate the practical computing situation. Additionally, we add the artifact image message flows into the Autoware system to simulate the situations that have one LiDAR and multiple cameras.

Table 2.2: Configurations of the Target Device (Jetson AGX Xavier)

Hardware	¹ CPUs	² GPUs	GPU Throughput	GPU Power	DLAs	DLA Throughput	DLA Power	Cost (USD)
	1	1 (1377 MHz)	2.8 TFLOPS FP16	30W	2 (1395.2 MHz)	2.5 TFLOPS FP16	0.5-1.5W	\$699
Software	Jetpack 4.4.1 SDK (L4T 32.4.4 OS), CUDA 10.2, OpenCV 3.4.3 compiled with CUDA, TensorRT 7.1.3.0, cuDANN 8.0.0							

¹ 8-core NVIDIA Carmel Arm v8.2 64-bit CPU 8MB L2 + 4MB L3

² NVIDIA Volta architecture with 512 NVIDIA CUDA cores and 64 Tensor cores.

2.2.3 Device

In the current industry, autonomous driving software as complicated as any of the six applications in ADApp relies on high-end autonomous driving devices. For instance, according to a previous literature (LTZG17), the platform from a leading level-4 autonomous driving company

consists of two compute boxes, each equipped with an Intel Xeon E5 processor and four to eight Nvidia GPU accelerators, connected with a PCI-E bus. The whole system costs over \$30K and, at its peak, consumes over 3000W power, making the whole solution unaffordable to average consumers.

Our targeted device is the off-the-shelf Jetson Xavier SoC. Table 2.2 lists the hardware and software configurations. The main computing units include one 8-core Carmel CPU based on ARM v8 ISA, one NVIDIA Volta-class integrated GPU, and two Deep Learning Accelerators (DLA). The integrated Volta GPU has 20 TOPS for INT8 operations and 2.8 TFLOPS for FP16 operations. DLA is a specialized accelerator for Deep Learning inference in cost-sensitive applications. It focuses on executing four common DNN operations: convolutions, activations, pooling, and normalization with power efficiency. Although it has 10 TOPS for INT8 operations, which is half lower than the integrated Volta GPU, its power consumption is much lower than the integrated Volta GPU. Jetson is also equipped with a Programmable Vision Accelerator, useful for image preprocessing. However, like many products in the autonomous driving industry, our autonomous driving system deploys preprocessing on sensors themselves, not central computing devices. It is hence not considered in this exploration. The memory on the device is 256-bit LPDDR4 Memory with a bandwidth of 136 GB/s. The overall cost of this Jetson Xavier SoC on the market is \$699.

The software configurations include Jetpack 4.4.1 SDK that runs L4T 32.4.4 OS. It is equipped with CUDA 10.2, OpenCV 3.4.3 compiled with CUDA, TensorRT 7.1.3.0, and cuDNN 8.0.0.

2.3 Observations and Analysis of the Default Executions

In the current industry, for an autonomous driving application as complicated as those in ADApp, the system usually deploys the tasks on multiple boards equipped with many computing units. This section reports the observations when we try to deploy each of these applications on a single Jetson AGX Xavier board in the default manner. These observations help us understand the limitations of the current deployment and resource management.

2.3.1 Observations

The default performance is shown in the top two segments of Table 2.3. The first segment is about the case that uses the default Linux time-sharing scheduling (version of Linux 2.6.23) (sch21). The second segment is about the case that uses ROSCH (SSA⁺18), the latest published scheduling algorithm specially designed for Autoware. (Note that in all those experiments, as well as all the other experiments reported throughout this dissertation, the planning module of the

autonomous driving applications is bound to two CPU cores isolated from the other tasks. Such isolation is a custom in autonomous driving as it is important to ensure the planning module gives out planned actions in time.)

From the table, we can see the following:

- In the Linux default case, at every time frame, all six applications exceed the allowed latency when processing the inputs. The most sluggish module is the 2D perception, taking time about twice as much as its allowed latency.
- ROSCH, the scheduling algorithm specially designed for autonomous driving, actually makes the applications perform even worse. None of the applications are able to make any progress on any input. The sensing module and the planning module are the only modules that produce outputs. The other modules are all time out. (Note, as mentioned in Section 2.2.1, the planning module always gives outputs in a fixed frequency; the outputs are based on obsolete information if there are delays in the earlier stages of the DAG.)

2.3.2 Analysis

The executions under the default Linux time-sharing schedule are no surprise: The scheduling scheme is not designed for real-time autonomous driving workload; its unsatisfying performance has been the motivation for the prior efforts for designing specialized scheduling algorithms for such workloads. The much worse results from ROSCH, an algorithm customized to autonomous driving, were a surprise.

Our detailed analysis shows that the reason is starvation caused by the scheduler. The scheduling algorithm in ROSCH is a variation of the most popular real-time scheduling algorithm Heterogeneous Earliest-Finish-Time (HEFT). Figure 1 shows the algorithm. This algorithm allows the computing units to have different speeds, but assumes a node in the DAG is a scheduling unit: The entire node is assigned to a processor. At a high level, the algorithm has two steps, ranking the tasks based on their rank metric and then scheduling the tasks in their ranking. It assigns a task to the processor that minimizes the estimated earliest finish time (EFT) of that task. If a computing unit is assigned multiple tasks, the algorithm assigns priorities to those tasks based on their ranks. The scheduling algorithm used in ROSCH is the same as HEFT except that they use different metrics for task ranking.

When this algorithm is applied to the autonomous driving applications on a single Jetson AGX Xavier card, starvation happens. The reason is as follows. Because there are only six CPU cores left after the reservation of two cores for the Planning module, many DAG nodes are

Algorithm 1: Original HEFT algorithm for DAG scheduling (THW02)

- 1: Set the computation cost for each node and the communication cost for each edge in the DAG with their mean values
- 2: Post-traverse the DAG calculate *rank* for each task

$$rank(n_i) = \overline{w}_i + \lim_{n_j \in succ(n_i)} (\overline{c}_{i,j} + rank(n_j))$$

where the $succ(n_i)$ is the set of immediately connected successors of task n_i , $\overline{c}_{i,j}$ is mean value of communication cost of edge (i, j) , and \overline{w}_i is average computation cost of task n_i . For the task n_{exit} , the rank value is

$$rank(n_{exit}) = \overline{w}_{exit}$$

- 3: Create an ordered list (taskList) of the nodes based on the descending order of their *rank*.
 - 4: **while** there are unscheduled tasks in the list **do**
 - 5: $n_i =$ next task in the taskList
 - 6: **for** each processor p_k in the processor-set Q **do**
 - 7: Compute the earliest finish time, $EFT(n_i, p_k)$, with the *insection-based scheduling formula*
 - 8: Assign task n_i to the processor p_j that minimizes EFT of task n_i .
 - 9: **end for**
 - 10: **end while**
 - 11: Assign priorities to the tasks assigned to the same processor based on their ranks.
-

scheduled to the same core. Some tasks in the Perception module are assigned to the same core as the tasks in the Sensing module. Based on the scheduling algorithm, the ROSCH scheduler gives higher priorities to the Sensing nodes than to the Perception nodes. Each node is a process consisting of multiple threads. All these threads inherit the priority of the process. As a result, the threads of the Sensing node, for their higher priorities, hog the CPU core throughout the execution. The Perception nodes on the same core are starved. As the other modules depend on the output of the Perception module, they make no progress either. The exception is the Planning module. As mentioned before, the module always gives outputs in a fixed frequency; as it receives no updates from the other modules, its outputs are out of date and useless. Such a problem does not show up in high-end devices where all main modules can get their dedicated computing units.

2.4 Just-In-Time Affinity and Priority Adjustment

To address the starvation problem, we change the scheduling algorithm such that the core affinity and priority adjustments are made in a just-in-time manner. Before explaining the improved scheduling scheme, it is necessary to briefly review the background of Linux scheduling systems.

2.4.1 Background on Linux Schedule

A task in Linux may be put into one of two scheduling queues: SCHED_FIFO and SCHED_OTHER (sch21). SCHED_FIFO is first in first out. It can only be applied to threads with static priorities above 0. When the SCHED_FIFO thread becomes runnable, it will immediately preempt any currently running SCHED_OTHER threads. SCHED_OTHER is used in the default Linux time-sharing scheduling. It is only applicable to threads at static priority 0 that do not require special real-time mechanisms.

2.4.2 Just-In-Time Adjustment

In ROSCH, every task is put into a SCHED_FIFO queue and has a statically assigned priority as calculated by the HEFT algorithm. In our improved algorithm, every task is put into a SCHED_OTHER queue and has the same default priority. Changes are made when a work item is put into the input queue of a task, and the main thread of that task is about to be invoked to process that item. At that moment, the main thread of that task is set to SCHED_FIFO, and its priority is changed to p , the level calculated by the default HEFT algorithm. Please note the changes

are made to only the main thread, and the other threads of that task keep the default priority. As soon as the main thread finishes processing the item, it is set back to `SCHED_Other`, and its priority is reset to the default.

Listing 2.1 shows the implementation. On Lines 1 and 2, `core_id` and `priority` hold the CPU core and priority that the default HEFT algorithm assigns to the node that this current main thread belongs to. Lines 10 to 15 bind the thread to the CPU core and put the thread into the `SCHED_FIFO` queue with priority. The thread is now being put into real-time mode to do its work. After the work is done, lines 22 to 28 unset the core affinity and put the thread back into the `SCHED_OTHER` queue with the default priority.

In this JIT scheme, the time for a node to hold high priority is reduced, and also the non-critical assistant threads are not promoted to a high priority, which also reduces the core contention.

Listing 2.1: Implementation of the Just-In-Time priority adjustment

```
1 /*
2 Codelet to run after the thread is triggered
3 */
4 int core_id = 1; // the core to bind
5 int priority = 98; // the priority to set
6 pthread_t this_thread = pthread_self();
7 cpu_set_t cpuset;
8 CPU_SET(core_id, &cpuset);
9
10 // set affinity and priority for real-time run
11 pthread_setaffinity_np(this_thread, sizeof(cpu_set_t), &cpuset);
12 struct sched_param params;
13 params.sched_priority = priority;
14 pthread_setschedparam(this_thread, SCHED_FIFO, &params);
15
16 /* ----- main work ----- */
17 ... ..
18 /* ----- */
19
20 // reset the affinity and priority
21 // to the default
22 for (int i = 0; i < USABLE_CORES; i++)
23     CPU_SET(i, &cpuset);
24 pthread_setaffinity_np(this_thread, sizeof(cpu_set_t), &cpuset)
25 params.sched_priority = 0;
26 pthread_setschedparam(this_thread, SCHED_OTHER, &params);
```

2.4.3 Performance and Analysis

Segment three in Table 2.3 reports the performance of the six applications after the improved scheduling scheme is applied to ROSCH.

- We can see that the starvation issue is gone. Every module makes progress.
- Compared to Default Linux Time-Sharing performance, the Sensing and Localization modules show significant speedups because the real-time priority helps them get the resource when their main thread needs to do the work.
- Compared to Default Linux Time-Sharing performance, the 2D perception and 3D perception, however, do not show speedups. Detailed analysis shows that it is because these tasks involve both CPU and GPU code, and the GPU code takes most of the time. So even though starvation is avoided, the six applications still miss the deadline in 100% of the time.

2.5 Migration to Take Advantage of All Accelerators

To understand how effectively the applications take advantage of the accelerators on Jetson AGX Xavier, we use the NVIDIA NSight to profile the executions of the applications (after the JIT improvement is made to ROSCH). The result indicates that the applications use the GPU extensively, but leave the two DLAs unused at all.

2.5.1 Changes

DLAs are special deep-learning accelerators designed to accelerate some common operations in DNNs. For a DNN to take advantage of DLAs, however, the applications must be written with TensorRT in some required form. TensorRT (ten21) is a library developed by NVIDIA for faster DNN inference on NVIDIA devices.

The default Autoware, despite being officially migrated to AGX Xavier, does not contain the use of the right TensorRT APIs to make use of the DLAs. As a result, the six applications in ADApp do not have those APIs used either. The lack of usage of special accelerators is not uncommon in autonomous driving software because of (i) the extra efforts needed to understand the accelerators and their usage, (ii) the benefits from those accelerators are often less satisfying (as we will soon see). To make the applications take advantage of DLAs, we revised their implementations. Take YOLOv3 as an example. In the workflow of TensorRT, as shown in Figure 2.3, to build a DLA version of the TensorRT engine for YOLOv3, we set the default execution device to DLA

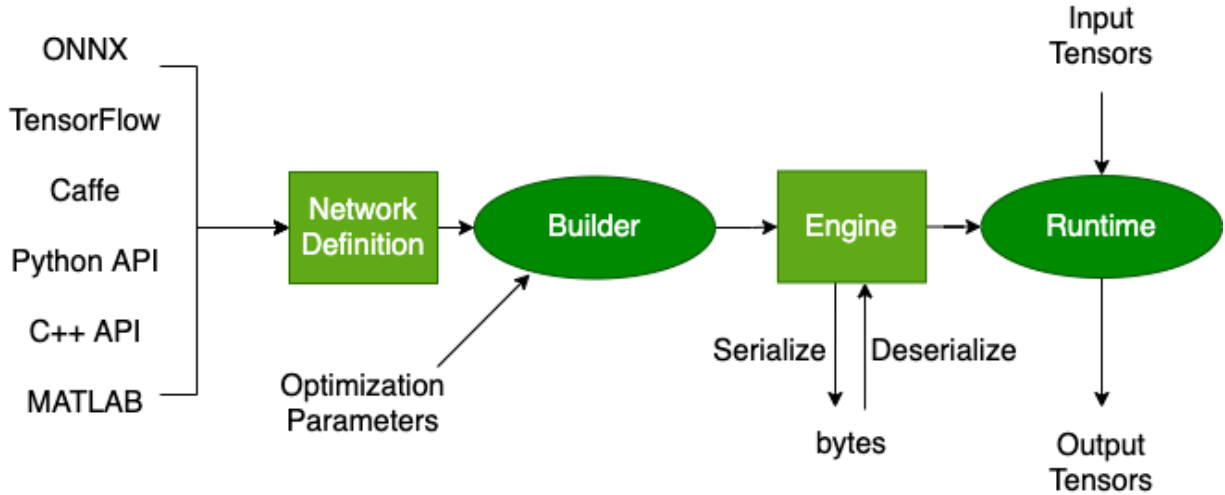


Figure 2.3: The TensorRT Workflow. (ten21)

through the invocation of the TensorRT API `setDefaultDeviceType(DeviceType::kDLA)`. DLAs do not support all kinds of DNN operators. So to prevent some layers that are unsupported by DLAs (e.g., *LeakyReLU* activation function), it is necessary to make those layers able to fall back to GPUs, which is achieved by adding the invocations of TensorRT API `setFlag(BuilderFlag::kGPU_FALLBACK)`. As there are two DLAs on a Jetson AGX Xavier, we can make a particular DLA to execute a kernel through `setDLACore(dlaCore)` (`dlaCore = 0` or `1`) after the deserialization of the TensorRT engine in Figure 2.3.

Now with DLAs being put into consideration, the scheduling algorithm would need to decide not only the schedules of the nodes on CPUs but also the assignments of the nodes to the three accelerators (one GPU, two DLAs). The design of ROSCH (and HEFT) assumes every node in the DAG can run on any computing unit, which is not the case for the autonomous driving workload where some nodes can run only on CPU, part of some the other nodes can run on GPU or DLA.

To address the issue, we extend the DAG scheduling algorithm in the previous section to make it workable for systems with multiple types of accelerators. The pseudo-code is shown in Algorithm 2. The design follows two principles (i) being practical and (ii) maximizing the quality of the final schedule. An observation is that as our target is a single low-end device, the number of accelerators is very limited (three in the case of Jetson AGX Xavier). Our design hence favors simplicity and result quality over scalability. The basic idea is to enumerate all the possible viable assignments of the task nodes to the accelerators. For each assignment, we instantiate the DAG with the measured performance of the tasks and the communication cost and then run the scheduling algorithm in the previous section on that DAG to obtain a

schedule. In the end, the schedule that gives the best performance is chosen. The assignment corresponding to the DAG gives the final assignments of the task nodes to the accelerators.

Algorithm 2: DAG Instantiation Based Scheduling

```

1:  $D$ : the DAG
2:  $A = \{a_i\}$ : the set of accelerators of one or more kinds
3:  $V = \{v_i\}$ : the set of task nodes in  $D$ 
4:  $E = \{e_i\}$ : the set of edges in the  $D$ 
5:  $b_{i,j}$ : 1 if  $v_i$  can run on  $a_j$ , 0 if  $v_i$  cannot run on  $a_j$ 
6:  $s$ : a valid assignment from  $V$  to  $A$ , that is,  $\{b_{i,s(i)} = 1$  for each  $v_i\}$ 
7:  $S = \{s\}$ : the set of valid assignments
8:  $Measures = \{\}$ 
9: for each  $s$  in  $S$  do
10:    $taskPerformance \leftarrow$  measure the performance of tasks under  $s$  in the default
      schedule
11:    $d =$  Instantiate  $D$  with  $taskPerformance$ 
12:    $sch = scheduleAlgorithm(d)$ 
13:    $appPerf \leftarrow$  measure the performance of the application under  $sch$ 
14:    $Measures.add(sch, appPerf)$ 
15: end for
16:  $finalSch = Measures.bestPerf()$ 

```

2.5.2 Performance and Analysis

Segment four in Table 2.3 reports the performance after the changes. Compared to the results in Segment 3. where DLAs are not used, the performance becomes worse. The 2D perception becomes faster in two applications (ADy608 and ADs608) but slower in the other applications. The 3D perception becomes slower in all applications.

The reason for the observed performance degradation is that the DLAs are used but only sporadically. DLAs are not as versatile as GPUs. They cannot support some layers in the DNNs. To see the effect of this, using the NVIDIA Nsight profiler, we profile the execution of a YOLOv3-608 \times 608 DNN that has been revised to use DLA. Figure 2.7 (b) shows the timeline of one inference by the DNN. Compared to the default execution, which does not use the DLA at all, this execution does utilize the DLA, but in only several small-time fragments; most of the time, the execution still happens on the GPU.

A detailed analysis uncovers the reasons. According to the DLA document (ten19), the *LeakyReLU* activation function in YOLOv3 is not supported by DLA. In YOLOv3, Convolution,

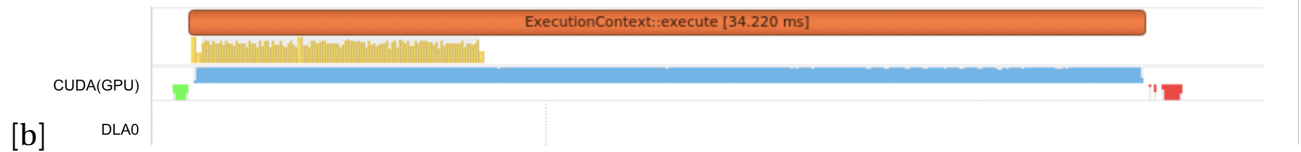


Figure 2.4: Default execution



Figure 2.5: After changes to use DLA

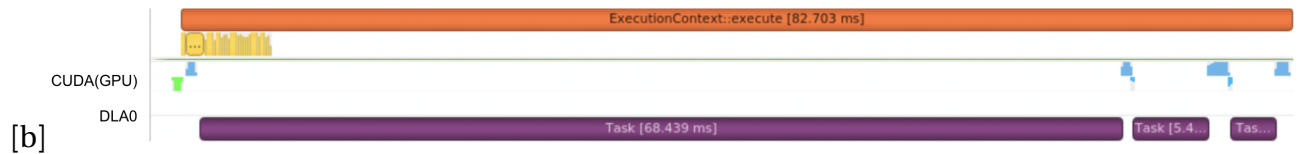


Figure 2.6: After hardware-aware model customization

Figure 2.7: The timeline of the execution of one inference by the YOLOv3-608 × 608 DNN, collected via NVIDIA Nsight.

normalization, and activation layers are grouped into one Convolution Block, and the process of feature extraction is composed of multiple Convolution Blocks (There are 57 Convolution Blocks in YOLOv3). If the DLA engine does not support a particular layer, the layer will fall back to GPU execution. In other words, the execution falls back to GPU at least 57 times when making one inference of YOLO-v3. Each of them causes extra memory allocation overhead and switching overhead on GPU and DLA.

Moreover, the computing engine in TensorRT can hold only up to eight subgraphs. In other words, the falling back to a GPU can happen only up to eight times in an execution. So after the eighth falling back, all remaining layers in the DNN have to run on GPU. As a result, most of the Convolution blocks of YOLOv3 still compete for the GPU resource rather than benefit from the DLAs.

The six autonomous driving applications include multiple 2D perception DNNs. The switching back and forth between GPUs and DLAs of those DNNs creates interference, especially in the applications with more 2D perception DNNs, which explains the observed increased running time.

2.6 Hardware-Aware Model Customization

To address the severe under-utilization of DLAs, the next measure we take is to make changes to the DNN models to make them better fit the restrictions of DLAs. We adhere to three principles: (i) the changes should be minimal; (ii) the changes should cause marginal or no accuracy loss; (iii) the changes can bring significant performance benefits.

2.6.1 Changes

Based on the analysis described in the previous section, the changes we make are to replace the *LeakyReLU* activation functions in every convolution block in the Yolov3 and Yolov3-SPP models with the standard *Relu* activation function—which is supported by DLAs. One potential concern is the impact of the changes on the accuracy of the DNN. Our experiments show that after re-training (on VOC 2007 dataset (EVGW⁺) which is used in the training of the original v3 and SPP models (ZHZ⁺19)), the modified models converge to the similar accuracy as the original models do, as Table 2.5 shows.

The results echo the observations made in the previous literature (XWCL15), the effect of using *LeakyReLU* over *Relu* is that the former helps the training converge slightly faster, but the two activation functions do not cause differences in the inference accuracy of Yolov3.

2.6.2 Performance and Analysis

Figure 2.6 shows the timeline of the execution of one inference of the modified YOLOv3-608 × 608. The changes indeed allow the entire DNN model to run on the DLA. It brings two-fold benefits: (i) No falling back to GPUs is needed, and hence the falling back overhead is avoided; (ii) The GPUs are now free of the interference and competition from those models offloaded to DLAs. Because the frequency of DLA is lower than GPU, the inference takes more time than it on GPU. But for an autonomous driving application with many tasks, the effective use of DLA allows the reduction of the load of GPU while still keeping the tasks on DLA meet the deadlines.

Segment five in Table 2.3 reports the performance of the six autonomous driving applications after the change. The same scheduling algorithm as in Section 2.5 is used. Now the performance shows significant improvement. All the modules in all of the six applications can now complete their work within the expected latency (As the table caption marks, 10% over the expected latency is tolerable as such a slack allows real systems to tolerate random fluctuations caused by system noise in real executions.) The applications meet the real-time requirements

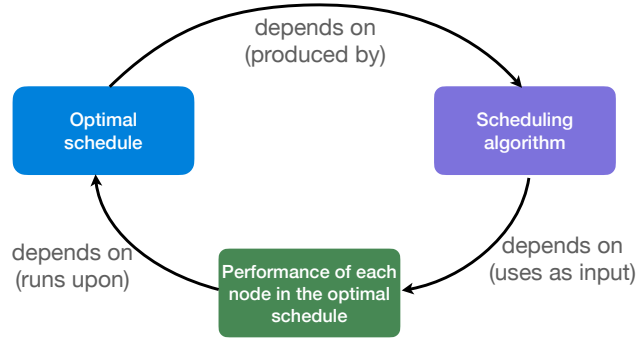


Figure 2.8: The cyclic dependence in the scheduling problem

entirely. Compared to the results in Segment 4, the 3D perception sees about $1.5\times$ speedups, and the 2D perception sees $2\text{-}2.2\times$ speedups.

2.7 Other Experimented Optimizations

In addition to those improvements, we have tried some other optimizations. One of them worth briefly mentioning is an idea we call *iterative co-run aware scheduling*.

Scheduling algorithms compute the schedules based on the performance of the tasks to schedule. But the performance of the tasks changes as the schedule changes. So ideally, the task performance used by all the scheduling algorithms as the basis *to find the optimal schedules* should be the performance of the tasks *under the optimal schedules*, which forms a cyclic dependence as illustrated in Figure 2.8.

All scheduling algorithms for autonomous driving have simply used the performance measured in some default schedules as the basis instead. We tried to improve it through an iterative scheme, which resembles the Expectation-Maximization algorithm (MK07) in machine learning. It starts with the measured performance of the tasks under a default schedule. It then enters a loop. Each iteration consists of two steps: (i) It applies the scheduling algorithm on the just measured performance of the tasks to calculate a new schedule; (ii) it runs the tasks under that new schedule to remeasure the task performance. It then goes back to step (i) and continues the loop until either the schedule remains unchanged (convergence) or the algorithm times out (i.e., exceeds a threshold).

The result shows that the method does not yield visible improvement, as Segment six in Table 2.3 reports. Our analysis shows that the reason is that the performance of those applications is mostly determined by the performance of the accelerators. Because the scheduling algorithm in Section 2.5, *DAG instantiation based scheduling* already considers all the possible placements of a task on each accelerator, and there are only three accelerators, the algorithm

already yields good schedules even without the iterative process. For other situations (e.g., many-core CPUs), the algorithm might be more useful; the exploration is beyond the scope of this work.

2.8 Power, Synergy, Insights and SCAD Kit

Making level-4 autonomous driving possible on a single Jetson AGX Xavier card also gives significant power benefits. Compared to the system mentioned in prior literature (LTZG17) that consumes as much as 3000W power, our system consumes only 32W, two orders of magnitude less. Even compared to level-2 autonomous driving systems such as NVIDIA Drive Pegasus (500W), the power consumption is $15\times$ less.

A note worth mentioning is that the significant performance improvements in Segment Five in Table 2.3 come not from an individual technique but from the synergy of the three measures, JIT priority adjustment, migration to accelerators, and hardware-aware model customization. If we apply only the hardware-aware model customization or the migration to accelerators, the applications would still suffer starvation. If we do not apply the code migration to accelerators, even if the DNN models went through the hardware-aware customization, their code still cannot run on DLAs.

