Memory-efficient DNN Training on Mobile Devices

In Gim and JeongGil Ko
School of Integrated Technology
College of Engineering
Yonsei University, Seoul, Korea
{hyunjun.kim,jeonggil.ko}@yonsei.ac.kr

ABSTRACT

On-device deep neural network (DNN) training holds the potential to enable a rich set of privacy-aware and infrastructure-independent personalized mobile applications. However, despite advancements in mobile hardware, locally training a complex DNN is still a non-trivial task given its resource demands. In this work, we show that the limited memory resources on mobile devices are the main constraint and propose *Sage* as a framework for efficiently optimizing memory resources for on-device DNN training. Specifically, *Sage* configures a flexible computation graph for DNN gradient evaluation and reduces the memory footprint of the graph using operator- and graph-level optimizations. In run-time, *Sage* employs a hybrid of gradient checkpointing and micro-batching techniques to dynamically adjust its memory use to the available system memory budget. Using implementation on off-the-shelf smartphones, we show that *Sage* enables local training of complex DNN models by reducing memory use by more than 20-fold compared to a baseline approach. We also show that *Sage* successfully adapts to run-time memory budget variations, and evaluate its energy consumption to show *Sage*'s practical applicability.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Human-centered computing** → **Ubiquitous and mobile computing systems and tools**.

KEYWORDS

Mobile DNN training framework, Mobile resource optimization, Memory adaptive model training

ACM Reference Format:
In Gim and JeongGil Ko. 2022. Memory-efficient DNN Training on Mobile Devices. In *The 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22), June 25–July 1, 2022, Portland, OR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3498361.3539765

MobiSys '22, June 25–July 1, 2022, Portland, OR, USA

ACM ISBN 978-1-4503-9185-6/22/06…$15.00
https://doi.org/10.1145/3498361.3539765

1 INTRODUCTION

Many mobile deep learning applications require or can benefit from, the ability to update their models on-the-fly. Especially, for applications that offer personalized feedback, a small amount of extra training with actual user data (e.g., model fine-tuning) can significantly enhance application quality. Services that exploit speech recognition [24], face verification [28, 39] or activity recognition [8, 42] can benefit from such additional training to improve accuracy and adapt to heterogeneous sensors and usage characteristics. Unfortunately, deep learning model training is a resource intensive task, and their hardware demands are much higher than that of inference operations. Thus, the common practice has been to offload the training task to a more powerful external server [19, 30, 45] rather than to perform model training on the resource-limited mobile or embedded computing devices themselves.

Albeit challenges, on-device model training provides substantial advantages over computation offloading regarding privacy protection and application scalability. Specifically, by updating models locally, user data can be kept local, and external data exchange can be minimized. Sharing information such as keyboard input history, voice recordings or face images with an external server may raise privacy concerns [14, 15, 29]. Furthermore, mobile applications with on-device training can quickly scale with minimal backend infrastructure and reduces their reliance on stable network connections. Furthermore, practical on-device training of complex models is a fundamental prerequisite for federated learning [23, 25], enabling scalable intelligence with private and large data.

Fortunately, recently released mobile GPUs have demonstrated impressive progress towards supporting heavy deep learning workloads, becoming powerful enough to not only serve inference operations but also some level of model training on mobile platforms. For example, modern smartphones equipped with dedicated accelerators for fast neural network processing (e.g., NPUs and GPUs), the Samsung Exynos 990 (released in 2020) processes 8 samples/sec for MobileBERT [1, 38], a language model with 25M parameters, being only 4× slower compared to a server-used NVIDIA GTX 1080 GPU (released in 2019). This performance gap differs with GPU models, but it is safe to conclude that processing power on mobile GPUs is scarce, but not in despair. Energy limitations can be another hurdle, but we can practically assume that training operations take place when the device is power-plugged and idle from other operations.

A more fundamental and prevailing challenge that hinders on-device model training is its *memory limitations*. Training operations mandate all intermediate activations to be retained in memory when computing gradients with automatic differentiation, making memory management a non-trivial task given limited memory on mobile platforms. For example, training BERT-small [6], a state-of-the-art language model, consumes more than 8 GB of memory,

whereas inference requires less than 500 MB. Considering that smartphones in the market are equipped with 4-12 GB of memory, locally training heavy models remains a challenging task.

Despite active research on mobile-targeted deep neural network (DNN) optimization for inference, model compression, and offloading, schemes for enabling effective on-device training is yet understudied. There are memory reduction schemes designed for on-server DNN training [13, 31–33, 43]. However, they rely on assumptions invalid on mobile architectures, such as host-side memory, ample memory bandwidth, and predictable resources. This suggests that memory management for mobile on-device model training requires a different approach, given distinctive characteristics.

In this work, by thoroughly analyzing various memory management approaches for DNN training, we show theoretically and empirically that naive application of existing low-memory training schemes to on-device training can cause sub-optimal performance due to three reasons: (i) mobile devices are more heterogeneous compared to server configurations, (ii) server-grade GPUs and mobile GPUs exhibit significantly different levels of parallel processing capability and memory bandwidth, and (iii) mobile memory resources are *extremely* limited and can show high levels of dynamics. These challenges suggest that assumptions on aggressive parallel processing made by currently available memory management schemes will not hold, and static memory planning approaches are unsuitable for mobile DNN model training.

Based on such observations, we propose a novel framework for memory-efficient on-device DNN training, *Sage*, which employs four core techniques: (i) a flexible automatic differentiation framework, (ii) a series of graph-level optimization, (iii) operator-level optimization, and (iv) a hybrid approach for run-time memory management. Specifically, to minimize memory consumption during the model training process, *Sage* configures a flexible computational graph for DNN gradient evaluation so that optimizations such as operator fusion (Sec.3.2.1) and subgraph reduction (Sec 3.2.2) can be applied. We note that *Sage* is the first work to explore the graph-level optimization for low-memory training. In operator-level optimization, memory-heavy operations (e.g., matrix multiplication) are manually tuned and memory transfer operations are minimized (Sec 3.3). As a final step, to dynamically adjust memory use to the available system resources, a novel hybrid algorithm of gradient checkpointing and gradient accumulation is designed (Sec. 3.4).

We implement *Sage* on three GPUs to perform extensive empirical evaluations. Specifically, we take two mobile platforms (e.g., Samsung Galaxy S10 and Note 20) and the NVIDIA RTX 3090 as target hardware to examine the impact of *Sage* in computing environments of different capabilities. Using these implementations, we test for four widely-used deep learning models ranging from lightweight models commonly used for mobile applications (e.g., MobileNet v2) to heavier models with higher accuracy and complex tasks (e.g., ResNet-50, DenseNet-121, BERT-small). Our evaluations show that *Sage* supports model training even when memory is extremely scarce on mobile GPUs, with competitive latency and 20-fold memory reduction compared to baseline approaches. Furthermore, we show that *Sage* well adapts to dynamically changing memory availabilities when training complex models, indicating the practical usability of *Sage* under real-world mobile usage scenarios.

Specifically, this work makes the following contributions.

- Through a preliminary study on the memory requirements for DNN training, we show that memory scarcity is the core limitation that restricts on-device training of complex deep learning models. Furthermore, we identify three accompanying challenges in implementing a low-memory training framework on mobile devices.
- We propose *Sage*, the first on-device training framework which incorporates various low-memory approaches given the constraints of mobile systems. *Sage* reveals a practical balance between memory consumption and training latency based on the hypothetical and empirical experiments on mobile platforms.
- We present extensive evaluations of *Sage* with four state-of-the-art deep learning models via implementations on the Samsung Galaxy S10 and Note 20. Our results show that *Sage* realizes on-device training of complex DNNs in a memory- and latency-efficient manner.
- Finally, we provide a fully open-sourced implementation of *Sage* at http://github.com/eis-lab/sage.

2 MOTIVATION AND BACKGROUND

2.1 On-device DNN Training

Recent advancements in mobile SoCs and software have catalyzed novel research on expanding the limits of deep learning model usage on resource-limited platforms. Performing on-device inference for fairly heavy models is no longer a challenge, and their simplified versions can now be trained locally. Yet, albeit aggressive research from the machine learning research community in designing more precise neural networks, mobile computing research has not addressed how such state-of-the-art but heavy and complex models can be trained and customized on mobile platforms.

