Systems for Machine Learning on Edge Devices

By

Shishir Girishkumar Patil

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph Edgar Gonzalez, Chair
Professor Prabal Dutta

Fall 2023

The thesis of Shishir Girishkumar Patil, titled Systems for Machine Learning on Edge Devices, is approved:

Chair _____ Date _____

_____ Date _____

University of California, Berkeley

Systems for Machine Learning on Edge Devices

Abstract

Systems for Machine Learning on Edge Devices

by

Shishir Girishkumar Patil

Master of Science in Computer Science

University of California, Berkeley

Professor Joseph Edgar Gonzalez, Chair

Modern edge applications increasingly rely on Machine Learning (ML) based predictions. ML models deployed in the real world rapidly degrade in quality due to the evolution of data and our interpretation of data with time. Cloud applications combat model staleness by retraining models frequently. However, this is challenging on Edge devices such as smartphones and wearables, since they are characterized by memory and energy constraints, while the ML models continue to grow bigger. To overcome this, we propose two complementary systems POET and Minerva.

POET is an algorithm designed to facilitate the training of large neural networks on memory-scarce, battery-operated edge devices. It optimizes the integrated search spaces of rematerialization and paging, two techniques that significantly reduce the memory requirements of backpropagation. By formulating a mixed-integer linear program (MILP), POET achieves energy-optimal training within given memory and runtime constraints. This approach not only allows for training substantially larger models on embedded devices but also enhances energy efficiency without compromising the mathematical correctness of backpropagation.

Complementing POET, Minerva offers an end-to-end ML model update system specifically tailored for microcontroller-based devices. At its core is a novel system abstraction known as capsules, which facilitates efficient and flexible ML model updates without necessitating disruptive full device firmware updates (DFUs). These capsules exploit the pure function nature of ML model inference, overcoming the challenges typically associated with updating parts of a running program. Recognizing the absence of ground-truth labels, Minerva presents a novel technique to evaluate deployed models. Minerva's efficient updates and the ability to shadow-test enable a wide array of embedded devices to receive regular model updates.

Together, POET enables training models locally, and in those scenarios where training requires the cloud, Minerva enables efficient deployment and testing thereby opening up newer domains of computing to exploit ML models.

To my Parents.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The advent of deep learning has revolutionized edge computing, with models now widely deployed for inference on edge devices such as smartphones and embedded platforms. However, training these models is still predominantly centralized, typically occurring on large cloud servers equipped with high-throughput accelerators like GPUs, TPUs, etc. This centralized training model necessitates the transfer of sensitive data from edge devices to the cloud, compromising user privacy and incurring additional data movement costs in network bandwidth and energy. To counter these issues, on-device training methods like federated learning have emerged, allowing for local training updates without consolidating data to the cloud, as seen in applications like Google Gboard and iPhone's Automatic Speech Recognition (ASR).

Despite these advancements, current on-device training methods struggle to support state-of-the-art models such as large language models (LLMs) due to the limited device memory insufficient for storing activations for backpropagation. To address this, prior work has suggested strategies like paging to auxiliary memory and rematerialization to reduce memory footprint, but these often lead to increased total energy consumption. Our work introduces POET (Power Optimal Edge Training) [36], a novel algorithm that optimally combines paging and rematerialization, scaling effective memory capacity with minimal energy overhead. POET reframes edge training as an optimization problem, enabling the training of sophisticated models like BERT on mobile-class edge devices. This is achieved by formulating an integer linear program (ILP) that discovers optimal schedules on weather to retain in main-memory, page to secondary storage, or rematerialize each later of a neural network resulting in minimal energy consumption. POET also strictly adheres to application specified memory and runtime constraints to accommodate training during device idle times, and is designed to work with existing architectures without approximations.

While training locally enables devices to keep up with personalized experiences, often the ML models need to be updated centrally. This could be because, the model developers came across a newer dataset previously not available, or the functionality of the model changed, or for many other reasons. In such scenarios, it is critical to have an efficient mechanism that would allow these ML models trained on the cloud to be deployed on edge devices, where they are invoked for inference.

Updating ML models on edge devices, especially microcontroller-based (MCU) applications is challenging due to their limited capabilities. Traditional model updates through device firmware updates (DFUs) are bandwidth-intensive and inflexible, often requiring a system-reboot and thereby losing application state. To overcome these limitations, we propose Minerva, an efficient model update system for MCU-based edge devices. Minerva leverages a novel system abstraction called capsules, enabling efficient, state-preserving, and low-bandwidth transparent updates for ML models. Capsules exploit the pure function nature of ML models, allowing updates without the need for reboots. This system dramatically reduces update times compared to standard DFUs and integrates seamlessly with existing ML platforms.

Together, POET and Minerva present an exhaustive paradigm for dealing with the challenges of deploying and maintaining ML models on edge devices. POET addresses the need for efficient on-device training, enabling large models to be trained on devices with stringent memory and energy constraints. Minerva, on the other hand, focuses on the deployment stage of the ML lifecycle, offering a flexible and efficient solution for updating models on MCU-based devices. These systems mark a significant stride in making edge computing more adaptable, robust, and efficient.

# Chapter 2

# POET: Training Neural Networks on Tiny Devices with Integrated Rematerialization and Paging

## 2.1 Introduction

Deep learning models are widely deployed for inference on edge devices like smartphones and embedded platforms. In contrast, training is still predominantly done on large cloud servers with high-throughput accelerators such as GPUs. The centralized cloud training model requires transmitting sensitive data from edge devices to the cloud such as photos and keystrokes, thereby sacrificing user privacy and incurring additional data movement costs.

To enable users to personalize their models without relinquishing privacy, on-device training methods such as federated learning [34] perform local training updates without the need to consolidate data to the cloud. These methods have been widely deployed to personalize keyboard suggestions in Google Gboard [19] and to improve Automatic Speech Recognition (ASR) on iPhones [39].

At the same time, current on-device training methods cannot support training modern architectures and large models. For example, Google Gboard fine-tunes a simple logistic regression model. Training larger models on edge devices is infeasible primarily due to the limited device memory which cannot store activations for backpropagation. A single training iteration for ResNet-50 [20] requires 200× more memory than inference.

Prior work has proposed strategies including paging to auxiliary memory [40] and rematerialization [8, 23, 26] to reduce the memory footprint of training in the cloud. However, these methods result in a significant increase in total energy consumption. The data transfers associated with paging methods often require more energy than recomputing the data. Alternatively, rematerialization increases energy consumption at a rate of $O(n^2)$ as the memory budget shrinks.

In this work, we show that *paging and rematerialization are highly complementary*. By

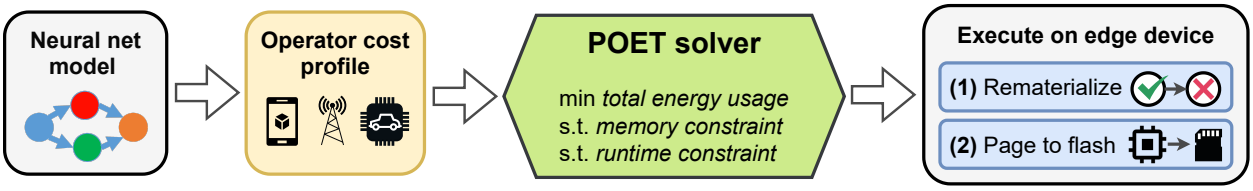


Figure 2.1: POET optimizes state-of-the-art ML models for training on Edge devices. Operators of the ML model are profiled on target edge device to obtain fine-grained profiles. POET adopts an integrated integrated rematerialization and paging to produce an energy-optimal training schedule.

carefully rematerializing cheap operations while paging results of expensive operations to auxiliary memory such as a flash or an SD card, we can scale effective memory capacity with minimal energy overhead. By combining these two methods, we demonstrate it is possible to train models like BERT on mobile-class edge devices. By framing edge training as an optimization problem, we discover optimal schedules with provably minimal energy consumption at a given memory budget. While the focus of this paper is edge deployemnts, the energy objective is increasingly becoming relevant even for cloud deployments [38].

We present POET (Private Optimal Energy Training), an algorithm for energy-optimal training of modern neural networks on memory-constrained edge devices (Fig 2.1). Given that it is prohibitively expensive to cache all activation tensors for backpropagation, POET optimally pages and rematerializes activations, thereby reducing memory consumption by up to 2x. We reformulate the edge training problem as an integer linear program (ILP) and find it is solved to optimality in under ten minutes by commodity solvers.

For models deployed on real-world edge devices, training happens when the edge device is relatively idle and spare compute cycles are available. For example, Google Gboard schedules model updates when the phone is put to charge. Hence POET also incorporates a hard training constraint. Given a memory constraint and the number of training epochs, POET generates solutions that also satisfy the given training deadline. POET transparently develops a comprehensive cost model by profiling the target hardware with the target network's operators. Finally, POET is mathematically value preserving (*i.e,* it makes no approximations), and it works for existing architectures out-of-the-box.

The novel contributions of this work include:

1. A formulation of an integer linear program to find the **energy-optimal** schedule to train modern deep neural networks **with a) memory and b) runtime as hard constraints**.

2. A unified algorithm for hybrid activation recomputation and paging.

3. The first demonstration of how to train ResNet-18 and BERT on tiny Cortex M class devices with memory and timing constraints.

## 2.2 Related Work

Scarcity of compute and memory is one of the largest constraint for machine learning on edge devices. Large models with state-of-the-art performance have largely been exorbitantly expensive for edge devices. The research community has predominantly focused on addressing *inference* on edge devices via methods like efficient DNN architecture [22, 46], quantization [14] or pruning [4].

Instead, we aim to make *training* large neural networks feasible on tiny edge devices. While compute is the limiting resource for inference on the edge, limited memory capacity constraints prevent training large models on the edge. Training via vanilla backpropagation requires caching the output of all intermediate layers (activations). We categorize methods to reduce memory usage of training as activation (1) compression, (2) rematerialization, and (3) paging. We then discuss prior work in energy-efficient training.

**Activation quantization:** ActNN [7], Weighted-Entropy-Based Quantization [35], and others have proposed techniques to quantize activations while performing full-precision multiply-accumulates (MACs). However, these techniques compromise accuracy and correctness. Moreover, poor hardware support for quantized operations under 8 bits limits the practical savings of these techniques. We do not consider methods for pruning during training like The Lottery Ticket Hypothesis [15] as they do not reduce the size of activations.

**Rematerialization:** Rematerialization discards activations in the forward pass and recomputes those values during gradient calculation. Chen et. al (2016) [8] proposed a simple and widely used algorithm for rematerialization where every $O(\sqrt{n})$ layer is retained for the backward pass. Griewank & Walther (2000) [17] propose an optimal algorithm for rematerialization on unit-cost linear auto-diff graphs. However, they force the strong assumption that models have uniform compute requirements across layers. Checkmate [23] identifies the optimal rematerialization schedule for arbitrary static graphs. Monet [43] extends Checkmate with operator implementation selection, but this is orthogonal to our work's scheduling problem. Dynamic Tensor Rematerialization (DTR) [26] finds an approximation of Checkmate that is near-optimal for common computer-vision models. Our work addresses the following limitations of Checkmate: (1) Checkmate does not consider energy nor latency as a constraint and (2) Checkmate does not page activations to secondary memory. POET is the first work that demonstrates provably optimal integrated paging and rematerialization.

**Paging:** SwapAdvisor [21] and DeepSpeed [42] page activations off a memory-scarce GPU to the CPU when out of memory. However, we find paging is very energy-intensive and is often less efficient than rematerialization. Capuchin [40] uses the Memory Saving Per Second (MSPS) heuristic to decide what to page. Only if paging is insufficient will Capuchin rematerialize activations thereby making it sub-optimal as demonstrated in Sec 2.6. POFO [2] formulates finding finding the optimal sequence combining rematerialization and paging as a dynamic programming problem. POFO makes many assumptions that limit generality: POFO only supports chain (linear) model graphs while we support arbitrary graphs such as BERT (Fig 2.3). POFO limits layers to a single rematerialization or page operation while POET can remat/page layers repeatedly. POFO forces all page-out operations to occur prior

| Method | General Graphs | Compute Aware | Memory Aware | Power Aware |
|---|---|---|---|---|
| Checkpoint all (PyTorch) | √ | × | × | × |
| Griewank & Walther (2000) [17] | × | × | × | × |
| Chen et. al (2016) $\sqrt{n}$ [8] | × | × | × | × |
| Chen et. al (2016) greedy [8] | × | × | ∼ | × |
| Checkmate [23] | √ | √ | √ | × |
| POFO [2] | × | √ | √ | × |
| DTR [26] | √ | √ | √ | × |
| POET (ours) | √ | √ | √ | √ |

Table 2.1: Comparison of baseline methods under power, compute, memory and generality metrics. POET satisfies all criteria, enabling end-to-end training on the edge.

to calculating the loss while we have no such restriction. And finally, while POFO assumes paging is asynchronous (e.g., CUDA) but this is not universally true for the edge devices we evaluate. Notice that POET is not only optimizing a different metric (energy vis-a-vis time) but a) adhere's to strict timing guarantees and b) is *provably optimal*.

**Energy-efficient training:** We are not the first to consider energy-optimal training. Prior work on energy-optimal training for the edge either a) required the design of new architectures [5, 46], or proposed b) new techniques of training by dropping activations, updating only select layers of the network, or c) used a different optimizer [51]. Compared to these techniques, POET is a) mathematically value preserving (makes no approximations, or modifications), and b) works for existing and new architectures out-of-the-box.

## 2.3 Background

A growing demand exists for edge machine learning applications for greater autonomy. In response, the community has developed systems to enable machine learning on edge devices. EdgeML [12], CoreML [25] and TensorFlow Lite [29] from Google are all efforts to meet this demand.

However, each of these efforts is application-specific and proposes new algorithms or optimizations to address the computational and memory requirements for machine learning inference on the edge [28]. While inference is already commonly deployed on edge devices, training remains ad-hoc and infeasible for large models.

Training on the edge is critical for privacy, cost, and connectivity. First, due to privacy concerns, many edge applications cannot transmit data to the cloud. Second, the energy

consumed by bulk data transmission can significantly reduce battery life [32].

Third, applications such as ocean sensing and communication [24], and those deployed in farms [49] are designed for offline operations – with no access to the internet.

Our objective of optimizing for energy is a non-trivial contribution. On edge devices, the energy objective can oftentimes conflict with the objective of running to completion. For example, on a given platform, rematerializing might consume lower energy, but paging might be quicker. This is because, on edge devices, it is common practise to turn-off/duty-cycle components that are not utilized (e.g., SD card, DMA, etc.) The energy profile may vary depending on the size of the tensor, and if the PCIe/DMA/SPI/I2C memory bus needs to be activated, etc. Exploiting this enables POET to find the most energy-efficient schedule which would not have been possible had we not optimized for energy.

While definitions differ on which devices are included in "the edge" (e.g., mobile phones, routers, gateways, or even self-driving cars). In the context of this paper, the edge refers to mobile phones and microcontrollers (Table 2.2). These devices are characterized by limited memory (ranging from KBs to a few GBs) and are commonly battery-powered ($\sim$ few hundred mAh) for real-world deployment. Further, in our research we found that it is quite common for these edge devices to be augmented with an off-chip secondary storage such as a flash or an SD card as seen in [49, 24, 37]. This presents us with an opportunity to exploit the off-chip memory for paging.