In Section 2.1, we have mentioned three main research questions. We summarize the answers offered by this work as follows:

- Answer to RQ1: The current perceived resource needs of autonomous driving are far more than necessary.
- Answer to RQ2: Resource under-utilization is the key issue to address to lower the cost of autonomous driving.
- Answer to RQ3: Level-4 autonomous driving can achieve real-time performance on a single off-the-shelf card for as little as \$699. To achieve that, it needs a synergy of just-in-time affinity and priority adjustment, appropriate code migration to accelerators, and hardware-aware model customization.

A side product from this study is the SCAD kit, a package that consists of not only the six ADApp applications but also a flexible generator of autonomous driving applications customized to fully utilize DLAs, builtin just-in-time priority adjustment, and several options of scheduling algorithms (non-iterative, iterative scheduling). The kit allows easy deployment of autonomous driving on a Jetson card, as well as the generation of various autonomous driving DAGs for experiments and benchmarking through its reconfiguration scheme. Nodes

of various tasks can be easily added or removed from the DAG by revising a configuration file. By addressing the many limitations of existing frameworks for autonomous driving research, SCAD offers a vehicle for the community to more quickly advance the research in this field. It will be released to the public after the publishing of this work.

2.9 Implications

The results from this work are significant. Before this work, no prior work had ever shown that level-4 autonomous driving is possible to achieve real-time performance on a single off-the-shelf low-end car-like Jetson AGX Xavier. The cost, \$699, is an order of magnitude less than those used in the current industry (e.g., the \$30k system described in previous literature (LTZG17)).

But note that besides the real-time performance requirement, a level-4 autonomous driving system must have enough redundancy, reliability, and security to ensure safety. Our autonomous driving system is a full system, the design of which consists of the implementations of security and redundancy (both heterogeneous and homogeneous approaches) besides the core module presented in the dissertation. Its software stack consists of a middleware to facilitate communications between nodes, as well as a Linux operating system to provide basic system services. This dissertation intentionally focuses on the presentation of the core module because it dominates the computing resource usage and execution time. Our proprietary middleware, for instance, is based on highly optimized nanomsg³, incurring less than 3% of CPU overheads and communication latency.

There are two approaches to adding redundancy, with both considered in our product design. A heterogeneous approach is to have multiple channels to create their own independent and diverse perceptions of the world (tru21) for cross-checks. A homogeneous approach is to run the same workload on two identical devices with redundant power supplies equipped on each device (TSV⁺20). What this dissertation focuses on is the performance of the core AI component. Being able to run the entire application on such a low-end card prepares a good basis: Even if the additional redundancy doubles or triples the cost, the cost would still be an order of magnitude lower than state-of-the-art.

Besides pointing out a promising path for the industry to drastically reduce the cost and power of autonomous driving systems, the overturning of the common perceptions by this work also suggests some new research opportunities. Some examples are as follows:

- Architecture design: As the cost and power consumption drops dramatically, it would be valuable to reexamine the design of the entire autonomous driving architecture in terms

³<https://nanomsg.org/documentation.html>

of the budget allocation for various components, reinforcement of security or reliability, and so on.

- **Software design:** The changed assumptions on the cost and computing resource suggest the need for rethinking the design of the autonomous driving software, such as the complexities and structures of adoptable DNNs, inter-component communication, and synchronizations, scheduling, and so on.
- **Optimizations:** There has been lots of research on the optimization of certain points in autonomous driving. They may be worth reexamination. For instance, as everything now runs on a single card, inter-card communication becomes less important, but how to effectively improve the reliability of the low-end device becomes more important. The many optimizations proposed before may work differently in this single-card setting. The research to schedule each layer of a DNN on GPUs to strike an accuracy-power-speed tradeoff (BL20), for example, may now need to consider the (core, data path, and memory) contentions from other DNN models running on both GPUs and other accelerators (e.g., DLAs).

2.10 Related Work

To cope with the complexity of level-4 driving, many autonomous machine companies resort to *ad hoc* solutions to ensure on-time autonomous machine product release. These *ad hoc* solutions are often product-specific, evolve through trial and error, and are hard to generalize to other autonomous machine designs, hence leading to high re-engineering costs for each product. For instance, in the solution presented in (YHX⁺20), the computing platform consists of a Xilinx Zynq UltraScale+ FPGA board, an on-vehicle PC machine equipped with an Intel Coffee Lake CPU and an Nvidia GTX 1060 GPU. Many manual efforts are spent in customizing the software deployment, scheduling, and resource management to such *ad hoc* solutions (YHX⁺20; LG20).

LoPECS (TLL⁺20) is an effort to make autonomous driving software runnable on an embedded low-end device. The autonomous driving software is, however, much simpler than level-4 autonomous driving applications. Its vision sensing, for instance, consists of only one camera, and its perception module consists of only one CNN model; there are no 3D perception, other 2D perception models, or perception fusions. Even with that simplicity, it still needs to offload extensive computations to the cloud, the uncertain delays incurred by which add risks to the safety of the system. Due to the simplicity of the problem in the prior work, none of the complexities encountered in this work—such as the priority-related starvation, the use

of non-GPU accelerators, and the scheduling for multi-type accelerators—manifests in their work.

Several studies proposed multi-DNN schedulers on heterogeneous SoC. ApNet (BL18) applies approximation approaches to each layer of the DNN network and makes a trade-off between accuracy and latency. PredJoule (BZZL18) optimizes energy for running DNN workloads. It adjusts power configuration based on the latency of workloads. NeuOS (BL20) provides a system for running multiple DNN workloads for autonomous systems. It tries to balance three dimensions of optimization: latency, accuracy, and power consumption. The idea of the NeuOS is to schedule multi-DNN layer by layer. In each layer boundary (layer end execution), the scheduler will decide the power config of the whole system and the accuracy of the next layer based on assigned policy (min energy, max accuracy, or balanced both) and the deadline of each network.

Even though those studies provide useful insights at certain focused research points, the practical situation is more complicated in the autonomous driving system. It has various modules such as sensing, localization, and so on. Each module not only contains multiple DNNs to support but also depends on each other. The entire application consists of tasks of varied kinds, some runnable only on CPU, some involving CPU parts and accelerator parts, and the parts for accelerators may use only GPU or both GPU and other accelerators (e.g., DLA).

Many recent studies present schedulers for running multiple DNN workloads on the cluster. These cluster schedulers (XBR⁺18; MBS⁺20; GCS⁺19; LSCL20; NSK⁺20) attempt to make an optimal decision to allocate diverse resources to many requests of DNN workloads. They have several optimizing objectives, such as fairness and throughput, to design-related scheduling policies. They adopt deployment optimizations such as space sharing in Gandiva and placement sensitivity in Themis and Tiresias to increase resource utilization. The settings, constraints, and considerations on embedded systems are all different.

2.11 Conclusion

This dissertation has presented an experience that manages to make level-4 autonomous driving applications achieve real-time performance on a single low-end card at a cost an order of magnitude lower than the devices used in today’s autonomous driving industry. It achieves it by addressing three major deficits in the current practices through several practical solutions, which include *just-in-time affinity and priority adjustment*, *model migrations to all types of accelerators*, *DAG instantiation-based scheduling* and *hardware-aware model customization*.

This work points out a promising path for the industry to drastically lower the cost and power consumption of autonomous driving. By overturning the common assumptions on

the required computing resource by autonomous driving, the work suggests rethinking the current architectures, software, and optimization for autonomous driving and opens up many potential research opportunities in various dimensions.

Table 2.3: Execution time (mean± std) of each module in the ADApp applications on Jetson AGX Xavier and the miss rates. The ∞ represents timeout. The miss rate of a module is how often the module misses its expected latency (shown in the parentheses in the table header)—up to 10% over is allowed to tolerate system noises. The column Miss Rate shows the miss rates of the most sluggish modules (whose times are prefixed with an *), that is, the modules with the largest miss rate in the application.

Application	Running Time of Each Module (ms) [expected latency in brackets]							Miss Rate
	<i>Sensing</i> [100ms]	<i>3D Percept</i> [100ms]	<i>2D Percept</i> [100ms]	<i>Localization</i> [100ms]	<i>Tracking</i> [100ms]	<i>Prediction</i> [100ms]	<i>Planning</i> [10ms]	
1. Default Linux Time Sharing (section 2.3)								
ADy288	14.3±5.2	94.7±12.8	*193.3±17.5	89.5±30.5	0.9±0.8	0.4±1.0	1.0±0.1	100%
ADy416	15.3±5.1	90.2±12.0	*167.6±12.7	89.1±29.1	0.9±0.7	0.5±0.9	1.1±0.2	100%
ADy608	14.8±4.8	89.0±18.7	*192.8±16.2	91.5±31.2	1.1±0.7	0.4±0.9	1.1±0.2	100%
ADs288	14.3±5.0	95.6±13.7	*195.2±18.2	88.7±28.9	1.0±0.8	0.4±1.0	1.1±0.1	100%
ADs416	14.8±4.8	91.3±13.4	*168.8±13.3	90.1±30.2	1.1±0.9	0.5±1.0	1.1±0.0	100%
ADs608	14.7±4.9	90.6±19.2	*194.2±17.7	91.2±29.3	0.9±0.6	0.4±1.1	1.1±0.1	100%
2. Default ROSCH (section 2.3)								
ADy288	8.8±1.0	* ∞	* ∞	* ∞	* ∞	* ∞	1.1±0.8	100%
ADy416	8.5±0.7	* ∞	* ∞	* ∞	* ∞	* ∞	1.3±0.9	100%
ADy608	8.5±0.8	* ∞	* ∞	* ∞	* ∞	* ∞	1.1±0.7	100%
ADs288	9.0±0.8	* ∞	* ∞	* ∞	* ∞	* ∞	1.2±1.0	100%
ADs416	8.4±1.0	* ∞	* ∞	* ∞	* ∞	* ∞	1.3±0.6	100%
ADs608	8.5±0.9	* ∞	* ∞	* ∞	* ∞	* ∞	1.5±0.8	100%
3. Just-In-Time (JIT) Priority Adjustment (section 2.4)								
ADy288	8.5±0.9	94.6±13.4	*194.7±16.3	43.5±10.2	1.0±0.7	0.6±1.1	1.2±0.4	100%
ADy416	8.4±1.0	91.7±11.2	*166.8±11.4	45.3±11.3	0.8±1.0	0.6±1.1	1.0±0.4	100%
ADy608	8.7±0.7	88.9±17.6	*190.9±17.9	47.2±9.9	1.1±0.6	0.5±0.9	1.0±0.5	100%
ADs288	8.9±0.9	96.1±12.7	*195.9±17.3	46.9±12.1	0.9±0.9	0.6±1.0	1.0±0.4	100%
ADs416	9.0±0.7	92.8±11.4	*169.7±13.3	48.1±11.9	1.0±0.5	0.5±1.1	1.3±0.4	100%
ADs608	8.7±0.9	91.2±20.3	*194.8±16.9	44.7±10.9	0.8±1.0	0.4±1.2	1.2±0.2	100%
4. JIT Adjustment + Migration to Accelerators (section 2.5)								
ADy288	8.7±0.8	123.8±18.5	*225.6±5.0	43.0±9.9	0.9±1.0	0.5±0.9	1.0±0.6	100%
ADy416	8.8±1.1	128.7±12.1	*177.6±3.3	46.7±10.5	1.0±0.8	0.5±1.0	1.2±0.3	100%
ADy608	9.1±0.9	144.3±8.1	*171.8±3.0	48.5±9.8	1.0±0.8	0.6±1.1	1.2±0.3	100%
ADs288	9.0±0.8	125.6±17.1	*225.6±6.3	47.3±11.3	1.1±0.7	0.4±1.1	1.3±0.2	100%
ADs416	8.8±1.1	130.5±13.2	*180.1±4.7	47.6±9.8	0.9±0.9	0.7±1.2	1.5±0.2	100%
ADs608	8.6±0.9	147.2±9.2	*174.3±4.5	47.6±10.4	0.9±0.9	0.6±1.2	1.2±0.5	100%
5. JIT Adjustment + Migration to Accelerators + Hardware-Aware Model Customization (section 2.6)								
ADy288	8.4±1.2	*89.0±15.3	95.6±5.1	46.3±9.8	0.9±0.9	0.7±0.9	1.0±0.4	0%
ADy416	9.0±0.9	72.0±9.0	88.1±4.3	44.9±10.7	1.0±0.8	0.6±0.9	1.3±0.2	0%
ADy608	8.8±1.2	80.8±10.6	*98.1±5.0	46.4±11.0	1.0±0.7	0.4±1.1	1.1±0.3	0%
ADs288	9.0±1.1	*92.0±14.3	96.4±5.4	45.8±10.1	1.1±0.7	0.4±1.1	1.2±0.6	0%
ADs416	8.9±0.8	74.2±9.3	90.0±4.2	47.2±10.0	1.0±0.8	0.5±0.9	1.2±0.2	0%
ADs608	8.8±1.0	83.7±9.7	*100.1±4.4	46.8±9.9	0.9±0.7	0.7±0.8	1.3±0.8	0%
6. JIT Adjustment + Migration to Accelerators + Hardware-Aware Model Customization + Iterative Co-run Aware Scheduling (section 2.7)								
ADy288	8.0±0.6	*90.2±16.1	94.7±5.5	47.7±10.0	1.0±0.7	0.5±1.1	1.3±0.2	0%
ADy416	8.5±0.5	72.6±10.1	87.8±4.5	43.2±9.1	0.9±0.6	0.6±0.8	1.0±0.1	0%
ADy608	8.7±0.4	82.1±9.9	*96.9±4.8	44.9±11.9	1.0±0.6	0.5±0.9	1.2±0.2	0%
ADs288	8.4±0.5	*93.2±14.2	97.0±5.2	46.4±8.9	0.9±0.7	0.4±1.1	1.0±0.3	0%
ADs416	8.3±0.5	73.8±9.9	91.3±4.7	47.3±10.2	1.0±0.8	0.6±0.9	1.1±0.8	0%
ADs608	8.6±0.4	84.2±10.0	*100.1±3.2	46.5±9.7	0.9±0.7	0.5±0.8	1.1±0.5	0%

Table 2.4: Power consumption for the ADApp applications on Jetson AGX Xavier by INA3221 power monitors (Xav21) (unit: milliWatts)

section 2.4				section 2.4 + section 2.5				section 2.4 + section 2.5 + section 2.6			
Application	GPU	DLA	Total System	Application	GPU	DLA	Total System	Application	GPU	DLA	Total System
ADy288	18,684	0	33,029	ADy288	18,378	1,683	32,919	ADy288	13,948	4,441	31,750
ADy416	18,401	0	30,266	ADy416	19,740	1,834	34,729	ADy416	14,237	4,590	33,410
ADy608	20,675	0	33,025	ADy608	20,343	1,987	35,064	ADy608	13,936	4,437	33,303
ADs288	18,323	0	32,013	ADy288	19,012	1,645	33,142	ADy288	14,589	4,358	32,019
ADs416	18,756	0	31,415	ADy416	20,032	1,785	35,165	ADy416	14,357	4,640	34,774
ADs608	20,876	0	33,048	ADy608	20,132	2,038	36,043	ADy608	14,128	4,549	34,237

Table 2.5: Accuracy before and after the hardware-aware model customization

Models	Original	Customized
Yolov3-288	77.58%	76.73%
Yolov3-416	80.77%	79.75%
Yolov3-608	78.37%	78.06%
Yolov3-SPP-288	78.58%	76.89%
Yolov3-SPP-416	81.52%	81.15%
Yolov3-SPP-608	78.05%	80.07%

CHAPTER

3

DECENTRALIZED APPLICATION-LEVEL ADAPTIVE SCHEDULING FOR MULTI-INSTANCE DNNs ON OPEN MOBILE DEVICES

3.1 Introduction

Deep Neural Networks (DNN) have attained remarkable success in various tasks. Recent years have witnessed increasing adoption of DNNs in mobile devices, thanks to the advancement in DNN compression (HMD15; NSL⁺21; NML⁺20), the increasing concerns on privacy, and the demands for real-time responses.

As more apps start to make use of AI, multiple DNN-equipped apps may run on a mobile device at the same time. For example, while a user is using her smartphone to examine some surveillance videos through a DNN-based object detection module, she may be speaking to the DNN-powered personal assistance app on her phone to take notes, while her social media app may be running some DNN-based recommendation algorithm in the background. We call such a co-run scenario *multi-instance DNN* executions.

Scheduling is important for multi-instance DNN executions, especially on resource-constrained systems. This dissertation particularly focuses on the *spatial* aspect of scheduling, which determines the placement of a DNN-based app on heterogeneous hardware units during each inference. It is critical for the computing efficiency of DNNs. On one hand, DNNs are computationally demanding, and their performance is heavily influenced by the type, configuration, and availability of the underlying computing resources.

On the other hand, modern mobile devices (e.g., smartphones, tablets) are commonly equipped with heterogeneous computing units. On a Samsung Galaxy S21, for instance, there is one big "primary" CPU core (ARM Cortex-X1), three medium-sized "performance" CPU cores (Cortex-A78), four small "efficiency" CPU cores (Cortex-A55), one Adreno 660 GPU, and other accelerators. As a result, the speed and power consumption of a DNN running on the different computing units in a mobile device may differ as much as several times as illustrated in Figure 3.1. Multi-instance DNN executions further complicate the scheduling of DNNs to the best computing units, due to the contentions for computing resources by other co-running DNNs.

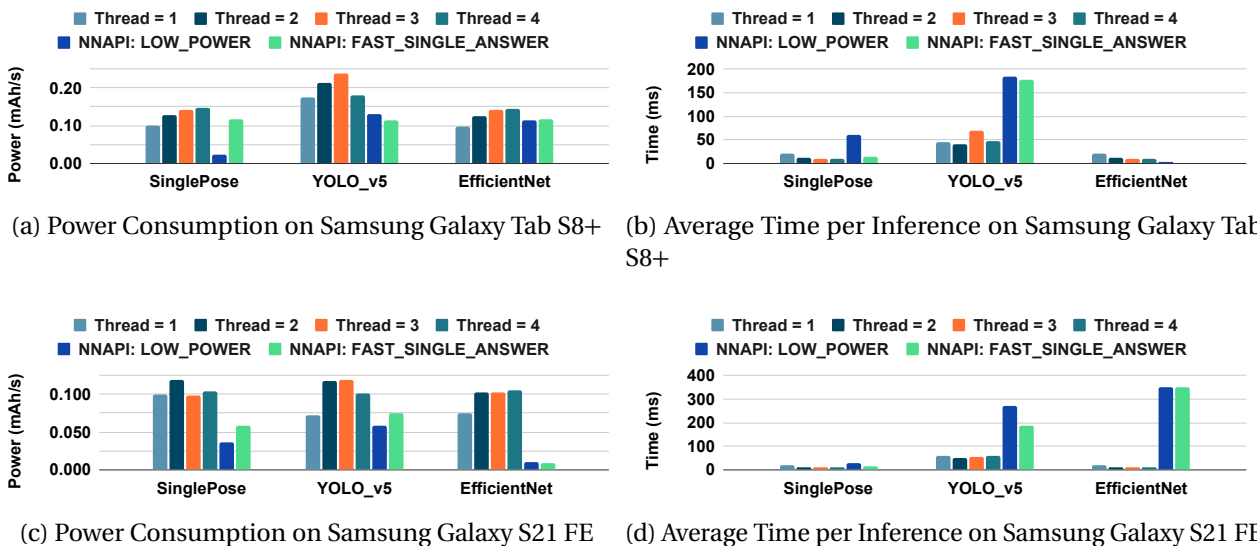


Figure 3.1: **Standalone Profiling of Three DNN Networks.** We profile the average power and inference time for different DNNs. They have their own best option for delegation. Thread = i means the model is executed on the CPU with i threads. NNAPI *low power* and *fast single answer* are two options that consider using GPU and accelerators.

The objective of this work is to address such spatial scheduling problems on *open* mobile devices. Here, *open* mobile devices refer to mobile devices on which users can install or uninstall

arbitrary apps anytime. In contrast, some devices (e.g., an autonomous driving system) are *closed*, where the applications to install and run are predetermined. The problem of multi-instance DNN scheduling also exists on *closed* devices, and has been explored in some previous studies (YWS⁺22; FZZ18; BL20; JLK⁺22). But those studies assume that the set of co-running apps are known beforehand and their schedules are fully controllable by a central agent (e.g., OS), which is not the case for *open* devices. Their solutions hence cannot apply to the *open* devices. To the best of our knowledge, no prior solutions have been proposed for multi-instance DNNs scheduling on open mobile systems.

This work proposes the first-known *decentralized* application-level adaptive spatial scheduler for multi-instance DNNs on open mobile devices. Being decentralized means that the spatial scheduling decisions are made by each application rather than by a centralized agent (e.g., OS). This distinctive design brings several benefits over centralized schemes: It is easy to adopt without the need for OS modifications; it requires no OS admin privileges; it preserves privacy and avoids inter-app communications or app-OS special communications and the associated overhead.

Specifically, the spatial scheduling considered in this work includes the decisions on running a DNN on GPU or on CPU and if on CPU how many CPU threads to use. We intentionally leave the temporal aspect of scheduling (i.e., at what time an app runs or gets evicted) and priority management as they are, because these tasks are what the OS is already taking care of. We meanwhile ensure that the spatial scheduling method can automatically adapt the scheduling decisions to the temporal scheduling by the underlying OS.

This design makes the solution easy to adopt (as no OS modifications are necessary), applicable across systems, and workable regardless of what the other co-running apps are and what scheduling policies they follow. It also retains the fairness guarantees and starvation-avoidance provided by the underlying OS.

Our solution achieves these properties by leveraging the adaptivity of Deep Reinforcement Learning (DRL) (MKS⁺15; VHGS16; PMJH18). We develop a DRL-based scheduling library. It is decentralized, working at the application level. Any app may call the library to dynamically determine, for the next DNN inference, whether CPU or accelerators is to be used, what modes (e.g., performance or power-efficiency modes) to use, and if CPU, what is the best number of threads to launch. It requires no direct knowledge about other apps. It gives recommendations based on the current state of the executing environment and the estimated rewards this application is expected to obtain for each of the possible schedules—produced by a model learned through the self semi-supervised approach of DRL.

By drawing on previous theoretical results on the Nash equilibrium of RL, we provide discussions on the convergence of the scheduling algorithm and the empirical evidences.

In a set of co-run scenarios formed by the subsets of nine DNNs, the proposed solution improves the average latency by as much as $4\times$, and saves the average energy consumption by as much as $3\times$ compared to Android NNAPI, the official Android tool that automatically selects the computing units to use for running a DNN. Experiments on a smartphone (Samsung Galaxy S21) and a tablet (Samsung Tab S8+) show that the benefits are consistently significant. Further experiments show that the benefits remain even if those DNNs co-run with uncontrolled apps (i.e., apps that do not employ the proposed scheduling algorithm). The scheduling algorithm converges quickly (within seconds) and adapts to the running environments automatically, making it an immediately adoptable solution across mobile systems.

Overall this work makes the following main contributions:

- To our best knowledge, the solution in this work is the first decentralized application-level adaptive scheduling for multi-instance DNNs on open mobile devices.
- This work uncovers a set of novel insights: (i) It is possible for a scheduler to work effectively without direct knowledge of other apps in multi-instance DNN scheduling; (ii) the DRL-based scheduling is effective in adapting to the factors in the execution environment (OS, other apps, priorities, etc.); (iii) the decentralized scheduling algorithm is quick in converging to a balance point.
- This work empirically evaluates the efficacy of the proposed solution, showing that it consistently gives significant speedups and energy savings across DNN workloads, hardware configurations, and running scenarios (with or without uncontrolled apps, various background/foreground combinations).

3.2 Background

Reinforcement Learning and Deep-Q-Network Reinforcement learning (RL) (PMJH18) is a machine learning approach in which an agent learns from environmental feedback and a series of decisions to maximize the total cumulative rewards to stimulate better decision-making. Q-learning (WD92) is a type of RL algorithm that seeks an offline policy to maximize the expected total rewards across all steps. "Q" refers to the policy function $Q(S, A)$ that inputs a state and action pair and outputs a Q-value. Two typical Q-learning schemes are Q-table and Deep-Q-network (DQN) (MKS⁺15). DQN improves the limited representation of the Q-table.

Q-table stores the state and action pairs with the estimated Q-value. The Q-value of each step can be obtained by table lookup. However, this approach only works when states and actions are discrete values and the sets are small. In contrast, DQN replaces the Q-table with a

neural network (NN) as a function approximator. Now, the sets of states and actions can be large, and the state space can be continuous. In the training phase, the loss function minimizes the squared error between the target Q-value and the predicted Q-value given by the NN. So the loss function of NN can thus be formulated as follows: $L = (Q(S, A) - (R + \gamma * \max(Q(S', A'))))^2$, where R is the immediate reward for taking action A in state S , γ is the discount factor (a value between 0 and 1 that determines the importance of future rewards), and S' and A' are the next state and the possible actions in that state, respectively.

Nash Equilibrium The Nash equilibrium is a concept in game theory that describes the optimal behavior of players in a game. It is a stable, self-enforcing, and Pareto optimal solution, where each player has chosen an optimal action, given the other players' actions. There are several methods for finding the Nash equilibrium of a game, including using best response functions and mixed strategies. The Nash equilibrium is a helpful tool for analyzing strategic interactions and predicting players' behavior in a game.

One way to find the Nash equilibrium is to use the best response function, which maps each player's action to the action that maximizes their reward, given the actions of the other players. The Nash equilibrium is then the set of actions where each player's action is the best response to the actions of the other players. Another approach is to use a mixed strategy, where players randomly choose their action with a certain probability. The Nash equilibrium is then the set of possibilities where each player's mixed strategy is the best response to the combined strategies of the other players.

3.3 Problem Statement and Research Questions

This section first provides a definition of the focused scheduling problem, and then lists the important research questions.

3.3.1 Problem Definition

Given: A set of apps A that may execute on a device V , and some of them may run at the same time. $A = A_c \cup A_u$, where, each app in A_c contains some DNNs and employs a policy P to decide the execution configuration of each of its DNNs, while the apps in A_u do not follow policy P . For each DNN, there are K possible configurations which affect the usage of CPUs and GPUs of the DNN differently.

