

Introduction to

Machine Learning Systems

Vijay
Janapa Reddi

Machine Learning Systems

Principles and Practices of Engineering Artificially Intelligent Systems

Prof. Vijay Janapa Reddi
School of Engineering and Applied Sciences
Harvard University

With heartfelt gratitude to the community for their invaluable contributions and steadfast support.

May 12, 2025

Table of contents

Preface	i
Global Outreach	i
Why We Wrote This Book	i
Want to Help Out?	ii
What's Next?	ii
Chapter 10 On-Device Learning	1
Purpose	1
10.1 Overview	2
10.2 On-Device Deployment Drivers	3
10.2.1 Why Learn On the Device	3
10.2.2 Application Domains	5
10.2.3 Centralized vs. Decentralized Training	6
10.3 Design Constraints and System Requirements	8
10.3.1 Model Constraints	9
10.3.2 Data Constraints	10
10.3.3 Compute Constraints	11
10.4 Adapting the Model	12
10.4.1 Weight Freezing and Bias-Only Updates	12
10.4.2 Residual Adaptation and Low-Rank Updates	14
10.4.2.1 Residual Adaptation via Adapters	15
10.4.2.2 Low-Rank Updates	15
10.4.2.3 Code Example: PyTorch Adapter Module	15
10.4.2.4 Use Case: Edge Personalization with Adapter Layers	16
10.4.2.5 Tradeoffs and Considerations	16
10.4.3 Task-Adaptive Sparse Updates	16
10.4.3.1 Sparse Update Formulation	17
10.4.3.2 Contribution-Based Layer Selection	17
10.4.3.3 Code Fragment: Selective Layer Updating (Py- Torch)	17
10.4.3.4 Use Case: Efficient Personalization with Tiny- Train	18
10.4.3.5 Tradeoffs and Considerations	18
10.4.3.6 Comparison of Model Adaptation Strategies	19
10.5 Working with Less Data	20
10.5.1 Few-Shot and Streaming Adaptation	20

10.5.2	Experience Replay and Memory-Based Learning	21
10.5.3	Compressed Data Representations	23
10.5.4	Tradeoffs and Comparative Summary	24
10.6	Distributed Local Learning: Federated Learning	25
10.6.1	Motivation	26
10.6.2	Federated Learning Protocols	27
10.6.2.1	Local Training	28
10.6.2.2	Federated Learning Protocols	28
10.6.2.3	Client Scheduling and System Participation . .	29
10.6.2.4	Communication-Efficient Updates	30
10.6.2.5	Personalization in Federated Learning	31
10.6.2.6	Privacy Considerations in Federated Learning	33
10.7	Designing Practical On-Device Learning Systems	33
10.8	Challenges and Limitations	35
10.8.0.1	System Heterogeneity	36
10.8.0.2	Data Fragmentation and Non-IID Distributions	37
10.8.0.3	Update Monitoring and Validation	37
10.8.0.4	Resource Competition and Cost	39
10.8.0.5	Deployment and Compliance Risks	40
10.8.0.6	Summary	41
10.9	Conclusion	42

REFERENCES	43
-------------------	-----------

References	45
-------------------	-----------

Preface

Welcome to Machine Learning Systems, your gateway to the fast-paced world of machine learning (ML) systems. This book is an extension of the [CS249r](#) course at Harvard University, taught by Prof. Vijay Janapa Reddi, and is the result of a collaborative effort involving students, professionals, and the broader community of AI practitioners.

We’ve created this open-source book to demystify the process of building efficient and scalable ML systems. Our goal is to provide a comprehensive guide that covers the principles, practices, and challenges of developing robust ML pipelines for deployment. This isn’t a static textbook—it’s a living, evolving resource designed to keep pace with advancements in the field.

“If you want to go fast, go alone. If you want to go far, go together.”
– African Proverb

As a living and breathing resource, this book is a continual work in progress, reflecting the ever-evolving nature of machine learning systems. Advancements in the ML landscape drive our commitment to keeping this resource updated with the latest insights, techniques, and best practices. We warmly invite you to join us on this journey by contributing your expertise, feedback, and ideas.

Global Outreach

Thank you to all our readers and visitors. Your engagement with the material keeps us motivated.

Why We Wrote This Book

While there are plenty of resources that focus on the algorithmic side of machine learning, resources on the systems side of things are few and far between. This gap inspired us to create this book—a resource dedicated to the principles and practices of building efficient and scalable ML systems.

Our vision for this book and its broader mission is deeply rooted in the transformative potential of AI and the need to make AI education globally accessible to all. To learn more about the inspiration behind this project and the values driving its creation, we encourage you to read the [Author’s Note](#).

Want to Help Out?

This is a collaborative project, and your input matters! If you'd like to contribute, check out our [contribution guidelines](#). Feedback, corrections, and new ideas are welcome—simply file a GitHub [issue](#).

What's Next?

If you're ready to dive deeper into the book's structure, learning objectives, and practical use, visit the About the Book section for more details.

#

AI Best Practices

Chapter 10

On-Device Learning



Figure 10.1: DALL-E 3 Prompt: Drawing of a smartphone with its internal components exposed, revealing diverse miniature engineers of different genders and skin tones actively working on the ML model. The engineers, including men, women, and non-binary individuals, are tuning parameters, repairing connections, and enhancing the network on the fly. Data flows into the ML model, being processed in real-time, and generating output inferences.

Purpose

How does enabling learning directly on edge devices reshape machine learning system design, and what strategies support adaptation under resource constraints?

The shift toward on-device learning marks a significant evolution in the deployment and maintenance of machine learning systems. Rather than relying exclusively on centralized infrastructure, models are now increasingly expected to adapt in situ—updating and improving directly on the devices where they operate. This approach introduces a new design space, where training must occur within stringent constraints on memory, compute, energy, and data availability. In these settings, the balance between model adaptability, system efficiency, and deployment scalability becomes critical. This chapter examines the architectural, algorithmic, and infrastructure-level techniques that enable effective learning on the edge, and outlines the principles required

to support autonomous model improvement in resource-constrained environments.

Learning Objectives

- Understand on-device learning and how it differs from cloud-based training
- Recognize the benefits and limitations of on-device learning
- Examine strategies to adapt models through complexity reduction, optimization, and data compression
- Understand related concepts like federated learning and transfer learning
- Analyze the security implications of on-device learning and mitigation strategies

10.1 Overview

Machine learning systems have traditionally treated model training and model inference as distinct phases, often separated by both time and infrastructure. Training occurs in the cloud, leveraging large-scale compute clusters and curated datasets, while inference is performed downstream on deployed models—typically on user devices or edge servers. However, this separation is beginning to erode. Increasingly, devices are being equipped not just to run inference, but to adapt, personalize, and improve models locally.

On-device learning refers to the process of training or adapting machine learning models directly on the device where they are deployed. This capability opens the door to systems that can personalize models in response to user behavior, operate without cloud connectivity, and respect stringent privacy constraints by keeping data local. It also introduces a new set of challenges: devices have limited memory, computational power, and energy. Furthermore, training data is often sparse, noisy, or non-independent across users. These limitations necessitate a fundamental rethinking of training algorithms, system architecture, and deployment strategies.

Definition of On-Device Learning

On-Device Learning is the *local adaptation or training* of machine learning models directly on deployed hardware devices, without reliance on continuous connectivity to centralized servers. It enables *personalization, privacy preservation, and autonomous operation* by leveraging user-specific data collected in situ. On-device learning systems must operate under *tight constraints on compute, memory, energy, and data availability*, requiring specialized methods for model optimization, training efficiency, and data representation. As on-device learning matures, it increasingly in-

corporates *federated collaboration, lifelong adaptation, and secure execution*, expanding the frontier of intelligent edge computing.

This chapter explores the principles and systems design considerations underpinning on-device learning. It begins by examining the motivating applications that necessitate learning on the device, followed by a discussion of the unique hardware constraints introduced by embedded and mobile environments. The chapter then develops a taxonomy of strategies for adapting models, algorithms, and data pipelines to these constraints. Particular emphasis is placed on distributed and collaborative methods, such as federated learning, which enable decentralized training without direct data sharing. The chapter concludes with an analysis of outstanding challenges, including issues related to reliability, system validation, and the heterogeneity of deployment environments.

10.2 On-Device Deployment Drivers

Machine learning systems have traditionally relied on centralized training pipelines, where models are developed and refined using large, curated datasets and powerful cloud-based infrastructure (Dean and Ghemawat 2008). Once trained, these models are deployed to client devices for inference. While this separation has served most use cases well, it imposes limitations in settings where local data is dynamic, private, or personalized. On-device learning challenges this model by enabling systems to train or adapt directly on the device, without relying on constant connectivity to the cloud.

10.2.1 Why Learn On the Device

Traditional machine learning systems rely on a clear division of labor between model training and inference. Training is performed in centralized environments with access to high-performance compute resources and large-scale datasets. Once trained, models are distributed to client devices, where they operate in a static inference-only mode. While this centralized paradigm has been effective in many deployments, it introduces limitations in settings where data is user-specific, behavior is dynamic, or connectivity is intermittent.

On-device learning refers to the capability of a deployed device to perform model adaptation using locally available data. This shift from centralized to decentralized learning is motivated by four key considerations: personalization, latency and availability, privacy, and infrastructure efficiency (Li et al. 2020).

Personalization is a primary motivation. Deployed models often encounter usage patterns and data distributions that differ substantially from their training environments. Local adaptation enables models to refine behavior in response to user-specific data—capturing linguistic preferences, physiological baselines, sensor characteristics, or environmental conditions. This is particularly important in applications with high inter-user variability, where a single global model may fail to serve all users equally well.

Latency and availability further justify local learning. In edge computing scenarios, connectivity to centralized infrastructure may be unreliable, delayed,

or intentionally limited to preserve bandwidth or reduce energy usage. On-device learning enables autonomous improvement of models even in fully offline or delay-sensitive contexts, where round-trip updates to the cloud are infeasible.

Privacy is another critical factor. Many applications involve sensitive or regulated data, including biometric measurements, typed input, location traces, or health information. Transmitting such data to the cloud introduces privacy risks and compliance burdens. Local learning mitigates these concerns by keeping raw data on the device and operating within privacy-preserving boundaries—potentially aiding adherence to regulations such as GDPR¹, HIPAA², or region-specific data sovereignty laws.

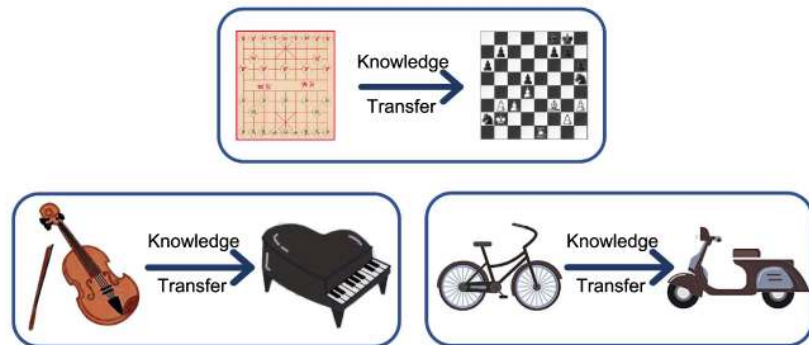
Infrastructure efficiency also plays a role. Centralized training pipelines require substantial backend infrastructure to collect, store, and process user data. At scale, this introduces bottlenecks in bandwidth, compute capacity, and energy consumption. By shifting learning to the edge, systems can reduce communication costs and distribute training workloads across the deployment fleet, relieving pressure on centralized resources.

