UNIVERSITY OF CALIFORNIA
Los Angeles

Making Video Analytics Applications Efficient and Affordable

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

by

Arthi Padmanabhan

2022

ABSTRACT OF THE DISSERTATION

Making Video Analytics Applications Efficient and Affordable

by

Arthi Padmanabhan
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2022
Professor Harry Guoqing Xu, Co-Chair
Professor Ravi Arun Netravali, Co-Chair

While using machine learning to analyze video data is seeing explosive growth, modern vision models are difficult and expensive to deploy in practice. This is because while models are getting more accurate and robust, they are also getting more complicated and thus more resource-intensive. At the same time, the environments in which they are used, such as self-driving cars, demand extremely fast and accurate results.

Traditionally, all video data was sent to cloud servers, where models were run over the frames on GPU machines. Recently though, the use of edge computing has shown promise in addressing this tension between performance and resource usage. Resources available at the edge are highly heterogeneous in terms of computational power and memory, and while most prior work assumes a well-equipped edge, we find that the devices used in practice are often inexpensive commodity hardware. This limits the amount of computation that can practically happen at the edge.

In this thesis, we aim to make the most of these resource-constrained edge devices. We present two systems that improve the tradeoff between performance and resource usage in live video analysis. Our first system, Reducto, uses the limited amount of compute available on smart cameras to run cheap computer vision techniques and filter out frames that are similar enough to the previous frame that we can reuse the previously computed result

as an approximation. This lowers GPU usage by over 50% and doubles processing speed. Our next system, GEMEL, addresses the memory bottleneck of running many models on an edge server by finding and merging common layers across a diverse set of models. This lowers the memory footprint by up to 60% and improves accuracy by up to 39%.

The dissertation of Arthi Padmanabhan is approved.

Ganesh Ananthanarayanan

Omid Salehi-Abari

George Varghese

Ravi Arun Netravali, Committee Co-Chair

Harry Guoqing Xu, Committee Co-Chair

University of California, Los Angeles

2022

To my mom,

Radhika Padmanabhan

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would first like to thank my advisors, Harry Xu and Ravi Netravali, for their mentorship throughout my PhD. They each had a unique impact on my growth as a researcher and person. I am grateful that Harry encouraged me to work on projects I found fascinating, and that he always kept his door open for impromptu brainstorming sessions. I thank Ravi for teaching me to think critically about research problems. He encouraged me to express my opinions even when I doubted myself and in the process helped me develop confidence as a researcher. I am especially grateful to both of them for being patient and supportive as I figured out my career direction.

I would like to thank George Varghese, who initially piqued my interest in networking and systems through fascinating courses. Thank you to Omid Abari for his valuable feedback on my work as a member of my thesis committee. I am extremely grateful to Ganesh Ananthanarayanan for taking a chance on me with an internship and enabling a great collaboration with the MNR group at Microsoft Research. I would also like to thank my collaborators. In addition to my advisors, I enjoyed working with and learned so much from: Yuanqi Li, Pengzhan Zhao, Yufei (Fiona) Wang, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, and Nikos Karianakis.

I am so lucky to have been a part of two labs, both with wonderful cultures and amazing people: Neil Agarwal, Shaghayegh Mardani, Pradeep Dogga, Lana Ramjit, Murali Ramanujam, Mike Wong, John Thorpe, Jon Eyolfson, Chenxi Wang, Christian Navasca, Usama Hameed, Yifan Xiao, Haoran Ma, Jiyuan Wang, Pengzhan Zhao, Shi Liu, and Yuanqi Li. Thank you for being great sounding boards for research ideas, for all the wonderful lab dinners, and for keeping me connected and sane with catch up meetings during the pandemic. I am particularly grateful to Jon Eyolfson, who patiently helped me iterate on my defense presentation, prepared me for interviews, and has been an amazing resource in computer science education.

I would like to thank Joseph Brown for taking care of all paperwork for my funding and leave of absence. I am also thankful for the Graduate Society of Women Engineers for providing a sense of community through trivia nights and coffee socials, as well as UCLA CAPS, particularly Colby Moss and Dr. Ian Mathis, for their impact on my well-being.

I would also like to acknowledge the amazing teachers and mentors who have supported my education and career. In particular, thank you to Yi Chen, who inspired me to try research at Pomona, advised my first project, and is someone I am very lucky to have in my professional and personal life. I am also grateful for Prof. Everett Bull, who taught my first CS class at Pomona and remains an invaluable resource, as well as Anand Rajeswaran, my Microsoft manager, mentor and friend.

Thank you to my Southern CA family for the lovely dinners, hikes, and tennis sessions over the last five years, especially Jamie, Nancy, Bob, the Currys, and the Thorpes. I am also very grateful for my cousins, aunts, and uncles - Kala's giant bags of homemade granola and Mika's cookie packages got me through some rough deadlines during the PhD! Thank you Vikram, Shreya, Alex and Shyam for many years of friendship and for always making time for wonderful gatherings over delicious food.

Of all the benefits of joining this lab at UCLA, meeting my life partner was the most unexpected. John has been a constant source of joy and support, and I feel lucky that we were able to go through the PhD experience together.

I am grateful beyond words for my dad, Ravi, for putting up with me, letting me talk through research frustrations, and celebrating accomplishments with me.

Finally, this thesis is dedicated to my mom and best friend, Radhika. While she didn't get to see the end of this adventure, her emphasis on intellectual curiosity and her never-ending belief in me were major reasons it happened.

The work presented in this thesis was supported by a UCLA Graduate Division Fellowship and contain modified versions of published co-authored material. Chapter 2 is a version of [89], and Chapter 3 is a version of [106] (PIs: Harry Guoqing Xu and Ravi Arun Netravali).

VITA

Research/Teaching Assistant — September 2017 - June 2022
University of California, Los Angeles, CA

Research Intern, Mobility and Networking Research — June 2020 - August 2020
Microsoft Research

B.A. Computer Science — August 2010 - May 2014
Pomona College

PUBLICATIONS

Arthi Padmanabhan*, Neil Agarwal*, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Harry Guoqing Xu, and Ravi Netravali. GEMEL: Model Merging for Memory-Efficient, Real-Time Video Analytics at the Edge. arXiv preprint arXiv:2201.07705, 2022

Arthi Padmanabhan, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Harry Xu, and Ravi Netravali. Towards Memory-Efficient Inference in Edge Video Analytics. In the Proceedings of the 3rd ACM Workshop on Hot Topics in Video Analytics and Intelligent (HotEdgeVideo 2021).

Yuanqi Li*, **Arthi Padmanabhan***, Pengzhan Zhou, Yufei Wang, Harry Xu, and Ravi Netravali. Reducto: On-Camera Frame Filtering for Resource-Efficient Real-Time Video Analytics. In the Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM 2020).

Arthi Padmanabhan, Lan Wang, and Lixia Zhang. Automated tunneling over IP Land: run NDN anywhere (poster). In the Proceedings of the 5th ACM Conference on Information-Centric Networking (ICN 2018).

CHAPTER 1

Introduction

The use of machine learning to analyze live video is seeing explosive growth, fueled by increasing amounts of video data being generated as well as the consistent improvement of machine learning models [37, 82, 86, 91, 130]. For example, in augmented reality applications, machine learning models track and understand the 3D world, allowing the experience to be interactive. In responding to Amber Alerts, models are run over all video footage within a certain area to look for a specific car and license plate number. In self-driving cars, models identify all objects and their locations relative to the car, allowing the car to make driving decisions. What all these applications have in common is the need for the machine learning model to produce results quickly and accurately. While in augmented reality applications, using a slow or faulty model might cause frustration to the user and lost revenue, in self-driving cars it could be fatal.

The need for fast and accurate results is complicated by the fact that these machine learning models, as they are getting more accurate and robust, are also getting larger and more complicated. They are now typically deep neural networks (DNNs) that require more computations, take longer to run, and use more memory. Figure 1.1 shows the trend in the number of parameters, which corresponds to the number of computations required, over the last 20 years. This tension between increased model complexity and the demanding environments in which models are run is the subject of this thesis. We first describe typical video analytics pipelines and then present methods to improve the trade-off space between performance and resource usage.

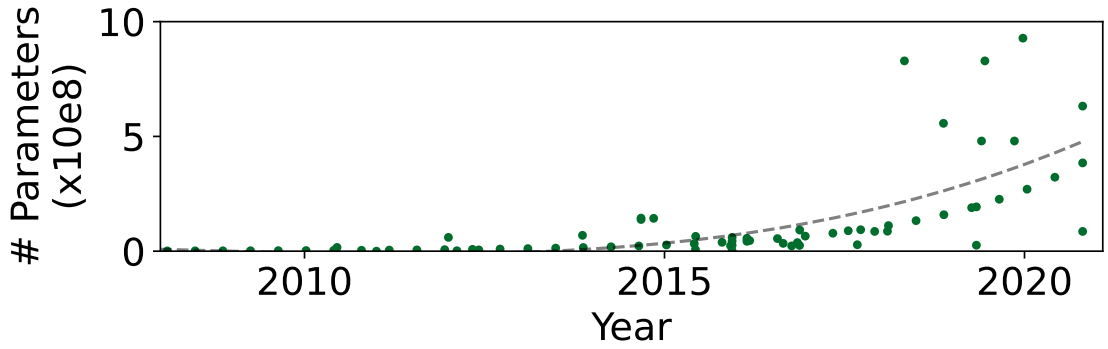


Figure 1.1: Parameter counts in popular vision DNNs over time. Data drawn from [121].

1.1 Background & Motivation

Video Analytics Pipelines. Video analytics pipelines [63, 74, 145, 158] typically consist of cameras that stream their video content over a network to a cloud server, where frames are pre-processed and then fed into a DNN, such as YOLO [116]. The results are used to respond to queries such as "return the location of all cars" or "return the number of buses", where responses are given per-frame.

The first major goal of such a pipeline is to respond with high accuracy. Typically, queries are given with an accuracy constraint, or the amount of error the query can tolerate, relative to running the most expensive DNN (which typically translates to the largest DNN available) over every frame in the video. The second goal is to respond with low latency. This often means running at 30 frames per second, which is the frame rate of real-time video. In other cases, the query will provide a service level agreement (SLA), specifying that the frame must be processed within a certain amount of time (e.g., 100 ms) after it is recorded by the camera; otherwise the result is no longer useful.

Achieving these two goals is challenging because this pipeline is extremely resource intensive, both in terms of network bandwidth and compute cost. Sending a single video at 1080p at 30 fps requires 2 Mbps, and if we consider that usually deployments consist of several (tens of) cameras sending video over the same network, the bandwidth consumes adds up quickly and causes delays in frames getting to the cloud server. In terms of compute cost, while models are getting more accurate, they are also getting larger and more computationally

intensive [19, 71, 72, 142]. For example, Faster RCNN, a state-of-the-art object detector, takes 6 seconds to process 1 second of video on a Tesla P100, a $6000 GPU machine.

One solution is to simply set up multiple networks and use several GPU machines per video stream. However, this is prohibitively costly to most organizations i.e., cities and enterprises will not spend hundreds of thousands of dollars to analyze their video streams. We thus aim to approach this problem with a focus on affordability: how can lower resource usage enough to make running these video analytics pipelines affordable, while still meeting accuracy constraints?

Edge Computing for Video Analytics. A key insight that this work leverages is that in video analytics pipelines, moving some computation to the edge enables several performance benefits [30, 101, 141]. For example, if we could use the compute at the edge to filter out unnecessary or redundant frames before they reach the cloud server, we could lower the bandwidth of sending all those frames, thus reducing network delays. This would also lower the compute requirements because only some frames would need to be run through the DNN [38, 40, 63, 76]. Moving computation to the edge could also be necessary to meet SLAs in cases where the result is needed at the edge itself, such as self-driving cars, as the round-trip time to the cloud would be prohibitive [58, 90, 159]. Further, in case of disconnection to the cloud, an edge component adds resilience to the pipeline [5, 103]; for example, we could use the edge to continue running high priority jobs, i.e., we could miss getting traffic flow patterns for a short time but still run Amber Alerts. Finally, moving computation to the edge can address privacy concerns around streaming video to the cloud [99, 110].

Resource-Constrained Edge Devices. The devices that can be used at the edge are highly varied, ranging from laptops to edge boxes to even cameras [1, 2, 43, 51, 102, 137]. While most prior work has assumed a fairly well-equipped edge in terms of computational power and memory, we find that in practice, deployments use resource-constrained inexpensive hardware. These contain less processing power than the cloud (i.e., CPU only and/or

lower clockspeed) as well as significantly less memory. This work studies these edge devices that are used in practice and explores how to make the most of their limited resources.

1.2 Thesis Approach

The key goal of this work is to improve the performance of video analytics pipelines using inexpensive resource-constrained edge devices. We present two strategies for approaching this goal that are each tailored to an edge device that is currently deployed. The first considers smart cameras with a weak CPU and uses the camera to efficiently decide when previously computed query results can be safely reused (i.e., when skipping a frame can still meet accuracy targets). The second strategy considers edge boxes whose GPUs contain limited memory compared the cloud. Here, we look *within* machine learning models for redundancies in layers and we merge redundant layers to lower the models' memory footprint.

1.3 Reusing Query Results When Possible

To cope with the high resource (network and compute) demands of real-time video analytics pipelines, recent systems have relied on frame filtering. However, filtering has typically been done with neural networks running on edge/backend servers that are expensive to operate. This paper investigates *on-camera filtering*, which moves filtering to the beginning of the pipeline. We studied existing deployments and found that in practice, cameras have limited compute resources that are not powerful enough to run neural networks. Our solution, **Reducto** (§2), presents a method to determine a cheap predicate that even the most resource-constrained cameras can run to determine which frames to send. This predicate is based on low-level differences between frames, such as pixel comparison or edge detection. Used incorrectly, such techniques can lead to unacceptable drops in query accuracy. To overcome this, Reducto dynamically adapts filtering decisions according to the time-varying correlation between feature type, filtering threshold, query accuracy, and video content. Experiments with a variety of videos and queries show that with Reducto, we can use the small CPU available in most existing deployments to filter out up to half the frames and lower the end-to-end latency of the pipeline by up to 26% while consistently meeting accuracy targets.

1.4 Exploiting Redundancy Between Models

While in Reducto, we treated the DNN as a black box (i.e., we changed the inputs and assessed the accuracy of the outputs), our second strategy involves diving into the DNNs themselves. Here, we consider a common pipeline where edge boxes are deployed close to the cameras and are aimed at running several DNNs per video stream while processing several video streams. We find that edge-box GPUs lack the memory needed to concurrently house the growing number of (increasingly complex) models for real-time inference. Unfortunately, existing solutions that rely on time/space sharing of GPU resources are insufficient as the required swapping delays result in unacceptable frame drops and accuracy loss. We present *model merging*, a new memory management technique that exploits architectural similarities between edge vision models by judiciously sharing their layers (including weights) to reduce workload memory costs and swapping delays. Our system, **GEMEL** (§3), efficiently integrates merging into existing pipelines by (1) leveraging several guiding observations about per-model memory usage and inter-layer dependencies to quickly identify fruitful and accuracy-preserving merging configurations, and (2) altering edge inference schedules to maximize merging benefits. Experiments across diverse workloads reveal that GEMEL reduces memory usage by up to 60%, and improves overall accuracy by up to 39% relative to time or space sharing alone.

CHAPTER 2

Reducto: On-Camera Filtering for Resource-Efficient Video Analytics

2.1 Overview

Significant work has been expended to improve the efficiency of video analytics pipelines [38, 40, 63, 74, 97, 158]. Across these systems, a prevailing (and natural) strategy is to improve efficiency by *filtering out* frames that do not contain relevant information for the query at hand [38, 40, 63, 76]. Conceptually, filtering out a frame requires understanding how that frame would affect a query result. To make such decisions without needing the actual query results (which would negate filtering benefits), existing systems employ various levels of *approximations* based on either (1) *compressed* object detection models (e.g., Tiny YOLO [116]) that compute lower-confidence results [63], (2) specialized *binary classification* models that eliminate frames that do not contain an object of interest [38, 76], or (3) simple *frame differencing* to eliminate frames whose low-level features (e.g., pixel values) have not changed substantially (based on a static threshold) and are expected to produce the same results [40].

On-camera filtering. In this work, unlike prior filtering approaches that typically run on edge [98] or backend servers, we seek to filter frames at the beginning of the analytics pipeline – *directly on cameras*. Like edge server approaches, on-camera filtering has the potential to alleviate not only backend *computation overheads* (by reducing the number of frames that must be processed by the backend object detector), but also end-to-end *network bottlenecks* between cameras and backend servers, particularly for wireless cameras [38, 58,

160]. Furthermore, an on-camera approach can also sidestep the management and cost overheads of operating edge servers [96, 117]. We note that in targeting on-camera filtering, our aim is to eliminate the reliance on edge servers for filtering by making use of currently unused resources. Our on-camera filtering techniques could also run on edge servers (if present), outperforming existing strategies while consuming fewer resources (§2.5).

Despite the potential benefits, our study of commodity cameras and surveillance deployments paints a bleak resource picture (§2.2.1). In contrast to edge servers, smartphones, or recent smart cameras that possess GPUs and AI hardware accelerators, deployed cameras often have low-speed CPUs (1 GHz) and modest amounts of RAM (256 MB). These resources preclude even compressed NNs for filtering (e.g., Tiny YOLO runs at < 1 fps), and instead can only tolerate specialized binary classification NNs or frame differencing strategies. Unfortunately, we find that these approaches are far more limited for filtering (§2.2.2). Binary classification strategies forego between 17-74% of potential frame filtering opportunities by filtering based on object presence rather than changes in query result, e.g., a parking lot can contain parked cars, but the overall count or locations of the cars may not change. In contrast, existing frame differencing strategies consistently violate query accuracy requirements by filtering out necessary frames (reasons described below).

Goal and insight. In this work, we ask: can we integrate on-camera filtering into video analytics pipelines in a way that achieves most of the potential filtering benefits without violating accuracy goals? Due to the aforementioned filtering limitations inherent to binary classification, we turn to *frame differencing with low-level features*.

Our key insight is that the lack of accuracy preservation with existing frame differencing strategies is *not* a problem inherent to low-level features, but rather a problem of these features not being used appropriately. For example, Glimpse [40] filters by comparing pixel-level frame differences against a static threshold, and is unable to adapt to the heterogeneous queries (e.g., detection, counting) and dynamic video content that analytics pipelines are faced with [74, 94]. This is because the same difference values may carry different meanings (in terms of changes in query results) for different video content and query types, e.g., a

traffic light may warrant a lower threshold than a busy highway. We assert that if we can (1) establish a correlation between feature types, their filtering thresholds, and query accuracy, and (2) dynamically adjust this correlation in response to changes in queries and video content, these cheap features can be surprisingly effective (more than NN-based techniques!) in indicating if filtering a frame will cause accuracy violations.

Reducto. Based on this insight, we developed Reducto, a simple and yet inexpensive solution to the real-time video analytics efficiency problem, that tackles three main challenges.

(C1) What low-level video features to use? The computer vision (CV) community [35, 40, 80, 83, 113, 114, 134, 139, 162, 163] has discovered a slew of low-level video features that extract frame differences [36], such as `Edge` and `Pixel`. To find the most appropriate features for on-camera filtering, we carefully studied a representative set of them (§2.3). An important observation we make is that the "best" feature (i.e., the one that most closely tracks changes in query results) to use varies across query classes more so than across different videos (see §2.4.2). This is because each feature uniquely captures a certain low-level video property; different query classes are interested in different video properties, and hence fit the best with different features. Based on this observation, the Reducto server performs offline profiling of historical video data to determine the best feature *for each query class*. The server notifies the camera of the feature it should use for each new query. Note that this is in contrast to existing strategies that always use the `Pixel` feature [40].

(C2) How to select filtering thresholds? Filtering frames using a differencing feature inherently requires cameras to select a parameter (i.e., a differencing threshold). Selecting the appropriate threshold is paramount as this value directly impacts the accuracy and filtering benefits of Reducto: too low of a threshold will sacrifice filtering benefits, while too high of a threshold may sacrifice accuracy. However, selecting this threshold value is difficult as the optimal threshold varies rapidly, on the order of seconds, due to the inherent dynamism in video content (§2.4.3). This rapid variance precludes the static thresholds used

by prior systems [40], and also prohibits servers from making threshold decisions. Instead, threshold selection must be *adaptive* and be done by *cameras, online*.

To overcome these challenges, we use lightweight machine learning techniques to predict, at a fine granularity on the camera (e.g., every few frames), which threshold to use for the selected feature. To do this, we train a cluster-based model *for each query and server-specified feature*, based on the observation that there is a strong correlation between the thresholds of the feature and the query accuracy (see §2.4.3). Clustering is done over all pairs of observed difference values (in the training set) and their highest feature values that hit the accuracy target. For each observed difference value, the camera selects the cluster in which the value falls and performs filtering using the average filtering threshold of that cluster. Note that such models are cheap regression models that can run in real time even under the camera's tight resource constraints.

(C3) What if the model is incomplete? The model used to predict thresholds for the selected feature may *lack sufficient coverage*, particularly when video characteristics drastically change (e.g., rush hour starts). Unfortunately, how and where to detect such scenarios is challenging because detection relies on analyzing the accuracy of recent frames; for example, a change may have occurred if we see a significant accuracy drop for recent frames. However, the question is how to see the accuracy drop – the camera is unaware of the true accuracy as it does *not* run DNN object detectors, while the server only receives a subset of frames that the camera deems as relevant.

To address this issue, the Reducto camera constantly checks if the feature value for the current frame falls into an existing cluster in the model. If not, this indicates a potentially significant (and previously unseen) change in video characteristics, so the camera halts filtering and notifies the server to retrain the model. Note that our linear model is not only efficient to run but also efficient to *train*, enabling the server to train a new model *online* upon a request from the camera. Once trained, the new model is streamed back to the camera, which uses it until a subsequent update is required.

Source code and experimental data for Reducto are available at https://github.com/reducto-sigcomm-2020/reducto.

2.2 Motivation

This section explores two questions: (1) what compute/memory resources do commodity and state-of-the-art smart cameras possess (§2.2.1)?, and (2) how well do existing filtering techniques perform in such settings (§2.2.2)?

2.2.1 Smart Camera Resource Overview

To better understand the available resources for filtering on cameras, we analyzed publicly available information about multiple surveillance deployments, and conducted a small-scale study of local city and campus-wide camera installations. We found that there is a large resource divide between state-of-the-art cameras and commodity cameras which are widely deployed. Given that large-scale camera deployments are financially expensive to install and maintain, we do not anticipate an immediate overhaul that replaces commodity cameras with state-of-the-art ones. Instead, we expect a more gradual shift, and thus believe that camera-based filtering must consider the resource availability on both classes of devices. Note that we only focused on smart cameras, or those with some non-zero amount of general purpose compute resources; cameras without such resources are unable to handle any on-device filtering.

State-of-the-art smart cameras. Recent smart cameras commonly include AI hardware accelerators built into their processors, which speed up tasks such as DNN execution and video encoding [8, 29, 95, 135]. For example, Ambarella [29]'s CV22 System on Chip includes a quad-core processor (1 GHz) along with the CVflow vector processor designed explicitly for vision-based CNN/DNN tasks (e.g., object tracking on 4k videos at 60 fps). Some cameras also ship with small on-board GPUs as an alternate way to accelerate similar workloads [16, 28]. For instance, DNNCam [16] ships with an NVIDIA TX2 GPU and 32 GB of flash storage and has a unit price of $2,418. These resources support real-time object recognition

RAM	Tiny YOLO: Object Detection			NoScope: Binary Classification		
	0.5 GHz	1.0 GHz	1.5 GHz	0.5 GHz	1.0 GHz	1.5 GHz
128 MB	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
256 MB	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
512 MB	0.19	0.39	0.64	28.39	56.25	85.9
1024 MB	0.20	0.42	0.66	26.9	58.36	84.1

Table 2.1: Inference speed (in fps) of compressed object detection and binary classification models in resource-constrained (camera-like) environments. *NA* means the model lacked sufficient resources to run. Pixel-based frame differencing (omitted for space) always ran at over 300 fps.

(i.e., 30 fps or higher) and thus can be used to run NN-based filtering techniques directly on cameras.

Commodity and deployed cameras. In contrast to the promising filtering resources on state-of-the-art cameras, deployed surveillance cameras paint a much bleaker resource picture [31, 146, 149]. These cameras are considerably cheaper (generally $20–100), and ship with far more modest compute resources typically involving a single CPU core, CPU speeds of 1-1.4 GHz, and 64-256 MB of RAM. We verified the widespread deployment of such low-resource cameras by speaking with security teams for UCLA and Los Angeles—none of their deployed cameras included AI hardware accelerators, GPUs, or colocated edge servers, but they all possessed cheap CPU resources.

2.2.2 Limitations of Existing Filtering Techniques

We now explore how existing filtering techniques would fare on deployed smart cameras in terms of speed and filtering benefits. From §2.1, there are three main classes of existing filtering techniques:

- The first approach runs a *compressed object detection* model (e.g., Focus [63]) to obtain approximate query results. This approach determines whether or not to send each frame for full model execution (rather than just sending the computed result) based on the confidence in the result that the compressed model produces.
- The second approach runs a cheaper and less general (e.g., trained for a specific query and video content) *binary classification model*, which detects whether an object of interest (for

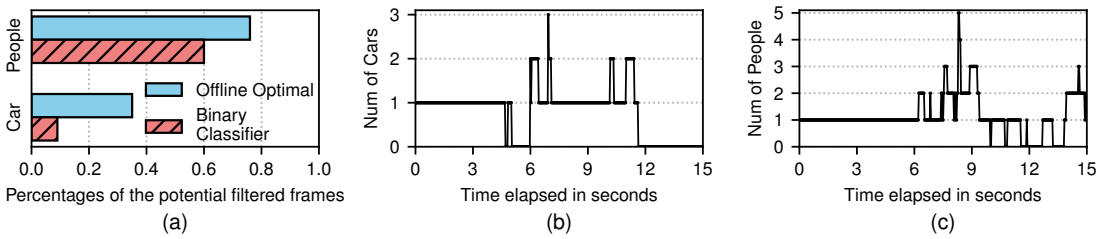


Figure 2.1: Binary classification yields limited filtering benefits: (a) the potential fraction of filtered frames, for standard people and car counting queries, as compared to an offline optimal (which filters based on query results), (b/c) representative video clips highlighting missed filtering opportunities with binary classification (i.e. non-zero but stable object counts).

the current query) is present in a frame or not. Only frames with the object of interest are sent to the server for processing (e.g., FilterForward [38], NoScope [76]).

- The third approach is to compute pixel-level *frame differences* and filter out frames which, according to a static/pre-defined differencing threshold, are largely unchanged from their predecessor and expected to yield the same query result (e.g., Glimpse [40]).

Speed. We started by evaluating the feasibility of running these three techniques *on cameras* for real-time filtering. We considered the canonical query of counting the number of cars in each frame. To evaluate frame differencing, we directly ran Glimpse's trigger frame selection algorithm using an arbitrary static threshold (more on this below) [40]. For compressed object detection, we used Darknet [115] to train a Tiny YOLO model [116] (8 convolutional layers) that only recognizes cars based on data labeled with YOLOv3. For binary classification, we trained a model that mimics the lightest classification model developed in NoScope [76]; this model has 2 convolutional layers (32 filters each) and a softmax hidden layer.[1] In both cases, training was done for each camera in our video dataset (§2.5.1) using 9 10-minute video clips from that camera.

We ran each technique on a new 10 minute clip from each camera under a sweep of resource configurations: a single core, 0.5-1.5 GHz CPU speed, and 128-1024 MB of RAM. Exper-

[1]We consider NoScope rather than FilterForward here because FilterForward's reliance on a DNN for feature extraction precludes its use on a camera; we empirically compare Reducto's filtering with that of FilterForward in §2.5.

iments were performed on a Macbook Pro laptop with a virtual machine that restricted resources to the specified parameters. Table 2.1 lists the filtering speeds in each setting. As shown, both NN models require at least 512 MB of RAM to operate, which precludes them from being used on many deployed cameras. Tiny YOLO is unable to achieve even 1 fps in any setting; note that even with the 11× speedup reported when also using background subtraction [63], Tiny YOLO is still far below real-time speeds. In contrast, when it has sufficient memory to run, the binary classification model consistently achieves real-time speeds, e.g., 28 fps and 86 fps with 0.5 GHz and 1.5 GHz processors, respectively. Further, pixel-based frame differencing is able to hit real-time speeds across all camera settings. Thus, the rest of the section focuses on frame differencing and binary classification (which is at least tenable in some camera settings).

Filtering efficacy. Now that we have identified potential filtering candidates for our resource-constrained environment, we ask, how effective are they at filtering out frames? We discuss the two candidates, binary classification and frame differencing, in turn.

