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

Arthi Padmanabhan[⋆§] Neil Agarwal[⋆¶] Anand Iyer[†] Ganesh Ananthanarayanan[†]
Yuanchao Shu[†] Nikolaos Karianakis[†] Guoqing Harry Xu[§] Ravi Netravali[¶]

[§]UCLA [†]Microsoft Research [¶]Princeton University

Abstract

Video analytics pipelines have steadily shifted to edge deployments to reduce bandwidth overheads and privacy violations, but in doing so, face an ever-growing resource tension. Most notably, 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, 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 60.7%, and improves overall accuracy by 8-39% relative to time or space sharing alone.

1 Introduction

Fueled by the proliferation of camera deployments and significant advances in deep neural networks (DNNs) for vision processing (e.g., classification, detection) [19, 28, 46, 69, 74], live video analytics have rapidly grown in popularity [25, 35, 60, 71, 113]. Major cities and organizations around the world now employ thousands of cameras to monitor intersections, homes, retail spaces, factories, and more [1, 5, 6, 10]. The generated video feeds are continuously and automatically queried using DNNs to power long-running applications for autonomous driving, footfall tracking, traffic coordination, business analytics, and surveillance [2, 11–13, 34].

In order to deliver highly-accurate query responses in real time, video analytics deployments have steadily migrated to the edge [25, 78, 107]. More specifically, pipelines routinely incorporate *on-premise* edge servers (e.g., Microsoft Azure Stack Edge [4], Amazon Outposts [3]) that run in hyper-proximity to cameras (in contrast to traditional edge servers [32, 37, 79, 104]), and possess on-board GPUs to aid video processing. These *edge boxes* are used to complement (or even replace [21, 29]) distant cloud servers by locally performing as many inference tasks on live video streams as possible [29, 53, 71, 117]. Generating responses directly on edge boxes reduces transfer delays for shipping data-dense video over wireless links [44, 73, 117] while also bringing resilience to outbound edge-network link failures [7, 80] and compliance with regional data privacy restrictions [77, 85].

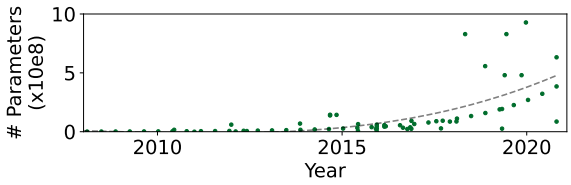

Figure 1: Parameter counts in popular vision DNNs over time. Data drawn from [92].

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 [4, 21, 95]. On the other hand, analytics deployments face rapidly increasing workloads due to the following trends: (1) more camera feeds to analyze [21, 53, 55], (2) more models to run due to increased popularity and shifts to bring-your-own-model platforms [16, 24, 38, 54], and (3) increased model complexity, primarily through growing numbers of layers and parameters (Figure 1) [15, 56, 57, 108]. 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 paper 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 [83, 86, 93], and is thus unlikely to keep pace with the ever-growing memory needs of DNNs (Figure 1). Second, we empirically find that existing memory management techniques that time/space-share GPU resources [26, 39, 50, 56, 94, 110] are insufficient for edge video analytics, resulting in skipped processing on 19-84% of frames, and corresponding accuracy drops up to 43% (§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 [94, 114]. 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 [22, 52, 64, 98]. Third, compared to computation bottlenecks [29, 39, 40, 60, 71], GPU memory restrictions during inference have been far less explored in video analytics.

⋆ These authors contributed equally to this work.

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 (§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 (§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 (§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 computer vision (CV) operations (e.g., convolutions) [22, 64], 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 [27, 31, 65, 66, 88, 106, 118, 119].

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 [62, 63] 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 [23, 70]. 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 (§5). Gemel 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 [94] 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.

Results. We evaluated Gemel on a wide range of workloads and edge settings (§2, §6.3). Overall, Gemel reduces memory requirements by up to 60.7%, which is 5.9-52.3% more than stem-sharing approaches that are fundamentally restricted to sharing contiguous layers from the start of models (Mainstream [59]), and within 9.3-29.0% of the theoretical maximum savings (that disregard layer weights). These memory savings lead to 13-44% fewer skipped frames and overall accuracy improvements of 8-39% compared to space/time-sharing GPU schedulers alone (Nexus [94]). Source code and datasets for Gemel are available at https://github.com/artpad6/gemel_nsdi23.

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 [16, 24, 38, 54], users in our deployment provided pretrained 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 [21,26,49,50,53,59–61,71,109]: 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 [36].

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 [4], Amazon Outposts [3], Sony REA [97], NVIDIA Jetson [8], and Hailo Edge-AI-box [43]. These servers each possess onboard 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 [13, 53], which runs upwards of 10 queries (or DNNs) on each feed [16, 21]. 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 §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.

