Efficient ML Model Updates for Deeply Embedded Microcontrollers

Shishir G. Patil
shishirpatil@berkeley.edu
UC Berkeley

Sam Kumar
samkumar@cs.ucla.edu
UCLA

Prabal Dutta
prabal@berkeley.edu
UC Berkeley

Joseph E. Gonzalez
jegonzal@berkeley.edu
UC Berkeley

Abstract

Frequent updates to machine learning (ML) models are essential for maintaining accuracy in dynamic environments. However, updating models deployed on IoT devices with low-power microcontrollers remains challenging. Because the model, application, and system are all linked together into a monolithic program, model updates are typically accomplished through full device firmware updates (DFUs). Unfortunately, DFUs are disruptive, inflexible, energy-intensive, and inefficient. We present Minerva, an end-to-end ML model update system for microcontroller-based IoT devices. At the heart of Minerva is a new abstraction and mechanism called *capsules*. Capsules encapsulate ML models as pure, independent functions and decouple the model from the rest of the system. These properties enable updates that are *lightweight*, avoiding the time and energy costs of a DFU both in communication and flash reprogramming, and *non-disruptive*, eliminating the need for a full reboot. Because capsules work at the application level, they require no modifications to the underlying OS or firmware, and thus apply broadly to existing IoT systems. Minerva reduces model update time by up to 89× compared to a full DFU, and up to 73× compared to a state-of-the-art embedded OS, enabling the growing number of deeply embedded devices to receive regular model updates. Minerva is an open-source project available at https://github.com/ShishirPatil/minerva.

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 [14, 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 throughout the entire lifecycle of an ML deployment.

This work is licensed under a Creative Commons Attribution 4.0 International License.
EUROSYS '26, Edinburgh, Scotland Uk
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3803613

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), for example, those built around Cortex-M family of ARM CPUs. For example, Farmbeats [49] uses microcontrollers in sensor boxes for precision agriculture. Recent efforts have looked at designing models for the constraints of the edge [42, 49, 50]. In particular, *continual learning* and *data-drift* scenarios—where application logic and system code remain stable over long periods while models evolve frequently (e.g., multiple times per day) in response to changing environmental conditions—drive frequent model updates at the edge [14, 47]. However, this requires updated models that are trained centrally, e.g. in the cloud, to be deployed on edge devices to improve prediction accuracy 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) [15, 20]. 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 [4, 25, 40, 44]. 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 [21]. But DFUs have several downsides, exacerbated in the context of model updates. First, although most updates only change the model (especially in continual learning or data-drift scenarios [14, 47]), 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. In practice, edge devices update models less frequently or not at all due to these downsides [6, 49].

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 [24, 28, 51], which are represented as a *diff* from the previous image instead of as a fresh new image. Unfortunately, this still requires device reboots, needs to store and restore context, and lacks the flexibility of the multi-process model. We discuss the limitations of delta updates further in §3.1.

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, a 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. ML models are pure functions—they do not maintain state across invocations or expose pointers to their internal code. 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 [7, 38].

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 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.

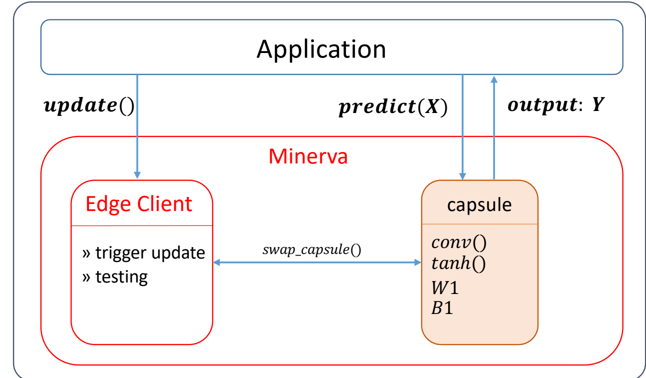

Figure 1. Minerva explores a user-space library design that interposes between ML models and the rest of the application. The operators (code) and weights are stored in self-contained and isolated capsules enabling rapid update of ML models transparent to the application logic.