These motivations are grounded in the broader concept of knowledge transfer, where a pretrained model transfers useful representations to a new task or domain. As depicted in Figure 10.2, knowledge transfer can occur between closely related tasks (e.g., playing different board games or musical instruments), or across domains that share structure (e.g., from riding a bicycle to driving a scooter). In the context of on-device learning, this means leveraging a model pretrained in the cloud and adapting it efficiently to a new context using only local data and limited updates. The figure highlights the key idea: pretrained knowledge enables fast adaptation without relearning from scratch, even when the new task diverges in input modality or goal.

¹ | GDPR: General Data Protection Regulation, a legal framework that sets guidelines for the collection and processing of personal information in the EU.

² | HIPAA: Health Insurance Portability and Accountability Act, U.S. legislation that provides data privacy and security provisions for safeguarding medical information.

Figure 10.2: Conceptual illustration of knowledge transfer across tasks and domains. The left side shows a pretrained model adapting to a new task, while the right side illustrates transfer across different domains.



This conceptual shift—enabled by transfer learning and adaptation—is essential for real-world on-device applications. Whether adapting a language model for personal typing preferences, adjusting gesture recognition to an individual’s movement patterns, or recalibrating a sensor model in a changing environment, on-device learning allows systems to remain responsive, efficient, and user-aligned over time.

10.2.2 Application Domains

The motivations for on-device learning are most clearly illustrated by examining the application domains where its benefits are both tangible and necessary. These domains span consumer technologies, healthcare, industrial systems, and embedded applications, each presenting scenarios where local adaptation is preferable—or required—for effective machine learning deployment.

Mobile input prediction is a mature example of on-device learning in action. In systems such as smartphone keyboards, predictive text and autocorrect features benefit substantially from continuous local adaptation. User typing patterns are highly personalized and evolve dynamically, making centralized static models insufficient. On-device learning enables language models to finetune their predictions directly on the device, without transmitting keystroke data to external servers. This approach not only supports personalization but also aligns with privacy-preserving design principles.

For instance, Google’s Gboard employs federated learning to improve shared models across a large population of users while keeping raw data local to each device (Hard et al. 2018). As shown in Figure 10.3, different prediction strategies illustrate how local adaptation can operate in real-time: next-word prediction (NWP) suggests likely continuations based on prior text, while Smart Compose leverages on-the-fly rescoring to offer dynamic completions, showcasing the sophistication of local inference mechanisms.

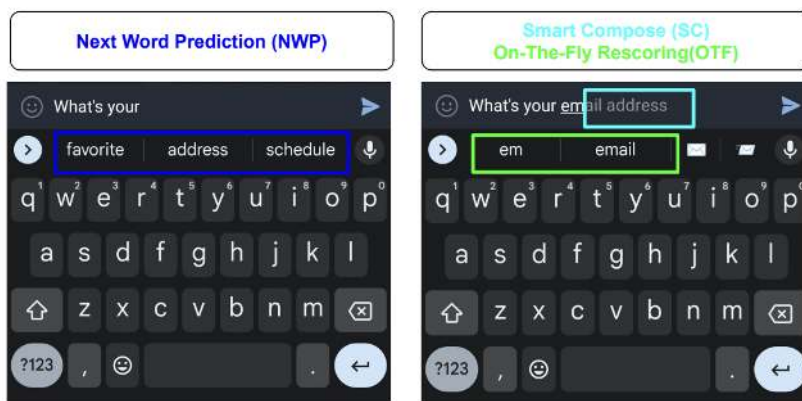


Figure 10.3: Illustration of two input prediction modes in Gboard. Left: Next Word Prediction (NWP). Right: Smart Compose (SC) with On-The-Fly Rescoring (OTF).

Wearable and health monitoring devices also present strong use cases. These systems often rely on real-time data from accelerometers, heart rate sensors, or electrodermal activity monitors. However, physiological baselines vary significantly between individuals. On-device learning allows models to adapt to these baselines over time, improving the accuracy of activity recognition, stress detection, and sleep staging. Moreover, in regulated healthcare environments, patient data must remain localized due to privacy laws, further reinforcing the need for edge-local adaptation.

Wake-word detection and voice interfaces illustrate another critical scenario. Devices such as smart speakers and earbuds must recognize voice commands

3 | Industrial Internet of Things (IIoT): Network of physical objects—devices, vehicles, buildings—that use sensors and software to collect and exchange data.

quickly and accurately, even in noisy or dynamic acoustic environments. Local training enables models to adapt to the user’s voice profile and ambient context, reducing false positives and missed detections. This kind of adaptation is particularly valuable in far-field audio settings, where microphone configurations and room acoustics vary widely across deployments.

Industrial IoT³ and remote monitoring systems also benefit from local learning capabilities. In applications such as agricultural sensing, pipeline monitoring, or environmental surveillance, connectivity to centralized infrastructure may be limited or costly. On-device learning allows these systems to detect anomalies, adjust thresholds, or adapt to seasonal trends without continuous communication with the cloud. This capability is critical for maintaining autonomy and reliability in edge-deployed sensor networks.

Embedded computer vision systems, including those in robotics, AR/VR, and smart cameras, present additional opportunities. These systems often operate in novel or evolving environments that differ significantly from training conditions. On-device adaptation allows models to recalibrate to new lighting conditions, object appearances, or motion patterns, maintaining task accuracy over time.

Each of these domains highlights a common pattern: the deployment environment introduces variation or uncertainty that cannot be fully anticipated during centralized training. On-device learning offers a mechanism for adapting models in place, enabling systems to improve continuously in response to local conditions. These examples also reveal a critical design requirement: learning must be performed efficiently, privately, and reliably under significant resource constraints. The following section formalizes these constraints and outlines the system-level considerations that shape the design of on-device learning solutions.

10.2.3 Centralized vs. Decentralized Training

Most machine learning systems today follow a centralized learning paradigm. Models are trained in data centers using large-scale, curated datasets aggregated from many sources. Once trained, these models are deployed to client devices in a static form, where they perform inference without further modification. Updates to model parameters—whether to incorporate new data or improve generalization—are handled periodically through offline retraining, often using newly collected or labeled data sent back from the field.

This centralized model of learning offers numerous advantages: high-performance computing infrastructure, access to diverse data distributions, and robust debugging and validation pipelines. However, it also depends on reliable data transfer, trust in data custodianship, and infrastructure capable of managing global updates across a fleet of devices. As machine learning is deployed into increasingly diverse and distributed environments, the limitations of this approach become more apparent.

In contrast, on-device learning is inherently decentralized. Each device maintains its own copy of a model and adapts it locally using data that is typically unavailable to centralized infrastructure. Training occurs on-device, often asynchronously and under varying resource conditions. Data never leaves the

device, reducing exposure but also complicating coordination. Devices may differ substantially in their hardware capabilities, runtime environments, and patterns of use, making the learning process heterogeneous and difficult to standardize.

This decentralized nature introduces unique systems challenges. Devices may operate with different versions of the model, leading to inconsistencies in behavior. Evaluation and validation become more complex, as there is no central point from which to measure performance (McMahan et al. 2017). Model updates must be carefully managed to prevent degradation, and safety guarantees become harder to enforce in the absence of centralized testing.

At the same time, decentralization introduces opportunities. It allows for personalization without centralized oversight, supports learning in disconnected or bandwidth-limited environments, and reduces the cost of infrastructure for model updates. It also raises important questions of how to coordinate learning across devices, whether through periodic synchronization, federated aggregation, or hybrid approaches that combine local and global objectives.

The move from centralized to decentralized learning represents more than a shift in deployment architecture—it fundamentally reshapes the design space for machine learning systems. In centralized training, data is aggregated from many sources and processed in large-scale data centers, where models are trained, validated, and then deployed in a static form to edge devices. In contrast, on-device learning introduces a decentralized paradigm: models are updated directly on client devices using local data, often asynchronously and under diverse hardware conditions. This change reduces reliance on cloud infrastructure and enhances personalization and privacy, but it also introduces new coordination and validation challenges.

On-device learning emerges as a response to the limitations of centralized machine learning workflows. As illustrated in Figure 10.4, the traditional paradigm (A) involves training a model on aggregated cloud-based data before pushing it to client devices for static inference. This architecture works well when centralized data collection is feasible, network connectivity is reliable, and model generalization across users is sufficient. However, it falls short in scenarios where data is highly personalized, privacy-sensitive, or collected in environments with limited connectivity.

In contrast, once the model is deployed, local differences begin to emerge. Region B depicts the process by which each device collects its own data stream—often non-IID⁴ and noisy—and adapts the model to better reflect its specific operating context. This marks the shift from global generalization to local specialization, highlighting the autonomy and variability introduced by decentralized learning.

Figure 10.4 illustrates this shift. In region A, centralized learning begins with cloud-based training on aggregated data, followed by deployment to client devices. Region B marks the transition to local learning: devices begin collecting data—often non-IID, noisy, and unlabeled—and adapting their models based on individual usage patterns. Finally, region C depicts federated learning, in which client updates are periodically synchronized via aggregated model updates rather than raw data transfer, enabling privacy-preserving global refinement.

⁴ | Non-IID Data: Datasets where samples are not independently and identically distributed, often seen in personalized data streams.

Figure 10.5 illustrates a pipeline that combines offline pre-training with online adaptive learning on resource-constrained IoT devices. The system first undergoes meta-training with generic data. During deployment, device-specific constraints such as data availability, compute, and memory shape the adaptation strategy by ranking and selecting layers and channels to update. This enables efficient on-device learning within limited resource envelopes.

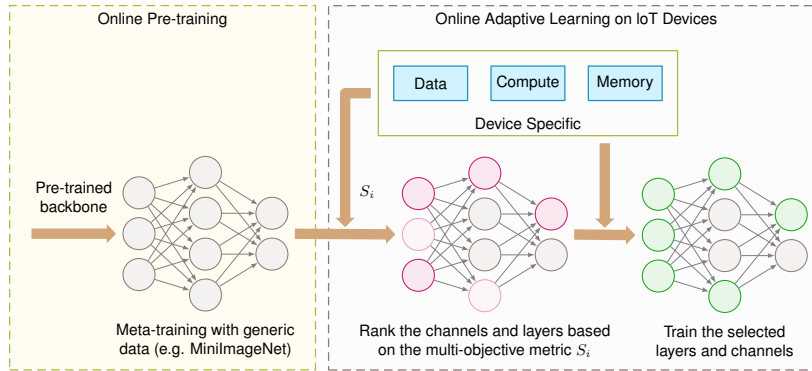


Figure 10.5: On-device adaptation framework.