Objective: Finding P such that the following is minimized:

$$\sum_{i \in A_c(DNN)} \sum_j L_{i,j} * W_{i,j}$$

where, $L_{i,j}$ and $W_{i,j}$ are respectively the latency and power consumption of the j^{th} inference of DNN_i , $A_c(DNN)$ is the DNNs in the apps in A_c . We use the product of latency and power to capture the common interest in both speed and energy usage on mobile devices.

Constraints: (1) The scheduling policy P in an app cannot access the information of another app (due to the isolation enforced by mobile systems); (2) each app’s priority and temporal scheduling are controlled by the underlying OS.

3.3.2 Design Considerations and Principles

The main aspect of scheduling focused in this work is the execution configuration that affects the usage of computing units—which is essential for the performance and power consumption of DNN. The relevant factors include some that are controllable at the application level, and some at the OS level. For a DNN written in TFLite (AAB⁺15) (a popular development framework for mobile AI), for instance, the application can explicitly specify whether the DNN should run on CPU or accelerators (e.g., GPU), and the number of CPU threads to use. It may also call APIs in NNAPI (Dev23) (the Android official library for running DNNs) with either a performance mode or an efficiency mode; the APIs will automatically determine the CPUs or accelerators to be used for the DNN. We uniformly regard such configurations as application-level controllable configurations, which specifically include the explicit specification of computing units and CPU thread number and the calls to other relevant libraries.

When a DNN runs with a certain configuration, the OS may exert further influence on the usage of the computing units. For instance, if the application-level control sets the DNN to run on a CPU with 2 CPU threads, the OS will ultimately determine which two CPU cores the threads will run on. If they are small cores, the speed may be much lower than on medium cores.

We design our solution with the following principles:

(1) The scheduling should be decentralized, functioning inside each application. This is because most of the relevant factors are inside the app, and application-level solutions are easier to adopt as they do not need changes to the underlying OS.

(2) The scheduling should be able to adapt to the influence of the underlying OS and other environmental factors (e.g., priorities of apps, background/foreground differences).

(3) The scheduler should work efficiently and adapt to changes in the system agilely.

3.3.3 Research Questions

Creating a decentralized application-level scheduler on open mobile systems has many differences from centralized scheduling on closed systems and hence raises many new questions. We summarize them into the following six open research questions (RQ). They are gradually addressed in the rest of this dissertation.

RQ1: How can the scheduler work effectively without direct knowledge of other apps?

The apps on a smartphone may be developed by many different authors. For security and privacy, open mobile systems typically impose strong isolations among apps. One app cannot access the direct info of another app. How can the scheduler work well under such a constraint?

RQ2: How can the solution deal with the effects of the scheduler in the underlying OS?

OS influences the execution of the Apps: Ultimately, it is the OS that allocates computing units and other resources to each App and determines when an app runs and its priority level. The OS schedulers differ from one version of OS to another. We avoid demanding changes to the underlying OS for easy adoption of our solution. The user-level scheduling hence must be made adaptive to OS.

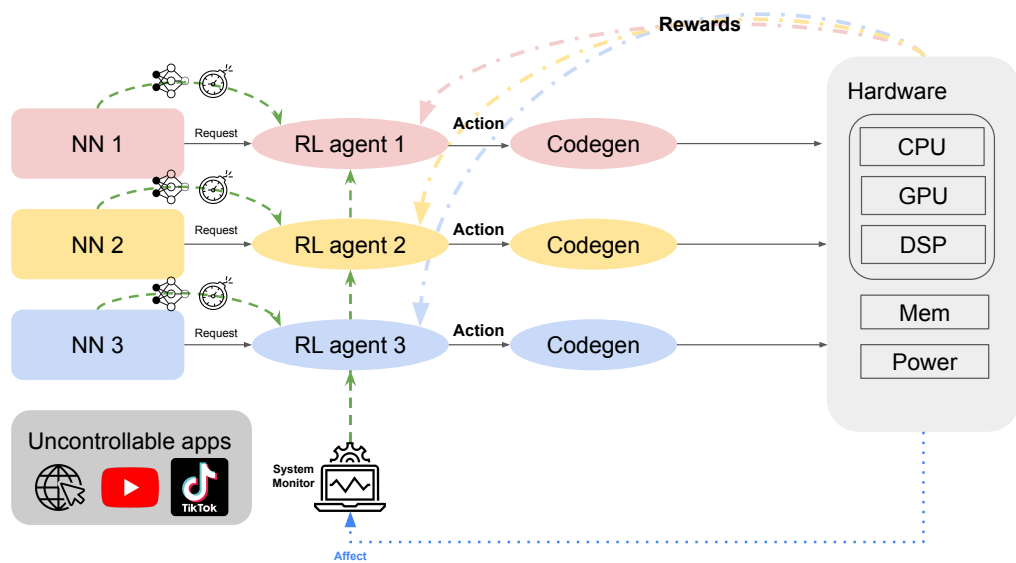


Figure 3.2: **The Structure of Decentralized DQN Scheduler.** The figure shows three controllable DNN applications that have their RL agent that selects actions (different code generation of the model). All of them collect system status as their RL agent input state. Also, we include the co-running uncontrollable apps that do not contain RL agents inside.

RQ3: Can decentralized scheduling converge to a good result?

On an open mobile system, apps come and go, and the workloads on the system may vary continuously. Without the knowledge of other co-running apps, can decentralized scheduling converge to good results?

RQ4: How fast can the decentralized scheduling learn and adapt?

Machine learning-based decision models usually take time to learn, but the dynamic nature of mobile systems demands fast responses. Can decentralized scheduling meet the speed needs?

RQ5: Can the solution work if there are uncontrollable apps?

In real mobile usage scenarios, not all Apps will adopt the same scheduling policy. The workload from uncontrollable apps can be unpredictable. Can the proposed solution still function well in the presence of such uncontrollable apps?

3.4 Decentralized DQN Scheduler

In this section, we introduce the design of the decentralized scheduler, which also answers RQ1 and RQ2.

The design is based on deep reinforcement learning (DQN). As Section 3.2 mentions, DQN is a deep reinforcement-learning method. As a semi-supervised method, it requires no manual labels, but actively explores the environment, learns the relations between actions and rewards automatically, and uses the learned model to predict the next suitable action. This nature makes it a good fit for the dynamic environment in our problem. In contrast, a DNN-based approach would require offline labels of many training cases and be slow in adapting to dynamic changes.

The DQN-based RL agent is also light-weighted and converges fast. For the storage overhead, the total extra memory footprint needed less than 250KB. For the computation overhead, the inference time of the network only takes less than 1ms and one training process only takes about 1ms to 2ms when three DQN agents co-running. In general real-time constraints are 50ms for each inference, and those extra computations overhead in each model can be neglected.

Figure 3.2 illustrates the role of the scheduler in a multi-instance scenario. Other than the uncontrolled apps, each DNN-based app contains a DQN agent for configurations. They do not have access to other apps but can obtain the state of the resource utilization of the whole system.

We define the learning procedure of our DQN-based agents with *States*, *Rewards*, and *Actions* in the multi-DNN execution environment as follows:

States We use the following variable to capture the static and variance environment information. For static features, we use the number of convolution layers, the number of fully-connected

Algorithm 3: DQN Algorithm for each Apps

Input: Environment E ; Replay Memory M ; Exploration Ratio ϵ ; The parameters of Policy Network θ and Target Network θ^- ; Discount Factor γ ; Batch Size B ; Update Steps C ; Huber loss function L

Output: $Q(s, a; \theta)$

Initialize: Take observation from E and generate current state s

```
1 while Inference start do
2   if rand() <  $\epsilon$  then
3     | Select action  $a$  randomly
4   else
5     | Select action  $a \leftarrow \operatorname{argmax}_a Q(s, a; \theta)$ 
6   Run inference on a target defined by action  $a$ 
7   When Inference ends, Calculate reward  $r$ 
8   Observe from  $E$  and generate next state  $s'$ 
9   Store transition  $(s, a, s', r)$  to  $M$ 
10  if  $M.size() > B$  then
11    | Sample a mini-batch  $N$  from  $M$ 
12    for each transition  $(s_j, a_j, s'_j, r_j)$  in  $N$  do
13      |  $y_j = r_j + \gamma \max_{a'} Q(s_j, a'; \theta^-)$ 
14      | Calculate Loss  $l_j = L(y_j, Q(s_j, a_j; \theta))$ 
15    | Batch Update  $\theta$  using SGD algorithm by loss vector  $l$ 
16   $s \leftarrow s'$ 
17   $i \leftarrow i + 1$ 
18  if  $i \bmod C == 0$  then
19    |  $\theta \leftarrow \theta^-$ 
-
```

layers, and the MAC operations of DNN models. For environment variance information, we use the CPU utilization, GPU utilization, and memory usage of co-run apps.

Rewards The reward function R is composed of three important metrics: latency, power consumption, and deadline. It is defined as follows:

$$R(L_i^{s,a}, W_i^a, d_i) = \begin{cases} -1000 & , \text{ if } L_i^{s,a} > d_i \\ -L_i^{s,a} \times Power_i^a & , \text{ otherwise} \end{cases}$$

where, $L_i^{s,a}$ represents the latency of the inference of DNN_i with action a on state s , W_i^a is the average power of the inference of DNN_i with action a , and d_i is the deadline for the inference of DNN_i .

Because (*latency* \times *power*) can be regarded as the energy consumption estimation, we call it **Energy Factor** in our dissertation. Also, it is used as one of the metrics in our evaluation (Section 3.6). The multiplication of power and latency gives a linear relationship, with no

skew towards either factor. For instance, if the latency doubles, the entire reward function doubles, and the same applies to power consumption. The simple production form of the reward function avoids additional hyper-parameters. A large negative reward is used when a deadline is passed to discourage missing deadlines.

Actions The action space involves the configurations that can affect the usage of computing resources on the hardware. In our study, we include six configurations as detailed in Section 3.6.1.

Neural Networks DQN includes a *policy neural network* and a *target neural network* inside, which learn about the relations between states, actions and rewards. We want to make the networks as simple as possible to control the runtime overhead. So we adopt a model that only maintains two fully-connected (FC) layers as our policy and target network. The first FC layer input channel is 8, and the output channel is 100. The second FC layer input channel is 100, and the output channel is 6. For the input, it is the vector of the state. Its dimension is 8, which contains four CPU and GPU usage pairs. The output is a vector that represents the q-values of six actions.

The DQN agents go through an exploration and learning stage until they reach convergence. As listed in Algorithm 3, at the start of each episode, each agent selects the action with the maximum q-value generated by the policy network with parameters θ and current state s but has some exploration rate ϵ to select action randomly. Here, one episode is one inference of the DNN model. After making the inference with the selected action, the power consumption and latency are collected to compute the reward r . Then, the agent observes the environment and generates the updated state s' . One transition (s, a, s', r) is then pushed into the replay memory M . The training process starts when the replay memory M has enough data. It goes as follows. It first samples a mini-batch N from M . Then, it calls the *target network* to generate the expected q-value y for each transition. After that, it uses the Huber Loss function (Fri01) to calculate the loss value between the expected q-value and the current q-value output by the *policy network*. The final step of the training process is to update the parameters θ based on the loss value. In every C episode, the parameters θ of the *policy network* are copied to the *target network* as its new parameters. In the targeted co-run scenario, each controllable App is equipped with a DQN agent that is trained for scheduling its DNN inferences.

3.5 Convergence Discussion

This section discusses the convergence of the DQN-based scheduling algorithm (RQ3). Prior works (Bow00) generalize the multi-agent Q-learning method as a general-sum stochastic game and prove all reward functions in each agent are guaranteed to converge. Specifically, Hu and Wellman (HW⁺98) present an algorithm to solve the general-sum stochastic games, and

Bowling (Bow00) strengthens the proof with further assumptions.

Their convergence theorem has four necessary assumptions, two of which are about exploration and the decay of the learning rate, and are similar to those used in the Deep-Q-Learning algorithm. It is assumed that they have been met. The remaining two assumptions (HW⁺98; Bow00) are as follows:

Assumption .1 *A Nash equilibrium $(\pi_*^1(s), \pi_*^2(s))$ for any stochastic game (Q_n^1, Q_n^2) satisfies one of the following properties:*

- 1. The equilibrium is a global optimal.*
- 2. The equilibrium receives a higher payoff if the other agent deviates from the equilibrium strategy.*

Assumption .2 *The Nash equilibrium of all stochastic games $Q_n(s)$, as well as $Q_*(s)$ must satisfy property 1 in Assumption 1 **or** the Nash equilibrium of all stochastic games, $Q_n(s)$, as well as $Q_*(s)$ must satisfy property 2 of Assumption 1*

Assumption 1 includes a property that states that there exists a set of strategies for the agents, where each agent individually obtains the highest possible payoff. This also guarantees that this set of strategies forms an equilibrium since no agents would gain from deviating from their chosen strategy. Assumption 2 includes another property in which the game's Nash equilibrium is a "saddle point." This implies that if an agent deviates from the equilibrium, the agent would not gain, but other agents would, which makes no agent want to deviate from the equilibrium.

Finding a globally optimal solution for the multi-agent problem is known to be NP-hard (CD06). However, by reaching Nash equilibrium during convergence, RL can ensure that each agent adheres to a strategy that gives a good payoff to both itself and the other agents. Reflected in our scheduling context, it means that all controllable Apps may adhere to a strategy that helps meet their deadlines while minimizing energy consumption. The achievement of Nash equilibrium eliminates fairness concerns among controllable Apps, as they all reach a stable state where each maximizes its benefits within the given constraints.

Based on prior studies (Bow00; HW⁺98), it has been demonstrated that a zero-sum stochastic game converges. Our scheduling problem for multi-DNN applications in this dissertation bears a resemblance to a zero-sum stochastic game. In our case, each agent corresponds to our DQN agent for each controllable DNN application. All DQN agents involved compete for limited resources, such as CPU and GPU, with a maximum utilization boundary. While our problem may not strictly adhere to all the definitions of a stochastic game, we empirically demonstrate its convergence under our circumstances in Section 3.6.

3.6 Evaluation

This section evaluates our decentralized application-level adaptive scheduler (called DQN for short in this section) by comparing it with three baseline scheduling methods that are designed for single DNN execution: two static scheduling settings used by Android Neural Networks API (NNAPI) (Dev23) (NNAPI `LOWER_POWER` that minimizes the power consumption for each DNN and NNAPI `FAST_SINGLE_ANSWER` that minimizes the inference latency for each DNN) and an offline profiling-based scheduling method (Best Standalone that based on offline profiling selects the best setting for each individual DNN among all delegate settings introduced in Section 3.6.1). This evaluation has three objectives as follows: 1) demonstrating that as the *first* decentralized DNN co-run scheduling method, DQN outperforms all baseline scheduling approaches that are designed for single DNN execution in multiple representative DNN co-run scenarios (Section 3.6.2); 2) verifying DQN’s convergence and fast converging speed, and studying the underlying reason why DQN outperforms baseline scheduling methods by a reward convergence analysis (Section 3.6.3); 3) proving DQN’s benefits remain even if the DNNs co-run with uncontrolled apps with both predictable and unpredictable workloads (Section 3.6.4).

3.6.1 Evaluation Methodology

Benchmarks. Table 3.1 characterizes the nine DNN models from various domains used in the evaluation¹. All models are in TensorFlow Lite (AAB⁺15) format. These DNN models form three groups: Image, Audio & Image, and Video & Image with three models in each group. Our evaluation runs models in each group simultaneously to simulate the real-world DNN co-run scenario. For example, Group 1 (G1) simulates an intelligent camera running varied AI capabilities simultaneously, pose detection (SinglePose), object detection (YOLO-v5), and image classification (EfficientNet). Other groups simulate more complex scenarios with audio and image or video and image co-processing. In addition, these DNNs have varied model sizes, standalone latency, and power consumption, representing three different cases: relatively balanced workloads, mild imbalanced workloads, and severe imbalanced workloads, respectively. Therefore, they show different behaviors in the evaluation. Please find more discussions in Section 3.6.2.

Software settings.

¹These DNNs are collected from Tensorflow Hub (tfh22) and GitHubs (Tan22).

Table 3.1: **Nine DNN Models Used in Our Evaluation.** They form three groups. DNN models in each group are executed simultaneously.

Group	Models	Sizes (KB)
G1: Image	SinglePose (VL21) (SP)	9,154
	YOLO-v5 (BWL21)	7,428
	EfficientNet (TL19) (ENet)	6,265
G2: Audio&Image	YamNet (KSSG20)	4,031
	MobileNetv1 (HZC ⁺ 17) (MNv1)	4,188
	WDSR (YSKL18)	1,252
G3: Video&Image	Movenet (kol)	24,440
	Esrgan (WYW ⁺ 18)	4,877
	MobileNetv2 (SHZ ⁺ 18) (MNv2)	13,666

Table 3.2: **Absolute Latency of DQN.** This table reports the absolute latency of DQN in Figure 3.3, Figure 4.6, and Figure 3.7.

	Figure 3.3		Figure 4.6		Figure 3.7	
(Unit: ms)	Tablet	Phone	Tablet	Phone	Tablet	Phone
G1: SP	8.5	10.7	35.0	32.5	32.1	34.4
G1: YOLOv5	331.1	397.1	591.9	318.7	542.4	446.6
G1: ENet	5.0	47.2	9.7	337.8	7.0	341.3
G2: YamNet	4.3	3.8	23.4	23.0	20.4	17.1
G2: MNv1	6.2	34.3	10.5	30.8	8.2	32.1
G2: WDSR	6.8	116.9	6.3	114.2	5.0	108.2
G3: Movenet	34.2	33.0	76.2	71.3	78.5	70.1
G3: Esrgan	62.3	61.5	64.6	75.2	94.4	69.8
G3: MNv2	40.2	25.2	80.2	42.5	17.1	35.4

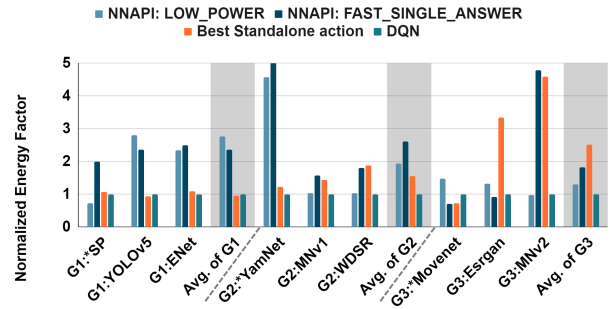
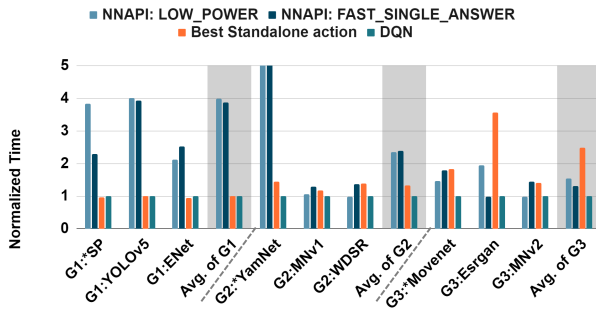
Our evaluation considers two co-run scenarios: *controllable DNN tasks co-run* and *uncontrollable tasks co-run*. *Controllable DNN tasks* refer to Apps that incorporate our RL agents, while *uncontrollable tasks* refer to Apps that do not involve our RL agent. *Uncontrollable tasks* may or may not use DNNs.

Controllable DNN Tasks Co-Run. We build an Android demo app with Java that can run each DNN individually. Users can control this demo app to start and stop DNN inference. This demo app relies on TensorFlow Lite (TFLite) (AAB⁺15) to run DNNs. Our evaluation employs multiple delegate settings in TFLite to run DNNs: using 1, 2, 3, or 4 CPU threads², respectively, and using two NNAPI (Dev23) modes, LOWER_POWER or FAST_SINGLE_ANSWER, respectively. Particularly, NNAPI is designed for accelerating TensorFlow Lite DNN execution on mobile devices with supported hardware accelerators including GPU, DSP, and NPU. It automatically

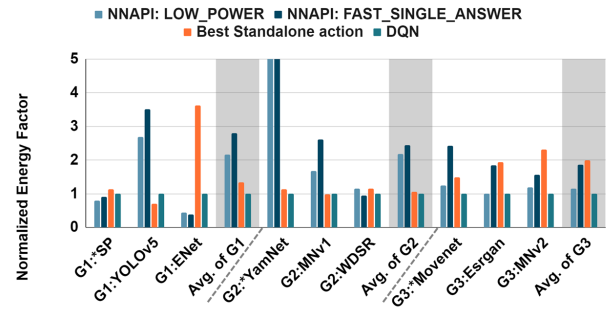
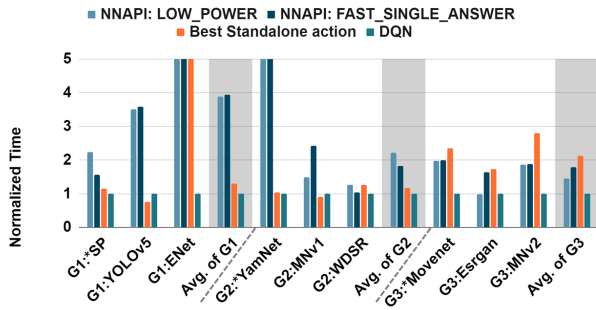
²The evaluated mobile chip has 8 CPU cores, but the Android OS only allows background Apps to access the 4 small CPU cores. Thus, we make it consistent throughout our evaluations: the foregrounds Apps access the 4 cores (prioritize to access the big core, medium core, then small cores), and the background Apps access the 4 small cores.

Table 3.3: **Absolute Energy Factor of DQN.** This table reports the absolute energy of DQN in Figure 3.3, Figure 4.6, Figure 3.7.

	Figure 3.3		Figure 4.6		Figure 3.7	
(Unit: Joule)	Tablet	Phone	Tablet	Phone	Tablet	Phone
G1: SP	18.9	15.9	13.48	18.1	25.2	21.8
G1: YOLOv5	950.3	446.3	1.6k	365.1	1.5k	509.8
G1: ENet	9.8	122.3	16.8	67.9	13.8	69.6
G2: YamNet	7.7	7.0	17.4	39.9	52.5	33.5
G2: MNv1	11.3	58.9	12.5	57.4	19.8	64.9
G2: WDSR	11.0	41.7	10.7	58.2	11.2	35.3
G3: Movenet	48.0	19.3	73.3	26.6	59.3	29.5
G3: Esrgan	33.2	14.5	36.0	20.4	23.4	16.4
G3: MNv2	27.7	32.8	11.8	38.2	60.8	29.6



(a) Normalized Average Inference Time on Samsung Galaxy Tab S8+ (b) Normalized Energy Factor on Samsung Galaxy Tab S8+ Galaxy Tab S8+



(c) Normalized Average Inference Time on Samsung Galaxy S21 FE (d) Normalized Energy Factor on Samsung Galaxy S21 FE Galaxy S21 FE

Figure 3.3: **Overall Comparison between DQN and Three Baselines on Three DNN Groups.** Compare (a) normalized average inference time and (b) rewards on Samsung Galaxy Tab S8+ and the same metrics (c) and (d) on Samsung Galaxy S21 FE when each group co-running three DNN apps (* denotes this DNN runs in the foreground and the others run in the background).

partitions a DNN model, maps each partition to a processor, and calls corresponding kernel codes for that processor. Thus, we can treat it as *static offline scheduling* for each individual DNN.

Our evaluation particularly employs two NNAPI modes as the baseline, LOWER_POWER which minimizes the power usage, and FAST_SINGLE_ANSWER which minimizes the inference latency. Besides them, our evaluation also employs an offline profiling-based scheduling method as a baseline: Best Standalone that selects the best setting (i.e., with the best energy factor defined as $power_consumption \times latency$) for each individual DNN among all delegate settings (including using 1 to 4 CPU threads and two NNAPI modes) based on offline profiling results³.

Uncontrollable Tasks Co-Run. We select two widely used real-world applications TikTok and Web Browser to experiment with two popular user behaviors, watching social media video and browsing web pages. Here we select the default web browser in Android, Google Chrome as our target.

Evaluation platforms. DQN is evaluated on two edge devices: (1) Samsung Galaxy S21 FE 5G mobile phone, equipped with Android 12 OS, and Qualcomm SM8350 Snapdragon 888 5G SoC with Octa-core CPU (1x2.84 GHz Cortex-X1 & 3x2.42 GHz Cortex-A78 & 4x1.80 GHz Cortex-A55), Adreno 660 GPU (Version 1), and Hexagon 780 DSP. Its storage capacity is 128GB with 6GB RAM and its voltage is 4.3V. (2) Samsung Tab S8+ tablet, equipped with Android 12 OS as well, and Qualcomm SM8450 Snapdragon 8 Gen 1 SoC with Octa-core (1x3.00 GHz Cortex-X2 & 3x2.50 GHz Cortex-A710 & 4x1.80 GHz Cortex-A510), Adreno 730 GPU, and Hexagon DSP. Its storage capacity is 128GB with 8GB RAM and its voltage is 4.1V.

3.6.2 Overall DQN Scheduling Performance

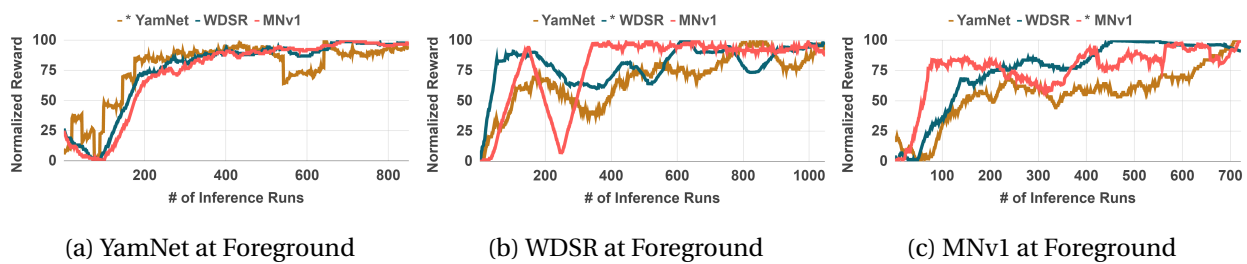


Figure 3.4: **Reward Convergence Trend for the Three Apps Co-Run in G2.** It reports DQN's normalized reward trend to prove DQN converges in different situations. It uses DNNs in G2 and places each DNN in the foreground.