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 [32] and Tock [35], provide general dynamic loading facilities that we could use instead of capsules. However, they are far more intrusive than capsules, require 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 demonstrate a novel technique to locally test the utility of model updates that extends an established statistical technique, 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()`).
Real-world Deployment: Minerva has been deployed in rural areas of California and India for over a month. The application uses ML to predict environmental factors, and the ML model was updated once per day.
Open Source: We open-source Minerva to facilitate research in ML on the edge.

This paper makes the following contributions. (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.

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 [37, 49], structural health monitoring [27, 52], and IoT [42].

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 [16]. 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 [42], Picovoice [43], Powerblade [11], Zanzibar [50] and Permamote [22]. In these deployments, application logic and system code are stable over long periods, while models evolve frequently due to data-drift. For example, Farmbeats benefits from daily model updates as soil and weather conditions change; electric grid monitoring systems (e.g., Gridware) may update models multiple times per day during storms, floods, and other weather events; and GesturePod updates models as new user data is collected.

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 a model—that is, to both the operators and weights. Additionally, as we shall explain in §2.3, models are often only a small fraction of the overall image size.

2.1 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, ≤ 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 [45] or Atmel SAM R21 series [3].

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 ≈ 1 MiB of RAM per MIPS of CPU (3M rule), the nRF52832 MCU, used by Farmbeats, has ≈ 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 [19, 25, 34].

Energy is a significant limitation as well. Deeply embedded devices may run off of a battery with limited capacity, or using an unreliable power source (e.g., a nearby solar panel), requiring them to use energy sparingly. Energy consumption on MCUs is dominated by radio usage and flash writes, both of which scale linearly with the number of bytes transferred or written [36]. In this context, the energy cost of DFUs can be significant, and reducing the update payload size translates directly to proportional energy savings.

2.2 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 [33], Contiki OS [12], or RIOT OS [25], 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 [18].

A notable exception is Tock OS [35]. 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) for 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 Minerva's performance to Tock's.

2.3 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 [23]. 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 2, the model is only a small fraction of the update size. 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 ("read-only memory"), 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 §5. The large size of DFUs exacerbates this overhead.

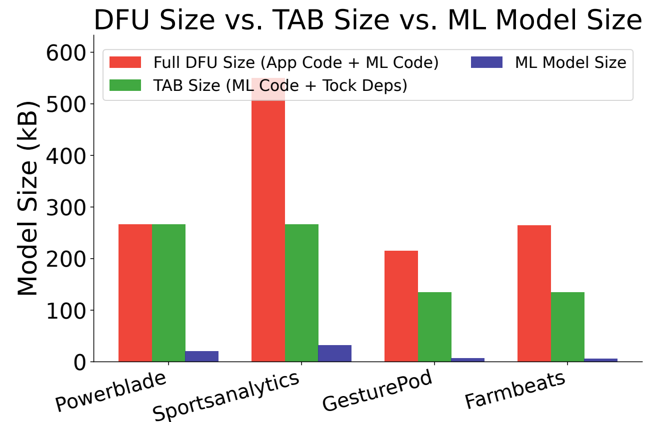

Figure 2. Comparison of machine learning model size to full DFU size and TAB size (Tock Application Binary, a compiled application to be loaded onto a board with Tock OS) for several edge applications. ML model size (blue) includes ML operators and weights. TAB Size includes the ML model size and Tock 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.

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 it fails, then the device can still boot the old image. While the two-bank model improves 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 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 §4.

3.1 Motivation and Opportunities

As described in §2.3, 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 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 [28] and are adopted in certain embedded OSes [24, 51]. While delta updates can significantly reduce the network bandwidth required to obtain the image, other issues with DFUs remain: delta updates still require, the two-bank model, the need to reboot, and inflexibility, remain.

Importantly, delta updates do not fully reduce reflash time, the memory required to support updates, or the energy cost of an update. The reason is that the two-bank model, the updated firmware must be written to the second bank; the process of writing the updated image to ROM is expensive in time and energy (§2.3), and the second bank in memory must be as large as the entire firmware image.

Importantly, these limitations are fundamental to delta updates and are not specific to the two-bank model. To understand why, 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 (§2.3). 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 Minerva capsules aim to do.

3.2 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.

