Newcastle University

School of Computing Science

Enabling Efficient ML-based IoT Applications: Edge-Cloud Collaboration for Deployment, Updates, and Optimization

Bin Qian

Submitted for the degree of Doctor of Philosophy in the School of Computing Science, Newcastle University, UK

Jul 2023

Abstract

The rapid integration of the physical world with the Internet through the IoT has led to a massive network of connected devices. ML has emerged as a crucial technology for processing and analyzing the vast amounts of diverse data generated by this IoT network, enabling intelligent IoT applications. The combination of ML and IoT has seen significant growth, enabling innovative use cases and leveraging cloud computing for data analysis and pattern extraction.

ML-based IoT applications face challenges in effectively analyzing the vast amounts of data generated by diverse IoT devices. Transferring data to centralized cloud centers can be inefficient for timely analysis, and cloud computing may not always be suitable for emerging IoT applications. To overcome these challenges, a federated approach that combines cloud and edge resources is crucial. Edge computing brings computing operations to resource-constrained edge devices, enabling real-time responses and reducing data transmission to the cloud. Collaborating edge and cloud resources in a federated system allows for efficient data processing at the edge and complex ML algorithms in the cloud. This collaboration improves response times, reduces latency, enhances scalability, and optimizes resource utilization in ML-based IoT applications.

An important challenge in edge-cloud collaboration is aggregating microservices in a way that meets application requirements such as latency, throughput, energy consumption, and model prediction accuracy. The edge-cloud collaboration aims to characterize Quality of Service (QoS) metrics based on microservice composition plans and adapt them to deployment sites, considering contextual factors and deployment locations. Furthermore, these plans need to be adaptable to fluctuations in computing environments throughout the application's execution. A feedback-driven orchestration mechanism is necessary to detect changes in infrastructure performance and QoS metrics.

In the edge-cloud computing paradigm, an additional challenge is the inconsistent model prediction performance observed in distributed environments. Models are con-

figured differently to accommodate resource constraints, leading to heterogeneity in model architectures and configurations. This heterogeneity can result in different outputs from models when provided with the same input, posing a systemic problem that hinders prediction agreement within the application. Currently, there is a need for a systematic design to efficiently detect and minimize model inconsistency in distributed deep learning applications.

To address the above mentioned challenges, the key contributions of this thesis are listed below:

- Designing ***OsmoticGate***, a video analytics task offloading framework that is capable of generating optimal workload balancing strategies, based on a Hierarchy Queue Model and a two-stage gradient-based algorithm.
- Based on the research outcomes in ***OsmoticGate***, implementing an online multi-agent reinforcement learning system named ***OsmoticGate2***, which involves a co-designed algorithm and system to achieve workload balancing in dynamic and distributed deep learning (DL) applications.
- Implementing ***DEEPCON***, an adaptive deployment framework to quickly detect and improve model consistency through over-the-air parallel training and online knowledge distillation that enables teacher-student learning among all deployed models.

Declaration

I declare that this thesis is my own work unless otherwise stated. No part of this thesis has previously been submitted for a degree or any other qualification at Newcastle University or any other institution.

Bin Qian

July 2023

Publications

Published

1. ***Bin Qian***, Yubo Xuan, Di Wu, Zhenyu Wen, Renyu Yang, Shibo He, Jiming Chen, and Rajiv Ranjan. "Edge-Cloud Collaborative Streaming Video Analytics with Multi-agent Deep Reinforcement Learning" *IEEE Networks Magazine* 2024.

2. **Bin Qian**, Zhenyu Wen, Junqi Tang, Ye Yuan, Albert Y. Zomaya, Rajiv Ranjan. "OSMOTICGATE: Adaptive Edge-based Real-time Video Analytics for the Internet of Things" *IEEE Transactions on Computers (TC)* 01 (2022): 1-14.

3. **Bin Qian**, Jie Su, Zhenyu Wen, Devki Nandan Jha, Yinhao Li, Yu Guan, Deepak Puthal et al. "Orchestrating the development lifecycle of machine learning-based IoT applications: A taxonomy and survey" *ACM Computing Surveys (CSUR)* 53, no. 4 (2020): 1-47.

4. Zhenyu Wen, Renyu Yang, **Bin Qian**, Yubo Xuan, Lingling Lu, Zheng Wang, Hao Peng, Jie Xu, Albert Y Zomaya, Rajiv Ranjan. "JANUS: Latency-Aware Traffic Scheduling for IoT Data Streaming in Edge Environments" *IEEE Transactions on Services Computing (TSC)* 53, no. 4 (2020): 1-47.

5. Jiaxu Qian, Zhenyu Wen, **Bin Qian**, Qin Yuan, Jianbin Qin, Qi Xuan, Ye Yuan. "Across Images and Graphs for Question Answering" *IEEE International Conference on Data Engineering (ICDE)* 2024.

6. Rudresh Dwivedi, Devam Dave, Het Naik, Smiti Singhal, Rana Omer, Pankesh Patel, **Bin Qian**, Zhenyu Wen, Tejal Shah, Graham Morgan, Rajiv Ranjan. "Explainable AI (XAI): Core ideas, techniques, and solutions" *ACM Computing Surveys (CSUR)* 55, no. 9 (2023) 1-33

7. Zhenyu Wen, Haozhen Hu, Renyu Yang, **Bin Qian**, Ringo WH Sham, Rui Sun, Jie Xu, Pankesh Patel, Omer Rana, Schahram Dustdar, Rajiv Ranjan. "Or-

chestrating Networked Machine Learning Applications Using Autosteer" *IEEE Internet Computing* 26, no. 6 (2022), 51-58

8. Shuyun Luo, Hang Li, Zhenyu Wen, **Bin Qian**, Graham Morgan, Antonella Longo, Omer Rana, and Rajiv Ranjan. "Blockchain-based task offloading in drone-aided mobile edge computing" *IEEE Network 35*, no. 1 (2021): 124-129.

9. Feng Lu, Wei Li, Song Lin, Chengwangli Peng, Zhiyong Wang, **Bin Qian**, Rajiv Ranjan, Hai Jin, and Albert Y. Zomaya. "Multi-scale Features Fusion for the Detection of Tiny Bleeding in Wireless Capsule Endoscopy Images" *ACM Transactions on Internet of Things* 3, no. 1 (2021): 1-19.

Under Review

1. **Bin Qian**, Jiaxu Qian, Zhenyu Wen, Yinhao Li, Shibo He, Jiming Chen, Albert Y. Zomaya and Rajiv Ranjan. "DeepCon: Improving Distributed Deep Learning Model Consistency in Edge-Cloud Environment via Distillation" (*IEEE Transactions on Computers (TC)* , under review)

2. **Bin Qian**, Yubo Xuan, Zhenyu Wen, and Rajiv Ranjan. "Edge-Cloud Collaborative Streaming Video Analytics on Open Environments" (In Submission)

Acknowledgements

I am privileged to be under the guidance and mentorship of Prof. Rajiv Ranjan, serving as my esteemed doctoral advisor. His consistent support and invaluable feedback have not just been in the academic area, but also in the well-being of my personal life. I also wish to extend my sincere appreciation to my co-supervisor, Prof. Zhenyu Wen, whose patient and diligent guidance has been instrumental in equipping me with the skills to conduct independent research as well as foster effective collaboration with others. The successful completion of this thesis owes a profound debt to the profound influence and support of both Prof. Rajiv Ranjan and Prof. Zhenyu Wen during my pursuit of the Ph.D. journey. Without their presence, this achievement would not have been possible.

I would like to thank my colleagues: Yinhao Li, Ayman Noor, Devki Nandan Jha, Jedsada Phengsuwan, Khaled Alwasel, Nipun Balan, Umit Demirbaga, Fawzy Habeeb, Deepak Puthal, Jie Su, Gagangeet Aujla, Xinyuan Liu, Chengming Zhou; I am grateful to many graduate students I met throughout the years at Newcastle University and Zhejiang University of Technology. I learned manything new from each of them. I would like to thank my co-authors Junqi Tang, Renyu Yang, Shuyun Luo, Di Wu, Yubo Xuan, and others for kindly sharing their valuable experiences in our work.

Finally, I would like to thank my parents for their patience and support over the years. In particular, I am very grateful to my girlfriend Yilin for her support, companion and motivation on the long journey.

Contents

List of Figures

List of Tables

1

Introduction

Contents

The rapid advancement of hardware, software, and communication technologies has significantly accelerated the integration of the physical world with the Internet through the Internet of Things (IoT). According to a report from Statista [1], it is projected that approximately 75.44 billion IoT devices will be connected to the Internet by 2025, indicating the tremendous scale of this interconnected network. These IoT devices generate an enormous volume of data with diverse modalities, presenting both opportunities and challenges. Effectively processing and analyzing this vast amount of data is crucial for the development of intelligent IoT applications. In this context, Machine Learning (ML) emerges as a pivotal technology that enables data intelligence, allowing us to gain insights and explore the complexities of the real world. The combination of ML and IoT-type applications is currently undergoing explosive growth, with numerous innovative use cases being developed, e.g., image processing, natural language processing, speech recognition, and other intelligent services [229]. This convergence leverages the power of ML algorithms to extract meaningful patterns and knowledge from IoT-generated data, with the support of cloud computing.

Cloud computing grants immediate access to a shared pool of computing resources, encompassing storage, applications, and services, accessible over the Internet. Cloud AI, on the other hand, offers scalable, cost-effective, and easily accessible artificial intelligence capabilities, empowering organizations to leverage robust computing resources, foster effective collaboration, and drive innovation without the need for significant infrastructure investments. Cloud AI research primarily focuses on enhancing model performance in various aspects such as generalization [37], robustness [258], fairness [200], and more. Through meticulous design choices, including model architectures, loss functions, and leveraging recent advancements in large language models [339], researchers aim to push the boundaries of AI performance within the realm of ample computing power and storage provided by cloud clusters.

Despite the immense capabilities of cloud AI, the exponential growth of the IoT presents a significant challenge: the unprecedented transfer of data from edge devices to data centers in large volumes. Additionally, cloud AI encounters significant obstacles in meeting the real-time latency and privacy requirements of applications

[1] https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

such as autonomous driving [152], real-time video analysis [230], and more. To address these concerns, the adoption of a decentralized computing paradigm called edge computing becomes imperative. Edge computing allows for distributed model training and inference closer to the data sources, effectively reducing transmission delays. This approach enables significant advancements in meeting real-time requirements and ensuring data privacy. In the field of edge computing, there is a pressing need to expand the frontiers of AI to encompass edge devices, thus unlocking their full potential in terms of performance and capabilities, even in resource-constrained environments. Designing lightweight model architectures, and on-device computation optimizations, are all promising research directions aiming at fast and energy-efficient edge computing.

ML-based IoT applications require collaboration between cloud computing and edge computing, considering their distinct characteristics, to achieve a comprehensive solution that encompasses both hardware and software aspects. This collaboration gives rise to a hierarchical computing architecture in which massive distributed end devices, edge servers, and central cloud servers play crucial roles. Extensive research has been conducted in academia and industry exploring various aspects of this collaboration. For example, Federated Learning [171] has emerged as a prominent research area, focusing on privacy-preserving and distributed model training. Large-scale recommendation systems [101] and distributed video analytics [229] are additional domains that have garnered significant attention. Through the orchestration of distributed and heterogeneous computing resources, edge-cloud collaboration introduces new enabling technologies that facilitate the provision of distributed, low-latency, and reliable intelligent services. These advancements are applicable during both the training and inference stages of ML-based IoT applications.

1.1 Research Questions

Figure 1.1 shows edge-cloud collaboration that spans two main infrastructure layers. Both layers consist of microservices for application-specific functionalities and network management. According to the high-level requirements of different applications, microservices are composed via networks within and across different layers. The edge

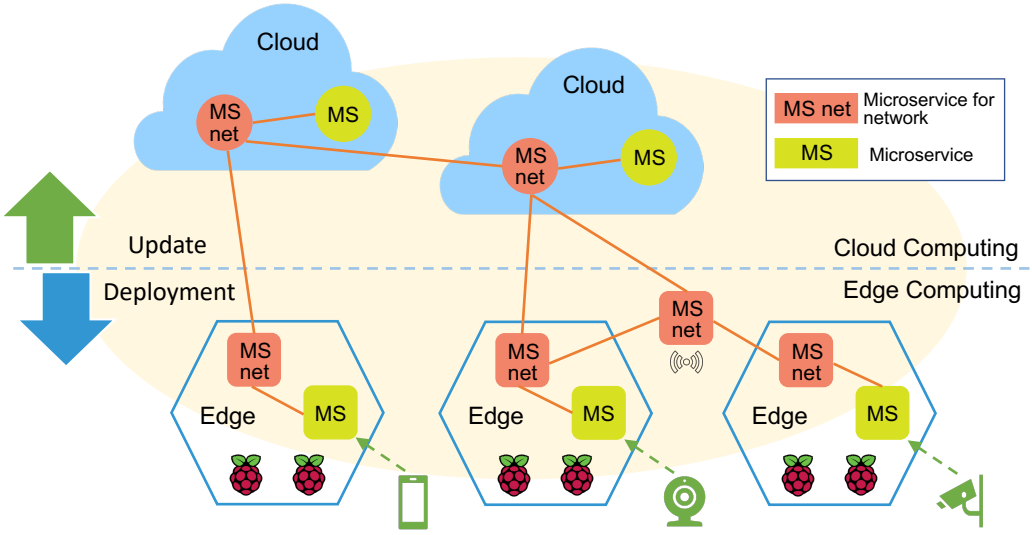


Figure 1.1: Deployment and Update Lifecycle in Edge-Cloud Computing

computing layer in ML-based IoT applications usually includes data-generating devices (i.e., cameras, sensors), data transmission devices (i.e., gateways), and computing nodes (i.e., Raspberry PI). These devices can perform various types of operations on the raw data collected in the environment, such as encoding or filtering of the incoming data streams. The cloud computing layer contains large-scale clusters to provide computing resources for high-performance microservices, including application serving services and updating services. The deployment and update lifecycle in edge-cloud computing involves the efficient provisioning, deployment, continuous updating, and optimization of ML-based IoT applications across edge and cloud resources.

The microservices deployment in a distributed and federated edge-cloud system requires suitable provisioning solutions that can aggregate various types of resources in both environments. An important problem in this context is *understanding how to aggregate these microservices in a way that supports application requirements* such as latency, throughput, energy consumption, and model prediction accuracy. Edge-cloud collaboration attempts to characterize these Quality of Service (QoS) metrics in response to the microservice composition plans and adapt to deployment sites, adapt them to deployment sites, taking into account the deployment locations and contextual factors. Moreover, it is essential for *these plans to be scalable and adaptable to fluctuations in the dynamic computing environments* that may occur over the entire application execution period. In this regard, a feedback-driven orchestration mech-

anism is necessary to detect changes in the infrastructure's performance and QoS metrics, while also being easily scalable to accommodate environments with multiple interconnected devices.

Another prevalent problem in distributed environments during application runtime is the issue of inconsistent model prediction results. In a typical distributed Deep Learning (DL) based application, models are configured differently to meet the requirements of resource constraints. For instance, a large ResNet56 model is deployed on the cloud server while a small lightweight MobileNet model is more suitable for the end-user device with fewer computation resources. However, the heterogeneity of the model architectures and configurations may bring a systemic problem - models may produce different outputs when given the same input. This inconsistency problem may cause the failure of prediction agreement inside the application. To tackle this issue, it is necessary to implement *a systemic design that can effectively detect model inconsistencies, along with the development of novel algorithms aimed at updating the models and minimizing such inconsistencies* in distributed deep learning (DL) applications.

Taking into account these problems and concerns mentioned above, we formulate the following three research questions (RQ) during application deployment and update:

- **(RQ1)** How can the relationships between system composition plans and system Quality of Service (QoS) metrics be effectively modeled in the edge-cloud computing paradigm during application deployment? Furthermore, what approaches can be employed to optimize these composition plans in order to maximize the desired QoS metrics?
- **(RQ2)** How can the extension and integration of these composition plans with running DL applications be achieved to ensure seamless scalability within dynamic and distributed edge-cloud environments?
- **(RQ3)** During application runtime, how can the efficiency of detecting and minimizing model inconsistency in distributed DL applications be improved through the collaboration between edge and cloud resources?

To answer **RQ1** needs to consider (1) the heterogeneity of the edge node: Each edge

node has a different processing rate based on current working situations (e.g., the number of IoT devices from which data are ingested). The proportion of the streaming data offloaded to the cloud should consider the computing and network load of each individual edge (e.g., CPU, upstream link utilization); (2) the interplay among edge nodes and cloud servers: all edge nodes may forward the data to the cloud simultaneously, without considering the offload policies of others. This may cause starvation on the cloud server where the data from some edge nodes may be delayed for processing; (3) the adaptation of modern streaming protocols for data analytics in ML-based IoT applications. Streaming protocols are essential for content delivery. They break streams into small segments, send them to target servers, and reassemble them at the destination. The streaming content analytical framework needs to carefully adapt to these protocols, in particular needs to consider how varying numbers of contents within a segment impact the offloading policy. Once the above parameters have been modeled, the optimization algorithm must take into careful consideration the complexity of the formulated optimization problem as well as the complexity of the algorithm itself.

To address **RQ2**, a comprehensive monitoring system is essential to capture the dynamic runtime performance of the distributed system. This entails monitoring the system's data input rate, current workload, communication bandwidth between the edge and cloud servers, and key quality-of-service (QoS) metrics such as latency, throughput, and prediction accuracy. Additionally, a distributed intelligent decision-making algorithm is necessary. This algorithm takes into account the system's status and generates decisions that optimize the system's workflow, ensuring a balanced operation. Furthermore, a holistic system design is crucial to enable online training of the algorithm. This design allows the algorithm to adaptively learn from the system's dynamics and make decisions that maximize the system's QoS metrics. This adaptive learning process ensures that the algorithm continually evolves and adjusts its configurations to achieve optimal performance.

Finally, **RQ3** studies the model consistency problem within geo-distributed Deep Learning applications. Existing research fails to consider how to detect and reduce such inconsistency when multiple models are collaborating in a real-world distributed application. An algorithm and system co-design solution is required to interact with the

distributed models and improve the consistency of system outputs. To be precise, we need to tackle the following challenges while building such distributed DL applications. 1) How to detect the inconsistency among the distributed models? An edge-based DL applications, the models are distributed and adaptively configured. This brings the challenge of how to efficiently interact with the different outputs of the models to provide a unified consistency measurement. 2) How to efficiently reduce the inconsistency among the heterogeneous models? In an edge-based DL application, the models deployed on the edge nodes are heterogeneous and distributed. Therefore, how to efficiently update (or fine-tune) these models becomes a challenge for both algorithm and system design. In particular, on the one hand, the proposed inconsistency reduction algorithm should have the flexibility and scalability to fine-tune multiple (greater than two) heterogeneous models simultaneously. On the other hand, the proposed system should have the ability to coordinate the models across the cloud and edge nodes in a distributed manner.

1.2 Contributions

In this thesis, I make the principal contributions as follows:

1. My first contribution is a framework ***OsmoticGate*** that considers the challenges in practical deployment of edge computing, and proposes a novel technique to uncover the influence of edge-cloud interaction during offloading policy design. I develop a new hierarchical queue model to describe the system dynamics of a video analytic system in an edge-cloud environment. The model is adapted to bitrate-based video streaming and focuses on modeling the processing latency and throughput of a video analytics system. Then I formulate the task offloading problem as a nonsmooth, non-convex, and constrained optimization problem, and propose a gradient-based algorithm to solve this problem efficiently. I also feed the model parameter through real-world benchmark and compare our algorithms with SOTA methods in both simulated environment and real-world testbed.

2. My second contribution is a distributed system ***OsmoticGate2*** that utilizes multi-agent reinforcement learning. Building upon my first contribution, this system features automated monitoring, controlling, and learning capabilities, and incorporates a distributed multi-agent reinforcement learning algorithm known as MAPPO. MAPPO forms the core of the system and is tightly integrated within the framework. This reinforcement learning agent features centralized training and decentralized deployment, enabling the system to make fast decisions and learn from its environment. To validate the effectiveness of the proposed system, I conduct experiments on a real-world testbed that incorporates edge-cloud collaboration. These experiments provide empirical evidence of the system's capabilities and its ability to achieve desired outcomes.

3. My third contribution is designing and implementing a system ***DEEPCON*** to realize our goal of quickly improving the model consistency of edge-cloud-based applications. I illustrate the importance of consistency in evaluating the performance of geo-distributed DL applications and define a new consistency metric (CC) for measurement. Then I propose DMML, a KD-based learning algorithm for cross-model learning, improving the consistency among the models. To improve the DMML's scalability, I develop the DMML-Par that can scale DMML to multiple GPU nodes. Then I design and implement DEEPCON to provide non-stop updates to improve the model consistency of geo-distributed DL application. Moreover, DEEPCON offers an algorithm and system co-design solution to maintain a geo-distributed DL application deployment life-cycle. Finally, I evaluate DMML with both vision and language classifications and evaluate the training speed of DEEPCON on the same dataset.

1.3 Thesis Structure

Chapter 1 describes the general background information and motivation behind the topic and illustrates the research questions and main contributions of this research.

Chapter 2 presents background material and a summary of work closely related to the original research described in this thesis.

Chapter 3 presents ***OsmoticGate*** which investigates video streaming processing task offloading in edge-cloud computing paradigm. Based on bitrate-based video streaming protocols, a Hierarchy Queue Model is proposed to capture system workload dynamics and its relation to system latency, and throughput. A two-stage gradient-based algorithm has been proposed to minimize system latency while ensuring minimal throughput. Extensive evaluation has been conducted to validate the effectiveness of *OsmoticGate* in both simulation and real-world testbed.

Chapter 4 presents ***OsmoticGate2***, an online multi-agent reinforcement learning system designed to achieve workload balancing in dynamic and distributed deep learning (DL) applications. Utilizing the state-of-the-art multi-agent reinforcement learning algorithm MAPPO, all agents actively interact with real-world environments and continuously learn from these interactions. Through this learning process, the system optimizes its performance in a dynamic setting. The experimental results demonstrate that ***OsmoticGate2*** effectively adapts to changing system configurations while ensuring stable runtime performance. These findings highlight the system's ability to successfully balance workloads and maintain desired performance levels in dynamic and distributed DL applications.

Chapter 5 presents ***DEEPCON***, an adaptive deployment framework to quickly improve model consistency through over-the-air parallel training. A whole pipeline is designed for quickly detecting the inconsistency within the systems along with an efficient learning algorithm (DMML) for improving the consistency between the models. In order to further accelerate the training process, a high-scalable algorithm DMMLPar, asynchronous parallel training of DMML is designed which adapts easily to various numbers of computation resources. ***DEEPCON*** is prototyped and implemented with a set of APIs for seamless communication between the edge and cloud layers. The evaluation results show the effectiveness of DMML in improving model consistency. The training speed-up of DMMLPar is also evaluated, which can guarantee the best consistency improvement while greatly reducing the training time.

Chapter 6 summarises and provides the conclusion of the work presented in this thesis and proposes directions for further work in the area.

2

Background and Literature Review

Contents

Summary

This chapter starts by describing some of the background information concerning the overall topic, including a brief primer on ML-based IoT applications, cloud AI, edge AI, and edge-cloud collaborations. At the same time, gaps in current research are highlighted and then briefly illustrated, and how this thesis fills these gaps.

2.1 ML-based IoT applications

2.1.1 ML-based IoT application - A Smart City Use Case

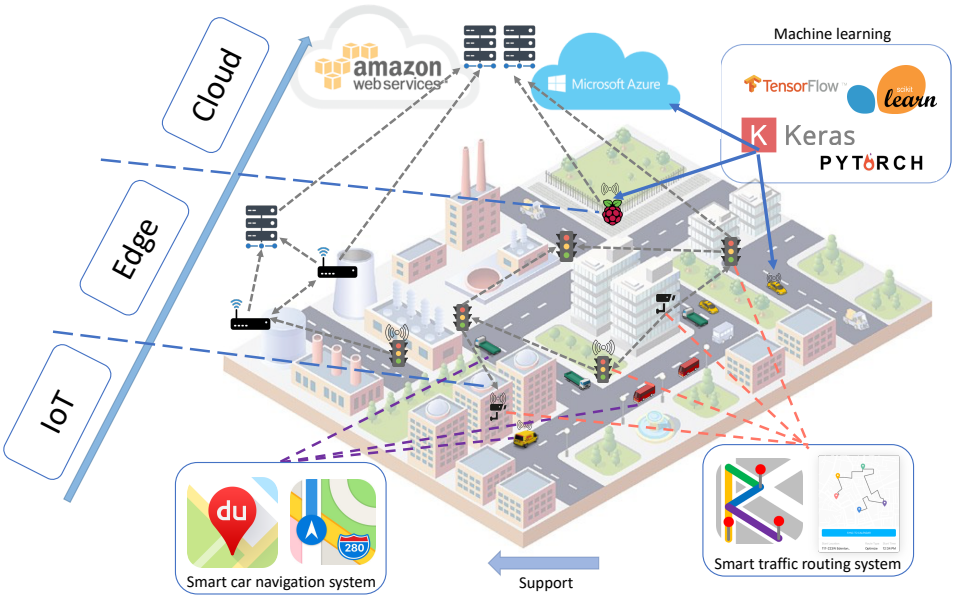


Figure 2.1: Smart City

Smart city uses modern communication and information techniques to monitor, integrate and analyze the data collected from core systems running across cities. Meanwhile, smart city makes intelligent responses to various use cases, such as traffic control, weather forecasting, industrial and commercial activities. Fig. 2.1 represents a smart city which consists of various IoT applications with many of them using Machine Learning (ML) techniques. For example, a *smart traffic routing* system consists of a large number of cameras monitoring the road traffic and a smart algorithm running on the cloud recommending the optimal routes for users [341]. On the other hand, a *smart car navigation system* [134] allows the passengers to set and change destinations

via built-in car audio devices. The two systems work together to provide real-time interactive routing services. More specifically, the user's voice commands are translated in the car edge side and sent to the cloud where the *smart traffic routing* system works. The best route is translated back to voice guiding users to their destinations. The applications mentioned above entail a wide array of computing resources, such as cloud, edge, and IoT devices, and employ various ML techniques. As a result, orchestrating the microservices within these ML-based IoT applications poses significant challenges for both the ML models and the IoT system.

2.1.2 Taxonomy of the literature review

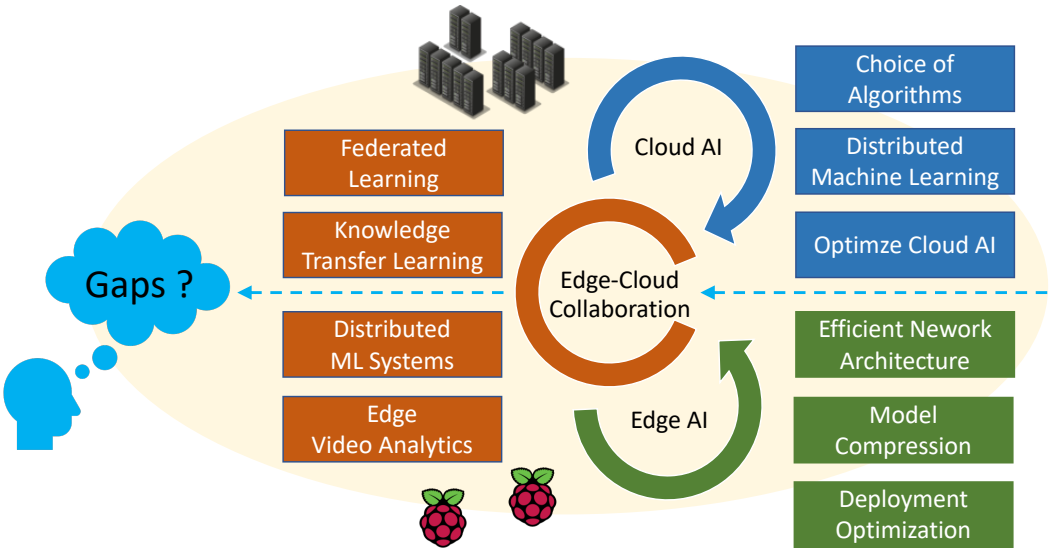


Figure 2.2: Literature Taxonomy

Figure 2.2 shows the taxonomy of the background and literature of this thesis. In the edge-cloud collaborative learning paradigm, different layers perform different types of computations. The first step is usually performed on the cloud server (see Cloud AI in §2.2), where we implement and train the model with various ML algorithms, optimizing them to achieve high efficiency without sacrificing too much accuracy. Efficient distributed machine learning techniques are utilized for fast model training with abundant cloud resources. The research on the edge AI (see §2.3), on the other side, focuses more on the efficient model architectures and deployment strategies that optimizes the on-device model performance. This includes less latency, and resource

usage, while sacrificing model prediction accuracy. Finally, based on the techniques mentioned above, the research on edge-cloud collaboration (see §2.4) utilizes computation and communication resources across the two layers, along with the abundant data source for realizing high-performance geo-distributed collaborative learning.

2.2 Cloud AI

AI has undergone remarkable development in recent years, surpassing human performance in various open-source competition benchmarks. A significant factor contributing to this success is the utilization of cloud computing, specifically large-scale distributed clusters, which greatly accelerates AI model training, referred to as cloud AI. In this review, we explore cloud models, optimization techniques, and distributed machine learning methods employed to expedite the model generation process.

2.2.1 TML vs. DL vs. RL

Model selection aims to find the *optimal* ML model to perform a user's specified tasks, whilst adapting to the complexity of IoT environments. In this section, we first discuss the model selection from three main categories i.e., TML, DL and RL, followed by a survey of well-known models (or algorithms) in each category and their corresponding criteria for model selection.

In this survey, we roughly divide the ML approaches/concepts into TML, DL and RL. Compared with the most popular DL , TML is relatively lightweight. It is a set of algorithms that directly transform the input data (to output), according to certain criteria. For supervised cases when a class label is available for training, TML aims to map the input data to the labels by optimising a model, which can be used to infer unseen data at the test stage. However, since the relationship between raw data and label might be highly non-linear, feature engineering— a heuristic trial-and-error process — is normally required to construct the appropriate input feature. The TML model is relatively simple, the interpretability (e.g., the relationship between the engineered features and the labels) tends to be high.

DL has become popular in recent years. Consisting of multiple layers, DL is powerful for modeling complex non-linear relationships (between the input and output) and thus does not require the aforementioned heuristic (and expensive) feature engineering process, making it a popular modelling approach in many fields such as computer vision and natural language processing. Compared with TML, DL models tend to have more parameters (to be estimated) and generally they require more data for reliable representation learning. However, it is crucial to guarantee the data quality and a recent empirical study[209] suggested the increasing number of noisy/less-representative training samples may harm DL's performance, making it less generalizable to unseen test data. Moreover, DL's multilayer structures make it difficult to interpret the complex relationship between input (i.e., raw features) and output. However, more and more visualisation techniques (e.g., attention map [330]) were used, which play an important role in understanding DL's decision-making process.

RL has become increasingly popular due to its success in addressing challenging sequential decision-making problems [272]. Some of these achievements are based on the combination of DL and RL, i.e., Deep Reinforcement Learning. It has shown its considerable performance in natural language processing [167, 304], computer vision [16, 57, 241, 271, 316], robotics [232] and IoT systems [192, 193, 337] and related applications like video games [16], visual tracking [241, 271, 316], action prediction [57], robotic grasping [232], question answering [304], dialogue generation [167], etc. In RL, there is usually one or more agent(s) interacting with the outside environment, where optimal control policies are learnt through experience. Fig. 2.3 illustrates the iterative interaction circle, where the agent starts without knowing anything about environment or task. Each time the agent takes action based on the environment states, and it receives a reward from the environment. RL optimises this process such that it learns to make decisions with higher rewards received.

Discussion. In IoT environments, a variety of problems can be modelled by using the aforementioned three approaches. The applications range from system and networking [193] [192], smart city [337] [168], to smart grid [299] [244], etc. To begin with modeling, it is essential for users to choose a suitable learning concept at the first stage. The main selection criteria can be divided into two categories: *Function-based selection*

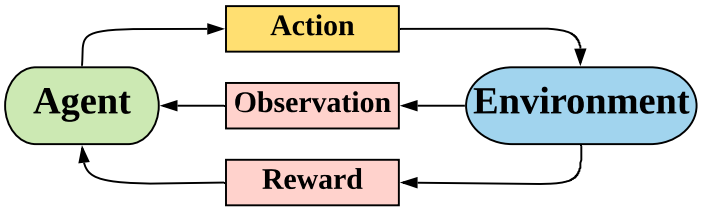


Figure 2.3: Reinforcement Learning Paradigm

and *Power Consumption-based selection*.

Function-based selection aims to choose an appropriate concept based on their functional difference. For example, RL benefits from its iterative environment $\leftrightarrow$ agent interaction property, and can be applied to various applications which need interaction with environment or system such as smart temperature control systems, or recommendation systems (with cold start problem). On the other hand, TML algorithms are more suitable for modelling structured data (with high-level semantic attributes), especially when interpretability is required. DL models are typically used to model complex unstructured data, e.g., images, audios, time-series data, etc. and are an ideal choice especially with high amount of training data and low requirement on interpretability.

Power Consumption-based selection aims to choose an appropriate model given constraints in computational power or latency. In contrast to TML, the powerful RL/DL models are normally computationally expensive with high overhead. Recently, model compression techniques were developed, which may provide a relatively efficient solution for using RL/DL models for some IoT applications. However, on some mobile platforms with very limited hardware resources (e.g., power, memory, storage), it is still challenging to employ compressed RL/DL models, especially when there are some performance requirements (e.g., accuracy, or real-time inference) [61]. On the other hand, lightweight TML may be more efficient, yet reasonable accuracy can only be achieved with appropriate features (e.g., high level attributes derived from the time-consuming feature engineering).

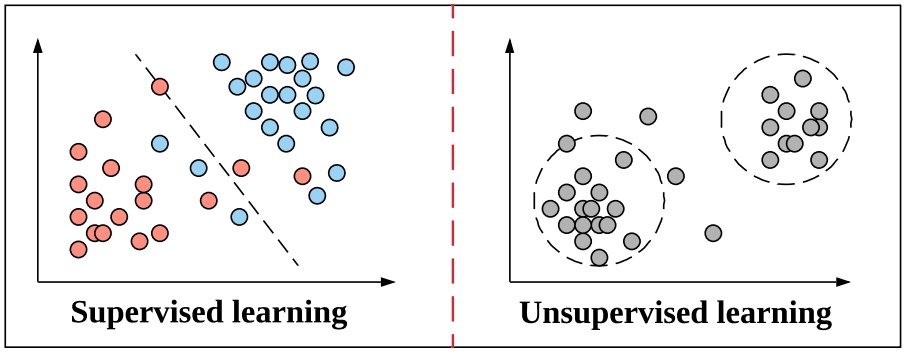


Figure 2.4: Examples of Supervised Learning (Linear Regression) and Unsupervised Learning (Clustering)

2.2.2 Traditional Machine Learning

Herein we demonstrate several popular TML algorithms, and discuss the criteria for choosing the TML algorithms. Given different tasks, TML can be further divided into *Supervised Learning* and *Unsupervised Learning*.

Supervised Learning. Supervised learning algorithm (i.e., Fig. 2.4) can be used when both the input data X and the corresponding labels Y are provided (for training), and it aims to learn a mapping function such that $Y :\leftarrow f(X)$. Supervised learning algorithms have been widely used in IoT applications, we introduce the most representative classifiers below.

Perceptron and *Logistic Regression (LR)* are probably the simplest linear classifiers. For both models, the model (i.e., weights and bias) is basically a simple linear transformation. Perceptron can perform binary classification simply based on the sign of the (linearly) transformed input data, while LR will further scale the transformed value into probability (via *sigmoid* function), before a thresholding function is applied for the binary classification decision. LR can also be extended to process multi-class classification scenarios by using *softmax* as the scaling function, with class-wise probabilities as output.

Artificial Neural Networks (ANN) is a general extension of the aforementioned linear classifiers. Compared with *Perceptron* or *LR* which linearly project input data to the output, *ANN* has an additional "hidden layer" (with a non-linear activation function),

which enables *ANN* to model non-linearity. However, in contrast to linear classifiers, this additional hidden layer makes it more difficult to see the relationship between the input and output data (i.e., low interpretability). Although in theory, with one hidden layer *ANN* can model any complex non-linear functions, in practice it has limited generalization capabilities when facing unseen data. *ANN* with more layers, also referred to as deep neural networks, tend to have better modelling capability, which will be introduced in the next subsection.

Decision Tree (DT) [233] and *Random Forest (RF)* [40] are two tree-structure based non-linear classifiers. Based on certain attribute-splitting criteria (e.g., *Information Gain* or *Gini Impurity*), *DT* can analyse the most informative attributes sequentially (i.e., splitting) until the final decision can be made. The tree structure makes it interpretable and it has reasonable accuracy with low-dimensional semantic attributes. However, it faces "the curse of dimensionality" problem and does not generalize well when the input feature quality is low. *RF*, on the other hand, can effectively address this overfitting issue. *RF* is an ensemble approach on aggregating different small-scale DTs, which are derived based on random sampling of the features/datasets. The random sampling mechanism can effectively reduce the dimensionality (for each individual *DT*) while the aggregation function can smooth the uncertainty of individual *DT*s, making *RF* a powerful model with great generalisation capabilities. However, the interpretability of *RF* tends to be less obvious than that of *DT*, owing to the random sampling and aggregation mechanisms.

Support Vector Machine (SVM) [71] is another popular supervised learning method. It is also called large margin classifier as it aims at finding a hyperplane that is capable of separating the data points (belonging to different classes) with the largest margin. For non-linearly separable datasets, various kernels (e.g., *RBF (Radial Basis Function)*) can be applied into the *SVM* framework with good generalization ability. Yet the time complexity for training this algorithm can be very high (i.e., $O(N^3)$ [8], where N represents the dataset size), making it less suitable for big datasets. On the other hand, *K-Nearest Neighbour (KNN)*[74], which does not require a training process (also referred to as lazy learning), is another powerful non-linear classifier. The classification is performed by distance calculation (between query and all the training examples),

distance ranking, and majority voting among the (K) nearest neighbours. So selecting suitable distance functions/metrics (for different tasks) is one of the key issues in *KNN*. Since for any query sample, the distance calculation has to be performed for every sample in the whole training set, it can be time-consuming and thus less scalable to large datasets. Different from the aforementioned methods, *Naive Bayesian (NB)* algorithm [98] takes the prior knowledge of the class distribution into account. Based on the assumption that the features are conditionally independent, the likelihood of each feature can be calculated independently, before being combined with the prior probability according to the Bayes' rule. If the feature-independence assumption is not significantly violated (e.g., low-dimensional structured data), it can be a very effective and efficient tool.

Unsupervised Learning. The unsupervised learning algorithm (see Fig. 2.4 right) aims at learning the inherent relationship between the data when only input data X exists (without class label Y). For example, the clustering algorithm can be used to find the potential patterns of some unlabelled data and the obtained results can be used for future analysis. *K-Means*[113] and *Principal Component Analysis (PCA)* [255] are the two most popular unsupervised learning algorithms. *K-means* aims to find K group patterns from data by iteratively assigning each sample to different clusters based on the distance between the sample and the centroid of each cluster. *PCA* is normally used for dimensionality reduction, which can de-correlate the raw features before selecting the most informative ones.

Discussion. For IoT applications, a common principle is to select the algorithm with the highest performance in terms of effectiveness and efficiency. One can run all related algorithms (e.g., supervised, or unsupervised), before selecting the most appropriate one. For effectiveness, one has to define the most suitable evaluation metrics, which can be task-dependent, e.g., accuracy or mean-f1 score for classification tasks, or mean squared errors for regression, etc. Before model selection, a number of factors should be taken into account: data structure (structured data, or unstructured data which may need additional preprocessing), data size (small or large), prior knowledge (e.g., class distribution), data separability (linearly, or non-linearly separable which may require additional feature engineering), dimensionality (low, or high which may require

dimensionality reduction), etc. There may also exist additional requirements from the users/stakeholders, e.g., interpretability for health diagnosis. Additionally, it is necessary to understand the efficiency requirement specific to an IoT application and one has to consider how the training/testing time grows with respect to data size. Take *KNN* as an example: although no training time is taken, *KNN*'s inference time can be very high (especially with a large training set), and thus presumably unsuitable for certain time-critical IoT applications. Also, the deployment environment is another non-negligible factor when developing IoT applications since many applications run (or partially run) on low power computing resources.

2.2.3 Deep Learning

In this section, we primarily introduce three classical deep models (i.e., *Deep Neural Networks* (DNN)/ *Multilayer Perceptron* (MLP), *Convolutional Neural Networks* (CNN) and *Recurrent Neural Networks* (RNN)) for supervised learning tasks on unstructured data such as image, video, text, time-series data, etc. We also brief two popular unsupervised models: *Autoencoder* (AE), and *Generative Adversarial Networks* (GAN).

Supervised DL. Next, we will introduce three basic supervised DL models: DNN, CNN and RNN, which require both the data and label for training.

Deep Neural Networks (DNN). As previously mentioned, a deep neural network (*DNN*) is an *ANN* with more than one hidden layer, and hence it is also called multilayer perceptron (*MLP*). Compared with *ANN* with a single hidden layer, *DNN* has more powerful modelling capabilities and its deep structure makes it easier for it to learn higher-level semantic features, which is crucial for classification tasks on complex data. However, for high-dimensional unstructured input data (such as images), there may be many model parameters to be estimated, and in this case, overfitting may occur if there is not enough labelled data. Nevertheless, generally *DNN* has decent performance when input dimensionality is not extremely high, and it has been successfully applied to various applications, for example human action recognition [284], traffic congestion prediction[83] and healthcare[212].

Convolutional Neural Network (CNN). When it comes to high-dimensional unstructured data such as images, in visual recognition tasks it is hard to directly map the raw image pixels into target labels due to the complex non-linear relationship. The traditional way is to perform feature engineering, which is normally a trial-and-error process, and may require domain knowledge in certain circumstances, before TML is applied. This heuristic approach is normally time-consuming, and there exist substantial recognition errors even in simple tasks since it is very challenging to hand-engineer the high-level semantic features. *CNN*, a deep neural network with convolutional layers and pooling layers, can address this issue effectively. The convolution operation can extract the higher level features while the pooling operation can keep the most informative responses and reduce the dimensionality. Compared with *DNN*, the weight sharing concept (of the convolution operation) enables *CNN* to capture the local pattern without suffering from the "curse of high-dimensionality" from the input. These operations and the hierarchical nature make *CNN* a powerful tool for extracting high-level semantic representations from raw image pixels directly, and successfully applied to various recognition tasks such as object recognition, image segmentation [96] and object detection [114]. Because of the decent performance on various visual analysis tasks, *CNN* is usually considered as the first choice for some camera-based IoT applications, for example traffic sign detection [256].