³we profile the power values through the Android Developer API "dumpsys batterystats".

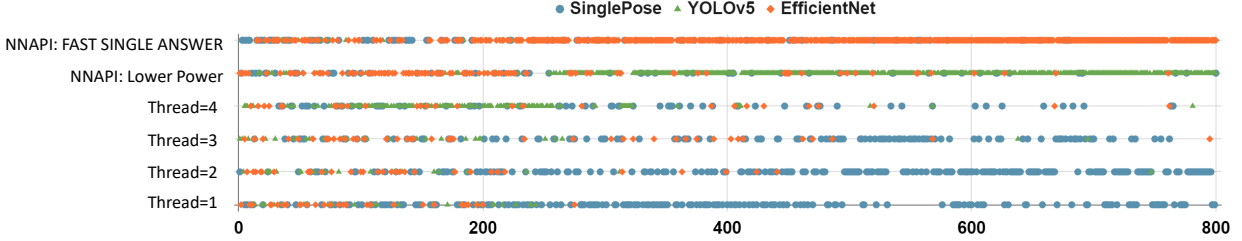


Figure 3.5: **Selected Action Convergence Trend.** This figure shows the action selection through runs for co-running results in G1 (all of them are in the background). The action selection will be converged into one or two options.

Table 3.4: **Selection Rates of Actions of Last 100 Runs of Figure 3.5.**

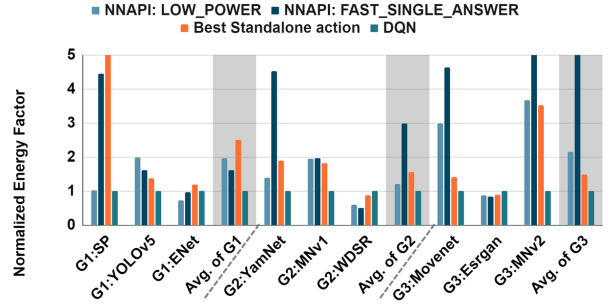
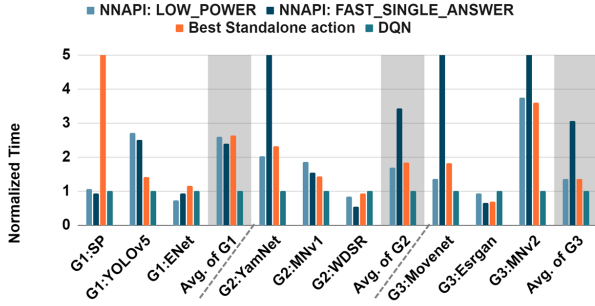
	Thread = 1	Thread = 2	Thread = 3	Thread = 4;	NNAPI: LOW_POWER	NNAPI: FAST_SINGLE_ANSWER
SinglePose	14%	75%	6%	1%	1%	3%
YOLO_v5	0%	1%	1%	0%	98%	0%
EfficientNet	0%	3%	0%	3%	0%	94%

This section evaluates DQN on the three groups of co-run DNNs in Table 3.1 by comparing it with the three scheduling baselines aforementioned: NNAPI LOWER_POWER, NNAPI FAST_SINGLE_ANSWER, and Best Standalone. Figure 3.3 shows the comparison results, in which the x-axis shows the three DNNs in each group and the average performance of each group. It is worth noting that the *star* (*) before the DNN name indicates that this DNN model is executed in the foreground (and two other DNNs in the same group are executed in the background) for this co-run⁴. We intentionally use this setting to simulate the real-world Apps co-run on the Android system (and bring the OS impact on the user-level scheduling into account).

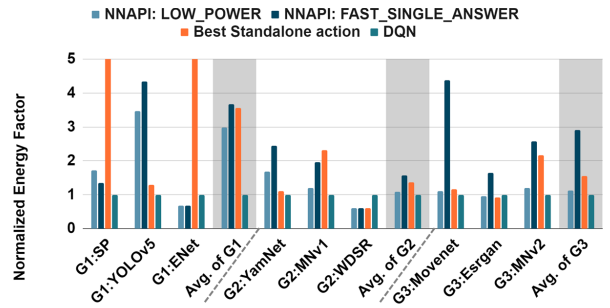
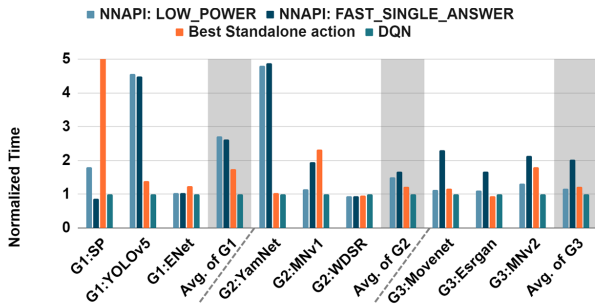
Figure 3.3 employs two metrics to compare our decentralized DQN system with three baselines: average inference latency (as shown in Figure 3.3a and Figure 3.3c) and energy factor (as shown in Figure 3.3b and Figure 3.3d). The energy factor is defined as $power_consumption \times latency$ for each inference, which is also used as our reward function in each DQN agent, the lower the better. To improve the readability, we normalize the results in Figure 3.3 by setting DQN performance as 1. Table 3.2 and 3.3 summarizes the absolute values for reference.

Figure 3.3 shows that for the average inference latency, our decentralized DQN-based scheduler achieves up to 4× speedup over two baselines of NNAPI (NNAPI LOWER_POWER and NNAPI FAST_SINGLE_ANSWER), and 2.7× speedup over Standalone Best action selection,

⁴Android OS grants foreground and background Apps different priorities/limitations, e.g., normally, background Apps have lower priority than foreground ones, and background Apps cannot access the big core of CPUs.



(a) Normalized Average Inference Time on Samsung Galaxy Tab S8+ (b) Normalized Energy Factor on Samsung Galaxy Tab S8+



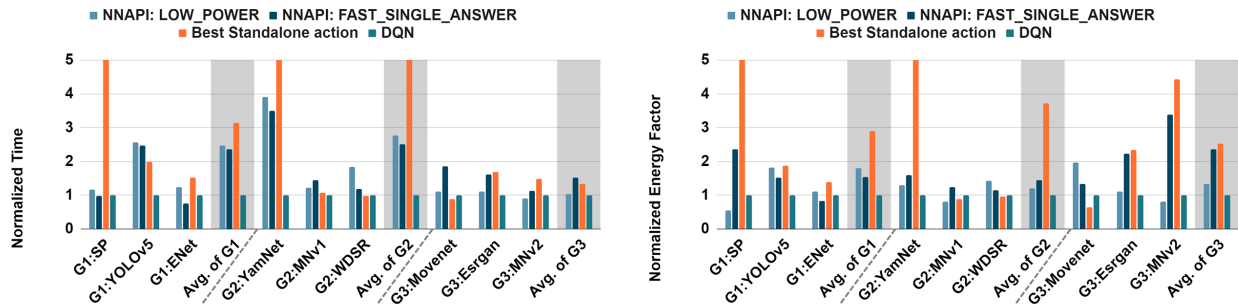
(c) Normalized Average Inference Time on Samsung Galaxy S21 FE (d) Normalized Energy Factor on Samsung Galaxy S21 FE

Figure 3.6: **Co-Run Three DNN Groups with Uncontrollable App TikTok.** Compare (a) normalized average inference time and (b) rewards on Samsung Galaxy Tab S8+ and the same metrics (c) and (d) on Samsung Galaxy S21 FE when each DNNs group co-running with uncontrollable app TikTok that plays video posts on the foreground.

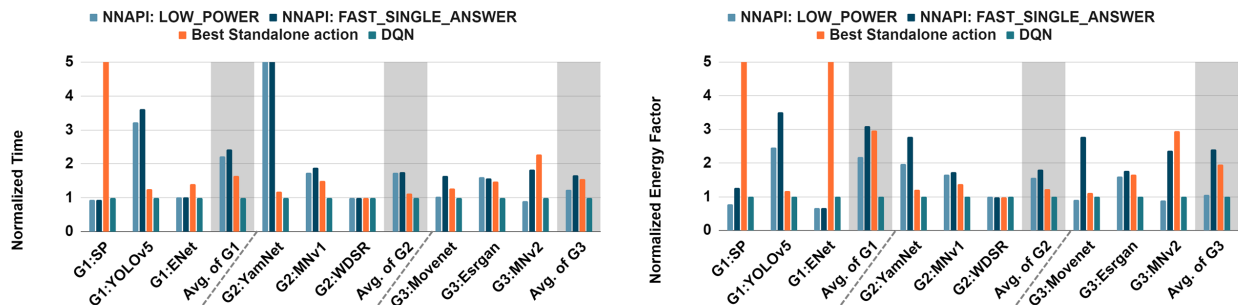
respectively, for average results of three co-running DNN groups (gray area). For the energy factor, our decentralized DQN-based scheduler achieves up to $3\times$ energy saving over NNAPI LOWER_POWER and NNAPI FAST_SINGLE_ANSWER, and $2.6\times$ energy saving over Standalone Best action selection, respectively, for average results of three co-run DNN groups. Comparing the DQN performance across three groups, we can see that as the workload imbalance increases (from G1 to G3), the benefit of DQN over Best Standalone grows while its benefit over both NNAPI schedulers drops. This is mainly because Best Standalone partitions each DNN workload into more processing units than NNAPI schedulers, so it reduces the resource competition caused by the workload imbalance. DQN has a similar effect. Moreover, although the average inference latency and energy factor vary for different groups (and each DNN model) under various settings, our decentralized DQN-based scheduler always performs better than baseline methods. These results prove that DQN is robust enough to deliver high-quality

scheduling results for various DNN applications and environment settings (e.g., varied foreground/background DNN settings, DNN structures/target domains, and executing devices). The following sections further verify this claim.

3.6.3 Reward Convergence Analysis



(a) Normalized Average Inference Time on Samsung Galaxy Tab S8+ (b) Normalized Energy Factor on Samsung Galaxy Tab S8+



(c) Normalized Average Inference Time on Samsung Galaxy S21 FE (d) Normalized Energy Factor on Samsung Galaxy S21 FE

Figure 3.7: **Co-Run Three DNN Groups with Uncontrollable App Web Browser (Google Chrome).** Compare (a) normalized average inference time and (b) rewards on Samsung Galaxy Tab S8+ and the same metrics (c) and (d) on Samsung Galaxy S21 FE when each DNNs group co-running with the uncontrollable app Google Chrome randomly browsing pages on the foreground.

This reward convergence analysis has three objectives: 1) to verify the rewards of multiple agents converging empirically, 2) to measure the DQN convergence speed⁵, and 3) to study

⁵Theoretical convergence proof only proves all DQN converges eventually without showing the convergence speed.

why DQN outperforms baseline scheduling by analyzing its trend of action selection.

We verify the convergence and measure the convergence speed by taking Group 2 (in Table 3.1) as an example due to the space limitation and other groups show similar trends. Figure 3.4 shows the convergence trends of DQN training on co-running three DNNs in Group 1 under three different settings that place YamNet, WDSR, or MNv1 at the foreground, respectively. The x-axis is the number of DQN inference runs (which is equivalent to the number of DNN inference runs), and the y-axis is the average action selection rewards of 100 times. Regardless of the environment settings, the DQN agents can reach convergence within 800-1000 inference runs. The scheduling time overhead for a single execution of the DNN App includes the forward- and backward-propagation phases of the DQN agent, which typically take 1.2 ms on average and only contribute to 0.5-5% of the overall DNN App execution time (which ranges 20-176ms). Additionally, the energy overhead amounts to approximately 0.5-2% of the DNN App execution. This convergence time is trivial compared to the time required to profile and configure each model manually.

We next study why the DQN agent can outperform the single DNN scheduling baselines by analyzing its trend of action selections. We take Group 1 (in Table 3.1) with SinglePose, YOLOv5, and EfficientNet as an example this time. All models run in the background, and the evaluation results are shown in Figure 3.5. In addition, Table 3.4 shows the percentage of action selection in the last one hundred runs to give an insight into action selection after the model is converged.

Figure 3.5 and Table 3.4 offer us two key insights in DQN: first, even though each DNN starts with multiple selections, their decision is converged into one or two actions, and second, a precise boundary exists between the actions only using CPU and those cooperating with GPU. The regular pattern of the action selection in the first insight implies that the DQN model converges at the end, empirically proving that the multi-agent DQN game converges (in another setting). The second insight tells us that DQN can effectively avoid computing resource contention. For example, when YOLOv5 and EfficientNet run on GPUs, DQN is able to schedule SinglePose on CPUs, thus preventing competition for the limited resource of accelerators; while other baseline scheduling methods fail to do this, resulting in GPU contention.

3.6.4 DQN Performance w/ Uncontrollable Apps

This section evaluates the performance of our decentralized DQN scheduler under uncontrollable application co-running situations. More specifically, we aim to 1) compare the performance between our decentralized DQN scheduler and those baselines under the popular real-world Apps co-running circumstance, and 2) illustrate our DQN solution has strong adap-

tivity when co-running with Apps with dynamically changing workloads.

Specifically, we simulate the real-world environment by simultaneously running DNNs and other apps with predictable workloads (i.e., repeatedly playing a TikTok video post), and unpredictable workloads (i.e., randomly browsing webpages in a web browser, Google Chrome). Both TikTok and Google Chrome run in the foreground, while DNNs (and their demo applications) run in the background. TikTok and Google Chrome require CPU and accelerators (e.g., GPUs), thus careful workload scheduling of DNNs is desired if we would like to achieve optimized system performance. To verify the predictability of TikTok and the unpredictability of Google Chrome, respectively, we use GPUWatch to monitor both applications’ execution and find that the major task, video processing in Tiktok requires a stable amount of GPU resources, while Google Chrome only consumes GPU resources when users touch or swipe across the screen, resulting in irregular GPU usage.

Figure 4.6 and Figure 3.7 compare DQN with all three baselines aforementioned, NNAPI LOWER_POWER, NNAPI FAST_SINGLE_ANSWER, and Best Standalone on two platforms under two uncontrollable cases (more predictable TickTok and more unpredictable Google Chrome), respectively. Our decentralized DQN-based approach outperforms all baselines for both cases in terms of both latency and energy for all three groups of DNNs. For the Tiktok case, DQN shows better performance because the DQN agent has more convincing history data that can predict more accurate action for the next step. The Google Chrome case empirically proves that our decentralized DQN-based approach is robust enough to handle DNNs co-run with an app that has unpredictable workloads.

Table 3.5: **Comparisons with simple Heuristic-based adaptation.** This table reports the average latency and energy factor for G2 in three co-run settings described in Figures 3.3, 4.6, and 3.7

Co-run Setting →		No other Apps		With Web Browser		With TikTok	
DNN Apps ↓	Metrics ↓	T&S (X=50)	DQN	T&S (X=50)	DQN	T&S (X=50)	DQN
YamNet	Time (ms)	3.690	4.250	127.120	20.482	53.400	23.050
	Energy Factor	0.461	0.502	15.890	3.392	4.429	2.707
SSD_MNv1	Time (ms)	8.260	6.170	191.740	8.248	60.670	30.860
	Energy Factor	1.161	0.734	24.351	1.284	8.037	3.890
WDSR	Time (ms)	126.290	6.750	56.320	5.042	142.000	114.220
	Energy Factor	15.701	0.714	8.599	0.729	18.034	3.948
Avg. Energy Factor		5.774	0.65	48.840	5.405	10.166	3.514

Compare with a Heuristic-Based Adaptive Method. To confirm that simple heuristic-based adaptation is insufficient for the scheduling problem, we implement a heuristic method called "trial and set" (T&S). In T&S, each action performs X (50 in our experiments) inference runs and selects the action with the minimum observed online energy factor in subsequent inferences. We compare it with our DQN results on G2 in each of three settings: three DNNs in G2 co-run (see Figure 3.3), G2 co-run with a predictable App (see Figure 4.6), and G2 co-run with an unpredictable App (see Figure 3.7). Each execution of the DNN-based app conducts 250 inferences in total. Table 3.5 shows the results. The result shows that the simple adaptation by T&S is insufficient for fitting the continuously changing execution environments. The schedules it picks cause 3–10 \times larger energy factors as well as frequently substantial slowdowns compared to the results by DQN.

3.7 Related Work

DNN workload under the stochastic runtime variance has been addressed in Autoscale (KW20). The authors propose a lightweight scaling engine for DNN inference on a cloud-edge environment. It applies an offline-trained Q-table that observes DNN characteristics and runtime variance as states and selects execution targets as action. It is, however, only for single DNN execution.

Multi-tenancy DNNs (YWS⁺22) have been an active research topic in recent years. NestDNN (FZZ18) proposes an efficient scheduler that works with different model pruning ratios. The scheduling decision is guided by the proposed minimum total cost and minmax cost. The solution enhances the multi-DNN inference accuracy and video frame processing rate while reducing energy consumption. NeuOS (BL20) proposes a layer-by-layer multi-DNN scheduler. At each layer boundary, the system will determine the power configuration for each DNN based on their deadlines. Band (JLK⁺22) presents a model analyzer and scheduler to organize a multi-DNN workload on a heterogeneous platform. The model analyzer partitions multiple DNN models into several subgraphs and dynamically designates them with an eligible execution target. The scheduling decision is based on subgraph execution latency estimation using their tensor size and FLOPS.

Those solutions are however all centralized approaches, assuming the set of DNNs is fixed and there is a central runtime scheduler managing all the instances, making them inapplicable to the multi-instance DNN scheduling on open mobile systems.

3.8 Conclusion

This dissertation proposes the first-known decentralized application-level adaptive scheduler for multi-instance DNNs on open mobile devices. It builds on DQN, a reinforcement learning algorithm that actively explores and learns the relations between the states, actions, and rewards in a dynamic environment. The exploration uncovers a set of insights. It shows that it is possible for a decentralized scheduler to work effectively without direct knowledge of other apps in multi-instance DNN scheduling. The DQN-based scheduling works well regardless of the differences among underlying systems, hardware, and execution settings. The algorithm is shown to converge quickly and effectively in improving co-run efficiency. As an application-level solution, it is ready to be immediately adopted across various mobile systems.

CHAPTER

4

COSMA: CONTENTION-CONSCIOUS CO-OPTIMIZATION OF DNNS AND SCHEDULING ON MOBILE SYSTEMS

4.1 Introduction

Mobile Artificial Intelligence (AI) is becoming increasingly important as Deep Neural Networks (DNN) is being quickly adopted in various applications running on mobile devices equipped with smartphones, robots, vehicles, and so on. The efficiency of mobile AI is crucial to user experience.

The environments of mobile AI have two prominent features. The first is that the co-running of applications is common. On a smartphone, while the user is watching videos on an AI-powered video player, a personal assistance app may be constantly trying to detect and recognize the user's demands in speech. On an automobile platform, while a navigation system is running, a drowsy detection module may be checking the driver's status so that it can alert the driver when necessary (RKA⁺19). Co-runs cause complicated interference among apps and hence substantial slowdowns (XBSV21; SCN⁺23; SXG⁺22).

The second feature, hardware heterogeneity, offers opportunities for mitigating the impact

of the contention. Mobile devices are known for the diverse computing units integrated on a single board. A typical modern smartphone contains multiple sizes of CPUs (big, medium, and small cores), GPUs, DSPs, NPUs, and so on. An AI app is often made executable on multiple types of computing units, which brings opportunities for the runtime to properly place those AI apps on the computing units to mitigate the co-run contentions and maximize the app’s performance. This method is called *DNN spatial scheduling* or *DNN scheduling* in short, which is what some recent studies have focused on (KW20; SCN⁺23; XYY⁺21).

Even though prior studies have shown some promise in DNN scheduling, they have all assumed that each DNN has a fixed structure.

In this work, we explore a new scheduling problem, *elastic DNNs optimization-scheduling problem (EDOSP)*. In EDOSP, each DNN may be configured dynamically and show different speed-accuracy tradeoffs. An example is *elastic multi-exit DNNs*, where the DNN is equipped with multiple exits, with each leading to a different computing demand and accuracy trade-off (HCL⁺18; KKKK19; LWW⁺21; BPX⁺20; YYX⁺19), as illustrated in Figure 4.1. Recent AI research (FZZ18; CYS⁺24; WLL⁺19; MHY⁺21; GZL⁺22) has shown that elastic DNNs can provide much better speed-accuracy quality, especially when the environment changes. However, none of the prior DNN scheduling studies have considered elastic DNNs.

Including dynamic optimization of DNNs with spatial scheduling significantly complicates contention-conscious DNN executions for three reasons. (i) There is an inter-dependence between DNN optimization and DNN scheduling. The appropriate placement of a DNN changes when the DNN is optimized differently, as that would change its latency, resource demands, and exerted resource contention; on the other hand, different placements of a DNN also affect the appropriate optimizations for the DNN because the placement changes the speed and resource usage of a DNN. (ii) The decision space is dramatically expanded. Because of their inter-dependence, the decisions for scheduling and optimization have to be made together rather than independently. If there are N DNNs with each having M potential placements and K exits, the decision space for co-optimization is as large as $(M \times K)^N$, growing rapidly with the power of N of both M and K . (iii) Due to the constant changes in runtime workloads and the repeated invocations of AI applications, decisions regarding optimization and scheduling must be made frequently at runtime. This process should be prompt, with minimal overhead, while also being able to adapt to dynamic changes in workloads, resource availability, and other factors during runtime.

To the best of our knowledge, no prior work has addressed this emerging problem. Applying prior solutions for spatial scheduling to this problem is inadequate. Previous methods for spatial scheduling mainly built on Deep Reinforcement Learning (DRL). They fall into two categories. The first is *centralized DRL*. An example is COSREL (XYY⁺21). It builds only one central DRL

agent, which schedules one DNN request each time. The second category is *decentralized DRL*, where each app carries its own DRL agent for predicting its appropriate placement. An example is the system by Sung et al. (SCN⁺23). Both have shown some promising results on DNNs of fixed structures, but none works well for the EDOSP. The *centralized DRL* method is subject to scalability limitations. Our experiments show that as the decision space grows, the quality of the decisions by that method drops drastically (details in Section 4.7.3 and 4.7.2). The *decentralized DRL*, on the other hand, is subject to a fundamental flaw: Its learning process lacks a convergence guarantee. In other words, its learning may not be able to converge to a policy that gives stable predictions, resulting in inferior performance (details in Section 4.7.4).

This dissertation presents our systematic exploration of the EDOSP and a novel solution named COSMA (contention-conscious co-optimization of DNNs and scheduling for mobile AI). We identify four principled challenges of EDOSP: (i) the large combinatorial decision space of EDOSP, (ii) the demands for high-quality decisions in all scenarios, (iii) the large differences among DNNs in their latency and performance metrics, (iv) and the dynamically changing nature of DNN usage in terms of invocation frequencies and timings.

COSMA addresses all these challenges. It is built on Centralized Training and Distributed Execution (CTDE), a type of cooperative Deep Reinforcement Learning (DRL) developed in the AI field and introduced into EDOSP by us. CTDE is efficient for training and prediction and, importantly, offers convergence guarantees. However, to materialize CTDE to effectively address those challenges, four major hurdles must be overcome. The first is to formulate the EDOSP such that it can be mapped to the theoretical framework of CTDE. The second is to come up with appropriate definitions of the key elements (states, actions, reward functions, etc.) of CTDE in this context by considering the specific complexities in EDOSP. The third is to develop the learning scheme to fit the properties of EDOSP. The final one is to create the software architecture to support the cooperative DRL in the mobile platforms to ensure efficacy and efficiency. We developed solutions for all of them and integrated them together into COSMA.

Our evaluations on a set of co-run settings and two mobile devices have validated the efficacy of COSMA in solving EDOSP. COSMA consistently outperforms centralized and decentralized approaches by $1.4\times-2\times$. Moreover, the experiments on co-runs of AI apps with real-world applications like TikTok, YouTube, and a Web Browser show robust results and consistently high gains by COSMA. The consistent benefits across hardware models further confirm the portability of COSMA as a solution for EDOSP on different hardware.

Overall, this dissertation makes the following major contributions:

- It provides the first systematic study on the co-optimization of DNNs and scheduling for mobile AI.

- It introduces CTDE into EDOSP for the first time.
- It develops a set of techniques for the principled challenges in EDOSP and composes them together into COSMA, the first known solution to EDOSP.
- It evaluates the efficacy of EDOSP, showing COSMA not only outperforms centralized and decentralized approaches by $1.4\times$ to $2\times$ but also demonstrates the robustness and portability in co-running scenarios with real-world applications across different hardware devices.

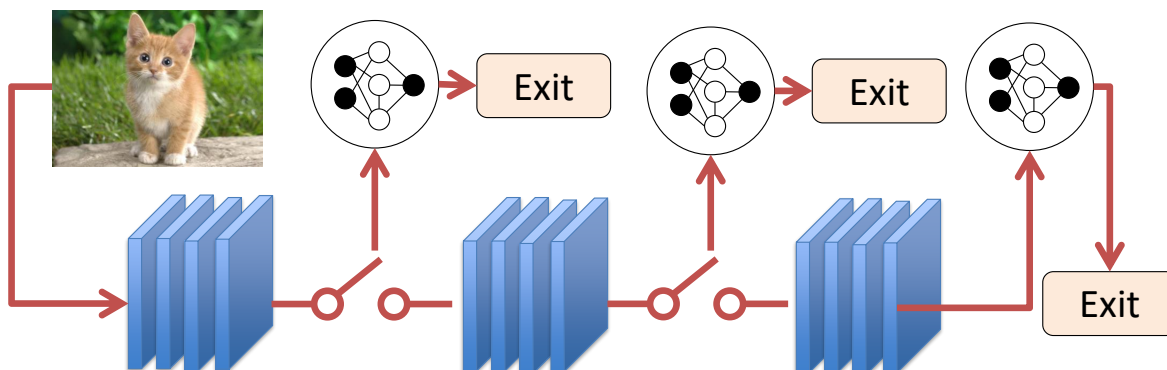


Figure 4.1: A multi-exit DNN with three potential outputs determined by thresholds like latency or accuracy. Deeper exits offer higher accuracy but greater latency. Blue layers are for feature extraction, circles indicate classifiers, and the cat image is the input.

