Spring 2021

Service Abstractions for Scalable Deep Learning Inference at the Edge

Peizhen Guo
Yale University Graduate School of Arts and Sciences, patrick.guopz@gmail.com

Abstract

Service Abstractions for Scalable Deep Learning Inference at the Edge

Peizhen Guo

2021

Deep learning driven intelligent edge has already become a reality, where millions of mobile, wearable, and IoT devices analyze real-time data and transform those into actionable insights on the device. Typical approaches for optimizing deep learning inference mostly focus on accelerating the execution of individual inference tasks, without considering the contextual correlation unique to edge environments and the statistical nature of learning-based computation. Specifically, they treat inference workloads as individual black boxes and apply canonical system optimization techniques, developed over the last few decades, to handle them as yet another type of computation-intensive applications. As a result, deep learning inference on edge devices still faces the ever increasing challenges of customization to edge device heterogeneity, fuzzy computation redundancy between inference tasks, and end-to-end deployment at scale.

In this thesis, we propose the first framework that automates and scales the end-to-end process of deploying efficient deep learning inference from the cloud to heterogeneous edge devices. The framework consists of a series of service abstractions that handle neural network model tailoring, model indexing and query, and approximate computation reuse respectively. Together, these services bridge the gap between deep learning training and inference, lower the barrier and reduce the burden for both deep learning researchers and the system (and application) developers, and eliminate redundant computation while executing the inference tasks.

To build efficient and scalable services, we take a unique algorithmic approach of harnessing the semantic correlation between the learning-based computation. Rather than viewing individual tasks as isolated black boxes, we optimize them collectively in a white box approach, proposing primitives to formulate the semantics of the deep learning workloads, and algorithms to assess their hidden correlation (in terms of the input data, the neural network models, and the deployment environments) and then merge common processing steps to minimize redundancy.

Service Abstractions for Scalable Deep Learning Inference at the Edge

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Peizhen Guo

Dissertation Director: Wenjun Hu

June 2021

Contents

List of Figures

List of Tables

Acknowledgements

The past six years at Yale is a priceless and unforgettable journey to me. Lots of people made my Yale life as a fantastic experience. First and foremost, my sincere thanks go to Prof. Wenjun Hu, my thesis advisor, for guiding this work. She gave me a wonderful opportunity to work for a Ph.D. at Yale, and her continuous support of my Ph.D. study and research enabled me to work on multiple exciting projects. Second, I want to thank my committee members, Professors James Aspnes, Lin Zhong, and Mahadev Satyanarayanan for the valuable feedbacks regarding my thesis. My lab mates and collaborators Bo Hu and Rui Li also gave me great help along the journey. Special thanks to them for collaborating with me on multiple projects and aiding me to finish my Ph.D. degree successfully. Finally, I would like to sincerely thank my family, especially my parents, and my girlfriend for their unconditional love and perpetual support.

To my family

Chapter 1

Introduction

1.1 Deep Learning at the edge

AI-driven intelligent edge has already become a reality [2], where millions of servers, mobile and IoT devices analyze real-time data and transform those into actionable insights on the device. For example, real-time video analytics (e.g., for traffic monitoring [3], security surveillance [4], and smart retail [5]), natural language understanding (e.g., virtual assistance, smart email composition [6], and machine translator [7]), visual assistance [8], and industrial automation (e.g., defect detection, assembly line management[9, 4]) are already everyday examples. It is projected that, by 2022, over 60% of the data locally generated by devices (e.g., IoT, sensors, and mobile devices) will drive real-time intelligent decisions; 80% of the IoT and mobile devices shipped will have on-device AI capabilities [10, 11].

Many AI functionalities today are powered by deep learning (DL), an emerging machine learning technique known by its remarkable predictive performance already surpassing human beings in a broad range of real-world tasks. Such remarkable performance comes with a price. The DL models and algorithms are designed with millions or even billions of parameters, involving trillions of tensor computations. Preparing these models to run inference tasks requires thousands of GPU hours on training, and relies on powerful and efficient hardware for inference. However, edge devices are power and resource constrained, and hence used to primarily offload related computation to the cloud to benefit from the extraordinary capability of DL models [12]. But recently, increasingly inference workloads are run natively on the edge devices to provide better interactive user experience (e.g.,

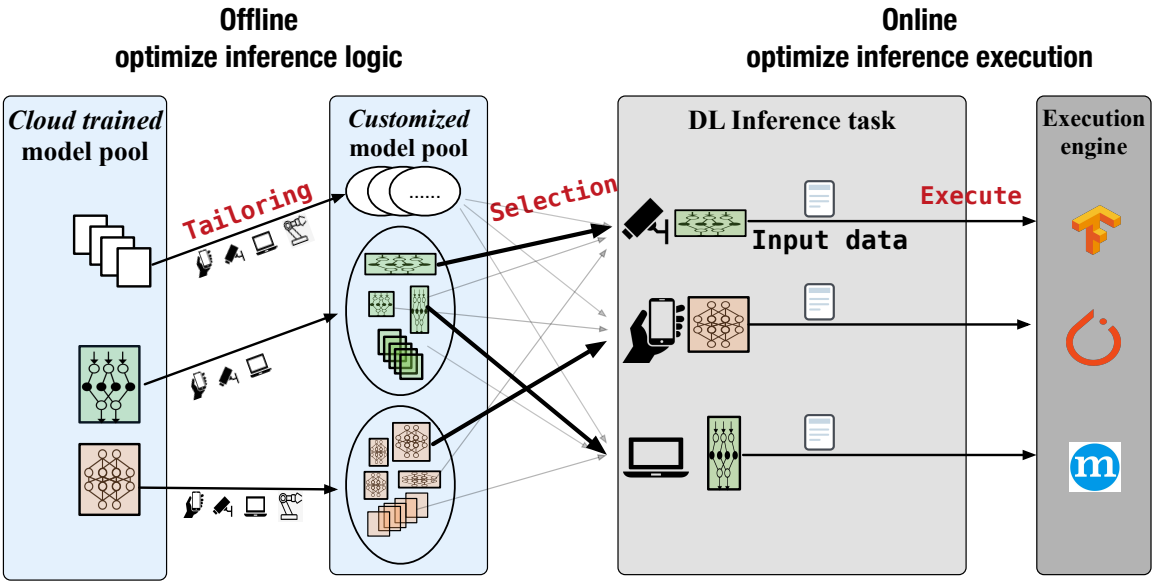


Figure 1.1: Lifecycle of deploying deep learning workloads on edge devices.

∼10 ms real-time response), data privacy (e.g., keeping sensitive medical data local), and reliability (e.g., avoiding being affected by the network connectivity and bandwidth issues [13, 14]).

In general, enabling efficient and scalable DL inference on edge devices relies on two parts: 1) *preparing* the DNN model that achieves good performance on the inference function with affordable resource usage (blue boxes in Figure 1.1); and 2) *executing* the inference tasks (on this DNN model) efficiently when input data arrive (grey boxes in Figure 1.1). Note that designing a state-of-the-art DNN model is an immense undertaking nowadays (e.g., taking hundreds of GPU hours, millions of dollars, and deep understanding of optimization theories [15, 7]). Hence, DNN model repositories, storing pre-trained models for diverse use scenarios, are increasingly adopted, becoming an essential component of the current DL ecosystems. Given this trend, "preparing" a DNN model essentially means *customizing and matching* the pre-trained DNN models designed for the cloud to the various application scenarios and diverse execution environments of the edge devices.

Therefore, to be more specific, we divide the whole deployment process into three logical steps (Figure 1.1). First, it necessitates ***tailoring*** (i.e., adapting) the complex deep neural network (DNN) model originally designed and trained on the cloud to edge settings with a range of hardware capability. By various estimates, in the next three to five years there will be over 50 billions IoT devices [16] with very diverse hardware profiles and runtime requirements. This creates a massive

search space and complexity. Then, given numerous DNN models available to use, the users (e.g., application developers, edge device administrators, etc.) have to ***select*** a model that is best suited to the required learning functionality and the multitude of constraints (e.g., resource usage, accuracy, latency, etc.). This typically involves profiling and identifying precisely which model to use from potentially hundreds of DNNs in a model repository (e.g., TF-Hub [17]), including the specific version for a particular DNN design. Finally, it needs scalable and efficient ***execution*** of the DNN inference tasks on the resource-constrained edge devices to meet the demand of interactive user experience with real-time input data arriving continuously. Given the dilemma between having lightweight and energy-saving devices and using complex models to achieve best device intelligence, there is a dynamic tradeoff made by the developers to achieve the desired performance by controlling both the systems and the learning-based application logic. These will be further explained in Section 2.1.2.

Existing works for optimizing deep learning inference at the edge include the following directions: building lightweight inference engines for DNN models (e.g., TF-Lite [18] and MCDNN [19]), proposing end-to-end compiler toolchains for graph-level optimization and accelerator kernel code generation (e.g., TVM [20]), and designing specialized compact neural network architectures for mobile and IoT devices (e.g., MCUNet [21] and EfficientNet [22]). In general, they either treat inference workloads as individual black boxes and apply system and compiler optimization techniques, developed over the last few decades, to handle the DL based applications as yet another type of computation-intensive workloads, or abstract the whole process as a non-convex optimization problem in deep learning theory (Section 2.2). Therefore, even though they each resolve specific problems while deploying DL inference workloads, we are still a long way from seamless deployment of these workloads on current edge devices. The overarching issue is around scalability, which we will further discuss next.

1.2 Scalability challenges in deploying Deep Learning inference to the edge

In this section, we will explain in detail the three fundamental factors that cause the scalability challenges, manifested in each logical step of deploying cloud-trained DNNs to the edge.

Table 1.1: Popular DL hardware specifications.

GPU	**Peak perf**	**Memory**	**Bandwidth**
V100	112 TFLOP	32 GB	900 GB/sec
2080	11.7 TFLOP	11 GB	480 GB/sec
Edge GPU	**Peak perf**	**Memory**	**Bandwidth**
Jetson AGX	11 TFLOP	16 GB	136 GB/sec
Jetson TX2	1.5 TFLOP	4 GB	58 GB/sec
Jetson nano	0.47 TFLOP	4 GB	25 GB/sec
ASIC	**Peak perf**	**Memory**	**Bandwidth**
Edge TPU [25]	4 TFLOP	-	-
Raspberry pi	6 GFLOP	2 GB	8.5 GB/sec

1.2.1 Heterogeneity in deployment environments

The deployment environments for edge devices are extremely heterogeneous, in terms of the *hardware profile*, *application requirements*, and *runtime resource availability*.

Hardware heterogeneity. The hardware profiles of the edge devices are incredibly diverse, ranging from embedded sensors, IoT devices, mobile phones/tablets, to edge servers, with a full spectrum of capability [23]. Table 1.1 lists the specifications of some selected GPU and ASIC accelerators, from high-end to low-end, widely employed at the edge for DNN-based workloads. For the same DNN inference workload, the completion times for low-end (e.g., nano) and high-end (e.g., 2080) devices differ by orders of magnitude (e.g., 229 ms and 9.8 ms to run inference over ResNet). Even when only considering smartphone platforms, to deploy a DL-based mobile app in the App Store requires considering over *hundreds* of types of hardware devices, from high-end iPhone 12 with dedicated neural processing units to 7-year-old Nexus 5 with much slower CPU processors [24]. Meanwhile, for the same hardware under different battery conditions and usage modes, the effective processing capability also differs significantly which further increases the complexity of the hardware heterogeneity.

Heterogeneous performance requirements. For a single DL function (e.g., object classification), the performance requirements, e.g., *latency* and *accuracy*, vary with deployment endpoints. Therefore, current pre-trained DNNs exhibit wide-ranging performance characteristics for the same functionality. Table 1.2 shows the inference time and accuracy of commonly used DNNs for two vision-based workloads and an NLP example (numbers from their original publications) [26, 27]. For the same

Table 1.2: Inference time and accuracy for different workloads and DNNs running on NVIDIA RTX 2070 GPU.

Workload	Network	Accuracy	Time (ms)
Obj recog.	MobileNet	68.8	3.47
	ResNeXt101	79.2	21.1
Obj Detect.	YOLO	57.9	22.5
	SSD512	74.9	156
Q&A	BiDAF	76.8	59
	BERT	88.5	323

workload, the DNN inference time differs by up to 8×, and the inference quality (e.g., in *accuracy*, *F1 score*) varies by as much as 25%. Furthermore, state-of-the-art algorithms could adapt a DNN model towards a wide range of performance targets. For instance, neural architecture search algorithms generate different variants of EfficientNet for object classification with 8× difference in memory usage, 6× difference in inference latency, and 7% difference in absolute classification accuracy [22]. Similarly, different variants of BERT model for NLP tasks differ by 10× in speed and memory usage, and 15% in accuracy [28].

Runtime resource availability. The availability of system resource (e.g., memory space, CPU cycles, and accelerator quotas), varies on the edge device during runtime due to other workloads competing for the same resource. For instance, when an edge device launches or completes workloads, or adjusts the resource allocation of the containers that serve the DNN model for inference tasks, the perceived resource availability to any active workloads changes [29, 30].

These numbers together outline a *huge*, *complex*, and *dynamic* design space to explore *fine-grained* tradeoff points between hardware, resource, and inference performance. For each single DL based task, each edge deployment scenario maps to a distinct tradeoff point that requires specific efforts for customization, because a sub-optimal tailoring and/or selection of DNN models could easily lose 10s of percentage points of accuracy or miss the latency requirement for real-time processing by hundreds of milliseconds [31].

1.2.2 Interdisciplinary expertise and manual efforts

The second set of contributors of the scalability challenge are the interdisciplinary expertise and manual efforts.

Regarding the expertise, the DL users (e.g., application developers, system administrators, etc.) need to understand the DL model and algorithm internals. The ML researchers (e.g., DNN developers) need to understand the low-level system primitives and execution details, keep track of the latest hardware platforms, and eventually translate them into upper-level neural network design constraints. Deep learning theory and systems are both fast evolving fields, with new techniques being proposed perhaps daily, and this presents an increasingly high barrier for the average developer to optimally deploy deep learning inference based applications on edge devices.

Manual efforts include configuring the algorithm parameters (e.g., learning rate), preparing the datasets (e.g., both compact and unbiased), annotating the models (e.g., to annotate deployment goals), and managing the execution runtime of the three deployment steps. More importantly, these efforts grow linearly or even exponentially with the number of models, deployment environments, DL workloads and their performance requirements.

We will further illustrate the expertise and manual efforts needed for completing the three aforementioned stages of enabling deep learning applications on edge devices.

First, to tailor DNNs towards heterogeneous deployment settings, there are two possible ways. The DL researchers should understand the resource budget and performance goal of each possible type of edge device, interpret them as DNN design demands and finally generate DNNs to cater to these demands as an additional step for model training. Alternately, the model users should prepare datasets, select and apply the right training algorithms to tailor already published DNNs towards their custom settings.

Second, to select the right one from potentially hundreds or thousands of DNN models to achieve the desired learning-based function within reasonable resource budgets, the onus is on the user to understand precisely which model performs best in their specialized scenario, and profile the model thoroughly on their deployment platform regarding the dynamic resource availability during runtime inference. A suboptimal model could miss the achievable accuracy target by 10%, break the real-time latency guarantee by seconds, or waste 20 $\times$ more resources. This is especially challenging for the average users who simply wish to easily select and embed a DNN model in their application.

Third, to efficiently execute the inference tasks on an edge device, users have to understand both the algorithmic aspects of the DNN model (e.g., each convolution layer extracts visual features at what granularity), and the system aspects of the basic operations (e.g., the computation and memory access

pattern for the parallelizable matrix multiplication operation), so as to develop application-specific solutions to optimize the end-to-end performance of their learning-based workloads.

1.2.3 Fuzzy computation redundancy

The approximation nature of deep learning inference as well as the spatial-temporal correlation between edge devices together lead to a new type of *fuzzy* computation redundancy (further explained in Chapters 3-5). For instance, tailoring a DNN model towards two devices with similar hardware capabilities currently involves two separate DNN structure learning processes, which seems obviously redundant. As another example, suppose two DL inference tasks take similar input data (e.g., photos of the same object taken from different vantage points) and run on similar models (e.g., ResNet and InceptionNet); they will likely produce the same classification result and running both are clearly redundant.

Note that such redundancy is fundamentally different from traditional meaning of computation redundancy, not characterized by *identical* input data, function logic, or execution settings. Instead, it is characterized by the level of *semantic correlation*. Hence, such widely existing fuzzy redundancy is not captured and handled by any existing system. Given that deploying DL inference already involves tremendous computation, time, and manual efforts, such redundancy becomes a significant contributing factor to the scalability issue, leading to extreme inefficiency in both model porting, selection, and actual inference execution.

Redundancy caused by correlated input data. Deep learning applications on mobile and IoT devices naturally involve computation on contextual data. These applications are typically invoked on multiple devices in close proximity, processing similar and correlated contextual data that map to the same outcome.

For instance, Google Lens [32] has become very popular, which enables visual search by recognizing objects in the camera view and rendering recognized information using DNN. Consider a scenario where the tourists near a famous landmark search for its history using the app. Clearly, it is redundant to run the same recognition function repeatedly on different devices for the same landmark. Although the devices capture different raw images, semantically the images are about the same landmark. If the recognition results are not shared among nearby devices, but instead

computed from scratch on each edge device, it leads to significant redundancy. An empirical study using Google Streetview API [33] to create an emulated landmark recognition scenario reveals that over 83% of the images trigger redundant landmark recognition logic.

Redundancy caused by correlated DNN models. New inference workloads are increasingly generated from similar deep neural network (DNN) models. The similarity between models stems from three trending practice: (i) The same standard datasets (e.g., ImageNet [34]) are used to train models to perform the same task; (ii) Model variants are derived from common base models incrementally via transfer learning; (iii) Knowledge distillation techniques leverage the well-trained existing models to guide the training of the new models and further produce semantically similar interpretations of the input data by the old and new DNN models. As a result, there is significant correlation between the inference workloads in terms of their *generalized* functional semantics. Note that such semantics corresponds to the general functionality the DNNs attempt to approximate, rather than the exact mathematical expressions of the DNNs.

Such correlation first leads to redundancy between inference tasks. Namely, when several inference tasks with "equivalent" models (or segments) are invoked on identical or highly similar inputs, the seemingly distinct workloads are in fact semantically equivalent, leading to redundant computation. As an example, we select a set of similar DNN models (e.g., InceptionV3, ResNet50, VGG19, MobileNet, and ResNeXt101) and feed them the same dataset. Surprisingly, we observe that over 95% of their outputs agree with one another, and this agreement ratio is even significantly higher than their inherent top-1 accuracy (e.g., 75%-85%), which reflects the "distance" between the DNN models and the decision logic they are trying to approximate.

Further, such correlation between different DNN models results in redundant efforts of edge developers for model selection. Existing model repositories (e.g., TF-Hub [17]) act as a remote filesystem only, with primitive APIs to publish and load a model. To select a good model, a user has to specify the precise URL to the model file and manually profile its performance in a loop until the best fit is found. Not capturing the hidden correlation between DNN models, all these efforts have to be repeated from scratch on all relevant DNN models every time a user searches for a suitable DNN, which requires significant user sophistication, extremely unscalable computation time (e.g., thousands of GPU hours) and cost (e.g., millions of dollars [35]).

Redundancy caused by correlated execution environments. Mentioned previously, the first step for enabling a cloud-designed DNN-based function on an edge device is to *tailor* (i.e. port) the model towards various deployment constraints and requirements. Unfortunately, the current practice of porting relies on laborious source code annotations and significant complexity on model architecture adaptation. For instance, preparing the model tailoring logic for ResNet50 requires around 30 lines of code edits scattered around several source files. Further, using state-of-the-art neural architecture search (NAS) algorithms to fit a neural network architecture to a single edge device deployment setting could lead to CO2 emission equivalent to 5 cars' lifetime [36]. The situation gets even worse when considering the intractable target space of heterogeneous edge device environments. Handling each porting process independently is extremely unscalable.

Meanwhile, we observe that multiple DNN architecture adaptation trials often share similar initial steps or training iterations and a large portion of them are interchangeable, i.e., unnecessarily repeated. For instance, when looking into 128 randomly generated DNN adaptation targets (each corresponding to an edge deployment setting), we find that up to 98% of the overall computation efforts can be eliminated if carefully reusing the common initial training iterations [37].

1.3 Requirement summary

Given these factors causing scalability challenges for end-to-end deep learning inference deployment, we summarize what it takes to address the challenges. Correspondingly, the requirements are as follow:

- *Fine-grained customization.* To handle the multi-dimensional heterogeneity of the edge deployment environments, we need scalable algorithms and mechanisms to reason about the functional semantics and resource profile of the DNN models so as to do fine-grained DNN tailoring and selection, where each edge deployment target is matched with a DNN model that best fits the desired functionality, hardware constraints and performance goals.

- *Redundancy elimination.* To resolve the emerging type of fuzzy computation redundancy, we need the right primitives to detect and measure the sources of redundancy, and algorithms to merge common computation steps to eliminate such redundancy.

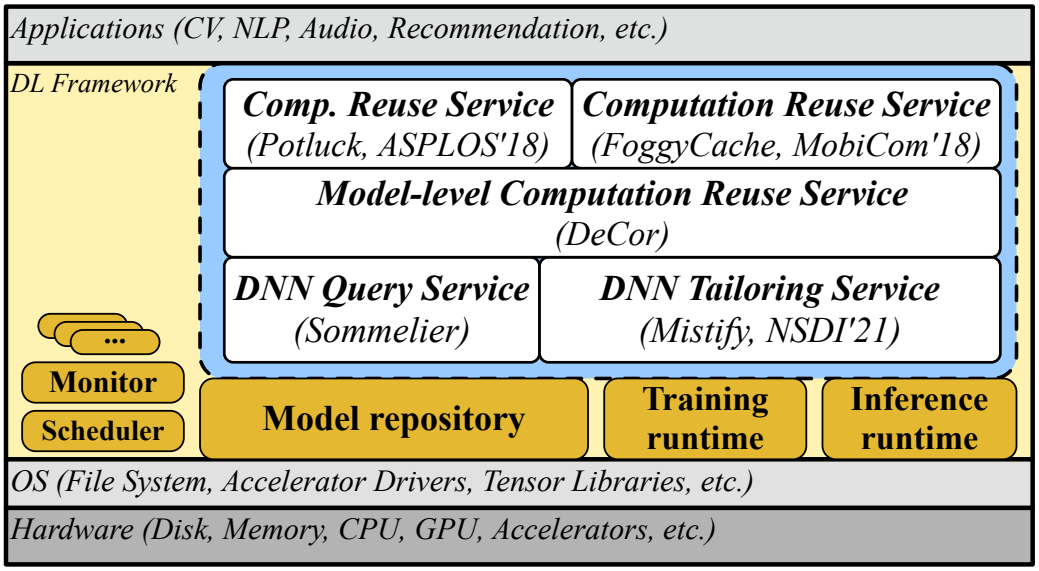


Figure 1.2: System stack (white boxes are our thesis components).

- *Service abstraction.* To simplify the manual efforts, lower the barriers for average users without the expertise in both DL algorithm and execution internals, and scale up the previous two algorithmic steps, we need the right service abstractions to abstract the complexity and build system support to automate and optimize the laborious processes.

1.4 Contributions

This thesis proposes the **first generic framework** towards automated, scalable, and efficient deployment of DL inference from the cloud to edge devices. The framework consists of five services, *Mistify*, *Sommelier*, *Potluck*, *FoggyCache*, and *DeCor*. Shown as the white boxes in Figure 1.2, these services collectively serve as an intermediate layer built upon the vanilla deep learning framework, supporting various deep learning applications.

Figure 1.3 further shows the end-to-end DL ecosystem spanning the cloud (or central servers) and edge devices. The blue boxes, namely the five thesis components, address the challenges at different stages of the end-to-end DL deployment process (from training to inference). Specifically, *Mistify* [37] develops a DNN model architecture tailoring service bridging the gap between the cloud (training) and the edge (inference) facilities. *Sommelier* provides DNN model indexing and query service, above the model repository stack, supporting fine-grained DNN model selection for the users,

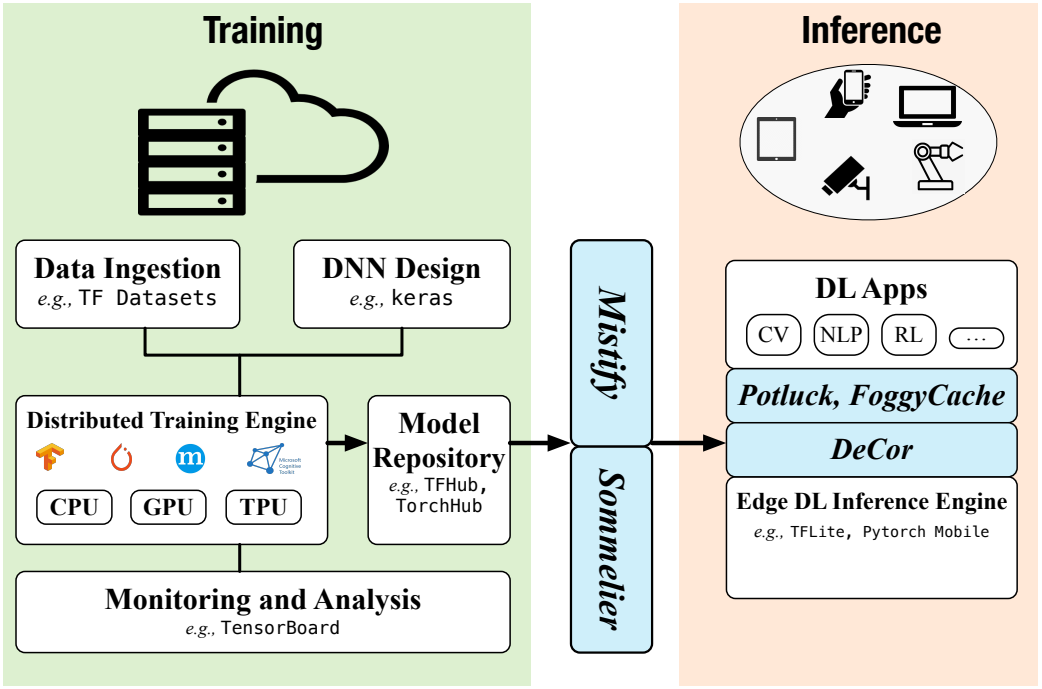


Figure 1.3: Big picture of the edge deep learning ecosystem (blue blocks are our thesis works).

workload developers, and the serving platforms. Above the DL inference execution engine, we have *Potluck* [38], *FoggyCache* [39], and *DeCor* that collectively perform the caching and computation reuse service to accelerate the speed and improve the resource efficiency of the inference tasks via eliminating the computation redundancy between them.

Referring to the aforementioned requirements to tackle the scalability challenge, we further elaborate how they are addressed by each individual piece.

Fine-grained customization: Mistify proposes algorithms to capture and merge common steps among individual tailoring requests, and collaboratively fine-tune model parameters afterwards with privacy-awareness. Together, they make fine-grained DNN structure tailoring towards each single edge deployment setting practical and scalable. Meanwhile, *Sommelier* proposes algorithms to express the semantics of DNN models and index structures to organize models according to their semantic correlation and resource profiles, based on which model selection with fine-grained resource and performance constraints (e.g., functional semantics, accuracy, latency, resource usage, and etc.) is fully supported with both high efficiency and good quality.

Redundancy elimination: Potluck and *FoggyCache* propose algorithms and runtime support to

eliminate redundancy by caching and reusing computation results with semantically correlated input data across applications and devices respectively; *DeCor* further extends the computation reuse scheme to consider inference tasks that run on semantically correlated models; *Sommelier* proposes semantic-based DNN indexing mechanisms to avoid repeated profiling and benchmarking overhead; and *Mistify* proposes collective adaptation to merge common processing steps of tailoring a DNN model towards different settings. Although they focus on different phases (i.e., tailoring, selection, and execution) and different aspects (input data, model logic, and execution settings) of deploying DL inference, these works, in common, achieve the general idea of gauging the "similarity" between individual processing tasks and merge them accordingly to eliminate redundant processing of the common parts.

Service abstraction: Each component of the thesis provides a novel service abstraction, outsourcing the interdisciplinary expertise, unnecessary development complexity, and laborious manual efforts from the users. For instance, with *Mistify*, the DL researchers can focus on designing new DNNs, without worrying about the deployment concerns. With *Sommelier*, the application developers can focus on writing the business logic of the app, avoiding the complexity of tuning the DL-based logic. With *Potluck*, *FoggyCache*, and *DeCor*, the platform developers can focus on optimizing the execution speed and resource efficiency of each individual inference task, and no longer need to develop sophisticated caching schemes for different types of workload and runtime setting. Meanwhile, these thesis components collectively form an overall framework. Namely, a DNN porting service by *Mistify*, model indexing and query service over the repository by *Sommelier*, and computation reuse service intercepting the inference execution path by *Potluck, FoggyCache* and *DeCor*. Together, these services lower the barrier for deploying DL inference on edge devices, automate the end-to-end process, and make it scalable and efficient.

Summary. This thesis makes the following contributions:

- We observe and quantify the existing challenges in the end-to-end process of deploying deep learning inference tasks from the cloud to heterogeneous edge devices and propose the **first universal framework** covering the whole deployment lifecycle. The framework decouples the currently intertwined DL design and deployment phases, removing the unnecessary complexity from both the ML researchers and application developers.

- From a system perspective, we propose a series of missing service abstractions and build runtime systems to handle the three essential steps of moving DL inference from the cloud to the edge, which scale the end-to-end deployment process, and reduce the required interdisciplinary expertise and laborious manual efforts from the users by orders of magnitude.

- From an algorithmic perspective, we leverage a unique paradigm of harnessing the semantic correlation between the learning-based computation to build efficient and scalable services. Rather than viewing individual tasks as isolated black boxes, we optimize them collectively in a white box approach, proposing primitives to formulate the semantics of the deep learning workloads, and algorithms to assess their hidden correlation (in terms of the input data, the neural network models, and the deployment environments) and then merge common processing steps to minimize redundancy.

1.5 Dissertation roadmap

Chapter 2 first explains the background of deep learning inference and then gives a broad overview of the existing efforts of optimizing DL workloads at the edge. Later sections elaborate on related work in more detail.

The following three chapters will detail the service abstractions we propose, the algorithms we design, and the systems we build. Following a problem driven narrative of how users deploy deep learning inference workloads to edge step by step, Chapter 3 explains the DL logic tailoring service. Then, given the ready-to-use models in the model repository, Chapter 4 explains details of how to automate and scale the DNN model selection while meeting the multi-dimensional requirements of the users. Finally, Chapter 5 focuses on how to harness the semantic correlation between input data and DNN models to design caching and computation reuse primitives to eliminate computation redundancy and accelerate deep learning inference execution.

Chapter 6 concludes this thesis and discusses a few potential directions extended from this thesis that can be further explored in the future.

Chapter 2

Background

2.1 Deep Learning Inference at the Edge

2.1.1 Current trend

The emerging trend of deploying deep learning inference functions on edge devices have become a mainstream, because of the increasing number of learning-based mobile and IoT applications and the rapidly growing power and capability of the edge devices [40, 13]. On-device DL inference brings the advantage of avoiding cloud communication latency and not affected by network connectivity issues. More importantly, it provides stronger privacy guarantee where sensitive user data (e.g., medical records) never has to leave the device. These all facilitate broader use scenarios of deep learning techniques on edge devices.

Broad scope of deep learning based applications. An increasingly wide range of DL based applications are used on mobile, IoT, and embedded devices. Typical use scenarios include real-time video analytics [41, 42, 43], autonomous driving [44, 45, 46], intelligent manufacturing and agriculture [4, 47], smart home and city [9, 5], personal cognitive assistance [32, 48], etc.

Rapid growth of edge device capability. Traditionally, these learning-based workloads are offloaded to central servers (e.g., cloud, cloudlet, etc.) to process, due to the limited space and processing capability of edge devices. Nowadays, increasingly powerful mobile processors and customized AI chips [49, 25, 50] become available, which can handle increasingly complex computation in real-time. For instance, state-of-the-art object detection pipeline can already run on NVIDIA

Jetson TX2 chip at 30 fps [26].

Need for on-device deep learning inference. Many DL applications, especially those with low latency or high data privacy requirements, need on-device inference execution instead of remote processing.

Although remote cloud processing relieves the burden from the processor of the edge devices, it adds additional cost for streaming data to the cloud and unpredictable communication latency to the end-to-end response time of the workload. For instance, LinkShare [51] observes that over 50% of the requests could break the real-time response requirements when multiple sources are contending the communication channel for offloading without an optimal scheduling strategy. This is unacceptable for the scenarios (e.g., manufacturing, autonomous driving) with low latency requirements. Overall, as the on-device computation time of DL workloads rapidly decrease over time, the trend of avoiding remote processing will soon become increasingly dominant.

Meanwhile, for emerging DL applications covering functions like surveillance, medicine, and recommendation, they will access very sensitive personal data (e.g., medical records, biometric data, and Internet browsing actions) and therefore have a strong preference of keeping the data local without streaming to remote cloud for processing [52, 53]. For instance, Federated Learning [54] is an emerging paradigm proposed to address the privacy concern by conducting on-device neural network training in a collaborative manner. On-device inference will only further guarantee the privacy of the sensitive data, which will unlock more privacy-sensitive fields to benefit from deep learning.

2.1.2 Lifecycle of deploying DL inference on edge devices

Noticeably, enabling efficient deep learning inference at the edge is far beyond simply focusing on the on-device workload execution. It involves the optimization efforts in two aspects, inference *logic* and inference *execution*. Figure 1.1 in Chapter 1 already illustrates the overall lifecycle of deploying a cloud trained DNN logic to edge devices for inference, where the two optimization stages: inference logic and inference execution are marked in blue and gray shades respectively. Next, we will further discuss the two aspects.

To optimize the inference logic, we need to take the dynamic and heterogeneous execution

environments of the edge devices into consideration and *tailor* the *platform-agnostic* (cloud designed and trained) DL model into a range of *platform-aware* models (e.g., variants that best fit tablets, smartphones, and cameras respectively). We will further elaborate on this stage in Chapter 3. Further, given the rapidly growing DL use cases and therefore the ever increasing amount of newly designed DNN models, to *select* the one that best fits the required functionality, performance goals, and resource budgets is an increasingly important optimization step for the average users who want to incorporate DL functionality in their programs. For instance, for a simple object classification task, there are over three hundreds pre-trained DNN models in TF-Hub. Each has its own resource usage profile and accuracy performance with respect to specific datasets. We discuss our support for DNN model selection in Chapter 4. Note that the inference logic optimization mainly considers machine and resource heterogeneity. We leave the fine-grained optimization regarding the dynamics of the input data as one of the future directions.

To optimize the inference execution on edge devices, it comes down to two sub-stages. Intuitively, it first involves using a global view to conduct cross-task optimization among different edge devices that are executing similar inference logic. Note that this is an essential stage that can harness the unique characteristics of edge computing where co-located devices and their contexts are often spatially and temporally correlated. Chapter 5 will further explain how the thesis optimizes inference task executions at this stage. Then, zooming into each specific task, the execution engine will leverage compiler and system optimization techniques to accelerate the inference task processing and reduce resource consumption.

In the next section, we will explain the existing works in three categories. Further, referring to Figure 1.1, we will discuss the missing points of each category of works and how they are manifested as the key challenges of enabling efficient DL inference at the edge.

2.2 Related work

This section reviews overall directions of optimizing deep learning inference. Each of the latter chapters will discuss other related work more specific to the topics at hand.

2.2.1 Edge-Centric Inference Engine

To run DL inference workloads efficiently on edge devices, several targeted engines are developed. They propose efficient graph executor runtime, optimize operator kernels and executables, and leverage additional hardware support to accelerate the deep learning workload execution.

Optimizing executor runtime. Several end-to-end DL inference engines have been built (e.g., Tensorflow-Lite [18], TensorRT [55], EIE [56], and others [57, 58, 59]) that expose straightforward APIs for users to import models and submit inference tasks, and meanwhile internally leverage full-stack optimizations (library, OS, and hardware) to accelerate the DL inference workloads by orders of magnitude faster than simply using CPU. The ultimate goal of these platforms is to outsource the efforts of performance tuning so that users can mainly focus on the learning logic. Many other works, such as Clipper [60], Pretzel [61] and others [62, 63, 64], propose additional layers and optimizations of the executor runtime of the inference engines and further focus on ease of deployment, resource sharing, caching, and inter-model optimizations.

Optimizing operator kernels. Typical efforts in optimizing neural network operator kernels fall into two categories: algorithm and hardware focused. For algorithm focused efforts, many recent works (e.g., Winograd [65] and others [66, 67, 68]) propose algorithmic variants of the vanilla neural network operators to carry out the same computation logic with less complexity. For instance, they leverage transformations (e.g., FFT) to change the number of matrix multiplications with a tolerable sacrifice of storage cost and numerical stability. For hardware focused efforts, they leverage the unique characteristics of certain hardware to design the data layout and transform the computation to best fit the hardware. For instance, Halide [69] and a few relevant works decouple the operator kernel scheduling from computation and match that with the GPU hierarchy to fully benefit from data locality. A few other works exploit the buffer capacity and architecture to achieve weight sharing while accelerating convolution operations [70, 71].

Additional hardware support. Following the theory-system codesign approach, there have been a large body of works building novel hardware architecture and use it to support deep neural network computation. These works profile the data access pattern, computation parallelism, and data locality characteristics of the most widely used DL operators and then use these as the principles to design new hardware, including number of processing unit, cache size, bus bandwidth, memory hierarchy, etc. A few typical works include GPU [72, 73], TPU [74], and DianNao [75, 76].

Application-specific solutions. For each specific type of the applications (e.g., video analytics, cognitive assistance, and wearable computer vision), there are a large body of works providing specialized support to optimize the execution of the DNN-based inference tasks, leveraging the unique characteristics of the applications. For instance, Starfish [77] supports efficient execution of concurrent vision workloads; DeepMon [78] accelerates real-time object detection on mobile devices by offloading convolution layers to the GPU; Gabriel and relevant works [79, 80] leverage nearby edge servers (i.e. cloudlets) and adaptive model (and execution path) switching techniques to achieve fast and resource-efficient execution of wearable cognitive applications; Noscope [41] supports large-scale DNN-based video analytics with multi-branch and early-exit techniques to optimize the query execution with minimal resource consumption.

2.2.2 Machine Learning Compiler

Deep neural networks are represented by computation graphs. Intuitively, it draws the connection to compiler optimizations where control graph abstractions are used as the intermediate representation (IR) for the programs to analyze and optimize their data and control flows. Therefore, a large body of works explore the DNN optimization from a compiler perspective. We classify them into three categories, end-to-end, high-level, and low-level optimization, to further summarize.

End-to-end optimization. End-to-end machine learning compiler optimizations are typically integrated as the compilation passes of the representative machine learning frameworks, handling both the high-level graph optimizations and the low-level operator optimizations and hardware-specific code generations. For instance, XLA [81] and MLIR [82] are proposed and initially embedded in Tensorflow ecosystem [83]; TVM [20] is combined with NNVM [84] (and its later variant Relay [85]) as the compiler optimization stack for the MXNet framework [86]; and Glow [87] serves as the

compiler for Pytorch environment [88]. These works take a user-defined neural network model as the input "program", first translate it into a high-level IR (typically graphs) to apply computation graph optimizations such as operator fusion, data layout transformation, memory planning, etc. Then, the optimized computation graph is further translated into a low-level IR (typically sequences of instructions) to generate platform-specific machine code with optimized loop unrolling, tiling, tensor scheduling, etc.

High-level graph optimization. In addition to the end-to-end solutions mostly lead by the industry, there are many recent works specifically focusing on the high-level graph optimization aspect. These works leverage the semantics kept in the computation graph to perform domain-specific optimizations. For instance, Metaflow [89] proposes iterative neural network operator fusing and graph rewriting while guaranteeing the graphs before and after rewriting are mathematically equivalent. ColocRL [90] uses reinforcement learning to discover more efficient strategies to place the computation graph on GPU devices. TASO [91] provides rule generation algorithms to automate the graph substitution process jointly considering the mathematical properties of the DNN operators and its effects on the overall performance.

Low-level operator optimization. Most of the low-level optimizations happen at the granularity of a single operator or instruction. Some existing works are already covered in the operator optimization paragraph in Section 2.2.1. Meanwhile, other works explore how to build automated pipelines for code generation and operator kernel optimization from a compiler perspective. For instance, Astra [92] incorporates lightweight profiling and the repetitiveness nature of deep learning computation to optimize the kernels automatically. TVM also proposes a learning-based algorithm to automatically search the best scheduling strategy for tensor programs [93].

2.2.3 Compact Neural Network Architecture

In production, both deep learning researchers and users fully realize the necessity of customizing cloud-designed sophisticated DNN models to edge device settings by tailoring the model structure and tuning the parameters.

DNN designer-driven approaches. Initially a series of DNN models hand-tuned by experts are proposed that perform both fast and accurate prediction on mobile devices [94, 95, 96, 97]. The

essential techniques include quantization, neuron sparsification, and neural block simplifications.

Recently, Distiller [98], AMC [99], MorphNet [100], OFA [101], ChamNet [102], EfficientNet [22], and many neural architecture search (NAS) works [103, 104, 105, 106, 107, 108] further explore how to leverage recurrent neural network (RNN) and evolution algorithms to perform DNN compression, adaptation, and architectural search in an algorithmic way, obviating the need for hand-tuning by experienced experts. These approaches adjust the model structure from parameter, connectivity, and operator perspectives by automatically searching in the whole space of different hyper-parameter combinations.

In addition, a series of works in deep learning theory start with a counter-intuitive observation of deep neural network training, the simultaneous growth of model complexity and generalization performance, and try to explain from a compression point of view by figuring out the lowest complexity to express specific DL logic. Some works give theoretic proof of why existing DNN models can be compressed to more compact variants without sacrificing performance [109, 110, 111]. Other works propose practical training algorithms like knowledge distillation [112, 113, 114, 115] to enhance the DNN compression and adaptation quality.

DNN user-driven approaches. Hereby, DNN users refer to the machine learning platform and application developers who "use" (run inference on) certain DNN model architecture, instead of proposing their own DNNs. Edge-targeted frameworks like TF-Lite [18], PyTorch Mobile [116], and application platforms like MCDNN [19] provide built-in model *compression*, *quantization* and *switching* support. With minimal understanding of the DNN model internals, these works can only support post-training, profiling-based compression such as 8 bit quantization, random dropout, etc.

2.2.4 Summary - the missing points

Unfortunately, existing works do not readily support efficient deep learning inference on edge devices. We will summarize the missing points of existing works, which also motivate this thesis.

Inference logic optimization. Although there are a huge body of existing works that aim to provide compact neural networks with high performance. They still face two fundamental challenges, corresponding to the designer-driven and user-driven approaches. 1) For the designer-driven works, the efforts are huge and unscalable to do fine-grained model tailoring and selection brought by

the dramatically increasing edge environment heterogeneity and DNN model diversity. 2) For the user-driven approaches, there are increasing technical difficulties (about DNN logic and inference system internals) and laborious manual efforts needed to customize a DNN model that best fits all requirements, without the right service abstractions to support.

Inference execution optimization. For optimizing the inference task execution (either from a system or compiler perspective), the missing points of existing works have two folds. First, they do not harness the approximation nature of the DL functions for computation graph optimizations. Note that neural network training can be viewed as a function approximation process. A same semantic can be approximated by totally different mathematical expressions. Existing DNN optimization approaches never realize this point to propose the right primitives to focus on the "approximate" semantics. Second, they miss the first stage of inference execution optimization, namely from a cross-task perspective. An intrinsic nature of edge devices is that they typically operate on contextual data. Therefore, co-located devices and/or correlated contexts will lead to repeated or partially overlapped inference tasks, which is eventually manifested as computation redundancy across different applications and devices. Such computation redundancy is largely neglected by existing works.

Chapter 3

Mistify: DNN Porting Service for Edge Devices at Scale

AI applications powered by deep learning inference have become a mainstay on edge devices. As we mentioned earlier, by 2022, over 60% of the data locally generated by devices (IoT, sensors, mobile devices) will drive real-time intelligent decisions; 80% of the IoT and mobile devices shipped will have on-device AI capabilities [10, 11]. While these used to primarily offload related computation to the cloud, an increasing interest of deep learning inference workloads is to run them natively on the edge device to provide better interactive user experience and data privacy. This often necessitates fitting a model originally designed and trained on the cloud to edge devices with a diverse range of hardware capability and performance requirements.

However, model porting is a non-trivial process even for a single target. From an *algorithmic* perspective, the core techniques involved are called *model tailoring* in the machine learning literature. There are two steps, adapting the architecture of a pre-trained model to fit a new specification, followed by fine-tuning the new model parameters. Although there have been numerous model tailoring *algorithms* [100, 117, 102, 101] to tailor model architectures and refine the parameter values, the complete porting process actually requires 1) manually "embedding" the algorithms by correctly annotating the model definition and the source code of the training logic; and 2) carefully handle the high computation complexity (over 100× GPU hours) of adapting the annotated model structure and tuning the parameters with the right data.

Challenges. By various estimates, there will be over 50 billions mobile and IoT devices [16] with ever increasing diversity of hardware profiles, which creates a massive space of distinct profiles for a cloud model to tailor. Unfortunately, the current practice of porting cannot scale with the sheer size of this tailoring space. Therefore, app developers currently perform little platform-specific customization to the intractable target space [23], even though lack of customization results in suboptimal performance (Section 3.1). The overarching issue is the scalability of porting DNN models towards diverse edge settings. Corresponding to Section 1.2, the scalability issue can be further manifested by two challenges: the unscalable complexity of customizing a model to heterogeneous edge settings; and the laborious manual efforts and expertise needed to complete the end-to-end model porting (i.e. tailoring) process due to the lack of the right service abstraction.

Solution overview. In this work, therefore, we propose *collective adaptation* and other complementary algorithms that capture and eliminate the redundancy between numerous model structure adaptation processes to resolve the scalability challenge of fine-grained customization; and build *Mistify*, a service abstraction to automate and scale the porting process from a pre-trained model to a suite of compact variants tailored to diverse edge resource specifications (Section 3.2), outsourcing the expertise and efforts from the users.

From an *algorithmic* perspective, we propose *collective adaptation* algorithm to generate new models at scale via eliminating duplicate iterations; *privacy-aware knowledge distillation* to balance training data privacy and model accuracy; and downtime-free run-time model generation and switching, all incorporated in our solution.

From a *system* perspective, we propose new abstractions to decouple the model semantics from the execution characteristics. Instead of requiring the user to annotate the original model, *Mistify generates* model adaptation logic from the configuration file to correctly tailor to the resource budgets and performance requirements of each device (Section 3.3). Further, *Mistify* coordinates the implicitly correlated edge data in a privacy-aware manner to optimize tuning performance (Section 3.4). During the run time of the inference, *Mistify* incorporates a feedback mechanism to generate new models as needed to adapt to fluctuating application demands and resource availability (Section 3.5).

To summarize, we make the following three contributions: First, we quantify the scalability challenge of porting pre-trained DNN models to edge settings. This necessitates system support

to automate this process. Second, we design and implement *Mistify* as a service framework for automated porting at scale. *Mistify* achieves scalability with collective adaptation and improves model quality with privacy aware knowledge distillation and run-time model adaptation. Third, *Mistify* provides a clean interface to separate DNN model design and deployment. This could lower the bar to wider usage of on-device deep learning at the edge leveraging the abundant resource in the cloud.

3.1 Background and motivation

The lifecycle of a DNN model spans design and deployment, and the need for automating model porting arises from the complexity of the process. We discuss these in detail before outlining the challenges and solutions.

3.1.1 Current DNN lifecycle

The lifecycle a DNN encompasses at least three stages: model design, publishing, and deployment.

DNN model design. DNN models today are designed towards either of two goals: optimal inference quality, or minimal resource footprint.

The former is typically assumed for workloads run on the cloud. Given increasing computation power, cloud-centric models employ advanced neural network topologies, millions of parameters and floating-point operations (FLOPs) to achieve the *highest accuracy*. For example, BERT [7] and ResNeXt [118] have 340 and 829 million parameters respectively, hence extremely computation intensive.