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 [7] requires an update to transform the old state appropriately, and Snap [38] 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. Unlike general live patching and dynamic software updating systems, which must perform explicit memory-state transformation—i.e., migrating global variables, heap objects, and data structures from the old firmware layout to the new one after code and data layout changes [7, 38]—capsules avoid such transformation entirely by construction. The only requirement is that 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.

3.3 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 ensure that the corresponding code or data are placed 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.

Converting ML models to C is a standard part of deploying models on MCUs and does not impose additional developer burden. ML frameworks such as TensorFlow Lite Micro [10] and PyTorch (via ONNX) automatically compile models from Python into self-contained C code with no external dependencies. Once the compiler outputs C code, annotating it for use with Minerva requires only adding `__attribute__((section(...)))` lines above each function and data item—approximately 10 lines per application—a step that could be trivially automated. While our current implementation targets C/C++, the same technique could apply to Rust, which we believe supports similar linker attributes (e.g., `#[link_section = ".capsule-data"]`).

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.

3.4 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 new 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 predictions generated by the new model are adequate. We provide a mechanism for this in §4.1.2.

The update buffer can be viewed as a second bank, similar to the two-bank model for DFUs (§2.3). 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, 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`.

Crucially, `predict` must not be called concurrently with `swap_capsule`. As synchronization mechanisms depend on

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

Figure 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 concurrency model and scheduler of the embedded OS, we leave it to the application to synchronize calls to `predict` and `swap_capsule`. 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.

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.

4.1 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.

4.1.1 Data Transfer and Integrity. With capsules, the model update can be sent to the device using TCP [30], CoAP [5], a specialized protocol like Flush [26], 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 [41] 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. We implement SHA-256 checksums, though simpler checksums could be used instead for better performance.

Minerva assumes that the network channel used for updates is free from adversarial tampering, consistent with many existing embedded DFU mechanisms [13]. Such a network channel can be obtained by using a cryptographic protocol like TLS. Re-using fixed function addresses in capsule updates does not introduce attack vectors beyond those already present in regular DFUs. Standard countermeasures—secure boot, signed capsules, and encrypted channels (e.g., TLS)—apply directly to Minerva and are orthogonal to the capsule abstraction. In the current implementation, Minerva protects against non-adversarial data corruption via SHA-256 checksums.

4.1.2 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 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 do acceptance testing for edge devices (§3.4).

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 [9, 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 [6, 11, 42, 43, 50]. We compare these predictions and confidence scores using the following metric:

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

Here, n is the size of the sample, $\mathbb{1}\{y_i^{\text{old}}! = y_i^{\text{new}}\}$ indicates the agreement in the labels between the two models, c_i^{new} is the

confidence of the new model in its prediction y_i^{new}, and $\sigma(i)$ is the rank of example i in descending order of confidence c_i^{old} for the old model. DCG_{minerva} is a scalar indicating the agreement between the two 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 does not 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_{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_{minerva} using Equation 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 dataset 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, Minerva is robust to data-drift and concept-drift.

4.2 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.