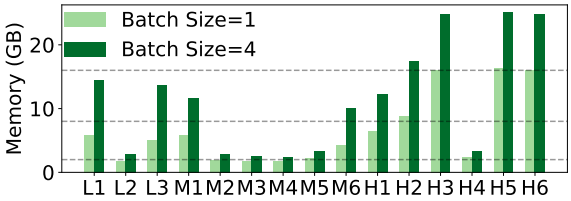

Figure 2: Per-workload memory requirements for two popular batch sizes used in video analytics [94]. Dashed lines represent the available GPU memory on several commercial edge boxes.

Each workload was randomly assigned to one of the cities, with the constituent models being randomly paired with the available videos. The extended version [82] details the workloads, each of which exhibits heterogeneity in terms of the families, tasks, videos, and (combinations of) target objects. 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 §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 [82]): (1) the minimum value (*min*), (2) 50% of the no_swap value (*50%*), and (3) 75% of the no_swap value (*75%*).

3 Motivation

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

Figure 2 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 [18]. 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.

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 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.

Table 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 (§2).

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 [56]. Space-sharing techniques [14, 17] 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 (§3.1)

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 [39, 94], 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) [26, 39, 50, 94, 110]. However, as we will show next, time-sharing techniques are bottlenecked by frequent model swapping, which severely limits their utility. More recently, SwapAdvisor [50] and Antman [110] 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 (§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 [94] 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 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.

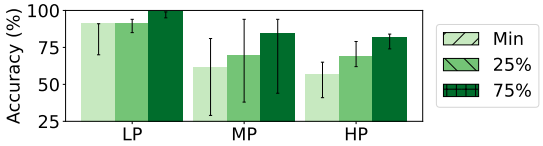

Figure 3: Achieved accuracy with time/space-sharing alone (i.e., using our Nexus variant) for different memory availability (following the definitions in §2). Bars list results for the median workload in each class, with error bars spanning min to max.

Figure 3 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 [94]. 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 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 [115], 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 [55, 60, 71, 76].

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 [40]. 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 [21, 100]; 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 (§2), it may not be possible to compress all of the models while delivering sufficient

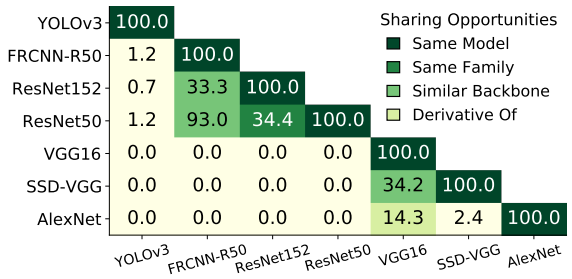

Figure 4: Percentage of architecturally identical layers across different model pairs. See Figure 20 for an extended version.

accuracy. However, even a handful of non-compressed models can exhaust the available GPU memory (§3.1). Second, compressed models exhibit sharing opportunities: our workloads include compressed and non-compressed models (§2), and our results show that Gemel is effective for both (§6).

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 (§3.2), 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 (§4.1), and then lay out the challenges associated with realizing merging in practice (§4.2).

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; Figure 20 presents our comprehensive results. Below, we summarize our findings; Figure 4 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 [20] and a given model can be employed on different video feeds or in search of different objects (§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 layers of ResNet18 are shared with ResNet34 (Figure 19). 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, 3 fully-connected; Figure 5 (left)).

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 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 [96]; VGG16 and AlexNet share 3 out of 16 layers, including 2 fully-connected layers at the end (Figure 5 (right)). Other examples include InceptionNetV3 [102] with GoogLeNet [101].

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

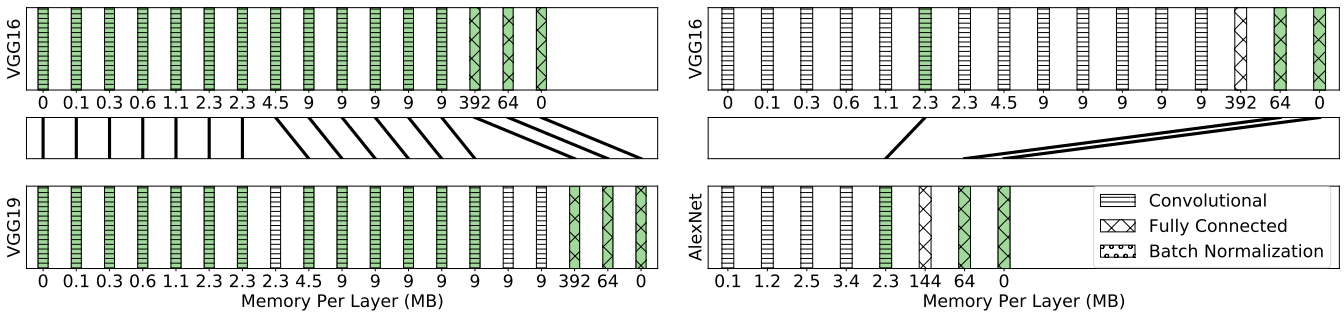

Figure 5: Sharing opportunities between VGG16 and VGG19 (left), and VGG16 and AlexNet (right).