4.2 Background

4.2.1 Reinforcement Learning and Value-Based Methods

Reinforcement Learning (RL) is a subset of machine learning where an agent interacts with an environment to make a sequence of decisions. The agent observes the state s_t from the state space S , takes an action a_t from the valid action space $A(s_t)$, receives a reward $r(s_t, a_t, s_{t+1})$, and transitions to the next state s_{t+1} . This process is stochastic and extends over discrete time steps until the termination of an episode. The aim is to discover a policy $\pi : S \rightarrow A$ that maximizes the expected cumulative discounted rewards over time, a concept formalized within Markov Decision Processes (MDP).

RL is semi-supervised learning. Unlike supervised learning, exploration and exploitation go hand in hand in RL. Offline learning is optional for RL. After deployment, RL continu-

ously explores the decision space and uses the feedback to refine its learned policy for better predictions.

Value-based methods in RL are concerned with estimating value functions that indicate the long-term rewards achievable by following certain policies from given states. The value function under policy π , denoted by $V^\pi(s)$, is the expected return of cumulative discounted rewards from state s , defined as:

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t, s_{t+1}) \middle| s_0 = s \right], \quad (4.1)$$

where $\gamma \in [0, 1]$ is the discount factor that weighs the importance of future rewards. The action-value function $Q^\pi(s, a)$, also known as the Q-value, is similarly defined but includes the expected return after taking an action a in the state s :

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t, s_{t+1}) \middle| s_0 = s, a_0 = a \right]. \quad (4.2)$$

Optimal value functions, denoted $V^*(s)$ and $Q^*(s, a)$, represent the maximum value that can be achieved under the best policy π^* . These are obtained by solving the Bellman optimality equations (Bel57):

$$V^*(s) = \max_{a \in A(s)} \sum_{s'} P(s'|s, a) [r(s, a, s') + \gamma V^*(s')], \quad (4.3)$$

$$Q^*(s, a) = \sum_{s'} P(s'|s, a) [r(s, a, s') + \gamma \max_{a'} Q^*(s', a')]. \quad (4.4)$$

Value-based methods such as Q-learning and Value Iteration use these equations to iteratively update the value estimates and converge to the optimal solutions, enabling the agent to make decisions that maximize cumulative rewards over time without the need to explicitly explore all policy combinations.

4.2.2 Deep Reinforcement Learning

Deep reinforcement learning (DRL) refers to the use of deep neural networks to approximate policies or value functions in reinforcement learning tasks. One of the frequently used algorithms is the deep Q-learning algorithm (DQN), a prominent method in value-action-based reinforcement learning

(MKS⁺15). DQN incorporates two key components: a *replay memory* (Lin92), which stores previous experiences as transaction tuples

(s, a, r, s', d) . The state s' is observed after taking the action a in state s and receiving reward r . d indicates episode completion; and a *target network* with slower weight updates compared to the *policy network*. These elements enhance sample efficiency and training stability by mitigating correlations among non-consecutive observations.

DQN’s primary objective is to utilize a deep neural network to approximate the function mapping a state s to the Q-value associated with each possible action a . While the original DQN paper employed Convolutional (CONV) and Fully Connected (FC) layers for its network, DQN offers flexibility in network architecture, allowing the use of CONV, FC, Transformers, RNNs, and more.

During training, DQN optimizes the network’s weights θ by sampling batches of b transitions from the replay memory and minimizing the *squared Temporal Difference* (TD) error:

$$L(\theta) = \sum_{i=1}^b [y_i - Q(s, a; \theta)]^2, \quad (4.5)$$

$$y_i = \begin{cases} r, & \text{if } d_t = \text{True}, \\ r + \gamma \max_{a'} Q(s', a'; \theta^-), & \text{if } d_t = \text{False}. \end{cases} \quad (4.6)$$

where the targeting network, represented by parameters θ^- , is periodically updated in sync with the policy network, with updates occurring every C iteration. This updated strategy maintains target network stability while adapting it to changes in the policy network over time.

4.3 EDOSP Problem Statement

To clarify, this section gives a formal definition of the EDOSP.

Given: A collection of applications (apps) D installed on a device equipped with DNNs, and a set of apps A that do not support DNNs. Multiple apps may run concurrently. Each DNN in D has multiple options in terms of its optimizations (e.g., exits) and in terms of its placements (or called execution modes) on hardware. Notation-wise, for every DNN, there are K distinct placements (or execution modes) with each impacting the utilization of hardware accelerators including CPU, GPU, DSP, and NPU in different ways; it meanwhile has V distinct model variants offering trade-offs between computational cost and performance accuracy. At each time step t , a policy π decides which placement, $k \in K$, and what variant, $v \in V$, each DNN in D shall take. The policy π can implemented as the policy followed by a central decision maker c or as a collection of policies followed by each DNN in D in making their own decisions.

Objective: Find a policy π that chooses $(k_{d,t}, v_{d,t}) \in K \times V$ for each DNN d at each time step t

such that the speed and quality of the DNNs are maximized.

Assumptions:

1. Job priority and temporal scheduling and associated scheduling fairness are taken care of by the underlying operating systems as in current mobile systems.
2. The system is open. New jobs and apps may come and go at any time; no assumptions are made about the predictability of their arrival.

Constraints:

1. Apps are isolated such that by default no app may access other apps' status or internal information.
2. Apps may communicate with a central agent and send and receive information related to scheduling and optimization decisions.
3. The system-level resource status is available for the scheduling agent to access.

4.4 Challenges and Design Principles

EDOSP features four fundamental challenges, which motivate the design principles of the solutions as follows.

(i) Huge searching space with real-time response requirement: efficiency and scalability required. As mentioned in the Introduction, the decision space for co-optimization, denoted as $(M \times K)^N$, expands exponentially with an increase in N . This complexity is compounded by a larger search space that includes all combinations of co-running DNNs, co-running Apps, and system programs. Testing all possibilities to find an optimal decision is impractical. Meanwhile, DNN jobs, typically more time-consuming than traditional programs, necessitate that the decision-making process be swift, ensuring real-time responses without impeding DNN inference. The latency of decision-making, therefore, must be considerably shorter than that of DNN inference.

(ii) Variations of DNNs: accommodating and fair treatment required. The large diversity of DNN models, along with their variants produced with optimization techniques like pruning, quantization, and multiple-exit architectures, calls for a solution that is adaptable to all DNN types and their variations. Moreover, this solution shall give fair treatment to DNNs and avoid penalizing a DNN because of its length or other non-priority factors. While optimizing for the collective benefit of all DNNs, it is crucial not to bias certain DNNs at the expense of others, ensuring reasonably prompt response times for all.

(iii) Dynamic nature of DNN usage: adaptivity required. DNN usage patterns are inherently dynamic, varying over time, with each DNN having distinct tasks and purposes. Additionally, the patterns of co-running DNNs change intermittently. For instance, image recognition and generative AI tasks might operate concurrently at times and separately at others. Furthermore, even identical DNNs may not experience consistent request frequencies, with the arrival of requests often being irregular. As such, the solution must exhibit strong adaptability to cater to these varying scenarios.

(iv) High-quality decision-making demands in all scenarios: convergence guarantee required. Considering the complexity of the decision and environment space outlined earlier, it is imperative that the solution consistently delivers high-quality decisions swiftly. "High-quality" here encompasses not just the accuracy of the decision but also its stability. Given the continuous influx of DNN requests, the solution must converge to a point where each decision yields a satisfactory outcome.

4.5 COSMA: Design and Implementation

This section describes our solution, COSMA.

4.5.1 Overview

COSMA is designed by following the four principles listed in the previous section. Its introduction of CTDE as the underlying algorithm to continuously learn and explore the decision space allows it to effectively avoid spending time in the non-promising subspaces and, at the same time, attains the convergence guarantees. Its high efficiency and flexibility in handling the special complexities of EDOSP come from several other innovations in COSMA, including its formulation of the EDOSP as a *decentralized partially observable Markov decision process (Dec-POMDP)* to fit CTDE, its definitions of the key elements in cooperative DRL to accommodate the variations of DNNs in latency and quality, its design of an effective learning scheme, and its software architecture to fit the mobile computing platforms. We give a detailed explanation of each of these components in the next several subsections.

4.5.2 Introducing CTDE

CTDE is a type of reinforcement learning working in a dynamic environment. It consists of multiple agents, with each holding a policy. At each time point, each of the agents makes a decision based on the current observation by following its policy, executes the decision by

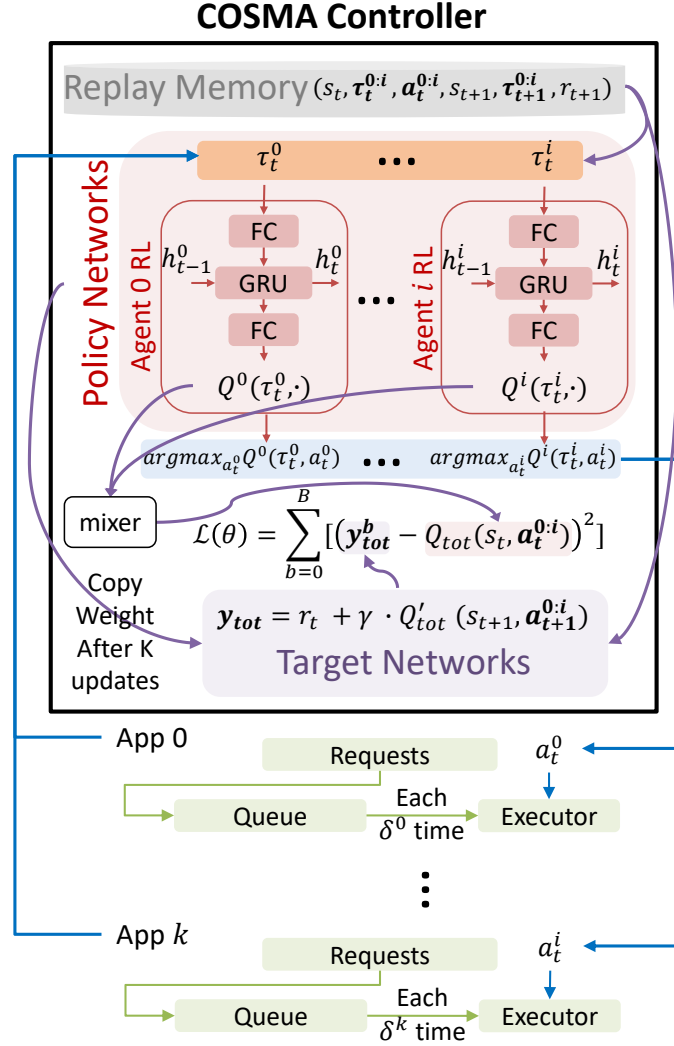


Figure 4.2: The COSMA framework orchestrates the co-scheduling of multiple DNNs. The execution workflow, depicted with blue arrows, involves: (1) gathering local states $\tau_t^0, \dots, \tau_t^i$ from the participating Apps; (2) obtaining the selected actions a_t^0, \dots, a_t^i for each agent from the Policy Network; (3) compiling the subsequent local states $\tau_{t+1}^{0:i}$, action set $\mathbf{a}_t^{0:i}$, global states s_{t+1} , and computing the total reward r_{t+1} for each agent in the transition from time slot t to $t + 1$, subsequently storing these into the replay memory. The training workflow, indicated by purple arrows, includes: (1) selecting a batch from the replay memory to feed into the Policy Networks and Target Networks; (2) calculating the loss value using Q_{tot} and y_{tot} ; (3) updating the weights of the policy networks. Following K updates, the policy networks transfer their weights to the target networks.

taking a corresponding action, and observes the changes in the environment states. The history of the actions and observations are then sent to a central trainer, which calculates the overall

rewards of the actions and then updates the policies of those agents. This central-training and decentralized execution process continues as long as the environment keeps changing.

COSMA chooses CTDE as the learning algorithm for several reasons.

First, the EDOSP has a natural mapping to CTDE: Each app can map to an agent in CTDE, each optimization option (placement, exit) can map to a decision in CTDE, the processing of a request by the app maps to an action in CTDE, and the model an app uses to select optimization options maps to the policy of an agent in CTDE.

Second, CTDE has some properties well aligned with the objective of COSMA. (i) It continues learning and predicting, which matches the needs of COSMA for adapting to continuous changes in workloads in mobile systems. (ii) Its distributed execution model makes it efficient and scalable, which matches the needs of COSMA for efficiency and scalability. (iii) Its centralized training model guarantees its learning to converge, which matches the needs of COSMA for general applicability. (iv) Its RL design makes it able to efficiently explore large decision spaces, which matches the needs of COSMA for tackling the enormous decision space of EDOSP.

4.5.3 Formulation of EDOSP

Despite the intuitive mapping between CTDE and the problems COSMA tries to solve, to materialize the connections into a complete solution to EDOSP, it is necessary to conduct some deeper examination and formulation of the dynamic process within EDOSP in light of CTDE.

We formulate EDOSP into a *decentralized partially observable Markov decision process* (Dec-POMDP) (BGIZ02), one of the kinds of dynamic processes that CTDE is designed to address.

As a Markov decision process, the state of a Dec-POMDP changes over time, and the state of the next time point is determined by the state of the previous time point and the actions taken at that time. Its decision process is *decentralized*, meaning that it consists of multiple decentralized agents that make their individual decisions. Its state is *partially observable*, meaning that each agent can observe only part of the states in the environment.

A Dec-POMDP is denoted as $G = (S, N, A, P, r, O, Z, \gamma)$. We next explain each of the components and how we define them in EDOSP. Our description will highlight how COSMA formulation avoids the biases caused by the differences in DNNs and how the mobile systems with DNNs can be modeled into a decentralized, partially observable Markov process.

States S . The state, represented as $s \in S$, encapsulates the current environmental conditions. The system conditions in the mobile device under EDOSP can be regarded as the global environmental conditions in Dec-POMDP, while each app's conditions form the local states of the

app. Specifically, we define s as (*tasks running ratio, memory usage, swap usage, CPU usage, and GPU usage*) in EDOSP as the global states, while the remaining jobs in the job queue of a DNN-based app as the local state.

Agents N and Actions A. Each DNN-based app is an agent $i \in N \equiv \{1, \dots, N\}$, and an action a_i by an agent is that it executes the DNN with placement k and optimization v ; $(k, v) \in K \times V$. The actions by all the agents collectively form a joint action $\mathbf{a} \in \mathbf{A} \equiv A^n$ in Dec-POMDP at each time step.

Transition function P. $P(s'|s, \mathbf{a}) : S \times \mathbf{A} \times S \rightarrow [0, 1]$ models how the environment evolves in response to a specific state and joint action combination. The value is the probability for the environment to change to state s' from s in action a . It corresponds to the mobile system state changes after the DNN apps' executions with (k, v) in a time step based on the mobile system's previous state.

Reward function r. All agents in Dec-POMDP must share a common reward function $r(s, \mathbf{a}) : S \times \mathbf{A} \rightarrow \mathbb{R}$. Defining r in EDOSP is crucial but also tricky. It should align with its objective, maximizing the DNN speeds and quality as stated in Section 4.3. However, uncaredful definitions could lead to undesired biases. For example, our preliminary approach involved a straightforward summation of negative latency and quality. Yet, this approach resulted in suboptimal throughput, as the system tended to deprioritize DNN tasks with lower latency in favor of those with higher latency, thereby compromising efficiency. We eventually identified the following definition, which captures the combined rewards of the speed and quality of DNNs while avoiding the biases from DNNs' differences:

$$\sum_{d \in D} \left(\alpha_d \frac{\#Completes_t^d(k_{d,t}, v_{d,t})}{\#Jobs_t^d(k_{d,t}, v_{d,t})} + \beta_d \cdot \text{Quality}_t^d(k_{d,t}, v_{d,t}) \right) \quad (4.7)$$

here, $\#Jobs_t^d$ represent the number of requests (or jobs) arrived for DNN d in time step t . $\#Completes_t^d$ is how many of the jobs are completed in time step t . So, the first component in the parentheses measures the speed aspect of the DNN's executions. By this definition, every incoming job is assigned equal importance, compelling each agent to maximize job completions within every time slot.

The second component, Quality_t^d , is the accuracy aspect of the DNN's executions; the higher the accuracy of the DNN variant is, the larger this component is. To neutralize biases stemming from various qualities and metrics of different DNN models, we have established standardized quality values (e.g., -0.5 for a low-fidelity variant, 0 for a mid-fidelity variant, and +0.5 for a high-fidelity variant). Selecting a variant with higher accuracy grants the agent bonus while choosing less accurate options results in penalties. Bonus and penalties are consistently applied to all agents.

The weighted sum of the speed and quality components combines the two objectives into one. Because of the normalization in the speed and the quantization of the quality, the two components are put to a similar scale, making the weighted sum a simple but intuitive way to balance them in the final score.

The weights α_d and β_d allow users to express their preferences over the speed and quality for a DNN; they are set to 2 and 1 by default.

In summary, the definition of this reward standardizes throughput during the time slot and the quality of the results across agents. It effectively eliminates biases between different agents as well as among various DNN variants. Consequently, this standardization ensures that no matter the type of DNN input, biases are mitigated.

Observation function O and Observations Z . Each agent makes its individual observations, denoted as $z \in Z$, determined by the observation function $O(s, i)$, which maps the current system state s to agent i 's observation: $S \times N \rightarrow Z$. In EDOSP, the observation function is implicit, and the observations are the local states observed by each DNN-based app.

Reward discount factor γ . This factor is a standard RL factor, quantifying how much an RL agent values immediate rewards compared to rewards that it might receive in the future. It is set to 0.99 by default.

4.5.4 Components for CTDE Learning

After formulating EDOSP into Dec-POMDP, we can now apply CTDE to solve the decision problems in EDOSP. Using the notations in Section 4.5.3, the objective of CTDE is to make each agent learn a stochastic policy that predicts the rewards an action may bring based on the history and the current state, referred to as $\tau^i \in T \equiv (Z \times A)^*$. stochastic policy is denoted as $\pi^i(a^i | \tau^i) : T \times A \rightarrow [0, 1]$.

To do that, we need to define the components needed by CTDE learning. Similar to other DRLs, CTDE learning can be based on Q-value factorization methods or actor-critique methods. Our design is based on the former for its simplicity and efficiency.

Policy network. Policy network is at the core of Q-value-based CTDE. It takes the action-observation history τ_t^i of agent i and an action a^i , and predicts the reward values of the action $Q^i(\tau_t^i, a^i)$. It is what CTDE tries to learn. In COSMA, the policy network consists of three Fully Connected (FC) layers for encoding inputs, one Gated Recurrent Unit (GRU) layer (CGCB14) to capture temporal dependencies, and one FC layer for decoding.

Target network. This network is a copy of the policy network used to generate stable target values for training. Unlike the policy network, its weights get updated less frequently once every K policy network updates.

Replay memory. It acts as a database to store a collection of experiences denoted by tuples (s_0, a_i^0, r_0, s_1) through $(s_t, a_t^i, r_t, s_{t+1})$. These tuples capture the states s , actions a^i of agent i , and rewards r . The experiences are sampled in batches for the purpose of training the networks.

Batch sampling. This process involves drawing samples from the replay memory, which are then used to train the policy and target networks.

Loss function $L(\theta)$. We define it as a mean-squared error loss function. It is used in training the policy network to reduce the difference between the predicted total action-value Q_{tot} and the target total action-value Y_{tot} .

Reward function. The reward function is the same as r as defined in Section 4.5.3.

4.5.5 Learning Algorithm

We use the existing *value function factorization* (RSDW⁺20) as the learning algorithm for COSMA. Compared to other learning algorithms for solving Dec-POMDP (Tan93; WEH20; RSDW⁺20; ZYL⁺18; VKR09), this algorithm is guaranteed to converge and gives competitive quality in general cases.

Convergence guarantee. As previous work has shown (RSDW⁺20), value function factorization is guaranteed to converge if the factorization adheres to the Individual-Global-Max (IGM) condition:

[IGM] A joint action-value function $Q_{j_t} : T^N \times A^N \rightarrow \mathbb{R}$ satisfies IGM if individual action-value functions $\{Q_i\}_{i=1}^N$ exist such that:

$$\operatorname{argmax}_a Q_{j_t}(\tau, a) = \begin{pmatrix} \operatorname{argmax}_{a_1} Q_1(\tau_1, a_1) \\ \vdots \\ \operatorname{argmax}_{a_N} Q_N(\tau_N, a_N) \end{pmatrix} \quad (4.8)$$

The IGM condition posits that the optimal joint action corresponds to the aggregation of optimal individual actions. This implies that if the IGM condition is met, the complexity of the multi-agent reinforcement learning (RL) problem effectively reduces to that of a single-agent RL problem. This critical factorization enables the application of single-agent convergence guarantees to the multi-agent context of the CTDE framework.

In our work, we explore the use of two types of Value Function Factorization: *additivity* and *monotonicity*. The former is derived from the VDN approach (SLG⁺18), while the latter is based on the QMix methodology (RSDW⁺20). Both *additivity* and *monotonicity* factorizations have been demonstrated to satisfy the IGM condition. Exploring them both helps reveal their strengths and weaknesses in the EDOSP context.

4.5.6 COSMA Software Architecture and Workflow

This part describes the design of the software architecture of COSMA in mobile systems and how it works with apps that may come and go. Noteworthy features include the use of *virtual agent* to forego the requirement for every app to be equipped with a CTDE agent, the smooth transitions when the set of active apps change in the system, and the flexible support for the policy networks training with or without the optional pre-training stage.

Figure 4.2 illustrates the architecture of COSMA. All components of CTDE are encapsulated into one app, named *COSMA controller*. It contains a *replay memory*, storing the action-observation histories, a policy network, and a target network. Each DNN-app that adopts the COSMA protocol sends its local observations to the *COSMA controller* at each time step, receives the optimization decisions from the *COSMA controller*, and executes its DNN accordingly. The *COSMA controller* is responsible for training the policy and target networks with the Value Function Factorization method using samples in the replay memory and making predictions of DNN optimization and scheduling for each active app. The learning and prediction go hand in hand, continuing throughout the use of the mobile system. COSMA does its job and smoothly adapts to the workload changes, such as an active app termination or a new app getting started.

Virtual Agent. An important feature of the architecture design is its concept of *virtual agent*, which foregoes the requirement for every app to be equipped with a CTDE agent. It is embodied by the policy networks in the COSMA controller, which consists of a collection of *virtual agent networks*. Each virtual agent network serves as the policy network for an *active* DNN-based app; the app then does not need to contain a CTDE agent anymore. When an app sends a query to the COSMA controller, it also passes its app ID; the corresponding virtual agent in COSMA will handle its query, replying with the appropriate optimization decision. The maximal number of virtual agents is fixed (3 and 5 in our experiments). When the total number of apps exceeds the maximum number of virtual agents, the virtual agent of the oldest non-active app will be reused for other active apps when necessary.

This design has several benefits: (i) It simplifies the requirements of users' apps and avoids increasing the size of apps. (ii) It makes it easier to upgrade the policy networks as all of them are now part of the *COSMA controller*. (iii) It removes the need to send updated policy network weights and hence reduces the communication overhead. (iv) When the set of active apps changes, the COSMA controller can smoothly adapt to the change by updating the virtual agents.

Training. The training of COSMA follows the standard CTDE scheme. It involves the replay memory randomly sampling transactions as batch samples to refine the policy network, thereby improving its decision-making algorithms. The Mixer component reflects the various methods of amalgamating the Q-values from all decentralized agents. Specifically, the Value De-

composition Networks (VDN) (SLG⁺18) aggregates these values, whereas the QMix (RSDW⁺20) incorporates several ReLU and fully connected (FC) layers to integrate global information. The gradients from the loss function are then back-propagated through the policy network, facilitating parameter updates. After a pre-determined number of updates, the policy network’s weights are synchronized with the target network. This dynamic interaction of action, observation, and feedback creates an environment conducive to continuous learning and adaptation, enabling the centralized training agent to iteratively refine its strategies based on the outcomes of decentralized execution.

In scenarios where the set of apps is largely fixed (e.g., on a vehicle assistant system), automatic pre-training can be taken as an option to condition COSMA agents ahead of deployment. COSMA offers a customizable *workload generator* which invokes DNN-based apps and generates requests to them to facilitate the pre-training by *COSMA controller*.

Inference. An inference is made by the corresponding virtual agent when an app sends a query to *COSMA controller* (along with its current state). The inference latency for each RL agent on the device is marginal (e.g., 45 μ s on the Samsung Galaxy S21 mobile phone), which is substantially lower than the latency associated with a typical DNN inference in each agent. Therefore, the added overhead from the RL inference is negligible and does not compromise the overall system performance.