10.3.1 Model Constraints

The structure and size of the machine learning model directly influence the feasibility of on-device training. Unlike cloud-deployed models that can span billions of parameters and rely on multi-gigabyte memory budgets, models intended for on-device learning must conform to tight constraints on memory, storage, and computational complexity. These constraints apply not only at inference time, but also during training, where additional resources are needed for gradient computation, parameter updates, and optimizer state.

For example, the MobileNetV2 architecture, commonly used in mobile vision tasks, requires approximately 14 MB of storage in its standard configuration. While this is feasible for modern smartphones, it far exceeds the memory available on embedded microcontrollers such as the Arduino Nano 33 BLE Sense, which provides only 256 KB of SRAM and 1 MB of flash storage. In such platforms, even a single layer of a typical convolutional neural network may exceed available RAM during training due to the need to store intermediate feature maps.

In addition to storage constraints, the training process itself expands the effective memory footprint. Standard backpropagation requires caching activations for each layer during the forward pass, which are then reused during gradient computation in the backward pass. For a 10-layer convolutional model processing 64×64 images, the required memory may exceed 1–2 MB—well beyond the SRAM capacity of most embedded systems.

Model complexity also affects runtime energy consumption and thermal limits. In systems such as smartwatches or battery-powered wearables, sustained model training can deplete energy reserves or trigger thermal throttling⁵. Training a full model using floating-point operations on these devices is often

⁵ | Reduction in computing performance to prevent overheating in electronic devices.

⁶ | MLPerf Tiny: A benchmark suite for evaluating the performance of ultra-low power machine learning systems in real-world scenarios.

infeasible. This limitation has motivated the development of ultra-lightweight model variants, such as MLPerf Tiny⁶ benchmark networks (Banbury et al. 2021), which fit within 100–200 KB and can be adapted using only partial gradient updates.

The model architecture itself must also be designed with on-device learning in mind. Many conventional architectures, such as transformers or large convolutional networks, are not well-suited for on-device adaptation due to their size and complexity. Instead, lightweight architectures such as MobileNets, SqueezeNet, and EfficientNet have been developed specifically for resource-constrained environments. These models use techniques such as depthwise separable convolutions, bottleneck layers, and quantization to reduce memory and compute requirements while maintaining performance.

These architectures are often designed to be modular, allowing for easy adaptation and fine-tuning. For example, MobileNets (Howard et al. 2017) can be configured with different width multipliers and resolution settings to balance performance and resource usage. This flexibility is critical for on-device learning, where the model must adapt to the specific constraints of the deployment environment.

10.3.2 Data Constraints

The nature of data available to on-device learning systems differs significantly from the large, curated, and centrally managed datasets typically used in cloud-based training. At the edge, data is locally collected, temporally sparse, and often unstructured or unlabeled. These characteristics introduce challenges in volume, quality, and statistical distribution, all of which affect the reliability and generalizability of learning on the device.

Data volume is typically limited due to storage constraints and the nature of user interaction. For example, a smart fitness tracker may collect motion data only during physical activity, generating relatively few labeled samples per day. If a user wears the device for just 30 minutes of exercise, only a few hundred data points might be available for training, compared to the thousands typically required for supervised learning in controlled environments.

Moreover, on-device data is frequently non-IID (non-independent and identically distributed) (Zhao et al. 2018). Consider a voice assistant deployed in different households: one user may issue commands in English with a strong regional accent, while another might speak a different language entirely. The local data distribution is highly user-specific and may differ substantially from the training distribution of the initial model. This heterogeneity complicates both model convergence and the design of update mechanisms that generalize well across devices.

Label scarcity presents an additional obstacle. Most edge-collected data is unlabeled by default. In a smartphone camera, for instance, the device may capture thousands of images, but only a few are associated with user actions (e.g., tagging or favoriting), which could serve as implicit labels. In many applications—such as detecting anomalies in sensor data or adapting gesture recognition models—labels may be entirely unavailable, making traditional supervised learning infeasible without additional methods.

Noise and variability further degrade data quality. Embedded systems such as environmental sensors or automotive ECUs⁷ may experience fluctuations in sensor calibration, environmental interference, or mechanical wear, leading to corrupted or drifting input signals over time. Without centralized validation, these errors may silently degrade learning performance if not detected and filtered appropriately.

Finally, data privacy and security concerns are paramount in many on-device learning applications. Sensitive information, such as health data or user interactions, must be protected from unauthorized access. This requirement often precludes the use of traditional data-sharing methods, such as uploading raw data to a central server for training. Instead, on-device learning must rely on techniques that allow for local adaptation without exposing sensitive information.

10.3.3 Compute Constraints

On-device learning must operate within the computational envelope of the target hardware platform, which ranges from low-power embedded microcontrollers to mobile-class processors found in smartphones and wearables. These systems differ substantially from the large-scale GPU or TPU infrastructure used in cloud-based training. They impose strict limits on instruction throughput, parallelism, and architectural support for training-specific operations, all of which shape the design of feasible learning strategies.

On the embedded end of the spectrum, devices such as the STM32F4 or ESP32 microcontrollers offer only a few hundred kilobytes of SRAM and lack hardware support for floating-point operations (Lai 2020). These constraints preclude the use of conventional deep learning libraries and require models to be carefully designed for integer arithmetic and minimal runtime memory allocation. In such cases, even small models require tailored techniques—such as quantization-aware training and selective parameter updates—to execute training loops without exceeding memory or power budgets. For example, the STM32F4 microcontroller can run a simple linear regression model with a few hundred parameters, but training even a small convolutional neural network would exceed its memory capacity. In these environments, training is often limited to simple algorithms such as stochastic gradient descent (SGD) or k-means clustering, which can be implemented using integer arithmetic and minimal memory overhead.

In contrast, mobile-class hardware—such as the Qualcomm Snapdragon, Apple Neural Engine, or Google Tensor SoC—provides significantly more compute power, often with dedicated AI accelerators and optimized support for 8-bit or mixed-precision matrix operations. These platforms can support more complex training routines, including full backpropagation over compact models, though they still fall short of the computational throughput and memory bandwidth available in centralized data centers. For instance, training a lightweight transformer on a smartphone is feasible but must be tightly bounded in both time and energy consumption to avoid degrading the user experience.

Compute constraints are especially salient in real-time or battery-operated systems. In a smartphone-based speech recognizer, on-device adaptation must

⁷ | Electronic Control Unit (ECU):
A device that controls one or more
of the electrical systems or subsys-
tems in a vehicle.

not interfere with inference latency or system responsiveness. Similarly, in wearable medical monitors, training must occur opportunistically, during periods of low activity or charging, to preserve battery life and avoid thermal issues.

10.4 Adapting the Model

Adapting a machine learning model on the device requires revisiting a core assumption of conventional training: that the entire model must be updated. In resource-constrained environments, this assumption becomes infeasible due to memory, compute, and energy limitations. Instead, modern approaches to on-device learning often focus on minimizing the scope of adaptation, updating only a subset of model parameters while reusing the majority of the pretrained architecture. These approaches leverage the power of transfer learning, starting with a model pretrained (usually offline on large datasets) and efficiently specializing it using the limited local data and compute resources available at the edge. This strategy is particularly effective when the pretrained model has already learned useful representations that can be adapted to new tasks or domains. By freezing most of the model parameters and only updating a small subset, we can achieve significant reductions in memory and compute requirements while still allowing for meaningful adaptation.

This strategy reduces both computational overhead and memory usage during training, enabling efficient local updates on devices ranging from smartphones to embedded microcontrollers. The central idea is to retain most of the model as a frozen backbone, while introducing lightweight, adaptable components—such as bias-only updates, residual adapters, or task-specific layers—that can capture local variations in data. These techniques enable personalized or environment-aware learning without incurring the full cost of end-to-end finetuning.

In the sections that follow, we examine how minimal adaptation strategies are designed, the tradeoffs they introduce, and their role in enabling practical on-device learning.

10.4.1 Weight Freezing and Bias-Only Updates

One of the simplest and most effective strategies for reducing the cost of on-device learning is to freeze the majority of a model's parameters and adapt only a minimal subset. A widely used approach is bias-only adaptation, in which all weights are fixed and only the bias terms—typically scalar offsets applied after linear or convolutional layers—are updated during training. This significantly reduces the number of trainable parameters, simplifies memory management during backpropagation, and helps mitigate overfitting when data is sparse or noisy.

Consider a standard neural network layer:

$$y = Wx + b$$

where $W \in \mathbb{R}^{m \times n}$ is the weight matrix, $b \in \mathbb{R}^m$ is the bias vector, and $x \in \mathbb{R}^n$ is the input. In full training, gradients are computed for both W and b . In

bias-only adaptation, we constrain:

$$\frac{\partial \mathcal{L}}{\partial W} = 0, \quad \frac{\partial \mathcal{L}}{\partial b} \neq 0$$

so that only the bias is updated via gradient descent:

$$b \leftarrow b - \eta \frac{\partial \mathcal{L}}{\partial b}$$

This drastically reduces the number of stored gradients and optimizer states, enabling training to proceed even under memory-constrained conditions. On embedded devices that lack floating-point units, this reduction can be critical to enabling on-device learning at all.

The following code snippet demonstrates how to implement bias-only adaptation in PyTorch:

```
# Freeze all parameters
for name, param in model.named_parameters():
    param.requires_grad = False

# Enable gradients for bias parameters only
for name, param in model.named_parameters():
    if 'bias' in name:
        param.requires_grad = True
```

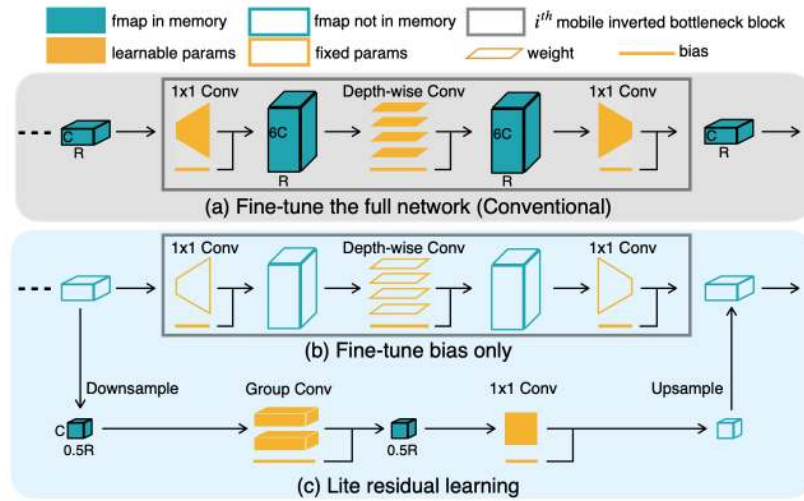
This pattern ensures that only bias terms participate in the backward pass and optimizer update. It is particularly useful when adapting pretrained models to user-specific or device-local data.

This technique underpins TinyTL, a framework explicitly designed to enable efficient adaptation of deep neural networks on microcontrollers and other memory-limited platforms. Rather than updating all network parameters during training, TinyTL freezes both the convolutional weights and the batch normalization statistics, training only the bias terms and, in some cases, lightweight residual components. This architectural shift drastically reduces memory usage during backpropagation, since the largest tensors—intermediate activations—no longer need to be stored for gradient computation.