Recurrent Neural Networks (RNN). Nowadays, with the increasing amount of generated stream and sequential data from various sensors, time series analysis has become popular among the machine learning (ML) community. *RNN* is a sequential modelling technique that can effectively combine the temporal information and current signal into the hidden units for time-series classification/prediction. An improved *RNN* named Long Short Term Memory (LSTM) [121], including complex gates and memory cells within the hidden units for "better memories", became popular in various applications such as speech recognition [104], video analysis [283], language translation [187], activity recognition [106] etc. Since data streaming is most common in the IoT environment, RNN (LSTM) is deemed as one of the most powerful modelling techniques, and there are various IoT applications such as smart assistant [97, 284], *smart car navigator system* [134], malware threat hunting [109], network traffic forecasting [235],

equipment condition forecasting [332], energy demand prediction system [206], load forecasting [156], etc.

Unsupervised DL. We will also introduce two unsupervised DL models: *Autoencoder (AE)* [23] and *Generative Adversarial Network (GAN)* [102].

Without requiring any label information, *AE* can extract compact features and reconstruct the original (high-dimensional) data with the extracted features. It is normally used for dimensionality reduction, latent distribution analysis or outlier detection. *GAN*, on the other hand, applies an adversarial process to learn the "real" distribution from the input data. More precisely, *GAN* consists of two parts, namely generator and discriminator. The generator aims at generating indistinguishable samples compared to the real data. While the discriminator works adversarially to distinguish the generated fake samples from the real data. It is an iterative competition process that will eventually lead to a state where the generated samples are indistinguishable from the real data. With the learnt "real" distribution, one can generate various samples for different purposes. *AE* and *GAN* are both powerful tools in the computer vision field, and their properties make them promising approaches for IoT applications. *AE* can be used for diagnosis/fault detection tasks [65, 217] or simply as a preprocessing tool (i.e., feature extraction/dimensionality reduction). *GAN* has been used for studies on generating rare category samples, and this upsampling approach may further improve the model performance [335, 336].

Discussion. The aforementioned DL models can be effective tools for processing different unstructured data types. The way of applying them is generally very flexible, and they can be used jointly to process complex data from various sources in the IoT environments. For example, although *CNN*/*RNN* could be used in an end-to-end manner (e.g., as image/time-series classifiers), they could also be used as feature extractors, based on which one can easily aggregate features extracted from different sources (e.g., audio, images, sensor data). With high-dimensional video data, one can either model by training *CNN*+ *LSTM* jointly [287], or use *CNN*/*AE* as feature extractors, before the sequential modelling (e.g., using *LSTM*). However, when modelling the data with limited labels (e.g., rare event), one needs to consider the potential overfitting effect when using DL directly. One may go back to the TML approaches or use some

upsampling techniques (e.g., *GAN*) to alleviate this effect.

2.2.4 Reinforcement Learning (RL)

In this section, we first introduce the strategies used to formulate the aforementioned video streaming example (see §2.2) with Reinforcement Learning (RL). As mentioned earlier, in RL an *agent* interacts with the *environment*, learning an optimal control policy through experience. It requires three key elements, *observation*, *action*, and *reward*. Based on these, we can formulate the adaptive bitrate streaming problem. Specifically, *observation* can be the buffer occupancy, network throughput, etc. At each step, the *agent* decides the bitrate of the next chunk. A *reward* (for example the quality of service feedback from the user) is received after the *agent* takes *action* (chunk bitrate). The algorithm proposed in [193] collects and generalizes the results of performing the past decisions and optimizes its policy from different network conditions. This RL-based algorithm can also make the system robust to various environmental noises such as unseen network conditions, video properties, etc.

As shown in Fig. 2.5, there is a plethora of algorithms in the whole reinforcement learning family. Here we focus on selecting appropriate RL algorithms based on different selection criteria.

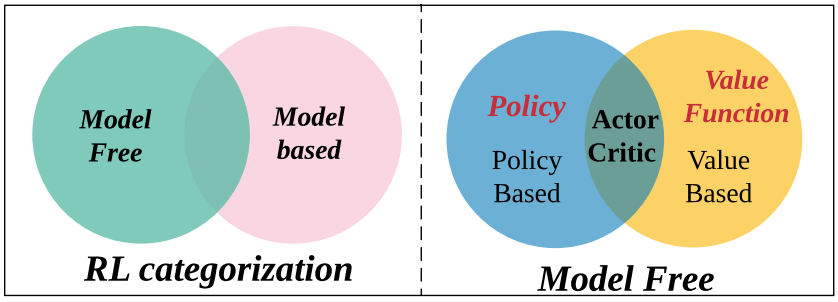


Figure 2.5: Reinforcement Learning Categorization

Environment Modelling Cost In RL modelling, sample efficiency is one of the major challenges. Normally the RL agent can interact either with the real world or a simulated environment during training. However, it can be difficult to simulate the heterogeneous IoT environments and complex IoT devices. RL models can also be trained directly in real world IoT environments, yet one major limitation is the heavy

training cost, which may range from seconds to minutes for each step. The model-based RL method, a method that can reduce the sample complexity, can decrease the training time significantly. It first learns a predictive model of the real world, based on which the decisions can be made. When compared with model-free approaches, model-based methods are still in their infancy, and because of the efficiency property, they may attract more attention in the near future.

Action Space: The action space of RL algorithms can be either continuous or discrete. For those RL algorithms with discrete action space, they choose from a finite number of actions at runtime. Take the video streaming task for example, the action space is different bitrates for each chunk. Another task formulated in discrete action space can be found in [192], where the action space is the "schedule of the job at i-th slot". Available algorithms for discrete action space tasks most reside in the policy gradient group, for example DQN, DDQN. The continuous action space, on the other hand, is infinite for all possible actions. Relationships exist between the actions that are usually sampled from certain distributions such as Gaussian distribution. For example, in an energy-harvesting management system, PPO algorithm [249] is used to control IoT nodes for power allocation. The action space, as stated in [207], is sampled from a Gaussian distribution to denote the load of each node ranging from 0% to 100%. Similarly, in another work [14] that studied energy harvesting WSNs, the Actor-Critic [154] algorithm is implemented to control the packet rate during transmission. One advantage of continuous action space lies in its ability to accurately control the system, thus a higher QoE is expected.

2.2.5 Distributed Machine Learning

Modern ML models such as neural networks require a substantial amount of data for the training process. These data are usually aggregated and stored in the cloud server where training happens. However, when the training process of large volume data outpaces the computing power of a single machine, we need to leverage multiple machines available in the server cluster. This requires the development of novel distributed ML systems and parallel training mechanisms which distribute and accelerate the machine learning workload.

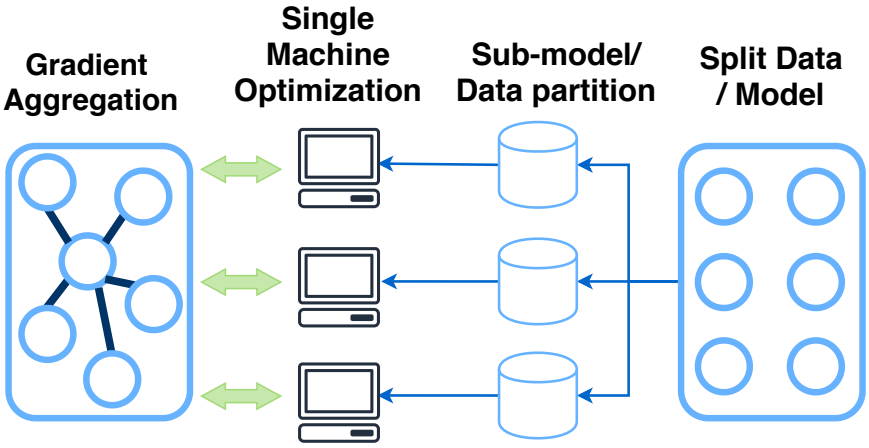


Figure 2.6: Distributed Machine Learning Pipeline

Fig 2.6 shows the schematic diagram of a distributed ML pipeline. It has multiple components which are engaged in *Training Concurrency*, *Single Machine Optimization*, and *Distributed System*. In *Training Concurrency*, either the models or the data are split into small chunks and placed on different devices. With *Single Machine Optimization* (which shares similar techniques as conventional ML, see § 2.2.6) that accelerates the training process, we get all local gradient updates. Finally, *Distributed System* discusses strategies that efficiently aggregate the gradient updates.

Training Concurrency in Distributed ML. In the distributed machine learning, the selection of parallel strategy depends on two factors: data size and model size. When either the datasets or the model parameters are too big for single-machine processing, it is straightforward to consider partitioning them into smaller chunks for processing at different places. Here we first introduce two basic methods *data parallel*, *model parallel*. We also introduce *pipeline parallel* and other hybrid approaches that take advantage of both approaches.

Data Parallel. In a multi-core system where a single core can not store all the data, data parallel is considered by either splitting the data samples or the features. Data parallel has been successfully applied to numerous machine learning algorithms [66] with each core working independently on a subset of data. It can be used for training ML algorithm for example decision trees and other linear models where the features are relatively independent. Parallel with the split of data features, though, it can not be used directly with neural networks because different dimensions of the features are

highly correlated.

In deep learning, data-parallel works by distributing the training dataset across different GPU units. The dominant data parallel approach is *batch parallelism* where mini-batch SGD is employed to compute local gradient updates on a subset of the data. A central server is responsible for aggregating all local updates to global parameter and pushing new models back to the working units. One of the earliest works trained with GPUs can be found in [234] where the authors implemented distributed mini-batch SGD unsupervised learning concurrently with thousands of threads in a single GPU. By varying the batch size [103, 263, 312], this method is effective in reducing the communication cost without too much accuracy loss. In the next paragraph, we will discuss more about the parallel SGD algorithms [197, 268, 315, 327, 346] for improving the communication efficiency, which can be seen as one way of improving the performance of data parallelism. Another type of data parallel that addresses the memory limit on single GPU is *spatial parallelism* [146]. *Spatial parallelism* considers partitioning spatial tensors into smaller subdivisions and allocating them to separate processing units. It thus differs from *batch parallelism* in that the latter puts the groups of data in the same process. *Spatial parallelism* approach has proven to show near linear speedup on modern multi-GPU systems.

Model Parallel. Data parallel suffers from the infeasibility of dealing with very large models especially when it exceeds the capacity of a single node. Model parallel addresses this problem by splitting the model with only a subset of the whole model running on each node [39, 79, 150, 164]. The computation graphs can be divided within the layers (horizontal) or across the layers (vertical). Mesh-tensorflow [254] allows linear within-layer scaling of model parameters across multiple devices after compiling a computation graph into a SPMD program. However, this approach requires high communication cost as it needs to split and combine model updates across a large number of units. [133] introduced decoupled parallel backpropagation to break the sequential limitation of the back-propagation between the nodes, greatly increasing the training speed without much accuracy loss. For CNN, as each layer can be specified as five dimensions including: samples, height, width, channels, and filters, existing literature [88, 89] studies the split of models among dimensions. Another research direction that

optimizes the communication overhead is by searching the optimal partition and device placement of computation graphs via reinforcement learning [202]. The literature [138, 291] followed this idea and shows interest in automatic search of optimal parallel strategies.

Pipeline Parallel. Although model parallel has proven successful in training extremely large models, the implementation is complicated due to the complexity of the neural network structure. This is especially true for CNNs since the convolution operators are highly correlated. Also, GPU utilization is low for model parallel. Due to the gradient interdependence between different partitions, usually only one GPU is in use each time. To solve the aforementioned problems, pipelining has been studied [224, 302] for speeding up the model training. With pipeline parallel, models are partitioned and displayed across different GPUs. Then mini-batches of training data are injected to the pipeline for concurrent processing of different inputs at the same time. Fewer worker GPUs are idle in the pipeline parallel setting as each node is allocated jobs, without waiting for other nodes to finish their work. According to the synchronization strategy we discussed earlier, gradients are aggregated by either synchronous pipeline model (GPipe [130]) or asynchronous pipeline model (PipeDream [112], SpecTrain [54], XPipe [105]). Theoretical analysis of pipeline parallel optimzation has also been studied and with Pipeline Parallel Random Smoothing (PPRS) [69], convergence rates can be further accelerated.

Hybrid. Data and model parallel are not mutually exclusive. Hybrid approaches that combine the benefits of both methods are effective in further accelerating the training process. Pipeline parallel [112, 130] can be seen as an approach built on top of data parallel and model parallel. Apart from that, [157] proposed combining data parallel and model parallel for different types of operators. With data parallel for CNN layers and model parallel for DNN layers, it achieved a 6.25× speed up with only 1% of accuracy loss on eight GPUs. Another implementation MAPS-Multi [27] borrows the idea of [157] and automates the partitioning of workload among multiple GPUs, achieving 3.12× speed up on four GTX 780 GPUs. Other forms of data parallel and model parallel hybrids exist in the literature [63, 79, 99, 100] that reduce the overall communication and computation overhead.

2.2.6 *Optimize Cloud AI*

Model training via a single machine is a common strategy for ML model generation. By placing the learning-related computation in the same place, the model learns from the data and updates its parameters. In this subsection, we highlight two approaches that leverage hardware for the training process acceleration: *Computation Optimization*, *Algorithm Optmization*.

Computation Optimization The basic computation unit of neural networks consists of vector-vector, vector-matrix and matrix-matrix operations. Efficient implementation of computations can accelerate the training and inference process. The Basic Linear Algebra Subprogram (BLAS) [1] standardizes the building blocks for basic vector, matrix operations. A higher level linear algebra library such as cuBLAS [2] implements BLAS on top of NVIDIA CUDA and is efficient in utilizing the GPU computation resource. Intel Math Kernel Library (MKL) [3], on the other hand, maximizes performance on Intel processors and is compatible with BLAS without the change of code.

Different DL architectures (e.g., DNNs, CNNs and RNNs) may require different optimizations in terms of basic computations. The DNN computation is usually basic matrix-matrix multiplication and the aforementioned BLAS libraries can efficiently accelerate the computations with GPU resources. The CNNs and RNNs are different in their convolution and recurrent computations. Convolutions can not fully utilize the multi-core processors and the acceleration can be achieved by unrolling the convolution [52] to matrix-matrix computation or computing convolutions as point-wise product [195]. For RNN (LSTM), the complex gate structures and consecutive recurrent layers differ from the DNNs and CNNs in that these computation units can not be split and deployed directly at different devices. This has made parallel computation difficult to apply. Optimization is possible though, with implementations on top of NVIDIA cuDNN [62]. Computations among the same gates can be grouped into larger matrix operations [15] and save intermediate steps. We can also accelerate by caching RNN units' weights with the GPU's inverted memory hierarchy [84]. The weights are

[1] http://www.netlib.org/blas
[2] https://docs.nvidia.com/cuda/cublas/
[3] https://software.intel.com/en-us/mkl

reusable between time steps, making a maximum 30× speed up on a TitanX GPU.

Algorithm Optimization Apart from the resource utilization optimization, the algorithmic level optimization is another important research direction for efficient model training and faster convergence. Optimization algorithms aim at minimizing/maximizing a loss function that varies for different machine learning tasks. They can be divided into two categories: *First Order Optimization* and *Second Order Optimization*.

First Order Optimization methods minimizing/maximizing the loss function with the gradient values with respect to the model parameters. Gradient Descent is one of the most important algorithms for neural networks. After back-propagation from the loss function, the model parameters are updated towards the opposite direction of the gradient. Gradient descent approaches fall into local optima when the absolute value is either too big or too small. Also it updates the gradient of the whole data set at one time, memory limitation is always a big problem. Variants have been proposed to address the aforementioned problems, including Stochastic gradient descent [35], mini-batch gradient descent [80]. Also, much famous research enables faster model convergence: Momentum [231], AdaGrad [93], RMSProp [118], ADAM [151]. *Second Order Optimization* methods take second order derivative for minimizing/maximizing loss function. Compared to the *First Order Optimization*, it consumes more computation power and is less popular for machine learning model training. However, *Second Order Optimization* considers the surface curvature performance and is less likely to get stuck on saddle points. Thus it sometimes outperforms the *First Order Optimization*. Famous *Second Order Optimization* methods include [19, 46, 116, 205, 218]. For a more systematic survey on the optimization methods for machine learning training, one can refer to [36].

2.3 Edge AI

The breakthrough of AI on edge devices has led to a plethora of intelligent applications. The traditional cloud-based paradigm, though effective, raises concerns about data privacy and reliance on network conditions due to data uploading. To address these challenges, edge inference has emerged, allowing us to deploy models partially

or entirely on mobile devices for local predictions. However, edge inference is complex as it must accommodate limited computing, storage, and energy resources on mobile devices. This section reviews essential techniques aimed at optimizing models for edge deployment.

2.3.1 Efficient Model Architecture

The state-of-the-art DL models often require high computational resources beyond the capabilities of IoT devices. Those models that perform well on large CPU and GPU clusters may suffer from unacceptable inference latency or even be unable to run on edge devices (e.g. Raspberry Pi). Tuning the deep neural network model architectures to increase the efficiency without sacrificing much accuracy has been an active research area. In this section, we will cover three main optimization directions: *Efficient architecture design*, *Neural architecture search* and *Model compression*.

Efficient architecture design. There exist neural network models that can specifically match the resource and application constraints. They aim to explore highly efficient basic architecture specially designed for platforms such as mobiles, robots as well as other IoT devices. MobileNets [124] is among the most famous works and proposed to use depth-wise separable convolutions [257] to build CNN models. By controlling the model hyper-parameters, MobileNets can strike an optimal balance between the accuracy and the constraints (e.g., computing resources). Later in MobileNetv2 [248], the inverted residual with linear bottleneck architecture was introduced to significantly reduce the operations and memory usage. Other important works include Xception [64] ShuffleNet [333], ShuffleNetv2 [188], CondenseNet [128]. These neural network models optimize on-device inference performance via efficient design of building blocks, achieving much less computational complexity while keeping or even raising accuracy on various computer vision datasets. Some work even outperforms the neural architectures generated through exhaustive automatic model search. Also, different building blocks can be combined together for even lighter models.

Neural architecture search (NAS). Another research direction named neural architecture search aims at searching an optimal model structure in a predefined search

space. There are usually three types of algorithms: reinforcement learning approach [180, 225], Genetic Algorithm (GA) based [181, 238], and other algorithms [21, 42].

The models generated by these methods are normally constrained to smaller model sizes. Model size and operation quality are the two most common metrics to be optimized, over other metrics such as inference time or power consumption. Representative works including MONAS [126], DPP-Net [85], RENA [344], Pareto-NASH [95] and MnasNet [276] are interested in finding the best model architectures to meet these constraints. These approaches are more straightforward as they optimize directly over real world performance. However, one drawback of NAS is the extensive computing power required for finding the optimal neural architectures. Thus, the already generated architectures can be utilized as guidance for future design for more efficient neural network model architecture.

2.3.2 Model Compression

As modern state-of-art DL models can be very large, reducing the model computation cost is crucial for deploying the models on IoT devices, especially for those latency-sensitive real-time applications. Model compression methods can be divided into four categories, *Parameter pruning and sharing*, *Low-rank factorization*, *Transferred/compact convolutional filters* and *Knowledge distillation*. Hereby we briefly summaries the categories of model compression techniques and list several important works.

Parameter pruning and sharing method aims to find and remove the redundant parameters (of DL models) for higher efficiency and generalization. One direction is to apply quantization techniques. The DL's parameters/weights are usually stored in memory with 32-bits, and quantization techniques can compress them into 16-bits [108], 8-bits [285] or even 1-bit [72, 73, 237]. On the other hand, weight pruning and sharing (in pre-trained DL models) has also attracted interest among the community. Some popular methods imposed L1 or L2 regularization constraints [163, 297], which can penalize models with more parameters, yielding the effect of pruning unnecessary parameters.

Low-rank factorization method decomposes the CNN or DNN tensors to lower ranks. The tensor matrix decomposition is implemented for each layer of the DL model. That is, once the decomposition for a certain layer is completed, the parameter size will be fixed (for this layer) and the decomposition will proceed to the next layer. Interesting work can be found in [162]. However, there are two major drawbacks. For very large DL models, it may be very expensive to perform decomposition owing to large parameter matrices. On the other hand, its layer-wise nature may also yield cumulative error, diverting the compression results to be far from optimal.

Transferred/compact convolutional filters method reduces the memory consumption by implementing special structural convolutional filters. Motivated by the equivariant group theory [68], the transferred convolutional filter transforms the model layers to a more compact structure, thus reducing the overall parameter space. The family of transformation functions [166, 253, 319] operates in the spatial domain of the convolutional filters to compress the whole model parameters. Compared to other model compression methods, transferred convolutional filters methods are less stable due to the strong transfer assumptions. However, when the assumption holds, the compact convolutional filter can have very good performance. In [273, 300], the filters were decomposed from 3×3 or bigger to 1×1 convolutions—ideal operations for IoT devices.

Knowledge distillation method learns a new, more compact model that mimics the function presented by the original complex DL model. The idea came from the work in [43], where a neural network model was applied to mimic the behavior of a large classifier ensemble system. Later this idea has been extended to the complex DL methods [119], more details can be found in [22, 243, 317]. However, currently the knowledge distillation methods are limited to classification tasks and further development is required.

Discussion. Types of model compression techniques have their own strengths and weaknesses and thus optimal choice is based on specific user requirements. *Parameter pruning and sharing* methods are the most commonly applied techniques for compression models from original models. It is stable as with proper tuning, this approach usually delivers no or few accuracy losses. On the other hand, *Transferred/compact*

convolutional filters methods address the compression from scratch. This end-to-end efficient design for improving the CNN performance approach shares similar insights to the efficient *neural architecture design* approach as we discussed earlier. *Knowledge distillation* methods are promising when working with relatively small datasets as the student model can benefit from the teacher model with less data. All these methods are not mutually exclusive, we can make combinations based on specific use cases to optimize the models that are more suitable for low-resource IoT devices.

When the ML model development process is finished, the developed models are to be deployed and composed as an application in the complex IoT environments. To simplify the deployment, the ML models and underlying infrastructure need to be specified (§2.3.3). Next, the optimization techniques can be applied to generate the deployment plans that select the suitable ML models for the deployment, optimizs the resource utilization of the model deployment and improve the reusability of the deployed models (§2.3.4). Once the deployment plans are generated, the models will be deployed over the specified infrastructure and the deployed models will be composed as well.

2.3.3 Declarative Machine Learning and Deployment

Declarative ML. Declarative ML aims to use high-level language to specify ML tasks by separating the applications from the underlying data representation, model training and computing resources. There are *three* general properties of declarative ML. First, the high-level specification only considers data types of input, intermediate results and output. They are exposed as abstract data types without considering the physical representation of the data or how the data is processed by the underlying ML models. Second, the ML tasks are specified as high-level operations through well-defined semantics. The basic operation primitives and their expected accuracy levels (or confidence interval) are defined accordingly. Based on the operation semantics, declarative ML systems select the features and underlying ML models automatically or semi-automatically, optimize the model performance and accuracy for varying data characteristics and runtime environments. Notably, the selection is based on the available models, provided as services. Finally, the correctness of the ML models must

be satisfied when a given model produces the equivalent results in any computing resources with the same input data and configurations. As a result, the declarative ML enables execution of the ML models over various hardware and computation platforms (such as Apache Spark) without any changes. Besides, these specification languages also bring flexibility and usability in the ML model deployment stage.

SystemML [31] is an implementation of declarative ML on Apache Spark. Through domain-specific languages, it specifies the ML models as abstract data types and operations, independent of their implementation. The system is able to specify the majority of ML models: matrix factorizations, dimension reduction, survival models for training and scoring, classification, descriptive statistics, clustering and regression. There is also other state-of-the-art research on declarative ML, including TUPAQ [265] and Columbus [322]. They utilize language specification and modelling technologies to describe the ML models for automatic model and feature selection, performance and resource optimization, model and data reuse.

Declarative Deployment. Hardware in the IoT environment consists of three basic types of device: *data generating* devices, *data processing* devices and *data transferring* devices. *Data generating* devices are also called "Things" (e.g., sensors, CCTV) and are used to collect environmental data. *Data transferring* devices such as router, IoT gateway, base station are used to transfer the generated data to the *data processing* devices. *Data processing* devices are used to run the analytic jobs. They can be GPU, CPU and TPU servers running in cloud or ARM based edge device such as Raspberry Pi and Arduino. An ML-based IoT application is usually running across a fully distributed environment, such that it requires correct specification of the component devices as well as the precise interoperation between these devices. [260] lists fundamental aspects that may simplify the hardware specification, i.e., processor, clock rate, general purpose input/output (GPIO), connectivity methods (Wi-Fi, Bluetooth, wired connection) and communication protocols (serial peripheral interface), universal asynchronous receiver-transmitter (UART).

Regarding the software, it is often categorized into three groups based on operation levels: *operating system (OS)*, *programming language* and *platform*. IoT *OS* allows users to achieve the basic behavior of a computer within internet-connected devices.

Software Specification	**Cloud**	**Edge**	**IoT devices**
Main OS	- Ubuntu - CentOS - Debian - RHEL - Windows Server - Amazon Linux	- Raspbian - NOOBS - Amazon FreeRTOS - RIOT - Google Fuchsia OS - Windows 10 IoT	- Amazon FreeRTOS - Contiki - TinyOS - RIOT - Ubuntu Core - Mbed OS
Programming Language	- Java - ASP.NET - Python - PHP - Ruby	- Java - Python - C - C++ - JavaScript	- C - C++ - Java - JavaScript - Python
Platforms	- AWS - Azure - Google Cloud Platform - IBM Cloud - Oracle Cloud	- Amazon Greengrass - EdgeX - Cisco IOx - Akraino Edge Stack - Eclipse ioFog	- AWS IoT - Azure IoT - GCP IoT - IBM Watson - Cisco IoT cloud connect

Table 2.1: List of OS, programming language and platform in IoT layers

The choice of OS in different layers of the IoT environment depends on the hardware properties such as memory and CPU. The *programming language* helps the developers to build various applications in different working environments with diverse constraints. The choice depends on the capability of devices and the purpose of the application [47]. The IoT software *platform* is a system that simplifies the development and deployment of the ML-based IoT application. It is an essential element of a huge IoT ecosystem which can be leveraged to connect new elements to the system. The details of the most popular OSs, programming languages, and platforms in IoT domain are in table 2.1 .

The heterogeneity of IoT infrastructures makes the deployment very complicated and difficult to automate. To overcome this issue, the infrastructure must be described and specified by machine understandable languages. Then, the declarative deployment systems are able to automatically map the ML models to the infrastructures and generate the deployment plans that optimize the performance and the accuracy.

The declarative TOSCA model [76] is able to specify the common infrastructures such as Raspberry Pis and cloud VM (hardware), MQTT and XMPP (communication protocol). The deployment logic can be defined through *TOSCA Lifecycle Interface* that allows users to customize the deployment steps. However, this declarative model is still very basic and can not handle complex deployments such as specifying the details of ML based application. Moreover, the IoT applications consist of installing devices and sensors which require *human tasks*. These tasks are not natively supported by any

available declarative deployment [41]. The imperative tool (e.g., kubectl commands) allows the technical experts with diverse knowledge of different deployment systems and APIs to interact with a deployment system and decide what actions should be taken. However, current imperative frameworks such as Juju, Kubernetes still do not support interactions such as sensor installation. In future, declarative deployment systems should interact with declarative ML systems to deploy a complex application over the heterogeneity of IoT infrastructure while supporting the *human* tasks through a more human centered imperative deployment model.

2.3.4 Deployment Optimization

When the infrastructures and deployment workflow of the ML models are specified, the deployment optimization problem can be formed as a mathematical expression subject to a set of system constraints. Then, resource allocation algorithms can be used to efficiently and precisely find the best solution for the given mathematical expressions. Moreover, the optimization objectives are a set of QoS parameters including storage and memory space, budget, task execution time and communication delay etc,. These algorithms can be divided into *two* classes based on whether an optimal solution can be guaranteed: *meta-heuristic method* and *iterative method* (or *mathematical optimization*). Nowadays, ML methods are becoming popular and being applied to solve these resource allocation problems by learning "good" solutions from the data. We investigate the representative works in resource allocation based on these *three* classes.

Iterative-based method. This class of algorithm generates a sequence of improved approximate solutions where each solution is driven by previous solutions. Eventually, the solutions will converge to an optimal point proved by a rigorous mathematical analysis. The heuristic-based iterative methods are also very common, but we categorize this type of algorithms into the *meta-heuristic based method*. The most popular algorithms of this class include newton's method [190, 191], gradient method [26] and ellipsoid method [185]. To apply and adapt iterative-based algorithms to optimize resource allocation requires strong mathematical background, which can be an obstruction for software developers to utilize these algorithms to optimize their deployment. Further-

more, the algorithms have the variety of performance for different problems in terms of efficiency and accuracy. As a result, more algorithms from iterative-based methods need to be studied and simplified by the system researchers, providing toolkits (or solvers) to tackle different optimization problems in IoT application deployment.

Meta-heuristic based method. The optimization problems in IoT applications can have large search spaces or be time-sensitive. The *meta-heuristic* based method is faster than *iterative-based method* in finding a near-optimal solution. This type of method consists of two subclasses: *trajectory*-based method and *population*-based method. The *trajectory*-based method finds a suitable solution with a trajectory defined in the search space. First, the resource allocation problems are mapped into a set of search problems such as variable neighborhood search, iterated local search, simulated annealing and tabu search. Then, the *meta-heuristic* algorithms are used to find the solutions. Many survey papers [111, 182, 262] have reviewed the algorithms applied for resource allocation in IoT, cloud computing, mobile computing. Additionally, *population*-based methods aim to find a suitable solution in the search space that is described as the evolution of a population of solutions. This method is also called evolutionary computation and the most well-known algorithm is the genetic algorithm. [320] investigates the resource allocation problems solved by evolutionary approaches in cloud computing.

Machine learning based method. ML based method is inspired by the ability of data to represent the performance and utilization of the contemporary systems. The ML based methods are used to build data-driven models that allows the target systems to learn and generate an optimized deployment plan. The proposed algorithms have been used to optimize various QoS parameters such as latency [193, 309], resource utilization [203], energy consumption [28] and many others. Zhang et al. [323] have given a comprehensive survey of the ML based methods used for resource allocation in mobile and wireless networking.

Deployment (or resource allocation) optimization problems have been studied for decades, and remain a huge legacy for overcoming the optimization problems in deploying ML-based IoT applications. Instead of developing new optimization algorithms, more efforts are required to model the complex optimization problems, in which the

system scale, conditions and diversity have been amplified significantly.

2.4 Edge-cloud Collaboration

Edge-cloud collaborative learning and inference has garnered significant interest across various research communities due to recent *privacy concerns* and *latency requirements*. For instance, when dealing with sensitive patient data, privacy is a paramount concern during the training and serving of AI models. This has motivated the development of federated learning, which distributes the training to local edge devices and updates the central cloud server, effectively reducing data transmission and addressing privacy concerns.

Moreover, knowledge transfer learning plays a crucial role in this ecosystem, facilitating the efficient transfer of knowledge between cloud and edge environments. This process accelerates model training and fosters collaborative intelligence among edge devices, ultimately resulting in enhanced model performance in the distributed setting.

The integration of distributed machine learning (§2.2.5) and edge-cloud collaboration leads to improved machine learning performance, responsiveness, and resource efficiency in distributed settings. In this section, we summarize the basics of federated learning (§2.4.1) and knowledge transfer learning (§2.4.2). We also highlight system design considerations that could benefit distributed learning in edgel-cloud computing paradigms (§2.4.3).

2.4.1 Federated Learning

In traditional distributed machine learning, the training usually happens in the cloud data center with aggregated training data generated by collecting, labeling, and shuffling raw data. The training data is thus considered *identical and independent distributed (IID)* and balanced. This facilitates the training process as one only needs to consider distributing the training task across various computation units and updating the model by aggregating all local gradient updates. However, this is not the case when IoT comes into play. The ML-based IoT applications differ from the traditional ML applications in that they usually generate data from heterogeneous geo-distributed

devices (e.g., user behavior data from mobile phones). These data can be privacy-sensitive as users usually prefer not to leak personal information, making conventional distributed ML algorithms infeasible for solving such problems. Thus novel optimzia-tion techniques are required to enable training in such scenarios.

Federated learning (FL) [155] is a type of distributed machine learning research that moves the training close to the distributed IoT devices. It learns a global model by aggregating local gradient updates and does not require the movement of the raw data to the cloud center. FederatedAveraging (FedAvg) [198] is a decentralized learning algorithm specifically designed for the FL. It implements synchronous local SGD [56] on each device with a global server averaging over a fraction of all the model updates per iteration. FedAvg is capable of training high-accuracy models on various datasets with many fewer communication rounds. Following this work, [155] proposed two approaches: *Structured updates* and *sketched updates* for reducing the communication cost, achieving higher communication efficiency. Further research addresses the privacy limitation of FL by Differential Privacy [199] and Secure Aggregation [33]. Finally, [32] delivers system-level implementation of FL based on previously mentioned techniques. It is able to train deep learning models with local data stored on mobile phones.

FL is still developing rapidly with many challenges remaining to be solved. On the one hand, FL shares similar challenges as in conventional distributed machine learning methods in terms of more efficient communication protocol, synchronization strategy as well as parallel optimzia tion algorithms. On the other hand, the distinct setting of FL requires more research preserving the privacy of training data, ensuring the fairness and addressing bias in the data. For a more thorough survey on details of FL, one can refer to [147].

2.4.2 Knowledge Transfer Learning

The knowledge learned from trained models can be transferred and adapted to new tasks, and it is especially helpful when limited data/labels are available. In this section, we introduce four types of knowledge transfer learning (KTL) approaches: Transfer learning, Meta-learning, Online learning, and Continual learning.

Transfer Learning [275] — transferring knowledge across datasets— is the most popular KTL approach. It trains a model in the source domain (with adequate data/labels, e.g., on ImageNet [81] for general visual recognition tasks), and fine-tunes the model parameters in the target domain to accommodate the new tasks (e.g., medical imaging analysis on rare diseases). The rationale behind is that low-level and mid-level features can be representative enough and thus shared across different domains. In this case, only the parameters related to high-level feature extraction need to be updated. This mechanism does not require a large amount of data annotation for learning reliable representation in the new tasks, which could be useful in cases when annotations are expensive (e.g., medical applications). **Meta Learning** [286] is another popular KTL approach; instead of transferring knowledge across datasets, it focuses on knowledge transfer across tasks. Meta learning means learning knowledge or patterns from a large number of tasks, then transfer this knowledge for more efficient learning of new tasks. When with continuous data streaming, it is also desirable to update the model with incoming data, and in this case **Online Learning** [122] can be used. However, it is difficult to model when the incoming data is from a different distribution or a different task. Most recently, **Continual Learning**[221] was proposed to address this issue. Not only can it accommodate the new tasks or data with unknown distribution, it can also maintain the performance on the old/historical tasks (i.e., no forgetting [148]), making it a practical tool for real-world IoT applications. These four KTL approaches are similar in concept yet have different use cases. Transfer/Meta learning are focused on knowledge transfer across datasets or tasks (irrespective of data types), while online/continuous learning are more suitable for data streaming and can transfer the knowledge continuously to the new incoming data or tasks.

2.4.3 Distributed ML systems

When we have acquired a local model update with partial data slice, multi-node and multi-thread collaboration are important for effectively updating the model. Network communication plays an important role in sharing information across the nodes. In this section, we present the three most important features in network communication: 1) network topology, 2) synchronization strategy and 3) communication efficiency.

Network Topology. The network topology defines the node connection approach in the distributed machine learning system. When the data and models are relatively simple, it is common to utilize existing Message Passing Interface (MPI) or MapReduce infrastructure for the training. Later when the systems are becoming more and more complex, new topologies should be designed to facilitate the parameter update.

The Iterative MapReduce (IMR) or AllReduce approaches are commonly used for synchronous data parallel training. Typical IMR engines (for example the Spark MLlib [201]) generalizes MapReduce and enables the iterative training required by most ML algorithms. Synchronous training can also be implemented by AllReduce topology. MPI[4] (Message Passing Interface) supports AllReduce and is efficient for CPU-CPU communication. Many researchers implement their own version of AllReduce for example Caffe2 Gloo[5], Baidu Ring AllReduce[6]. In the ring-Allreduce topology, all nodes connect to each other without a central server, just like a ring. The training gradients are aggregated through their neighbors on the ring. To provide more efficient communication for DL workload in the GPU cluster, libraries such as Nvidia NCCL [70] are developed and support the AllReduce topology. In NCCL2 [137], the multi-node distribution feature is also introduced. Horovod [251] replaces the Baidu ring-Allreduce backend with NCCL2 for efficient distribution.

A Parameter Server (PS) infrastructure [169] is usually composed of a set of worker nodes and a server node which gathers and distributes computation from worker nodes. As asynchronous training of PS neglects stragglers, it provides better fault tolerance capability when some of the nodes break down. Parameter server also features high scalability and flexibility. Users can add nodes to the cluster without restarting the cluster. Famous projects such as DMTK Microsoft Multiverso [94], Petuum [303] and DistBelief [79] enable training of even larger networks.

Synchronization Strategy. In distributed ML, model parameter synchronization between worker nodes is cost-extensive. The trade-off between the communication and the fresher updates has great impact on the parallelism efficiency.

Bulk Synchronous Parallel (BSP) [196] is the simplest strategy for ensuring model

[4]https://computing.llnl.gov/tutorials/mpi/
[5]https://github.com/facebookincubator/gloo
[6]https://github.com/baidu-research/baidu-allreduce

consistency of all worker nodes. For each training iteration, all nodes wait for the last (slowest) node to finish the computation and the next iteration does not start before the all the model updates are aggregated. Total Asynchronous Parallel (TAP) [79] approaches are proposed to address the problem of the stragglers within the network. With TAP, all worker nodes access the global model via a shared memory. They can pull and update global model parameters any time when the training is finished. As there is no update barrier for this approach, the system fault tolerance is greatly improved. However, stale model updates can not guarantee convergence to global optimum. Many famous frameworks use the TAP strategy, including Hogwild! [239] and Cyclades [220].

Stale Synchronous Parallel (SSP) [120] compromises between fully-synchronous and asynchronous schemes. It allows maximum staleness by allowing faster working nodes to read global parameters without waiting for slower nodes. As a result, the workers spend more time doing valuable computation, thereby improving the training speed greatly. But when there is too much staleness within the system, the convergence speed can be significantly reduced. Many state-of-the-art distributed training systems implement BSP and SSP for efficient parallelism, for example tensorflow [7], Geeps [75], Petuum [303].

In contrast to the SSP which limits the staleness of the model update, the Approximate Synchronous Parallel [125] (ASP) limits the correctness. In Gaia [125], for each local model updates, the global parameter is aggregated only if the parameter change exceeds a predefined threshold. This "significance" only strategy eliminates unnecessary model updates and is efficient in utilizing the limited bandwidth. However, the empirical determination of the threshold only considers the network traffic and is insufficient for dealing with dynamics in the IoT environment. [292] has addressed this problem by also considering resource constraints for efficient parallelism.

Communication Efficiency. Communication overhead is the key and often the bottleneck in distributed machine learning [170]. The sequential optimization algorithms implemented in the worker nodes require frequent read and write from the globally shared parameters which poses a great challenge on balancing network bandwidth and communication frequency. To increase communication efficiency, we can either

reduce the size of the model gradient (communication content) or the communication frequency.

Communication content. The gradient size between working nodes is correlated to both the model size itself and the gradient compression rate. We have reviewed four types of *model compression* techniques in §2.3.1 which are effective in reducing the overall gradient size. Hereby we focus on the techniques that compress the gradient before transmission, and discuss the gradient *quantization* and *sparsification*.

Gradient quantization differs from the weight quantization (§2.3.1) as the former compresses the gradient transmission between worker nodes while the latter focuses on faster inference via smaller model size. Works that reduce the gradient precision [78] have been proposed and 1-bit quantization [250, 269] is effective in greatly reducing the computation overhead. Based on the idea, QSGD [11] and Terngrad [298] consider stochastic quantization where gradients are randomly rounded to lower precision. Additionally, weight quantization and gradient quantization can also be combined [123, 131, 301, 324, 343] for efficient on-device acceleration.

The weights of the DNNs are usually sparse and due to the large number of unchanged weights in each iteration, the gradient updates are even more sparse. This sparsification nature of the gradient transmission has been utilized for more efficient communication. Gradient sparsification works by sending only important gradients when exceeding a fixed threshold [269] or adaptive threshold [90]. Gradient Dropping [9] uses layer normalization to keep the convergence speed. DGC [178] uses local gradient clipping for sending important gradients first while the less important ones are aggregated with momentum correction for later transmission.

Communication Frequency. Local (Parallel) SGD [197, 268, 315, 327, 346] entails performing local updates several times before parameter aggregation. Motivated by reducing the inter-node communication, this approach is also called model averaging. One-shot averaging [197, 346] considers only one aggregation during the whole training process. While [327] argues that one-shot averaging can cause inaccuracy and proposes more frequent communications, many works [177, 228, 314, 331] prove the applicability of the model averaging approach in various deep learning applications. In an asynchronous setting, the communication frequency can also be maneuvered

through the push and pull operations in the worker nodes. DistBelief [79] has adopted this approach with a larger push interval compared to the pull interval.

2.5 Edge Video Analytics (EVA)

Video analytics (VA) refers to the process of extracting valuable information from video data utilizing computer vision as well as other deep learning techniques. VA tasks includes recognizing patterns, identifying objects, and tracking movements to analyze and understand video content, enabling data-driven decision-making. VA finds applications in various domains, such as security and surveillance for monitoring camera feeds and detecting intrusions, traffic management for optimizing traffic flow and identifying incidents, manufacturing and industrial applications for identifying product defects and ensuring worker safety.

Edge video analytics (EVA), as defined in [345], entails performing analytics that leverages the computing resources across the end, edge and the cloud to maximize the performance of VA applications, i.e., the accuracy, latency, scalability of VA systems. EVA is a specialized area within the broader field of Video Analytics (VA) that concentrates on tackling the specific difficulties related to real-time video processing, data transmission, and system optimization. Current research in EVA primarily centers around enhancing the performance of systems and applications. This involves endeavors like creating streamlined Video Analytics Pipelines (VAPs) by eliminating unnecessary computations [306], maximizing resource utilization through load balancing [132], and improving execution efficiency through on-device workload scheduling [179].

2.5.1 EVA Application Architectures

The design of an EVA application heavily relies on factors such as scalability, flexibility, and specific system characteristics like the number of executors and resource availability. To enhance application performance and mitigate challenges in future maintenance, EVA architectures have undergone several transformations. These include transitioning from monolithic architectures to microservices architectures, and ultimately to serverless microservices architectures.

(1) Monolithic Architectures: Traditionally, applications followed a tightly coupled approach where all processes were bundled together as a single service, known as the monolithic architecture [77, 227]. In this architecture, if one module of the application faces high demand, scaling the entire architecture becomes necessary by provisioning a virtual machine (VM) to accommodate a new instance of the application. The monolithic architecture poses a risk to service availability because the interdependence and coupling among modules amplify the impact of a failure in a single module. Some early work on EVA adopts a monolithic architecture, such as vehicle detection [295], license plate recognition [53, 92], and traffic congestion analysis [288]. These services may suffer from slow responses, errors, or even downtime, due to the fact that the entire application is enclosed within a single codebase and must be deployed and scaled as a unified entity.