## 2.4 Integrated paging and rematerialization

Rematerialization and paging are two techniques to lower the memory consumption of large, state-of-the-art ML models. In rematerialization, an activation tensor is deleted as soon as they are no longer needed, most often, during a forward pass. This frees up precious memory that can be used to store the activations of the following layers. When the deleted tensor is needed again, for example, to compute gradients during backpropagation, it is recomputed from the other dependent activations as dictated by the lineage. Paging, also known as offloading, is a complementary technique to reduce memory. In paging, an activation tensor that is not immediately needed is paged-out from the primary memory to a secondary memory such as a flash or an SD card. When the tensor is needed again, it is paged back in.

This is best understood with the representative neural-network training timeline from Figure 2.2. Along the X-axis, each cell corresponds to a single layer of an eight-layered, linear, neural-network. The Y-axis represents the logical timesteps over one epoch. An occupied cell indicates that an operation (forward/backward pass computation, rematerialization, or paging) is executed at the corresponding timestep. For example, we can see that the activation for Layer 1 (L1) is computed at the first timestep (T1). At timestep T2 and T3, the activations of L2 and L3 are computed respectively. Suppose layers L2 and L3 happen to be memory-intensive but cheap-to-compute operators, such as non-linearities (tanH, ReLU, etc,) then rematerialization becomes the optimal choice. We can delete the activations ({T3,

L2}, {T4, L3}) to free up memory, and when these activations are needed during backward propagation we can rematerialize them ({T14, L3}, {T16, L2}).

Suppose layers L5 and L6 are compute-intensive operators such as convolutions, dense matrix-multiplication, etc. For such operations, rematerializing the activations would lead to an increase in run-time and energy and is sub-optimal. For these layers, it is optimal to page-out the activation tensor to secondary storage ({T6,L5}, {T7, L6}), and page-in when they are needed ({T10,L6}, {T11, L5}).

One major advantage of paging is that depending on how occupied the memory bus is, it can be pipelined to hide latency. This is because modern systems have DMA (Direct Memory Access) which can move the activation tensor from the secondary storage to the primary memory while the compute engine is running in parallel. For example, at timestep T7, we are both paging L6 out and computing L7. However, rematerialization is compute-intensive, cannot be parallelized. This leads to an increase in run-time. For example, we have to dedicate timestep T14 to recompute L3 thereby delaying the rest of the backward pass execution.

## 2.5 POET: Private Optimal Energy Training

We introduce Private Optimal Energy Training (POET), a graph-level compiler for deep neural networks that rewrites training DAGs for large models to fit within the memory constraints of edge devices while remaining energy-efficient. POET is hardware-aware and first traces the execution of the forward and backward pass with associate memory allocation requests, runtime, and per-operation memory and energy consumption. This fine-grained profiling for each workload happens only once for a given hardware, is automated, cheap, and provides the most accurate cost model for POET. POET then generates a Mixed Integer Linear Programming (MILP) which can be efficiently solved. The POET optimizer searches for an efficient rematerialization and paging schedule that minimizes end-to-end energy consumption subject to memory constraints. The resulting schedule is then used to generate a new DAG to execute on the edge device. While the MILP is solved on commodity hardware, the generated schedule shipped to the edge device is only a few hundred bytes, making it highly memory efficient.

Rematerialization is most efficient for operations that are cheap-to-compute yet memory-intensive. These operations can be recalculated with low energy overhead. Paging, however, is best suited to compute-intensive operations where rematerialization would otherwise incur significant energy overhead. POET jointly considers both rematerialization and paging in an integrated search space.

Without a minimum training throughput limit, it is possible that the energy optimal strategy is also far too slow to train in practical applications. In reality, training needs to run while the device is idle where spare compute cycles are available. For example, Google Android schedules ML model updates when the phone is charging. To maintain high training

throughputs, the POET optimizer can maintain a minimum training throughput to ensure that training completes during downtime.

Given a memory budget $\mu_{RAM}$ and a training time budget $\mu_{deadline}$, POET finds an energy optimal schedule by choosing to either a) rematerialize or b) page the tensors to/from secondary storage such as an SD card. Our method scales to complex, realistic architectures and is *hardware-aware* through the use of microcontroller-specific, profile-based cost models. We build upon the formulation proposed by Checkmate [23] and adapt it to jointly consider integrated rematerialization and paging, to optimize for an energy objective rather than the runtime, and to implement a minimum throughput constraint.

[t!]

$$
\begin{aligned}
\text{arg min} \quad & \sum_{T} [R\Phi_{compute} + M_{in}\Phi_{pagein} + M_{out}\Phi_{pageout}]_T \\
\text{subject to} \quad & R_{t,i} + S^{RAM}_{t,i} \geq R_{t,j} && \forall t \in V \;\; \forall (v_i, v_j) \in E \\
& R_{t-1,i} + S^{RAM}_{t-1,i} + M^{in}_{t-1,i} \geq S^{RAM}_{t,i} && \forall k \in K \;\; \forall t \geq 2 \;\; \forall i \\
& S^{AUX}_{t-1,i} + M^{out}_{t-1,i} \geq S^{AUX}_{t,i} && \forall t \geq 2 \;\; \forall i \\
& S^{AUX}_{t,i} \geq M^{in}_{t,i} && \forall k \in K \;\; \forall t \geq 2 \;\; \forall i \\
& S^{RAM}_{t,i} \geq M^{out}_{t,i} && \forall k \in K \;\; \forall t \geq 2 \;\; \forall i \\
& U^{RAM}_{t,i} \leq \mu_{RAM} && \forall t \in V \;\; \forall i \in V \\
& \sum_{T} [R\Psi_{compute}]_T \leq \mu_{deadline} \\
& S_{1,i} = 0 && \forall i \in V \\
& R_{v,v} = 1 && \forall v \in V \\
& R, S_{SD}, S_{RAM}, M_{in}, M_{out} \in \{0,1\}^{T \times T}
\end{aligned}
\tag{2.1}
$$

**Assumptions:** We assume operations execute sequentially on edge devices without inter-operator parallelism. Moreover, we assume parameters and gradients are stored in a contiguous memory region without paging. Unlike prior work in rematerialization [8, 26], we do not limit rematerialization to occur once. We assume auxiliary storage (e.g., flash/ SD card) is available. However, if auxiliary storage is not available, the POET optimizer will fall back to only performing rematerialization.

## Optimal Rematerialization

Following the design of Checkmate [23], we introduce the formulation of the rematerialization problem. Given a directed acyclic dataflow graph $G = (V, E)$ with $n$ nodes, a topological ordering $\{v_1, \ldots, v_n\}$ is computed which constrains execution to that order of instructions. Two key decision variables are introduced: (1) $R \in \{0,1\}^{n \times n}$ where $r_{t,i}$ represents the decision to (re)materialize an operation $v_i$ at timestep $t$ and (2) $S \in \{0,1\}^{n \times n}$ where $s_{t,i}$ represents whether the result of an operation $v_i$ is resident in memory at timestep $t$.

From the rematerialzation matrix $R$, and the storage matrix $S$, we define a series of constraints to maintain graph dependencies. All arguments for an operation $j$ must be resident in memory prior to running that operation, yielding constraint $R_{t,i} + S_{t,i} \geq R_{t,j} \quad \forall (i,j) \in E \quad \forall t \in \{1, \ldots, n\}$. Similarly, the result of an operation is only resident in memory in one of the two cases: a) if it was already resident in memory before, or b) if it was (re)materialized ($S_{t,i} \leq S_{t-1,i} + R_{t-1,i} \quad \forall i \in V \quad \forall t \in \{1, \ldots, n\}$).

To adhere to the strict constraints on the peak memory used during training, an intermediate variable $U \in \mathbb{R}^{n \times n}$ is defined. $U_{t,i}$ is the total memory used by the system during training at timestep $t$ when evaluating operation $i$. By bounding the maximum value of $U_{t,i} \quad \forall i \in V \quad \forall t \in \{1, \ldots, n\}$ to the user-specified memory limit $\mu_{RAM}$, we limit the total memory consumption during training.

## Optimal integrated paging and rematerialization

While rematerialization can provide significant memory savings, it introduces significant energy consumption overheads from duplicate recomputations. Similarly, paging if done wrong will result in a wasteful shuffling of data between memories. Here, we formalize a joint search space for rematerialization and paging to enable the discovery of the energy-optimal hybrid schedule.

Like rematerialization, the discovery of the optimal paging schedule is a challenging combinatorial search problem. However, we find that independently solving for paging first, and then solving for rematerialization will not produce globally optimal solutions. As an example, consider a graph where the output depends on the result of two operations $v_1$ and $v_2$ where both nodes have equivalent memory costs but $v_2$ is cheaper to evaluate. A paging strategy may evict $v2$ which would force rematerialization to recompute the more expensive $v_1$ rather than $v_2$.

We represent a schedule as a series of nodes that are either being saved $S^{RAM}$, (re)computed $R$ or paged from secondary storage $S^{AUX}$. To model when a node is copied from secondary storage to RAM, we introduce a variable $M^{in} \in \{0,1\}^{n \times n}$ where $M^{in}_{t,i}$ represents paging a tensor from secondary storage to RAM between timesteps $t-1$ and $t$. Similarly, we model page-out with $M^{out}$.

We now present the intuition behind adding the following constraints to the optimization problem in order to search over optimal schedules for paging and rematerialization:

1c For $S^{RAM}_{t,i}$ to be in memory at time-step $t$, either compute $R_{t,i}$ at timestep $t$, or retain $S^{RAM}_{t,i}$ in memory from the previous timestep $t-1$, or page-in if $S^{RAM}_{t-1,i}$ is resident on flash (at $t-1$).

1d Each node $i$ can reside on flash $S^{AUX}_{t,i}$, either if it resided on flash at timestep $t-1$ ($S^{AUX}_{t-1,i}$), or it was paged out at time-step $t-1$ ($M^{out}_{t-1,i}$).

1e To page-in $M^{in}_{t,i}$ at time-step $t$, it has to be resident on flash $S^{AUX}_{t,i}$ at timestep $t$.

1f Each node $i$ can reside in memory $S_{t,i}^{RAM}$ at timestep $t$, only if it was paged out of flash ($M_{t,i}^{out}$).

Algorithm 2.5 defines the complete optimization problem.

## Expressing an energy consumption objective

If we only consider rematerialization, then minimizing runtime will generally correlate with decreased energy usage. However, this is no longer true when considering paging; paging can be more energy-efficient than rematerializing a compute-intensive operation. To address this, we introduce a new objective function to the optimization problem that minimizes the combined energy consumption due to computation, page-in, and page-out.

When paging occurs on an edge device, the vast majority of energy consumed is due to powering-on the flash/SD block device. As this power is in addition to any power the CPU is consuming, the total power consumption is a linear combination of paging and CPU energy. We precompute each of these values, generally as the integral of the power of active components of the edge device integrated over the runtime of the operation. $\Phi_{compute}$, $\Phi_{pagein}$ and $\Phi_{pageout}$ represent the energy consumed for each node for computing, paging in, and paging out respectively.

Therefore, the new objective function combining paging and rematerialization energy usage is:

$$\sum_{T} [R\Phi_{compute} + M_{in}\Phi_{pagein} + M_{out}\Phi_{pageout}]_T \tag{2.2}$$

## Ensuring minimum training throughput

If we attempt to find the minimum energy schedule subject to only a memory constraint, the POET solver may select solutions with poor end-to-end training throughput. Ideally, training should occur in the downtime between interactive workloads on an edge device. To ensure this, we introduce a new constraint to the optimization problem that ensures schedules meet a minimum training throughput threshold. This constraint effectively trades off between energy consumption and training throughput.

To enforce a particular throughput, we compute a latency target. Via profiling, we capture $\Psi_{compute}$ denoting the runtime of each operation. We then constrain total runtime with the constraint:

$$\sum_{T} [R\Psi_{compute}]_T \leq \mu_{deadline} \tag{2.3}$$

## Paging latency hiding via transfer planner

POET outputs the DAG schedule in terms of which nodes of the graph ($k$) to rematerialize, and which to page-in ($M_{t,k}^{in}$) or page-out ($M_{t,k}^{out}$) at each time-step ($t$). Our Algorithm 2.5

| Device | Clock | RAM | FPU? |
|---|---|---|---|
| M0 (MKR1000) | 48 MHz | 32 KB | × |
| M4 (nrf52840) | 64 MHz | 256 KB | √ |
| A72 (RPi-4B+). | 1.5 GHz | 2 GB | √ |
| A57 (Jetson TX2) | 2 GHz | 8 GB | √ |

Table 2.2: We evaluate a wide variety of battery-powered edge devices. All devices have at least 32GB of flash memory via an SD card or flash to enable paging of activations or tensors. FPU is floating-point unit.

takes the ILP solves to generate and dictate the strategy that determines which tensors are resident-in-memory ($S^{aux}_{t,k}$) at a fine-grained (operator) level.

We factor in the latency introduced by paging. As described in Section 2.6, POET is hardware-aware by profiling the latency per platform for paging activations to secondary storage. Fine-grained profiling helps in fine-tuning when to start paging, such that the activation tensors arrive just-in-time. We then modify the page-in ($M^{in}_{t,k}$) and the page-out ($M^{out}_{t,k}$) schedule to ensure there is no contention for the memory bus as the tensors are paged-in just-in-time. For example, if ($M^{in}_{t,i}$) can contend with ($M^{in}_{t,j}$), then we schedule one of them to page-in at an earlier time ($M^{in}_{t-1,i}$) and update the in-memory schedule ($S^{aux}_{t-1,i}$) to account for the earlier paging-in. While this ensures the activations are paged in just-in-time, in parallel, ($R_{t',j}$) informs the PyTorch DAG scheduler to deallocate the tensors that we have chosen to rematerialize ($S^{aux}_{t,k}$) at a future timestep ($t'$).

[t] **Input:** Graph $G = (V, E)$, schedule $R, M_{in}, M_{out}$
**for** *t=1,..,|V|* **do**
  **for** *k=1,..,|V|* **do**
    **if** $M^{in}_t$ **then**
      add `%r = pagein` $v_k$ to $P$
    **if** $R_{t,k}$ **then**
      add `%r = compute` $v_k$ to $P$
    **if** $M^{out}_{t,k}$ **then**
      add `%r = pageout` $v_k$ to $P$
    **for** $i \in DEPS[k] \cup \{k\}$ **do**
      **if** $M^{out}_{t,k} \vee FREE_{t,i,k}$ **then**
        add `deallocate %r` to $P$
**Output:** execution plan $P = (v_1, .., v_n)$

## 2.6 Evaluation

In our evaluation of POET we seek to answer three key questions. First, how much energy consumption does POET reduce across different models and platforms? Second, how does POET benefit from the hybrid paging and rematerialization strategy? Lastly, how does POET adapt to different runtime budgets?

### Experimental setup

We evaluate POET on four distinct hardware devices listed in Table 2.2: the ARM Cortex M0 class MKR1000, ARM Cortex M4F class nrf52840, A72 class Raspberry Pi 4B+, and Nvidia Jetson TX2.

POET is fully hardware-aware and relies on fine-grained profiling. For example, on the Jetson-TX2 hardware we profile each operator along with its variations in dimensionality (e.g., `conv2d` with varying kernel-sizes, strides, padding, etc.) These fine-grained time, energy, and memory profiles then inform POET about the exact specifications. These devices test a diverse set of memory, compute, and power configurations. As POET is hardware and energy-aware, it takes device-specific characteristics into account.

We evaluate POET on VGG16 [44] and ResNet-18 [20] trained on the CIFAR-10 dataset as well as BERT [13]. In all of our baselines, we limit all MILP solves to no more than 10 min on commodity CPUs. Our experiments are with a batch-size of 1. We compare POET to work PyTorch's default scheduler, Chen et. al[8], Griewank & Walther[17], DTR [26], and Checkmate [23].