Figure 10.6 illustrates the architectural differences between a standard model and the TinyTL approach. In the conventional baseline architecture, all layers are trainable, and backpropagation requires storing intermediate activations for the full network. This significantly increases the memory footprint, which quickly becomes infeasible on edge devices with only a few hundred kilobytes of SRAM.

In contrast, the TinyTL architecture freezes all weights and updates only the bias terms inserted after convolutional layers. These bias modules are lightweight and require minimal memory, enabling efficient training with a drastically reduced memory footprint. The frozen convolutional layers act as a fixed feature extractor, and only the trainable bias components are involved in adaptation. By avoiding storage of full activation maps and limiting the

Figure 10.6: TinyTL freezes weights and batch norm statistics, adapting only the biases and lightweight components to enable memory-efficient on-device training.



number of updated parameters, TinyTL enables on-device training under severe resource constraints.

Because the base model remains unchanged, TinyTL assumes that the pre-trained features are sufficiently expressive for downstream tasks. The bias terms allow for minor but meaningful shifts in model behavior, particularly for personalization tasks. When domain shift is more significant, TinyTL can optionally incorporate small residual adapters to improve expressivity, all while preserving the system's tight memory and energy profile.

These design choices allow TinyTL to reduce training memory usage by more than 10 \times . For instance, adapting a MobileNetV2 model using TinyTL can reduce the number of updated parameters from over 3 million to fewer than 50,000. Combined with quantization, this enables local adaptation on devices with only a few hundred kilobytes of memory—making on-device learning truly feasible in constrained environments.

10.4.2 Residual Adaptation and Low-Rank Updates

Bias-only updates offer a lightweight path for on-device learning, but they are limited in representational flexibility. When the frozen model does not align well with the target distribution, it may be necessary to allow more expressive adaptation—without incurring the full cost of weight updates. One solution is to introduce residual adaptation modules (Houlsby et al. 2019),⁸ or low-rank parameterizations⁹, which provide a middle ground between static backbones¹⁰ and full fine-tuning (Hu et al. 2021).

These methods extend a frozen model by adding trainable layers—typically small and computationally cheap—that allow the network to respond to new data. The main body of the network remains fixed, while only the added components are optimized. This modularity makes the approach well-suited for on-device adaptation in constrained settings, where small updates must deliver meaningful changes.

⁸ Residual Adaptation Modules: Layers added to existing networks to improve adaptability without extensive retraining.

⁹ Low-rank Parameterizations: Techniques that decompose parameters into low-rank matrices to save computation.

¹⁰ Static Backbones: Unchangeable core parts of a neural network model, typically pre-trained.

10.4.2.1 Residual Adaptation via Adapters

A common implementation involves inserting adapters—small residual bottleneck layers—between existing layers in a pretrained model. Consider a hidden representation h passed between layers. A residual adapter introduces a transformation:

$$h' = h + A(h)$$

where $A(\cdot)$ is a trainable function, typically composed of two linear layers with a nonlinearity:

$$A(h) = W_2 \sigma(W_1 h)$$

with $W_1 \in \mathbb{R}^{r \times d}$ and $W_2 \in \mathbb{R}^{d \times r}$, where $r \ll d$. This bottleneck design ensures that only a small number of parameters are introduced per layer.

The adapters act as learnable perturbations on top of a frozen backbone. Because they are small and sparsely applied, they add negligible memory overhead, yet they allow the model to shift its predictions in response to new inputs.

10.4.2.2 Low-Rank Updates

Another efficient strategy is to constrain weight updates themselves to a low-rank structure. Rather than updating a full matrix W , we approximate the update as:

$$\Delta W \approx UV^\top$$

where $U \in \mathbb{R}^{m \times r}$ and $V \in \mathbb{R}^{n \times r}$, with $r \ll \min(m, n)$. This reduces the number of trainable parameters from mn to $r(m + n)$. During adaptation, the new weight is computed as:

$$W_{\text{adapted}} = W_{\text{frozen}} + UV^\top$$

This formulation is commonly used in LoRA (Low-Rank Adaptation) techniques, originally developed for transformer models (Hu et al. 2021) but broadly applicable across architectures. Low-rank updates can be implemented efficiently on edge devices, particularly when U and V are small and fixed-point representations are supported.

10.4.2.3 Code Example: PyTorch Adapter Module

```
class Adapter(nn.Module):
    def __init__(self, dim, bottleneck_dim):
        super().__init__()
        self.down = nn.Linear(dim, bottleneck_dim)
        self.up = nn.Linear(bottleneck_dim, dim)
        self.activation = nn.ReLU()

    def forward(self, x):
        return x + self.up(self.activation(self.down(x)))
```

This adapter adds a small residual transformation to a frozen layer. When inserted into a larger model, only the adapter parameters are trained.

10.4.2.4 Use Case: Edge Personalization with Adapter Layers

Adapters are especially useful when a global model is deployed to many devices and must adapt to device-specific input distributions. For instance, in smartphone camera pipelines, environmental lighting, user preferences, or lens distortion may vary between users (Rebuffi, Bilen, and Vedaldi 2017). A shared model can be frozen and fine-tuned per-device using a few residual modules, allowing lightweight personalization without risking catastrophic forgetting¹¹. In voice-based systems, adapter modules have been shown to reduce word error rates in personalized speech recognition without retraining the full acoustic model. They also allow easy rollback or switching between user-specific versions.

¹¹ | Catastrophic Forgetting: A phenomenon where a neural network forgets previously learned information upon learning new data.

10.4.2.5 Tradeoffs and Considerations

Residual and low-rank updates strike a balance between expressivity and efficiency. Compared to bias-only learning, they can model more substantial deviations from the pretrained task. However, they require more memory and compute—both for training and inference.

When considering residual and low-rank updates for on-device learning, several important tradeoffs emerge. First, these methods consistently demonstrate superior adaptation quality compared to bias-only approaches, particularly when deployed in scenarios involving significant distribution shifts¹² from the original training data (Quiñero-Candela et al. 2008). This improved adaptability stems from their increased parameter capacity and ability to learn more complex transformations.

However, this enhanced adaptability comes at a cost. The introduction of additional layers or parameters inevitably increases both memory requirements and computational latency during forward and backward passes. While these increases are modest compared to full model training, they must be carefully considered when deploying to resource-constrained devices.

Additionally, implementing these adaptation techniques requires system-level support for dynamic computation graphs¹³ and the ability to selectively inject trainable parameters. Not all deployment environments or inference engines may support such capabilities out of the box.

Despite these considerations, residual adaptation techniques have proven particularly valuable in mobile and edge computing scenarios where devices have sufficient computational resources. For instance, modern smartphones and tablets can readily accommodate these adaptations while maintaining acceptable performance characteristics. This makes residual adaptation a practical choice for applications requiring personalization without the overhead of full model retraining.

¹² | Distribution shifts refer to changes in the input data's characteristics, which can affect model performance when different from the training data.

¹³ | Dynamic Computation Graphs: Structures that allow changes during runtime, enabling models to adapt structures based on input data.

10.4.3 Task-Adaptive Sparse Updates

Even when adaptation is restricted to a small number of parameters—such as biases or adapter modules—training remains resource-intensive on constrained devices. One promising approach is to selectively update only a task-relevant

subset of model parameters, rather than modifying the entire network or introducing new modules. This approach is known as task-adaptive sparse updating (Zhang, Song, and Tao 2020).

The key insight is that not all layers of a deep model contribute equally to performance gains on a new task or dataset. If we can identify a *minimal subset of parameters* that are most impactful for adaptation, we can train only those, reducing memory and compute costs while still achieving meaningful personalization.

10.4.3.1 Sparse Update Formulation

Let a neural network be defined by parameters $\theta = \{\theta_1, \theta_2, \dots, \theta_L\}$ across L layers. In standard fine-tuning, we compute gradients and perform updates on all parameters:

$$\theta_i \leftarrow \theta_i - \eta \frac{\partial \mathcal{L}}{\partial \theta_i}, \quad \text{for } i = 1, \dots, L$$

In task-adaptive sparse updates, we select a small subset $\mathcal{S} \subset \{1, \dots, L\}$ such that only parameters in \mathcal{S} are updated:

$$\theta_i \leftarrow \begin{cases} \theta_i - \eta \frac{\partial \mathcal{L}}{\partial \theta_i}, & \text{if } i \in \mathcal{S} \\ \theta_i, & \text{otherwise} \end{cases}$$

The challenge lies in selecting the optimal subset \mathcal{S} given memory and compute constraints.

10.4.3.2 Contribution-Based Layer Selection

A principled strategy for selecting \mathcal{S} is to use contribution analysis—an empirical method that estimates how much each layer contributes to downstream performance improvement. For example, one can measure the marginal gain from updating each layer independently:

1. Freeze the entire model.
2. Unfreeze one candidate layer.
3. Finetune briefly and evaluate improvement in validation accuracy.
4. Rank layers by performance gain per unit cost (e.g., per KB of trainable memory).

This layer-wise profiling yields a ranking from which \mathcal{S} can be constructed subject to a memory budget.

A concrete example is TinyTrain, a method designed to enable rapid adaptation on-device (C. Deng, Zhang, and Wu 2022). TinyTrain pretrains a model along with meta-gradients that capture which layers are most sensitive to new tasks. At runtime, the system dynamically selects layers to update based on task characteristics and available resources.

10.4.3.3 Code Fragment: Selective Layer Updating (PyTorch)

```
## Assume model has named layers: ['conv1', 'conv2', 'fc']
## We selectively update only conv2 and fc

for name, param in model.named_parameters():
    if 'conv2' in name or 'fc' in name:
        param.requires_grad = True
    else:
        param.requires_grad = False
```

This pattern can be extended with profiling logic to select layers based on contribution scores or hardware profiles.

10.4.3.4 Use Case: Efficient Personalization with TinyTrain

Consider a scenario where a user wears an augmented reality headset that performs real-time object recognition. As lighting and environments shift, the system must adapt to maintain accuracy—but training must occur during brief idle periods or while charging.

TinyTrain enables this by using meta-training during offline preparation: the model learns not only to perform the task, but also which parameters are most important to adapt. Then, at deployment, the device performs task-adaptive sparse updates, modifying only a few layers that are most relevant for its current environment. This keeps adaptation fast, energy-efficient, and memory-aware.

10.4.3.5 Tradeoffs and Considerations

Task-adaptive sparse updates offer a highly efficient alternative to full model finetuning, but they require careful design (S. Wang et al. 2020). Task-adaptive sparse updates introduce several important system-level considerations that must be carefully balanced. First, the overhead of contribution analysis—while primarily incurred during pretraining or initial profiling—represents a non-trivial computational cost. This overhead is typically acceptable since it occurs offline, but it must be factored into the overall system design and deployment pipeline.

Second, the stability of the adaptation process becomes critical when working with sparse updates. If too few parameters are selected for updating, the model may underfit the target distribution, failing to capture important local variations. This suggests the need for careful validation of the selected parameter subset before deployment, potentially incorporating minimum thresholds for adaptation capacity.