4.2.1 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.

```
# Code for model training
...

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

Figure 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.

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 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`, `saves` and `exports` their new model, and then `deploys` 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.

4.2.2 Determining Update Payload. After generating the new image, 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—we 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 (§3.4) parses the header and updates the code and data capsules in device ROM accordingly.

5 Evaluation

We evaluate the size 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. §5.2 presents microbenchmarks and §5.3-§5.6 present end-to-end experiments. §5.7 evaluates Minerva's DCG-based acceptance testing and §5.8 reports on our real-world deployment of Minerva.

5.1 Experimental Methodology/Setup

To understand the performance of Minerva, we measure 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.

5.1.1 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 §2.3.

5.1.2 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.

5.1.3 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.

5.1.4 Applications. We perform our evaluations on four real-world applications. Farmbeats is described in §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 [42]: 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.

Sportsanalytics: In Sportsanalytics we consider a plug-and-play device that boosts the engagement of the audience and players during live sports at stadiums. It is a non-intrusive embedded device that collects high-quality sensor data to capture key parameters in real time as the game is played. Like Farmbeats, Sportsanalytics 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.

	Powerblade	Sportsanalytics	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 1. Microbenchmarks for a capsule update (milliseconds). Hashing the capsule far outweighs 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 dominates the time taken to perform the capsule update.

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

5.2 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 1, 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 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.

	Powerblade	Sportsanalytics	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 4. Memory footprint (KB). Full Device Firmware Updates are approximately 30× machine learning model sizes.

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 shows the estimated download times of these TABs. 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.

5.3 Size Reduction

Table 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 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.

The small size of Minerva capsules brings two significant benefits. First, it reduces the memory overhead of supporting model updates, because Minerva's update buffer (§3.4) only needs to be as large as the capsule. In contrast, with full DFUs, each bank in the two-bank model must be large enough to fit the entire firmware image. Second, it reduces the energy consumption of a model update. The energy cost of a firmware update is linear in the size of the update, due to the cost of transferring the update over the network and of writing to ROM. We expect that the smaller size of capsules in Minerva compared to DFUs would translate to commensurately lower energy consumption.

5.4 Download Time

Figure 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 is on the order of thousands of milliseconds, while the time

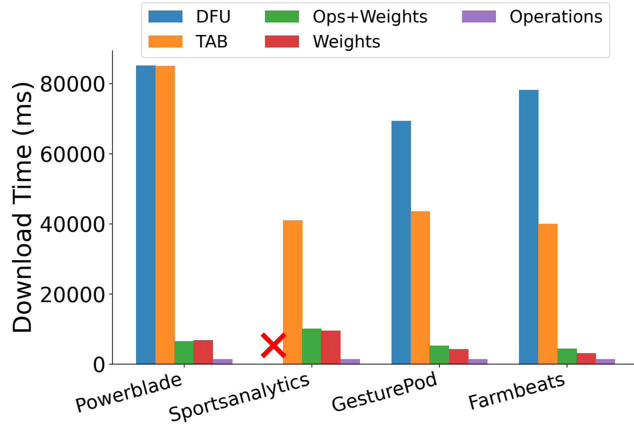

Figure 5. Time taken to Download Update. Downloading the full DFU and TAB far outweigh the time taken to download Minerva updates, shown in green {ops+weights}, red {weights}, and purple {ops}. Due to the application size, a DFU is not even possible for Sportsanalytics!

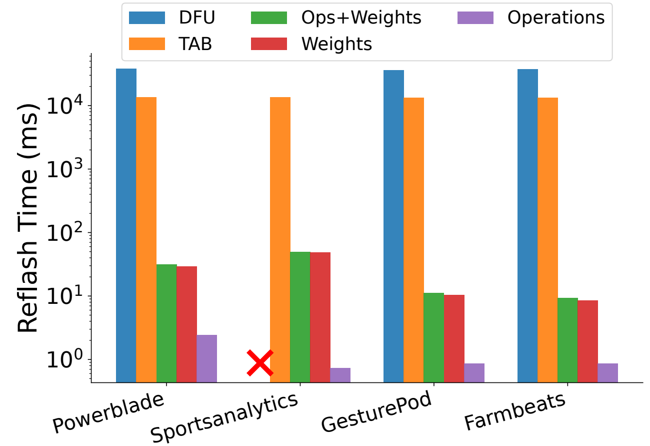

Figure 6. Time taken to re-flash update. Notice that this is a log-linear graph. Blue represents reflashing the entire application binary as part of a DFU, orange represents Tockloader re-installing the TAB, and green, red, and purple represent Minerva updates of just the ML model, just the weights, and just the operators respectively.

taken to download full DFU payloads and TABs is 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 Sportsanalytics in the two-bank model (i.e., it exceeds 500 KiB). Therefore, we cannot gather benchmark data for a DFU of the Sportsanalytics 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 Sportsanalytics 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.

5.5 Re-Flash Time

Figure 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 also highlights 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.

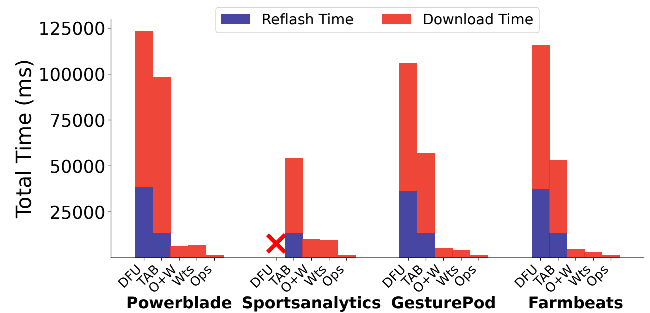