The latter goal is geared towards resource-constrained edge devices, including IoT nodes, smartphones and tablets. The desirable models (e.g., MobileNet [94] and SqueezeNet [95]) are exceedingly compact, requiring only a few MBs for storage and affordable computing budget, ready to run across diverse device hardware. However, these DNNs sacrifice accuracy in exchange for super lightweight execution, aiming at *maximal deployment coverage*.

Once well trained, these models are published to public repositories for deployment.

DNN deployment at the edge. Many DL inference engines have been developed to serve DNN workloads on edge devices. They focus on deployment optimizations such as cross-platform com-

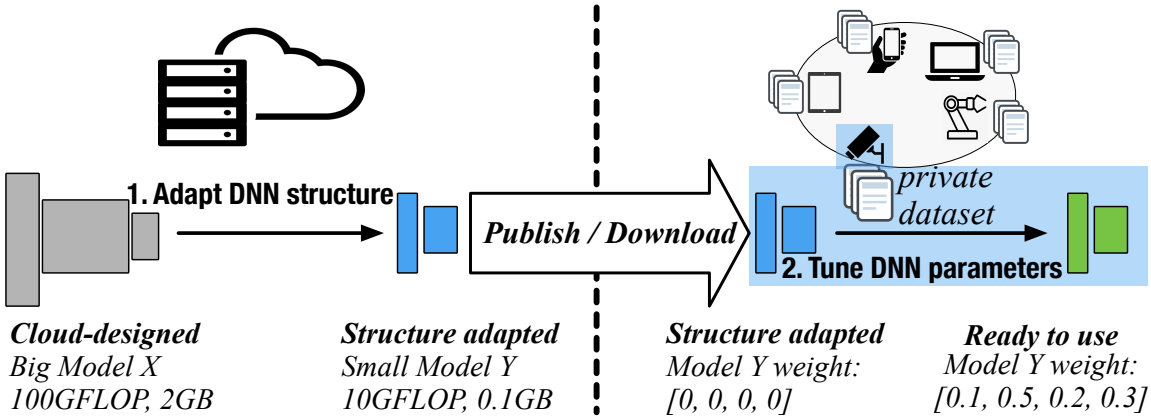


Figure 3.1: Steps to port a DNN model to an edge setting.

patibility, trimming executable size, and low-power operator kernels [18, 20]. Once a DNN model is loaded (e.g., from model repositories or custom URLs), these engines can execute the inference efficiently.

Transition from design to deployment. When a pre-trained model is ill-suited to a desirable deployment setting, it needs to be *tailored* to the new resource budget and performance goals. Illustrated in Figure 3.1, this requires *adapting* the model architecture (e.g.,by trimming network connections, skipping layers, pruning and quantizing parameters) and then *fine-tuning* (i.e., retraining) the parameters with local datasets. However, the end-to-end model porting process is complex. The source model needs to be correctly annotated to enable its architecture to be adapted for a particular setting. Fine-tuning also requires careful use of the training data to balance training quality (effective specialization without overfitting) and data privacy.

3.1.2 The complexity of porting DNN models

As more edge devices adopt on-device inference, porting cloud-based models to edge settings becomes increasingly complex, facing several challenges: (i) the range of model adaptation targets is huge as a result of the diversity in the hardware specification; (ii) the porting process involves several stages, each requiring coordination between multiple parties; (iii) runtime dynamics and new deployment settings necessitate frequent model re-adaptations.

Heterogeneous execution environment. Edge devices are incredibly diverse, ranging from embedded sensors, IoT devices, mobile phones/tablets, to edge servers, with a full spectrum of hardware

capability [23]. Table 1.1 lists the specification of some GPU and ASIC accelerators and processors, from high-end to low-end, widely employed at the edge for DNN-based workloads. For the same DNN inference workload, the completion times for low-end (e.g., nano) and high-end (e.g., 2080) devices can differ by orders of magnitude (e.g., 229 ms and 9.8 ms to run inference over ResNet).

Even when considering only smartphone platforms, to deploy a DL-based mobile app on App Store need to consider over *hundreds* of types of hardware devices, from high-end iPhone11 pro with dedicated neural processing units to 7-year-old Nexus 5 with orders of magnitude slower processor [24]. Meanwhile, for the same hardware under different battery conditions and dynamic latency requirements, the optimal DNN model also differs a lot which further increases the adaptation targets to be considered.

Meanwhile, state-of-the-art algorithms could adapt a DNN model towards a wide range of performance characteristics. For instance, different variants of EfficientNet for object classification differ by 8× in memory usage, 6× in latency, and 7% in absolute accuracy [22]. Similarly, different variants of BERT model for NLP tasks differ by 10× in speed and memory usage, and 15% in accuracy [28].

These numbers outline a massive design space to explore different tradeoff points between inference accuracy and latency, where a sub-optimal choice could incur up to 10% accuracy loss (e.g., when running EfficientNet-B0 unnecessarily on the latest iPhone model) or miss the latency requirement for real-time processing by over 100 ms (e.g., running ResNet on a low-end smartphone) [31].

Clearly, one size does not fit all, but nor would a few sizes only. Instead, it is desirable to tailor to each target at a fine granularity. For instance, EfficientNet-B4 (a popular model occupying a sweet spot of computation complexity and prediction accuracy) is suitable for Samsung S9, achieving 83% accuracy and 50 fps real-time response rate. However, using the same DNN on its immediate predecessor (S8) and successor (S10) would reduce the response rate by 14 fps for S8 and the accuracy by nearly 1% for S10. These are significant to the model designers where even 0.1% accuracy improvement merits tremendous effort (both intellectually and computationally) into model design and training. Given the ever increasing size of this adaptation space, it is impractical to either cover all plausible operation points with a few DNN models, or manually exhaust the entire space to customize the adaptation tradeoff for each possible individual edge setting.

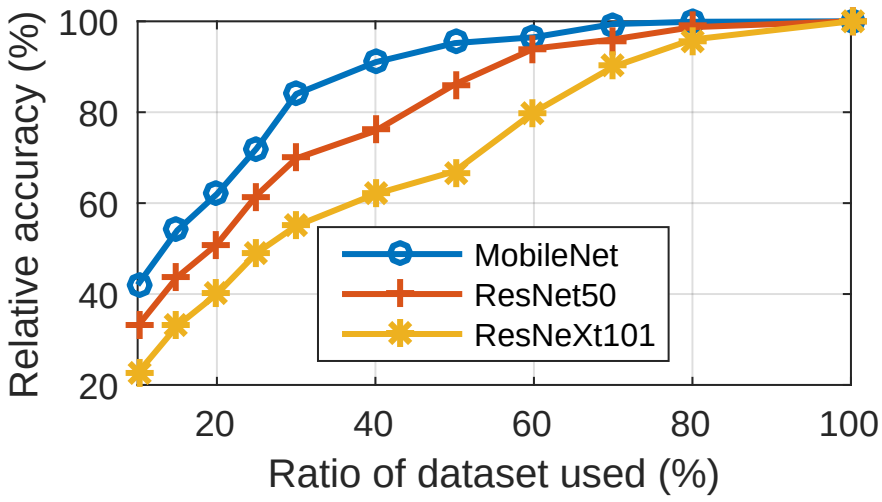


Figure 3.2: Training dataset size influences accuracy.

Multi-stage multi-party efforts. Tailoring a DNN model involves first adapting to the right model architecture, and then fine tuning the model parameters (Figure 3.1).

The first stage takes place where the original models are trained (i.e., in the *cloud*), with target specification from the edge device. The second stage increasingly takes place at the *edge* given the push for on-device inference and federated learning. Edge devices collect and maintain specialized data relevant to the local context for model training [41, 54] and local data are typically privacy sensitive [119]. However, smaller networks with less abundant datasets are well-known to be much harder to train, as it is easy to overfit the model with the training data such that the model may not generalize well to unseen test data [120, 112]. Thus, it is also preferable for the edge to take advantage of relevant datasets available elsewhere (e.g., in the cloud or on other devices) to enhance the training dataset and improve training quality. We illustrate this effect by training three DNNs (MobileNet, ResNet50, and ResNeXt101) multiple times, each time using a different subset of the Cifar100 dataset [121]. Figure 3.2 plots the relative test accuracy, compared to the default accuracy when the entire Cifar100 dataset is used for training. Even though all models are trained to converge to the same error values, the validation accuracy degrades by up to 80% when using less training data.

To sum up, both stages of model tailoring require coordination between the cloud and the edge, and resolving the conflict between data privacy and fine-tuning quality.

Fluctuating runtime characteristics. The runtime characteristics of deep learning inference tasks

is highly dynamic, reflected in two aspects.

First, the performance requirements, e.g., accuracy and response time, of an inference task change frequently. For instance, the accuracy requirements of a vision-based security surveillance workload are different during crucial and trivial moments, while the latency requirement fluctuates across peak and off-peak hours (e.g., daytime and night) [122, 41]. Further, FLOPS is sometimes an inaccurate proxy to statically estimate runtime latency [123]. Second, the resource availability (e.g., memory space, CPU cycles, accelerator quotas), varies on the edge device due to other workloads competing for the same resource. For instance, when an edge device launches or completes workloads, or adjusts the resource allocation of the containers that serve the DNN model for inference tasks, the perceived resource availability to any active workloads changes [29, 30].

The frequent changes in the performance requirements and resource available necessitate a mechanism to better serve individual combinations of the operation point, including a suite of models to switch dynamically, and asynchronously tailoring new ones as the demand warrants.

3.1.3 The need to automate DNN porting

Current practice. To tailor DNNs towards heterogeneous deployment settings, currently either the model designers should generate different DNNs to cater to each possible resource budget and performance goal, or the model users should prepare datasets, select and apply algorithms to tailor already published DNNs towards their custom settings. Either way, a source model needs to be manually annotated to incorporate a suitable adaptation algorithm and then fine-tuned.

Latest adaptation algorithms, such as AutoML [117], EfficientNet [22], and others [124, 100, 102, 101], all address target-specific adaptation *case by case* as an additional step in the *design* phase. They use different techniques (e.g., gradient-based, evolutionary, and recurrent neural network based) to revise the model architecture closer to the required resource and performance target with successive training iterations.

Problems with current porting practice. The overarching problem of existing efforts is they do not scale from a system perspective (e.g., hundreds of GPU hours for a single setting [125, 117]) and largely rely on manual efforts (e.g., thousands lines of code spread across source files [126]). Such manual tailoring process is not easily turned to a configuration style that is agnostic to the number of

cases because distinct structure adapting terms have to be added to different DNN models/layers and at specific positions, which makes it difficult and error-prone. Furthermore, it is infeasible for the model designers to prepare for all possible deployment settings, or for model users to be well versed in machine learning literature to build the right algorithm.

The need for an automated framework. The current model porting process implicitly couples DNN model design and deployment, even though they are conceptually separate stages, and incurs unnecessary complexity to both model designers and model users. This motivates adding a separate *model porting* stage to the model lifecycle, i.e., an intermediary to decouple design from deployment and automatically port pre-trained DNN models towards heterogeneous edge settings.

Mistify is therefore built as an intermediate framework to encapsulate diverse adaptation algorithms and address the end-to-end porting challenges outlined above, analogous to scheduler *frameworks* for distributed systems implementing scheduling *algorithms* and providing supporting services.

3.1.4 System requirements

To address the challenges above, an automated model porting framework should meet the following requirements.

Avoiding deeply embedded and unscalable manual code changes. Since existing model adaptation step is often coupled with the model design itself, a side effect is that relevant code changes are embedded deep into the model design code. Therefore, the system challenge is to simplify the code modifications needed to specify the adaptation target.

Mistify addresses this challenge in two steps (Section 3.3). First, we expose the right high-level abstractions of adaptation choices to users. This elevates per-model code edits (embedded in the particular script specifying the model) to framework level configuration parameter changes. Second, we parse the adaptation requirements from the configuration files and merge implicitly correlated model adaptation requests to reduce duplicate effort and improve scalability.

Cloud-edge multi-party coordination. To automate the two-stage model tailoring process with the best training outcome, the main challenge is to simultaneously ensure private data stay locally but parameter tuning can benefit from the data distributed across devices. We address this by estimating

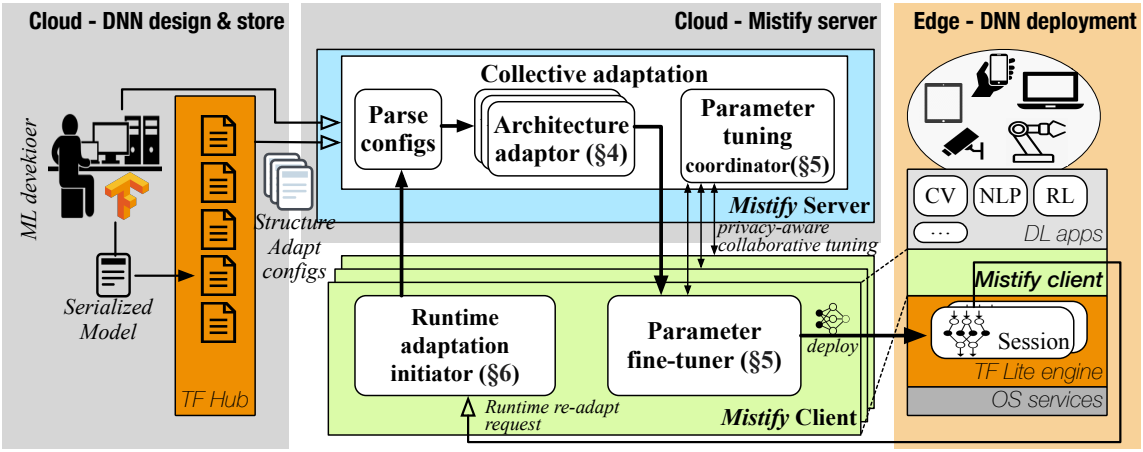


Figure 3.3: *Mistify* system architecture.

semantic correlation between data across devices without explicitly exchanging and examining the raw data. This way our system implicitly coordinates multiple devices running the same tailoring process to maximally "share" available training data in a privacy preserving fashion (Section 3.4).

Fast response to runtime dynamics. Fundamentally, the system challenge is to effectively handle the mismatch between a statically trained model and the dynamic execution environments during runtime. Specifically, this requires generating new models as needed and switching to them with minimal downtime. We address the challenge with a feedback mechanism between the model deployment points (e.g., edge devices) and the model tailoring point (e.g., a central server or cloudlet) to perform real-time DNN re-adaptation (Section 3.5).

3.2 *Mistify* demystified

The overarching goal for *Mistify* is two-fold: (i) *Mistify* should separate the model design and deployment stages with a clean interface; and (ii) *Mistify* should bridge the two stages with a runtime that automatically explores the design space at scale and generates models best suited to user-specified tradeoff points, hiding such complexity from both sides.

Therefore, *Mistify* is designed as an intermediate framework between DNN model design toolkits and deployment engines, as shown in Figure 3.3. The arrows across different shaded blocks show how *Mistify* interacts with model designs and users. *Mistify* exposes APIs to model users and inference engines to specify their *porting configurations*, either in a *batch* mode during initialization or in a *streaming* mode incrementally during runtime. Example configurations are shown in Figure 3.4. Such

configuration includes hardware profile (e.g., FLOPS/s, memory bandwidth), resource availability (e.g., slices of memory and CPU cycles), performance targets (e.g., inference accuracy and latency), the original model to be adapted, and the datasets for fine-tuning. The source models are fetched either from model repositories (e.g., TF-Hub) or developer specified custom URLs.

Mistify operates in two modes, a static mode (analogous to compile time) and a dynamic mode (analogous to run time). The static mode can handle either asynchronous (i.e., porting a previously designed model) or synchronous (i.e., connecting the porting stage directly to model design) model porting in an offline fashion, whereas the dynamic mode can handle runtime model adaptation. The static mode typically involves generating one model per device specification, but many specifications in one batch; In the dynamic mode, we aim to generate multiple model variants for the same hardware architecture, but with varying levels of resource consumption and inference result quality target.

The primary challenge of *Mistify* is therefore how to generate a large number of adapted DNN models with minimal computation and manual intervention. In general, our approach is *collective adaptation*, i.e., parsing adaptation goals and harnessing the implicit correlations among the goals to reduce unnecessary computation (Section 3.3). *Mistify* parses a collection of individual adaptation goals into a dependency tree with each node corresponding to a distinct goal, so that each goal is adapted only from its immediate parent via a desirable adaptation algorithm that is automatically chosen. Next, the adapted models are distributed to the endpoints, where the *Mistify* client runtime will prepare the deployment of the adapted model by fine-tuning the parameters (Section 3.4). Finally, the models start running on edge devices, and the *Mistify* client monitors the execution environment (e.g., resource availability and desirable performance goals). The *Mistify* client will trigger on-demand model re-adaptation asynchronously when the environment changes warrant a new model (Section 3.5).

Example deployment. The *Mistify* server can be deployed in the cloud by the DNN application developers, coupled with the model repository (e.g., TF-Hub), exposing APIs to the public. Alternately, the server can be maintained by the model users (e.g., edge device administrators) on their private cloud to serve local devices (e.g., IoT nodes). Such deployment assumptions align well with the common practices of on-device DL inference. [18, 4]. *Mistify* clients are simply deployed on the edge devices along with the native DL engine as an extension module.

Use cases. For DL function (e.g. Google lens) developers, the porting system will save the manual efforts of tweaking the DNN to fit each new model of smartphone. Instead, the developers simply need to expose an API for the user devices to describe their deployment settings and then generate models tailored to these settings. Take edge device owners as another case. When they want to deploy deep learning based functions on their devices, they can simply search for a well-performing pre-trained DNN model that fits their need, and then focus on their core business logic. The porting system will handle the complexity of turning the model into variants that match heterogeneous device specifications optimally.

***Mistify* server.** The *Mistify* server consists of two functional modules: an *architecture adaptor* and a *parameter tuning coordinator*. Once the *architecture adaptor* receives the original DNN model and the adaptation settings from model users and/or the *Mistify* client, it generates the adapted models and sends those to the corresponding clients (Section 3.3). The *parameter tuning coordinator* serves as the central point to coordinate the parameter fine-tuning processes among the *Mistify* clients (Section 3.4.2), whereas the actual tuning logic is executed on each client locally (Section 3.4.1).

***Mistify* client.** The *Mistify* client consists of a *runtime adaptation initiator*, a *parameter fine-tuner* and a *runtime performance monitor*. The *runtime adaptation initiator* intercepts the native DNN model loading path of the inference engine to automatically trigger model adaptation during initialization, and then listens for runtime re-adaptation requests. The *parameter fine-tuner* takes an adapted DNN model as the starting point, optimizes its parameters jointly based on the local (private) training data and the guidance from the correlated neighboring counterparts (coordinated by the *Mistify* server). This approach aims to overcome overfitting while maintaining data privacy. The *runtime monitor* tracks the current performance as well as resource availability. Once these profiles change significantly, it will trigger an online model switching as well as an offline re-adaptation request.

3.3 Scalable model architecture adaptation

Instead of requiring the user to manually annotate the source models, *Mistify* provides expressive configuration interfaces to specify adaptation goals and constraints (Section 3.3.1) and suitable abstractions to capture common algorithmic steps that meet these constraints (Section 3.3.2). To further scale to a large target space, *Mistify* merges adaptation instances to avoid duplicate efforts

```
[ // start of all configurations
    {
        "id": "1",
        "model": "Resnet",
        "dataset": {
            "train": "/path/to/train",
            "test": "/path/to/test"
        },
        "algorithm": {
            "name": "Morphnet",
            "config": {
                "threshold": 0.1,
                "init_reg_strength": 1e-9
            }
        },
        "adaptation_goal": {
            "latency": "30ms",
            "accuracy": 0.80,
            "FLOP": "5G",
            "num_of_params": "20M"
        }
    },
    // more configurations ...
    // ... more configurations,
    {
        "id": "9",
        "model": "Efficientnet",
        "dataset": {
            "train": "/path/to/train",
            "test": "/path/to/test"
        },
        "algorithm": {
            "name": "Chamnet",
            "config": {
                "init_population": 10,
                "crossover_rate": 0.7,
                "mutation": 0.08
            }
        },
        "adaptation_goal": {
            "latency": "30ms",
            "accuracy": 0.80,
            "FLOP": "5G",
            "num_of_params": "20M"
        }
    }
] // end of all configurations
```

Figure 3.4: Example porting configuration.

(Section 3.3.3) with *collective adaptation*.

3.3.1 Adaptation goal specification

An adaptation goal reflects the desirable inference performance given static and dynamic device conditions. Therefore, the user simply provides three sets of inputs: *hardware profile*, *resource availability*, and *performance targets*. Hardware profile includes compute power (GFLOP/s), memory bandwidth (GB/s), and quantization strategy (8/16/32 bits). Resource availability includes memory limit (GB), CPU/GPU shares, and GPU memory allocation. Performance target includes latency (s/task) and accuracy. These descriptions of an adaptation goal are consistent with most state-of-the-art adaptation algorithms. Note that our collective adaptation approach is not limited to aforementioned device and runtime factors. Custom finer-grained profiling libraries and tools can be incorporated by implementing the `Measure()` interface of the adaptation executor (Section 3.3.2). We leverage a JSON-like format to specify multiple goals in a single configuration file, which will be parsed automatically during adaptation.

We next formulate the *cost budgets* of a given DNN structure based on the specification of the adaptation goal provided by the user. In terms of computation cost, each layer contributes $C_{in} * C_{out} * S_{kernel} * S_{out}$ multiplications and additions. C_{in} and C_{out} denote the input and output channels; S_{kernel}

and S_{out} denote the convolution kernel and output size for Conv operations; for normal Matmul operations, S_{kernel} and S_{out} both equal 1 as they are equivalent to 1×1 convolution on 1×1 inputs. For memory cost, each layer contributes $C_{in} * C_{out} * S_{kernel}$ parameters (S_{kernel} is 1 for Matmul operations similarly). Combined with the quantization strategy, the total memory consumption of a NN layer can be calculated. For latency cost, we first calculate the previous two costs C_{comp} and C_{mem} respectively. Then, we leverage the hardware specifications (peak computation power and the memory bandwidth) to translate these costs into the latency cost as $C_{mem}/mem\_bandwidth + C_{comp}/comp\_power$.

We can further incorporate an energy consumption budget in the specification, but we leave this to future work as the energy budget is more hardware specific and not as generic as the above three cost items.

3.3.2 Adaptation Executor

Common workflow of DNN adaptation algorithms. State-of-the-art DNN adaptation algorithms follow a similar process. They take a source DNN model and adaptation goals, and search for variants of the base model architecture that fits each scenario. The search explores a high-dimensional vector space, where each hyperparameter of the DNN (e.g., #layers, #filters, kernel size, and quantization) corresponds to a specific dimension. The search process runs iteratively until the costs of the current model optimally match the adaptation goal. Common search strategies include evolutionary search [127, 102], gradient descent [124, 100], RNN-based search [123, 22], and so on.

Adaptation executor. In light of this common process, we design an abstraction, an `Adaptation Executor`, that collects all adaptation settings as a closure, and exposes three function APIs (`Init()`, `Measure()`, and `Adjust()`). `Init()` loads the adaptation settings, the model, and the constraints, and then instantiates the executor that runs the chosen adaptation algorithm (default or user specified). This function is also responsible for identifying the right places to insert the training loss terms in the entire training workflow. `Measure()` is called after each adaptation iteration to determine the *costs* of the current model in specific metrics (e.g., model size, accuracy, or custom costs profiled by user-specified functions). `Adjust()` will then tune the control knobs of the algorithms (e.g., dimension-wise step size, threshold, or learning rate) to steer the cost refinements towards the adaptation goals in an optimal direction. These APIs abstract away the inner workings of

heterogeneous adaptation algorithms in a universal approach, obviating the need to directly annotate the models to embed the adaptation logic. A new adaptation algorithm can interface with *Mistify* by implementing the above APIs, and the user can specify the preferred algorithm in the configuration.

Case study: running MorphNet algorithm via adaptation executor. The vanilla DNN training starts with defining an accuracy loss function ($\mathscr{L}_{output}$) based on the difference between the model outputs and the ground-truth labels. The loss is back-propagated to each layer i (with parameter θ_i) as $\mathscr{L}_i(\theta_i)$. Each layer calculates the gradient of the loss, and optimizes the parameters (θ_i) iteratively by minimizing the loss via gradient descent. Namely, $\theta_i^{new} = \theta_i^{old} - \eta \cdot \nabla_{\theta_i} \mathscr{L}_i(\theta_i)$.

MorphNet (a recent gradient based search algorithm [100]) converts the resource costs of DNNs as additional penalty terms of the loss function. This way, the DNN architecture is iteratively optimized via gradient descent along with the vanilla DNN training. For instance, the "useless" weight parameters will be suppressed to zero and trimmed during training when minimizing the overall loss, as they do not contribute to reducing the accuracy loss but increasing the architectural loss. The adaptation process completes when each structure-related cost (e.g., number of FLOPs) satisfies the corresponding constraint, or when the pre-defined maximal running time is reached for the non-converged cases.

However, to actually adapt a model with MorphNet, a user needs to (i) pick the penalty term representation for each operator included in the DNN (e.g., *Gamma* regularizer for BatchNorm op), (ii) specify the input and output operator of the model, (iii) instantiate the penalty term with the arguments such as trimming threshold and learning rate, (iv) add the penalty term into the overall training loss, and (v) set the cost monitoring and termination conditions. All these steps are repeated for each adaptation target, and require modifying the source code of the DNN model definition and training scripts. In contrast, with the clean APIs and the adaptation executor abstraction of *Mistify* the users only need to specify the high-level configurations (e.g., adaptation algorithm, trimming threshold) and all adaptation goals (e.g., memory usage, number of FLOPs) collectively in a single JSON file.

To encapsulate the MorphNet algorithm in an *Mistify* adaptation executor, we implement the APIs as follows. For `Init()`, we additionally implement the operations of deriving the positions (e.g., Conv layers) to add architectural loss terms, essentially by first finding the input layers, and

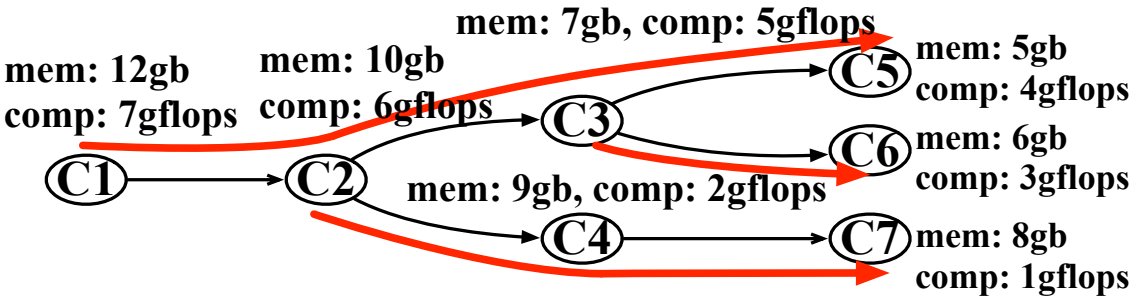


Figure 3.5: Configuration tree example.

then traversing the whole DNN graph topologically along the dependencies to insert the loss terms to corresponding layers until the outputs. `Measure()` simply calculates the resource and performance costs of a given DNN independent of the adaptation algorithms. For `Adjust()`, we implement the logic of setting the learning rate of the loss term corresponding to each resource and/or performance constraint. The implementation of these APIs is lightweight (Section 3.6).

3.3.3 Collective adaptation

We observe that multiple adaptation goals often share similar initial steps or training iterations. Handling each adaptation goal independently is very inefficient when deploying a model to a range of devices. Therefore, *Mistify* provides a mechanism to "merge" adaptation goals to avoid duplicating the same action. We parse the adaptation goals into an n-ary tree structure following certain rules. Goals along a branch are adapted one-by-one serially in a single pass. We also design a mechanism to navigate the adaptation direction, to meet the constraints (e.g., memory and computation) of a single goal simultaneously.

Adaptation goals compilation. As mentioned above, each single goal consists of several resource and performance constraints, and can be abstracted as a multi-dimensional vector. Then, combining with the hardware specifications we build a partial order of all goal vectors (from the least demanding to the most) according to their constraints. Figure 3.5 shows an example of 7 goals ($C1$ to $C7$), each with two hardware resource constraints (memory usage *mem* and computation complexity *comp*). Goals C_i and C_j have a strict order $C_i < C_j$ only when $C_i.mem < C_j.mem$ and $C_i.comp < C_j.comp$.

Following the partial ordering relations between goals, we further generate a tree structure, with each node representing a goal, and each edge leading to one of its immediate more demanding goals.

Hence, each branch of the tree corresponds to an independent adaptation path (marked with a red arrow in Figure 3.5). Along each path, every two goals are consistently ordered on all constraints. This ensures that they can be collectively adapted in one pass without conflicts. Note that the accuracy does not strictly increase over the path. When *Mistify* starts to traverse a path from one point to the next, the accuracy will first drop to a certain level, and then climb back while the training continues. Meanwhile, the resource profiles will move to the desired position.

Given the tree structure, we first uniformly expand the architecture of the original DNN so that, for each constraint dimension, its actual cost value is larger than that of the root node (the least demanding goal). Then, starting from the root, we run the encapsulated adaptation algorithm to trim the DNN architecture iteratively along each adaptation path. Every time a goal is satisfied, the corresponding version of DNN is stored as a checkpoint for future use.

Note that even though the mapping between partially ordered goals to a tree structure is usually not unique, we find that there is only marginal difference in the overall adaptation time between different mappings. Hence, it is not worth optimizing the mapping given it is NP-hard.

Structure loss scheduling. When executing the adaptation along a path, an essential question is how to control the adaptation towards the optimal direction (via `Adjust()`), i.e., how to meet multiple desirable constraints simultaneously. Although some existing works achieve this by forking a new adaptation schedule for each change of the adaptation "direction" [128], they are not scalable to large number of adaptation goals and fine-grained continuous controls. For instance, the common practice of training a single DNN model involves adapting learning rate periodically by at least over 10 times. Further multiplying this with the total number of adaptation goals in the configuration tree, the overall trials will easily exceed tens or hundreds of thousand trials, which is too heavyweight for almost all training platforms.

To address this, we adjust the control knobs based on the weighted combination of the corresponding architecture losses. The overall DNN loss function is the sum of the normal loss (denoted as $\mathscr{L}$) and a set of architecture losses corresponding to each constraint $\{\mathscr{G}_i\}$. For each $\mathscr{G}_i$, their control knob (e.g., learning rate for gradient-based algorithms) can be viewed as a weight parameter w_i, which leads the overall loss $\mathscr{L}_{all} = \mathscr{L} + \sum_i w_i \cdot \mathscr{G}_i$. Now, to adjust the adaptation "direction" towards a specific constraint f_i, we only need to increase the weight w_i of the loss $\mathscr{G}_i$.

Initially, all the weights w_i are equal and sum to 1. Suppose for the loss of constraint f_i we have the initial value $\mathscr{G}_i^{(0)}$ and the target value $\mathscr{G}_i^{(+)}$. Then, for every k training iterations (empirically set to 200), we reschedule the weights once. The n-th iteration weight $w_i^{(n)}$ is calculated as $Share_i^{(n)} / \sum_i Share_i^{(n)}$, where $Share_i^{(n)} = \frac{\mathscr{G}_i^{(+)} - \mathscr{G}_i^{(n-1)}}{\mathscr{G}_i^{(+)} - \mathscr{G}_i^{(0)}}$. In essence, we proportionally assign the next values of the weights $w_i^{(n)}$ according to how far the corresponding loss values $w_i^{(n-1)}$ deviate from the targets, and finally normalize these weights.

3.4 Privacy-aware fine-tuning at the edge

After adjusting the model architecture with respect to the resource and performance constraints, the weight parameters need to be fine-tuned before actual deployment. If all training data are collected and stored in the cloud, parameter tuning simply follows the standard training process for the adapted DNN. Instead, we will consider how to fine-tune parameters when edge devices collect and store their private data locally. The challenge arises when specializing the DNNs only using the local contexts of edge devices.

Recall (Section 3.1.2) that DNNs are hard to train with a small dataset, usually the case for individual edge devices, and can easily overfit. On the other hand, the data local to each device is often more relevant but private, making it difficult or infeasible to aggregate the data from different end devices into a larger dataset for centralized training. Therefore we need to balance protecting edge data privacy and training quality (in terms of how well individual models generalize). While many existing works (e.g., Federated learning [54] and others [129, 130, 131]) address decentralized private DNN training, they assume different endpoints train the *same DNN structure* with different local datasets. The situation is different for *Mistify*, where the models on different devices have *different model structures* to meet specific adaptation goals.

Knowledge distillation (KD). To tackle the aforementioned dilemma, we need a mechanism for DNN "knowledge" sharing between distinct peer models and without explicitly exchanging private data between devices. Fortunately, *mutual knowledge distillation* [113, 115] comes to the rescue. When training a DNN model (M_1, the *student* model) from scratch, leveraging additional help from another similar but independently trained model (M_2, the *peer teacher* model) can significantly improve the validation accuracy of M_1.

Specifically, the optimization of parameter θ_i follows:

$$\theta_i = \theta_i - \eta \nabla_{\theta_i} \{\phi(y, M_1(x)) + \varphi(M_2(x), M_1(x))\}$$

where ∇_θ denotes taking derivatives with respect to the variable θ, ϕ and φ denote the loss functions (e.g., cross-entropy) respectively defined for the ground-truth labels and the teacher model M_2's outputs, and η denotes the learning rate as usual. The corresponding parameter values in M_2 are incorporated as added constraints. This way, the student model receives extra supervision from the teacher model during training, beyond optimizing for conventional learning objectives like the cross-entropy loss subject to the ground-truth training labels. The idea behind the formulation is to take advantage of the extra supervision provided by the teacher model (M_2) while training the student model (M_1), beyond optimizing for conventional learning objectives, such as the cross-entropy loss subject to the ground-truth training labels. *Mistify* builds on the idea to coordinate parameter tuning between the clients and the server in a privacy aware manner.

3.4.1 Client: KD-enhanced parameter tuning

Observe that a DNN trained locally on an edge device embeds the "knowledge" extracted from the private local data. Therefore, to take full advantage of the edge data distributed across devices without exchanging the private data, our algorithm instead shares the DNN models trained independently on each device. The ensemble of DNNs from other devices serves as the "teacher" to guide the current device's model just like mutual knowledge distillation. In this way, the "knowledge" extracted from different datasets are shared and the data privacy is preserved.

Our algorithm proceeds as follows.

(i) Each participating endpoint device (E_i) first tunes their adapted version of DNN model (M_i) with locally available training data until convergence.

(ii) Each endpoint sends its current model along with its loss and accuracy statistics to the central coordinator and waits for a response, namely a set of models (M_1 to M_n) trained on the other devices. An operator is added over the n outputs of these models (M_1 to M_n), taking their average as the final output of the model ensemble.

(iii) KD-enhanced tuning is then invoked to optimize the parameters θ_i of model M_i:

$$\theta_i = \theta_i - \eta \nabla_{\theta_i} \{\phi(y, M_i(x)) + \varphi(\frac{1}{n-1} \sum_{j \neq i} M_j(x), M_i(x))\}$$

Namely, the outputs of each local model M_i are compared with both the ground-truth labels y and the outputs of the assembled teacher model to calculate loss. We follow similar hyperparameter settings as in [112], using cross-entropy loss for ϕ and Kullback-Leibler (KL) divergence [132] for φ to measure the distance between the teacher and local models, and a default value 0.001 for η.

(iv) Loop over steps (i) to (iii) until the model finally converges. Noticeably, to improve generalization and avoid being skewed by some poor performing models, we randomly skip $max(n/10, 1)$ of the models used for each round of KD-enhanced tuning in step (iii).

Privacy-aware tuning. Although less privacy-sensitive than the training data, DNN models can still leak the information from the private training data. To overcome this privacy leak challenge, we can add noise to the fine-tuning process to achieve differential privacy [133, 134]. The noise can be added to either the training data, or directly the model parameters to be sent to the *Mistify* server. However, the latter provides less privacy protection, easier to "denoise", and does not provide fine-grained control easily.

Therefore, we augment the algorithm above with an optional step after (i). Specifically, we add Laplacian noise to the local training data, and train the model (M_i) for additional epochs until convergence. Then, this noisy model (M_i') is sent to the central coordinator in step (ii). This provides differential privacy to the model parameters and reduces information leakage from the private data. The level of noise added is chosen empirically according to existing privacy-preserving machine learning practice (e.g., PATE [133] and Myelin [134]) with the same level of privacy loss preference (e.g., $\varepsilon < 5$).

Note that *Mistify* is amenable to this differentially private approach by design. As *Mistify* aims to scale to a large batch of end devices (hundreds of more), potentially there is a large number of peer models to draw from during the intermediate steps. Therefore, even though the individual noisy intermediate model (M_i') is less accurate than its noiseless counterpart, the accuracy loss is compensated for by the ensemble of other peer models [135].

3.4.2 Server: client model coordination

One particular concern of our aforementioned algorithm is whether the models used for KD-enhanced tuning indeed add knowledge rather than noise. This is supported by the widely existing spatio and temporal correlation between datasets from nearby edge devices [38, 136, 39]. Further, it is also proposed that given datasets sufficiently similar in their semantic contexts (e.g., types of objects, hidden feature occurrence frequencies) as the training data, the trained models perform semantically equivalent functionality and can provably generalize to achieving the same capability [137, 138, 139].

In practice, we use commonly used spatio-temporal hints (e.g., location, time, view angle) sent by each client along with their models as a coarse-grained mechanism to estimate the correlation between datasets. There are myriad alternative lightweight approaches to measure dataset similarity without piece-wise comparison of the actual raw data (e.g., by calculating dataset feature summaries [140, 141]). They are easily pluggable into *Mistify* by implementing the corresponding APIs. Regardless of the exact metrics used to measure correlation, they are represented as multi-dimensional vectors. The central coordinator located on the *Mistify* server maintains a Locality-Sensitive Hashing (LSH [142]) structure to index these vectors (representing edge endpoint contexts), which guarantees sublinear complexity for query and insertion [143]. When an edge endpoint needs available models to fine-tune its own parameters, it will select the "nearest" (most correlated) models, based on the aforementioned methods.

3.5 Runtime model adaptation

Existing algorithms and libraries only port DNN models statically in a batch mode. Instead, *Mistify* further extends DNN porting with a runtime streaming mode, where the client actively monitors runtime changes of the resource and performance constraints and generates new models to such dynamics. *Mistify* does not aim to address general runtime adaptation issues, such as resource allocation and job scheduling. Nor does *Mistify* specifically deals with switching different *existing* models during runtime [19, 144]. These orthogonal decisions are left to the operating system scheduler or the regular inference serving environment. In contrast, the runtime of *Mistify* is invoked to generate *new* models when existing models are inadequate. The adaptation mechanism involves two paths, foreground and background.

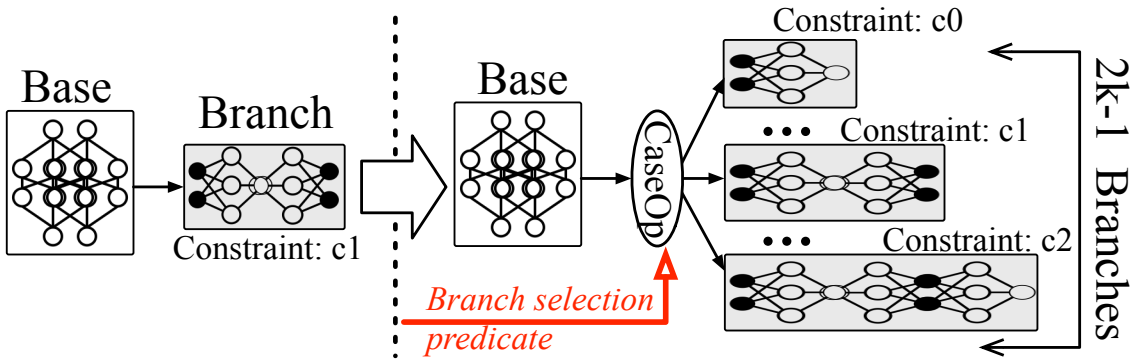


Figure 3.6: Multi-branch model construction.

3.5.1 Constructing a multi-branch model.

To support on-the-fly adaptation to fluctuating resource constraints, each DNN model is constructed in a multi-branch form during the architecture adaptation process (Section 3.3). Figure 3.6 illustrates the process. First, the aforementioned adaptation algorithm is triggered as usual until the constraints specified in the configuration are satisfied. Now, besides continuing to adapt to other configurations, a new adaptation thread is spawned. This thread separately adapts the current DNN into a k-branch DNN. For instance, a 5-branch DNN is built by freezing the first few layers and adapting the remaining layers towards 5 different configurations, whose resource budgets range from $\frac{1}{3}$ of to $3\times$ times that of the original DNN model. The branches share the same base, achieve the same inference task, but satisfy different resource budget and performance goals. In practice, k is set based on the observation of the typical fluctuations of the resource availability and performance targets. After fine-tuning the parameters of the multi-branch DNN, we add a case conditional operator (e.g., `tf.case` for Tensorflow) between the base and different branches.

Foreground path: downtime free branch switching. Foreground path is tightly coupled with the user-facing inference serving logic, performing real-time adjustments of the current DNN model based on the dynamic constraints. To achieve this, *Mistify* picks a branch from the multi-branch DNN with the closest resource and performance profile. The branch switching is done on-the-fly by setting the corresponding value of the conditional variable (red arrow in Figure 3.6) of the case operator in the DNN, saving additional overhead such as allocating memory and preparing runtime resources. This guarantees on-the-fly adjustment of the model resource consumption and performance statistics without additional overhead such as reloading computation graph from files, allocating memory,

and preparing runtime executing resources. For instance, when the amount of resource available to an inference workload shrinks by 45%, the foreground path will first select the branch with the closest constraints (e.g., resource budget tightened to half) to run the subsequent tasks. Meanwhile, the process to generate a new model tailored to the new resource constraints will be invoked in the background, to eventually replace the current model.

3.5.2 Background path

Meanwhile, in the background, the *Mistify* client will send the new adaptation configuration to *Mistify* server, where it is compared with existing ones in terms of their resource constraints, based on the partial ordering explained in Section 3.3, in order to retrieve the immediate predecessor of this incoming config. Then, the new DNN model is incrementally adapted from the corresponding "predecessor" DNN, until the constraints of the new configuration are met. This avoids redundant adaptation iterations between successive model adaptation instances and speeds up the process of generating a new model. The new model will be sent to the client for subsequent processing.

3.5.3 Discussion

Admittedly, memory consumption as a resource constraint is not adjustable when achieving downtime-free switching in foreground path. Instead, it consumes additional memory to hold multiple branches. However, we consistently observe that among different resource and performance constraints, memory space is in most cases the most loose one. Further, dynamic memory allocation (enabled by many platforms [86]) further alleviates the concerns for memory.

Meanwhile, as the bottom line, we provide a config parameter that controls if the downtime free branch switching in foreground path is activated. When deactivated, the immediate task after branch switching will wait the reloading and resource allocation of the DNN model (with a different branch) to finish, before running. Such process makes sense for those cases with scarce memory space and a stable runtime environment.

3.6 Implementation

We build *Mistify* over TensorFlow (TF) 1.13 [145] (Figure 3.3) and plan to open-source the codebase. The server side consists of around 6K lines of Python code for the model architecture adaptor and parameter tuning coordinator. The client side consists of around 2.5K lines of Python code for the parameter fine-tuner and the runtime monitor. The former is a library extension of the TF core, whereas the latter runs as a runtime daemon of the inference serving engine (e.g., TF-serving or TF-Lite). The client and the server communicate via a lightweight RPC library using ZeroMQ [146] and ProtoBuf [147].

Interfacing with the native environment. Recall that *Mistify* can be activated at two stages (Section 3.2), when initializing inference serving or during the run time. For the former, the function `tfhub.load()` is intercepted to trigger the model porting process (when fed the special argument). For the latter case, the *Mistify* runtime monitor is by default registered with the live `Session` of TF serving engine to collect runtime statistics (`tf.RunMetadata`), and to invoke the *Mistify* client to initiate the re-adaptation process on demand. The foreground branch switching is implemented by assigning a suitable value to the predicate variable of `tf.case` op.

Encapsulating adaptation algorithms. *Mistify* implements wrappers over two representative, state-of-the-art adaptation algorithms, MorphNet [100] (using sparsifying regularizers) and ChamNet [102] (using evolutionary algorithms). These are representatives of common categories of techniques. Adding new adaptation algorithms to *Mistify* is fairly easy, following the process outlined in the MorphNet case study in Section 3.3.2. Each wrapper implementation around these algorithms for *Mistify* requires around 100 lines of code (LoC), which is fairly modest compared to the thousands of LoC in the original codebases of these algorithms.

3.7 Evaluation

Hardware setup. Following Figure 3.3, a Linux server with 8-core 2.1 GHz Intel Xeon CPU, and NVIDIA 2070 GPU acts as the server side of *Mistify*; For the client-side operations of *Mistify*, we use devices with a low-end NVIDIA P600 GPU, a Google Edge TPU [25], and a Samsung S9 smartphone respectively as representatives of diverse edge hardware.

Application benchmarks. Computer Vision (CV) and Natural Language Processing (NLP) tasks currently dominate deep learning use scenarios, accounting, for example, for 95% of the DL workloads in the Google data centers [74]. We select one workload each, *Object Recognition* and *Question & Answering* corresponding to the two application categories, as representative benchmarks. While there are numerous other CV and NLP applications, for example, *scene segmentation* for CV, *machine translation* for NLP, these are based on DNN models derived from the same base structures as those used for our benchmarks (For example, ResNet blocks for object recognition, detection, and segmentation; Transformer blocks for Q&A, named entity recognition, sentiment analysis). Therefore, the results obtained for our benchmarks are representive of a wide range of scenarios.

Specifically, we select three state-of-the-art DNNs, MobileNet [94], ResNet50 [148], and ResNeXt101 [118], with increasing computation complexity (0.5 to 16 GFLOPs), parameter size (16 to 320 MB), and accuracy (68% to 79%) for object recognition. MobileNet is originally designed for mobile devices, whereas the other two mostly run in the cloud. For Q&A, the input is a question along with a context paragraph containing the answer to the question. A popular "accuracy" metric for this is the Exact Match (EM) score, i.e., the answer that exactly matches the question means a correct inference result, otherwise wrong. We prepare two DNNs, for the previous-generation and current state of the art, BiDAF [149], and BERT [7]. The former is lightweight but task-specific (customized for Q&A) (10$\times$ MB), whereas BERT is much larger, generically supporting various downstream tasks. Hereby, we use these DNNs with different specifications to evaluate if *Mistify* could successfully tailor them into different execution settings by balancing the accuracy and resource consumption (and correspondingly the latency).

Datasets. We use domain specific standard datasets to adapt network architectures, fine-tune their parameters, and validate their performance. Specifically, ImageNet [34] and Cifar100 [121] are used for object recognition, whereas SQuADv1.1 [150] is used for Q&A.