As challenging as it is, on-device model training holds the potential to accelerate research in mobile computing as well as federated learning [25]. By supporting applications to train deep neural networks (DNNs) locally, services can be designed to be more aware of protecting privacy-sensitive data and minimize their infrastructure dependency so that users can be served under any external circumstances. With on-device training, model fine-tuning can be applied to an initially-provided generic deep learning model, which can be highly beneficial as locally collected data can be used to directly and easily update model parameters.

Specifically, the main technical hurdle, as we further argue in the following sections, is that model training requires a significant amount of memory resources. A slower processor may increase the training latency, but insufficient memory is a game-stopper by itself. Even the most recently released mobile platforms cannot support the memory requirements of well-known state-of-the-art deep learning models. Thus, in current practice, applications that require (or can benefit from) user personalization, either utilize a simple model or delegate the training task to an external server.

On the contrary, there is still a rapidly growing interest in designing "more accurate" deep learning models at the cost of additional processing and memory resources [44]. Graphics Processing Units (GPUs) designed for deep learning servers are more powerful than ever, capable of executing complex operations in large batches, allowing sophisticated deep learning models to execute efficiently.

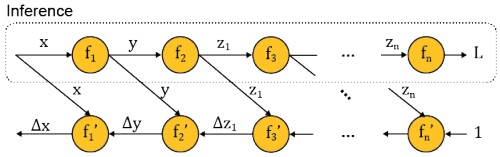

Figure 1: Computational graph representation of model inference and training.

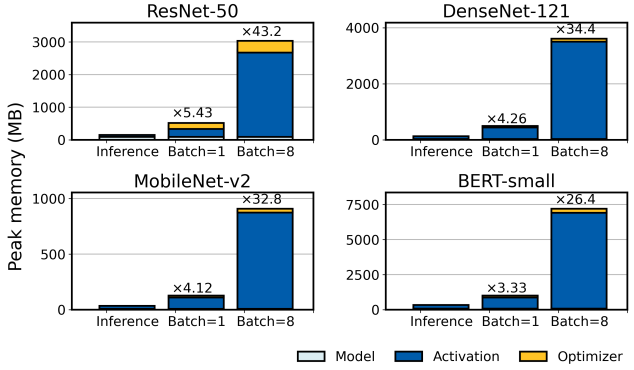

Figure 2: Memory breakdown for four widely-used DNNs when performing inference and training (with different batch sizes).

Mobile GPUs are also experiencing similar improvements, yet, they still lack processing power and peripheral resources compared to their server-side counterparts. Naturally, system designers are forced to sacrifice the performance, as fully exploiting a complex deep learning model, including the training operations, is not yet feasible on mobile platforms.

2.2 The Memory Blowup Problem

A major bottleneck in training deep learning models on-device is their memory scarcity. Given that DNN training is an iterative procedure requiring model states and weights to be retained between iterations, memory usage can be 5 to 100 times more compared to inference operations depending on the input and batch size. This translates to multiple GBs of memory on platforms where memory is not abundant.

A fundamental reason behind this memory blowup originates from how a model's gradients are computed when training the neural network. Automatic differentiation, an essential technique for computing DNN gradients, is cored on the chain rule of derivatives. For example, take Figure 1, a neural network layer $\vec{y} = f(\vec{x})$ with loss $L = g(\vec{y})$. The gradient $\nabla\vec{x} = (\frac{\partial L}{\partial x_1}, ..., \frac{\partial L}{\partial x_n})$ is computed as:

$$\frac{\partial L}{\partial x_n} = \frac{\partial L}{\partial y_n} \times \frac{\partial y_n}{\partial x_n}, \tag{1}$$

which indicates that $\vec{x}$ and $\nabla\vec{y}$ should be computed *a priori* and be present in memory when computing the gradient $\nabla\vec{x}$.

For inference operations of model $f(\vec{x})$, the computational graph is straightforward and intermediate states are discharged after its use. However, for model training, the graph topology becomes more complex. Each gradient node ($\nabla\vec{x}$) depends on both its output gradient ($\nabla\vec{y}$) and original value ($\vec{x}$) due to the nature of automatic differentiation (Eq. 1). Therefore, all intermediate states are retained in memory until the final loss is computed. Roughly, the memory requirement for inference is determined by the largest-sized activation in the network, whereas memory requirements for training are determined by the sum of all activation sizes.

With this in mind, we quantify the required memory for four widely-used DNN architectures in Figure 2. We examine the memory requirements for inference operations and model training with different batch sizes and break down the memory usage into three portions to show how much memory the model, optimizer, and activations consume, respectively. Training relatively lighter models such as MobileNet v2 requires only ~1 GB of memory. Given recent smartphones' memory budgets, this is (although still substantial) not infeasible to guarantee. However, more complex state-of-the-art models, such as BERT-small, can require up to 8 GB of memory for training. Since the mobile OS and its background apps will consume a few GBs of memory as a baseline, training these complex models becomes a challenging, if not infeasible, task. Nevertheless, with increased model complexity comes higher accuracy and broader application scenarios. Thus, there is a need for a framework that effectively manages the memory usage of DNN training so that they properly function on mobile platforms.

2.3 Memory Reduction Approaches

The memory blowup issue in DNN training has become increasingly prominent as deep learning research has evolved towards utilizing deeper models and larger mini-batches to seek higher accuracy. Even on server-scale GPUs, this increased memory requirement is becoming challenging to accommodate, leading to various efforts to reduce model training memory overhead, which we broadly classify into five categories below and illustrate in Figure 3.

• **Gradient accumulation** approaches reduce memory usage in model training by incrementally computing mini-batch operations [37]. Typically, computing the gradient of an entire mini-batch is preferable to enhance parallelism. Unfortunately, computing a full mini-batch can be overwhelming for resource-limited platforms due to its memory needs. Mathematically, it is possible to split the mini-batch into smaller units and separately compute sub-mini-batch gradients and then accumulate them to evaluate the mini-batch gradient. Thus, for models where larger mini-batch sizes are beneficial, gradient accumulation is an efficient method to reduce memory use with no accuracy loss (with exceptions for batch-norm, as we discuss in Sec. 5). The key challenge is determining *how* the mini-batch is split as batches too small can result in hardware under-utilization.

• **Gradient checkpointing** directly trade-offs memory and additional computation by retaining only a subset of activations on the memory, and recomputing non-stored ones on demand [9]. The key here is determining which activations to store and which to drop. Step-based gradient checkpointing stores activations based on a predefined distance (e.g., one every $\sqrt{n}$) [3]. Some studies compute an optimal schedule to minimize computation latency given available hardware profiles [17, 22]. Recently, dynamic gradient checkpointing exploits heuristics to determine on-memory activations with minimal computing overhead [20]. As Figure 3 shows, gradient checkpointing reduces memory footprints over activation layers (c.f., Fig. 3 (b)), while accumulation operates over mini-batches (c.f., Fig. 3 (a)). These approaches are orthogonal and can easily co-exist.

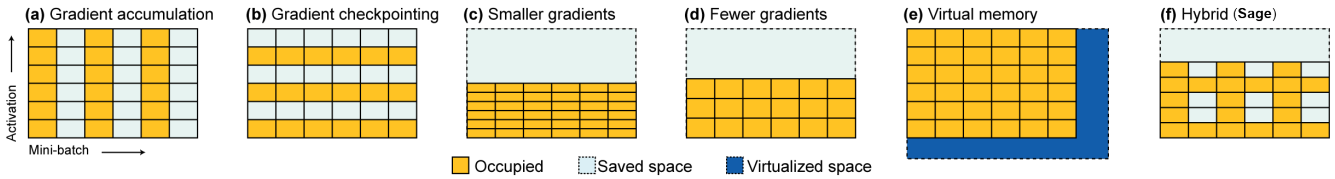

Figure 3: Different memory reduction schemes that can be applied for DNN training optimization.

• **Using smaller activation sizes** is a widely adopted strategy for reducing a model's memory consumption (c.f., Fig. 3 (c)). For example, using lower-precision floating points or integer units when calculating and storing intermediate states reduces model operation costs [26]. Most GPUs, including mobile GPUs have built-in support for 16-bit floating points or 8-bit integers, offering computational efficiency compared to commonly used 32-bit floating points. Sparsification also reduces gradient size by suppressing computation for (near) zero-value elements [4]. However, current hardware has shown inefficiencies in storing/computing sparse matrices [7], which limit their practical and general use.