**Hyperparameters:** POET only decides on the optimal scheduling of nodes in the training graph and does *not* change the training routine (learning rate, optimizer, etc.). Hence, our system is robust to hyper-parameters.

Sensitivity to Batch-size: POET is mathematically preserving and can be easily scaled to arbitrary batch sizes without loss of generality. Of course, this is conditioned on the underlying device's memory capacity. It is possible that as batch size varies, the underlying operator implementation might change. POET, with its fine-grained profiling is robust to these changes and transparently adapts to artifacts.

### How much energy consumption does POET reduce across models and platforms?

Figure 2.3 shows the energy consumed for a single epoch of training. Each column represents a unique hardware platform as defined in Table 2. We notice that across all platforms, POET generates the most energy-optimal (Y-axis) schedule all the while reducing the peak memory consumed (X-axis) and adhering to the timing budget.

For the BERT model on the Cortex M4 and the TX2 platform, we noticed an interesting behavior: our ILP solves time-out. This is because we limit all solves to no more than 10 min. With a longer ILP solve budget (<30 min), POET can predictably find more optimal

| | ResNet-18 Training | |
|---|---|---|
| | POET | POFO (Beaumont et al. 2021) |
| Memory | 285,873 kB | 311,808 kB |
| Runtime | 82.36 ms | 94.79 ms |

Table 2.3: POET's MILP formulation lowers peak memory consumption by 8.3% and improves throughput by 13% compared to POFO [2] on Nvidia's Jetson TX2 edge device

solutions. Further, notice that a) POET has an additional timing budget which none of the other baselines do, and b) all of our baselines are already mature. Checkmate [23] is provably optimal for rematerialization, while DTR [26] closely approximated Checkmate. Furthermore, POET tried to solve a much "harder" problem as its search space with rematerialization and paging together is larger.

## How does POET benefit from integrated rematerialization and paging?

We compare our joint optimal paging and rematerialization schedule with Capuchin which optimizes each with a heuristic. Capuchin will effectively page until no longer feasible and only then will it begin to rematerialize. Instead, POET begins rematerializing cheap operations like ReLU much earlier which yields considerable energy savings (up to 141% lower overhead).

In Figure 2.5, we benchmark POET and Capuchin when training ResNet-18 on the A72. As the RAM budget decreases (to the right), Capuchin consumes 73% to 141% more energy than a baseline with full memory. In comparison, POET incurs less than a 1% energy overhead. This trend holds for all architectures and platforms we tested.

In Table 2.3 we benchmark POET and POFO when training ResNet-18 on Nvidia's Jetson TX2. We find that POET finds an integrated rematerialization and paging schedule that lowers *peak memory consumption by 8.3%* and *improves throughput by 13%*. This showcases the benefit of POET's Mixed-integer linear programming (MILP) solver, which is able to optimize over a much larger search-space. While POFO only supports linear models, POET generalizes to non-linear models as demonstrated in Fig 2.3.

## How does POET adapt to varying runtimes?

Figure 2.4 highlights the benefit of the integrated strategies that POET adopts across different timing constraints. The run-time budget refers to the total time available for one epoch of training naïvely (without paging or rematerialization). For each of the runtimes, we plot the total energy consumed if we were to restrict to either of a) paging or b) rematerialization only, and the c) integrated solution.

We find that rematerialization is energy-optimal compared to paging at higher (looser) timing budgets. This is reflected in the POET (paging+remat) green curve, closely tracking the POET (remat only) yellow curve at runtime budgets of 0.6 - 0.9 ms. However, at lower runtime budget (0.5 ms), paging is preferable as rematerialization strategies become infeasible. This is because, rematerialization is a compute intensive serial operation, however, our Algorithm 2.5 benefits from the ability to hide paging latencies by pipelining (see Section 2.6) to realize the tighter deadline bounds. POET's optimal, integrated solution consumes up to 40% lower energy compared to paging or rematerialization only solutions.

## 2.7 Conclusion

Enabling large models to be trained on edge devices is important due to privacy constraints as well as offline operation. Edge devices deployed in the real-world are powered by tiny microcontrollers that are low-powered, and have limited memory (e.g. 32 KB). The low-power and limited memory, coupled with tight timing constraints imposed by real-time systems makes training on the edge challenging.

Our novel mixed-integer linear programming based Power Optimal Edge Training (POET) algorithm enables training on tiny chips with memory as low as 32 KB. Given a memory budget and a timing constraint, POET finds the most energy optimal schedule to train the model by choosing to either rematerialize or page the tensors to secondary storage.

Across a diverse set of models and devices, we discover low-power training schedules at less memory than baselines. POET enables new applications for privacy-preserving personalization of large models like BERT on tiny devices at the edge for the first time. Future directions include integrating activation compression as well as expanding POET's search space to paging parameters.

## Acknowledgement

We thank Prateek Jain, Charles Packer, Daniel Rothchild, Alex Smola, Pete Warden, and the anonymous reviewers whose insightful comments, and feedback helped improve the paper. This research is supported by a NSF CISE Expeditions Award CCF-1730628, and gifts from Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Scotiabank, and VMware. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

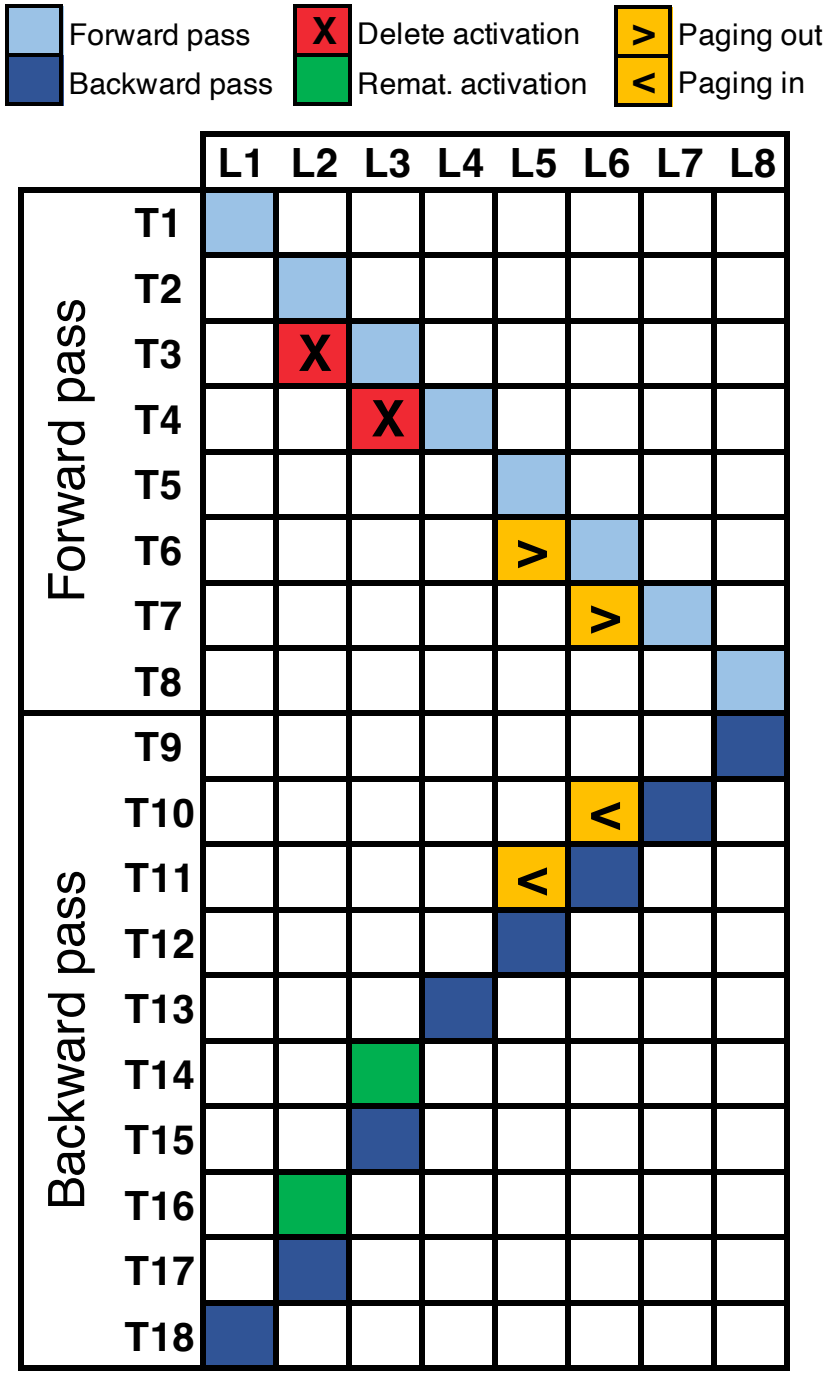


Figure 2.2: Rematerialization and paging are complementary. This plot visualizes the execution schedule for an eight layer neural network. We represent logical timesteps in increasing order on the y-axis while different layers are represented by the x-axis. Layers 2 and 3 are cheap-to-compute operators and therefore can be rematerialized at low cost. However, layers 5 and 6 are compute intensive so it is more energy-efficient to page them to secondary flash storage.

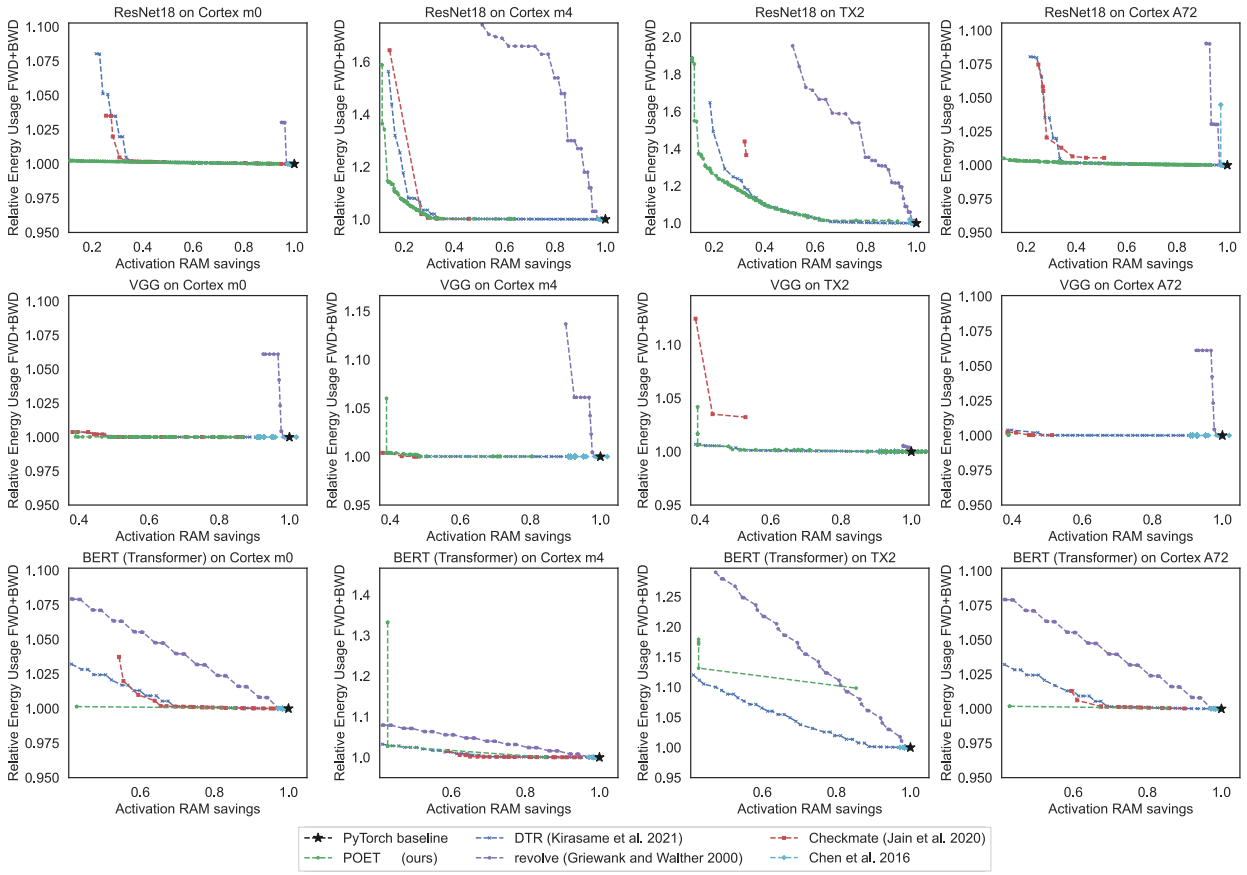


Figure 2.3: **POET consumes less energy across diverse models and devices**: We profile the energy usage of each method relative to a full-memory configuration as the device's available memory capacity shrinks. For ResNet-18 (top row), VGG (middle row) and BERT (bottom), POET outperforms competitive methods in most configurations. When training ResNet-18 on the TX2, POET consumes up to 35% less energy than DTR while discovering solutions at tighter memory budgets.

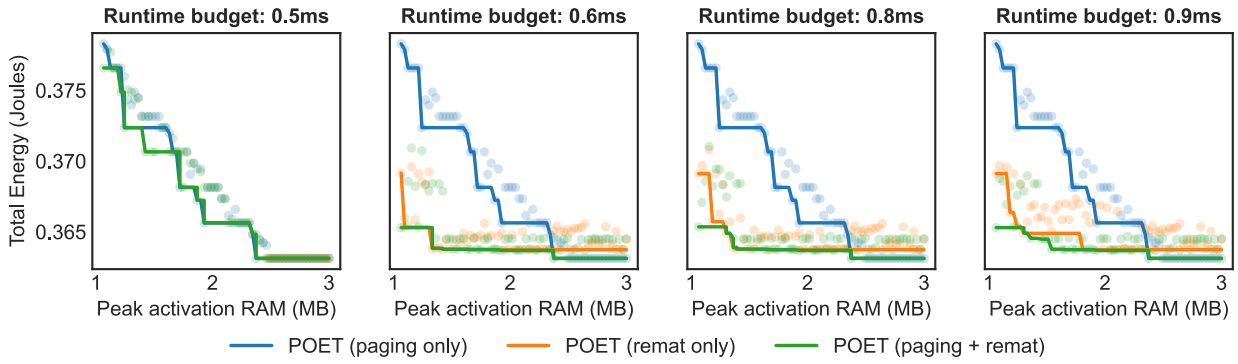


Figure 2.4: **Both rematerialization and paging are necessary for low-energy schedules with limited memory**: We compare ablations of POET on VGG for CIFAR-10 and find that both rematerialization and paging are required to achieve low-energy solutions at limited memory budgets across all runtime constraint values.