3.7.1 Collective Architecture adaptation

Collective adaptation time. We generate 128 different adaptation configurations based on four DNNs (MobileNet, ResNet50, ResNeXt101, and BERT). Among these, the least and most demanding configurations respectively constrain the adapted DNNs to 2$\times$ and 0.5$\times$ the default DNN memory

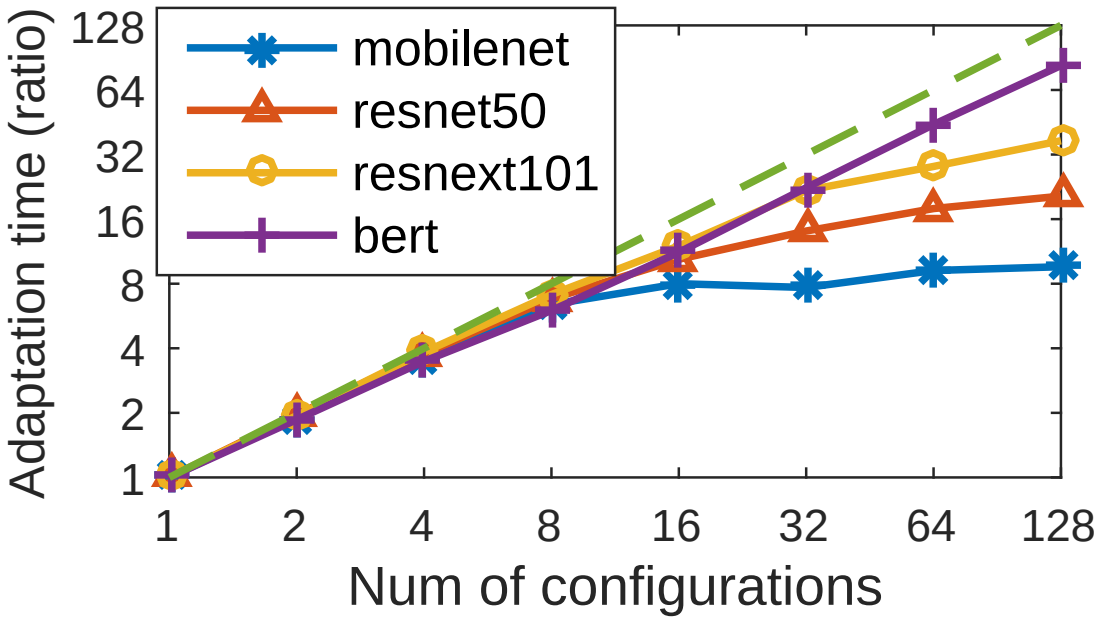


Figure 3.7: Completion time comparison for adapting a DNN from 0.5× to 2× resource consumption.

usage and computation complexity. Then, we select different subsets of these 128 configurations, adapt all of them with and without *Mistify*, and compare their overall time consumption to evaluate our collective adaptation approach (Section 3.3.3). Figure 3.7 shows the relative time needed without over with *Mistify*. *Mistify* accelerates the overall adaptation time almost linearly with the number of configurations when it is less than 10, consistently achieving around 10x acceleration even for DNNs as small as MobileNet. For large DNNs, such as BERT, that are structurally more amenable to adaptation (i.e., easier to prune a subset of the network without affecting validation accuracy), the acceleration scales well with over 100 configs.

Adaptation quality. Then, we examine the quality of the DNNs collectively adapted by *Mistify* or adapted per-config by default. Table 3.1 shows two rows for each network, corresponding to 4× compression and expansion with respect to the complexity and memory consumption of the original DNN. This spans the range from low- to high-end hardware [23]. For instance, the respective inference time of the compressed and expanded ResNet50, running on a Google Nexus 5 (low-end 2013 model) and a Samsung Galaxy 10 (high-end 2019 model), are both around 30 ms, low enough for practical usage. "Accuracy" corresponds to the EM score (exactly matching the ground-truth answer) for NLP. To avoid being affected by the parameter tuning quality, all adapted DNNs are trained with the whole datasets, and without considering any device-specific constraints. We see that

Table 3.1: Accuracy of Collectively Adapted Models (*Mistify*) vs Individually Adapted Models (Per-case)

DNN	Per-case (%)	*Mistify* (%)	Relative diff (%)
MobileNet	55.8	54.7	-2.0%
	69.4	69.5	+0.1%
ResNet50	68.2	68.0	-0.3%
	72.9	72.5	-0.6%
ResNeXt101	74.0	74.3	+0.4%
	77.6	77.9	+0.3%
BERT	71.4	70.6	-1.2%
	79.1	78.8	-0.4%

Mistify's adaptation algorithm achieves almost equivalent accuracy compared to the per-case default algorithm, with less than 0.5% accuracy loss for most cases and only 1% for the worst scenario (e.g., when adaptation configurations are incompatible with total ordering, causing the overall adaptation path to detour substantially), within the typical range of accuracy loss in exchange for resource efficiency [151].

3.7.2 Parameter tuning

We use a more specialized dataset Cifar100 to evaluate parameter tuning on the edge. The whole dataset is partitioned into subsets, as the local data of each edge device.

Convergence speed and quality. We compare the convergence speed and test accuracy for three different networks (MobileNet, ResNet50, and ResNeXt101), with and without *Mistify* support for parameter tuning (Section 3.4.1). Figure 3.8 shows that even without additional data, KD-enhanced parameter tuning (solid lines) already achieves over 3× faster convergence as well as better accuracy.

Scalability. We assess the scalability of the parameter tuning algorithm (Section 3.4.1) in terms of the ratio of communication time over training time, under different network bandwidth settings. We consider two extreme cases, MobileNet (very compact) and BERT (very sophisticated). In Figure 3.9 and Figure 3.10, each line corresponds to a specific network bandwidth in MB/s. When the network bandwidth is over 5 MB/s, our algorithm is consistently scalable, communication merely taking less than 15% of the time relative to training. For MobileNet case, even 1 MB/s narrow bandwidth could fully support the algorithm. Note that federated learning is multiple device collaboratively train one

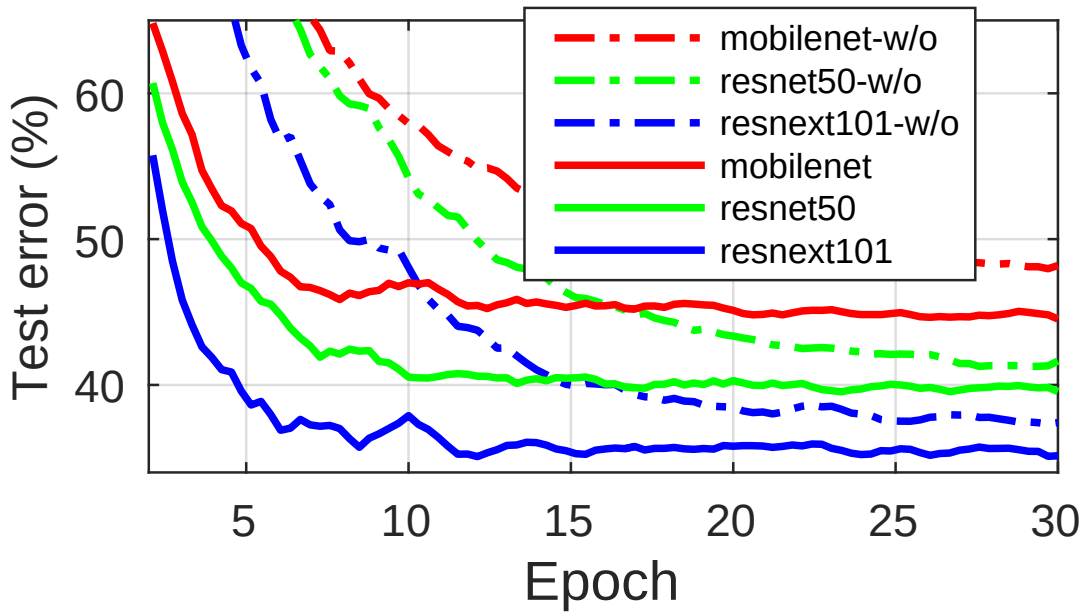


Figure 3.8: Comparison of convergence speed and performance for default approach and with *Mistify* support.

single model, whereas for our case, each device fine-tune their own adapted model, just sharing parameters as "knowledge" to enhance the training quality, so they are not directly comparable. Further, the lines almost flatten beyond three neighbor networks, so using more neighbor networks for our tuning does not impact scalability.

Accuracy of parameter tuning. We randomly partition Cifar100 and SQuAD each into 5 subsets, each used by an edge device for local training. Then, we compare the fine-tuning accuracy using different approaches. Table 3.2 shows that knowledge distillation (KD) improves parameter tuning accuracy by 40% over local training alone. Compared to the ideal distillation case where an exceptionally accurate teacher network is available (a pre-trained, cloud version), the ensemble of 4 peer networks achieves within 10% of optimal KD, despite using half the training data and meanwhile adding differential privacy to the model parameters.

3.7.3 Runtime overhead of *Mistify*

Foreground path on *Mistify* client. To switch to another network in response to the runtime dynamics (Section 3.5.1), there are two types of overhead. First, additional space is needed to store other networks in memory and switch to them seamlessly. Second, loading the new DNN

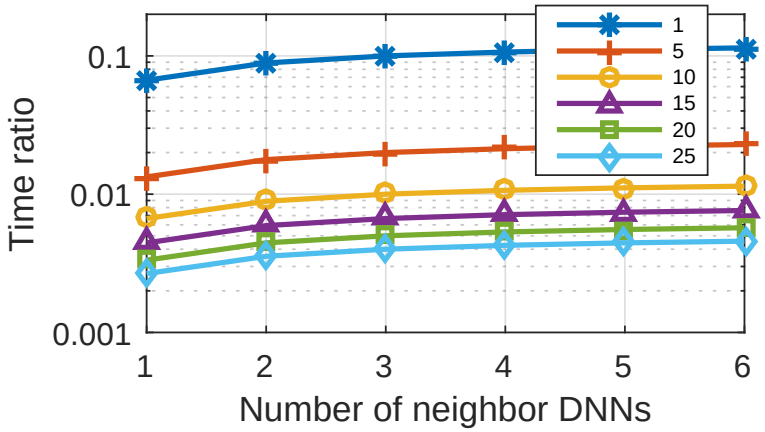


Figure 3.9: The ratio of communication time over training time, reflecting the algorithm scalability for MobileNet (compact model).

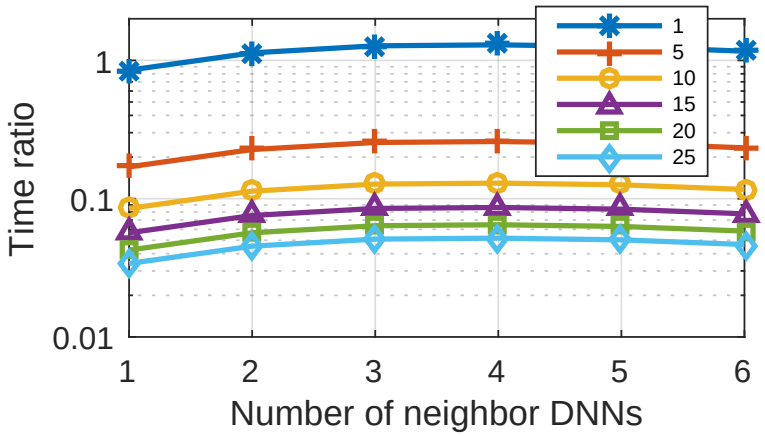


Figure 3.10: The ratio of communication time over training time, reflecting the algorithm scalability for BERT (huge model).

Table 3.2: The accuracy of tuning parameters with *Mistify*.

Scenario	DNNs		
	MobileNet (%)	ResNet50 (%)	BERT (%)
Local training	39.7	43.9	22.5
KD	66.4	75.3	78.8
1-peer tuning	53.8	61.5	51.9
2-peer tuning	58.1	67.2	65.6
4-peer tuning	59.8	69.0	71.8

and preparing the runtime execution resources on-demand saves memory but needs downtime. Specifically, the model size corresponds to the runtime memory consumption of the DNN, instead of the size of the serialized model file. Table 3.3 illustrates the trends of additional memory consumption or time consumption for different networks. The suffix "-kb" means adding k branches to the adapted

Table 3.3: Additional number of parameters and network switching time overhead.

DNN	Additional/original params (M)	Time (s)
MobileNet-b3	2.67/3.43	2.11
MobileNet-b5	4.57/3.43	
ResNet50-b3	18.2/23.9	3.34
ResNet50-b5	31.7/23.9	
ResNeXt101-b3	33.9/44.3	4.19
ResNeXt101-b5	57.6/44.3	
BERT-b3	92.4/110	21.84
BERT-b5	171/110	

Table 3.4: Latency (ms) for building config tree.

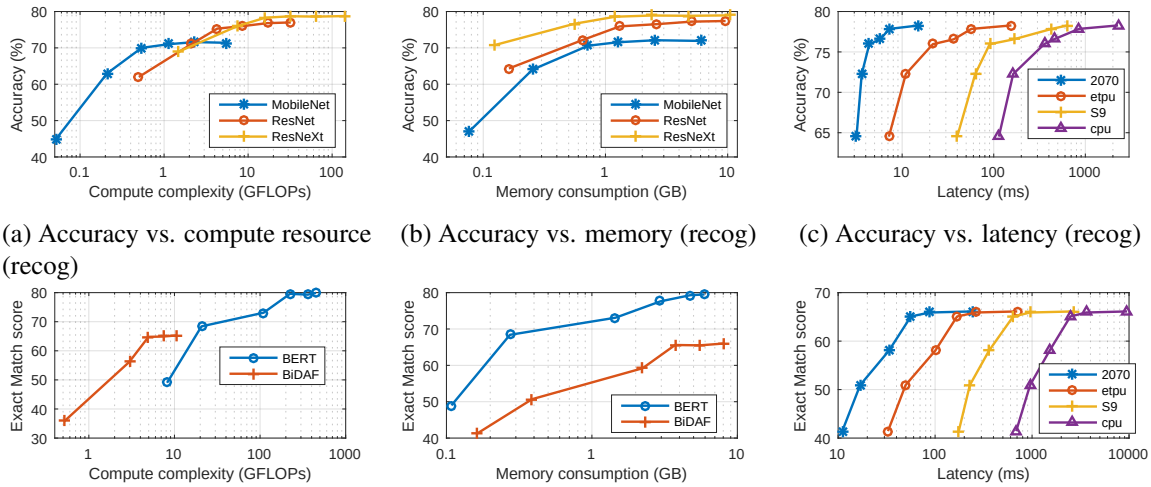
Num of constraints	Num of configs		
	10	100	1000
1	0.03	0.34	5.22
2	0.05	0.71	11.94
3	0.08	0.92	15.46
4	0.14	2.13	37.91

DNN. We can see from the table that loading a network to memory for inference serving takes 2-20 seconds, whereas saving such switching overhead requires storing around 75% additional parameters in memory for 3-branch cases, and around 1.5× for 5-branch cases, affordable for most modern hardware. For modern hardware, typically a few GB is affordable in avoiding over 20 second service downtime, critical to many scenarios.

Latency of parsing adaptation configurations. Recall that we support four types of constraints (inference latency, accuracy, memory consumption, computation complexity), we measure the latency of generating a configuration tree (Section 3.3.3) given various numbers of configurations. Table 3.4 shows that when considering all four constraints and given 1000 different adaptation configurations, it only takes around 38 ms to generate the config tree, a negligible latency compared to the overall adaptation time.

3.7.4 End-to-end performance

Settings. Based on the device specifications in Table 1.1 and typical latency requirements for vision and NLP tasks [9, 6], we generate a set of execution settings (with different combinations of memory, complexity, and latency constraints), feed them to *Mistify* with different source DNNs, and evaluate



(a) Accuracy vs. compute resource (recog)

(b) Accuracy vs. memory (recog)

(c) Accuracy vs. latency (recog)

(d) Accuracy vs. compute resource (Q&A)

(e) Accuracy vs. memory (Q&A)

(f) Accuracy vs. latency (Q&A)

Figure 3.11: The dynamic tradeoff between latency, accuracy, and resource consumptions with *Mistify*.

Table 3.5: Comparison of overhead for porting DNN to edge with/without *Mistify*.

	2 configurations				10 configurations				100 configurations			
	Man.	Mor.	Chm.	***Mist.***	Man.	Mor.	Chm.	***Mist.***	Man.	Mor.	Chm.	***Mist.***
Lines of Code	>0.2k	55	97	**6**	>1k	138	159	**14**	>10k	782	511	**104**
Num of Files	6	4	5	**1**	30	12	32	**1**	300	102	302	**1**
Total time (%)	100			**54.2**	100			**12.5**	100			**2.86**

the DNN models produced by *Mistify*, using Morphnet [100] and Chamnet [102] algorithms.

Balancing performance and resource usage. We first evaluate how well *Mistify* balances the accuracy performance and resource consumption. We select common resource and latency constraints as the execution settings for different edge devices. We set the memory consumption budget from 0.1 GB to 10 GB covering embedded IoT devices to edge server scenarios. We set the computation complexity constraints for running inference on a DNN between 0.1 to 100× GFLOPs, which could further translate to staying within around 10× ms inference latency from resource-constrained devices to powerful edge servers.

Figure 3.11 shows the three-way trade-offs between accuracy, latency, and resource consumption.

The top three correspond to recognition, the lower three to Q&A. *Mistify* reduces the computation complexity by over 20× with less than 5% accuracy loss for the vision workload, and could achieve 50× reduction of complexity in exchange for 12% relative quality of result degradation. Note that the accuracy loss is due to the adaptation algorithms, not *Mistify* itself. Similarly, *Mistify* consistently achieves a near-optimal and practically usable accuracy (comparable to existing hand-tuned on-device models in production [96, 102]) with between 0.5 to 10 GB memory consumption during run time, hence significantly decreasing the deployment complexity for many state-of-the-art DNN models on the edge. Mapping resource consumption to inference time, *Mistify* consistently achieves a near-optimal accuracy performance even when the latency requirements vary by 8 to 10×, for accelerator hardware ranging from advanced data center grade to low-power and lightweight devices.

Simplifying manual overhead. We further assess the end-to-end manual effort and time overhead needed to port a pre-designed DNN to different edge devices. The manual overhead is quantified with two metrics: lines of code (LoC) needed for code addition or modification, and number of files (NoF) touched. The former depicts the overall overhead, and the latter one captures the scatteredness of the modifications, which correlates with the probability of making mistakes. For NoF, we follow a typical file organization [152], i.e., model definition, training, evaluation, and other stages are separated into different files or folders.

We compare *Mistify* with manual porting, MorphNet [100] and ChamNet [102] (two state-of-the-art "automatic" model tailoring toolkits and libraries). Different deployment scales and scenarios are covered by varying the number of distinct configurations from 2 to 100.

Table 3.5 demonstrates that *Mistify* reduces the overall LoC modification needed by 7 to 10×. More importantly, *Mistify* exposes high-level configuration files to users, obviating the need for source script modifications. *Mistify* only requires editing one file. Thus, it saves the number of files users need to access by orders of magnitude (over 100×). Finally, *Mistify* can manage adapting to 100 execution settings using less than 3% of the time of the other approaches, highlighting the enormous potential of harnessing the correlation among configurations to optimize the overall porting efficiency.

3.8 Related work

We are not aware of prior work that aims at providing an automatic porting service bridging DNN design and seamless edge deployment. The most related work revolves around model adaptation and knowledge distillation *algorithms*.

Model adaptation. In production, DNN models hand-tuned by experts have already been shown to run both fast and accurately on mobile devices [94, 95, 96, 97]. The essential techniques include quantization, sparsification, and neural block optimization. Recently, Distiller [98], AMC [99], MorphNet [100], OFA [101], ChamNet [102], and many neural architecture search (NAS) works [103, 104, 105, 106, 107] further explore how to algorithmically explore the search space and find the optimal neural network structure, obviating the hand-tuning by experienced experts. However, none of them is directly usable like *Mistify*, because they are all still *algorithms*, require manually annotating source code to construct the adaptation logic, hence not scalable to edge scenarios with multiple adaptation instances. None supports the on-device tuning scenarios or considers runtime adjustments. *Mistify* is orthogonal as an automated system framework and can incorporate them as pluggable algorithmic modules.

While frameworks like TF-Lite [18], PyTorch [88], and MCDNN [19] provide some model *compression* and *switching* support, *Mistify* differs in techniques supported and the level of manual efforts. To generate a good model, careful model architecture design is essential, which normally requires significant expertise. *Mistify* abstracts the model architecture adaptation process with the configuration APIs to make it accessible to non-experts, easy to automate the end-to-end process and optimize for batch model generation at an abstract graph level.

Knowledge distillation. This was initially proposed as an optimization for model training by transferring knowledge (i.e., parameter values) from a teacher network to a student network [112]. Later works extend the idea to a mutual distillation setting among peer models [113, 114, 115]. *Mistify* adopts and revises the general idea in a selective distillation manner to improve edge training accuracy while enhancing privacy.

Edge-centric deep learning inference engines. Emerging frameworks such as TF-Lite [18] and more [58, 73, 153] are optimized for inference serving on mobile and IoT devices, aiming to hide

the deployment complexity from developers and device users. However, the interface exposed by existing engines only permits model downloading from the cloud (or the central server), without tailoring to edge runtime requirements and constraints, proactively or reactively. In contrast, *Mistify* provides an interface for two-way state exchange and a feedback loop between the cloud and the edge, facilitating targeted model design and efficient execution on the edge.

3.9 *Mistify* summary

Deep learning models today are typically trained on the cloud and then ported to edge devices manually. Not only is manual porting unscalable, it indicates a lack of separation between model design (optimized for accuracy) and deployment (optimized for resource efficiency).

In this chapter, we design and implement *Mistify*, a framework to automate this porting process, which reduces the DNN porting time needed to cater to a wide spectrum of edge deployment scenarios by over 10×, incurring orders of magnitude less manual effort. *Mistify* not only provides a useful service to complete the transition from DL workload design to deployment on the edge, but cleanly separates these two stages. We believe the system will further facilitate advanced model design and seamless model deployment.

Chapter 4

Sommelier: DNN Model Indexing and Query Service

Deep learning (DL) inference accounts for the explosive growth of analytics workloads everywhere, in the cloud and on edge devices. Computer vision (CV) and natural language processing (NLP) tasks are the dominant deep learning workloads currently deployed. According to Facebook statistics [154], the volume of their workloads tripled in less than two years, increasingly supporting diverse applications. These workloads are usually resource intensive but also increasingly user facing, hence under stringent latency requirements.

As it requires significant expertise and computation resources to design deep neural network (DNN) models, it is increasingly common to use a pre-trained model (e.g., ResNet [148] for image recognition), either verbatim or as the basis to *transfer* the model to the target application (e.g., object detection and semantic segmentation). Typical edge inference engines provide an API to load an existing model from a given repository. Model designers often start with an existing model or incrementally generate new models. Model testers use a set of similar models to identify adversarial inputs that lie at the decision boundaries. (Section 4.1.1). As a result, DNN model repositories have become essential players in existing machine learning ecosystems, e.g., TF-Hub for the TensorFlow ecosystem [17], PyTorch Hub for PyTorch [88] and Model zoo for MXNet [86]. These are even more helpful for the general public.

Challenges. However, existing model repositories provide a barebone interface for the user to

retrieve a specific model. While it is common for a repository to house hundreds of models, the onus is on the user to profile and identify precisely which model to use from potentially hundreds of DNNs, including the specific version for a particular DNN design. This level of repository support barely relieves an average user of the expertise required to design the model in the first place. This is especially cumbersome for a model user who simply wishes to deploy one in an application [155]. A suboptimal model could miss the achievable accuracy target by 10% or waste 20× more resources. Having to manually profile and search through individual models further inhibits effective runtime adaptation to fluctuating resource availability during prediction serving. Section 4.1.2 discusses these issues further.

In summary, the challenges are 1) the redundant expensive efforts of exhaustively profiling and benchmarking DNN models from the repository by different edge DL-based application developers every time they select a DNN model to use; and 2) the interdisciplinary expertise needed for understanding both the algorithm and the execution details of the DL algorithms to figure out the optimal DNN model to use for a particular DL inference workload.

Contribution. In this work, we propose *Sommelier*, an indexing and query service over typical DNN model repositories, which automatically searches through the repository for the most suitable model based on a desirable semantic requirement and resource budget, without requiring the user to explicitly profile individual models.

For the algorithmic aspect, recognizing the difficulty to quantify the exact semantics of each DNN model, *Sommelier* defines a notion of *generalized semantic equivalence* between models (Section 4.2) and formulates the query goal as finding a model most interchangeable with a well-known reference model (e.g., ResNet). Further, *Sommelier* proposes algorithms and indexing structures to measure and organize DNN models based on this equivalence notion (Section 4.3). This eliminates the aforementioned redundancy (and therefore the key scalability challenge) of exhaustively profiling and benchmarking DNN models.

Meanwhile for the system aspect, *Sommelier* resolves the challenge of requiring the users to have expertise in both the theory and execution details of the DNN models by exposing a model query service abstraction on the existing model repository facility (Section 4.4), as the delegate for model selection, which automates the process of DNN model semantic understanding and resource

profiling, and hides the complexity and efforts from the users.

In summary, we make the following contributions:

First, we formulate the lookup requirements of model repository users as quantifiable constraints around DNN resource consumption and semantic equivalence between them.

Second, we design an algorithmic primitive and code library to automatically extract semantic equivalence across DNN models, especially between model segments. To our knowledge, no previous work automatically infers DNN sub-structures. We provide both proofs and empirical analysis to show the "distance" between models.

Finally, we build a query system, *Sommelier*, that indexes DNN models and their resource footprint and selects the most suitable model given a performance specification and resource budget. *Sommelier* facilitates more extensive and precise usage of model repository with minimal expertise or manual effort, and can further contribute to explainable AI.

4.1 Motivation

4.1.1 The need for a DNN model repository

Building DNN from scratch is too expensive. The efficacy of deep learning rests on the quality of the DNN model, but training a sophisticated DNN from scratch is an immense undertaking. This demands comprehensive understanding of optimization theory and neural network internals, enormous amounts of training data, and computation resources. For example, training a ResNet50 network involves carefully choosing optimizers and hyper-parameters, and writing hundreds to thousands of lines of code spanning Python, C++, and specialized libraries (e.g., CUDA, MKL [156]) to deploy the whole training pipeline; the entire training run could take 14 GPU-days, processing nearly 1 TB of training data [148].

Given the colossal cost of training new models, repositories of DNN models are increasingly adopted in DL ecosystems, to store pre-trained models for diverse model reuse possibilities. For instance, we analyzed around 150 active DL projects on GitHub and over 94% of them involves building and training upon existing models loaded from some model repository. Further, a small set of six common neural network models is chosen by over 60 projects to carry out their learning

functionality. This confirms the importance of model repositories.

Usage of model repository. Model repository significantly reduces the learning curve of adopting deep learning in practice. Application developers can directly choose a pre-trained model from the repository that matches the required functionality (e.g., object detection) or certain model segments (e.g., visual feature extractors) to build their learning-based applications without domain knowledge of the DNNs [157, 158].

(i) *Inference serving*. Various deep learning inference serving frameworks (e.g., Tensorflow-serving [12], Clipper [60]) have been developed to provide runtime support for low-latency, accurate, and robust DNN-based prediction tasks. These systems interpose between end-user applications and the deep learning engines, hiding the complexity of implementing and deploying the DL logic from the application developers. Applications directly specify the model and provide input data, and the inference serving system, integrated with a model repository, will load the specified model in memory, and execute the input data.

(ii) *Model design*. With the emergence of training techniques such as transfer learning [159] and knowledge distillation [160], model repositories are becoming indispensable to deep learning researchers. These techniques facilitate *incremental* new model design by copying (a segment of) an existing well-trained network as the basis and adapting towards new use cases with substantially less effort (in model designing time and training data size). For instance, common well-trained neural networks (e.g., ResNet [148], BERT [7]) are widely utilized for downstream CV and NLP tasks [161, 162], accelerating the training time from *days* to *minutes* [154, 7].

(iii) *DNN model testing*. DNN models deployed in safety-critical applications such as autonomous driving need to be robust against adversarial input data, namely specific input values that could trigger abnormal inference results and potentially dangerous behaviors, e.g., misclassifying a stop sign as a red flower. Key to the model verification process is to identify corner case input data. These are typically found by loading a few similar DNN models from the model repositories [163, 164] and exploring the intersection of their decision boundaries of these models.

4.1.2 Limitations of existing model repositories

Existing model respositories (e.g., TF-Hub [17]) act as a remote filesystem only, with primitive APIs to publish and load a model. To retrieve a model a user has to specify the precise URL to the model file. This requires significant user sophistication regarding the right model choice.

Inference serving. Since the rationale behind an inference serving system is to hide complexity, developer interactions with the associated model repository should only include *high-level specifications* of DNN models (e.g., accuracy, latency, and resource usage) as the inputs, rather than the exact model name and version. This is even more so for anyone not familiar with detailed DNN model profiles. From the perspective of a DL inference serving system, the *runtime execution environment* (e.g., queue length, caching strategy) and *application performance goal* (e.g., critical vs. non-critical period) fluctuate [41, 165], making manual and static model selection a poor match for myriad runtime optimization needs which necessitates the model repository to automatically suggest an optimal selection.

Model design. Even for domain experts, appreciating the accuracy and resource usage of all models in advance is impossible. For instance, ResNeXt101 and MobileNet, two models trained with the same ImageNet dataset for classification, differ by 10% in accuracy and 20× in memory consumption. Further, such numbers are measured under a specific setting only, and could vary with the datasets and hardware platforms. Enumerating models by name and profiling each until the best fit is found is extremely unscalable.

DNN model testing. Since an important model testing step is to find the "tricky" input data using a set of similar but not identical models, the quality of model selection dictates the coverage and soundness of the testing process. Ideally, the repository should automatically determine which set of models is similar yet provides sufficient local differences to explore adversarial examples. Instead, this is currently done manually, with significant amount of repeated efforts when the same model is tested multiple times.

Summary. All three use scenarios manifest the same fundamental limitation of existing model repositories: there is ***no query support*** for the model repository, only a barebone filesystem. This then leaves model selection to manual operations, which is time-consuming and often results

in suboptimal performance. For instance, manual selection effectively precludes runtime model switching when serving inference tasks. Only ML experts with deep knowledge of DNNs can take full advantage of such model repositories without worrying about suboptimal selections.

4.1.3 Requirements for DNN query support

Given the shortcomings of existing model repositories, we build *Sommelier* to provide model query support. We next discuss the requirements and challenges for a query system.

Canonical model lookup requirements. Fundamentally, *Sommelier* supports a query operation, `query()`, where the method signature captures the user requirements. An example query might take the format of "find a model for vision on embedded devices that uses less than 50 MB memory but only allows within 5% accuracy loss to ResNet model" [94]. For a DNN, its inference accuracy, resource usage, and computation latency are the three key factors forming the multi-dimensional design space that concerns a user. Latency can be estimated from the resource usage when given the hardware specification. Therefore, a DNN model query should essentially specify two lookup conditions: (i) *model semantics* (e.g., object recognition with over 95% accuracy over 1000-class ImageNet syntax), and (ii) *resource consumption* (e.g., less than 1 GB memory and 0.5 TFLOP). To fulfill such query requests over neural network models, we need to organize the models in a reasonable way, and search through these models efficiently, which poses several challenges.

Challenge 1: assessing DNN model semantics. While organizing the DNN models along the resource consumption axis is intuitive, it is not straightforward along the model semantics axis (e.g., by an accuracy target for a specific functionality). The latter relies on a measurable definition of DNN semantics and the right primitive to compare and rank DNNs, neither of which is obvious. DNN models are described by directed graphs of mathematical operators, and hence have unique mathematical expressions. However, we observe that using these expressions to define and "compare" DNN semantics does not suit practical scenarios. Instead, given the nature of DNNs and their interaction with existing repositories, we find that assessing pair-wise semantic correlation between models is more insightful than attempting to quantify the model semantics in absolute terms (Section 4.2).

Challenge 2: quantifying semantic relations between DNN models. Given the primitive to assess DNN semantics, the next challenge is designing algorithms to measure the semantic relations between

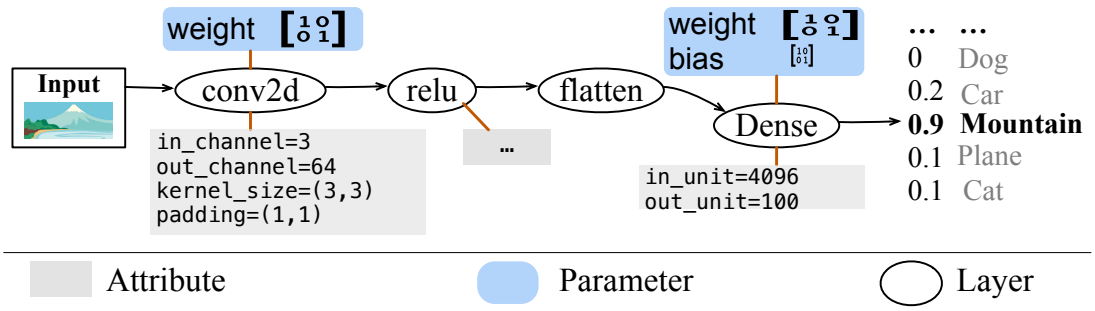


Figure 4.1: Anatomy of a DNN based inference task.

models. First, the nature of such relation differs depending on whether it is between two DNN models holistically or between model segments, which requires different methods to handle. Second, DNN models have complex topologies, diverse operators, and different validation dataset availability, which requires generic and yet extensible algorithmic design. Third, DNN structures are increasingly complex, hence requiring efficient and scalable techniques for model traversal. Section 4.3 describes our approach to address these issues.

Challenge 3: query specification and processing. From a system perspective, the challenge centers around (i) how to design an expressive query interface and specification to cover the user requirements broadly; (ii) how to design index data structures and process the query accurately and efficiently. These questions are answered in Section 4.4.

4.2 Characterizing DNN semantics

The anatomy of a DNN. Figure 4.1 shows an example recognition task using a DNN. It takes an image of mountain (represented as a tensor) as the input, extracts and processes input features layer by layer, and finally generates an output vector whose largest dimension reflects the result "mountain".

A DNN is typically expressed as a directed acyclic graph (DAG), following a dataflow model. Each node in the DAG is a base layer in a network, considered as an atomic unit carrying out a certain operation (e.g., 2D convolution) on its input. Each DNN layer is characterized by *attributes* and *parameters*. Attributes (the grey boxes in Figure 4.1) describe the type and shape of the input/output tensors and their dependency. Parameters (the blue boxes) capture the internal states of a layer (e.g., the weight and bias tensor of a Dense layer).

4.2.1 The futility of conventional view

A DNN model is simply a sequence of primitive mathematical operators. Therefore, it is intuitive, and a common practice, to use the mathematical expression of a DNN to denote its task semantics, and then compare the semantics between DNN models by assessing the difference between their expressions. However, this common practice is problematic.

Reason 1: there is no unique formal representation of the DNN model and the DL task. Training deep neural network models are theoretically understood as a function approximation process, where the exact "functionality" is unknown, but is gradually *approximated* through a finite set of input data and output labels. The function thus derived is evaluated by how well it *generalizes*, i.e., how accurately it performs the inference task when fed unseen data. This implies that the same "function semantics" can be "described" via totally different mathematical representations. From the viewpoint of formal verification, the strongest postcondition for a DL task is not unique because multiple outcomes can be *acceptable* given the same input [166, 167, 168]. This departs substantially from the traditional sense of program semantics. For a function like `sort()`, we can specify the strongest postcondition uniquely since it produces a unique *correct* output given an input.

Reason 2: program correctness is no longer a "yes" or "no" binary state, but extends to a tunable performance metric, *accuracy*. It is common to revise the neural network structure to adjust the tradeoffs between accuracy and other performance metrics, but this does not change the functional semantics of the task. For instance, neural architecture search algorithms (e.g., OFA [101] and MnasNet [123]) adjust the neural network structural complexity, between $0.1\times$ to $10\times$ in exchange for the right accuracy targets that vary by almost 10%, in order to balance between acceptable performance and resource footprint on edge devices. Minerva [151] prunes computation on the hardware to achieve $3\times$ processing speedup on edge devices, at the expense of 2 to 10% accuracy drop.

Quantitative evidence. We next empirically show the discrepancy between the *mathematical difference* and *semantic equivalence* between various DNNs. We select 5 widely used DNN models (Resnet50, Inception, ResNext101, VGG19 and MobileNet), all pre-trained with ImageNet for image classification, and feed the same test input to all of them. In Figure 4.2, the off-diagonal entries show the fraction of results (corresponding to the top-1 accuracy) that agree completely, while the diagonal

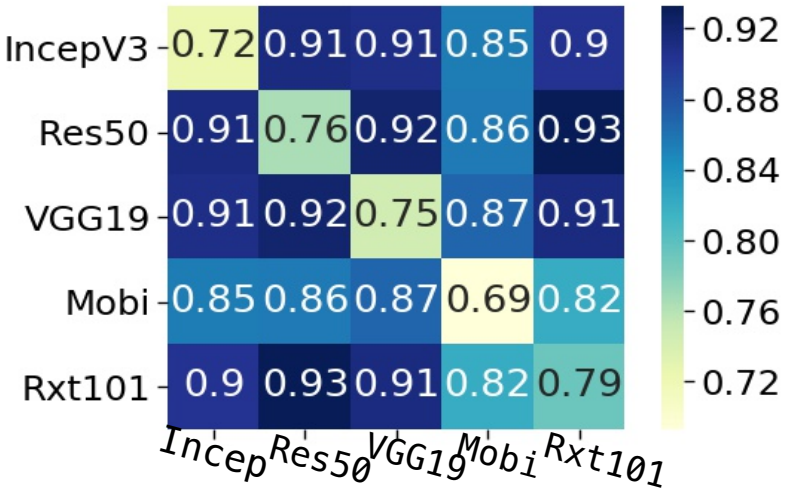


Figure 4.2: Extent of equivalence between DNN models.

entries show the inherent top-1 accuracy for each model. Interestingly, the output agreement ratio *between* models is significantly higher than their inherent accuracy values. This implies that these models are functionally equivalent and highly interchangeable in practice, yet none of the models is the definitive one for the image classification task.

Observation. The correlation between models is not surprising. Intuitively, if two models are both expected to identify a cat, say, they both need to learn the feline features from the training data. Given most DNNs are trained with the same few standard datasets, there is far less diversity among the *features* represented than the *models* extracting these features. In other words, there is inherent correlation between the features identified by different models. This is more deterministic than the individual feature extraction process, which sheds light on an alternate view of DNN model semantics.

4.2.2 Alternate view: Model equivalence

Inspired by the observation above, we explore DNN semantics by instead harnessing the correlation between models. Users typically know about the accuracy and resource profiles of a few well-published models, e.g., ResNet for computer vision and BERT for NLP. Therefore, we can assess the semantics of a model with respect to a well-known reference, supplied by the user or defined by default.

Semantic equivalence between DNN models. We formally define the *semantic equivalence* be-

tween DNN models as ***the interchangeability of the models to achieve the inference task.*** Suppose we have a model M_1 and an arbitrary dataset D_1 containing input data and groundtruth results of the inference task. A second model M_2 is semantically equivalent to M_1 iff feeding D_1 to M_2 achieves a quality of result (e.g., 95% classification accuracy) comparable to M_1, differing by less than a user-specified threshold ε (e.g., 5%). The notion of generalized semantic equivalence can be applied beyond DNN, but we restrict our discussion to DNNs in this chapter.

The rationale is that (i) the equivalence measurement between DNN models is decoupled from their concrete mathematical representations; and (ii) the *threshold* is a control knob for users to customize the level of relaxation acceptable to suit their unique needs. Note that, in machine learning theory, while many terms can be used to express the *similarity* between two functions, these do not translate to the *interchangeability* between models (and their substructures). Therefore, the above definition is needed for our particular consideration.

4.3 Assessing semantic equivalence

Having transformed the problem of uniquely specifying the semantics of a DNN model to assessing the equivalence between models, we next develop an algorithm to quantify the level of equivalence between models.

Given the type of model variants commonly seen today, there are two cases to consider. First, two DNNs might be designed differently but trained with similar data to achieve the same task (e.g., recognizing animals). In particular, one model could be a structurally more compact version of the other. In either case, the two models exhibit semantic equivalence holistically, and we can draw on typical model evaluation approaches. Second, model variants are frequently derived from the common model base, but transferred and fine-tuned to different downstream tasks (e.g., emotion detection and question answering), then the semantic equivalence relation exists between the common base segments of two models. For this, we develop a novel algorithm based on generalization theory, because there is no notion of "accuracy" defined on the output of intermediate DNN layers for evaluation.

4.3.1 Detecting whole model equivalence

We can simply treat each DNN model as a black box and compare them as a whole in two steps. We first check the "structure" of the input and the output, manifested as the data types and the shape of the tensor, to quickly filter out completely different models (e.g., for tasks that are not comparable), and then use a validation dataset to get the quality of result (QoR, e.g. accuracy) difference and compare to the acceptable threshold ε. Analogous to a compiler optimization process, the equivalence detection follows a static analysis of first type-checking and then comparing the "values".

Input and output layer check. For the input layer, we check the input tensor shapes of the candidate models to determine if they possibly capture the same semantics. However, this can be misleading because resizing and other preprocessing can be applied to the same raw input source causing them to have different shapes. To cope with this, the model designers can specify in the configuration how the inputs should be preprocessed, as well as register custom preprocessors if needed. The strict comparison between input shapes is invoked only when no preprocessing is specified.

The model outputs are typically derived in two ways based on the task category, *classification* (semantics defined by the largest dimension of the output), *regression* (semantics defined by the whole output vector), or a combination of them when there are multiple outputs. For regression-style outputs (e.g., object detection, word embedding), we simply observe the output shapes. If the shapes are identical, we pass the two models to the next checking phase. For classification-style outputs (e.g., object recognition), a finer-grained check can be additionally conducted if users specify the output syntax, namely the syntax label of each output dimension (e.g., dimension i maps to "dog", dimension j maps to "cat"). Such an analysis could exclude those models having the same output shape but carrying different semantics.

Finally, the models with matching input and output layer structures are passed to the next step.

Assessing semantic equivalence. We next feed the validation dataset to two candidate models, measure the average QoR difference, and compare with the (default or user-specified) threshold ε. In most scenarios, QoR goals are the same as the optimization objective in the model training phase. However, if unspecified, the default QoR difference is computed as the average l_2 distance between the outputs of the two models.

So far, the QoR difference measured is an empirical value which might be specific only to this

validation dataset. To generalize, we leverage the generalization theory of DNNs [169, 170] as a guide to generate an upper bound of the QoR difference *independent of the validation dataset selection*. In brief, the upper bound of QoR difference is calculated by adding to the empirical value a generalization bound dependent on the architecture of the neural network model. The generalization bound is expressed as: $\tilde{\mathcal{O}}\{(\frac{1}{\gamma^2 n} d^2 max\|f(x)\|_2 \sum_{i=1}^{d} \frac{1}{\mu_i^2 \mu_{i\rightarrow}^2})^{1/2}\}$, where γ is determined by the specific metric chosen by the inference task, n is the size of the validation dataset, d denotes the total number of layers, $\|f(x)\|_2$ denotes the model output vector l_2 length, μ_i and $\mu_{i\rightarrow}$ are inter-layer factors calculated from the weight matrix of the layer (details in [109]).

Finally, we determine the two models to be semantically equivalent if the upperbound QoR difference is smaller than the pre-defined threshold ε. Specifically, another model M' is judged semantically equivalent to model M when $|max(QoR_M, QoR_{M'}) - \mathsf{Diff}(M \rightarrow M') - QoR_M| < Threshold$. We denote the output QoR difference as Diff, original QoR values of the two models as QoR_M and $QoR_{M'}$, and the threshold (app-specified or otherwise default) for QoR difference $Threshold$.

4.3.2 Equivalence between model segments

In the case of transfer learning and model adaptation based use scenarios of the model repository, two DNNs may not be equivalent in their entirety, but share semantically equivalent segments (e.g., a stack of layers). This motivates us to further analyze equivalence between DNN model internals. Again, the detection proceeds in two steps, checking the structure of the model to extract potentially equivalent model segments, and then assessing the semantic equivalence of them.

Extracting model segments. Unlike the whole model scenario, only checking the first and last layer around intermediate model segments is far less informative for filtering out unrelated models. Instead, we view the neural network models as DAGs and extract the common sub-graphs as the candidates that are possibly equivalent. However, detecting common sub-graphs between two graphs is well-known as an NP-hard problem and is simply not scalable. Fortunately, unlike general graphs with arbitrary node connectivity, DNNs tend to connect layers sequentially, and only involve a small set of parallel branches locally (e.g., residual connections in ResNet [148]). Therefore, we propose a detection algorithm that instead finds the longest common *operational sequence* as the candidate segments, which reduces the complexity to $\mathcal{O}(N^2)$, where N is the number of layers, without missing

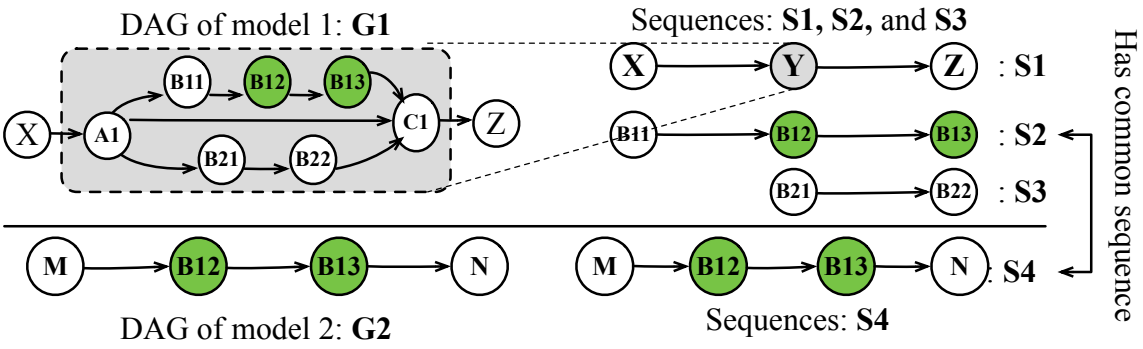


Figure 4.3: Extracting model segments recursively.

useful segments.

Our algorithm first recursively extracts the longest sequence of operations from each DAG. As an example shown in Figure 4.3, the operation sequence $S1$ is extracted first; then, zooming into the "operator" Y, another two sequences of operations, $S2$ and $S3$, are further extracted from the graph $G1$. Now, with a set of sequences (i.e., $S1$ to $S3$) extracted from each DAG, we find the longest common sequences (green shaded) between the two sets of sequences as the candidate model segments for further assessment.

Revising equivalence definition for model segments. For model segments, the definition of semantic equivalence involves an additional step because intermediate segments themselves do not have validation datasets and quality of result (QoR) metrics. Suppose we have a segment S from model M, and another segment S' structurally identical to S. We can derive a *twin* model M' from M by replacing the segment S with S'. Now, we can translate the semantic equivalence of S' with S to the semantic equivalence of M' with M, as defined previously (Section 4.2.2).

Layer-wise output difference estimate. Since a large number of model segments can be extracted from a single model, assessing whole model equivalence for each segment or their combinations is far too complicated for practical purposes. Instead, we take random vectors as the inputs, *calculate the theoretic upper bound* of the output vector difference brought by the candidate segments in the model and their possibly equivalent counterparts. Finally, we derive the QoR (e.g., accuracy) difference from the theoretic output difference upper bound accordingly.

Given the same input, the upper bound output difference between two model segments can be calculated layer-wise from the input layer to the output layer by induction, namely, we derive the

output difference for layer i from the counterparts of layer $i-1$ following the data dependency.

We classify neural network layers (operations) into three categories and handle them individually: *linear* operations, *non-linear* operations, and *multi-source* combinations. Linear operations essentially cover all kinds of layers invoking matrix multiplication (`FullyConnected`, `Convolution`, `Embedding`, etc.). Non-linear operations include activation (`ReLU`, `tanh`, `sigmoid`, etc.), pooling (`maxpooling`, `meanpooling`, etc.), and normalization. Multi-source combination refers to merging multiple input sources into a single output (`add`, `multiply`, `concat`, etc.). Note that even though recurrent operations (RNN, GRU, LSTM, etc.) are typically viewed as independent operators, their essential computing logic are no difference than a combination of aforementioned basic operations. Therefore, each recurrent operator itself can be treated as a model segment to conduct error analysis. Furthermore, recently proposed self-attention layers are recommended to replace the recurrent operations, which simply consists of several typical matrix multiplication operations [171].

Starting with the *linear* operations (the computation kernel of almost all DNN layers), the output difference of the current layer comes from two sources: (i) the output difference of previous layers propagated to the current layer; and (ii) the additional output difference incurred by the weight parameter differences between the current layer and its equivalent counterpart. Suppose W' is the current layer's weight matrix, W is the counterpart layer's weight matrix, and ΔX is the upper bound of the difference vector. The propagation of the upper bound of the difference vector ΔX is deducted as follows. Assuming the current operator is at layer i, we denote the input and output difference vectors by ΔX^i and ΔX^{i+1}, and the weight matrix difference by $\Delta W = W' - W$. The relationship between ΔX^{i+1} and ΔX^i can now be expressed as:

$$\Delta X^{i+1} = (\Delta W) \cdot X^i + W' \cdot \Delta X^i$$

Then, we can deduce the upper bound layer-wise as:

$$max\|\Delta X^{i+1}\| \leqslant \lambda_{max}(\Delta W) \cdot max\|X^i\| + \lambda_{max}(W') \cdot max\|\Delta X^i\|$$

Whereas $\lambda_{max}(W)$ denotes the largest singular value of matrix W, and $max\|X^{i+1}\| = \lambda_{max}(W) \cdot max\|X^i\|$. Note that, for `Convolution` layers, the kernels are always internally reshaped (according

to `im2col`) into a single 2D matrix before calculating output. Therefore, they are treated the same way as `FullyConnected` layers even though the latter often involve multiple kernels on multi-dimensional inputs.