• **Reducing the number of activations** is also feasible for alleviating model computation memory by reducing intermediate states in a model (c.f., Fig. 3 (d)). For example, a series of arithmetic operations in a computational graph can be merged into one composite operation to garner multiple gradients into a single state. Many DNN compilers [2, 35] employ such graph-level optimization for acceleration and memory footprint reduction of inference operations, but its use in training operations is yet to be explored.

• **Memory virtualization** in GPUs exploits host-side memory to logically extend usable memory (c.f., Fig. 3 (e)). A number of previous works have applied such schemes to server-scale DNN training [11, 31, 34]. Unfortunately, since mobile SoCs exploit a unified memory architecture for the GPU and CPU, this is not suitable for mobile environments.

2.4 Technical Challenges

Locally training a complex DNN model with mobile GPUs introduces several unique challenges not present in server-used GPUs. Among many, we summarize three major challenges and present discussions on how they affect the design of an on-device DNN training framework.

• **Device heterogeneity:** Optimizing a DNN training operation requires a well-specified target environment as the ideal approach varies for different memory budgets and hardware capabilities. In server-side training scenarios, specific hardware and platform (e.g., CUDA) can be targetted for the system design, allowing optimization solutions to be determined at the compilation time. On the other hand, finding generic solutions for mobile platforms is challenging due to device heterogeneity. For example, even the same smartphone models can have different memory capacity and processing power depending on their version. This complicates the design of a globally applicable optimization scheme and suggests that memory management schemes for on-device DNN training should be flexible and runtime-adaptable.

• **Limited processing power:** While the performance of mobile GPUs has improved to a level where they can support some level of DNN training, their limited parallelism and memory bandwidth invalidate several presumptions of existing memory management algorithms tailored for powerful GPUs. For example, the benefits of using larger batches when applying gradient accumulation may be insignificant in mobile GPUs due to the limited processing power. Therefore, memory optimization strategies for mobile on-device training must consider such performance limitations and fully exploit the available hardware.

• **Scarce and dynamic memory:** Unlike server platforms where the CPU and GPU are paired with separate hardware memory, most mobile SoCs adopt a unified memory architecture, where the different processors share a common physical memory. This means that host-side memory consumption caused by the operating system, background workloads, and applications can constrain the available memory for DNN training. As a result, the unified memory architecture can cause unpredictable memory churn. Therefore, an on-device DNN training framework should dynamically adjust its operations to suit the real-time memory constraints.

With these challenges in mind, the following section presents details on *Sage*, our proposed framework that optimizes the memory usage on mobile platforms for supporting on-device training of complex DNN models.

3 *SAGE* DESIGN

Training approaches discussed in the previous section show distinctive latency overhead depending on the memory budget and underlying hardware. Thus, various methods can achieve similar memory reduction but with different latency overhead. The design goal of *Sage* is to (i) combine multiple low-memory training methods to support on-device training with an extreme memory budget while (ii) ensuring practical latency overhead on mobile platforms.

Sage optimizes training operation via a three-step process as illustrated in Figure 4. First, given a DNN model, *Sage* constructs a computation graph with automatic differentiation (AD). Unlike typical AD implementations, gradients exist as separate nodes in the computation graph. This enables the second step, where a series of graph- and operator-level optimizations are performed to lighten static memory footprints. Finally, during the graph evaluation for the gradient descent, *Sage* employs a hybrid strategy to adaptively combine gradient checkpointing and gradient accumulation based on the available system memory and computation power. The rest of this section provides details on these operations.

3.1 Automatic Differentiation

As briefly discussed in Sec 2.2, AD is a fundamental algorithm for computing DNN parameter gradients. There can be various ways to implement the AD framework, but a typical way adopted by most deep learning frameworks (e.g., TensorFlow and PyTorch) is to use forward and backward passes for gradient computation. For

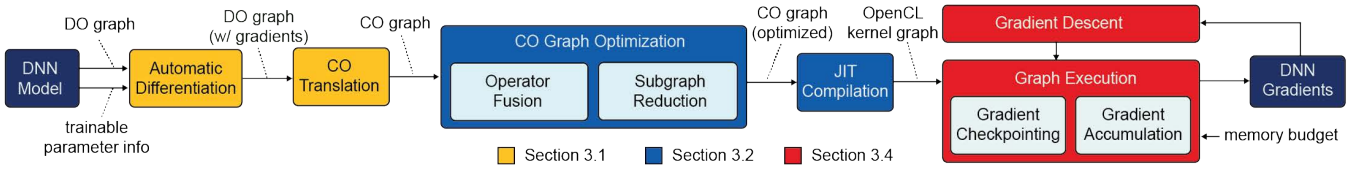

Figure 4: *Sage*'s operational flow, consisting of graph construction, graph optimization, and runtime optimization.

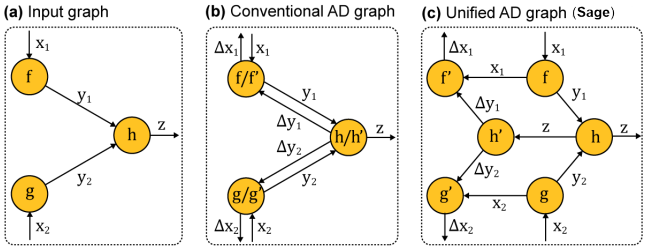

Figure 5: A sample unified computation graph generated by *Sage*'s automatic differentiation framework compared to a conventional computation graph.

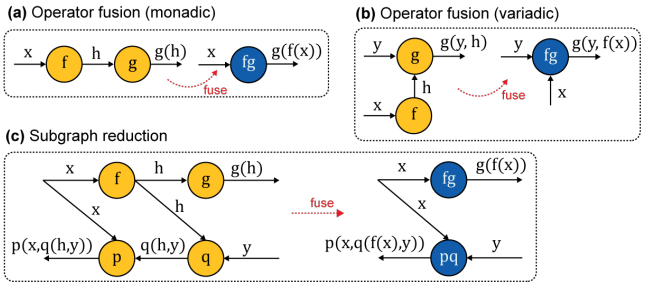

Figure 6: Graph-level optimization schemes (i.e., operator fusion and subgraph reduction) used in *Sage*.

example, the forward pass computes and stores model activations, and the backward pass traverses the model in the reverse order to compute gradients using stored activations, as in **(b)** of Figure 5.

However, such separated passes for gradient evaluation limit and complicate the use of graph-level optimizations such as operator fusion, as fusing operations in one direction can invalidate the mathematical logic in the other. To fully integrate graph optimization in DNN training, we re-design the AD framework to generate a unified computational graph (combining the two passes) for gradient evaluation. Specifically, *Sage* uses a two-layer abstraction to express a DNN node (i.e., *differentiable operations (DO)*) and *computable operations (CO)* to decouple the *evaluation* and *differentiation* processes of DNN computation graphs.

Specifically, a DNN model is defined as a DO graph, where DO nodes include a set of high-level tensor operations necessary to define DNNs, such as arithmetic and convolution operations, coupled with its derivative definition, also expressed in the DO graph. In performing automatic differentiation, each DO node augments the graph with its gradient DO graph. As Figure 5 shows, *Sage* outputs a unified graph containing both the forward and backward pass information. DO nodes can be translated into a corresponding CO graph, where CO nodes describe lower-level operations for tensor computation and memory layout modifications. As CO graph is not tied to the differentiability constraints, it can be freely altered, merged, or removed as long as it gives the same computation result, allowing subsequent graph-level optimizations to take place.

3.2 Graph-level Optimizations

Once the CO graph is produced from the DO graph, *Sage* executes a series of graph-level optimizations to minimize the memory consumption and execution overhead by reducing the graph size. Optimization approaches presented below take place sequentially in their respective dimensions, and we illustrate details on each of their operations in Figure 6.

3.2.1 Operator fusion. Fusing multiple operators and generating a single (larger) operator is an effective approach for reducing memory footprints of a DNN's graph representation. At a high level, all CO operations can be categorized into three groups: *symmetrical operation*, *asymmetrical operation*, and *opaque operation*. Using the rules described below, we target to fuse most (if not all) operators of a CO graph to minimize its memory footprint.