Figure 7. Time taken for complete update. This graph combines download time with reflash time. Full DFUs (leftmost bar of each cluster) and TAB re-installs (rightmost bar of each cluster) split time between downloading the update and re-flashing the binary. Minerva update times, on the other hand, are dominated by time taken to download the update. Note the re-flash time for Minerva updates, shown in blue, are on the order of tens of milliseconds and thus not visible.

5.6 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) in Figure 7. These results clearly emphasize the end-to-end improvements obtained through a Minerva update over a full Device Firmware Update.

	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 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.

5.7 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 [42], 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 both the 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 5. In this evaluation, threshold is set to zero (a hyperparameter) - if 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.

5.8 Deployment Experiences

Minerva is an open-source project. We deployed Minerva on two sensor systems that are deployed on energised electric poles (utility poles). These sensors employ an 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 on electric poles for utilities across different states, for this deployment, we deployed Minerva on these platforms in northern California, and rural India.

System setup: These 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 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 every day, 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 every day, and each model was tested with the DCG metric. On three occasions, due to differences in weather patterns from the data 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 2—Farmbeats [6, 49], GesturePod [42], Sportsanalytics, 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.

6 Related Work

Delta updates [24, 28, 51] are a technique for reducing the bandwidth of DFUs. Minerva brings important benefits that delta updates alone do not—it reduces memory overhead and energy consumption (see §5.3 for details), avoids reboots, and improves flexibility. We describe why delta updates do not bring these benefits in §3.1. We also discuss how delta updates can be used with Minerva in §4.2.2.

Efficient ML for Edge: ML models that achieve state-of-the-art results on classical datasets are exorbitantly expensive for edge devices. In this setting, prior works have proposed techniques such as quantization, sparsification, neural architecture search, etc., to enable ML model inference on memory-compute limited edge devices [16, 29]. Minerva benefits from all the above developments (Figure 2). As models get smaller (in memory footprint), Minerva provides greater improvements when compared to performing a full DFU.

There is growing interest [8, 31] in cloud based prediction serving frameworks that support model updates. Clipper [8] 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 ML model capsule, but our approach is focused more on separating application and model logic. Pretzel [31] 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 [2, 12, 17, 33, 35]. Among them, Mbed OS [2] does not support dynamic linking/loading, and it requires a full device firmware update to modify modules. This is not necessary in our setting. Other operating systems TinyOS [39], SOS [17], Contiki [12], and Zephyr [41] 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 [35], 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. We note that Tock uses the term "capsules" to refer to untrusted Rust modules that run in the kernel; unlike Minerva capsules, Tock capsules cannot be dynamically loaded or replaced at runtime. Tock "processes" are more analogous to Minerva capsules, but they are OS-level abstractions that require a specific runtime, PIC support, and kernel involvement, whereas Minerva capsules operate entirely at the application level and require no OS changes.

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.

Live Patching: Live patching techniques for general-purpose systems [46] allow updating running code without a reboot, but they must handle complex state migration and memory layout changes. Minerva avoids this complexity entirely by exploiting the purity of ML inference—capsule updates require no state transformation by construction.

Secure DFUs: The advancements on secure DFUs for IoT (e.g., [13]); the security mechanisms described therein (e.g., signed firmware, encrypted channels) are orthogonal to and compatible with Minerva.

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 updatable 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 (§4). This may incur ROM fragmentation, as capsules occupy fixed positions in ROM. Or, one can have a single *physical* capsule entrypoint that dispatches to each *logical* entrypoint, like a syscall 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 may still be preferable to DFUs due to lower reflash times.