Next, we derive the output difference bound for *non-linear* operations. Consider activation layers (`RELU`, `tanh`, and `sigmoid`) first. For ReLU activation, its expression $relu(x) = max(0,x)$ ensures that $|relu(x)| < |x|$. The same holds for its recent invariants (`LeakyReLU`, `PreLU`, `ELU`, `Swish`, etc.). For `sigmoid` and `tanh` activations, their expressions are $\sigma(x) = \frac{1}{1+e^{-x}}$ and $\tanh(x) = 2\sigma(2x) - 1$ respectively. Their derivatives are not greater than $1/4$ and 1 respectively, which all ensure $|activation(x)| < |x|$. Therefore, even though activations are non-linear, it is safe to upper bound the l_2 norm of the activation output by simply by the input, because $activation(X_i)^2 < X_i^2$ for all elements of the input vector X. Namely, all these activation layers and their variants (e.g., `LeakyReLU`) follow $|activation(x)| < |x|$, which means the input difference bound itself could serve as the upper bound of the output difference. Then, for `Pooling` layers, it is easily proved that the l_2 difference of the outputs is always smaller than or equal to that of the input difference. For normalization layers, the output difference is scaled by a factor determined by the length of the original output vectors. Thus, we can simply derive the upper bound as $\Delta X^{i+1} = \|\Delta X^i\|/\|X^i\|$.

Finally, we handle *multi-source* combination operations. Intuitively, we treat the difference vectors of each input source as an independent random variable. Then, the statistical feature of the output difference vectors can be derived by that of the input vectors according to the way in which the output vector is mathematically expressed.

To sum up, for each type of popular neural network layer (operator), we propose an approach to derive the output difference upper bound from its input difference and the weight matrix of its possibly equivalent counterpart.

Assessing semantic equivalence. Based on the previous bound analysis, the algorithm proceeds as follow. (i) Once all segments S_i that have structurally identical counterparts S_i' from other existing models are extracted from the current model M forming a set $S_M = \{S_i\}$, a forward pass of model M is conducted to derive the output difference upper bound with respect to S_M. (ii) Then, we add Gaussian noise (scaled to center on the value of the output difference upper bound) to all the output vectors collected previously, and then use these noisy outputs to measure the average quality of result

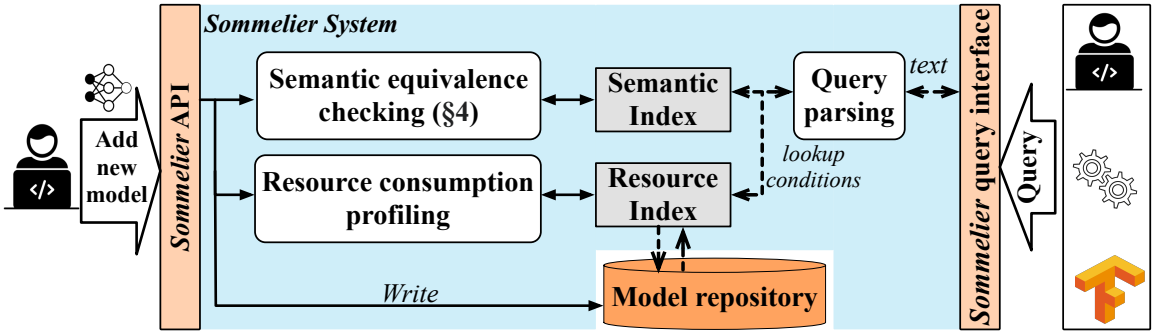


Figure 4.4: *Sommelier* system architecture.

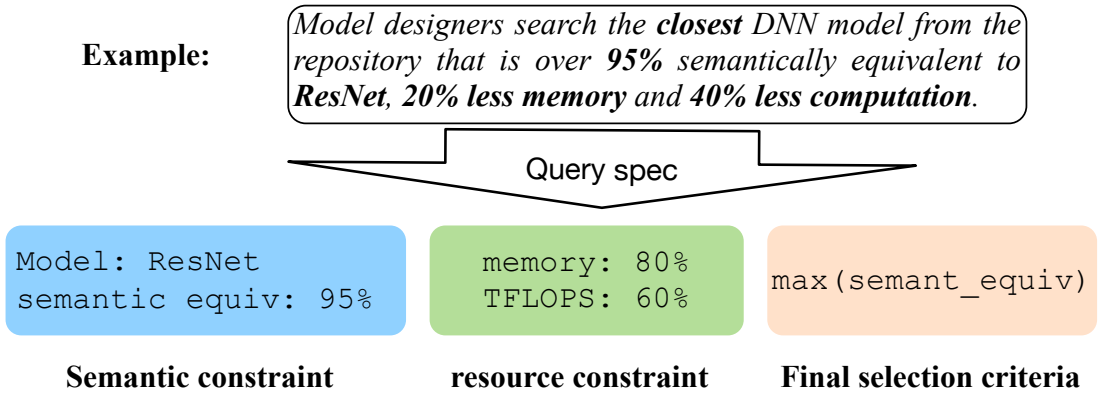


Figure 4.5: Specifying a concrete use case as a DNN query

$QoR^{(S_M)}$. Subtracting $QoR^{(S_M)}$ from the original QoR (measured with the validation dataset) finally gives the QoR difference, which is compared with the threshold ε. (iii) If the QoR difference exceeds the threshold, we gradually remove segments from the set S_M starting with the one with the lowest computation complexity, recalculate the new QoR difference via steps (i) and (ii), until the bound falls within the threshold. The algorithm returns the current *semantic equivalence segment set* (S_M), or null if S_M is empty.

4.4 DNN model query with *Sommelier*

Building on the semantic analysis of DNN models discussed in the proceeding sections, we design *Sommelier* to support DNN model queries. Figure 4.4 shows the system architecture, key components, and its interface with the model repository and other parties. The core system is built on a pair of index structures, a semantic index and a resource profile index, keyed with the same DNN model.

4.4.1 Formulating DNN model queries

Recall the model lookup requirements in Section 4.1.3. We first define a query specification to express desirable model semantics and resource usage, as shown in Section 4.5.

A user specifies the *semantic constraint*, the *resource budget*, or the *final selection criteria* in their queries. In particular, a semantic constraint is defined by a reference model (in a serialized format) and the semantic equivalence threshold with respect to this reference. Resource constraints include computation complexity and memory consumption. These are meant to indicate the user's resource budget or resource usage target. A user does not need to know the exact resource footprint of any model. Final selection criteria outline any additional method(s) to select the final output among the retrieved candidate models (e.g., the model with the "most similar" semantics, or select with user-defined utility functions). If users have no preference for or prior knowledge of any specific DNN model as the reference, they can specify the inference task category instead and *Sommelier* supplies a default reference model for that task.

Figure 4.5 shows a query example, where, say, a model designer wishes to find a DNN model that is `most` interchangeable with the latest version of `ResNet` (i.e., equivalent to ResNet 95% of the time) but consumes `20%` less memory and `40%` less computation time. Note that such queries are not about exact matches, but like range queries that jointly consider multiple lookup conditions. We use TFLOPS and memory as the two most representative resource metrics to explain the *Sommelier* design, but *Sommelier* is not restricted to these two metrics. Custom metrics (e.g., latency) can be plugged in easily (shown later in Figure 4.6).

4.4.2 Semantic index

Sommelier leverages an index structure to track the semantic equivalence relations between stored DNN models, so as to process the queries efficiently without having to compare the semantics between each pair of DNNs. The top-level structure of the index is a hashtable. For each entry in the table, the key is the hash fingerprint of a DNN model, and the value is a list of candidate records, each of which consists of a candidate DNN model and its semantic equivalence level to the keyed model. The records within each candidate model list is maintained in descending order according to the semantic equivalence level. Therefore, the hashtable maintains the mapping between a model to

all its semantically equivalent counterparts.

Insertion to the index. When a new DNN model (M_n) is added, *Sommelier* randomly selects 5 existing models in the repository and conducts a pairwise semantic analysis between M_n and the selected models. The difference between (M_n) and the other models can be derived transitively: suppose models X and Y differ by A, Y and Z by B, then the semantic difference between X and Z is bounded by $|A-B|$ and $|A+B|$. Empirically, this sampling approach dramatically improves scalability without degrading query quality much. Currently, there is one index for the entire repository, since the majority of existing models are for either CV or NLP tasks. This can be easily extended to one index per inference task category. We assess the semantic equivalence between two models regardless of their actual usage.

A new entry (Rn) is created in the index table representing M_n. (i) For whole models, suppose an existing model M_1 has semantic equivalence level $L_{1\leftrightarrow n}$ to the new model M_n. Then, the model M_1 along with $L_{1\leftrightarrow n}$ is added to the candidate list of the entry Rn. (ii) For model segments, suppose a segment S_1 of an existing model M_1 has semantic equivalence level $L_{s1\rightarrow sn}$ to a segment S_n of the new model M_n (e.g., interchangeable from S_1 to S_n for M_n). Then, a model M_n' synthesized from M_n by replacing S_n with S_1 is added to the candidate list of entry Rn along with the equivalence level $L_{s1\rightarrow sn}$. Besides, for each entry of the existing models, the new model M_n is added to their candidate list in the same way as explained.

Lookup with the index. When a query is submitted with the reference model M_n and the semantic equivalence threshold as the arguments, *Sommelier* will first locate the key by calculating the fingerprint of the reference model, and then, from the candidate list, collect as the output all the models whose equivalence level exceeds the threshold. An output model M_i can be an existing real model that is holistically equivalent to the input model M_n. Alternately, M_i can be synthesized by replacing a segment S_n (from input model M_n) with S_j (from an existing real model M_j) such that S_n and S_j are equivalent.

4.4.3 Resource profile index

Likewise, *Sommelier* builds another index structure to record the resource profile of each DNN model.

For each entry of the resource index, the key is a vector whose fields correspond to the usage number of a certain resource (e.g., memory), and the value is the fingerprint of the DNN model. *Sommelier* uses Locality Sensitive Hashing (LSH) [142] to organize the entries for fast distance-based search over resource vectors. A LSH structure has multiple hash functions which collaboratively map "close" vectors to the same bucket (and distant vectors to different bucket) with high probability. Thus, the buckets convey a sense of "locality" where similar vectors are stored together. When searching for similar vectors to the input, LSH uses these hash functions to locate the buckets corresponding to the input and returns the vectors stored in these buckets as the output. Adding a new resource vector to the LSH structure simply involves mapping and storing the vector to the corresponding buckets.

Insertion to the index. Inserting a new DNN model involves updating both the model storage and the model index in the background. The former simply follows the practice of existing model repositories (e.g., TF-Hub, Model zoo).

The essential step for inserting a new DNN model into the index is to generate the resource consumption vector. For computation complexity, we simply sum up the FLOPs of all computation-intensive operators in the model. The memory and latency dimensions additionally involve preparing the model runtime, as execution configurations could significantly affect the numbers [102]. For instance, static versus dynamic memory allocation will affect the runtime memory consumption; the GPU architecture (bus bandwidth and #cores) will affect the latency of the model. To overcome such challenges, we use the same execution settings to load any new model into memory once and note down the actual memory consumption. To estimate the model latency, *Sommelier* separately maintains an operator latency table, which includes the runtime latency of each type of basic neural network operator. Then, given a new DNN model, the estimated latency is the combination of the latencies of all its sequential operators.

Lookup with the index. When a query is submitted with the resource constraints, *Sommelier* first converts the resource specification into the constraint vectors as mentioned, and then uses the vector to query the LSH-based index (i.e., finding all keys with the value field smaller than the constraint vector). Finally, among all the returned models, those that satisfy the constraints in all dimensions will be the outputs.

4.4.4 Query processing

A query submitted to *Sommelier* is first parsed into an abstract syntax tree (AST), from which the user-specified query conditions are extracted to formulate three query processing steps. Each step is determined by a filtering constraint, the *semantic constraint*, the *resource consumption constraint*, or the *final selection criteria*.

Based on the two indices, DNN queries submitted to *Sommelier* is handled as pipelines of *filtering* operations. The filtering operations involve three stages, *model semantic* filtering, *resource consumption* filtering, and *final candidate* filtering. Each stage takes the information from the corresponding part of the query to configure the filter, execute the filtering logic on the index structure, and then intersect the output models from the current stage with the models from the previous stages. Noticeably, for resource consumption filter, the filtering condition is further represented as a multi-dimensional vector. For instance, *memory less than 200 MB, computation complexity less than 50 GFLOPS, and latency less than 30 ms* is simply represented as a vector $(200, 50, 30)$.

4.4.5 Discussion

Supporting developer annotations. The above description for the index generation assumes no metadata is available for any model. If any annotation is available, for example, noting down the model accuracy and resource footprint in a particular setting, *Sommelier* can incorporate and "translate" this information to our standard indexing metrics, in place of the corresponding analysis. However, such annotations are unlikely to replace the built-in analysis provided by *Sommelier* unless they can cover all information about the model in any runtime settings exhaustively.

Framework independency. It is sometimes inevitable to interact with the DL framework runtime to accurately measure the resource consumption and analyze the model semantics. However, neither the model semantics analysis nor the resource profiling differs significantly between frameworks (Tensorflow, Pytorch, MxNet, etc.). Hence, *Sommelier* is not tied to or limited by any specific DL framework.

Resource metrics. In this chapter, we use FLOPS to capture the computational complexity of a model, which is widely adopted by most platform-aware DNN adaptation papers [172, 100, 117]. Although this metric is independent of specific types of hardware and frameworks, a drawback is

that it is not always accurate when further translated into platform-specific metrics such as latency. To overcome this problem, *Sommelier* prepares the inference engine runtime for each new incoming model on locally available hardware platforms (e.g., CPU, GPU, and TPU) and collect the actual performance number of the additionally required metrics (e.g., latency). According to publicly available statistics [158, 74], a small set of common types of platforms could support over 95% of all types of workloads in a large company. This confirms the feasibility of using a small set of hardware in the *Sommelier* runtime to support platform-aware metrics besides FLOPS [102].

4.5 Implementation

We implement *Sommelier* as a standalone query engine taking the existing model repositories as its data connectors. The implementation consists of around 6000 lines of C++ and CUDA code for neural network graph and operator definitions, semantic equivalence assessment, and query processing, as well as around 1000 lines of Python code to import and export DNN models between *Sommelier* and ONNX format [173], a universal neural network model representation compatible with all mainstream frameworks such as TensorFlow and PyTorch. In particular, the module to assess semantic equivalence can be separated out as a common library for model analysis. We plan to open source the code.

APIs. *Sommelier* connects with a DNN model repository specified by the user during initialization. *Sommelier* further exposes a `query()` API in place of the original interactions between users and the model repository. It takes a query command (syntax shown in Figure 4.6) as the input and returns a list of selected DNN models, or `null` if none satisfies all the query predicates. The emphasized terms are supplied by users or other DL framework components (e.g., inference serving systems). The ref-model is the name (or ID) of a reference model (when left empty, a default model is chosen based on the type of inference task), threshold is the semantic equivalence threshold, and the optional exec-spec outlines additional execution settings (e.g., hardware information, running mode, and batch size) in key-value pairs to help building DNN resource usage profiles.

Porting to other DL frameworks *Sommelier* can be easily ported to interface with different DL frameworks. Neural network representations are interchangeable between different frameworks via ONNX [173]. Further, existing model repository APIs are mostly equivalent. For instance,

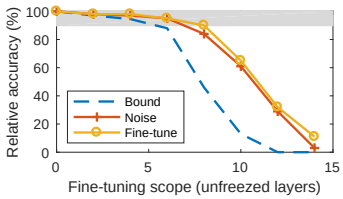
```
query = “CORR” ref_model
        “ON” ( attr (“>”|“<”) threshold )+
        “ORDER BY” ( metrics(attr) )*
        “WITH” ( device_spec )*

attr  = “SEMANTIC_CORR” | “MEM” | “CPU” | “LATENCY”

metrics = “MAX” | “MIN” | UDF
```

Figure 4.6: *Sommelier* query syntax.

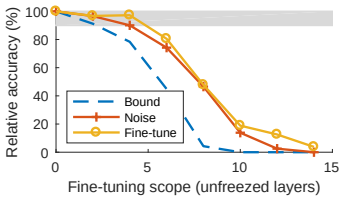
loading models from TF-Hub and Pytorch Hub needs a call of `torch.hub.load(path, name, pretrained)` and `tfhub.KerasLayer(model_url)` respectively. Hence, only 3 lines of configuration change is needed to migrate *Sommelier* across model repositories.

4.6 Evaluation

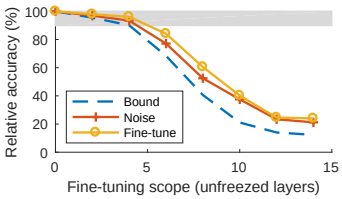
The key to the *Sommelier* performance is to build the semantic and resource indices effectively and efficiently, which in turn depends on the algorithms in Section 4.3. Therefore, our goals here are to (i) evaluate the algorithms; (ii) show how the query system can be used for the use cases outlined in Section 4.1.1; (iii) use TF-Hub as a case study to evaluate the index structures and analyze what they reveal about the models; and (iv) evaluate the overhead of various operations.



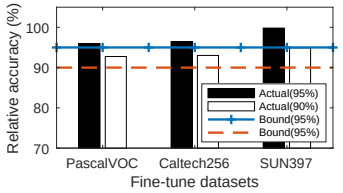
(a) Diff fine-tune scopes (recognition).



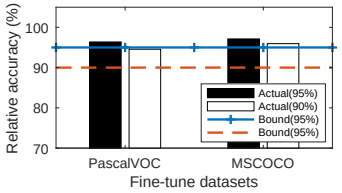
(b) Diff fine-tune scopes (detection).



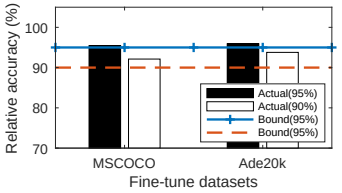
(c) Diff fine-tune scopes (segment).



(d) Different datasets (recognition).



(e) Different datasets (detection).



(f) Different datasets (segmentation).

Figure 4.7: QoR bound and actual QoR loss given varying levels of fine-tuning and datasets.

4.6.1 General setup

DNN model benchmarks. To evaluate the semantic equivalence algorithms as well as the *Sommelier* query system, we prepare two sets of DNN models: (i) a synthetic repository of models we generate ourselves, transferred from six widely used pre-trained models: three for vision (*image recognition* [148], *object detection* [161], and *semantic segmentation* [162]), and the other three for NLP (*sentiment analysis*, *question and answering (Q & A)*, and *named entity recognition*) [7]. This gives us fine-grained control in terms of different semantic equivalence levels to extensively evaluate the algorithms; (ii) We also use some TF-Hub model collections to show how *Sommelier* indices perform in realistic settings (Section 4.6.4).

Datasets. We use a few widely-used datasets to tune, validate, and assess semantic equivalence between models: ImageNet [34], Caltech256 [174], and SUN397 [175] for object and scene recognition; PascalVOC [176] and MSCOCO [177] are used to fine-tune object detection; Ade20k [178] to fine-tune segmentation; SQuAD1.1 [150], IMDB [179], and CoNLL03 [180] to fine-tune Q & A, sentiment analysis, and named entity recognition workloads respectively.

Hardware. A Linux server with a quad-core 2.3 GHz Intel Xeon CPU, 64 GB memory, and an NVIDIA RTX2070 GPU is used to evaluate *Sommelier* model query system. This covers inference scenarios broadly, since they are run on a single server whether at the edge or in the cloud.

4.6.2 Assessing semantic equivalence

We first quantify how well *Sommelier* captures semantic equivalence between models, reflected in two metrics: (i) whether the proposed quality of result (QoR) lower bound threshold is *reliable*, i.e., always below the actual QoR; (ii) whether the bound is *tight*, i.e., not far from the actual QoR so as to avoid missing actually equivalent models.

Semantic equivalence between model segments. Recall that we are concerned with model segments in the case of transfer learning. We use the three CV workloads, whose models are all transferred from the pre-trained Resnet50. We fine-tune the three models with different task-specific datasets and by freezing different numbers of base model layers. Separately, we derive additional reference models by adding noise to each fine-tuned model to mimic a worst-case fine-tuned result. Then, we replace the transferred part of the newly tuned model with the counterpart in the original one, and evaluate the relative accuracy (%) based on the absolute accuracy before replacement (normalized to 100%). This relative accuracy is compared with our accuracy lower bound derived from the original and fine-tuned models.

Figures 4.7a to 4.7c each plots three lines. The y-axis shows the relative result quality (QoR) of the "partially replaced" model with respect to the fine-tuned model. The dashed line corresponds to the estimated low-bound accuracy in our algorithm, whereas the two solid lines ("noise" and "fine-tune") reflect the actual accuracy relative to the normal and worse-case (by adding random noise) fine-tuned models. It illustrates that our algorithm generates lower bounds that are reliable and, more importantly, closely tracks the actual accuracy when the actual accuracy is less than 10% (shaded region), which covers the operation region for practical usage. Figure 4.7d to 4.7f further justifies the tightness of the bound given by *Sommelier* as it is consistently close to the actual QoR with at most 4.5% relative difference under various fine-tune datasets.

Whole model semantic equivalence. Now we examine the QoR (i.e., accuracy) bound performance for whole models. ResNet50 is selected as the reference model, for which the bound is calculated (under different validation dataset size) with respect to Inception-V3, VGG19, and MobileNet, all three achieving the same image recognition functionality. The actual accuracy while interchanging these models for their tasks is measured 20 times with the same validation datset size and we compare the accuracy lower bound with the lowest and average actual accuracy value. Table 4.1 justifies the

Table 4.1: Lower bound vs actual accuracy(%). A cell (X/Y/Z) reports "bound/min/average" of actual accuracy.

Data Size	InceptionV3	VGG19	MobileNet
100	54 / 64 / 72	54 / 66 / 75	50 / 57 / 69
1k	62 / 68 / 72	62 / 70 / 75	58 / 66 / 69
10k	67 / 70 / 72	70 / 72 / 75	62 / 66 / 69

bound is safe and further shows that the accuracy bound is increasingly close to the actual accuracy when larger validation dataset is used. When over 1000 records are used for validation, the bound is around 10% close to the actual accuracy.

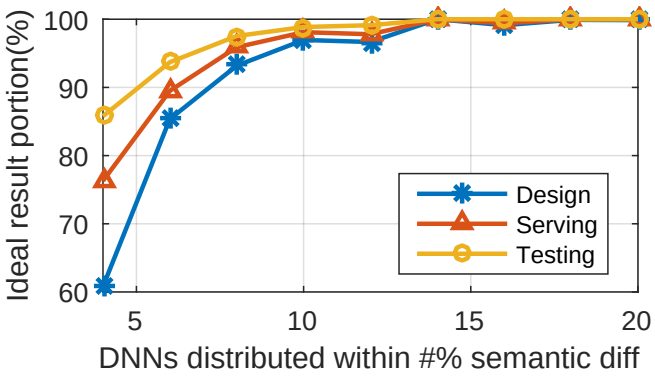
4.6.3 End-to-end performance

Settings. We use the three aforementioned motivating examples (Section 4.1.1), i.e., model design, model testing, and inference serving, to evaluate *Sommelier* in an end-to-end fashion. The first two examples represent offline scenarios, and the last example represents online scenarios. The evaluation setups (e.g., models, workloads, and datasets) are the same as described in Section 4.6.1. These cover both similar whole models and model segments. *Model testing* also borrows the model settings specified in DeepXplore [163].

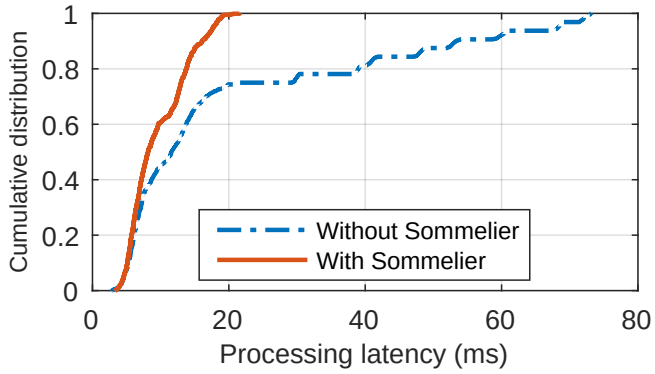
Performance metrics. For offline scenarios (i.e., model design and testing), we care about the *query quality*, namely whether the portion of DNN models selected by *Sommelier* is the ideal selection. For online scenarios (i.e., inference serving), we examine the *inference latency* of the serving engine using a vanilla model repository (which loads a single model statically) versus a repository backed by *Sommelier* (which lends to adaptive model selection to runtime dynamics). Finally, we compare the *time and manual effort*, in lines of code (LoC), needed for *Sommelier* versus manually.

Query quality. Figure 4.8a shows the portion of query output models matching the ideal model. Even when all models are semantically different from one other by at most 4% (the most extreme case where all models are "usable"), *Sommelier* still consistently returns the ideal one for over 60% cases. When model differences are distributed evenly between 0% to 10%, *Sommelier* returns the ideal model for over 95% cases.

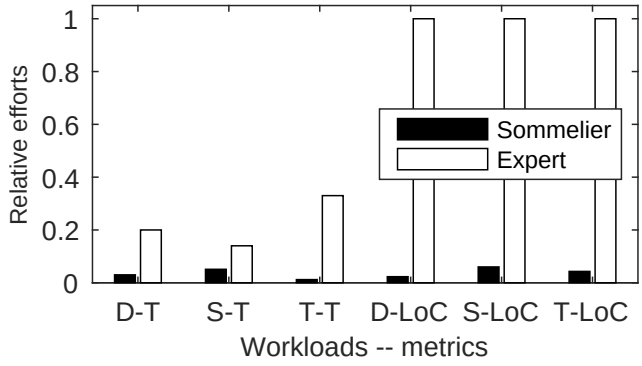
Inference latency. Figure 4.8b shows that, when *Sommelier* is used with an inference server, it could cut down the heavy tail (90-percentile) latency of inference tasks by over 6× by switching models



(a) Query quality (*Sommelier* vs ideal).



(b) Runtime inference time CDF.

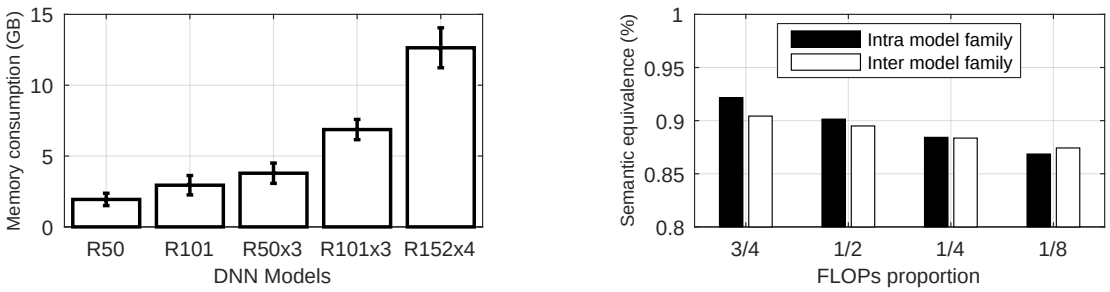


(c) Time and manual efforts saving.

Figure 4.8: End-to-end performance.

automatically, which is not possible for current model repositories.

Time and manual efforts. Finally, comparing to manually achieving such performace, *Sommelier* significantly reduces the time consumption and lines of code needed. Figure 4.8c shows the relative time and LoC needed, normalized with respect to the non-expert user (e.g., a sophomore student without deep experience of Deep Learning). *Sommelier* reduces the profiling time needed by up to 30×, and replaces hundreds LoC of script writing with 10 lines of *Sommelier* queries.



(a) Resource variance.

(b) Semantic equivalence.

Figure 4.9: Resource and semantic index effectiveness.

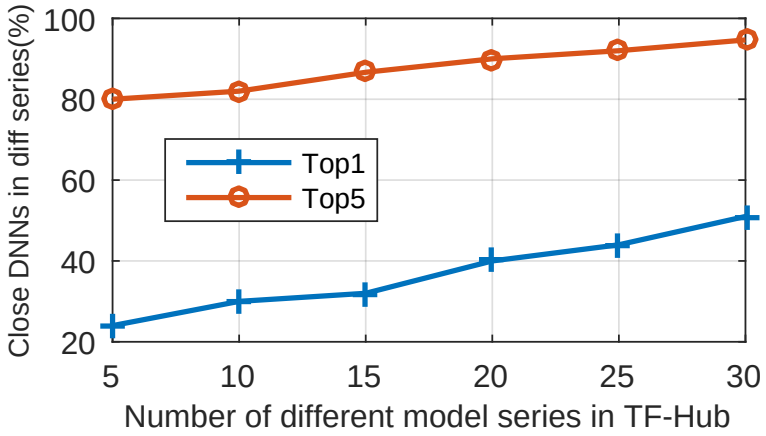


Figure 4.10: Cross-series DNN correlation.

4.6.4 Tensorflow Hub case study

We conduct a case study of 163 DNN models belonging to 30 series in Tensorflow Hub [17] to illustrate how well *Sommelier* uncovers hidden correlation between models. Each series is a family of models derived from a common basis. Consider two widely used state-of-the-art model series for image classification, BiT [108] and EfficientNet [22]. They include a sequence of 5 and 8 models respectively with increasing resource consumption (from tens to hundreds million paramters) and accuracy (from 74% to 86%). Currently, manual model selection is done intra-series for simplicity, not considering interchangeable models in different series.

First, Figure 4.9 zooms into organization across BiT and EfficientNet only. For the resource index, Figure 4.9a shows the variance of the memory consumption for BiT models relative to their "standard" memory usage. Under different execution settings (e.g., GPU specification and batch size), the memory consumption can vary by 25% but *Sommelier*'s resource index avoids such inaccuracy without exhaustive profiling. For the semantic index, we use the largest version (the R152x4 model)

Table 4.2: Time of gauging semantic equivalence.

Metrics	Alexnet	ResNet	VGG19	BERT
# Params (M)	62	60	143	340
Time (Seg)	1.89s	2.77s	5.46s	14.10s
Time (Whole)	1.25s	4.46s	6.18s	22.92s

in the BiT collection as the reference model, and measure the semantic equivalence between the reference model and any model in BiT and EfficientNet collections with a similar resource profile. Figure 4.9b surprisingly shows that when choosing a model 8× smaller to replace the R152x4 model, the better one is *from EfficientNet*, not from the same BiT collection. This is hard to identify manually.

Next, we incrementally index more series, eventually to all 163 models. For each model, we identify its topK semantic equivalents (i.e., the models with the K highest semantic equivalence scores) within and across model series. In Figure 4.10, the x-axis shows how many randomly selected series are indexed, and the y-axis shows the portion of series with models having the topK semantic equivalents *outside* their own series (repeated 5 times). On average, up to 40% and 80% series find top 1 and 5 semantic equivalent DNNs *in another series*. This suggests the extent of hidden correlation is widespread, highlighting the value of automatic semantic assessment in *Sommelier*. Further considering partial model equivalence relations, the aforementioned percentage went above 50% and 90% respectively even with less than 5 series.

4.6.5 *Sommelier* system overhead

Sommelier introduces several query operations during run time. In this section, we profile them individually.

Latency of semantic equivalence detection. Recall that *Sommelier* assesses semantic equivalence between models offline (Section 4.3), which is not on the critical path of processing inference workloads. We mainly consider whether *Sommelier* can handle huge DNN models. We use four models (Table 4.2 column titles) as the inputs to test run the whole model and model segment equivalence detection algorithms respectively. Table 4.2 shows the time needed. We can clearly see that the algorithm scales well when the model size is extremely large. Even for the huge BERT model which would consume over 12 GB of memory during run time, our algorithm still finishes

Table 4.3: Runtime query latency (ms).

Predicate	Num of records			
	100	1K	10K	100K
Resource	0.22	0.54	1.63	4.32
Semantic	0.01	0.03	0.04	0.06
Both	0.24	0.61	2.30	6.69

Table 4.4: Memory footprint (MB) with the indices.

# Models	10	100	1k	10K	100K
Resource	0.001	0.008	0.091	0.87	6.5
Semantic	0.006	0.58	23	55	71

within around 20 s, reasonable for offline index insertion.

Latency of runtime queries. The query operations are on the online path (Section 4.4). The main latency overhead is searching the LSH-based resource index and the 2D pair-wise DNN semantic index. This is relatively slower than the extremely fast (ns level), hash-based search for current, naive DNN model lookups. We prepare the model repository with different numbers of models varying from 100 to 100K. In each case, the storage is queried 20 times, and we time the average search latency when given *either* a resource or semantic constraint alone, as well as when given both constraints. Table 4.3 shows the average query time versus storage size. The query is fast enough even considering both searching predicates. In practice, the repository size needed is mostly smaller than 100K model records, where around 6 ms is the typical retrieval latency, even orders of magnitude lower than the actual processing time of a typical inference task.

Memory overhead of *Sommelier* indices. Since *Sommelier* leverages index structures to track the semantic correlation and resource profiles of DNN models to accelerate query processing (Section 4.4), additional memory consumption is therefore inevitable. However, this should be negligible since only the metadata of the models need to be kept in-memory, whereas the model itself still resides on disk. Table 4.4 shows the added memory consumption incurred by randomly picking different numbers of DNN models and building the two index structures. The additional memory footprint is mostly under 80 MB, indeed negligible compared to the memory capacity of modern hardware. This also leaves space for further caching the most frequently used models in memory to further mask the model loading latency from a (remote) disk.

4.7 Related Work

We are not aware of any DNN query service based on their model semantics and resource consumption profiles. Our work takes a leaf out of theoretical work on DNN semantic analysis and explanatory queries in database literature.

Semantic analysis of DNN models. Until recently, there was little consideration of harnessing the statistical nature of DNN [181]. Some relevant works studied the functional semantics of DNN models to ensure neural networks are robust, safe, and interpretable. Reluplex [166], AI2 [167], and others [182, 183] focus on DNN model semantics robustness against adversarial inputs using SMT solvers and other verification techniques. Manifold [138], DeepTest [184] and DeepXplore [163] validate DNN robustness solely via iteratively refined test datasets. However, they all verify local properties (e.g., adding perturbation to an input) of DNNs, and hence not applicable to assess task-level semantics between models. Recent work on interpretable AI, such as OMG [185] and ARG [186] leverage continuity features of the input and human visual perception respectively to *explain* why and how a given model matches the functional semantics of a DL task. Instead, *Sommelier* provides the "inverse" function, serving the optimal DNNs to the users with DL tasks and performance goals at hand.

Explanation query engines. Recent work in the database literature has explored the functionality of the databases to detect causality, analyze internal correlations of the data, and answer explanation queries for the users. These include theoretical analysis frameworks [187, 188], relational interfaces for explanation [189], and optimizations for specialized data types (e.g., performance traces [190], and error logs [191]). Although these techniques are agnostic to the specific data type, they lack the capability to extract the relevant information from DNN models to effectively handle explanation queries. *Sommelier* develops essential tools for this purpose.

4.8 *Sommelier* summary

DNN model repositories have become indispensible players in today's machine learning ecosystems. However, existing model repositories require the user to profile and identify precisely which model to use.

Instead, we propose *Sommelier*, an indexing and query system over typical DNN model repositories, using a novel primitive to quantify semantic equivalence between DNN models. *Sommelier* is built as a standalone query engine that can interface with an existing repository. Extensive evaluation shows that *Sommelier* can identify the ideal model for over 95% of the queries, and reduce the 90th percentile tail latency of inference tasks by a factor of 6 when interfaced with an inference server for runtime model switching.

We believe *Sommelier* is a promising approach to expand the utility of model repositories with minimal learning curve and manual efforts, paving way for future system optimizations. Looking ahead, the DNN semantic analysis in *Sommelier* could further contribute to explainable AI.

Chapter 5

Computation Reuse Service for Deep Learning Inference

5.1 Overview

5.1.1 Contextual data driving DL inference at the edge

Emerging learning-based mobile applications increasingly interact with the environment, process large amounts of sensory input, and assist the edge device users with a range of tasks. For example, a personal assistance application can "see" the environment and generate alerts or audio information for visually-impaired users [192]. A driving assistance application [193] can render 3D scenes overlaid on the physical environment to help the driver to visualize the surroundings beyond the immediate views by understanding the semantics of the surrounding environments. A smart home application [48] can parse audio commands to control home appliances and/or translate natural language to search useful information. A smart agriculture solution consists of a fleet of autonomous vehicles collaboratively maintain the health of the crops (e.g., watering the crops if they appear dehydrated) by capturing the images and other sensor data (e.g., ambient light intensity, humidity, and temperature), analyzing the crop status, and deciding the actions accordingly in real time. The core logic of these applications, namely deep learning inference on the complex contextual data, are usually computation-intensive and latency-sensitive, while running on resource-constrained devices.

The standard approaches to resolving these challenges involve either offloading these computation

to a cloud(let) [194, 195, 79, 196, 12, 197] or applying local system optimizations to speed up the on-device processing [198, 199]. The former depends on the network bandwidth availability and meanwhile suffers from unpredictable communication latency affecting the real-time performance. The latter often trades off computation quality for faster response, yet still not sustainable as the power growth of the mobile processors do not scale with the dramatic growth of the learning-based application complexity [74].

5.1.2 Redundancy among DL inference workloads

Note that these aforementioned DL inference applications often operate on similar, correlated input data and share common processing components, both within the same (type of) applications and across different ones. While the input data are rarely the same, they share temporal, spatial, and semantic correlation due to the overlapping contextual data collected or the similar functional requirements of the applications. A closer look at these applications suggests there is widely existing redundancy in such computation across applications (Section 5.2.1), edge devices (Section 5.3.1), and different DNN models (Section 5.4.1). Applications with the similar functionality are used by multiple devices over time, operating on a similar context (e.g., common physical locations).

Eliminating such redundancy across applications and devices is a promising direction to optimize the resource efficiency of running DL inference tasks on resource-constrained edge devices, ultimately achieving low latency, efficient resource usage, and high accuracy at the same time.

However, there is a defining difference between our cases and traditional redundancy elimination - *exact matching is no longer the criteria to define redundancy*. Instead, the most common input types, i.e., images, speech, and sensor reading, come from analog sources. The input values are rarely exactly identical, but correlated temporally, spatially, or semantically, and mapped to the same output leading to redundant processing.

5.1.3 Missing service abstraction: caching and computation reuse

Apparently, there is a missing piece in the existing DL inference execution facilities that could cache and reuse computation results to eliminate redundant computation between DL inference workloads (with similar input data and/or running on similar DNN models), across applications and/or edge

devices. To build the missing services requires resolving the following issues: 1) Existing primitives fail to capture the characteristics of *fuzzy* computation redundancy; 2) There is no algorithm and data structure that achieve computation deduplication and reuse approximately based on semantic correlation on the input data and the computation logic; and 3) Although it is feasible to develop application-specific solutions to capture and eliminate the redundancy, the engineering efforts are largely repeated for each application. On the other hand, developing generic solutions needs careful design to decouple yet bridge the application-specific metrics and system-generic computation reuse workflow.

5.1.4 Solution overview

Aforementioned issues all necessitate a generic service abstraction to outsource the common complexity among all applications, supporting caching and computation reuse based on the approximate semantic relations between the inference tasks. The rest of this chapter will delve into the details of three works *Potluck* (Section 5.2), *FoggyCache* (Section 5.3), and *DeCor* (Section 5.4). Algorithmically, these works propose new abstractions, algorithms, and data structures to harness the semantic correlation between input data and the computation logic to eliminate the redundancy between the inference tasks. Further, they jointly serve as caching service on the edge execution engine. With these service abstractions, edge developers could now solely focus on optimizing their core application logic, without worrying about caching and memoization issues.

Specifically, *Potluck* [38] and *FoggyCache* [39] both capture the inference task redundancy by gauging the input data correlation. We will first explain *Potluck*, which proposes the algorithms to measure inference task similarity based on input data correlation, and designs the first caching prototype that achieves approximate computation reuse *between applications*. Based on the approximate cache design of *Potluck*, we will explain *FoggyCache*, which further refines the algorithms and develops cache synchronization mechanisms and other techniques to achieve approximate computation reuse *across different edge devices*.

DeCor further extends the computation reuse scheme to detect redundancy between inference tasks running on similar DNN models. It is built on the algorithms of measuring the semantic correlation between DNN models explained in Section 4.3.

5.2 Cross-Application Approximate Computation Reuse

In this section, we present *Potluck*, a cross-application approximate deduplication service to achieve the above goal. *Potluck* essentially stores and shares processing results between applications and leverages a set of algorithms to assess the input similarity to maximize deduplication opportunities, as detailed in Section 5.2.2. We carefully design an input matching algorithm to improve the processing performance without compromising the accuracy of the results. *Potluck* is implemented as a background service on Android that provides support across applications (Section 5.2.3). Extensive evaluation shows that our system can potentially reduce the processing latency for our benchmark augmented reality and vision applications by a factor of 2.5 to 10 (Section 5.2.4).

In summary, we make the following contributions:

First, we highlight deduplication opportunities across emerging vision-based and AR-based mobile applications. These arise from various sources of correlation in their input, common processing components they leverage, and the co-installation of these applications.

Second, in view of the opportunities above, we propose a set of cross-application approximate deduplication technique to achieve both fast processing and accurate results. To the best of our knowledge, this is the first such attempt.

Third, we build *Potluck* as a background service. Extensive evaluation confirms its benefit is significant, accelerates the inference execution by an order of magnitude without sacrificing accuracy.

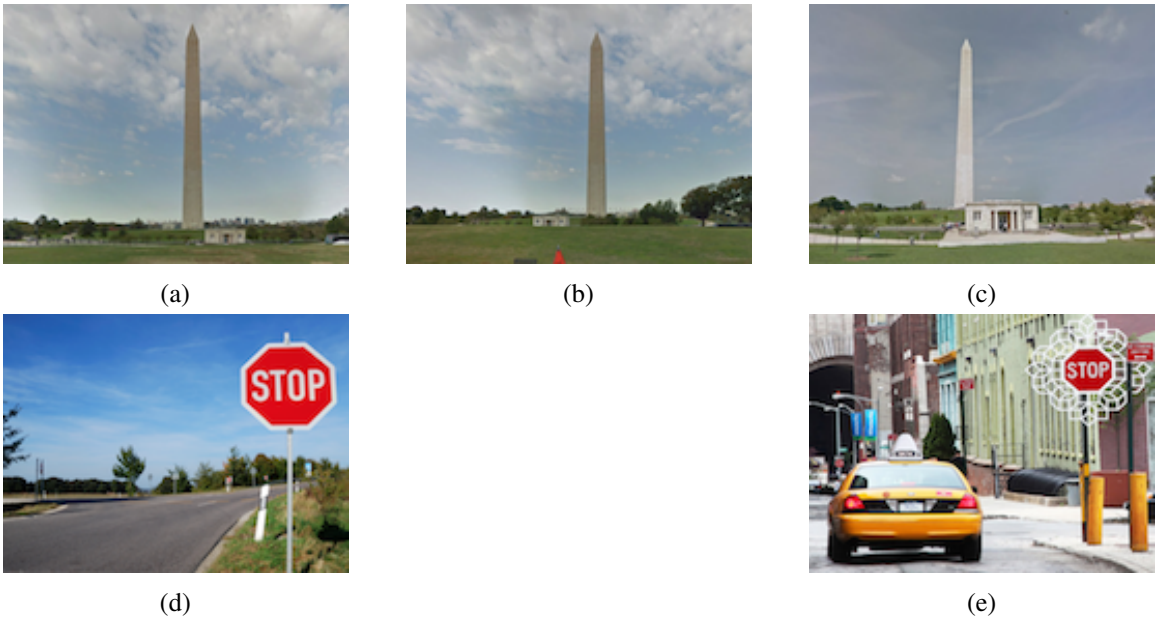
5.2.1 Motivation

5.2.1.1 Motivating applications

Among the fastest growing applications, vision-based cognitive assistance applications and augmented reality (AR) based applications are two representative categories.

As an example cognitive application, Google Lens [32] continuously captures surrounding scenes via the camera, recognizes objects using deep learning techniques, and then presents related information to assist the user. These applications increasingly provide personal assistance.

On the other hand, AR applications such as IKEA Place [200] for home improvement, Google's Visual Positioning System for indoor navigation [201], and PokeMon Go [202] blend the virtual and physical experience. They overlay 3D graphic effects on real world scenes to enrich and enhance the



(a) (b) (c)

(d) (e)

Figure 5.1: (a) and (b) are two snapshots taken successively along the same road 136 m apart in October 2016. (c) is taken at a similar location but in August 2014. (d) and (e) are captured in completely different places at different times, but both prominently feature a stop sign.

interface between human eyes and the physical world.

Common themes. Most of these are lifestyle applications. According to the measurement study of the smartphone usage [203], there is a high probability of such applications being co-installed, *even though they may not be running simultaneously*. Further, they often operate in similar physical environments, share common processing steps, and map a group of similar input values to the same output. We discuss these in detail next.

5.2.1.2 Input correlation and similarity

The above applications all take input from the environment or some context, directly or indirectly. Such input exhibits similarity, within an application or across applications, due to the activities of the mobile user showing spatial and temporal correlation.

Temporal correlation. We can view the combined video input to all the applications as a continual camera feed. In other words, assuming there is a never-ending centralized camera feed to the mobile device, different applications simply take a subset of the frames as needed. From standard video analysis, significant temporal correlation exists between successive frames because the scene

rarely changes completely within a short interval, and this has been leveraged extensively in video compression. In most cases, the main objects of interest in these scenes are slightly distorted versions of one another by some translation and/or scaling factor.

Spatial correlation. It is common for humans to follow along recurrent trajectories, for example, due to their regular commuting schedules or frequenting a favorite restaurant from time to time. Therefore, there is some level of recurrence of the scenes obtained as part of those activities, though potentially taken from different view points and partially different environments, such as different lighting conditions and surrounding backgrounds. The actual images might show different color bias, for example. Such correlation can be identified using SURF [204] like approaches.

Semantic correlation. A further situation arises when the same object or the same type of objects appears in completely unrelated background scenes and at different times. For example, when a road sign is detected at different places and times, regardless of the exact sign, a driver assistance app simply generates an alert. Since many applications interpret the scene to related abstract notions of objects or faces, many seemingly different images can be classified to the same category and considered semantically equivalent.

Similar but not identical. However, these correlated input frames are rarely exactly the same, for various reasons. In some cases, the scene is actually changing (e.g., the user walking or driving along a street). In other cases (e.g., approaching the same intersection from different directions), we get more or less the same scenes, but at different view angles. More generally, there might be distortion across frames due to image blur (different focus or motion-induced blur).

Correlation in the results. Generalizing the semantic correlation, these similar input values are often mapped to the same output values in the aforementioned applications, due to the resolution of the results. For example, adjacent pixels in an image may be mapped to the same feature details. Image recognition functions may attach the same label to different images. For an AR application, there is no need to render a new scene if it is visually indistinguishable to our eyes from a previous one.

Examples. Figure 5.1 illustrates these similarities. The images are taken from Google Street View. Images (a) - (c) could be perceived “the same” by a Google Len like app for showing the Washington Monument, whereas images (d) and (e) show “stop sign”.