The first category of operations, symmetrical operation, group tensor operations where all inputs and outputs hold the same tensor shape, with computation occurring in an element-wise manner. Basic arithmetic operations, element-wise mathematical functions, and activation functions fall into this category of operations. For these, we observe that all neighboring symmetrical operations in the graph can be fused into a single operation. On the contrary, the second operation type, asymmetrical operation, covers broader tensor operations with heterogeneous input and output shapes. Examples in this category include matrix multiplication, convolution, reduction and contraction operators. Their characteristics suggest that asymmetrical operations can be fused with neighboring *symmetrical* operations, but not with adjacent asymmetrical operations. Finally, opaque operations (e.g., sorting and argmax) cannot be fused with different operators given their unique characteristics.

One important detail that *Sage* considers in the fusing process is in identifying CO nodes whose output needs to reside in the memory. Model gradients or outputs with multiple dependencies may fall into this category. For these, *Sage* assures that nodes are either not fused or at least kept at the last part of the fused operator to secure the output.

3.2.2 Subgraph Reduction. *Sage* conducts a subgraph reduction phase upon completing operator fusion to pare the graph size further. Note that a computation graph generated from automatic differentiation commonly induces complex dependencies between CO nodes, possibly limiting the impact of operator fusion. Subgraph reduction further aggressively simplifies the CO graph via inline rematerialization, a common compiler optimization technique to

curtail memory operations by recomputing certain values instead of storing them in memory. This approach allows for an aggressive node reduction in the CO graph and contributes to a significant reduction in memory use.

Sage implements this concept by deliberately neglecting nodes with multiple dependencies during operator fusion that satisfies three conditions: (i) the node is a symmetrical operation, (ii) if the node is a variadic fused operation, its input number should be less than K_w, and (iii) if the node is a monadic fused operation, its fusion count should be less than K_h. Conditions (ii) and (iii) prevent memory bandwidth-heavy or computation-heavy operations from being rematerialized. K_w and K_h are both set to 5 in our work. We show an example of this operation in Figure 6 (c).

While subgraph reduction seemingly trades-off memory space and latency, this approach has empirically shown improved latency on mobile hardware due to the limited memory bandwidth, as in-line recomputation is often faster than reading a computed value from memory.

3.2.3 Just-in-time Compilation. To realize the benefits of graph-level optimization, operations must be compiled and executed as a single GPU kernel. Considering mobile platform heterogeneity, compiling CO graphs into kernel codes prior to deployment is not ideal, since device-specific optimizations are essential in generating efficient GPU kernel codes. Specifically, the compiling process should consider factors such as workgroup size selection and shared memory configurations based on underlying GPU specifications.

For this reason, *Sage* performs just-in-time (JIT) compilation of CO graphs via OpenCL. First, a high-level OpenCL code fragment is generated for each CO node by translating the CO operation with predefined code templates. This process includes adding index translation code based on tensor memory layouts. Next, based on the graph optimization information, kernel code is generated by merging code fragments from fused and rematerialized nodes. Finally, the generated kernel code is compiled into machine code (i.e., SPIR-V) by the system's OpenCL library. As a result, the compilation process yields a kernel code graph, which is a collection of compiled codes with execution dependencies. We note that JIT compilation induces some overhead (e.g., 260 ms for compiling ResNet-50 on Samsung Galaxy Note 20) during model initialization, but its impact on the overall performance is neglectable since the compiled kernels can be cached.

3.3 Operator-level Optimizations

The memory footprint of some computation-heavy CO nodes, such as the matrix multiplication and convolution, are manually optimized in *Sage*. For example, it is common to implement GPU matrix multiplication with additional memory copy operations (e.g., transpose and pad) for better vectorization performance, and image-to-column (im2col) algorithms are frequently used to implement convolutions. However, such utilization of memory resources over (slightly) better latency may not be preferable for mobile platforms with harsh memory constraints. Therefore, we employ a single-kernel implementation [27] for matrix multiplication and convolution that does not induce additional memory overheads.

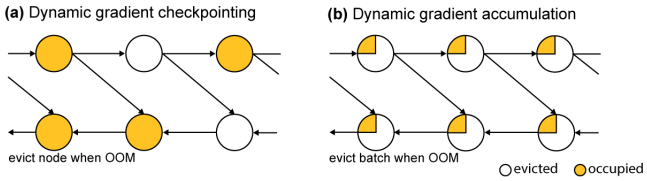

Figure 7: Illustration of dynamic gradient checkpointing and dynamic gradient accumulation strategies.

Memory transfer also occurs commonly when memory copy operations are explicitly issued by tensor reshaping CO (e.g., transpose and permutation). Several sophisticated DNN components, such as attention mechanisms [41], exploit a noticeable amount of tensor reshaping to perform complex tensor operations. Naturally, these operations increase memory use in a DNN training process. Thus, *Sage* aggressively employs index translation within the kernel to in-line compute results without memory copies. We note that using direct tensor operations over composite tensor operations can cause a slight increase in computation time. However we argue that given the tight memory constraints on mobile devices, reducing the memory overhead should be prioritized over computation speed. We discuss the latency impact of this process in Section 4.

3.4 Run-time Memory Management

The aforementioned optimizations focus on building memory-efficient computation graphs. Nevertheless, even an optimized graph still possesses complicated node dependencies, resulting in excessive memory use. Furthermore, the heterogeneous and dynamic memory availability prevalent in mobile platforms should be carefully considered, which is challenging to address using only graph-level optimization.

We now discuss a run-time solution to optimize memory consumption during graph execution. As we will detail, this phase is essential for *Sage* to adjust and adapt to extremely scarce and dynamic memory budgets. We start by introducing the two core concepts, namely, dynamic gradient checkpointing and dynamic gradient accumulation, then introduce a hybrid approach that suits the constraints of mobile/embedded computing environments.

3.4.1 Dynamic gradient checkpointing. During graph execution, nodes are visited in topological order. However, due to the complex dependencies caused by automatic differentiation where nodes with low topological order (e.g., input data) connect to high-order nodes (e.g., gradient of the first model layer), all intermediate states are destined to be retained in the memory, resulting in significant memory consumption. As Figure 7 (a) illustrates, gradient checkpointing alleviates this overhead by storing only a subset of intermediate states (i.e., checkpoints) in the memory. Those not stored are recomputed on-demand. This approach can be extremely effective in minimizing the memory consumption of model training operations.

Among a handful of work that target to identify the ideal set of nodes to remain in the memory (and which to drop), *Sage* selectively employs the on-line checkpointing approach, as it offers high flexibility in managing memory consumption at runtime [21]. Specifically, using this scheme, *Sage* retains all intermediate computation results in the memory until it meets a memory usage

threshold. If the storing of the next intermediate state exceeds the threshold, *Sage* will greedily (and iteratively) evict memory contents by identifying lowest heuristic value items. *Sage* takes the *computational cost per bit* as its heuristic to assess the "memory worthiness" and among all states stored on the memory, the one with the lowest *computational cost / tensor size* is victimized as this leaves the least recomputation overhead. Note that we use computation latency when evaluating the computational cost of a tensor. For this, we record the latency for all tensor computations. The current implementation of *Sage* only focuses on GPU operations, and we leave the use of heterogeneous embedded processors (e.g., NPU) and employing cost values based on energy consumption as part of future work.

3.4.2 Dynamic gradient accumulation. As illustrated in Figure 7 (b), gradient accumulation, also known as micro-batching, splits the original mini-batch into smaller micro-batches. The use of smaller batch sizes result in less (instantaneous) memory use; thus, is effective in reducing peak memory requirements of model training. Once all micro-batch computations complete, per-iteration accumulated results are used to recover the mini-batch gradient.

The key to applying gradient accumulation is to determine the micro-batch size. We devise and incorporate a simple algorithm that dynamically adjusts micro-batch sizes to the available memory budget. Note that a large micro-batch offers minimal memory reduction, while a too-small micro-batch may under-utilize the GPU. Thus, *Sage* targets to identify the maximum possible micro-batch size that it can support given its current memory availability. Once the memory is scarce, a single batch column for all intermediate states in the memory is collectively evicted from the memory (c.f., Fig. 3 (b)) until the target budget is met. Micro-batched gradients are stored, and the graph evaluation process iteratively continues until all micro-batch gradients accumulate.