To evaluate the *potential* filtering benefits with binary classification, we analyzed object detection results (captured by YOLO [116]) for all videos in our dataset and computed the fraction of frames that do not contain any object of interest. Note that this represents an *upper bound* on the benefits that systems such as NoScope [76] and FilterForward [38] can achieve. As a reference, we also considered an *offline optimal* strategy, where each frame is filtered if its query result is identical to that of its predecessor. As shown in Figure 2.1, binary classification is very limited in its filtering abilities: compared to the offline optimal, binary classification filters out **73.5%** and **16.7%** fewer frames for the car and person queries, respectively. The reason is that there exist many scenarios where query results remain unchanged across consecutive frames but have non-zero objects of interest. For example, a car count will be consistently greater than 1 if the camera is facing parked cars — although one frame would be sufficient to accurately count the number of cars, binary classifiers would send all such frames since they all contain objects of interest. Figures 2.1(b) and 2.1(c) illustrate this property for several representative video clips.

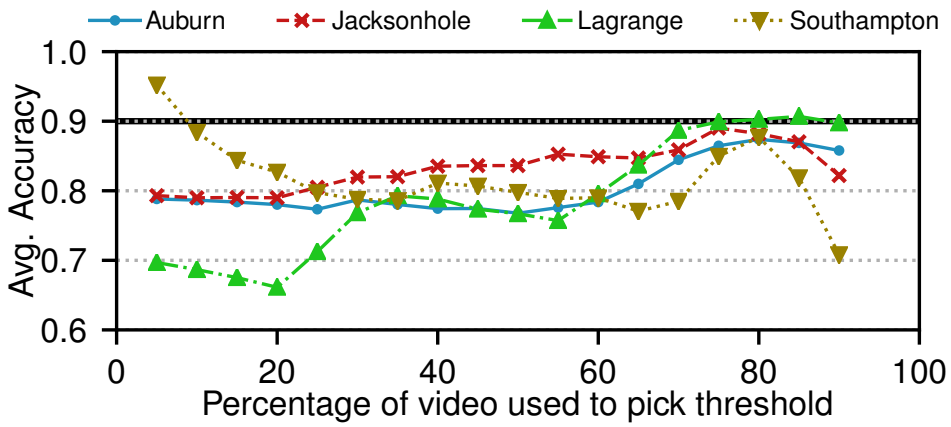


Figure 2.2: Glimpse [40] is unable to meet query accuracy requirements due to its use of a static threshold. The x-axis lists the fraction of each video used to select the best static threshold (i.e., max filtering while meeting the accuracy goal of 90%); the remainder of each video is used for evaluating the threshold.

Quantifying the potential filtering benefits with frame differencing techniques is challenging as they vary based on the tunable filtering threshold. Instead, the key limitation with respect to filtering efficacy is that existing frame differencing systems employ static and pre-defined filtering thresholds, which complicate accuracy preservation. To illustrate the limitations of static thresholds, we evaluated Glimpse [40] on 4 random videos in our dataset. For each video, to pick the static threshold to use, we varied the amount of video (from the start) to use for selecting the *best possible static threshold*, i.e., the threshold that filtered the most frames while achieving the target accuracy. We then evaluated the query accuracy on the rest of the video that was not used for threshold selection. As shown in Figure 2.2, even with the best possible static threshold, Glimpse is almost never able to meet the target accuracy. Note that this is true even when we used 90% of each video for threshold selection, and despite the fact that this evaluation was done on adjacent video from the same camera. The reason, which we will elaborate on in §2.4.2, is that the best filtering threshold depends heavily on video content, which can be highly dynamic.

Key takeaway. These results collectively paint a challenging picture for on-camera filtering. Due to resource restrictions, to use existing techniques in real time, cameras must resort to either binary classification models or frame differencing. However, binary classification is

Feature	On-camera tracking speed	Server tracking speed
SURF	1.27	26.55
SIFT	1.83	10.71
HOG	2.86	5.90
Corner	27.93	144.86
Edge	65.72	799.14
Area	71.80	1105.11
Pixel	308.60	2714.26

Table 2.2: Tracking speed (fps) for our candidate raw video features for frame differencing. High- and low-level features are shown on the top and bottom, respectively. Camera resources were 1 core, 1.0 GHz, and 512 MB RAM, while servers had 4 cores, 4 GHz, and 32 GB of RAM.

largely suboptimal as it hides many filtering opportunities (i.e., where objects are present but query results do not change across frames). Existing frame differencing strategies, on the other hand, use static thresholds and are unable to reliably meet accuracy targets.

To make effective use of frame differencing, the key question is whether it is possible to correlate frame differences with *pipeline accuracy* so that we can make a more informed decision as to whether a frame can be filtered out. We answer this question affirmatively in the next sections, where we describe how *lightweight differencing features* across video frames can serve as cheap monitoring signals that are highly correlated with changes in query results. If applied judiciously (and dynamically), these strong correlations enable large filtering benefits that are even comparable to those with the ideal baseline described earlier in this section.

2.3 Filtering using Cheap Vision Features

Given the limitations of existing filtering strategies for on-camera filtering (§2.2.2), we seek a clean-slate approach to filtering based on frame differencing. In this section, we focus on identifying candidate features, and in §2.4, we present Reducto, which determines when and how to use those features for effective on-camera filtering.

Our goal is to identify *a set of raw video features* (1) that are cheap enough to be tracked on cameras in real-time, and (2) whose values are highly correlated with changes in query

results for broad ranges of queries and videos (unlike prior systems that purely focus on detection [40]). We began with a representative list of differencing features used by the CV community [36], and grouped them in terms of the amount of computation required for extraction. *Low-level features* such as pixel or edge differences can be observed directly from raw images, but contain moderate amounts of noise. The main concern of using these features is whether or not this noise outweighs the true differencing values in certain cases. In contrast, *high-level features*, such as *scale-invariant feature transform* (SIFT) and *speeded up robust features* (SURF), aim to extract highly distinctive qualities of an image that are invariant to light, pose, etc., by analyzing properties such as local pixel intensities and shapes; these features have more semantic information, and many applications use such information to relate specific contents across frames. These features require multiple steps of computation on raw video values for extraction, but contain less noise as the noise is often smoothed out by the computation. The main concern of using high-level features is, clearly, their high extraction overheads.

Table 2.2 shows a representative list [64] of (high- and low-level) features we considered, and summarizes their computation overheads (in terms of fps) in both on-camera and server settings. Their detailed descriptions can be found in Table 5.1 (§5.3) with more information in these survey papers [34, 64, 164]. Further, we note that other features may meet the aforementioned goals and can be easily plugged into Reducto.

Tracking speed. As shown in Table 2.2, tracking high-level features is far too slow to operate in real-time on cameras; many of these features cannot be extracted fast enough even on servers! For instance, SURF and SIFT are restricted to frame rates under 2 fps. In contrast, low-level features can be extracted on cameras at 28-309 fps. Overall, these results eliminate high-level features from consideration for on-camera filtering and direct us to focus on identifying the appropriate low-level features that can satisfy our correlation requirements. We also exclude the low-level `Corner` feature that falls just short of our real-time (30 fps) tracking goal.

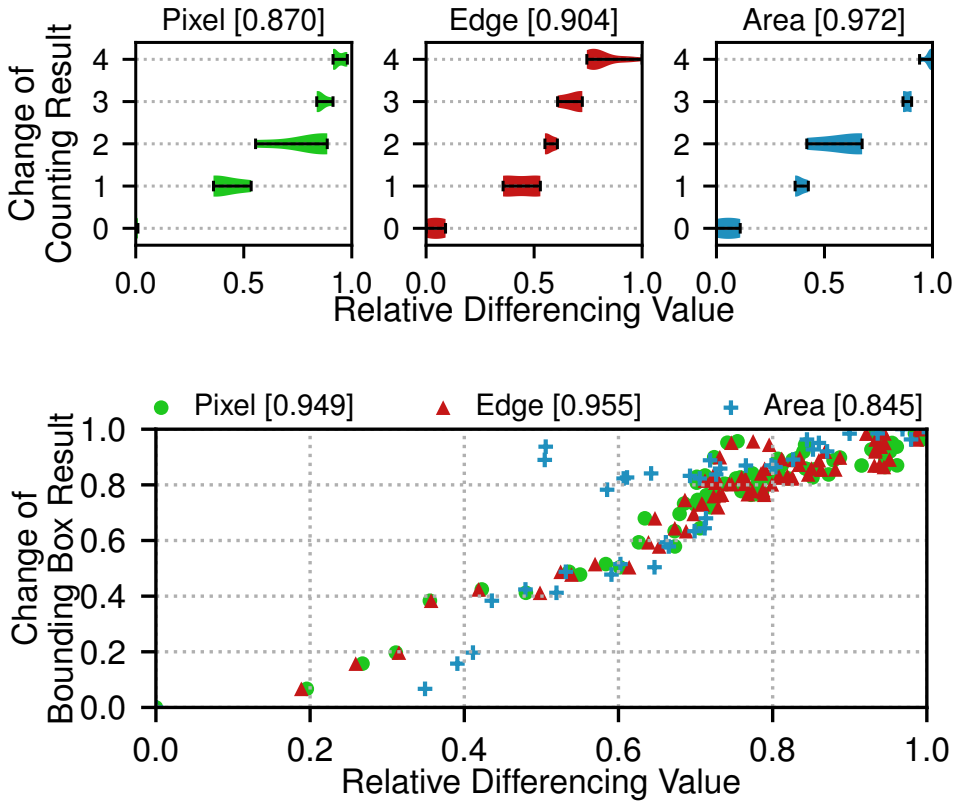


Figure 2.3: Correlations between differencing values and changes in query results for a 10 seconds clip in Auburn [4]. *Top* shows a car counting query where each line includes tick marks for min, max, and average feature value, with ribbons summarizing the distribution; *bottom* shows a car bounding box detection query. Results are for a random video. The legend lists the Pearson correlation coefficient per feature.

Correlation with changes in query results. Recall that our goal is to use differencing features to "predict" whether a change in query results may occur. Thus, the features we use need not capture the precise *change in magnitude* between query results for two frames, but instead must have strong correlation with *whether* a change occurs. Figure 2.3 summarizes the correlation between the values for each feature that can operate in real-time and changes in query results. The two figures highlight the fact that the three low-level features `Pixel` (i.e., directly compares pixels), `Edge` (i.e., captures differences in contours of objects), and `Area` (i.e., captures differences in areas) are indeed highly correlated (to varying degree–see §2.4) with changes in query results despite being potentially noisy on short time intervals. For example, on counting queries, a change of just 1 in object count leads to average changes of 0.42, 0.38, and 0.44 for the differencing values *w.r.t.* the `Pixel`, `Area`, and `Edge` features,

respectively. As a reference, changes in these feature values are only 0.01, 0.11, and 0.09 when the count results are unchanged. For the bounding box query, even though the precise bounding box coordinates for an object change progressively across frames, the correlation remains strong, with easily visible differences in feature values for even minor adjustments in bounding box coordinates. Note that these trends hold for the other videos in our dataset as well (Figures 5.1-5.3 in §5.1).

2.4 Reducto Design and Implementation

2.4.1 Overview

Figure 2.4 depicts the high-level query execution workflow with Reducto. Currently, Reducto supports the three primary classes of queries used in prior video analytics systems [75, 94]: tagging, counting, and bounding box detection. Descriptions of these queries are presented in §2.5.1.

***Offline* server profiling ❶ (§2.4.2).** The Reducto server first runs an *offline profiler* ❶ over several minutes of video that characterize the typical scenes for that camera. The profiler ❶ then runs traditional pipelines ❸ on that video and stores the results for subsequent feature selection. As this characterization data is collected, the profiler processes each frame in the video to extract the three low-level differencing features presented in §2.3. Our observation (Figure 2.5) is that there often exists a single feature that works the best for a query class across different videos, cameras, and accuracy targets. Hence, during profiling, the server finds the best feature for each query class that it wishes to support. At the end of this phase, the best feature for each query class is identified and stored at the server.[2]

***Per-frame* diff extraction ❹ (§2.4.4).** The camera does not stream any frames until it receives a query. Upon the arrival of a user-specified query and target accuracy, the server informs the camera of the best feature for that query. To filter, the diff extractor ❹ continuously tracks the differences in the specified feature between consecutive frames.

[2]Best features for common query types (e.g., detection) can be pre-programmed or shared across servers, thereby avoiding profiling.

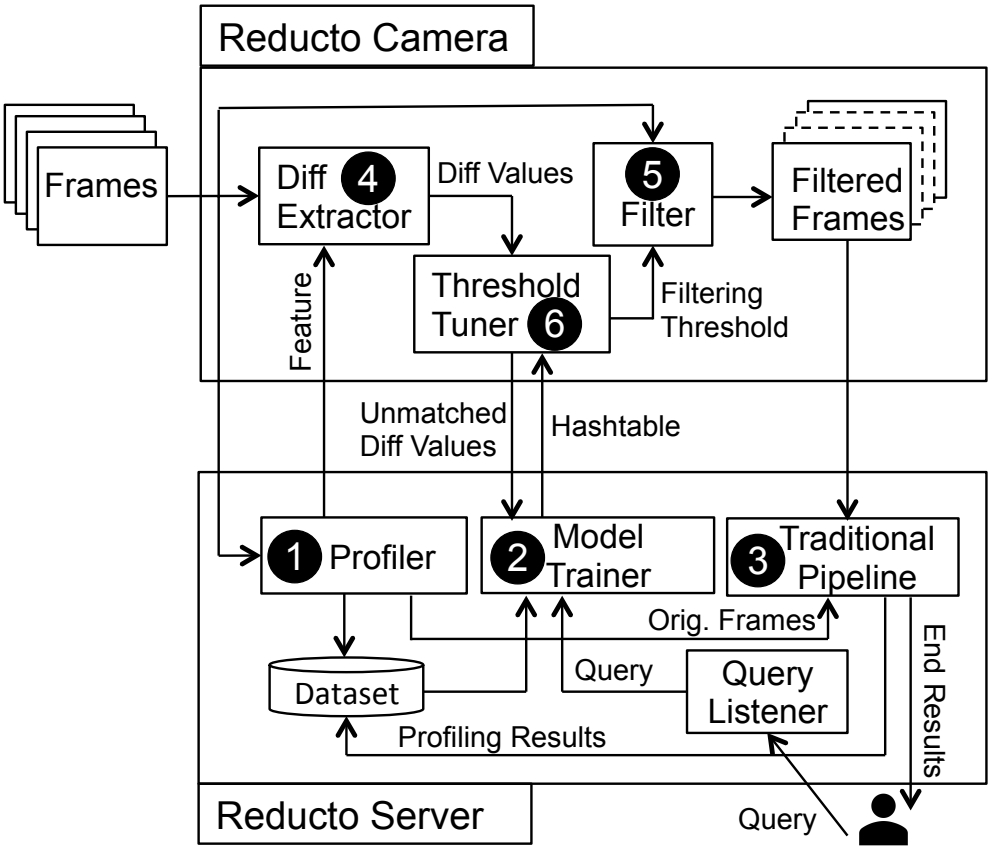


Figure 2.4: Overview of Reducto.

The key question at this point is how to know, for each pair of consecutive frames, if the difference between them is *sufficiently insignificant* so that if the camera sends only the first one to the server (which reuses its query result for the second), the accuracy would not drop below the target. In other words, what is the right filtering threshold to use?

***Per-query* model training ❷ (§2.4.3).** To answer this question, the server uses a model trainer ❷ that quickly trains, for each query, a simple (regression) model characterizing the relationships between differencing values, filtering thresholds, and query result accuracy. The model is trained by performing K-means-based clustering over the original frames sent by the camera during a short period after the query arrives. Training typically takes several seconds to finish due to the simple models used. The generated model is encoded as a hash table, where each entry represents a cluster of differencing values whose corresponding thresholds are within the same neighborhood — each key is the average differencing value and each

value is the threshold for that cluster which delivers the required accuracy. Together with the selected feature, this hash table is also sent to the camera for each query.

Per-frame **threshold tuning ❻ and filtering ❺ (§2.4.4).** When the camera receives the feature and the hash table for the query, it starts filtering frames. To do so, the filter ❺ queries the threshold tuner ❻ for the threshold to use. The tuner looks up the hash table using the differencing value produced by the diff extractor ❹, finds the matching key-value entry, and applies the listed threshold (i.e., the value of the entry).

Occasional **model retraining ❷ (§2.4.5).** In some cases, the differencing value may not map to any table entry (e.g., the distances between the value and the existing keys are too large). This indicates a potential change in video dynamics and implies that the new scene cannot be effectively captured by the existing clusters. As an example, the burst of cars at the start of rush hour can lead to a differencing value significantly different from those seen during training. In these cases, the threshold tuner ❻ sends these unmatched values (together with their original frames) to the model trainer ❷, which adds these new data points into its dataset (along with the generated query results), re-trains the model, and sends the tuner ❻ an updated hash table to ensure that the model stays applicable despite changes in the video.

The user can decide whether the camera deletes the frames that are not sent to the server. If the user wishes to save the frames for later retrieval or retrospective queries [63, 151], the camera archives all frames onto cheap local storage.

Tracking granularity. Since Reducto's goal is to ensure that the specified accuracy is continuously met, Reducto analyzes differencing features at the granularity of *video segments* rather than individual frames. Video segments represent small windows (e.g., N seconds) of consecutive frames. Analyzing features over segments enables Reducto to smooth out intermittent noise in feature values (§2.3). Thus, the Reducto camera buffers frames for each segment and selects the filtering threshold for the feature (using the hash table) when all

frames of the segment arrive. The camera then applies the filter with the selected threshold to each buffered frame to decide whether it needs to be sent.

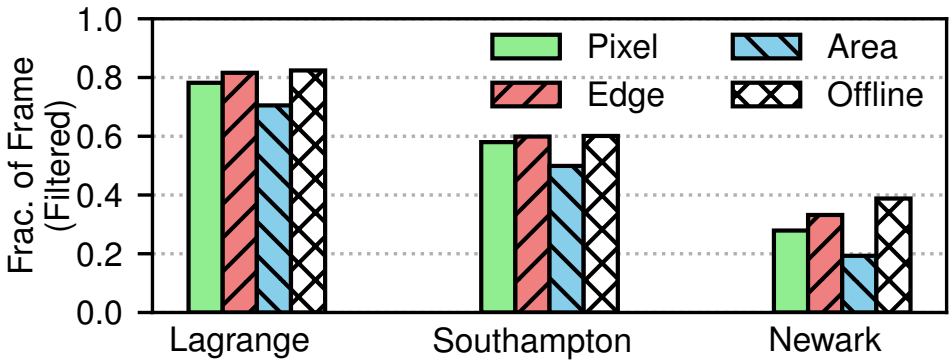
Selecting the right segment size is important: a small segment size is susceptible to inaccuracy due to noisy feature values, while a large segment size better handles noise but requires more frames to be buffered prior to making filtering decisions (delaying query results). We empirically observe that $N = 1$ *second* sufficiently balances these properties, and we present results analyzing how sensitive Reducto's results are to segment size in §2.5.

Discussion. We note that the presented design for Reducto (and our current implementation) focuses on single queries for a given camera's video feed. However, the described filtering approach can be extended to handling multiple queries in a straightforward manner: filtering decisions can initially be made independently per query (as described), and then aggregated by taking the union of frames deemed important for *any* query. Additionally, we note that Reducto currently targets detection-based queries that do not carry over information across frames. For instance, in its current form, Reducto does not support activity detection queries. We leave support for these more complex queries to future work.

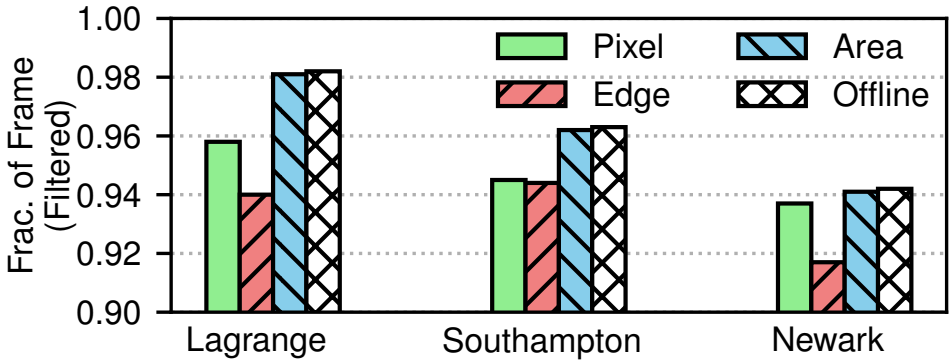
2.4.2 Feature Selection via Server-side (Offline) Profiling

During the offline profiling phase, the profiler ❶ uses several minutes of representative video frames to compute, for each frame, (1) the object detection results using traditional pipelines ❸, and (2) the low-level differencing values for each candidate feature. Using these results, the server determines which feature the camera should use.

The best feature to use is the one that maximizes the filtering benefits (i.e., filters out the most frames) while meeting the accuracy requirement specified by the user. In order to identify the best feature, the server analyzes the profiling results on a per-segment basis. For each segment and for each feature, the server considers a large range of possible thresholds for the feature. For each candidate feature, the server then aggregates the largest filtering benefits (obtained from using the best performed threshold on each segment) across all



(a) Query: Car bounding box detection



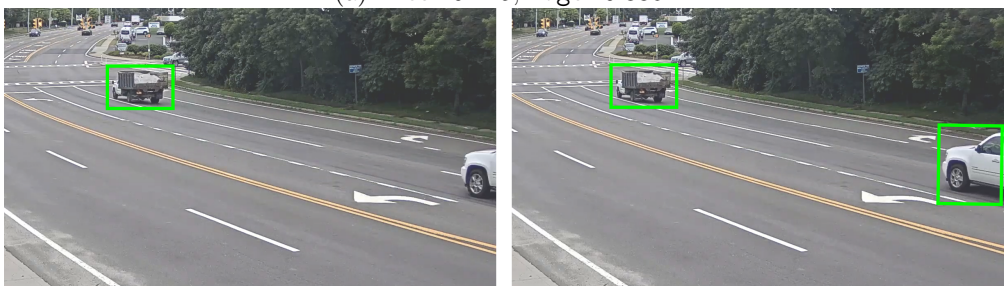
(b) Query: Car Counting

Figure 2.5: Filtering efficacy of the 3 low-level features across 3 videos and 2 queries. Y-axis reports the percentages of frames filtered (the higher the better). Across these videos, `Area` is best for counting, but `Edge` is best for bounding box detection. Results used YOLO and a target accuracy of 90%.

segments. These aggregated benefits are used to pick the best feature for each query class supported by Reducto.

Observation 1: Interestingly, we observe that the best feature tends to vary across query classes, but remains stable across cameras, videos, and target accuracies for each class. For example, consider Figure 2.5, which shows that the Area feature provides the largest filtering benefits for counting queries across 3 representative videos. In contrast, the Edge feature provides the most filtering benefits for bounding box queries. For reference, `Area` outperforms `Pixel` and `Edge` on counting queries by **11–70%**; and `Area` trails the two other features by **5–41%** on bounding box queries.

(a) `Area`: 0.145, `Edge`: 0.886



(b) `Area`: 0.830, `Edge`: 0.908

Figure 2.6: Car detection results for two sets of adjacent frames from the Southampton video; subcaptions list the corresponding differencing feature values. For bounding box detection queries, slight variations can change the query result; `Edge` picks up on these subtle changes (top) but `Area` does not. In contrast, counting queries are better served by `Area`, which reports significant differences when counts change (bottom), but not when counts stay fixed (top).

The reason is that different features and queries operate at different granularities, and their values change at varied levels with respect to changes across frames. In other words, minor frame differences may affect certain queries and feature values more than others. For example, consider the definitions of the `Area` and `Edge` features (Table 5.1 in §5.3). `Area` compares the size of the areas of motion across frames, but does not consider the distance that those areas move. In contrast, `Edge` is finer-grained and observes changes in the locations of the edges of objects.

Figure 2.6 illustrates how these divergences affect the suitability of each feature with respect to filtering for two query types: bounding box detection and counting queries. As shown in Figure 2.6(a), *any* motion for an object of interest can alter the corresponding bounding box coordinates. Whereas the `Area` feature is largely insensitive to such minor changes (making it ill-suited for filtering, as it suggests that the query result should not change), the `Edge` feature will detect (even minor) movements to the object's edges and yield a high differencing value. In contrast, Figure 2.6(b) shows that the coarse-grained nature of the `Area` feature is

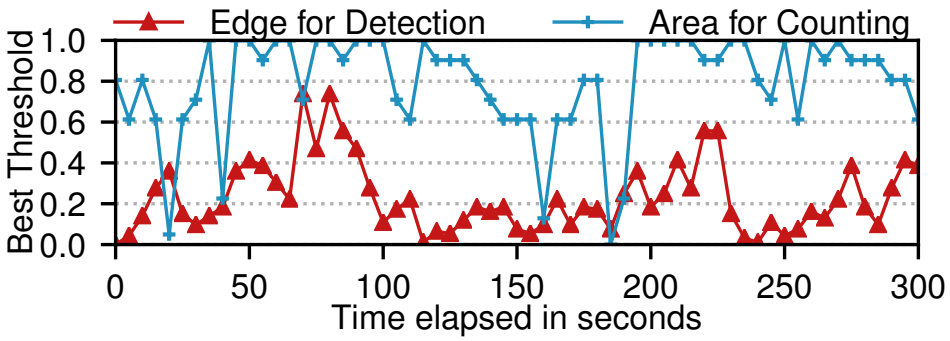


Figure 2.7: Best filtering thresholds vary across (even adjacent) video contents. This experiment used the Southampton video, and two features over two queries (`Area` over counting and `Edge` over bounding box detection, both for cars); the target accuracy is 90%. Trends hold for other queries, videos, and accuracy targets (§5.2).

well-suited for counting queries: when a new object enters a scene, it represents a new area of motion and results in a high differencing value. Thereafter, until the object count changes, the `Area` value remains low. The `Edge` feature, on the other hand, reports significant frame differences even when the overall object count stays unchanged (e.g., Figure 2.6(a)), making it too conservative for filtering for counting queries.

We verified that this stability in best feature holds across other query classes (e.g., tagging), objects of interest (e.g., people), target accuracies (e.g., 80%), and detection models (e.g., Faster R-CNN) as well; results are shown in §5.2 (Figures 5.4- 5.5) due to space restrictions. This observation implies that the server need not select features dynamically, and instead can make one-time feature decisions for all the query classes it wishes to support.

2.4.3 Model Training for Threshold Tuning

Knowing which feature to use is not enough; the camera also needs to know how to tune the filtering threshold for the feature so that filtering does *not* create unacceptable degradation in query accuracy.

Observation 2: While for each query class the best feature remains stable over time, the best threshold (i.e., highest one which meets the accuracy target) to use for a given feature does not. Figure 2.7 illustrates this point for two different query classes and their corresponding best features. As shown, the best threshold for each feature varies rapidly,

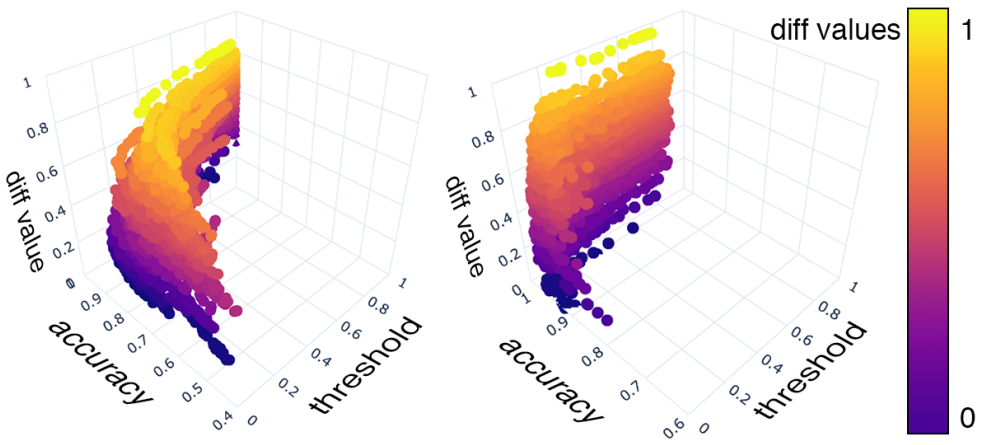


Figure 2.8: Simplified clustering results for two car queries: detection (left) and counting (right) over the Jackson Hole video.

on the order of segments. Thus, the camera needs a way to dynamically tune the threshold of the feature to prevent any unacceptable accuracy drops. However, making this decision requires understanding how different thresholds relate to the accuracy of query results. If the server can establish a mapping between differencing values, thresholds, and result accuracy, the camera can use such information to quickly find the best thresholds to use.