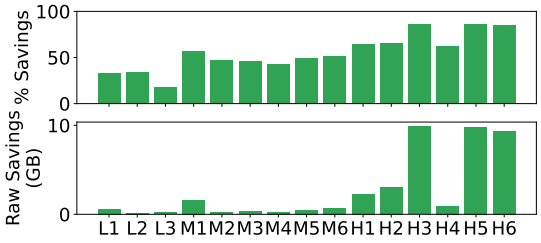

Figure 6: Potential memory savings when all architecturally identical layers are shared across the models in each workload.

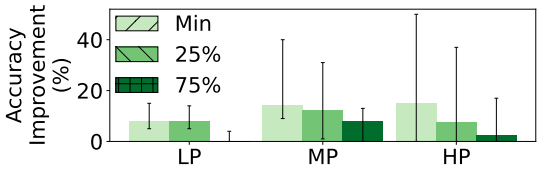

Figure 7: Potential accuracy improvements when sharing all architecturally identical layers. Memory availability is defined in §2, bars list medians, and error bars span min to max.

is, they all ingest pixel representations of raw images, and employ a series of traditional CV operations [22, 64], 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 [27, 49, 65, 66, 71, 118, 119].

Prior work has capitalized on such similarities for efficient multi-task learning [30, 59, 112] and architecture search [75, 84]. 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 §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 (§4.2 and §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 6 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 (§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-end accuracy by comparing the performance of the Nexus variant from §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 (§5.4). As shown in Figure 7, 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.

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 swapping. Concretely, there are two core challenges in practically exploiting the architectural commonalities from §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 8 illustrates this trend by sharing different numbers of identical layers across representative pairs of models that vary on the aforementioned properties, e.g., tar-

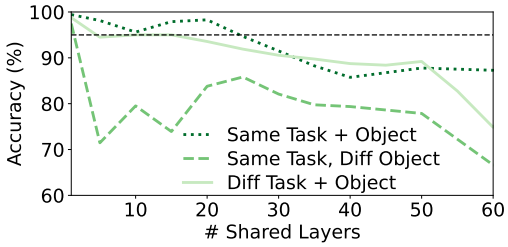

Figure 8: Accuracy after 5 hours of retraining when sharing additional architecturally-identical layers for different model pairs (starting from their origins). Tasks cover detection (Faster RCNN) and classification (ResNet50), and two objects: people, vehicles. Results list the lower per-model accuracies per pair.

get 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 [23, 70]. 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 [62, 63]. 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 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 8 have largely different 'breaking points.' The result

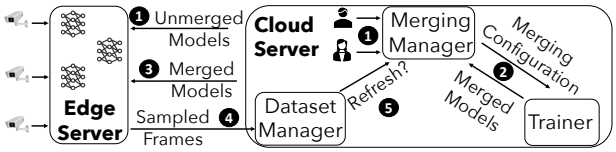

Figure 9: Gemel architecture.

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.

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 §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.

5.1 Overview

Figure 9 shows Gemel's cloud merging and edge inference workflows. As in existing pipelines [16,60,71], 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 §3.2 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 [60, 116]) 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 ex-

pended, (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 [71, 100], 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 uses PyTorch [18] to manage cloud merging and edge inference, and is implemented in ≈3500 LOC. More details are presented in A.1.

5.2 Guiding Observations

Two key empirical observations guide Gemel's approach to tackling the challenges in §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 10 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 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 [47]. 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 10. DenseNet has the same pattern [51].

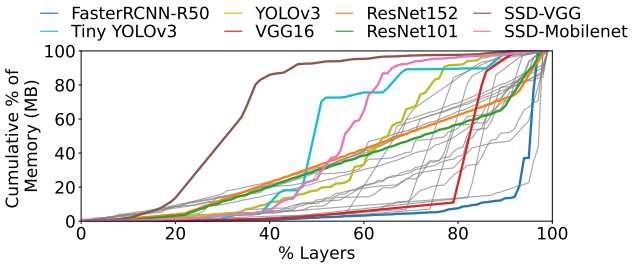

Figure 10: Cumulative memory consumed by each model's layers moving from start to end of the model. §A.4 has full legend.

Figure 10 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 (§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 8). 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 [112]. 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 2).

	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 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.

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 2). This is consistent with our finding that sharing more layers leads to larger accuracy degradations (Figure 8) 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 §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 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.

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 [48]), 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 (§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 §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 predefined 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.

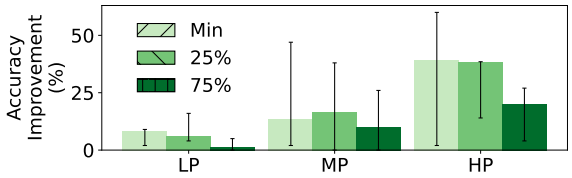

Figure 11: Accuracy improvements with Gemel compared to time/space-sharing alone for different GPU memories (defined in §2). Bars list median workloads, with error bars as min-max.