3.4.3 Hybrid strategy in Sage. While gradient checkpointing and accumulation can theoretically be applied together, generating an optimal hybrid approach is not straightforward, given the distinct memory-latency tradeoff characteristics of different strategies. Take a scenario in which nodes are asked to be evicted from the memory. There can be many ways in which the two strategies coalesce. For example, they may operate on separately allocated memory spaces, or be applied simultaneously, or may prioritize one over the other. While these options may lead to similar memory reduction, their latency overhead can significantly differ.

Sage employs a hybrid strategy of combining dynamic gradient checkpointing and dynamic gradient accumulation based on an empirical observation made for mobile GPUs. Note that the major concern of aggressive micro-batching is in under-utilizing the GPU. We noticed that for mobile GPUs, under-utilization only occurs when the micro-batch is relatively small (e.g., ≤6 for ResNet-50) due to computing power limitations. Thus, it makes sense to aggressively exploit gradient accumulation until reaching a pre-defined tipping point as memory reduction can come at no cost.

Based on such observation, we design *Sage* to prioritize dynamic gradient accumulation until reaching the GPU under-utilization point (i.e., *MTP*). Then, *Sage* employs gradient checkpointing for further memory reduction (c.f., Algo. 1 ; Fig. 3 (f)). We show through our evaluations that this hybrid approach is highly efficient for mobile devices compared to other possible options.

Algorithm 1 *Sage*'s runtime memory management

Require: G ▷ List of gradient nodes in the graph
Require: $batch$ ▷ Mini-batch size
$Q \leftarrow \text{BinaryHeap}(G)$ ▷ Sorted in topological order
$D \leftarrow \text{Set}$ ▷ Prevents infinite loop in checkpointing
$m \leftarrow batch$
while $batch \geq 0$ **do**
  while $node \leftarrow \text{Pop}(Q)$ **do**
    if $node \in memory$ **then continue**
    $\text{Insert}(D, \text{Parents}(node))$
    if $\text{Parents}(node) \subset memory$ **then**
      while not sufficient memory **do**
        if $m \geq MTP$ **then** ▷ Accumulation
          free all last batches in memory
          $m \leftarrow m - 1$
        else ▷ Checkpointing
          $C \leftarrow memory \cap D^C$
          free $c \in C$ with the smallest $\frac{\text{Latency}(c)}{\text{Memory}(c)}$
        end if
      end while
      $\text{Compute}(node)$
      $\text{Remove}(D, \text{Parents}(node))$
    else
      $\text{Push}(Q, node)$
      $\text{Push}(Q, \text{Parents}(node))$
    end if
  end while
  $batch \leftarrow batch - m$
end while

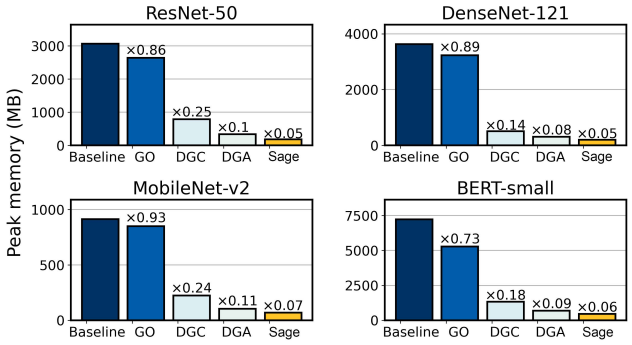

Figure 8: Minimum memory requirements for baseline, graph optimization (GO), gradient checkpointing (DGC), gradient accumulation (DGA) and *Sage*.

4 EXPERIMENT AND EVALUATION

4.1 Experimental Settings

We implement *Sage* with ~15K lines of Rust code [1], and evaluate its performance using four deep learning models that are widely-used

[1] https://github.com/eis-lab/sage

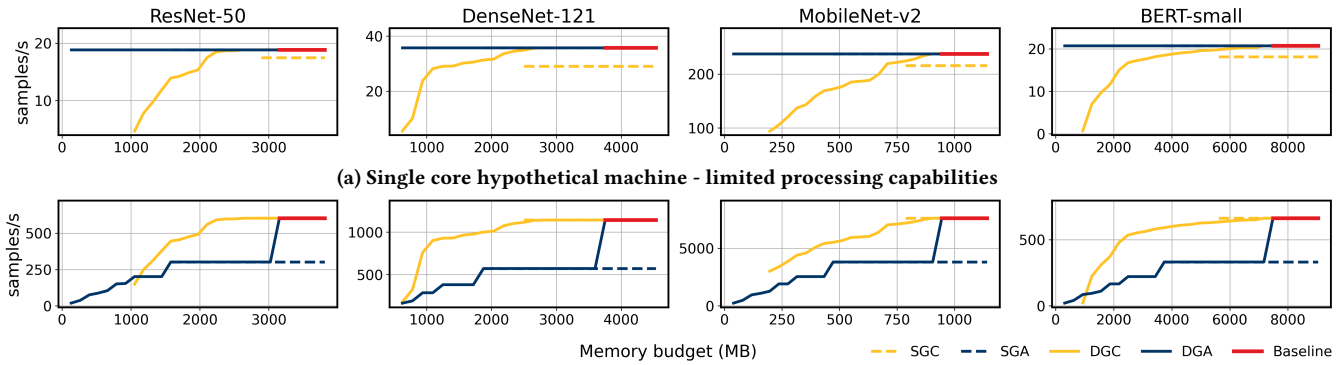

Figure 9: Theoretical computation throughput of various gradient accumulation and gradient checkpointing schemes for two (extreme) hypothetical computing environments.

in natural language processing (i.e., BERT-small [6]) and computer vision (i.e., ResNet-50 [10], DenseNet-121 [12], MobileNet-v2 [36]). We implement the DO graph of each model with parameters taken from the original work and employ input data of shape (64, 512) for BERT-small, and use (3, 224, 224) for other models. For all cases, the mini-batch size is set to 32. Specifically, we evaluate *Sage* using three platforms: (i) Samsung Galaxy S10 (Exynos 9820) with Mali-G76 GPU and 8 GB RAM, (ii) Samsung Galaxy Note 20 (Snapdragon 865) with Adreno 650 GPU and 8 GB RAM, and (iii) an NVIDIA RTX 3090 GPU server. Smartphone implementations run on Android 11.

4.2 Memory Requirements for Training

We start our evaluations by presenting results on how much memory is required for training deep learning models when applying *Sage*. Figure 8 compares the minimum memory requirements for training the four deep learning models of our interest with different configurations. Specifically, here, we compare *Sage* with a baseline approach (no memory management), graph and operator optimizations (GO; Sec 3.3 and Sec 3.2), dynamic gradient checkpointing (DGC; Sec 3.4.1), and dynamic gradient accumulation (DGA; Sec 3.4.2).

As the plots show, *Sage* significantly reduces the required memory for DNN model training (20-fold reduction compared to baseline for ResNet-50 and DenseNet-121 and more than 13-fold for all tested cases). We can also notice that compared to graph optimization schemes, run-time optimization methods, especially dynamic gradient accumulation, aggressively reduces memory usage. From these plots, we can conclude that by effectively combining the individual strategies, *Sage* reduces the memory requirements for model training to suit the tight memory availability of mobile platforms.

4.3 DNN Training Performance

We now present benchmark evaluation results on the performance of *Sage* in training various DNNs. Here, we focus on examining the impact of different memory-reduction techniques used in *Sage* in greater detail.

4.3.1 Comparison methods. We exploit the five following configurations to compare the run-time memory management performance *Sage*. First, we consider a baseline approach where no memory optimization is used. Second, we implement static gradient checkpointing (SGC) in which only one sample every $\sqrt{n}$ samples are stored in the memory [3]. Third, we also implement a static gradient accumulation (SGA) scheme, where we configure the batch size to $\frac{1}{2}$ of the original (i.e., 16). Finally, we present results for dynamic gradient checkpointing (DGC) and dynamic gradient accumulation (DGA), integrated in *Sage*, separately.