To generate this mapping, the server requires the camera to send unfiltered frames over a short window right after the query is registered. These frames are used as an initial training set — the server runs the full pipeline on them, producing *complete results* about each segment of frames — including query accuracy, fraction of frames filtered, and extracted feature values — for a broad range of candidate thresholds. For each segment, we compose a 29-dimension vector for the segment. This vector contains the average differencing feature value across the pairs of adjacent frames in the segment, i.e., a 1-second segment contains 30 frames (30 fps), resulting in 29 differencing values. We then add a data point to our training set for each candidate threshold; each data point is keyed at the corresponding 29-dimension differencing vector, and labeled with the tested threshold and the resulting query accuracy. Lastly, we remove any data points whose accuracy falls below the target accuracy for the query. The server then clusters these data points using the standard K-means algorithm based on their differencing vectors. Selecting the number of clusters entails balancing the overhead of the clustering algorithm and robustness of the resulting clusters

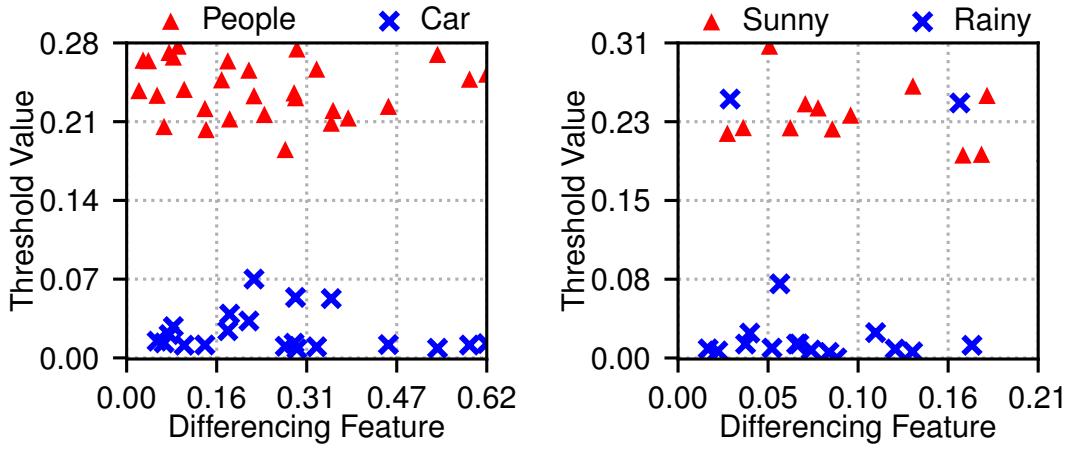
to noisy inputs; we empirically observe that setting a target of 5 clusters strikes the best balance between these factors, and we leave an exploration of more adaptive tuning strategies to future work [50, 57].

Figures 2.8 illustrates the clustering results for a random 10-minute clip. As shown, the data is highly amenable to such clustering, and the results follow a fairly intuitive pattern: to meet a given accuracy target, the filtering threshold *decreases* as the differencing feature value *increases*. This is because high feature values imply that frames are changing significantly (e.g., due to motion) — these changes give Reducto a reason to believe that the query result *may* change and thus the camera needs to send more frames.

Once clustering is done, the results are encoded into a hash table where each entry encodes information about a cluster — keys represent aggregated differencing values in the cluster augmented with the size measurement of the cluster (discussed shortly), while values represent the aggregated labels (i.e., thresholds). In particular, each key is of the form $\langle center, variance \rangle$, where *center* is a 29-dimension vector computed by performing element-wise averaging across the vectors in the cluster and *variance* is another 29-dimension vector where the i^{th} element represents the *longest distance* between the i^{th} elements in any possible pairs of data points in the cluster. In other words, *center* encodes the *central point* of the cluster while *variance* measures the *size* of the cluster (i.e., how far apart data points can be). Each value is the averaged filtering threshold of all data points in the corresponding cluster.

2.4.4 On-Camera Filtering

To filter out frames in real time, the camera continuously tracks differencing values for the selected feature. At the end of each segment, the camera decides which frames in the segment should be sent to the server. To do this, the camera simply looks up the hash table provided by the server. Specifically, the camera composes a similar 29-dimension vector a for the segment (by averaging the vectors for the constituent frames) and queries the hash table. The lookup algorithm finds the key-value pair $\langle \langle c, v \rangle, l \rangle$ such that (1) the euclidean distance between a and c is $\leq$ to that between a and any other key in the hash table, and (2) the



(a) Different object

(b) Different video contents

Figure 2.9: Offline training would be limited: comparisons of hash table entries (i.e., clusters) between (a) detection of different objects (i.e., people and car) and (b) different video contents (i.e., sunny and rainy) show that the clusters differ significantly under these circumstances; results were obtained from analyzing the entire Auburn video.

distance between the i^{th} elements in a and c is $\leq$ the i^{th} element in v, which represents the longest distance for the i^{th} dimension in the cluster. This indicates that the new data point falls well into the cluster (i.e., video contents changed in a similar way as in the past). Once such a table entry is found, the camera uses the threshold (i.e., the entry's value) to filter out frames in the segment. The remaining frames are compressed using H.264 at the original video's bitrate, and sent to the server.

2.4.5 Online Model Retraining

In scenarios where no matching key-value pair can be found (i.e., a does not belong to any cluster listed in the hash table), Reducto speculates that the current video properties are different from those used by the server to compute the table. In order to prevent degradations to below the accuracy target, the camera halts filtering and sends all frames in the segment to the server. The server computes query results over these original frames so no accuracy loss can occur. The server also adds these unfiltered frames to its dataset and re-clusters. The updated hash table is streamed back to the camera once it is computed, and upon reception, the camera resumes filtering.

Camera location	FPS	Resolution
Jackson Hole, WY [7]	15	1920 × 1080
Auburn, AL [4]	15	1920 × 1080
Banff, Canada [3]	15	1280 × 720
Southampton, NY [14]	30	1920 × 1080
Lagrange, KY [9]	30	1920 × 1080
Casa Grande, AZ [6]	30	640 × 360
Newark, NJ [10]	10	640 × 360

Table 2.3: Summary of our video dataset.

Online vs. offline training. In our implementation, model training (i.e., hash table generation) and retraining (i.e., hash table updates) are handled in the same way. Upon receiving a query, the server sends the selected feature and an *empty hash table* to the camera. The threshold tuner ❻ would not find any matching entry in the table and thus would have to stop filtering and send all frames for model training. Similarly, retraining is also triggered by misses in table lookups. A question the careful reader may ask is: is it necessary to perform model training/retraining online? In other words, does an *offline-learned* linear model suffice? To answer this question, we compared the hash table entries (i.e., clusters) generated under different queries and video contents. The results are illustrated in Figure 2.9. As shown, the clusters (and threshold values) differ significantly under these different circumstances, indicating that an offline training approach would be limited for unseen queries and video contents. Thus, even though Reducto's initial hash table can benefit from historical video data, in order to cope with the fact that it is impractical to foresee all possible queries and video properties, Reducto also supports online training/retraining.

2.5 Evaluation

2.5.1 Methodology

Table 2.3 summarizes the video dataset on which we evaluated Reducto. Our dataset comprises public video streams from 7 live surveillance video cameras deployed around North America. From each data source, we collected 25 10-minute video clips that cover a 24-hour period. As a result, video content for a given camera varied over time with respect to illumination, weather characteristics, and density of people and cars. Video content also varied across cameras *w.r.t.* quality, orientation (e.g., certain cameras were mounted on traffic

Figure 2.10: Screenshots from several of the videos in our dataset. Left is Jackson Hole, WY, and right is Newark, NJ.

lights, while others were recording streets from a side angle), and speed/density of objects (e.g., rural vs. metropolitan). Figure 2.10 provides some example screenshots.

In our evaluation, we considered three main classes of queries, each with a unique definition of accuracy:

- **Tagging queries** return a binary decision regarding whether or not an object of a given type appears in a frame. Accuracy is defined as the percentage of frames which are tagged with the correct binary value.
- **Counting queries** return a count for the number of objects of a given type that appear in a frame. Accuracy for a frame is defined as the absolute value of the percent difference between the correct and returned values.
- **Bounding box detection queries** return bounding box coordinates around each instance of a given object that is detected in a frame. Accuracy is measured using the standard mAP metric [47] that evaluates, for each returned bounding box, whether the enclosed object is of the correct type and whether the bounding box has sufficiently large overlap (intersection over union) with the correct box.

We ran each query class across our entire video dataset for two types of objects: people and cars. Unless otherwise noted, ground truth for all video frames and queries was computed using YOLO [116]. Reported accuracy numbers for each of Reducto's segments were

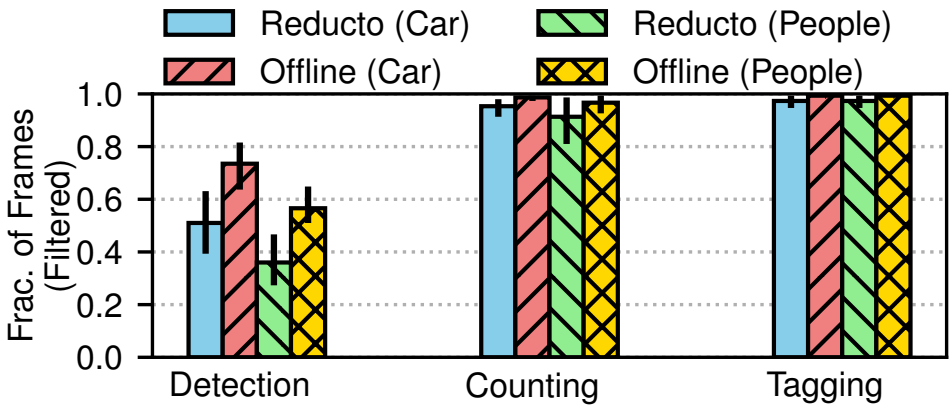
computed by averaging the accuracy values for each of the segments' constituent frames; Reducto used segments of 1 second unless further specified.

Server components ran on an Ubuntu Amazon EC2 p3.2xlarge instance with 8 CPU cores and 1 NVIDIA Tesla V100 GPU. The camera was either a Raspberry Pi or a VM whose resources were provisioned based on the RAM and CPU speeds observed in our study of deployed cameras (§2.2); the recorded video was fed into the camera sequentially and in real time. For brevity, in the VM scenario, we present results for the resource configuration of 256 MB of RAM and a 1 GHz CPU (single core). However, we note that the reported trends persist in the other settings in Table 2.1. The camera and server were connected over a variety of live (LTE and WiFi) and emulated networks via Mahimahi [100].

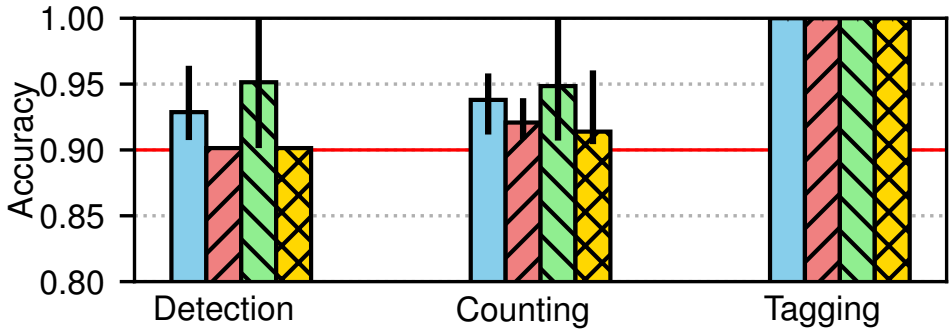
2.5.2 Overall Performance

To understand Reducto's filtering efficacy, we first compared it to a *baseline* video analytics pipeline in which cameras do not perform any filtering, and servers compute query results for all frames. To contextualize our results, we compared both systems with the *offline optimal* (§2.2) that uses actual query results to perfectly filter out each frame whose result sufficiently matches that of its predecessor. Note that the offline optimal represents an *upper bound* of what Reducto can hope to achieve without direct knowledge of query results.

Figure 2.11 shows that, across our entire video dataset, a target accuracy of 90%, and a variety of query types, Reducto is able to filter out a median of 51-97% of frames, which is within 2.8-36.7% of the offline optimal. As expected, filtering benefits vary based on query type, object of interest, and video. For instance, across the dataset, Reducto filtered out a median of 97% and 51% for tagging and detecting cars, respectively. This follows from the fact that the bounding box position for a moving car changes very *quickly* (e.g., across consecutive frames), while the presence of any car (i.e., what a tagging query searches for) often remains stable for long durations. Indeed, almost all frames could be filtered for tagging queries because query results changed very *infrequently*.



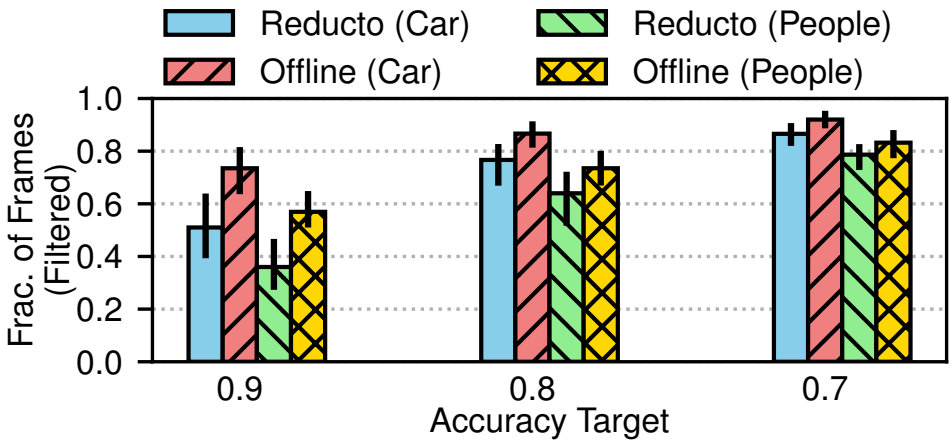
(a) Fraction of filtered frames



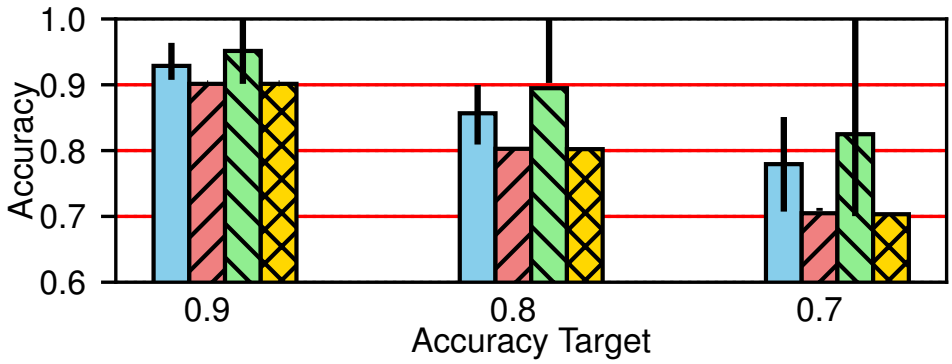
(b) Achieved accuracy

Figure 2.11: Comparing Reducto and the offline optimal filtering strategy for three query types and two objects of interest across our entire dataset. Results are for the distribution across all Reducto segments. Each bar reports the median with the error bar showing the 25th and 75th percentiles. The target query accuracy is 90%.

Despite this aggressive filtering, Figure 2.11 illustrates that Reducto is able to *always* deliver per-segment accuracy values above the target (90%). Reducto consistently delivers higher accuracy than the offline optimal, which nearly perfectly matches the target (due to knowing the ground truth). This is a result of Reducto's cautious selection of filtering thresholds (§2.4.4). In other words, whereas Reducto conservatively selects the filtering threshold to overshoot the accuracy target (filtering out fewer frames than possible), the offline optimal perfectly hovers over the target, thereby optimizing the fraction of frames that can be filtered within the accuracy constraint.



(a) Fraction of filtered frames



(b) Achieved accuracy

Figure 2.12: Analyzing Reducto's results for different accuracy targets. Results are for bounding box detection queries of cars and people across our entire video dataset.

Varying accuracy targets. We also evaluated how Reducto's filtering benefits vary with different accuracy targets. In this experiment, we primarily focused on bounding box detection queries which show the largest variation across accuracy targets due to their fine-grained nature. As expected, Reducto's filtering benefits increase as the accuracy target decreases (Figure 2.12). For instance, when the object of interest is people, filtering benefits rise from 36% to 79% as accuracy drops from 90% to 70%. The reason is that Reducto can be more aggressive with filtering and tolerate more substantial inter-frame differences in feature values, without violating a lower accuracy target. Importantly, Reducto always met the specified accuracy target.

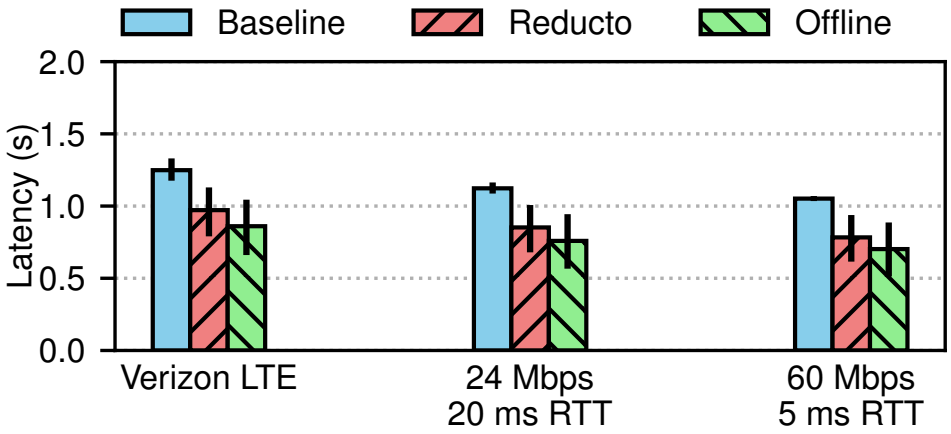


Figure 2.13: Distribution of per-frame query response times on different camera-server networks. Each bar reports the median, with error bars showing 25th and 75th percentiles. Results are for detecting cars on our entire video dataset, and the target accuracy is 90%.

System	Accuracy (%)	Fraction Filtered (%)	Bandwidth Saving (%)	Backend Processing (fps)
Baseline	100.00	0.00	0.00	41.13
Reducto	90.49	53.42	22.30	86.21
Optimal	90.16	72.80	39.33	140.04

Table 2.4: Breaking down the impact of Reducto's filtering on network and backend computation overheads. Results are for detecting cars and are averaged across our entire dataset. The target accuracy is 90%.

Query response times. The promise of frame filtering is ultimately to reduce resource overheads and deliver (highly accurate) query results with low latency. Figure 2.13 illustrates that, across several network conditions, Reducto is able to reduce median per-frame response times by 22-26% (0.26-0.28s) compared to the baseline pipeline; Reducto's response times are within 12–13% the offline optimal. Table 2.4 further breaks down these query response time speedups into network and backend improvements. As shown, Reducto's filtering results in an average bandwidth saving of 22% compared to the baseline pipeline; backend processing speeds, on the other hand, more than doubled due to the decrease in frames to be processed.

On-camera evaluation. Our experiments thus far have considered a resource-constrained VM as the camera component of the video analytics pipeline. In order to evaluate the feasibility of running Reducto directly on a camera, we replaced the aforementioned VM in our pipeline with a Raspberry Pi Zero [13] that embeds a 1.0 GHz single-core CPU and 512

MB of RAM; this resource profile falls into the range of on-camera resources that we observed in our study of commodity cameras and surveillance deployments (Table 2.1 and §2.2.1). We note that Raspberry Pi computing boards are intended to run alongside sensor devices (e.g., cameras) to provide minimal and affordable computation resources. We implemented Reducto on the Raspberry Pi using OpenCV [17] for feature extraction and frame differencing calculations, and a hash table lookup to make threshold selections and filtering decisions. Unfiltered frames were encoded using Raspberry Pi's hardware-accelerated video encoder for the H.264 standard. As we did with the VM, we fed in each recorded video in Table 2.3 sequentially and in real-time to the Raspberry Pi.

Overall, we observed that Reducto's filtering results for each video *identically* matched those from our VM-based implementation (i.e., results in Figure 2.12). More importantly, Reducto was able to operate at 47.8 fps on the Raspberry Pi, highlighting the ability to perform real-time filtering. Digging deeper, we found that the bulk of the processing overheads were due to per-frame feature extraction with OpenCV; this task could operate at 99.7 fps, as compared to frame differencing calculations and hash table lookups that ran at 129.5 and 318.6 fps, respectively.

Sensitivity to segment size. We varied the segment size that Reducto used for on-camera filtering between 0.5-10 seconds. Figure 2.14 illustrates three trends. First, as segment size decreases, Reducto's median filtering benefits are largely unchanged. We note that the distribution of filtering benefits widens largely because there are fewer opportunities to experience different video conditions within a small segment. For instance, a segment of 4 frames may be mostly unchanged and require only 1 frame to be sent; such a filtering fraction is less likely as segment sizes grow. Second, as segment size increases, bandwidth savings increase. This is because larger segments enable more aggressive bandwidth savings from standard video encodings: more frames can avoid redundant transmission due to fewer key frames. Third, per-frame query response times grow as segment sizes increase. Recall that Reducto cameras only filter out and ship frames to servers *after* a segment is captured.

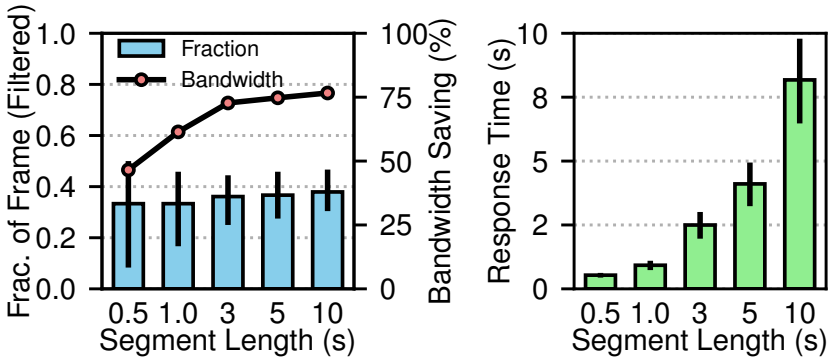


Figure 2.14: Impact of Reducto's on-camera segment length on filtering benefits for object detection of cars on two randomly selected videos in our dataset; the target accuracy was 90%. Results are distributions across segments, with bars representing medians and error bars spanning 25th to 75th percentile.

System	Accuracy (%)	Fraction Filtered (%)	Bandwidth Saving (%)	Backend Processing (fps)
Reducto	90.49	53.42	22.30	86.21
Tiny YOLO	90.22	24.46	13.68	53.66
FilterForward	90.10	27.70	14.49	56.32

Table 2.5: Comparing Reducto with existing real-time filtering systems. Results are for detecting cars in our entire dataset, and the target accuracy is 90%.

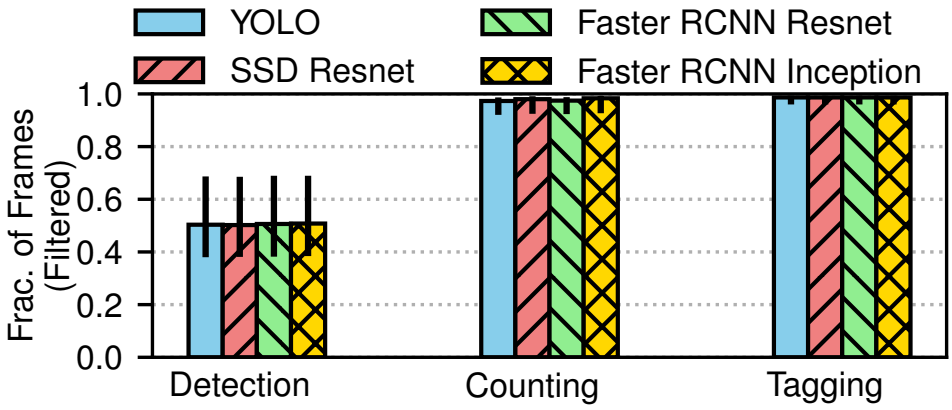
Thus, frames that are early in a given segment must experience query response times that are at least as long as the segment size.

Sensitivity to different object detection models: We verified (Figure 2.15) that Reducto's overall filtering benefits and accuracy preservation persist across other models, i.e., SSD ResNet, Faster R-CNN with Inception ResNet.

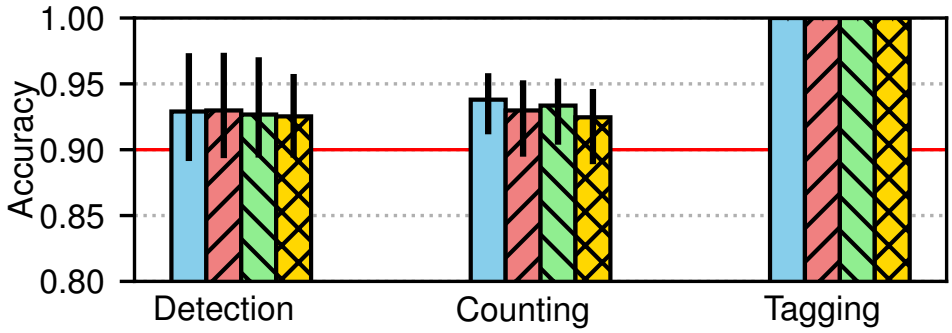
2.5.3 Comparison with Other Filtering Strategies

We also compared Reducto with two existing filtering approaches that are both able to consistently meet a desired accuracy target; recall from §2.2.2 that Glimpse [40] was unable to do so due to its static threshold approach.

Tiny YOLO. We considered a filtering system that computes approximate query results using a compressed detection model (Tiny YOLO). Frames whose result confidence is sufficiently high (80% in this experiment; tuned to the target accuracy) can benefit from (1)



(a) Fraction of frames filtered



(b) Achieved accuracy

Figure 2.15: Comparing Reducto with different detection models for three query types and two objects of interest across our entire dataset.

filtering, if the frame does not contain an object of interest, or (2) result reuse which avoids running the backend detector. This approach is *loosely* inspired by Focus's ingest-time processing [63] which targets retrospective queries; we omit Focus' clustering strategy, which is primarily useful for the tagging queries that Focus targets. We trained a Tiny YOLO model on 90 minutes of video from each feed in our dataset to detect cars; we then tested on separate 30-minute clips from the same feed.

FilterForward. We also ran FilterForward [38], a binary classification-based filtering system designed for edge servers. With FilterForward, micro-classifers ingest feature maps computed by different layers of a full-fledged object detector, and determine whether an object of interest is present or not in each frame; if not, the frame is filtered at the edge

System	Accuracy (%)	Fraction Filtered (%)	Bandwidth Saving (%)	Backend Processing (fps)
Reducto	90.49	53.42	22.30	86.21
Cloudseg 2x	85.78	0.00	56.82	32.33
Cloudseg 4x	60.86	0.00	82.46	31.13
Reducto	99.10	97.11	80.23	1360.71
Cloudseg 2x	99.67	0.00	56.82	32.19
Cloudseg 4x	99.55	0.00	82.46	31.57

Table 2.6: Comparing Reducto with CloudSeg [144]. Results are for detecting cars (top) and tagging cars (bottom), both with an accuracy target of 90%.

server. FilterForward reports comparable performance to NoScope [76], which is intended for retrospective queries. In our experiments, we directly ran FilterForward's open-source code and trained a micro-classifer in the same way as Tiny YOLO above.

Results. Table 2.5 shows that Reducto achieves significantly larger filtering benefits compared to both systems. Average frame savings with Reducto are 53.42%, while Tiny YOLO and FilterForward filter only 24.46% and 27.7%, respectively. This translates to improvements of 54-63% and 53-61% in network bandwidth expenditure and backend processing costs, respectively. Key to this performance discrepancy is the limitation in binary classification-based filtering (§2.2.2). We note that, unlike Reducto, neither FilterForward nor Tiny YOLO can run in real time on a camera; the filtering benefits described here, however, are unaffected by resource constraints.

2.5.4 Comparison with Complementary Video Analytics Systems

We also compared Reducto with two systems that improve the efficiency of real-time video analytics pipelines, CloudSeg [144] and Chameleon [74]. Each system aims to improve a different aspect of the analytics pipeline, and both approaches are conceptually complementary to Reducto.