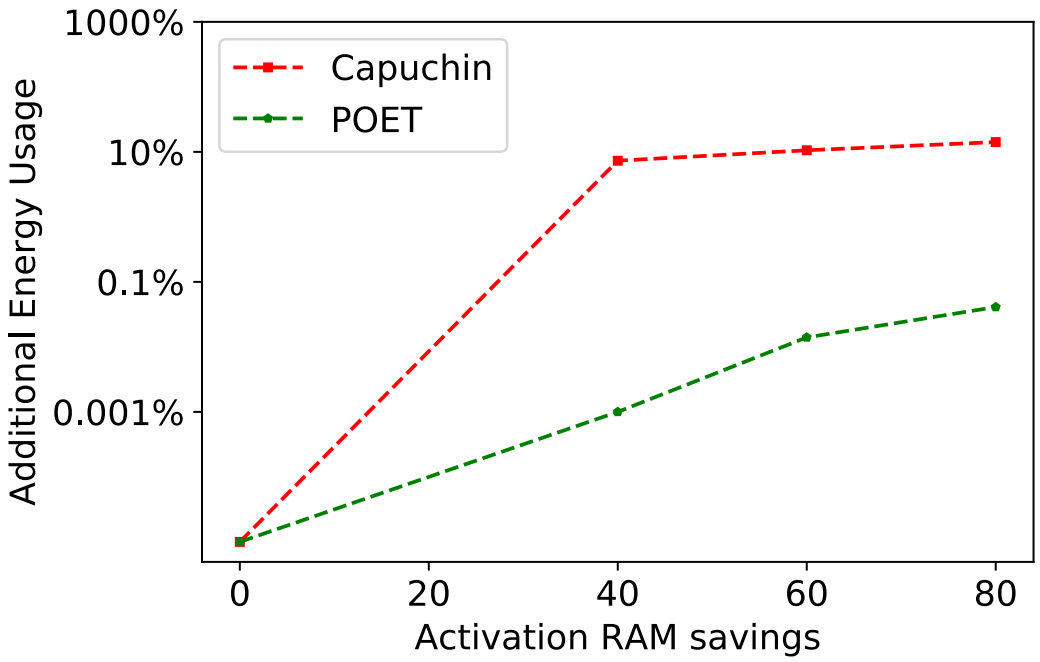


Figure 2.5: **Optimal integrated rematerialization and paging outperforms Capuchin (log scale)**: POET incurs 73% to 140% less energy overhead relative to a full-memory baseline by rematerializing earlier alongside paging. Capuchin strongly prefers paging before falling-back on rematerializing activations which makes it sub-optimal.

# Chapter 3

# Minerva: Efficient ML Model Updates for Deeply Embedded Microcontrollers

## 3.1 Introduction

Frequent updates of machine learning (ML) models are necessary to respond to changes in the environment and data, the arrival of new datasets, user feedback, and rapid innovation in model design [16, 47]. ML model updates include not only data (e.g., the weights of a neural network) but also model architecture and the associated code of its operators, which may change in an update. This cycle of feedback, re-training, and re-deployment continues for the lifetime of a deployment.

At the same time, ML applications that have historically resided in the cloud are moving closer to the edge. In this paper, we focus on the extreme edge, consisting of embedded microcontrollers (MCUs) built around Cortex-M family of ARM CPUs. For example, Farmbeats [49] has used microcontrollers in sensor boxes for precision agriculture. Recent efforts have looked at designing models for the constraints of the edge [49, 37, 50]. However, updated models that are trained centrally, e.g. in the cloud, must then be deployed on edge devices to provide better predictions for the embedded applications they support. In this paper, we focus on this critical *deployment* stage of the ML lifecycle for edge devices.

On server-class systems, the model is served from a separate process or microservice using Remote Procedure Calls (RPCs) [**meta˙microservices**, **deathstarbench**]. This allows the model executable to be updated independently of the rest of the application. Such decoupling of the application from the model also provides *flexibility*—it is easy to test the new model on samples of traffic and roll back changes to earlier models if necessary, permitting *AB testing* and *acceptance testing*.

Unfortunately, this multi-process approach does not work on MCU-based edge devices. The reason is that MCUs lack MMUs and have only *kilobytes* of RAM, making them ill-suited to running full-fledged operating systems like Linux [**telos**, **hamilton**, **at86rf233**, **nrf52840**]. On MCUs, the application and system share an address space and are typically linked into a

single logical program called the *image*. While possible, it is difficult to update only part of the image, because the layout of code and data (e.g., the lengths of functions, etc.) may change with each update. This would break any jumps, branches, and function calls into the updated part of the image. And, unless the device is rebooted, any data structures and pointers in RAM must be transformed for use with the updated code and data.

Instead, ML model updates on MCU-based devices are typically carried out through *device firmware updates (DFUs)*, in which an entire new image is installed on the device. DFUs are a natural solution for classical use cases, such as infrequent reprogramming of a sensor network to a new application or patching a vulnerability [**dfu-vulnerability**]. But DFUs have several downsides, exacerbated in the context of model updates. First, although only the model is updated, DFUs download a full program image, which is bandwidth-intensive, especially over low-rate links like NB-IoT, LoRA, or even satellites. Second, each DFU requires a reboot, losing application and system state, which is costly because model updates may be frequent and it may take time to recreate precious state. Third, DFUs are less flexible than the process-based approach, increasing friction and overhead for local model testing and rollback. Fourth, it restricts software to only *half* of the available memory, to have space to download an update before switching to it. In practice, edge devices update models less frequently or not at all due to these downsides [49, 6].

In this context, we study how to enable efficient model updates for edge devices. We propose a system, Minerva, that does so with minimal application changes, with no added overhead in obtaining a model prediction, and while retaining much of the flexibility of the multi-process approach.

A seemingly natural starting point for Minerva is to make DFUs more efficient. For example, one can reduce the bandwidth for DFUs using delta updates [**ota˙compression**, **mbedos˙delta**, **freertos˙delta**], which are represented as a *diff* from the previous image instead of as a fresh new image. Unfortunately, this still requires device reboots and lacks the flexibility of the multi-process model.

Instead, our approach is to bring a service decomposition similar to the multi-process model to MCUs, within a single address space. The key observation is that models are *pure functions*. In particular, an model can be naturally abstracted as a single `predict` function that (1) returns its output by value, and (2) maintains no state across invocations. Our key insight is that this property allows us to build a carefully-structured, updatable memory segment that we call a *capsule*, which contains the ML model.

To understand how capsules work, consider the challenges we described earlier in updating only part of the image. Because ML models are pure functions, they *never* expose pointers to their internal code or data or maintain state across invocations. Thus, there is no need to update data structures and pointers throughout RAM when updating the capsule. By leveraging this property of ML models, we sidestep a significant source of complexity in performing live updates [**mrpc**, **snap**].

Another challenge is that updates may break branches, jumps, and function calls into the capsule. To address this, we structure the capsule such that the `predict` function is always at the same memory address after each update. Because the application and model only

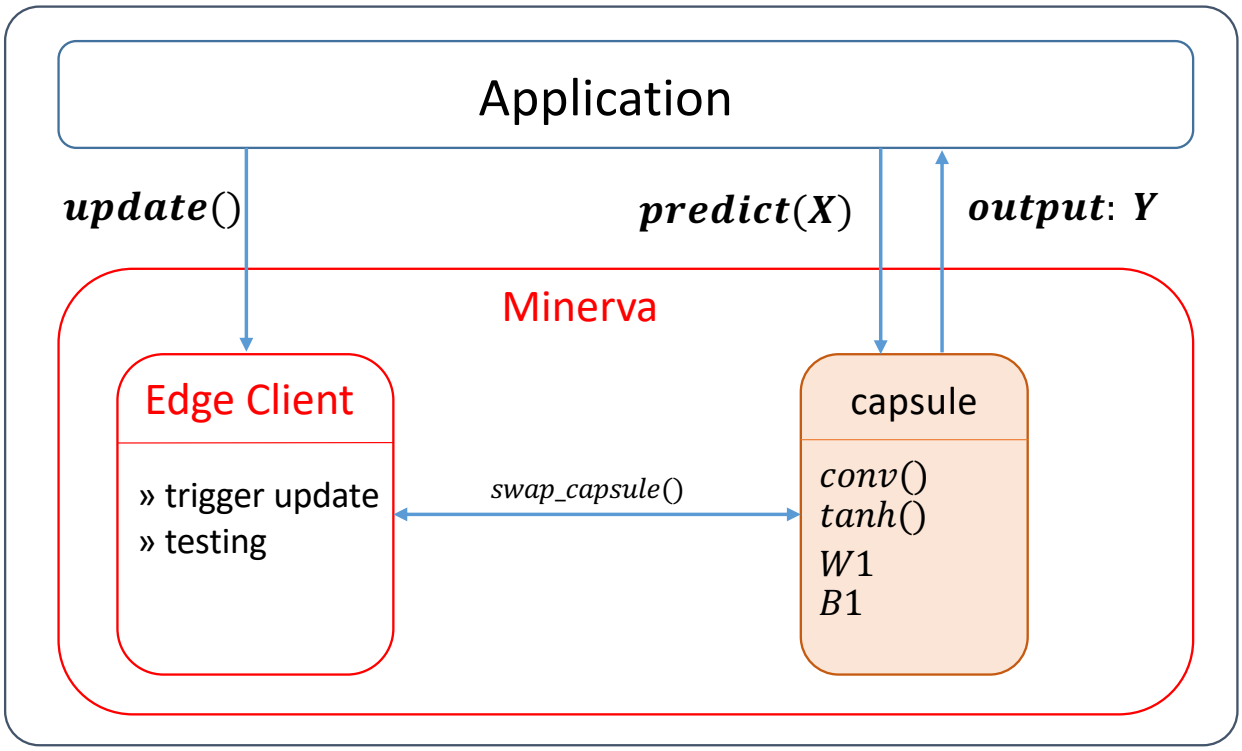


Figure 3.1: Minerva explores an user-space library design that interposes between ML models and the rest of the application. The operators (code) and weights are stored in self-contained and isolated capsules enabling rapid update of ML models transparent to the application logic.

interact via the `predict` function, the application can use the updated model without having to patch any code outside of the capsule. As `predict` is called just like a normal function, capsules add no overhead in the critical path of ML inference. Importantly, capsules are simple to integrate into an application. The programmer merely *annotates* functions and data belonging to the capsule and uses a special linker script that structures the capsule correctly at build time. To update the capsule, the device first downloads the update, and then invokes a lightweight library to apply it to the capsule.

Certain systems for MCUs, like Maté [**mate**] and Tock [33], provide general dynamic loading facilities that we could use instead of capsules. However, they are far more intrusive than capsules, requiring applications to run in a software interpreter or be ported to a particular runtime environment. Unlike these approaches, capsules do not require tight integration with the underlying system and are largely platform-agnostic. This is critical for wide applicability because MCU-based edge devices exhibit heterogeneity in hardware and software.

To demonstrate the utility of capsules, we also design Minerva, an end-to-end capsule-

based ML model update system. Minerva coordinates updates using a pull-based model and uses a simplified form of delta updates to efficiently transmit updated capsules to edge devices. Additionally, Minerva leverages the flexibility afforded by capsules to perform local acceptance testing for model updates. Although devices may lack access to ground-truth labels, we show that we can locally test the utility of model updates using an established statistical technique called discounted cumulative gain (DCG).

We have implemented our system on four real-world applications and found that Minerva enables model updates up to 89× faster than a standard DFU and up to 73× faster than an application update in Tock (a widely-used embedded OS). Further, eliminating the need to re-flash non-ML code or perform a reboot enables applications to maintain state without checkpoints and restores. We designed Minerva to be easy to deploy and to integrate well with existing ML platforms. Concretely, it requires four additional lines of code to interface with TensorFlow/PyTorch, and two additional lines of code in the edge application—one to `include` the library, and the other to apply a capsule update (`swap_capsule()`).

With Minerva, (1) We introduce capsules, which enable state-preserving low-bandwidth transparent updates for ML models. (2) We introduce a mathematical formulation for locally testing the utility of the received model using discounted cumulative gain (DCG) and the ability to perform acceptance testing for model updates. (3) We implement Minerva on MCUs and evaluate it on four real-world embedded machine learning applications. We demonstrate up to 89× reduction in DFU time.

## 3.2 Background

Although the term *edge* can refer to a variety of devices and services, including mobile phones, routers, gateways, and CDNs, this paper focuses on the ***extreme edge* consisting of deeply embedded devices**. These include sensors for environmental monitoring [**habitat˙monitoring**, 49], structural health monitoring [**structural˙health˙monitoring**, **golden˙gate˙bridge˙monitoring**], and IoT [37].

It is increasingly common to deploy ML models on such devices. For example, **Farmbeats** [49] is an end-to-end IoT platform for precision agriculture that processes data collected from various sensors deployed in farms. To identify faulty sensors, Farmbeats recently incorporated Fall-curves [6] to detect sensor faults using an ML-based predictor [18]. As the underlying soil condition changes, weather patterns change, and newer data is ingested, Farmbeats benefits from frequent model updates. Other examples of MCU-based edge ML devices are GesturePod [37], Picovoice [41], Powerblade [11], Zanzibar [50] and Permamote [**permamote**].

A model is composed of *operators* and *weights*. Operators are the building blocks of an ML architecture, such as `tanh`, ReLU, convolutions, and max-pooling. Weights are the parameters associated with these operators, such as convolution filter weights and biases. Model updates could include updates to both the code and data associated with an model—that is, to both

the operators and weights. Additionally, as we shall explain in §3.2, models are often only a small fraction of the overall image size.

## MCU-Based Device Hardware

Deeply embedded devices must be cheap to produce, easy to embed in the physical world, and operate for long periods of time (e.g., for years on a small, cheap, $\leq 100$ mAh battery). As such, they are typically built around microcontrollers (MCUs) that are cheap, small, and energy-efficient, such as the Nordic nRF52 series [**nrf52**] or Atmel SAM R21 series [**samr21**].

Such MCUs typically use ARM Cortex-M CPUs that lack MMUs and have only *kilobytes* of RAM. Importantly, they have less RAM per unit of compute power than conventional systems. The reason is that MCUs use only SRAM (instead of DRAM) to minimize energy consumption, and even so, are limited in RAM size by SRAM leakage current. Whereas non-MCU systems have $\approx 1$ MiB of RAM per MIPS of CPU (3M rule), the nRF52832 MCU, used by Farmbeats, has $\approx 100$ DMIPS of CPU (ARM Cortex-M4F @ 64 MHz) but only 64 KiB of RAM. Thus, while embedded CPUs have grown more powerful over the years, making algorithms like ML attractive, embedded RAM has not grown commensurately.

To deal with limited RAM size, these MCUs also include nonvolatile, flash-backed "read-only memory," or ROM. On these devices, ROM is more plentiful than RAM; the nRF52832, for example, has 64 KiB of RAM but 512 KiB of ROM. As a result, RAM is only used for data that may change at runtime, like stack, heap, and mutable globals. Any data that will not change at runtime, like program text (code) and static data, are stored in ROM. This is possible because ROM and RAM are in a unified physical address space, enabling the program to execute code and access data directly out of ROM without first loading that data into RAM.

Network bandwidth is also a constrained resource for devices at the extreme edge. Low-power wireless network technologies, like LoRa and IEEE 802.15.4, provide only *kilobits* per second of bandwidth. Even if network bandwidth is available, battery-powered devices must use it sparingly, as network usage can dominate energy consumption [**ipisdead**, **hamilton**, 32].

## MCU-Based Device Software

MCU-based devices do not run fully-fledged operating systems like Linux or Windows. Instead, they typically run specialized *embedded OSes*, like TinyOS [31], Contiki OS [**contiki**], or RIOT OS [**hamilton**], that are carefully designed to be lightweight. Embedded OSes generally provide a scheduler, timers, device drivers, and an IP-based network stack. However, as MCUs lack hardware support for address translation (i.e., MMUs), embedded OSes generally lack memory virtualization, protection/isolation, and dynamic linking/loading. The application and system are linked into a monolithic program, called an *image*, and execute together in the same address space. Applications request system services via direct function calls and can freely obtain and follow pointers to OS data structures. This design stems from the

classical view that, for maximum efficiency, the system and application should be tightly integrated and co-specialized [**tinyos˙asplos**].

A notable exception is Tock OS [33]. Tock uses hardware support for memory protection in modern MCUs to provide multiprogramming with isolation. Still, it *does not support virtual memory or address translation*, requiring apps to be compiled with position-independent code (PIC) to support dynamic loading. Furthermore, Tock's dynamic loading only applies to data in *RAM*, not in *ROM*—all loadable apps must be part of the device's ROM when the device is initially programmed, and new apps cannot be downloaded over the network at runtime. We use Tock as a starting point in designing capsules and compare Minvera's performance to Tock's.