4.3.2 Performance on hypothetical platforms. Prior to presenting experimental results, for a concrete and comprehensive understanding of how *Sage* exploits mobile hardware characteristics to support complex DNN training, we configure an emulation environment that exploits two extreme hypothetical devices: (i) a single core machine with limited computing power and (ii) a machine with infinite cores of ideal processing capability and maximum parallelism. In Figure 9 we present the computational throughput (in samples per second) for different comparison methods with varying (statically configured) memory availability.

From the trends of the plots, we make a few interesting observations. First, from the fact that the plots for static approaches (SGA, SGC) are shorter than those of their dynamic counterparts, we can notice that static schemes offer less memory reduction benefits than dynamic approaches. This is due mainly to the fact that it is challenging to identify static parameters prior to run-time. Next, for cases when the processing capabilities are limited (Fig. 9 (a)), DGC shows superior performance over DGA, as no overhead is induced in split-computing the mini-batch. On the other hand, with ideal processing power (Fig. 9 (b)), DGC shows a better performance. These results, suggest that DGA should be prioritized in computing environments with less parallel processing capability, whereas DGC should be preferred with powerful processors. Note that we did not test with *Sage* in this configuration. This is due to the fact that when using these extreme conditions, *Sage* will naturally merge to the best performing option, as *Sage* takes a hybrid approach.

4.3.3 Experimental benchmark. Figure 10 presents the same evaluations as above, this time, on three real hardware platforms, Samsung Galaxy S10 and Note 20 together with the NVIDIA RTX 3090. Given that mobile GPUs show less parallelism compared to the RTX 3090

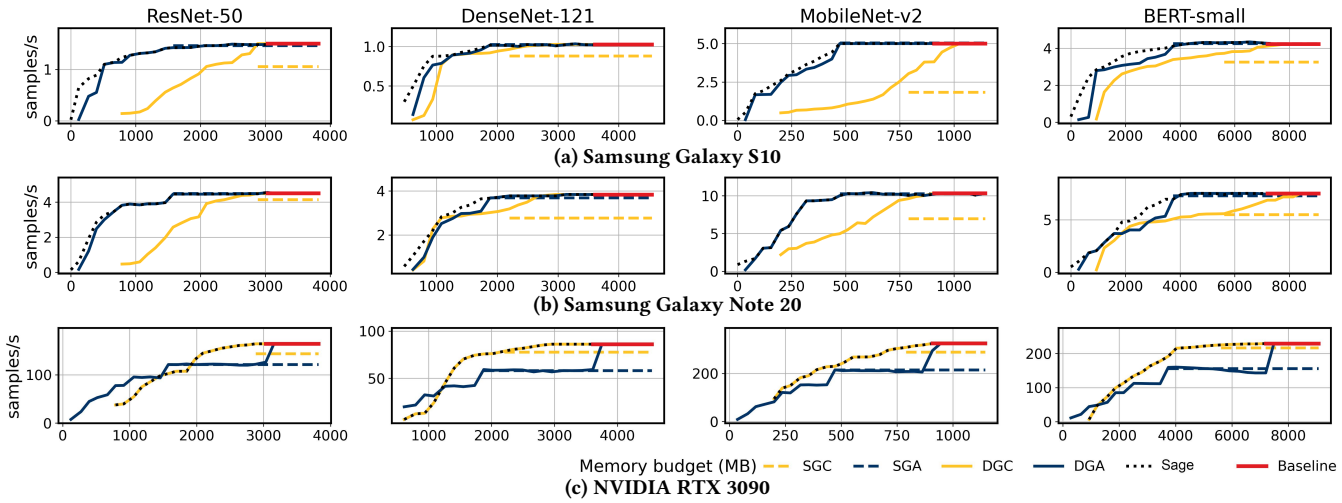

Figure 10: Empirical throughput performance for different runtime memory management schemes compared to *Sage* on the Samsung Galaxy S10, Note 20 and the NVIDIA RTX 3090.

(~100× difference in processing speed), the performance trends agree with our findings from the emulations in Figure 9. Here, the computing performance of RTX 3090 closely resembles the high-processing power environment, while the performance of the two mobile GPUs generally follows the limited computing power case. Nevertheless, note that when provided with sufficient memory, *Sage* shows DGA-like performance, but once the amount of available memory becomes scarce, *Sage* issues checkpointing operations, to achieve a more memory efficient solution. As a result, when observing the minimum required memory for model training, on mobile devices, we can see that *Sage* enables models to be trained even with extremely low memory availability (e.g., 50% reduction compared to the maximum reduction of DGA and 420% to DGC in ResNet-50). In terms of throughput, *Sage* can process up to 7 samples (of model training) per second for BERT-small. This implies that *Sage* successfully achieves its initial design goal of offering a memory-efficient deep neural network training environment for mobile computing platforms.

4.3.4 Performance impact of graph and operator optimization. Next, we take a deeper look into the impact of applying graph- and operator-level optimization strategies used in *Sage* using Table 1. Here, we take the four models of our interest and present number of nodes in the resulting CO-graph, memory consumption and processing latency on the Samsung Galaxy S10 with different optimizations enabled. Starting from operator-level optimization, replacing composite tensor operations with direct operations (Direct) shows a small decrease in the CO node count, and all four models exhibit effective memory reduction at the cost of (marginal) added computation latency. With operation fusion (OF), all models show noticeable improvements in reducing both memory usage and latency. Subgraph reduction (SR), a more aggressive fusing scheme, amplifies these benefits. *Sage*, which combines all three operations, achieves both memory and latency reductions by effectively curtailing memory copy operations and fusing adjacent operations so that the number of computation-needing components are minimized.

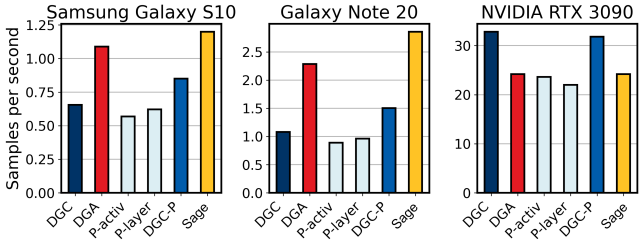

Figure 11: Throughput comparison of different DGA-DGC hybrid strategies.

4.3.5 Evaluation on different runtime optimization approaches. Finally, we focus on evaluating *Sage*'s run-time optimization scheme. As mentioned in Section 3.4, gradient accumulation and checkpointing strategies can be combined in different ways. While *Sage* prioritizes gradient accumulation until reaching the GPU under-utilization point, then applies gradient checkpointing, other combinations can also be possible. In Figure 12 we present the average processing throughput of the ResNet-50 model on three platforms for six different hybrid approaches: (i) DGC only, (ii) DGA only, (iii) half of memory drop using DGC and the other half with DGA , (iv) DGA-DCG used interchangably, (v) DGC prioritized, and (vi) *Sage* - DGA prioritized. The plots show that among all cases, *Sage* shows the highest throughput performance on mobile platforms. Interestingly, on the RTX 3090, fully exploiting and prioritizing gradient checkpointing shows better performance. This is because the cost of recomputing gradients is less significant given the computing resources available on server-scale GPUs. Nevertheless, our evaluations on different hybrid approaches show that *Sage* is the most suitable approach for mobile platforms.

4.4 Performance with Dynamic Memory

Given a reduced computational graph from the graph optimization phase, the goal of run-time memory optimization is to adapt the

Model	ResNet-50			DenseNet-121			MobileNet-v2			BERT-small		
	CO nodes	Memory usage (MB)	Latency (ms)	CO nodes	Memory usage (MB)	Latency (ms)	CO nodes	Memory usage (MB)	Latency (ms)	CO nodes	Memory usage (MB)	Latency (ms)
Original	1762	3303	1161	5120	3607	1511	1478	908	290	2509	7197	572
Direct	1699	3147 (-4.7%)	1240 (+6.8%)	4912	3408 (-5.5%)	1677 (+10.9%)	1320	867 (-4.5%)	323 (+11.3%)	2275	6578 (-8.6%)	602 (+5.2%)
Direct+OF	760	2903 (-12.1%)	760 (-34.5%)	3895	3253 (-9.8%)	1226 (-18.8%)	791	846 (-6.8%)	237 (-18.0%)	1730	5368 (-25.4%)	380 (-33.5%)
Direct+OF+SR	650	2857 (-14.5%)	685 (-41.0%)	3330	3207 (-11.1%)	978 (-35.3%)	629	838 (-7.7%)	207 (-28.8%)	1547	5246 (-27.1%)	335 (-41.5%)

Table 1: CO graph size, memory and latency impact of different graph and operator optimization used in *Sage*.