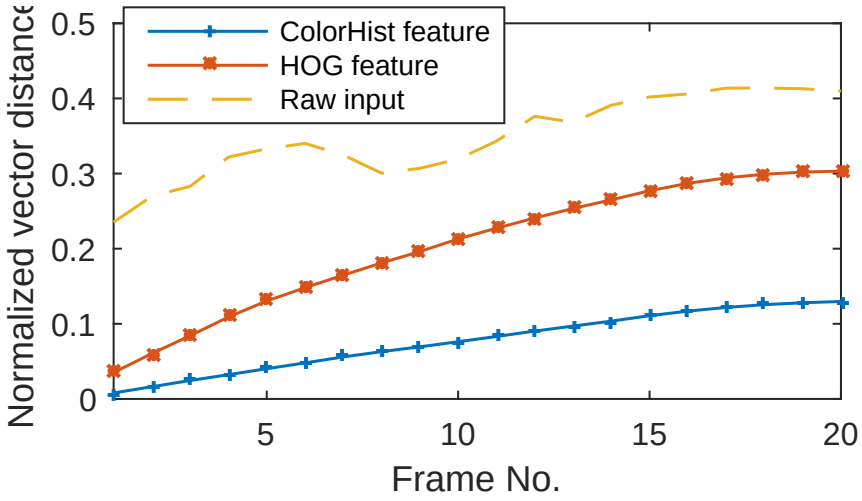


Figure 5.2: Similarity between frames

As another example, we select a video segment from an HEVC test dataset [205], and compute several features (color histogram [206], HoG feature [207]) for consecutive frames. Figure 5.2 shows the relative differences between the first and later frames, calculated as the Euclidean distance between the normalized vectors of the matched features. Shorter distances indicate higher levels of similarity, although there are no universal criteria to define similarity levels. The features show consistent correlation levels across a long sequence of frames whereas the raw images do not.

5.2.1.3 Common processing steps

As noted previously [77, 19], many computer vision applications share similar, incremental processing steps. However, we also observe common processing steps in other types of applications (e.g., speech recognition) and across different types of applications (e.g., vision and AR applications).

Figure 5.3 shows the schematic processing flows for a cognitive application (Google Lens), and two AR based applications, *IKEA place* (as an example of AR shopping application) and *indoor navigation*.

The indoor navigation app first recognizes the environment in the input image feed, which essentially invokes the object recognition procedure. This is also the core step of the Google Lens cognitive assistance app. Similar situations could be very common, since AR application logic typically starts with understanding the spatial context. Therefore, AR applications can share essential recognition functions with image recognition apps.

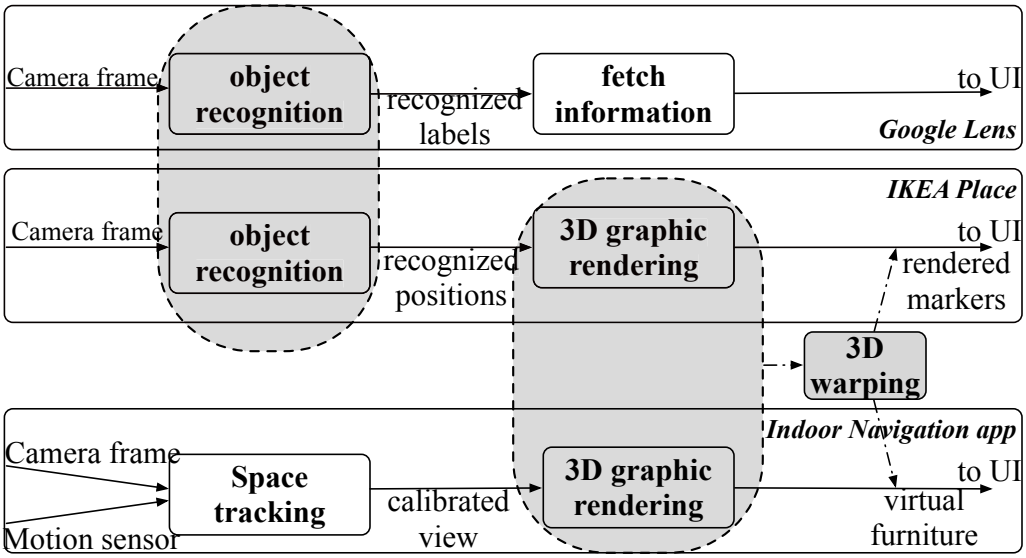


Figure 5.3: Schematic processing pipelines for three apps.

The two AR applications both require 3D graphic rendering. IKEA place would render virtual furniture at certain positions to visualize a furnished room, while the indoor navigation app would render a virtual map of merchandise to help direct customers. When the latter takes place in a furniture shop, the rendering logic would be essentially the same as what is needed for IKEA place.

Common functions are also used in non-vision based applications. For example, two location based applications can share the processing for GPS data or related contextual information close in time. A call assistant might use the mic to capture the audio to identify the location and ambient environment to determine whether to mute the call [208]. Similarly, the same procedures can be used for home occupancy detection as part of smart home management to determine whether to turn off the lights and turn on the alarm.

More generally, emerging learning-based application ecosystems further presents common APIs and libraries, and the possibility of sharing common processing steps between applications. For instance, Alexa Skills [48] enable developers to deploy various services on smart IoT devices like Amazon Echo. Deep learning frameworks such as Tensorflow [83] provide high-level programming models for app developers. Services and applications within such ecosystems leverage the same human-device interface, processing pipelines, and the underlying implementations to capture the input, understand the context, and execute tasks.

5.2.1.4 Opportunities and challenges

Given the similarities between applications discussed so far, *deduplication* is a natural approach for performance optimization. As long as these applications collectively are used frequently, there is significant potential for deduplication. For example, the home occupancy assessment and home personal assistant are usually used multiple times throughout a day; the Google Lens and indoor navigation are both likely to be used daily or more frequently.

Note that these "sharing" applications do not need to be run concurrently. Deduplication works as long as the previous results are still cached, and the interval could easily be days or longer provided there is enough space to store the cached results.

Challenges. In order to effectively deduplicate the processing across applications, we need support for identifying the equivalence between input values, cross-application sharing, and appropriate cache management criteria.

Since the input images are rarely the same, we need to be able to quantify and assess the extent of similarity between them, based on the semantics of the function.

The deduplication opportunities may straddle application boundaries, so we need a service shared between applications. This will naturally support in-app deduplication as well, though incurring a slight overhead by crossing the application boundaries.

Deduplication means we need to cache previous results. However, since our cache serves a different purpose than those of traditional caches, we cannot manage cache entries based on the least recent access or other traditional cache entry replacement algorithms.

We address these challenges by designing a cache service, *Potluck*, shared between applications.

5.2.2 *Potluck* System Design

5.2.2.1 Overview

Potluck caches previously computed results to provide approximate deduplication across applications. The processing flow is conceptually simple. When an application obtains an input (e.g., a frame from a video feed) and calls certain processing functions, it first queries the cache for any existing results. The query proceeds in several steps. First, the input data are turned into a feature vector, which serves as the key. Second, a lookup attempt is made with the key and the name of the function

called, by matching the input key to any existing key within a given similarity threshold. If there is a hit, the cached result is returned. Otherwise, the application processes the raw input and then puts the result in the cache. Third, the put action may trigger an adjustment of the input similarity threshold. We discuss the individual steps next.

5.2.2.2 Computing the key

Definition of keys. The key is essentially a variable-length feature vector generated from the input image, such as SIFT [209], SURF [204], HoG [207], colorHist [206], FAST [210], and Harris [211]. For example, the feature vector might be a 768-bit vector to represent the color histogram, a vector of $N \times 64$ bytes to describe SURF features from the input image, or a vector of $m \times n$ bytes to represent the down-sampled version of the raw input image into $m \times n$ pixels.

An essential requirement is that the key must be defined in a metric space, in which a notion of distance can then be defined. This need not necessarily be the Euclidean distance, although it is the one commonly used.

Given the raw input (e.g., images or speech segments), as application requirements might differ, we give the application the freedom to choose the exact key generation and similarity assessment mechanisms. App developers can customize the implementation of both or select from a library of mechanisms provided within *Potluck*.

Converting the raw input to a feature vector is important, because this step can eliminate noise in the raw data, "homogenize" inputs with different formats and scales, and save space when storing these items.

5.2.2.3 The usefulness of cache entries

The *Importance* metric. Conventional caches operate within a single application, where the entries store frequently accessed data. The number of data access attempts simply reflects the value of the data and determines whether the data should be retained.

Instead, our cache is different, since not all cache entries of computation results are created equal, and the variance across applications is even larger. The access frequency is the only factor that determines the value of the cached result. Therefore, we assess the usefulness of a cache entry by a

new metric, called *importance*, computed as $computation\ overhead \times access\ frequency/entry\ size$ In addition, each cache entry is tagged with a validity period. When that expires, the entry will be automatically cleared from the cache in the background.

The *importance* value indicates how frequently an entry *has been used and might save on future computation times*, but has no correlation with the accuracy of the result. Therefore, it is only used for evicting a cache entry. The lookup operation does not take into account this value.

Calculation and update of importance. The importance value for an entry is dynamic and its recalculation happens in two cases. A `lookup()` call increments the access frequency of the fetched entry by 1, and the corresponding importance value is updated accordingly. With a `put()` call, on the other hand, a new importance value is calculated for the entry. Specifically, the *computation overhead* is calculated as the elapsed time between the `lookup()` miss and the `put()` operation of this entry, and the *access frequency* is initialized to 1. The expiration time is simply that of the overall entry, set during the `put()` call.

5.2.2.4 Querying the cache

Threshold-restricted nearest neighbor query. A query involves finding the closest match for an input key. When given a feature vector as the key, we initiate a k nearest neighbour search, iterating over all entries in the key index.

After that, we discard those returned entries whose distance from the input key vector exceeds a certain threshold. By default, to balance the lookup time and quality, we set k to 1. We experimented with a few values and find that this value provides the fastest lookup time without sacrificing quality.

Random dropout. When a cache query operation is invoked, with a probability (currently set to 0.1) *Potluck* will simply return `null` without actually querying the cache. This is a randomization mechanism to enforce a `put()` operation at least periodically. This refreshes cache entries as well as triggers a recalibration of the threshold. The latter is valuable, in case the threshold has been loosened too much, as explained next. We will discuss how to set the "dropout" probability at the end of Section 5.2.4.2.

Algorithm 1: NN-based threshold tuning algorithm

```
1  initialize threshold ← 0;
   // params are customizable
2  initialize k ← 4, α ← 0.8, z ← 100;
   Wait: z entries inserted to cache by Put operations
3  while service not terminated do
4  |  wait for new Put operation;
5  |  read (key, val) pair from the operation;
6  |  (key′, val′) ← lookup(key);
7  |  if ||key′ − key|| ⩽ threshold and val′ ≠ val then
8  |  |  threshold ← threshold/k;
9  |  else if ||key′ − key|| > threshold and val′ = val then
10 |  |  threshold ← (1 − α) × ||key′ − key|| + α × threshold;
11 end
```

5.2.2.5 Tuning the similarity threshold

The threshold controls to what extent different raw inputs are consider "the same". Part of our argument is that many raw images are similar and therefore we can avoid duplicating the subsequent processing. Clearly, there is a tradeoff between performance speedup from reusing previous results and the accuracy of the results. We manage this by adaptively tuning the similarity threshold based on the ground truth and the observation of the nearest neighbour entry, as shown in Algorithm 1.

The algorithm. The idea is straightforward. The threshold is initialized to 0, meaning no distance between input images is permitted. After caching enough entries (100 by default), the algorithm kicks into action and we then gradually increase ("loosen") or decrease ("tighten") it as needed, triggered by each `put()` operation. In general, the threshold is loosened conservatively but tightened aggressively. If the threshold is too tight, we might miss deduplication opportunities. When the threshold is too loose, the cache lookups might return false positives, i.e., input images that are not actually similar but considered so due to the threshold.

Given the new key and value to be stored in the cache, the algorithm finds the nearest neighbor in the feature vector space to the new key. Two cases should be noted. If the key distance is larger than the threshold and both keys map to the same values (line 9 in the pseudo-code), the threshold is too tight and should be loosened with an exponentially weighted moving average. Conversely, if the key distance is within or equal to the current threshold, but the keys map to different values (line 7), the threshold is too loose and should be tightened. Note that the latter case will not arise naturally. If two

keys are within the threshold, the cache query would normally return the cached result (incorrectly). Therefore we adopt the "random dropout" in the cache lookup process to artificially trigger this case from time to time, as a quality control mechanism.

Intuition and correctness. The threshold-tuning algorithm is essentially based on finding k nearest neighbors (kNN). It observes the distance between the (key, value) pairs of the nearest neighbours and compares the stored results with the ground-truth to adjust the maximum diameter of the "similar" result cluster accordingly. This diameter is then the threshold value we adopt. kNN is a widely used non-parametric, case-based machine learning algorithm, which makes no assumptions of the input data model. It has been extensively studied for decades and proven correct [212] for handling data with unknown features. In the same vein, our NN-based threshold tuning algorithm can provide reasonable hints on the correlation between input similarity and the reusability of the result even if we have no prior knowledge of the input data.

Quality of results and security considerations. While leveraging results across applications in *Potluck* can yield performance benefits, it breaks the isolation between applications. This can leave the system vulnerable to malicious apps polluting the cache by inserting spurious results.

Fortunately, the combination of the threshold-based kNN and random dropout algorithms can guarantee the quality of results (QoS) is not completely affected by a polluted cache and act as a defense mechanism against malicious apps. The protection can be further enhanced by incorporating a reputation system (such as Credence [213]) into *Potluck*. Each cache entry can be tagged with the application source. The threshold-tuning phase can then establish a reputation record for each application, and malicious apps can be identified and barred from time to time.

It is worth mentioning that sharing results in our context does not present privacy concerns. The input data tend to be derive from the contextual information for the mobile device, and hence common to all applications on the device.

5.2.2.6 Cache management

Inserting and indexing cache entries Several steps are involved to insert a cache entry (namely a `put()` operation). *Potluck* first collects the auxiliary information about the entry to compute its importance. Second, we invoke the threshold tuning algorithm (Section 5.2.2.5). Finally, we store

the key, the computed result, and the importance value.

The key is then added to the right position in the index. The processing time depends on the index data structure. However, unlike cache lookup, the indexing process runs in the background asynchronously and does not affect the application response time.

Eviction policy and expiry. The cache entries can be discarded in two ways. First, cache entries can expire, and the timeout is currently set to be an hour. Second, if the cache is full when a new `put()` request comes, the least important entry will be evicted and replaced with the new entry.

5.2.2.7 Supporting multiple key types

So far we have explained the processing flow from a single-app (single key type) perspective. In practice, different applications may prefer to map input to feature vectors of different specifications. In other words, we need to support multiple definitions of the key, or *types*. Each application should be able to perform cache operations using their preferred key types, and we automate typecasting between keys to further support cross-application deduplication.

Multi-index structure. For multi-key-type settings, we construct a cache query index for each type of the keys, so that the query index can be optimized for the unique properties of the particular key type to ensure highly efficient lookups. Cache entries generated by different applications but using the same key type will be managed in the same index.

Cache lookup. The cache lookup will take one more argument, specifying the key type being looked up. This then sends the query to the corresponding key index.

Cache insertion. Whenever a `put()` operation introduces a new key-value pair to the cache, we propagate this entry to all key indices. This triggers operations to iterate through all existing input key types, mapping the raw input to each key type, invoke the threshold tuning procedure per key index, and then insert the key to each corresponding index.

Cache entry eviction. Unlike cache insertion, cache eviction is not propagated to all indices. Instead, for each key type, the corresponding index will select the entry to be evicted and delete the key. The actual cached computation result will be cleared via garbage collection when no indices have references to it.

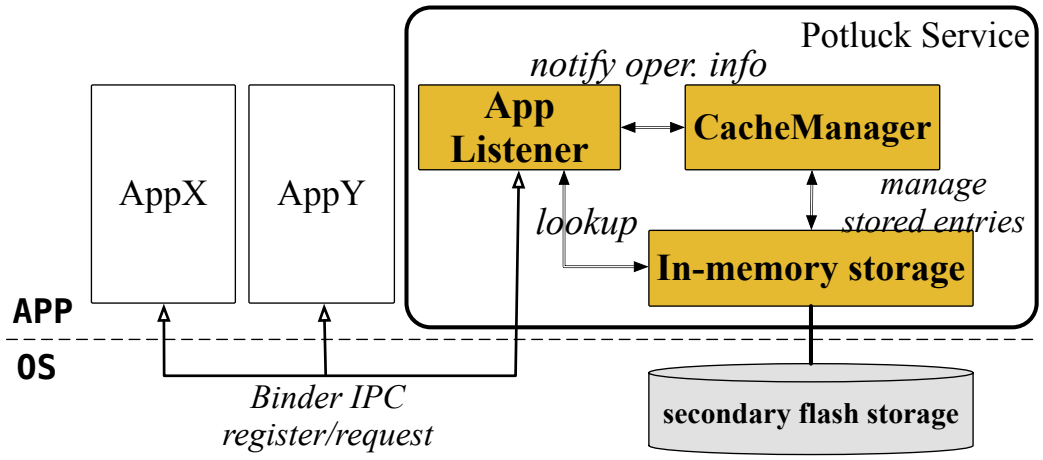


Figure 5.4: System architecture.

5.2.3 Implementation

5.2.3.1 Architecture

We implement *Potluck* as a background application level service in Android Marshmallow OS with API version 23. Figure 5.4 shows the architecture of the system.

The deduplication service consists of the following modules. The `AppListener` maintains a threadpool, handles the requests from upper-level applications, and carries out the corresponding procedures, including registering apps to the service, executing the `lookup()` or `put()` requests, and invoking the threshold tuning or reset procedure. The `CacheManager` maintains the *importance* metric of stored entries by monitoring the execution time of the functions and the access frequency of the stored entries. Based on such information, it handles the expiry and eviction in the background. The `DataStorage` is the storage layer which keeps previous computation results, and indexes the entries to speed up lookup requests.

5.2.3.2 Deduplication service

Key generation and comparison. Generally, our system supports variable-length vectors to serve as the keys. They are implemented as `Vector` instances from `java.util.Collection`, `String` instances of `java.lang.String`, or `INDArray` instances (a third-party class for fast numerical vector computation) [214]. We implement the extraction of the features mentioned

in Section 5.2.2. Most of them are already implemented in the openCV library [215], and we invoke the corresponding functions to process the input image.

By default, we support comparison and similarity measurement for scalars and vectors, as well as lexical ordering and comparison for strings.

Support for custom key definition and matching. We expose an interface to the application, through which the application can customize its own key generation and comparison logic if desired. For example, app developers can implement Mel Frequency Cepstral Coefficents (MFCC) [216] computation for an audio file and Principal Component Analysis (PCA) [217] based dimensionality reduction for high-dimensional input data. Any customized classes and methods are then incorporated via *dynamic class loading*, supported via reflection in Java. In this way, app-specific components are meshed with the system logic in *Potluck*. Our implementation leverages the `OpenHFT.compiler` open-source package [218] to achieve this.

Cache organization. Figure 5.5 shows the cache layout. There are three variables, the function called, the feature vector specification (i.e., the key type), and the value of the key, that collectively correspond to a stored result. Therefore, we organize the cache entries into multiple levels, first by the functions invoked, then by the key types, and finally the specific keys.

First, when an insertion or lookup is needed, we add or match a function. This is implemented with a `HashMap`. Note that this means only applications using exactly the same function can share results. Since the type of applications that might benefit from *Potluck* typically use common libraries (such as OpenCV or some deep learning framework), we believe the current approach is reasonable tradeoff between simplicity and effectiveness. Second, we use another `HashMap` to organize all key types corresponding to a function. Third, we use appropriate data structures to organize different key values, either a Locality Sensitive Hash (LSH) [142], KD-tree [219], treemap, or a hashmap, depending on the key dimension and how similarity assessment work. The final "values" stored are simply references (memory addresses) to the actual value stored in the memory.

A hashmap is useful for the exact matching, achieving $O(1)$ time complexity for key search. A Treemap is implemented as a balanced binary tree which supports nearest neighbor and range searches in $O(logN)$ time. Scalar or vector keys which are compared by their lexical order could benefit from using this data structure. Further, KD-trees and LSHs are data structures to support

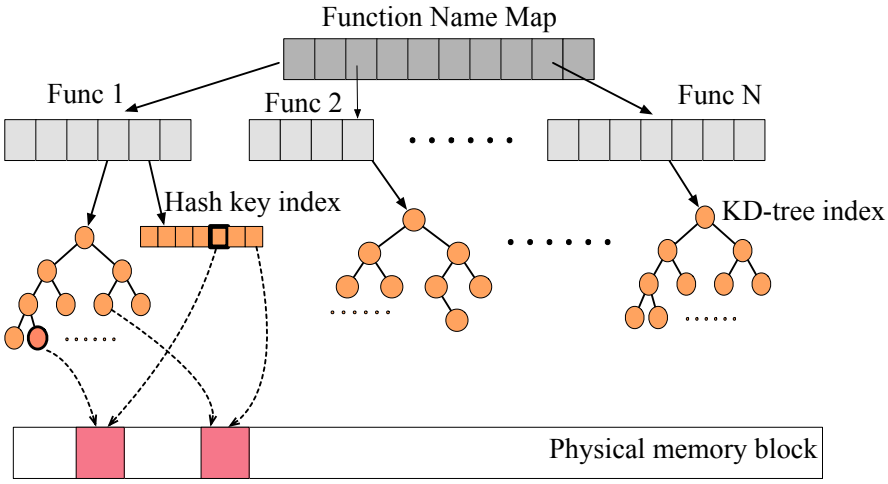


Figure 5.5: Cache layout.

spatial indexing and efficient nearest neighbor and range searches (with $O(logN)$ average complexity) for multi-dimensional vectors, where we can only calculate distances between keys but not derive a global order for them.

Cache eviction and expiry. Cache entry eviction and expiry are handled by a separate management thread running in the background. If the cache is full when `put()` is called, the management thread will iterate through all indices to find the entry with the lowest importance to be discarded.

Separately, the management thread also maintains a queue that orders all cache entries by their expiration times. This thread will be waken up when the current head item in the queue reaches its expiration time. The thread clears all (at the same time) expired entries from the cache and the priority queue, and sets the next wake-up time according to the expiration time of the new head item.

Communication between components. The communication between the apps and the deduplication service leverages Binder with AIDL [220], the IPC mechanisms natively supported by the Android OS. Interactions between the internal modules of the service are simply through shared memory with mutual exclusion locks.

The `AppListener` receives a `Request` message from an application, which consists of the request type (register or operation), function name, key type, lookup key, and computation results to store. It replies to the application with a `Reply` message containing the request type and the corresponding return values. The `AppListener` also sends the query information to the `CacheManager`.

The `CacheManager` maintains a queue of query requests. It also manages the data in the `dataStorage`, inserting new entries, evicting the least important entries when necessary, updating the *importance* value of those accessed entries, and discarding expired entries.

5.2.3.3 APIs and patches to the application code

There are two sets of APIs exposed to the application, on the control and data paths of the application respectively.

Registration on the control path. Applications start using *Potluck* with a `register()` call. This function registers a handle with the cache service, loads any custom-defined key generation methods, and initializes the application-specific key index. It also resets the input similarity threshold.

Cache operations on the data path. Applications can call `put()` and `lookup()`, two intuitive functions to insert and look up an entry. Therefore, the changes needed to leverage *Potluck*is negligible.

Discussion. Currently we need to patch the application source code to add handles to *Potluck*, but this makes sense because fuzzy input matching requires having the exact input values, not just their memory representations. Further, we want to expose some interface to the application to control the accuracy and performance tradeoff.

5.2.4 Evaluation

5.2.4.1 General setup

Application benchmarks. We built three simplified applications as benchmarks, one image recognition application and two augmented reality (AR) applications. The image recognition application includes pre-trained models and performs deep-learning based inference using the AlexNet neural network [221]. For the AR applications, one uses the current 3D orientation of the device and its location to render virtual objects, while the other first runs image recognition on the current frame in the camera view, and then renders virtual objects overlaid on the detected physical objects.

Data sets. While the above applications can run in real time, evaluating the recognition performance using real-time camera feeds is difficult, since it is impractical to enumerate all possible scene

sequences as the input for evaluation. Further, any single camera feed only captures a single scenario, and does not necessarily represent the general case. Therefore, we turn to standard datasets used to train and test image classification algorithms. In such datasets, images are crowdsourced and well calibrated, which eliminates the spatio-temporal correlation between them. In light of this, they present less favorable (i.e., more challenge) scenarios for *Potluck* than datasets collected from real applications. Experiment results from these data sets are then indicative of the worst-case performance for *Potluck*, and we can expect better performance for real applications.

We use two commonly used image classification datasets, CIFAR-10 [121] and MNIST [222], which serves as a controlled, generic scenario. We also capture several video feeds in real life to emulate real application scenarios. The comparison between the results from these datasets cross-validates our belief that the performance of *Potluck* in practice will be better than reported in this section.

The CIFAR-10 dataset consists of 60,000 32×32 color images categorized into 10 classes, 6,000 each. There are 50,000 training images and 10,000 test images. We use images within the same class to mimic deduplication opportunities where similar objects appeared in different backgrounds.

The MNIST dataset is a database of handwritten digits, consisting of a training set of 60,000 examples and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image.

We found that experiment results from the two datasets were similar, and therefore we mostly present results based on CIFAR-10, as it covers a wider range of image scenes.

Experiment environment. All experiments in this section are run a Google Nexus 5 (with a quad-core 2.26 GHz Qualcomm Snapdragon as the CPU and an Adreno 330 graphics processor) as our mobile device, running Android Marshmallow OS with API version 23. Later in the section we also use a PC (with a quad-core 2.3 GHz Intel Core i7 CPU and NVIDIA GeForce GT 750M GPU) to compare the processing times. The PC is around an order of magnitude faster than the phone.

Metrics. We evaluate *Potluck* in terms of *accuracy*, *processing time*, and *missed opportunity*.

The first two characterize the performance benefit and tradeoff of capturing the input similarity to reduce duplicate computation. The third one is analogous to the notion of *recall* commonly used to characterize machine learning algorithms. Roughly speaking, *recall* measures the portion of test

Table 5.1: Key generation time

Feature	Size (KB)	Time (ms)	Usage
SIFT	124	1568	Recognition
SURF	32	446	Recognition
Harris	35	91	Detection
FAST	28	4.6	Detection
Downsamp	1	5.8	Deep learning

data recognized based on the training data. In our case, we first characterize the optimal case for deduplication under each specific experiment setting, which defines the upperbound performance of our system, and then quantify *missed opportunity* by the gap between the performance of *Potluck* and the particular optimal case.

Since the input is the main determining factor for the performance of our system, our results are interpreted with respect to the input data setting of each experiment.

5.2.4.2 Input and key management

Key generation. We randomly select a set of 600×400 images from our dataset, and measure the time taken to generate a feature vector following different feature extraction methods. Around 500 features are detected in each image. Table 5.1 shows that generating SIFT and SURF features as the key takes orders of magnitude longer than the others but captures more information about the raw image. They are suited to recognition tasks. Harris and FAST features are based on edge detection and a good fit for object detection workloads. Detection is the first step of recognition, and the latter requires much more detailed information. *Downsamp* refers to down-sampling the raw image to fewer dimensions, which is then vectorized to be fed into deep neural networks. Since key generation is the first step to use *Potluck*, there is clearly a tradeoff between the processing time and the level of feature expressiveness required for a specific app. For our later experiments, we use *Downsamp* for the deep learning based image recognition app and FAST for motion estimation within the AR applications.

Threshold tuning. The similarity threshold determines the amount of deduplication we can achieve as well as the accuracy of the lookup result. The threshold is loosened or tightened depending on the cached entries. We perform two experiments to evaluate the tuning algorithm.

First, we investigate how many entries should be cached before we start calibrating the threshold

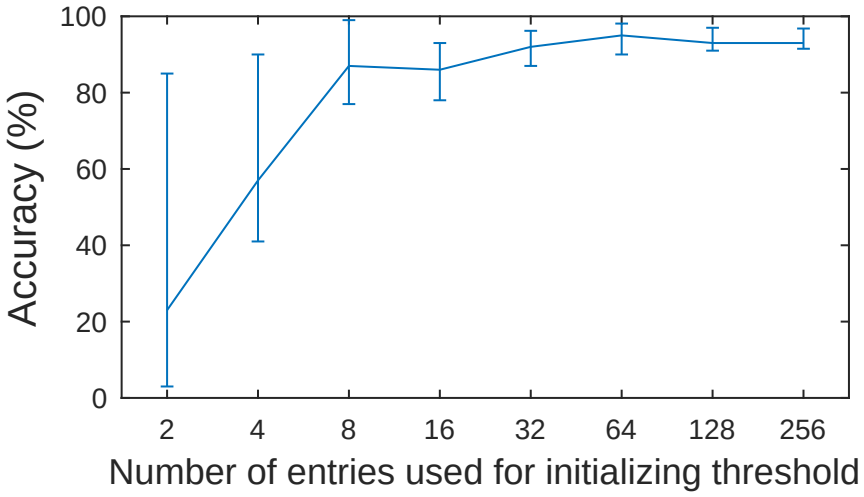


Figure 5.6: The accuracy of the similarity threshold.

and thus *enabling* the deduplication service. We consider a threshold "accurate" if the results from the cache query are similar to the ground truth. We randomly pick a variable number of images from the training set of CIFAR-10, put the recognition results into the cache, and calculate the initial value of the threshold. Then, we take 400 images from the test set and obtain the recognition results by both running the recognition algorithm and retrieving the nearest match from the cache results. These steps are repeated 10 times and we collect the average and variance information.

Figure 5.6 shows the *normalized recognition accuracy* of the threshold vs the number of cache entries used for initializing the threshold. Since the recognition accuracy without leveraging deduplication is not 100% anyway, we use that as a baseline to normalize the accuracy of our system. In other words, the y-axis shows the accuracy with *Potluck* divided by the baseline accuracy value. The line shows the average value, while the errorbars show the maxima and minima.

The accuracy stabilizes quickly as more cache entries are available. With at least 32 entries (over 1 second for a normal 30 fps video feed), the accuracy exceeds 95% with less than 5% error. The time overhead for computing a new threshold turns out to be less than 1 ms and negligible.

Second, we analyze how quickly the threshold is tightened. Recall that we loosen the threshold slowly and conservatively to minimize the possibility of false positives, but try to tighten it quickly. This is also because the threshold is loosened more frequently, invoked by each natural `put()` operation. In contrast, it is only tightened after a random dropout mechanism (Section 5.2.2.4), which happens rarely. In this experiment, we start with a certain threshold (normalized to 1), and then count

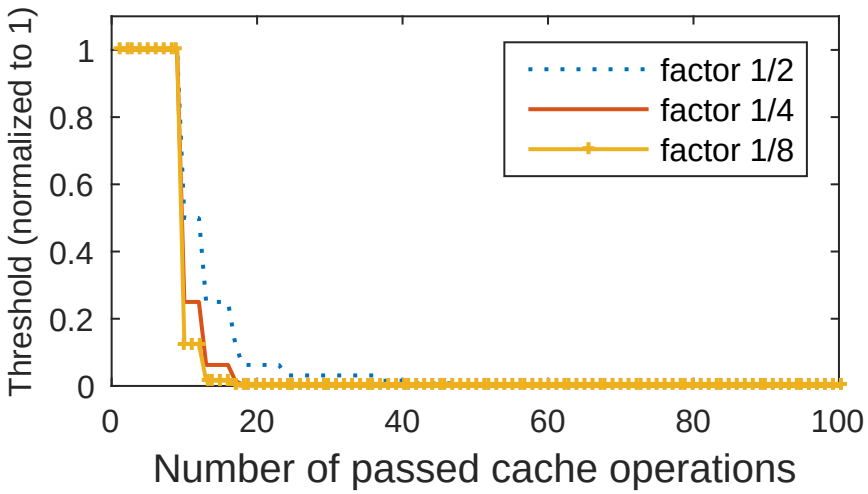


Figure 5.7: Threshold changes with lookup operations.

how many cache entries are needed to adjust the threshold to 0.

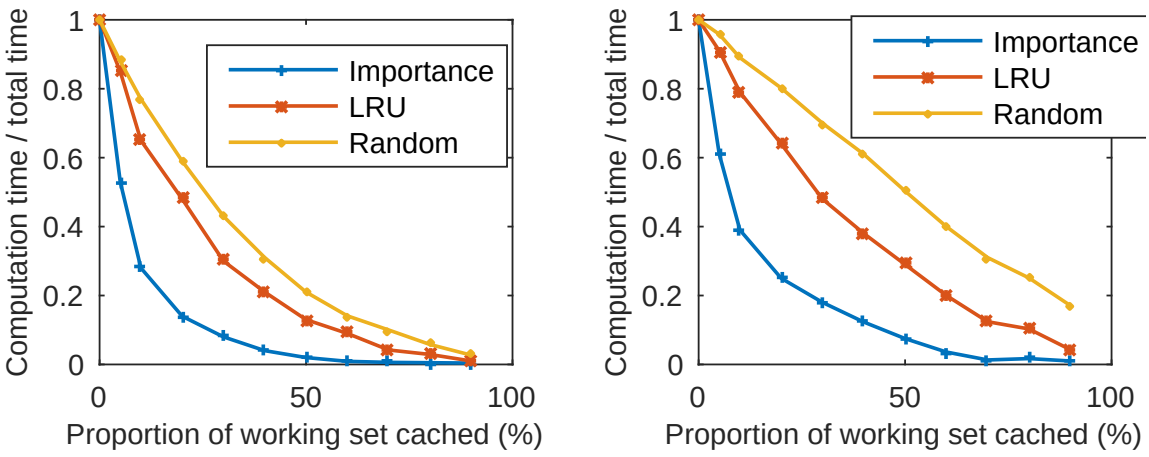
Figure 5.7 illustrates that, when the decrease factor (the parameter k defined in Alg. 1) is over 1/4 and the dropout probability 0.1 (the respective default value used), within around 20 cache operations (including `lookup()` and `put()`), the threshold shrinks by a factor of 20. With only 30 operations on average, we could further shrink it by a factor of 100. In other words, when switching to a new scene, for a 30 fps camera, the threshold could be adjusted accordingly within seconds, which is an acceptable latency for most use cases.

5.2.4.3 Cache entry replacement strategy

To evaluate our cache replacement strategy, we consider two cache hit patterns, uniform distribution and exponential distribution, and compare our importance-based strategy with two commonly used cache replacement strategies, *least recently used* (LRU) and *random discard*.

The number of cache hits, or the occurrences of reusable results can be modeled by a uniform distribution or an exponential distribution. Uniform distribution is often seen for single-app or in-app deduplication, as it is common to obtain input frames at fixed intervals and each component in the processing pipeline is invoked once per new input. Exponential distribution fits the multi-application scenario as the relative application popularity can be modeled by an exponential distribution [223].

For this experiment, we first define 100 different workloads, each of which takes a different amount of computation time ranging from 1 ms to 10 s. Then we create two request arrival sequences



(a) Exponential distribution

(b) Uniform distribution

Figure 5.8: Comparison of cache entry replacement strategies given different access patterns.

with 10,000 requests each, generated from these 100 workloads. Within the two request sequences, the number of occurrence of each workload is uniformly and exponentially distributed respectively. Next, we vary the proportion of working sets cached from 10 to 90 (meaning caching 10% - 90% of all workloads). Under each value, we submit the request sequence to the cache and measure the portion of the total computation time required due to cache misses, using the three different cache replacement algorithms.

Figure 5.8 shows that our algorithm consistently outperforms LRU by a large margin. For both request patterns, using the importance metric to retain entry caches can save an additional 40% of the computation while caching less than 20% of the previous results. The fraction of computation time drops further to below 5%, when the proportion cached grows to over 40% and 60% of the active working set respectively for the exponential and uniform workload distributions. The non-uniform distribution in the request patterm will propagate to the importance values, skewing the distribution of the latter. These results suggest that our algorithm successfully retains the results from the computation-intensive workloads.

5.2.4.4 System overhead

Cache lookup and insertion overhead. The cache lookup overhead depends on the organization, the current cache size, and the key length. The computation complexity of a key matching operation is determined by the key length. We submit 100 requests to the cache and measure the average

Table 5.2: Lookup latency

# of entry	key size (bytes)	LSH (μs)	enum (μs)
100	100	3.2	50
1000	100	3.6	170
10000	100	4.4	2210
100000	100	6.7	21340
100000	1000	7.5	205070
100000	5000	8.1	–

completion time.

Table 5.2 compares the lookup time using Locality Sensitive Hash (LSH) and by naively enumerating through all keys. LSH based lookup is very efficient, takes less than $10\mu s$, and scales well with an increasing cache size. Without a carefully designed key structure, the best one could do is to resort to naive enumeration. It incurs an acceptable latency when the total size of the keys is no larger than 10 MB, but cannot scale well to hundreds of MB.

The insertion overhead is at micro-second level even for a 500 MB cache (about the upper limit, since using more space is not practical for mobile devices), which is negligible.

IPC latency. We sequentially submit 500 requests and divide the total response time by 500. The average end-to-end latency using the Binder and AIDL mechanism is about 0.36 ms per request.

Space overhead. Android sets a per-device limit for the maximum heap space an app could use, ranging from 16 MB to 512 MB. This simply prevents applications from exhausting the memory, and our service operates within the limit.

Our key structure is also efficient regarding space usage. Consider a raw image of 400×400 pixels, about 500 KB in size. Its SIFT or SURF feature vectors are only 48 KB and 24 KB in size when 400 features points are extracted. Other feature vectors (such as FAST features) are often more compact. Even if all these vectors are used simultaneously, their combined size is still an order of magnitude smaller than that of the raw image.

Further, as we mentioned previously, even though we use multiple key indices, the corresponding "values" are only memory addresses, not the actual recognition results. This way, the recognition results are not stored redundantly.

Generalization to other hardware models. The overhead numbers listed above further imply that

the benefit of *Potluck* is not device or CPU dependent, but bound by the input. The read/write speed for memory and flash storage access varies little across phone models, while the latency due to the lookup/pipeline overhead is at least 3 orders of magnitude lower than running the same computation on a high-end GPU-equipped device. In fact, we will show later (Section 5.2.4.6) that an old phone running deduplication could outperform a powerful PC.

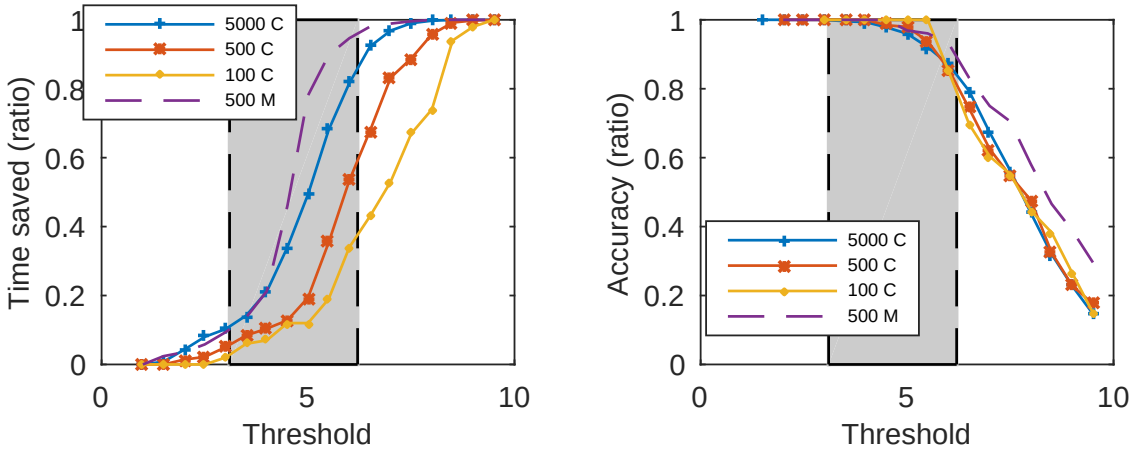
5.2.4.5 Single-application performance

We next evaluate the end-to-end performance of *Potluck* for applications individually. We run both the deep learning application and the AR application that loads and renders 3D models based on the current location and device orientation.

Performance and accuracy tradeoff. We randomly select 100, 500, and 5000 images along with their (ground-truth) recognition labels from the CIFAR-10 training set and 500 images from the MNIST dataset as the pre-stored entries, and then select 100 images from the test set as the inputs to the cache lookups. Figure 5.9 shows the processing time saved and the accuracy respectively as the threshold changes. The actual threshold values produced by our tuning algorithm stay within the shaded region in either figure.

The performance of *Potluck* is measured by dividing the accuracy and time saving by the respective optimal values. The optimal accuracy is defined as the accuracy when using the pre-trained AlexNet deep neural network to recognize the test images. The optimal time saving is 100% assuming all lookups result in cache hits (with the right results).

We make several observations from the figures. First, our threshold tuning algorithm results in a reasonable tradeoff by saving up to 80% of total the computation time at the expense of less than 10% accuracy drop. Second, when there is a larger number of stored results, the accuracy starts to drop slightly earlier. This makes sense because more cached results can increase the noise and the chance of mis-classification (i.e., false positives for key matches). Third, not surprisingly, the total time saving increases significantly with the number of stored entries, as there is a higher probability of cache hits. Lastly, the two different datasets shows consistent trends for the tradeoff between the accuracy and total time saved. This suggests the generic behavior of the system under various scenarios.



(a) Time saved

(b) Accuracy

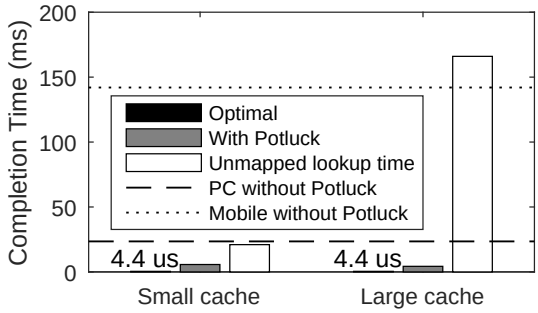
Figure 5.9: Time saving and accuracy vs the extent of deduplication opportunities. "C" and "M" correspond to the CIFAR-10 and MNIST datasets respectively

Note that we cannot assess the "accuracy" easily for AR applications, since the rendered scenes are evaluated with a number of visual quality metrics, such as the image resolution, and there is no absolute notion of "accurate".

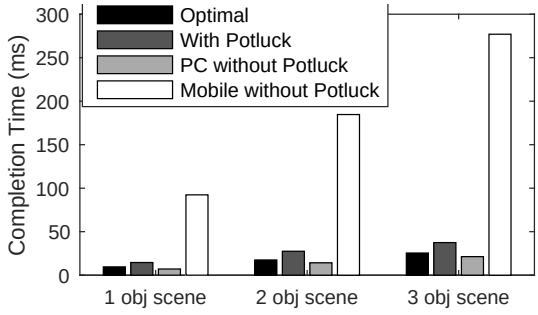
Mobile processing vs offloading to a PC. To further gauge the benefit of *Potluck*, we compare the application processing time on the mobile device and on the PC mentioned earlier. The latter is a proxy for offloading computation to a power server but *without incurring network transfer latency*.

For the *deep learning based image recognition application*, the experiment setting is the same as for the previous experiment, except that we run our threshold tuning algorithm live to automatically adjust the threshold, instead of fixing its value. Figure 5.10(a) shows the normalized average completion time for each image with optimal deduplication, with and without *Potluck* on the mobile device, and on the PC. The performance of *Potluck* is within 5 ms of the optimal case. It reduces the completion time of the native application on the mobile by a factor of 24.8, and even reduces the native execution time on the powerful laptop PC by a factor of 4.2.

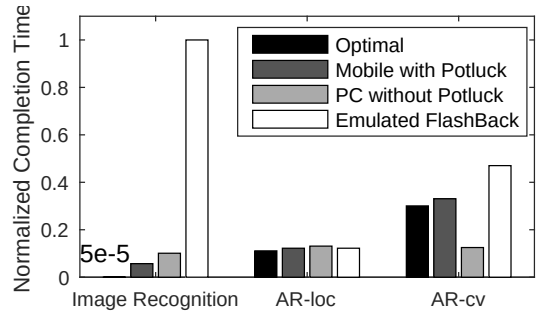
For the *3D graphic rendering* part of the AR application, our target results are three 2D scenes with depth information, each containing virtual 3D objects of different rendering complexity. Normally, a 3D object is rendered and then projected onto the display. With *Potluck*, the processing flow is simplified to looking up rendered 2D images with the most similar orientation, estimating the transform matrix, and warping the original 2D image to fit the current orientation [224]. The 3D



(a) Deep learning



(b) AR rendering



(c) Three apps concurrently

Figure 5.10: Single and multi-app performance of *Potluck*.

orientation and location of the device are used as the key for the cache lookups in *Potluck*.

Since we only care about the transform matrices between scenes, and not the actual scenes in the video, for this experiment we generate a video feed of three virtual 3D models viewed from different angles and sample non-consecutive frames to synthesize the workload to emulate a real scenario.

Figure 5.10(b) compares the per-frame rendering time needed by *Potluck* with the times for native rendering on the mobile, on the PC, and the optimal deduplication case. The performance of *Potluck* is within 9.2% of the optimal deduplication performance. It reduces the running time of the native application on the mobile device by a factor of 7, and only takes 47% longer than rendering on the PC.

5.2.4.6 Multi-application performance

Finally, we run the three applications together, two AR applications and an image recognition app as described at the beginning of the section. Note that the applications are not required to run simultaneously in the foreground in the classical sense of concurrency. Rather, we emulate a scenario where the invocations of these applications are interleaved in similar spatio-temporal contexts. We record several 30-second video segments from the real world at 60 fps, extract 200 frames, evenly spaced, from each video sequence, as our input sequences, and evaluate the performance gain from *Potluck*.

Figure 5.10(c) shows the normalized completion time of the three applications respectively. *Potluck* reduces the per-frame completion time by 2.5 to 10 times, and almost achieves the same performance as optimally reusing previous results. For the deep learning and the location based AR applications, running deduplication on the mobile device is even faster than running the whole workload on the PC.

The last set of bars in the figure represents an emulated version of FlashBack [225]. This is a system to achieve fast graphics rendering for *virtual reality* applications, by precomputing all possible input combinations and simply looking up the corresponding results during the actual run. This is the closest to reusing previous results for AR applications, even though the input handling is different. Assuming the same result from the input handling techniques in FlashBack and *Potluck*, the benefit of FlashBack only extends to in-app result reuse for only the rendering portion of our AR applications. In view of our benchmark applications, therefore, the emulated FlashBack can benefit the location-base AR application similar to *Potluck* does, benefit the rendering portion of the second AR application, but does nothing for the deep learning application.

We also evaluated *Potluck* on the MNIST dataset. The images in this set show higher semantic correlation than those in CIFAR-10. While *Potluck* delivered a similar time saving for the location-based AR workload as shown in Figure 5.10(c), it reduced the processing time of the image recognition application by a factor of 16 compared to native computation on the phone. This highlights the benefit of *Potluck* when the input data exhibit stronger correlation, as *Potluck* is able to eliminate more potentially duplicated processing. Further, these results again suggest that *Potluck* could bring significant performance gain in a broad range of scenarios.

5.2.5 Related Work

To the best of our knowledge, no existing work has explored cross-application approximate deduplication in mobile scenarios. Further, *Potluck* provides a more generic mechanism for deduplication even within the same (type) of applications. We discuss a few approaches closest to ours.

Application-specific solutions. Starfish [77], Flashback [225], and MCDNN represent recent efforts accelerating computation-intensive mobile applications. **Starfish** extends common computer vision (CV) libraries [215] with a centralized mechanism, including a cache to store previous function call arguments and results. However, it does not readily work for newer DNN-based applications [226, 227], and its memoization requires precise matching between the inputs. **FlashBack** is a pre-rendering system specifically designed for virtual reality (VR) applications. It utilizes nearest neighbour matching to select pre-rendered frames and adjust them for new frames. However, the design assumes fixed environment and known data pattern, which is not the case for non-VR applications, as the scenes for AR applications are unbounded and constantly changing. **MCDNN** accelerates the execution of deep neural networks on mobile devices. One particular optimization is to share the execution (results) of the common layers of the neural networks from different applications. But, the sharing is synchronous and does not involve explicit caching. If the exact same input is passed to the neural network twice, the whole computation will be performed twice.

In contrast, *Potluck* is more flexible in several ways. It targets cross-application deduplication, does not require applications to run concurrently to share results, and makes little assumptions of the specifics of the sharing applications or the shared input data. The input similarity is determined semantically, rather than based on the raw binary representation.

Deduplication vs frame sampling. Though not designed for the same settings, several previous efforts related to video analytics [228] or continuous vision [199] considered some form of *frame sampling* to reduce the computation complexity. These systems selectively process "the most interesting" input frames and skip the rest. They include algorithms to identify the frames of interest for further processing.