## DFUs for MCU-Based Devices

Currently, the most accessible way to update an ML model on an MCU-based device is to perform a *device firmware update (DFU)*. In a DFU, an entirely new image, containing not only the model but also the application and system, is built and deployed to a device. In this section, we describe how DFUs work and explain why they are inefficient for model updates.

**Step 1: Obtaining the New Image.** Embedded OS solutions often rely on a physical connection (e.g., J-Tag) or close proximity (e.g., Bluetooth) to install or update an application. Unfortunately, it is difficult to physically access edge devices deployed in the field, particularly for devices in remote locations like farmlands [49] or the deep sea [24]. As a result, it is desirable to use *over-the-air (OTA) updates*, in which updates are transferred over the network. Herein lies the first challenge of DFUs—while edge devices are constrained in network bandwidth, OTA DFUs are bandwidth-intensive. As an extreme example, Farmbeats uses LoRa with a bit rate of only $\approx 100$ *bits* per second, making it slow to download a $\approx 100$ KiB DFU. DFUs are particularly inefficient in the context of model updates because, as shown in Figure 3.2, the model is only a small fraction of the update size. Increase font size in figure? Chart is not red/green colorblind friendly. Also notice the green bars highlighting the "bloatware" introduced by embedded OSes. Even a compiled binary containing only ML code and no other application logic is still considerably larger than the standalone size of the ML model.

**Step 2: Reflashing ROM with the New Image.** The new image must be stored in ROM. The process of updating data in ROM is called *reflashing*. ROM, as its name ("read-only memory") suggests, is optimized for reading; writing to ROM is far slower and more energy-intensive than writing to RAM. As a result, reflashing ROM with the new update is a significant source of overhead in the update process, as we will see in §3.5. The large size of DFUs exacerbates this overhead.

**Step 3: Verifying the New Image.** Reliability for DFUs is accomplished via a two-bank memory model. ROM is partitioned into two separate regions called *banks*, one of which stores the active image. In a DFU, the new image is downloaded into the other bank and checked for integrity. If the integrity check passes, then either the new image is copied into the active bank, or the bank containing the new image is marked as active. If the update

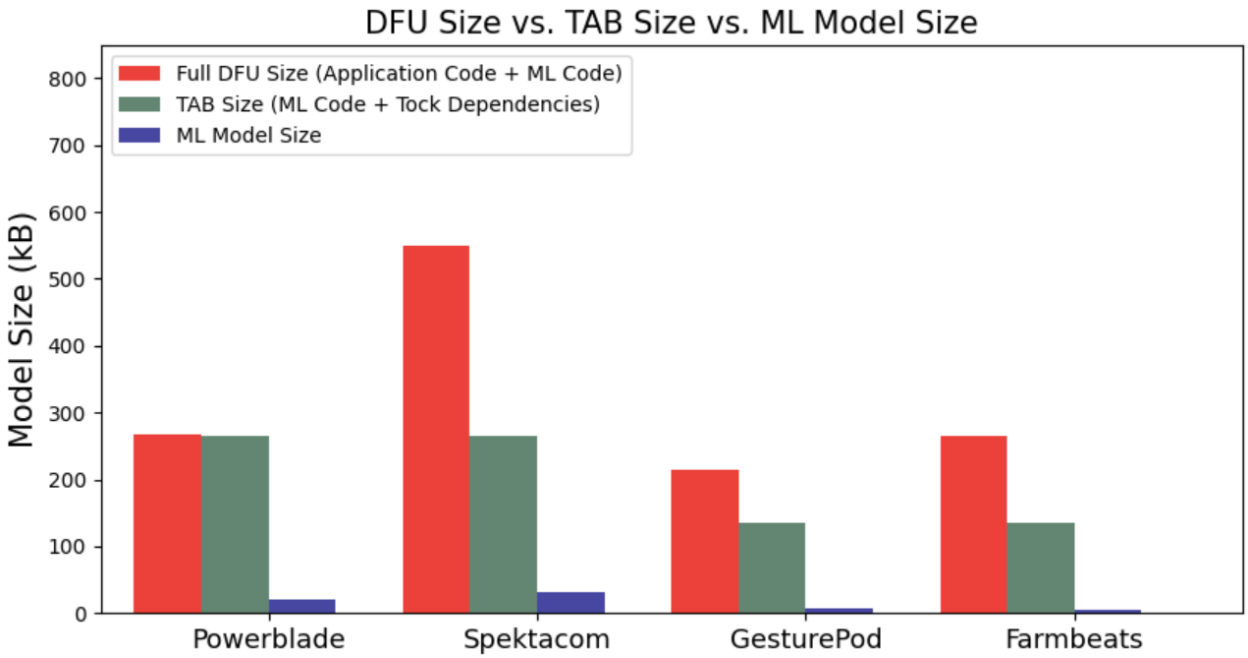


Figure 3.2: Comparison of machine learning model size to full DFU size and TAB size (Tock Application Binary, a compiled application to be loaded onto a board with Tock OS) for several edge applications. ML model size (blue) includes ML operators and weights. TAB Size includes the ML model size and Tock support modules. The full DFU size (red) includes ML model size along with application logic, and networking modules. **Unlike** the cloud setting, machine learning model sizes are only a fraction of the total edge application size.

fails, then the device can still boot the old image. While the two-bank model helps ensure reliability, it effectively halves the available ROM space.

**Step 4: Booting the New Image.** Every DFU requires a reboot, which causes the application to lose the contents of RAM (application state). While this can be addressed by checkpointing state in persistent memory and restoring it after the reboot, this is challenging because state is often spread throughout the application (e.g., networking state, application logic, etc.). Additionally, rebooting a device involves re-initializing and synchronizing sensors, serial communication modules, and other peripherals, which takes time.

Rebooting also reduces flexibility. Concretely, ML deployments on server-class devices benefit from acceptance testing. Acceptance testing is important for edge deployments because not all edge devices observe the same data distribution. In the precision agriculture example, a sensor in a farm in Alaska observes a different data distribution than a sensor in a farm in Texas. Hence, even though the new model is better than the old one in most cases (*in expectation*), it may not be universally better across all nodes of a deployed fleet. One way to solve this is to build profiles of each deployed device and test the model in the cloud. However, this is not always feasible as bandwidth constraints prevent streaming data

to the cloud. This calls for local acceptance testing. Having to reboot to switch models adds friction to this process.

## 3.3 Capsules

In this section, we describe *capsules*, which allow ML model updates to be efficiently and flexibly deployed on MCU-based devices. We focus here only on the mechanisms of capsules—how capsules are structured, how to generate them, and how to apply updates on a device. Other details of an end-to-end model update system, like how to trigger updates and how to transfer them over the air, are described in §3.4.

### Motivation and Opportunities

As described in §3.2, DFUs are unnecessarily large in the context of model updates—they contain not only the updated model, but also the entire application and system. A seemingly natural approach for efficient model updates is to only include the updated model in DFUs. An existing approach in this direction is *delta updates*, in which the DFU is represented as a *diff* from the previous image. Delta updates have been studied in academia [**ota˙compression**] and are adopted in certain embedded OSes [**mbedos˙delta**, **freertos˙delta**]. While delta updates can significantly reduce the network bandwidth required to obtain the image, the other issues with DFUs, such as the two-bank model, the need to reboot, and inflexibility, remain.

Importantly, delta updates do not fully reduce reflash time. As a thought experiment, consider an update that increases the length of a function, causing all code located after it in the image to be shifted to a greater address. Delta updates can express this change efficiently when it is transferred over the network, but all code after the updated function must still be shifted in ROM, which is expensive (§3.2). Our conclusion from this thought experiment is that an efficient model update solution must *isolate model code and data in ROM* so that they can be updated independently of the rest of the device code and data. This is exactly what capsules aim to do.

### High-Level Capsule Design

We partition ROM to separate the ML model—both its operators and its weights—from the rest of the application. This is achieved by building the image with a special linker script that creates a fixed-size memory section in ROM large enough to store the ML model. This memory section is called a *capsule*. Because the ML model code and data are in a fixed-size section of ROM, any layout changes in a model update are limited to that section and do not affect the ROM layout of the rest of the image. Thus, capsules can be updated independently from the rest of the image.

Say something here about the analogy to services?

While conceptually simple, this approach is challenging for general updates. When a capsule is updated, any pointers to the capsule's code and data—including branches, jumps, and function calls—may be broken and must be patched. As a concrete example, consider placing the system's IP network stack in a capsule and updating it. Any downcalls from the application to the network API, and any upcalls from the radio driver into the network stack, must be identified in ROM and patched. Avoiding a reboot is particularly challenging because any in-RAM state for the old capsule must be transformed to work with the new one. In a network stack, for example, IP routing state and open TCP connections must work with the new capsule, even if the new capsule uses different data structures. Solutions to this problem are typically complex and intrusive—for example, mRPC [**mrpc**] requires an update to transform the old state appropriately, and Snap [**snap**] requires state to be serialized to a canonical format before an update.

Our key observation is that, for model updates, these problems do not arise. The reason is that ML models are *pure functions*. Specifically, the model exposes only a single point of invocation—a function for ML model inference, which we call `predict`—that (1) returns its output by value, and (2) maintains no state across invocations. This captures a broad range of ML algorithms, from simple signal processing like threshold detection to deep neural networks. Even ML algorithms with an explicit notion of state, like reinforcement learning, are typically implemented as pure functions that explicitly pass state as input arguments and return values.

Because ML models are pure functions that maintain no state across invocations, *they exist entirely within ROM, with absolutely no data stored in RAM between invocations*. This obviates the need to update in-RAM state, sidestepping a core challenge in implementing live updates. The only requirement is that capsule updates are *atomic*—that is, model updates must not happen concurrently with model inference.

We must also address the fact that updates may break pointers into the capsule. Fortunately, the ML model has a narrow interface to the rest of the device—the `predict` function. The application only interacts with the ML model via the `predict` function, and the model never exposes pointers to its internals. Thus, updating the capsule can only break calls to `predict`. To address this, we simply keep `predict` at the same location within the capsule each time it is updated. This ensures that calls to `predict` are not broken upon capsule update.

Capsules naturally fit into the standard pattern of interpreted execution of neural networks adopted by all major deep learning frameworks including TensorFlow Lite Micro [10]. Interpreted execution enables rapid updates of models to a wide range of heterogeneous devices without the need to deploy new operator code. Conversely, improvements in a specific device operator code does not require modification to the model or even application code.

## Capsule Generation

We now discuss how to generate an image containing a properly structured capsule. This is needed both to program a device, and to generate a new capsule for an OTA update.

A capsule is a fixed-size ROM segment at a fixed address in ROM with the following two properties: (1) it contains the code and/or data for an ML model, and (2) it contains an entrypoint function, called `predict` above, at a fixed address. Thus, in order to build an image containing a capsule, the application source code must specify which functions and data belong to the capsule (i.e., are part of an ML model), and which function is the capsule entrypoint (i.e., `predict`).

Our solution assumes that the program is written in C, which is standard for MCU-based devices, and relies on `__attribute__` annotations in `gcc` to specify the above. As a concrete example, capsule functions (e.g., ML operators) are annotated with `.capsule-code` and capsule data (e.g., ML weights) are annotated with `.capsule-data`. The capsule entrypoint (e.g., the `predict` function) is annotated with `.capsule-entry`. These annotations place the corresponding code or data into sections in the compiled object file called `.capsule-code`, `.capsule-data`, and `.capsule-entry`. When generating the image, a special linker script assembles these sections into a capsule.

Importantly, the ML model in the capsule is compiled in relation to the rest of the application source in order to get absolute addressing at link-time correct, as well as preserve any references from within the model code to variables which are in the general application as a whole, rather than just the capsule. To program a device for the first time, the capsule-containing image can be copied into the device's ROM as usual. For an OTA update, the capsule's data is extracted from the compiled image and sent to the device.

How should the linker script structure the capsule? To minimize fragmentation within the capsule, we place the `.capsule-entry` section at the start of the capsule, before the `.capsule-code` or `.capsule-data` section. This not only keeps `predict` at a consistent address across updates, but also keeps the rest of capsule memory remains contiguous, providing maximum flexibility to fit the other capsule code and data within the fixed-size capsule.

## Capsule Update

Once an updated capsule is downloaded to a device and stored in an *update buffer* in RAM, the update must be activated. We cannot simply activate the capsule while it is in RAM, because (1) our system cannot (without modifications) execute code from RAM, and (2) existing calls to `predict` reference the address of the old capsule in ROM. Thus, we must overwrite the old capsule in ROM with the updated data.

However, directly overwriting the old capsule is inflexible; if a problem is detected with the new model, it is impossible to roll back. Instead, we *swap* the contents of capsule memory and the update buffer, allowing the client to swap back to the original capsule if necessary. The swap is implemented with the help of a temporary buffer provided by the caller.

How might the application determine whether to swap back to the old capsule? Capsules are agnostic to the particular techniques that applications use; one possibility is to check if the predictions generated by the new model are adequate. We discuss how the application might determine this in §3.4.

```
// Includes
#include "minerva.h"
...
int main(){
  if (update_condition)
    swap_capsule();
  // Application Logic
  ...
}
```

Figure 3.3: Changes to source-code required on the Edge device. Only two lines are necessary to support - one to `include` the library, and one to trigger the swap.

The update buffer can be viewed as a second bank, similar to the two-bank model for DFUs (§3.2). However, unlike the case of DFUs, the update buffer only needs to be as large as the *capsule*, not as large as the whole image. Additionally, while our implementation stores the update buffer in RAM, it can, in principle, be stored in ROM, in case ROM is plentiful.

On the edge device, the user must make a few changes to the device's application code. As shown in Figure 3.3, the Minerva API exposes a function `swap_capsule` which updates the capsule based on a global pointer to a buffer which contains the updated capsule. The application on the device is responsible for downloading the update and calling `swap_capsule`. What about the case where only a function is updated, not the whole capsule?

Crucially, capsule updates must be atomic—`predict` must not be called concurrently with `swap_capsule`. As synchronization mechanisms depends on the embedded OS scheduler, we leave it to the application to synchronize calls to `predict` and `swap_capsule` to ensure atomicity. On a system with multithreading, the application can use a mutex. On an event-based system, the application may post calls to `predict` and `swap_capsule` to the scheduler's event loop.

## 3.4 Minerva

While capsules provide a mechanism to efficiently and flexibly update ML models, there are several aspects to the end-to-end model update process that they do not handle. Our system Minerva fills in the missing pieces, such as generating the C code for the model, efficiently transferring model updates to devices over the air, and acceptance testing of the new model to decide whether or not to accept the update.

## Minerva Client

The Minerva Client resides on the edge application and communicates with the Minerva Server to update its ML model. An important question that arises is: How is the update triggered? Our answer is to use a *pull model*. In Minerva, the application running on the edge device decides when to update the ML model; to do so, it invokes the Minerva API, which pulls the new model from the server. We do this because the least disruptive time to update the ML model depends on the application, making it natural for the edge node to decide when to trigger the update. For example, devices in Farmbeats [49] rely on solar power and may decide when to request an ML model when energy is plentiful.

### Data Transfer and Integrity

With capsules, the model update can be sent to the device using TCP [**tcplp**], CoAP [**coap**], a specialized protocol like Flush [**flush**], or any other bulk transfer protocol appropriate for the network setup and application. In our Minerva implementation, we use the HTTP library in Zephyr OS [**zephyr**] to pull updates.