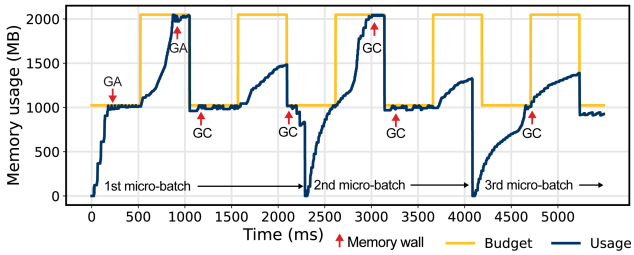

Figure 12: Time-series trace of *Sage*'s memory usage with dynamic memory availability.

training process towards the dynamic memory budgets of a mobile computing environment. To confirm that *Sage* performs properly (and well) under such dynamics, we configure the Galaxy S10 with ResNet-50 and dynamically vary the amount of available memory. In Figure 12 we present the memory use trace and memory management strategy applied at each point for one mini-batch training iteration. Specifically, in the initial phase, no micro-batch is defined (e.g., batch size = 32) until approaching the first memory capacity wall. At this point, gradient accumulation issues for a micro-batch size reduction (e.g., batch split to 20 + 12). When the next memory wall is met, the micro-batch is reduced once more (e.g., 14 + 14 + 4). At memory walls that follow, *Sage* determines that further micro-batch reduction would cause GPU under-utilization and applies gradient checkpointing for the remainder of the iteration. As this example shows, *Sage* successfully manages memory usage with respect to the available memory resources.

4.5 Energy Consumption

Finally, Figure 13 plots the per-training iteration energy consumption on the Galaxy S10 and Note 20 for the four models of our interest. For all cases, we present results for different memory budgets and use the BatteryHistorian power monitoring tool for measurements [5]. Note that we measure and deduct the base energy usage of the OS to extract *Sage*'s energy use. Given that the Galaxy S10 is equipped with a battery of ~60,000 Joules and with the ResNet-50 consuming ~4.9 J per training iteration, a 1,000-iteration training operation (enough for model fine-tuning) with ResNet-50 will result in consuming 8% of the entire battery. These results suggest that while not explicitly optimized for minimal energy consumption, *Sage* is still suitable as an practical on-device DNN training framework for mobile platforms.

5 DISCUSSIONS

Our evaluations indicate that *Sage* effectively allows the training of memory demanding state-of-the-art complex deep neural network models on mobile computing platforms. Based on our experiences in designing, implementing and evaluating *Sage*, we outline some interesting discussion points that can potentially lead to meaningful future research below.

• **Thermal constraints**: An important factor that we do not consider is the thermal profile of the mobile platform when on-device DNN training takes place. As the results in Section 4.3.3 show, despite *Sage* effectively managing memory contents to realize DNN training, the process still can take a noticeable amount of time and energy. This opens the possibility of the device's thermal throttling to function. We see such effect as a possibility and plan to address such issue as part of future work.

• **Impact on mobile user experience**: To enable on-device complex DNN training, *Sage* is designed to fully exploit all available resources on the mobile platform. However, if we consider the case where on-device training is used as a background service of an application, the exhaustive use of computing/memory resources can easily impact the user experience. While out of the scope for this work, we see this as an important next research direction.

• **Handling batch normalization**: Through our experiments, we showed that dynamic gradient accumulation is highly efficient in reducing the training memory budgets in mobile platforms. However, for DNN architectures that employ batch-wise regularization methods (e.g., batch normalization), gradient accumulation approaches can interfere with the training results. For this, a simple workthrough is to compute and store the entire mini-batch mean and variance data in *inference mode* prior to each training iteration, and exploit this stored data in the following training operations. This will add latency overhead induced by the computing of mini-batch statistics separately, but will not affect the overall training memory requirements.

• **Low-precision training**: Low-precision floating points operations can coalesce with *Sage* for additional memory reduction. While not included as part of our evaluations, our elementary (and incomplete) implementation of half-precision (FP16) training suggests that ~40% linear reduction can be made in overall memory consumption. While not included in the scope of this work, where we target to keep the original precision of the model, we plan to add a complete half-precision feature in our open source software of *Sage*.

6 RELATED WORK

The task of proposing a memory-efficient DNN training framework for mobile platforms has not been actively studied in previous work. In Section 2.3, we presented a number of memory reduction techniques that can be (or have been) applied to model training operations. As we show throughout this work, server-scale GPUs

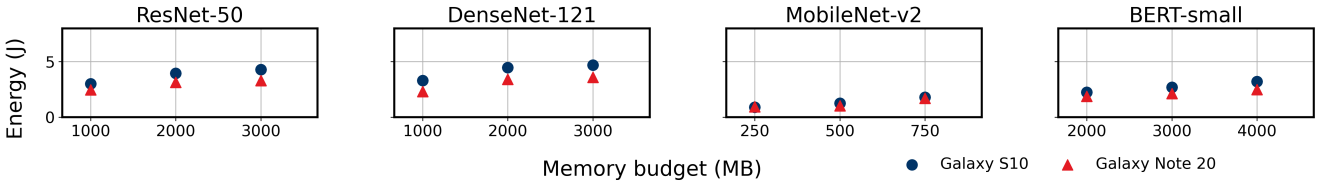

Figure 13: Per-training iteration energy consumption on the Galaxy S10 and Note 20 for four DNN models.

and those equipped on mobile platforms show noticeably different performance characteristics. Specifically, the assumptions that previously proposed schemes are based on do not hold; thus, they cannot be applied naively for mobile on-device DNN training.

While not being memory saving mechanisms themselves, mobile deep learning frameworks such as TensorFlow Lite [40], Alibaba MNN [18], and Apple's Core ML [16] are designed to support some basic level of on-device training. We note that the implementation of *Sage* on such industry-scale mobile DNN frameworks can potentially enhance the system's real-world applicability. Unfortunately, at this point, many components of *Sage* (e.g., graph-level optimizations and runtime memory management) require considerable design modifications for full integration. Furthermore, some frameworks implement internal optimizations (e.g., tensor pooling) that can interfere with our design hypotheses. Thus, we believe that coalescing the proposed memory-reducing optimizations in *Sage* with existing latency-reducing framework designs will be the next step to democratize on-device training.

7 SUMMARY

This work targets to address the challenge of training complex deep learning models on today's mobile computing platforms. When made possible, we believe that the consequences of on-device model training is significant. A mobile device can lessen its dependency towards powerful servers and privacy-sensitive data can remain local while fully enjoying the capabilities of precise and personalized neural networks. We present *Sage* as a solution to realize this goal. Specifically, we identify memory budget limitations as the most significant hurdle and propose *Sage* as a framework for dynamically managing the memory usage in on-device model training scenarios. Specifically, *Sage* reforms the computational graph for gradient evaluation with minimal memory transfer operators and performs operator fusion and subgraph reduction to reduce the graph size. Using a combination of gradient accumulation and checkpointing, *Sage* also adapts its memory use with respect to the dynamic memory budgets on mobile platforms. Extensive evaluations with real implementations of *Sage* on the Samsung Galaxy S10 and Note 20 indicate that *Sage* is robust enough to train heavy state-of-the-art DNNs, such as ResNet-50 and BERT-small, which were previously infeasible using mobile GPUs. We foresee *Sage* as an important milestone in catalyzing a rich set of mobile applications that fully exploit the capabilities of highly accurate neural networks.

ACKNOWLEDGEMENTS AND NOTES

The authors would like to thank our shepherd, Professor Mi Zhang, and the anonymous reviewers for their valuable feedback on the work. This work was supported by the Ministry of Science and ICT's NRF Basic Science Research Program (2021R1A2C4002380), IITP (IITP-2022-2022-0-00240), ITRC Program supervised by IITP (IITP-2021-2020-0-01461), Ministry of Culture, Sports and Tourism and Korea Creative Content Agency (R2021040018), and by the Ministry of Trade, Industry and Energy and KIAT through the International Cooperative R&D program (P0016150). In Gim submitted this work as Hyunjun Kim. JeongGil Ko is the corresponding author for this work (jeonggil.ko@yonsei.ac.kr).