We leave a full exploration of generalizability to future work.

8 Conclusion

This paper 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. Beyond raw latency improvements, capsules enable capabilities not possible with DFUs: local acceptance testing without a reboot, rollback to a previous model without restoring checkpoints, and model updates on devices where DFUs are infeasible due to ROM constraints. 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.

Availability

Minerva is an open-source project available at https://github.com/ShishirPatil/minerva and on Zenodo [1].

Acknowledgments

We would like to thank Tim Barat, and Riley Lyman for their contributions to the project. This work was supported by the iCyPhy (Industrial Cyber-Physical Systems) Research Center, including in part by Denso, as well as an Okawa Foundation Research Grant, and the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This research is also supported by a NSF CISE Expeditions Award CCF-1730628, and gifts from Amazon Web Services, Ant Group, Ericsson, Facebook, Google, Intel, Microsoft, Scotiabank, and VMware.

References

[1] Minerva (zenodo). 2026. https://doi.org/10.5281/zenodo.18861016.

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

[3] Atmel. Atmel SAM R21E / SAM R21G, SMART ARM-based wireless microcontroller, 2016. https://www.mouser.com/ds/2/268/Atmel-42223-SAM-R21_Datasheet-1065540.pdf.

[4] Atmel Corporation. *Low Power, 2.4GHz Transceiver for ZigBee, RF4CE, IEEE 802.15.4, 6LoWPAN, and ISM Applications*, 2014. Preliminary Datasheet.

[5] Carsten Bormann, Angelo P. Castellani, and Zach Shelby. CoAP: An application protocol for billions of tiny Internet nodes. *IEEE Internet Computing*, 16(2):62–67, 2012.

[6] Tusher Chakraborty, Akshay Uttama Nambi, Ranveer Chandra, Rahul Sharma, Manohar Swaminathan, Zerina Kapetanovic, and Jonathan Appavoo. 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, pages 95–107, New York, NY, USA, 2018. ACM.

[7] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. Remote procedure call as a managed system service. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 141–159, Boston, MA, April 2023. USENIX Association.

[8] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[9] Kunal Dahiya, Deepak Saini, Anshul Mittal, Ankush Shaw, Kushal Dave, Akshay Soni, Himanshu Jain, Sumeet Agarwal, and Manik Varma. DeepXML: A deep extreme multi-label learning framework applied to short text documents. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*, pages 31–39, 2021.

[10] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, Rocky Rhodes, Tiezhen Wang, and Pete Warden. Tensorflow lite micro: Embedded machine learning on tinyml systems, 2020.

[11] Samuel DeBruin, Branden Ghena, Ye-Sheng Kuo, and Prabal Dutta. PowerBlade: A low-profile, true-power, plug-through energy meter. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, 2015.

[12] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki – A lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, page 455–462, USA, 2004. IEEE Computer Society.

[13] Saad El Jaouhari and Eric Bouvet. Secure firmware over-the-air updates for IoT: Survey, challenges, and discussions. *Internet of Things*, 18:100508, 2022.

[14] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):1–37, 2014.

[15] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. ProtoNN: Compressed and accurate kNN for resource-scarce devices. In *Proceedings of the International Conference on Machine Learning*, August 2017.

[17] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, page 163–176, New York, NY, USA, 2005. Association for Computing Machinery.

[18] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, page 93–104, New York, NY, USA, 2000. Association for Computing Machinery.

[19] Jonathan W. Hui and David E. Culler. IP is dead, long live IP for wireless sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, page 15–28, New York, NY, USA, 2008. Association for Computing Machinery.

[20] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, Boston, MA, July 2023. USENIX Association.

[21] Muhammad Ibrahim, Andrea Continella, and Antonio Bianchi. Aot-attack on things: A security analysis of iot firmware updates. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, 2023.

[22] Neal Jackson, Joshua Adkins, and Prabal Dutta. Capacity over capacitance for reliable energy harvesting sensors. In *The 18th International Conference on Information Processing in Sensor Networks*, IPSN'19. ACM, April 2019.

[23] Junsu Jang and Fadel Adib. Underwater backscatter networking. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 187–199, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Jan Jongboom. Towards firmware updates over LoRa: cryptography and delta updates.