Model updates could suffer from data corruption in transit. As an end-to-end integrity check, the Minerva Server includes a checksum with the update each time the client requests a new model version. On the client, the `swap_capsule` function validates the checksum before applying the update. Our implementation uses SHA-256 for the checksum, though simpler checksums could be used instead for better performance.

Our design only protects against *non-adversarial* data corruption. In principle, one could protect against adversarial data corruption by using a secure channel, such as one provided by TLS, to transfer the data.

### Acceptance Testing

A new model update may improve performance for most nodes in a deployed fleet, but this improvement may not be consistent across *all* nodes. To determine if a new update is performs well on a specific edge device, we test the model before applying the update, a process known as "acceptance testing". Capsules provide the flexibility to perform acceptance testing for edge devices (§3.3).

Unfortunately, applying acceptance testing in the edge setting is often complicated by the lack of ground-truth labels needed to evaluate model accuracy. In this section, we show how to perform acceptance testing in the absence of ground-truth labels by applying a modified version of discounted cumulative gain (DCG), which is commonly used in recommendation and ranking systems [3, 48]. The high-level idea is to keep track of the most confident predictions made by the prior model and ensure that the new model largely agrees with the old model on those examples.

To get an estimate of how the new model compares to the old model on the distribution of observations made by the device, we maintain a sample of observations and their corresponding predictions and confidence scores for the currently deployed model. For this paper we consider

a uniform sampling procedure implemented using reservoir sampling but other sampling procedures (e.g., recency biased) could be used depending on the needs of the application. This sample is automatically maintained by the Minerva client library.

To evaluate how the new model's predictions perform relative to the old model, we then run the new model on all the data in the sample generating new predictions and confidence scores. Here we assume that the prediction task is multi-class classification which is common to many edge applications [41, 37, 6, 45, 11, 50]. We compare these predictions and confidence scores using the following metric:

$$DCG_{\text{minerva}} = \sum_{i=1}^{n} \frac{(-1)^{1\{y_i^{\text{old}} != y_i^{\text{new}}\}} (c_i^{\text{new}})}{\log_2 (\sigma(i) + 1)}. \tag{3.1}$$

Here, $n$ is the size of the sample, $1\{y_i^{\text{old}} != y_i^{\text{new}}\}$ indicates the agreement in the labels between the two models, $c_i^{\text{new}}$ is the confidence of the new model in it's prediction $y_i^{\text{new}}$, and $\sigma(i)$ is the rank of example $i$ in descending order of confidence $c_i^{\text{old}}$ for the old model. The magnitude of the scalar $DCG_{\text{minerva}}$ indicates the agreement between the models (higher is better).

Intuitively, this metric rewards agreement and penalizes disagreement on predictions proportional to the confidence of the new model. Meanwhile, the denominator discounts the examples that had low confidence under the old model. This ensures that the new model doesn't degrade performance in settings where the old model was confident. For example, if we have a keyboard auto-correction model, the performance may change (hopefully improve) on rare words, but not on common English words.

Minerva uses the $DCG_{\text{minerva}}$ metric to decide whether to accept the new model or revert to the old model. On downloading the new model, we swap the new capsule in and compute the $DCG_{\text{minerva}}$ using Equation 3.1. If $DCG_{\text{minerva}} <$ threshold then the new model update is retained. Else, we swap back to the old ML model. The threshold is hyperparameter that can be tuned to favor fresh models or stability.

The Minerva approach to acceptance testing has the following advantages: a) Unlike metrics such as accuracy, precision, and recall, we do not need access to the entire data-set to evaluate and compare the two models. This is relevant in our setting, where we have limited memory. b) We also do not need ground-truth labels, which can be hard if not impossible to obtain on the edge. c) The new and the old models are compared locally, preserving bandwidth, and guaranteeing data-privacy. d) As we operate over the latest data on the edge, the Minerva is robust to data-drift and concept-drift.

## Minerva Server

The Minerva Server service takes the model programmed by the application developer and generates a corresponding C implementation. It cross-compiles this implementation for the edge devices and, upon request, transmits the updated capsule contents along with a checksum to a Minerva Client.

```
# Code for model training
...

# Minerva interface
server = MinervaServer(model)
server.save()
server.export()
server.deploy(deviceList)
```

Figure 3.4: Changes to ML training source-code required to interface with Minerva. Only the four additional lines are necessary to interface Minerva with Tensorflow/PyTorch, while the rest of the source-code remains unchanged.

### Compilation

Upon receiving the updated model from the application developer, the Minerva Server service generates the corresponding machine code that will replace the edge device's model. First, C code is generated from the given model. In our implementation, we hand-wrote the translation of models from Python to C, but this system is agnostic to how the C code is generated. In a production environment, TFLite, PyTorch with ONNX and Ell, or a different method could be used to generate the corresponding implementation from the Python source code.

Once the model has been converted to C code, the Minerva Server service includes the new model into the overall application source-code and compiles using an ARM-`gcc` compiler optimized for the target platform. The generated C code includes `.capsule-code`, `.capsule-data`, and `.capsule-entry` annotations that allow the linker to properly assemble the capsule, as described in §3.3. Finally, the Minerva Server extracts the contents of the memory sections which contain the ML graph and model, thus producing the machine-code required by the edge device. These extracted contents are what eventually get sent to the edge device by the Minerva Server during a Minerva update.

Using the Minerva interface, changes to user code are minimal. Minerva can integrate with most existing edge workloads utilizing TensorFlow models for prediction with just six lines of code added to application code on the server. Figure 3.4 shows the changes required to the user's Python source code in order to upload the newly trained model onto a set of edge devices. The user constructs an instance of `MinervaServer`, `save`s and `export`s their new model, and then `deploy`s it to a list of edge devices listed in the `deviceList`. The rest of the code which comes above these final four lines remains completely untouched.

### Determining Update Payload

After generating the new iamge, Minerva Server determines what to send to the device. Capsules are efficient because they only require the ML model to be sent to the device, but

DFUs also have some benefits. With delta updates, DFUs only include the *diff* from the previous image, which could be even smaller than the model. Additionally, while capsules require the new model to fit within the capsule memory segment allocated in the image, DFUs allow any change to the image.

To get the best of both worlds, Minerva combines capsules with delta updates and DFUs to generate an optimized update payload. First, our design is to apply delta updates to capsules, only transferring the *diff* from the previous capsule instead of the full capsule. This reduces the network bandwidth required to download a model update. Second, if the new model code and weights do not fit in the capsule segment on the edge device, then we fall back to a full DFU. In this case, the Minerva Server can increase the size of the capsule segment to fit the new model (plus some additional slack, if desired). That way, future updates that do not further increase the model size can be done efficiently without falling back to a full DFU. We expect large changes that require a full DFU to be rare because models are usually updated incrementally.

While Minerva can use fully general delta updates in principle, our implementation uses a simplified form of delta updates. Our approach is to store operators and weights in separate capsules that can be updated independently. In the common case where an update only changes the model weights, only the data capsule needs to be updated; the code capsule's contents need not be sent over the network. Similarly, if an update only changes the model operators (e.g., for efficiency optimizations), then only the code capsule must be updated. Additionally, Minerva can update just a single function within the code capsule, as long as it does not grow in length. This is accomplished by including a header with each update indicating the update's length and offset within the capsule. The `swap_capsule` function (§3.3) parses the header and updates the code and data capsules in device ROM accordingly.

## 3.5 Evaluation

We evaluate the memory, and latency improvements to ML model updates resulting from using capsules. Our results indicate that capsules significantly reduce the size of the update. This results in commensurate reductions in the time to download the update and re-flash the device. §3.5 presents the microbenchmarks and §3.5-§3.5 present end-to-end experiments. §3.5 evaluates Minerva's DCG-based acceptance testing and §3.5 reports on our real-world deployment of Minerva.

### Experimental Methodology/Set-Up

To understand the performance of Minerva, we measures how long it takes to update ML models on MCUs over a wireless network and compare it to two widely used baselines: a full DFU and an application re-installation onto the Tock embedded OS using Tockloader. We break each measurement down into two main parts. First, we measure how long it takes to download the full DFU or Tock Application Binary (TAB) and contrast it with the time

taken to download just the ML model. Second, we compare the time it takes to re-flash the new firmware image or TAB, versus the time it takes to re-flash just the ML model. For ML model updates we evaluate all the three cases: a) when only the weights need to be updated, b) when only the operators (executable code) need to be updated, and finally c) when the entire ML model (both the operators and weights) need to be updated.

### Full Device Firmware Update

We use an Amazon AWS S3 bucket to store the Device Firmware Update binaries, and a Nordic nRF52840 as the client. The popular nRF52840 has an ARM Cortex-M4 CPU @ 64 MHz with 1 MB ROM and 256 KB RAM. The client requests the update from the server ("pull" model) and downloads the update over cellular IoT on a nRF9160 module. We use a two-bank model for reliability, as discussed in §3.2.

### Updating Applications in Tock

We compare the performance of Minerva to that of Tock. For consistency we again use the Nordic nRF52840 but this time with Tock OS installed. We use the Tockloader tool to re-flash a compiled Tock Application Binary (TAB) file containing the updated ML model onto the board, overwriting and updating an existing application containing the old model. Tockloader uses J-Tag to communicate with the board and thus requires a physical connection; Tock does not support OTA. For the sake of comparison, we give Tock the benefit of doubt and use the nRF52840 network module. Does the TAB include the whole application or just the ML model? Explain how you use the network module?

### Minerva Capsule Update

We use an Amazon AWS EC2 instance for the Minerva server, to train, statically analyze, compile, and generate the optimized update payload. Similar to the DFU update described above, we store the update binaries in the same S3 object store, and use the same hardware for the client—a Nordic nRF52840 which downloads the update over cellular IoT on a nRF9160 module. Computing the optimized update payload always took less than 10 seconds.

### Applications

We perform our evaluations on four real-world applications. Farmbeats is described in §3.2, and we describe the rest here. One undergraduate student was able to port the models from all four applications to use Minerva in less than half a day.

**GesturePod [37]:** GesturePod is a real-time gesture recognition device attached to white canes to help people with visual impairments easily access their phone. As more data is generated by users, newer models are trained, and need to be updated on the user's cane. GesturePod is powered by the MKR1000 development board - an ARM Cortex-M0+ CPU @ 48 MHz with 32 KB of RAM and 256 KB of ROM.

| | Powerblade | Spektacom | GesturePod | Farmbeats |
|---|---|---|---|---|
| Capsule Size (KB) | 20.32 | 31.87 | 7.23 | 6.00 |
| SHA-256 Hash (ms) | 31.49 | 48.96 | 11.20 | 9.28 |
| `memcpy` (ms) | 0.28 | 0.44 | 0.10 | 0.08 |

Table 3.1: Microbenchmarks for a capsule update (milliseconds). Hashing the capsule far outweights the time taken to overwrite the capsule by two orders of magnitude. This suggests that manipulation of the data to provide security and integrity guarantees dominate the time taken to perform the capsule update.

**Spektacom [45]:** Spektacom is a plug-and-play device that boosts the engagement of the audience and players during live sports at stadiums. It is a non-intrusive sticker that collects high-quality sensor data to capture key parameters in real time as the game is played. Like Farmbeats, Spektacom is powered by the nRF52832 MCU, and has a BLE module.
**PowerBlade [11]:** Powerblade is a small, low-power AC plug load meter. It measures real, reactive, and apparent power, and reports this data over BLE radio. ML based local detection saves network bandwidth by allowing Powerblade to transmit just the appliance-class information instead of the complete current and voltage trace. As Powerblade is deployed across homes, more data is ingested, and this is used to constantly train and deploy better models. Powerblade is powered by the nRF51822 MCU with an ARM Cortex-M0 CPU @ 16 MHz, 256 KB of ROM and 32 KB of RAM.

## Microbenchmarks

A Minerva capsule update consists of two parts, hashing the received capsule section to check for data integrity, and using `memcpy` to overwrite the previous capsule contents with the updated contents. As seen in Table **??**, the hash and compare section of the capsule update far outweighs the time it takes to overwrite the capsule contents by two orders of magnitude. Using a lighter weight checksum than SHA-256 could improve performance.

Performing a full OTA DFU involves downloading the update from a server, erasing the memory where the binary will be stored, copying the update into the correct location in memory (re-flash), resetting and swapping the program counter to execute the updated application, and finally rebooting. The time taken to perform the full DFU, as shown in Table 3.2 is largely the download time and re-flash time. By using Minerva, we reduce the size of the updates, and thus significantly reduce the amount of data that needs to be re-flashed and downloaded by many orders of magnitude.

When performing an application update in Tock, we compile the application with the updated ML model into a TAB file which the Tockloader tool can then install onto the board to replace the existing application containing the old model. Table 3.3 shows the estimated

| | **Powerblade** | **GesturePod** | **Farmbeats** |
|---|---|---|---|
| Connect to Server (ms) | 664 | 425 | 374 |
| Download Update (ms) | 85,154 | 69,288 | 78,220 |
| Erase Flash (ms) | 5,364 | 4,656 | 5,235 |
| Re-flash (ms) | 31,272 | 30,876 | 31,312 |
| Reset (ms) & Swap Bank | 84 | 62 | 35 |
| Reboot (ms) | 822 | 511 | 508 |

Table 3.2: Microbenchmarks for full DFU (milliseconds). Downloading the update and reflashing the application occupy on average 94% of the time taken to perform the DFU. This suggests that reducing the size of data needing to be downloaded and reflashed will speed up model updates.

| | **Powerblade** | **GesturePod** | **Farmbeats** |
|---|---|---|---|
| Connect to Server (ms) | 664 | 425 | 374 |
| Download Update (ms) | 84,981 | 43,589 | 39,946 |
| Read Board & App Metadata (ms) | 557 | 569 | 553 |
| Re-flash and Verify (ms) | 7,063 | 6,675 | 6,669 |
| Erase post-userspace flash page (ms) | 5,976 | 6,109 | 6,098 |

Table 3.3: Microbenchmarks for Tockloader application install (milliseconds). The time taken to erase the post-user space flash page is comparable to the application re-flash time itself.

download times of these TABs. I think you should cut the rest of this paragraph! It just describes Tock... Upon starting the re-flash process, Tockloader first establishes a J-Tag connection to the board and reads relevant board metadata as well as the application headers of any pre-existing Tock applications to determine what it needs to replace. It then re-flashes the TAB onto the board and verifies that it was successfully installed at the requested memory location. Finally, Tockloader erases a flash page at the end of the memory region containing the installed application so that the kernel can successfully find the end of the memory region containing user applications. Let's discuss.

## Memory Footprint

Table 3.4 shows that on average, capsules in Minerva (operators and weights) are 0.04× the size of a full DFU and 0.07× the size of the corresponding TAB. This is because non-ML

| | **Powerblade** | **Spektacom** | **GesturePod** | **Farmbeats** |
|---|---|---|---|---|
| Full DFU (KB) | 266.74 | 550.53 | 214.91 | 264.53 |
| TAB (KB) | 266.2 | 266.2 | 135.2 | 135.2 |
| Operators & Weights (KB) | 20.32 | 31.88 | 7.23 | 6.00 |
| Operators (KB) | 1.39 | 0.39 | 0.52 | 0.53 |
| Weights (KB) | 18.93 | 31.48 | 6.72 | 5.47 |

Table 3.4: Memory footprint (KB). Full Device Firmware Updates are approximately 30× machine learning model sizes.

code, including large portions of the code associated with handling the full DFU, are not included in the Minerva capsule. While the TABs generally have a smaller footprint than the full DFU, they still contain Tock standard library modules which make them larger than Minerva capsules.