CloudSeg. CloudSeg uses super resolution techniques to significantly compress live video prior to shipping it to servers for analytics tasks; super resolution models at the server are used to (mostly) recover the original high resolution image, which is then fed into the analytics pipeline. To implement CloudSeg, we used bilinear interpolation in OpenCV [17]

System	Accuracy (%)	Bandwidth Saving (%)	Backend Processing (fps)
Baseline	100.00	0.00	13.04
Reducto	90.08	32.16	103.40
Chameleon	92.00	0.00	93.75

Table 2.7: Comparing Reducto with Chameleon [74] on a car counting query. The target accuracy was 90%.

to compress all frames by 2-4× on our camera VM. We then used the same super resolution model as CloudSeg, CARN [87], to recover the original video on the server.

As shown in Table 2.6, we initially tried a compression factor of 4× for CloudSeg. Despite heavy tuning, we were unable to hit our accuracy target for detection. Thus, we focused our discussion on the 2× compression which narrowly misses the 90% accuracy goal. As expected, for detection, CloudSeg achieves superior bandwidth savings compared to Reducto (57% compared to 22%). However, CloudSeg does not filter out frames, and instead opts purely for compression, i.e., all frames must go through costly backend processing. As a result, Reducto's filtering results in 2.7× improvements in backend processing overheads. Results for tagging follow a similar pattern, but we note that Reducto achieves superior bandwidth savings because most frames can be filtered out; for the same reason, the discrepancy in backend processing overheads is more pronounced. These approaches are complementary in that Reducto can also apply super resolution encoding on cameras (in real time) after filtering.

Chameleon. Systems such as Chameleon [74] and VideoStorm [158] reduce backend computation costs by profiling different configurations of pipeline knobs (e.g., video resolution, frame sampling rate, etc.) and selecting those that are predicted to minimize resource utilization while meeting the user-specified accuracy requirement. Chameleon improves upon VideoStorm in that it profiles periodically rather than once, upfront. To implement Chameleon, we considered configurations based on the following knobs: 5 levels of image resolution (1080p, 960p, 720p, 600p, 480p), 2 pre-trained object detection models (Faster R-CNN and YOLOv3), and 5 levels of frame rate (30fps, 10fps, 5fps, 2fps, 1fps). For each video in our dataset, we selected the best configuration for each 4-second segment (which is

Chameleon's profiling rate); we used the same segment size for Reducto. For ease of implementation, profiling for each segment was done offline. For fair comparison, Reducto used the more expensive Faster R-CNN model, which Chameleon treats as ground truth.

As shown in Table 2.7, both systems significantly outperform the baseline pipeline, but Reducto achieves 37% better backend processing speeds. Further, by filtering directly at the video source, Reducto is also able to achieve network bandwidth improvements that Chameleon cannot. While both systems reap filtering benefits (e.g., decreased sample rates with Chameleon), they are largely complementary in that Chameleon considers knobs which Reducto does not, i.e., detection model, image resolution.

2.6 Related Work

Edge-cloud split. One class of edge-based approaches, exemplified by FilterForward [38], sends frames to the server based on the objects present, approximated by a light-weight neural network running at the edge. Wang et. al. [140] use MobileNet [62] on drones. Similarly, Vigil [159] uses an edge node that can run object detection and sends frames with a higher object count. Gammeter et al. [49] send a frame only when object tracking on the mobile device has consistently low confidence. Alternatively, a server could receive partial information from the edge and decide whether it needs more based on inference results [107]. While this model can save significant bandwidth, round trips between server and edge impedes the system's ability to respond to queries in real time. Chinchali et al. [41] also use a server-driven approach, but the edge device can adapt (for DNN input) both the information it sends and the encoding method based on feedback from the server. Finally, Emmons et al. [46] propose a DNN split inference, where the edge runs as many layers as possible before sending the intermediate values to the cloud. In contrast to all of these solutions, Reducto, is aimed at cameras with resources that do not even support small NNs.

Resource scheduling. VideoStorm [158] and Chameleon [74] profile pipeline knobs to identify cheap and accuracy-preserving configurations (§3.6), while VideoEdge [68] also considers placement plans over a hierarchy of clusters. DeepDecision [112] and MCDNN [59] treat re-

source scheduling as an optimization problem and maximize key metrics such as accuracy or latency, while LAVEA [152] allocates computation among multiple edge nodes, optimizing for latency. These systems are largely complementary to Reducto, as the resource-accuracy tradeoff could be further tuned on the set of Reducto-chosen frames. Another complementary class of systems focuses on efficient GPU task scheduling [124].

Querying video. NoScope [76], BlazeIt [75], and Focus [63] lower resource consumption for efficient retrospective video querying. In contrast, Reducto uses the relationship between video features and query result, rather than presence of objects, for an early determination of relevant frames.

Computer vision. The idea of filtering frames based on their features is widely seen in the CV community [33, 48, 80, 114, 143, 147, 148]. Many of these methods are used for the task of retrospectively classifying or recognizing events in videos [42, 105, 148]. AdaFrame, for example, trains a Long Short-Term Memory network to adaptively select frames with important information. Others are used for key frame extraction [114, 155]. The are two main barriers to directly using these methods to filter frames in a setting like Reducto. One is that the task of choosing which frames a DNN should process is different from choosing frames for video classification or key frame extraction, because the importance of a frame in Reducto is determined solely by whether the DNN output changes. Second, these methods rely on neural networks that are too expensive to run on a camera.

2.7 Reducto Summary

Recall that our key goal in this thesis is to improve the performance of video analytics pipelines by using inexpensive hardware at the edge. This chapter presents Reducto, a system that supports efficient real-time querying by leveraging the inexpensive, previously unused hardware on cameras to perform frame filtering. Given the limited resources available on cameras, Reducto uses cheap frame differencing techniques to determine when previously computed results can be used while still meeting accuracy targets. Our evaluation found that careful use of such techniques yields up to a 50% decrease in the number of frames that

need to be sent to the server and run through the DNN. This lowers the end-to-end latency by up to 26%.

Reducto's approach to optimizing the pipeline exploited the temporal redundancy in video content, or the *inputs* to the machine learning models. However, it treated the model itself as a black box; we never changed the inner workings of the model. Next, we look at a similar video analytics pipeline, but we instead delve into methods for optimizing the models themselves.

CHAPTER 3

GEMEL: Model Merging for Memory-Efficient, Real-Time Video Analytics at the Edge

3.1 Overview

While many video analytics pipelines use both an edge component and a cloud component to run inference (e.g., Reducto), other deployments incorporate *on-premise* edge servers (e.g., Microsoft Azure Stack Edge [2], Amazon Outposts [1]) that run in hyper-proximity to cameras and possess on-board GPUs to run DNNs at the edge itself. These *edge boxes* are typically used to replace [24] distant cloud servers by locally performing as many inference tasks on live video streams as possible [38, 68, 159]. Recall that generating responses directly on edge boxes reduces transfer delays for shipping data-dense video over wireless links [58, 90, 161] while also bringing resilience to outbound edge-network link failures [5, 103] and compliance with regional data privacy restrictions [99, 110].

To reap the above benefits, video analytics deployments must operate under the limited computation resources offered by edge boxes. On the one hand, due to cost, power, and space constraints, edge boxes typically possess weaker GPUs than their cloud counterparts [2, 24, 125]. On the other hand, analytics deployments face rapidly increasing workloads due to the following trends: (1) more camera feeds to analyze [24, 68, 70], (2) more models to run due to increased popularity and shifts to bring-your-own-model platforms [20, 27, 52, 69], and (3) increased model complexity, primarily through growing numbers of layers and parameters (Figure 1.1) [19, 71, 72, 142]. Taken together, the result is an ever-worsening resource picture for edge video analytics.

Problems. Although GPU computation resources are holistically constrained on edge boxes, this work focuses on *GPU memory restrictions*, which have become a primary bottleneck in edge video analytics for three main reasons. First, GPU memory is costly due to its high-bandwidth nature [108, 111, 122], and is thus unlikely to keep pace with the ever-growing memory needs of DNNs (Figure 1.1). Second, we empirically find that existing memory management techniques that time/space-share GPU resources [32, 53, 65, 71, 123, 150] are insufficient for edge video analytics, resulting in skipped processing on 19-84% of frames, and corresponding accuracy drops up to 43% (§3.3). The underlying reason is that the costs of loading vision DNNs into GPU memory (i.e., swapping) are prohibitive and often exceed the corresponding inference times, leading to sub-frame-rate ($<$ 30 fps) processing and dropped frames due to SLA violations [123, 156]. Such accuracy drops are unacceptable for important vision tasks, especially given that each generation of vision DNNs brings only 2-10% of accuracy boosts – that after painstaking tuning [25, 67, 79, 128]. Third, compared to computation bottlenecks [38, 53, 54, 74, 89], GPU memory restrictions during inference have been far less explored in video analytics.

Contributions. We tackle this memory challenge by making two main contributions described below. The design and evaluation of our solution are based on a wide range of popular vision DNNs, tasks, videos, and resource settings that reflect workloads observed in both our own multi-city pilot video analytics deployment and in prior studies (§3.2).

Our first contribution is *model merging*, a fundamentally new approach to tackling GPU memory bottlenecks in edge video analytics that is complementary to time/space-sharing strategies (§3.4). With merging, we aim to share *architecturally identical* layers across the models in a workload such that only one copy of each shared layer (i.e. one set of weights) must be loaded into GPU memory for all models that include it. In doing so, merging reduces both the number of swaps required to run a workload (by reducing the overall memory footprint) and the cost of each swap (by lowering the amount of new data to load into GPU memory).

Our merging approach is motivated by our (surprising) finding that vision DNNs share substantial numbers of layers that are architecturally (i.e. excluding weights) identical (§3.4.1). Such commonalities arise not only between identical models (100% sharing), but also across model variants in the same (up to 84.6%) and in different (up to 96.3%) families. The reason is that, despite their (potentially) different goals, vision DNNs ultimately employ traditional CV operations (e.g. convolutions) [25, 79], operate on unified input formats (e.g. raw frames), and perform object-centric tasks (e.g. detection, classification) that rely on common features such as edges, corners, and motion [35, 40, 80, 83, 114, 139, 162, 163].

Our analysis reveals that exploiting these architectural commonalities via merging has the potential to substantially lower memory usage (17.9-86.4%) and boost accuracy (by up to 50%) in practice. However, achieving those benefits is complicated by the fact that edge vision models typically use different weights for common layers due to training non-linearities [77, 78] and variance in target tasks, objects, and videos; and yet, merging requires using unified weights for each shared layer. Digging deeper, we observe that there exists an *inverse relationship* between the number of shared layers and achieved accuracy during retraining. Intuitively, this is because for shared layers to use unified weights, other layers must adjust their weights accordingly during retraining; the more layers shared, the harder it is for (the fewer) other layers to find weights to accommodate such constraints and successfully learn the target functions [26, 88]. Worse, determining the right layers to merge is further complicated by the fact that (1) it is difficult to predict precisely how many layers will be shareable before accuracy violations occur, and (2) each instance of retraining is costly.

Our second contribution is GEMEL, an end-to-end system that practically incorporates model merging into edge video analytics by automatically finding and exploiting merging opportunities across user-registered vision DNNs (§3.5). tackles the above challenges by leveraging two key observations: (1) vision DNNs routinely exhibit power-law distributions whereby a small percentage of layers, often towards the end of a model, account for most of the model's memory usage, and (2) merging decisions are agnostic to inter-layer dependencies, and accordingly, a layer's mergeability does not improve if other layers are also shared.

Building on these observations, GEMEL follows an *incremental* merging process whereby it attempts to share one additional layer during each iteration, and selects new layers in a memory-forward manner, i.e. prioritizing the (few) memory-heavy layers. In essence, this approach aims to reap most of the potential memory savings as quickly, and with as few shared layers, as possible. GEMEL further accelerates the merging process by taking an adaptive approach to retraining that detects and leverages signs of early successes and failures. At the end of each successful iteration, GEMEL ships the resulting merged models to the appropriate edge servers, and carefully alters the time/space-sharing scheduler – a merging-aware variant of Nexus [123] in our implementation – to maximize merging benefits, i.e. by organizing merged models to reduce the number of swaps, and the delay for each one. Importantly, GEMEL verifies that merging configurations meet accuracy targets *prior* to deployment at the edge, and also periodically tracks data drift.

We plan to open-source GEMEL and our datasets.

3.2 Methodology & Pilot Study

We begin by describing the workloads used in this paper. They were largely derived from our experience in deploying a pilot video analytics system in collaboration with two major US cities (one per coast), for road traffic monitoring.

Models and tasks. In line with other video analytics frameworks [20, 27, 52, 69], users in our deployment provided pre-trained models when registering queries to run on different video feeds. Due to the complexity of model development, we observe that users opt to leverage existing (popular) architectures geared for their target task (e.g., YOLOv3 for object detection), and train those models for specific object(s) of interest and datasets (e.g. detecting vehicles at Main St.) to generate a unique set of weights. Despite being allowed, custom architectures were never provided in our deployment.

Accordingly, we selected the 7 most popular families of models across our pilot deployment and recent literature [24, 32, 63, 65, 68, 73, 74, 76, 89, 144]: YOLO, Faster RCNN, ResNet, VGG, SSD, Inception, and Mobilenet. From each family, we selected up to 4 model variants

(if available) that exhibit different degrees of complexity and compression. For instance, from YOLO, we consider {YOLOv3, Tiny YOLOv3}; similarly, we consider ResNet{18, 50, 101, 152}. The selected models focus on two tasks – object classification and detection – and for each, we train different versions for all combinations of the following objects: people and vehicles (e.g., cars, trucks, motorbikes). Classification and detection accuracy are measured using F1 and mAP [47].

Videos. Our dataset consists of video streams from 12 cameras in our pilot deployment that span two metropolitan areas. From each region, we consider cameras at adjacent intersections, and those spaced farther apart within the same metropolitan area; this enables us to consider different edge box placements, i.e., at a traffic intersection vs. further upstream to service a slightly larger geographic location. From each stream, we scraped 120 minutes of video that cover 24-hour periods from four times of the year.

Edge boxes. Our review of on-premise edge boxes focused on 5 commercial offerings: Microsoft Azure Stack Edge [2], Amazon Outposts [1], Sony REA [127], NVIDIA Jetson [11], and Hailo Edge-AI-box [56]. These servers each possess on-board GPUs and offer 2-16 GB of total GPU memory. Since edge inferences do not typically span multiple GPUs, we focus on model merging and inference scheduling *per GPU*. This does not restrict GEMEL to single-GPU settings; rather, it means that our merging and scheduling techniques are applied separately to the DNNs in each GPU, with the assumption that each merged model runs on only one GPU.

Workload construction. Recent works highlight that 10s of videos are usually routed to each edge box [15, 68], which runs upwards of 10 queries (or DNNs) on each feed [20, 24]. Our experience was similar: it was typical to direct the max possible number of feeds to an edge box, with the goal of *minimizing the number of edge boxes required to process the workload*. To cover this space, and since we focus on per-GPU inference optimization, we generated an exhaustive list of all possible workloads sized between 2-50 DNNs using the models above. We then sorted the list in terms of the potential (percentage) memory savings

(using the methodology from §3.4), and selected 15 workloads: 3 random workloads from the lower quartile (i.e., *Low Potential (LP1-3)*), 6 from the middle 50% (i.e., *Medium Potential (MP1-6)*), and 6 from the upper quartile (i.e., *High Potential (HP1-6)*). We chose this ratio to match that from our deployment. MP and HP workloads each constituted 30-50% of the total workloads since (1) users tended to employ the same few model variants from a limited set of popular families, and (2) each user typically used the same architecture (but not weights) for different feeds in a region. LP workloads were less common (<20%), and arose from different users opting for different model families.

Each workload was randomly assigned to one of the cities, with the constituent models being randomly paired with the available videos. §5.4 details the workloads, each of which exhibits heterogeneity in terms of the families, tasks, videos, and (combinations of) target objects of the included models. In summary, the workloads contain 3-42 queries (avg: 15) across 3-7 video feeds (avg: 5), featuring 2-10 unique models (avg: 6) and 2-5 different objects (avg: 4). We consider additional workloads, models, objects, and videos in §3.6.3.

Result presentation. End-to-end accuracy depends on the available GPU memory. However, each workload requires a different minimum amount of memory to run, i.e. the GPU should be able to load/run the most memory-intensive model in isolation for a batch size of 1. Further, the memory needed to avoid swapping (i.e. to load all models and run one at a time) also varies per workload; we call this no_swap. To ensure comparability across all presented accuracy results and to focus on memory-bottlenecked scenarios, we assign each workload three memory settings to be evaluated on (listed in §5.6): (1) the minimum value (*min*), (2) 50% of the no_swap value (*50%*), and (3) 75% of the no_swap value (*75%*).

3.3 Motivation

3.3.1 Memory Pressure in Edge Video Analytics

To run inference with a given model, that model's layers and parameters must be loaded into the GPU's memory, with sufficient space reserved to house intermediate data generated while running, e.g. activations. The amount of data generated (and thus, memory consumed)

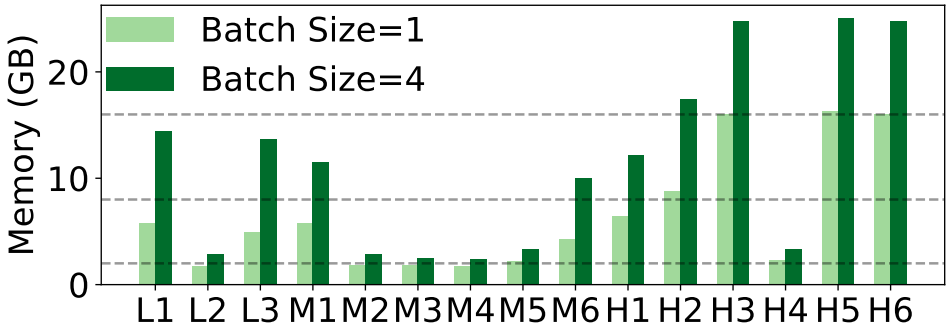


Figure 3.1: Per-workload memory requirements for two popular batch sizes used in video analytics [123]. Dashed lines represent the available GPU memory on several commercial edge boxes.

Model	Load Memory (Time)	Run Memory (Time)		
		BS=1	BS=2	BS=4
YOLOv3	0.24 (49.5)	0.52 (17.0)	0.73 (24.0)	1.22 (39.9)
ResNet152	0.24 (73.3)	0.65 (24.8)	0.98 (26.3)	1.71 (26.7)
ResNet50	0.12 (27.1)	0.35 (8.4)	0.50 (8.5)	0.84 (8.5)
VGG16	0.54 (72.2)	0.74 (2.1)	0.89 (2.4)	1.18 (2.4)
Tiny YOLOv3	0.04 (6.7)	0.15 (3.0)	0.18 (5.2)	0.24 (5.2)
Faster RCNN	0.73 (117.3)	3.70 (115.4)	6.96 (210.1)	12.47 (379.4)
Inceptionv3	0.12 (11.8)	0.19 (9.1)	0.23 (9.1)	0.34 (9.1)
SSD-VGG	0.11 (16.1)	0.23 (16.5)	0.33 (25.7)	0.51 (44.6)

Table 3.1: Memory (GB) and time (ms) requirements for loading/running inference with 3 different batch sizes (in frames). Run memory values include load values, but exclude memory needs of serving frameworks. Results use a Tesla P100 GPU.

during inference depends on both the model architecture and the batch size used; a higher batch size typically requires more memory.

Figure 3.1 shows the total amount of memory (i.e., for both loading and running) required for each of our workloads and two batch sizes; the listed numbers exclude the fixed memory that ML frameworks reserve for operation, e.g., 0.8 GB for PyTorch [22]. As shown, many workloads do not directly fit into edge box GPUs, and the number of edge boxes necessary to support a given workload can quickly escalate. For instance, even with a batch size of 1 frame, 73% of our workloads need more than one edge box possessing 2 GB of GPU memory; with a batch size of 4, 60% and 27% require more than one edge box with 8 GB and 16 GB of memory.

Table 3.1 breaks this memory pressure down further by listing the amount of loading and running memory required for representative models in our workloads. When analyzed in

the context of the scale of edge video analytics workloads, the picture is bleak, even with a batch size of 1. For example, a 2 GB edge box can support only 1, 2, or 3 VGG16, YOLOv3, or ResNet50 models, respectively, after accounting for the memory needs of the serving framework. Moving up to 8 and 16 GB edge boxes (of course) helps, but not enough, e.g. an 8 GB box can support 13 YOLOv3 or 2 Faster RCNN models, both of which are a drastic drop from the 10s of models that workloads already involve (§3.2).

3.3.2 Limitations of Existing GPU Memory Management

Space and time sharing. Existing learning frameworks recommend model allocation at the granularity of an entire GPU [71]. Space-sharing techniques [18, 21] eschew this exclusivity and partition GPU memory per model. Although space-sharing approaches are effective when a workload's models can fit together in GPU memory, they are insufficient when that does not hold, which is common at the edge (§3.3.1)

There are two natural solutions when a workload's models cannot fit together in the target GPU's memory. The first is to place models on *different* GPUs [53, 123], which resource-constrained edge settings cannot afford. The second is to *time share* the models' execution in the GPU by *swapping* them in and out of GPU memory (from CPU, via a PCIe interface) [32, 53, 65, 123, 150]. However, as we will show next, time-sharing techniques are bottlenecked by frequent model swapping, which severely limits their utility. More recently, SwapAdvisor [65] and Antman [150] proposed swapping at finer granularities, e.g., individual or a few layers. However, even these approaches are limited in our case because a handful of layers in vision DNNs typically account for most memory usage (§3.5.2); edge boxes often lack the GPU memory to concurrently house even these expensive singular layers.

We evaluated time-sharing strategies in our setting by considering a hybrid version that *packs* models into GPU memory, and executes as many models as possible while ensuring that swapping costs for the next model to run are hidden. Concretely, we extend Nexus [123] to incorporate such pipelining. Our variant first organizes models in round-robin order (as Nexus does), and profiles the workload offline to determine the best global list of per-model batch sizes that maximizes the minimum achieved per-model throughput while adhering to an

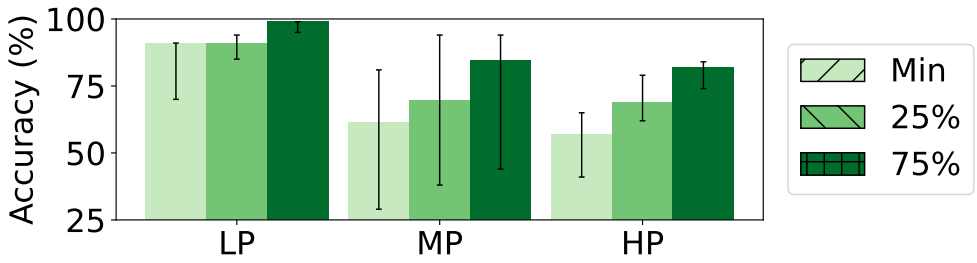


Figure 3.2: Achieved accuracy with time/space-sharing alone (i.e. using our Nexus variant) for different memory availability (following the definitions in §3.2). Bars list results for the median workload in each class, with error bars spanning min to max.

SLA (i.e., a per-frame processing deadline). Using those batch sizes, the scheduler traverses the round robin order with the goal of minimizing GPU idle time: when loading the next model, if there does not exist sufficient memory to load both parameters and intermediates, the most recently run model (i.e., the one whose next use is in the most distant future) is evicted to make space.

Figure 3.2 shows the accuracy of the Nexus variant on our workloads with an SLA of 100 ms; we saw similar trends for other common SLAs in video analytics [123]. As shown, accuracy drops are substantial, growing up to 43% relative to a setting when there exists sufficient memory to house all models at once. The root cause is the disproportionately high loading times of vision DNNs that must be incurred when swapping. As shown in Table 3.1, per-model loading delays are 0.98-34.4× larger than the corresponding inference times (for batch size 1). When facing the strict SLAs of video analytics, these loading costs result in the inability to keep pace with incoming frame rates, and thus, dropped (unprocessed) frames; the Nexus variant skipped 19-84% of frames.

Predicting workload characteristics. Another approach is to selectively preload models based on predictions of the target workload [157], e.g. deprioritizing inference on streams at night due to lack of activity. However, in edge video analytics, spatial correlation between streams results in model demands being highly correlated [70, 74, 89, 93].

Compression and quantization. These techniques generate lighter model variants that impose lower memory (and compute) footprints and deliver lower inference times. Some

families offer off-the-shelf compressed variants (e.g. Tiny YOLOv3), and techniques such as neural architecture search can be used to develop cheaper variants that are amenable to deployment constraints [54]. Regardless, in reducing model complexity, these cheaper model variants typically sacrifice accuracy and are more susceptible to drift, relative to their more heavy-weight counterparts [24, 131]; consequently, determining the feasibility of using such models in a given setting requires careful tuning and analysis by domain experts.

We consider compression and quantization as orthogonal to merging for two reasons. First, in common workloads that involve a mix of models and tasks (§3.2), it may not be possible to compress all of the models while delivering sufficient accuracy. However, even a handful of non-compressed models can exhaust the available GPU memory (§3.3.1). Second, compressed models exhibit sharing opportunities: our workloads include compressed and non-compressed models (§3.2), and our results show that GEMEL is effective for both (§3.6).

3.4 Our Approach: Model Merging

To address the high model loading costs that plague existing memory management strategies when workloads cannot fit together in a GPU's memory (§3.3.2), we propose *model merging*. Merging is complementary to time/space sharing of GPU memory, and its goal is straightforward: share layers across models such that only one copy of each shared layer (i.e., layer definition and weights) must be loaded into GPU memory and can be used during inference for all of the models that include it. The benefits are two-fold: (1) reduce the overall memory footprint of a workload, thereby enabling edge boxes to house more models in parallel and perform fewer swaps (or equivalently, lower the number of edge boxes needed to run the workload), and (2) accelerate any remaining swaps by reducing the amount of extra memory that the next model to load requires. Note that merging does not involve sharing intermediates (i.e. layer outputs) for a common layer because models may run on different videos (and thus, inputs). We next highlight the promise for merging in edge video analytics (§3.4.1), and then lay out the challenges associated with realizing merging in practice (§3.4.2).

3.4.1 Opportunities

Commonality of layers. A layer is characterized by both its architecture and its weights. In ML frameworks (e.g. PyTorch, TensorFlow), the architecture is defined by first specifying a layer type (e.g. convolutional, linear, batch normalization), which in turn indicates how the layer transforms inputs, and dictates the set of defining parameters that must be specified (e.g. convolutional: kernel, stride, etc., linear: # of input features, bias, etc.). A layer's weights are a matrix of numbers whose dimensions match the layer structure. To successfully share a layer across a set of models, that layer must be *architecturally* identical in each model, but its weights need not be the same across appearances.

Architectural equivalence is determined directly from the model definition in the ML framework (i.e., no inference required): the layers must be of the same type, with identical values for type-specific properties. Using this approach, we studied pairs of 24 different models to identify and analyze layers with identical architectures. §5.7 and Figures 5.6-5.8 present our comprehensive results and break down sharing opportunities by layer type. Below, we summarize our findings; Figure 3.3 lists results for representative model pairs.

Model pairs fall into one of three categories: (1) instances of the same model, (2) different models in the same family (e.g. ResNet variants), and (3) different models in different families. Multiple instances of the same model unsurprisingly match on every layer; this favorable scenario is not uncommon in edge video analytics, as several model architectures tend to dominate the landscape [23] and a given model can be employed on different video feeds or in search of different objects (§3.2). More interestingly, we also observe sharing opportunities across different models from the same (up to 84.6%) and divergent (up to 96.3%) families.

Models within the same family exhibit significant sharing opportunities as larger variants are typically extended versions of the original base model. For instance, ResNet models share ResNet blocks (groups of 2-3 convolutional layers) that are repeated at different frequencies, as well as the first convolutional layer and final fully-connected layer. As a result, all 41

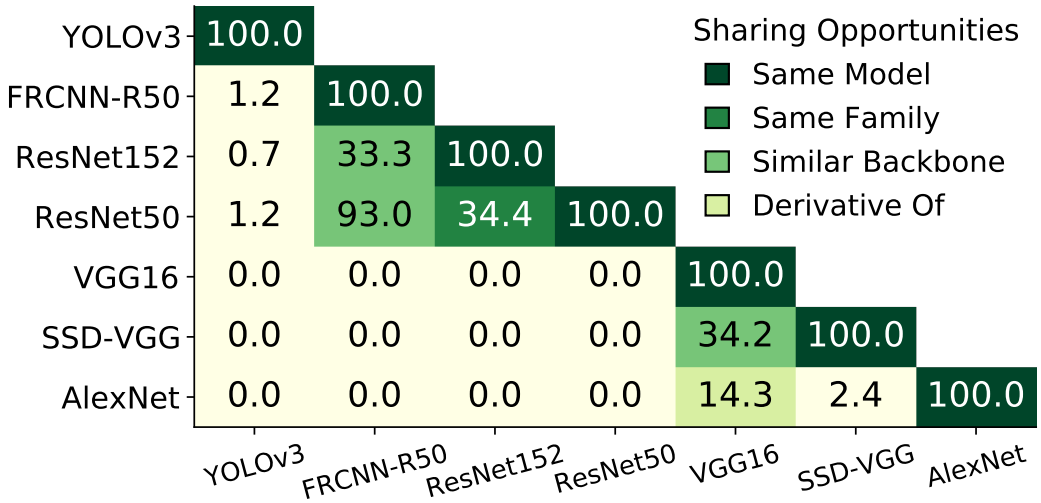


Figure 3.3: Percentage of architecturally identical layers across different model pairs.

layers of ResNet18 are shared with ResNet34 (Figure 5.8). Similarly, in the VGG family, models share the exact same base architecture and add different numbers of convolutional layers, e.g., VGG19 shares all 16 of VGG16's layers (13 convolutional and 3 fully-connected; Figure 5.6).