Third, the selection of updateable parameters must account for hardware-specific characteristics of the target platform. Beyond just considering gradient magnitudes, the system must evaluate the actual execution cost of updating specific layers on the deployed hardware. Some parameters might show high contribution scores but prove expensive to update on certain architectures, requiring a more nuanced selection strategy that balances statistical utility with runtime efficiency.

Despite these tradeoffs, task-adaptive sparse updates provide a powerful mechanism to scale adaptation to diverse deployment contexts, from microcontrollers to mobile devices (Levy et al. 2023).

10.4.3.6 Comparison of Model Adaptation Strategies

Each adaptation strategy for on-device learning offers a distinct balance between expressivity, resource efficiency, and implementation complexity. Understanding these tradeoffs is essential when designing systems for diverse deployment targets—from ultra-low-power microcontrollers to feature-rich mobile processors.

Bias-only adaptation is the most lightweight approach, updating only scalar offsets in each layer while freezing all other parameters. This significantly reduces memory requirements and computational burden, making it suitable for devices with tight memory and energy budgets. However, its limited expressivity means it is best suited to applications where the pretrained model already captures most of the relevant task features and only minor local calibration is required.

Residual adaptation, often implemented via adapter modules, introduces a small number of trainable parameters into the frozen backbone of a neural network. This allows for greater flexibility than bias-only updates, while still maintaining control over the adaptation cost. Because the backbone remains fixed, training can be performed efficiently and safely under constrained conditions. This method supports modular personalization across tasks and users, making it a favorable choice for mobile settings where moderate adaptation capacity is needed.

Task-adaptive sparse updates offer the greatest potential for task-specific finetuning by selectively updating only a subset of layers or parameters based on their contribution to downstream performance. While this method enables expressive local adaptation, it requires a mechanism for layer selection—either through profiling, contribution analysis, or meta-training—which introduces additional complexity. Nonetheless, when deployed carefully, it allows for dynamic tradeoffs between accuracy and efficiency, particularly in systems that experience large domain shifts or evolving input conditions.

These three approaches form a spectrum of tradeoffs. Their relative suitability depends on application domain, available hardware, latency constraints, and expected distribution shift. Table 10.1 summarizes their characteristics:

Table 10.1: Comparison of model adaptation strategies.

Technique	Trainable Parameters	Memory Overhead	Expressivity	Use Case Suitability	System Requirements
Bias-Only Updates	Bias terms only	Minimal	Low	Simple personalization; low variance	Extreme memory/compute limits
Residual Adapters	Adapter modules	Moderate	Moderate to High	User-specific tuning on mobile	Mobile-class SoCs with runtime support
Sparse Layer Updates	Selective parameter subsets	Variable	High (task-adaptive)	Real-time adaptation; domain shift	Requires profiling or meta-training

10.5 Working with Less Data

On-device learning systems operate in environments where data is scarce, noisy, and highly individualized. Unlike centralized machine learning pipelines that rely on large, curated datasets, edge devices typically observe only small volumes of task-relevant data—collected incrementally over time and rarely labeled in a supervised manner (Chen et al. 2019). This constraint fundamentally reshapes the learning process. Algorithms must extract value from minimal supervision, generalize from sparse observations, and remain robust to distributional shift. In many cases, the available data may be insufficient to train a model from scratch or even to finetune all parameters of a pretrained network. Instead, practical on-device learning relies on data-efficient techniques: few-shot adaptation, streaming updates, memory-based replay, and compressed supervision. These approaches enable models to improve over time without requiring extensive labeled datasets or centralized aggregation, making them well-suited to mobile, wearable, and embedded platforms where data acquisition is constrained by power, storage, and privacy considerations.

10.5.1 Few-Shot and Streaming Adaptation

In conventional machine learning workflows, effective training typically requires large labeled datasets, carefully curated and preprocessed to ensure sufficient diversity and balance. On-device learning, by contrast, must often proceed from only a handful of local examples—collected passively through user interaction or ambient sensing, and rarely labeled in a supervised fashion. These constraints motivate two complementary adaptation strategies: few-shot learning, in which models generalize from a small, static set of examples, and streaming adaptation, where updates occur continuously as data arrives.

Few-shot adaptation is particularly relevant when the device observes a small number of labeled or weakly labeled instances for a new task or user condition (Y. Wang et al. 2020). In such settings, it is often infeasible to perform full finetuning of all model parameters without overfitting. Instead, methods such as bias-only updates, adapter modules, or prototype-based classification are employed to make use of limited data while minimizing capacity for memorization. Let $D = \{(x_i, y_i)\}_{i=1}^K$ denote a K -shot dataset of labeled examples collected on-device. The goal is to update the model parameters θ to improve task performance under constraints such as:

- Limited number of gradient steps: $T \ll 100$
- Constrained memory footprint: $\|\theta_{\text{updated}}\| \ll \|\theta\|$
- Preservation of prior task knowledge (to avoid catastrophic forgetting)

Keyword spotting (KWS) systems offer a concrete example of few-shot adaptation in a real-world, on-device deployment (Warden 2018). These models are used to detect fixed phrases—such as “Hey Siri” or “OK Google”—with low latency and high reliability. A typical KWS model consists of a pretrained acoustic encoder (e.g., a small convolutional or recurrent network that transforms input audio into an embedding space) followed by a lightweight classifier. In commercial systems, the encoder is trained centrally using thousands of hours of labeled speech across multiple languages and speakers. However,

supporting custom wake words (e.g., “Hey Jarvis”) or adapting to underrepresented accents and dialects is often infeasible via centralized training due to data scarcity and privacy concerns.

Few-shot adaptation solves this problem by finetuning only the output classifier or a small subset of parameters—such as bias terms—using just a few example utterances collected directly on the device. For example, a user might provide 5–10 recordings of their custom wake word. These samples are then used to update the model locally, while the main encoder remains frozen to preserve generalization and reduce memory overhead. This enables personalization without requiring additional labeled data or transmitting private audio to the cloud.

Such an approach is not only computationally efficient, but also aligned with privacy-preserving design principles. Because only the output layer is updated—and often with a simple gradient step or prototype computation—the total memory footprint and runtime compute are compatible with mobile-class devices or even microcontrollers. This makes KWS a canonical case study for few-shot learning at the edge, where the system must operate under tight constraints while delivering user-specific performance.

Beyond static few-shot learning, many on-device scenarios benefit from streaming adaptation, where models must learn incrementally as new data arrives (Hayes et al. 2020). Streaming adaptation generalizes this idea to continuous, asynchronous settings where data arrives incrementally over time. Let $\{x_t\}_{t=1}^{\infty}$ represent a stream of observations. In streaming settings, the model must update itself after observing each new input, typically without access to prior data, and under bounded memory and compute. The model update can be written generically as:

$$\theta_{t+1} = \theta_t - \eta_t \nabla \mathcal{L}(x_t; \theta_t)$$

where η_t is the learning rate at time t . This form of adaptation is sensitive to noise and drift in the input distribution, and thus often incorporates mechanisms such as learning rate decay, meta-learned initialization, or update gating to improve stability.

Aside from KWS, practical examples of these strategies abound. In wearable health devices, a model that classifies physical activities may begin with a generic classifier and adapt to user-specific motion patterns using only a few labeled activity segments. In smart assistants, user voice profiles are finetuned over time using ongoing speech input, even when explicit supervision is unavailable. In such cases, local feedback—such as correction, repetition, or downstream task success—can serve as implicit signals to guide learning.

Few-shot and streaming adaptation highlight the shift from traditional training pipelines to data-efficient, real-time learning under uncertainty. They form a foundation for more advanced memory and replay strategies, which we turn to next.

10.5.2 Experience Replay and Memory-Based Learning

On-device learning systems face a fundamental tension between continuous adaptation and limited data availability. One common approach to alleviating this tension is experience replay—a memory-based strategy that enables

models to retrain on past examples. Originally developed in the context of reinforcement learning and continual learning, replay buffers help prevent catastrophic forgetting and stabilize training in non-stationary environments.

Unlike server-side replay strategies that rely on large datasets and extensive compute, on-device replay must operate with extremely limited capacity—often tens or hundreds of samples—and must avoid interfering with user experience (Rolnick et al. 2019). Buffers may store only compressed features or distilled summaries, and updates must occur opportunistically (e.g., during idle cycles or charging). These system-level constraints reshape how replay is implemented and evaluated in the context of embedded ML.

Let \mathcal{M} represent a memory buffer that retains a fixed-size subset of training examples. At time step t , the model receives a new data point (x_t, y_t) and appends it to \mathcal{M} . A replay-based update then samples a batch $\{(x_i, y_i)\}_{i=1}^k$ from \mathcal{M} and applies a gradient step:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} \left[\frac{1}{k} \sum_{i=1}^k \mathcal{L}(x_i, y_i; \theta_t) \right]$$

where θ_t are the model parameters, η is the learning rate, and \mathcal{L} is the loss function. Over time, this replay mechanism allows the model to reinforce prior knowledge while incorporating new information.

A practical on-device implementation might use a ring buffer¹⁴ to store a small set of compressed feature vectors rather than full input examples. The following pseudocode illustrates a minimal replay buffer designed for constrained environments:

¹⁴ Ring Buffer: A circular buffer that efficiently manages data by overwriting old entries with new ones as space requires.

```
# On-device replay buffer using compressed feature vectors
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []
        self.index = 0

    def store(self, feature_vec, label):
        if len(self.buffer) < self.capacity:
            self.buffer.append((feature_vec, label))
        else:
            self.buffer[self.index] = (feature_vec, label)
            self.index = (self.index + 1) % self.capacity

    def sample(self, k):
        return random.sample(
            self.buffer,
            min(k, len(self.buffer))
        )
```

This implementation maintains a fixed-capacity cyclic buffer, storing compressed representations (e.g., last-layer embeddings) and associated labels.

Such buffers are useful for replaying adaptation updates without violating memory or energy budgets.

In TinyML applications, experience replay has been applied to problems such as gesture recognition, where devices must continuously improve predictions while observing a small number of events per day. Instead of training directly on the streaming data, the device stores representative feature vectors from recent gestures and uses them to finetune classification boundaries periodically. Similarly, in on-device keyword spotting, replaying past utterances can improve wake-word detection accuracy without the need to transmit audio data off-device.

While experience replay improves stability in data-sparse or non-stationary environments, it introduces several tradeoffs. Storing raw inputs may breach privacy constraints or exceed storage budgets, especially in vision and audio applications. Replay from feature vectors reduces memory usage but may limit the richness of gradients for upstream layers. Write cycles to persistent flash memory—often required for long-term storage on embedded devices—can also raise wear-leveling concerns¹⁵. These constraints require careful co-design of memory usage policies, replay frequency, and feature selection strategies, particularly in continuous deployment scenarios.

10.5.3 Compressed Data Representations

In many on-device learning scenarios, the raw training data may be too large, noisy, or redundant to store and process effectively. This motivates the use of compressed data representations, where the original inputs are transformed into lower-dimensional embeddings or compact encodings that preserve salient information while minimizing memory and compute costs.