(2) Microservices Architectures: The microservices architecture, in contrast, involves constructing an application using a set of loosely coupled and finely grained microservices [77, 82, 214, 227]. These microservices can be developed independently using different programming languages, deployed separately on diverse infrastructures, and managed by distinct operational teams. They communicate with each other through well-defined and lightweight application programming interfaces (APIs). In the event of a surge in video feeds, the vehicle detection microservice can be scaled independently while the other microservices continue to function normally. The microservices architecture has gained significant popularity in EVA applications due to its adaptability and scalability. Examples such as the application pipeline of Microsoft Rocket [12, 50, 184] consists of a series of microservices, including decoding, background subtraction, light DNN object detector, heavy DNN object detector, and databases. Can be independently configured to meet different needs, executed across edge hardware and Azure cloud. And, the NVIDIA multi-camera intelligent garage application [211] divides pipes into edges and clouds. Edge decoding, detection, localization, tracking, and sending metadata to the cloud for event detection and data aggregation. With the increase of cameras, only edge microservices need to be expanded.

(3) Serverless Architectures: Serverless computing is a cloud computing model in

which the infrastructure required to run and scale applications is managed by a cloud provider [183]. Tasks such as load balancing, auto-scaling, fault tolerance, networking, and maintenance are handled by the cloud provider, allowing application developers to focus solely on writing code. In a serverless architecture, applications are decomposed into smaller, independent functions (often just a few lines of code) that are executed on-demand rather than continuously running on a server [25, 49]. These functions are triggered by specific events, such as an HTTP request or a database change, and the cloud provider allocates the necessary resources for their execution. Once the execution is complete, the resources are released. Typical serverless EVA applications include smart cars[29, 149], real-time security monitoring[223, 289], and traffic flow analysis and management in smart cities[91].

Serverless computing can be combined with a microservices architecture [24, 49], resulting in a widely adopted architectural approach known as a serverless microservices architecture [136]. In a serverless microservices architecture, a microservice is implemented as a set of event-driven functions. This architecture offers improved granularity and resource utilization. However, the choice between a microservices architecture and a serverless microservices architecture depends on the specific needs and requirements of an EVA application.

2.5.2 Techniques for optimizing the performance of EVA applications

2.5.2.1 Performance profiling:

The performance of an EVA depends on various configurations and workload placement strategies that trade off between different resource usages and analytics quality. Selecting an appropriate configuration can minimize resource demands while maintaining the desired level of analytics quality. Workload placement is also crucial in edge computing, as application performance can vary based on the availability of resources on different executors. Profiling [210] is a commonly used technique to gather information about program characteristics during execution, enabling analysis of the program's dynamic behavior. Profiling information, referred to as a profile, can be utilized to optimize configuration and placement strategies. Profiling can be conducted

either offline or online, depending on the specific requirements and constraints.

Offline profiling is a common approach used for obtaining an accurate profile, which involves running running EVA tasks multiple times with various inputs and configurations on all available devices. The profiler collects metrics such as accuracy, execution latency, resource usage, memory etc. Typically, a portion of the video data is labeled using high-quality configurations, which are representative of the target application scenarios, which is then used for profiling. However, the cost of profiling can be high due to the exponential growth of the search space as the number of parameters, such as configurations and placements, increases. For example, VideoStorm [325]'s offline profiler generates query resource-quality profile, which took 20 CPU days on a 10 minute video with 414 configurations. VideoEdge [132] considers a more complex profiling problem in hierarchical clusters, with a search space of 1800 combinations from different resolutions, frame rates, object detectors, trackers and placements plans. In order to accelerate the profiling process, VideoEdge merges common components among multiple queries and caches intermediate results.

Offline profiling faces the challenge of not capturing the diverse visual characteristics present in real-world scenes. This could result in decisions that do not account for the inherent trade-offs between resources and accuracy, leading to sub-optimal VA configurations. Content-aware profiling offers a promising solution to this problem by incorporating content features as additional dimensions in the profile. This approach involves training a prediction model based on the augmented profile, and then estimate the performance online. By considering video characteristics in the prediction, the model can adapt dynamically at runtime. ApproxNet [306] measures the frame complexity and temporal continuity on the edge devices, while ApproxDet [307] further employs video characteristics such as the number of objects, the sizes, and their movement speeds to meature their patterns of motion. THus, ApproxDet can better predict the accuracy and latency of different configurations and select the ones that reveals superiority. CEVAS [329] also measures the video content dynamics via metrics such as average number of objects for better profiling performance. The offline profiling approach, however, heavily depend on domain expertise for selecting features, and entails low generalizability. While they may work well in certain scenarios, they

may not perform effectively in others. Spatula [135] also leverages offline profiling for efficient cross-camera video analytics on large camera networks. By leveraging spatial and temporal correlations across cameras, the profiling search space has been greatly reduced, thus reduce the cost of cross-camera analytics.

Online Profiling addresses this issue by periodically updating the profiles for streaming EVA applications. Therefore, the main challenge of online profiling is how to reduce the computation overhead of the periodic profiling. Chameleon [142] employs online profiling to thoroughly profile all configurations and produce several candidate configurations. Through the use of cross-camera correlation and video content consistency, these candidate configurations are distributed and propagated across different spatial and temporal contexts. This strategy helps reduce the overall cost associated with extensive profiling. However, it's worth noting that this approach is relatively coarse-grained and relies on the establishment of a predefined time interval for conducting the profiling process. Awstream [321] integrates both offline profiling and online profiling in its approach, where online profiling gradually improves upon the initial bootstrap profile obtained from offline profiling. To enhance efficiency, Awstream selectively profiles a subset of configurations that are considered Pareto-optimal. A comprehensive profiling is triggered solely when additional resources become accessible, ensuring the current profile remains up to date.

2.5.2.2 Workload Scheduling:

In an EVA application, users can remotely submit various types of requests, such as queries or continuous video feeds. The service provider is responsible for fulfilling these requests and ensuring SLAs. To efficiently manage network, computation, and storage resources, the service provider must determine the optimal devices for handling these requests. One feasible approach is to implement the application in a microservice architecture that allows for independent deployment, scaling, and updating of each component. This architecture enables flexibility in workload placement and task offloading between different devices, enabling service providers to maximize resource utilization and revenue.

Optimized transmission: The common approach of offloading workloads from IoT

edges to on-premise edges or clouds is driven by the limited computing capacities of IoT edges. Nonetheless, blindly offloading every frame without considering redundancy can result in unnecessary overhead due to limited network resources. To mitigate these issues, the use of a filter [48, 59, 174, 293] to eliminate redundant frames prior to offloading is beneficial, along with techniques like frame compression or subsampling [277] to minimize communication overhead. Notably, solutions like CloudSeg [293] employs the streaming of low-resolution videos, which are then processed in the cloud to recover high-resolution frames with super-resolution techniques. Similarly, DDS [91] continuously sends low-quality videos to an edge server for inference, selectively resending regions with higher quality based on feedback from the server. VPaaS [326] introduces a video streaming protocol that optimizes bandwidth and reduces round-trip time (RTT) by transmitting low-quality video to the cloud. The system can then identify specific regions within video frames that require further processing at the fog ends. At the fog ends, a lightweight DNN model is used to correct any misidentified labels in these regions. Spider [176] presents a multi-hop millimeter-wave (mmWave) wireless relay network design that addresses the challenges of physical blockages in mmWave links. Spider combines a low-latency Wi-Fi control plane with a mmWave relay data plane to enable efficient rerouting around blockages. It also presents innovative algorithms for video bit-rate allocation and scalable routing, which prioritize maximizing video analytics accuracy rather than solely focusing on data throughput.

Analytics task offloading: An Video Analytics Pipeline (VAP) consists of multiple components with varying resource demands. For example video encoding and decoding are CPU-dependent while CNN-based object detection relies heavily on GPU resources. Partitioning the VAP into individual tasks and assigning them to suitable devices can enhance execution efficiency. This approach offers greater flexibility in how the pipeline is executed, enabling tasks to be distributed among different executors. By doing so, resource allocation can be optimized, leading to improved overall efficiency of the system.

For example, LAVEA [311] and CEVAS [329] investigates serverless VAP that executes different stateless functions across the edge and the cloud devices. To avoid unnecessary computation, MCDDN [110] and Mainstream [139] take into account the

possibility of combining shared tasks from different applications that operate on the same stream. If two applications require object detection in input frames, these tasks can be merged to eliminate redundancy. GEMEL [219] further extends the idea by "model merging" technique that shares layers and weights between similar architectures in edge models. This technique could reduce memory and swapping delay as compared to stem-sharing approach in Mainstream. Distream [318] is a distributed live video analytics system which dynamically adapts to workload changes, achieving low-latency, high-throughput, and scalable live video analytics by effectively balancing workloads between smart cameras and the edge cluster. EdgeDuet [310] combines local detection of large objects with offloading the detection of small objects to the edge. The primary focus is on reducing the latency associated with small object detection through tile-level parallelism. By optimizing the offloaded detection pipeline at the tile level instead of the entire frame, EdgeDuet achieves a balance between high accuracy and low latency. Zhou [342] and EdgeFlow [127] propose spatial partitioning of the feature maps for model parallel processing. On the other hand, REMIX [144] presents a image partitioning mechanism where an input frame is dynamically partitioned into multiple blocks, and an appropriate object detector is assigned to each block based on different characteristics.

2.5.2.3 Other research works

Continuous learning: Ekya [29], as presented in the paper by Bhardwaj et al. (2022), focuses on mitigating data drifting issues during inference, where deployed models encounter real data that deviates from the training data. By effectively managing the tradeoff between the accuracy of the retrained model and inference accuracy via a micro-profiler, Ekya tackles the challenge of simultaneously supporting inference and retraining tasks on edge servers.

Privacy-preserving VAP: With the growing importance of video-analytics-as-a-service in the cloud industry, maintaining the privacy of analyzed videos is a significant concern. Although trusted execution environments (TEEs) offer promise in preventing direct disclosure of private video content, they are still vulnerable to side-channel attacks. Visor [226] addresses these challenges by providing protection for user video

streams and machine learning models, even in the presence of compromised cloud platforms and untrusted co-tenants. By implementing a hybrid TEE that spans both the CPU and GPU, Visor effectively mitigates side-channel attacks resulting from data-dependent access patterns in video modules and also addresses potential leakage in the CPU-GPU communication channel.

2.6 Gaps and challenges in edge-cloud collaboration

The above literature review provides an overview of key research content related to cloud AI, edge AI, and edge-cloud collaborations. these research highlights the importance of edge-cloud collaborations for improving QoS performance in ML-based IoT applications. However, current research mainly focuses on the optimization of the models during the training process; there remains a huge gap in terms of the optimization of the models after development.

As shown in §2.1.1, a smart car navigator system comprises multiple ML models, including speech recognition, text classification, text generation and text-to-speech (TTS) model. Deployment of ML models in a pipeline requires proper composition plans to maximize the user QoS:

1) Action composition is defined by composing a set of basic actions for complex decisions. In a self-driving car operating system, actions can be accelerating, braking, turning left and right, etc. The combination of various action spaces increases the difficulties of learning optimal decisions in such complex systems. Hierarchical abstract machines (HAM) [261] are well studied in the context of reinforcement learning [222, 282] by allowing agents to select from a constrained list of action spaces, speeding up the learning and adaptation to the new environment.

2) Model composition aims to create an ML-based IoT application by using reusable, portable, self-contained modules via inserting new components or removing existing components. Apache Airflow [7] is an open-source platform for creating, scheduling and monitoring workflows in Python. The Valohai[8] operator is an extension of Airflow

[7]https://airflow.apache.org/
[8]https://valohai.com/

that utilizes the docker container to build self-contained modules for each model while providing flexibility for users to define the steps to execute.

The following gaps and challenges are highlighted for compositing the ML models in the edge-cloud computing paradigm:

- How to chain the dependent models. Each individual ML model has its own specification and data format of the inputs and outputs. The challenge is to design a data messaging system to orchestrate the data flow across different models while considering their required specifications and data format.

- How to allocate computing resources automatically for different models. An application chained by various ML models requires different computing resources across heterogeneous infrastructures, i.e., edge devices and cloud servers. It is challenging to provision computing resources efficiently for the chained ML models while meeting their performance requirements.

- In distributed and dynamic environments, how to effectively adapt the provision plans to ensure system instability. In such environments, it is challenging to design an algorithm that could interact with many devices and dynamic environments to generate an optimal strategy.

- How to monitor failure. The composed application consists of a set of ML models that needs to be monitored, ensuring that everything is executed as anticipated. However, ML models differ from traditional software in that ML models are highly stochastic, leading to possible disagreement in prediction results. It is challenging to define and detect such system failures.

- How to update the models to reduce system failure. It is challenging to design a cross-model learning algorithm to reduce the disagreement among the models. Also, how to leverage computation resources across different computation devices to speed up training is also a challenging topic.

2.7 Conclusion

In this section, I conduct a thorough survey on the hot topics of edge-cloud collaboration. Starting from an example of an ML-based IoT application, I highlight several famous types of Cloud AI algorithms that are commonly developed in this context. Then, I introduce Edge AI techniques that enable efficient on-device model optimization and deployment. I also list common edge-cloud collaborative computing paradigms in the literature and highlight that the optimization of model performance in terms of deployment and update remains a huge gap. The rest of the thesis aims to fill this gap in terms of enabling efficient deployment, and updates in ML-based IoT applications, via optimized edge-cloud Collaboration techniques.

3

OSMOTICGATE: Adaptive Edge-based Real-time Video Analytics for the Internet of Things

Contents

Summary

This chapter presents OsmoticGate, a video analytics task offloading framework that is capable of generating optimal workload balancing strategies, based on a Hierarchy Queue Model and a two-stage gradient-based algorithm. Experiments on both simulation and real-world testbed show that OsmoticGate can achieve low latency given different system configurations.

3.1 Introduction

The adoption of Internet-of-Things (IoT) devices, including set-top boxes, embedded computers, and mobile devices, have increased in the context of video generation, delivery, and processing applications. A case in point, each minute 300 hours of video are uploaded onto Youtube. As noted by ourselves [229] and others [115], deep learning technologies and its offshoots are becoming cornerstone for enabling IoT video analytics applications such as security surveillance [3], image object tracking [4], home or industrial building automation [2].

Although the computing power of IoT devices has improved significantly, more research is required in order to optimize the execution of deep learning models on the same. For example, there is a need to develop optimization techniques that can balance the IoT devices' resource constraint (e.g., processing power and memory) and deep learning model's complexity and performance (e.g., processing latency, accuracy). HUAWEI mate 10 pro only has 200MB memory for its *Neural Processing Units* (NPU) which is not enough to run many advanced deep learning models, such as Faster R-CNN [242]. Moreover, NVIDIA Jetson Nano can only process 5 video frames in each second by using ResNet model [216], far from meeting the requirements of real-time processing. This bottleneck will be dramatically amplified as the IoT data volume and velocity continues to grow exponentially.

Developing an offloading technique to balance the video analytics workload, driven by the edge and cloud resources' computing and processing capabilities, has evolved as promising approach [236, 277]. DeepDecision [236] is the first attempt that combines

the low power edge devices with more powerful cloud servers to execute deep learning model both locally and remotely. However, this approach does not support fine-grained offloading of video frames across edge and cloud within a single window, which may lead to inefficient resource utilization. The inefficiency is caused by the possible idle time of either edge devices or cloud servers when the window size is not well tuned (see more details in §5.2). FastVA [277] proposes a local frame buffer for fine-grained splitting of video data within a window. This solution is not suitable when number of frames within a window can change over time. For example, when cameras are configured to adaptive bitrate protocols. Furthermore, FastVA fails to coordinate video analytics workload across multiple edge nodes which may cause queuing delay in the cloud. We discuss this issue through a primary benchmark in §5.2. VideoPipe [247], VideoEdge [132], VideoStorm [325] and Chameleon [142] focus on scheduling and configuring the video analytics jobs (queries) on edge or cloud computing clusters. These approaches have two key limitations: (i) they may run out of capacity at the edge layer as most of these devices are resource constraint and (ii) unable to run complex deep learning models on resource constraint devices, which may be required for more complex IoT application scenarios (e.g., city level traffic modeling).

In the practical deployment of edge computing, an offloading policy needs to consider *three facts*: 1) *heterogeneity of edge node* – Each edge node has different processing rate based on current working situations (e.g., number of IoT devices to ingest data from). The proportion of the video offloaded to cloud should consider the computing and network load of each individual edge (e.g., CPU utilization, upstream link utilization). 2) *interplay among edge nodes and cloud servers* – All edge nodes may forward the video to the cloud simultaneously, without considering others' offloading policies. This may cause starvation on the cloud server where the video from some edge nodes may be delayed for processing. 3) *modern video streaming protocols adaption* – Video streaming protocols are essential for video delivery. They break video into small segments, send to target servers and reassemble them at destination. Video analytical framework needs to carefully adapt to this protocol, in particular needs to consider how varying number of frames within a segments impact the offloading policy.

Objective. Our approach OsmoticGate therefore considers the facts in practical

deployment of edge computing, and proposes a novel technique to uncover the influence of these facts to offloading policy design. In particular, we develop a hierarchical queue model to capture the heterogeneity of edge node, in which the resource constraints (e.g., computing capacity and network bandwidth) of each edge nodes are used to parameterize the *local queue* and the *global queue* performance models. Thereafter, we attempt to minimize the processing latency of each video stream to achieve the real-time analytics. Specially, we formulate our problem as a non-smooth, non-convex, constrained min-latency optimization problem. To efficiently find an approximate solution of this problem, we develop a two-stage gradient-based algorithm and compare it with the state-of-the-art (SOTA) solutions (e.g., FastVA, DeepDecision, HillClimb [245]). Our evaluation demonstrates the advantages of workload-based modeling and the proposed algorithms reduce 2 $\times$ latency compared with SOTA methods in 5G network.

The contributions of this paper are as follows:

- We develop a new hierarchical queue model to describe the system dynamics of a video analytic system in edge-cloud environment. The proposed model is adapted to bitrate-based video streaming and focus on modeling the processing latency and throughput of a video stream analysis system (§3.3).
- We formulate the task offloading problem as a non-smooth, non-convex and constrained optimization problem (§3.4), and propose a gradient-based algorithm to efficiently solve this problem (§3.5).
- We feed the model parameter through real-world benchmark and evaluate our algorithms by comparing SOTA method in simulated environment (§3.6).

Position of the work. Our framework is generic for modeling the workload of complex video analytic system while resolving the optimal workload balance problem. The focus of the proposed systems is to minimize the system latency while maintaining the throughput within the user-defined bounds. In the future, the framework can be easily extended to include new decision variables including accuracy, energy consumption, resolution, bit-rate as well as as the choice of deep learning models.

3.2 Background and Motivation

3.2.1 Edge-Cloud Computing Paradigm for Video Analytics

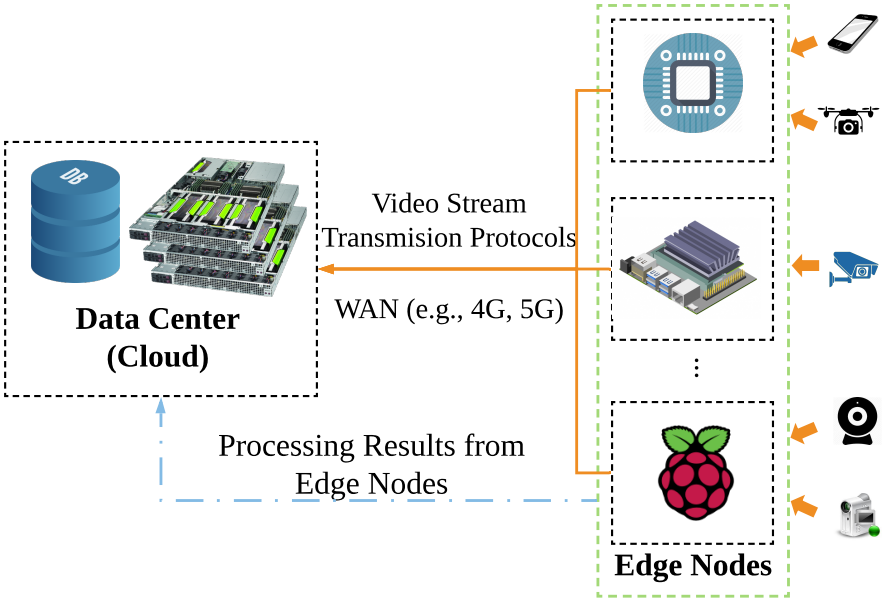
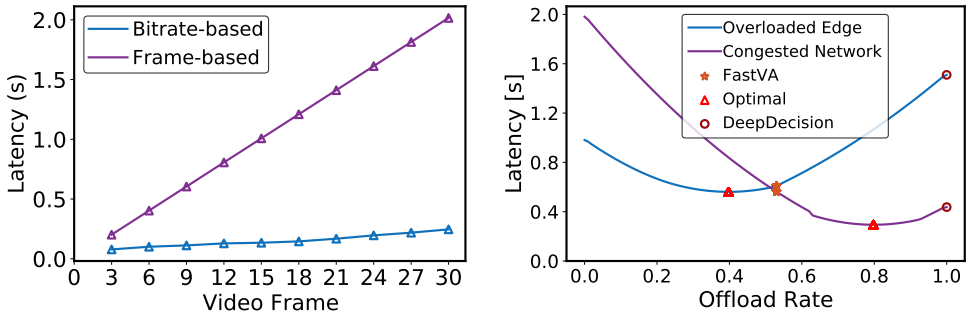


Figure 3.1: Video Analytics in Edge-Cloud Computing Paradigm

In this work, we target to provide a general solution for efficiently performing video analysis on the emerging edge-cloud computing paradigm [229] as shown in Fig. 3.1. In this computing paradigm, a set of video generating devices (e.g. traffic surveillance cameras, drones, mobile phones) are generating live video stream which can be processed either on low-power edge computing device (e.g. Raspberry pi, Jetson Nano, computing chips), or cloud data center with GPU cluster. Extensive research [240, 242] have been conducted in deep learning and adapted to assist video analytics. The communication among edge nodes and data center is via Wide Area Network (WAN), using various video stream transmission protocols.

3.2.2 Motivation

In order to efficiently perform video analysis on edge-cloud computing paradigm, the key is *when* and *how* to offload the video streaming to data center. In the following, we use a real-world traffic monitoring application and conduct two benchmark experiments to study the impact of the three complexities mentioned in §3.1.



(a) Transmission Latency Comparison (b) Heavy System Workload

Figure 3.2: What is Affecting the Performance of Cloud-edge Video Analysis System?

Setup. We deploy a retrained YOLOv3-tiny (detailed in §3.6.1) on both a Jetson Nano and a GPU server to count the number of cars in each video frame, with an emulated 5G network environment.

Bitrate-based V.S. Frame-based Video Transmission. We crop input video into segments containing 3 to 30 frames and deliver them from the edge to the cloud server using two types of transmission methods: 1) bitrate-based method that compresses the input video in H.264 format before transmission; and 2) frame-based method that transfers the video frames sequentially. As shown in Fig. 3.2(a), with the increase of video size, the difference of the transmission time between bitrate-based and frame-based method increases dramatically. The bitrate-based one is almost 10× faster than the frame-based one, when the number of frame is 30.

Understanding the System Workload is Essential. In this experiment, we inject video traffic for 60 seconds between edge node and cloud server while emulating following system dynamics: i.e., I) Heavy traffic between the edge node and server: we inject 10 seconds of video chunks to the communication queue beforehand; II) Heavy load in the edge: we inject 10 seconds of video chunks to the edge processing queue beforehand. Next, we benchmark two SOTA offloading techniques (FastVA and DeepDecision) and an exhaustive method. Fig. 3.2(b) shows that changing system workload impacts the optimal offloading rate. Moreover, we conclude that SOTA solutions are unable to achieve optimal offloading rate when subjected to system dynamics I and II.

Challenges. Bitrate-based video stream transmission is efficient and commonly used in real-world applications. However, adapting the video analytics pipeline to this

transmission protocol requires video encoding/decoding process and smooth feeding of the data into the analytics model. Hence, the first requirement (**R1**) is to study and design a mechanism for splitting the video streaming processing task to a granular level that allows efficient encoding/decoding, assisting optimal offloading performance tuning. The second requirement (**R2**) is to develop a novel model that automatically captures the dynamic workload in the proposed video analytics system. In the practical deployment, the edge node heterogeneity, network variation, as well as the interplay between the edge nodes and cloud servers can induce complicated and dynamic workload within the system. Finally, the third requirement (**R3**) is to design a new algorithm that adapts to the proposed model and optimizes the system performance.

3.3 System Model

In this section, we first show a new design that allows our OsmoticGate to adapt to bitrate-based video streaming, and then propose a *Hierarchical Queue Model (HQM)* to describe the system behaviors for video analytics. Finally, we model *two* key metrics for measuring system performance: *processing latency* and *system throughput*.

3.3.1 Adapting Bitrate-based Video Streaming

A video chunk is a segment of video stream which contains a sequence of video frames. In order to utilize the bitrate-based video streaming and meet the requirement **(R1)**, we split a time window of video into m chunks. We therefore are capable of offloading a proportion β of the chunks to server for processing.

On edge node k, each video is parameterised by a bitrate $b^{(k)}$ **encoding** as defined in Eq. 3.1.

$$b^{(k)} = \alpha^{(k)} * f^{(k)} * r^{(k)} * c^{(k)} \tag{3.1}$$

We use superscript k to denote metrics on edge node k. $f^{(k)}$ is the video frame rate, and $r^{(k)}$ is the resolution of a raw video frame. Next, $\alpha^{(k)}$ represents the bits required to represent each image pixel, $c^{(k)}$ is the compression rate with video encoding. Next, for each **split**, the start time and chunk duration are set. As video encoding must store key frame (i.e., I-frame) within the video as well as cross-frame information variation

at a fixed time interval, we locate I-frames as well as the group of pictures (GOPS)[1] within the duration. Different video encoding techniques can adapt $c^{(k)}$ and $\alpha^{(k)}$ to different settings. After this stage, video chunks can either be offloaded to cloud or processed locally. Finally, the **decode** is the process of converting video chunks into video frames. The extracted frames are processed by deep learning models where real detection happens.

3.3.2 *Hierarchical Queue Model (HQM)*

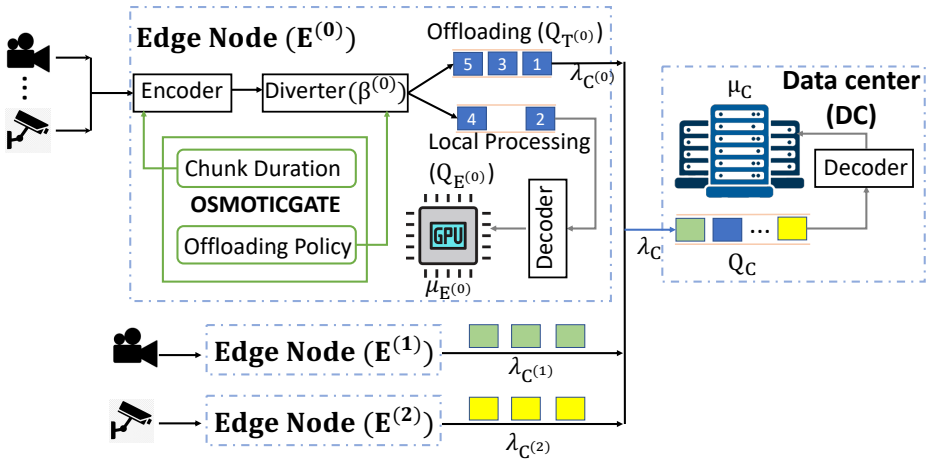


Figure 3.3: Hierarchical Queue Model in OsmoticGate

To meet **(R2)**, we develop HQM as in Fig. 3.3, which uncovers the system bottleneck in edge-cloud video processing system. Each edge node $E^{(k)}$ contains one offloading queue $Q_T^{(k)}$ and local processing queue $Q_E^{(k)}$. Q_C is the data center queue that receives the video chunks from any $Q_T^{(k)}$. All video in $Q_E^{(k)}$ and Q_C must be decoded into frames in **Decoder** before being processed.

The computed inputs of $Q_E^{(k)}$ and $Q_T^{(k)}$ are as follows:

$$\begin{aligned} \lambda_E^{(k)}(\beta_E^{(k)}) &= \beta_E^{(k)} * n^{(k)}/\Delta t \\ \lambda_T^{(k)}(\beta_E^{(k)}) &= (1-\beta_E^{(k)}) * n^{(k)}/\Delta t \end{aligned} \tag{3.2}$$

For $n^{(k)}$ chunks to be processed during time interval Δt, a proportion $\beta_E^{(k)}$ is placed at $E^{(k)}$. $\lambda_E^{(k)}(\beta_E^{(k)})$ and $\lambda_T^{(k)}(\beta_E^{(k)})$ are input rate of $Q_E^{(k)}$ and $Q_T^{(k)}$ respectively. As

[1] https://en.wikipedia.org/wiki/Group_of_pictures

they can contain decimal numbers, We round the inputs to the closest integer number throughout the following paper.

3.3.3 Latency Model

In this subsection, we aim to model the end-to-end latency of processing each video chunk. For example, there are n chunks of video that are injected to our system during interval Δt (including those already in the system). The latency is the average time of processing all n chunks. First, we assume that the size for each chunk is $s^{(k)}$ (bits/chunk). The processing latency $l_E^{(k)}$ for each video chunk on $E^{(k)}$ has positive correlation with $s^{(k)}$ as formalized in Eq. (3.3) and is assumed the same for chunks in the same $Q_E^{(k)}$:

$$l_E^{(k)} = a_E^{(k)} + b_E^{(k)} * s^{(k)} \tag{3.3}$$

where $b_E^{(k)}$ is a constant that defines relation between the processing latency and the size of each chunk. $a_E^{(k)}$ is the oscillation introduced by the underlying hardware such as the video decoding latency. Similarly, the processing latency $l_C(r)$ in the servers can be expressed as :

$$l_C = a_C + b_C * s^{(k)} \tag{3.4}$$

b_C indicates the coefficient between the processing latency and the size of a given chunk, and a_C is the underlying hardware influence.

Assuming uploading bandwidth between $E^{(k)}$ and DC is $B^{(k)}$, the average transmission latency per chunk $l_T^{(k)}$ is:

$$l_T^{(k)} = \frac{s^{(k)}}{B^{(k)}} \tag{3.5}$$

$l_T^{(k)}$ is the same for all chunks in $Q_T^{(k)}$.

Compute the latency of $Q_E^{(k)}$. Let $Size(Q_E^{(k)})$ denotes the number of video chunks in $Q_E^{(k)}$ at time t. The input rate of $Q_E^{(k)}$ is $\lambda_E^{(k)}(\beta_E^{(k)})$, linearly correlated with $\beta_E^{(k)}$ (see Eq. (3.2)).

Given system parameters, the queuing latency of i_{th} video chunk injected to $Q_E^{(k)}$ during Δt (with respect to the offloading rate $\beta_E^{(k)}$) is denoted by $T_{Q_E}^{(k)}(i, \beta_E^{(k)})$:

$$T_{Q_E}^{(k)}(i, \beta_E^{(k)}) = (\text{Size}(Q_E^{(k)}) + i) * l_E^{(k)} - \frac{i}{\lambda_E^{(k)}(\beta_E^{(k)})} \tag{3.6}$$

$(Size(Q_E^{(k)}) + i) * l_E^{(k)}$ indicates the timestamp that the i_{th} chunk is popped out from $Q_E^{(k)}$, $\frac{i}{\lambda_E^{(k)}(\beta_E^{(k)})}$ is the timestamp of the chunk arrived at the queue.

The sum of the processing latency for all items injected into the queue can be expressed in Eq. (3.7).

$$T_E^{(k)}(\beta_E^{(k)}) = \sum_{i=1}^{Size(Q_E^{(k)})} i * l_E^{(k)} + \sum_{i=1}^{\lambda_E^{(k)}(\beta_E^{(k)})\Delta t} \max\{T_{Q_E^{(k)}}(i, \beta_E^{(k)}), l_E^{(k)}\} \tag{3.7}$$

$T_E^{(k)}$ is the cumulative latency with two components: the first part is the latency for processing (including queueing latency) the remaining video chunks in $Q_E^{(k)}$, i.e., $i * l_E^{(k)}$ is the cumulative latency of popping the i_{th} chunk out of the queue; the second part is the cumulative time cost of popping the injected video chunks during time Δt out of the queue. The *max* term indicates that minimum processing latency must be greater than pure edge processing latency $l_E^{(k)}$ (processed instantly after entering the queue).

Compute the latency of $Q_T^{(k)}$. Similar computation applies to the *Transmission Queue*. Denote $Size(Q_T^{(k)})$ as the number of video chunks in the *Transmission Queue* at time t, the input rate is $\lambda_T^{(k)}(\beta_E^{(k)})$ (see Eq. (3.2)). The queuing latency for i_{th} chunk in the *transmission queue* is computed through:

$$T_{Q_T}^{(k)}(i, \beta_E^{(k)}) = (\text{Size}(Q_T^{(k)}) + i) * l_T^{(k)} - \frac{i}{\lambda_T^{(k)}(\beta_E^{(k)})} \tag{3.8}$$

The accumulated processing latency is computed in Eq. (3.9):

$$T_T^{(k)}(\beta_E^{(k)}) = \sum_{i=1}^{Size(Q_T^{(k)})} i * l_T^{(k)} + \sum_{i=1}^{\lambda_T^{(k)}(\beta_E^{(k)})\Delta t} \max\{T_{Q_T^{(k)}}(i, \beta_E^{(k)}), l_T^{(k)}\} \tag{3.9}$$

Compute the latency of Q_C. Assume that $Size(Q_C)$ represents the number of video chunks in the cloud queue at time t, $\mu_T^{(k)} = \frac{1}{l_T^{(k)}}$ indicates the output rate of $Q_T^{(K)}$, the offload rate of all edge nodes is $\boldsymbol{\beta}$:

$$\boldsymbol{\beta} := [\beta_E^{(1)}, \beta_E^{(2)}, ..., \beta_E^{(k)}] \tag{3.10}$$

During Δt, the total number of video chunks coming into each *transmission queue* is

denoted by $N_{Q_T}^{(k)}(\beta_E^{(k)})$:

$$N_{Q_T}^{(k)}(\beta_E^{(k)}) = \lambda_T^{(k)}(\beta_E^{(k)}) * \Delta t + Size(Q_T^{(k)}) \tag{3.11}$$

which adds up the injected chunks $\lambda_T^{(k)}(\beta_E^{(k)}) * \Delta t$ and the chunks already in the queue $Size(Q_T^{(k)})$.

Thus, the overall input rate $\lambda_C(\boldsymbol{\beta})$ of Q_C is defined by:

$$\lambda_C(\boldsymbol{\beta}) = \sum_{k=1}^{K} \lambda_C^{(k)}(\beta_E^{(k)}) = \sum_{k=1}^{K} \min\{\mu_T^{(k)}, \frac{N_{Q_T}^{(k)}(\beta_E^{(k)})}{\Delta t}\} \tag{3.12}$$

Input rate $\lambda_C^{(k)}(\beta_E^{(k)})$ from each node is limited by the minimum between the video transmission rate $\mu_T^{(k)}$ and average producing rate $N_{Q_T}^{(k)}(\beta_E^{(k)})/\Delta t$ from each *transmission queue*. $\lambda_C(\boldsymbol{\beta})$ sums up the output from all K transmission queues.

Again, we compute the queuing latency of i_{th} chunk arrived at Q_C by:

$$T_{Q_C}^{(k)}(i, \beta_E^{(k)}) = (Size(Q_C) + i) * l_C - \frac{i}{\lambda_C(\beta_E^{(k)})} \tag{3.13}$$

The cumulative processing latency of Q_C is:

$$T_C(\boldsymbol{\beta}) = \sum_{i=1}^{Size(Q_C)} i * l_C + \sum_{i=1}^{\lambda_C(\beta)\Delta t} \max\{T_{Q_C}^{(k)}(i, \beta_E^{(k)}), l_C\} \tag{3.14}$$

As a result, the cumulative latency for processing all video chunks generated during time Δt in the system is:

$$T(\boldsymbol{\beta}) = \sum_{k=1}^{K} [T_E^{(k)}(\beta_E^{(k)}) + T_T^{(k)}(\beta_E^{(k)})] + T_C(\boldsymbol{\beta}) \tag{3.15}$$

3.3.4 Throughput Model

We define system throughput as the number of chunks that can be processed within time Δt. It is total throughput for the proposed system.

Compute the throughput of $E^{(k)}$. Assume that the output rate of $Q_E^{(k)}$ is $\mu_E^{(k)} = \frac{1}{l_E^{(k)}}$, total number of chunks in $Q_E^{(k)}$ during Δt is:

$$N_E^{(k)}(\beta_E^{(k)}) = \lambda_E^{(k)}(\beta_E^{(k)})\Delta t + Size(Q_E^{(k)}) \tag{3.16}$$

which is affected by offloading ratio $\beta_E^{(k)}$.

The throughput of the $E^{(k)}$ is shown in Eq. (3.17):

$$I_E^{(k)}(\beta_E^{(k)}) = \min\{\mu_E^{(k)}, \frac{N_E^{(k)}(\beta_E^{(k)})}{\Delta t}\} \tag{3.17}$$

If the queued chunks are greater than the processing capacity $\mu_E^{(k)}$, the throughput $I_E^{(k)}$ equals $\mu_E^{(k)}$. Otherwise, $I_E^{(k)}$ equals the processing rate of available chunks, i.e., $N_E^{(k)}(\beta_E^{(k)})/\Delta t$.

Compute the throughput of Cloud DC. The number of chunks in Q_C at time t is constrained by cumulative number of chunks 1) injected into, 2) popped out from all $Q_T^{(k)}$, which are defined by $N_{Q_{Tin}}$ and $N_{Q_{Tout}}$ respectively:

$$N_{Q_{Tin}} = \sum_{k=1}^{K}(\lambda_T^{(k)}\Delta t) \text{ and } N_{Q_{Tout}} = \sum_{k=1}^{K}(\mu_T^{(k)}\Delta t) \tag{3.18}$$

The cloud throughput is affected by the output rate of Q_C and the total number of chunks in Q_C, which can be formalized by the following:

$$I_C(\boldsymbol{\beta}) = \min\{\frac{N_{Q_{Tin}}+Size(Q_C)}{\Delta t}, \frac{N_{Q_{Tout}}+Size(Q_C)}{\Delta t}, \mu_C\} \tag{3.19}$$

The first two terms indicate two upper limits for all available chunks, which includes transmitted chunks from $E^{(k)}$ and chunks already in the cloud queue $Size(Q_C)$. $\mu_C^{(k)} = \frac{1}{l_C}$ is the output rate of Q_C.

Finally, the system throughput model is:

$$I(\boldsymbol{\beta}) = \sum_{k=1}^{K} I_E^{(k)}(\beta_E^{(k)}) + I_C(\boldsymbol{\beta}) \tag{3.20}$$

3.4 Constrained Min-Latency Problem

3.4.1 Problem Formulation

To satisfy the requirement **(R3)**, we aim to minimize the system latency, while achieving the defined minimal throughput constraint. To simplify the problem, we fix the video encoding configurations and optimize the offloading parameters. Here we relax

the parameters $\beta_E^{(k)}$ and allow them to be any real numbers between 0 and 1 in order to fit our continuous optimization schemes. After we finish the optimization stage, we round them to the nearest values which will enforce $\lambda_E^{(k)}(\beta_E^{(k)})\Delta t$ to be integers at the end. Recall $\boldsymbol{\beta} := [\beta_E^{(1)}, \beta_E^{(2)}, ..., \beta_E^{(K)}]$, our formulation for the optimization objective reads:

$$\underset{\boldsymbol{\beta}\in\mathbb{R}^K}{\arg\min} \quad T(\boldsymbol{\beta}) \tag{3.21}$$

$$s.t. \quad C_1 : I(\boldsymbol{\beta}) \geq I^* \tag{3.22}$$

$$C_2 : \boldsymbol{\beta} \in [0,1]^K \tag{3.23}$$

C_1 represents that the system throughput must equal or greater than a predefined constraint I^*. C_2 denotes that the video offloading rates $\beta_E^{(k)}$ are restricted between $[0, 1]$.

3.4.2 Challenges in the optimization task

Constrained optimization. In our formulation, we enforce constraints C_1 and C_2 that regularize the decision boundary of the input variable $\beta_E^{(k)}$. Hence, the optimization algorithms must take the constraints into account and ensure feasibility and convergence simultaneously.

Non-smooth optimization with Lipschiz continuity. Our objective function $T(\cdot)$ is a Lipschitz-continuous function which satisfies:

$$\|T(x) - T(y)\|_2 \leq M\|x - y\|_2, \forall x, y \in \mathbb{R}^K \tag{3.24}$$

When $M < +\infty$, Eq. (3.21) is a non-smooth function globally, because its components have a piece-wise smooth structure of the form $\max\{f_1(\boldsymbol{\beta}), f_2(\boldsymbol{\beta})\}$. For example, the Eq. (3.7)) includes the term $\max\{(\text{Size}(Q_E^{(k)}) + i) * l_E^{(k)} - \frac{i}{\lambda_E^{(k)}(\beta_E^{(k)})}, l_E^{(k)}\}$, where we have $f_1(\boldsymbol{\beta}) = (\text{Size}(Q_E^{(k)}) + i) * l_E^{(k)} - \frac{i}{\lambda_E^{(k)}(\beta_E^{(k)})}$, $f_2(\boldsymbol{\beta}) = l_E^{(k)}$ which is non-differentiable when $f_1(\boldsymbol{\beta}) = f_2(\boldsymbol{\beta})$. Since both $f_1(\boldsymbol{\beta})$ and $f_2(\boldsymbol{\beta})$ are smooth functions, $T(\cdot)$ is smooth in most time when the following condition is met: $x \in \mathbb{R}^K$ which satisfies $f_1(x) \neq f_2(x)$, there exists a radius $\varepsilon(x) > 0$:

$$\|\nabla T(x) - \nabla T(y)\|_2 \leq L(x)\|x - y\|_2, \forall y : \|x - y\|_2 \leq \varepsilon(x) \tag{3.25}$$

where we denote $L(x) < +\infty$ as the local-smoothness parameter at the point x which is the smallest possible positive value to ensure Eq. (3.25) to hold.