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 [81, 94], Gemel can directly modify the schedule as described above to maximize benefits. Other schedulers [39] 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 [105] or priority scheduling [111] will only see merging-induced reductions in loading costs if merged models are (by chance) neighbors in the order. Note that finer-grained [50, 110] and space-sharing [9, 14, 17, 21] 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.

6 Evaluation

We primarily evaluated Gemel across the diverse workloads and settings from §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 [59] (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.

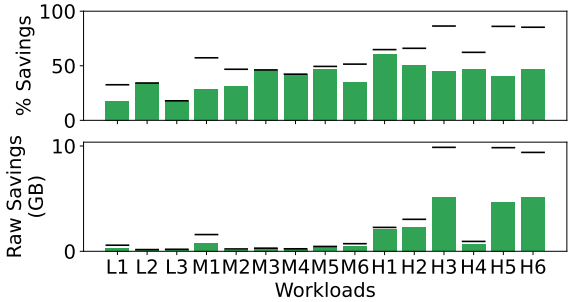

Figure 12: Gemel's per-workload memory savings. Lines above bars show the theoretical optimal savings from Figure 6.

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 §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 §6.2.

Figure 11 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 11 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.

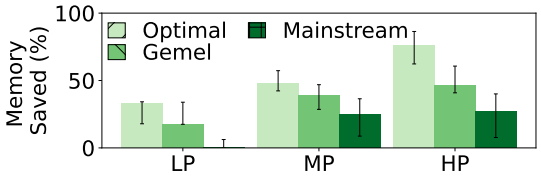

Figure 13: Memory savings with Gemel, an optimal that ignores accuracy, and Mainstream [59]. Bars list the median workload per class, with error bars spanning min to max.

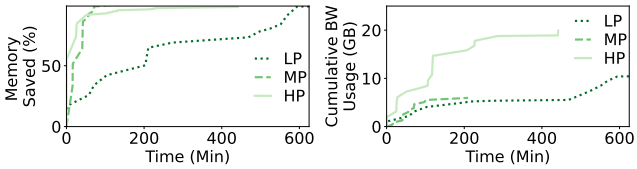

Figure 14: Gemel's memory savings (left) and cloud-to-edge bandwidth usage (right) over time during incremental merging. Results show the median workload per class.

Memory Reductions. Figure 12 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 heuristic in §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* [59], 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 [90]) 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 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 13 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 (§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 8). 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%).

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 14 (left), 73% of Gemel's achieved memory savings for the median HP workload are realized within the first 24 minutes 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 14 (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 profile the time spent in each of Gemel's components. Training delays are configurable (Figure 14), but dominate cloud merging, with the remaining $<$2% of time spent on identifying shareable layers (0.7-1.4s per workload) and serializing/saving weights from successful training (9.1-19.5s per round). The majority of time spent at the edge steadily shifts from model loading to inference as Gemel's incremental merging results stream in; at the median, time spent blocked reduces from 32.8%, 48.3%, and 52.0% to 22.1%, 34.6%, and 27.9% for the LP, MP, and HP

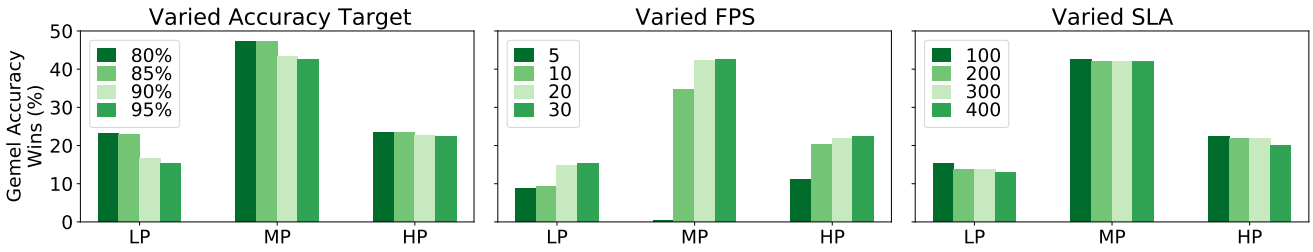

Figure 15: Gemel's accuracy wins (compared to time/space-sharing alone) with varied accuracy targets, FPS, and SLAs.

workloads respectively. Applying results takes <.15s and is not blocking.

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 15 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.

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 §2, and we report memory saved over time. Figure 16 shows results for two representative workloads (HP3, MP2); the remaining workload results are in §A.4. 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.

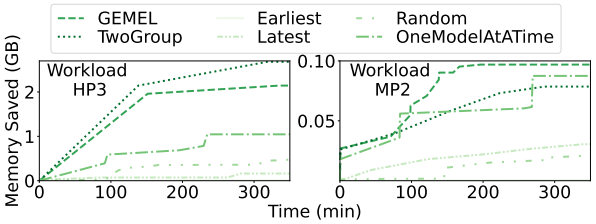

Figure 16: Comparing variants of Gemel's merging heuristic on two representative workloads.