Potluck can be viewed as a different take on frame sampling. Our service computes the full results for selected input images and find a nearest match for the rest. One important difference is that *Potluck* makes no assumptions about the sequence of input images and is more flexible for

applications launched in an ad hoc fashion.

Computation reuse in clusters. In distributed clusters, Differential dataflow reuses results between iterations within the same program [229], while DryadInc [230], SEeSAW [231] and Nectar [232] leverage cross-job (not application) computation reuse with a centralized cache service and a program rewriter that replaces redundant computation with the corresponding cached results. These solutions do not readily apply to a mobile setting, since the resource constraints are completely different, and there is no distributed filesystem as a basic structure to synchronize data globally. UNIC [233] is a recent work specifically designed for deduplication security, which is orthogonal to our design. The techniques proposed by UNIC can be incorporated into *Potluck* for better program-level integrity and secrecy.

Approximate caching. There is a loose analogue between deduplicating *computation* in *Potluck* and deduplicating *storage* in approximate caching (such as Doppelg$'$anger [234]) and the compression of image sets [235] or nearly identical videos [236]. All cases are motivated by the similarity between input images, and various feature extraction mechanisms can be used to quantify the similarity for further compression. However, *Potluck* further reasons about the computation resulted from the input similarity, whereas the other schemes aim to reduce the space usage of the input.

5.2.6 *Potluck* summary

In this section, we argue for an unorthodox approach to optimize the performance of computation-intensive mobile applications via cross-application approximate deduplication. This is based on the observation that many emerging applications, such as computer vision and augmented reality applications take similar scenes as the input and sometimes invoke the same processing functions. Therefore, there are ample opportunities for reusing previously computed results.

We build *Potluck*, a background service that conceptually acts as a middleware to support multiple applications. *Potluck* converts the input image to a feature vector, which, along with the function invoked, then serves as a key to the previously computed result. The design further includes mechanisms to dynamically tune the input similarity threshold and manage cache entries based on their potential for reuse. Evaluation shows that we can speed up the processing significantly via deduplication.

Looking ahead, we believe there is scope to explore further deduplication opportunties. We have mainly focused on image-based applications, since they tend to be among the most computationally intensive. However, the design and implementation presented are general and can apply to other types of input data. We can also apply the deduplication concept across devices. Further, the applications could exploit optimization opportunities by adding post-lookup logic to perform incremental computation.

5.3 Cross-Device Approximate Computation Reuse

Section 5.2 explains a first prototype of caching and reusing computation results from the inference tasks of different applications that operate on similar input data. Recall that the overall approximate reuse process involves several steps and key challenges: 1) capturing and quantifying the *input* similarity in a metric space, 2) fast search for the most similar records, and 3) reasoning about the quality of *previous output* for reuse.

Step one is straightforward. Existing, domain-specific techniques can already turn these raw input values into feature vectors, and we can then define a metric to compute the distance between them, for example, the Euclidean distance. There are two implications, however. First, leveraging these feature extraction techniques decouples the application specific processing from generic system-wide procedures applicable to any such applications. Second, the app developer can use well-established techniques and libraries, and there is no need to manually annotate or manage the input features.

In this Section 5.3, to achieve approximate computation reuse *across device boundary* poses greater challenges to the latter two steps, and therefore we need to refine the algorithms. The challenges arise from two fundamental constraints regardless of the underlying scenario: (i) The input data distributions are dynamic and not known in advance, and (ii) similarity in the input does not directly guarantee the reusability among the output and there is no metrics measuring the reusability likelihood. To address (i), we propose a variant of locality sensitive hashing (LSH), which is commonly used for indexing high-dimensional data. The standard LSH is agnostic to the data distribution and does not perform well for skewed or changing distributions. Therefore, our *adaptive locality sensitive hashing (A-LSH)* dynamically tunes the indexing structure as the data distribution varies, and achieves both very fast and scalable lookup speed and constant lookup quality regardless of the exact data distribution. For (ii), we propose a variant of the well-known k nearest neighbor (kNN) algorithm. kNN is a suitable baseline since it makes no assumptions about the input data distribution and works for almost all cases. However, kNN performs poorly in a high-dimensional space due to the *curse of dimensionality*, insufficient amounts of data and skewed distribution in the data [212]. Our *homogenized kNN (H-kNN)* overcomes these hurdles to guarantee highly accurate reuse and provides control of the tradeoff between the reuse quality and aggressiveness.

Now, given the refined algorithms with better control of the approximate reuse and higher

efficiency, we further incorporate approximate computation reuse as a multi-tier service, called *FoggyCache*, extending the current computation offloading runtime. *FoggyCache* employs a two-level cache structure that spans the local device and the nearby server to achieve cross-device computation reuse. To maximize reuse opportunities, we further optimize the client-server cache synchronization with stratified cache warm-up on the client and speculative cache entry generation on the server.

FoggyCache is implemented on the Akka cluster framework [237], running on Ubuntu Linux servers and Android devices respectively. Using ImageNet [34], we show that A-LSH achieves over 98% lookup accuracy while maintaining constant time lookup performance. H-kNN achieves the pre-configured accuracy target (over 95% reuse accuracy) and provides tunable performance. We further evaluate the end-to-end performance with three benchmarks, simplified versions of real applications corresponding to the motivating scenarios. Given a combination of standard image datasets, speech segments, and real video feeds, and an accuracy target of 95%, *FoggyCache* consistently harnesses over 90% of all reuse opportunities, reducing computation latency and energy consumption by a factor of 3 to 10.

In summary, we make the following contributions:

First, we observe cross-device *fuzzy redundancy* in upcoming mobile and IoT scenarios, and highlight eliminating such redundancy as a promising optimization opportunity.

Second, we propose *A-LSH* and *H-kNN* techniques to quantify and leverage the fuzzy redundancy for *approximate computation reuse*, independent of the application scenarios.

Third, as an example realization, we design and implement *FoggyCache* that provides approximate computation reuse as a service, which achieves a factor of 3 to 10 reduced computation latency and energy consumption with little accuracy degradation for realistic application benchmarks.

5.3.1 Motivation

5.3.1.1 Example scenarios

Smart home. Many IoT devices connected to a smart home service platform [238] run virtual assistance software that takes audio commands to control home appliances. The intelligence of such software is supported by inference functions, such as speech recognition, stress detection, and

speaker identification [239]. Statistics [240] show that a small set of popular audio commands, e.g., *"turn on the light"*, are often repeatedly invoked. Say two household members issue this command to their respective device in different rooms. Currently, each command triggers the entire processing chain. However, processing both is unnecessary, as the two commands are semantically the same. It would be more efficient if one could reuse the processing output from the other.

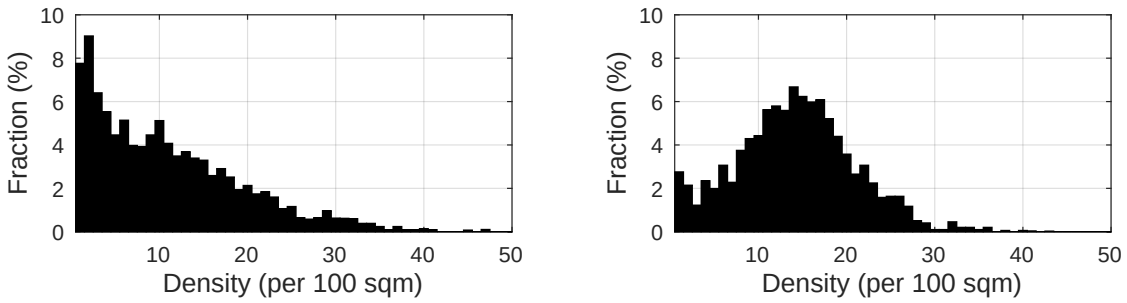
Cognitive assistance apps. Google Lens [32] has become very popular, which enables visual search by recognizing objects in the camera view and rendering related information. Key to the app is the image recognition function. Consider a scenario where the tourists near a famous landmark search for its history using the app. Clearly, it is redundant to run the same recognition function repeatedly on different devices for the same landmark. Although the devices capture different raw images, semantically the images are about the same landmark. If the recognition results can be shared among nearby devices, e.g., by a base station, we can avoid the redundant processing on individual devices.

Intelligent agriculture. Robotic assistance has been deployed to automate agricultural tasks. As an example [241], a fleet of autonomous vehicles move along pre-defined routes to measure and maintain the health of the crops, e.g., watering the crops if they appear dehydrated. Each vehicle captures images and other sensor data (for ambient light intensity, humidity, and temperature) every few meters, recognizes the crop status, and then acts accordingly in real time. The vehicles on adjacent paths record significantly correlated data, and running the same processing function on these correlated sensor data will largely produce the same results. Such repeated processing is unnecessary if the processing outputs can be shared among the vehicles, e.g., through the command center of the robots.

5.3.1.2 Fuzzy redundancy

Common to all three scenarios above, the application logic revolves around recognition and inference. There is redundancy in the processing of each application, even when presented with *non-identical* input data. We refer to this as *fuzzy redundancy*. This is due to the *similarity* in the input data, the *error tolerance* of the processing logic, and the *repeated invocations* of the same functions.

Input similarity stems from the same contextual information being captured, such as major landmarks, ambient noise, and road signs. For such information, there is (i) temporal correlation



(a) Outdoors (b) Indoors

Figure 5.11: Device density distribution from trace [1].

between successive samples (e.g., frames in a video feed), (ii) spatial correlation between what nearby devices observe, and (iii) common objects involved (e.g., traffic lights at each intersection).

Error tolerance arises from a common input-output mapping process, i.e., the input values that are "close" enough are often mapped to the same output. A large number of workloads exhibit this property, where high-dimensional input values (e.g., images of handwritten digits) are mapped to lower-dimensional output values (e.g., "0", "1", ..., "9"), and therefore the possible output values are constrained to a finite set. Learning-based workloads (e.g., recognition, classification, and AI agent) and graphics rendering [224] both exhibit such resilience, and increasingly they have been run in mobile scenarios.

Repeated invocations are manifested in three ways. (i) Given the popularity of some mobile apps, the same app (e.g., Google lens and PokeMon) can be launched by the same device or across devices repeatedly. (ii) Significant correlation exists in spatio-temporal contexts and smartphone usage [242]. For instance, IKEA Place [200], an augmented reality furnishing app, is mostly run by shoppers in IKEA stores. (iii) Mobile applications often rely on standard libraries. This is very common for computer vision (OpenCV [215]), graphics (OpenGL [243]), and deep learning (TF-Lite [18]), which means even different applications can invoke the same library functions.

5.3.1.3 Quantitative evidence

To gauge the extent of *fuzzy redundancy*, we estimate the amount of correlated processing for landmark recognition in the aforementioned cognitive assistance scenario. This is measured with the proportion of input images showing semantically equivalent scenes across mobile devices.

First, we estimate the mobile device density distribution by leveraging a WiFi trace from

Table 5.3: Proportion of redundant scenes (%).

Setting	**Device density** (# devices / 100 m^2)				
	0-10	*10-20*	*20-30*	> 30	*Average*
Indoors	49.63	74.43	99.57	100	64.14
Outdoors	61.01	85.76	99.51	100	82.67

CRAWDAD [1]. Mobile devices frequently scan WiFi access points (APs) to switch or renew their association, by broadcasting a probe request every few seconds. The trace contains probe requests from clients within range of different APs over three months. The AP locations include auditoria, office buildings, malls, and scenic spots. We select the traces at two types of locations, scenic spots and office buildings, for an outdoor and an indoor scenario respectively. The device density (i.e., number of devices per $100m^2$) is calculated by counting the number of distinct devices sending probe requests within a 30-second window. Figure 5.11 shows the device density distribution at these two locations.

Next, we use Google Streetview API [33] to download streetviews and create an "outdoor input image set" as perceived by a phone camera. The number of images selected is proportional to the corresponding device density distribution measured above. For the sampled images, we count the number of images capturing the same scenes (i.e., buildings, landmarks, and traffic signals) and convert that to a percentage of all images to quantify *the amount of fuzzy redundancy*. Similarly, we use the NAVVIS indoor view dataset [244] for indoors and repeat the above procedures to estimate the portion of redundant scenes.

Table 5.3 shows the proportion of redundant scenes given different device density. On average, around 64% and 83% of the images exhibit *fuzzy redundancy* for indoor and outdoor scenarios respectively. The amount of redundancy increases significantly with the device density. This highlights substantial redundancy elimination opportunities to optimize the performance of these contextual recognition based applications.

5.3.2 Approximate Computation Reuse

To eliminate *fuzzy redundancy*, we follow the philosophy for conventional computation reuse, i.e., caching previous outputs and later retrieving them instead of computing from scratch every time. However, existing *precise* reuse techniques cannot handle the *approximation* we need.

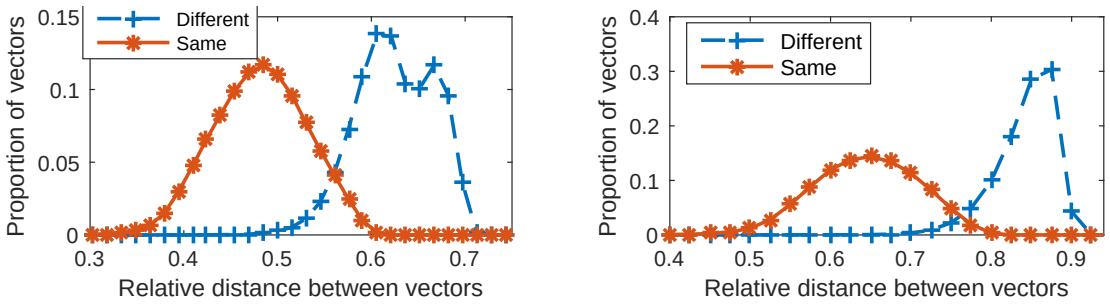
Problems with *precise computation reuse*. Conventional reuse [233, 231] determines the reusability of a previous computation output on the basis of hash-based *exact input matching*. Unfortunately, this is too restrictive for *fuzzy redundancy*, where the input values are correlated but rarely identical. We need to relax the criterion such that computation records are reusable if the input values are *sufficiently similar*.

Challenges for *approximate computation reuse*. Extending exact reuse is non-trivial and requires solving several problems: (i) embedding application-specific raw input data into a generic metric space, (ii) fast and accurate search for the nearest-match records in a high-dimensional metric space, and (iii) reasoning about the *quality of reuse* among the potential search outputs. Challenge (i) can be addressed with well-established domain-specific feature extraction approaches (Section 5.3.2.1). To address (ii) and (iii), we propose adaptive locality sensitive hashing (A-LSH, Section 5.3.2.2) and homogenized k nearest neighbors (H-kNN, Section 5.3.2.3).

Reuse process. Armed with these techniques, *approximate computation reuse* proceeds on a per-function basis. We always turn function input into feature vectors to serve as cache and query keys. Once an inference function is actually executed, a key-value pair is added to the A-LSH data structure. The *value* is simply the function output. When an application invokes a particular function, this triggers a reuse query for that function. We retrieve several key-value pairs from the A-LSH whose *keys* are the nearest-match to the query key (i.e., a feature vector from the *new* function input). Among the *values* of these key-value pairs, we then select the final query result (i.e., the *new* function output) with H-kNN. Section 5.3.2.4 discusses the generality of this process.

Crucially, while the input matching is approximate, the ideal output identified for reuse is *precise*, the same as the result from the full-fledged computation, due to the error tolerance discussed earlier (Section 5.3.1.2).

Terminology. Thoughout the paper, "input" refers to the raw input data to the inference function or the corresponding feature vectors serving as the cache or query key, while "output" refers to the previously computed results, the cached *value* matching a cache *key* or the reuse query result.



(a) Image data

(b) Audio data

Figure 5.12: Distance distribution between feature vectors of the same and different semantics.

5.3.2.1 Application-specific feature extraction

Different contextual recognition applications vary by their input data type (such as images, audio, and text) and inference logic. Therefore, the first step is to embed heterogeneous raw input data into a generic representation while preserving the notion of *similarity*.

There are two implications from this step. First, it decouples application-specific processing from general reuse procedures. Second, it obviates the need for app developers to manually annotate data features.

Assessing similarity is far more challenging than checking for *equality*. Fortunately there are well established techniques to map raw data to multi-dimensional vectors in a metric space. We can then compute the Euclidean distance between vectors to gauge their similarity.

Domain-specific approaches. For *images and videos*, their local and global characteristics can be captured in feature vectors such as SIFT [209] and GIST [245], which have been shown [246] to effectively measure image similarity. For *audio*, MFCC [216] and PLP [247] are widely used to capture acoustic features in compact vectors for speech applications [248].

Autoencoding. More generally, recent Autoencoder techniques [249, 250] use deep neural networks to *automatically* learn state-of-the-art feature extraction schemes for various data sources, including text, images, and speech.

Examples. Figure 5.12 shows that we can indeed quantifying data similarity with the distance between feature vectors mapped from the raw images and audio samples. The data are randomly selected from three arbitrary classes from ImageNet [34] and the TIMIT acoustic dataset [251]. We use SIFT to turn 256×256 images into 1000-dimension vectors and MFCC to convert 30-ms speech

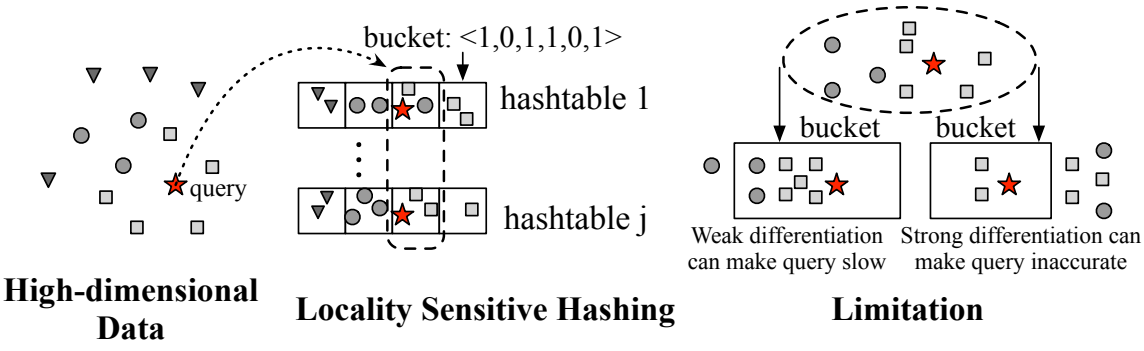


Figure 5.13: Locality sensitive hashing.

segments (sampled at 16 kHz, 16-bit quantization) to 39-dimension vectors. Figures 5.12a and 5.12b plot the distribution of the distance between pairs of feature vectors, for the image and audio data respectively. The distances are normalized in their respective scale space. We can see that the feature vectors for the same scene (or utterance) are geometrically "closer".

5.3.2.2 Adaptive Locality Sensitive Hashing

After turning the raw input data into high-dimensional feature vectors, we need a mechanism to index them for fast and accurate lookup.

Locality sensitive hashing (LSH) as a strawman. LSH [142] is widely used to search for the nearest matches in a high-dimensional space [252]. The data structure consists of multiple hashtables, each of which employs carefully selected, distinct hash functions and a set of buckets. The hash functions will map similar data to the same bucket in their corresponding hashtables with high probability. The buckets convey a sense of "locality".

Figure 5.13 shows LSH operations for three clusters of data, represented by three shapes. Ideally, each cluster should be mapped to a distinct, corresponding bucket across hashtables. When searching for the nearest matches, LSH first locates the bucket corresponding to the query input within each hashtable. The entries in all these buckets form a candidate set, from which the final output is selected based on its distance to the query input.

The time complexity for retrieving the nearest neighbors using LSH is $O(n^{\rho}\log n)$, where n is the number of data records indexed and ρ is a variable far smaller than one. In comparison, other spatial indexing data structures such as R-Tree, KD-Tree, and VP-Tree [253, 219, 254] are not as practical

Table 5.4: Lookup speed comparison (10,000 entries)

Dimension	R-Tree (ms)	LSH (ms)
4	0.018	0.002
64	2.279	0.009
128	6.342	0.011
1024	87.504	0.010

or efficient as LSH when dealing with high-dimensional data such as images or audio. R-tree has $O(2^d \log n)$ complexity, where d refers to the index key dimension. The factor 2^d significantly limits its usage in high-dimensional scenarios. Table 5.4 shows the lookup speed using different data structures when given random high-dimensional vectors as keys. While LSH consistently caps the lookup time at around 0.01 ms, that time for R-tree increases exponentially, to 87 ms, with the number of dimensions.

Limitation of LSH. The standard LSH is statically configured, however, limiting its performance.

Recall that each hashtable in the LSH leverages a set of hash functions $h : R^d \rightarrow \mathbb{N}$. Each hash function maps a vector v to an integer by $h_i(v) = \lfloor \frac{a_i \cdot v + b_i}{r} \rfloor$, where a_i, b_i are random projection vectors and *the parameter r captures the granularity of the buckets, i.e., how well buckets differentiate among dissimilar entries*. The concatenation of the j integers together forms a bucket, $< h_1(v), h_2(v), ..., h_j(v) >$, within the hashtable.

Thus, *configuring r is crucial for LSH performance*. Lookup is both fast and accurate when the hash bucket granularity matches the distribution of the cache keys.

The rightmost part of Figure 5.13 shows two examples of parameter misconfiguration. The star represents the query input, and should ideally be hashed to a bucket containing all the squares but only squares. When the buckets are too coarse-grained, the *hashing differentiation* is weak. Many dissimilar keys are hashed to the same bucket. Searching through a large bucket is slow, but we can be confident that all relevant entries are in the bucket and the best match can be found. Conversely, fine-grained buckets contain few entries each and are quick to search through, but might not contain the best match.

In practice, a major challenge is that the distribution of the input data is unknown and often time-varying. The performance of the standard LSH is thus at the mercy of the data distribution. This necessitates an algorithm to tune the LSH configuration during run time.

Adaptive LSH (A-LSH). In the LSH query complexity expression $O(n^{\rho}\log n)$, we aim to keep the parameter ρ constantly low for optimal lookup performance.

Analytically, ρ is determined by r [255]: $\rho(r) = \log_{p_r} p_1$, $p_r = 1 - 2\Phi(-r/c) - \frac{2}{\sqrt{2\pi}r/c}(1 - e^{-(r^2/2c^2)})$, and p_1 is simply p_r when $r = 1$. $\Phi(\cdot)$ is the cumulative distribution function of the standard normal distribution, $N \sim (0,1)$.

c is in fact called the *locality sensitivity factor*. Its ideal value should divide the entire data set into disjoint subsets, such that the variance is small within each subset but large across different subsets. The bucket granularity (r) can then be determined based on the intra-subset variance. Therefore, to obtain the optimal ρ, we first estimate the value of the *data-dependent* parameter c in the previous formula and then optimize the parameter r accordingly.

Since c varies with the current cached key distribution but no assumptions can be made in advance, we approximate c with the statistics of the *key* distribution. Specifically, for each cache key, we first find its k^{th} nearest neighbors, where k is a pre-selected constant (Section 5.3.2.3). The distance to this k^{th} neighbor, D_k, measures the radius of the immediate neighborhood cluster of the cached key. Across all cached keys we then have a distribution of D_k. Then, we calculate c as the smaller of $5 \times mean(D_k)$ and the 95th percentile within the distribution of D_k. This is an empirical rule we learned from experiments, covering a wide variety of data. Finally, we can tune r to reach the local minimum of the function $\rho(r)$ during the run time, leveraging existing optimization methods such as gradient descent [256].

5.3.2.3 Homogenized k Nearest Neighbors

After retrieving several "closest" cached records, we need to determine the reuse output from these records. Intuitively, we want to reuse aggressively as opportunities arise, but also conservatively to ensure the reused result would be identical to a newly computed result. This requires balancing the reuse quality and aggressiveness.

k nearest neighbors (kNN) as a strawman. Selecting a reusable record can be modeled as a data classification problem, so we first consider kNN [212], an algorithm most widely used for this purpose. The algorithm finds k records closest to the query input, identifies the cluster label associated with each, and then returns the *mode* of the cluster labels through majority voting. When

applied to the cached *key-value pairs* for our reuse scenario, step one above is based on matching *keys*, while the "cluster label" is the *value* field of each pair.

The primary advantage of kNN is its non-parametric nature, namely, no data-specific training is needed *a priori*. Despite the simple idea, kNN has been proved to approach the lowest possible error rate when given sufficient data [257]. State-of-the-art improvements, such as weighted kNN [258], assign different weights to the nearest records to further improve the kNN accuracy.

Problem with native kNN. The ideal situation for native kNN is when the k records form a single dominant cluster that truly matches the query key. The *value* associated with this cluster is then unambiguously the correct result for reuse. In practice, however, neither condition is guaranteed. Therefore, native kNN and its variants cannot always ensure accurate reuse. Nor do they give much control over the reuse quality or the aggressiveness. The limitations are manifested in both the input and output processing.

First, existing kNN variants cannot always assess input similarity accurately. The Euclidean distance between high-dimensional vectors (i.e., the cache keys) becomes less informative with increased dimensionality and fails to reflect the similarity in the keys. The *curse of dimensionality* causes the noise in certain dimensions to disproportionally skew the overall distance measurement [212].

Second, a dominant *value* cluster is often absent due to insufficient data or a skewed data distribution, and existing kNN variants provide inadequate *tie-breakers* between multiple clusters. As an example, suppose an input key K_1 is located at the intersection of two clusters, corresponding to the computation outputs V_1 and V_2 respectively. Among the nearest keys of K_1, half of their values are V_1, and the rest are V_2. In this case, either V_1 or V_2 can be valid, and it is impossible to select one correctly without further information.

Consequently, the *perceived input similarity* does not guarantee *output reusability*, and it is hard to gauge the confidence level of correct reuse merely from the cache *keys*. Unfortunately, native kNN and variants make decisions based on the *keys* but not the cached *values*.

To address the above limitations, we propose a novel refinement, called *homogenized kNN* (H-kNN). It utilizes the cached *values* to remove outliers and ensure a dominant cluster among the k records initially chosen. This lets us improve and explicitly control the *quality of reuse*.

Homogeneity factor θ. Observe that the kNN performance issues arise from the lack of a suitable

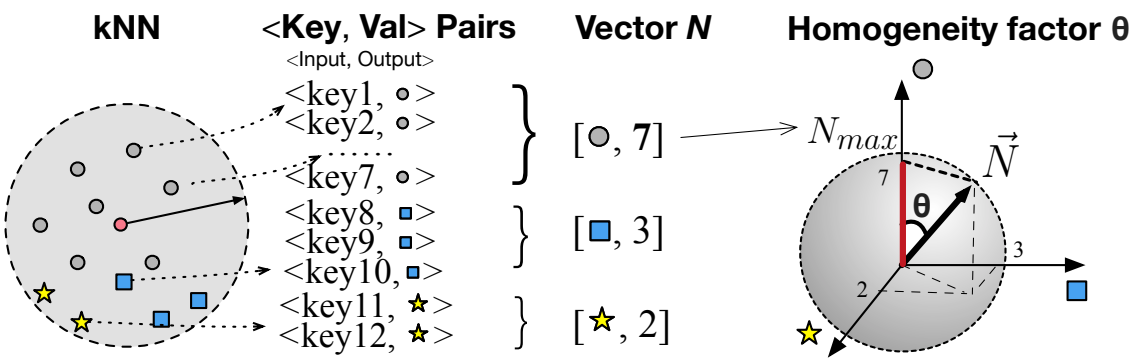


Figure 5.14: Calculating the homogeneity factor θ.

mechanism to assess the dominance of the clusters from the k records, hence the correctness of reuse. We therefore define a metric, the *homogeneity factor* (θ), for this purpose.

From the k key-value pairs, we first prepare a frequency vector $\vec{N} = [N_1, N_2, ..., N_k]$, where each element N_i records the frequency of the i^{th} distinct *value*. Then, $\theta(\vec{N}) = N_{max}/||\vec{N}||_2$, where $N_{max} = \max(N_i), \forall\, i \in [1,k]$. Figure 5.14 shows an example. 12 nearest keys correspond to 3 distinct values. Therefore, we derive $\vec{N} = [7,3,2]$, and $\theta = 7/\sqrt{7^2+2^2+3^2}$.

A geometric interpretation gives an intuition behind θ. Cached records (key-value pairs) with the same *value* form a cluster. Each cluster is mapped to a distinct dimension in $\vec{N}$, and the cluster size mapped to the length of the projection onto that dimension. The homogeneity factor θ is actually the *cosine* distance between $\vec{N}$ and its longest projection. A small *cosine* distance implies the existence of a large dominant *value* cluster, i.e., a high level of homogeneity among the k records selected. In that case, we can be highly confident that this dominant *value* is the correct result for reuse.

This definition of θ applies to discrete output values, which covers most classification scenarios. If, instead, the output values are continuous, θ can simply be defined to be inversely proportional to the *variance* of the k values and normalized to the proper scale.

Homogenized kNN (H-kNN). With the homogeneity factor, we can then set a threshold θ_0 to control the reuse quality. Algorithm 2 describes the operations of H-kNN. The intuition behind our refinement is to first remove outliers from the k records initially chosen and then assess the homogeneity of the remaining records. Reuse proceeds only if there is a dominant cluster. Note that $mean(D_k)$ is the average k^{th} nearest neighbor distance D_k (also used when adapting the LSH parameters in Section 5.3.2.2).

Algorithm 2: Homogenized kNN

```
// queryKey and k are arguments
Select k nearest neighbors with native kNN, get List<record> neighborList;
neighborList.filter {record ⇒ distance(record.key, queryKey) < mean(D_k)};
Calculate N⃗ and θ from neighborList;
if θ > θ_0 then
    return value corresponding to N_max;
end
return null;
```

The value of θ_0 simultaneously affects the correctness of the returned results and the proportion of non-null query outputs, the latter of which can be interpreted as the aggressiveness of reuse. Therefore, with H-kNN, the *quality of reuse* can be enhanced and explicitly controlled by adjusting θ_0. A lower θ_0 permits more reuse but potentially less accurate results. θ_0 can be set empirically by default (discussed in Section 5.3.5.3) or dynamically by the application according to its preference for the aggressiveness of reuse.

Bounding accuracy loss. For H-kNN, first note the reuse accuracy is tunable through θ_0. Next, we investigate the error inherent in the H-kNN algorithm.

For an input x, the error probability of reuse can be denoted by $error(reuse) = P_{x\sim C}(reuse(x) \neq compute(x))$, where C is the input distribution. According to the Probably Approximately Correct (PAC) learnability framework [259], for each given constant values of ε and δ, the error rate is bounded by ε with at least $1-\delta$ probability, when the number of participating samples, n, exceeds the value of the polynomial $p(\frac{1}{\varepsilon}, \frac{1}{\delta}, n, dim(f))$. n in our case is the total number of the cached records that are usable by H-kNN. $dim(f)$ is a factor determined by the intrinsic complexity of the learning task. For native kNN, $dim(f)$ quantifies how well the nearest neighbor data points can be unambiguously clustered (i.e., the *VC dimension* of a local sphere constituted by the nearest neighbour data [260]).

In other words, we can bound the potential accuracy loss of *kNN* with a specified confidence level by tuning the bucket granularity of the records stored in A-LSH, i.e., the parameter r mentioned in Section 5.3.2.2. As *H-kNN* improves on kNN, the reuse error can be reduced by further factor determined by the intrinsic VC dimensionality of the cached *values*.

5.3.2.4 Generality

We emphasize that the approximate reuse algorithms (A-LSH and H-kNN) are applicable to different applications.

Application-agnostic process. A-LSH and H-kNN are agnostic to the application logic because they operate at the granularity of individual functions (e.g., an image recognition method) instead of an application as a whole (e.g., the Google Lens app). If an application successively calls image recognition and speech recognition, say, two separate reuse queries will be issued.

A learning-based application typically executes in three stages: (i) acquiring and preprocessing the input, (ii) invoking relevant machine learning pipelines, and (iii) combining the outputs to generate the final result.

For example, the cognitive assistance app (Section 5.3.1.1) might acquire an image from the camera view, run image recognition to identify a landmark label, search for the landmark on the Internet, and finally display a selected page to the user. Depending on the input image quality, additional preprocessing can be employed, e.g., illumination correction, noise removal, and segmentation. Similarly, the final output generation steps, e.g., combining outputs from multi-modal learning pipelines, searching for related information and rendering on the screen, would be distinct for each specific case. However, the core machine learning functions, i.e., image recognition, are common across invocations of the app.

Stages (i) and (iii) vary by application and potentially even between different runs of the same application. In contrast, Stage (ii) only varies by the type of learning operations but not the specific application contexts. Operating only on stage (ii) enables A-LSH and H-kNN to be application-agnostic.

Beyond classification. Although we have used classification examples throughout this section, the reuse framework is broadly applicable to different types of machine learning functions. A-LSH is designed for fast and accurate nearest neighbour lookup upon high-dimensional data, and hence is generic to machine learning models. H-kNN can be applied to learning tasks with either discrete output (i.e., classification) or continuous output (i.e., regression, prediction), as explained in Section 5.3.2.3. A sufficient condition of H-kNN is local smoothness of the model, which is shown to be satisfied by a majority of the machine learning techniques [261, 262].

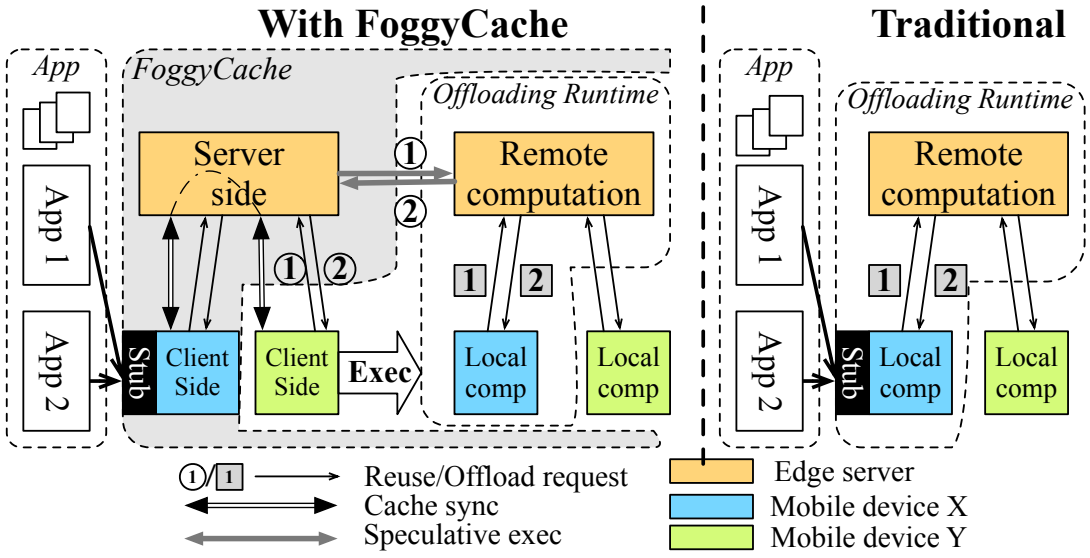


Figure 5.15: *FoggyCache* architecture.

5.3.3 *FoggyCache* service design

The techniques discussed in the previous section are generic to any approximate computation reuse scenarios. In this section, we discuss how to implement these techniques as a service for *cross-device* reuse outlined in Section 5.3.1. We target contextual based recognition and inference applications.

Mobile computing paradigms today typically require coordination between the smart devices, nearby edge servers [263], and the remote cloud. An essential component in such systems is an offloading runtime, such as MAUI, Odessa, and Comet [197, 264, 265]. The runtime dynamically partitions the processing pipeline into fine-grained tasks and places their execution locally on the device or remotely on a server.

Therefore, we re-design the traditional offloading runtime by incorporating approximate reuse as a service called *FoggyCache*, interposed between the application and the offloading runtime as an intermediate layer. *FoggyCache* intercepts the application call to the offloading runtime interface, as shown in Figure 5.15, to invoke approximate reuse regardless of where the computation task is eventually executed.

5.3.3.1 System overview

FoggyCache follows a typical client-server architecture. The *FoggyCache* client can be on a smartphone, tablet, or IoT device. The *FoggyCache* server is a central point of coordination between the clients. Given the advent of mobile edge computing for low-latency computation offloading, edge servers or cloudlets [195] are ideally suited to deploying the *FoggyCache* server.

Server. The server-side *FoggyCache* consists of an A-LSH cache gathering previous computation records (input & output) of all clients, a service daemon handling reuse queries, and a module that handles the client-server coordination.

Client. The client-side *FoggyCache* consists of an on-device A-LSH cache and a service endpoint which interacts with the server-side cache and the offloading runtime or the applications. The local cache stores a subset of the computation records from the server-side to minimize remote lookup.

Plugging *FoggyCache* in the offloading runtime. To avoid modifying the application, we retain the native interface between the application and the offloading runtime. The *FoggyCache client* intercepts the offloading call inside the entry point to the offloading runtime.

Take MAUI as an example. Once a method is declared as `remoteable`, its invocation will prompt the standard offloading runtime to schedule the code execution, locally or remotely. With *FoggyCache*, the method invocation first triggers the reuse pipeline before a scheduling decision is made. If any previous results are reusable, these are returned directly to the application without further computation. Otherwise, the normal offloading action resumes to schedule and execute the task. The APIs are detailed in Section 5.3.4). The *FoggyCache server* runs in its own process or container, separately from the remote end of the offloading runtime.

Challenge: two-level cache coordination. Both the server-side and client-side caches adopt the *least frequently used* (LFU) policy for cache entry replacement. However, coordination is crucial between the two levels of cache. As new computation requests are initiated from the clients, yet cross-device reuse is supported at the *FoggyCache* server side. Therefore, how new computation records propagate from the clients to the server and vice versa notably affects the *FoggyCache* performance. Our solution has two parts, corresponding to the two directions of data flow between the client and the server, shown in Figure 5.15 by the arrows of cache sync and speculative execution.

Algorithm 3: Initial cache warm-up algorithm

```
// Subset size s and num of nearest neighbors k are inputs, subset is the
   output
Initialize empty subset[s];
Create inverted index Idx: {Value ↦ List<Entry> lst} from the cache;
Sort the Value from large to small in Idx w.r.t. lst.size();
Store the sorted Values into a List<Value> vlist;
while vlist is not empty && subset is not full do
    Value v = vlist.get(0);
    List<Entry> elist = Idx.get(v);
    Sample min(k, elist.size(), subset.spaceLeft()) entries from elist and append them to subset;
    vlist.remove(v);
end
if vlist is empty then
    Proportionally sample entries from the cache to fill up the subset;
end
```

5.3.3.2 Client side cache management

Two mechanisms are needed to synchronize the client side caches with the server side: *client warm-up* and *cache miss handling*. The former is needed when a client appears in the vicinity of the server for the first time to boot-strap the service. The latter is triggered when no locally cached outputs could be reused.

Client cache warm-up. Intuitively, the client cache should receive broadly distributed entries to jump start the reuse service and maximize the probability of a random reuse query being matched with a reusable output from the cache.

Without relying on any assumptions on the input data distribution, we adopt *stratified sampling* to generate a subset of the server cache. The size of the subset is determined by the client or follows a default value. Algorithm 3 details the operations. The key idea is to first select as many *types* of cached *keys* with popular cached *values* as possible, where the *popularity* of a cached value is estimated based on the number of cache keys mapped to this value. This ensures a broad coverage of the records in the subset so that our approximate reuse algorithms could proceed in most cases. If space remains in the client cache after this first sampling pass, the algorithm then proportionally samples from the rest of the entire server cache. This algorithm achieves a dynamic trade-off between the subset coverage, the distribution of the cache keys, and the limited client cache space.

Note that existing data-dependent prefetching techniques [266, 267] are orthogonal to our design.

While we aim for reasonable performance *without prior knowledge*, other prefetching techniques could be adopted instead if *prior knowledge about the input data is known*.

Cache miss handling. Cache miss handling is on the critical path of the application, and therefore the processing logic and the data transmitted should be lightweight and minimal.

When the *FoggyCache* client does not carry reusable outputs, it sends a request to the server including the query input and the *homogeneity threshold* θ_0. Different clients could therefore customize the tradeoff between time saving and reuse accuracy by querying the same server cache with different thresholds θ_0.

The *FoggyCache* server executes the query and returns the reused output along with k nearest records, the minimum needed to carry out the H-kNN algorithm. This way, it reduces a potential cache miss from a similar, future query on the client device. Meanwhile, k is small enough to avoid incurring non-negligible communication overhead.

5.3.3.3 Server side cache updates

From the server perspective, it is desirable to collect newly generated computation records from the clients in a timely fashion for cross-device reuse. Intuitively, each *FoggyCache* client can batch updates to the server periodically. However, the cache entries might reach the server too slowly and unreliably this way, especially in the face of client mobility or unstable network connectivity. Moreover, not all computation records are created equal. For instance, a computation record with few nearest neighbour records stored in the *FoggyCache* server could potentially benefit all clients that submit reuse queries with similar inputs, and hence should be synchronized to the server as soon as possible. However, only the *FoggyCache* server knows such information. Therefore, we devise a speculative execution mechanism on the *FoggyCache* server to speed up its updates proactively.

Speculative computation. Once a reuse query comes, the server additionally estimates the *importance* (i.e., the probability of future reuse) of the computation task that corresponds to the query. Based on this probability, the server decides whether to speculatively execute the task and add the input-output record to the cache for future reuse queries.

Although prediction-based speculative execution algorithms are widely used [268, 269], they are not directly applicable. Due to the approximate nature, the *importance* of a computation record is no

longer solely decided by the access statistics about the record itself.

Instead, the likelihood of a computation record being reused in the future is jointly determined by three factors: the average access frequency (F_k), the average distance from the reuse query ($Dist_k$), and the homogeneity factor (θ_k) among the k nearest neighbors of the query input. The *FoggyCache* server also maintains the average access frequency F_{avg}, the average distance to k nearest neighbor $Dist_{avg}$ among all cached records, and the default homogeneity threshold θ_0. We then calculate $P_f = \min(F_{avg}/F_k, 1)$, $P_d = \min(Dist_{avg}/Dist_k, 1)$, and $P_\theta = \min(\theta_k/\theta_0, 1)$, as the corresponding normalized factors ranging in (0, 1] so that they can be used as probabilities. Then, $importance = 1 - P_f \cdot P_d \cdot P_\theta$. The multiplication captures the independence between these three factors. The *FoggyCache* server will then invoke speculative computation for this query with a probability equals to *importance* . The intuition behind the *importance* value is that we first take access frequency F_k as a baseline estimate, and then further consider the approximate nature, where the input distribution ($Disk_k$) and the output distribution (θ_k) both play an important role in determining the reused output (Section 5.3.2).

Note that the decision to proceed with speculative execution does not consider the load on the edge server. Basically, we decide *whether* to speculatively compute a record based on its importance, but we let the task scheduler on the edge server to decide *when* to execute the speculation task. For instance, the task can be separately assigned a low priority to avoid it contending with latency-sensitive tasks.

5.3.3.4 Additional consideration

Incentives: Various approaches [270, 271, 272] have been proposed to incentivize participation in decentralized systems. *FoggyCache* follows the "give and take" approach to incentives, similar to the proposal in [270]. Each *FoggyCache* client is allocated free credits at the beginning, while additional credits are given *proportional* to the number of computation outputs it contributes. The credits are used to query reusable computation from the server. The exact numerical value of the proportion parameters vary based on the global balance of queries and contributions under each specific scenario.

Security. The main security concern for *FoggyCache* arises from malicious devices polluting the

cache with false computation outputs. To address this, we can incorporate existing object-based reputation system (e.g., Credence [213]) in *FoggyCache* with negligible additional overhead. Each computation record is additionally labelled with an anonymous identity of the contributing client. Clients implicitly vote on cached records while running the reuse query. Specifically, if a cached record is selected by Step 2 of the H-kNN but its output is not chosen in the end, this constitutes a negative vote. Conversely, a successful reuse is a positive vote.

Privacy. Good enough privacy could be achieved by anonymizing participating devices when reporting data to the server. Since *FoggyCache* targets locality-based scenarios, the raw input of the approximate reuse is mostly local contextual information. Such information is meant to be collected by all nearby entities and hence public by nature. Location privacy is less of a concern here. Moreover, the *FoggyCache* client does not have to store and operate on raw input data. This means that different applications or vendors can employ their custom encryption schemes to protect the raw data without affecting cross-device reuse, as long as they feed the feature vectors extracted to *FoggyCache*.

5.3.4 Implementation

5.3.4.1 Architecture

We implement *FoggyCache* following a typical client-server model. A two-level cache structure that spans the edge server as well as the local device serves as our storage layer. The communication layer builds on the Akka [237] framework.

Cache layout. The two-level storage adopts the same layout. The highest level of each cache structure is a Java `HashMap`, which maps a function name (`String`) to an in-memory key-value store, where an A-LSH is generated from the key region among computation records of this function collected from all the clients. Additionally, the server side cache system includes utility functions to serialize and deserialize its data partially to disk.

Concurrency. *FoggyCache* is built using the Akka toolkit, which adopts the actor model [273] to handle concurrent interactions between system components. Each function module is implemented in a separate class extending the Akka `AbstractActor` class. Concurrency is managed implicitly by the Akka framework via message passing. We further leverage the Akka cluster module to provide a

fault-tolerant, decentralized membership service.

5.3.4.2 APIs and patches

***FoggyCache* APIs.** As much as possible, *FoggyCache* aims to make the processing logic transparent to the offloading runtime and applications. Therefore, three intuitive APIs are exposed: `ConfigFunc(func_name, config)`, `QueryCompOutput(func_name, input, params)`, and `AddNewComp(func_name, input, output)`. The first specifies reuse configurations for each native function (e.g., serialization, feature extraction, and vector distance calculation). The latter two trigger reuse queries and feed the native processing outputs back to *FoggyCache*.

Application or library patches. To interact with *FoggyCache*, short patches should be applied to the offloading runtime, or the application code when no runtime is used. No more than 10 lines of code is needed to wrap around the native pipeline. `QueryCompOutput` and `AddNewComp` are added to the native code within a conditional statement to determine whether to invoke the native processing pipeline. `ConfigFunc` enables on-demand customizations.

5.3.5 Evaluation

5.3.5.1 General setup

Application benchmarks. Following the motivating examples in Section 5.3.1, we build three stripped-down versions of real applications as benchmarks, two for image recognition (plant and landmark detection) and one for speaker identification. These are implemented in Java, using the DL4J [57], OpenCV [215], and Sphinx [274] libraries. The workload settings follow those in related papers [248, 275], using the same pre-trained neural network models that are widely adopted by real applications. Compared to the real applications, our benchmarks skip supporting functionalities such as the user interface, since they can interfere with the timing and energy measurements of the core computation modules. Our benchmarks can also be instrumented easily for various measurements, which is difficult with proprietary applications.

Datasets. We use two standard image datasets, ImageNet [34] and Oxford Buildings [276], an audio dataset, TIMIT acoustics [251], and several real video feeds.

Table 5.5: Data correlation in different settings.

Setting	Avg norm distance
ImageNet (same synset)	1.00 +/- 0.15
Video (10 frames apart)	0.31 +/- 0.04
Video (30 frames apart)	0.53 +/- 0.27

The ImageNet plant subset includes over 4000 types of labeled plant images, taken from different viewpoints under various lighting conditions. The Oxford Buildings dataset consists of 5000 images of 11 specific landmarks in Oxford, hand-picked from Flickr. The TIMIT acoustic dataset [251] contains broadband recordings of 630 speakers of eight major dialects of English, and we use it for speaker identification. For end-to-end performance evaluation, we also use several 10-minute real video feeds, taken on a university campus and in a grocery store, multiple times at either location.