Sharing for models in different families comes in two main forms: (a) 'similar backbones' and (b) 'derivatives of.' Scenario (a) includes pairs of detectors that use the same (or similar) backbone networks for feature extraction, e.g., SSDs that use any VGG backbone, or FasterRCNNs that use any ResNet backbone. (a) also includes pairs of classifiers and detectors where the classifier (or a close variant) is used as the detector's backbone. For instance, every layer in the ResNet50 backbone of FasterRCNN (which constitutes 51% of the detector's layers) appears in the ResNet101 classifier. Similar examples include SSD-VGG with any VGG variant, and SSD-MobileNet with MobileNet. Scenario (b) involves cases where one model family was derived directly from another. For example, VGG was developed by replacing AlexNet's large kernels with multiple smaller ones [126]; VGG16 and AlexNet share 3 out of 16 layers, including 2 fully-connected layers at the end (Figure 5.7). Other examples include InceptionNetV3 [133] with GoogLeNet [132].

In summary, 43% of all pairs of different models present sharing opportunities. Of those with substantial ($\geq$ 10%) common layers, 51% have models in the same family, while 49%

involve models from different families; for the latter, 76% are 'similar backbones' and 24% are 'derivatives of.'

These layer similarities generally follow from the fact that the considered models are all vision processing DNNs. That is, they all ingest pixel representations of raw images, and employ a series of traditional CV operations [25, 79], e.g. a convolutional layer is applying a learned filter to raw pixel values in preparation for downstream processing. Moreover, the target tasks are rooted in identifying and characterizing objects in the scene using low-level CV features such as detected edges and corners [35, 63, 80, 83, 89, 162, 163].

Prior work has capitalized on such similarities for efficient multi-task learning [39, 73, 154] and architecture search [92, 109]. Those efforts aim to reduce computation overheads by sharing "stems" of models, i.e. contiguous layers (and their generated intermediates) starting from the beginning of the models. In contrast, we aim to exploit architectural similarities to reduce memory overheads via merging. As a result, merging only requires layer definitions and weights to be shared, but not generated intermediate values. This distinction is paramount because, as we will discuss in §3.5.2, memory-heavy layers typically reside towards the end of vision DNNs. Consequently, stem sharing would require almost all model layers to be shared to reap substantial memory savings, which in turn brings unacceptable accuracy drops (§3.4.2 and §3.6). Merging, on the other hand, can share only those memory-heavy layers to simultaneously deliver substantial memory savings and preserve result accuracy.

Potential memory savings and accuracy improvements. Figure 3.4 shows the memory savings from sharing all of the common layers across the models in each of our workloads; this represents an *upper bound* on merging benefits as it disregards the challenge of identifying an acceptable set of weights per shared layer (§3.4.2). As shown, the memory savings are substantial: per-workload memory usage dropped by 17.9-86.4% relative to no merging, translating to raw savings of 0.2-9.9 GB. Importantly, these savings result in 2 and 4 new workloads fitting entirely (no swapping) on edge boxes with 2 GB and 8 GB of GPU memory (with batch size 1). Similarly, the number of 2 GB edge boxes needed to support each workload drops from 1-9 to 1-4. We further evaluated the resulting impact on end-to-

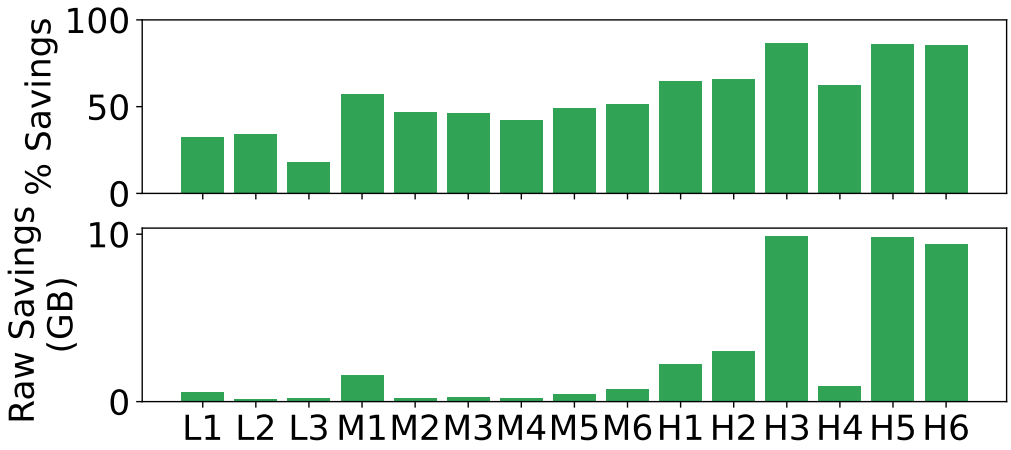


Figure 3.4: Potential memory savings when all architecturally identical layers are shared across the models in each workload.

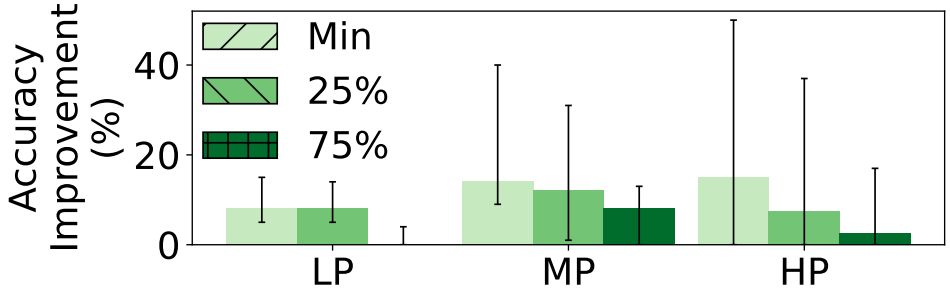


Figure 3.5: Potential accuracy improvements when sharing all architecturally identical layers. Memory availability is defined in §3.2, bars list medians, and error bars span min to max.

end accuracy by comparing the performance of the Nexus variant from §3.3.2 when run on workloads with and without (maximal) merging. Models in both cases were ordered in the same way, to maximize the benefits of merging (§3.5.4). As shown in Figure 3.5, merging can boost accuracy by up to 50% across our workloads. These benefits are a direct result of lower swapping costs, and the resulting ability to run on 29-61% more frames.

3.4.2 Challenges

Merging layers for memory reductions requires using shared weights across the models in which those layers appear. However, those shared weights must not result in accuracy violations for any of the models, despite their potentially different architectures/tasks, target objects/videos, etc.; such accuracy drops would forego merging benefits from faster swap-

ping. Concretely, there are two core challenges in practically exploiting the architectural commonalities from §3.4.1.

Challenge 1: sharing vs. accuracy tension. To maximize memory savings, merging seeks to share as many architecturally identical layers as possible across a workload's models. However, we observe that accuracy degradations steadily grow as the number of shared layers increases. Figure 3.6 illustrates this trend by sharing different numbers of identical layers across representative pairs of models that vary on the aforementioned properties, e.g. target task. These results were obtained when we increase the number of shared layers by moving from start to end in the considered models, but similar trends are observed for other selection strategies (e.g. random) and models.

The reason for this behavior is intuitive: the retraining performed to assess the feasibility of a sharing configuration is *end-to-end* across the considered models. During this process, weights are being tuned for all of the layers in all of the models, with the constraint being that the shared layers each use a unified set of weights. Sharing more layers has three effects: (1) more constraints are being placed on the training, (2) it is harder to find weights for (the shrinking number of) unshared layers that simultaneously accommodate the growing constraints, and (3) learning each model's desired function becomes more difficult as there exist fewer overall parameters to tune [26, 88]. It is for these reasons that isolated merging strategies such as averaging weights across copies of each shared layer (while keeping other layers unchanged) do not suffice; we find that sharing even single layers in this way almost always results in unacceptable accuracy dips.

Digging deeper, the issue stems from non-convex optimization of DNNs, which leads to several equally good global minima [77, 78]. Thus, training even two identical models on the same dataset, and for the same task/object, often results in divergent weights across each layer, despite the resultant models exhibiting similar overall functionality.

Challenge 2: retraining costs. The retraining involved in determining whether a set of layers to share can meet an accuracy target, and if so, the weights to use, can be prohibitively

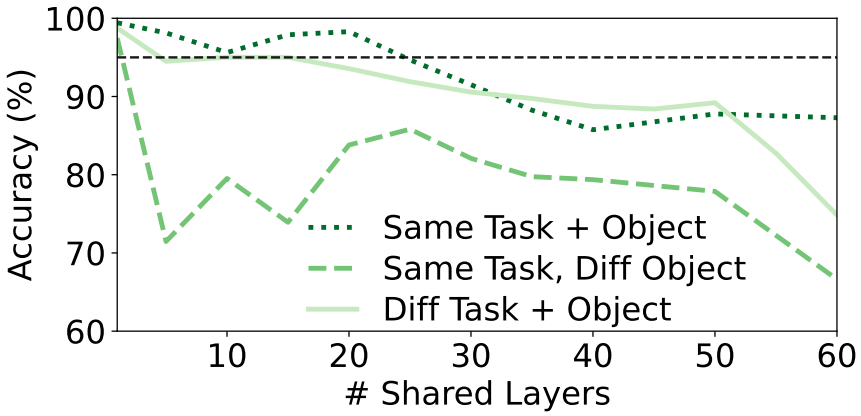


Figure 3.6: Accuracy after 5 hours of retraining when sharing additional architecturally-identical layers for different model pairs (starting from their origins). Tasks cover detection (Faster RCNN) and classification (ResNet50), and two objects: people, vehicles. Results list the lower per-model accuracies per pair.

expensive. For instance, each epoch when jointly retraining two Faster RCNN models that detect cars at nearby intersections (i.e. a simple scenario) took ≈35 mins, and different combinations of layer sharing required between 1-10 epochs to converge. These delays grow as more models are considered since training data must reflect the behavior of all of the unmerged models that are involved, e.g. by using the original training datasets for each of those models. Worse, it is difficult to know, a priori, which sharing configurations can meet accuracy targets (and which will not) in a reasonable time frame. For example, the model pairs in Figure 3.6 have largely different 'breaking points.' The result also fails to support the use of intuitive trends to predict the success of sharing configurations: models targeting the same task or object do not exhibit any discernible advantage.

3.5 GEMEL Design

GEMEL is an end-to-end system that practically integrates model merging into edge video analytics pipelines by addressing the challenges in §3.4.2. We first provide an overview of GEMEL's operation, and then describe the core observations (and resulting optimizations) that it leverages to enable timely merging without violating accuracy requirements.

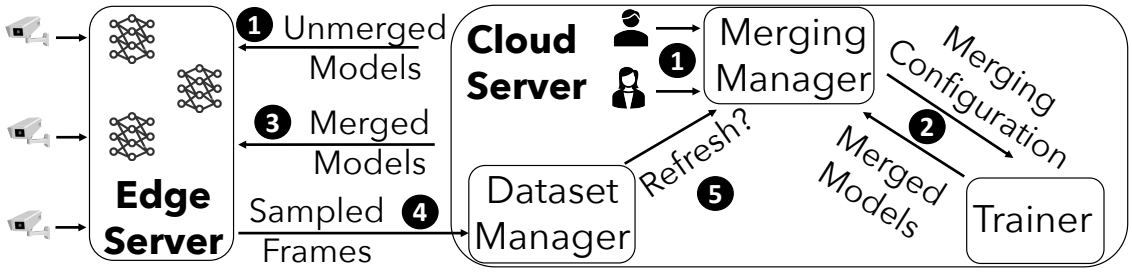


Figure 3.7: GEMEL architecture.

3.5.1 Overview

Figure 3.7 shows GEMEL's cloud merging and edge inference workflows. As in existing pipelines [20, 74, 89], users register inference tasks (or "queries") at GEMEL's cloud component by providing a DNN, and specifying the input video feed(s) to run on as well as the required accuracy for the results. Upon receiving new queries, GEMEL bootstraps edge inference by sending unaltered versions of the registered models to the appropriate edge box(es) ❶. When GPU memory is insufficient to house all of those models, edge boxes run the Nexus variant from §3.3.2 that pipelines inference and model loading to maximize the min per-model throughput.

After initiating edge inference, GEMEL's cloud component begins the merging process, during which it *incrementally* searches through the space of potential merging configurations across the registered models, and evaluates the efficacy of each configuration in terms of both its potential memory savings and its ability to meet accuracy requirements ❷. The evaluation of each configuration involves joint retraining and validation of the models participating in merging. Since GEMEL's goal is to ensure that the retrained models deliver sufficient accuracy (relative to the originals) on the target feeds, data for these tasks can be obtained in one of two ways: users can supply the data used to train the original models, or GEMEL can automatically generate a dataset by running the supplied model (or a high-fidelity one [74, 158]) on sampled frames from the target feed.

At the end of each merging iteration, if the considered configuration was successfully retrained to meet the accuracy targets for all constituent models, GEMEL shares the updated

merged models with the appropriate edge boxes ❸. New merging results may result in altered edge inference schedules to maximize merging benefits for reducing swapping costs and boosting inference throughput. The iterative merging process for the current workload then continues until (1) the cloud resources dedicated to merging have been expended, (2) no configurations that can deliver superior memory savings are left to explore, or (3) models with sharing opportunities are either newly registered or deleted by users.

GEMEL periodically assesses *data drift* for its merged models. As in prior systems [89, 131], edge servers periodically send sampled frames (and their inference results, if collected) to GEMEL's cloud component ❹. These sampled frames are used to augment the datasets considered for retraining merged models, and to track the accuracy of recent results generated at the edge by deployed merged models. For the latter, GEMEL runs the original user models on the sampled videos and compares the results to those from the merged models. If accuracy is below the target for any query, GEMEL reverts edge inference to use the corresponding original (unmerged) models, and resumes merging and retraining, starting with the previously deployed weights ❺.

Implementation. GEMEL's main components are training models at the cloud server and running the scheduler at the edge. GEMEL uses PyTorch [22] to manage cloud merging and edge inference. During training, a single optimizer manages the weights across all considered models; the optimizer holds a single copy of weights for each layer that is shared across the models. Aside from this, GEMEL's training process mirrors classic multi-task training[39]: it forms a collective pool of an equal number of data samples from all models and randomly selects batches from this pool. Samples are run through their respective models, each model calculates its loss individually, and losses are summed over all models. In this way, layers that are shared are updated by the concurrent training of multiple models within a single batch.

The Nexus-variant scheduler chooses when to load and evict models as described in §3.5.4. To load a model into GPU memory, the scheduler simply calls ".cuda()" on that model's PyTorch object. PyTorch automatically only loads layer weights not already in GPU memory.

However, when evicting a model, PyTorch, by default, removes all of the layers' weights from GPU memory. This poses a problem if some of those weights are needed by models still in GPU memory (i.e. they are shared). To avoid this, the scheduler: (1) maintains a running list of shared layers that are needed by models currently in GPU memory or next in line to be loaded, and (2) when a model needs to be evicted, only evicts weights corresponding to layers not in the list. Overall, GEMEL is implemented in ≈3500 LOC: 500 for finding shared layers and sharing them according to the heuristic, 2500 for dataset management and retraining, and 500 for scheduling models at the edge.

3.5.2 Guiding Observations

Two key empirical observations guide GEMEL's approach to tackling the challenges in §3.4.2. We describe them in turn.

Observation 1: power-law memory distributions. We find that vision DNNs commonly exhibit power-law distributions in terms of memory usage, whereby a few "heavy-hitter" layers account for most of the overall model's memory consumption. Figure 3.8 illustrates this, showing that for 80% of considered models, 15% of the layers account for 60-91% of memory usage. For example, a single layer in VGG16 is responsible for 392 MB (the entire model is 536 MB) and corresponds to the steep slope around the x=80% mark. Similarly, Tiny YOLOv3 has three layers (around the 38%, 45%, and 65% marks) that together use 35 MB of its total 42 MB.

Heavy-hitter layers come in one of two forms. The first are the convolutional layers at the end of the feature extractor that condense the numerous low-level features extracted by prior layers (e.g., shapes, colors) into higher-level, more abstract features (e.g., eyes, nose). The second are the subsequent fully-connected layer(s) that learn more robust patterns from all possible combinations of those high-level features, e.g., eyes, nose, and fur might each suggest a dog, but the combination is a stronger indicator. Note that models generally include one such fully-connected layer per sub-task, e.g., detectors have one for finding bounding boxes and one for classifying objects. Memory-heavy fully-connected layers are spatially close to

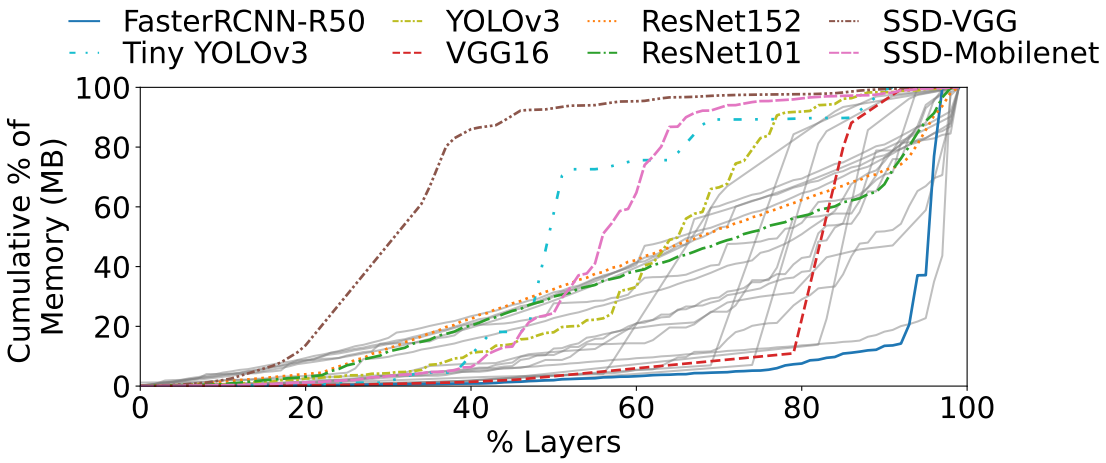


Figure 3.8: Cumulative memory consumed by each model's layers moving from start to end of the model. §5.7 has full legend.

one another (within a few layers), and are usually followed by 1-2 cheap fully-connected layers that extract predictions from the final feature vector.

The main exception is ResNet, whose models use residual layers to address accuracy saturation limitations of prior deep models [60]. ResNet models have memory-heavy ResNet blocks (set of convolutional layers) that repeat at varying frequencies, thereby distributing memory more evenly across the models, e.g., ResNet101 and ResNet152 repeat the same ResNet block 23 and 36×, leading to gradual slopes in Figure 3.8. DenseNet has the same pattern [66].

Figure 3.8 also shows that heavy-hitter layers most often appear in the latter half of a model's architecture (since both forms involve condensing features from earlier layers), complicating the use of stem sharing for memory savings (§3.4.2). For example, Faster RCNN's expensive fully-connected layers fall at layers 101 and 104 out of 106, and together account for 76% of total memory. The few cases with heavy-hitters in the middle of a model (between the 20-60% marks) are "single-shot" detectors (SSD-VGG, SSD-Mobilenet, Tiny YOLOv3, YOLOv3) that find bounding boxes and classify objects at once, rather than as disparate subtasks. These models replace the few memory-heavy fully-connected layers (for those subtasks) with many cheaper convolutional layers; doing so extends model lengths and shifts the large jump from memory-heavy *feature extractor* layers to earlier.

These observations have two implications on merging. First, strategies can reap most potential memory benefits by targeting the few heavy-hitter layers in models. Thus, the tension between memory savings and accuracy is far more favorable than that between the number of shared layers and accuracy (Figure 3.6). Second, strategies should be agnostic to the position of heavy hitters in models, and must support the common case where heavy hitters appear towards the end.

Observation 2: independence of per-layer merging decisions. In DNNs, layers are configured based on input formats, target task, execution time, etc. Hence, a natural assumption is that the ability to share any one layer is dependent on sharing decisions for other layers, e.g. a layer may be shareable if and only if other layers are shared. Prior work has highlighted that inter-layer dependencies primarily arise between neighboring layers, e.g. with transfer learning, performance drops are largest when splitting neighboring layers [154]. Thus, to determine the existence of layer-wise dependencies as it pertains to merging, we focus our analysis on (potential) dependencies between neighboring layers; we also consider other layers via random selection. Using the 25% most memory-heavy layers for each model in our workloads, we test whether accuracy targets are met under different sharing configurations (described in Table 3.2).

As shown, we *never* observe a case where a layer is unable to meet an accuracy target on its own, but it is able to meet the accuracy target when some other layers are also shared (shaded row in Table 3.2). This is consistent with our finding that sharing more layers leads to larger accuracy degradations (Figure 3.6) since additional constraints are placed on the weights for those layers, and fewer (unconstrained) non-shared layers exist to help satisfy the constraints. The implication is that layers can be considered independently during merging without harming their potential merging success.

Takeaway. Collectively, these observations motivate an incremental merging process (detailed in §3.5.3) that attempts to share one new layer at a time, and prioritizes heavy-hitter layers that consume the most memory (and are thus the most fruitful to share). In this

	Only Alone	Only Alternate	Both	Neither
1 Each Side	1.1%	0.0%	97.6%	1.3%
2 Each Side	3.7%	0.0%	95.0%	1.3%
Random	8.5%	0.0%	90.2%	1.3%

Table 3.2: Sharing a layer alone vs. *alternate* approaches (sharing a layer with one or two neighbors on each side, or with 3 random sets of 1-10 layers). Results are % of runs that meet accuracy targets (aggregated across 80, 90, 95%), and list cases where the layer alone met but an alternate did not, an alternate met but the layer alone did not, both met, and neither met.

manner, memory-heavy layers are considered in the most favorable settings (i.e. with the fewest other shared layers), and each increment only modestly adds to the likelihood of not meeting accuracy targets.

Note. Despite arising across our diverse workloads, these observations are not guarantees. Importantly, violation of these observations only results in merging delays (inefficiencies), but not accuracy breaches; accuracy is explicitly vetted prior to shipping merged models to the edge for inference.

3.5.3 Merging Heuristic

GEMEL begins by enumerating the layers that appear in a workload, and annotating each with a listing of which models the layer appears in (and where) and the total memory it consumes across the workload; we refer to all appearances of a given layer as a 'group.' GEMEL then sorts this list in descending order of memory consumption, e.g. a 100 MB layer that appears in 4 models would be earlier than a 120 MB layer that appears 3 times. Thus, memory-heavy groups, or those that would yield the largest memory savings if successfully merged, are towards the start of the list.

GEMEL then maintains a running merging configuration, and simultaneously merges and trains layers across models in an *incremental* fashion. To begin, GEMEL selects the first group from the sorted list (i.e. the one that consumes the most memory in the workload) and attempts to share it across all of the models in which it appears; this group is added to the running configuration. While a subset of models could be considered instead, GEMEL

aggressively opts to first try sharing across all models in the group, and then to selectively remove appearances of the layer when the resulting accuracy is insufficient. The reason is that we did not observe any model clustering strategies (e.g. based on task) that identified models consistently unable to share layers.

To retrain and merge the current running configuration, GEMEL selects initial weights for the newly added group from a random model that includes that layer. We tried selecting weights from each model (including the one with the highest accuracy) but found no difference in the # of epochs needed to meet accuracy. We also tried default initialization techniques (e.g., Kaiming[61]), which led to lower accuracy. Retraining continues until the merged models each meet their accuracy targets, or a preset time budget elapses (10 epochs by default). If retraining is successful, GEMEL adds the next group in the sorted list to the running configuration, and resumes retraining from the weights at the end of the previous iteration. The generated merged models are sent to the edge box and incorporated into edge inference (§3.5.4).

If retraining is not successful at the end of an iteration, GEMEL must decide whether to prune layers from the current group and try again, or to discard the group altogether and move on to the next one in the sorted list. To do this, GEMEL follows a strategy that aims to balance fast memory savings and avoidance of unsuccessful training rounds, with priority on the latter since failures can consume 3-10 epochs (each up to 30 min) and provide no new memory savings. Specifically, recall that each time a new group is considered, the number of shared layers in the merging configuration grows by the size of the group. To counter this 'additive increase,' upon unsuccessful retraining, GEMEL halves the current group, eliminating half of the layer appearances. If the resulting layer appearances consume more memory than the next group, GEMEL considers those layers; else, GEMEL removes the current group from the running policy, and moves to the next one. In either case, retraining resumes from the weights at the end of the last successful iteration. We compare against alternate merging heuristics in §3.6.2.

Accelerating retraining. Each iteration requires GEMEL to run retraining over many epochs, and validate the results accuracy-wise. To accelerate training and validation, GEMEL takes an adaptive approach. During validation, as per-model accuracy values approach their targets, it is often unnecessary to train further on full epochs of data. Instead, GEMEL reduces the training data once the accuracy is within a pre-defined threshold of the target. Specifically, GEMEL reduces the amount of data so it is inversely proportional to the gap in accuracy normalized by the lift since the previous training. Reducing data on such *early success* directly translates to lower training times. Similarly, GEMEL detects *early failures* by looking at the validation results and removing models that are not improving at the same pace as the others after some time (3 epochs by default). We empirically observe that early success and early failure detection drastically (28% on average) reduces retraining times.

3.5.4 Edge Inference

Upon receiving a new set of merged models from GEMEL's cloud component, an edge server quickly incorporates those models into its inference schedule. However, to ensure that merging benefits are maximized, the schedule is altered to reduce the amount of data that must be loaded across the anticipated swaps. During the offline profiling Nexus uses to select per-model batch sizes, GEMEL estimates per-workload-iteration swapping delays based on per-model computation costs and swapping delays (both influenced by merging). The idea is that, when merging is used, in addition to ordering models to reduce the number of swaps, models that share the most layers should be placed next to one another in the load order. This lowers the cost of each swap by enabling finer-grained swapping, where only those layers in the next model that are not already in GPU memory must be loaded.

More generally, all schedulers will reap merging benefits in the event that GEMEL enables a workload to entirely fit on an edge box (without swapping). Additional benefits depend on the specific scheduler. For schedulers that employ a statically-configured load order [104, 123], GEMEL can directly modify the schedule as described above to maximize benefits. Other schedulers [53] dynamically select the load order to optimize for a certain metric. Such schedulers typically incorporate model loading times when estimating the efficacy of different

orders, and thus would naturally factor in the effects of merging per swap. Note that merging benefits would be considered in the context of meeting the optimization metric(s) rather than minimizing global loading delays (as in GEMEL's Nexus variant). Lastly, schedulers that ignore load times in favor of policies such as FIFO [138] or priority scheduling [153] will only see merging-induced reductions in loading costs if merged models are (by chance) neighbors in the order. Note that finer-grained [65, 150] and space-sharing [12, 18, 21, 24] schedulers follow the same principles: shared layers should be adjacent in the load orders for the former, while models with the most shared layers should be placed in the same GPU partition for the latter.

3.6 Evaluation

We primarily evaluated GEMEL across the diverse workloads and settings from §3.2. Our key findings are:

- GEMEL improves per-workload accuracies by 8-39% compared to time/space-sharing strategies alone; these improvements result from GEMEL processing 13-44% more frames (while adhering to SLAs).
- GEMEL lowers memory needs by 17.5-60.7% (0.2-5.1 GB); savings are 5.9-52.3% more than Mainstream [73] (stem sharing), and within 9.3-29.0% of an optimal that ignores weights (and accuracy drops) when sharing layers.
- More than 70% of GEMEL's memory savings are achieved within the first 24-210 minutes of merging+retraining due to its incremental merging heuristic.