## Download Time

Figure 3.5 compares the download times for a full DFU, TAB, and Minerva update over a cellular network (LTE-M). Across the board, the time taken to download Minerva capsules are on the order of thousands of milliseconds, while the time taken to download full DFU payloads and TABs are on the order of tens of thousands. The main factor behind these improvements is that Minerva sends less data over the network. Our device does not have enough ROM to execute a full DFU for Spektacom in the two-bank model (i.e., it exceeds 500 KiB). Therefore, we can not gather benchmark data for a DFU of the Spektacom application. However, Minerva adds negligible overhead to the application binary when compared with a full DFU, and we are able to gather benchmark data for Spektacom on Minerva. For applications using lower-rate networks (e.g., Farmbeats, deployed in fields on LoRaWAN networks with $\approx 200$ bits per second), the difference in download time may be even more significant.

## Re-Flash Time

Figure 3.6 compares the re-flash of an entire image, a Tock application update, and a Minerva capsule update. The difference between the time taken to re-flash the Minerva updates compared to both the full DFU and the Tock application update is orders of magnitude in size, showing that Minerva's updates greatly improve upon the efficiency of code updates for this application-space. This is largely because the device does not need to be reset during a Minerva update, and because the memory footprint which must be re-flashed is also far reduced in this case. A Minerva update only updates the code in the capsule in memory and the program can immediately continue execution. Moreover, Table 3.3 also highlights

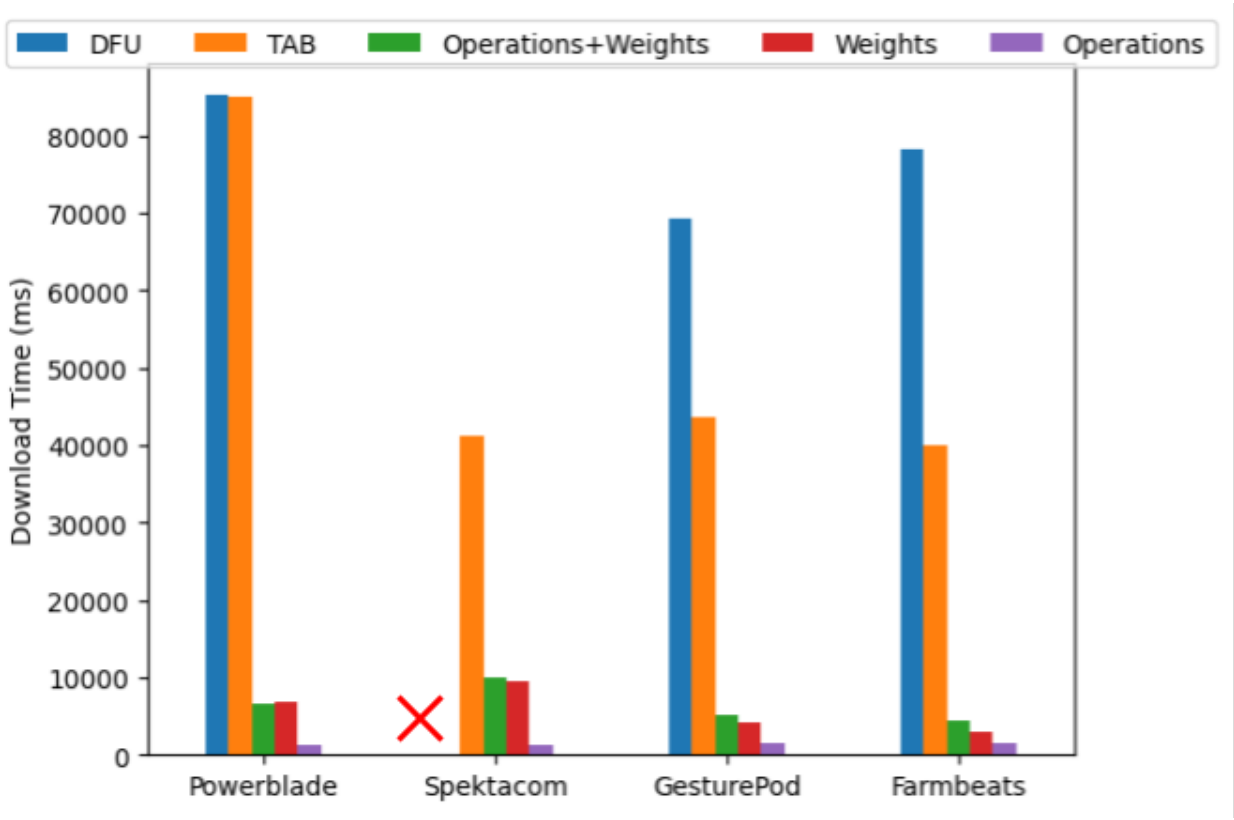


Figure 3.5: Time taken to Download Update. Downloading the full DFU and TAB far outweigh the time taken to download Minerva updates, shown in green {ops+weights}, red {weights}, and purple {ops}. Due to the application size, a DFU is not even possible for Spektacom!

how within a Tock application update a significant portion of time is taken to erase the flash page at the end of user applications after the actual re-flashing of the updated application is complete. This highlights how Minerva updates are not only faster than Tockloader but also more efficient.

## Total Time

A significant portion of total time taken by both DFUs and Tock application updates corresponds to re-flashing the binary. However when performing Minerva updates, the time taken to re-flash almost disappears in comparison to the time taken to download the update (re-flash times in purple are unobservable in the graph, compared to the download times in red). These results clearly emphasize the end-to-end improvements obtained through a Minerva update over a full DFU.

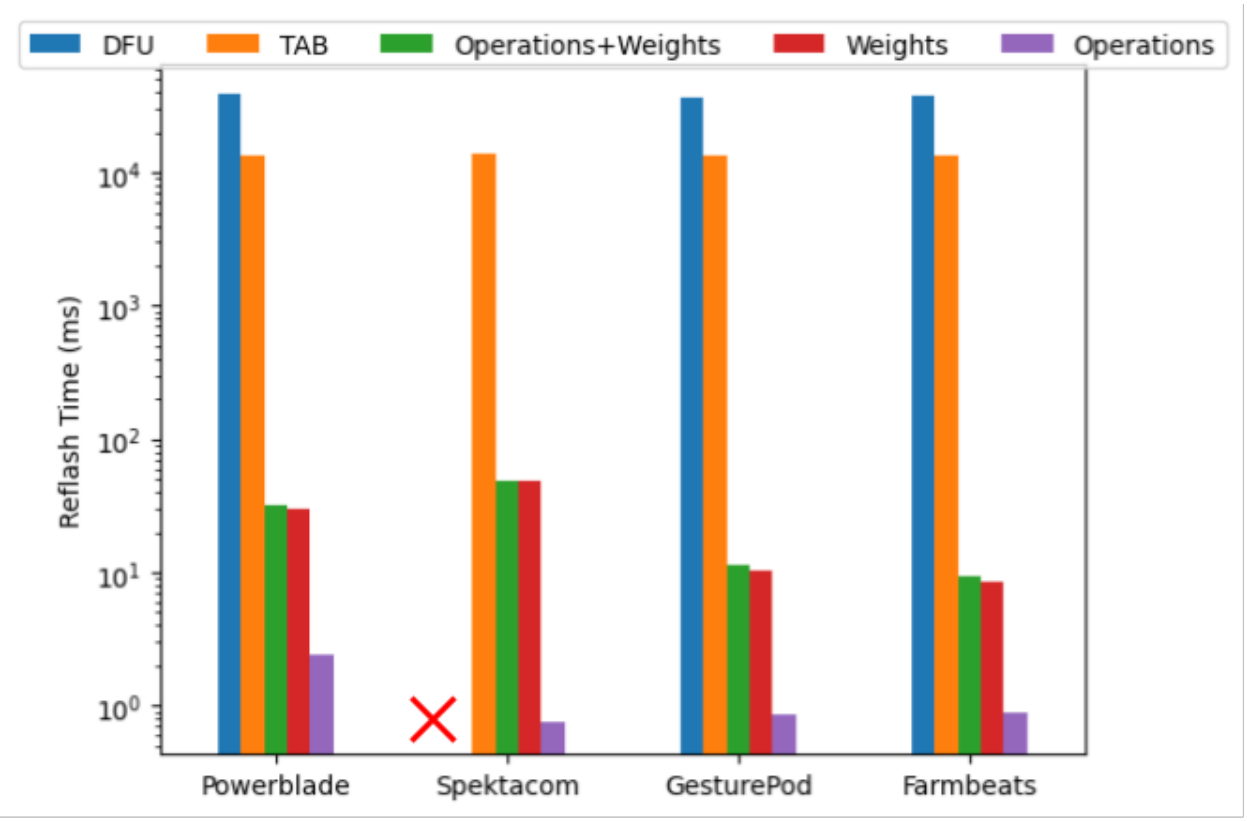


Figure 3.6: Time taken to re-flash update. Notice that this is a log-linear graph. Blue represents reflashing the entire application binary as part of a DFU, orange represents Tockloader re-installing the TAB, and green, red, and purple represent Minerva updates of just the ML model {operators (ops) + weights}, {weights}, and {operators} respectively.

## Acceptance Testing

Each Minerva client can uniquely determine if it should accept a new update or not based on the $DCG_{minerva}$ framework. Evaluating $DCG_{minerva}$ is challenging; even if we track when edge devices accept/reject updates, it is impossible to tell if it was the right decision without ground-truth labels. Hence, we use the GesturePod dataset [37], which contains real raw sensor data together with ground-truth labels, to simulate using $DCG_{minerva}$ to *accept* or *reject* updates. We identified three users from the dataset, *UID 01*, *UID 02*, *UID 03*, to use in our evaluation. These were the users who had common gestures in the both training data and the test data. We trained a classification model to predict the gesture, simulated each user running the model on their local data, and measured the local impact of an update to the model. Because the GesturePod dataset contains ground-truth labels, we can compare the decisions made on the basis of DCG to ground-truth changes in model performance. The results are in Table 3.5. In this evaluation, threshold is set to zero (a hyperparameter) - if

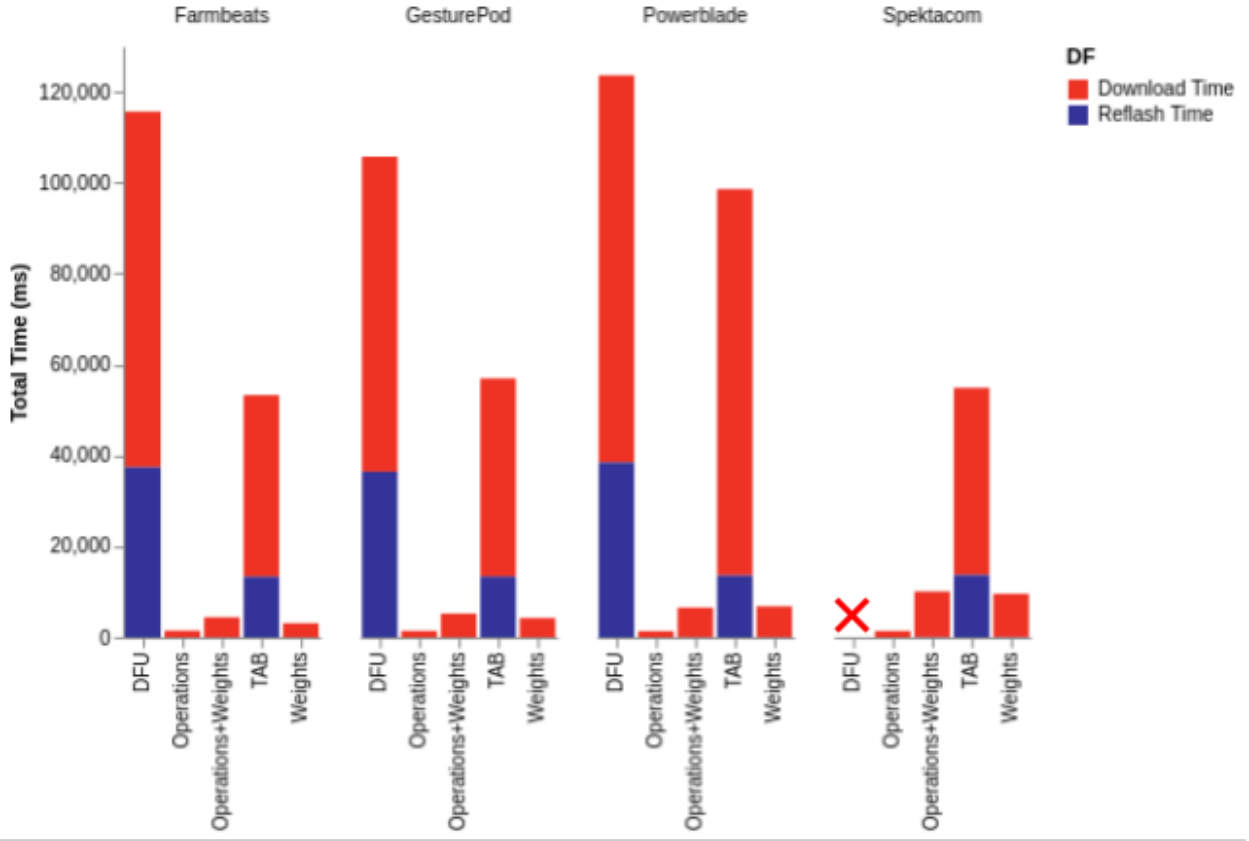


Figure 3.7: Time taken for complete update. This graph combines download time with reflash time. Full DFUs (leftmost bar of each cluster) and TAB re-installs (rightmost bar of each cluster) split time between downloading the update and re-flashing the binary. Minerva update times, on the other hand, are dominated by time taken to download the update. Note that the re-flash time for Minerva updates, shown in blue, are on the order of tens of milliseconds and thus not visible.

the score is positive accept the new model, else retain the original model. When we compare Minerva's predictions with the labels from the dataset, we can demonstrate the efficiency in using $IDCG_{minerva}$ to carry out acceptance testing especially when getting access to labeled data is challenging.

## Deployment Experiences

We plan to open-source Minerva upon publication. We deployed Minerva on two sensor systems that are deployed on energised electric poles (utility poles). These sensors employ a ML model to detect high risk, yet diverse anomaly events such as vegetation contacts, conductor shorts, etc, to detect forest fires. While these sensor systems have been deployed

| | Accuracy w/o Acceptance Testing (%) | w/ Acceptance Testing (%) | $DCG_{minerva}$ score |
|---|---|---|---|
| UID 01 | 73 → **76** | 73 → **76** | 18.42 |
| UID 02 | **76** → 75 | 76 → **76** | -6.94 |
| UID 03 | **63** → 61 | 63 → **63** | -10.73 |

Table 3.5: Accuracy changes for three model updates, with and without acceptance testing, based on ground-truth labels. DCG accurately predicts when model updates are helpful, without ground-truth labels.

on electric poles for utilities across different states, for this deployment, we deployed Minerva on these platforms in northern California, and rural India.

**System set-up:** The sensor systems are powered by an nRF52840 (ARM Cortex M4f) microcontroller. These devices are solar-battery powered with a 30Wh battery, and a full charge can take between 5-10 days depending on weather, season, and deployment conditions. These devices consume about 20mA quiescent, 120mA when transmitting, and 50mA when receiving. They are communicate over LTE-M and LoRa (RN2903 module) depending on availability, at 30kbps and 0.4-1kbps respectively.

**Minerva updates:** With Minerva we updated the ML model on the system everyday, for a month, and faced no challenges or errors even after a month. The performance impact of Minerva was imperceptible during normal operation: the application triggered the update at a fixed time at night everyday, and each model was tested with the DCG metric. In three occasions, due to differences in weather pattern, from data that was used to train the model, the Minerva Edge client rejected the update on the basis of $DCG_{minerva}$.