REFERENCES

[1] 2020. ML Commons Benchmark: Mobile Inference v0.7 Results. https://mlcommons.org/en/inference-mobile-07/

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[3] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *CoRR* abs/1604.06174 (2016). arXiv:1604.06174 http://arxiv.org/abs/1604.06174

[4] Tim Dettmers and Luke Zettlemoyer. 2019. Sparse Networks from Scratch: Faster Training without Losing Performance. *CoRR* abs/1907.04840 (2019). arXiv:1907.04840 http://arxiv.org/abs/1907.04840

[5] Android Developers. 2013. Profile battery usage with Batterystats and Battery Historian. https://developer.android.com/topic/performance/power/setup-battery-historian.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[7] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 17. https://doi.org/10.1109/SC41405.2020.00021

[8] Taesik Gong, Yeonsu Kim, Jinwoo Shin, and Sung-Ju Lee. 2019. MetaSense: few-shot adaptation to untrained conditions in deep mobile sensing. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems, SenSys 2019, New York, NY, USA, November 10-13, 2019*, Raghu K. Ganti, Xiaofan Fred Jiang, Gian Pietro Picco, and Xia Zhou (Eds.). ACM, 110–123. https://doi.org/10.1145/3356250.3360020

[9] Andreas Griewank and Andrea Walther. 2000. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Softw.* 26, 1 (2000), 19–45. https://doi.org/10.1145/347837.347846

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[11] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 1341–1355. https://doi.org/10.1145/3373376.3378530

[12] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2261–2269. https://doi.org/10.1109/CVPR.2017.243

[13] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and

Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 103–112. https://proceedings.neurips.cc/paper/2019/hash/093f65e080a295f8076b1c5722a46aa2-Abstract.html

[14] Sinh Huynh, Rajesh Krishna Balan, and JeongGil Ko. 2022. IMon: Appearance-Based Gaze Tracking System on Mobile Devices. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 5, 4, Article 161 (dec 2022), 26 pages. https://doi.org/10.1145/3494999

[15] Sinh Huynh, Rajesh Krishna Balan, JeongGil Ko, and Youngki Lee. 2019. VitaMon: Measuring Heart Rate Variability Using Smartphone Front Camera. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems* (New York, New York) *(SenSys '19)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3356250.3360036

[16] Apple Inc. 2013. CoreML Documentation. https://developer.apple.com/documentation/coreml.

[17] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph Gonzalez. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/book/320.pdf

[18] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lyu, and Zhihua Wu. 2020. MNN: A Universal and Efficient Inference Engine. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/book/287.pdf

[19] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor N. Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Yunji Chen, Olivier Temam, and John Carter (Eds.). ACM, 615–629. https://doi.org/10.1145/3037697.3037698

[20] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2021. Dynamic Tensor Rematerialization. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=Vfs_2RnOD0H

[21] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2021. Dynamic Tensor Rematerialization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Vfs_2RnOD0H

[22] Mitsuru Kusumoto, Takuya Inoue, Gentaro Watanabe, Takuya Akiba, and Masanori Koyama. 2019. A Graph Theoretic Framework of Recomputation Algorithms for Memory-Efficient Backpropagation. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 1161–1170. https://proceedings.neurips.cc/paper/2019/hash/e555ebe0ce426f7f9b2bef0706315e0c-Abstract.html

[23] Wei Yang Bryan Lim, Nguyen Cong Luong, Dinh Thai Hoang, Yutao Jiao, Ying-Chang Liang, Qiang Yang, Dusit Niyato, and Chunyan Miao. 2020. Federated Learning in Mobile Edge Networks: A Comprehensive Survey. *IEEE Commun. Surv. Tutorials* 22, 3 (2020), 2031–2063. https://doi.org/10.1109/COMST.2020.2986024

[24] Ian McGraw, Rohit Prabhavalkar, Raziel Alvarez, Montse Gonzalez Arenas, Kanishka Rao, David Rybach, Ouais Alsharif, Hasim Sak, Alexander Gruenstein, Françoise Beaufays, and Carolina Parada. 2016. Personalized speech recognition on mobile devices. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2016, Shanghai, China, March 20-25, 2016*. IEEE, 5955–5959. https://doi.org/10.1109/ICASSP.2016.7472820

[25] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA (Proceedings of Machine Learning Research, Vol. 54)*, Aarti Singh and Xiaojin (Jerry) Zhu (Eds.). PMLR, 1273–1282. http://proceedings.mlr.press/v54/mcmahan17a.html

[26] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=r1gs9JgRZ

[27] Cedric Nugteren. 2018. CLBlast: A Tuned OpenCL BLAS Library. In *Proceedings of the International Workshop on OpenCL, IWOCL 2018, Oxford, United Kingdom, May 14-16, 2018*, Simon McIntosh-Smith and Ben Bergen (Eds.). ACM, 5:1–5:10. https://doi.org/10.1145/3204919.3204924

[28] HyeonJung Park, Jong-Seok Lee, and JeongGil Ko. 2020. Achieving Real-Time Sign Language Translation Using a Smartphone's True Depth Images. In *2020 International Conference on COMmunication Systems NETworkS (COMSNETS)*. 622–625. https://doi.org/10.1109/COMSNETS48256.2020.9027420

[29] HyeonJung Park, Youngki Lee, and JeongGil Ko. 2021. Enabling Real-Time Sign Language Translation on Mobile Platforms with On-Board Depth Cameras. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 5, 2, Article 77 (jun 2021), 30 pages. https://doi.org/10.1145/3463498

[30] Jaeyeon Park, Hyeon Cho, Rajesh Krishna Balan, and JeongGil Ko. 2020. HeartQuake: Accurate Low-Cost Non-Invasive ECG Monitoring Using Bed-Mounted Geophones. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 3, Article 93 (sep 2020), 28 pages. https://doi.org/10.1145/3411843

[31] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 891–905. https://doi.org/10.1145/3373376.3378505

[32] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning. In *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 59:1–59:14. https://doi.org/10.1145/3458817.3476205

[33] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 18:1–18:13. https://doi.org/10.1109/MICRO.2016.7783721

[34] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 18:1–18:13. https://doi.org/10.1109/MICRO.2016.7783721

[35] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *CoRR* abs/1805.00907 (2018). arXiv:1805.00907 http://arxiv.org/abs/1805.00907

[36] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. Computer Vision Foundation / IEEE Computer Society, 4510–4520. https://doi.org/10.1109/CVPR.2018.00474

[37] Charles Michael Stein, Dinei A. Rockenbach, Dalvan Griebler, Massimo Torquati, Gabriele Mencagli, Marco Danelutto, and Luiz Gustavo Fernandes. 2021. Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units. *Concurr. Comput. Pract. Exp.* 33, 11 (2021). https://doi.org/10.1002/cpe.5786

[38] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 2158–2170. https://doi.org/10.18653/v1/2020.acl-main.195

[39] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. 2014. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*. IEEE Computer Society, 1701–1708. https://doi.org/10.1109/CVPR.2014.220

[40] TensorFlow. 2022. TensorFlow Lite: ML for Mobile and edge devices. https://www.tensorflow.org/lite.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[42] Jindong Wang, Yiqiang Chen, Shuji Hao, Xiaohui Peng, and Lisha Hu. 2019. Deep learning for sensor-based activity recognition: A survey. *Pattern Recognit. Lett.* 119 (2019), 3–11. https://doi.org/10.1016/j.patrec.2018.02.010

[43] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, Andreas Krall and Thomas R. Gross (Eds.). ACM, 41–53. https://doi.org/10.1145/3178487.3178491

[44] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. 2018. Recent Trends in Deep Learning Based Natural Language Processing [Review Article]. *IEEE Comput. Intell. Mag.* 13, 3 (2018), 55–75. https://doi.org/10.1109/MCI.2018.2840738

[45] Shuai Yu, Xin Wang, and Rami Langar. 2017. Computation offloading for mobile edge computing: A deep learning approach. In *28th IEEE Annual International Symposium on Personal, Indoor, and Mobile Radio Communications, PIMRC 2017, Montreal, QC, Canada, October 8-13, 2017*. IEEE, 1–6. https://doi.org/10.1109/PIMRC.2017.8292514