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 16 (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 16 (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 16 (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.

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 [101], DenseNet [51]). In total, our analysis involves 17 videos (8 scene types), 13 objects, and 16 models; the extended version [82] 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

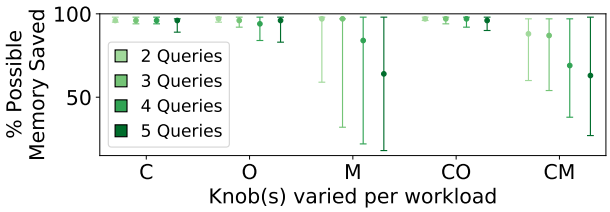

Figure 17: Memory savings across subset of generalization workloads, organized by workload size (color) and knobs varied (Camera, Object, Model). Distributions show median and 25-75%ile; accuracy target was 95%. Figure 22 has full graph.

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 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 17, 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 19). 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 (§4.2).

7 Additional Related Work

Certain systems reuse model components [91], most relatedly via stem sharing for compute savings [59] or sharing operators with identical weights anywhere in models [68]; in contrast, Gemel targets memory savings, and enables sharing architecturally-identical layers anywhere in models even if they have different weights. Layer sharing in multi-task learning is often studied in the context of transfer learning, where models for a task with insufficient data leverage the dataset of a related task [30,99,103]; Gemel considers multiple sets of pretrained weights for sharing, each with different goals (e.g., detection vs. classification, different objects).

Other platforms optimize model serving either by tuning video analytics-specific knobs to lower compute footprints [29,35,49,55,60,61,87,109,116,117], or by identifying lightweight variants of individual models that match specific hardware resources [45, 89]; Gemel focuses on memory (not compute) bottlenecks, and optimizes across models. Some frameworks reuse results across frames [31,33,42,67,71], reducing frame rates for inference and alleviating the impact of model loading delays. Gemel provides benefits at lower FPS (§6.2), and also can alleviate memory pressure across spatially correlated feeds that exhibit limited reuse opportunities at the same time (§3.2).

There exist training optimizations that trade off memory usage for computation overheads [83, 86, 93]; we eschew such techniques given the holistic constraint on compute resources that edge boxes face (§1). Finally, another body of work develops metrics to quantify how similar models will behave [41, 58, 72]. While Gemel does not consider model similarity metrics in its heuristic (we quantitatively observe that 'model similarity' is not reflected in layer merging potential), we leave it to future work to explore the relationship between 'model similarity' and 'layer similarity' in improving Gemel's prediction of layer merging potential.

8 Conclusion

Model merging is a new memory management technique that exploits architectural similarities across vision DNNs by sharing their common layers (including parameters but not intermediates). Gemel efficiently carries out model merging by quickly finding and retraining accuracy-preserving layer sharing configurations, and scheduling edge inference to maximize merging benefits (8-39% accuracy boosts).

Acknowledgements. We thank Ramesh Govindan and Jennifer Rexford for their valuable feedback on earlier drafts of the paper. We thank our shepherd, Wenjun Hu, and the anonymous NSDI reviewers for their constructive comments. This work was supported in part by a Sloan Research Fellowship, research grants from Cisco, ONR grant N00014-18-1-2037, and NSF CNS grants 2152313, 2153449, 2147909, 2140552, 1703598, 1763172, 1907352, 2007737, 2006437, 2128653, and 2106838.

References

[1] Absolutely everywhere in beijing is now covered by police video surveillance. https://qz.com/518874/.

[2] Are we ready for ai-powered security cameras? https://thenewstack.io/are-we-ready-for-ai-powered-security-cameras/.

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

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

[5] British transport police: Cctv. http://www.btp.police.uk/advice_and_information/safety_on_and_near_the_railway/cctv.aspx.

[6] Can 30,000 cameras help solve chicago's crime problem? https://www.nytimes.com/2018/05/26/us/chicago-police-surveillance.html.

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

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

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

[10] Paris hospitals to get 1,500 cctv cameras to combat violence against staff. https://bit.ly/2OYiBz2.

[11] Powering the edge with ai in an iot world. https://www.forbes.com/sites/forbestechcouncil/2020/04/06/powering-the-edge-with-ai-in-an-iot-world/.

[12] Video analytics applications in retail - beyond security. https://www.securityinformed.com/insights/co-2603-ga-co-2214-ga-co-1880-ga.16620.html/.

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

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

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

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

[17] NVIDIA TensorRT, April 2021.

[18] Pytorch, April 2021.

[19] Pytorch-yolov3. https://github.com/eriklindernoren/PyTorch-YOLOv3, 2021.

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

[21] 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.

[22] 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.

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] S. R. E. Datondji, Y. Dupuis, P. Subirats, and P. Vasseur. A survey of vision-based traffic monitoring of road intersections. *Trans. Intell. Transport. Sys.*, 17(10):2681–2698, Oct. 2016.