**Ease of porting:** Minerva has been ported to all four motivating applications in Figure 3.2—Farmbeats [49, 6], GesturePod [37], Spektacom [45], and Powerblade [11]—in addition to the sensor system for electric poles described above. It took an undergraduate student half a day to apply Minerva to all four applications.

## 3.6 Related Work

Prior work most relevant to our system can be broadly classified into two categories—efficient ML on the edge, and efforts in developing an operating system for the edge.

**Efficient ML for Edge:** ML models which achieve state of the art results on classical datasets have largely been exorbitantly expensive for edge devices. In this setting, prior works have proposed techniques such as quantization, sparsification, neural architecture search, etc., to enable machine learning model inference on memory-compute limited edge devices [27, 18]. As described in Figure 3.2, our system benefits from all the above developments. As models get smaller (in terms of memory footprint), Minerva provides greater improvements when compared to performing a full DFU.

There has been recent interest [9, 30] in cloud based prediction serving frameworks that support model updates. Clipper [9] adopted a blackbox view on models and leveraged containers with simple predict APIs to both abstract and isolate individual model logic from the the serving systems. This is similar to our machine learning model capsule though our approach is focused more on separating application and model logic. Pretzel [30] adopted a whitebox view on models and introduced a range of optimizations for prediction pipelines. These optimizations are complementary to our work.

**Operating Systems for Edge Devices:** Prior works have looked at developing OS for edge devices [**sos**, **contiki**, 33, 31, 1].

Among them, [1] do not support dynamic linking/loading, and we need to perform a full device firmware update to modify modules. This is not relevant in our setting. Other operating systems TinyOS [**tinyos-dynamic**], SOS [**sos**], Contiki [**contiki**], and Zephyr [**zephyr**] do support incremental code updates but suffer from the following. SOS's design necessitates the use of position independent code (PIC), which, due to compiler limitations, is not fully supported on common platforms. Contiki uses protothreads as the underlying mechanism, and this requires application modules to be re-written around the protothreads paradigm. These have proven to be impediments for wide spread adoption of SOS and Contiki. TinyOS is tightly coupled with the NesC language - which introduces challenges in porting to the new language. Further, the Tiny Manager running on the edge introduces significant memory and performance overheads (about 7.7% of RAM, and 32% of the program memory). Tockloader makes it possible to update applications on Tock[33], however, it also relies on PIC, and further requires a system re-boot. In fact, it is not possible to perform OTA using Tock, which is a critical requirement in our setting. Lastly, the embedded systems landscape is characterized by heterogeneity, and all the above mentioned operating systems require significant efforts to be ported onto new platforms. Minerva, on the other hand, incurs no performance overheads at runtime, and as a user-level library is easily ported across platforms.

## 3.7 Generalizability of Capsules

On server- and laptop-class devices, service decomposition abounds. For example, microservices decompose applications into independent components, and microkernels do the same for operating systems. Capsules, too, are a form of service decomposition—they decouple ML inference from the rest of the application as an independently updateable service. Can capsules generalize beyond ML inference, to bring a more wholesale form of service decomposition to MCU software?

- Capsules directly generalize to pure functions other than ML inference. Any "pure" algorithms for local data processing would benefit from capsules out-of-the-box, including simple signal processing like threshold detection.

- To support multiple entrypoints, one can place each one in its own capsule, similar to how Minerva separates operators from weights (§3.4). This may incur ROM fragmentation,

as capsules occupy fixed positions in ROM. Alternatively, one can have a single *physical* capsule entrypoint that dispatches to each *logical* entrypoint, like a system call handler.

- Capsules do not cleanly generalize to functions that are stateful or expose internal pointers. A reboot would be required on updates. We cautiously speculate that capsules would still be preferable to DFUs due to lower reflash times.

We leave a full exploration of generalizability to future work.

## 3.8 Conclusion

This systems investigates the model deployment stage of the ML lifecycle for MCU-based IoT devices. Our main insight is that ML inference is a pure function. This enables capsules, a mechanism that allows ML model updates to be deployed to MCU-based edge devices more efficiently and flexibly than full DFUs. We use capsules to build Minerva, an end-to-end model update system, and demonstrate that it is up to two orders of magnitude faster compared to a full DFU and to an application update in a state-of-the-art embedded OS.

# Chapter 4

# Conclusion

The growing complexity and size of ML models pose significant challenges for deployment on edge devices, which are often limited by their memory and compute capabilities. These devices, including smartphones, wearables, and microcontroller-based systems, are integral to our daily lives, yet their constrained resources make it difficult to leverage the full potential of advanced ML models.

The need for local training on these devices is underscored by its benefits: enhancing user privacy, reducing energy consumption, and minimizing bandwidth usage. The Power Optimal Edge Training (POET) algorithm addresses the challenge of training memory-intensive ML models on edge devices with extremely limited memory, as low as 32 KB. By employing a novel mixed-integer linear programming approach, POET optimizes the training process under strict memory and timing constraints, focusing on energy efficiency. This breakthrough allows for the deployment of SotA models, such as BERT, directly on small, low-powered devices. This paves the way for privacy-preserving personalization and real-world applications that were previously unfeasible due to hardware limitations. Future advancements in this field could include integrating activation compression techniques.

The Minerva project focuses on the efficient deployment of cloud-updated ML models in microcontroller-based IoT devices. The key innovation in Minerva is the use of capsules, which exploit the state-less, self-contained nature of ML inference to streamline the model update process. This approach significantly outperforms traditional full device firmware updates (DFUs), achieving update speeds up to two orders of magnitude faster. The capsules enable more frequent and less disruptive updates, a crucial factor in enabling wide spread adoption of ML-based predictions.

Together, POET and Minerva help maintaining the relevance and effectiveness of ML models in rapidly evolving real-world scenarios. These systems take a step in which edge devices are not just passive data collectors but active participants in data analysis, all while maintaining a strong emphasis on user privacy and energy efficiency. As these technologies develop further, we can expect a new-breed of edge-intelligent-applications to emerge!

# Bibliography


[1] Arm. *Mbed OS, Arm.* https://www.mbed.com/en/platform/mbed-os/. Accessed 2020. 2019.

[2] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. "Efficient Combination of Rematerialization and Offloading for Training DNNs". In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato et al. Vol. 34. Curran Associates, Inc., 2021, pp. 23844–23857. URL: https://proceedings.neurips.cc/paper/2021/file/c8461bf13fca8a2b9912ab2eb1668e4b-Paper.pdf.

[3] K. Bhatia et al. *The extreme classification repository: Multi-label datasets and code.* 2016. URL: http://manikvarma.org/downloads/XC/XMLRepository.html.

[4] Davis Blalock et al. "What is the State of Neural Network Pruning?" In: *Proceedings of Machine Learning and Systems*. Ed. by I. Dhillon, D. Papailiopoulos, and V. Sze. Vol. 2. 2020, pp. 129–146. URL: https://proceedings.mlsys.org/paper/2020/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf.

[5] Han Cai, Ligeng Zhu, and Song Han. "ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware". In: *International Conference on Learning Representations*. 2019. URL: https://arxiv.org/pdf/1812.00332.pdf.

[6] Tusher Chakraborty et al. "Fall-curve: A Novel Primitive for IoT Fault Detection and Isolation". In: *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. SenSys '18. Shenzhen, China: ACM, 2018, pp. 95–107. ISBN: 978-1-4503-5952-8. DOI: 10.1145/3274783.3274853. URL: http://doi.acm.org/10.1145/3274783.3274853.

[7] Jianfei Chen et al. "ActNN: Reducing Training Memory Footprint via 2-Bit Activation Compressed Training". In: *International Conference on Machine Learning*. 2021.

[8] Tianqi Chen et al. "Training Deep Nets with Sublinear Memory Cost". In: *CoRR* abs/1604.06174 (2016). arXiv: 1604.06174. URL: http://arxiv.org/abs/1604.06174.

[9] Daniel Crankshaw et al. "Clipper: A Low-Latency Online Prediction Serving System". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 613–627. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw.

[10] Robert David et al. "TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems". In: *Proceedings of Machine Learning and Systems*. Ed. by A. Smola, A. Dimakis, and I. Stoica. Vol. 3. 2021, pp. 800–811. URL: https://proceedings.mlsys.org/paper/2021/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf.

[11] Samuel DeBruin et al. "PowerBlade: A Low-Profile, True-Power, Plug-Through Energy Meter". In: *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. SenSys '15. Seoul, South Korea: ACM, 2015, pp. 17–29. ISBN: 978-1-4503-3631-4. DOI: 10.1145/2809695.2809716. URL: http://doi.acm.org/10.1145/2809695.2809716.

[12] Don Kurian Dennis et al. "EdgeML: Machine Learning for resource-constrained edge devices". In: http://github.com/Microsoft/EdgeML. 2019. URL: https://github.com/Microsoft/EdgeML.

[13] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[14] Zhen Dong et al. "HAWQ: Hessian AWare Quantization of Neural Networks With Mixed-Precision". In: *The IEEE International Conference on Computer Vision (ICCV)*. Oct. 2019.

[15] Jonathan Frankle and Michael Carbin. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=rJl-b3RcF7.

[16] João Gama et al. "A survey on concept drift adaptation". In: vol. 46. 4. ACM New York, NY, USA, 2014, pp. 1–37.

[17] Andreas Griewank and Andrea Walther. "Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation". In: *ACM Trans. Math. Softw.* 26.1 (Mar. 2000), pp. 19–45. ISSN: 0098-3500. DOI: 10.1145/347837.347846. URL: https://doi.org/10.1145/347837.347846.

[18] C. Gupta et al. "ProtoNN: Compressed and Accurate kNN for Resource-scarce Devices". In: *Proceedings of the International Conference on Machine Learning*. Aug. 2017.

[19] Andrew Hard et al. "Federated learning for mobile keyboard prediction". In: *arXiv preprint arXiv:1811.03604* (2018).

[20] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[21] Chien-Chin Huang, Gu Jin, and Jinyang 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*. 2020, pp. 1341–1355.

[22] Forrest N. Iandola et al. *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5MB model size*. 2016. arXiv: 1602.07360 [cs.CV].

[23] Paras Jain et al. "Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization". In: *arXiv preprint arXiv:1910.02653* (2020).

[24] Junsu Jang and Fadel Adib. "Underwater Backscatter Networking". In: *Proceedings of the ACM Special Interest Group on Data Communication*. SIGCOMM '19. Beijing, China: Association for Computing Machinery, 2019, pp. 187–199. ISBN: 9781450359566. DOI: 10.1145/3341302.3342091. URL: https://doi.org/10.1145/3341302.3342091.

[25] Gaurav et al. Kapoor. "CoreML, Apple". In: 2019. URL: https://developer.apple.com/documentation/coreml.

[26] Marisa Kirisame et al. "Dynamic Tensor Rematerialization". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=Vfs_2RnOD0H.

[27] A. Kumar and M. Goyal S. amd Varma. "Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things". In: *Proceedings of the International Conference on Machine Learning*. Aug. 2017.

[28] Nicholas D. Lane et al. "DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices". In: *Proceedings of the 15th International Conference on Information Processing in Sensor Networks*. IPSN '16. Vienna, Austria: IEEE Press, 2016. ISBN: 9781509008025.

[29] Juhyun Lee et al. "On-Device Neural Net Inference with Mobile GPUs". In: *CoRR* abs/1907.01989 (2019). arXiv: 1907.01989. URL: http://arxiv.org/abs/1907.01989.

[30] Yunseong Lee et al. "PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 611–626. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/lee.

[31] Philip Levis et al. "TinyOS: An operating system for sensor networks". In: *Ambient intelligence*. Springer, 2005, pp. 115–148.

[32] Philip Levis et al. "Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks". In: *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*. NSDI'04. San Francisco, California: USENIX Association, 2004, p. 2.

[33] Amit Levy et al. "Multiprogramming a 64 kB Computer Safely and Efficiently". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: ACM, 2017, pp. 234–251. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132786. URL: http://doi.acm.org/10.1145/3132747.3132786.

[34] Tian Li et al. "Federated Learning: Challenges, Methods, and Future Directions". In: *IEEE Signal Processing Magazine* 37.3 (2020), pp. 50–60. DOI: 10.1109/MSP.2020.2975749.

[35] E. Park, J. Ahn, and S. Yoo. "Weighted-Entropy-Based Quantization for Deep Neural Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017, pp. 7197–7205. DOI: 10.1109/CVPR.2017.761.

[36] Shishir G Patil et al. "Poet: Training neural networks on tiny devices with integrated rematerialization and paging". In: *International Conference on Machine Learning*. PMLR. 2022, pp. 17573–17583.

[37] Shishir G. Patil et al. "GesturePod: Enabling On-device Gesture-based Interaction for White Cane Users". In: *Proceedings of the 32Nd Annual ACM Symposium on User Interface Software and Technology*. UIST '19. New Orleans, LA, USA: ACM, 2019, pp. 403–415. ISBN: 978-1-4503-6816-2. DOI: 10.1145/3332165.3347881. URL: http://doi.acm.org/10.1145/3332165.3347881.

[38] David Patterson et al. "The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink". In: *arXiv preprint arXiv:2204.05149* (2022).

[39] Matthias Paulik et al. "Federated Evaluation and Tuning for On-Device Personalization: System Design & Applications". In: *CoRR* abs/2102.08503 (2021). arXiv: 2102.08503. URL: https://arxiv.org/abs/2102.08503.

[40] Quan Peng et al. "Capuchin: Tensor-based GPU Memory Management for Deep Learning". In: *ASPLOS*. Mar. 2020. URL: https://www.microsoft.com/en-us/research/publication/capuchin-tensor-based-gpu-memory-%5C%20management-for-deep-learning/.

[41] PicoVoice. *Edge Voice AI Platform*. https://picovoice.ai/. Accessed 2020. 2019.

[42] Jie Ren et al. "ZeRO-Offload: Democratizing Billion-Scale Model Training". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 551–564. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/ren-jie.

[43] Aashaka Shah et al. "Memory Optimization for Deep Networks". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=bnY0jm4l59.

[44] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[45] Spektacom. "Spektacom Inc." In: 2019. URL: https://www.spektacom.com/.

[46] Mingxing Tan and Quoc V. Le. "EfficientNetV2: Smaller Models and Faster Training". In: *CoRR* abs/2104.00298 (2021). arXiv: 2104.00298. URL: https://arxiv.org/abs/2104.00298.

[47] Alexey Tsymbal. "The Problem of Concept Drift: Definitions and Related Work". In: May 2004.

[48] Hamed Valizadegan et al. "Learning to Rank by Optimizing NDCG Measure". In: *Advances in Neural Information Processing Systems 22*. Ed. by Y. Bengio et al. Curran Associates, Inc., 2009, pp. 1883–1891. URL: http://papers.nips.cc/paper/3758-learning-to-rank-by-optimizing-ndcg-measure.pdf.

[49] Deepak Vasisht et al. "Farmbeats: An iot platform for data-driven agriculture". In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 515–529.

[50] Nicolas Villar et al. "Project Zanzibar: A Portable and Flexible Tangible Interaction Platform". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: Association for Computing Machinery, 2018. ISBN: 9781450356206. DOI: 10.1145/3173574.3174089. URL: https://doi.org/10.1145/3173574.3174089.

[51] Yue Wang et al. "E2-Train: Training State-of-the-art CNNs with Over 80% Energy Savings". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 5138–5150. URL: http://papers.nips.cc/paper/8757-e2-train-training-state-of-the-%5C%20art-cnns-with-over-80-energy-savings.pdf.