3.6.1 Overall Performance

End-to-end Accuracy Improvements. We first compare GEMEL with time/space-sharing solutions alone, i.e. the Nexus variant running with only unmerged (original) models. Our experiments consider all workloads and resource settings from §3.2, a per-frame processing SLA of 100 ms, and an accuracy target of 95%; trends hold for other accuracy targets and SLAs, which we consider in §3.6.2.

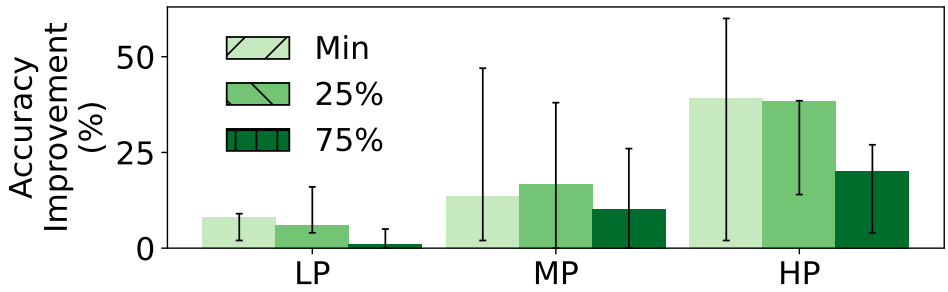


Figure 3.9: Accuracy improvements with GEMEL compared to time/space-sharing alone for different GPU memories (defined in §3.2). Bars list median workloads, with error bars as min-max.

Figure 3.9 presents our results, showing that GEMEL improves accuracy by 8.0%, 13.5%, and 39.1% for the median LP, MP, and HP workloads, respectively, when the edge box GPU's memory is just enough to load and run the largest model in each workload, i.e. the *min* setting. The origin of these benefits is GEMEL's ability to reduce the time blocked on swapping delays by 17.9-84.0%, which enables processing on 13-44% more frames than without merging.

Our results highlight two other points. First, GEMEL's benefits are highest for workloads that are most significantly bottlenecked by memory restrictions (and thus loading costs). For instance, workloads HP1 and LP1 exhibit largely different memory vs. computation profiles: loading costs are 66% of computation costs in the former, but only 15% in the latter. Accordingly, GEMEL's accuracy wins across the available memory settings are 11-60% and 5-16% for workloads HP1 and LP1. Second, Figure 3.9 shows that, as expected, GEMEL's benefits per workload decrease as the available GPU memory grows, e.g. accuracy improvements drop to 17.5% and 10.2% for the median MP workload when GPU memory grows to 50% and 75% of the total workload memory needs. The reason is straightforward: larger GPU memory yields fewer required swaps without merging.

Memory Reductions. Figure 3.10 lists the memory reductions that GEMEL delivers for each considered workload by sharing model layers and the associated weights, i.e. parameter reductions. We note that reported values here are based on GEMEL's final merging results and an accuracy target of 95%; we analyze the incremental nature of GEMEL's merging

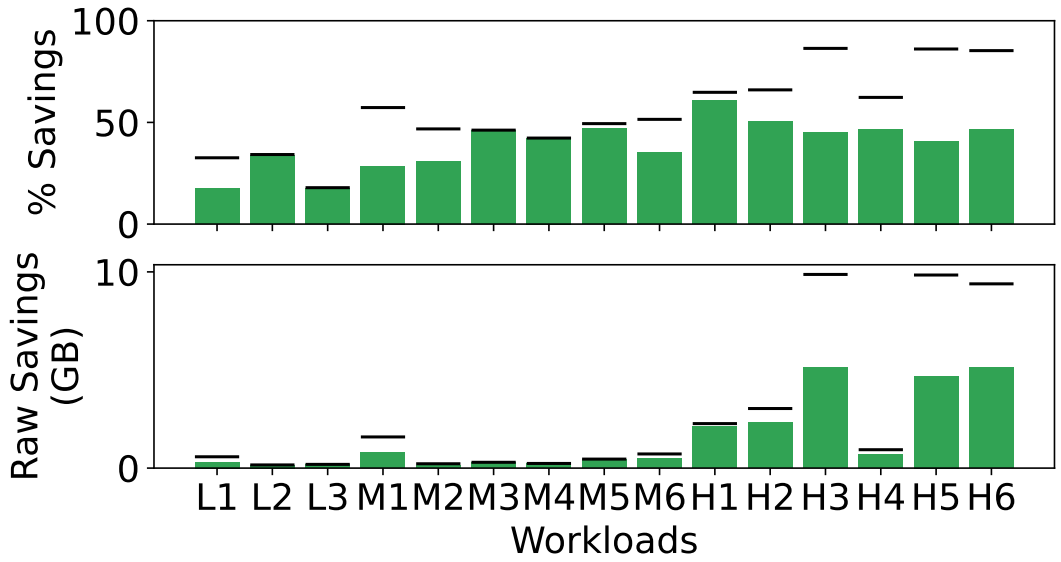


Figure 3.10: GEMEL's per-workload memory savings. Lines above bars show the theoretical optimal savings from Figure 3.4.

heuristic in §3.6.2. As shown, parameter reductions are 17.5-33.9% for LP workloads, 28.6-46.9% for MP workloads, and 40.9-60.7% for HP workloads; the corresponding raw memory savings are 0.2-0.3 GB, 0.2-0.8 GB, and 0.7-5.1 GB, respectively. When analyzed in terms of overall memory usage during inference (i.e. including the parameters, inference framework, and intermediate data generated during model execution), reductions are 4.5-48.1% across the workloads. Wins are generally higher for workloads with larger parameter reductions, with the exception of Workloads LP1 and LP3 (reductions of 6.3% and 4.5%) whose intermediates are particularly large relative to the parameters.

To better contextualize the above memory savings, we compare GEMEL with two alternatives. First, we consider a theoretical optimal (*Optimal*) that shares all layers that are architecturally identical across a workload's models, without considering accuracy (and the need to find shared weights for those layers). Thus, Optimal represents an *upper bound* on GEMEL's potential memory savings. Second, we compare with *Mainstream* [73], a recent stem-sharing approach. To run Mainstream, we trained each model in our workloads several times, each time starting with pre-trained weights (based on ImageNet [119]) and freezing up to different points, e.g. freeze up to layer 10, freeze up to layer 15, etc. We selected the configuration for each model that kept the most layers frozen while meeting the accuracy

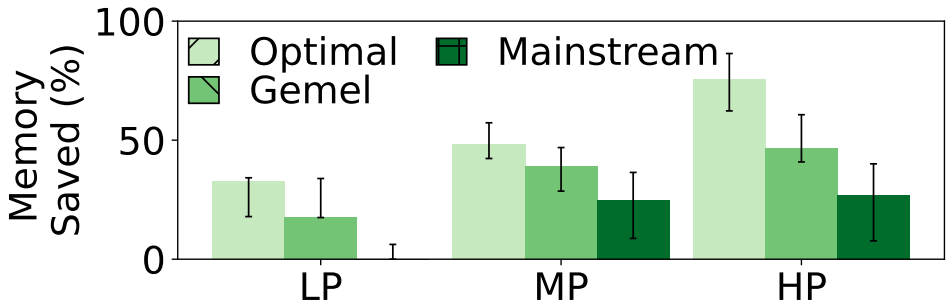


Figure 3.11: Memory savings with GEMEL, an optimal that ignores accuracy, and Mainstream [73]. Bars list the median workload per class, with error bars spanning min to max.

target (95% relative to no freezing). Then, within each workload, we merged all layers that were shared across the frozen layer set of the constituent models (note that these layers have identical weights) and recorded the resultant memory savings.

Figure 3.11 shows our results, from which we draw two conclusions. First, GEMEL's memory savings are within 9.3%, 15.0%, and 29.0% of Optimal for the median LP, MP, and HP workloads. Second, GEMEL's memory reductions are 5.9-52.3% larger than Mainstream's across all workloads. This is a direct consequence of GEMEL's prioritization of memory-heavy layers that routinely appear towards the end of models (§3.5.2). By requiring shared stems from the start of the models, Mainstream would have to share all layers up to the memory-heavy ones; we find that sharing nearly-entire models is rarely possible while meeting accuracy targets (Figure 3.6). The high variance in Mainstream's results are due to the fact that different models drop in accuracy at different rates when more layers are frozen. Classifiers drop relatively slowly (savings up to 70.1%), while detectors are a harder task with faster accuracy drops (Mainstream was unable to share many layers, with savings as low as 1.0%).

3.6.2 Analyzing GEMEL

Incremental memory savings. Key to GEMEL's practicality are its efficient merging heuristic and retraining optimizations that aim to reap memory savings early in the process; indeed, this is important not only to reap accuracy-friendly memory wins quickly, but also to quickly respond to workload changes. As shown in Figure 3.12 (left), 73% of GEMEL's achieved memory savings for the median HP workload are realized within the first 24 minutes

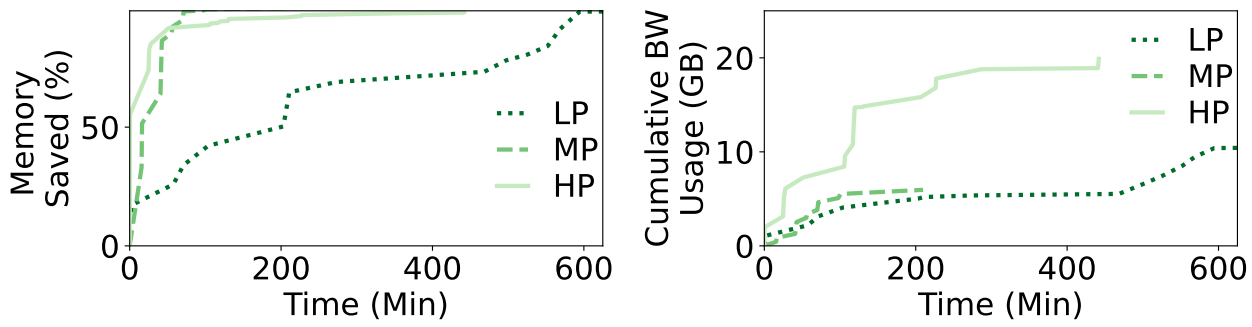


Figure 3.12: GEMEL's memory savings (left) and cloud-to-edge bandwidth usage (right) over time during incremental merging. Results show the median workload per class.

of merging. Similarly, 86% and 64% of the total memory savings are achieved in the first 42 and 210 minutes of merging for median MP and LP workloads, respectively.

Network bandwidth usage. After each successful merging iteration, GEMEL ships weights to edge servers for all updated models. As shown in Figure 3.12 (right), cumulative bandwidth usage during merging is 6.0-19.4 GB for the three workloads. Importantly, bandwidth consumption largely grows after substantial memory savings are already reaped. For example, for the median MP workload, 86% of memory savings are achieved in 42 minutes, while only 2.1 GB (of the total 6.0 GB) of bandwidth is used during that time. The reason is that later merging iterations explore the larger number of lower-memory layers. Thus, GEMEL can often deliver large memory savings even in constrained settings with bandwidth caps. Note that shipping weights uses cloud-to-edge (not precious edge-to-cloud) bandwidth.

Micro-benchmarks. We break down time spent in each component of GEMEL's end-to-end system for our main evaluation workloads.

- *Merging.* The total time spent on merging is configurable, with Figure 3.12 illustrating the memory savings that GEMEL reaps over time for different workloads. Regardless, training dominates the time spent in the overall merging process. Other tasks include (1) finding shared layers, which is done once per workload and takes 0.7-1.4s, or <0.01% of

merging time (we consider a merging time of 6 hours as across workloads, most merging happens by this time), and (2) serializing and saving (to disk) weights after each successful round of training, which takes 9.1-19.5s each time (total of 0.8%-1.44% of merging time).

- *Edge inference.* When using the Nexus variant without any merged models, the time spent blocked while waiting for models to load is 32.8%, 48.3%, and 52.0% at the median of LP, MP, and HP workloads. This time steadily reduces as incremental merging results arrive from GEMEL's cloud merging component, and culminates at 22.1%, 34.6%, and 27.9% when the final merging results arrive. When new merging results arrive, depending on the model size, it takes between 0.03 and 0.58s to load each set of weights into their respective model. However, this time is not blocking, as GEMEL creates a new version of each affected model in CPU with the up-to-date sharing and weights, and substitutes it into the schedule at the next time the corresponding old model is evicted. Therefore, swapping the new model for the old one does not incur any delay.

Varying accuracy, FPS, and SLA. To evaluate the impact of each parameter, we conducted experiments using one randomly selected workload from each class. In each experiment, we only vary one parameter, while keeping the other two at the fixed values from above (95%, 30 FPS, 100 ms).

Figure 3.13 presents our results, which exhibit three trends. First, GEMEL's accuracy wins over time/space-sharing alone grow (by 1.1-7.8% for the three workloads) as accuracy targets drop (from 95% to 80%). This is because certain layers failed to meet 95% during retraining, but did meet a lower accuracy target. Second, GEMEL's accuracy wins drop as input video frame rates (FPS) drop, e.g. from 6.2-42% across the workloads when FPS drops from 30 fps to 5 fps. The reason is that lower FPS values reduce the amount of inference in any time window (assuming a fixed SLA), which in turn adds tolerance to high loading delays. Third, GEMEL's benefits grow as SLAs become stricter: accuracy wins for the three workloads rise by 0.4-2.3% when SLA drops from 400 to 100 ms. This is because tighter SLAs imply more skipped frames for a given swapping delay.

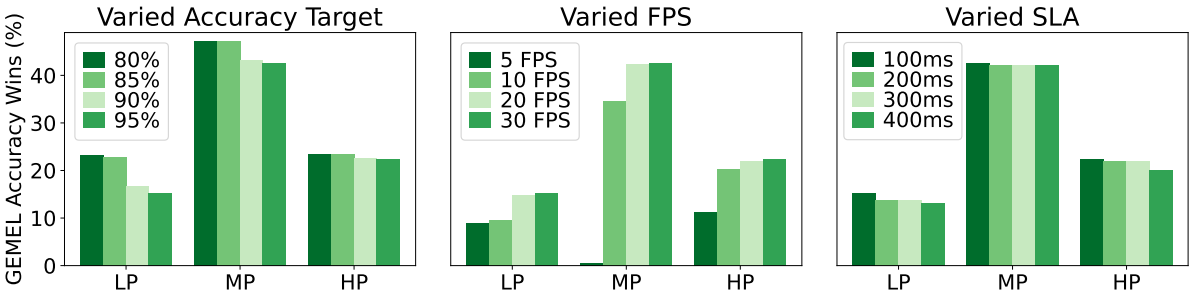


Figure 3.13: GEMEL's accuracy wins (compared to time/space-sharing alone) with varied accuracy targets, FPS, and SLAs.

Comparison to other merging heuristics. We consider variants that differ from GEMEL in one of two ways: they choose layers to merge in a different order or they merge a different number of layers at a time. We describe the variants of each type below, along with the corresponding results. Our experiments use all workloads from §3.2, and we report memory saved over time. Figure 3.14 shows results for two representative workloads (HP3, MP2); the remaining workload results are in §5.7. In summary, no variant consistently outperforms GEMEL, and the degradations (in saved memory or merging delays) that each brings to certain workloads (from being overly aggressive or cautious) are substantial.

Rather than merging layers in descending order of memory usage (irrespective of position) as GEMEL does, the variants we consider start by merging the models' earliest layers (*Earliest*), latest layers (*Latest*), and three random layer orderings (*Random*). Across all workloads, these heuristics all resulted in significantly lower memory savings. Among the three, *Latest* performed the best (median of 13.5% of GEMEL's savings), as memory-heavy layers often appear later in a model (but not necessarily the end). For the same reason, *Earliest* performed the worst (0.2% of GEMEL's savings). *Random*'s performance varied dramatically (0.2% - 72.9%, median of 5.7% of GEMEL's savings) based on whether a memory-heavy layer was selected.

We consider two variants to GEMEL's approach of adding one layer group at a time across all models that layer appears in. First, *TwoGroup* more aggressively adds two groups at a time. This can result in faster memory savings than GEMEL (3/15 workloads, including

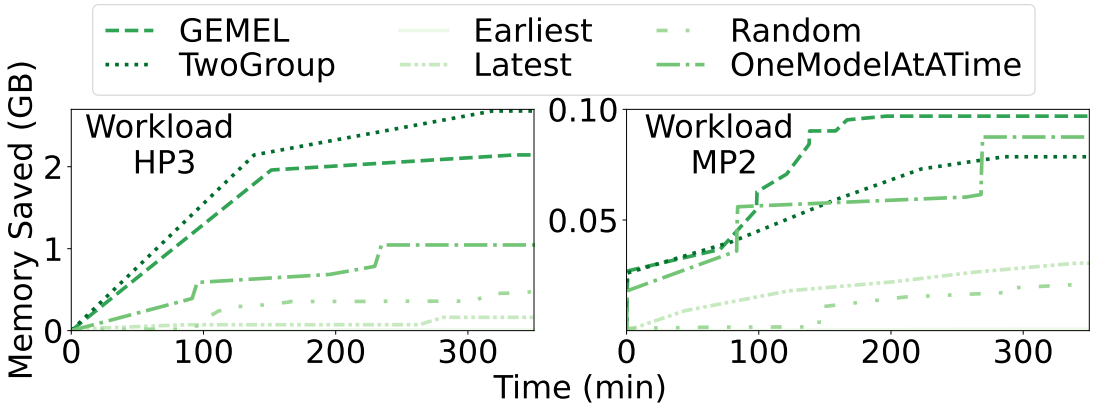


Figure 3.14: Comparing variants of GEMEL's merging heuristic on two representative workloads.

Figure 3.14 (left)), but most often (8/15 workloads) misses accuracy targets and results in substantial slowdowns (78 min longer to max savings for the median workload). The reason is that, on failure, *TwoGroup* restarts training with 1 group, adding long delay without memory savings, e.g., x=75-220 min in Figure 3.14 (right). Second, *OneModelAtATime* less aggressively shares the selected group's layer iteratively across the models it appears in. This reaches within 5% of GEMEL's memory savings in 8/15 workloads, but is often unnecessarily slow, e.g., in Figure 3.14 (left), GEMEL successfully considers 5 models at once, while *OneModelAtATime* individually adds models (some of which fail) leading to the flat stretch from 0-91 min.

3.6.3 Generalization Study

We evaluate GEMEL on over 850 more workloads that extend our main ones by adding: (1) new scene types and the objects they bring (e.g., bags, hats, and people at a beach, boats in a canal), and (2) new models, including more variants in the same families (e.g. ResNet, VGG), and entirely new architectures (e.g. GoogLeNet [132], DenseNet [66]). In total, our analysis involves 17 videos (8 scene types), 13 objects, and 16 models; §5.5 lists the values.

Constructing workloads. Each query in a workload is parameterized by a set of knobs: *camera feed* (and corresponding *scene* type), *model*, and *object of interest*. To study the impact of varying each knob (or combination of knobs) on GEMEL's merging, we construct workloads as follows. For each set of target knobs to vary, we start with a random query

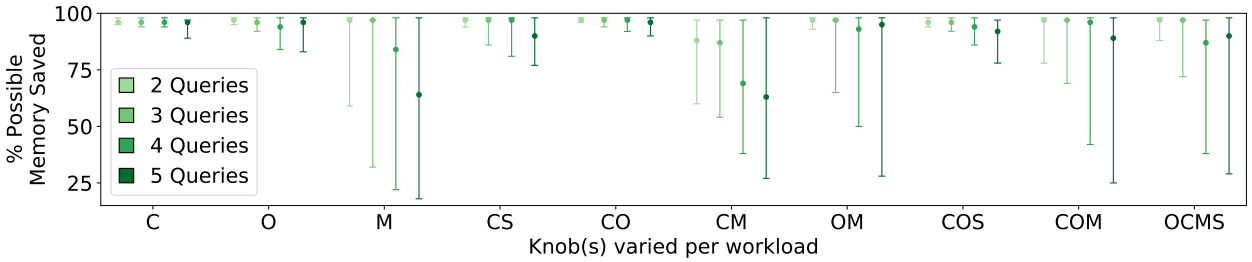


Figure 3.15: Memory savings across 872 workloads, organized by workload size (color) and knobs varied (x-axis). We plot the median of each distribution (error bars spanning 25-75P). Knobs are labeled as follows: C:Camera, O:Object, M:Model, S:Scene.

and incrementally add new queries that only vary values for the target knobs to generate workloads with 2-5 queries each. We did this up to 30 times each for all target knob sets (as their values permit), excluding only (1) target knob sets that vary the *scene* but not *camera* knob, (2) queries for an *object* that never appears in a given *camera* feed, and (3) workloads with no possible memory sharing opportunities.

Findings. As shown in Figure 3.15, GEMEL's memory savings are high for 2-query workloads (89-98% of optimal at medians), but steadily degrade as workloads grow. This is expected as increasing workload size is (by design, and unrealistically) increasing heterogeneity in this experiment. The nature of degradation depends on the knob(s) being varied. For all combinations of {*camera, object, scene*}, degradations are mild moving from 2- to 5-query workloads (0-8%), showing GEMEL's robustness to variations on those properties. Since *model* is constant in these cases, degradations are because the same set of shareable layers must support more diverse scenarios (making it harder to find shared weights).

The *Model* knob (alone or with other knobs) presents a different picture, with larger drops in median memory savings (2-33%) and broader distributions. We can decompose this into two aspects as workload sizes increase:

- Previously-shared layers appear in the new model: the effect on memory savings heavily depends on where the shared layer appears in the new model; recall that layers can appear in different positions (and thus, serve different roles) across models (Figure 5.8). Cases

where the new model introduces drastically different positions for shared layers (e.g., ResNet variants) account for the low-end of the resultant distributions, while memory savings largely persist when positions of shared layer(s) are similar in the new model (e.g., merging across VGG variants).

- New layers are shareable with the new model: the extra sharing opportunities increase potential savings, but are more challenging to realize as they reduce the number of non-shared layers whose weights help compensate for the constraints from sharing (§3.4.2).

3.7 Related Work

Model Sharing. Sharing parts of models has been explored in the ML literature [120], and more recently in video analytics through Mainstream [73], which aims to share outputs from common *stems* of early layers across models. However, unlike GEMEL which addresses memory bottlenecks, the main goal of these works is to reduce computation. This leads to two limitations for our problem. First, these approaches only apply to models that operate on the same underlying data, limiting their applicability to realistic workloads with many videos. Second, and more importantly, because memory-heavy layers are often towards the end of vision DNNs (§3.5.2), stem-sharing approaches would have to share almost all model layers to reap large memory savings – doing so almost always comes with accuracy violations (§3.4.2 and §3.6.1). In contrast, by only sharing weights (not intermediates), GEMEL is able to quickly share only late-stage layers that enable memory savings and accuracy preservation.

PRETZEL [85] focuses on reusing operators in *non-deep* ML models, such as featurizers. The key observation PRETZEL makes is that *both the operator and the parameters are shared* across models, and hence storing them in a shared object store for reuse leads to better memory utilization; the vast majority of savings arise from sharing parameters. However, in edge video analytics, the assumption that shared layers across models have the same parameters (weights) is often violated, as it is common for vision DNNs to be specialized to diverse tasks, objects, and videos (§3.4.2). Indeed, the core challenges that GEMEL tackles are in (quickly) determining which architecturally identical layers consume substantial memory *and* can be retrained to use unified weights without violating accuracy requirements.

Multi-task learning is a well-known technique in machine learning that can learn multiple *related* tasks simultaneously [39]. Some works also study how to share layers in multi-task learning[129, 136]. However, the problem is usually studied in the context of transfer learning, where one model does not have enough data to train on, and instead is trained together with another model; this is in contrast to our setting which considers two sets of pretrained weights that must be shared. Additionally, the related tasks are usually variations of the same model (e.g. spam classification) and thus the data for each individual task can be pooled together. In contrast, objectives vary across edge video analytics DNNs, e.g. detection vs. classification, different objects.

Optimized Video Analytics Pipelines. Many systems aim to lower the resource usage (e.g. computation and bandwidth) of video analytics pipelines. Chameleon [74] profiles pipeline knobs to identify computationally cheaper configurations that preserve accuracy. VideoStorm [158] proposes scheduling techniques that leverage lag tolerance to optimize analytics results. NoScope [76] and Focus [63] build support for low-latency queries on large scale video streams. These systems are complementary to GEMEL, which focuses on alleviating memory (not compute) bottlenecks in edge video analytics. Other systems partially process frames at the edge to reduce both computation and bandwidth costs in video analytics pipelines [38, 45, 70, 112, 144, 161]. Unlike these systems, GEMEL runs inference entirely at the edge.

Previous work has also explored optimizing model serving systems to reduce computation over large and heterogeneous workloads. MCDNN [59] and INFaaS [118] generate model variants with a range of resource profiles and when running inference, dynamically choose which variant to run. While MCDNN generates these variants by compressing models (i.e., retraining), InFaaS uses methods such as pruning and quantization. These methods optimize each model individually; GEMEL supports such variants and provides additional benefits by optimizing *across* models.

Result Reuse. Other systems reduce required model inference by reusing previously computed results, within a query [40, 84, 89] and across queries [44, 55]. Within a query, these systems filter frames that are similar enough to a previously computed frame and reuse results from the previous computation [40, 89]. Across queries, they reuse results when the models and inputs are both the same [44] or the models are the same and the inputs are similar [55]. Less inference leads to a higher tolerance for loading delays, so these systems can alleviate memory pressure, as described in §3.6.2. However, these methods highly depend on how much inference can be avoided, and with spatially correlated inputs (like GEMEL's), all queries typically require high inference loads at the same time (e.g., dynamic, busier scenes). Therefore, reusing results is insufficient to address the memory bottleneck. These approaches could be combined with GEMEL, allowing for lower loading costs and when possible, higher tolerance for those loading costs.

Finally, there exist training optimizations that trade off memory usage for computation overheads [108, 111, 122]. We eschew such techniques given the holistic constraint on compute resources that edge boxes face (§3.1).

3.8 GEMEL Summary

In this chapter, we studied the memory bottleneck of running several DNNs on a single edge box. Due to the limited memory available on edge boxes, fitting multiple DNNs on such boxes quickly exhausts GPU memory, and the existing solution of swapping models in and out of GPU memory results in an unacceptable accuracy drop. We showed that model merging, a new memory management technique, can exploit architectural similarities across vision DNNs by sharing their common layers (including parameters but not intermediates). We then presented GEMEL, a system that efficiently carries out model merging by quickly finding and retraining accuracy-preserving layer sharing configurations, and scheduling edge inference to maximize merging benefits, resulting in an accuracy improvement of 8-39%.

CHAPTER 4

Conclusion & Future Directions

While machine learning models are consistently improving in their accuracy and robustness, their increasing size and complexity cause them to be extremely resource intensive, posing a challenge for practical deployment. A large body of recent work tackles this problem from a systems perspective, treating the machine learning model as one component of a larger system and studying and alleviating system bottlenecks. This thesis focuses on running such machine learning models over live video. Video analytics pipelines to process live video have seen a push towards using edge computing. Among other benefits (e.g., privacy, fault tolerance), this opens up opportunities to improve end-to-end pipeline efficiency.

Most systems assume a fairly well-equipped edge as far as computation and memory. However, in this work, we find a discrepancy between the assumed resources at the edge and the commodity hardware used in practice. Since we do not anticipate cities and enterprises spending considerably more to upgrade all hardware in the near future, we ask how we can make the most of inexpensive hardware at the edge.

We first studied cameras that are used in existing deployments and found that they usually contain a modest CPU and limited memory. Based on these findings, we built Reducto (§2), a system that uses the cheap CPU to compare low-level properties of frames, such as edges, to decide which frames can be filtered out while still meeting accuracy targets.

We then considered an existing camera deployment that used an edge box. This box was equipped with a GPU but lacked sufficient memory to house all models it was intended to service. To tackle this, we looked *inside* the models for redundancies and found that

redundant layers can be merged to lower the memory footprint of a set of models. We built GEMEL, which integrates merging into video analytics pipelines by identifying which layers can be merged while preserving accuracy.

Going forward, as machine learning models get larger, solutions must optimize each part of the video analytics pipeline to keep up with (increasing) demand for processing live video. While Reducto focused on redundancy in the inputs, or the video data itself, and GEMEL (§3) exploited redundancy in the models, we discuss one potential future direction that looks elsewhere in the pipeline.

Constructing Pipelines from Complex Queries. As tasks get more complicated (e.g. identifying abandoned luggage at an airport), queries move from running a single model to running multiple components. For example, the abandoned luggage detector must detect objects that are in the foreground, have been static for a certain amount of time, are not people, and are not within a certain distance of any person. This requires multiple components, such as a foreground detector, an object detector, and concepts such as "within" and "for x sec".