[35] 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.

[36] 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.

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

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

[39] 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.

[40] 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.

[41] P. Guo, B. Hu, and W. Hu. Sommelier: Curating dnn models for the masses. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1876–1890, 2022.

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

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

[44] 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.

[45] 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, page 123–136, New York, NY, USA, 2016. Association for Computing Machinery.

[46] K. He, G. Gkioxari, P. Dollár, and R. B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017.

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

[48] 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.

[49] 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, Oct. 2018. USENIX Association.

[50] 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.

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

[52] 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-5425656ae359, 2018.

[53] 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.

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

[55] 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.

[56] 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.

[57] 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.

[58] H. Jia, H. Chen, J. Guan, A. S. Shamsabadi, and N. Papernot. A zest of LIME: Towards architecture-independent model distances. In *International Conference on Learning Representations*, 2022.

[59] 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.

[60] 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, page 253–266, New York, NY, USA, 2018. Association for Computing Machinery.

[61] 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.

[62] 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.

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

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] Y. Li, Z. Zhang, B. Liu, Z. Yang, and Y. Liu. ModelDiff: testing-based DNN similarity comparison for model reuse detection. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, jul 2021.

[73] 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.

[74] 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.

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

[76] 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.

[77] 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.

[78] 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.

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

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

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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, 2018.

[88] 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.

[89] 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.

[90] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[91] 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.

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

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

[94] 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.

[95] 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.

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

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

[98] 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.

[99] 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.

[100] 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.

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

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

[103] 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.

[104] 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.

[105] 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.

[106] 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.

[107] 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.

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

[109] 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.

[110] 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.

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

[112] 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.

[113] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98:289–330, 2019.

[114] 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.

[115] 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.

[116] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, Boston, MA, Mar. 2017. USENIX Association.

[117] 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.

[118] 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.

[119] 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.

A Appendix

A.1 Implementation Details

Gemel's main components are training models at the cloud server and running the scheduler at the edge. 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 [30]: 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 §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.

A.2 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 3: Knob values considered in generalization study.

A.3 Workload Memory Settings

Workload	L1	L2	L3
Min	4.50	1.45	4.50
50%	5.12	1.59	4.72
75%	5.43	1.66	4.83

Table 4: Edge box memory settings for LP workloads (in GB).

Workload	M1	M2	M3	M4	M5	M6
Min	3.35	1.45	1.32	1.32	1.45	3.35
50%	4.56	1.62	1.55	1.45	1.83	3.77
75%	5.16	1.70	1.65	1.52	2.02	3.99

Table 5: Edge box memory settings for 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 6: Edge box memory settings for HP workloads (in GB).

A.4 Additional Figures

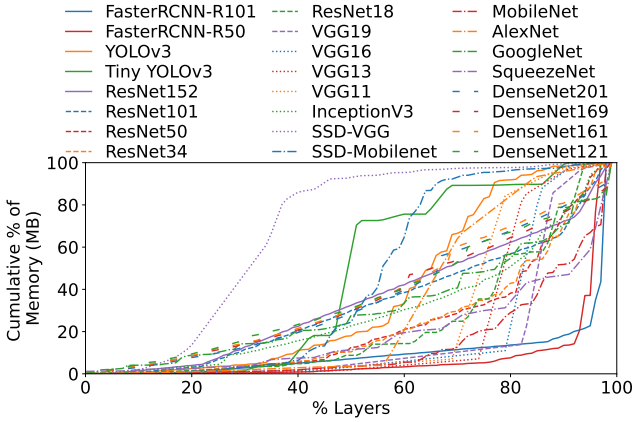

Figure 18: Extended version of Figure 10. Cumulative memory consumed by each model's layer groups moving from start to end of the model.

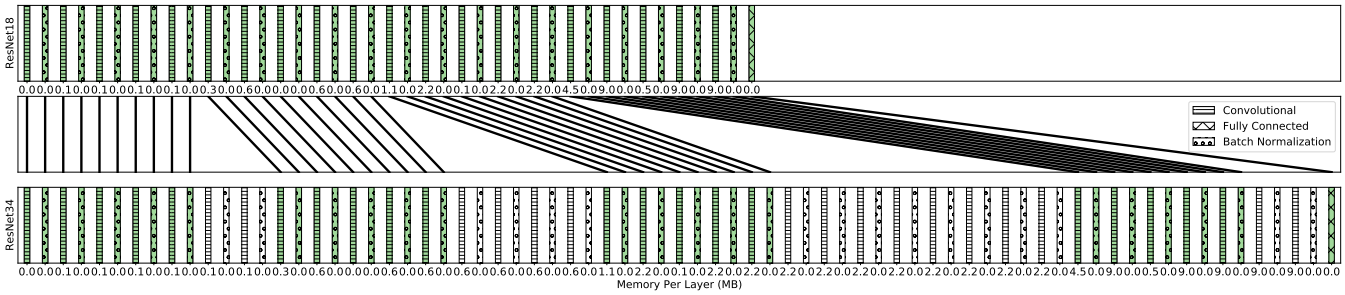

Figure 19: ResNet18 and ResNet34 are variants within the ResNet model family [47]. They share 41/73 layers (20 convolutional, 1 fully-connected and 20 batch normalization).