Table 4.1: DNN Models and Variants and their Standalone Latencies in different Setting of TFLite Runtime Interpreter (unit: million-seconds). The Android Neural Networks API (NNAPI) (Dev23) is an Android C API designed for running computationally intensive operations for machine learning on Android devices. K threads means use K threads to run on CPU

Models	Quality	Size (KB)	1 Thread	2 Threads	3 Threads	4 Threads	GPU	NNAPI
0:EfficientNet (ms)	73.8	11,608	36	26	22	20	15	60
	75.4	18,125	38	30	24	26	14	51
	77.6	20,644	53	39	32	34	17	65
1:MobileNet (ms)	65.9	7,691	14	11	10	11	7	26
	72.9	13,662	31	24	20	22	12	43
	76.5	23,806	55	39	33	34	17	63
2:YoloX (ms)	32.6	19,895	575	363	278	254	103	318
	40.2	35,166	862	595	456	408	157	500
	46.6	99,049	2334	1721	1393	1122	311	993

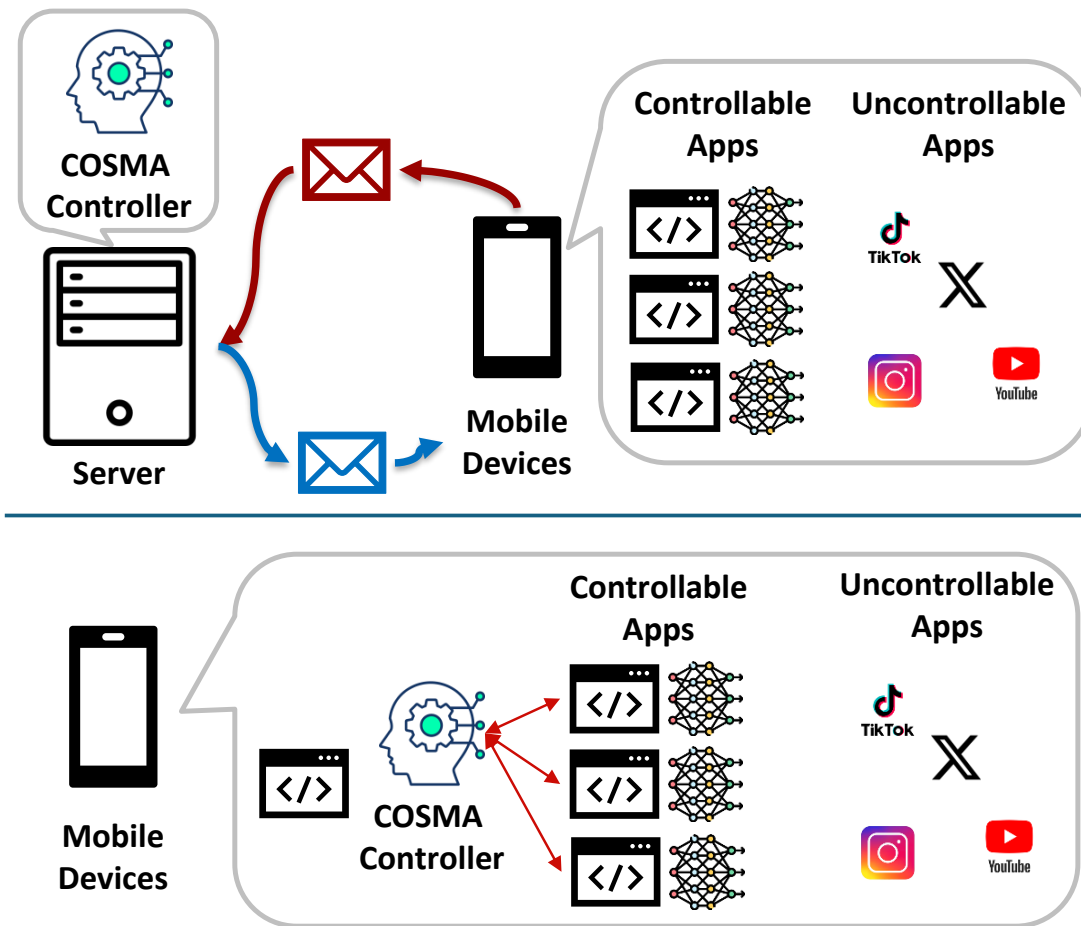


Figure 4.3: At the top is the Server Cooperative Learning Structure. The red message encompasses the current and next states, along with additional details such as hit/miss counts, aiding the COSMA controller in reward calculation and action generation for subsequent steps. The blue message denotes the chosen actions for each controllable app, along with the signal to initiate the next step. At the bottom is the On-Device Learning Structure, featuring the COSMA controller.

4.6 On-Device Learning v.s. Server Cooperative Learning

We explore two system design approaches for the COSMA controller: *server cooperative learning* versus *on-device learning*.

Server Cooperative Learning: Illustrated at the top of Figure 4.3, this method involves the mobile device transmitting the current status of App co-run execution to the server via message passing packets. This includes job queue status in each application, model ID, incoming jobs, hit-and-miss counts, and other system-related data. The RL agent learns on the server and

Table 4.2: The comparison of two structure designs: Server Cooperative vs. On-device learning

Structure	On-Device	Server Cooperative
Message Passing (forward and back)	<0 ms	1000 ms ~3000 ms (Include resend loss packet)
Weights Update	250 ms ~350 ms	10 ms ~20 ms
Mem Requirement of RL on device	743 KB	0 KB (No RL DNN holds on Device)

sends back action decisions (i.e., placement and version selection) to the mobile device. While this approach enables the RL agent to learn weights faster (10ms 20ms, indicated in Table 4.2) and more accurately due to the superior computational resources of the server, it may face challenges such as packet loss and communication overhead (refer to Table 4.2), which could hinder rapid and frequent updates.

In real-world scenarios, this approach is more suitable for automotive systems (LLT⁺19; way21; bai21) where both servers and devices operate within a closed system, meaning mobile apps and device services are owned by the same entity. In such environments, concerns regarding privacy issues related to sharing workload information between devices are less significant. Additionally, performing expensive backpropagation operations during RL agent learning on a server is more feasible, considering latency and energy consumption factors.

On-device Learning: Depicted at the bottom of Figure 4.3, in this approach, the RL agent resides on the mobile device, learning and acting locally with direct access to the device’s execution status. This method allows users to keep their sensitive App execution data locally and enables the RL agent to be responsive without the overhead of message communication. However, the tradeoff includes higher energy and memory consumption, longer weight update latency (refer to Table 4.2), and a potentially less precise weight learning process (LZC⁺22; CGZH20) due to the limited computational resources on the device. In real-world scenarios, this approach is more suitable for personal devices such as mobile phones and tablets, where most device Apps operate in an open environment managed by different companies—users who may be less inclined to share information about their app usage.

In summary, the COSMA controller can adapt flexibly to either of these structures depending on the scenario. Additionally, a *hybrid* scheme is possible: with user permission, the company can perform server learning by pre-training the RL agent, and the user can conduct on-device learning using the transferred weights before use, enabling personalized adaptability. Subsequently, the company can conduct post-learning on the server after collecting more workload data from users. This flexibility is inherent in our design and implementation.

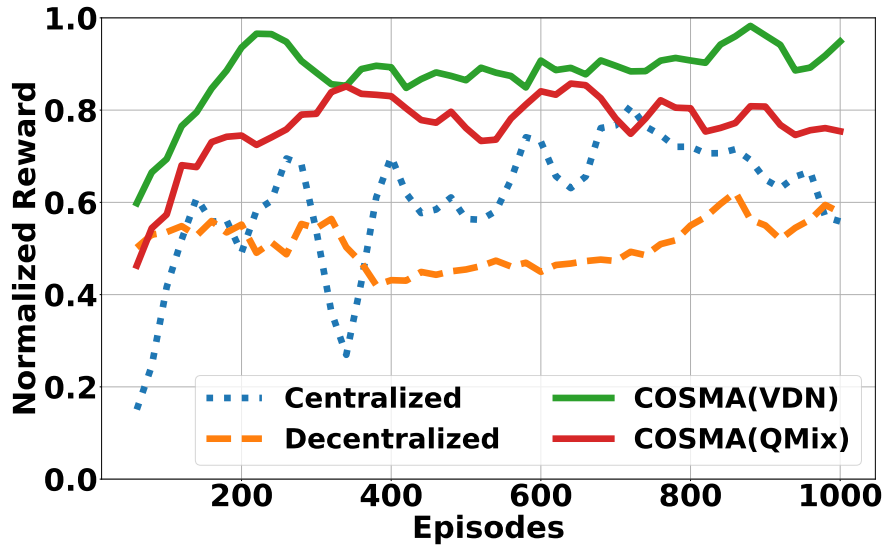


Figure 4.4: Static Selected **Three** models in each Episode for co-scheduling multiple DNNs without Co-running Apps.

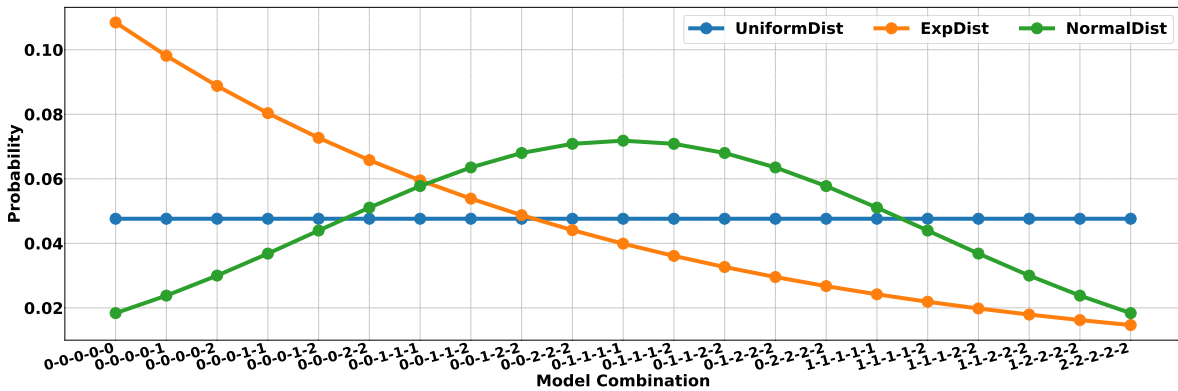


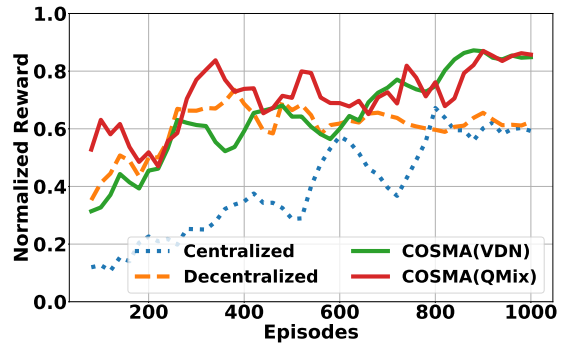
Figure 4.5: Three distributions of different model combinations in each episode. (0-0-1-1-2), for instance, means that two 0:EfficientNet, two 1:MobileNet, and one 2:YoloX in Table 4.1 will be co-scheduled together.

4.7 Evaluation

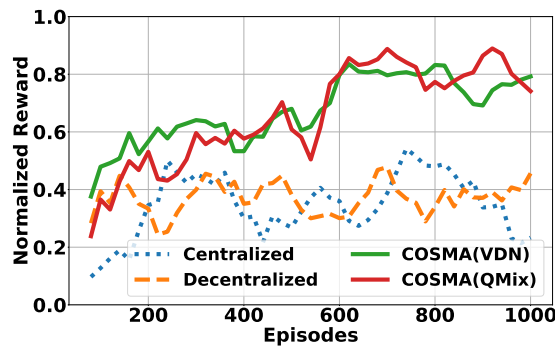
This section evaluates the efficacy of COSMA in optimizing DNNs and their schedules in a dynamic environment.



(a) Rewards Curve in the Uniform Distribution



(b) Rewards Curve in the Exponential Distribution



(c) Rewards Curve in the Normal Distribution

Figure 4.6: Randomly Selected **Five** models under three distributions under Figure 4.5 in each Episode for co-scheduling multiple DNNs without Co-running Apps.

4.7.1 Methodology

Evaluation platforms. We conducted our experiments on two mobile devices. (i) The first is a Samsung Galaxy S21 FE 5G mobile phone, which runs on Android 12 OS and is powered by Qualcomm SM8350 Snapdragon 888 5G SoC. The SOC consists of an Octa-core CPU (1x2.84 GHz Cortex-X1 & 3x2.42 GHz Cortex-A78 & 4x1.80 GHz Cortex-A55), Adreno 660 GPU, and Hexagon 780 DSP. The storage capacity is 128GB with 6GB RAM, and the voltage is 4.3V. (ii) The second mobile device is a Galaxy S10e having an Octa-core CPU (1x2.84 GHz Kryo 485 & 3x2.42 GHz Kryo 485 & 4x1.78 GHz Kryo 485) and Adreno 640 GPU. The storage capacity is 128GB with 8GB RAM, and the voltage is 4.3V. Because the observations on the two platforms are quite consistent, we focus our discussions in the following parts mostly on Galaxy S21 FE, and briefly report the observations on Galaxy S10e in Section 4.7.6.

DNN Models and Workloads We developed a workload generation app on Android phones to create various co-run scenarios and dynamic workload changes. It invokes multiple instances

of the DNNs through subthreads to create co-running among DNNs, along with other apps, based on workload generation plans.

The experimented workloads involve instances of three popular DNN models along with some real-world apps, including TikTok, YouTube, and Web-Browser. Each of the DNN models has three variants of different sizes, speeds, and accuracies, representing the multi-exits of the model. The details are shown in Table 4.1. The quality scores of the variants are set to -0.5 , 0.0 , and $+0.5$ for the Low, Mid, and High qualities of the model variants. All DNN models are in TensorFlow Lite (AAB⁺15) format.

To examine the robustness of COSMA, we let the workload generator produce workloads in four scenarios: (i) Scenarios with a static DNN set; (ii) Scenarios with a dynamic changing DNN set; (iii) Scenarios with a dynamic changing DNN set and additional random interference; (iv) Scenarios with a dynamic changing DNN set and other uncontrolled co-running apps. The details of the workloads will be given in the next several subsections, along with the experimental results.

We set every agent to go through three steps (take the current state as an input, do inference, and get predicted actions) in each episode by considering the trade-off between the training quality of the RNN structure in the RL agent and the minimization of total training time. Before starting a new episode, the system updates its internal parameters (weights).

Options for Scheduling There are six options for the spatial scheduling of a DNN, as shown in the rightmost six columns in Table 4.1. Specifically, a DNN may run on one to four CPU threads, on GPU, or on a mix computing units determined by NNAPI which uses APU when possible and falls back to GPU or CPU otherwise.

Methods to Compare Our comparisons include four methods:

- **Centralized:** policy networks only contain ONE agent RL but its input is $(\tau_t^0, \dots, \tau_t^i)$ and output is action set (a_t^0, \dots, a_t^i) for each App
- **Decentralized:** policy networks are divided into independent agent RLs. Each of them trains and does inference independently, which is similar structure in Sung et al. (SCN⁺23).
- **COSMA(VDN):** Our COSMA which uses VDN (SLG⁺18) as the value function factorization
- **COSMA(QMix):** Our COSMA which uses QMix (RSDW⁺20) as the value function factorization

We keep the common components the same setting for all the methods: The Replay Memory is set to 500, the batch size for learning is 16, the learning rate is 0.001, the target update rate (θ) is 0.005, the discount factor is 0.99, the learning optimizer is RMSprop (TH12) with settings as

($\alpha = 0.99$, $\text{eps} = 1e - 05$, $\text{weight_decay} = 1e - 5$), and all of them use Double Q-learning (VHGS16) as the weights update scheme.

4.7.2 Scenario I: Static Models Set

First, we assess the performance of the methods on three pre-selected co-running models in CPU and GPU co-running scenarios: EfficientNet (TL19), MobileNet (SHZ⁺18), and YoloX (GLW⁺21), over 1000 episodes. Figure 4.4 shows the reward comparison among Centralized, Decentralized, and our COSMA methods (VDN and QMix). The Centralized and Decentralized methods are represented by blue and orange dashed lines, respectively, while VDN and QMix are indicated by solid green and red lines.

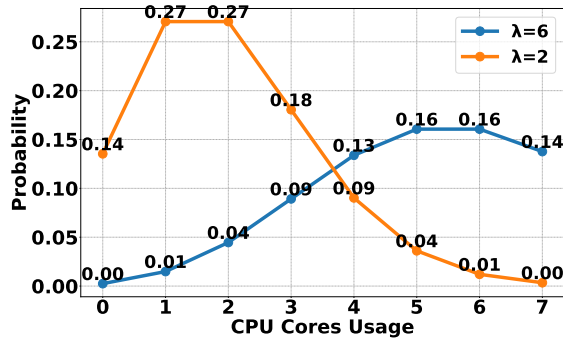
In Figure 4.4, the reward trajectory of the Centralized solution is characterized by significant fluctuations. Although the Decentralized solution initially outperforms Centralized approach, its performance gradually deteriorates. This decline reflects their limited adaptability in dynamic co-running environments. Conversely, both VDN and QMix show superior performance, maintaining higher normalized rewards with greater consistency. Notably, VDN often leads, benefiting from its effective state-value decomposition in a multi-agent context.

QMix closely trails VDN, adeptly maximizing rewards despite occasional fluctuations, likely due to its adaptive policy learning amidst the computational demands of co-running models. Additionally, compared to VDN, QMix incorporates the global state into its decision-making process. However, this approach has its limitations; when the global state remains unchanged, it may introduce unnecessary complexity, potentially reducing performance by introducing noise into the system.

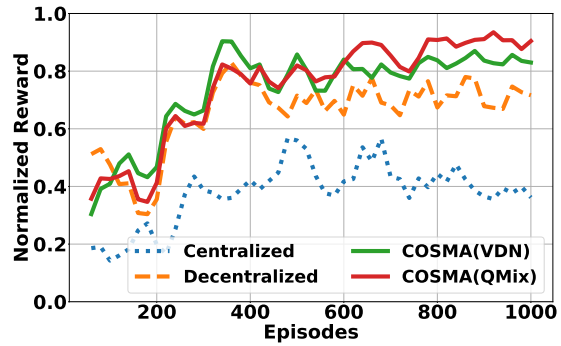
4.7.3 Scenario II: Random Models Set

Following the statically selected scenario, Figure 4.6a illustrates a more complex case with five co-running models, which are repeatedly selected five times from the three models listed in Table 4.1. The workload generator generates workloads by following each of three distributions of the co-run combinations: Uniform, Exponential, and Normal, as shown in Figure 4.5.

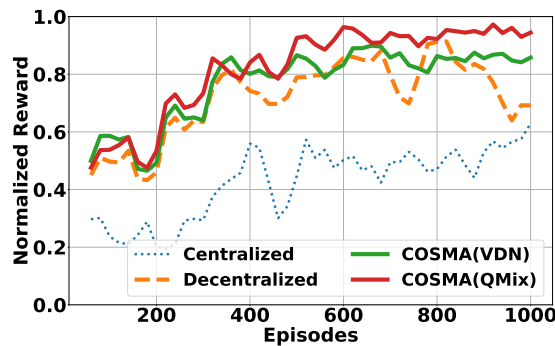
Figure 4.6a shows the results observed on the workload of the Uniform Distribution. The reward curves exhibit significant variability across episodes. Notably, the performance of the Centralized approach displays higher instability compared to other approaches and converges to suboptimal results. The Decentralized approach shows a more gradual improvement but fails to achieve the stability and higher rewards demonstrated by the COSMA. In contrast, both VDN and QMix exhibit robust performance, with VDN maintaining a slight advantage over QMix. This advantage arises from our COSMA's ability to strike a balance between



(a) CPU Cores Usage under Poisson Distributions



(b) Rewards Curves with Low CPU Interference



(c) Rewards Curves with High CPU Interference

Figure 4.7: Selected **Five** models under Uniform distributions in each Episode for co-scheduling multiple DNNs with CPU Usage Co-running Apps.

shared knowledge and individual model learning, allowing them to adapt efficiently to random changes.

Next, we evaluate the adaptiveness of COSMA under different distributions that make certain model combinations more likely to be selected. Figure 4.6b and Figure 4.6c show the results for the Exponential and Normal Distributions, respectively. The Centralized approach still exhibits significant instability, similar to the uniform distribution, primarily due to the vast action space it encounters. Notably, the Decentralized approach struggles to maintain a consistent upward trajectory, showcasing its sensitivity to the skewed model selection process. On the other hand, COSMA continues to outperform both the Centralized and Decentralized approaches. Interestingly, the final performance of QMix catches up with that of VDN. It is attributed to the increased complexity of both the action space and the environment, wherein the mixer in QMix begins to demonstrate its capabilities.

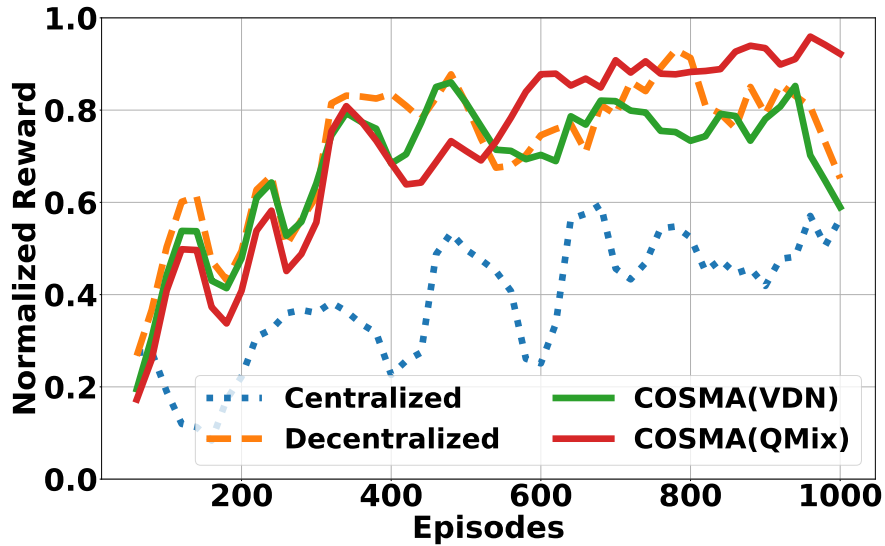


Figure 4.8: Selected **Five** models under Uniform distributions in each Episode for co-scheduling multiple DNNs with CPU/GPU Usage Co-running Apps.

4.7.4 Scenario III: Dynamic Workloads with Random Models and Additional Random Interference

We also evaluate the methods when DNNs run with additional random interference. The workloads in this experiment are the same as those in the previous subsection, except that additional CPU or GPU interferences are created by two contrived apps, one having a random CPU usage and the other having high GPU usage. They create an unstable global state. More specifically, the CPU interference app occupies N cores in each time step, with N following one of the Poisson distributions in Figure 4.7a. The GPU interference app is a GPU-based TFLite model that keeps conducting inferences; through GPU monitor in Android, the GPU utilization is up to 100% when that app is running.

Figure 4.7 reports some of the observations. COSMA, especially QMix, exhibits superior adaptability to CPU load variations. Figures 4.7b and 4.7c show that QMix outshines VDN by effectively reallocating tasks between CPU and GPU during high CPU load situations. These results demonstrate the benefit of incorporating global system state information (e.g., CPU, GPU, and Memory status) into the training process, as evidenced by QMix’s performance. This is particularly true under conditions where both CPU and GPU resources are heavily taxed, as shown in Figure 4.8. The adeptness of QMix at utilizing comprehensive system state information enables it to surpass other methods, such as VDN, which lack this broader contextual understanding. The findings underscore the necessity of an extensive system perspective

in crafting reinforcement learning strategies that enhance the co-scheduling of models in resource-restricted environments.

4.7.5 Scenario IV: Random Models set with Real-world Uncontrolled Apps

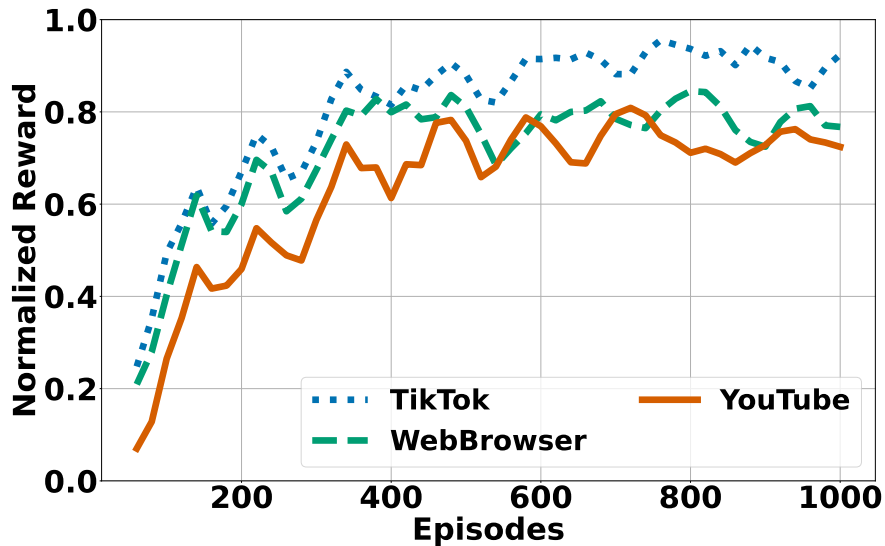


Figure 4.9: Selected **Five** models under Uniform distributions in each Episode for co-scheduling multiple DNNs with Real-world Apps including **TikTok**, **YouTube**, and **WebBrowser**. All results adopt **COSMA(QMix)**.

We also experiment with the cases where DNNs co-run with uncontrolled, real-world apps. An app is uncontrolled if it does not have COSMA protocol incorporated and hence does not communicate with the COSMA controller.

Figure 4.9 shows the performance of co-scheduling multiple deep neural networks (DNNs) with three different real-world applications: TikTok, YouTube, and Google Chrome Browser. The performance is evaluated under five selected DNNs in uniform distributions in Figure 4.5 for each episode, and all results adopt the QMix method.

As the number of episodes increases, all applications show an overall improvement in performance, indicating enhanced capabilities of the models. Web Browser, in particular, exhibits a highly volatile performance with notable fluctuations in normalized reward. This variability could mean that Web Browser’s workload poses greater challenges for COSMA or that its performance is more influenced by changes in the scheduling strategy. In contrast,

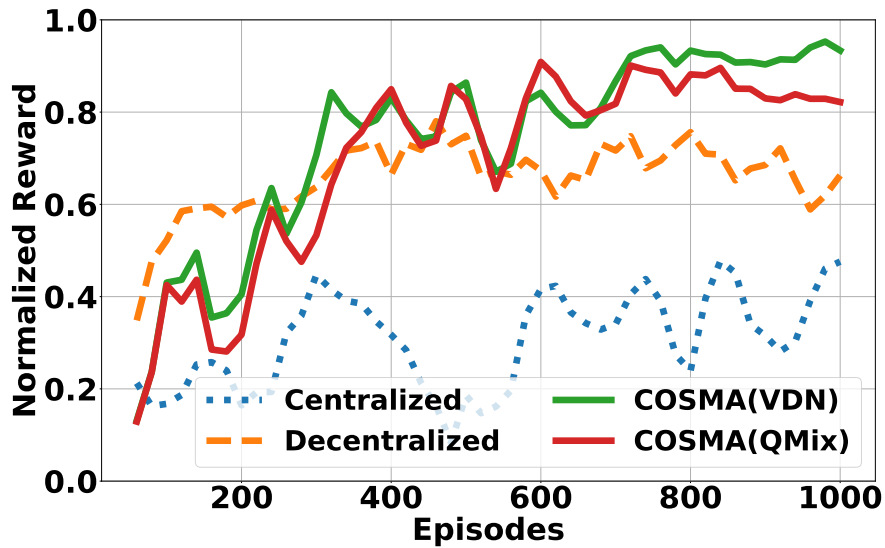


Figure 4.10: Used **Samsung Galaxy S10e** and Randomly Selected **Five** models under Normal distributions under Figure 4.5 in each Episode for co-scheduling multiple DNNs without Co-running Apps.

the performance of Tiktok is comparatively stable, showing fewer fluctuations, implying a more consistent or predictable task load that is easier to handle through COSMA. YouTube’s performance becomes more stable in the later episodes than the Web Browser. This suggests that the COSMA may find an optimal strategy in the presence of YouTube tasks during the end period of the episodes.