[25] Hyung-Sin Kim, Michael P Andersen, Kaifei Chen, Sam Kumar, William J. Zhao, Kevin Ma, and David E. Culler. System architecture directions for post-SoC/32-bit networked sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, page 264–277, New York, NY, USA, 2018. Association for Computing Machinery.

[26] Sukun Kim, Rodrigo Fonseca, Prabal Dutta, Arsalan Tavakoli, David Culler, Philip Levis, Scott Shenker, and Ion Stoica. Flush: A reliable bulk transport protocol for multihop wireless networks. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, page 351–365, New York, NY, USA, 2007. Association for Computing Machinery.

[27] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steven Glaser, and Martin Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *2007 6th International Symposium on Information Processing in Sensor Networks*, pages 254–263, 2007.

[28] Ryozo Kiyohara, Satoshi Mii, Mitsuhiro Matsumoto, Masayuki Numao, and Satoshi Kurihara. A new method of fast compression of program code for OTA updates in consumer devices. *IEEE Transactions on Consumer Electronics*, 55(2):812–817, 2009.

[29] Ashish Kumar, Saurabh Goyal, and Manik Varma. Resource-efficient machine learning in 2 KB RAM for the Internet of Things. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1935–1944. PMLR, 06–11 Aug 2017.

[30] Sam Kumar, Michael P Andersen, Hyung-Sin Kim, and David E. Culler. Performant TCP for Low-Power wireless networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 911–932, Santa Clara, CA, February 2020. USENIX Association.

[31] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, Carlsbad, CA, October 2018. USENIX Association.

[32] Philip Levis and David Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 85–95, New York, NY, USA, 2002. Association for Computing Machinery.

[33] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. TinyOS: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[34] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *First Symposium on Networked Systems Design and Implementation (NSDI 04)*, San Francisco, CA, March 2004. USENIX Association.

[35] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64 kB computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 234–251, New York, NY, USA, 2017. ACM.

[36] Farouk Mahfoudhi, Ashish Kumar Sultania, and Jeroen Famaey. Over-the-air firmware updates for constrained NB-IoT devices. *Sensors*, 22(19):7572, 2022.

[37] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, WSNA '02, page 88–97, New York, NY, USA, 2002. Association for Computing Machinery.

[38] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[39] Waqaas Munawar, Muhammad Hamad Alizai, Olaf Landsiedel, and Klaus Wehrle. Dynamic TinyOS: Modular and transparent incremental code-updates for sensor networks. In *2010 IEEE International Conference on Communications*, pages 1–6, 2010.

[40] Nordic Semiconductor. nRF52840 system on chip: Multiprotocol Bluetooth 5.4 SoC supporting Bluetooth LE, Bluetooth mesh, NFC, Thread and Zigbee.

[41] Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P. Midkiff. Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, page 32, USA, 2009. USENIX Association.

[42] Shishir G. Patil, Don Kurian Dennis, Chirag Pabbaraju, Nadeem Shaheer, Harsha Vardhan Simhadri, Vivek Seshadri, Manik Varma, and Prateek Jain. 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, pages 403–415, New York, NY, USA, 2019. ACM.

[43] PicoVoice. Edge voice AI platform. https://picovoice.ai/, 2019. Accessed 2020.

[44] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*. ACM/IEEE, 2005.

[45] Nordic Semiconductor. nRF52 Series. https://infocenter.nordicsemi.com/topic/struct_nrf52/struct/nrf52.html.

[46] Chengzhi Su, Xin Xing, Xiang Cheng, Rui Guo, and Chao Luo. LPAH: Illustrating efficient live patching with alignment holes in kernel data. *IEEE Transactions on Computers*, 73(10):2434–2448, 2024.

[47] Alexey Tsymbal. The problem of concept drift: Definitions and related work. 05 2004.

[48] Hamed Valizadegan, Rong Jin, Ruofei Zhang, and Jianchang Mao. Learning to rank by optimizing ndcg measure. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 1883–1891. Curran Associates, Inc., 2009.

[49] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. FarmBeats: An IoT platform for Data-Driven agriculture. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 515–529, Boston, MA, March 2017. USENIX Association.

[50] Nicolas Villar, Daniel Cletheroe, Greg Saul, Christian Holz, Tim Regan, Oscar Salandin, Misha Sra, Hui-Shyong Yeo, William Field, and Haiyan Zhang. Project Zanzibar: A portable and flexible tangible interaction platform. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, New York, NY, USA, 2018. Association for Computing Machinery.

[51] Prasad Vyawahare. Delta over-the-air updates.

[52] Ning Xu, Sumit Rangwala, Krishna Kant Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and Deborah Estrin. A wireless sensor network for structural monitoring. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, page 13–24, New York, NY, USA, 2004. Association for Computing Machinery.