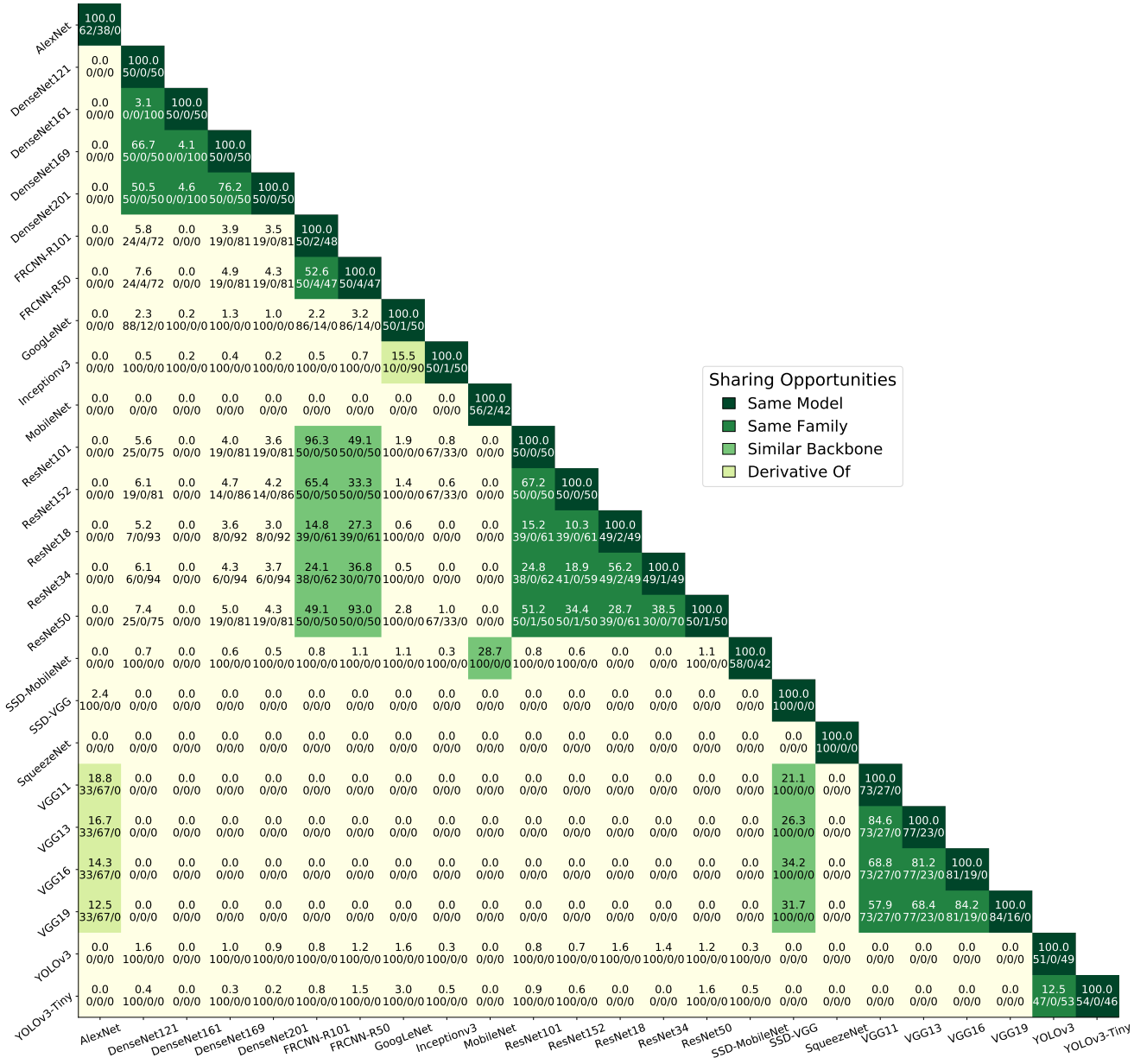

Figure 20: Extended version of Figure 4. For each unique pair of models, we show the percentage of architecturally identical layers and of those layers, the percent breakdown across layer types (%Convolutional / %Linear / %BatchNorm).