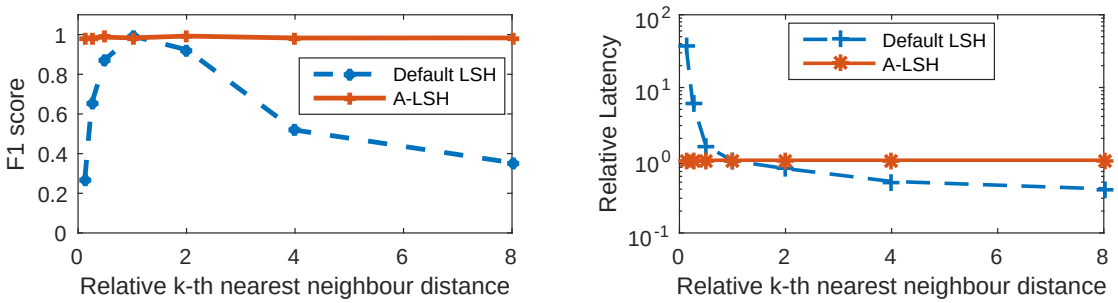
Table 5.5 compares the average feature vector distance between two images from the same syntax set in ImageNet and two frames from a video feed. The distance is significantly larger (by more than 50%) for ImageNet than for successive video frames, because no *spatio-temporal correlation* exists between images in ImageNet.

Therefore, we mainly use the standard image and audio datasets in our evaluation. Although they appear less realistic than real audio or video feeds, they present more challenging cases for computation reuse and help us gauge the lower-bound performance of *FoggyCache*.

Hardware setup. With a 64-bit NVIDIA Tegra K1 processor, Google Nexus 9 is one of the most powerful commodity Android mobile devices. Thus, we use the tablet (running Android OS 7.1) as the client side device to assess the potential benefit from saving computation with *FoggyCache*. The *FoggyCache* server runs on a Ubuntu (14.04) Linux desktop server with a quad-core 2.3 GHz Intel Xeon CPU and 16 GB of memory.

5.3.5.2 Microbenchmarks

A-LSH performance. We first select a subset from ImageNet, optimize the parameter r for the default LSH, and calculate the average k^{th} nearest neighbor distance ($mean(D_k)$ in Section 5.3.2.2). Recall that this distance captures the density of the data in the LSH (a large distance indicates a low density and vice versa). Then, we select other subsets of images where their average k^{th} nearest



(a) F1 score

(b) Latency

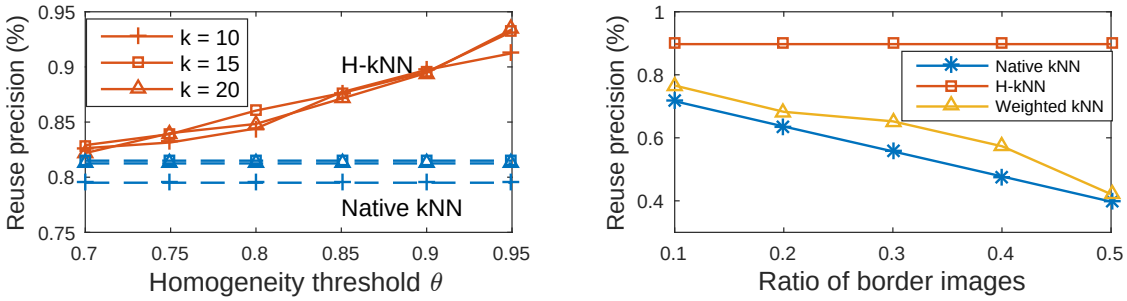
Figure 5.16: Lookup quality and latency for the default LSH and A-LSH.

distances range from $1/8$ of the D_k to $8\times$. These subsets serve as the input to the default LSH and A-LSH. k is set to the default value 10. The lookup quality is measured by the *F1 score*, the harmonic mean of *precision* (the correct rate of the results) and *recall* (the percentage of the correct results found), ranging from 0 (the worst) to 1 (the optimal).

Figure 5.16a shows that the lookup quality of the default LSH fluctuates dramatically given different data densities, whereas A-LSH consistently maintains an F1 score over 0.98. The default LSH only achieves a high lookup quality when the data distribution matches the pre-determined value of r. Figure 5.16b further shows that the lookup time for A-LSH remains constant. However, there is no guarantee for the default LSH, especially when the data are densely stored and thus highly clustered into the same few hash buckets. Note that although LSH appears to incur a lower lookup time for sparsely populated data, the corresponding lookup quality is low. Together the figures show that A-LSH accurately adapts the parameters to the dynamics of the input data distribution, and *consistently achieves a near-optimal balance between the lookup quality and speed*.

H-kNN performance. We compare H-kNN with naive kNN and a state-of-the-art variant, weighted kNN [258]. The performance metric is the reuse *precision*, which is upper-bounded by 100%.

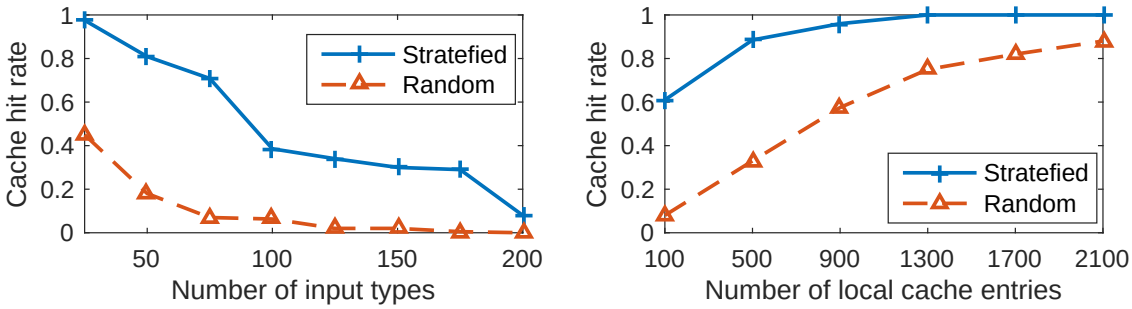
First, we select 1100 images from 4 types of syntax sets in ImageNet. 1000 of them are fed into the cache, and the other 100 images as inputs for H-kNN k and θ_0 (the homogeneity threshold) values vary. The solid and dashed lines in Figure 5.17a represent the H-kNN and native kNN performance, respectively. H-kNN outperforms native kNN by an increasing margin as θ_0 increases, which confirms that (i) the homogenization process improves the reuse accuracy, and (ii) the level of accuracy is indeed tunable through the parameter θ_0. This means that *applications can customize the level of reuse based on desirable accuracy guarantees*. The value of k makes little difference,



(a) Effects of θ_0

(b) Effects of data distribution

Figure 5.17: Reuse precision of H-kNN and alternatives.



(a) cache size 1000

(b) 100 types of inputs

Figure 5.18: Client cache hit rates and server cache sampling strategies.

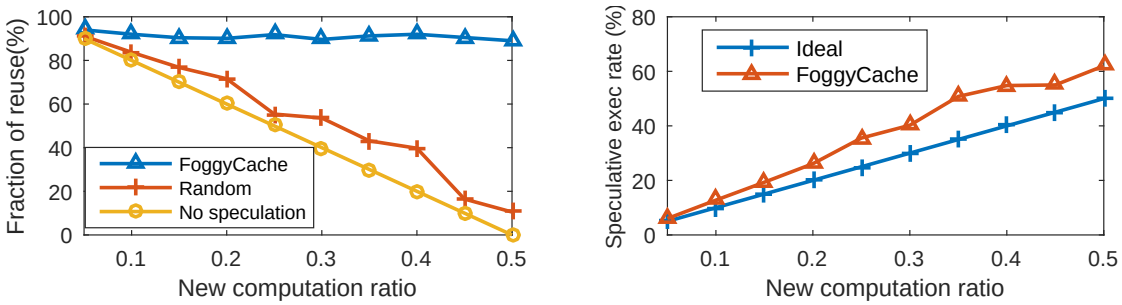
however. Based on the results, we set $k = 10$ and $\theta_0 = 0.9$ throughout this section. More detailed tradeoff is shown in Figure 5.20b.

Second, we investigate how H-kNN copes with two intersecting clusters (the example against native kNN in Section 5.3.2.3), by adjusting the proportion of cache keys at the intersection of two clusters. Figure 5.17b indicates that H-kNN maintains a consistent and high reuse precision regardless of the key distribution. Unfortunately, both native and weighted kNN suffer, as predicted in Section 5.3.2.3, with the reuse precision dropping by 40%.

Client cache warm-up. We next evaluate the benefit of stratified sampling for client cache warm-up (Section 5.3.3.2), and compare that to randomly sampling server cache entries.

We generate different key-value pairs from ImageNet data to store in the *FoggyCache* server cache and also for reuse queries. Then, we bootstrap the client cache with stratified sampling and random sampling (as the baseline) respectively. The performance of the algorithms is shown in terms of the client cache hit rate.

First, we set the client cache size to 1000 entries, change the number of syntax sets of images



(a) Speculation performance

(b) Speculation efficiency

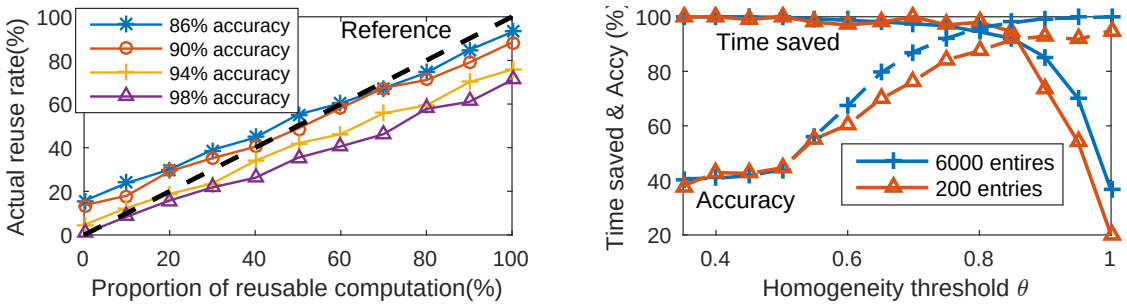
Figure 5.19: Performance comparison of speculative execution in *FoggyCache* and alternatives.

at the server, and observe the cache hit rates. We make two observations from Figure 5.18a. (i) When fewer than 100 types of images are cached at the server, stratefied sampling achieves over 50% cache hit rate, which confirms that popular images in ImageNet are adequately prioritized. (ii) When more types of images are cached at the server, the client cache hit rate from random sampling drops to nearly zero, whereas stratified sampling still manages over 25% hit rate, showing better type coverage in the latter.

Then, we select 100 types of images from the server cache and vary the client cache size. Figure 5.18b shows both strategies achieve 80% hit rate, but stratefied sampling requires only a quarter of the cache space needed by random sampling.

Speculative server cache updates. Finally, we gauge the benefit of incorporating speculative computation (Section 5.3.3.3) in *FoggyCache*. We select a subset of ImageNet dataset to create multi-device reuse query streams, where the fraction of "new" computation (no reusable results exist at all) ranges from 5% to 50%. We compare our speculative execution algorithm with two alternatives, *random* and *no speculation*.*Random* means invoking speculative execution with the same probability as in *FoggyCache*, but selecting inputs randomly. The ideal reuse proportion is 100%.

Figure 5.19a illustrates that *FoggyCache* consistently caches in around 90% of the reuse opportunities, whereas *random* and *no speculation* cannot keep up as the fraction of "new" computation increases, because *FoggyCache* accurately predicts the *importance* of a computation record for future reuse and preemptively generates that record before the actual reuse request. Figure 5.19b compares the fraction of computation that is speculatively executed in the ideal case (each speculatively generated record is visited later) and in *FoggyCache*, and our algorithm only triggers 10% unnecessary



(a) Reuse opportunities captured

(b) Reuse vs accuracy tradeoff

Figure 5.20: Tradeoff between captured reuse opportunities and computation accuracy.

computation at most compared to the ideal case.

5.3.5.3 Tradeoff between reuse and accuracy

Accuracy. We run object recognition using ResNet50 [15] on selected ImageNet images to assess the tradeoff between the aggressiveness of reuse and the accuracy.

First, we quantify how well *FoggyCache* recognizes reuse opportunities when the fraction of reusable queries in the whole query stream varies. The dashed line in Figure 5.20a shows the ideal case and serves as a reference. Any points above indicate false negatives (missed reuse), while points below the line indicate false positives (inaccurate reuse).

FoggyCache consistently captures the reuse opportunities in all data combinations while maintaining high accuracy. Both the false positive and false negative rates are below 10% (the 0% and 100% reuse points) while the reuse accuracy exceeds 90%. Even if we reuse conservatively to ensure a 98% accuracy, we only miss fewer than 30% of all reuse chances.

Second, we examine the trend of the total computation time saved and the relative accuracy (compared to native recognition accuracy), both as the homogeneity threshold θ_0 varies. We run the experiments for various caching levels, ranging from 200 to 6000 cached entries. For legibility only the lines for 200 and 6000 entries are plotted. The other lines fall between these two.

The dashed and solid lines in Figure 5.20b plot the relative accuracy and the time saved respectively. We can see that setting θ_0 to between 0.8 to 0.95 would ensure both higher than 90% accuracy and less than 20% loss of the reusable opportunity. This confirms that *FoggyCache* achieves a decent balance between accuracy and computation time reduction.

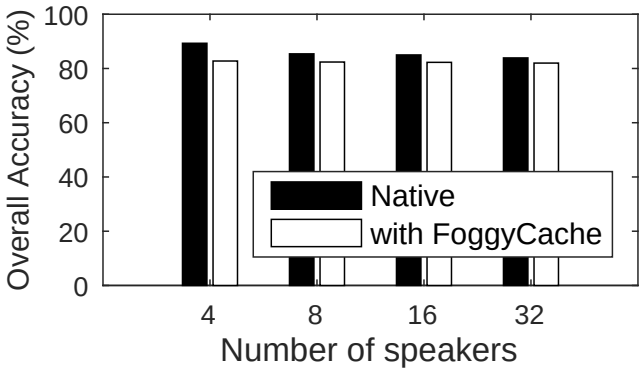
Table 5.6: End-to-end *FoggyCache* performance.

Workload			**Latency (ms)**				**Energy (mJ)**	
App.	Description		Offl. (w/o)	Offl. (w/)	w/o.	w/	w/o.	w/
Speaker Id.	Num of speakers	4	28.1	***8.4***	13.1	***4.2***	30.4	***9.8***
		8	28.1	11.8	13.1	5.5	30.4	13.3
		16	28.1	12.1	13.1	5.9	30.4	13.7
		32	28.1	13.2	13.1	6.4	30.4	15.0
Landmark Det.	DNNs	AlexNet [221]	24.6	16.3	37.1	19.5	365.9	39.5
		ResNet50 [15]	32.8	17.7	102.4	27.9	1315	110.7
		VGG16 [277]	53.8	***21.4***	269.6	***57.3***	3132	***246.9***
	Video feed (campus)		53.8	***12.0***	269.6	***25.4***	3132	***114.2***
Plant Recog.	DNNs	AlexNet	24.6	16.6	37.1	21.4	316.8	113.9
		GoogleNet [278]	29.2	17.9	65.3	32.2	817.4	236.8
		VGG16	53.8	***27.9***	269.6	***99.8***	3132	***901.4***
	Video feed (grocery)		53.8	***16.5***	269.6	***30.8***	3132	***131.1***

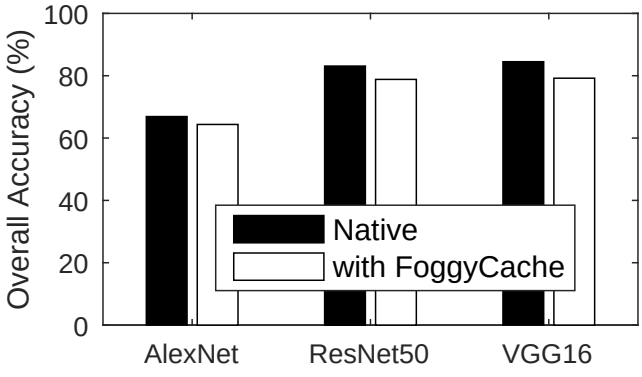
User experience. We conduct an informal user survey among students on our campus to gauge how approximate reuse affects user experience. In the context of the cognitive assistance application, students are asked whether they are satisfied with different combinations of the percentage accuracy loss and the reuse benefits (in terms of percentage reduction of battery consumption and latency), with data points taken from Figure 5.20b. From 100 completed questionnaires, 92 are satisfied with the user experience when the accuracy loss is under 5%, and 80 satisfied when the accuracy loss is under 10%. *FoggyCache* performs well for both cases. For more accuracy-sensitive applications, such as autonomous driving and medical pill recognition, the accuracy of *FoggyCache* can be tuned by carefully selecting the value of θ_0 and the number of cached records.

5.3.5.4 End-to-end system performance

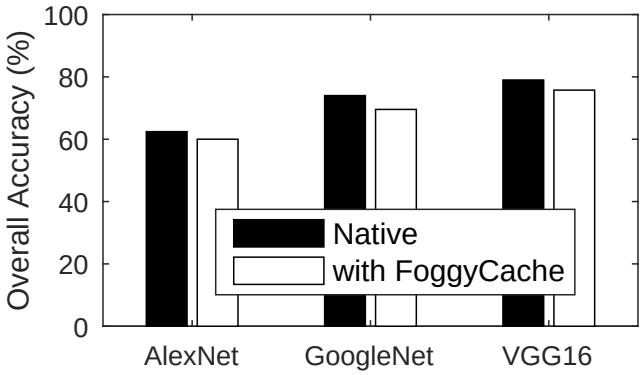
We investigate the end-to-end performance of *FoggyCache* using the three aforementioned application benchmarks. We separately consider two modes of execution for mobile applications, local processing on the mobile device and edge offloading. The real-time decision made by the offloading runtime between the two modes is orthogonal to the *FoggyCache* behavior. The performance metrics are *latency*, *energy consumption*, and *accuracy*. The *latency* is measured end-to-end from the arrival of a request to its completion. The *accuracy* is defined as the percentage of correct results. The *energy consumption* is calculated based on the real-time battery status collected with the Android debugging



(a) Speech identification



(b) Landmark recognition



(c) Plant recognition

Figure 5.21: The accuracy of the processing pipeline with and without *FoggyCache*.

API, `adb dumpsys batterystats`.

Experiment settings. We use Nexus 9 tablets as the *FoggyCache* clients, configured with a 15 MB local cache size. The *FoggyCache* server is deployed on a Linux machine which also serves as the edge offloading destination. The network latency between the clients and the server is around 20 ms, a typical value for the edge setting [279]. We also tried lower latencies but they only made *FoggyCache* perform better.

For *Speaker identification*, we randomly select 3200 speech segments from 4, 8, 16, and 32 speakers respectively from the TIMIT dataset, add different ambient noise, and extract the PLP feature vectors We store the computation records in the server cache, and populate 10% of them to the tablet for client cache warmup. Another 200 speech segments are selected from TIMIT and preprocessed the same way to serve as the test inputs. The core computation of the workload follows the same setting as for DeepEar [248].

For *landmark detection* and *plant recognition*, we take 5000 images each from the Oxford dataset and ImageNet and both the campus and grocery store video feeds. The standard datasets exhibit no *spatio-temporal correlation* between successive inputs, while the real video feeds contain common *imperfections*, e.g., motion induced or out-of-focus blur. We extract feature vectors, warm up the client cache with 10% of the data, and process another 1000 images as test inputs. Four fine-tuned neural network models (AlexNet, GoogleNet, ResNet50, and VGG16) are used to evaluate the performance.

Performance. Table 5.6 records the performance in all the experiments. "With" and "Without" refer to whether *FoggyCache* is enabled, and "Offl." refers to cases where the actual computation happens at the edge server instead of the local device. The numbers in bold highlight the most remarkable performance of *FoggyCache*. *FoggyCache* achieves a 50-70% latency reduction for both local processing and edge offloading for the standard image datasets. When using the real video feeds, the processing latency could be reduced by 88%. The energy consumption is only measured for local processing. *FoggyCache* reduces the native energy consumption by a factor of 3 for the standard datasets and 20 for the video feeds. Figure 5.21 shows that *FoggyCache* caps the overall accuracy penalty under 5% while achieving good performance. This confirms that A-LSH and H-kNN can ensure the reuse quality regardless of the specific settings. The accuracy penalty for the video feeds is constantly under 1%, hard to tell from the bars and thus not shown in the figure.

To sum up, *FoggyCache* effectively reduces the latency and energy consumption (for on-device processing) of the native processing pipelines, and the benefit is more pronounced when the native logic is more resource intensive.

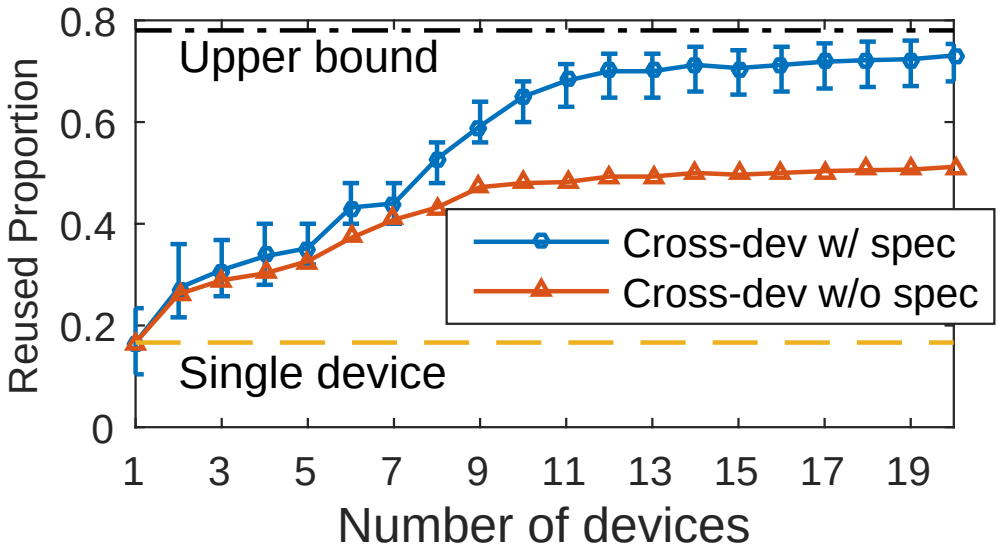


Figure 5.22: Single- or cross-device reuse achieved with *FoggyCache*, with or without speculation.

5.3.5.5 Large-scale experiment

Finally, we run the landmark detection benchmark and examine how the number of devices affects the overall computation reuse opportunities.

Each client device is supplied with 300 input images about 5 landmarks, randomly selected from the Oxford dataset. This corresponds to feeding to the application a 10-second video at 30 fps with no inter-frame correlation. The server and clients caches are configured to be the same sizes as in Section 5.3.5.4 but remain empty before the start, so that the caches evolve solely with the reuse needs on all client devices. We start with a single *FoggyCache* client, and then increase the number of concurrent clients in successive runs.

Figure 5.22 compares the reuse opportunities captured in several scenarios: reuse within the same device only (the "single-device reuse" line), cross-device reuse with or without speculative execution, and the theoretic upper bound which quantifies the intrinsic reuse opportunity within the input dataset. The error-bars are obtained from 10 runs. As the number of devices in the system increases, the percentage of successful reuse also climbs quickly. Once we have more than 10 devices in the system, the reuse proportion stays above 70%. Additional devices provide marginal benefit, given the upper bound is around 80%. The figure also shows that *FoggyCache* outperforms intra-device reuse by more than 55% and *FoggyCache* without speculation by 25%.

5.3.6 Related Work

We are aware of little existing work exploring approximate computation reuse *algorithms* in mobile scenarios. Further, the *FoggyCache system* incorporates these approximate algorithms into the existing execution runtime on mobile devices. We discuss a few approaches closest to ours.

Precise redundancy elimination. Redundancy elimination (RE) is widely employed, e.g., in mobile applications, data analytics [229, 230, 231, 232], networking [280, 281], and storage systems [282]. However, existing schemes involve *exact matching* while *FoggyCache* handles *fuzzy redundancy*.

Starfish [77], MCDNN [19], and Flashback [225] all include caching while accelerating computation-intensive mobile applications. However, they either involve exact matching with cache entries or consider only low-dimensional input values within a pre-defined set. In contrast, *FoggyCache* handles fuzzy input matching without prior knowledge of the data distribution.

UNIC [233] targets security aspects of RE, which are orthogonal to our design and can be combined with *FoggyCache*.

Approximate techniques. Approximate *caching* techniques such as Doppelgänger [234] and Image set compression [283] leverage similarity between data pieces to reduce storage overhead. Approximate *computing* techniques such as ApproxHadoop [284] and Paraprox [285] selectively skip inputs and tasks to reduce computation complexity with tolerable errors. *FoggyCache* adopts similar insights such as exploiting similarity and suppressing error propagation but approximates *repeated computation* using different techniques.

Cachier[286] and Potluck[38] are the closest to *FoggyCache*. Cachier alludes to the notion of approximate reuse but focuses on cache entry optimization, assuming certain query patterns. Our own prior work, Potluck, experiments with avoiding duplicated image recognition and augmented reality rendering for single device. In contrast, *FoggyCache* is more general, achieving high quality reuse and tunable performance, without assumptions about the workload.

Cross-device collaboration. Collaborative sensing and inference systems such as CoMon [287] and Darwin [288] revolve around multi-device coordination in the same context. However, unlike *FoggyCache* eliminating fuzzy redundancy between devices, these cross-device collaboration works focus on partitioning a big job into correlated or independent subtasks, distributing them among the devices, and then collecting the individually results.

5.3.7 *FoggyCache* summary

In this section, we argue for *cross-device approximate computation reuse* for emerging mobile scenarios, where the same application is often run on multiple nearby devices processing similar contextual inputs. Approximate reuse can simultaneously achieve low latency and accurate results, and is a promising optimization technique.

We design techniques, adaptive locality sensitive hashing (A-LSH) and homogenized k nearest neighbors (H-kNN), to address practical challenges to achieve generic approximate computation reuse. We then build *FoggyCache*, which extends the mobile offloading runtime to provide approximate reuse as a service for mobile edge computing. Evaluation shows that, when given 95% accuracy target, *FoggyCache* consistently harnesses over 90% of all reuse opportunities, which translates to reduced computation latency and energy consumption by a factor of 3 to 10. *FoggyCache* provides tuning mechanisms to further improve the accuracy.

While *FoggyCache* is optimized for multi-device mobile and edge scenarios, our reuse techniques A-LSH and H-kNN are generic and have broader applicability. We will investigate other approximate reuse paradigms in future work.

5.4 Harnessing DNN Semantic Correlation for Computation Reuse

Zooming into the workload growth, we already explained in Chapter 4 that new inference workloads are increasingly generated from *similar* deep neural network (DNN) models. The similarity between models stems from three trending practice: (i) Different vendors design custom models to achieve the same goal; (ii) Model variants are derived from the same base models through tuning or compression, though largely yielding the same outcomes; and (iii) Transfer learning facilitates generating new models incrementally from old ones. As a result, there is significant correlation between inference workloads regarding their application semantics.

The problem around DL inference then is that, at the system level, all of the above are treated as if individually new models and distinctive workloads. In other words, we actually need the abstractions to efficiently represent and exploit the *similarity* between models as the key to further accelerate DL inference workloads by eliminating redundancy between DNN models. Fortunately, the semantic equivalence detection techniques mentioned in Section 4.3 is actually the right tool that we can use.

Harnessing the semantic equivalence between deep learning models, we build the corresponding operator for *semantic computation reuse* that approximates a new model with an existing model (Section 5.4.3). This is implemented over MXNet, and further as part of a prediction serving runtime transparent to the application (Section 5.4.4). Extensive evaluation (Section 5.4.5) shows that we improve the inference throughput by 13× with less than 5% accuracy drop, and still up to 7× even with less than 2% accuracy loss.

In summary, we make the following contributions:

First, we highlight the extent of semantic correlation between emerging deep learning inference workloads resulting in redundant inference computation.

Second, based on the DNN semantic equivalence detection techniques (Section 4.3), we build *DeCor* as a runtime service for DNN semantic computation reuse between deep learning inference workloads. Evaluation shows this is a promising approach to accelerate inference workloads without additional resource consumption.

5.4.1 Correlated models leading to computation redundancy

Given the widely existing correlation between DNN models (explained in Section 4.2), if the semantically equivalent models then process the same input during inference, they are likely to incur redundant computation. This is best explained through an intuitive understanding of the inference process.

Equivalence between models. Training deep neural network models are theoretically understood as a function approximation process, where the exact "functionality" is unknown, but are gradually *approximated* through sufficient input data and output labels. The same "function semantics" can be "described" by totally different model representations. The correlated models (or model segments) mentioned in previous chapters are simply examples of models capturing the same task semantics but with different structures and/or parameter values. Therefore, these correlated models should be viewed as *equivalent* during inference execution.

Note that this notion of function semantics is unique to DL and not captured in the traditional sense of program semantics. For a conventional function like `sort()`, it is possible to specify the strongest postcondition uniquely since it produces a unique output given any input. In contrast, the strongest postconditions for a DL task is not unique because multiple outcomes may be acceptable given the same input.

Context-driven inputs. Most real-time DL inference workloads process visual, audio, and textual inputs collected from the local context, and therefore the same inputs recur with a high probability [289]. This observation has motivated several systems [290, 39, 61, 60] to incorporate caching as a key component in their designs. For instance, a single video stream from a surveillance camera triggers the execution of all video analytic workloads installed on the device. Further, successive frames in a stream are almost identical in most cases. Similarly, the same piece of text (e.g., email, webpage) could trigger multiple NLP workloads such as named entity recognition for information extraction and machine translation for non-native readers.

Computation redundancy. When several inference tasks with "equivalent" models (or segments) are invoked on identical or highly similar inputs, the seemingly distinct workloads are in fact semantically equivalent. In this case, processing all the workloads then incurs redundant computation. Ideally, a distinct input should only trigger one of these equivalent workloads, not all.

Table 5.7: Performance speedup opportunity when realizing *equivalent* model segments.

Model	Time saving	Performance comparison
Faster-rcnn	46% saved	0.8 / 0.78 mAP
PSPNet	74% saved	0.84 / 0.82 mIoU

Empirical motivation for optimization. We empirically quantify the level of redundancy between correlated models and segments, namely the performance optimization opportunity. We prepare three workloads, respectively for *recognition*, *detection*, and *segmentation*, following a production setting [154]. The DNN models (PSPNet and Fast-rcnn) for the latter two are transferred from Resnet (trained for *recognition*), with their own successive layers added and fine-tuned based on domain-specific datasets. We emulate computation reuse by comparing the outputs from the *recognition* tasks to the corresponding outputs obtained from the other two workloads. Table 5.7 illustrates the computation time speedup and accuracy loss for the *detection* and *segmentation* tasks following this reuse strategy. Notably, over 3x of performance speedup with less than 2% result of quality loss shows that exploiting the equivalence between models is a promising optimization opportunity.

5.4.2 Measuring equivalence between DNNs

In Section 4.3, we have proposed the algorithms for measuring the correlation (namely, the interchangeability) between DNN models and model segments. Hereby, we further discuss the design and implementation for achieving an automatic equivalence measuring system with high performance.

Whole model equivalence detection. The whole model equivalence detection is mainly built on the idea of using a validation dataset to check the output similarity between two DNNs. Therefore, we implement this module based on the abstraction of TensorFlow Estimator. Specifically, all the configuration settings around the validation dataset and the two DNNs are kept in a `session_config`. Then, the validation dataset are randomly shuffled and fed to the estimator in an infinite loop, wrapped in an `input_fn`. Finally, when the desired generalization level (or the pre-defined maximal size) is reached, the `input_fn` will throw an `out_of_input` exception to finish the `estimator.predict()` function, which eventually completes the whole model equivalence measuring process.

Model segment equivalence detection. The model segment equivalence detection is essentially implemented as a DNN graph traversal pass, following the topological order between the operators.

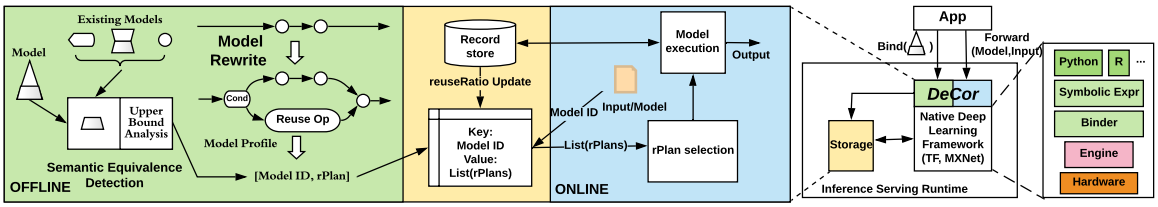


Figure 5.23: *DeCor* system architecture.

The graph traversal logic is the same as the normal forward and backward passes of the DNN computation graph. The main difference for the equivalence detection pass is the `visit()` function for each DNN operator, which will invoke the backend computation kernel associated to the operator to derive the output difference upper bound, based on the upper bound of the input difference. Note that the initial input difference to the two DNN segments are particularly important, as it serves as the bridge to incorporate input semantic equivalence with the DNN segment equivalence. For instance, if the threshold used to determine input data semantic equivalence can be used as the initial input difference upper bound to carry out the model segment equivalence detection.

Extending to new operators. Extending the automatic equivalence measuring module to incorporate new operators is fairly easy, highly similar to adding a new operator (layer) in an existing DL framework. First, we specify the hyper-parameters of the new operator (e.g., input shape, output shape, data type, etc.) and register them to the graph definition template (e.g., NNVM for MXNet, GraphDef for TensorFlow) of our equivalence measuring module. Then, we associate the operator with a backend computation kernel that achieve the semantic correlation analysis. The backend computation kernel accelerates the relevant numerical computation (e.g., calculating the eigenvalue of a matrix, and flattening a convolution kernel into 2D matrix, etc.), which can then be implemented on CUDA, MKL, or other CPU and accelerator libraries for better efficiency. In summary, the two steps are very straightforward, requiring only a few hundred lines of code to plug in to the model segment equivalence measuring module.

5.4.3 Semantic computation reuse

In this section, we explain ***DeCor***, a system that achieves cross-job *semantic computation reuse*, which harnesses the aforementioned semantic equivalence relations across inference jobs to optimize Deep Learning system performance by eliminating semantically redundant computation.

Typical DL framework workflow. Initially, a model should be defined programatically or loaded from a file via certain frontend language (i.e. Python) interface. The DL framework will parse the model, allocate runtime resources, and generate the model executor when the `Bind(model)` API is called. Then, the executor can be used to run inference based on new inputs by calling `Forward(input)` API of the DL framework.

***DeCor* runtime.** Our runtime has two triggering paths, *offline* and *online*, intercepting the two native APIs `Bind(model)` and `Forward(input)` respectively. Figure 5.23 shows *DeCor* architecture. The offline path detects semantic equivalence relations of the model, rewrites it to embed reuse operations, and updates the index of the semantic-centric storage service. The online path carries out the actual semantic reuse operations for each inference task.

5.4.3.1 Offline path

Semantic equivalence detection. As the first module in the offline path, it receives the symbolic computation graph representation (i.e. `nnvm::symbol` for MXNet) of the new model registered. Then, the algorithms described in Section 4.3 and Section 5.4.2 are conducted upon the new model and existing models, which finally mark the start and end positions of all semantically equivalent model segments (or the whole model).

As the whole graph semantic equivalence checking can be costly, we further optimize it to avoid doing pair-wise checking between the new model and all existing models, making the algorithm scalable with the total number of models. Specifically, suppose the existing models A_1 to A_n are semantically equivalent, instead of checking the new model B with all n models, we randomly select three to check. If all three models are determined as equivalent to B, then we associate B with all n existing models as equivalent to each other.

Automatic model rewriting. Having the positions of semantic equivalent segments within a neural network model, the next step is to modify the model so that the reuse operations (including query and insertion to the storage service) can be triggered automatically.

Figure 5.23 gives a simple example of how model rewriting works. Essentially, the model rewriter leverages the control flow operators (supported by almost all mainstream deep learning frameworks [145, 86, 291]) to embed the conditional logic where the actual execution of the marked

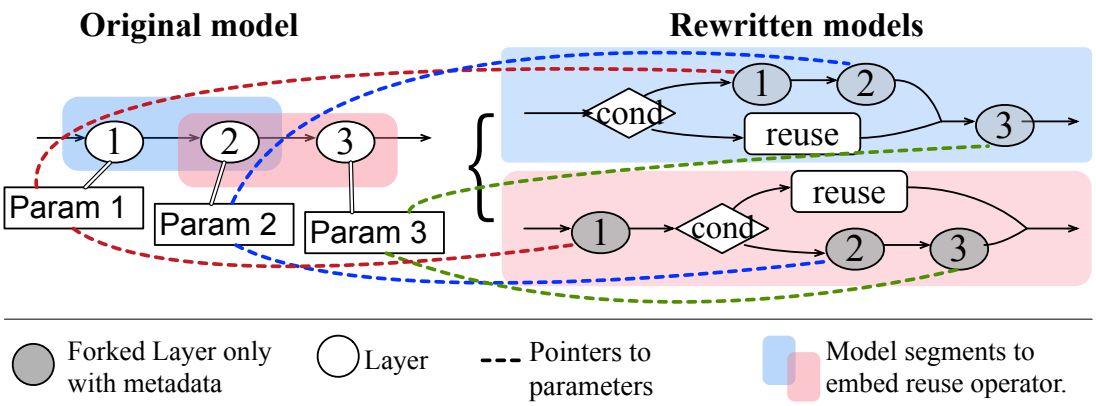


Figure 5.24: Model rewrite with overlapping segments.

segments depends on whether there are existing computation results stored that are reusable. To be specific, suppose we have a model expressed as $G_a \rightarrow G_b \rightarrow G_c$, where G_a, G_b, and G_c are three computation graph segments connected sequentially. If G_b has a semantically equivalent counterpart, for a conditional operator expressed as `cond(ifSym, thenSym, elseSym)`, the `ifSym` corresponds to the operations of querying reusable computation records; `thenSym` corresponds to the operations of determining the reuse outputs; and `elseSym` corresponds to the original computation segment G_b followed by a storage insertion. Finally, the model is rewritten as $G_a \rightarrow cond(if, then, G_b) \rightarrow G_c$.

When multiple reusable model segments overlap with each other, shown by Figure 5.24, their model rewriting plans conflict. In this case, the original model is forked multiple copies and we rewrite them differently for each plan. Note that only the metadata, namely the model attributes with the dataflow dependency between layers, are actually copied which is extremely lightweight. The memory space holding model parameters are not copied but just referenced by corresponding nodes (grey ones in Figure 5.24) in the forked models.

After rewriting, the rewritten model is passed to the native framework to generate the *model executor*. Finally, a profiling step is triggered to measure the *additional overhead* brought by the embedded reuse operations and the *saved computation time* via reuse. After filtering out inefficient models, these numbers along with the *QoR difference bound* derived by previous analysis will be kept with the model executor as its profile, which in total forms a semantic reuse plan (`rPlan` in Figure 5.23).

5.4.3.2 Semantic-centric storage service

To skip computation, we also need a storage service to store inference results from the "equivalent" models. It performs three basic functions: query, insertion, and management.

Semantic-centric indexing. The key design difference compared to traditional storage systems is the indexing structure of our storage service. First, the top-level index of the computation records is no longer the name or hash of the inference model. Instead, we index computation records by their "semantics" (each of which is given a `sID` to denote). Namely, all (intermediate) computation records from different but semantically equivalent model segments will be stored in the same place. Hence, when retrieving reusable computation records, all these semantically equivalent records can all be considered, breaking the intra-model reuse boundary.

Further, to achieve semantic-centric indexing, an additional data structure is necessary, *model-semantic mapping table*, which keeps track of the mapping between each model and its all semantic equivalence information. Each entry in the table corresponds to a distinct upper-level DL model, which contains the mapping from the model ID (`ID`) to a list of semantic reuse plans (`rPlan`). Each `rPlan` includes the specific rewritten model executor (`rModExec`) with reuse operations embedded, and the `rProfile` (computation saving capability, additional overheads, and QoR difference bound) of current plan. Further, the reuse success ratio of each `sID` in the current reuse plan is also kept in the profile. They are used and updated during online operations (Section 5.4.3.3).

Computation record store. The computation records under the same "semantics" are stored and organized by the Locality Sensitive Hashing (LSH) structure [142] to support exact, nearest, and range query. To meet the strict latency requirements of online inference workloads, computation records are all kept in the main memory. Even though, we will show in the evaluation that comparing to the memory footprint needed to execute the DL model, far less additional memory is required to keep previous computation records.

Query and insertion. Query happens when an input invokes a specific model. The model-semantic mapping table is first searched to decide the optimal reuse plan (Section 5.4.3.3). Then, the corresponding model executor is launched, and when the embedded reuse operator is triggered, the LSH structure associated to the corresponding semantics will be queried for reusable computation records based on the input. Insertion happens when new models are registered with reuse plans generated, or

when finishing actual computation where new input-output records are generated. The former case will update the index and mapping table. The latter inserts the record under corresponding semantics.

Storage management. Semantic-centric storage has a runtime profiler carrying out two important management functions, (i) updating real-time reuse statistics, *reuseFreq* for online reuse plan selection (Section 5.4.3.3); and (ii) trim the least efficient reuse-embedded execution graph with respect to the memory consumption limit. During runtime, after accumulating enough real-time statistics (i.e. reuse success ratio) and the memory reaches limit, the least performant reuse plan will be removed by the profiler to save space.

5.4.3.3 Online path

Reuse plan selection. Once a new inference model is invoked with an input data, the first step is to lookup the model-semantic mapping table to select the optimal model executor to run (if multiple exists), which is based on an intuitive heuristic, performance improvement (*Score*), where $Score = \Sigma_{i=1}^{n} reuseFreq_i \cdot compSaved_i - (1 - reuseFreq_i) \cdot overhead_i$ among all components i within the whole model that are wrapped with reuse operations. The reuse plan with the highest *Score* will be finally selected. Noticeably, $compSaved_i$, $overhead_i$, and $reuseFreq_i$ are all kept in the reuse plan.

Reuse operations. Reuse operations include two parts, retrieving computation records and determining reuse output based on these records, which are wrapped in the `if` and `then` statement of the reuse operator respectively. Retrieving computation record is simply a storage system lookup using the current input as the key. If no records are retrieved, reuse fails and will invoke the actual computation logic. If multiple records are returned, the final output is determined as the average among all outputs of the retrieved records.

5.4.3.4 Discussion

Security and privacy. It is a traditional topic about secure and private computation reuse [233]. Further, deep learning poses unique security challenges in overcoming model reverse engineering [292] and adversarial examples [293]. These aspects are not the main focus of our work, and our system does not impede the combination with these approaches. Moreover, as the nature of semantic reuse, the exact input-output records are much harder, if not totally impossible, to be inferred simply by

retrieving the reuse storage.

Scalability. Different from training, online inference workloads hardly run in distributed mode. Therefore, scaling up the model serving system by launching more endpoints (e.g. VMs, containers) still works when *DeCor* module is plugged in. Furthermore, the semantic-centric storage service could easily scale up by migrating it upon distributed in-memory storage (e.g. Redis [294] and Ignite [295]).

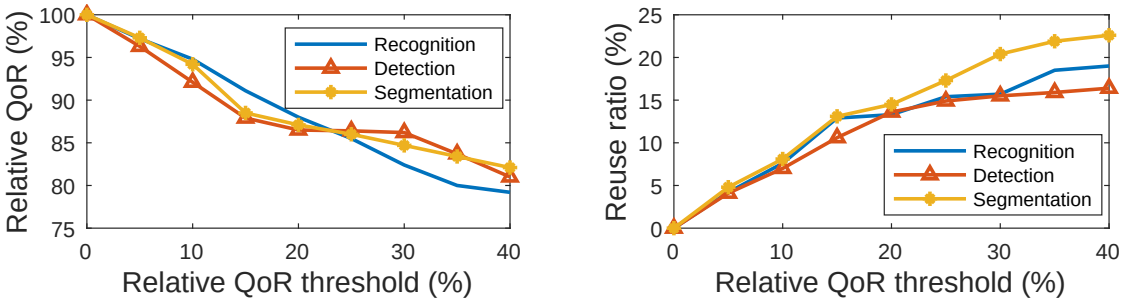
5.4.4 Implementation

We implement *DeCor* over MXNet 1.4 [86], which can run both on the edge (mobile devices and edge servers) and in datacenters. Our implementation consists of around 600 lines of C++ code in the backend for automatic model-rewriting, as well as around 3000 lines of Python code that builds a minimalist inference serving system prototype along with the storage system as a background service.

All the offline and online components are implemented in the backend, so as to minimize frontend-backend API calls. Specifically, semantic equivalence detection module is implemented in `nnvm::graph_editor.cc` that directly handle `nnvm::Symbol`. Then, function rewriter is implemented as a `nnvm::Pass` that operates on internal computation graph `nnvm::Graph`.

***DeCor* interface.** *DeCor* sits between the inference workloads and the native DL framework. It exposes a set of configuration items for each upper-level task to specify their preferences for semantic computation reuse (i.e., equivalence threshold, QoR metrics, validation datasets). On the framework side, *DeCor* intercepts the two native MXNet APIs, `Bind()` and `Forward()`. Invocations of these two native APIs by upper-level applications trigger the offline and online paths of *DeCor* respectively. Meanwhile, the users can specify, in their per-app config file, the error metric and the accuracy target.

Porting to other DL frameworks. Although our implementation is specific to MXNet, the system can be ported easily to other DL frameworks. Prevailing DL platforms follow similar APIs and workflows. For example, TensorFlow provides two APIs: `Create()` creates a runnable session with the newly loaded model, and `Run()` runs the session associated to a model when new inputs are fed. These are the counterparts, respectively, to `Bind()` and `Forward()` in MXNet. Further, the ONNX [173] forum provides cross-framework interpret-ability of DNN models, which raises the



(a) Relative accuracy.

(b) Reuse ratio.

Figure 5.25: Tradeoff between accuracy loss and reuse ratio.

possibility of semantic reuse even across platforms.

5.4.5 Evaluation

5.4.5.1 General setup

Application benchmarks. Using the same set of typical DL application examples (Section 4.6.3), we build six realistic DL inference workloads, three for vision (*image recognition* [148], *object detection* [161], and *semantic segmentation* [162]), and the other three for NLP (*sentiment analysis*, *question and answering (Q & A)*, and *named entity recognition*) [7]. All models and configurations of the six workloads are reproduced from their original papers.

Datasets. We use domain specific standard datasets to fine-tune the six workloads and generate input data streams according to real settings. Specifically, ImageNet [34], Caltech256 [174], and SUN397 [175] are used for object and scene recognition; PascalVOC [176] and MSCOCO [177] are used to fine-tune object detection; Ade20k [178] is used to fine-tune segmentation; SQuAD1.1 [150], IMDB [179], and CoNLL03 [180] are used to fine-tune Q & A, sentiment analysis, and named entity recognition workloads respectively.

Hardware. A Linux server running Ubuntu (16.04) acts as our model server, with a quad-core 2.3 GHz Intel Xeon CPU, 64 GB memory, and an NVIDIA K80 GPU with 24 GB graphic memory. We focus on the server inference scenario in this section, since resource-constrained edge devices only benefit more from *DeCor* based on our empirical results.

5.4.5.2 Accuracy loss vs saving computation

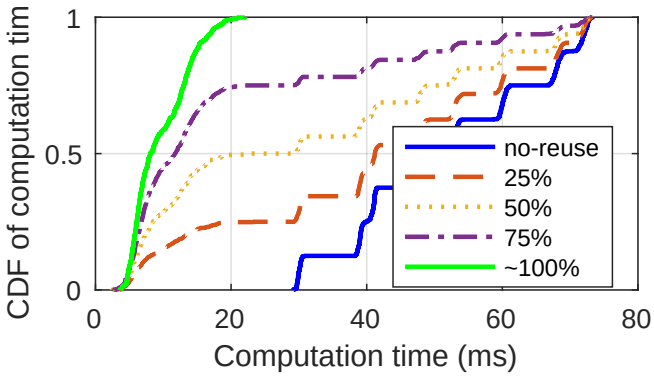
We first prepare 4 similar models, the pre-trained Resnet50 as the base, and transferring it to generate three other models specific to each vision workloads mentioned earlier. The three new models are fine-tuned by randomly selecting 40 parameters and retraining those with task-specific training datasets to different levels, ensuring the relative QoR loss caused by reusing results from pre-trained base model falls evenly between 0% to 40%, with same input streams assembled from the the vision related standard datasets.