We propose using the semantics of such queries to find and eliminate redundant or unnecessary computation. The first challenge is in building a query language that can incorporate spatio-temporal aspects of video data. This language must be simple enough that someone without a machine learning background can use it, yet robust enough to express concepts such as distance and time. The second challenge is in constructing the pipeline, specifically the order in which to run each component and where each should run (e.g., edge device or cloud). The order of such components is important because each component has a different resource profile and eliminates a certain amount of computation downstream in the pipeline. For example, foreground detection is relatively cheap and if there is nothing in the foreground, the abandoned luggage detector need not do anything else; therefore, it makes sense to go near the beginning of the pipeline. The location of each component is also important as it determines how much information, if any, must be sent to the cloud, which affects the bandwidth needed for the end-to-end pipeline.

The existence of such a query language with automatic pipeline construction could expand the video analytics community by allowing those without an in-depth knowledge of machine learning models to easily write queries. With a better idea of the queries people want to run, we can improve our understanding of the bottlenecks and the optimizations to video analytics pipelines.

Ethical Considerations. The ability to process live video is applicable to many domains, including traffic safety, self-driving cars, security, and surveillance. When this line of work is fully usable and pipelines can process video in real-time, I believe there should be conversations around the ethical use of such technology and the privacy implications. A recent body of work aims to answer queries while preserving the privacy of the individuals in videos. I hope that such measures are considered in policy decisions around this line of work.

CHAPTER 5

Appendix

5.1 Reducto: Correlations

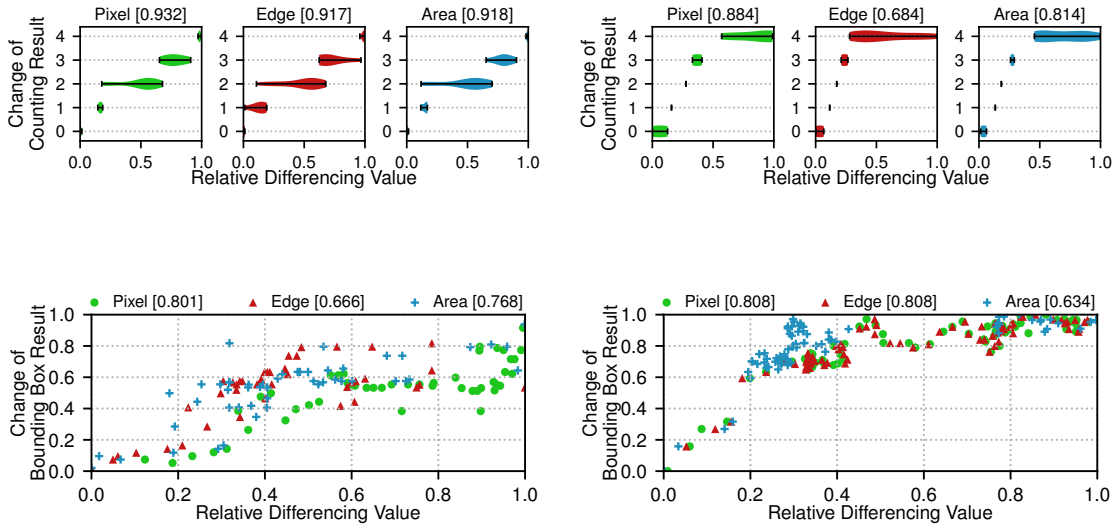


Figure 5.1: Correlations between differencing values and changes in query results (above: counting, below: detection) for a 10 seconds clip in Jacksonhole[7](left) and Southampton[14] (right).

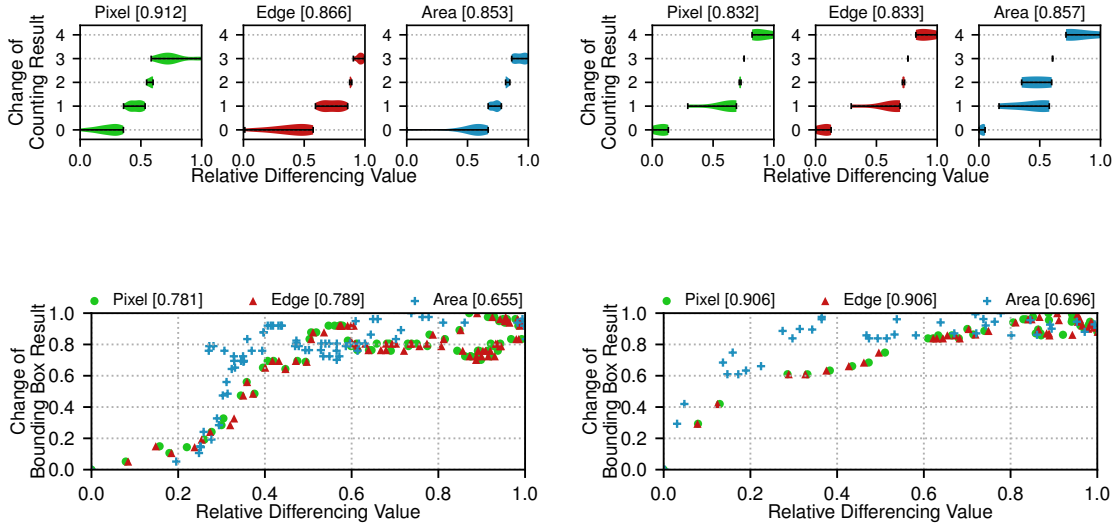


Figure 5.2: Correlations between differencing values and changes in query results (above: counting, below: detection) for a 10 seconds clip in Lagrange[9](left) and Newark[10] (right).

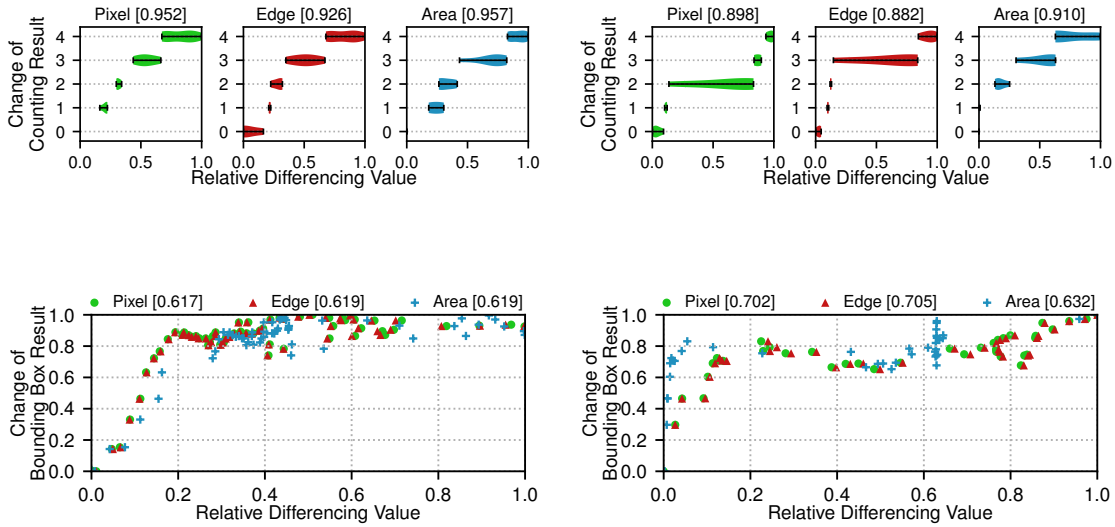
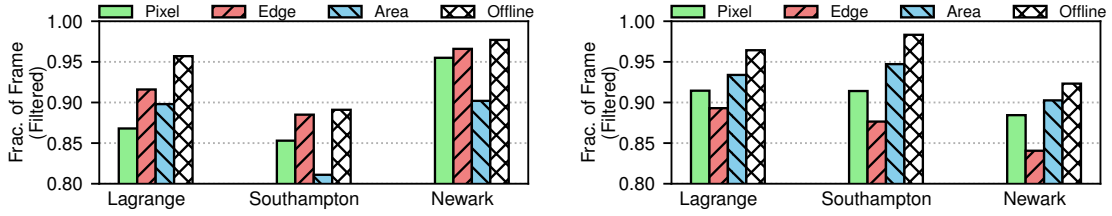


Figure 5.3: Correlations between differencing values and changes in query results (above: counting, below: detection) for a 10 seconds clip in Banff[3](left) and Casa Grande[6] (right).

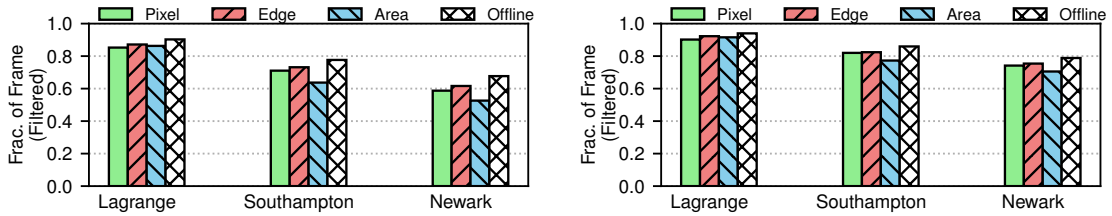
5.2 Reducto: Filtering Efficacy



(a) Query: People bounding box detection

(b) Query: People counting

Figure 5.4: Filtering efficacy of the 3 low-level features across 3 videos and 2 queries. This figure shows that the best feature holds across other objects of interest.



(a) Car bounding box detection with 80% accuracy target

(b) Car bounding box detection with 70% accuracy target

Figure 5.5: Filtering efficacy of the 3 low-level features across 3 videos and 2 queries. This figure shows that the best feature holds across other accuracy targets.

5.3 Reducto: Feature Descriptions

Feature	Description
Area	Calculates the areas of motion in the frame and sends frame if the largest area (as a fraction of total pixels) is above the given threshold.
Pixel	Finds pixels that changed value from the last frame, rounds very small changes to 0, and sends frame if the resulting fraction of changed pixels is above the given threshold.
Edge	Separates the pixels that belong to edges and sends frame if the pixel differences among edge pixels is above the given threshold.
Corner	Detects the pixels that belong to corners and sends frame if the pixel differences among corner pixels is above the given threshold.
SIFT	Detects key points based on contrast, assigns them orientations based on the "neighborhood" of surrounding pixels, and matches them between frames.
SURF	Detects key points using a blob detector, assigns them orientations such that the key points remain if the object is either scaled or rotated, and matches key points between frames.
HOG	Divides frame into small cells, collects a distribution of gradient directions across cells, and compares the distribution across frames.

Table 5.1: Description of differencing features considered in our survey (§2.3).

5.4 GEMEL: Workload Details

Model	Video Feed	Object
frcnn-r101	A1	people
r101	A1	person, car, bus, truck
r50	A2	person, car, bus, truck
r152	A3	person, vehicle
mnet	A4	person, car, truck
yolo	A5	people
tiny-yolo	A1	people
ssd-vgg	A6	cars
ssd-vgg	A1	cars
ssd-mnet	A5	cars
ssd-mnet	A4	cars
ssd-mnet	A6	cars
inception	A3	person, vehicle

Table 5.2: Workload LP1

Model	Video Feed	Object
r152	B1	person, vehicle
r101	B2	person, car, bus, truck
ssd-vgg	B3	people

Table 5.3: Workload LP2

Model	Video Feed	Object
ssd-mnet	B4	cars
frcnn-r101	B3	people
r152	B1	person, vehicle
r18	B3	person, car, bus, truck, motorbike
inception	B1	person, vehicle

Table 5.4: Workload LP3

Model	Video Feed	Object
frcnn-r50	B1	cars
frcnn-r50	B1	people
r50	B2	person, car, bus, truck
r50	B1	person, vehicle
r152	B3	person, car, bus, truck, motorbike
r152	B4	person, car, bus, truck
r18	B5	person, car, bus, truck
r18	B4	person, car, bus, truck
tiny-yolo	B3	cars
tiny-yolo	B2	cars
yolo	B5	cars
yolo	B1	cars
ssd-vgg	B4	cars
ssd-vgg	B3	people
inception	B3	person, car, bus, truck, motorbike

Table 5.5: Workload MP1

Model	Video Feed	Object
r50	B3	person, car, bus, truck, motorbike
r50	B1	person, vehicle
r152	B3	person, car, bus, truck, motorbike
r18	B5	person, car, bus, truck
ssd-mnet	B1	cars
ssd-mnet	B2	cars

Table 5.6: Workload MP2

Model	Video Feed	Object
yolo	B4	cars
yolo	B3	people
tiny-yolo	B1	people
tiny-yolo	B3	cars
ssd-vgg	B1	cars
ssd-mnet	B5	cars

Table 5.7: Workload MP3

Model	Video Feed	Object
yolo	A4	people
yolo	A6	cars
r50	A2	person, car, bus, truck

Table 5.8: Workload MP4

Model	**Video Feed**	**Object**
yolo	A5	people
yolo	A4	people
r152	A1	person, car, bus, truck
r152	A4	person, car, truck
mnet	A4	person, car, truck

Table 5.9: Workload MP5

Model	**Video Feed**	**Object**
frcnn-r50	B5	cars
frcnn-r50	B4	cars
r50	B2	person, car, bus, truck
mnet	B3	person, car, bus, truck, motorbike
tiny-yolo	B3	people

Table 5.10: Workload MP6

Model	**Video Feed**	**Object**
vgg	B4	person, car, bus, truck
vgg	B1	person, vehicle
vgg	B3	person, car, bus, truck, motorbike
vgg	B5	person, car, bus, truck
ssd-vgg	B5	cars
ssd-mnet	B5	cars
mnet	B4	person, car, bus, truck
tiny-yolo	B3	cars
tiny-yolo	B1	people
frcnn-r50	B4	cars
frcnn-r50	B5	cars

Table 5.11: Workload HP1

Model	Video Feed	Object
frcnn-r101	B4	cars
frcnn-r101	B5	cars
frcnn-r101	B1	cars
frcnn-r101	B2	cars
frcnn-r50	B1	people
r50	B3	person, car, bus, truck, motorbike
r18	B3	person, car, bus, truck, motorbike
ssd-mnet	B3	people
ssd-mnet	B1	people
mnet	B4	person, car, bus, truck
yolo	B3	people
tiny-yolo	B5	cars
tiny-yolo	B1	people
vgg	B4	person, car, bus, truck
inception	B2	person, car, bus, truck
inception	B3	person, car, bus, truck, motorbike

Table 5.12: Workload HP2

Model	Video Feed	Object
frcnn-r50	A3	cars
frcnn-r50	A3	people
frcnn-r50	A1	cars
frcnn-r50	A1	people
frcnn-r50	A5	cars
frcnn-r50	A5	people
frcnn-r50	A2	cars
frcnn-r50	A4	cars
frcnn-r50	A2	trucks
frcnn-r101	A3	people
yolo	A3	cars
yolo	A3	people
yolo	A1	people
yolo	A7	buses
yolo	A7	cars
yolo	A7	people
yolo	A7	trucks
yolo	A5	trucks
yolo	A5	people
yolo	A6	cars

Table 5.13: Workload HP3

Model	Video Feed	Object
r152	A3	person, vehicle
r152	A1	person, car, bus, truck
r152	A7	person, car, bus, truck
r152	A6	car, bus, truck
r152	A2	person, car, bus, truck
r152	A4	person, car, truck
r50	A3	person, vehicle
r50	A7	person, car, bus, truck
r50	A6	car, bus, truck
r50	A2	person, car, bus, truck
r50	A6	car, bus, truck
ssd-vgg	A3	people
ssd-vgg	A1	cars
ssd-vgg	A5	people
ssd-vgg	A6	cars
ssd-vgg	A4	cars
vgg	A2	person, car, bus, truck
r18	A2	person, car, bus, truck

Table 5.14: Workload HP3 (continued)

Model	Video Feed	Object
yolo	B1	cars
yolo	B5	cars
tiny-yolo	B2	cars
tiny-yolo	B1	cars
tiny-yolo	B3	people
ssd-vgg	B5	cars
ssd-vgg	B3	people
ssd-mnet	B5	cars
ssd-mnet	B3	people
ssd-mnet	B2	cars
ssd-mnet	B1	people
mnet	B3	person, car, bus, truck, motorbike
mnet	B5	person, car, bus, truck
r152	B4	person, car, bus, truck
r152	B3	person, car, bus, truck, motorbike
r152	B1	person, vehicle

Table 5.15: Workload HP4

Model	Video Feed	Object
frcnn-r50	A3	cars
frcnn-r50	A3	people
frcnn-r50	A1	cars
frcnn-r50	A1	people
frcnn-r50	A5	cars
frcnn-r50	A5	people
frcnn-r50	A2	cars
frcnn-r50	A4	cars
frcnn-r50	A2	trucks
frcnn-r101	A3	people
yolo	A3	cars
yolo	A3	people
yolo	A1	people
yolo	A7	buses
yolo	A7	cars
yolo	A7	people
yolo	A7	trucks
yolo	A5	trucks
yolo	A5	people
yolo	A6	cars

Table 5.16: Workload HP5

Model	Video Feed	Object
r152	A3	person, vehicle
r152	A1	person, car, bus, truck
r152	A7	person, car, bus, truck
r152	A6	car, bus, truck
r152	A2	person, car, bus, truck
r152	A4	person, car, truck
r50	A3	person, vehicle
r50	A7	person, car, bus, truck
r50	A6	car, bus, truck
r50	A2	person, car, bus, truck
r50	A6	car, bus, truck
ssd-vgg	A3	people
inception	A3	person, vehicle
inception	A1	person, car, bus, truck
inception	A7	person, car, bus, truck
inception	A6	car, bus, truck
inception	A4	person, car, truck
vgg	A2	person, car, bus, truck
r18	A2	person, car, bus, truck
r18	A2	person, car, bus, truck
r18	A2	person, car, bus, truck

Table 5.17: Workload HP5 (continued)

Model	Video Feed	Object
frcnn-r50	A3	cars
frcnn-r50	A3	people
frcnn-r50	A1	cars
frcnn-r50	A1	people
frcnn-r50	A5	cars
frcnn-r50	A5	people
frcnn-r50	A2	cars
frcnn-r50	A4	cars
frcnn-r50	A2	trucks
frcnn-r101	A3	people
yolo	A3	cars
yolo	A3	people
yolo	A1	people
yolo	A7	buses
yolo	A7	cars
yolo	A7	people
r101	A1	person, car, bus, truck
r101	A7	person, car, bus, truck
r101	A6	car, bus, truck

Table 5.18: Workload HP6

Model	Video Feed	Object
r101	A1	person, car, bus, truck
r152	A3	person, vehicle
r152	A1	person, car, bus, truck
r152	A7	person, car, bus, truck
r152	A6	car, bus, truck
r152	A2	person, car, bus, truck
r152	A4	person, car, truck
r50	A3	person, vehicle
r50	A7	person, car, bus, truck
r50	A6	car, bus, truck
r50	A2	person, car, bus, truck
r50	A6	car, bus, truck
tiny-yolo	A1	people
tiny-yolo	A5	people
inception	A3	person, vehicle
inception	A1	person, car, bus, truck
inception	A7	person, car, bus, truck
inception	A6	car, bus, truck
inception	A4	person, car, truck
vgg	A2	person, car, bus, truck
r18	A2	person, car, bus, truck
r18	A2	person, car, bus, truck
r18	A2	person, car, bus, truck

Table 5.19: Workload HP6 (continued)

5.5 GEMEL: Generalization Workload Query Knobs

Knob	Values
Object	Truck, Person, Bus, Boat, Shoe, Skateboard, Car, Hat, Backpack, Wine Glass, Traffic Light, Parking Meter, Surfboard
Camera	A0, A1, A2, A3, B0, B1, B2, B3, B4, B5, B6, Restaurant, Mall, Beach, Canal, Parking Lot, Street
Model	SSD-VGG, AlexNet, YOLOv3, Tiny-YOLOv3, DenseNet, SqueezeNet, GoogLeNet, ResNet-18, ResNet-34, ResNet-50, ResNet-101, ResNet-152, VGG-11, VGG-13, VGG-16, VGG-19
Scene	CityA Traffic, CityB Traffic, Restaurant, Beach, Mall, Canal, Parking Lot, Street

Table 5.20: Knob values considered in generalization study.

5.6 GEMEL: Workload Memory Settings

Workload	L1	L2	L3	M1	M2	M3	M4	M5	M6
Min	4.50	1.45	4.50	3.35	1.45	1.32	1.32	1.45	3.35
50%	5.12	1.59	4.72	4.56	1.62	1.55	1.45	1.83	3.77
75%	5.43	1.66	4.83	5.16	1.70	1.65	1.52	2.02	3.99

Table 5.21: Edge box memory settings for LP and MP workloads (in GB).

Workload	H1	H2	H3	H4	H5	H6
Min	3.35	4.50	4.50	1.45	4.50	4.50
50%	4.87	6.60	10.25	2.17	10.41	10.26
75%	5.63	7.66	13.13	2.53	13.36	13.14

Table 5.22: Edge box memory settings for HP workloads (in GB).

5.7 GEMEL: Additional Figures

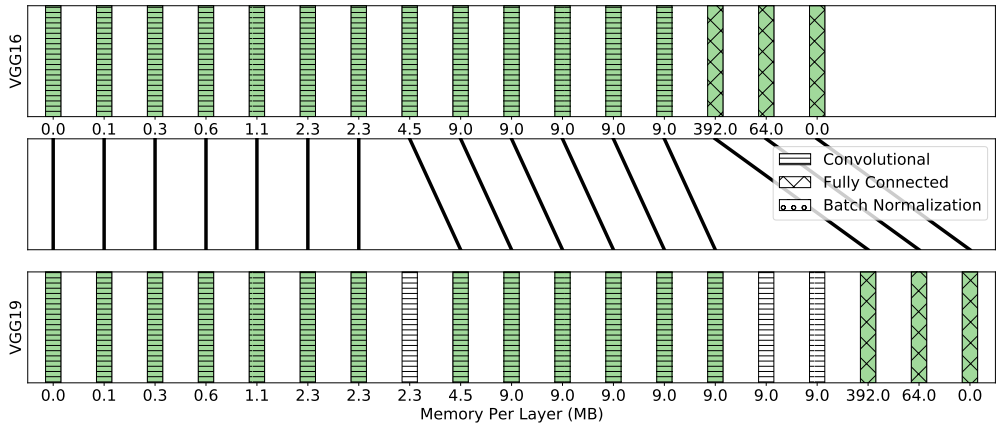


Figure 5.6: VGG16 and VGG19 are variants within the VGG model family [126]. They share 16/19 layers (13 convolutional and 3 fully-connected). Note that 'batch normalization' layers are not present in these models.

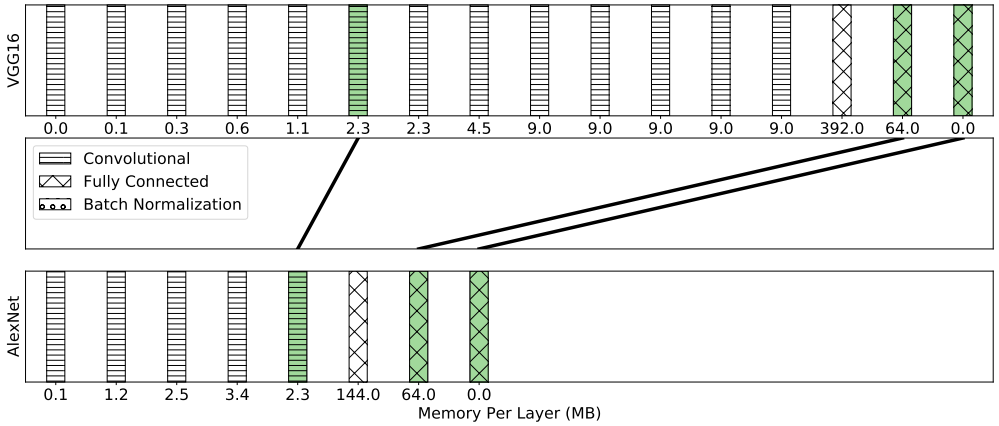


Figure 5.7: VGG [126] was developed by replacing AlexNet's [81] large kernels with multiple smaller ones. VGG16 and AlexNet share 3/16 layers (1 convolutional and 2 fully-connected). Note that 'batch normalization' layers are not present in these models.

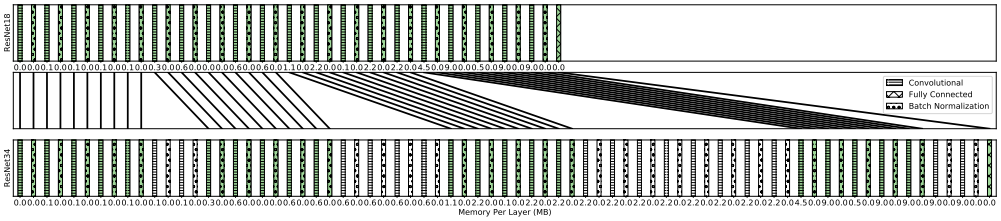


Figure 5.8: ResNet18 and ResNet34 are variants within the ResNet model family [60]. They share 41/73 layers (20 convolutional, 1 fully-connected and 20 batch normalization).

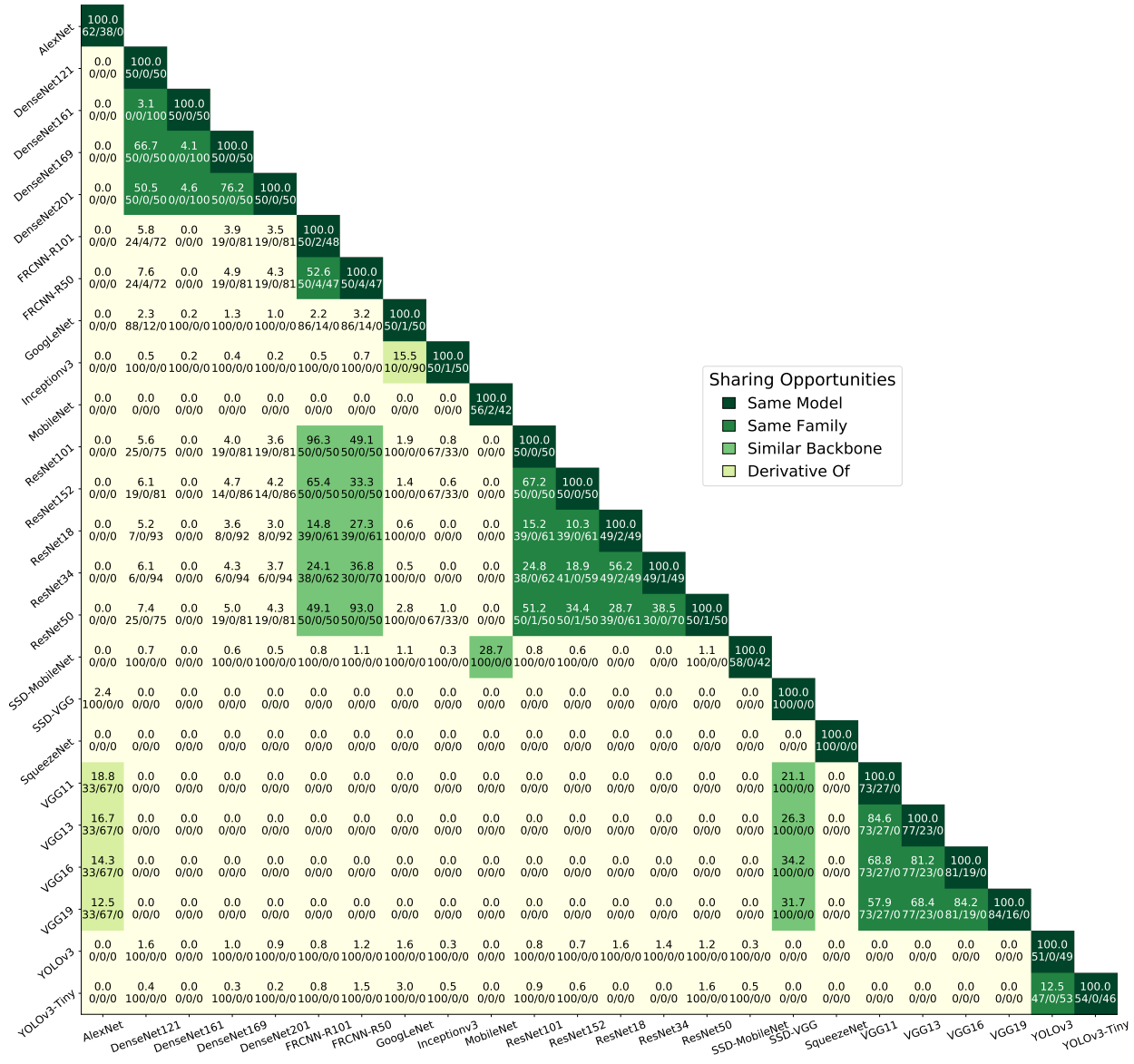


Figure 5.9: Extended version of Figure 3.3. For each unique pair of models, we show the percentage of architecturally identical layers and of those layers, the percent breakdown across layer types (%Convolutional / %Linear / %BatchNorm).