Non-convex optimization. We observe that objective function $T(\boldsymbol{\beta})$ (see Eq. (3.15) is also non-convex. Since each component of $T_E^{(k)}(\beta_E^{(k)})$, $T_T^{(k)}(\beta_E^{(k)})$ and $T_C(\boldsymbol{\beta})$ includes a sum of non-convex functions. For example, in Eq. (3.7) we have $\sum_{i=1}^{\lambda_E^{(k)}(\beta_E^{(k)})\Delta t} \max\{(\text{Size}(Q_E^{(k)})+ i) * l_E^{(k)} - \frac{i}{\lambda_E^{(k)}(\beta_E^{(k)})}, l_E^{(k)}\}$, where $\lambda_E^{(k)} = \beta_E^{(k)} * n^{(k)}/\Delta t$, and then the first term in the max is a concave function in $\beta_E^{(k)}$, while $l_E^{(k)}$ is a constant. Hence all the elements in the sum is either non-convex or a constant function. As a result, It is a non-convex optimization problem that is a NP-Hard problem and in general is impossible to be globally optimized in polynomial time [86]. We hence aim to find a local minimum solution.

3.4.3 Problem Transformation

To solve the non-convex problem through a gradient-based method, in this section, we derive the adaptation of the objective function and the gradient accordingly. The adaptation discussed in this section will assist in derivation of gradient information as discussed in §3.5.

Adaptation of $\sum(\cdot)$. In Eq. (3.7), (3.9) and (3.14) we have the summation in the form of $\sum_{i=1}^{f_1(\beta_E^{(k)})}(i * f_2(\beta_E^{(k)}))$ which can not be directly used for computation against the offloading rate $f(\beta_E^{(k)})$. To adapt them into an expression with respect to the offloading rate $f(\beta_E^{(k)})$, such that gradient can be computed wherever possible.

For Eq. (3.7), we have $T_E^{(k)}(\beta_E^{(k)}) = a + \sum_{i=1}^{f_1(\beta_E^{(k)})} f_2(i, \beta_E^{(k)})$ where a is a constant, $f_1 = \beta_E^{(k)} * \Delta t$ and $f_2(i, \beta_E^{(k)}) = \max\{(\text{Size}(Q_E^{(k)}) + i) * l_E^{(k)} - \frac{i}{\lambda_E^{(k)}(\beta_E^{(k)})}, l_E^{(k)}\}$. We first extract the i from $f_2(i, \beta_E^{(k)})$ into the form $f_2(\beta_E^{(k)})$, so that:

$$\begin{aligned} f(\beta_E^{(k)}) \quad &= \textstyle\sum_{i=1}^{f_1(\beta_E^{(k)})} i * f_2(\beta_E^{(k)}) \\ &= \tfrac{1}{2}(1 + f_1(\beta_E^{(k)})) * f_1(\beta_E^{(k)}) * f_2(\beta_E^{(k)}) \end{aligned} \tag{3.26}$$

The gradient of $f(\beta_E^{(k)})$ is expressed as $g(\beta_E^{(k)})$:

$$g(\beta_E^{(k)}) = \quad (\tfrac{1}{2} + f_1(\beta_E^{(k)})) * f_2(\beta_E^{(k)}) + \tfrac{1}{2}(1 + f_1(\beta_E^{(k)})) * f_1(\beta_E^{(k)}) * f_2'(\beta_E^{(k)}) \tag{3.27}$$

To ease the computation of gradient, we smooth the value space of $\beta_E^{(k)}$ by transforming it from discrete to continuous space. We choose the closest discrete value after deciding the final offloading ratio $\beta_E^{(k)}$.

Relaxation of $\min(\cdot)$ **and** $\max(\cdot)$**.** The non-smoothness in our optimization function is introduced by $\min(\cdot)$ and $\max(\cdot)$ terms. We need to relax these terms to remove the non-smoothness. For example in $f_2(\beta_E^{(k)}) = \max\{f_3(\cdot), f_4(\cdot)\}$. where $f_3(\cdot) = \text{Size}(Q_E^{(k)}) + i) * l_E^{(k)} - \frac{i}{\lambda_E^{(k)}(\beta_E^{(k)})}$ and $f_4(\cdot) = l_E^{(k)}\}$. The relaxation can be implemented by:

$$\max(f_3(\cdot), f_4(\cdot)) = \begin{cases} f_3(\cdot) & f_3(\cdot) > f_4(\cdot) \\ f_4(\cdot) & f_3(\cdot) < f_4(\cdot) \end{cases} \tag{3.28}$$

With the system parameters and offloading rates, the value of $f_3(\cdot)$ and $f_4(\cdot)$ are determined, and thus the $min(\cdot)$ term is relaxed. By doing so, we can ensure that non-smoothness in the function is removed while not changing the problem statement of formulation. However, there is no expression when $f_3(\cdot) = f_4(\cdot)$. This problem needs to be resolved with our Alg. 2 (detailed in §3.5.3) . Similarly, $min(f_3(\cdot), f_4(\cdot))$ terms in Eq. (3.12) and (3.19) can be relaxed as the following:

$$\min(f_3(\cdot), f_4(\cdot)) = \begin{cases} f_3(\cdot) & f_3(\cdot) < f_4(\cdot) \\ f_4(\cdot) & f_3(\cdot) > f_4(\cdot) \end{cases} \tag{3.29}$$

Relaxation of L_C**.** To enable the adaptation of Eq. (3.26) in Eq. (3.14), we make the following relaxation. l_C is the latency of processing video chunks in server side. To simplify the computation, we assume that $l_C(r)$ is the same for any types of chunks from various edge nodes, i.e., $L_C = \sum_{k=1}^{K} L_C^{(k)}/K$.

3.5 Two-stage Algorithm Design

In the previous section we have presented the formulation of the constrained min-latency optimization objective. Now in this section we complete the algorithmic design by adapting the gradient-based optimization methods as the solver according to the characteristics of our objective. The proposed formulation of the optimization task and the adaptation of the gradient-based algorithms jointly form a core contribution of this work.

Why Gradient-based Algorithm. The gradient-based algorithms [213, 281], although do not have theoretical guarantees for finding the global optima, have been adapted to provide numerical solutions for many non-convex optimization problems. Motivated by the characteristics of our objective function, a two-stage gradient algorithm is proposed for solving Eq. (3.21). We use the gradient-based methods to find an approximate solution. The first motivation behind such choice is that the objective function's gradients can be efficiently computed ($O(\sum_{k=1}^{K} n^{(k)})$ floating point operations). Meanwhile as it is locally smooth almost everywhere, gradient-based methods can guarantee decrease at each iteration with suitable step-sizes and fast convergence rates.

We consider the gradient-based methods to be the most suitable choices for our optimization task. One may consider the zeroth-order methods such as the HillClimb, which does not exploit the gradient information (note that in our setting the gradient evaluation is as efficient as the function value query) and have poor convergence rates [267] especially in high dimensions. The higher-order methods such as the Newton-type methods, are also unsuitable for our tasks, since they typically need significantly more computational cost [38] for deriving the descent direction and handling the constraints compare to gradient-based methods. Moreover it is known that in non-convex settings, the benefit of higher-order oracles become very little [153], which makes the use of higher-order methods rather unnecessary.

Alternatively, one may consider adapting the stochastic gradient descent (SGD) methods [34, 280] to our task. Such methods compute efficient approximations of the gradients $\nabla T(\cdot)$ on randomly subsampled minibatches of the loss function $T(\cdot)$ as the

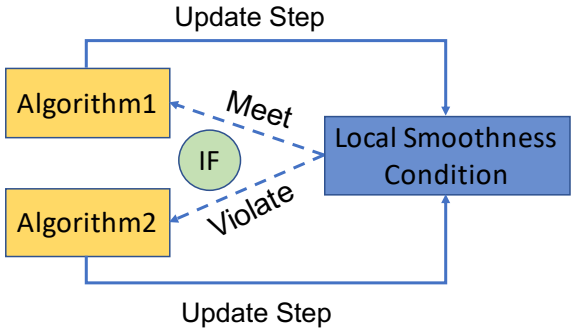


Figure 3.4: High-level Overview of Two-stage Algorithm

descent direction. The SGD methods are tailored for huge-scale tasks such as training deep neural nets on a large dataset. Since our optimization task is rather mild-scale (and also considering the practical limitations and downsides[2] of SGD discussed in [279]), we deem it as unnecessary to adopt stochastic gradient techniques at the moment, and leave the investigation for the practicality of SGD-type algorithms in our task as a future direction.

3.5.1 Overview of Two-Stage Gradient Algorithm

Denoting $\mathcal{Q}$ as the constraint set resulted by C_1 and C_2:

$$\mathcal{Q} := \{v \in [0,1]^K \mid I(v) \geq I^*\}, \tag{3.30}$$

we define the projection operator as the following:

$$\mathcal{P}_{\mathcal{Q}}(x) := \arg\min_{y \in \mathcal{Q}} \frac{1}{2}\|x - y\|_2^2. \tag{3.31}$$

The projection operator takes any point $x \in \mathbb{R}^K$ and returns its closest point within the constraint set. Then our algorithms for solving Eq. (3.21) can be generally expressed as the following iterative form:

$$\boldsymbol{\beta}_{j+1} = \mathcal{P}_{\mathcal{Q}}(\boldsymbol{\beta}_j - \eta_j \boldsymbol{\gamma}_j), \tag{3.32}$$

where η_j is the step size chosen at iteration j, while $\boldsymbol{\gamma}_j$ is some descent direction which seeks to decrease the function value $T(\cdot)$. The most simple and efficient choice is the

[2]Compared to the full gradient descent methods, SGD methods can have slower convergence rates in some scenarios/regimes [279], and require significantly more frequent calls on the projection operators [278]. Moreover, they are less compatible with the line-search schemes, and have less parallelizability [281].

gradient direction:

$$\boldsymbol{\gamma}_j = \nabla T(\boldsymbol{\beta}_j), \tag{3.33}$$

Since the objective function is globally non-smooth, for each step we need to choose a step-size η_j adapts to the local smoothness as described in Eq. (3.25). Therefore, our proposed algorithm consist of two stages as shown in Fig. 3.4. If the local-smoothness condition, a generalized sufficient decrease property, is met, the efficient algorithm PGD-VAO (§3.5.2) is applied for updating the steps. Otherwise, PGS-VAO (§3.5.3) is applied for updating the steps. Based on this switch, the algorithm converges after several iterations. The detail of how to switch between two algorithm is discussed in §3.5.4. Finally, we analyze the algorithm complexity in §3.5.6.

3.5.2 Projected Gradient Descent for Video Analytic Offloading (PGD-VAO)

We first present our projected gradient descent algorithm tailored for solving optimization task Eq. (3.21) in Alg. 1. In **Step 0**, we first compute the gradient at point $\boldsymbol{\beta}_j$ by transferring $\min(\cdot)$ and $\max(\cdot)$ through Eq. (3.28) and Eq. (3.29). Then, the gradient can be computed by Eq. (3.27). Since the local smoothness parameter defined in Eq. (3.25) at point $\boldsymbol{\beta}$ – the $L(\boldsymbol{\beta})$ is not known for each update but the practical step-size choice of gradient step is dependent on this parameter, we adopt the weak-wolfe line-search [165] scheme which estimates the local smoothness and allow us to use an adaptive step-size throughout iterations (see **Step 1**). Also, we discuss how to choose line search algorithms to improve the performance of PGD-VAO in §3.5.2.1. In **Step 2**, we compute next point y_{j+1}. Then, in **Step 3**, we check whether y_{j+1} meets the constraint, if not it will be returned to the closet point within the constraint set, denoted $\boldsymbol{\beta}_{j+1}$.

3.5.2.1 Choice of Line Search

In practice the theoretically ideal step-size of $\eta_j = O(1/L(\boldsymbol{\beta}_j))$ cannot be exactly computed since it is computationally intractable and unnecessary to get the exact value of $L(\boldsymbol{\beta}_j)$, hence the line-search schemes have been widely-adopted as numerical solutions. For instance, the back-tracking line-search scheme, being the most simple

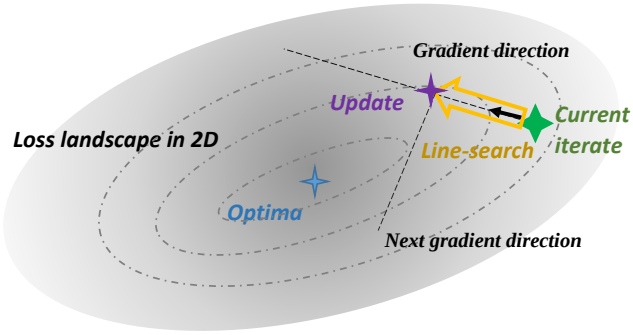


Figure 3.5: Illustration of the Weak-Wolfe line search mechanism, which in each iteration seeks a step-size to optimally decrease the objective function value and make sure that the next gradient direction to be as orthogonal as possible to current gradient direction.

yet widely-applied choice, can be expressed as finding step size η_j such that a sufficient decrease is numerically enforced:

$$T(\boldsymbol{\beta}_j - \eta_j \boldsymbol{\gamma}_j) \leq T(\boldsymbol{\beta}_j) - w_1 \eta_j \langle \nabla T(\boldsymbol{\beta}_j), \boldsymbol{\gamma}_j \rangle, \tag{3.34}$$

with parameter $w_1 \in (0, 1)$ configurable, while the (weak) Wolfe line-search scheme [165] uses an additional condition:

$$\langle \nabla T(\boldsymbol{\beta}_j - \eta_j \boldsymbol{\gamma}_j), \boldsymbol{\gamma}_j \rangle \leq w_2 \langle \nabla T(\boldsymbol{\beta}_j), \boldsymbol{\gamma}_j \rangle, \tag{3.35}$$

with a tunable parameter $w_2 \in [w_1, 1)$. The idea behind the Wolfe-type line-search is simple, as we can observe from above experession, that besides seeking a decrease in functional value, it enforce the gradient direction of the forthcoming iteration to be as orthogonal as possible w.r.t the current descent direction $\boldsymbol{\gamma}_j$ (which in the case of PGD-VAO $\boldsymbol{\gamma}_j = \nabla T(\boldsymbol{\beta}_j)$). We also illustrate in 2D the mechanism of such a line-search method in Figure 3.5. The Wolfe-type line search methods are known for providing a more accurate estimation of the ideal step-sizes compare to the backtracking line-search.

3.5.3 Projected Gradient Sampling for Video Analytic Offloading (PGS-VAO)

In this subsection we present our PGS-VAO algorithm tailored for solving our constrained optimization task as Alg. 2. The gradient sampling was proposed in [45]

Algorithm 1: Projected Gradient Descent for Video Analytic Offloading (PGD-VAO)

Input: Initial point $\boldsymbol{\beta}_0$ at which T is differentiable,
weak wolfe line search parameter w_1, w_2, constraints $\mathcal{Q}$, total number of iterations J
for $j = 1, 2, ..., J$ **do**
  Step 0: Compute the gradient
  $\boldsymbol{\gamma}_j = \nabla\, T(\boldsymbol{\beta}_j)$
  Step 1: Step length calculation
  $\eta_j = \text{line\_search}(\boldsymbol{\beta}_j, \boldsymbol{\gamma}_j, w_1, w_2)$
  Step 2: Update $y_{j+1} = \boldsymbol{\beta}_j - \eta_j \boldsymbol{\gamma}_j$
  Step 3: Projection $\boldsymbol{\beta}_{j+1} = \arg\min_{x \in \mathcal{Q}} \frac{1}{2}||x - y_{j+1}||_2^2$
end

whereby addressing the non-smooth property of the objective function.

Step 1: With sampled points in **Step 0**, we compute all gradients at these points following the same procedure as **Step 0** in Alg. (1) . Following the recent research in [44], we use the non-normalized search direction $-\boldsymbol{\gamma}_j$ as opposed to $-\boldsymbol{\gamma}_j/||\boldsymbol{\gamma}_j||_2$ in [45]. If the norm of the gradient $||\boldsymbol{\gamma}_j||$ is smaller than the optimality tolerance ν_j, we terminate this loop, reduce the sampling radius ϵ_j and optimality tolerance ν_j, continue next iteration.

Step 2: We numerically observe that the original backtracking line search applied in [44] does not provide consistent results. We hence replace backtracking line search by bisection line search satisfying Weak Wolfe conditions.

Step 3: After each update, if the resulting points is out of boundary $y_{j+1} \notin \mathcal{Q}$, we implement projection step 4, otherwise we continue next loop.

Step 4: We solve for each step, y_{j+1} that do not meet both C_1 and C_2 will be projected back to the points within the original variable space $\mathcal{Q}$ that are closed to the current position y_{j+1} in terms of euclidean distance and we obtain $\boldsymbol{\beta}_{j+1}$ to move to the next iteration.

3.5.4 Switching between PGD-VAO and PGS-VAO

In this subsection we discuss the condition that makes the PGD-VAO non-convergent and we have to switch to the PGS-VAO iterations. Once our algorithm reaches the point that the PGD iterations are able to make progress, our algorithm may switch

Algorithm 2: Projected Gradient Sampling for Video Analytic Offloading (PGS-VAO)

Input:
Initial point $\boldsymbol{\beta}_0$ at which T is differentiable, closed unit ball $\mathbb{B}$, maximum step N.
initial sampling radius $\epsilon_0 > 0$, sample radius reduction factor $\theta_\epsilon \in [0, 1]$, sample size $m \geq K + 1$.
Optimality tolerance $\nu_j \geq 0$. Optimality tolerance reduction factor $\theta_\nu \in [0, 1]$
weak wolfe line search parameter w_1, w_2, constraints $\mathcal{Q}$
for $j \in N$ **do**
  Step 0: Gradient Sampling
  Independently sample $\{\boldsymbol{\beta}_{j,1}, \boldsymbol{\beta}_{j,2}, ..., \boldsymbol{\beta}_{j,m}\}$ from $\mathbb{B}(\boldsymbol{\beta}_j, \epsilon_j)$;
  Step 1: Search direction calculation
  Compute $\boldsymbol{\gamma}_j$ as the solution of $\arg\min_{g \in \mathcal{G}} ||g||_2^2$, where
  $\mathcal{G} = \text{Conv}\{\nabla T(\boldsymbol{\beta}_{j,0}), \nabla T(\boldsymbol{\beta}_{j,1}), ..., \nabla T(\boldsymbol{\beta}_{j,m})\}$;
  if $||\boldsymbol{\gamma}_j|| \leq \nu_j$ **then**
    $t_j = 0$, set $\nu_{j+1} = \theta_\nu \nu_j$, $\epsilon_{j+1} = \theta_\epsilon \epsilon_j$;
    go to step 3
  else
    set $\nu_{j+1} = \nu_j$, $\epsilon_{j+1} = \epsilon_j$;
    go to step 2
  end
  Step 2: Step length calculation
  $\eta_j = \text{line\_search}(\boldsymbol{\beta}_{j,0}, \boldsymbol{\gamma}_j, w_1, w_2)$
  Step 3: Update $y_{j+1} = \boldsymbol{\beta}_j - \eta_j \boldsymbol{\gamma}_j$
  if $y_{j+1} \notin \mathcal{Q}$ **then**
    go to step 4;
  else
    set $\nu_{j+1} = \theta_\nu \nu_j$, $\epsilon_{j+1} = \theta_\epsilon \epsilon_j$;
    continue
  end
  Step 4: Projection $\boldsymbol{\beta}_{j+1} = \arg\min_{x \in \mathcal{Q}} \frac{1}{2}||x - y_{j+1}||_2^2$
end

back to PGD-VAO.

A theoretical analysis on the convergence of PGD-VAO. We start by presenting in Lemma 3.5.1 a generalized sufficient decrease property when the local smoothness holds, which is essential for the convergence of our algorithm. The proof strategy of this lemma is standard and follows similar steps as the sufficient-decrease result in [87] which was originally derived for globally-smooth composite objectives. We include the proof in the Appendix A for completeness. Next we will use this lemma to analyze when the projected gradient descent with practical line-search schemes will converge to a stationary point.

Lemma 3.5.1 (Local Sufficient Decrease Property) *Let $x \in \mathcal{Q}$ and $T(\cdot)$ is locally smooth*

around x with a radius $\varepsilon(x)$ such that:

$$\|\nabla T(x) - \nabla T(v)\|_2 \leq L(x)\|x - v\|_2, \forall v : \|x - v\|_2 \leq \varepsilon(x), \tag{3.36}$$

where we denote $L(x) < +\infty$ as local-smoothness parameter at the point x which is the smallest possible positive value to ensure (3.36) to hold, and $z = \mathcal{P}_{\mathcal{Q}}(x - \eta\boldsymbol{\gamma})$ with the step-size η is suitably chosen such that $\|z - x\|_2 \leq \varepsilon$, then for any $a > 0$ we have:

$$\begin{aligned} 0 \quad \leq \quad & T(x) - T(z) + \frac{1}{2aL(x)}\|\boldsymbol{\gamma} - \nabla T(x)\|_2^2 \\ & + [\frac{L(x)(a+1)}{2} - \frac{1}{2\eta}]\|x - z\|_2^2 \end{aligned}$$

We now apply the local sufficient decrease property by Lemma 3.5.1 to show that when local smoothness holds with a lower-bounded radius $\varepsilon(\boldsymbol{\beta}_j) \geq \varepsilon > 0$, the updating sequence $\boldsymbol{\beta}_j$ generated by Alg.1 converges to a stationary point. We start by defining the generalized gradient map at any position x as:

$$G(x) := \frac{1}{\eta}[x - \mathcal{P}_{\mathcal{Q}}(x - \eta\nabla T(x))]. \tag{3.37}$$

When the vector $x - \eta\nabla T(x)$ is in the constraint set $\mathcal{Q}$, it is clear that $G(x) = \nabla T(x)$.

Theorem 3.5.2 (Convergence of PGD-VAO under local-smoothness) *Suppose for all iteration j, the updates $\boldsymbol{\beta}_j$ admit local smoothness with a radius lower bounded as $\varepsilon(\boldsymbol{\beta}_j) \geq \varepsilon > 0$, $\eta_j = \frac{1}{2aL(\boldsymbol{\beta}_j)}$ for some $a > 1$, then the accumulative average gradient norm, $\boldsymbol{G}(J) := \frac{1}{J}\sum_{j=1}^{J}\|G(\boldsymbol{\beta}_j)\|_2^2$ converges at a rate $O(1/J)$:*

$$\boldsymbol{G}(J) \leq \frac{8aT(\boldsymbol{\beta}_0)\max_{j\in[J]} L(\boldsymbol{\beta}_j)}{J}, \tag{3.38}$$

and meanwhile $\|G(\boldsymbol{\beta}_J)\|_2^2 \to 0$ as $J \to +\infty$.

We provide the proof in Appendix B. Theorem 3.5.2 suggests that for the case where we have local-smoothness lower bounded above 0 throughout the iterations, the sequence generated by Alg.1 strictly converges to a stationary point of Eq. (3.21). However, when the iteration violates this condition, the gradient norm $\boldsymbol{G}(J)$ will be unbounded and we cannot guarantee convergence for stationary point at this case. This potential weakness of PGD-VAO motivates us to adopt the class of gradient sampling algorithms of Burke et al [44, 45] which is tailored for addressing such a non-convergent issue of line-search gradient descent methods in non-smooth optimization [3]. While the PGD-

[3]We refer the readers to [44, Figure 1] for an illustration of the non-convergent issue of gradient descent (with line-search) and how gradient sampling can overcome this.

VAO is computationally efficient, our PGS-VAO algorithm demands significantly more computation cost. Ideally we wish to run the efficient PGD-VAO algorithm whenever the local-smoothness holds with a non-decreasing radius.

Practical implementation. From our analysis we can see that it is easy to check in practice when should we switch from PGD-VAO to PGS-VAO. We can observe from the proof of Theorem 3.5.2 presented in the supplemental material, that only when $\eta_j = O(1/L(\boldsymbol{\beta}_j))$ can we derive the bound in Eq. (3.38). Meanwhile if the radius $\varepsilon(\boldsymbol{\beta}_j) \to 0$, then the local sufficient decrease cannot hold unless the step size must also shrink $\eta_j \to 0$. In the context of Theorem 3.5.2, it is equivalent to regard this case as $\eta_j = \frac{1}{2aL(\boldsymbol{\beta}_j)}$ with $a \to +\infty$, and hence the right-hand-side of (3.38) become unbounded: $\frac{8aT(\boldsymbol{\beta}_0)\max_{j\in[J]} L(\boldsymbol{\beta}_j)}{J} \to +\infty$, and at such case the PGD-VAO cannot have guaranteed convergence to stationary point and we need to switch to PGS-VAO.

Recall that we use a line-search algorithm to estimate the local-smoothness and adaptively determine the step-size. If we observe that the step-sizes given by the line-search scheme keep decreasing towards 0 for a number of iterations, then it suggests that sequence arrives at a regime where the local smoothness fails to hold. At such case we need to switch to PGS-VAO for further progress.

To be more specific, a most practical scheme for determining the switching point could be: first selecting a small step-size threshold η_a which is close to 0; then if the step-sizes chosen by the line-search scheme are below this threshold consecutively for a number of iterations, we may switch to the Alg 2. Similarly, after a few iterations of using costly Alg 2, if the step-sizes chosen by the line-search scheme are above this threshold, we may switch back to use the efficient PGD-VAO iterations.

Discussion. Although the gradient-based methods cannot guarantee convergence to the global optima for non-convex objectives in general, numerically we found that our algorithms consistently converge to well-performing solutions which are sufficient in practice. This phenomenon seems to suggest that our objective function is likely to be a well-behaved non-convex function, such that local minimas are almost as good as global optima. However, we plan to investigate this problem in the future at a greater depth.

3.5.5 Difference between projected gradient descent and projected gradient sampling algorithm

Projected gradient descent is an iterative optimization algorithm that aims to find the optimal solution within a constrained set of feasible solutions. It operates by iteratively updating the current solution in the direction of the negative gradient of the objective function while ensuring that the updated solution remains within the feasible set. This is achieved by projecting the updated solution onto the feasible set, effectively enforcing the constraints. The process continues until a convergence criterion is met or a maximum number of iterations is reached.

The projected gradient descent, however, can not be directly used in case of non-smooth objective functions as the function gradients at non-smooth points can not be directly obtained. When using projected gradient sampling for solving non-smooth optimization problems, it mainly focuses on generating samples from a probability distribution that reflects the underlying nonsmooth objective function. The gradient at non-smooth points can thus be approximated by approaches such as averaging over samples in close proximity or importance sampling methods. The two approaches differ mainly in that gradient sampling aims to generate representative samples from a distribution, enabling exploration and sampling tasks rather than convergence to a specific optimum.

3.5.6 The Complexity of the Algorithms

The complexity per iteration of the PGD-VAO algorithm is relatively low. To evaluate the gradient of $T_E^{(k)}(\beta_E^{(k)})$, $T_T^{(k)}(\beta_E^{(k)})$ and $T_C(\boldsymbol{\beta})$ in E.q 3.7, 3.9 and 3.14, the number of element-wise gradient evaluation is $\beta_E^{(k)} n^{(k)}$, $(1-\beta_E^{(k)})n^{(k)}$ and $\sum_{k=1}^{K} \min\{\mu_T^{(k)}, (1-\beta_E^{(k)})n^{(k)} + \text{Size}(Q_T^{(k)})\}$ respectively. Denote $\bar{n} = \frac{1}{K}\sum_{k=1}^{K} n^{(k)}$ (where K is the total number of edge nodes) we can see that the gradient evaluation takes $O(\bar{n}K)$ floating point operations, while the line search takes the same order of complexity and projection step takes $O(K)$ floating point operations, and hence the total complexity per iteration of PGD-VAO is $O(\bar{n}K)$.

However, the PGS-VAO algorithm is much more computationally expensive per itera-

tion. It first needs to compute at least $K+1$ gradients which cost $O(\bar{n}K^2)$, then solving the QP subproblem in step 1 takes $O(K^3)$ floating point operations. The complexity of PGS-VAO per iteration is $O(\bar{n}K^2 + K^3)$. Hence the PGS-VAO iterations are much more computationally expensive than PGD-VAO iterations.

3.6 Evaluation

In the evaluation, we first benchmark system performance of the cloud-edge video processing system (as shown in Fig 3.1) in a real world test-bed and then feed the benchmarked parameters to our model. Next, we evaluate the performance of OsmoticGate through simulations, and compare its performance with the SOTA solutions.

Assumptions: In this chapter, the streaming video analytics system is evaluated on a road traffic monitoring application. For real-time analytics applications as such, the latency requirement is stringent, less than 30 ms per frame [308]. The optimization goal in this chapter is thus minimizing processing latency to meet the real-time requirements. In order to realize such goals, we establish a testbed that comprises the edge and the cloud server. We assume both ends are equipped with GPUs for processing deep learning model workloads. The edge has limited computation power. The cloud has more computing power, but still has its limitation. This assumption is realistic as the cloud service such as Asure is on a pay-as-you-go basis. Based on that, we build a testbed with Jetson nanos and a workstation with 1080ti GPU as edge and cloud respectively.

3.6.1 Obtaining the parameters for HQM *via real-world benchmark*

Parameters. To make the HQM capture the system behaviors of the cloud-edge video processing system, we conduct a set of real-world benchmark experiments to obtain the modeling parameters. Specifically, in §3.3.2, we model the relationship between the processing latency of system components against the video size. Hereby we benchmark three sets of parameters including edge inference latency (Eq. (3.3)), cloud inference

latency (Eq. (3.4)) and network transmission latency (Eq. (3.5)). We also add 10% of oscillation to the network bandwidth to simulate real-world condition.

Environment Set-up. We use NVIDIA Jetson Nano (with ARM Cortex-A57 CPU and 4GB RAM) as the edge node and the cloud server is a bare metal Ubuntu machine, with 20 cores (Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz), GeForce GTX1080 Ti graphics card and 32 GB RAM. The network is configured by by using Linux traffic control (TC) as shown in Table 3.1, according to real world measurement[4].

Dataset and Deep Learning Model. We aim to achieve real-time road traffic monitoring which requires to detect the number of cars in each video frame. We therefore train a YOLOv3-tiny, a variant of the YOLOv3 [240] with 9 layers by using Jackson video dataset [48]. Since the dataset does not provide the labels, we extract image frames from the video and manually annotate 2500 frames for training purpose. The labeled dataset is available at [5]. The models are implemented with Darknet[5] and trained on our server. The trained YOLOv3-tiny is deployed on both server and edge. Moreover, TensorRT[6] is used to optimize the deployment on Jetson Nano for high-performance inference.

Standard MOTChallenge [7] dataset provides a rich collection of datasets for stimulating novel research and algorithms for optimizing the analytics performance in emerging tasks like cross-camera vehicle tracking. However, the aim of this is to reduce system processing latency by leveraging computation resource across the edge and the cloud. The task-offloading strategy developed in this chapter can be applied for different video processing workloads. For simplicity, we choose Jackson video dataset, the same as in FilterForward [48] that also aims to develop low-latency edge-cloud streaming video analytics application.

Benchmarking Results. Table. 3.2 shows the inference speed of the cloud server and edge node, we record the end-to-end latency from video decoding to completion of the inference procedure. We see that the inference latency increase linearly against the

[4]https://www.opensignal.com/reports/2018/04/uk/state-of-the-mobile-network
[5]http://pjreddie.com/darknet
[6]https://developer.nvidia.com/tensorrt
[7]https://motchallenge.net/

Network	Latency	Upload Bandwidth
2G	500ms	0.1Mbps
3G	80ms	1Mbps
4G	40ms	8Mbps
5G	20ms	20Mbps

Table 3.1: Emulated Network Configuration

chunk size. Also, cloud server has 10x of processing speed, compared that to edge node. We feed these benchmarked parameters to model in Eq. (3.3)and (3.4).

Also, we record end-to-end transmission latency under difference network environments. With the benchmark results, we can approximate the relation between the chunk size and transmission latency as formulated in Eq. (3.5).

Chunk Size to Chunk Duration. As illustrated in Eq. (3.3), (3.4) and (3.5), we are interested in the relation between the latency and the chunk size. However, chunk size is not directly configurable with FFMpeg processing. We resolve this by mapping the chunk duration to chunk size as shown in Table 3.2, such that all experiments can be implemented with chunk duration as a control variable. Our benchmark result indicates that chunk size maintain a linear relation with the chunk duration, and that the processing latency fits our modeling proposition in Eq. (3.3), (3.4) and (3.5).

Chunk Duration (s)	**Chunk Size (KB)**	**Cloud Inference Latency(s)**	**Edge Inference Latency (s)**	**Transmission Latency (s)**			
				2G	**3G**	**4G**	**5G**
0.1	315	0.065	0.497	15.62	1.565	0.196	0.114
0.2	330	0.08	0.698	19.81	2.013	0.252	0.142
0.3	341	0.095	0.923	21.98	2.242	0.28	0.162
0.4	348	0.11	1.143	25.41	2.573	0.322	0.183
0.5	352	0.125	1.353	26.11	2.683	0.336	0.191
0.6	358	0.14	1.552	28.44	2.916	0.364	0.211
0.7	365	0.165	1.751	31.9	3.363	0.42	0.242
0.8	371	0.17	1.952	38.11	3.921	0.49	0.281
0.9	378	0.185	2.151	41.13	4.364	0.546	0.313
1.0	384	0.20	2.354	47.43	4.923	0.616	0.355

Table 3.2: Testbed Benchmarking

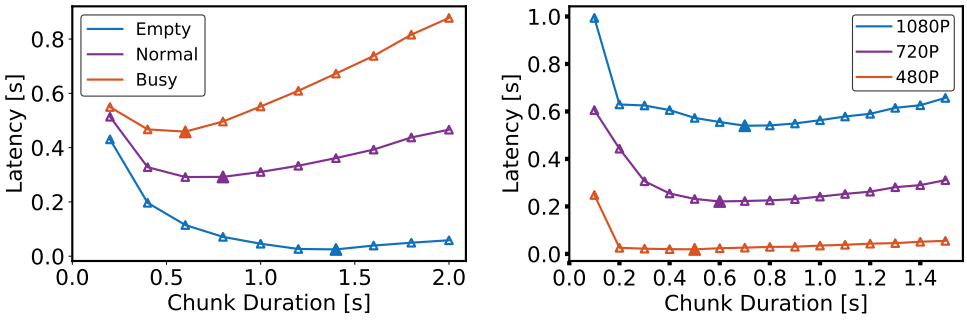
Additionally, the video which is encoded with lower bitrate, can have less latency and bandwidth cost as well as higher throughput, compared to that is encoded with higher bitrate. However, the lower bitrate may lead to lower model accuracy. In order to mitigate the accuracy loss while maximizing the system throughput, video encoding configuration has to be carefully designed.

Bitrate (kb/s)	Precision	Recall	F1
1000	0.9626	0.9338	0.9433
500	0.9661	0.9247	0.9400
250	0.9670	0.9052	0.9286
100	0.9607	0.7358	0.8121

Table 3.3: Model Accuracy under Various Bitrates and the Video Resolution is 1080P.

In the following, we evaluate the relation between model inference accuracy and bitrate. DeepDecision [236] has evaluated that resolution does not impact greatly on model inference accuracy. The model prediction accuracy is computed against the annotated car objects in the video frame. The overlapping between the detection and annotation bounding boxes is computed using IoU (Interaction of Union) metrics. We identify one detection as true positive when computed IoU metric exceeds a threshold (0.7 in out setting). As both the prediction accuracy and coverage are important, we consider F1 score as metric to evaluate model performance. Table 3.3 shows that the prediction precision does not decrease too much with the reduction of bitrate. Note that the accuracy decrease dramatically when the bitrate is lower than 100kb/s. In this paper, we set the bitrate as 1000kb/s for the rest of experiments. We are aware that accuracy of YOLOv3-Tiny could be inferior to YOLO. However, for our developed real-world dataset, yolov3-tiny already achieves acceptable accuracy. This work focus on minimizing workload inference latency, we consider YOLOv3-Tiny throughout the experiment. Developing methodologies to benchmark and improve the accuracy of these models is subject to future work. In future work, an resource aware encoding method that tunes the bitrate automatically based on the available resources can be plugged into OsmoticGate.

Determine the Chunk Duration. As discussed in §3.3.2, We adapted video analytics task offloading based on bitrate video streaming. The following evaluates the impacts of the video split granularity (i.e., chunk duration) to the system performance. To this end, we first configure the input rate of video stream and network as 30 frame/s and 5G, and consider *two* scenarios below: 1) *Varying system workload*: We set three system status, i.e. no workloads (*empty*), normal workloads (*normal*) and heavy workloads (*busy*) across the whole system. We also encode video bitrate and resolution at 1000kb/s and 1080P, receptively. 2) *Varying video resolution*: we adjust the resolution



(a) Varying System Workload (b) Varying Video Resolution

Figure 3.6: Latency V.S. Chunk duration

of input video among 1080P, 720P and 480P, and set the system in normal condition. In both scenarios, we vary the chunk duration from 0.2 second to 2 second and report its influence to latency.

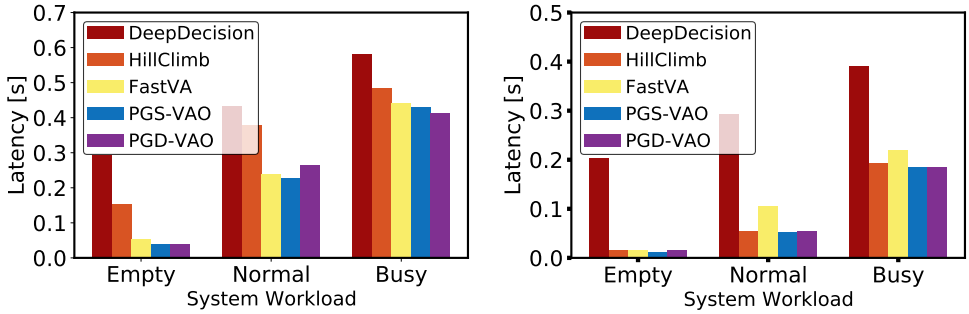
Fig 3.6(a) shows that the optimal chunk duration decreases with the increase of the system workload, i.e., the optimal chunk duration for busy, normal and empty are 0.6 seconds, 0.8 seconds and 1.4 seconds, respectively. In Fig. 3.6(b), our finding is that the smaller the video resolution, the smaller optimal chunk duration. Due to the fact that the lower video resolution requires less time to process each video chunk, the whole system configuration should be reduced as well for achieving smaller system latency.

Another finding from Fig. 3.6 is that when the video chunk is too small, the latency increases dramatically. It is because only a few frame is included in a chunk and the benefit of video compression is not sufficiently utilized. Also, more computing resources are wasted in encoding and decoding.

In the following evaluations, we use the obtained optimal settings of chunk duration. For more complex environment, in future, we can train reinforcement learning models to decide the optimal chunk duration automatically [193].

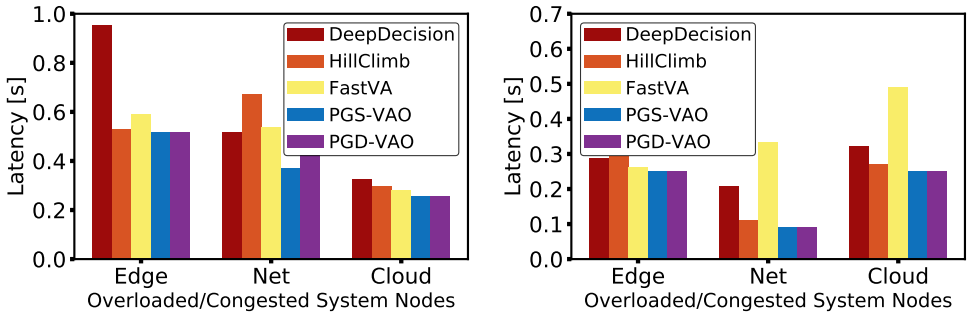
3.6.2 Simulation

Configuration. We configure the system parameters according to Table 3.2, the input rate of each edge node is 30 frame/s, and video resolutions are 1080P, 720P and 480P, respectively. The network is the same as benchmarking experiments (see Table 3.1).



(a) Uniform System Workload under 3G Network

(b) Uniform System Workload under 5G Network



(c) Imbalanced System Workload under 3G Network

(d) Imbalanced System Workload under 5G Network

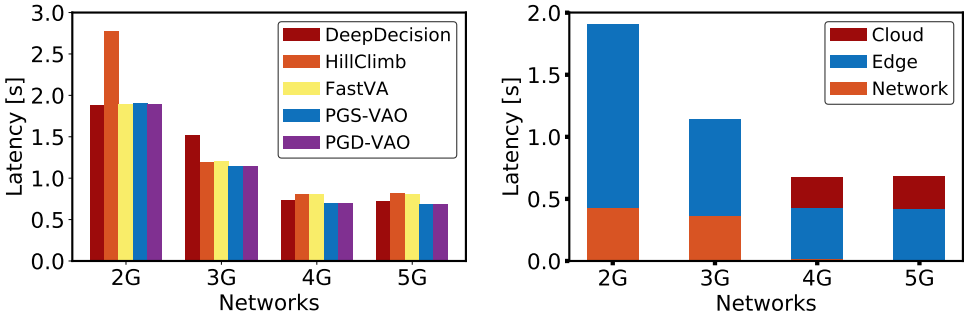
Figure 3.7: Performance under Different System Workloads

The system workload is set as no workloads (*empty*), normal workloads (*normal*) and heavy workloads (*busy*), with 0, 5 and 10 video chunks in the respective queues. Time interval Δt is 30 seconds for all experiments.

Evaluation Metric. We mainly consider two metrics in our evaluation: 1) *Latency* – the average latency of processing each video frame; this include the data transmission time and processing time either on edge or cloud. 2) *Throughput*–the number of the video frame processed in each second.

Algorithms. Although we proposed a two-stage algorithmic strategy to overcome a potential issue that the PGD iterations could converge at non-stationary points, our extensive numerical experiments do not observe such a case. Hence we compared our algorithms PGD-VAO and PGS-VAO separately with *three* baseline solutions below. We implemented and parameterized these baseline techniques based on our system configurations.

- DeepDecision: considers the optimization problem that during each time interval, video is processed only on the edge side or the cloud side depending on system



(a) Algorithm Comparison

(b) System Latency Proportion (PGS-VAO)

Figure 3.8: Performance under Different Network Bandwidth

throughput.

- FastVA: considers making most use of network transmission to offload the video chunks to the cloud.
- HillClimb: utilizes the Hill Climbing [245] algorithm to solve our optimization problem.

3.6.3 Comparison With Existing Approaches

In this subsection, we analyze the algorithm performance under various system conditions. For all experiments, optimal chunk duration is chosen based on the benchmarking results as shown in Fig. 3.6. We configure the number of edge node , cloud server and video resolution as 10, 1 and 720P respectively, if not otherwise stated.

3.6.3.1 The Impact of Network Bandwidth

Fig. 3.8(a) shows that with the increase of the network bandwidth, the latency is greatly reduced. In 3G condition, the latency is 2× of that in 4G and 5G. However, when network is 4G and 5G, there is no obvious performance difference. It is because more data are transmitted to the cloud when the network speed increase, more computing resources from cloud server is utilized until it is saturated.

In order to better understand the trends, we show the latency introduced by different components using PGS-VAO, according to our HQM in Fig. 3.8(b). In 2G and 3G network, the latency is introduced by data queuing for processing in the edge nodes and

transmitting to the cloud. As the improvement of network, our algorithm can adapt to this change, offloading more data to the cloud to utilize the computing resources from cloud, thereby reducing latency.