Compressed representations serve two complementary goals. First, they reduce the footprint of stored data, allowing devices to maintain longer histories or replay buffers under tight memory budgets (Sanh et al. 2019). Second, they simplify the learning task by projecting raw inputs into more structured feature spaces, often learned via pretraining or meta-learning, in which efficient adaptation is possible with minimal supervision.

One common approach is to encode data points using a pretrained feature extractor and discard the original high-dimensional input. For example, an image x_i might be passed through a convolutional neural network (CNN) to produce an embedding vector $z_i = f(x_i)$, where $f(\cdot)$ is a fixed feature encoder. This embedding captures visual structure (e.g., shape, texture, or spatial layout) in a compact representation—typically 64 to 512 dimensions—suitable for lightweight downstream adaptation.

Mathematically, training can proceed over compressed samples (z_i, y_i) using a lightweight decoder or projection head. Let θ represent the trainable parameters of this decoder model, which is typically a small neural network that maps from compressed representations to output predictions. As each example is presented, the model parameters are updated using gradient descent:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} \mathcal{L}(g(z_i; \theta), y_i)$$

Here:

¹⁵ | Wear leveling is a technique used in flash memory management to distribute data writes evenly across the memory, prolonging lifespan.

- z_i is the compressed representation of the i -th input,
- y_i is the corresponding label or supervision signal,
- $g(z_i; \theta)$ is the decoder's prediction,
- \mathcal{L} is the loss function measuring prediction error,
- η is the learning rate, and
- ∇_{θ} denotes the gradient with respect to the parameters θ .

This formulation highlights how only a compact decoder model—with parameter set θ —needs to be trained, making the learning process feasible even when memory and compute are limited.

Advanced approaches go beyond fixed encoders by learning discrete or sparse dictionaries that represent data using low-rank or sparse coefficient matrices. For instance, a dataset of sensor traces can be factorized as $X \approx DC$, where D is a dictionary of basis patterns and C is a block-sparse coefficient matrix indicating which patterns are active in each example. By updating only a small number of dictionary atoms or coefficients, the model can adapt with minimal overhead.

Compressed representations are particularly useful in privacy-sensitive settings, as they allow raw data to be discarded or obfuscated after encoding. Furthermore, compression acts as an implicit regularizer, smoothing the learning process and mitigating overfitting when only a few training examples are available.

In practice, these strategies have been applied in domains such as keyword spotting, where raw audio signals are first transformed into Mel-frequency cepstral coefficients (MFCCs)—a compact, lossy representation of the power spectrum of speech. These MFCC vectors serve as compressed inputs for downstream models, enabling local adaptation using only a few kilobytes of memory. Instead of storing raw audio waveforms, which are large and computationally expensive to process, devices store and learn from these compressed feature vectors directly. Similarly, in low-power computer vision systems, embeddings extracted from lightweight CNNs are retained and reused for few-shot learning. These examples illustrate how representation learning and compression serve as foundational tools for scaling on-device learning to memory- and bandwidth-constrained environments.

10.5.4 Tradeoffs and Comparative Summary

Each of the techniques introduced in this section, few-shot learning, experience replay, and compressed data representations, offers a strategy for adapting models on-device when data is scarce or streaming. However, they operate under different assumptions and constraints, and their effectiveness depends on system-level factors such as memory capacity, data availability, task structure, and privacy requirements.

Few-shot adaptation excels when a small but informative set of labeled examples is available, especially when personalization or rapid task-specific tuning is required. It minimizes compute and data needs, but its effectiveness hinges on the quality of pretrained representations and the alignment between the initial model and the local task.

Experience replay addresses continual adaptation by mitigating forgetting and improving stability, especially in non-stationary environments. It enables reuse of past data, but requires memory to store examples and compute cycles for periodic updates. Replay buffers may also raise privacy or longevity concerns, especially on devices with limited storage or flash write cycles.

Compressed data representations reduce the footprint of learning by transforming raw data into compact feature spaces. This approach supports longer retention of experience and efficient finetuning, particularly when only light-weight heads are trainable. However, compression can introduce information loss, and fixed encoders may fail to capture task-relevant variability if they are not well-aligned with deployment conditions. Table 10.2 summarizes key tradeoffs:

Table 10.2: Summary of on-device learning techniques.


Technique	Data Requirements	Memory/Compute Overhead	Use Case Fit
Few-Shot Adaptation	Small labeled set (K-shots)	Low	Personalization, quick on-device finetuning
Experience Replay	Streaming data	Moderate (buffer & update)	Non-stationary data, stability under drift
Compressed Representations	Unlabeled or encoded data	Low to Moderate	Memory-limited devices, privacy-sensitive contexts

In practice, these methods are not mutually exclusive. Many real-world systems combine them to achieve robust, efficient adaptation. For example, a keyword spotting system may use compressed audio features (e.g., MFCCs), finetune a few parameters from a small support set, and maintain a replay buffer of past embeddings for continual refinement.

Together, these strategies embody the core challenge of on-device learning: achieving reliable model improvement under persistent constraints on data, compute, and memory.

10.6 Distributed Local Learning: Federated Learning

On-device learning enables models to adapt locally using data generated on the device, but doing so in isolation limits a system’s ability to generalize across users and tasks. In many applications, learning must occur not just within a single device, but across a fleet of heterogeneous, intermittently connected systems. This calls for a distributed coordination framework that supports collective model improvement without violating the constraints of privacy, limited connectivity, and device autonomy. Federated learning (FL) is one such framework.



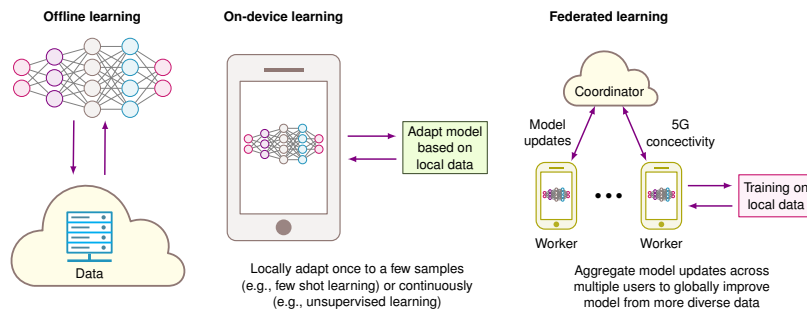
Definition of Federated Learning

Federated Learning is a *decentralized machine learning approach* in which training occurs across a population of distributed devices, each using

its *private, locally collected data*. Rather than transmitting raw data to a central server, devices share only *model updates*—such as gradients or weight changes—which are then aggregated to improve a shared global model. This approach *preserves data privacy* while enabling *collective intelligence across diverse environments*. As federated learning matures, it integrates *privacy-enhancing technologies, communication-efficient protocols, and personalization strategies*, making it foundational for scalable, privacy-conscious ML systems.

To better understand the role of federated learning, it is useful to contrast it with other learning paradigms. Figure 10.7 illustrates the distinction between offline learning, on-device learning, and federated learning. In traditional offline learning, all data is collected and processed centrally. The model is trained in the cloud using curated datasets and is then deployed to edge devices without further adaptation. In contrast, on-device learning enables local model adaptation using data generated on the device itself, supporting personalization but in isolation—without sharing insights across users. Federated learning bridges these two extremes by enabling localized training while coordinating updates globally. It retains data privacy by keeping raw data local, yet benefits from distributed model improvements by aggregating updates from many devices.

Figure 10.7: A comparison of learning paradigms: Offline learning occurs centrally with all data aggregated in the cloud. On-device learning adapts models locally based on user data but does not share information across users. Federated learning combines local adaptation with global coordination by aggregating model updates without sharing raw data, enabling privacy-preserving collective improvement.



This section explores the principles and practical considerations of federated learning in the context of mobile and embedded systems. It begins by outlining the canonical FL protocols and their system implications. It then discusses device participation constraints, communication-efficient update mechanisms, and strategies for personalized learning. Throughout, the emphasis remains on how federated methods can extend the reach of on-device learning by enabling distributed model training across diverse and resource-constrained hardware platforms.

10.6.1 Motivation

Federated learning (FL) is a decentralized paradigm for training machine learning models across a population of devices without transferring raw data to

a central server (McMahan et al. 2017). Unlike traditional centralized training pipelines, which require aggregating all training data in a single location, federated learning distributes the training process itself. Each participating device computes updates based on its local data and contributes to a global model through an aggregation protocol, typically coordinated by a central server. This shift in training architecture aligns closely with the needs of mobile, edge, and embedded systems, where privacy, communication cost, and system heterogeneity impose significant constraints on centralized approaches.

The relevance of federated learning becomes apparent in several practical domains. In mobile keyboard applications, such as Google’s Gboard, the system must continuously improve text prediction models based on user-specific input patterns (Hard et al. 2018). Federated learning allows the system to train on device-local keystroke data—preserving privacy—while still contributing to a shared model that benefits all users. Similarly, wearable health monitors often collect biometric signals that vary greatly between individuals. Training models centrally on such data would require uploading sensitive physiological traces, raising both ethical and regulatory concerns. FL mitigates these issues by enabling model updates to be computed directly on the wearable device.

In the context of smart assistants and voice interfaces, devices must adapt to individual voice profiles while minimizing false activations. Wake-word models, for instance, can be personalized locally and periodically synchronized through federated updates, avoiding the need to transmit raw voice recordings. Industrial and environmental sensors, deployed in remote locations or operating under severe bandwidth limitations, benefit from federated learning by enabling local adaptation and global coordination without constant connectivity.

These examples illustrate how federated learning bridges the gap between model improvement and system-level constraints. It enables personalization without compromising user privacy, supports learning under limited connectivity, and distributes computation across a diverse and heterogeneous device fleet. However, these benefits come with new challenges. Federated learning systems must account for client variability, communication efficiency, and the non-IID nature of local data distributions. Furthermore, they must ensure robustness to adversarial behavior and provide guarantees on model performance despite partial participation or dropout.

The remainder of this section explores the key techniques and tradeoffs that define federated learning in on-device settings. We begin by examining the core learning protocols that govern coordination across devices, and proceed to investigate strategies for scheduling, communication efficiency, and personalization.

10.6.2 Federated Learning Protocols

Federated learning protocols define the rules and mechanisms by which devices collaborate to train a shared model. These protocols govern how local updates are computed, aggregated, and communicated, as well as how devices participate in the training process. The choice of protocol has significant implications for system performance, communication overhead, and model convergence.

In this section, we outline the core components of federated learning protocols, including local training, aggregation methods, and communication strategies. We also discuss the tradeoffs associated with different approaches and their implications for on-device learning systems.

10.6.2.1 Local Training

Local training refers to the process by which individual devices compute model updates based on their local data. This step is important in federated learning, as it allows devices to adapt the shared model to their specific contexts without transferring raw data. The local training process typically involves the following steps:

1. **Model Initialization:** Each device initializes its local model parameters, often by downloading the latest global model from the server.
2. **Local Data Sampling:** The device samples a subset of its local data for training. This data may be non-IID, meaning that it may not be uniformly distributed across devices.
3. **Local Training:** The device performs a number of training iterations on its local data, updating the model parameters based on the computed gradients.
4. **Model Update:** After local training, the device computes a model update (e.g., the difference between the updated and initial parameters) and prepares to send it to the server.
5. **Communication:** The device transmits the model update to the server, typically using a secure communication channel to protect user privacy.
6. **Model Aggregation:** The server aggregates the updates from multiple devices to produce a new global model, which is then distributed back to the participating devices.