Towards the end of the episodes, the performance of the three applications appears to converge, suggesting that the models may have reached a plateau in learning or that the differences in co-scheduling complexity between the applications diminish over time.

4.7.6 Cross-Hardware Portability Test

As a continuous learning method, COSMA is inherently adaptive to hardware. Our experiments on a different mobile device, Samsung Galaxy S10e, confirm the portability. As Figure 4.10 shows, on Samsung Galaxy 21s FE, COSMA consistently outperforms other methods. As the learning process continues, the two COSMA methods both exhibit significantly larger rewards than the centralized and decentralized methods do. Their trends are consistent with those shown on Galaxy 21s in Figure 4.10.

4.7.7 Multi-exit DNN models

Experiments Setting To introduce greater complexity into the design space of each DNN model, we implement a multi-exit DNN model setup using MDSNet (HCL⁺18) in our experiments. This setup entails two dimensions shaping an action space for each model: the number of exits and TFLite execution engine options. In our experiments, we utilize a 5-exit MDSNet and 1 to 8 threads TFLite execution engine options, resulting in 40 possible actions. Additionally, we concurrently execute 5 MDSNet models in each episode, leading to a total action space of 40^5 , which is significantly large, rendering classical ML methods and traditional RL solutions ineffective. These multi-exit network experiments were conducted on a Samsung Galaxy S21 FE 5G mobile device. The first experiment involves a comparison of COMSA with decentralized and centralized RL solutions, while the second experiment aims to assess the robustness and stability of COSMA in co-scheduling multi-exit networks within a real-world app environment.

No Real-world Apps co-scheduling We evaluate the DNN models using 5 exits. With 8 available model placements (1-8 threads), and 5 models running concurrently, there will be a total of 40^5 combinations. Such an extensive action space necessitates a centralized (single) RL agent to learn large weight embeddings. This can easily lead to out-of-memory issues, regardless of whether learning occurs on-device or on a server. Therefore, employing a multi-agent learning solution is more appropriate in this scenario.

In Figure 4.11, we demonstrate that utilizing the COSMA controller with VDN (indicated by the orange solid line) or QMix (indicated by the green solid line) leads to learned rewards converging to 35% higher than those achieved with a decentralized solution (depicted by the dashed line) (SCN⁺23). Additionally, the COSMA controller exhibits more stable learning outcomes, underscoring the robustness of our proposed approach. VDN and QMix yield similar performance in this specific setting, where no co-scheduling application exists prior to the execution of the five models, ensuring that centralized execution workloads remain stationary for both cases.

Real-world Apps co-scheduling Even as the action space expands due to the added complexity of the multi-exit option in the model setting, the RL agent demonstrates resilience in learning the co-running DNN workload through multi-agent learning with COSMA (QMix), especially when background factors involve uncontrollable real-world applications. Illustrated in Figure 4.12, amidst background workloads like TikTok, YouTube, or WebBrowser, the RL agent within COSMA achieves convergence after approximately 400 episodes of learning.

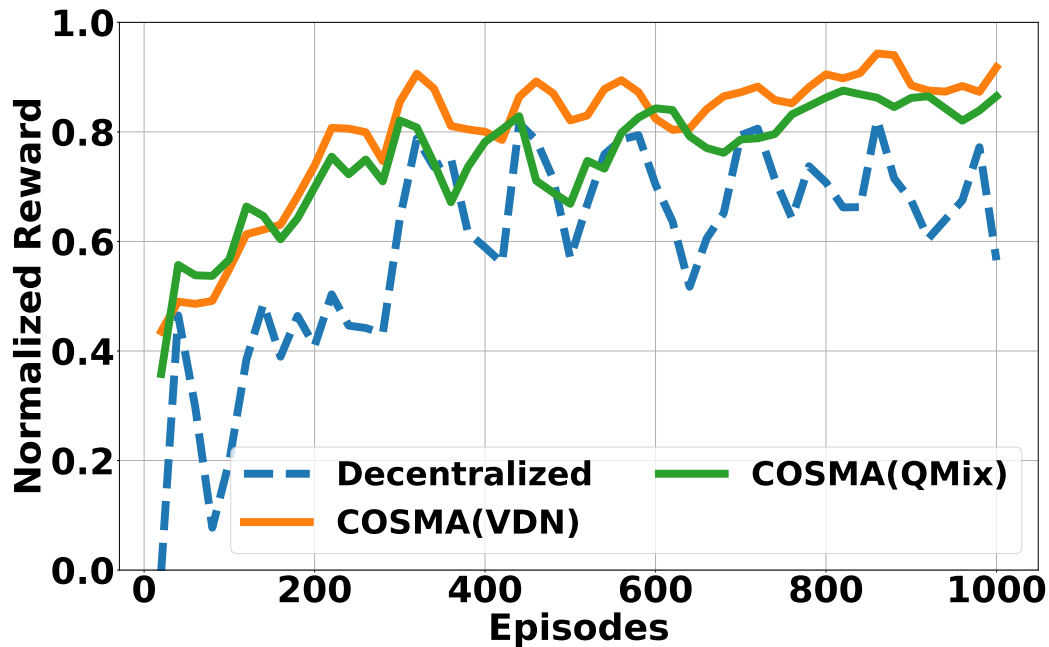


Figure 4.11: Concurrently co-scheduling **five** multi-exit models, each with 5 exits, and 8 placements (1-8 threads) per model, presents significant challenges. The vast action space in the Centralized structure (resulting in 40^5 possible outputs) demands extensive memory resources, totaling 12.64 GB, leading to out-of-memory issues on the RTX 4060.

4.7.8 Transfer Learning

Experiments Setting In this experiment, our objective is to assess the adaptability of COMSA, specifically how quickly it can adjust to new environments or changes within an environment. To achieve this, we employ transfer learning by utilizing the trained weights from one environment as the initial weights for another new environment. We examine three scenarios: (1) variations in the time pattern of model combinations, (2) the introduction of a new DNN app, and (3) modifications in the co-scheduling of a real-world app. These scenarios mirror the types of changes that COMSA may encounter in real-world scenarios, showcasing its adaptiveness and robustness. The terms "Normal" and "Uniform" refer to the distribution of model combinations in each episode, as illustrated in Figure 4.5. In the "Scratch" scenario, the weights of the RL agent network are randomly initialized. When training the RL agent network from scratch, we set a higher exploration rate, allowing for a 20% random action set selection before 400 episodes. Conversely, in the "Transfer" learning scenario, where the RL agent network learns from transferred knowledge, we set a lower exploration rate, with 20% random action set selection only before 200 episodes, indicating a quicker decrease in the exploration rate.

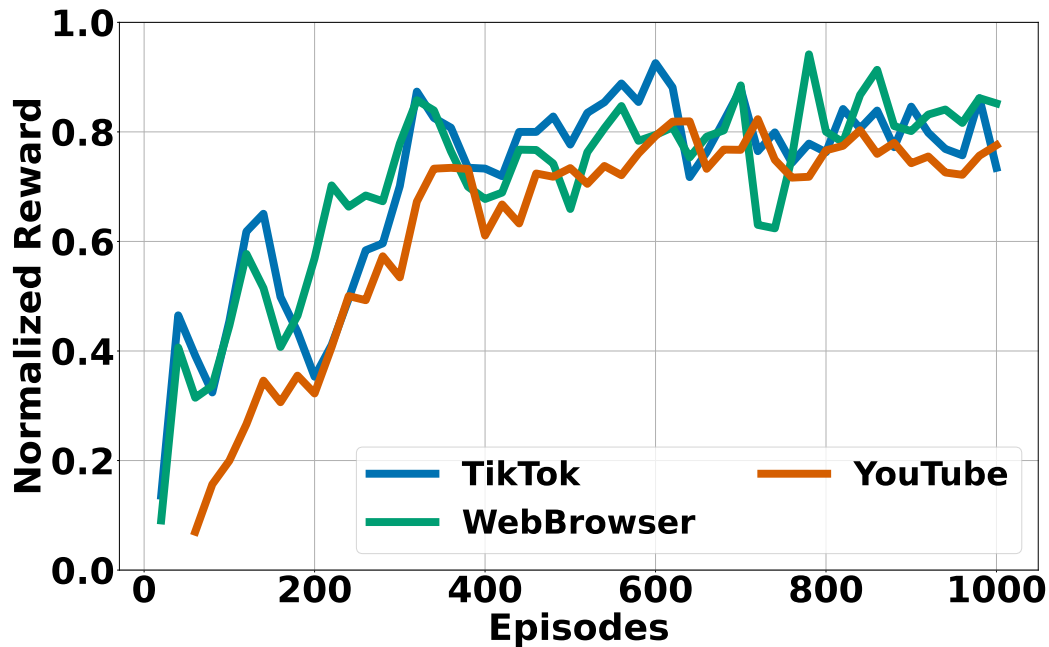


Figure 4.12: Co-scheduling **Five** multi-exit models with Real-world Apps including **TikTok**, **YouTube**, and **WebBrowser**. All results adopt **COSMA(QMix)**.

The change pattern of Model Combination Figure 4.13 illustrates the robustness of the RL agent’s learning capacity through transfer learning between weights learned from normal and uniform distribution. The dashed lines depict the reward convergence trajectory when training from scratch on normal and uniform distributions (refer to the model combinations pattern in Figure 4.13), while the solid lines represent training the agent from a normal distribution pattern to a uniform distribution and vice versa. While training from scratch typically requires about 400 episodes to achieve stable performance, transfer learning enables convergence to stability from early episodes, irrespective of the initial weight patterns learned. This indicates that even when employing the same set of models with different usage combinations, the RL agent can swiftly adapt from one distribution to another, regardless of the training method used for the original weights.

New DNN App participation Figure 4.14 depicts the adaptability of the RL agent’s learning capability as a new model is introduced. We examine two scenarios: (1) represented by the orange line, where the agent first learns from two models of similar size and latency (0: EfficientNet and 1: MobileNet), followed by the inclusion of a model (2: YoloX) with dissimilar size and latency within the original set. (2) represented by the green line, where the agent learns from two models with dissimilar size and latency initially (0: EfficientNet and 2: YoloX),

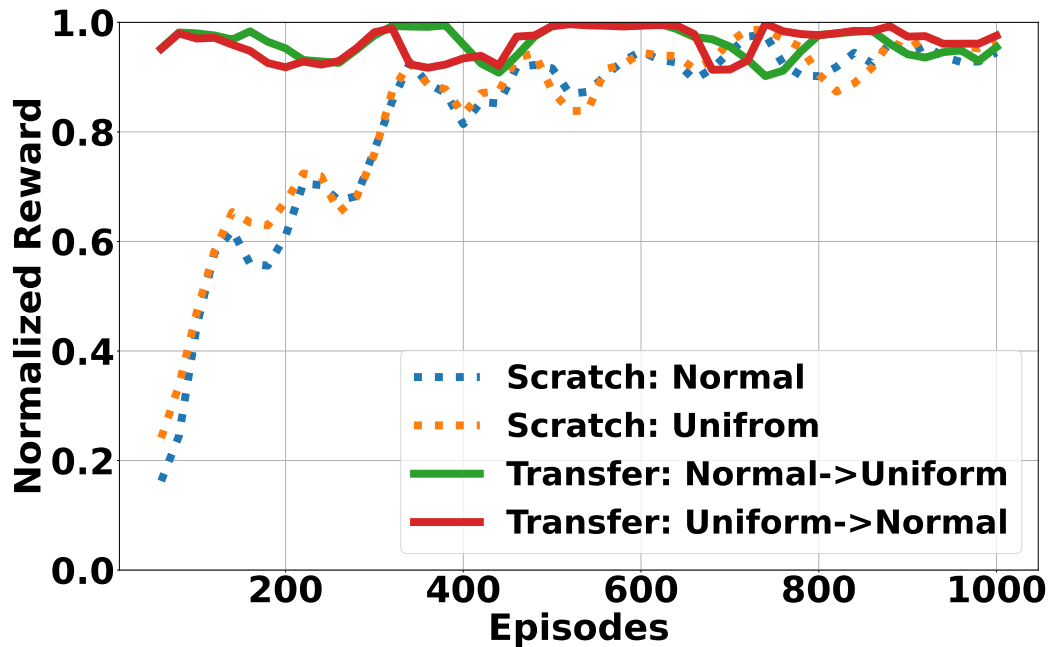


Figure 4.13: The dotted lines represent the training of **five** selected models with varying patterns in each episode, starting from scratch. Conversely, the solid lines depict transfer learning from models with opposite patterns.

then integrates a model (2: YoloX) with similar size and latency from the original set. Unlike training from scratch on three models, indicated by dashed lines, transfer learning results in convergence at early episodes. Notably, the transfer learning outcome of scenario (2) exhibits less fluctuation compared to scenario (1) (scenario (1) dips to less than 0.9 normalized rewards around 300 and 700 episodes). We suggest that this discrepancy is attributed to the relative challenge for the RL agent in accommodating unseen model sizes for which the initial weights are trained.

The change of Real-world App Figure 4.15 illustrates the benefits of employing transfer learning to adapt co-run workloads that involve real-world App. Initially, when the RL agent starts learning from scratch on model combinations with an uncontrollable workload (e.g., the "TikTok" App), its normalized reward typically converges at approximately 0.7. However, when the agent learns from an environment including an intricate workload (e.g., the "YouTube" App) and subsequently learns to adapt to another, it can achieve a convergence of normalized reward approximately 17% higher (reaching beyond 0.8) compared to starting from scratch. A similar outcome is observed when this process is reversed, underscoring the advantage of transfer learning in enabling the RL agent to better adapt to workloads with variance in

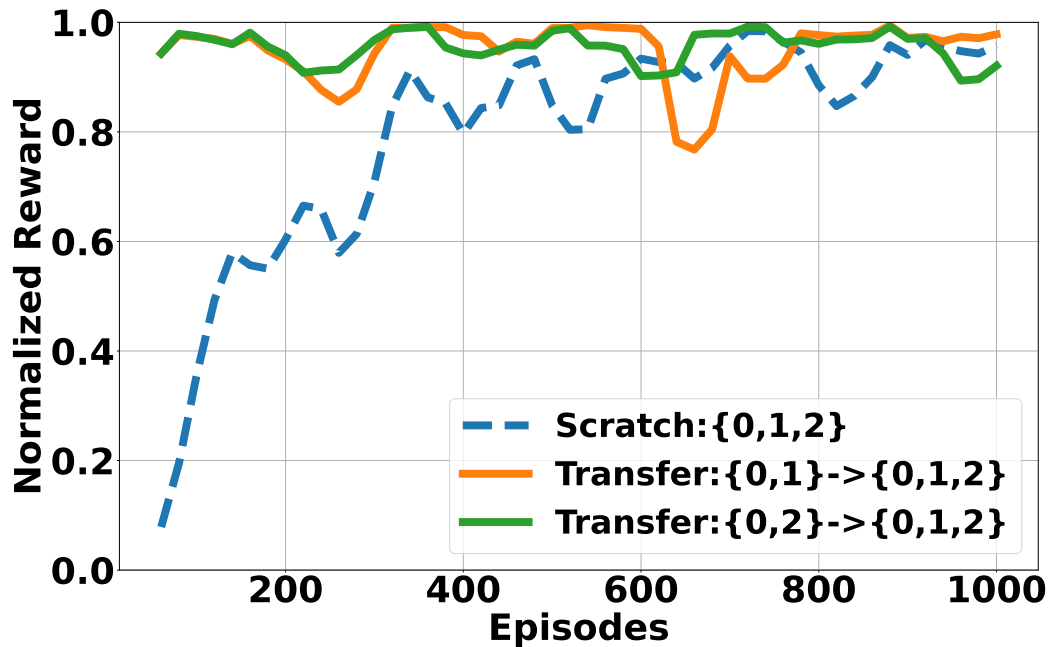


Figure 4.14: The dashed lines depict the training of **five** models, each representing one of the **three** types detailed in Table 4.1, starting anew in every episode. Conversely, the solid lines demonstrate transfer learning, utilizing trained weights from two distinct model types.

complexity.

4.8 Related Work

Multi-DNNs co-running on heterogeneous devices. Multiple agent reinforcement learning algorithms have shown significant accuracy improvement in domains like traffic control and video games. However, efficient deployment of multi-DNN co-scheduling on mobile devices has received less exploration.

One of the relevant studies is AutoScale (KW20), which leverages Q-table reinforcement learning to make informed decisions regarding model allocation across different device types, including CPU, GPU, Cloud, or remote devices during each inference cycle. However, an inherent limitation of AutoScale lies in the constrained capacity of its Q-table, rendering it unsuitable for handling scenarios with more than one DNN model running in a system—a scenario increasingly common in mobile devices.

To address the limitations associated with small state and action spaces, COSREL (XYY+21) employs Deep Q-learning reinforcement learning—the basis of the *centralized method* in our

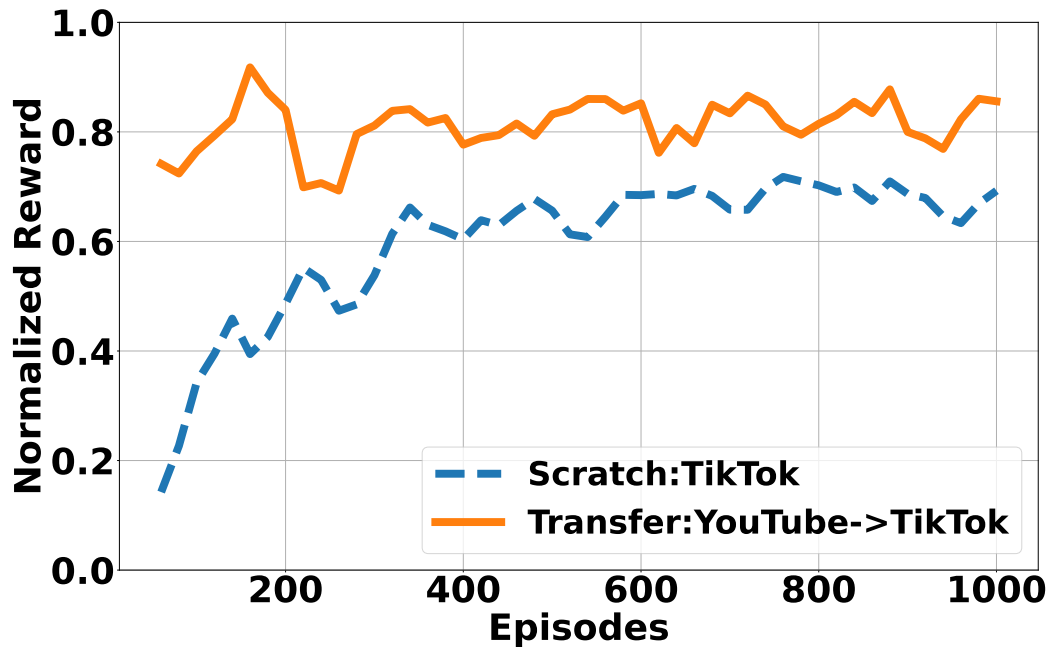


Figure 4.15: The dashed lines represent the selection of **five** models under Uniform distributions in each episode for co-scheduling multiple DNNs with the Real-world App, TikTok, starting from scratch. Conversely, the solid lines illustrate transfer learning from trained weights derived from the same setup but with the Real-world App, YouTube.

evaluation. This approach predicts the allocation of multiple models to various devices during discrete time slots, ensuring that it adheres to the principles of the Markov decision process in the context of multiple DNNs. However, COSREL faces a notable drawback in the form of training overhead. As the number of models and hardware components increases, the state and action space grow exponentially, necessitating increasingly complex and resource-intensive reinforcement learning models. Another issue of COSREL is that it only picks one model from multiple DNNs in each step to reduce the action space. This approach inadvertently results in significant underutilization of hardware resources, a challenge that needs to be addressed for optimal efficiency.

Recently, Sung and others (SCN⁺23) have introduced decentralized reinforcement learning agents for managing the decision-making processes related to DNNs on heterogeneous hardware within mobile devices. It is the basis of the *decentralized method* in our evaluation. While their empirical findings suggest that these agents operate without communication or cooperation, yielding satisfactory results, a fundamental drawback of this method is its lack of convergence guarantees and, hence, stable performance as demonstrated in our evaluations. **Multi-agent deep reinforcement learning.** Cooperative multi-agent deep reinforcement learn-

ing is an active research topic in AI. The Independent Q-Learning (IQL) algorithm (Tan93; LR00; MLLFP07; TMK⁺17; OPA⁺17; FIMY18) encapsulates a scenario where each agent operates independently without considering its influence on other agents. It treats fellow agents as components of the environment. While this approach has been useful in certain contexts, it is limited by non-stationarity in the environment and is particularly effective only in smaller-scale problems.

More closely related to our proposed solutions, the Fully Observable Critic approach (WEH20; FFA⁺18; RSP18; SWP⁺19; CY17; YNI⁺20; JDL18; IS19; YLL⁺18; KMH⁺19; LWT⁺17; MZXG19) addresses the non-stationarity challenges of the IQL method by leveraging a global state observed by all critic models. Discussions on the convergence properties of that approach are however limited and still need further investigations.

The method adopted in this work, value Function Factorization or Value-Decomposition (VD) (RSDW⁺20; SLG⁺18; MJM⁺18; SKK⁺19), assigns specific portions of the global reward to each individual agent. They have shown advantages in mitigating non-stationarity and preventing agent idleness.

Consensus (ZYL⁺18; KMP13; LYH18; MCZS14; MTH⁺17; CYS20; ZYB18; ZZ19) and Learning to Communicate (VKR09; PYW⁺17; FADFW16; SF⁺16; SJS18; LPB17; DKM⁺17) involves inter-agent communication to facilitate information exchange and tackle non-stationarity. However, these approaches are not well-suited to our specific context, primarily due to the high cost or impractical nature of agent communication within the framework of our research.

4.9 Conclusion

This dissertation has presented COSMA, a novel solution to the EDOSP in mobile AI. Integrating the CTDE approach, COSMA addresses the co-optimization of DNNs and scheduling with a focus on efficiency and convergence. The dissertation presents how EDOSP can be formulated into a Dec-POMDP process, making it amenable for CTDE methods to solve. It describes a set of techniques to materialize CTDE effectively into a complete solution to EDOSP. In a set of extensive evaluations, COSMA has consistently demonstrated superior performance over traditional centralized and decentralized methods, improving the rewards by $1.4\times$ to $2\times$ across various scenarios. The adaptability of COSMA in co-running scenarios with real-world applications, coupled with its cross-hardware portability, makes it an appealing option for addressing the co-optimization of DNN models and their runtime schedules on mobile devices.

CHAPTER

5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

This dissertation has explored and contributed to the domain of runtime optimization for multi-tenant Deep Neural Network (DNN) executions on heterogeneous mobile devices, offering innovative solutions with significant implications the industry.

Chapter 3 presents a foundational approach for enhancing autonomous driving technologies by implementing cost-effective solutions, including just-in-time affinity and priority adjustment, model migrations, DAG instantiation-based scheduling, and hardware-aware model customization. These strategies have proven to drastically reduce both cost and power consumption in autonomous driving systems, urging a reevaluation of prevailing assumptions about necessary computational resources and fostering a new research opportunities.

Chapter 4 shows a pioneering decentralized application-level adaptive scheduler for multi-instance DNNs on open mobile devices. Utilizing a Deep Q-Network (DQN) reinforcement learning algorithm with decentralized strategy, this scheduler autonomously adapts to dynamic environments without requiring explicit knowledge of other concurrently running applications. This innovation demonstrated rapid convergence and effectiveness in optimizing co-run efficiency, setting a new benchmark for decentralized scheduling approaches across diverse mobile systems.

Chapter 5 presents COSMA, an advanced solution to the Execution Dependency Optimization Problem (EDOSP) in mobile AI contexts. By leveraging the Centralized Training with Decentralized Execution (CTDE) framework and reformulating EDOSP into a Decentralized Partially Observable Markov Decision Process (Dec-POMDP), COSMA outperforms existing centralized and decentralized methods. The robustness of COSMA, evidenced by its ability to improve rewards by up to two-fold in varied scenarios, underlines its utility in co-optimizing DNN models and their runtime scheduling, making it a highly adaptable solution across different hardware environments.

In conclusion, the contributions of this dissertation are manifold and groundbreaking, significantly advancing the field of mobile AI by providing scalable, efficient, and cost-effective solutions. The frameworks and methodologies developed here not only enhance the performance and feasibility of autonomous systems and mobile AI applications but also open up new pathways for future research and development in runtime optimization and scheduling of DNNs on other applications on other heterogeneous devices. These advancements may lead to changes to the landscape of mobile computing, setting the stage for broader adoption and more innovative applications in the field.

5.2 Future Work

Building upon the findings and methodologies presented in this dissertation, there are several avenues for future research that can further refine and expand the scope of runtime optimization for multi-tenant DNN executions on heterogeneous mobile devices:

Enhanced Learning Algorithms for Scheduling: Future research could explore the integration of more advanced machine learning algorithms beyond DQN, such as Proximal Policy Optimization (PPO) or Asynchronous Advantage Actor-Critic (A3C), to improve the adaptability and efficiency of decentralized scheduling systems. These algorithms could offer faster convergence and better scalability, enhancing the system's performance in more complex and dynamic environments.