3.6.3.2 The Impact of System Workload

We first test the general performance of the algorithms with uniform system workloads, i.e., *empty*, *normal* and *busy* workload for all system queues. Then, we tune the workload imbalance by setting overloaded edge and cloud nodes, and congested network conditions respectively. We set these nodes as *busy* while keeping others as *normal*. We conduct the experiments in both 3G and 5G networks.

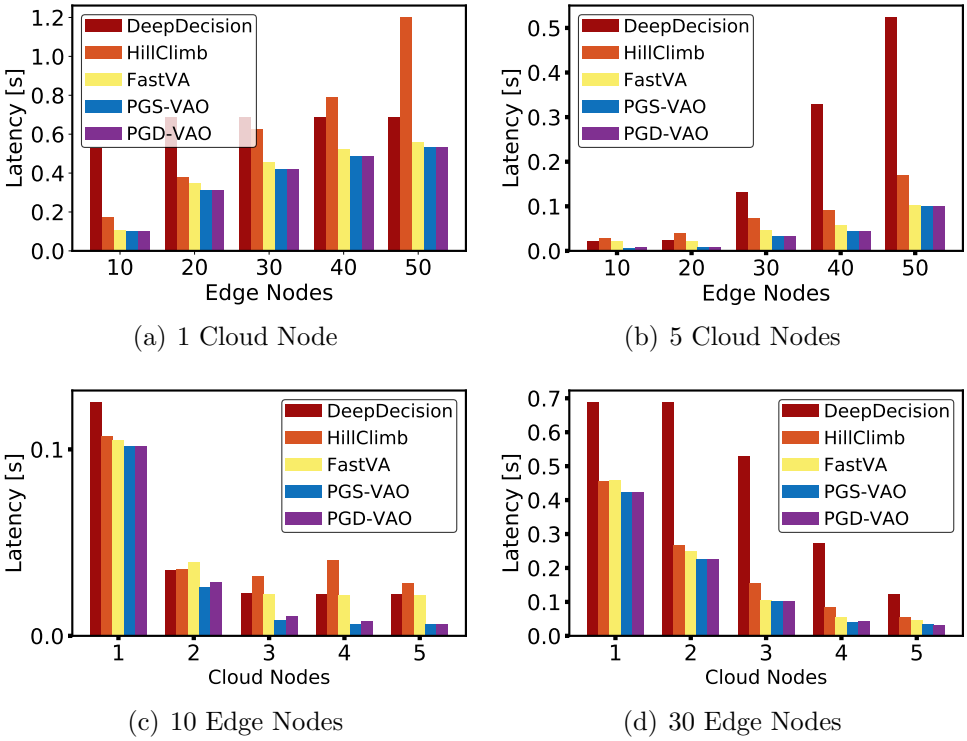
Fig. 3.7(a) and 3.7(b) indicate that with the increase of the workload, the system latency is increasing as well. It is also obvious that HQM-based algorithms (e.g., PGS-VAO, PGD-VAO) perform better than non-HQM-based algorithms (e.g., FastVA), which performs better in 5G (see Fig. 3.7(b)).

Fig. 3.7(a) illustrates that our proposed algorithm can capture the the system bottle which is the network speed and understand that maximizing the network utility is the best strategy. FastVA shares the similar offloading strategy, thereby achieving similar performance compared to our algorithms. However, FastVA cannot adapt to the change of system status, using the same offloading strategy in 5G network (as shown in Fig. 3.7(b)). Since the network is not the bottleneck, the ideal offloading policy should based on the processing capacity of both edge nodes and cloud server.

To further highlight the advantage of our algorithms, we imbalance the workload on different components modeled by our HQM. In particular, when the bottleneck is on the cloud/network, i.e., a lot of data is queuing for either transmitting or processing on the cloud, both FastVA and DeepDecision fail to make right offloading decision. In 5G network (see Fig 3.7(d)), the situation is magnified because FastVA is able to push more data to cloud server, without considering the queued data on network or cloud.

3.6.3.3 The Impact of Computing Resources

In this experiment, we evaluate the performance of the algorithms with different computing resources. We first fix the number of cloud servers from 1 to 5 and vary the



(a) 1 Cloud Node

(b) 5 Cloud Nodes

(c) 10 Edge Nodes

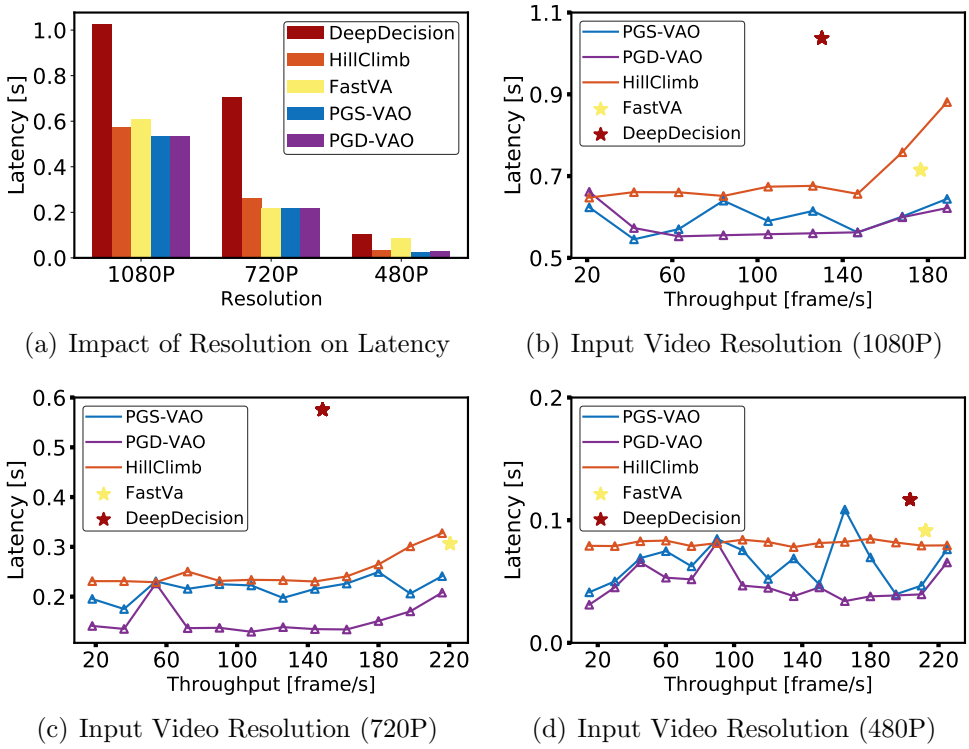
(d) 30 Edge Nodes

Figure 3.9: The Latency with Various Edge Nodes and Cloud Servers

number of edge nodes from 10 to 50 to report the system latency. Then we set edge nodes as 10 and 30 and vary the cloud nodes from 1 to 5. The experiment is conducted in 5G network, and results are shown in Fig. 3.9.

The findings are two-folds. First, from Fig. 3.9(a) and 3.9(b), we can see that the cloud processing capability has great impact on the overall system processing latency. When there is only one cloud node, the processing latency is approximately 5 $\times$ than the system with 5 cloud nodes. However, if the capacity of cloud server become sufficient, adding more cloud resources will not affect the overall system processing latency (see Fig. 3.9(c) and 3.9(d)). When the ratio of number of edge node and cloud server is 10:1, the system latency can be reduced to less than 0.1s. The latency reaches 0.2s when the ratio is around 15:1. This is also revealed in Fig. 3.9(c) and 3.9(d), when the ratio exceeds 15:1, the overall latency increases dramatically. This indicates that one cloud server is not sufficient for supporting more than 15 edge nodes. In order to scale up the system, appropriate cloud servers should be added with the increase of the edge nodes .

Moreover, our proposed method always performs better than baseline methods in any



(a) Impact of Resolution on Latency

(b) Input Video Resolution (1080P)

(c) Input Video Resolution (720P)

(d) Input Video Resolution (480P)

Figure 3.10: Impact of Throughput Constraint on System Latency with Varying Resolution

condition. There is no significant performance difference between PGS-VAO and PGD-VAO, but as discussed earlier, PGS-VAO is more stable in dealing with non-smooth value functions, it is still worth to use PGS-VAO when the computation reaches break point.

3.6.3.4 The Impact of Video Resolution

In this subsection, we vary the video resolution from 1080P to 480P and show its impact on latency in Fig. 3.10(a). Our proposed algorithm outperform all other baseline methods. In general, the latency of 720P and 480P video is about 1/2 and 1/6 of that in 1080P video, near linear reduction with resolution decreasing. The reduction of the latency is caused by two reasons: 1) less data needs to be transmitted 2) faster inference time. Video resolution also affects the system throughput and we will discuss it in next subsection.

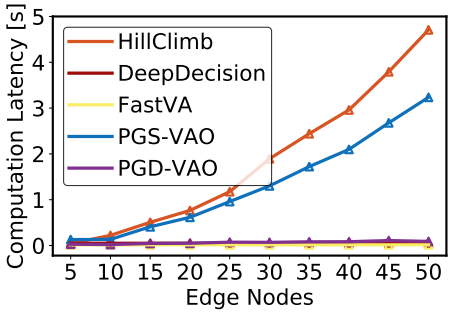


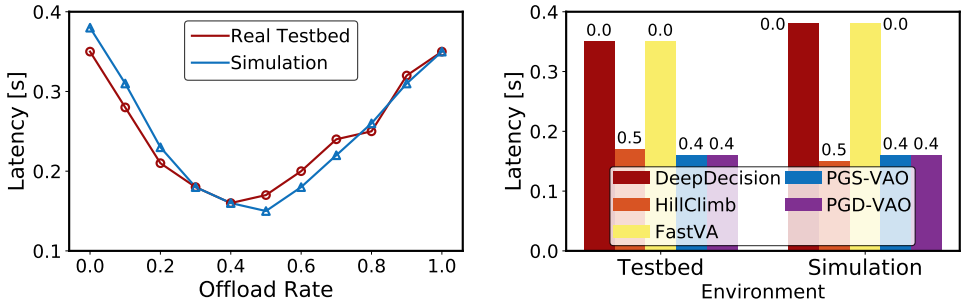
Figure 3.11: Algorithm Computation Latency with Different Edge Nodes

3.6.4 Impact of Throughput Constraint

Recall E.q 3.21, our optimization problem is to minimize the system latency which is bounded by a pre-defined throughput threshold (I^* in C_1). In order to study the impact of the I^* on latency, our experiments are conducted under different video resolution, while we adjust the throughput constraint with different values.

Fig. 3.10(b), 3.10(c) and 3.10(d) shows that our proposed algorithms can archive less latency, compared to other methods, and ensure the latency is fluctuating in a certain range with the varying of throughput constraints. This fluctuation is caused by the network oscillation that we add into the simulated network environments. HillClimb is implemented based on HQM, but it fails to obtain the optimal solution in some cases. For example, Fig. 3.10(b), 3.10(c) shows that when the throughput is set higher than 180, the latency increases significantly. Moreover, FastVA and DeepDecision are not based on HQM, and they therefore do not have throughout constraint. We report the monitored throughput by using two algorithms.

Additionally, it is a challenge to set an optimal throughput constraint that is affected by many factors such as input stream rate, network conditions and video resolution etc. Throughout, in this paper, for each set of the experiment, we manually set throughput constraint below the maximum system throughput to ensure smooth running of the optimization algorithms. More advance method is desired to maximize the throughput in future work.



(a) System Latency with Different Offloading Rate

(b) Algorithm Performance in Different Environments

Figure 3.12: Testbed vs Simulation

3.6.5 The Complexity Analysis of the Algorithms

In section 3.5.6, we analyzed the algorithm complexity and we validate the analysis in this subsection. Fig. 3.11 shows the execution time of the HQM-based algorithms to compute an offloading solution. PGD-VAO outperforms other two algorithms in terms of efficiency; its computation time increase *linearly* as the edge nodes increase. The computation time of PGS-VAO has a *superlinear growth* when the number of edge node increase. The results experimentally confirm our time complexity analysis for both PGD-VAO and PGS-VAO.

3.6.6 Real-world Test-bed

We evaluate the performance of the OsmoticGate using a lab test-bed in order to compare the real-system performance against simulation.

Lab Test-bed Configuration. The real-world test-bed is configured with Jetson Nanos and servers. We connect 4 Jetson-Nanos to the GPU server and set the connection as 5G with TC. To emulate the workload of the transmission queues, we maintain all transmission queues with 30 video chunks to saturate the cloud server. Time interval Δt is set as 30 seconds.

Fig. 3.12(a) reports the average queuing latency for both the simulation and the test-bed environment given different offloading rates. Generally, the computed simulation latency is quite close to the real test-bed. The difference between the two is less than 0.01 seconds for most cases and the two curves nearly overlap at several points. Small oscillation has been noted for the real test-bed as well. The complexity of the data

transmission and processing pipelines, as well as the varying working conditions of the HQM, are all factors that introduce oscillation to the test-bed performance.

Fig. 3.12(b) compares our proposed algorithm and other baselines in both test-bed and simulation environments. Overall, the HQM based algorithms (i.e., PGS-VAO, PGD-VAO, HillClimb) can achieve better performance as compared to non-HQM based algorithms (i.e., DeepDecision, FastVA). HillClimb, PGS-VAO and PGD-VAO all reach near-optimal point around 0.5 and 0.4, as seen in Fig. 3.12(a), while DeepDecision and FastVA consider transmitting all processing tasks to the cloud. PGS-VAO and PGD-VAO outperform the other baselines by 2x. This huge difference has proved the necessity to consider system workload dynamics when making offloading decisions.

3.7 Related Work

Content Delivery Network. Content Delivery Network (CDN) has been well studied in many areas including vehicle monitoring [305], Unmanned Aerial Vehicle (UAV) monitoring [17] and smart city [58]. The main goal of CDN is to use various technologies such as caching [159, 215] or machine learning (ML) [51, 246] to optimizing streaming data delivery in network level. Our OsmoticGate can be built up on these underlying systems to have better performance of video analytic task offloading.

Video Streaming System. Video streaming systems aim to deliver video data under different network conditions, while meeting various QoS requirements, including latency, throughput and system re-buffering level. To this end, the adaptive video streaming algorithms aim to configure the video stream to achieve efficiency video delivery. The existing work usually consider two factors: i) Rate-based algorithm that decides the bitrate based on network bandwidth assumption [143, 175, 270]. ii) Buffer-based algorithms that consider the client's playback buffer [129, 266]. This method keeps the system buffer at a stable level without sacrificing the video quality at large. Our work is built upon the modern video streaming technologies. Unlike the traditional video streaming systems that focus the QoS of video delivery. Instead, we target to offload the video processing tasks over edge+cloud environment

Video Analytic Task Offloading. [132, 142, 247, 325] have been conducted on schedul-

ing and configuring the video analytics jobs (queries). DeepDecision [236] considers both edge and cloud for conducting video analytics. However, it chooses CNNs only on one node (edge/cloud) for processing the videos. FastVA [277] considers offloading from local NPU to edge server, CrowdVision [186] considers client-server task offloading. However, They all fail to consider the system workload thus can not deliver optimal decisions under heavy-loaded systems. Also, these works treated video as sequence of images, whereas we consider bitrate-based video analytics that benefit from modern video streaming protocols.

3.8 Conclusions

In OsmoticGate, we investigated video streaming processing task offloading in cloud-edge computing paradigm. Based on bitrate-based video streaming protocols, we propose a HQM that is capable of capturing system workload dynamics. We model the system latency and throughput and then formulate a non-smooth, non-convex, constrained min-latency optimization problem. A two-stage gradient-based has been proposed which features switching between PGS-VAO and PGD-VAO algorithms. We have analyzed the convergence bound of PGA-VAO. Using this bound, we give practical implementation criteria for switching between the two algorithms. Extensive benchmarking has been conducted that serve as the foundations of our experiments. Simulation results showed that the our algorithm outperforms baseline works. Also, the two-stage algorithm is stable given different throughput constraints and various system conditions, which confirmed its effectiveness.

3.9 Proof of Lemma 5.1

By the definition of z we have:

$$z = \arg\min_{v \in \mathcal{Q}} \{\langle \boldsymbol{\gamma}, v - x \rangle + \frac{1}{2\eta} \|v - x\|_2^2\}. \tag{3.39}$$

Then for any point $y \in \mathcal{Q}$ statisfies $\|x - y\|_2 \leq \varepsilon$:

$$\langle \boldsymbol{\gamma}, z - x \rangle + \frac{1}{2\eta} \|z - x\|_2^2 \leq \langle \boldsymbol{\gamma}, y - x \rangle + \frac{1}{2\eta} \|y - x\|_2^2, \tag{3.40}$$

and hence:

$$0 \leq \langle \boldsymbol{\gamma}, y - z \rangle + \frac{1}{2\eta}(\|x - y\|_2^2 + \|x - z\|_2^2) \tag{3.41}$$

Now due to the fact that for $t \in [0, 1]$ we have $\frac{\partial T}{\partial t}(x+t(x-z)) = \langle \nabla T(x+t(x-z)), x-z \rangle$, and also consider the local-smooth condition, we have:

$$\begin{aligned} & |T(z) - T(x) - \langle \nabla T(x), z - x \rangle| && (3.42)\\ = & \int_0^1 \langle \nabla T(x + t(z - x)) - \nabla T(x), z - x \rangle dt && (3.43)\\ \leq & \int_0^1 \|\nabla T(x + t(z - x)) - \nabla T(x)\|_2 \|x - z\|_2 dt && (3.44)\\ \leq & \frac{L(x)}{2}\|x - z\|_2^2. && (3.45) \end{aligned}$$

Hence we have:

$$T(z) - T(x) \leq \langle \nabla T(x), z - x \rangle + \frac{L(x)}{2}\|x - z\|_2^2. \tag{3.46}$$

Now set $y = x$ in (3.41) and take the sum, we have:

$$0 \leq T(x) - T(z) + \langle \nabla T(x) - \boldsymbol{\gamma}, z - x \rangle + [\frac{L(x)}{2} + \frac{1}{2\eta}]\|x - z\|_2^2, \tag{3.47}$$

Next we use Young's inequality to have the following decomposed upper bound for $\langle \nabla T(x) - \boldsymbol{\gamma}, z - x \rangle$, that $\forall a > 0$:

$$\langle \nabla T(x) - \boldsymbol{\gamma}, z - x \rangle \leq \frac{1}{2aL(x)}\|\nabla T(x) - \boldsymbol{\gamma}\|_2^2 + \frac{aL(x)}{2}\|x - z\|_2^2, \tag{3.48}$$

and hence:

$$\begin{aligned} 0 \quad \leq \quad & T(x) - T(z) + \frac{1}{2aL(x)}\|\nabla T(x) - \boldsymbol{\gamma}\|_2^2 \\ & + [\frac{L(x)}{2} + \frac{aL(x)}{2} + \frac{1}{2\eta}]\|x - z\|_2^2, \end{aligned}$$

3.10 Proof of Theorem 5.2

We first apply the local sufficient decrease property by Lemma 3.5.1, setting $\boldsymbol{\gamma}_j = \nabla T(\boldsymbol{\beta}_j)$ and $a > 1$, then for all iteration $j = 1, 2, ..., J$ we can write:

$$T(\boldsymbol{\beta}_{j+1}) \leq T(\boldsymbol{\beta}_j) + [\frac{a}{2}L(\boldsymbol{\beta}_j) - \frac{1}{2\eta_j}]\|\boldsymbol{\beta}_{j+1} - \boldsymbol{\beta}_j\|_2^2, \tag{3.49}$$

til the first iteration:

$$T(\boldsymbol{\beta}_1) \leq T(\boldsymbol{\beta}_0) + [\frac{a}{2}L(\boldsymbol{\beta}_0) - \frac{1}{2\eta_1}]\|\boldsymbol{\beta}_1 - \boldsymbol{\beta}_0\|_2^2. \tag{3.50}$$

Then by summing up all these inequalities from 1 to J, and noting that $\|\boldsymbol{\beta}_{j+1} - \boldsymbol{\beta}_j\| = \eta_j^2\|G(\boldsymbol{\beta}_j)\|_2^2$, we have:

$$\sum_{j=1}^{J} \eta_j^2[\frac{1}{2\eta_j} - \frac{a}{2}L(\boldsymbol{\beta}_j)]\|G(\boldsymbol{\beta}_j)\|_2^2 \leq T(\boldsymbol{\beta}_0) - T(\boldsymbol{\beta}_J) \leq T(\boldsymbol{\beta}_0). \tag{3.51}$$

To provide the sharpest bound we set $\eta_j = \frac{1}{2aL(\boldsymbol{\beta}_j)}$ which would maximize the term $\eta_j^2[\frac{1}{2\eta_j} - \frac{a}{2}L(\boldsymbol{\beta}_j)]$. Then we have:

$$\frac{1}{8aL_{\max}}\sum_{j=1}^{J}\|G(\boldsymbol{\beta}_j)\|_2^2 \leq \sum_{j=1}^{J}\frac{1}{8aL(\boldsymbol{\beta}_j)}\|G(\boldsymbol{\beta}_j)\|_2^2 \leq T(\boldsymbol{\beta}_0), \tag{3.52}$$

where we denote $L_{\max} = \max_j L(\boldsymbol{\beta}_j)$. Then immediately we have the accumulative average gradient norm convergence rate as $O(1/J)$:

$$\boldsymbol{G}(J) \leq \frac{8aL_{\max}T(\boldsymbol{\beta}_0)}{J}, \tag{3.53}$$

and since as $J \to +\infty$, $\boldsymbol{G}(J) \to 0$ so $\|G(\boldsymbol{\beta}_J)\|_2^2 \to 0$.

4

OsmoticGate2: Edge-Cloud Collaborative Real-time Video Analytics with Multiagent Deep Reinforcement Learning

Contents

Summary

The content presented in this chapter is based on and expands upon the previously introduced **Osmoticagate** framework from chapter 3. This chapter introduces **Osmoticagate2**, an online multi-agent reinforcement learning system designed for edge-cloud collaborative real-time video analytics. The experiments conducted in real-world environments confirm that **Osmoticagate2** can effectively adapt to dynamic settings and achieve high prediction accuracy in real-time scenarios.

4.1 Introduction

Video analytic is of utmost importance in a range of computer vision applications, including video surveillance [328], augmented reality, and autonomous driving. Presently, Deep Neural Networks (DNNs) serve as the foundational technology for cutting-edge video analytic algorithms, ensuring exceptional precision for the end users. Nevertheless, the deployment of DNN models for video analytics in real-world settings presents numerous obstacles. These models are highly computationally demanding, featuring hundreds of layers, which leads to significant inference latency. Moreover, the sheer magnitude of streaming video data gives rise to apprehensions regarding the transmission of raw data to the cloud for inference due to the exorbitant costs associated with bandwidth and the ensuing insufferable delays in transmission.

One promising means to tackle bandwidth expenses and data transmission delays in video analytic applications is deploying DNN models on edge nodes situated in close proximity to users. These edge nodes can swiftly receive streaming videos from such end devices as cameras or mobile phones with minimized delays. However, they inevitably fall short in processing capabilities and become overwhelmed especially during peak times with non-negligible delays. Hence, video analytics systems imperatively require elaborate considerations of both edge and cloud resources for optimal performance guarantee in overloaded situations.

There are many existing studies on video analytics pipelines in the continuum of edge and cloud computing. For instance, A^2 [141] and EdgeAdaptor [338] delved into the

choice of distinct DNN models to optimize delay and accuracy. DeepDecision [236], FastVA [277], and Osmoticgate [230] investigated new offloading mechanisms between the edge and the cloud to fine-tune video analytics setups. They primarily take into account video preprocessing and model selection on the edge side to strike a balance between inference accuracy and delay. Reducto [173], and ClodSeg [294] employed frame filtering and resolution downsizing techniques to minimize communication costs during video transmission while preserving accuracy. However, these approaches can hardly adapt to dynamic environments where the fluctuation of available computing and communication resources is the norm rather than the exception across distributed edge nodes.

When it comes to optimizing video analytics pipelines in a highly distributed and dynamic environment, two main challenges remain unsettled: i) It is impractical to scale up the system due to the large configuration space in distributed systems. System states and configurations will drastically increase with the increment of the device number, and it is time-consuming for a centralized controller to search for an optimal configuration that maximizes the system resource usage across the edge and the cloud. For example, Chicago police analyze 30,000 camera streams in real time [1], and making decisions in a centralized manner is impossible. Thus the decentralized controlling system is imperative in such a distributed system. ii) Achieving a self-configuring plan for both cloud and edge nodes is challenging. In an edge-cloud setting, every agent, whether cloud or edge, strives to optimize their resource usage, which can impact the entire system's performance. For instance, if the network bandwidth is sufficient, each edge node may be more inclined to offload more data to the cloud, which may saturate the cloud and reduce the performance of the entire system.

To address the aforementioned challenges, we present OsmoticGate2, a distributed orchestration system for optimizing video analytics across the edge and the cloud. In our system, multiple edge devices work collaboratively and choose the appropriate configurations to maximize system Quality of Service (QoS) performance. To improve the system scalability, we deploy on each edge node a controller for generating video analytics configurations, based on local status only. Additionally, we have developed a mechanism grounded in multi-agent reinforcement learning, which uti-

lizes the Centralized Training and Decentralized Execution (CTDE) approach. This strategy allows all agents to engage in collaborative learning within the OsmoticGate2 server. The resulting OsmoticGate2 offers notable advantages, including exceptional scalability, adaptability, and efficiency, all achieved through a carefully optimized system-algorithm co-design.

Experiments on a real testbed with edge-cloud collaboration validate the system's effectiveness in enabling real-time and accurate video analytics. The paper's main contributions are summarized as follows:

- Design OsmoticGate2, an edge-cloud collaborative video analytics system in which the edge nodes and the cloud server can collaborate for video encoding and video analytics.
- Design an online multi-agent reinforcement learning algorithm (MAPPO) for orchestrating the system configurations within the systems. The agents of multiple edge nodes collaboratively learn the optimal policy by sharing their information to maximize their respective rewards.
- We evaluate the performance the proposed algorithm with real-world datasets and testbeds. Our method can achieve the best performance in terms of rewards, inference accuracy, as well as the target system latency.

4.2 System Overview

An overview of the proposed streaming video analytics architecture OsmoticGate2 is depicted in Fig. 4.1, where a large number of edge devices work collaboratively with the cloud server to perform video analytics jobs on streaming video generated from end devices. The streaming video is first encoded into small chunks and then processed locally on edge devices, with all results aggregated on the cloud server in the end. When the edge device is unable to process all the video streams in real time, a proportion of the video chunks are transmitted to the cloud for processing as well.

The extra video transmission overhead for cloud computing raises challenges on balancing the communication and computation resources in edge-cloud collaborative ana-

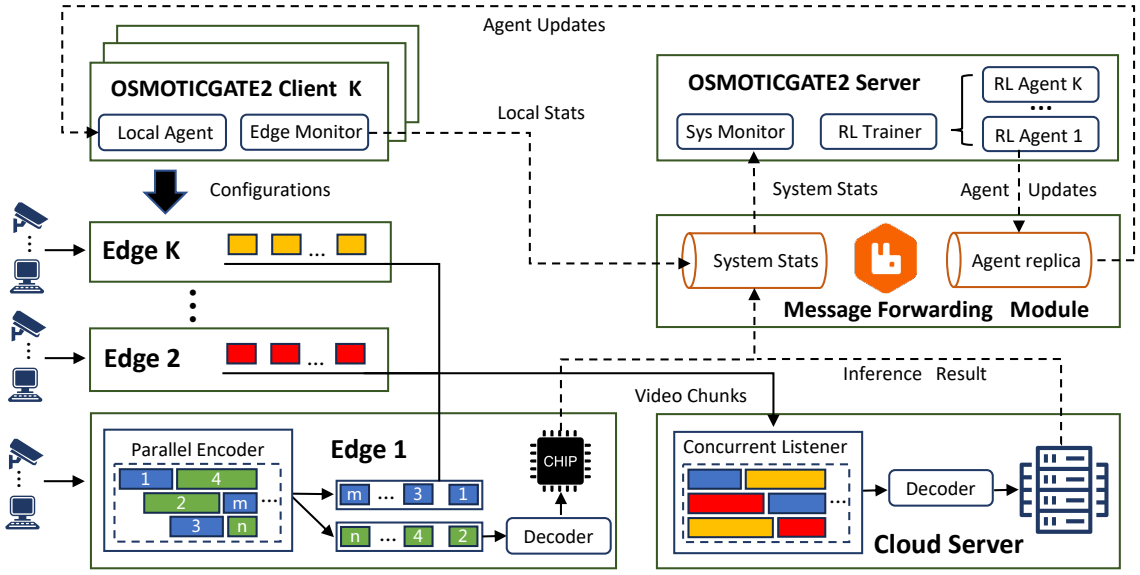


Figure 4.1: RL-based Edge-Cloud Collaborative Video Analytics in OsmoticGate2. The streaming videos are encoded in the edge nodes and then processed on both the edge and the cloud. Our OsmoticGate2 agents control the edge behaviors with various configurations. The two modules are communicated via a message forwarding module across the edge and the cloud.

lytics. As compared to edge computing, cloud computing incurs extra communication overhead but benefits lower processing latency. The balance between the communication and computation is further complicated when the system scales with more edge devices in the system. Thus, an adaptive mechanism that learns the system dynamics is necessary in consideration of the system scalability.

OsmoticGate2 develops a novel mechanism capable of adaptively generating system configurations to improve the system QoS metrics, i.e., accuracy and latency. It is composed of three major components: the *video analytics modules*, the *multi-agent controllers* as well as the *message-forwarding modules*. The three components are modularized and communicated via the *message-forwarding modules*, featuring seamless "monitoring - updating" across the edge-cloud computing paradigm, enabling the smooth running of the OsmoticGate2. In general, OsmoticGate2 has three main design considerations:

Adaptability: We design multi-agent RL-based controllers for generating configurations and updating controllers based on local and global system status.

Scalability: We implement centralized training and decentralized execution (CTDE)

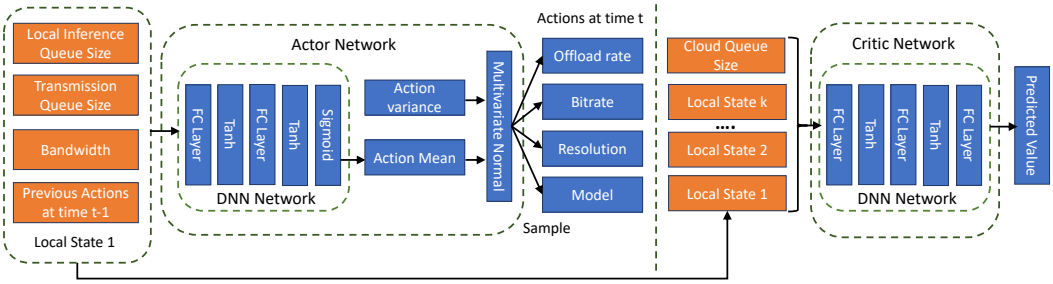


Figure 4.2: RL Agent architecture

strategies for the controllers, alleviating the problems of generating configurations from high-dimension search spaces, especially when there are many edge devices in the system.

Efficiency: We implement on-device acceleration techniques on both computation and communication for video processing across the edge and the cloud server.

4.3 Multi-agent RL-based Controllers

We consider a classic real-time video analytics application, as depicted in Fig. 5.2. This application comprises multiple edge devices denoted as E_k, for $k \in \{1, 2, ..., K\}$, and a cloud server referred to as S. Each E_k is responsible for receiving video streams and performing video analytics jobs in real time. Before the encoded video streams are input into the deep learning models for video analytics, they are placed in task queues, where they await decoding into image frames at both endpoints. Then an appropriate model is selected to meet the quality of service (QoS) requirements specific to the application. All the outcomes generated are then consolidated and sent to the cloud monitor, alongside other system status information.

For N chunks of video streams generated from E_k, the overall average processing latency l_k^t averages over two components: 1). *Edge processing latency* edge processing pipeline includes four components: the video encoding latency, the waiting time in the queue, the video decoding latency, and the inference latency. 2). *Cloud processing latency* cloud processing places extra system overhead. During cloud inference, except aforementioned latency, one extra latency comes from the transmission latency between the edge and the cloud. The notation table is shown in Table 4.1.

Table 4.1: Notation

Notation	Description
t	Current time step t.
K	The total number of edge devices.
E_k	Edge device k, $k \in$K.
S	The cloud server.
r_k^t	Video resolution of edge device k at time step t.
b_k^t	Video bitrate of edge device k at time step t.
m_k^t	Inference model candidate of edge device k at time step t.
μ_k^t	The offloading ratio for edge device k at time step t.
QI_k^t	The size of local inference queue for edge device k at time step t.
QT_k^t	The size of transmission queue for offloading at time step t.
Q_S^t	The size of inference queue for server s at time step t.
B_k^t	The network bandwidth of edge device k at time step t.
A_k^t	The action for edge device k at time step t.
acc_k^t	The average accuracy from edge device k at time steps t.
l_k^t	The inference latency of edge device k at time steps t.
l_k^{target}	The latency target of edge device k.
O_k^t	The local state of edge device k at time step t.
G^t	The global state of the server S at time step t.
R_k^t	The reward of edge device k at time step t.
l^{thres}	The latency threshold in reward function
F	The reward penalty when latency exceeds threshold l^{thres}
M	Maximum training episode
T	Maximum steps in one episode

4.3.1 Optimization Objective

Throughout the whole T running time within the system, we consider two optimization objectives: the average system processing latency and the average inference accuracy. Assume the average inference accuracy during one time interval Δt is acc_k^t, our optimization objective is formulated as follows:

$$\begin{aligned} \underset{A_k^t}{\text{maximize}} \quad & \frac{1}{T}\sum_{t=1}^{T} acc_k^t \\ \text{subject to} \quad & A_k^t = \{r_k^t, b_k^t, m_k^t, \mu_k^t\}; \\ & |l_k^t - l_k^{target}| \leq \epsilon, \forall t \in T \end{aligned} \tag{4.1}$$

For each E_k, we aim to maximize the average inference accuracy during the system running time T, while ensuring the average processing latency is close to a preset latency threshold l_k^{target}. We set a slack variable ϵ such that the small latency variance

$|l_k^t - l_k^{target}|$ is acceptable during the application production. Subject to different QoS metrics of the real-time stream processing systems, l_k^{target} can be variably adjusted as well for each edge device. The generated configurations for video processing include the video bitrate b_k^t, frame resolution r_k^t, and the choice of model m_k^t used for video analytics. Additionally, the smart agent must make decisions about the offloading rate μ_k^t, representing the proportion of stream data to be transferred from device k to the server S.

The key problem for the RL agent lies in figuring out the optimal value, denoted as A_k^t, for E_k at every time step, with the goal of enhancing the precision of system predictions. The difficulty lies in factoring in dynamic system limitations, like computational resources and network capacity, while being contextually aware. The objective is to strike the right balance between ensuring stable system response times and achieving the highest level of accuracy by making informed choices regarding video analytics configurations.

Based on the system model and the optimization objective as represented in Equation 4.1, we elaborate on the intricate blueprints of the fundamental elements in the reinforcement learning (RL) agent, i.e., the architecture of the RL agent, the RL state, the RL action, the reward function. In addition, we also demonstrate how to train the RL agent in the system.

4.3.2 Architecture of RL agents

Actor-critic framework: Figure. 4.2 illustrates the architecture of RL agents in our system. We adopt an architecture of actor-critic paradigm where multiple actor networks are deployed on each edge device E_k for outputting the optimal policy. Following the design in [313], each edge device k has its own pair of actor and critic network on the central server S. The advantage of this design is that each edge device E_k can focus on optimizing its own policy while sharing the learning from local trails through a critic network on the cloud. We use the Proximal Policy Optimization (PPO) [249] as the actor-critic RL algorithm in this paper and denote the architecture of PPO used as the multiple agents PPO (MAPPO).

The actor network π is responsible for learning the agent's policy. It maps agent observations O to the mean and standard deviation of a Multivariate Gaussian Distribution, from which an action is sampled, in continuous action spaces. The actor network maximizes the expected cumulative reward over time by adjusting its network parameters in a way that leads to better actions in various states, ultimately improving the agent's performance. In this work, the input of the actor network is the local state of each E_k, and the output is probability distributions of 4 actions. They are later cast to the system configurations in the environment. The architecture of the actor network includes a 2-layer DNN network with a sigmoid activation function and a hyper-parameter action variance.

We sample the actions via a multivariate normal distribution with the action mean generated from the DNN network and the action variance. There have been many techniques for enabling efficient model exploration during training, as reflected in [160]. In this paper, we employ a parameterized exploration technique for action sampling modules in agent actors. Specifically, the distribution variance exponentially decreases along with different training episodes, i.e., standard deviation $std = 0.5 * 0.96^{episode}$. As compared to exploration techniques such as extra entropy loss in optimization objective, the benefit of such mechanism is enabling fast training of the algorithm, which is especially important in real-world systems.

The critic network, on the other hand, estimates the value of being in a particular state or taking a particular action in a given state. It takes states and actions as input and outputs a value, which represents the expected cumulative reward that can be obtained from that state-action pair. By comparing the estimated values with the actual rewards, the agent can update its policy network to maximize the expected rewards. In this work, the input of the critic network is an embedding layer that concatenates all local states and the cloud queue size. After performing feedforwarding with a two-layer DNN network, the output of the critic network is the predicted value given the global state.

4.3.3 RL States and Actions

The system state reflects the working status within the system at every time step t. Concretely, we distinguish two states, i.e., the local state (O_k^t) on each edge device E_k and the global state (G^t) on the server S. On the edge side, each device E_k takes the local state O_k^t as the input at time step t to output the optimal action A_k^t. The global state G^t is used as the input of the global critic network at time step t.

The local state O_k^t for each E_k includes the size of the transmission queue QT_k^t, the size of the local inference queue QI_k^t, the average network bandwidth B_k^t between E_k and the cloud S, and the agent's action at the previous time step $t-1$. At time slot t, we denote the local state of each agent k as $O_k^t = \{QI_k^t, QT_k^t, B_k^t, A_k^t\}$

On the cloud server S, the global state G^t includes all local states and the size of cloud queue Q_S^t from all edge devices. The global critic network takes the global state and estimates the value of state-action value function, i.e., the expected cumulative reward an RL agent can achieve following its policy. The global state G^t at time step t is defined as $G^t = \{Q_S^t, O_1^t, O_2^t, ..., O_k^t\}$. where k is the number of all edge devices.

The RL action generated by each edge device k should consider the critical decisions with the system model, i.e., the resolution of generated images r_k^t, the bitrate for encoding and decoding the video stream b_k^t, the candidates of different models m_k^t, and the offloading ratio of the generated images μ_k^t. The RL agent action for each edge device k at time step t can be expressed as $A_k^t = \{r_k^t, b_k^t, m_k^t, \mu_k^t\}$. Specifically, we design three decision variables r_k^t, b_k^t, and m_k^t as discrete variables. The resolution choices r_k^t are three-folds: 240P, 360P, and 540P. The video bitrate b_k^t has three choices as well: 500kbps, 1000kbps, and 3000kbps. The model choice on the edge side m_k^t includes three model variants of YOLOv8, named YOLOv8-m, YOLOv8-s, YOLOv8-n. While on the cloud side, we use YOLOv8-m only, with the best accuracy and largeset inference latency as well. The data offloading ratio μ_k^t is designed as a continuous variable from [0,1] to concisely control the proportion of data transferred to the cloud. During system implementation, μ_k^t portions of the video chunks are offloaded to the cloud while the rest are processed locally. The combination of the chosen actions can cover a wide range of accuracy and latency requirements that are suited for different types

of QoS requirements, i.e., highest accuracy or moderated accuracy with acceptable system latency.

4.3.4 Reward Function

The reward function guides the optimization direction and is the core of the RL agent. Based on our system optimization goal in Equation 4.1, for each time step t, we formulate the reward R_k^t as follows:

$$R_k^t = \begin{cases} acc_k^t - \omega * |l_k^t - l_k^{target}| & l_k^t \leq l^{thres} \\ -\omega * F - (l_k^t - l^{thres}) & \text{otherwise} \end{cases} \tag{4.2}$$

where acc_k^t is the average accuracy at time step t, and $\omega * |l_k^t - l_k^{target}|$ is the latency constraint with a positive ω balancing the weights between the accuracy and latency. When the latency l_k^t exceeds the pre-defined threshold l^{thres}, we place another linear penalty term $-\omega * F - (l_k^t - l^{thres})$, where with F a positive number to penalize the action that leads to high latency. ω weighting term is included to ensure that the rewards under normal conditions are not overshadowed. By doing so, the agent is motivated to complete the task as close to the desired time as possible while maximizing accuracy.

4.3.5 Centralized Training and Decentralized Execution (CTDE) in OsmoticGate2

In our paper, the number of edge devices, the number of bitrates, resolutions, model choices and the offloading rates all could increase the complexity of the decision-making. In MAPPO, we have designed centralized training and decentralized deployment strategy for alleviating the scalability problems. Specifically, by using centralized training, the agents can learn and coordinate their actions based on a global view of the environment. This allows them to capture complex interactions and dependencies more effectively during the learning process. In our paper, for each configuration we select three values that could reveal performance variations across them. Increasing the number of configurations will not raise scalability issues. Since the agents operate in a decentralized manner, making decisions based on their local observations and learned

Algorithm 3: RL agent training in OsmoticGate2

Input: the number of the edge device K, maximum episode M, maximum time step T
Output: the actor of each device k, $[\pi_k, \forall k \in K]$
for *each episode* $m \in M$ **do**
  $\tau_1, ..., \tau_K = []$ empty list
  for *each device* $E_k, k \in K$ *in parallel* **do**
    Initialize parameters θ_k for the actor π_k, and ϕ_k for the critic V_k
    for *each time step* $t \in T$ **do**
      1. observe the agent's local state O_k^t and global state G_k^t
      2. Get actions $A_k^t = \pi_k(O_k^t; \theta_k^t)$
      3. execute the action $A_k(t)$ in environment
      4. observe the agent's local state O_k^{t+1} and global state G_k^{t+1}
      5. get the reward with R_k^t with Eq. 4.2
      6. $\tau_k += [A_k^t, O_k^t, G_k^t, R_k^t, O_k^{t+1}, G_k^{t+1}]$
    end
    Update θ_k via Adam
    Update ϕ_k via Adam
  end
end
return *RL agents*

policies. Each agent acts independently and does not require access to the full state or information of other agents. In our system, we implement in OsmoticGate2 server the RL trainer and deploy in OsmoticGate2 client the local agents. Auto-scaling can be implemented for the cloud server as well to support inference tasks for more devices in the future.