We consider two metrics: the portion of reuse leveraged, the accuracy loss relative to what the native model achieves. In Figure 5.25, the x-axis corresponds to the semantic equivalence threshold, controlling the tradeoff between the two metrics. Figure 5.25b shows that *DeCor* consistently captures over 85% as reusable, among all actually reusable fine-tuned models. Meanwhile, Figure 5.25a shows that the actual QoR loss caused by reuse is always below the QoR threshold. These suggest *DeCor* could perform well on both metrics. Although it gradually lose track of reuse opportunities when setting semantic equivalence threshold over 10%, such loose semantic equivalence threshold hardly makes any practical sense.

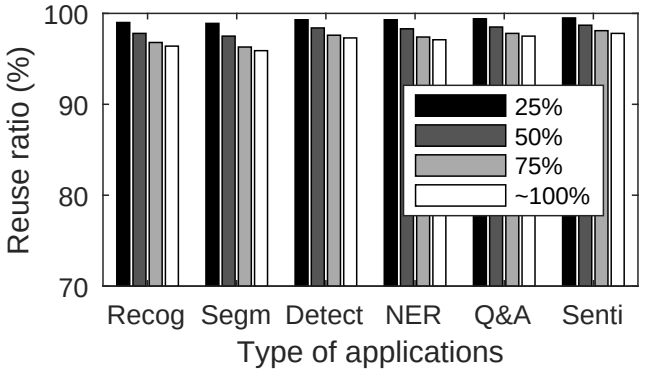
5.4.5.3 End-to-end performance

Settings. We use all six CV and NLP workloads described in Section 5.4.5.1. 90% of the CV tasks have models transferred from the Resnet50 base model, whereas the other 10% are chosen from the MXNet model zoo for the same task. The NLP workloads are all transferred from the BERT [7] base model, with different layers added to achieve different downstream tasks. The input data are synthesized from mixing randomly selected data from the nine standard datasets (all collected from real scenarios) described earlier, to emulate input streams from real settings. Specifically, we generate 1000-entry input data streams with different data correlation frequency (25% to 100%) to measure our system performance. The semantic equivalence threshold is set to 5% for most experiments, which means we want less than 5% QoR loss incurred by semantic computation reuse. This corresponds to the median accuracy drop within the range of 2-10% considered acceptable for speedup [151]. We also experiment with 2% accuracy drop as a conservative scenario.

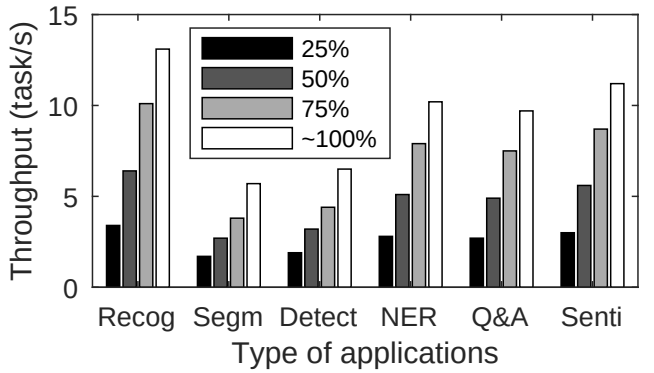
Performance. We evaluate the system using four metrics: relative QoR, processing time distribution



(a) Processing time CDF.



(b) Relative QoR.



(c) Task throughput.

Figure 5.26: End-to-end performance.

of all inference tasks, system throughput (tasks handled per second) and the corresponding resource consumption interpretation. Figure 5.26 shows the performance of *DeCor* for all workloads and input streams. The four bars in each group correspond to the performance at different input correlation levels (25% to 100%), i.e., the maximum portion of reuse possible.

DeCor reduces the median task processing time by 40% with 50% reusable inputs, and could reduce the task processing time by up to 85% when nearly all inputs are reusable. Compared to the

Table 5.8: Time of gauging semantic equivalence.

Metrics	Alexnet	ResNet	VGG19	BERT
# Params (M)	62	60	143	340
Time (Seg)	1.89s	2.77s	5.46s	14.10s
Time (Whole)	1.25s	4.46s	6.18s	22.92s

native framework, *DeCor* further improves the task throughput by up to 13×, which translates to a 10× reduction of resource consumption to achieve the same response time. *DeCor* achieves such performance improvement with less than 4% relative QoR penalty. Even if the equivalence threshold is 2%, *DeCor* could still achieve a median task throughput improvement of around 2×, and up to 7.4×.

Performance predictability. Admittedly, the reported performance is workload-specific, but it provides useful hints to predict *DeCor* performance given specific settings. 100% reuse (optimal) typically provides an order of magnitude improvement on the processing latency. On that basis we can estimate the overall performance improvement by further considering the proportion of recurring input. Multiplying the improvement numbers at optimal reuse (100% reuse opportunities leveraged) by the input recurrence ratio provides a simple but effective estimate.

5.4.5.4 Additional system overhead

DeCor introduces several additional operations to the system runtime. In this section, we profile them individually.

Latency of semantic equivalence detection. Recall that *DeCor* assesses semantic equivalence between models offline (Section 5.4.3.1), which is not on the critical path of processing inference workloads. We mainly consider whether *DeCor* can handle huge DNN models.

We use the four models (listed as Table 5.8 column titles) as the inputs to dry-run the whole model and model segment equivalence detection algorithms respectively, and measure the average time consumption.

Table 5.8 shows the time needed to detect equivalent model segments and determine whole model equivalence. We can clearly see that the algorithm scales well when the model size is extremely large. Even for the huge BERT model which would consume over 12 GB memory during runtime, our algorithm still finishes within around 20s, which is far less than the average time that a new model is

Table 5.9: Reuse operation latency.

Records	5K	50K	500K	5M
Time (ms)	0.02	0.05	3.6	13.7

Table 5.10: Memory footprint with/without semantic reuse.

Model	Memory	# Reuse plans	Addi. (MB)
Resnet50	0.89 GB	1 / 5 / 10	16 / 41 / 65
VGG19	3.28 GB	1 / 5 / 10	12 / 34 / 59

designed and registered to serve inference.

Latency of reuse operations. The reuse operations are on the online paths (Section 5.4.3.3). The main latency overhead is from the LSH-based nearest record search [142] useful for CV workloads, since completely identical inputs are rare [39]. This is much slower than the extremely fast (ns level), hash-based exact search previously used for computation reuse.

We prepare the storage with records of 2 KB each, varying the number of records from 5K to 5M. In each case, the storage is queried 20 times, and we time the search operations. Table 5.9 shows the average search time vs storage size. The online reuse operation is fast enough even considering similarity-based retrieval. In practice, the storage size needed is far smaller than 10 GB, and 5 ms is the typical retrieval latency, much faster than actually processing an inference task.

Memory and computation overhead for rewritten models. Since *DeCor* rewrites the original computation graph of a new model to generate reuse plans by embedding reuse operators (Section 5.4.3.1), additional memory is needed. However, this should be negligible since only the metadata of the model are copied and modified. Table 5.10 shows the added memory consumption incurred by randomly generating 1 - 10 reuse plans for Resnet50 and VGG19. The additional memory footprint is mostly under 50 MB, indeed negligible compared to the memory footprint of the original model. This can be further reduced when the background profiler trims inefficient plans.

Rewritten models may also incur computation overhead unnecessarily during run time in the *absence* of reuse records, despite the *presence* of similar models at "compile" time. *DeCor* avoids this with the offline profiling stage of model writing. Table 5.11 shows that *DeCor* incurs up to 7% additional execution time when embedding as many as 6 reuse operators (corresponding to 6 different possibilities of model origin). In practice, it is unlikely that a single model is derived from more than six sources of base models.

Table 5.11: Computation overhead for rewritten models.

# Reuse Ops	1	2	3	4	5	6
Addi. time (%)	3.8	4.9	5.6	6.1	6.5	7

5.4.6 Related Work

We are not aware of prior work that discusses or leverages semantic equivalence across DL inference workloads for system optimization. The most related work otherwise revolves around accelerating DL inference or redundancy elimination.

System optimizations for Deep Learning inference. To accelerate DL inference workloads, existing approaches variously optimize the computation graph [20], the prediction serving runtime [61, 60], and hardware support [74, 72]. These require proposing new programming models, revising system workflows and resource management, or hardware-specific support. In contrast, we harness the intrinsic semantic correlation between inference workloads for inter-job optimization. Our approach is agnostic to the workload semantics, the underlying hardware or the framework. The *DeCor* runtime is transparent to applications and does not involve intrusive modification to the native DL frameworks.

Redundancy elimination. Redundancy elimination (RE) is widely employed, e.g., in DL serving systems, data analytics [229, 230, 231, 232], databases [296], and storage systems [282]. Unlike *DeCor*, however, none of the existing systems harnesses the *semantic equivalence* between functions.

Clipper [60], Pretzel [61] and MCDNN [19] all include caching to accelerate latency-sensitive inference tasks. They either treat the DL model as a blackbox or require the app developer to manually annotate the exact segment to be shared globally. In contrast, *DeCor* automatically extracts semantic equivalence relations between jobs. For data analytics frameworks, DryadInc [230], Nectar [232], Differential dataflow [229], and SEeSAW [231] achieve cross-job RE with a centralized cache service and a program rewritter to suppress redundant execution paths. Although deep learning workloads are run on these frameworks, only recurrent jobs (the same job run repeatedly) benefit from the RE mechanism. The system abstractions and programming models are also quite different. UNIC [233] targets security aspects of RE, which are orthogonal to our design and can be combined with *DeCor*. All existing proposals assume *precise function matching*, i.e., reuse is conditioned on the same function being invoked again and again. While this might suit traditional (discrete) programs, it will

increasingly miss truly "overlapping" DL inference computation, since DL inference workloads are more amorphous.

5.4.7 *DeCor* summary

As new deep learning inference workloads are increasingly generated from variants of existing pre-trained models or transfer learning, we can observe significant correlation between models and workloads. However, existing system solutions treat all inference workloads as individually distinct, often navigating a difficult tradeoff between minimizing response time, inference error, and resource consumption.

In this section, we provide primitives to efficiently represent and expose the semantic similarity between DNN models, and an immediate application is to provide semantic computation reuse. Our system, *DeCor*, is built as a runtime extension of MXNet, transparent to the application. Extensive evaluation shows that our system can accelerate the inference workloads by 13× while incurring within 5% accuracy drop relative to the performance of the native models.

Looking ahead, we expect the equivalence detection to serve as a building block for further system optimizations, for example, to help restructure the computation pipeline across workloads. We will explore those in future work.

Chapter 6

Conclusion and Future Direction

6.1 Conclusion

This thesis takes a first step in building the end-to-end service abstractions for enabling efficient deep learning inference at the edge. We first highlight the existing challenges in the end-to-end process of deploying deep learning inference tasks from cloud to heterogeneous edge devices and propose a universal framework to resolve them, which consists of a series of new service abstractions to the users. Specifically, we build *Mistify* - an automated DNN model porting service that accelerates and scales the customization of DNN logic towards heterogeneous edge settings (Chapter 3); *Sommelier* - a DNN model indexing and query service that simplify the manual, time and computation efforts of selecting the optimal DNN model to achieve the desired functionality under specific constraints and budgets (Chapter 4); and *Potluck*, *FoggyCache*, and *DeCor* that collectively perform approximate caching and computation reuse on the inference workloads to accelerate the speed of inference execution and meanwhile improve resources and energy efficiency (Chapter 5).

From a system perspective, these services abstract away the required interdisciplinary expertise and laborious manual efforts from the users, and significantly lower the barrier of deploying deep learning inference logic. From an algorithmic perspective, we adopt a unique paradigm in our solutions, treating learning-based workloads collectively as white boxes, proposing algorithms that harness the hidden correlation between them to optimize the resource efficiency of the whole process of deploying deep learning inference tasks.

While building these systems, we further form a deeper understanding of DNN based workloads

and develop an initial approach to measure the semantics of DNN models. Looking forward, we believe this will lead to opportunities far beyond just optimizing inference workload preparation and execution. It should be further extended to use as a novel primitive to explore the explainable AI, ethical AI, and AI fairness. I hope my thesis could be a stepping stone that helps facilitate more future works in this area, which would ultimately bring Deep Learning and AI technology seamlessly running around all of us.

6.2 Future directions

My generic approach of understanding and harnessing the features of AI-driven workloads to design service abstractions that automate manual services and achieve seamless execution of them at the edge is a promising direction far beyond this thesis. AI is moving extremely fast. In the foreseeable future, the learning-based functions will become increasingly dynamic and heterogeneous in their processing logic, execution preferences, and etc. Therefore, a promising direction based on this thesis is to build more future-proof system framework that dynamically coordinates both the cloud and the heterogeneous edge devices and covers the whole life cycle of machine learning, including data ingestion, training, validation and testing, porting, and inference. Moreover, the unique algorithmic approaches proposed by this thesis could be leveraged in a much broader scope (e.g., from a database and/or compiler perspective) to further optimize the ML systems. Following are a few promising directions to explore.

Continuous learning. According to the recent research, the lifecycle of the DNN model is becoming long-lasting and dynamic, instead of the single-shot training and inference. Under such trend of online continuous learning, the data ingestion and model evolution process become essential to the success of a DL function. Leveraging the primitives to analyze semantic correlation between new data collected and the model checkpoints along the evolution paths, we can build systems for continuous learning focusing on higher data efficiency, faster iterations, and robust to data noise and malicious training behaviors.

Data management for explainable AI. Machine Learning testing as well as deep learning explainability is an area gaining much focus. When using ML techniques for autonomous driving and medical scenarios, the security, robustness and predictability of DNN models are critical. Several recent systems (e.g., DeepXplore [163]) proposed to automatically detect adversarial scenarios by reasoning the decision boundary between semantically similar models. In light of this, it would be interesting to build tools to reason, compare, and analyze DNN model internals (e.g., *diff*, *debug*, and *generative* tools), based on the *semantic correlation* notion mentioned above. These systems could tell the fine-grained relations between two models and generate tailored data for DNN testing under targeted scenario. An immediate usage could be generating artificial models from a pre-trained model to construct the environment for DNN testing, contributing to much more automated, robust,

and controllable DNN robustness testing. Eventually, our notion of *semantic correlation* would serve as a key towards explainable AI, uncovering the missing connections between DNN (or even human brain) neuron topology and the functional semantics.

Semantic-invariant deep learning compiler. Another promising direction to explore is adding the novel primitives of measuring semantic correlation to the existing DL compiler stack (e.g., TVM [20]). Existing deep learning compilers optimize the computation graph of a DNN model by performing greedy rule-based graph transformations and operator kernel generation, both of which only consider transformations that strictly guarantee mathematical equivalence. Harnessing the function approximation nature of machine learning, we can further explore the opportunities of extending the current DL workload optimization from mathematical to semantic invariant.

Bibliography

[1] M. V. Barbera, A. Epasto, A. Mei, S. Kosta, V. C. Perta, and J. Stefa. CRAWDAD dataset sapienza/probe-requests (v. 2013-09-10). Downloaded from `https://crawdad.org/sapienza/probe-requests/20130910`, September 2013.

[2] MobiSys'19 IoT Day. `https://www.sigmobile.org/mobisys/2019/iot_day_program/`.

[3] N. V. Kim and M. A. Chervonenkis. Situation control of unmanned aerial vehicles for road traffic monitoring. *Modern Applied Science*, 9(5):1, 2015.

[4] Foghorn: The Original Edge-Native AI Platform. `https://www.foghorn.io/edge-ai-platform/`.

[5] Driving intelligent retail with AI. `https://www.nvidia.com/en-us/industries/retail/`.

[6] S. Ravi and Z. Kozareva. Self-governing neural networks for on-device short text classification. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 887–893, 2018.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[8] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong. Redeye: analog convnet image

sensor architecture for continuous mobile vision. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 255–266. IEEE Press, 2016.

[9] Aetina: dedicated AI computing solution at edge. https://www.aetina.com/index.php.

[10] Avnet. Ai at the edge: The next frontier of the internet of things, 2018.

[11] Gartner highlights 10 uses for ai-powered devices. https://www.gartner.com/en/newsroom/press-releases/2018-03-20-10-uses-for-ai-powered-devices.

[12] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[13] W. Qualcomm. We are making on-device ai ubiquitous, 2018.

[14] S. A. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan. The emerging landscape of edge computing. *GetMobile: Mobile Computing and Communications*, 23(4):11–20, 2020.

[15] 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.

[16] D. Evans. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, 1(2011):1–11, 2011.

[17] Tensorflow hub: A repository of reusable assets for machine learning with tf. https://github.com/tensorflow/hub.

[18] Deploy machine learning models on mobile and IoT devices. https://www.tensorflow.org/lite.

[19] 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.

[20] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[21] J. Lin, W. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han. Mcunet: Tiny deep learning on iot devices. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[22] M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114, 2019.

[23] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

[24] Share of smartphone models sold in the united states from 2017 to 2020. `https://www.statista.com/statistics/755671/united-states-smartphone-market-share-by-model/`.

[25] Edge tpu: Google's purpose-built asic designed to run inference at the edge. `https://cloud.google.com/edge-tpu`.

[26] S. Bianco, R. Cadene, L. Celona, and P. Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018.

[27] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[28] Y. Mao, Y. Wang, C. Wu, C. Zhang, Y. Wang, Y. Yang, Q. Zhang, Y. Tong, and J. Bai. Ladabert: Lightweight adaptation of bert through hybrid model compression. *arXiv preprint arXiv:2004.04124*, 2020.

[29] J. Wang, J. Pan, and F. Esposito. Elastic urban video surveillance system using edge computing. In *Proceedings of the Workshop on Smart Internet of Things*, page 7. ACM, 2017.

[30] N. Chen, Y. Chen, Y. You, H. Ling, P. Liang, and R. Zimmermann. Dynamic urban surveillance video stream processing using fog computing. In *2016 IEEE second international conference on multimedia big data (BigMM)*, pages 105–112. IEEE, 2016.

[31] C. Xia, J. Zhao, H. Cui, X. Feng, and J. Xue. Dnntune: Automatic benchmarking dnn models for mobile-cloud computing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(4):1–26, 2019.

[32] World around you with Google Lens and the Assistant. `https://www.blog.google/products/assistant/world-around-you-google-lens-and-assistant/`.

[33] Google Street View Image API.

[34] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[35] Economist. The cost of training machines is becoming a problem. `https://www.economist.com/technology-quarterly/2020/06/11/the-cost-of-training-machines-is-becoming-a-problem`, 2020.

[36] E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for deep learning in NLP. In A. Korhonen, D. R. Traum, and L. Màrquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 3645–3650. Association for Computational Linguistics, 2019.

[37] Mistify: Automating DNN model porting for on-device inference at the edge. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[38] P. Guo and W. Hu. Potluck: Cross-application approximate deduplication for computation-intensive mobile applications. In *Proceedings of the Twenty-Third International Conference*

on Architectural Support for Programming Languages and Operating Systems, ASPLOS '18, pages 271–284, New York, NY, USA, 2018. ACM.

[39] P. Guo, B. Hu, R. Li, and W. Hu. Foggycache: Cross-device approximate computation reuse. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, pages 19–34, New York, NY, USA, 2018. ACM.

[40] X. Wang, Y. Han, V. C. Leung, D. Niyato, X. Yan, and X. Chen. Convergence of edge computing and deep learning: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 22(2):869–904, 2020.

[41] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[42] 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.

[43] 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.

[44] M. Bansal, A. Krizhevsky, and A. S. Ogale. Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst. In A. Bicchi, H. Kress-Gazit, and S. Hutchinson, editors, *Robotics: Science and Systems XV, University of Freiburg, Freiburg im Breisgau, Germany, June 22-26, 2019*, 2019.

[45] Y. He, N. Zhao, and H. Yin. Integrated networking, caching, and computing for connected vehicles: A deep reinforcement learning approach. *IEEE Transactions on Vehicular Technology*, 67(1):44–55, 2017.

[46] R. Q. Hu et al. Mobility-aware edge caching and computing in vehicle networks: A deep reinforcement learning. *IEEE Transactions on Vehicular Technology*, 67(11):10190–10203, 2018.

[47] D. Vasisht, Z. Kapetanovic, J. Won, X. Jin, R. Chandra, S. Sinha, A. Kapoor, M. Sudarshan, and S. Stratman. Farmbeats: An iot platform for data-driven agriculture. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 515–529, 2017.

[48] Alexa Skills Kit: Add Voice to Your Big Idea and Reach More Customers. `https://developer.amazon.com/alexa-skills-kit`.

[49] W. Qualcomm. Mobile processors that are smooth, fast, and powerful. `https://www.qualcomm.com/products/mobile-processors`.

[50] S. Mittal. A survey on optimized implementation of deep learning models on the NVIDIA jetson platform. *J. Syst. Archit.*, 97:428–442, 2019.

[51] B. Hu and W. Hu. Linkshare: device-centric control for concurrent and continuous mobile-cloud interactions. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 15–29, 2019.

[52] M. Fredrikson, E. Lantz, S. Jha, S. Lin, D. Page, and T. Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 17–32, 2014.

[53] A. Frome, G. Cheung, A. Abdulkader, M. Zennaro, B. Wu, A. Bissacco, H. Adam, H. Neven, and L. Vincent. Large-scale privacy protection in google street view. In *2009 IEEE 12th international conference on computer vision*, pages 2373–2380. IEEE, 2009.

[54] 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.

[55] H. Vanholder. Efficient inference with tensorrt, 2016.

[56] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[57] Deep learning for Java. https://deeplearning4j.org/.

[58] Integrate machine learning models into your app. https://developer.apple.com/documentation/coreml.

[59] Y. Ma, D. Yu, T. Wu, and H. Wang. Paddlepaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing*, 1(1):105–115, 2019.

[60] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[61] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. {PRETZEL}: Opening the black box of machine learning prediction serving systems. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 611–626, 2018.

[62] W. Wang, S. Wang, J. Gao, M. Zhang, G. Chen, T. K. Ng, and B. C. Ooi. Rafiki: machine learning as an analytics service system. *arXiv preprint arXiv:1804.06087*, 2018.

[63] Z. Ahmed, S. Amizadeh, M. Bilenko, R. Carr, W.-S. Chin, Y. Dekel, X. Dupre, V. Eksarevskiy, S. Filipi, T. Finley, et al. Machine learning at microsoft with ml. net. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2448–2458, 2019.

[64] Mxnet model server (mms). https://github.com/awslabs/mxnet-model-server, 2018.

[65] A. Lavin and S. Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[66] J. Cong and B. Xiao. Minimizing computation in convolutional neural networks. In *International conference on artificial neural networks*, pages 281–290. Springer, 2014.

[67] M. Mathieu, M. Henaff, and Y. LeCun. Fast training of convolutional networks through ffts. In Y. Bengio and Y. LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[68] D. H. Bailey, K. Lee, and H. D. Simon. Using strassen's algorithm to accelerate the solution of linear systems. *The Journal of Supercomputing*, 4(4):357–371, 1991.

[69] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):1–11, 2016.

[70] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

[71] Y.-H. Chen, J. Emer, and V. Sze. Using dataflow to optimize energy efficiency of deep neural network accelerators. *IEEE Micro*, 37(3):12–21, 2017.

[72] 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.

[73] M. Dukhan, Y. Wu, and H. Lu. Qnnpack: open source library for optimized mobile deep learning.

[74] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[75] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, pages 269–284. ACM, 2014.

[76] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, et al. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE, 2014.

[77] R. LiKamWa and L. Zhong. Starfish: Efficient concurrency support for computer vision applications. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 213–226. ACM, 2015.

[78] L. N. Huynh, Y. Lee, and R. K. Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 82–95, 2017.

[79] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81. ACM, 2014.

[80] J. Wang, Z. Feng, S. George, R. Iyengar, P. Pillai, and M. Satyanarayanan. Towards scalable edge-native applications. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 152–165, 2019.

[81] M. Abadi, M. Isard, and D. G. Murray. A computational model for tensorflow: an introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 1–7, 2017.

[82] C. Lattner, J. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[83] 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 *OSDI*, volume 16, pages 265–283, 2016.

[84] NNVM: reusable graph IR for deep learning systems. `https://github.com/apache/tvm/tree/main/nnvm`.

[85] J. Roesch, S. Lyubomirsky, L. Weber, J. Pollock, M. Kirisame, T. Chen, and Z. Tatlock. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 58–68, 2018.

[86] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[87] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[88] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[89] Z. Jia, J. Thomas, T. Warszawski, M. Gao, M. Zaharia, and A. Aiken. Optimizing dnn computation with relaxed graph substitutions. *SysML 2019*, 2019.

[90] 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 *International Conference on Machine Learning*, pages 2430–2439. PMLR, 2017.

[91] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[92] M. Sivathanu, T. Chugh, S. S. Singapuram, and L. Zhou. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 909–923, 2019.

[93] T. Chen, L. Zheng, E. Q. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy. Learning to optimize tensor programs. In S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 3393–3404, 2018.

[94] 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.

[95] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[96] 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.

[97] S. Ravi. Projectionnet: Learning efficient on-device deep networks using neural projections. *arXiv preprint arXiv:1708.00630*, 2017.

[98] Neural Network Distiller by Intel AI Lab: a Python package for neural network compression research. `https://github.com/NervanaSystems/distiller`.

[99] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[100] A. Gordon, E. Eban, O. Nachum, B. Chen, H. Wu, T.-J. Yang, and E. Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1586–1595, 2018.

[101] H. Cai, C. Gan, and S. Han. Once for all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019.

[102] X. Dai, P. Zhang, B. Wu, H. Yin, F. Sun, Y. Wang, M. Dukhan, Y. Hu, Y. Wu, Y. Jia, et al. Chamnet: Towards efficient network design through platform-aware model adaptation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11398–11407, 2019.

[103] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[104] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. In *5th International Conference on Learning Representations, ICLR*

2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net, 2017.

[105] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 8697–8710. IEEE Computer Society, 2018.

[106] H. Liu, K. Simonyan, and Y. Yang. DARTS: differentiable architecture search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[107] E. Real, C. Liang, D. R. So, and Q. V. Le. Automl-zero: Evolving machine learning algorithms from scratch. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 8007–8019. PMLR, 2020.

[108] A. Kolesnikov, L. Beyer, X. Zhai, J. Puigcerver, J. Yung, S. Gelly, and N. Houlsby. Big transfer (bit): General visual representation learning. 2019.

[109] S. Arora, R. Ge, B. Neyshabur, and Y. Zhang. Stronger generalization bounds for deep nets via a compression approach. In J. G. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 254–263. PMLR, 2018.

[110] D. Zou, Y. Cao, D. Zhou, and Q. Gu. Gradient descent optimizes over-parameterized deep relu networks. *Machine Learning*, 109(3):467–492, 2020.

[111] S. Arora, S. Du, W. Hu, Z. Li, and R. Wang. Fine-grained analysis of optimization and generalization for overparameterized two-layer neural networks. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 322–332. PMLR, 09–15 Jun 2019.

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

[113] R. Anil, G. Pereyra, A. Passos, R. Ormandi, G. E. Dahl, and G. E. Hinton. Large scale distributed neural network training through online distillation. *arXiv preprint arXiv:1804.03235*, 2018.

[114] T. Li, J. Li, Z. Liu, and C. Zhang. Knowledge distillation from few samples. *arXiv preprint arXiv:1812.01839*, 2018.

[115] 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.

[116] Pytorch mobile. `https://pytorch.org/mobile/home/`, 2019.

[117] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[118] D. Mahajan, R. Girshick, V. Ramanathan, K. He, M. Paluri, Y. Li, A. Bharambe, and L. van der Maaten. Exploring the limits of weakly supervised pretraining. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 181–196, 2018.

[119] V. Popov, M. Kudinov, I. Piontkovskaya, P. Vytovtov, and A. Nevidomsky. Distributed fine-tuning of language models on private data. 2018.

[120] J. Yim, D. Joo, J. Bae, and J. Kim. A gift from knowledge distillation: Fast optimization, network minimization and transfer learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4133–4141, 2017.

[121] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-10 and cifar-100 datasets. *URl: https://www. cs. toronto. edu/kriz/cifar. html*, 6, 2009.

[122] S. Jain, G. Ananthanarayanan, J. Jiang, Y. Shu, and J. Gonzalez. Scaling video analytics systems to large camera deployments. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 9–14. ACM, 2019.

[123] 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 *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[124] J. M. Alvarez and M. Salzmann. Learning the number of neurons in deep networks. In *Advances in Neural Information Processing Systems*, pages 2270–2278, 2016.

[125] E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for deep learning in nlp. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3645–3650, 2019.

[126] MorphNet: a library of learning deep network structure during training. `https://github.com/google-research/morph-net`.

[127] Y. Xiong, R. Mehta, and V. Singh. Resource constrained neural network architecture search: Will a submodularity assumption help? In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1901–1910, 2019.

[128] A. Shin, D. Y. Kim, J. S. Jeong, and B.-G. Chun. Hippo: Taming hyper-parameter optimization of deep learning with stage trees. *arXiv preprint arXiv:2006.11972*, 2020.

[129] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.

[130] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

[131] R. Shokri and V. Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1310–1321, 2015.

[132] S. Kullback and R. A. Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[133] N. Papernot, S. Song, I. Mironov, A. Raghunathan, K. Talwar, and Úlfar Erlingsson. Scalable private learning with pate. In *ICLR*, 2018.

[134] N. Hynes, R. Cheng, and D. Song. Efficient deep learning on multi-source private data. *CoRR*, abs/1807.06689, 2018.

[135] N. Papernot, M. Abadi, Ú. Erlingsson, I. J. Goodfellow, and K. Talwar. Semi-supervised knowledge transfer for deep learning from private training data. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[136] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu. Deepcache: Principled cache for mobile deep vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 129–144, 2018.

[137] M. Mirman, T. Gehr, and M. Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning*, pages 3578–3586, 2018.

[138] J. Zhang, Y. Wang, P. Molino, L. Li, and D. S. Ebert. Manifold: A model-agnostic framework for interpretation and diagnosis of machine learning models. *IEEE transactions on visualization and computer graphics*, 25(1):364–373, 2018.

[139] A. Ruoss, M. Baader, M. Balunović, and M. Vechev. Efficient certification of spatial robustness. *arXiv preprint arXiv:2009.09318*, 2020.

[140] S. Parthasarathy and M. Ogihara. Exploiting dataset similarity for distributed mining. In *International Parallel and Distributed Processing Symposium*, pages 399–406. Springer, 2000.

[141] A. B. L. Larsen, S. K. Sønderby, H. Larochelle, and O. Winther. Autoencoding beyond pixels using a learned similarity metric. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 1558–1566. JMLR.org, 2016.

[142] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.

[143] J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17(5):419–428, 2001.

[144] R. Lee, S. I. Venieris, L. Dudziak, S. Bhattacharya, and N. D. Lane. Mobisr: Efficient on-device super-resolution through heterogeneous mobile processors. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[145] 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 Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[146] P. Hintjens. *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

[147] Protocol buffers: language-neutral, platform-neutral extensible mechanism for serializing structured data. `https://developers.google.com/protocol-buffers/`.

[148] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[149] M. Seo, A. Kembhavi, A. Farhadi, and H. Hajishirzi. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*, 2016.

[150] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. Squad: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, 2016.

[151] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278. IEEE, 2016.

[152] TensorFlow Official Models. `https://github.com/tensorflow/models`.

[153] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani. Deep Learning for IoT Big Data and Streaming Analytics: A Survey. *IEEE Communications Surveys & Tutorials*, 2018.

[154] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. Khudia, J. Law, P. Malani, A. Malevich, S. Nadathur, et al. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*, 2018.

[155] N. J. Yadwadkar, F. Romero, Q. Li, and C. Kozyrakis. A case for managed and model-less inference serving. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 184–191, 2019.

[156] Intel(r) math kernel library for deep neural networks (intel(r) mkl-dnn). `https://intel.github.io/mkl-dnn/`.

[157] Open source deep learning code and pretrained models. `https://modelzoo.co/`.

[158] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

[159] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[160] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.

[161] S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[162] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia. Pyramid scene parsing network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2881–2890, 2017.

[163] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[164] H. Karimi, T. Derr, and J. Tang. Characterizing the decision boundary of deep neural networks. *arXiv preprint arXiv:1912.11460*, 2019.

[165] C. Canel, T. Kim, G. Zhou, C. Li, H. Lim, D. G. Andersen, M. Kaminsky, and S. R. Dulloor. Scaling video analytics on constrained edge nodes. *SysML*, 2019.

[166] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

[167] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2018.

[168] G. Singh, T. Gehr, M. Püschel, and M. Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[169] B. Neyshabur, S. Bhojanapalli, and N. Srebro. A PAC-bayesian approach to spectrally-normalized margin bounds for neural networks. *arXiv preprint arXiv:1707.09564*, 2017.

[170] P. L. Bartlett, D. J. Foster, and M. J. Telgarsky. Spectrally-normalized margin bounds for neural networks. In *Advances in Neural Information Processing Systems*, pages 6240–6249, 2017.

[171] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[172] C. Cortes, X. Gonzalvo, V. Kuznetsov, M. Mohri, and S. Yang. Adanet: Adaptive structural learning of artificial neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 874–883. JMLR. org, 2017.

[173] ONNX: Open Neural Network Exchange Format. `https://onnx.ai/`.

[174] G. Griffin, A. Holub, and P. Perona. Caltech-256 object category dataset. 2007.

[175] J. Xiao, J. Hays, K. A. Ehinger, A. Oliva, and A. Torralba. Sun database: Large-scale scene recognition from abbey to zoo. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 3485–3492. IEEE, 2010.

[176] 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.

[177] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[178] B. Zhou, H. Zhao, X. Puig, S. Fidler, A. Barriuso, and A. Torralba. Scene Parsing through ADE20K Dataset. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[179] A. L. 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, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[180] E. F. Sang and F. De Meulder. Introduction to the conll-2003 shared task: Language-independent named entity recognition. *arXiv preprint cs/0306050*, 2003.

[181] P. Kraft, D. Kang, D. Narayanan, S. Palkar, P. Bailis, and M. Zaharia. Willump: A statistically-aware end-to-end optimizer for machine learning inference. *arXiv preprint arXiv:1906.01974*, 2019.

[182] G. Singh, R. Ganvir, M. Püschel, and M. Vechev. Beyond the single neuron convex barrier for neural network certification. In *Advances in Neural Information Processing Systems*, pages 15072–15083, 2019.

[183] M. Balunovic and M. Vechev. Adversarial training and provable defenses: Bridging the gap. In *International Conference on Learning Representations*, 2020.

[184] Y. Tian, K. Pei, S. Jana, and B. Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314, 2018.

[185] D. Kang, D. Raghavan, P. Bailis, and M. Zaharia. Model assertions for monitoring and improving ml model. *arXiv preprint arXiv:2003.01668*, 2020.

[186] U. Ehsan, P. Tambwekar, L. Chan, B. Harrison, and M. O. Riedl. Automated rationale generation: a technique for explainable ai and its effects on human perceptions. In *Proceedings of the 24th International Conference on Intelligent User Interfaces*, pages 263–274, 2019.

[187] R. Fagin, R. Guha, R. Kumar, J. Novak, D. Sivakumar, and A. Tomkins. Multi-structural databases. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 184–195, 2005.

[188] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1579–1590, 2014.

[189] F. Abuzaid, P. Kraft, S. Suri, E. Gan, E. Xu, A. Shenoy, A. Ananthanarayan, J. Sheu, E. Meijer, X. Wu, et al. Diff: a relational interface for large-scale data explanation. *Proceedings of the VLDB Endowment*, 12(4):419–432, 2018.

[190] D. Y. Yoon, N. Niu, and B. Mozafari. Dbsherlock: A performance diagnostic tool for transactional databases. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1599–1614, 2016.

[191] S. Roy, A. C. König, I. Dvorkin, and M. Kumar. Perfaugur: Robust diagnostics for performance anomalies in cloud services. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1167–1178. IEEE, 2015.

[192] Openshade with visually impaired users. `http://www.openshades.com/`.

[193] A. Shashua, Y. Gdalyahu, and G. Hayun. Pedestrian detection for driving assistance systems: Single-frame classification and system level performance. In *Intelligent Vehicles Symposium, 2004 IEEE*, pages 1–6. IEEE, 2004.

[194] J. Flinn. Cyber foraging: Bridging mobile and cloud computing. *Synthesis Lectures on Mobile and Pervasive Computing*, 7(2):1–103, 2012.

[195] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4), 2009.

[196] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-aware Adaptation for Mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 276–287, New York, NY, USA, 1997. ACM.

[197] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[198] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn. Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Mobile Cloud Gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 151–165, New York, NY, USA, 2015. ACM.

[199] 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. ACM, 2015.

[200] How stores will use augmented reality. `https://www.technologyreview.com/s/601664/`.

[201] Google's indoor VPS navigation by Tango-ready phone. `http://mashable.com/2017/05/google-visual-positioning-service-tango-augmented-reality/`.

[202] PokeMon Go augmented reality game. `http://www.pokemongo.com/`.

[203] X. Liu, H. Li, X. Lu, T. Xie, Q. Mei, H. Mei, and F. Feng. Understanding Diverse Smarpthone Usage Patterns from Large-Scale Appstore-Service Profiles. *arXiv preprint arXiv:1702.05060*, 2017.

[204] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer, 2006.

[205] J. Le Feuvre, J. Thiesse, M. Parmentier, M. Raulet, and C. Daguet. Ultra high definition HEVC DASH data set. In *Proceedings of the 5th ACM Multimedia Systems Conference*, pages 7–12. ACM, 2014.

[206] J. Hafner, H. S. Sawhney, W. Equitz, M. Flickner, and W. Niblack. Efficient color histogram indexing for quadratic form distance functions. *IEEE transactions on pattern analysis and machine intelligence*, 17(7):729–736, 1995.

[207] F. Suard, A. Rakotomamonjy, A. Bensrhair, and A. Broggi. Pedestrian detection using infrared images and histograms of oriented gradients. In *Intelligent Vehicles Symposium, 2006 IEEE*, pages 206–212. IEEE, 2006.

[208] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo. LEO: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proceedings of the Annual International Conference on Mobile Computing and Networking, MobiCom16*. ACM, 2016.

[209] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[210] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. In *European conference on computer vision*, pages 430–443. Springer, 2006.

[211] C. Harris and M. Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988.

[212] N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[213] K. Walsh and E. G. Sirer. Experience with an object reputation system for peer-to-peer filesharing. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 1–1, Berkeley, CA, USA, 2006. USENIX Association.

[214] N-Dimensional Arrays for Java. http://nd4j.org/.

[215] Open-source computer vision. http://opencv.org/.

[216] P. Mermelstein. Distance measures for speech recognition, psychological and instrumental. *Pattern recognition and artificial intelligence*, 116:374–388, 1976.

[217] I. Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[218] Openhft java runtime compiler. https://github.com/OpenHFT/Java-Runtime-Compiler.

[219] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)*, 27(5):126, 2008.

[220] Android Interface Definition Language for IPC. https://developer.android.com/guide/components/aidl.html.

[221] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[222] Y. LeCun, C. Cortes, and C. J. Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2, 2010.

[223] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 179–194. ACM, 2010.

[224] W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3d warping. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 7–ff. ACM, 1997.

[225] K. Boos, D. Chu, and E. Cuervo. Flashback: Immersive virtual reality on mobile devices via rendering memoization. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 291–304. ACM, 2016.

[226] Z. Wang, D. Liu, J. Yang, W. Han, and T. Huang. Deep networks for image super-resolution with sparse prior. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 370–378, 2015.

[227] G. Bertasius, J. Shi, and L. Torresani. Deepedge: A multi-scale bifurcated deep network for top-down contour detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4380–4389, 2015.

[228] T. Zhang, A. Chowdhery, P. V. Bahl, K. Jamieson, and S. Banerjee. The design and implementation of a wireless video surveillance system. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 426–438. ACM, 2015.

[229] F. D. McSherry, R. Isaacs, M. A. Isard, and D. G. Murray. Differential dataflow, October 20 2015. US Patent 9,165,035.

[230] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: Reusing Work in Large-scale Computations. In *HotCloud*, 2009.

[231] K. Kannan, S. Bhattacharya, K. Raj, M. Murugan, and D. Voigt. Seesaw-similarity exploiting storage for accelerating analytics workflows. In *HotStorage*, 2016.

[232] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *OSDI*, volume 10, pages 1–8, 2010.

[233] Y. Tang and J. Yang. Secure deduplication of general computations. In *USENIX Annual Technical Conference*, pages 319–331, 2015.

[234] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger. Doppelgänger: a cache for approximate computing. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 50–61. ACM, 2015.

[235] H. Wu, X. Sun, J. Yang, W. Zeng, and F. Wu. Lossless Compression of JPEG Coded Photo Collections. *IEEE Transactions on Image Processing*, 25(6):2684–2696, 2016.

[236] H. Wang, T. Tian, M. Ma, and J. Wu. Joint Compression of Near-Duplicate Videos. *IEEE Transactions on Multimedia*, 2016.

[237] Build powerful reactive, concurrent, and distributed applications more easily. https://akka.io/.

[238] C. Dixon, R. Mahajan, S. Agarwal, A. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 25–25. USENIX Association, 2012.

[239] Amazon Alexa: a first look at reach figures. https://omr.com/en/amazon-alexa-skill-marketing/.

[240] Smart home, seamless life: Unlocking a culture of convenience, January 2017.

[241] S. Vijayarangan, P. Sodhi, P. Kini, J. Bourne, S. Du, H. Sun, B. Poczos, D. Apostolopoulos, and D. Wettergreen. High-throughput robotic phenotyping of energy sorghum crops. *Field and Service Robotics. Springer*, 2017.

[242] H. Verkasalo. Contextual patterns in mobile service usage. *Personal and Ubiquitous Computing*, 13(5):331–342, 2009.

[243] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[244] R. Huitl, G. Schroth, S. Hilsenbeck, F. Schweiger, and E. Steinbach. TUMindoor: An extensive image and point cloud dataset for visual indoor localization and mapping. In *Proc. of the International Conference on Image Processing*, Orlando, FL, USA, September 2012. Dataset available at http://navvis.de/dataset.

[245] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision*, 42(3):145–175, 2001.

[246] R. Hu and J. Collomosse. A performance evaluation of gradient field hog descriptor for sketch based image retrieval. *Computer Vision and Image Understanding*, 117(7):790–806, 2013.

[247] H. Hermansky. Perceptual linear predictive (plp) analysis of speech. *the Journal of the Acoustical Society of America*, 87(4):1738–1752, 1990.

[248] N. D. Lane, P. Georgiev, and L. Qendro. Deepear: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 283–294, New York, NY, USA, 2015. ACM.

[249] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[250] M. Zhao, D. Wang, Z. Zhang, and X. Zhang. Music removal by convolutional denoising autoencoder in speech recognition. In *Signal and Information Processing Association Annual Summit and Conference (APSIPA), 2015 Asia-Pacific*, pages 338–341. IEEE, 2015.

[251] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, D. S. Pallett, N. L. Dahlgren, and V. Zue. Timit acoustic-phonetic continuous speech corpus. *Linguistic data consortium*, 10(5):0, 1993.

[252] A. Gionis, P. Indyk, R. Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, volume 99, pages 518–529, 1999.

[253] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *Acm Sigmod Record*, volume 19, pages 322–331. Acm, 1990.

[254] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, volume 93, pages 311–321, 1993.

[255] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.

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

[257] C. J. Stone. Consistent nonparametric regression. *The annals of statistics*, pages 595–620, 1977.

[258] Y. Li and B. Cheng. An improved k-nearest neighbor algorithm and its application to high

resolution remote sensing image classification. In *Geoinformatics, 2009 17th International Conference on*, pages 1–4. Ieee, 2009.

[259] F. Laviolette and M. Marchand. Pac-bayes risk bounds for stochastic averages and majority votes of sample-compressed classifiers. *Journal of Machine Learning Research*, 8(Jul):1461–1487, 2007.

[260] S. Kpotufe. k-nn regression adapts to local intrinsic dimension. In *Advances in Neural Information Processing Systems*, pages 729–737, 2011.

[261] D. Duvenaud, O. Rippel, R. Adams, and Z. Ghahramani. Avoiding pathologies in very deep networks. In *Artificial Intelligence and Statistics*, pages 202–210, 2014.

[262] S. Gu and L. Rigazio. Towards deep neural network architectures robust to adversarial examples. *arXiv preprint arXiv:1412.5068*, 2014.

[263] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young. Mobile edge computing—a key technology towards 5g. *ETSI White Paper*, 11(11):1–16, 2015.

[264] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 43–56. ACM, 2011.

[265] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen. Comet: Code offload by migrating execution transparently. In *OSDI*, volume 12, pages 93–106, 2012.

[266] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review*, 26(3):22–36, 1996.

[267] A. Parate, M. Böhmer, D. Chu, D. Ganesan, and B. M. Marlin. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*, pages 275–284. ACM, 2013.

[268] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148. IEEE Computer Society Press, 1981.

[269] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 191–205. ACM, 2005.

[270] S. Gaonkar, J. Li, R. R. Choudhury, L. Cox, and A. Schmidt. Micro-blog: sharing and querying content through mobile phones and social participation. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 174–186. ACM, 2008.

[271] L. Buttyan and J.-P. Hubaux. *Security and cooperation in wireless networks: thwarting malicious and selfish behavior in the age of ubiquitous computing*. Cambridge University Press, 2007.

[272] P. Mohan, V. N. Padmanabhan, and R. Ramjee. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 323–336. ACM, 2008.

[273] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[274] Open source speech recognition toolkit. `https://cmusphinx.github.io/`.

[275] A. Gordo, J. Almazán, J. Revaud, and D. Larlus. Deep image retrieval: Learning global representations for image search. In *European Conference on Computer Vision*, pages 241–257. Springer, 2016.

[276] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2007.

[277] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[278] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, et al. Going deeper with convolutions. Cvpr, 2015.

[279] W. Hu, Y. Gao, K. Ha, J. Wang, B. Amos, Z. Chen, P. Pillai, and M. Satyanarayanan. Quantifying the impact of edge computing on mobile applications. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, page 5. ACM, 2016.

[280] A. Anand, V. Sekar, and A. Akella. Smartre: an architecture for coordinated network-wide redundancy elimination. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 87–98. ACM, 2009.

[281] S. Sanadhya, R. Sivakumar, K.-H. Kim, P. Congdon, S. Lakshmanan, and J. P. Singh. Asymmetric caching: improved network deduplication for mobile devices. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 161–172. ACM, 2012.

[282] L. DuBois, M. Amaldas, and E. Sheppard. Key considerations as deduplication evolves into primary storage. *White Paper*, 223310, 2011.

[283] Z. Shi, X. Sun, and F. Wu. Feature-based image set compression. In *Multimedia and Expo (ICME), 2013 IEEE International Conference on*, pages 1–6. IEEE, 2013.

[284] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 383–397. ACM, 2015.

[285] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 35–50. ACM, 2014.

[286] U. Drolia, K. Guo, J. Tan, R. Gandhi, and P. Narasimhan. Cachier: Edge-caching for recognition applications. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 276–286. IEEE, 2017.

[287] Y. Lee, Y. Ju, C. Min, S. Kang, I. Hwang, and J. Song. Comon: Cooperative ambience monitoring platform with continuity and benefit awareness. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 43–56. ACM, 2012.

[288] E. Miluzzo, C. T. Cornelius, A. Ramaswamy, T. Choudhury, Z. Liu, and A. T. Campbell. Darwin phones: the evolution of sensing and inference on mobile phones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 5–20. ACM, 2010.

[289] S. Jain, G. Ananthanarayanan, J. Jiang, Y. Shu, and J. Gonzalez. Scaling video analytics systems to large camera deployments. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 9–14. ACM, 2019.

[290] S. Yang. Iot stream processing and analytics in the fog. *IEEE Communications Magazine*, 55(8):21–27, 2017.

[291] R. Collobert, S. Bengio, and J. Mariéthoz. Torch: a modular machine learning software library. Technical report, Technical Report IDIAP-RR 02-46, IDIAP, 2002.

[292] S. J. Oh, M. Augustin, B. Schiele, and M. Fritz. Towards reverse-engineering black-box neural networks. *arXiv preprint arXiv:1711.01768*, 2017.

[293] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58. ACM, 2011.

[294] Redis: in-memory data structure store. `https://redis.io/`.

[295] Ignite: memory-centric distributed database, caching, and processing. `https://ignite.apache.org/`.

[296] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *ACM SIGMOD Record*, volume 30, pages 307–318. ACM, 2001.