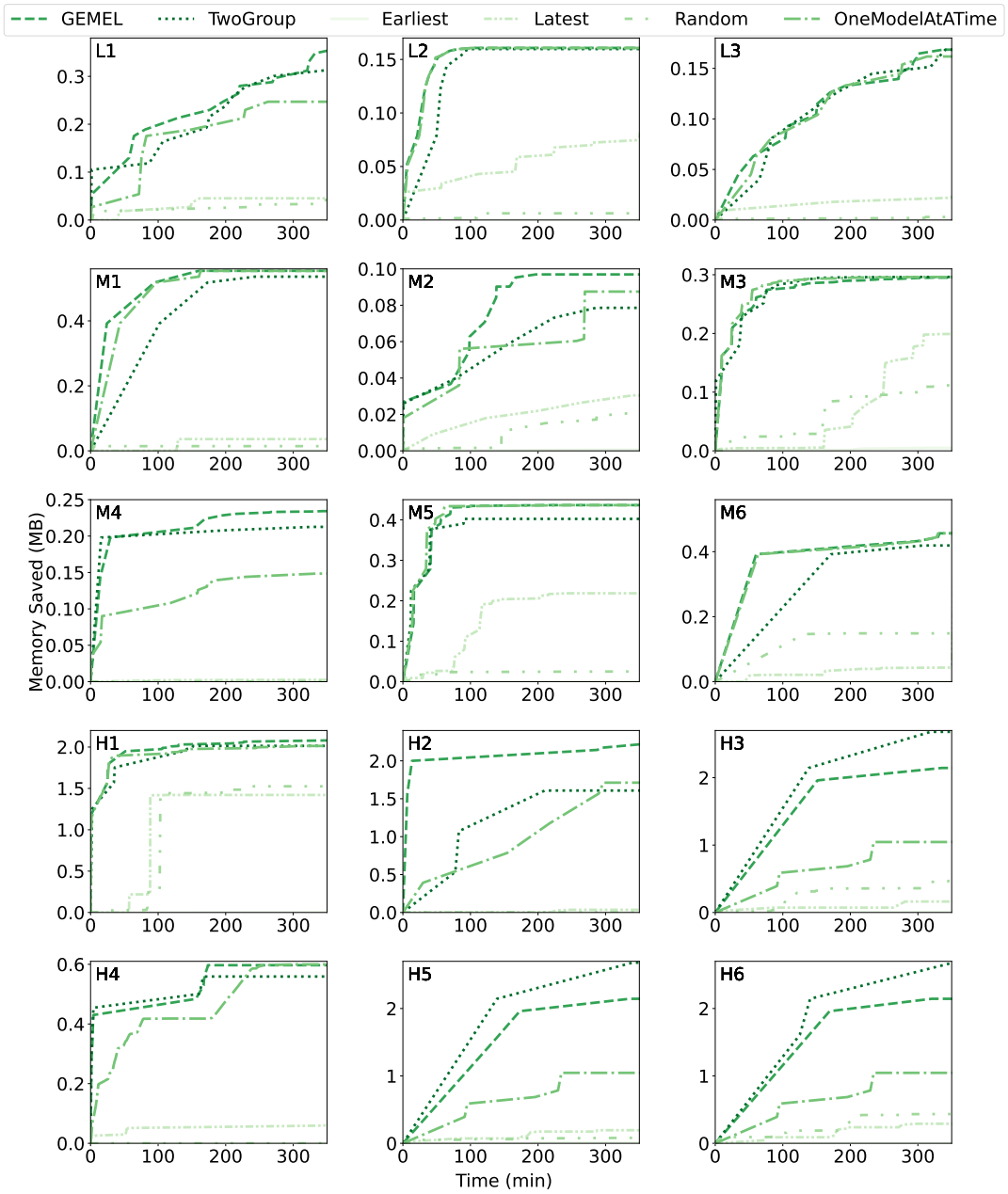

Figure 21: Complete version of Figure 16. Comparison of Gemel with other merging heuristics.

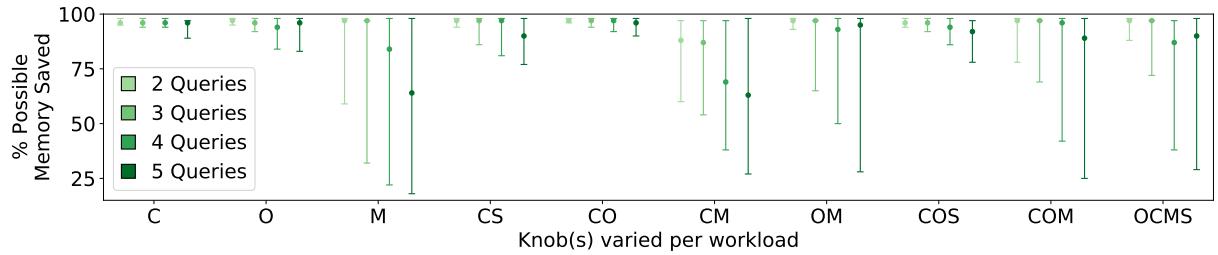

Figure 22: Extended version of Figure 17. Memory savings across 872 workloads, organized by workload size (color) and knobs varied (x-axis). We plot the median of each distribution (error bars spanning 25-75P). Knobs are labeled as follows: C:Camera, O:Object, M:Model, S:Scene.