Algorithm 3 illustrates how we train the actor and critic network of each RL agent. First of all, we initialize the environment for each device, and initialize the parameters θ_k of the actor π_k, and the parameters ϕ_k of the critic V_k. When the RL agents are ready, both the server and the edge nodes start receiving the video streams. During the training process, each edge node is allocated a buffer list τ_k. At time step t, the agent observes the local state O_k^t and the global state G_k^t, then the action is generated $A_k^t = \pi_k(O_k^t; \theta_k^t)$. The A_k^t is executed in the system until time step $t+1$. At this time step, the new local state O_k^{t+1}, global state G_k^{t+1}, as well as the reward R_k^t are observed in the cloud server. All these stats are collected in the system as one buffer for updating the model parameters. The agents continue to interact with the system until the end of the episode where $t == T$. After one episode, for each pair of actor θ_k

and ϕ_k, all collected samples in the trajectory are fed to the objective function of the respective actor and critic networks, and the network parameters are updated with ADAM optimizers.

4.4 Implementation Details

4.4.1 Video Analytics Module

The video analytics module is built upon the work in *Osmoticgate* [230] while we extend the framework with several parallel techniques to speed up the video processing operations. The processing process in OsmoticGate2 includes video encoding, inference, and video transmission across the edge and the cloud server. We illustrate in the following subsections the techniques employed to speed up the processing within the system.

4.4.1.1 Parallel Video Encoder

When the video streams are fed into the edge devices, they are encoded within the system with the configuration command from the OsmoticGate2 clients. The encoding process entails compressing the video streams to target bitrate and resolutions, in order to accelerate the inference speed. As encoding computation is highly CPU-intensive, we utilize the multi-core architecture and implement multi-process encoding for accelerating the encoding process in the first place. Specifically, the video streams are split into small chunks and encoded concurrently in different processes. The results are then pushed to the local processing queue for local inference or transmission queue to be processed by the cloud server.

4.4.1.2 Inference Engine

We implement inference engines for both the edge and cloud devices for making predictions on the encoded video chunks. On the edge devices, models of various configurations are loaded into the memory. OsmoticGate2 clients decide the appropriate model to be used for processing the current chunks. On the cloud server, only a large

model with the best performance is deployed for making predictions. All received video chunks are decoded into batches of frames before being loaded into the models for inference.

4.4.1.3 Concurrent Listener

When transmitting the raw video contents from the edge to the cloud, the packet re-assembly may take much time and it is easy for the socket to get stuck at this time. This is especially true when multiple clients are transmitting to the cloud simultaneously, where the clients have to wait for the other clients to send all the content.

Thus, we design a socket pool with each containing a worker process listening to connections and receiving the packets. The concurrent processes are continuously monitoring a whole socket pool, re-assembling the received packets. All received packets are sent to the queue, then the cloud cluster for further processing.

4.4.2 Multi-agent Controllers

The core of the OsmoticGate2 is the coordination of components for alleviating the computation and communication bottlenecks across the edge and cloud servers. In order to do so, we deploy on each edge device an OsmoticGate2 *client* and a global OsmoticGate2 *server* on the cloud. Each OsmoticGate2 *client* contains an agent and generates system configurations for controlling the local processing operation in real-time. The configurations are generated based on local system stats collected via the *Edge Monitor* and the previously generated configurations.

Specifically, the *System Monitor* is deployed on the OsmoticGate2 *server* collecting all local stats. A *RL Trainer* is deployed on the cloud server along with all local agents. Via collected information from the *System Monitor*, the *RL Trainer* is able to jointly update all agents, such to achieve the goal of optimal system performance.

4.4.3 Message-forwarding Module

In order to realize the seamless control between the *video analytics module* and the *multi-agent controllers* mentioned above, we implement the *Message-forwarding mod-*

ule as a middleware to exchange information between these two modules across the edge and the cloud.

The module is based on RabbitMQ where we initiate two queues: 1) *System Stats* queue for forwarding the inference results from all devices as well as the local/global stats. All stats from both the edge devices and cloud server are then aggregated in the system monitor to assist the agent training process 2) *Agent Replica* queue for forwarding and deploying the updated agents to the edge side. OsmoticGate2 *server* contains all agent replicas within the system. Once the agents are updated via the RL trainer, the newly generated models are sent to the target device, ensuring all agents are up-to-date, conforming to the system states.

4.5 PERFORMANCE EVALUATION

4.5.1 Experimental Setting

Testbed: We use 4 NVIDIA Jetson Xavier NX (with ARMv8.2 CPU and 8GB RAM) as the edge nodes and the cloud server is a bare metal Ubuntu machine, with 8 cores (Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz), GeForce RTX3090 graphics card and 48 GB RAM. The cloud operating system is Ubuntu 20.04.

Dataset: We use road traffic video datasets from UA-DETRAC [296] to construct our video analytics system, which contains 10 hours of video captured at 24 different locations at Beijing and Tianjin in China [1]. The videos are recorded at 30 frames per second (fps), with a resolution of 960×540 pixels. We subtract from the simple and medium category a total of 120s of video data to construct our training dataset.

In order to simulate real-world bandwidth connection, we use public traces from oboe [10] with TC to control the bandwidth between the edge and the cloud server. We sample a sub-trace from all the traces and repeatedly emulate the bandwidth throughout the whole training process. In particular, during the training of each episode, the network condition changes every 20 seconds, and the duration of each time step t is set to 10 seconds.

[1] https://detrac-db.rit.albany.edu/

Table 4.2: The average inference accuracy and processing latency for each video chunk with different models, including the encoding, decoding latency. This does not include the queue waiting time, and the transmission latency.

Edge	Accuracy			Latency(s)		
Model(YOLOv8)	m	s	n	m	s	n
540P	0.844	0.794	0.746	2.117	1.646	1.025
360P	0.817	0.763	0.721	1.512	1.306	0.851
240P	0.762	0.731	0.653	1.289	1.174	0.754
Cloud	Accuracy			Latency(s)		
Bitrate(kbps)	3000	1000	500	3000	1000	500
540P	0.846	0.814	0.741	0.707	0.677	0.668
360P	0.842	0.818	0.775	0.569	0.548	0.542
240P	0.815	0.797	0.767	0.483	0.473	0.467

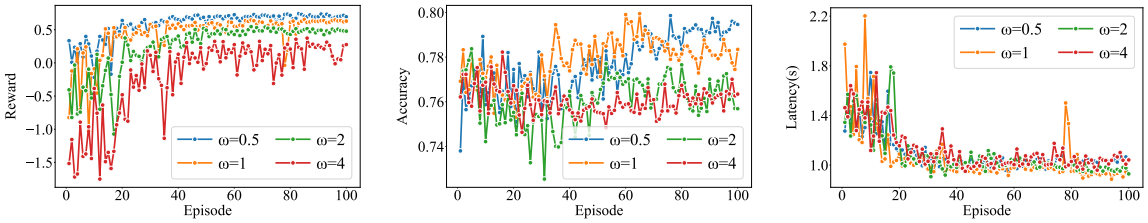
System Configuration: We consider object detection when analyzing the video data in UA-DETRAC and deploy yolo [6] models for detection. We use the same architectures as in [6] and deploy on the edge device YOLOv8-n, YOLOv8-s, YOLOv8-m models with 3.2M, 11.2M, 25.9M parameters respectively. As cloud has enough computation power, we only deploy the YOLOv8-m model with highest accuracy. The models are pre-trained on COCO dataset fine-tuned on UA-DETRAC subset later. During the fine-tuning process, we start with a learning rate of 0.01 and a batchsize of 64 and train for 300 epochs, and then adjust the learning rate to 0.001 and continue training 300 epochs to get the final model.

We show in Table 4.2 the running performance of different models in the system. Due to the different processing pipelines, the inference accuracy of different edge models are correlated with the video frame resolution as encoding to smaller bitrate is unnecessary. Whereas the cloud performance is influenced by both the resolution and video stream bitrate. The system configuration space is further complicated by the offloading rate as well. In the reward function, F is set to 1 and l^{thres} is set to 1.5 in all experiments.

Baseline methods: We assess the performance of our approach by contrasting it with the following baseline.

1) OSMOTICGATE [230]: the cloud server models the system and collects global information, using a two-stage gradient-based algorithm to provide offloading strategies for each edge node.

2) FastVA [277]: considers making the most use of network transmission to offload the



(a) Reward Curve (b) Accuracy Curve (c) Latency Curve

Figure 4.3: Convergence and Performance of OsmoticGate2 under Different Penalty Weights

video chunks to the cloud.

3) Edge-Median: edge nodes choose the smallest offload rate, the medium resolution, the medium bitrate, and the medium model.

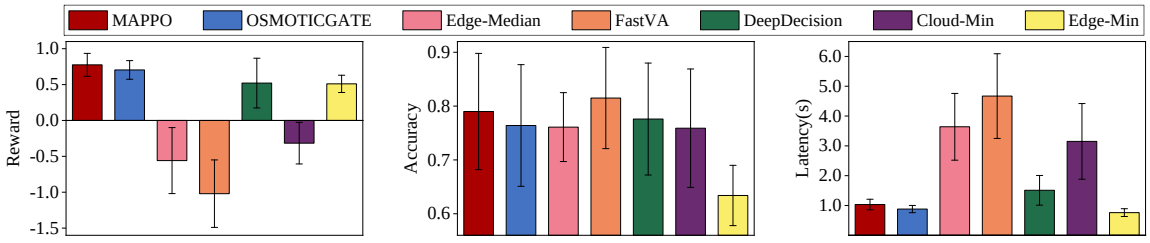
4) DeepDecision [236]: addresses an optimization challenge wherein video processing occurs selectively either at the edge or the cloud side, contingent upon the prevailing system throughput within each time interval.

5) Cloud-Min: edge nodes choose the largest offload rate, and choose the smallest resolution and the lowest bitrate.

6) Edge-Min: edge nodes choose the smallest offload rate, resolution, lowest bitrate, and the smallest model.

4.5.2 Convergence and Performance under Different Penalty Weights

We analyze the convergence of the proposed OsmoticGate2 under different penalty weights ω, i.e., 0.5, 1, 2, 4. We set l^{target} as 1 in this experiment. As seen in Fig. 4.3, all 4 sets of experiments can converge to a stable policy. The convergence can be fast at around 40 episodes, which validates the effectiveness of the proposed MAPPO algorithm. In Fig. 4.3(a), the overall reward decreases when ω gets larger, with around 0.77, 0.63, 0.48, and 0.23 reward for 4 penalty weights respectively. This is mainly due to the larger penalty incurred from the deviations of the latency target. However, the reward does not fully reflect the system performance as it combines both the latency and accuracy of the system.



(b) Average reward (c) Average inference accuracy (d) Average latency

Figure 4.4: Performance of OsmoticGate2 and baseline with Penalty Weights of 0.5

From the accuracy curve and latency curve in Fig. 4.3(b) and 4.3(c), when the value of ω is set to 0.5, our algorithm attains its highest accuracy of 0.79. This is primarily because a smaller value of ω places greater emphasis on achieving high accuracy. During the training process, the accuracy curve gradually converges to a point and the latency is stable at the end. As reflected in Fig. 4.3(c), the convergence latency is around 1s, which satisfies the pre-defined l^{target}. The training curve reveals that our algorithm learns configurations that prioritize meeting the latency target over maximizing accuracy.

4.5.3 Performance Comparison with Baselines

In this experiment, we compare the performance of the proposed algorithm with several baselines. We set ω to be 0.5, and lt^{target} to be 1. We report the average reward for different methods in Fig. 4.4(b) and the detailed system performance in Fig. 4.4(c) and 4.4(d).

From Fig. 4.4(b), we can see that MAPPO outperforms baselines in all experiments. OsmoticGate is the most competitive baseline in the literature. However, as its primary goal was to minimize system processing latency, OsmoticGate finally achieves 0.88 latency and 0.76 accuracy, an inferior performance as compared to us. FastVA achieves the highest accuracy 0.815 and latency 4.67, as it tends to transmit as many as possible video chunks to the cloud, but it leads to cloud overload. Other simple heuristics such as Edge-Median and Edge-Min all lead to worse performance in our experiments. In conclusion, our OsmoticGate2 with MAPPO method strictly conforms to the system performance target, i.e., maximize accuracy while ensuring around 1.0 system latency. OsmoticGate2 can ensure real-time streaming video processing while maximizing

the prediction accuracy. The experimental results validate the superiority of the multi-agent reinforcement learning in our OsmoticGate2.

4.6 Conclusion

In this paper, we study the problem of real-time video analytics across the edge and cloud environments. We propose OsmoticGate2 for orchestrating the configurations and performing task offloading across the edge and the cloud. In order to adapt to distributed and dynamic environments, we introduce a sophisticated online multi-agent reinforcement learning system crafted. Powered by the cutting-edge multi-agent reinforcement learning algorithm MAPPO, all agents are actively engaged in real-world environments, continually learning from these interactions. This dynamic learning process enables the system to optimize its performance effectively in a changing environment. Through rigorous experimentation, we show exceptional adaptability to varying system configurations while maintaining stable runtime performance.

5

DEEPCON: Improving Geo-distributed Deep Learning Model Consistency in Edge-Cloud Environment via Distillation

Contents

Summary

This chapter studies a model inconsistency problem that incurs when deploying many models on distributed edge-cloud computing paradigms. I design and implement DeepCon, an adaptive deployment system across the edge-cloud layer for over-the-air model updates. I also implement DMML-Par, an asynchronous parallel training algorithm for quickly updating the models and improving consistency. My experiment results on both vision and language tasks demonstrate that DMML could improve the model consistency up to 4%, 7%, and 13% at CIFAR10/100 and IMDB datasets without sacrificing the accuracy of individual models. I also show that the DMML-Par is up to 60% faster compared to simple synchronous parallel training.

Chapter 4 proposes a streaming video analytics system that features high accuracy and low processing latency. Thus, the evaluation was thus performed with videos on the proposed OsmoticGate2 system. One problem raised after system deployment is the inconsistent predictions of many deployed models in the application. Thus, Chapter 5 aims to address model consistence problems and proposes an algorithm named DMML-par for fast training of the deep learning models to increase model consistency. The evaluation set up and dataset choice is aligned with the research problems in Chapter 5. We use images as it's the basic processing unit of deep learning models and we use cloud servers with multiple GPUs for evaluating the training speed of the DMML-par.

5.1 Introduction

The deployment platforms of DL models are distributed across various hardware devices, including cloud or edge GPU, end-user mobile phones, and IoT devices, as highlighted in [229]. These platforms form the fundamental infrastructure of distributed Deep Learning applications. However, since these devices differ in their hardware capabilities, it is necessary to adaptively deploy multiple DL models that share similar functions but with different architectures and parameters, to efficiently utilize the available resources.

(a) Recyclable Waste Classification

Correct / Incorrect		Case A: Consistent					Case B: Inconsistent				
		Predictions				Picked	Predictions				Picked
Examples	Class	M_A	M_B	M_C	M_D	By	M_{A1}	M_{B1}	M_{C1}	M_{D1}	By
X_1 - X_{12}	...	...				...	...				...
X_{13}	A	A	A	A	A	M_A	B	B	A	A	M_B
X_{14}	A	A	A	A	A	M_A	B	B	A	A	M_B
X_{15}	B	B	B	B	B	M_B	D	D	B	B	/
X_{16}	B	B	B	B	B	M_B	D	D	B	B	/
X_{17}	C	D	D	D	D	M_D	C	C	B	B	/
X_{18}	C	D	D	D	D	M_D	C	C	B	B	/
X_{19}	D	C	C	C	C	M_C	D	D	A	A	/
X_{20}	D	C	C	C	C	M_C	D	D	A	A	/
Acc		16/20=80%				80%	16/20=80%				60%
Result		A	B	C	D		A	B	C	D	

(b) Example Classification Results for 2 Sets of Models

Figure 5.1: The Failures Caused by Model Inconsistency in Recyclable Waste Classification

Example. A distributed Recyclable Waste Classification application [194], for example, consists of multiple robot arms with each responsible for picking up a specific type of item and placing it into corresponding bins (see Figure 5.1(a)). However, different items require different robotic arms, such as rigid robotic arms for metals and flexible robotic arms for lamps. These robotic arms are often manufactured by different companies and use different hardware platforms, thus deploying different DL models (e.g., ResNet101 [115], MobileNet [124] and VGG13 [259]) to classify recyclable objects. The heterogeneity of the hardware and software may lead to potential issues with system compatibility as follows.

Systemic problem. Multiple models within the application may produce different outputs when given the same input because the models are usually trained individually or calibrated to different configurations after development. Table 5.1(b) shows two cases of 4 arms classifying and picking items from the conveyors. In case A, M_A, M_B, M_C, M_D are designed to pick item A, B, C and D respectively. The setting is the same for case B as well, with one difference that, in case B, M_{A1}, M_{B1}, M_{C1}, M_{D1} may generate different classification results for the same item: $X_{13} - X_{20}$. We assume that all models have achieved their best classification accuracy, i.e., 80% classification accuracy. We then compare the picking results for 2 sets of arms. For the first 12 examples, we assume that all models are making correct predictions (marked as green), such that these items are correctly picked by respective arms. However, for the rest

8 examples, the 2 sets of models behave differently. In case A $X_{17} - X_{20}$, items C and D are incorrectly classified (marked red) as D and C respectively. Thus, arm M_D picks X_{17} and X_{18}, arm M_C pick X_{19} and X_{20} to their respective bins, causing misclassification bad cases in the system. For example $X_{13} - X_{16}$, 4 models are making correct predictions as well. Finally, the system is able to correctly pick 16 items out of 20 examples.

The situation is much worse for case B with models producing different classification results for the same item. For item A in example X_{13} and X_{14}, and item B in example X_{15} and X_{16}, they are only possible to be correctly picked when M_{A1} or M_{B1} makes correct predictions respectively. However, both M_{A1} and M_{B1} mistakenly classify object A as B for X_{13} and X_{14}, thus M_{B1} picks them into bin B, causing a system failure. M_{A1} and M_{B1} also misclassify B as D for X_{15} and X_{16}, and omit them in the conveyor, waiting for other arms to pick them up. M_{C1} and M_{D1} make the correct classification for all these 4 items but they are just not designed to pick either A or B, thus X_{15} and X_{16} are finally left on the conveyor, with no arms capable of picking them. Similar situations are observed for example $X_{17} - X_{20}$, even though there are models making correct classifications, they are just not targeting the specific items. Thus the whole system omits these items and left them on the waiting conveyor, leading to a clog of the running system.

When comparing 2 cases, we observe that the unpredictable inconsistent results between different models may 1) fail on previously correct picks and 2) completely omit items on the conveyers. This inconsistency reduces the ability of the system to correctly sort objects (from 80% to 60%) and potentially collapses the whole system and causes much energy waste.

To overcome this problem, the models need to work collaboratively to identify the inconsistent cases, and interact with each other, learning from each to achieve a "consensus". Some previous works have attempted to address this problem as well. [145, 208, 252, 264] study model "irreproducibility/disagreement" problem during model training. These works report different factors that models may not make consistent predictions, including activation functions [252], model randomness [264], model architectures [208], optimizers [145]. They also design specific techniques to reduce such "irreproducibil-

ity/disagreement" during model training. Some preliminary work [20, 30, 140] are proposed to reduce the model inconsistency during model re-training as well. However, these methods only consider the inconsistency when the same model is continuously updated during the training process. They are not applicable to a number of models with different parameters and architectures, which is a common case in real-world IoT applications.

In conclusion, some preliminary research has looked at factors that cause and approaches to reduce model inconsistency. However, they have not considered how to *detect* and *reduce* such inconsistency when multiple models are collaborating in a real-world distributed application. An algorithm and system co-design solution is required to interact with the distributed models and improve the consistency of system outputs. To be precise, we need to tackle the following challenges while building such distributed DL applications. ***1) How to detect the inconsistency among the distributed models?*** An edge-based DL-applications, the models are distributed and adaptively configured. This brings the challenge of how to efficiently interact with the different outputs of the models to provide a unified consistency measurement. ***2) How to efficiently reduce the inconsistency among the heterogeneous models?*** In an edge-based DL application, the models deployed on the edge nodes are heterogeneous and distributed. Therefore, how to efficiently update (or fine-tune) these models becomes a challenge for both algorithm and system design. In particular, on the one hand, the proposed inconsistency reduction algorithm should have the flexibility and scalability to fine-tune multiple (greater than two) heterogeneous models simultaneously. On the other hand, the proposed system should have the ability to coordinate the models across the cloud and edge nodes in a distributed manner.

We design and implement DeepCon to realize our goal of quickly improving the model consistency of an edge-based application. We first develop the core technology of DeepCon which is a learning algorithm - Deep Mixup Mutual Learning based on knowledge distillation (KD) [119] that forces models to generate more similar outputs. Similar to KD, DMML enables a student model to mimic the behavior of the teacher model by learning from its output and ground truth labels. Furthermore, we extend DMML and implement asynchronous parallel DMML (DMML-Par) for accelerating

the model training in multiple GPU workers. Finally, we develop a set of APIs to support seamless communications between edge and cloud for data collecting (Edge), model consistency validating (Cloud), and model updating (GPU cluster).

Overall, this paper makes the following key contributions:

- **DMML for improving model consistency (§5.2 and §5.3).** We illustrate the importance of consistency in evaluating the performance of geo-distributed DL applications and define a new consistency metric (CC) for measurement. Then we propose DMML, a KD-based learning algorithm for cross-model learning, improving the consistency among the models. To improve the DMML's scalability, we develop the DMML-Par that can scale DMML to multiple GPU nodes.
- **Design and implementation of DeepCon (§5.4).** DeepCon provides non-stop updates to improve the model consistency of geo-distributed DL applications. Moreover, DeepCon offers an algorithm and system co-design solution to maintain a geo-distributed DL application deployment life-cycle.
- **Comprehensive evaluation of DeepCon (§5.5).** We evaluate DMML with both vision and language classifications: CIFAR-10, CIFAR-100, and IMDB. We also evaluate the training speed of DeepCon on the same dataset. Our results show that DMML can achieve 34.1% to 56.8% CC improvement on pre-trained models. Our DeepCon can also achieve up to 60% speed up compared to the simple model parallel algorithm.

5.2 Overview of DeepCon

DeepCon aims to develop an adaptive deep learning model deployment system that combines the computing resources from the cloud server, GPU cluster, and edge devices, which enables the models to be updated in OTA (over-the-air) manner. To this end, we build DeepCon that comprises two main components: *Deep Mixup Mutual Learning (DMML)* algorithm and *OTA controller*.

The DMML algorithm is running on a cloud, which is developed to optimize the parameters of the models to mitigate the inconsistency when the inconsistency is detected or

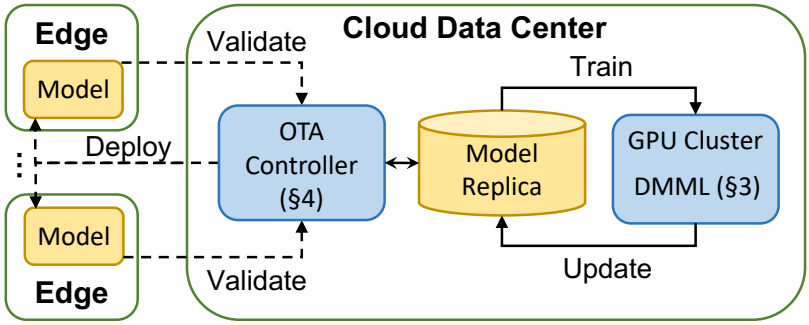


Figure 5.2: DeepCon Overview

exceeded a threshold (see §5.3.3). The OTA controller orchestrates the model update and re-deployment across cloud and edge devices (see §5.4). It maintains a deep learning application deployment life-cycle through the following operations, i.e., Deploy → Validate → Train → Update → Deploy. The high-level system overview of DeepCon is shown in Fig 5.2.

The system achieves the following three goals.

- **Seamless OTA.** DeepCon utilizes the computing resources both from edge devices and the cloud to achieve seamless OTA. This algorithm and system co-design solution ensures the non-stop update of the models on the host devices.

- **Generability.** We develop a generic deep learning model updating algorithm for easy fine-tuning of models without being affected by the number of models and the model architectures.

- **Scalability.** The proposed model updating algorithm is able to fine-tune the models in parallel and automatically scale to multiple computing nodes. Therefore, if the application developers want to reduce update latency, he/she only needs to add more computing resources and require zero code changes.

5.3 Design of DMML

In this section, we describe the design of DMML. Following the phenomenon observed in Table 5.1(b), we first provide a formal definition of the consistency in DeepCon (Section 5.3.1). Then we present a transformation that transfers the problem

	Metrics				
	Acc_1	Acc_2	C	CC	<Acc-CC>
CIFAR-10					
ResNet20+ResNet20	91.73	91.78	92.65	88.56	3.17
ResNet20+ResNet56	91.73	93.27	92.51	89.30	2.43
ResNet20+VGG13	91.73	93.26	92.34	89.15	2.58
CIFAR-100					
ResNet20+ResNet20	66.86	67.47	70.00	58.28	8.58
ResNet20+ResNet56	66.86	70.47	69.89	59.47	7.39
ResNet20+VGG13	66.86	71.57	68.75	59.74	7.12

Table 5.1: Different Metrics Reported on CIFAR10/100

of improving the consistency of various models to knowledge distillation tasks (section 5.3.2). Finally, we develop an online knowledge distillation-based method to solve the problem (section 5.3.3).

5.3.1 From Accuracy to Consistency

The key idea of this paper is to improve the consistency between the models, while not sacrificing individual model accuracy. In this section, we first provide a definition of consistency, and compare the difference between accuracy and consistency.

Accuracy (Acc). Accuracy measures the probability of a model correctly predicting the ground truth labels. For any given task, a set of N models $\{M_1 \ ... \ M_N\}$, Dataset $D(X, Y)$, we have the following definition:

$$Acc(M_n) = \mathop{\mathbb{E}}_{(x,y)\sim D}[1\{M_n(x) = y\}] \tag{5.1}$$

For any model M_n and data $\{x, y\} \in D(X, Y)$, $Acc(M_n)$ measures the probability of a model M_n correctly predicting the labels y.

Consistency (C). Consistency measures the probability of multiple models producing the same result given the same input. Given $\{x, y\} \in D$ and N models $\{M_1 \ ... \ M_N\}$, we have the following definition:

$$C(M_1...M_N) = \mathop{\mathbb{E}}_{(x,y)\sim D}[1\{M_1(x) = ... = M_N(x)\}] \tag{5.2}$$

C measures the probability of models $\{M_1...M_N\}$ predicting the same result at a given point x.

Correct Consistency (CC): the Intersection between Accuracy (Acc) and Consistency (C). Consistency (C) only measures if all models produce the same results, no matter if they are correct or not. In real-world applications, we are more interested in *consistent and correct predictions* for a set of models within the same application. Thus, we use correct consistency (CC) to define this as shown in Eq. 5.3.

$$CC(M_1...M_N) = \mathop{\mathbb{E}}_{(x,y)\sim D}[1\{M_1(x) = ... = M_N(x) = y\}] \tag{5.3}$$

CC measures the probability of multiple models $\{M_1$ ... $M_N\}$ correctly predicting the target y at the same input point x.

CC is at the intersection between the Consistent (C) and correct outputs (Acc) produced by all the models, as formulated in Eq. 5.5.

$$CC(M_1...M_N) = C(M_1...M_N) \cap Acc(M_1)... \cap ...Acc(M_N) \tag{5.4}$$

Theoretical Gap of Improvement between Accuracy and Correct Consistency: Gap <Acc-CC>. For a set of reported model accuracy $Acc(M_1)$ ... $Acc(M_N)$, the maximum ideal CC is $\min\{Acc(M_1), ..., Acc(M_N)\}$: the accuracy of the smallest/worst performed model.

Thus for multiple models $\{M_1$ ... $M_N\}$, the gap for improvement is defined as:

$$< Acc - CC >= \min\{Acc(M_1)...Acc(M_N)\} - CC(M_1...M_N). \tag{5.5}$$

Table 5.1 compares the Top-1 accuracy and consistency metrics (C, CC, and <Acc-CC>) of CIFAR-10 and CIFAR-100 dataset obtained from our pre-trained models. Three pairs of model parameters are generated as follows: (i) The same model architecture but trained twice with different initialization parameters (ResNet20 and ResNet20). (ii) Models with the same backbone module but with different depths (ResNet20 and ResNet56). (iii) Models with totally different architectures (ResNet20 and VGG13).

From the experimental results, we have the following observations: (i) The inconsistency is ubiquitous even for the same model that is trained twice with different initialization weights. (ii) The Gap <Acc-CC> between (Acc) and (CC) increases

with more complex tasks, from around 3% in CIFAR10 to around 8% in CIFAR100.

The gap <Acc-CC> may come from different model architectures, random initialization weights, different data orders [264], etc. This motivated the paper to build a system that can efficiently improve the *CC* of a distributed DL application.

5.3.2 Problem definition

To ensure the consistency of various models, we have the following optimization goal.

$$\begin{aligned} \arg\max_{M_1..M_N} \quad & CC(M_1, ..., M_N) \\ \text{s.t.} \quad & Acc(M_n; D) - Acc(M_n^{'}; D) <= \xi \\ & \forall M_n \in \{M_1..M_N\} \\ & \xi \geq 0 \end{aligned} \tag{5.6}$$

For any dataset D, we aim to maximize the CC for all models $\{M_1 \ ... \ M_N\}$. Denote M_n and $M_n^{'}$ as models before/after re-parameterization, we also add a constraint for model accuracy. The slack factor ξ allows the optimized $M_n^{'}$ to stand a maximum accuracy loss by ξ as compared to the original M_n. For most of our experiments, we set $\xi = 0$, such that we ensure all models do not drop accuracy during the optimization process.

Solving the optimization problem with Knowledge Distillation. Knowledge distillation (KG) [119] allows teaching or "distilling" the knowledge from one or a set of models to other models, which naturally meets the optimization goal illustrated in Eq. 5.6. To transfer the optimization problem to a KD task, we first have the following goals:

$$\arg\min_{M_S} {}_{\{x,y\}\in D}\mathcal{L}(M_S(x), y) + \mathcal{L}(M_S(x), M_T(x)) \tag{5.7}$$

Given a teacher model M_T and labeled dataset $D(X, Y)$, the goal of KD is to generate a student model M_S by learning from both M_T and $D(X, Y)$ [119]. Two loss functions $\mathcal{L}(M_S(x), y)$ and $\mathcal{L}(M_S(x), M_T(x))$ measures the difference between M_S and D, M_T respectively. By minimizing both loss functions at the same time, the learned M_S retains the knowledge from M_T and the dataset D.

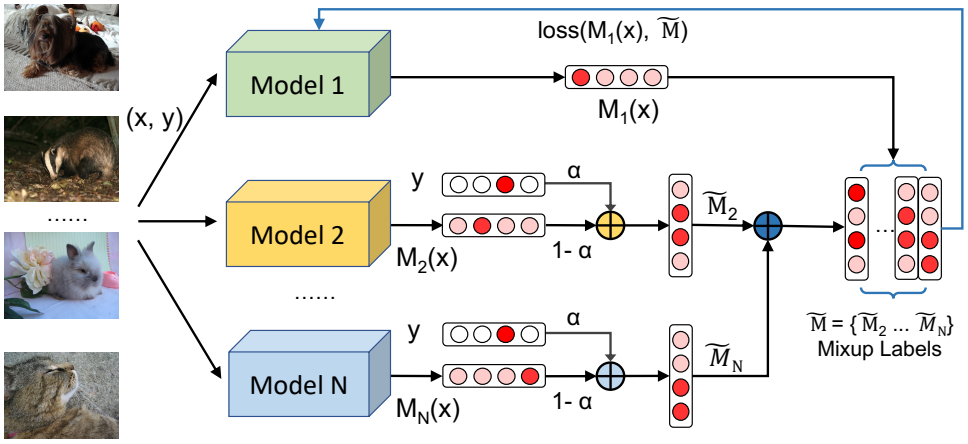


Figure 5.3: The overview of **Basic Computation of Deep Mixup Mutual Learning (DMML)**. The same inputs are fed to all models and get results M_1 ... M_N. Then, each M_n and true label y jointly generate a mixup label $\tilde{M}_n$ controlled by a weighting parameter α. For each model, the loss function is computed by comparing its output M_x against all other mixup labels $\tilde{M}$. For example, L_1 is computed by M_1 and $\tilde{M} = \{\tilde{M}_2 ... \tilde{M}_N\}$.

We note the ground truth label y could also be regarded as the output of a perfect model. In order to agree with the output of all models and the ground truth label, our optimization goal could be formulated as:

$$\arg \min_{M_1..M_N} {}_{\{x,y\}\in D}\mathcal{L}(y, M_1(x)...M_N(x)) \tag{5.8}$$

Whereby the loss function measures the difference between all models and the ground truth label. However, it is infeasible to optimize all the models at the same time. More practically, we can recursively reparameterize a model to minimize its difference against all other models and the ground truth label:

$$\arg \min_{M_n} {}_{\{x,y\}\in D}\mathcal{L}(y, M_1(x)...M_N(x)), \forall M_n \in \{M_1..M_N\} \tag{5.9}$$

The remaining problem is to construct the loss function $\mathcal{L}(M_n; D)$ of each model M_n on dataset D, which will be discussed in the following section.

5.3.3 Basic Deep Mixup Mutual Learning (DMML)

In order to minimize the loss function defined in E.q 5.9, we design a DMMLalgorithm that consists of two components *Deep Mixup Label* and *Multi-model Distillation* (see

Fig. 5.3).

5.3.3.1 Deep Mixup Label

Deep Mixup Label is a learning target that allows DMML to improve consistency among models while ensuring that individual models still maintain or even improve original accuracy after training. The Deep Mixup Label is a weighted average between the model's output and ground truth label as defined in Eq. 5.10.

$$\tilde{M}(x, y, M_n, \alpha) = \alpha M_n(x) + (1 - \alpha)y \tag{5.10}$$

At point $\{x, y\}$, $\tilde{M}$ averages the ground truth label y and pseudo label (output) generated from model $M_n(x)$, controlled by the weighting parameter α. The Deep Mixup Label can be applied to most deep learning tasks, $M_n(x)$ can be the output of any form generated by the models, i.e., bounding box coordinates for detection tasks. For the classification task we evaluate in this paper, $M_n(x)$ can be the outputs after softmax indicating the probabilities of one sample x belonging to any class $y \in Y$.

5.3.3.2 Multi-model Distillation

Fig. 5.3 shows the details of how to use multiple generated Deep Mixup Labels to train a target model through DMML. Assume we have N models in total, and we would like to update the k-th model M_k. Then, for each of other $N - 1$ models and data points $\{x, y\}$, we extract the computed Deep Mixup Label $\tilde{M}(x, y, M_n, \alpha)$, $\forall n \in N, n \neq k$. For each Deep Mixup Label, we compute the difference of $M_k(x)$ against the Deep Mixup Label $\tilde{M}(x, y, M_n, \alpha)$ with a loss function. All loss functions are summed and averaged to get the final loss function for model M_k, as shown in E.q 5.11:

$$\mathcal{L}(M_k, \{x, y\}) = \frac{1}{N-1} \sum_{n=1, n\neq k}^{N} \mathcal{L}(M_k(x), \tilde{M}(x, y, M_n, \alpha)) \tag{5.11}$$

Finally, the computed loss is sent back to the model M_k for computing the gradients and updating the model parameters. (see example M_1 in Fig. 5.3). In this paper, we use cross entropy for computing each loss in E.q 5.11.

Algorithm 4: Deep Mixup Mutual Learning (DMML)

Input: Training data D (X,Y), learning rate $\gamma(t)$, Mixup Ratio α, N pre-trained models $M = \{M_1, M_2 \ ... \ M_N\}$
Initialize: $t \leftarrow 0$
while *Not Convergence* **do**
  for *randomly sample batch data* $\{x,y\} \in D$ **do**
    $\tilde{M} \leftarrow new\ list()$
    for $n \in N$ **do**
      // Compute $\tilde{M}_n(x, y, M_n, \alpha)$ with E.q. 5.10
      $\tilde{M}_n \leftarrow \tilde{M}_n(x, y, M_n, \alpha)$
    end
    for $n \in N$ **do**
      $\mathcal{L}_n \leftarrow 0$
      for $k \in N$ **do**
        if $k \neq n$ **then**
          $\mathcal{L}_n \leftarrow \mathcal{L}_n + \mathcal{L}(M_n, \tilde{M}_k)$
        end
      end
      $\mathcal{L}_n \leftarrow \mathcal{L}_n \ / \ (N-1)$
      $M_n \leftarrow M_n + \gamma_t \frac{\partial \mathcal{L}_n}{\partial M_n}$
    end
  end
  t = t + 1
  update learning rate $\gamma(t)$
end

The design of mixup labels and loss function of multi-model distillation forces the model to update parameters such to produce consistent outputs compared to other models. The accuracy of the models, though, remains stable during training as we mixup up the ground truth label in the *Deep Mixup Labels*. We carefully tuned α such that the consistency is maximized while the accuracy of individual models is preserved. We validate the effects of mixup ratio α in Fig. 5.8 as well.

5.3.3.3 DMML Algorithm

Based on Deep Mixup label and multi-model Distillation, we develop an algorithm, namely DMML, that optimizes all models' parameters simultaneously. Alg. 4 shows detail of the proposed DMML.

For a given model $M_n \in M$ and the sampled data $\{x, y\}$, we compute the Deep Mixup

Label $\tilde{M}_n$ (see line 7). Then, we use E.q. 5.11 to compute the average of the difference between the output of M_n and other deep mixup labels (from line 10 to 19). Next, we update $\mathcal{L}_n$ and M_n as shown in line 17 and 18. In each iteration, the learning rate $\gamma(t)$ will be updated based on convergence curves to speed up algorithm convergence [204]. Obviously, DMML is not very scaled with the increasing number of models. The next subsection introduces a parallel DMML algorithm.

5.4 Over-the-Air Update in DeepCon

Over-the-air (OTA) updates play a crucial role in distributed deep learning applications, ensuring seamless integration of the latest advancements and improvements, empowering the models to continuously evolve and adapt to emerging system failures. By integrating computation and communication across the edge and the cloud, OTA updates can accomplish the fine-tuning of the deployed models via efficient data sampling and parallel computation, thereby significantly improving the update efficiency. The key observation in DMML is that the models are updated synchronously in the central server with a global data set. In DeepCon, we propose the OTA update approach of distributed models with data sampling and asynchronous parallel training, emphasizing the coordination between the cloud and edge nodes for real-time monitoring and agile parallel training in order to rapidly enhance model consistency after deployment.

The overview of implementation on OTA is shown in Fig.5.4, and its high-level API call are listed in table 5.2. Finally, we discuss how to improve the scalability of the proposed method (section 5.4.2).

5.4.1 Over-the-air update in DeepCon

Cloud. The cloud holds replicas of all models within the system and implements the *OTA Controller* (see Fig. 5.2) via multiple modules for coordinating with the GPU cluster and various edge nodes, updating and deploying models. When the *Consistency Validation* module detects the inconsistency of models, the *Training Task Scheduler* will be triggered and performs the model fine-tuning process over GPU cluster. The

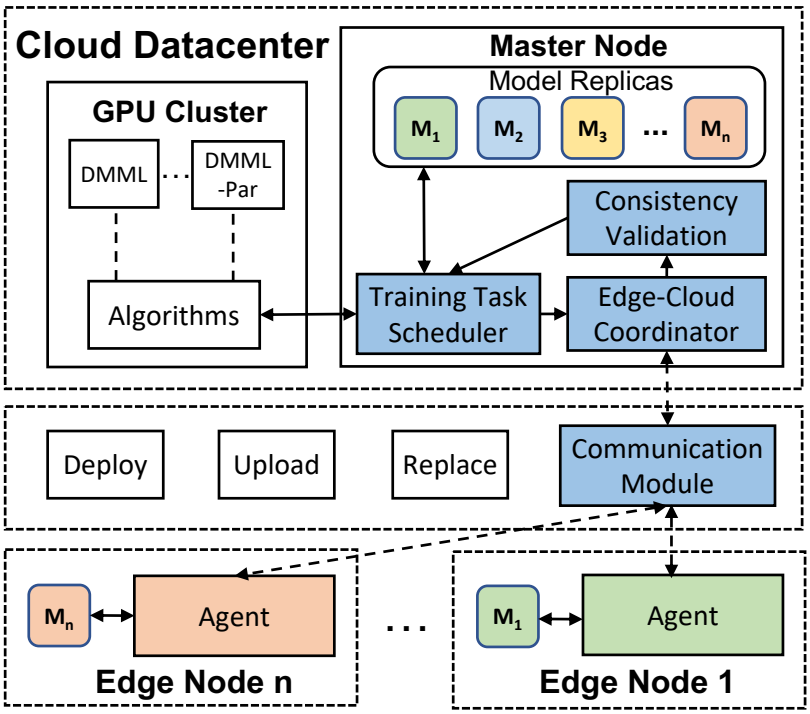


Figure 5.4: The High-level Implementation of DeepCon

updated models, thereafter, are pushed to the corresponding edge nodes for further deployment. The *Edge-Cloud Coordinator* communicates with the GPU cluster, CPU server (Master node), and edge nodes through the *Communication Module* to perform the following operations, i.e., model deployment, data uploading, model training, model validation, and model update.

Edge node. One each edge node, a pre-trained model is deployed and makes predictions for incoming stream data. The *Agent* caches the data samples and their inference results and shares them with the cloud in a defined time interval. These data will be used to check the consistency of models deployed on various edge nodes.

I: Training Task Scheduler. The scheduler is designed to orchestrate training algorithms to update the models in the GPU cluster. We, therefore, define two main APIs: $trainModels(w, m, t)$ and $getModels(t)$. $trainModels(w, m, t, a)$ defines that in task t a set of models m are trained on w workers (i.e., GPU nodes) via algorithm a. $getModels(t)$ returns the updated models in task t.

II: Consistency Validation. After the sampled inference results and data are gathered, $getValid(m, r, l)$ check the consistency of a set of models m based on the inference results r and true labels l of the collected sample data. The labels l can be obtained

Table 5.2: DeepCon APIs

Cloud APIs	Description
$trainModels(w, m, t, a)$	In task t a set of models m are trained on w workers via algorithm a.
$getModels(t)$	Returns the updated models in task t.
$getValid(m, r, l)$	Returns the CC of a set of models m based on the inference results r and true labels l of the collected sample data.
$getDeploy(m, e)$	Specifies the pre-trained model m to be deployed on edge node e.
$getReplace(m, e)$	Replace the running model on the edge node e with the updated model m.
Edge node APIs	**Description**
$install(m)$	Builds a instance of model m.
$migrate(m_1, m_2)$	Moves the input streams from model m_1 to model m_2 for inference.
$stop(m)$	Stop the instance of model m.
$getUpload(d, r)$	Upload the sampled data stream d and their corresponding inference results r to the cloud.

through automated data labeling method [60, 172] or human workers (e.g., Amazon Mechanical Turk).