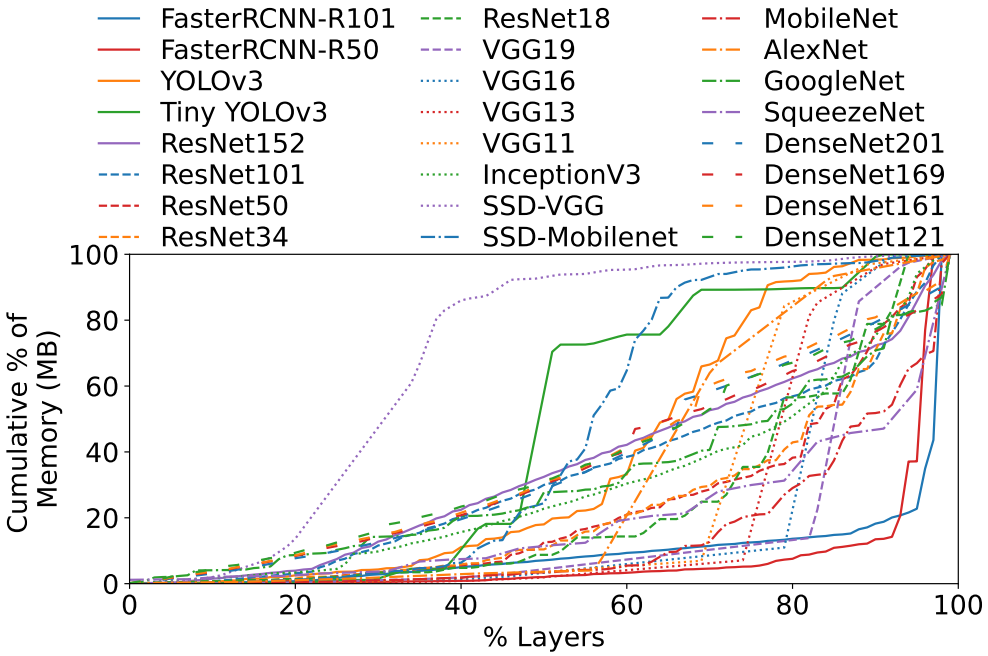


Figure 5.10: Extended version of Figure 3.8. Cumulative memory consumed by each model's layer groups moving from start to end of the model.

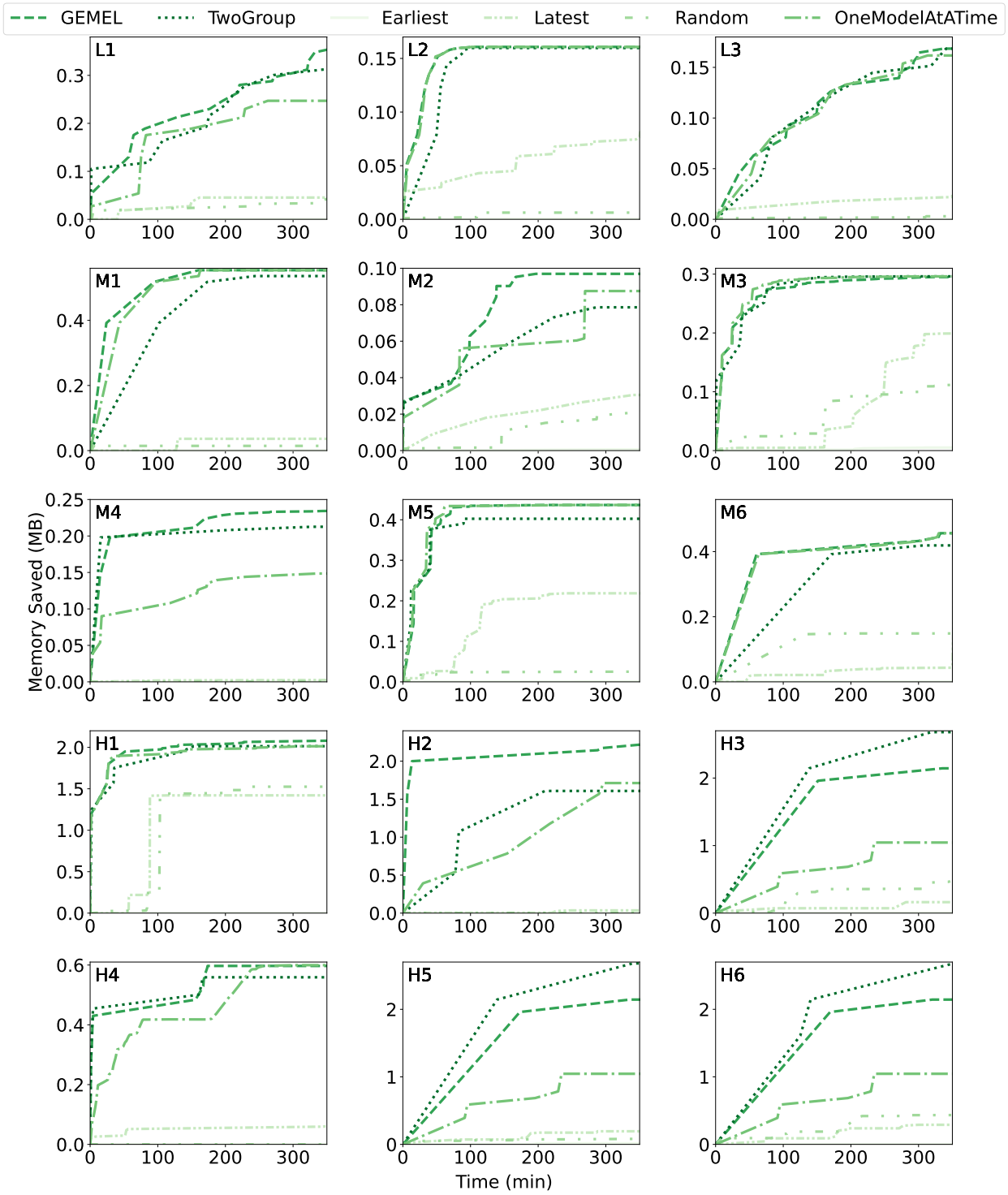


Figure 5.11: Complete version of Figure 3.14. Comparison of GEMEL with other merging heuristics.

Bibliography

[1] AWS Outposts. https://aws.amazon.com/outposts/.

[2] Azure Stack Edge. https://azure.microsoft.com/en-us/products/azure-stack/edge/.

[3] Banff Live Cam, Alberta, Canada. https://www.youtube.com/watch?v=9HwSNgcdQ7k.

[4] City of Auburn Toomer's Corner Webcam. https://www.youtube.com/watch?v=hMYIc5ZPJL4.

[5] Edge computing at chick-fil-a. https://medium.com/@cfatechblog/edge-computing-at-chick-fil-a-7d67242675e2.

[6] Gebhardt Insurance Traffic Cam Round Trip Bike Shop. https://www.youtube.com/watch?v=RNi4CKgZVMY.

[7] Jackson Hole Wyoming USA Town Square Live Cam. https://www.youtube.com/watch?v=1EiC9bvVGnk.

[8] JeVois Smart Machine Vision Camera. http://jevois.org.

[9] La Grange, Kentucky USA - Virtual Railfan LIVE. https://www.youtube.com/watch?v=pJ5cg83D5AE.

[10] Newark Police Citizen Virtual Patrol. https://cvp.newarkpublicsafety.org.

[11] NVIDIA Jetson: The AI platform for edge computing. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/.

[12] NVIDIA Multi-Instance GPU . https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[13] Raspberry Pi Zero. https://www.raspberrypi.org/products/raspberry-pi-zero.

[14] TwinForksPestControl.com SOUTHAMPTON TRAFFIC CAM. https://www.youtube.com/watch?v=y3NOhpkoR-w.

[15] The vision zero initiative. http://www.visionzeroinitiative.com/.

[16] DNNCam^{TM} AI camera. https://groupgets.com/campaigns/429-dnncam-ai-camera, 2019.

[17] Open source computer vision library. https://https://opencv.org, 2020.

[18] Cuda multi-process service, April 2021.

[19] Live Video Analytics with Microsoft Rocket for reducing edge compute costs, May 2021.

[20] Microsoft rocket video analytics platform, April 2021.

[21] NVIDIA TensorRT, April 2021.

[22] Pytorch, April 2021.

[23] Traffic Video Analytics – Case Study Report, May 2021.

[24] R. B. , Z. Xia, G. Ananthanarayanan, J. Jiang, Y. Shu, N. Karianakis, K. Hsieh, V. Bahl, and I. Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *USENIX NSDI*, April 2022.

[25] M. Alam, M. Samad, L. Vidyaratne, A. Glandon, and K. Iftekharuddin. Survey on deep neural networks in speech and vision systems. *Neurocomputing*, 417:302–321, 2020.

[26] Z. Allen-Zhu, Y. Li, and Y. Liang. Learning and generalization in overparameterized neural networks, going beyond two layers. *CoRR*, abs/1811.04918, 2018.

[27] Amazon. Rekognition. https://aws.amazon.com/rekognition/.

[28] Amazon. AWS DeepLens. https://aws.amazon.com/deeplens/, 2019.

[29] Ambarella. CV22 - Computer Vision SoC for Consumer Cameras. https://www.ambarella.com/wp-content/uploads/CV22-product-brief-consumer.pdf.

[30] G. Ananthanarayanan, V. Bahl, L. Cox, A. Crown, S. Nogbahi, and Y. Shu. Video analytics - killer app for edge computing. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, pages 695–696, New York, NY, USA, 2019. Association for Computing Machinery.

[31] AXIS. Axis for a safety touch at the Grey Cup Festival. https://www.axis.com/files/success_stories/ss_stad_greycup_festival_58769_en_1407_lo.pdf.

[32] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514. USENIX Association, Nov. 2020.

[33] S. Bhardwaj, M. Srinivasan, and M. M. Khapra. Efficient video classification using fewer frames. *CoRR*, abs/1902.10640, 2019.

[34] D. Brezeale and D. J. Cook. Automatic video classification: A survey of the literature. *Trans. Sys. Man Cyber Part C*, 38(3):416–430, May 2008.

[35] S. Brutzer, B. Hoferlin, and G. Heidemann. Evaluation of background subtraction techniques for video surveillance. In *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '11, pages 1937–1944, Washington, DC, USA, 2011. IEEE Computer Society.

[36] N. Buch, S. A. Velastin, and J. Orwell. A review of computer vision techniques for the analysis of urban traffic. *Trans. Intell. Transport. Sys.*, 12(3):920–939, Sept. 2011.

[37] Z. Cai, M. Saberian, and N. Vasconcelos. Learning complexity-aware cascades for deep pedestrian detection. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, pages 3361–3369, Washington, DC, USA, 2015. IEEE Computer Society.

[38] 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. In *2nd SysML Conference*, 2019.

[39] R. Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997.

[40] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168, 2015.

[41] S. P. Chinchali, E. Cidon, E. Pergament, T. Chu, and S. Katti. Neural networks meet physical networks: Distributed inference between edge devices and the cloud. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, pages 50–56, New York, NY, USA, 2018. Association for Computing Machinery.

[42] Chong-Wah Ngo, Yu-Fei Ma, and Hong-Jiang Zhang. Video summarization and scene detection by graph modeling. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(2):296–305, Feb 2005.

[43] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. OSDI, 2014.

[44] 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, Boston, MA, Mar. 2017. USENIX Association.

[45] K. Du, A. Pervaiz, X. Yuan, A. Chowdhery, Q. Zhang, H. Hoffmann, and J. Jiang. Server-driven video streaming for deep learning inference. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 557–570, New York, NY, USA, 2020. Association for Computing Machinery.

[46] J. Emmons, S. Fouladi, G. Ananthanarayanan, S. Venkataraman, S. Savarese, and K. Winstein. Cracking open the dnn black-box: Video analytics with dnns across the camera-cloud boundary. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, HotEdgeVideo'19, pages 27–32, New York, NY, USA, 2019. Association for Computing Machinery.

[47] M. Everingham, L. Gool, C. K. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *Int. J. Comput. Vision*, 88(2):303–338, June 2010.

[48] H. Fan, Z. Xu, L. Zhu, C. Yan, J. Ge, and Y. Yang. Watching a small portion could be as good as watching all: Towards efficient video classification. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 705–711. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[49] S. Gammeter, A. Gassmann, L. Bossard, T. Quack, and L. Van Gool. Server-side object recognition and client-side object tracking for mobile augmented reality. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*, pages 1–8, June 2010.

[50] A. K. Ghosh. On optimum choice of k in nearest neighbor classification. *Computational Statistics & Data Analysis*, 50(11):3113–3123, 2006.

[51] Google. Google edge network. https://peering.google.com/#/infrastructure, 2016.

[52] Google. Cloud vision api. https://cloud.google.com/vision, 2021.

[53] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, Nov. 2020.

[54] P. Guo, B. Hu, and W. Hu. Mistify: Automating DNN model porting for on-device inference at the edge. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 705–719. USENIX Association, Apr. 2021.

[55] P. Guo and W. Hu. Potluck: Cross-application approximate deduplication for computation-intensive mobile applications. *SIGPLAN Not.*, 53(2):271–284, mar 2018.

[56] HAILO. Edge AI Box. https://hailo.ai/reference-platform/edge-ai-box/, 2021.

[57] P. Hall, B. U. Park, R. J. Samworth, et al. Choice of neighbor order in nearest-neighbor classification. *The Annals of Statistics*, 36(5):2135–2152, 2008.

[58] B. Han, F. Qian, L. Ji, and V. Gopalakrishnan. Mp-dash: Adaptive video streaming over preference-aware multipath. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 129–143, New York, NY, USA, 2016. ACM.

[59] 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*, MobiSys '16, pages 123–136, New York, NY, USA, 2016. ACM.

[60] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

[61] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[62] 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*, abs/1704.04861, 2017.

[63] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 269–286, Carlsbad, CA, 2018. USENIX Association.

[64] W. Hu, N. Xie, Li, X. Zeng, and S. Maybank. A survey on visual content-based video indexing and retrieval. *Trans. Sys. Man Cyber Part C*, 41(6):797–819, Nov. 2011.

[65] C.-C. Huang, G. Jin, and J. Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1341–1355, New York, NY, USA, 2020. Association for Computing Machinery.

[66] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. Densely connected convolutional networks, 2016.

[67] J. Hui. Object detection: speed and accuracy comparison (Faster R-CNN, R-FCN, SSD, FPN, RetinaNet and YOLOv3). https://jonathan-hui.medium.com/object-detection-speed-and-accuracy-comparison-faster-r-cnn-r-fcn-ssd-and-yolo-3 2018.

[68] 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, Oct 2018.

[69] IBM. Maximo remote monitoring. https://www.ibm.com/products/maximo/remote-monitoring, 2021.

[70] S. Jain, X. Zhang, Y. Zhou, G. Ananthanarayanan, J. Jiang, Y. Shu, V. Bahl, and J. Gonzalez. Spatula: Efficient cross-camera video analytics on large camera networks. In *ACM/IEEE Symposium on Edge Computing (SEC 2020)*, November 2020.

[71] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association.

[72] M. Jeon, S. Venkataraman, J. Qian, A. Phanishayee, W. Xiao, and F. Yang. Multi-tenant gpu clusters for deep learning workloads: Analysis and implications. *Technical report, Microsoft Research*, 2018.

[73] A. H. Jiang, D. L.-K. Wong, C. Canel, L. Tang, I. Misra, M. Kaminsky, M. A. Kozuch, P. Pillai, D. G. Andersen, and G. R. Ganger. Mainstream: Dynamic stem-sharing for multi-tenant video processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 29–42, Boston, MA, July 2018. USENIX Association.

[74] 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*, SIGCOMM '18, pages 253–266, New York, NY, USA, 2018. ACM.

[75] D. Kang, P. Bailis, and M. Zaharia. Blazeit: Fast exploratory video queries using neural networks. *CoRR*, abs/1805.01046, 2018.

[76] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: Optimizing neural network queries over video at scale. *Proc. VLDB Endow.*, 10(11):1586–1597, Aug. 2017.

[77] K. Kawaguchi, J. Huang, and L. P. Kaelbling. Every local minimum value is the global minimum value of induced model in nonconvex machine learning. *Neural Computation*, 31(12):2293–2323, Dec 2019.

[78] K. Kawaguchi and L. P. Kaelbling. Elimination of all bad local minima in deep learning. *CoRR*, abs/1901.00279, 2019.

[79] S. H. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan, and M. Shah. Transformers in vision: A survey. *CoRR*, abs/2101.01169, 2021.

[80] H. Kim, S. Leutenegger, and A. J. Davison. Real-time 3D reconstruction and 6-DoF tracking with an event camera. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VI*, pages 349–364, 2016.

[81] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, may 2017.

[82] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[83] B. Kueng, E. Mueggler, G. Gallego, and D. Scaramuzza. Low-latency visual odometry using event-based feature tracks. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 16–23, Oct 2016.

[84] A. Kumar, A. Balasubramanian, S. Venkataraman, and A. Akella. Accelerating deep learning inference via freezing. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[85] 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, Carlsbad, CA, Oct. 2018. USENIX Association.

[86] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua. A convolutional neural network cascade for face detection. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5325–5334, June 2015.

[87] Y. Li, E. Agustsson, S. Gu, R. Timofte, and L. Van Gool. Carn: Convolutional anchored regression network for fast and accurate single image super-resolution. In *The European Conference on Computer Vision (ECCV) Workshops*, September 2018.

[88] Y. Li and Y. Liang. Learning overparameterized neural networks via stochastic gradient descent on structured data. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 8168–8177, Red Hook, NY, USA, 2018. Curran Associates Inc.

[89] Y. Li, A. Padmanabhan, P. Zhao, Y. Wang, G. H. Xu, and R. Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 359–376, New York, NY, USA, 2020. Association for Computing Machinery.

[90] Z. Li, Y. Shu, G. Ananthanarayanan, L. Shangguan, K. Jamieson, and V. Bahl. Spider: A multi-hop millimeter-wave network for live video analytics. In *ACM/IEEE Symposium on Edge Computing*. ACM/IEEE, December 2021.

[91] T. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 936–944, July 2017.

[92] H. Liu, K. Simonyan, and Y. Yang. DARTS: differentiable architecture search. *CoRR*, abs/1806.09055, 2018.

[93] X. Liu, P. Ghosh, O. Ulutan, B. S. Manjunath, K. Chan, and R. Govindan. Caesar: Cross-camera complex activity recognition. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, SenSys '19, page 232–244. Association for Computing Machinery, 2019.

[94] Y. Lu, A. Chowdhery, and S. Kandula. Optasia: A relational platform for efficient large-scale video analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 57–70, New York, NY, USA, 2016. ACM.

[95] M5STACK. K210 RISC-V 64 AI Camera. https://m5stack.com/blogs/news/introducing-the-k210-risc-v-ai-camera-m5stickv, 2019.

[96] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys Tutorials*, 19(4):2322–2358, Fourthquarter 2017.

[97] I. Markit. Ihs markit's top video surveillance trends for 2018. https://cdn.ihs.com/www/pdf/Top-Video-Surveillance-Trends-2018.pdf, 2018.

[98] Microsoft. Microsoft azure data box. https://azure.microsoft.com/en-us/services/databox/, 2019.

[99] Microsoft. Enabling Data Residency and Data Protection in Microsoft Azure Regions. https://azure.microsoft.com/en-us/resources/achieving-compliant-data-residency-and-security-with-azure/, 2021.

[100] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. Proceedings of ATC '15. USENIX, 2015.

[101] S. A. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan. The emerging landscape of edge computing. *GetMobile: Mobile Comp. and Comm.*, 23(4):11–20, May 2020.

[102] E. Nygren, R. K. Sitaraman, and J. Sun. The akamai network: A platform for high-performance internet applications. *SIGOPS*, 2010.

[103] OfCom. Residential landline and fixed broadband services. https://www.ofcom.org.uk/__data/assets/pdf_file/0015/113640/landline-broadband.pdf, 2017.

[104] 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, 2017.

[105] M. Otani, Y. Nakashima, E. Rahtu, and J. Heikkilä. Rethinking the evaluation of video summaries. *CoRR*, abs/1903.11328, 2019.

[106] A. Padmanabhan, N. Agarwal, A. Iyer, G. Ananthanarayanan, Y. Shu, N. Karianakis, G. H. Xu, and R. Netravali. Gemel: Model merging for memory-efficient, real-time video analytics at the edge, 2022.

[107] C. Pakha, A. Chowdhery, and J. Jiang. Reinventing video streaming for distributed vision analytics. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA, July 2018. USENIX Association.

[108] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 891–905. Association for Computing Machinery, 2020.

[109] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient neural architecture search via parameter sharing. *CoRR*, abs/1802.03268, 2018.

[110] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa. Visor: Privacy-preserving video analytics as a cloud service. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1039–1056. USENIX Association, Aug. 2020.

[111] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He. Zero-infinity: Breaking the GPU memory wall for extreme scale deep learning. *CoRR*, abs/2104.07857, 2021.

[112] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 1421–1429, April 2018.

[113] H. Rebecq, T. Horstschaefer, G. Gallego, and D. Scaramuzza. EVO: A geometric approach to event-based 6-dof parallel tracking and mapping in real time. *IEEE Robotics and Automation Letters*, 2(2):593–600, 2017.

[114] H. Rebecq, T. Horstschaefer, and D. Scaramuzza. Real-time visual-inertial odometry for event cameras using keyframe-based nonlinear optimization. In *British Machine Vision Conference 2017, BMVC 2017, London, UK, September 4-7, 2017*, 2017.

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

[116] J. Redmon and A. Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[117] Y. Ren, F. Zeng, W. Li, and L. Meng. A low-cost edge server placement strategy in wireless metropolitan area networks. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6, July 2018.

[118] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.

[119] 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. *Int. J. Comput. Vision*, 115(3):211–252, Dec. 2015.

[120] S. S. Sarwar, A. Ankit, and K. Roy. Incremental learning in deep convolutional neural networks using partial network sharing. *IEEE Access*, 8:4615–4628, 2019.

[121] J. Sevilla, P. Villalobos, and J. Cerón. Parameter counts in Machine Learning. https://www.lesswrong.com/posts/GzoWcYibWYwJva8aL/parameter-counts-in-machine-learning, 2021.

[122] A. Shah, C. Wu, J. Mohan, V. Chidambaram, and P. Krähenbühl. Memory optimization for deep networks. *CoRR*, abs/2010.14501, 2020.

[123] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis.

In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[124] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[125] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[126] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[127] Sony. REA-C1000 Edge Analytics Appliance. https://pro.sony/ue_US/products/ptz-cameras/rea-c1000-edge-analytics-appliance, 2021.

[128] F. Sultana, A. Sufian, and P. Dutta. Evolution of image segmentation using deep convolutional neural network: A survey. *Knowledge-Based Systems*, 201-202:106062, 2020.

[129] X. Sun, R. Panda, R. Feris, and K. Saenko. Adashare: Learning what to share for efficient deep multi-task learning. *arXiv preprint arXiv:1911.12423*, 2019.

[130] Y. Sun, X. Wang, and X. Tang. Deep convolutional network cascade for facial point detection. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '13, pages 3476–3483, Washington, DC, USA, 2013. IEEE Computer Society.

[131] A. Suprem, J. Arulraj, C. Pu, and J. Ferreira. Odin: Automated drift detection and recovery in video analytics. *Proc. VLDB Endow.*, 13(12):2453–2465, July 2020.

[132] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions, 2014.

[133] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision, 2015.

[134] Z. Tang, G. Wang, H. Xiao, A. Zheng, and J. Hwang. Single-camera and inter-camera vehicle tracking and 3d speed estimation based on fusion of visual and semantic features. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 108–115, June 2018.

[135] Tencent. DeepGaze AI Camera. https://open.youtu.qq.com/#/open/solution/hardware-ai.

[136] S. Vandenhende, S. Georgoulis, B. De Brabandere, and L. Van Gool. Branched multi-task networks: deciding what layers to share. *arXiv preprint arXiv:1904.02920*, 2019.

[137] L. M. Vaquero and L. Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *CCR*, 44(5):27–32, Oct. 2014.

[138] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[139] A. R. Vidal, H. Rebecq, T. Horstschaefer, and D. Scaramuzza. Ultimate slam? combining events, images, and IMU for robust visual SLAM in HDR and high-speed scenarios. *IEEE Robotics and Automation Letters*, 3(2):994–1001, 2018.

[140] J. Wang, Z. Feng, Z. Chen, S. George, M. Bala, P. Pillai, S.-W. Yang, and M. Satyanarayanan. Bandwidth-efficient live video analytics for drones via edge computing. pages 159–173, 10 2018.

[141] 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, SEC '19, page 152–165, New York, NY, USA, 2019. Association for Computing Machinery.

[142] M. Wang, C. Meng, G. Long, C. Wu, J. Yang, W. Lin, and Y. Jia. Characterizing deep learning training workloads on alibaba-pai, 2019.

[143] S. Wang, H. Lu, P. Dmitriev, and Z. Deng. Fast object detection in compressed video. *CoRR*, abs/1811.11057, 2018.

[144] Y. Wang, W. Wang, J. Zhang, J. Jiang, and K. Chen. Bridging the edge-cloud barrier for real-time advanced vision analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[145] P. N. Whatmough, C. Zhou, P. Hansen, S. K. Venkataramanaiah, J. Seo, and M. Mattina. FixyNN: Efficient hardware for mobile computer vision via transfer learning. *CoRR*, abs/1902.11128, 2019.

[146] Wi4Net. Axis is on the case in downtown Huntington Beach. http://www.wi4net.com/Resources/Pdfs/huntington %20beach%20case_study%5BUS%5Dprint.pdf.

[147] C. Wu, C. Feichtenhofer, H. Fan, K. He, P. Krähenbühl, and R. B. Girshick. Long-term feature banks for detailed video understanding. *CoRR*, abs/1812.05038, 2018.

[148] Z. Wu, C. Xiong, C. Ma, R. Socher, and L. S. Davis. Adaframe: Adaptive frame selection for fast video recognition. *CoRR*, abs/1811.12432, 2018.

[149] Wyze. Wyze camera. https://www.safehome.org/home-security-cameras/wyze/, 2019.

[150] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020.

[151] T. Xu, L. M. Botelho, and F. X. Lin. Vstore: A data store for analytics on large videos. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 16:1–16:17, New York, NY, USA, 2019. ACM.

[152] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li. Lavea: Latency-aware video analytics on edge computing platform. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2573–2574, June 2017.

[153] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *JSSPP*, 2003.

[154] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 3320–3328, Cambridge, MA, USA, 2014. MIT Press.

[155] Yueting Zhuang, Yong Rui, T. S. Huang, and S. Mehrotra. Adaptive key frame extraction using unsupervised clustering. In *Proceedings 1998 International Conference on Image Processing. ICIP98 (Cat. No.98CB36269)*, volume 1, pages 866–870 vol.1, Oct 1998.

[156] A. R. Zamani, M. Zou, J. Diaz-Montes, I. Petri, O. Rana, A. Anjum, and M. Parashar. Deadline constrained video analysis via in-transit computational environments. *IEEE Transactions on Services Computing*, 13(1):59–72, 2020.

[157] X. Zeng, B. Fang, H. Shen, and M. Zhang. Distream: Scaling live video analytics with workload-adaptive distributed edge intelligence. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, SenSys '20, page 409–421, New York, NY, USA, 2020. Association for Computing Machinery.

[158] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 377–392, Berkeley, CA, USA, 2017. USENIX Association.

[159] T. Zhang, A. Chowdhery, P. Bahl, K. Jamieson, and S. Banerjee. The design and implementation of a wireless video surveillance system. pages 426–438, 09 2015.

[160] 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*, MobiCom '15, pages 426–438, New York, NY, USA, 2015. ACM.

[161] 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*, MobiCom '15, pages 426–438, New York, NY, USA, 2015. ACM.

[162] A. Z. Zhu, N. Atanasov, and K. Daniilidis. Event-based feature tracking with probabilistic data association. In *2017 IEEE International Conference on Robotics and Automation, ICRA 2017, Singapore, Singapore, May 29 - June 3, 2017*, pages 4465–4470, 2017.

[163] A. Z. Zhu, N. Atanasov, and K. Daniilidis. Event-based visual inertial odometry. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5816–5824, July 2017.

[164] Z. Zou, Z. Shi, Y. Guo, and J. Ye. Object detection in 20 years: A survey. *CoRR*, abs/1905.05055, 2019.