This process is repeated iteratively, with devices periodically downloading the latest global model and performing local training. The frequency of these updates can vary based on system constraints, device availability, and communication costs.

10.6.2.2 Federated Learning Protocols

At the heart of federated learning is a coordination mechanism that enables many devices—each with access to only a small, local dataset—to collaboratively train a shared model. This is achieved through a protocol in which client devices perform local training and periodically transmit model updates to a central server. The server aggregates these updates to refine a global model, which is then redistributed to clients for the next training round. This cyclical procedure decouples the learning process from centralized data collection, making it especially well-suited to mobile and edge environments where user data is private, bandwidth is constrained, and device participation is sporadic.

The most widely used baseline for this process is Federated Averaging (FedAvg), which has become a canonical algorithm for federated learning (McMahan et al. 2017). In FedAvg, each device trains its local copy of the model using

stochastic gradient descent (SGD) on its private data. After a fixed number of local steps, each device sends its updated model parameters to the server. The server computes a weighted average of these parameters—weighted by the number of data samples on each device—and updates the global model accordingly. This updated model is then sent back to the devices, completing one round of training.

Formally, let \mathcal{D}_k denote the local dataset on client k , and let θ_k^t be the parameters of the model on client k at round t . Each client performs E steps of SGD on its local data, yielding an update θ_k^{t+1} . The central server then aggregates these updates as:

$$\theta^{t+1} = \sum_{k=1}^K \frac{n_k}{n} \theta_k^{t+1}$$

where $n_k = |\mathcal{D}_k|$ is the number of samples on device k , $n = \sum_k n_k$ is the total number of samples across participating clients, and K is the number of active devices in the current round.

This basic structure introduces a number of design choices and tradeoffs. The number of local steps E impacts the balance between computation and communication: larger E reduces communication frequency but risks divergence if local data distributions vary too much. Similarly, the selection of participating clients affects convergence stability and fairness. In real-world deployments, not all devices are available at all times, and hardware capabilities may differ substantially, requiring robust participation scheduling and failure tolerance.

10.6.2.3 Client Scheduling and System Participation

Federated learning operates under the assumption that clients—devices holding local data—periodically become available for participation in training rounds. However, in real-world systems, client availability is intermittent and highly variable. Devices may be turned off, disconnected from power, lacking network access, or otherwise unable to participate at any given time. As a result, client scheduling plays a central role in the effectiveness and efficiency of distributed learning.

At a baseline level, federated learning systems define eligibility criteria for participation. Devices must meet minimum requirements such as being plugged in, connected to Wi-Fi, and idle, to avoid interfering with user experience or depleting battery resources. These criteria determine which subset of the total population is considered “available” for any given training round.

Beyond these operational filters, devices also differ in their hardware capabilities, data availability, and network conditions. For example, some smartphones may contain many recent examples relevant to the current task, while others may have outdated or irrelevant data. Network bandwidth and upload speed may vary widely depending on geography and carrier infrastructure. As a result, selecting clients at random can lead to poor coverage of the underlying data distribution and unstable model convergence.

Moreover, availability-driven selection introduces participation bias: clients with favorable conditions—such as frequent charging, high-end hardware, or consistent connectivity—are more likely to participate repeatedly, while others

are systematically underrepresented. This can skew the resulting model toward behaviors and preferences of a privileged subset of the population, raising both fairness and generalization concerns.

To address these challenges, systems must carefully balance scheduling efficiency with client diversity. A key approach involves using stratified or quota-based sampling to ensure representative client participation across different groups. For instance, asynchronous buffer-based techniques allow participating clients to contribute model updates independently, without requiring synchronized coordination in every round (Nguyen et al. 2021). This model has been extended to incorporate staleness awareness (Rodio and Neglia 2024) and fairness mechanisms (Ma et al. 2024), preventing bias from over-active clients who might otherwise dominate the training process.

To address these challenges, federated learning systems implement adaptive client selection strategies. These include prioritizing clients with underrepresented data types, targeting geographies or demographics that are less frequently sampled, and using historical participation data to enforce fairness constraints. Systems may also incorporate predictive modeling to anticipate future client availability or success rates, improving training throughput.

Selected clients perform one or more local training steps on their private data and transmit their model updates to a central server. These updates are aggregated to form a new global model. Typically, this aggregation is weighted, where the contributions of each client are scaled—such as by the number of local examples used during training—before averaging. This ensures that clients with more representative or larger datasets exert proportional influence on the global model.

These scheduling decisions directly impact system performance. They affect convergence rate, model generalization, energy consumption, and overall user experience. Poor scheduling can result in excessive stragglers, overfitting to narrow client segments, or wasted computation. As a result, client scheduling is not merely a logistical concern—it is a core component of system design in federated learning, demanding both algorithmic insight and infrastructure-level coordination.

10.6.2.4 Communication-Efficient Updates

One of the principal bottlenecks in federated learning systems is the cost of communication between edge clients and the central server. Transmitting full model weights or gradients after every training round can quickly overwhelm bandwidth and energy budgets—particularly for mobile or embedded devices operating over constrained wireless links. To address this, a range of techniques have been developed to reduce communication overhead while preserving learning efficacy.

These techniques fall into three primary categories: model compression, selective update sharing, and architectural partitioning.

Model compression methods aim to reduce the size of transmitted updates through quantization, sparsification, or subsampling. For instance, instead of sending full-precision gradients, a client may transmit 8-bit quantized updates or communicate only the top- k gradient elements with highest magnitude.

These techniques significantly reduce transmission size with limited impact on convergence when applied carefully.

Selective update sharing further reduces communication by transmitting only subsets of model parameters or updates. In layer-wise selective sharing, clients may update only certain layers—typically the final classifier or adapter modules—while keeping the majority of the backbone frozen. This reduces both upload cost and the risk of overfitting shared representations to non-representative client data.

Split models and architectural partitioning divide the model into a shared global component and a private local component. Clients train and maintain their private modules independently while synchronizing only the shared parts with the server. This allows for user-specific personalization with minimal communication and privacy leakage.

All of these approaches operate within the context of a federated aggregation protocol. A standard baseline for aggregation is Federated Averaging (FedAvg), in which the server updates the global model by computing a weighted average of the client updates received in a given round. Let \mathcal{K}_t denote the set of participating clients in round t , and let θ_k^t represent the locally updated model parameters from client k . The server computes the new global model θ^{t+1} as:

$$\theta^{t+1} = \sum_{k \in \mathcal{K}_t} \frac{n_k}{n_{\mathcal{K}_t}} \theta_k^t$$

Here, n_k is the number of local training examples at client k , and $n_{\mathcal{K}_t} = \sum_{k \in \mathcal{K}_t} n_k$ is the total number of training examples across all participating clients. This data-weighted aggregation ensures that clients with more training data exert a proportionally larger influence on the global model, while also accounting for partial participation and heterogeneous data volumes.

However, communication-efficient updates can introduce tradeoffs. Compression may degrade gradient fidelity, selective updates can limit model capacity, and split architectures may complicate coordination. As a result, effective federated learning requires careful balancing of bandwidth constraints, privacy concerns, and convergence dynamics—a balance that depends heavily on the capabilities and variability of the client population.

10.6.2.5 Personalization in Federated Learning

While compression and communication strategies improve scalability, they do not address a critical limitation of the global federated learning paradigm—its inability to capture user-specific variation. In real-world deployments, devices often observe distinct and heterogeneous data distributions. A one-size-fits-all global model may underperform when applied uniformly across diverse users. This motivates the need for personalized federated learning, where local models are adapted to user-specific data without compromising the benefits of global coordination.

Let θ_k denote the model parameters on client k , and θ_{global} the aggregated global model. Traditional FL seeks to minimize a global objective:

$$\min_{\theta} \sum_{k=1}^K w_k \mathcal{L}_k(\theta)$$

where $\mathcal{L}_k(\theta)$ is the local loss on client k , and w_k is a weighting factor (e.g., proportional to local dataset size). However, this formulation assumes that a single model θ can serve all users well. In practice, local loss landscapes \mathcal{L}_k often differ significantly across clients, reflecting non-IID data distributions and varying task requirements.

Personalization modifies this objective to allow each client to maintain its own adapted parameters θ_k , optimized with respect to both the global model and local data:

$$\min_{\theta_1, \dots, \theta_K} \sum_{k=1}^K \left(\mathcal{L}_k(\theta_k) + \lambda \cdot \mathcal{R}(\theta_k, \theta_{\text{global}}) \right)$$

Here, \mathcal{R} is a regularization term that penalizes deviation from the global model, and λ controls the strength of this penalty. This formulation enables local models to deviate as needed, while still benefiting from global coordination.

Real-world use cases illustrate the importance of this approach. Consider a wearable health monitor that tracks physiological signals to classify physical activities. While a global model may perform reasonably well across the population, individual users exhibit unique motion patterns, gait signatures, or sensor placements. Personalized finetuning of the final classification layer or low-rank adapters enables improved accuracy, particularly for rare or user-specific classes.

Several personalization strategies have emerged to address the tradeoffs between compute overhead, privacy, and adaptation speed. One widely used approach is local finetuning, in which each client downloads the latest global model and performs a small number of gradient steps using its private data. While this method is simple and preserves privacy, it may yield suboptimal results when the global model is poorly aligned with the client's data distribution or when the local dataset is extremely limited.

Another effective technique involves personalization layers, where the model is partitioned into a shared backbone and a lightweight, client-specific head—typically the final classification layer (Arivazhagan et al. 2019). Only the head is updated on-device, significantly reducing memory usage and training time. This approach is particularly well-suited for scenarios in which the primary variation across clients lies in output categories or decision boundaries.

Clustered federated learning offers an alternative by grouping clients according to similarities in their data or performance characteristics, and training separate models for each cluster. This strategy can enhance accuracy within homogeneous subpopulations but introduces additional system complexity and may require exchanging metadata to determine group membership.

Finally, meta-learning approaches, such as Model-Agnostic Meta-Learning (MAML), aim to produce a global model initialization that can be quickly adapted to new tasks with just a few local updates (Finn, Abbeel, and Levine 2017). This technique is especially useful when clients have limited data or operate in environments with frequent distributional shifts. Each of these strategies reflects a different point in the tradeoff space. These strategies vary in their system implications, including compute overhead, privacy guarantees, and adaptation latency. Table 10.3 summarizes the tradeoffs.

Table 10.3: Comparison of personalization strategies in federated learning, evaluating their system-level tradeoffs across multiple design dimensions.