III: Communication Module. This module interacts the Edge-Cloud Coordinator in Cloud with the Agent in each edge node to provide seamless OTA solution. To this end, in Edge-Cloud Coordinator the $getDeploy(m, e)$ specifies the pre-trained model m to be deployed on the edge node e. Next, the $getReplace(m, e)$ is designed to replace the running model on the edge node e with the updated model m. To provide "no stop" model replacement operation, the three high-level APIs i.e., $install(m)$, $migrate(m_1, m_2)$ and $stop(m)$ in edge node agent are design. $install(m)$ builds a instance of model m, and $migrate(m_1, m_2)$ moves the input streams from model m_1 to model m_2 for inference, and $stop(m)$ is to stop and remove the instance of model m. Finally, the $getUpload(d, r)$ uploads the sampled data stream d and their corresponding inference results r to the cloud for further validation. In this paper, we assume that both cloud and edge node have right to access the raw data, and the privacy preserving will be considered in future work.

5.4.2 Parallel Training of DMML (DMML-Par)

In this subsection, we propose DMML-Par, an approach that parallelizes the DMML, which can be scaled up to multiple GPU nodes in a distributed manner. Alg. 2 illustrates the details of DMML-Par. We develop an asynchronous mechanism that allows DMML-Par to offer a lock-free, non-wait model parameter exchange solution during the fine-tuning phase.

Algorithm 5: Parallel Deep Mixup Mutual Learning (DMML-Par)

Input: Training data D (X,Y), learning rate $\gamma(t)$, Mixup Ratio α, N pre-trained models $M = \{M_1, M_2 \ ... \ M_N\}$, GPU workers $c \in C$, server S

Initialize: t $\leftarrow$ 0
Server model replica: $M^S \leftarrow M$
run_model $\leftarrow$ new list([False for n in N])
nodes_idle $\leftarrow$ new list([True for c in C])

while *Not Convergence* **do**
  for $c \in C$ **do**
    if *nodes_idle[c] == False* **then**
      continue
    else
      for $n \in N$ **do**
        if *run_model[n] == False* **then**
          nodes_idle[c] $\leftarrow$ False
          run_model[n] $\leftarrow$ True
          // Remote procedure call to train M_n on worker c
          Train(M_n, α, t, c, n)
          break
        end
      end
    end
  end
  t = t + 1
  update learning rate $\gamma(t)$
end

Update(c, n, M_n)**:**
// Update M_n in S
$M_n^S \leftarrow M_n$
nodes_idle[c] $\leftarrow$ True
run_model[n] $\leftarrow$ False

Train(M_n, α, t, c, n):
// Worker c pulls latest model from S
$M \leftarrow M^S$
for *randomly sample batch data* $\{x,y\} \in D$ **do**
  $\mathcal{L}_n \leftarrow 0$
  for $k \in N$ **do**
    if $k \neq n$ **then**
      // Compute $\tilde{M}_k(x, y, M_k, \alpha)$ with E.q. 5.10
      $\tilde{M}_k \leftarrow \tilde{M}_k(x, y, M_k, \alpha)$
      $\mathcal{L}_n \leftarrow \mathcal{L}_n + \mathcal{L}(M_n, \tilde{M}_k)$
    end
  end
  $\mathcal{L}_n \leftarrow \mathcal{L}_n \ / \ (N-1)$
  $M_n \leftarrow M_n + \gamma_t \frac{\partial \mathcal{L}_n}{\partial M_n}$
end
// Remote procedure call to update model n and release the worker c
Update(c , n, M_n)

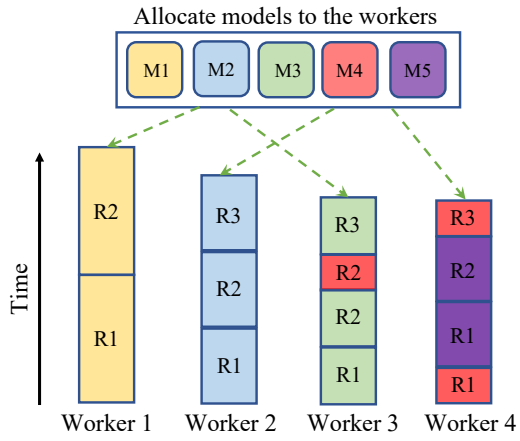


Figure 5.5: Example of DMML-Par on 5 Models and 4 workers

We first initialize two lists *run_model* and *nodes_idle* for maintaining the states of the models and workers, where their index represents the corresponding worker and model. All models are not yet allocated and all workers are not occupied (see line 4 and 5). Then for all the workers $c_n \in C$ in sequence, we first check if worker c_n is idle (line 8), and if not we will move to the next idle worker. If the worker is idle, we check the model status in *run_model* and allocate the selected model $M_n \in M$ to an idle worker c_n. Then, M_n and worker c_n is marked as running and occupied (line 13). At the same time, the training process is distributed to the target worker c_n via **Train()**.

For training M_n (from line 30 to 46), the first state is to pull the latest model replica from the global model replica M^S. Then, we compute the deep mixup labels and loss against $M_n(x)$, compute the gradients, and update M_n. We iterate through the dataset D until all data has been trained once (from line 33 to 43). Finally, the **Update()** function is called to update the M_n to the server and release the worker c (from line 25 to 29).

Discussion. The asynchronous design of DMML-Par ensures maximum utilization of available computation resources across the GPU clusters. We ensure that all models get the same rounds of training, such to mitigate the performance drop brought by the asynchronous parallel training (see Fig. 5.7). DMML-Par is also easily scalable and adaptive to a dynamic number of worker nodes, to meet the latency requirements of the users.

Example of DMML-Par with 5 models on 4 workers. Fig. 5.5 shows how DMML-Par

allocates the models to various workers for parallel training. In the first round, we train models 1 to 4 on workers 1 to 4 respectively. When worker 4 finishes its process, only model 5 (M_5) is not trained. Thus, M_5 is allocated to worker 4. Next, when worker 3 completes the training process for M_3, two models i.e., M_3, and M_4 are able to be allocated to worker 3 (the other three models are still doing first-round training). Since M_3 was trained on worker 3 already, we directly run the second-round training for M_3 on worker 3. Following the same scheduling policy, we allocate M_2, M_1, and M_4 to workers 1, 2, and 3, respectively. Notably, before starting a training process on a worker and a training process finishes, DMML-Par will pull the latest version of all models from the master node (see Alg. 2 line 32) and push the trained model to the master node (see Alg. 2 line 27), this operation allows the models to be updated asynchronously maximizing the utility of computing resources.

5.5 Evaluation

5.5.1 Experiment Setup

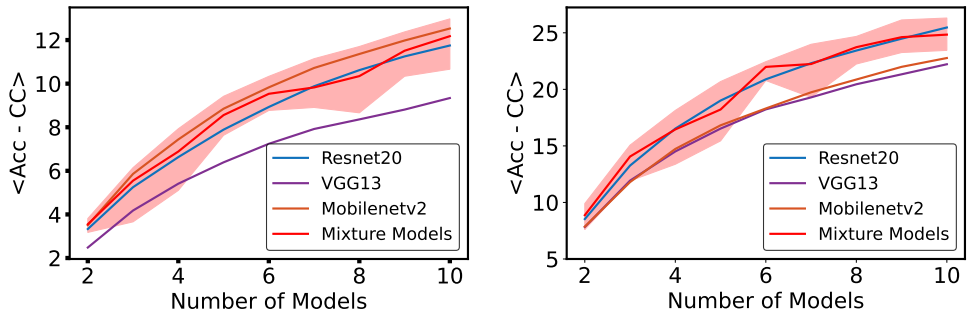
Datasets. CIFAR-10 and CIFAR-100 [158] datasets contain 10 and 100 classes respectively. They are split into 50,000 images for training and 10,000 for testing, with each 32 x 32 colored pixel. IMDB [189] is a binary text classification task that contains 25,000 data samples for both the train and test sets.

Models. For the image classification task, we use three network architectures and vary their architecture depth when generating models: ResNet-20, ResNet-56 [115] and MobileNetv2_x0_5, MobileNetv2_x1_4 [124] and VGG13 [259]. For text classification task, 5 different structures are implemented for the experiments: TextRNN, BiLSTM, TextCNN, TextRCNN and Self-Attention. All models use pre-trained glove embeddings for feature representation.

Cluster Setup. We implement all neural networks and training processes with Pytorch and perform experiments on a Ubuntu 16.04 Linux server equipped with 5 Nvidia Tesla V100 GPU, 40 Intel Gold 5218 CPUs, and 100GB memory. All pre-trained models are generated by ourselves serving as base models for further fine-tuning. For pre-training

CIFAR models we use SGD optimizer and the initial learning rate is set to 0.1, 100 epochs and drop every 30 epochs. All fine-tuning learning rate is set to 0.01, 50 epochs, and drop every 20 epochs. For IMDB training, ADAM is used and the learning rate is set to 0.0001 for both pre-training and fine-tuning with 200 and 100 epochs respectively. We also construct 100-dim glove embeddings for IMDB and fix the embedding during all experiments. Data augmentation is applied for both vision and language tasks: vision datasets are augmented with random crops of 4 padding sizes and horizontal flips; language tasks randomly replace words with default tokens. Note that for pre-training we do not aim to reach the best-benchmarked accuracy for respective model architectures and datasets. For all our settings, the pre-trained models have converged to a stable point and we show the benefits of different techniques for improving the CC and reducing the <Acc-CC> gap.

Baselines. We compare our method against the following baselines. 1) *Vanilla-KD* [119] assume all students learn from the teachers and true labels with distillation loss and cross-entropy loss. KD is widely studied for training small networks that mimics the behaviors of large networks. 2) *Label Smooth* [20, 140, 274] regulates the true label and is capable of reducing model churn during training. We extend KD with label smoothing for comparison. 3) *Mixup Distillation* [140] proposes distillation based training for reducing model churn. We extend the method to multiple models where all models but the best-performed model learn from the best models. 4) *Ensemble Distillation*: Ensemble methods are not directly applicable to models with different architectures. We implement ensemble distill where each model learns from the ground truth label and the average ensemble logits from all models in the group. 5) *KDCL* [107] is similar to ensemble distillation but implements random augmentation to different models during model training. The training loss consists of both entropy loss with true label and distillation loss with ensemble digits. 6) *Deep Mutual Learning* [334] is an online-KD method that jointly learns multiple models at the same time. The loss consists of entropy loss with true label and pair-wise distillation loss against all other models. 7) *Co-Distillation* [13] is another online-KD method that enables parallel training of multiple models. The training loss consists of entropy loss and distillation loss against average logits over all other models. For a fair comparison,



(a) CIFAR-10 (b) CIFAR-100

Figure 5.6: Acc and CC Gap (Eq. 5.6) with Different Model Numbers and Architectures.

all baseline results and DMML are implemented with the same optimizer and training epochs. We vary the weighting parameters and report the results with the best *Correct Consistency* (CC) metric.

Metrics. The evaluation metrics we used in the paper include: 1) Correct Consistency (E.q 5.3 CC): the percentage of all models to all produce the positive results given the same input. 2) Performance gap between ACC and CC: <Acc-CC> = $\min\{Acc(M_1), ..., Acc(M_N)\} - CC(M_1...M_N)$ (E.q 5.5).

All experiments are repeated 5 times with average value, and standard deviation reported in the results. We only report results that maintain or increase individual model accuracy, making sure no accuracy loss during training.

5.5.2 Identify and Quantify Gap between Acc and CC

We have revealed in Table. 5.1 that CC between models is always inferior to any model's Acc. The maximum CC between the models can not exceed the accuracy of the worst performed model, i.e., $CC_{max}(M_1..M_N) = min\{Acc(M_1)\ ...\ Acc(M_n)\}$. Ideally, we want to increase CC such that when the worst-performed models correctly classify an input, all other models produce the same correct predictions as well.

As the goal of DeepCon is to improve the CC metric among models, we first need to quantify the theoretical maximum gap between a set of pre-trained models. In this subsection, we quantify the extent of the <Acc-CC> gap with different numbers and architectures of models. We repeat training Resnet20, VGG13, and MobileNetv2_x0_5

	CIFAR10		CIFAR100		IMDB	
	CC ↑	<Acc-CC> ↓	CC ↑	<Acc-CC> ↓	CC ↑	<Acc-CC> ↓
	ResNet20 + VGG13		ResNet20 + VGG13		TextRCNN + SelfAtten	
Pretrained	89.15	2.58	59.74	7.12	62.76	11.89
vanilla kd	89.34±0.05	2.69±0.07	60.02±0.07	7.36±0.18	66.58±0.43	7.79±0.38
ls	89.45±0.06	2.63±0.06	59.79±0.02	7.35±0.08	67.12±0.23	7.53±0.13
mixup	89.48±0.18	2.64±0.13	60.31±0.11	7.19±0.13	67.21±0.32	7.44±0.41
ensemble-distill	90.00±0.21	2.21±0.16	61.81±0.07	5.69±0.15	67.44±0.18	6.66±0.26
kdcl	89.87±0.14	2.21±0.12	61.84±0.11	5.95±0.05	66.89±0.22	6.74±0.17
dml	90.44±0.12	1.88±0.12	62.59±0.24	5.06±0.23	67.35±1.15	6.29±0.23
co-distill	90.40±0.05	1.96±0.09	**62.91±0.07**	5.01±0.18	67.30±0.74	6.25±0.45
dmml	**90.74±0.07**	**1.7±0.02**	62.83±0.06	**4.57±0.18**	**69.75±0.13**	**5.14±0.33**
	5 Models		5 Models		5 Models	
Pretrained	84.27	8.34±0.78	50.35	19.69±2.13	46.86	26.61±1.97
vanilla kd	85.00±0.10	7.85±0.61	50.82±0.15	19.4±1.90	52.67±0.10	21.88±1.16
ls	85.29±0.09	7.64±0.62	50.77±0.05	19.57±2.09	53.01±0.26	21.62±1.10
Mixup	85.61±0.11	7.36±0.63	52.35±0.20	18.32±2.08	53.3±0.03	21.32±1.09
ensemble-distill	86.85±0.17	6.39±0.73	55.22±0.20	16.45±2.27	51.41±2.61	21.96±0.79
kdcl	86.51±0.20	6.66±0.70	54.34±0.15	17.26±2.25	51.05±3.21	22.18±0.91
dml	87.50±0.17	5.80±0.67	56.11±0.18	15.74±2.20	57.21±0.24	18.2±0.66
co-distill	87.23±0.10	6.05±0.72	55.64±0.09	16.15±2.29	57.21±0.07	18.31±0.63
dmml	**88.17±0.07**	**4.98±0.52**	**57.38±0.07**	**13.93±2.11**	**59.45±0.19**	**16.31±0.59**

Table 5.3: Correct Consistency (%) and Acc,cc Gap on the CIFAR10/100 and IMDB Dataset with 2 and 5 Models.

20 times and report the average gap between model accuracy and consistency. We then report the mean, upper, and lower bound of the gap.

Figure 5.6 shows our benchmarking results. We make the following three observations: 1) The <Acc-CC> gap increases with the increase of model numbers. When more models join the evaluation, the gap increase is almost linear to the model numbers, for both CIFAR10 and CIFAR100. In CIFAR-100, this gap is over 20% for all cases with 10 models, revealing a huge gap between the model accuracy and consistency. 2) For the same model, MobileNetv2 and ResNet20 report the highest gap on two datasets. Also, we evaluated the mixed model scenario, its gap is highest with large variance as well, compared to the case where models have the same architecture. 3) Gap increases with the complexity of the tasks: Comparing both datasets, CIFAR100 report nearly double the gap compared to CIFAR100 under all circumstances. The above findings necessitate the need to improve model consistency, which is paramount for geo-distributed deep learning applications with multiple models and varying configurations.

5.5.3 DMML Performance on Vision and Language Tasks

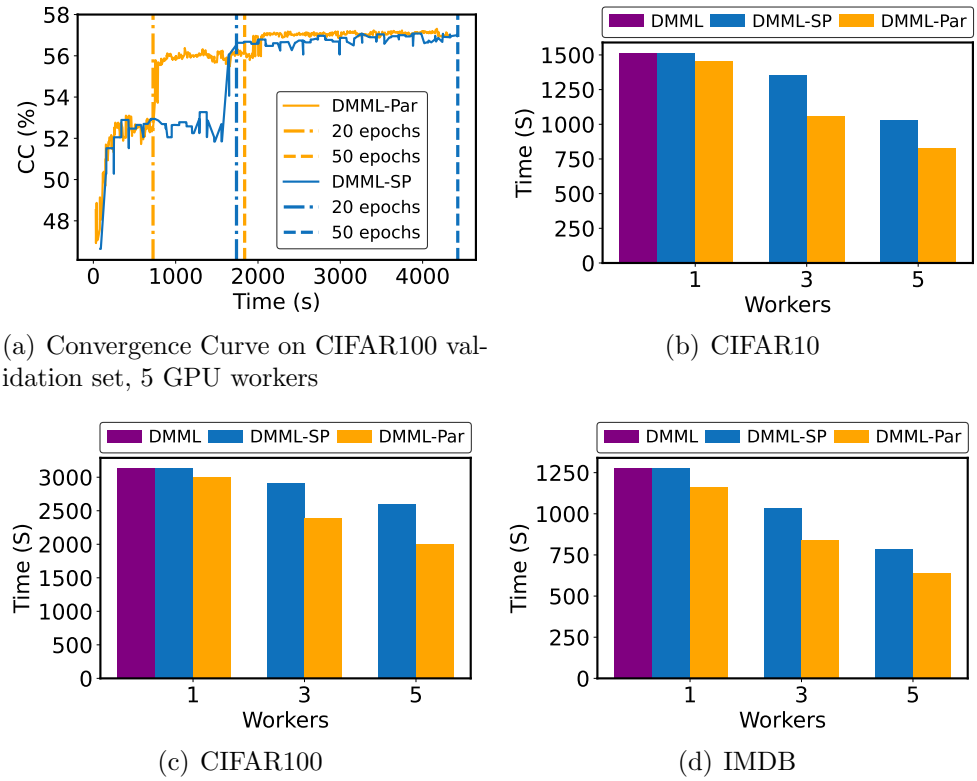
Table 5.3 compares the performance of DMML to other baselines on three popular datasets: CIFAR10, CIFAR100 and IMDB. We conduct 2 sets of experiments with 2 models and 5 models respectively.

Generally, in all experiments, the implemented baselines as well as DMML are capable of improving the consistency *CC* among the models, indicating that cross-model learning is effective in generating more similar models, regardless of the model architectures and parameters. Online KD methods (e.g., dml, codistill, dmml, kdcl and ensembledistill) are more effective as compared to offline-KD (e.g., vanilla kd, label smooth, mixup) in overall performance.

In experiments with 2 models, DMML is able to reduce about 34.1%, 35.8%, and 56.8% of <Acc-CC> for 3 datasets respectively. DMML achieves higher *CC* than baseline solutions in most cases. However, DMML reports slightly worse CC than codistillation on ResNet20 + VGG13, CIFAR100 experiment. This is because the co-distillation improves all models' Acc thereby resulting the better CC. However, this method only works in rare cases and very randomly.

On IMDB dataset (NLP), we observe much higher inconsistency for pre-trained TextRCNN and SelfAttention models, due to the bigger difference between their model architectures. After applying our methods to the models, the CC can be greatly improved, meaning that DMML is more effective for models of greater difference. KDCL [107] as reported in the paper can improve model invariance by adding image distortion for each model but is less effective in improving model consistency. In the following, we conduct experiments with 5 models of different architectures to show the scalability of DMML in improving model consistency.

Comparing the experiments of 5 models and 2 models, we see a larger difference of <Acc-CC>: 8.34%, 19.69%, and 26.61% for the three datasets. After applying DMML, the <Acc-CC> has reduced to 4.98%, 13.93%, and 16.31%, indicating 40.28%, 29.25% and 38.70% reduction to this gap. Comparing the computer vision tasks and NLP tasks, we also see a larger gap in the latter, despite that all models use the same glove embeddings at the first layer. In the future, we will investigate the performance of



(a) Convergence Curve on CIFAR100 validation set, 5 GPU workers

(b) CIFAR10

(c) CIFAR100

(d) IMDB

Figure 5.7: Evaluation of DMML-Par with 5 models on 3 datasets

DMML for other basic language tasks, i.e., semantic parsing [117].

5.5.4 Performance of DMML-Par

Fig. 5.7 plots the performance of DMML-Par implemented in our DeepCon. We train the model on 1, 3, and 5 workers (GPU nodes) with three datasets. As a comparison, we extend DMML to a simple synchronous parallel training algorithm namely DMML-SP. We iteratively allocate model training tasks to available workers and wait for all the models to be trained once. Then we update the whole server model replica and push them to the next round of training. We also report the performance of DMML as a baseline. All training setup and parameter setting are the same as the DMML, including the Mixup ratio, the learning rate, and optimizers.

Fig. 5.7(a) shows the convergence curve of DMML-SP and DMML-Par with 5 models on the CIFAR100 validation set. Generally, we see that the training speed of DMML-Par is much faster than DMML-SP. It takes 725 and 1838 seconds for DMML-Par to complete 20 and 50 rounds of model training, nearly 60% reduction as compared to DMML-SP (i.e., 1739 and 4427 seconds respectively.) As far as the CC metric, DMML-

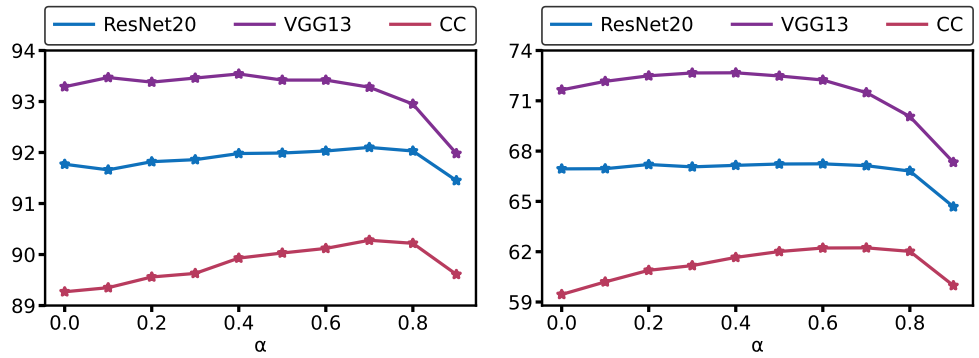
Par also reports faster and stable growth as compared to the DMML-SP. After finishing 50 rounds of training, the highest CC reported is 56.45, slightly worse than the best CC (57.38) reported in Tab. 5.3. However, when we continue training for a few rounds, we can still guarantee the best CC as compared to synchronous DMML-SP.

Fig. 5.7(b), 5.7(c), and 5.7(d) report latency of three algorithms. We report the minimum time it takes for each algorithm to reach the best CC as marked by Tab. 5.3. Overall, on three datasets, DMML-Par can reduce 20%, 14%, and 20% training time compared to DMML-SP and DMML on average, while reaching the same CC. In reality, we can opt to sacrifice few CC for much less training time, e.g., in Fig. 5.7(a), DMML-Par already achieves 56.12% CC (1% less than best CC) when training for only 1100s, much faster than 1950s when reaching the best CC as reported in Fig. 5.7(c).

Also, we can observe that the training time of both DMML-SP and DMML-Par decreases with the increase in the number of workers. However, DMML-SP has to wait for all models to finish their round of training and then update the models, which may cause some nodes to be idle.

5.5.5 *The impact of parameter* α

We analyze how the weight parameter α affects the model accuracy and correct consistency (CC) metric. In Fig. 5.8 we report the hyper-parameter tuning results of α on the two datasets, with varying the α from 0 to 0.9.



(a) CIFAR-10 (b) CIFAR-100

Figure 5.8: Metrics (%) with Weighting Parameter α, Resnet20 + VGG13

We notice that with the increase of α (more model output in the label mixup), the CC metric is getting better, while the accuracy remains stable, and slightly outperforms

the accuracy of the pre-trained models respectively. The maximum point is reached around 0.7/0.8 and then starts to drop when further increasing α. This indicates that around 20% of information from the true label is already a strong indicator, enough to keep the model stable from the accuracy perspective.

5.6 Related Work

Model Consistency. model consistency measures the ability of models to produce the same outputs when fed with same/similar inputs. It is different from accuracy in that the later measures the prediction of each individual model, without considering the relationship between different models.

Model consistency has been studied in several related areas by different terminologies. [145, 208] attempt to quantify the disagreement level between two networks trained on the same dataset. They reveal that model disagreement could arise from different architectures and initializations, training samples and optimizers. [252, 264] study the reproducibility problem during model re-training and conclude that factors like activation functions or data order could lead to drastic prediction differences between the fine-tuned model and the base model. [67, 340] report model instability which measures the output variation of a given model when slight perturbation is added to the input sources. A stability loss is added to the training loss to mitigate the model instability. *Prediction Churn* [20, 30, 140] propose churn, another definition to measure model consistency during the training process. By using techniques such as mixup label [140], adaptive label smoothing [20], models at different phases are forced to produce same results during the training process. [290] is most related to the definition in our paper, which apply ensemble-based techniques during model training to improve model consistency. However, the ensemble technique is limited in applicability when models are different in sizes or architectures.

In summary, the existence of model inconsistency is ubuiquitous during model training and deployment. Some attempts have been made to improve model consistency, but are limited to model training phase with the same model architecture. There is no study on the severity of model consistency with the growth of system complexity,

i.e., the growing model size, different model architectures. Also, a general method is required to guarantee consistency between the models in a ML system that needs consistency guarantee.

Model Distillation. knowledge distillation [18, 119] is first proposed to train a shallow network that mimics the behavior of a deep network. The training process leverages knowledge from both the ground true label and the teacher network. Knowledge representation from the teacher model could be from the model output at the final layer [119] or feature maps from middle layers [243], with loss computed by KL divergence loss, or MSE loss. When there is a lack of pre-trained teacher models, however, online knowledge distillation [334] enable simultaneous training of both the teacher and student models at the same time. In Deep Mutual Learning (DML) [334], the model update is implemented via averaged pair-wise distillation loss for the current model against all other models. Co-distillation [107] is similar to DML and is directly applicable to training large-cohort models in a collaborative way. It is also studied that co-distillation in effective in reducing model inconsistency. Many other works extend the training paradigm of online KD by by using ensembles of model outputs and the true label [107], adding diverse peers [55] or multi-branch architecture [161].

We compare DMML with many of the above baselines on online-KD and techniques on improving model consistency. We show that DMML is easy to implement, effective in improving model consistency and generic on different data sources, i.e., vision and language datsets.

5.7 Conclusion

In this paper, we propose DeepCon, an adaptive deployment framework for quickly improving the model consistency via over-the-air parallel training. We design a whole pipeline for quickly detecting the inconsistency within the systems, and propose an efficient learning algorithm (DMML) based on knowledge distillation for improving the consistency between the models. In order to further accelerate the training process, we implement DMML-Par, asynchronous parallel training of DMML, a high-scalable algorithm that is easily adapted to various numbers of computation resources. We

prototype DeepCon and implement a set of APIs for seamless communication between the edge and cloud layers. The evaluation results show the effectiveness of DMML in improving model consistency. We also evaluate the training speed up of DMML-Par, which can guarantee the best consistency improvement while greatly reducing the training time. In the future, we plan to extend the implementation and experience with more real-world applications, e.g., PersonReID, and Object Detection.

6

Conclusion

Contents

Summary

In this chapter, we summarize the research work presented in this thesis. Then, we outline the contributions and propose future research directions for addressing the existing challenges in the current state-of-the-art.

6.1 Thesis Summary

The relentless advancement of computing power has propelled the evolution of AI, from cloud AI to edge AI, and now to the collaborative frontier of edge-cloud AI. This dynamic edge-cloud computing paradigm has ushered in a new era of innovation, significantly augmenting the development of ML-based IoT applications. The synergy between edge and cloud computing fosters a flourishing environment where high accuracy, low latency, and unwavering stability converge. This thesis showcases the implementation of multiple distributed ML-based IoT applications, alongside the strategic design of a comprehensive suite of edge-cloud collaborative optimization algorithms. The core objective of this research is to enhance the quality of service (QoS) metrics of these applications, with a particular focus on optimizing system latency, prediction accuracy, and model consistency after deployment.

Chapter 1 provides an overview of ML-based IoT applications within the broader context of edge-cloud collaborative computing. It also touches upon the deployment and update Lifecycle in Edge-Cloud Computing. Additionally, it addresses challenges and research questions, highlighting the contributions made by the thesis.

Chapter 2 introduces a practical ML-based IoT application set in the context of a smart city. Furthermore, I delve into the fundamental concepts of cloud AI, edge AI, and the innovative domain of edge-cloud collaborative AI. The primary emphasis of this thesis lies in tackling the intricate challenges related to performance optimization after deployment, which we address through the application of cutting-edge edge-cloud collaborative AI techniques.

Chapter 3 presents OsmoticGate, an in-depth exploration of video streaming processing task offloading in the context of edge-cloud computing paradigm. Leveraging

bitrate-based video streaming protocols, I introduce a novel Hierarchy Queue Model to comprehensively capture the dynamics of system workload and its direct impact on system latency and throughput. To optimize system performance, a two-stage gradient-based algorithm is meticulously devised. This algorithm aims to minimize system latency while ensuring the maintenance of a minimal throughput level. Rigorous evaluation and testing of OsmoticGate are carried out, encompassing both simulation scenarios and real-world testbed experiments, to establish and validate its effectiveness and practicality.

Chapter 4 introduces OsmoticGate2, a sophisticated online multi-agent reinforcement learning system crafted to achieve workload balancing in dynamic and distributed deep learning (DL) applications. Powered by the cutting-edge multi-agent reinforcement learning algorithm MAPPO, all agents are actively engaged in real-world environments, continually learning from these interactions. This dynamic learning process enables the system to optimize its performance effectively in a changing environment. Through rigorous experimentation, OsmoticGate2 showcases its exceptional adaptability to varying system configurations while maintaining stable runtime performance. The experimental results serve as compelling evidence of OsmoticGate2's capability to successfully balance workloads and sustain desired performance levels in the context of dynamic and distributed DL applications.

Chapter 5 introduces DEEPCON, a highly adaptable deployment framework aimed at swiftly enhancing model consistency through over-the-air parallel training. The framework encompasses a comprehensive pipeline designed to rapidly detect inconsistencies within the systems. It incorporates an efficient learning algorithm, DMML, specifically tailored to improve model consistency between different models. To further expedite the training process, an innovative high-scalable algorithm, DMMLPar, facilitates asynchronous parallel training of DMML, effectively adapting to varying computation resource availability. The implementation of DEEPCON includes a seamless set of APIs that enable seamless communication between the edge and cloud layers. The evaluation results unequivocally demonstrate the efficacy of DMML in improving model consistency. Furthermore, the training speed-up achieved by DMMLPar is rigorously assessed, showcasing its ability to guarantee the most significant consistency

improvement while substantially reducing training time.

Research Impact: Generally, the work presented in this thesis is generic applicable for improving analytics performance of deep learning based IoT applications. More specifically, the system modelling methodology implemented in OsmoticGate can precisely capture the implicit and complex relationship between system configurations and QoS metrics. Meanwhile, OsmoticGate can greatly reduce system analytics latency via balancing workloads across the edge and the cloud. Then, OsmoticGate2 can be used for generating configurations that jointly optimize latency and accuracy, in a more dynamic environment that comprises more edge devices. Finally, DeepCon ensures system stability by enforcing consistent analytics results among all models within the application.

Based on the research outcomes in this thesis, a prototype system for streaming data analytics is established. The system comprises a data plane that can process various types of streaming data with computation resources across the edge and the cloud. Meanwhile, a control plane is closely integrated within the system that can system dynamics and generate configuration for optimizing various system QoS performance. The prototype, however, is still in its early stage. More enhancement can be made from the following perspectives:

6.2 Future Research Directions

6.2.1 Agile adaptation of decision-making agents in open environment.

To promote deployment of deep learning applications in real-world, some recent research has started to explore coordination of agents in open environments. This usually entails online adaption of the agents when the environment changes. For example, in large-scale distributed environments, the cameras and devices could join in or leave the application anytime. It is thus necessary for the agents to adapt themselves to the new environments and new trends. Meanwhile, exploration and adaptation would inevitably generate configurations that may harm the stability of the underling system. How to enable safe exploration during adaptation is also important in this case.

6.2.2 Improving generalization via adapting large language models for networking.

Many networking tasks now employ deep learning (DL) or reinforcement learning (RL) techniques to solve complex system optimization problems. However, the DL or RL-based algorithms entails intensive engineering overhead for different tasks. Moreover, the deep learning models tend to achieve poor generalization performance across different or unseen environments. In the case of task-offloading in video analytics, the deployed models face challenges in generalizing to complex and dynamic networks, workloads, computation resources. Motivated by the recent success of large language models (LLMs), a promising future research direction is the adaptation of LLM for improving the generalization capability of decision-making agents in network-based video analytics applications.

6.2.3 Precise control on delayed system feedback.

The system feedbacks could be delayed in streaming analytics scenarios. For example, the retrieval of analytics results of all video chunks during a time-interval may be delayed due to the inadequate computation resources during this period. The delayed feedbacks pose a challenge of training task-offloading agents using logs with incomplete information. When being applied to real-world applications, the challenges becomes more severe as the streaming video analytics orchestration agents need to be re-trained frequently and the system logs need to be collected over short time scales. Existing approach simply ignore the unobserved feedbacks, resulting in bias in the training and thus harm the accuracy of the task-offloading agents. A novel approach that addresses delayed feedback during agent training is thus necessary and a promising research direction in improving the decision-making accuracy in streaming applications.

References

[1] Can 30,000 cameras help solve chicago's crime problem? μ`https://www.nytimes.com/2018/05/26/us/ chicago-police-surveillance.html`. Accessed: 2023-10-31.

[2] Dust networks applications: Industrial automation.

[3] Piper: All-in-one wireless security system.

[4] Provigial target tracking and analysis.

[5] Training dataset.

[6] Yolov8. μ`https://github.com/ultralytics/ultralytics`. Accessed: 2024-03-19.

[7] M Abadi, P Barham, J Chen, Z Chen, A Davis, J Dean, M Devin, S Ghemawat, G Irving, M Isard, *et al.* Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on OSDI*, pages 265–283, 2016.

[8] A Abdiansah and R Wardoyo. Time complexity analysis of support vector machines (svm) in libsvm. *International journal computer and application*, 2015.

[9] A. F Aji and K Heafield. Sparse communication for distributed gradient descent. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 440–445, 2017.

[10] Z Akhtar, Y. S Nam, R Govindan, S Rao, J Chen, E Katz-Bassett, B Ribeiro, J Zhan, and H Zhang. Oboe: Auto-tuning video abr algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 44–58, 2018.

[11] D Alistarh, D Grubic, J Li, R Tomioka, and M Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[12] G Ananthanarayanan, Y Shu, L Cox, and V Bahl. Project rocket platform—designed for easy, customizable live video analytics—is open source. *Microsoft Research Blog*, 2020.

[13] R Anil, G Pereyra, A Passos, R Ormandi, G. E Dahl, and G. E Hinton. Large scale distributed neural network training through online distillation. In *International Conference on Learning Representations*, 2018.

[14] F. A Aoudia, M Gautier, and O Berder. Rlman: an energy manager based on reinforcement learning for energy harvesting wireless sensor networks. *IEEE Transactions on Green Communications and Networking*, 2(2):408–417, 2018.

[15] J Appleyard, T Kocisky, and P Blunsom. Optimizing performance of recurrent neural networks on gpus. *arXiv preprint arXiv:1604.01946*, 2016.

[16] K Arulkumaran, M. P Deisenroth, M Brundage, and A. A Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

[17] A Asheralieva and D Niyato. Game theory and lyapunov optimization for cloud-based content delivery networks with device-to-device and uav-enabled caching. *IEEE Transactions on Vehicular Technology*, 68(10):10094–10110, 2019.

[18] J Ba and R Caruana. Do deep nets really need to be deep? *Advances in Neural Information Processing Systems*, 27, 2014.

[19] J Ba, R Grosse, and J Martens. Distributed second-order optimization using kronecker-factored approximations. 2016.

[20] D Bahri and H Jiang. Locally adaptive label smoothing for predictive churn. *arXiv preprint arXiv:2102.05140*, 2021.

[21] B Baker, O Gupta, R Raskar, and N Naik. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*, 2017.

[22] A. K Balan, V Rathod, K. P Murphy, and M Welling. Bayesian dark knowledge. In *Advances in Neural Information Processing Systems*, pages 3438–3446, 2015.

[23] P Baldi. Autoencoders, unsupervised learning, and deep architectures. In *Proceedings of ICML workshop on unsupervised and transfer learning*, pages 37–49, 2012.

[24] I Baldini, P Castro, K Chang, P Cheng, S Fink, V Ishakian, N Mitchell, V Muthusamy, R Rabbah, A Slominski, *et al.* Serverless computing: Current trends and open problems. *Research advances in cloud computing*, pages 1–20, 2017.

[25] D Barcelona-Pons, M Sánchez-Artigas, G París, P Sutra, and P García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th international middleware conference*, pages 41–54, 2019.

[26] A Beck, A Nedić, A Ozdaglar, and M Teboulle. An $o(1/k)$ gradient method for network resource allocation problems. *IEEE Transactions on Control of Network Systems*, 1(1):64–73, 2014.

[27] T Ben-Nun, E Levy, A Barak, and E Rubin. Memory access patterns: the missing piece of the multi-gpu puzzle. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[28] J. L Berral, Í Goiri, R Nou, F Julià, J Guitart, R Gavaldà, and J Torres. Towards energy-aware scheduling in data centers using machine learning. In *Proceedings of the 1st International Conference on energy-Efficient Computing and Networking*, pages 215–224. ACM, 2010.

[29] R Bhardwaj, Z Xia, G Ananthanarayanan, J Jiang, Y Shu, N Karianakis, K Hsieh, P Bahl, and I Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 119–135, 2022.

[30] S Bhojanapalli, K Wilber, A Veit, A. S Rawat, S Kim, A Menon, and S Kumar. On the reproducibility of neural network predictions. *arXiv preprint arXiv:2102.03349*, 2021.

[31] M Boehm, M. W Dusenberry, D Eriksson, A. V Evfimievski, F. M Manshadi, N Pansare, B Reinwald, F. R Reiss, P Sen, A. C Surve, *et al.* Systemml: Declarative machine learning on spark. *Proceedings of the VLDB Endowment*, 9(13):1425–1436, 2016.

[32] K Bonawitz, H Eichner, W Grieskamp, D Huba, A Ingerman, V Ivanov, C Kiddon, J Konecny, S Mazzocchi, H. B McMahan, *et al.* Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.

[33] K Bonawitz, F Salehi, J Konečnỳ, B McMahan, and M Gruteser. Federated learning with autotuned communication-efficient secure aggregation. *arXiv preprint arXiv:1912.00131*, 2019.

[34] L Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[35] L Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.

[36] L Bottou, F. E Curtis, and J Nocedal. Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311, 2018.

[37] O Bousquet, S Boucheron, and G Lugosi. Introduction to statistical learning theory. In *Summer School on Machine Learning*, pages 169–207. Springer, 2003.

[38] S Boyd and L Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[39] J. K Bradley, A Kyrola, D Bickson, and C Guestrin. Parallel coordinate descent for l1-regularized loss minimization. *arXiv preprint arXiv:1105.5379*, 2011.

[40] L Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[41] U Breitenbucher, K Kepes, F Leymann, and M Wurster. Declarative vs. imperative: How to model the automated deployment of iot applications? *Proceedings of the 11th Advanced Summer School on Service Oriented Computing*, pages 18–27, 2017.

[42] A Brock, T Lim, J. M Ritchie, and N Weston. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017.

[43] C Buciluǎ, R Caruana, and A Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006.

[44] J. V Burke, F. E Curtis, A. S Lewis, M. L Overton, and L. E Simões. Gradient sampling methods for nonsmooth optimization. In *Numerical Nonsmooth Optimization*, pages 201–225. Springer, 2020.

[45] J. V Burke, A. S Lewis, and M. L Overton. A robust gradient sampling algorithm for nonsmooth, nonconvex optimization. *SIAM Journal on Optimization*, 15(3):751–779, 2005.

[46] R. H Byrd, S. L Hansen, J Nocedal, and Y Singer. A stochastic quasi-newton method for large-scale optimization. *SIAM Journal on Optimization*, 26(2):1008–1031, 2016.

[47] B Cabé, E. I. W Group, *et al.* Iot developer survey 2018. *SlideShare, April*, 13, 2018.

[48] C Canel, T Kim, G Zhou, C Li, H Lim, D. G Andersen, M Kaminsky, and S Dulloor. Scaling video analytics on constrained edge nodes. *Proceedings of Machine Learning and Systems*, 1:406–417, 2019.

[49] P Castro, V Ishakian, V Muthusamy, and A Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, 2019.

[50] R Chandra. Farmbeats: Empowering farmers with affordable digital agriculture solutions. In *ASA, CSSA and SSSA International Annual Meetings (2019)*. ASA-CSSA-SSSA, 2019.

[51] Z Chang, L Lei, Z Zhou, S Mao, and T Ristaniemi. Learn to cache: Machine learning for network edge caching in the big data era. *IEEE Wireless Communications*, 25(3):28–35, 2018.

[52] K Chellapilla, S Puri, and P Simard. High performance convolutional neural networks for document processing. In *10th International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[53] C. L. P Chen and B Wang. Random-positioned license plate recognition using hybrid broad learning system and convolutional networks. *IEEE Transactions on Intelligent Transportation Systems*, 23(1):444–456, 2022.

[54] C.-C Chen, C.-L Yang, and H.-Y Cheng. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv preprint arXiv:1809.02839*, 2018.

[55] D Chen, J.-P Mei, C Wang, Y Feng, and C Chen. Online knowledge distillation with diverse peers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3430–3437, 2020.

[56] J Chen, X Pan, R Monga, S Bengio, and R Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[57] L Chen, J Lu, Z Song, and J Zhou. Part-activated deep reinforcement learning for action prediction. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 421–436, 2018.

[58] M Chen, L Wang, J Chen, X Wei, and L Lei. A computing and content delivery network in the smart city: Scenario, framework, and analysis. *IEEE Network*, 33(2):89–95, 2019.

[59] T. Y.-H Chen, L Ravindranath, S Deng, P Bahl, and H Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM conference on embedded networked sensor systems*, pages 155–168, 2015.

[60] Y Chen, S Liu, X Zhang, K Liu, and J Zhao. Automatically labeled data generation for large scale event extraction. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 409–419, 2017.

[61] Y Cheng, D Wang, P Zhou, and T Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

[62] S Chetlur, C Woolley, P Vandermersch, J Cohen, J Tran, B Catanzaro, and E Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[63] T Chilimbi, Y Suzue, J Apacible, and K Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on OSDI*, pages 571–582, 2014.

[64] F Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[65] P Chopra and S. K Yadav. Fault detection and classification by unsupervised feature extraction and dimensionality reduction. *Complex & Intelligent Systems*, 2015.

[66] C.-T Chu, S. K Kim, Y.-A Lin, Y Yu, G Bradski, K Olukotun, and A. Y Ng. Map-reduce for machine learning on multicore. In *Advances in neural information processing systems*, pages 281–288, 2007.

[67] E Cidon, E Pergament, Z Asgar, A Cidon, and S Katti. Characterizing and taming model instability across edge devices. *Proceedings of Machine Learning and Systems*, 3, 2021.