Cross-Platform Optimization Techniques: While this dissertation has addressed optimization across various mobile devices, extending these techniques to include a broader range of platforms, including edge and cloud computing environments, could provide a more holistic approach to DNN execution optimization. Research could direct its attention towards exploring additional DNN model structures, such as LLM, on another computing architectures, such as microchip controllers or Nvidia Tegra SoCs.

Energy Efficiency Metrics and Tools: There is a need for developing more sophisticated metrics and diagnostic tools to measure and optimize energy consumption in real-time DNN

applications. Future studies could focus on creating frameworks that not only assess the computational efficiency but also the energy footprint of algorithms, particularly in the context of autonomous driving and mobile AI.

Real-World Deployment and Testing: The practical application of the scheduling and optimization techniques developed in this dissertation should be tested in real-world scenarios to evaluate their practical viability and performance. Pilot studies or partnerships with industry stakeholders can provide valuable insights and feedback for further refinement.

By addressing these areas, future research can build on the current dissertation to provide more robust, efficient, and adaptive solutions for runtime optimization in heterogeneous computing environments, ultimately pushing the boundaries of what mobile and autonomous systems can achieve.

REFERENCES

- [AA20] Autoware-AI. Rosbag demo · autoware-ai/autoware.ai wiki, Jun 2020.
- [AAB⁺15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. <https://tensorflow.org/>, 2015.
- [amp23] Nvidia ampere architecture in-depth, Mar 2023.
- [Aut18] Apollo Auto. Apollo auto. <https://www.apollo.auto/>, Accessed: 2024-01-18.
- [bai21] Baidu apollo team (2017), apollo: Open source autonomous driving, July 2021.
- [Bel57] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.
- [BGIZ02] Daniel S Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The complexity of decentralized control of markov decision processes. *Mathematics of operations research*, 27(4):819–840, 2002.
- [BL18] Soroush Bateni and Cong Liu. Apnet: Approximation-aware real-time neural network. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 67–79. IEEE, 2018.
- [BL19] Soroush Bateni and Cong Liu. Predictable data-driven resource management: an implementation using autoware on autonomous platforms. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 339–352. IEEE, 2019.
- [BL20] Soroush Bateni and Cong Liu. Neuos: A latency-predictable multi-dimensional optimization framework for dnn-driven autonomous systems. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 371–385, 2020.
- [Bow00] Michael Bowling. Convergence problems of general-sum multiagent reinforcement learning. In *ICML*, pages 89–94, 2000.
- [BPX⁺20] Maxim Berman, Leonid Pishchulin, Ning Xu, Matthew B Blaschko, and Gérard Medioni. Aows: Adaptive and optimal network width search with latency constraints. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11217–11226, 2020.

- [BS03] Peter Biber and Wolfgang Strasser. The normal distributions transform: A new approach to laser scan matching. In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*(Cat. No. 03CH37453), volume 3, pages 2743–2748. IEEE, 2003.
- [BWL21] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov5. <https://github.com/ultralytics/yolov5>, 2021.
- [BZZL18] Soroush Bateni, Husheng Zhou, Yuankun Zhu, and Cong Liu. Predjoule: A timing-predictable energy optimization framework for deep neural networks. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 107–118. IEEE, 2018.
- [CD06] Xi Chen and Xiaotie Deng. Settling the complexity of two-player nash equilibrium. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 261–272. IEEE Computer Society, 2006.
- [CGCB14] Junyoung Chung, Çağlar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.
- [CGZH20] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. Tinytl: Reduce memory, not parameters for efficient on-device learning. In Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.
- [Cor23] NVIDIA Corporation. Nvidia h100 tensor core gpu architecture, 2023. Accessed: 2023-11-28.
- [Cou92] R Craig Coulter. Implementation of the pure pursuit path tracking algorithm. Technical report, Carnegie-Mellon UNIV Pittsburgh PA Robotics INST, 1992.
- [CR20] Yujeong Choi and Minsoo Rhu. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 220–233. IEEE, 2020.
- [CY17] Xiangxiang Chu and Hangjun Ye. Parameter sharing deep deterministic policy gradient for cooperative multi-agent reinforcement learning. *CoRR*, abs/1710.00336, 2017.
- [CYS20] Lucas Cassano, Kun Yuan, and Ali H Sayed. Multiagent fully decentralized value function learning with linear convergence rates. *IEEE Transactions on Automatic Control*, 66(4):1497–1512, 2020.

- [CYS⁺24] Minseok Choi, Won Joon Yun, Seok Bin Son, Soohyun Park, and Joongheon Kim. Joint delay-sensitive and power-efficient quality control of dynamic video streaming using adaptive super-resolution. *IEEE Trans. Green Commun. Netw.*, 8(1):103–117, 2024.
- [Dev23] Android Developers. Android neural networks api. <https://developer.android.com/ndk/guides/neuralnetworks/index.html>, Accessed: January 10, 2023.
- [DKM⁺17] Abhishek Das, Satwik Kottur, José MF Moura, Stefan Lee, and Dhruv Batra. Learning cooperative visual dialog agents with deep reinforcement learning. In *Proceedings of the IEEE international conference on computer vision*, pages 2951–2960, 2017.
- [DTMD10] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, and James Diebel. Path planning for autonomous vehicles in unknown semistructured environments. *The international journal of robotics research*, 29(5):485--501, 2010.
- [EVGW⁺] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. <http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html>.
- [FADFW16] Jakob Foerster, Ioannis Alexandros Assael, Nando De Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. *Advances in neural information processing systems*, 29, 2016.
- [FFA⁺18] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.
- [FIMY18] Taiki Fuji, Kiyoto Ito, Kohsei Matsumoto, and Kazuo Yano. Deep multi-agent reinforcement learning using dnn-weight evolution to optimize supply chain performance. 2018.
- [Fou18] Autoware Foundation. Why autoware. <https://autoware.org/about/why-autoware/>, Accessed: 2024-01-18.
- [Fri01] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.
- [FZZ18] Biyi Fang, Xiao Zeng, and Mi Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 115–127, 2018.
- [GCS⁺19] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 485–500, 2019.

- [GLW⁺21] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. YOLOX: exceeding YOLO series in 2021. *CoRR*, abs/2107.08430, 2021.
- [Goo] Google. Google tensor g3 for pixel 8. <https://blog.google/products/pixel/google-tensor-g3-pixel-8/>.
- [Goo18a] Google. Ai in pixel. <https://store.google.com/intl/en/ideas/articles/ai-in-pixel/>, Accessed: 2024-01-18.
- [Goo18b] Google. Pixel feature drop - december 2023. <https://store.google.com/intl/en/ideas/articles/pixel-feature-drop-december-2023/>, Accessed: 2024-01-18.
- [GZL⁺22] Jiawei Guan, Feng Zhang, Jiesong Liu, Hsin-Hsuan Sung, Ruofan Wu, Xiaoyong Du, and Xipeng Shen. Trec: Transient redundancy elimination-based convolution. *Advances in Neural Information Processing Systems*, 35:26578–26589, 2022.
- [HCL⁺18] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Q. Weinberger. Multi-scale dense networks for resource efficient image classification. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.
- [HH20] Malik Haris and Jin Hou. Obstacle detection and safely navigate the autonomous vehicle from unexpected obstacles on the driving lane. *Sensors*, 20(17):4719, 2020.
- [HMD15] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.
- [HNR68] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100--107, 1968.
- [HW⁺98] Junling Hu, Michael P Wellman, et al. Multiagent reinforcement learning: theoretical framework and an algorithm. In *ICML*, volume 98, pages 242–250, 1998.
- [HWF⁺20] Zhanchao Huang, Jianlin Wang, Xuesong Fu, Tao Yu, Yongqi Guo, and Rutong Wang. Dc-spp-yolo: Dense connection and spatial pyramid pooling based yolo for object detection. *Information Sciences*, 522:241–258, 2020.
- [HZC⁺17] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.
- [IS19] Shariq Iqbal and Fei Sha. Actor-attention-critic for multi-agent reinforcement learning. In *International conference on machine learning*, pages 2961–2970. PMLR, 2019.

- [JDL18] Jiechuan Jiang, Chen Dun, and Zongqing Lu. Graph convolutional reinforcement learning for multi-agent cooperation. *CoRR*, abs/1810.09202, 2018.
- [jet21] 2021.
- [JLK⁺22] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. Band: coordinated multi-dnn inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 235–247, 2022.
- [KCK⁺21] Hakpyeong Kim, Heeju Choi, Hyuna Kang, Jongbaek An, Seungkeun Yeom, and Taehoon Hong. A systematic review of the smart energy conservation system: From smart homes to sustainable smart cities. *Renewable and sustainable energy reviews*, 140:110755, 2021.
- [KKKK19] Dohyun Kim, Joongheon Kim, Junseok Kwon, and Tae-Hyung Kim. Depth-controllable very deep super-resolution network. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2019.
- [KMH⁺19] Daewoo Kim, Sangwoo Moon, David Hostallero, Wan Ju Kang, Taeyoung Lee, Kyunghwan Son, and Yung Yi. Learning to schedule communication in multi-agent reinforcement learning. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.
- [KMP13] Soumya Kar, José M. F. Moura, and H. Vincent Poor. Qd-learning: A collaborative distributed strategy for multi-agent reinforcement learning through *Consensus + Innovations*. *IEEE Trans. Signal Process.*, 61(7):1848–1862, 2013.
- [kol] Movenet: Ultra fast and accurate pose detection model. <https://www.tensorflow.org/hub/tutorials/movenet>. Accessed: 2023-09-30.
- [KSSG20] Ievgeniia Kuzminykh, Dan Shevchuk, Stavros Shiaeles, and Bogdan Ghita. Audio interval retrieval using convolutional neural networks. In *Internet of Things, Smart Spaces, and Next Generation Networks and Systems: 20th International Conference, NEW2AN 2020, and 13th Conference, ruSMART 2020, St. Petersburg, Russia, August 26–28, 2020, Proceedings, Part I 20*, pages 229–240. Springer, 2020.
- [KTM⁺18] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296. IEEE, 2018.
- [KW20] Young Geun Kim and Carole-Jean Wu. Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1082–1096. IEEE, 2020.

- [LG20] Shaoshan Liu and Jean-Luc Gaudiot. Autonomous vehicles lite self-driving technologies should start small, go slow. *IEEE Spectrum*, 57(3):36–49, 2020.
- [LHL⁺21] Quyuan Luo, Shihong Hu, Changle Li, Guanghui Li, and Weisong Shi. Resource scheduling in edge computing: A survey. *IEEE Communications Surveys & Tutorials*, 23(4):2131–2165, 2021.
- [Lin92] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8:293–321, 1992.
- [Liu20] Shaoshan Liu. Critical business decision making for technology startups: A perceptin case study. *IEEE Engineering Management Review*, 48(4):32–36, 2020.
- [LLT⁺19] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.
- [LLT⁺20] Shaoshan Liu, Liyun Li, Jie Tang, Shuang Wu, and Jean-Luc Gaudiot. Creating autonomous vehicle systems. *Synthesis Lectures on Computer Science*, 8(2):i–216, 2020.
- [LPB17] Angeliki Lazaridou, Alexander Peysakhovich, and Marco Baroni. Multi-agent cooperation and the emergence of (natural) language. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.
- [LR00] Martin Lauer and Martin A Riedmiller. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *Proceedings of the seventeenth international conference on machine learning*, pages 535–542, 2000.
- [LSCL20] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. Allox: compute allocation in hybrid clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.
- [LTL⁺21] Liu Liu, Jie Tang, Shaoshan Liu, Bo Yu, Yuan Xie, and Jean-Luc Gaudiot. π -rt: A runtime framework to enable energy-efficient real-time robotic vision applications on heterogeneous architectures. *Computer*, 54(4):14–25, 2021.
- [LTZG17] Shaoshan Liu, Jie Tang, Zhe Zhang, and Jean-Luc Gaudiot. Computer architectures for autonomous driving. *Computer*, 50(8):18–25, 2017.
- [LVC⁺19] Alex H Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 12697–12705, 2019.
- [LWT⁺17] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in neural information processing systems*, 30, 2017.

- [LWW⁺21] Changlin Li, Guangrun Wang, Bing Wang, Xiaodan Liang, Zhihui Li, and Xiaojun Chang. Dynamic slimmable network. In *Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition*, pages 8607–8617, 2021.
- [LYH18] Donghwan Lee, Hyungjin Yoon, and Naira Hovakimyan. Primal-dual algorithm for distributed reinforcement learning: Distributed gtd. In *2018 IEEE Conference on Decision and Control (CDC)*, pages 1967–1972. IEEE, 2018.
- [LYL⁺21] Shaoshan Liu, Bo Yu, Yahui Liu, Kunai Zhang, Yisong Qiao, Thomas Yuang Li, Jie Tang, and Yuhao Zhu. Brief industry presentation: The matter of time—a general and efficient system for precise sensor synchronization in robotic computing. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021.
- [LZC⁺22] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. On-device training under 256kb memory. *Advances in Neural Information Processing Systems*, 35:22941–22954, 2022.
- [MBS⁺20] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient {GPU} cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.
- [MCZS14] Sergio Valcarcel Macua, Jianshu Chen, Santiago Zazo, and Ali H Sayed. Distributed policy evaluation under multiple behavior strategies. *IEEE Transactions on Automatic Control*, 60(5):1260–1274, 2014.
- [MHY⁺21] Pavlo Molchanov, Jimmy Hall, Hongxu Yin, Jan Kautz, Nicolò Fusi, and Arash Vahdat. HANT: hardware-aware network transformation. *CoRR*, abs/2107.10624, 2021.
- [MJM⁺18] David Mguni, Joel Jennings, Sergio Valcarcel Macua, Sofia Ceppi, and Enrique Munoz de Cote. Inducing efficient equilibria in multi-agent systems. 2018.
- [MK07] Geoffrey J McLachlan and Thriyambakam Krishnan. *The EM algorithm and extensions*, volume 382. John Wiley & Sons, 2007.
- [MKS⁺15] 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.
- [MLLFP07] Laëtitia Matignon, Guillaume J Laurent, and Nadine Le Fort-Piat. Hysteretic q-learning: an algorithm for decentralized reinforcement learning in cooperative multi-agent teams. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 64–69. IEEE, 2007.

- [MTH⁺17] Sergio Valcarcel Macua, Aleksi Tukiainen, Daniel García-Ocaña Hernández, David Baldazo, Enrique Munoz de Cote, and Santiago Zazo. Diff-dac: Distributed actor-critic for multitask deep reinforcement learning. *CoRR*, abs/1710.10363, 2017.
- [MZYG19] Hangyu Mao, Zhengchao Zhang, Zhen Xiao, and Zhibo Gong. Modelling the dynamic joint policy of teammates with attention multi-agent ddpg. *AAMAS '19*, page 1108–1116, Richland, SC, 2019. International Foundation for Autonomous Agents and Multiagent Systems.
- [NML⁺20] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 907–922, 2020.
- [NSK⁺20] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhemiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 481–498, 2020.
- [NSL⁺21] Wei Niu, Mengshu Sun, Zhengang Li, Jou-An Chen, Jiexiong Guan, Xipeng Shen, Yanzhi Wang, Sijia Liu, Xue Lin, and Bin Ren. RT3D: Achieving real-time execution of 3d convolutional neural networks on mobile devices. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35:10, pages 9179–9187, 2021.
- [NVI] NVIDIA. Deep learning accelerator. <https://developer.nvidia.com/deep-learning-accelerator>. Accessed: 2024-01-20.
- [nvi21] Nvidia drive - autonomous vehicle development platforms, May 2021.
- [OPA⁺17] Shayegan Omidshafiei, Jason Pazis, Christopher Amato, Jonathan P How, and John Vian. Deep decentralized multi-task multi-agent reinforcement learning under partial observability. In *International Conference on Machine Learning*, pages 2681–2690. PMLR, 2017.
- [PMJH18] Santiago Pagani, PD Sai Manoj, Axel Jantsch, and Jörg Henkel. Machine learning for power, energy, and thermal management on multicore processors: A survey. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(1):101–116, 2018.
- [PYW⁺17] Peng Peng, Quan Yuan, Ying Wen, Yaodong Yang, Zhenkun Tang, Haitao Long, and Jun Wang. Multiagent bidirectionally-coordinated nets for learning to play starcraft combat games. *CoRR*, abs/1703.10069, 2017.
- [Quaa] Qualcomm. Autonomous driving. <https://www.qualcomm.com/products/automotive/autonomous-driving>. Accessed: 2024-01-21.
- [Quab] Qualcomm. Hexagon dsp sdk. <https://developer.qualcomm.com/software/hexagon-dsp-sdk/dsp-processor>. Accessed: 2024-01-19.

- [Qua18] Qualcomm. Mobile ai in snapdragon smartphones. <https://www.qualcomm.com/products/mobile/snapdragon/smartphones/mobile-ai>, Accessed: 2024-01-18.
- [RF18] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.
- [RKA⁺19] Muhammad Ramzan, Hikmat Ullah Khan, Shahid Mahmood Awan, Amina Ismail, Mahwish Ilyas, and Ahsan Mahmood. A survey on state-of-the-art drowsiness detection techniques. *IEEE Access*, 7:61904–61919, 2019.
- [RSDW⁺20] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Monotonic value function factorisation for deep multi-agent reinforcement learning. *The Journal of Machine Learning Research*, 21(1):7234–7284, 2020.
- [RSP18] Heechang Ryu, Hayong Shin, and Jinkyoo Park. Multi-agent actor-critic with generative cooperative policy network. *CoRR*, abs/1810.09206, 2018.
- [Sam18] Samsung. Enter the new era of mobile ai with samsung galaxy s24 series. <https://news.samsung.com/global/enter-the-new-era-of-mobile-ai-with-samsung-galaxy-s24-series>, Accessed: 2024-01-18.
- [Sch17] Matthias Schreier. Bayesian environment representation, prediction, and criticality assessment for driver assistance systems. *at-Automatisierungstechnik*, 65(2):151--152, 2017.
- [sch21] 2021.
- [SCN⁺23] Hsin-Hsuan Sung, Jou-An Chen, Wei Niu, Jiexiong Guan, Bin Ren, and Xipeng Shen. Decentralized application-level adaptive scheduling for multi-instance dnns on open mobile devices. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 865–877, 2023.
- [SDR20] Benjamin K Sovacool and Dylan D Furszyfer Del Rio. Smart home technologies in europe: A critical review of concepts, benefits, risks and policies. *Renewable and sustainable energy reviews*, 120:109663, 2020.
- [SF⁺16] Sainbayar Sukhbaatar, Rob Fergus, et al. Learning multiagent communication with backpropagation. *Advances in neural information processing systems*, 29, 2016.
- [SHZ⁺18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

- [SJS18] Amanpreet Singh, Tushar Jain, and Sainbayar Sukhbaatar. Learning when to communicate at scale in multiagent cooperative and competitive tasks. *CoRR*, abs/1812.09755, 2018.
- [SKK⁺19] Kyunghwan Son, Daewoo Kim, Wan Ju Kang, David Earl Hostallero, and Yung Yi. Qtran: Learning to factorize with transformation for cooperative multi-agent reinforcement learning. In *International conference on machine learning*, pages 5887–5896. PMLR, 2019.
- [SLG⁺18] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z. Leibo, Karl Tuyls, and Thore Graepel. Value-decomposition networks for cooperative multi-agent learning based on team reward. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '18*, page 2085–2087, Richland, SC, 2018. International Foundation for Autonomous Agents and Multiagent Systems.
- [SSA⁺18] Yukihiro Saito, Futoshi Sato, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. Rosch: Real-time scheduling framework for ros. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 52–58. IEEE, 2018.
- [ST17] Biljana L Risteska Stojkoska and Kire V Trivodaliev. A review of internet of things for smart home: Challenges and solutions. *Journal of cleaner production*, 140:1454–1464, 2017.
- [SWP⁺19] Guillaume Sartoretti, Yue Wu, William Paivine, TK Satish Kumar, Sven Koenig, and Howie Choset. Distributed reinforcement learning for multi-robot decentralized collective construction. In *Distributed Autonomous Robotic Systems: The 14th International Symposium*, pages 35–49. Springer, 2019.
- [SXG⁺22] Hsin-Hsuan Sung, Yuanchao Xu, Jiexiong Guan, Wei Niu, Bin Ren, Yanzhi Wang, Shaoshan Liu, and Xipeng Shen. Brief industry paper: Enabling level-4 autonomous driving on a single \$1k off-the-shelf card. In *28th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2022, Milano, Italy, May 4-6, 2022*, pages 297–300. IEEE, 2022.
- [Tan93] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.
- [Tan22] Freedom Tan. Some super resolution tflite models, 2022. GitHub repository.
- [ten19] 2019.
- [ten21] 2021.
- [Tes18] Tesla. Autopilot. <https://www.tesla.com/autopilot>, Accessed: 2024-01-18.

- [tfh22] Tensorflow hub, 2022. Platform for hosting and serving machine learning models.
- [TH12] Tijmen Tieleman and Geoffrey Hinton. Rmsprop: Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning. *COURSERA Neural Networks Mach. Learn*, 17, 2012.
- [the15] 2015.
- [THW02] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.
- [TL19] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.
- [TLL⁺20] Jie Tang, Shaoshan Liu, Liangkai Liu, Bo Yu, and Weisong Shi. Lopecs: A low-power edge computing system for real-time autonomous driving services. *IEEE Access*, 8:30467–30479, 2020.
- [TMK⁺17] Ardi Tampuu, Tabet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. Multiagent cooperation and competition with deep reinforcement learning. *PLoS one*, 12(4):e0172395, 2017.
- [tru21] Mobileye true redundancy, July 2021.
- [TSV⁺20] Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti, et al. Compute solution for tesla’s full self-driving computer. *IEEE Micro*, 40(2):25–35, 2020.
- [TYLG20] Jie Tang, Rao Yu, Shaoshan Liu, and Jean-Luc Gaudiot. A container based edge offloading framework for autonomous driving. *IEEE Access*, 8:33713–33726, 2020.
- [VDW19] Toan H Vu, An Dang, and Jia-Ching Wang. A deep neural network for real-time driver drowsiness detection. *IEICE TRANSACTIONS on Information and Systems*, 102(12):2637–2641, 2019.
- [VHGS16] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.
- [VKR09] Paulina Varshavskaya, Leslie Pack Kaelbling, and Daniela Rus. Efficient distributed reinforcement learning through agreement. *Distributed Autonomous Robotic Systems 8*, pages 367–378, 2009.
- [VL21] Ronny Votel and Na Li. Next-generation pose detection with movenet and tensorflow.js. *TensorFlow Blog*, 4, 2021.

- [Way18] Waymo. Waymo driver. <https://waymo.com/waymo-driver/>, Accessed: 2024-01-18.
- [way21] Waymo driver, July 2021.
- [WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.
- [WEH20] Rose E. Wang, Michael Everett, and Jonathan P. How. R-MADDPG for partially observable environments and limited communication. *CoRR*, abs/2002.06684, 2020.
- [WLL⁺19] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8612–8620, 2019.
- [WYW⁺18] Xintao Wang, Ke Yu, Shixiang Wu, Jinjin Gu, Yihao Liu, Chao Dong, Yu Qiao, and Chen Change Loy. Esrgan: Enhanced super-resolution generative adversarial networks. In *Proceedings of the European conference on computer vision (ECCV) workshops*, pages 0–0, 2018.
- [Xav21] 2021.
- [XBR⁺18] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.
- [XBSV21] Yuanchao Xu, Mehmet Esat Belviranli, Xipeng Shen, and Jeffrey Vetter. Pccs: Processor-centric contention-aware slowdown model for heterogeneous system-on-chips. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1282–1295, 2021.
- [XWCL15] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *CoRR*, abs/1505.00853, 2015.
- [XY⁺21] Zhiyuan Xu, Dejun Yang, Chengxiang Yin, Jian Tang, Yanzhi Wang, and Guoliang Xue. A co-scheduling framework for dnn models on mobile and edge devices with heterogeneous hardware. *IEEE Transactions on Mobile Computing*, 2021.
- [YHX⁺20] Bo Yu, Wei Hu, Leimeng Xu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1067–1081. IEEE, 2020.
- [YLL⁺18] Yaodong Yang, Rui Luo, Minne Li, Ming Zhou, Weinan Zhang, and Jun Wang. Mean field multi-agent reinforcement learning. In *International conference on machine learning*, pages 5571–5580. PMLR, 2018.

- [YNI⁺20] Jiachen Yang, Alireza Nakhaei, David Isele, Kikuo Fujimura, and Hongyuan Zha. CM3: cooperative multi-goal multi-stage multi-agent reinforcement learning. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.
- [YSKL18] Sangdoon Yun, Seokjun Seo, Jiwon Kim, and Kyoung Mu Lee. Wide activation for efficient and accurate image super-resolution. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 557–572, 2018.
- [YWS⁺22] Fuxun Yu, Di Wang, Longfei Shangguan, Minjia Zhang, Chenchen Liu, and Xi-ang Chen. A survey of multi-tenant deep learning inference on GPU. *CoRR*, abs/2203.09040, 2022.
- [YYX⁺19] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas S. Huang. Slimmable neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.
- [ZHZ⁺19] Zhi Zhang, Tong He, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. Bag of freebies for training object detection neural networks. *CoRR*, abs/1902.04103, 2019.
- [ZYB18] Kaiqing Zhang, Zhuoran Yang, and Tamer Basar. Networked multi-agent reinforcement learning in continuous spaces. In *2018 IEEE conference on decision and control (CDC)*, pages 2771–2776. IEEE, 2018.
- [ZYL⁺18] Kaiqing Zhang, Zhuoran Yang, Han Liu, Tong Zhang, and Tamer Basar. Fully decentralized multi-agent reinforcement learning with networked agents. In *International Conference on Machine Learning*, pages 5872–5881. PMLR, 2018.
- [ZZ19] Yan Zhang and Michael M Zavlanos. Distributed off-policy actor-critic reinforcement learning with policy consensus. In *2019 IEEE 58th Conference on decision and control (CDC)*, pages 4674–4679. IEEE, 2019.