[68] T Cohen and M Welling. Group equivariant convolutional networks. In *International conference on machine learning*, pages 2990–2999, 2016.

[69] I Colin, L Dos Santos, and K Scaman. Theoretical limits of pipeline parallel optimization and application to distributed deep learning. In *Advances in Neural Information Processing Systems*, pages 12350–12359, 2019.

[70] N Corporation. Nvidia collective communications library (nccl), 2015.

[71] C Cortes and V Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[72] M Courbariaux, Y Bengio, and J.-P David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in NIPS*, 2015.

[73] M Courbariaux, I Hubara, D Soudry, R El-Yaniv, and Y Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv*, 2016.

[74] T. M Cover, P Hart, *et al.* Nearest neighbor pattern classification. 1967.

[75] H Cui, H Zhang, G. R Ganger, P. B Gibbons, and E. P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[76] A. C. F da Silva, U Breitenbücher, P Hirmer, K Képes, O Kopp, F Leymann, B Mitschang, and R Steinke. Internet of things out of the box: Using tosca for automating the deployment of iot environments. In *CLOSER*, pages 330–339, 2017.

[77] L De Lauretis. From monolithic architecture to microservices architecture. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 93–96. IEEE, 2019.

[78] C. M De Sa, C Zhang, K Olukotun, and C Ré. Taming the wild: A unified analysis of hogwild-style algorithms. In *Advances in neural information processing systems*, pages 2674–2682, 2015.

[79] J Dean, G Corrado, R Monga, K Chen, M Devin, M Mao, A Senior, P Tucker, K Yang, Q. V Le, *et al.* Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[80] O Dekel, R Gilad-Bachrach, O Shamir, and L Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13(Jan):165–202, 2012.

[81] J Deng, W Dong, R Socher, L.-J Li, K Li, and L Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[82] S Deng, Z Xiang, J Taheri, M. A Khoshkholghi, J Yin, A. Y Zomaya, and S Dustdar. Optimal application deployment in resource constrained distributed edges. *IEEE transactions on mobile computing*, 20(5):1907–1923, 2020.

[83] S Devi and T Neetha. Machine learning based traffic congestion prediction in a iot based smart city. 2017.

[84] G Diamos, S Sengupta, B Catanzaro, M Chrzanowski, A Coates, E Elsen, J Engel, A Hannun, and S Satheesh. Persistent rnns: Stashing recurrent weights on-chip. In *International Conference on Machine Learning*, pages 2024–2033, 2016.

[85] J.-D Dong, A.-C Cheng, D.-C Juan, W Wei, and M Sun. Dpp-net: Device-aware progressive search for pareto-optimal neural architectures. In *Proceedings of the ECCV*, pages 517–531, 2018.

[86] D. L Donoho. For most large underdetermined systems of linear equations the minimal 1-norm solution is also the sparsest solution. *Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences*, 59(6):797–829, 2006.

[87] D Driggs, J Tang, J Liang, M Davies, and C.-B Schönlieb. Spring: A fast stochastic proximal alternating method for non-smooth non-convex optimization. *arXiv preprint arXiv:2002.12266*, 2020.

[88] N Dryden, N Maruyama, T Benson, T Moon, M Snir, and B Van Essen. Improving strong-scaling of cnn training by exploiting finer-grained parallelism. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 210–220. IEEE, 2019.

[89] N Dryden, N Maruyama, T Moon, T Benson, M Snir, and B Van Essen. Channel and filter parallelism for large-scale cnn training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–20, 2019.

[90] N Dryden, T Moon, S. A Jacobs, and B Van Essen. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pages 1–8. IEEE, 2016.

[91] K Du, A Pervaiz, X Yuan, A Chowdhery, Q Zhang, H Hoffmann, and J Jiang. Server-driven video streaming for deep learning inference. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 557–570, 2020.

[92] S Du, M Ibrahim, M Shehata, and W Badawy. Automatic license plate recognition (alpr): A state-of-the-art review. *IEEE Transactions on circuits and systems for video technology*, 23(2):311–325, 2012.

[93] J Duchi, E Hazan, and Y Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[94] A Elk. Distributed machine learning toolkit: Big data, big model, flexibility, efficiency, 2019.

[95] T Elsken, J. H Metzen, and F Hutter. Multi-objective architecture search for cnns. *arXiv preprint arXiv:1804.09081*, 2, 2018.

[96] M Everingham, L Van Gool, C. K. I Williams, J Winn, and A Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[97] G Frewat, C Baroud, R Sammour, A Kassem, and M Hamad. Android voice recognition application with multi speaker feature. In *2016 18th MELECON*, pages 1–5. IEEE, 2016.

[98] N Friedman, D Geiger, and M Goldszmidt. Bayesian network classifiers. *Machine learning*, 29(2-3):131–163, 1997.

[99] A. L Gaunt, M. A Johnson, M Riechert, D Tarlow, R Tomioka, D Vytiniotis, and S Webster. Ampnet: Asynchronous model-parallel training for dynamic neural networks. *arXiv preprint arXiv:1705.09786*, 2017.

[100] A Gholami, A Azad, P Jin, K Keutzer, and A Buluc. Integrated model, batch, and domain parallelism in training neural networks. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 77–86, 2018.

[101] Y Gong, Z Jiang, Y Feng, B Hu, K Zhao, Q Liu, and W Ou. Edgerec: recommender system on edge in mobile taobao. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 2477–2484, 2020.

[102] I Goodfellow, J Pouget-Abadie, M Mirza, B Xu, D Warde-Farley, S Ozair, A Courville, and Y Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[103] P Goyal, P Dollár, R Girshick, P Noordhuis, L Wesolowski, A Kyrola, A Tulloch, Y Jia, and K He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[104] A Graves, A rahman Mohamed, and G. E Hinton. Speech recognition with deep recurrent neural networks. *ICASSP*, pages 6645–6649, 2013.

[105] L Guan, W Yin, D Li, and X Lu. Xpipe: Efficient pipeline model parallelism for multi-gpu dnn training. *arXiv preprint arXiv:1911.04610*, 2019.

[106] Y Guan and T Plötz. Ensembles of deep lstm learners for activity recognition using wearables. *IMWUT*, 1(2):11, 2017.

[107] Q Guo, X Wang, Y Wu, Z Yu, D Liang, X Hu, and P Luo. Online knowledge distillation via collaborative learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11020–11029, 2020.

[108] S Gupta, A Agrawal, K Gopalakrishnan, and P Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

[109] H HaddadPajouh, A Dehghantanha, R Khayami, and K.-K. R Choo. A deep recurrent neural network based approach for internet of things malware threat hunting. *Future Generation Computer Systems*, 85:88–96, 2018.

[110] S Han, H Shen, M Philipose, S Agarwal, A Wolman, and A Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 123–136, 2016.

[111] P Hansen, N Mladenović, and J. A. M Pérez. Variable neighbourhood search: methods and applications. *Annals of Operations Research*, 175(1):367–407, 2010.

[112] A Harlap, D Narayanan, A Phanishayee, V Seshadri, N Devanur, G Ganger, and P Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

[113] J. A Hartigan and M. A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1), 1979.

[114] K He, G Gkioxari, P Dollar, and R. B Girshick. Mask r-cnn. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2980–2988, 2017.

[115] K He, X Zhang, S Ren, and J Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[116] X He, D Mudigere, M Smelyanskiy, and M Takác. Distributed hessian-free optimization for deep neural network. In *Workshops at the Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[117] C Hidey, F Liu, and R Goel. Reducing model jitter: Stable re-training of semantic parsers in production environments. *arXiv preprint arXiv:2204.04735*, 2022.

[118] G Hinton. Rmsprop, 2019.

[119] G Hinton, O Vinyals, and J Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[120] Q Ho, J Cipar, H Cui, S Lee, J. K Kim, P. B Gibbons, G. A Gibson, G Ganger, and E. P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

[121] S Hochreiter and J Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[122] S. C Hoi, D Sahoo, J Lu, and P Zhao. Online learning: A comprehensive survey. *arXiv preprint arXiv:1802.02871*, 2018.

[123] L Hou, R Zhang, and J. T Kwok. Analysis of quantized models. In *International Conference on Learning Representations*, 2019.

[124] A. G Howard, M Zhu, B Chen, D Kalenichenko, W Wang, T Weyand, M Andreetto, and H Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[125] K Hsieh, A Harlap, N Vijaykumar, D Konomis, G. R Ganger, P. B Gibbons, and O Mutlu. Gaia: Geo-distributed machine learning approaching {LAN} speeds. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 629–647, 2017.

[126] C.-H Hsu, S.-H Chang, J.-H Liang, H.-P Chou, C.-H Liu, S.-C Chang, J.-Y Pan, Y.-T Chen, W Wei, and D.-C Juan. Monas: Multi-objective neural architecture search using reinforcement learning. *arXiv preprint arXiv:1806.10332*, 2018.

[127] C Hu and B Li. Distributed inference with deep learning models across heterogeneous edge devices. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 330–339. IEEE, 2022.

[128] G Huang, S Liu, L Van der Maaten, and K. Q Weinberger. Condensenet: An efficient densenet using learned group convolutions. In *Proceedings of the IEEE Conference on CVPR*, pages 2752–2761, 2018.

[129] T.-Y Huang, R Johari, N McKeown, M Trunnell, and M Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 187–198, 2014.

[130] Y Huang, Y Cheng, A Bapna, O Firat, D Chen, M Chen, H Lee, J Ngiam, Q. V Le, Y Wu, *et al.* Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.

[131] I Hubara, M Courbariaux, D Soudry, R El-Yaniv, and Y Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[132] C.-C Hung, G Ananthanarayanan, P Bodik, L Golubchik, M Yu, P Bahl, and M Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 115–131. IEEE, 2018.

[133] Z Huo, B Gu, Q Yang, and H Huang. Decoupled parallel backpropagation with convergence guarantee. *arXiv preprint arXiv:1804.10574*, 2018.

[134] J Ivanecky and S Mehlhase. An in-car speech recognition system for disabled drivers. In *International Conference on Text, Speech and Dialogue*, pages 505–512. Springer, 2012.

[135] S Jain, X Zhang, Y Zhou, G Ananthanarayanan, J Jiang, Y Shu, P Bahl, and J Gonzalez. Spatula: Efficient cross-camera video analytics on large camera networks. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 110–124. IEEE, 2020.

[136] S. Y Jang, B Kostadinov, and D Lee. Microservice-based edge device architecture for video analytics. In *SEC*, pages 165–177, 2021.

[137] S Jeaugey. Nccl 2.0, 2017.

[138] Z Jia, M Zaharia, and A Aiken. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

[139] A. H Jiang, D. L.-K Wong, C Canel, L Tang, I Misra, M Kaminsky, M. A Kozuch, P Pillai, D. G Andersen, and G. R Ganger. Mainstream: Dynamic {Stem-Sharing} for {Multi-Tenant} video processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 29–42, 2018.

[140] H Jiang, H Narasimhan, D Bahri, A Cotter, and A Rostamizadeh. Churn reduction via distillation. *arXiv preprint arXiv:2106.02654*, 2021.

[141] J Jiang, Z Luo, C Hu, Z He, Z Wang, S Xia, and C Wu. Joint model and data adaptation for cloud inference serving. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 279–289, 2021.

[142] J Jiang, G Ananthanarayanan, P Bodik, S Sen, and I Stoica. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*, pages 253–266, 2018.

[143] J Jiang, V Sekar, H Milner, D Shepherd, I Stoica, and H Zhang. {CFA}: A practical prediction system for video qoe optimization. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 137–150, 2016.

[144] S Jiang, Z Lin, Y Li, Y Shu, and Y Liu. Flexible high-resolution object detection on edge devices with tunable latency. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 559–572, 2021.

[145] Y Jiang, V Nagarajan, C Baek, and J. Z Kolter. Assessing generalization of sgd via disagreement. In *ICLR*, 2021.

[146] P Jin, B Ginsburg, and K Keutzer. Spatially parallel convolutions. 2018.

[147] P Kairouz, H. B McMahan, B Avent, A Bellet, M Bennis, A. N Bhagoji, K Bonawitz, Z Charles, G Cormode, R Cummings, *et al.* Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*, 2019.

[148] R Kemker, M McClure, A Abitino, T. L Hayes, and C Kanan. Measuring catastrophic forgetting in neural networks. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[149] M Khani, G Ananthanarayanan, K Hsieh, J Jiang, R Netravali, Y Shu, M Alizadeh, and V Bahl. {RECL}: Responsive {Resource-Efficient} continuous learning for video analytics. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 917–932, 2023.

[150] J. K Kim, Q Ho, S Lee, X Zheng, W Dai, G. A Gibson, and E. P Xing. Strads: a distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[151] D. P Kingma and J Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations (ICLR)*, 2015.

[152] B. R Kiran, I Sobh, V Talpaert, P Mannion, A. A Al Sallab, S Yogamani, and P Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(6):4909–4926, 2021.

[153] J Kohler, L Adolphs, and A Lucchi. Adaptive norms for deep learning with regularized newton methods. *NeurIPS 2019 Workshop: Beyond First-Order Optimization Methods in Machine Learning*, pages arXiv–1905, 2019.

[154] V. R Konda and J. N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[155] J Konečnỳ, H. B McMahan, D Ramage, and P Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *arXiv preprint arXiv:1610.02527*, 2016.

[156] W Kong, Z. Y Dong, Y Jia, D. J Hill, Y Xu, and Y Zhang. Short-term residential load forecasting based on lstm recurrent neural network. *IEEE Transactions on Smart Grid*, 10(1):841–851, 2017.

[157] A Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[158] A Krizhevsky, V Nair, and G Hinton. Cifar-10 cifar-100 (canadian institute for advanced research).

[159] T.-Y Ku, J. D Shin, Y.-S Chung, and H Choi. Hybrid cache architecture using big data analysis for content delivery network. In *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*, pages 273–274. IEEE, 2014.

[160] P Ladosz, L Weng, M Kim, and H Oh. Exploration in deep reinforcement learning: A survey. *Information Fusion*, 85:1–22, 2022.

[161] X Lan, X Zhu, and S Gong. Knowledge distillation by on-the-fly native ensemble. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 7528–7538, 2018.

[162] V Lebedev, Y Ganin, M Rakhuba, I Oseledets, and V Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

[163] V Lebedev and V Lempitsky. Fast convnets using group-wise brain damage. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[164] S Lee, J. K Kim, X Zheng, Q Ho, G. A Gibson, and E. P Xing. On model parallelization and scheduling strategies for distributed machine learning. In *Advances in neural information processing systems*, pages 2834–2842, 2014.

[165] A. S Lewis and M. L Overton. Nonsmooth optimization via quasi-newton methods. *(Mathematical Programming) 2013*, 141(1):135–163, 2013.

[166] H Li, W Ouyang, and X Wang. Multi-bias non-linear activation in deep neural networks. In *International conference on machine learning*, pages 221–229, 2016.

[167] J Li, W Monroe, A Ritter, M Galley, J Gao, and D Jurafsky. Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541*, 2016.

[168] L Li, Y Lv, and F.-Y Wang. Traffic signal timing via deep reinforcement learning. *IEEE/CAA Journal of Automatica Sinica*, 3(3):247–254, 2016.

[169] M Li, D. G Andersen, J. W Park, A. J Smola, A Ahmed, V Josifovski, J Long, E. J Shekita, and B.-Y Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on OSDI*, pages 583–598, 2014.

[170] M Li, D. G Andersen, A. J Smola, and K Yu. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*, 2014.

[171] T Li, A. K Sahu, A Talwalkar, and V Smith. Federated learning: Challenges, methods, and future directions. *IEEE signal processing magazine*, 37(3):50–60, 2020.

[172] X Li and Y Guo. Adaptive active learning for image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 859–866, 2013.

[173] Y Li, A Padmanabhan, P Zhao, Y Wang, *et al.* Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, page 359–376, 2020.

[174] Y Li, A Padmanabhan, P Zhao, Y Wang, G. H Xu, and R Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 359–376, 2020.

[175] Z Li, X Zhu, J Gahm, R Pan, H Hu, A. C Begen, and D Oran. Probe and adapt: Rate adaptation for http video streaming at scale. *IEEE Journal on Selected Areas in Communications*, 32(4):719–733, 2014.

[176] Z Li, Y Shu, G Ananthanarayanan, L Shangguan, K Jamieson, and P Bahl. Spider: A multi-hop millimeter-wave network for live video analytics. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 178–191. IEEE, 2021.

[177] T Lin, S. U Stich, K. K Patel, and M Jaggi. Don't use large mini-batches, use local sgd. *arXiv preprint arXiv:1808.07217*, 2018.

[178] Y Lin, S Han, H Mao, Y Wang, and W. J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[179] N Ling, K Wang, Y He, G Xing, and D Xie. Rt-mdl: Supporting real-time mixed deep learning tasks on edge platforms. In *Proceedings of the 19th ACM conference on embedded networked sensor systems*, pages 1–14, 2021.

[180] C Liu, B Zoph, M Neumann, J Shlens, W Hua, L.-J Li, L Fei-Fei, A Yuille, J Huang, and K Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018.

[181] H Liu, K Simonyan, O Vinyals, C Fernando, and K Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017.

[182] L Liu, M Zhang, Y Lin, and L Qin. A survey on workflow management and scheduling in cloud computing. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 837–846. IEEE, 2014.

[183] W Lloyd, S Ramesh, S Chinthalapati, L Ly, and S Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE international conference on cloud engineering (IC2E)*, pages 159–169. IEEE, 2018.

[184] F Loewenherz, V Bahl, and Y Wang. Video analytics towards vision zero. *Institute of Transportation Engineers. ITE Journal*, 87(3):25, 2017.

[185] Q Lu, T Peng, W Wang, W Wang, and C Hu. Utility-based resource allocation in uplink of ofdma-based cognitive radio networks. *International Journal of Communication Systems*, 23(2):252–274, 2010.

[186] Z Lu, K Chan, S Pu, and T La Porta. Crowdvision: A computing platform for video crowdprocessing using deep learning. *IEEE Transactions on Mobile Computing*, 18(7):1513–1526, 2018.

[187] M.-T Luong, H Pham, and C. D Manning. Effective approaches to attention-based neural machine translation. *EMNLP*, 2015.

[188] N Ma, X Zhang, H.-T Zheng, and J Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

[189] A Maas, R. E Daly, P. T Pham, D Huang, A. Y Ng, and C Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, pages 142–150, 2011.

[190] R Madan, J Borran, A Sampath, N Bhushan, A Khandekar, and T Ji. Cell association and interference coordination in heterogeneous lte-a cellular networks. *IEEE Journal on selected areas in communications*, 28(9):1479–1489, 2010.

[191] R Madan, S. P Boyd, and S Lall. Fast algorithms for resource allocation in wireless cellular networks. *IEEE/ACM Transactions on Networking (TON)*, 18(3):973–984, 2010.

[192] H Mao, M Alizadeh, I Menache, and S Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, 2016.

[193] H Mao, R Netravali, and M Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210, 2017.

[194] W.-L Mao, W.-C Chen, C.-T Wang, and Y.-H Lin. Recycling waste classification using optimized convolutional neural network. *Resources, Conservation and Recycling*, 164:105132, 2021.

[195] M Mathieu, M Henaff, and Y LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

[196] W McColl. Bulk synchronous parallel computing. *Abstract Machine Models for Highly Parallel Computers, Oxford University Press, Oxford*, 1995.

[197] R McDonald, K Hall, and G Mann. Distributed training strategies for the structured perceptron. In *Human language technologies: The 2010 annual conference of the North American chapter of the association for computational linguistics*, pages 456–464. Association for Computational Linguistics, 2010.

[198] H. B McMahan, E Moore, D Ramage, S Hampson, *et al.* Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629*, 2016.

[199] H. B McMahan, D Ramage, K Talwar, and L Zhang. Learning differentially private recurrent language models. *ICLR 2018*, 2017.

[200] N Mehrabi, F Morstatter, N Saxena, K Lerman, and A Galstyan. A survey on bias and fairness in machine learning. *ACM computing surveys (CSUR)*, 54(6):1–35, 2021.

[201] X Meng, J Bradley, B Yavuz, E Sparks, S Venkataraman, D Liu, J Freeman, D Tsai, M Amde, S Owen, *et al.* Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[202] A Mirhoseini, A Goldie, H Pham, B Steiner, Q. V Le, and J Dean. A hierarchical model for device placement. 2018.

[203] A Mirhoseini, H Pham, Q. V Le, B Steiner, R Larsen, Y Zhou, N Kumar, M Norouzi, S Bengio, and J Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2430–2439. JMLR. org, 2017.

[204] G Montavon, G Orr, and K.-R Müller. *Neural networks: tricks of the trade*, volume 7700. springer, 2012.

[205] P Moritz, R Nishihara, and M Jordan. A linearly-convergent stochastic l-bfgs algorithm. In *Artificial Intelligence and Statistics*, pages 249–258, 2016.

[206] M. S Munir, S. F Abedin, M. G. R Alam, D. H Kim, and C. S Hong. Rnn based energy demand prediction for smart-home in smart-grid framework. 2017.

[207] A Murad, F. A Kraemer, K Bach, and G Taylor. Autonomous management of energy-harvesting iot nodes using deep reinforcement learning. *arXiv:1905.04181*, 2019.

[208] P Nakkiran and Y Bansal. Distributional generalization: A new kind of generalization. *arXiv preprint arXiv:2009.08092*, 2020.

[209] P Nakkiran, G Kaplun, Y Bansal, T Yang, B Barak, and I Sutskever. Deep double descent: Where bigger models and more data hurt. *arXiv preprint arXiv:1912.02292*, 2019.

[210] M. A Namjoshi and P. A Kulkarni. Novel online profiling for virtual machines. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 133–144, 2010.

[211] M Naphade, V Kolar, and S Biswas. Multi-camera large-scale intelligent video analytics with deepstream sdk. *Accessed: March*, 9, 2023.

[212] P Naraei, A Abhari, and A Sadeghian. Application of multilayer perceptron neural networks and support vector machines in classification of healthcare data. In *FTC*, pages 848–852, Dec 2016.

[213] Y Nesterov. Gradient methods for minimizing composite functions. *Mathematical Programming*, 140(1):125–161, 2013.

[214] C Nguyen, A Mehta, C Klein, and E Elmroth. Why cloud applications are not ready for the edge (yet). In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 250–263, 2019.

[215] J Ni and D. H Tsang. Large-scale cooperative caching and application-level multicast in multimedia content delivery networks. *IEEE Communications Magazine*, 43(5):98–105, 2005.

[216] Nvidia. *Jetson Nano: Deep Learning Inference Benchmarks*, 2021 (accessed 19, 1, 2021).

[217] D Oh and I Yun. Residual error based anomaly detection using auto-encoder in smd machine sound. *Sensors*, 18(5):1308, 2018.

[218] K Osawa, Y Tsuji, Y Ueno, A Naruse, R Yokota, and S Matsuoka. Second-order optimization method for large mini-batch: Training resnet-50 on imagenet in 35 epochs. *arXiv preprint arXiv:1811.12019*, 2018.

[219] A Padmanabhan, N Agarwal, A Iyer, G Ananthanarayanan, Y Shu, N Karianakis, G. H Xu, and R Netravali. Gemel: Model merging for {Memory-Efficient},{Real-Time} video analytics at the edge. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 973–994, 2023.

[220] X Pan, M Lam, S Tu, D Papailiopoulos, C Zhang, M. I Jordan, K Ramchandran, and C Re. Cyclades: Conflict-free asynchronous machine learning. In *Advances in Neural Information Processing Systems*, pages 2568–2576, 2016.

[221] G. I Parisi, R Kemker, J. L Part, C Kanan, and S Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 2019.

[222] R Parr and S. J Russell. Reinforcement learning with hierarchies of machines. In *NIPS*, 1998.

[223] I Pelle, J Czentye, J Dóka, A Kern, B. P Gerő, and B Sonkoly. Operating latency sensitive applications on public serverless edge cloud platforms. *IEEE Internet of Things Journal*, 8(10):7954–7972, 2020.

[224] A Petrowski, G Dreyfus, and C Girault. Performance analysis of a pipelined backpropagation parallel algorithm. *IEEE Transactions on Neural Networks*, 4(6):970–981, 1993.

[225] H Pham, M. Y Guan, B Zoph, Q. V Le, and J Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[226] R Poddar, G Ananthanarayanan, S Setty, S Volos, and R. A Popa. Visor:{Privacy-Preserving} video analytics as a cloud service. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1039–1056, 2020.

[227] F Ponce, G Márquez, and H Astudillo. Migrating from monolithic architecture to microservices: A rapid review. In *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–7. IEEE, 2019.

[228] D Povey, X zhang, and S Khudanpur. Parallel training of dnns with natural gradient and parameter averaging. *ICLR 2015* -, Aug 2017.

[229] B Qian, J Su, Z Wen, D. N Jha, Y Li, Y Guan, D Puthal, P James, R Yang, A. Y Zomaya, *et al.* Orchestrating the development lifecycle of machine learning-based iot applications: A taxonomy and survey. *ACM Computing Surveys (CSUR)*, 53(4):1–47, 2020.

[230] B Qian, Z Wen, J Tang, Y Yuan, A. Y Zomaya, and R Ranjan. Osmoticgate: Adaptive edge-based real-time video analytics for the internet of things. *IEEE Transactions on Computers*, 72(4):1178–1193, 2022.

[231] N Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[232] D Quillen, E Jang, O Nachum, C Finn, J Ibarz, and S Levine. Deep reinforcement learning for vision-based robotic grasping: A simulated comparative evaluation of off-policy methods. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6284–6291. IEEE, 2018.

[233] J. R Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[234] R Raina, A Madhavan, and A. Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual ICML*. ACM, 2009.

[235] N Ramakrishnan and T Soni. Network traffic prediction using recurrent neural networks. In *2018 17th IEEE ICMLA*. IEEE, 2018.

[236] X Ran, H Chen, X Zhu, Z Liu, and J Chen. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 1421–1429. IEEE, 2018.

[237] M Rastegari, V Ordonez, J Redmon, and A Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 2016.

[238] E Real, A Aggarwal, Y Huang, and Q. V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.

[239] B Recht, C Re, S Wright, and F Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[240] J Redmon and A Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[241] L Ren, X Yuan, J Lu, M Yang, and J Zhou. Deep reinforcement learning with iterative shift for visual tracking. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 684–700, 2018.

[242] S Ren, K He, R Girshick, and J Sun. Faster r-cnn: towards real-time object detection with region proposal networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems-Volume 1*, pages 91–99, 2015.

[243] A Romero, N Ballas, S. E Kahou, A Chassang, C Gatta, and Y Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

[244] F Ruelens, B. J Claessens, S Vandael, B De Schutter, R Babuška, and R Belmans. Residential demand response of thermostatically controlled loads using batch reinforcement learning. *IEEE Transactions on Smart Grid*, 8(5):2149–2159, 2016.

[245] S Russell and P Norvig. Artificial intelligence: a modern approach. 2002.

[246] A Sadeghi, G Wang, and G. B Giannakis. Deep reinforcement learning for adaptive caching in hierarchical content delivery networks. *IEEE Transactions on Cognitive Communications and Networking*, 5(4):1024–1033, 2019.

[247] M Salehe, Z Hu, S. H Mortazavi, I Mohomed, and T Capes. Videopipe: Building video stream processing pipelines at the edge. In *Proceedings of the 20th International Middleware Conference Industrial Track*, pages 43–49, 2019.

[248] M Sandler, A Howard, M Zhu, A Zhmoginov, and L.-C Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[249] J Schulman, F Wolski, P Dhariwal, A Radford, and O Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[250] F Seide, H Fu, J Droppo, G Li, and D Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[251] A Sergeev and M Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[252] G. I Shamir, D Lin, and L Coviello. Smooth activations and reproducibility in deep networks. *arXiv preprint arXiv:2010.09931*, 2020.

[253] W Shang, K Sohn, D Almeida, and H Lee. Understanding and improving convolutional neural networks via concatenated rectified linear units. In *ICML*, pages 2217–2225, 2016.

[254] N Shazeer, Y Cheng, N Parmar, D Tran, A Vaswani, P Koanantakool, P Hawkins, H Lee, M Hong, C Young, R Sepassi, and B Hechtman. Mesh-TensorFlow: Deep learning for supercomputers. In *Neural Information Processing Systems*, 2018.

[255] J Shlens. A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100*, 2014.

[256] A Shustanov and P Yakimov. Cnn design for real-time traffic sign recognition. *Procedia engineering*, 201:718–725, 2017.

[257] L Sifre and S Mallat. Rigid-motion scattering for image classification. *Ph. D. dissertation*, 2014.

[258] S. H Silva and P Najafirad. Opportunities and challenges in deep learning adversarial robustness: A survey. *arXiv preprint arXiv:2007.00753*, 2020.

[259] K Simonyan and A Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[260] K. J Singh and D. S Kapoor. Create your own internet of things: A survey of iot platforms. *IEEE Consumer Electronics Magazine*, 6(2):57–68, 2017.

[261] S. P Singh. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the National Conference on Artificial Intelligence*. JOHN WILEY & SONS LTD, 1992.

[262] S Singh and I Chana. A survey on resource scheduling in cloud computing: Issues and challenges. *Journal of grid computing*, 14(2):217–264, 2016.

[263] S. L Smith, P.-J Kindermans, C Ying, and Q. V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[264] R. R Snapp and G. I Shamir. Synthesizing irreproducibility in deep networks. *arXiv preprint arXiv:2102.10696*, 2021.

[265] E. R Sparks, A Talwalkar, D Haas, M. J Franklin, M. I Jordan, and T Kraska. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 368–380. ACM, 2015.

[266] K Spiteri, R Urgaonkar, and R. K Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking*, 28(4):1698–1711, 2020.

[267] S. U Stich, C. L Muller, and B Gartner. Optimization of convex functions with random pursuit. *SIAM Journal on Optimization*, 23(2):1284–1309, 2013.

[268] S. U Stich. Local sgd converges fast and communicates little. In *ICLR 2019 ICLR 2019 International Conference on Learning Representations*, number CONF, 2019.

[269] N Strom. Scalable distributed dnn training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[270] Y Sun, X Yin, J Jiang, V Sekar, F Lin, N Wang, T Liu, and B Sinopoli. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 272–285, 2016.

[271] J Supancic III and D Ramanan. Tracking as online decision-making: Learning a policy from streaming videos with reinforcement learning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 322–331, 2017.

[272] R. S Sutton and A. G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[273] C Szegedy, S Ioffe, V Vanhoucke, and A. A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[274] C Szegedy, V Vanhoucke, S Ioffe, J Shlens, and Z Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[275] C Tan, F Sun, T Kong, W Zhang, C Yang, and C Liu. A survey on deep transfer learning. In *International conference on artificial neural networks*, pages 270–279. Springer, 2018.

[276] M Tan, B Chen, R Pang, V Vasudevan, M Sandler, A Howard, and Q. V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *CVPR*, 2019.

[277] T Tan and G Cao. Fastva: Deep learning video analytics through edge processing and npu in mobile. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1947–1956. IEEE, 2020.

[278] J Tang and M Davies. A fast stochastic plug-and-play admm for imaging inverse problems. *arXiv preprint arXiv:2006.11630*, 2020.

[279] J Tang, K Egiazarian, M Golbabaee, and M Davies. The practicality of stochastic optimization in imaging inverse problems. *IEEE Transactions on Computational Imaging*, 6:1471–1485, 2020.

[280] J Tang, M Golbabaee, F Bach, and M. E davies. Rest-katyusha: Exploiting the solution's structure via scheduled restart schemes. In *Advances in Neural Information Processing Systems 31*, pages 427–438. Curran Associates, Inc., 2018.

[281] J Tang, M Golbabaee, and M. E Davies. Gradient projection iterative sketch for large-scale constrained least-squares. In *International Conference on Machine Learning*, pages 3377–3386. PMLR, 2017.

[282] S Thrun and A Schwartz. Finding structure in reinforcement learning. In *NIPS*, 1995.

[283] A Ullah, J Ahmad, K Muhammad, M Sajjad, and S. W Baik. Action recognition in video sequences using deep bi-directional lstm with cnn features. *IEEE Access*, 6:1155–1166, 2018.

[284] M Vacher, B Lecouteux, J. S Romero, M Ajili, F Portet, and S Rossato. Speech and speaker recognition for home automation: Preliminary results. In *SpeD*. IEEE, 2015.

[285] V Vanhoucke, A Senior, and M. Z Mao. Improving the speed of neural networks on cpus. 2011.

[286] J Vanschoren. Meta-learning: A survey. *arXiv preprint arXiv:1810.03548*, 2018.

[287] S Venugopalan, H Xu, J Donahue, M Rohrbach, R Mooney, and K Saenko. Translating videos to natural language using deep recurrent neural networks. *arXiv preprint arXiv:1412.4729*, 2014.

[288] J Wan, Y Yuan, and Q Wang. Traffic congestion analysis: A new perspective. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1398–1402, 2017.

[289] I Wang, E Liri, and K Ramakrishnan. Supporting iot applications with serverless edge clouds. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–4. IEEE, 2020.

[290] L Wang, D Ghosh, M. T. G Diaz, A Farahat, M Alam, C Gupta, J Chen, and M Marathe. Wisdom of the ensemble: Improving consistency of deep learning models. *arXiv preprint arXiv:2011.06796*, 2020.

[291] M Wang, C.-c Huang, and J Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[292] S Wang, T Tuor, T Salonidis, K. K Leung, C Makaya, T He, and K Chan. When edge meets learning: Adaptive control for resource-constrained distributed machine learning. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 63–71. IEEE, 2018.

[293] Y Wang, W Wang, J Zhang, J Jiang, and K Chen. Bridging the {Edge-Cloud} barrier for real-time advanced vision analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[294] Y Wang, W Wang, J Zhang, J Jiang, and K Chen. Bridging the Edge-Cloud barrier for real-time advanced vision analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[295] Z Wang, J Zhan, C Duan, X Guan, P Lu, and K Yang. A review of vehicle detection techniques for intelligent vehicles. *IEEE Transactions on Neural Networks and Learning Systems*, 34(8):3811–3831, 2022.

[296] L Wen, D Du, Z Cai, Z Lei, M Chang, H Qi, J Lim, M Yang, and S Lyu. UA-DETRAC: A new benchmark and protocol for multi-object detection and tracking. *Computer Vision and Image Understanding*, 2020.

[297] W Wen, C Wu, Y Wang, Y Chen, and H Li. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*, pages 2074–2082, 2016.

[298] W Wen, C Xu, F Yan, C Wu, Y Wang, Y Chen, and H Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, pages 1509–1519, 2017.

[299] Z Wen, D O'Neill, and H Maei. Optimal demand response using device-based reinforcement learning. *IEEE Transactions on Smart Grid*, 6(5):2312–2324, 2015.

[300] B Wu, F Iandola, P. H Jin, and K Keutzer. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 129–137, 2017.

[301] S Wu, G Li, F Chen, and L Shi. Training and inference with integers in deep neural networks. In *International Conference on Learning Representations*, 2018.

[302] Y Wu, M Schuster, Z Chen, Q. V Le, M Norouzi, W Macherey, M Krikun, Y Cao, Q Gao, K Macherey, *et al.* Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[303] E. P Xing, Q Ho, W Dai, J. K Kim, J Wei, S Lee, X Zheng, P Xie, A Kumar, and Y Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

[304] C Xiong, V Zhong, and R Socher. Dcn+: Mixed objective and deep residual coattention for question answering. *arXiv preprint arXiv:1711.00106*, 2017.

[305] W Xiong, C Shan, Z Sun, and Q Meng. Real-time processing and storage of multimedia data with content delivery network in vehicle monitoring system. In *2018 6th International Conference on Wireless Networks and Mobile Communications (WINCOM)*, pages 1–4. IEEE, 2018.

[306] R Xu, R Kumar, P Wang, P Bai, G Meghanath, S Chaterji, S Mitra, and S Bagchi. Approxnet: Content and contention-aware video object classification system for embedded clients. *ACM Transactions on Sensor Networks (TOSN)*, 18(1):1–27, 2021.

[307] R Xu, C.-l Zhang, P Wang, J Lee, S Mitra, S Chaterji, Y Li, and S Bagchi. Approxdet: content and contention-aware approximate object detection for mobiles. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 449–462, 2020.

[308] R Xu, S Razavi, and R Zheng. Edge video analytics: A survey on applications, systems and enabling techniques. *IEEE Communications Surveys & Tutorials*, 2023.

[309] N. J Yadwadkar, B Hariharan, J. E Gonzalez, and R Katz. Multi-task learning for straggler avoiding predictive job scheduling. *The Journal of Machine Learning Research*, 17(1):3692–3728, 2016.

[310] Z Yang, X Wang, J Wu, Y Zhao, Q Ma, X Miao, L Zhang, and Z Zhou. Edgeduet: Tiling small object detection for edge assisted autonomous mobile vision. *IEEE/ACM Transactions on Networking*, 2022.

[311] S Yi, Z Hao, Q Zhang, Q Zhang, W Shi, and Q Li. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–13, 2017.

[312] Y You, I Gitman, and B Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

[313] C Yu, A Velu, E Vinitsky, J Gao, Y Wang, A Bayen, and Y Wu. The surprising effectiveness of ppo in cooperative multi-agent games. *Advances in Neural Information Processing Systems*, 35:24611–24624, 2022.

[314] H Yu, S Yang, and S Zhu. Parallel restarted sgd for non-convex optimization with faster convergence and less communication. *arXiv preprint arXiv:1807.06629*, 2(4):7, 2018.

[315] H Yu, S Yang, and S Zhu. Parallel restarted sgd with faster convergence and less communication: Demystifying why model averaging works for deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 5693–5700, 2019.

[316] S Yun, J Choi, Y Yoo, K Yun, and J Young Choi. Action-decision networks for visual tracking with deep reinforcement learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2711–2720, 2017.

[317] S Zagoruyko and N Komodakis. Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer. *arXiv preprint arXiv:1612.03928*, 2016.

[318] X Zeng, B Fang, H Shen, and M Zhang. Distream: scaling live video analytics with workload-adaptive distributed edge intelligence. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 409–421, 2020.

[319] S Zhai, Y Cheng, Z. M Zhang, and W Lu. Doubly convolutional neural networks. In *Advances in neural information processing systems*, pages 1082–1090, 2016.

[320] Z.-H Zhan, X.-F Liu, Y.-J Gong, J Zhang, H. S.-H Chung, and Y Li. Cloud computing resource scheduling and a survey of its evolutionary approaches. *ACM Computing Surveys (CSUR)*, 47(4):63, 2015.

[321] B Zhang, X Jin, S Ratnasamy, J Wawrzynek, and E. A Lee. Awstream: Adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 236–252, 2018.

[322] C Zhang, A Kumar, and C Ré. Materialization optimizations for feature selection workloads. *ACM Transactions on Database Systems (TODS)*, 41(1):2, 2016.

[323] C Zhang, P Patras, and H Haddadi. Deep learning in mobile and wireless networking: A survey. *IEEE Communications Surveys & Tutorials*, 2019.

[324] H Zhang, J Li, K Kara, D Alistarh, J Liu, and C Zhang. Zipml: Training linear models with end-to-end low precision, and a little bit of deep learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 4035–4043. JMLR. org, 2017.

[325] H Zhang, G Ananthanarayanan, P Bodik, M Philipose, P Bahl, and M. J Freedman. Live video analytics at scale with approximation and {Delay-Tolerance}. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, 2017.

[326] H Zhang, M Shen, Y Huang, Y Wen, Y Luo, G Gao, and K Guan. A serverless cloud-fog platform for dnn-based video analytics with incremental learning. *arXiv preprint arXiv:2102.03012*, 2021.

[327] J Zhang, C De Sa, I Mitliagkas, and C Ré. Parallel sgd: When does averaging help? *arXiv preprint arXiv:1606.07365*, 2016.

[328] L Zhang, J Xu, Z Lu, and L Song. Crossvision: Real-time on-camera video analysis via common roi load balancing. *IEEE Transactions on Mobile Computing*, pages 1–13, 2023.

[329] M Zhang, F Wang, Y Zhu, J Liu, and Z Wang. Towards cloud-edge collaborative online video analytics with fine-grained serverless pipelines. In *Proceedings of the 12th ACM Multimedia Systems Conference*, pages 80–93, 2021.

[330] Q.-s Zhang and S.-C Zhu. Visual interpretability for deep learning: a survey. *Frontiers of Information Technology & Electronic Engineering*, 19(1):27–39, 2018.

[331] S Zhang, A. E Choromanska, and Y LeCun. Deep learning with elastic averaging sgd. In *Advances in neural information processing systems*, pages 685–693, 2015.

[332] W Zhang, W Guo, X Liu, Y Liu, J Zhou, B Li, Q Lu, and S Yang. Lstm-based analysis of industrial iot equipment. *IEEE Access*, 6:23551–23560, 2018.

[333] X Zhang, X Zhou, M Lin, and J Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *CVPR*, pages 6848–6856, 2018.

[334] Y Zhang, T Xiang, T. M Hospedales, and H Lu. Deep mutual learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4320–4328, 2018.

[335] C Zhao, C Chen, Z He, and Z Wu. Application of auxiliary classifier wasserstein generative adversarial networks in wireless signal classification of illegal unmanned aerial vehicles. *Applied Sciences*, 8(12):2664, 2018.

[336] C Zhao, M Shi, Z Cai, and C Chen. Research on the open-categorical classification of the internet-of-things based on generative adversarial networks. *Applied Sciences*, 2018.

[337] D Zhao, Y Chen, and L Lv. Deep reinforcement learning with visual attention for vehicle classification. *IEEE Transactions on Cognitive and Developmental Systems*, 2016.

[338] K Zhao, Z Zhou, X Chen, R Zhou, X Zhang, *et al.* Edgeadaptor: Online configuration adaption, model selection and resource provisioning for edge dnn inference serving at scale. *IEEE Transactions on Mobile Computing*, 22(10):5870–5886, 2023.

[339] W. X Zhao, K Zhou, J Li, T Tang, X Wang, Y Hou, Y Min, B Zhang, J Zhang, Z Dong, *et al.* A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[340] S Zheng, Y Song, T Leung, and I Goodfellow. Improving the robustness of deep neural networks via stability training. In *Proceedings of the ieee conference on computer vision and pattern recognition*, pages 4480–4488, 2016.

[341] B Zhou, J Cao, X Zeng, and H Wu. Adaptive traffic light control in wireless sensor network-based intelligent transportation system. In *VTC*, pages 1–5. IEEE, 2010.

[342] L Zhou, M. H Samavatian, A Bacha, S Majumdar, and R Teodorescu. Adaptive parallel execution of deep neural networks on heterogeneous edge devices. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 195–208, 2019.

[343] S Zhou, Y Wu, Z Ni, X Zhou, H Wen, and Y Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[344] Y Zhou, S Ebrahimi, S. Ö Arık, H Yu, H Liu, and G Diamos. Resource-efficient neural architect. *arXiv preprint arXiv:1806.07912*, 2018.

[345] Z Zhou, X Chen, E Li, L Zeng, K Luo, and J Zhang. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proceedings of the IEEE*, 107(8):1738–1762, 2019.

[346] M Zinkevich, M Weimer, L Li, and A. J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010.