Strategy	Personalization Mechanism	Compute Overhead	Privacy Preservation	Adaptation Speed
Local Finetuning	Gradient descent on local loss	Low to Moderate	High (no data sharing)	Fast (few steps)
Personalization Layers	post-aggregation Split model: shared base + user-specific head	Moderate	High	Fast (train small head)
Clustered FL	Group clients by data similarity, train per group	Moderate to High	Medium (group metadata)	Medium
Meta-Learning	Train for fast adaptation across tasks/devices	High (meta-objective)	High	Very Fast (few-shot)

Selecting the appropriate personalization method depends on deployment constraints, data characteristics, and the desired balance between accuracy, privacy, and computational efficiency. In practice, hybrid approaches that combine elements of multiple strategies—such as local finetuning atop a personalized head—are often employed to achieve robust performance across heterogeneous devices.

10.6.2.6 Privacy Considerations in Federated Learning

While federated learning is often motivated by privacy concerns—keeping raw data localized rather than transmitting it to a central server—the paradigm introduces its own set of security and privacy risks. Although devices do not share their raw data, the transmitted model updates (such as gradients or weight changes) can inadvertently leak information about the underlying private data. Techniques such as model inversion attacks and membership inference attacks demonstrate that adversaries may partially reconstruct or infer properties of local datasets by analyzing these updates.

To mitigate such risks, modern federated learning systems commonly employ protective measures. Secure Aggregation protocols ensure that individual model updates are encrypted and aggregated in a way that the server only observes the combined result, not any individual client’s contribution. Differential Privacy techniques inject carefully calibrated noise into updates to mathematically bound the information that can be inferred about any single client’s data.

While these techniques enhance privacy, they introduce additional system complexity and tradeoffs between model utility, communication cost, and robustness. A deeper exploration of these attacks, defenses, and their implications for federated and on-device learning is provided in a later security and privacy chapter.

10.7 Designing Practical On-Device Learning Systems

On-device learning presents opportunities for personalization, privacy preservation, and autonomous adaptation, but realizing these benefits in practice

requires disciplined system design. Constraints on memory, compute, energy, and observability necessitate careful selection of adaptation mechanisms, training strategies, and deployment safeguards.

A key principle in building practical systems is to minimize the adaptation footprint. Full-model fine-tuning is typically infeasible on edge platforms, instead, localized update strategies—such as bias-only optimization, residual adapters, or lightweight task-specific heads—should be prioritized. These approaches enable model specialization under resource constraints while mitigating the risks of overfitting or instability.

The feasibility of lightweight adaptation depends critically on the strength of offline pretraining (Bommasani et al. 2021). Pretrained models should encapsulate generalizable feature representations that allow efficient adaptation from limited local data. Shifting the burden of feature extraction to centralized training reduces the complexity and energy cost of on-device updates, while improving convergence stability in data-sparse environments.

Even when adaptation is lightweight, opportunistic scheduling remains essential to preserve system responsiveness and user experience. Local updates should be deferred to periods when the device is idle, connected to external power, and operating on a reliable network. Such policies minimize the impact of background training on latency, battery consumption, and thermal performance.

The sensitivity of local training artifacts necessitates careful data security measures. Replay buffers, support sets, adaptation logs, and model update metadata must be protected against unauthorized access or tampering. Lightweight encryption or hardware-backed secure storage can mitigate these risks without imposing prohibitive resource costs on edge platforms.

However, security measures alone do not guarantee model robustness. As models adapt locally, monitoring adaptation dynamics becomes critical. Lightweight validation techniques—such as confidence scoring, drift detection heuristics, or shadow model evaluation—can help identify divergence early, enabling systems to trigger rollback mechanisms before severe degradation occurs (Gama et al. 2014).

Robust rollback procedures depend on retaining trusted model checkpoints. Every deployment should preserve a known-good baseline version of the model that can be restored if adaptation leads to unacceptable behavior. This principle is especially important in safety-critical and regulated domains, where failure recovery must be provable and rapid.

In decentralized or federated learning contexts, communication efficiency becomes a first-order design constraint. Compression techniques such as quantized gradient updates, sparsified parameter sets, and selective model transmission must be employed to enable scalable coordination across large, heterogeneous fleets of devices without overwhelming bandwidth or energy budgets (Konečný et al. 2016).

Moreover, when personalization is required, systems should aim for localized adaptation wherever possible. Restricting updates to lightweight components—such as final classification heads or modular adapters—constrains the risk of catastrophic forgetting, reduces memory overhead, and accelerates adaptation without destabilizing core model representations.

Finally, throughout the system lifecycle, privacy and compliance requirements must be architected into adaptation pipelines. Mechanisms to support user consent, data minimization, retention limits, and the right to erasure must be considered fundamental aspects of model design, not post-hoc adjustments. Meeting regulatory obligations at scale demands that on-device learning workflows align inherently with principles of auditable autonomy.

The flowchart in Figure 10.8 summarizes key decision points in designing practical, scalable, and resilient on-device learning systems.

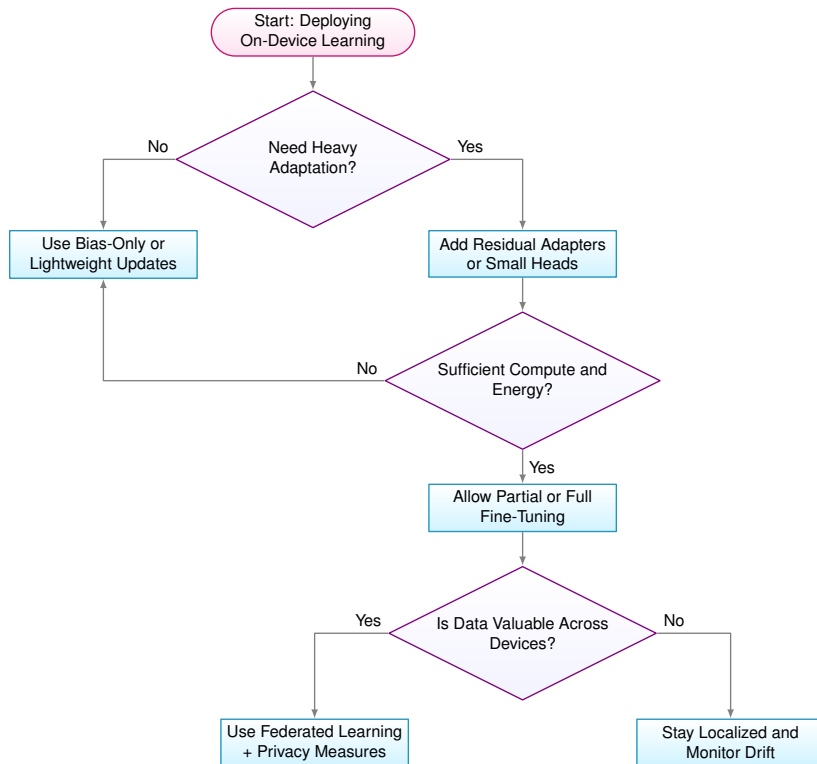


Figure 10.8: Decision flowchart for designing practical on-device learning systems.

10.8 Challenges and Limitations

While on-device learning holds significant promise for enabling adaptive, private, and efficient machine learning at the edge, its practical deployment introduces a range of challenges that extend beyond algorithm design. Unlike conventional centralized systems, where training occurs in controlled environments with uniform hardware and curated datasets, edge systems must contend with heterogeneity in devices, fragmentation in data, and the absence of centralized validation infrastructure. These factors give rise to new systems-level tradeoffs and open questions concerning reliability, safety, and maintainability. Moreover, regulatory and operational constraints complicate the deployment

of self-updating models in real-world applications. This section explores these limitations, emphasizing the systemic barriers that must be addressed to make on-device learning robust, scalable, and trustworthy.

10.8.0.1 System Heterogeneity

Federated and on-device learning systems must operate across a vast and diverse ecosystem of devices, ranging from smartphones and wearables to IoT sensors and microcontrollers. This heterogeneity spans multiple dimensions: hardware capabilities, software stacks, network connectivity, and power availability. Unlike cloud-based systems, where environments can be standardized and controlled, edge deployments encounter a wide distribution of system configurations and constraints. These variations introduce significant complexity in algorithm design, resource scheduling, and model deployment.

At the hardware level, devices differ in terms of memory capacity, processor architecture (e.g., ARM Cortex-M vs. A-series), instruction set support (e.g., availability of SIMD or floating-point units), and the presence or absence of AI accelerators. Some clients may possess powerful NPUs capable of running small training loops, while others may rely solely on low-frequency CPUs with minimal RAM. These differences affect the feasible size of models, the choice of training algorithm, and the frequency of updates.

Software heterogeneity compounds the challenge. Devices may run different versions of operating systems, kernel-level drivers, and runtime libraries. Some environments support optimized ML runtimes like TensorFlow Lite Micro or ONNX Runtime Mobile, while others rely on custom inference stacks or restricted APIs. These discrepancies can lead to subtle inconsistencies in behavior, especially when models are compiled differently or when floating-point precision varies across platforms.

In addition to computational heterogeneity, devices exhibit variation in connectivity and uptime. Some are intermittently connected, plugged in only occasionally, or operate under strict bandwidth constraints. Others may have continuous power and reliable networking, but still prioritize user-facing responsiveness over background learning. These differences complicate the orchestration of coordinated learning and the scheduling of updates.

Finally, system fragmentation affects reproducibility and testing. With such a wide range of execution environments, it is difficult to ensure consistent model behavior or to debug failures reliably. This makes monitoring, validation, and rollback mechanisms more critical—but also more difficult to implement uniformly across the fleet.

Consider a federated learning deployment for mobile keyboards. A high-end smartphone might feature 8 GB of RAM, a dedicated AI accelerator, and continuous Wi-Fi access. In contrast, a budget device may have just 2 GB of RAM, no hardware acceleration, and rely on intermittent mobile data. These disparities influence how long training runs can proceed, how frequently models can be updated, and even whether training is feasible at all. To support such a range, the system must dynamically adjust training schedules, model formats, and compression strategies—ensuring equitable model improvement across users while respecting each device's limitations.

10.8.0.2 Data Fragmentation and Non-IID Distributions

In centralized machine learning, data can be aggregated, shuffled, and curated to approximate independent and identically distributed (IID) samples—a key assumption underlying many learning algorithms. In contrast, on-device and federated learning systems must contend with highly fragmented and non-IID data. Each device collects data specific to its user, context, and usage patterns. These data distributions are often skewed, sparse, and dynamically shifting over time.

From a statistical standpoint, the non-IID nature of on-device data leads to challenges in both optimization and generalization. Gradients computed on one device may conflict with those from another, slowing convergence or destabilizing training. Local updates can cause models to overfit to the idiosyncrasies of individual clients, reducing performance when aggregated globally. Moreover, the diversity of data across clients complicates evaluation and model validation: there is no single test set that reflects the true deployment distribution.

The fragmentation also limits the representativeness of any single client's data. Many clients may observe only a narrow slice of the input space or task distribution, making it difficult to learn robust or generalizable representations. Devices might also encounter new classes or tasks not seen during centralized pretraining, requiring mechanisms for out-of-distribution detection and continual adaptation.

These challenges demand algorithms that are robust to heterogeneity and resilient to imbalanced participation. Techniques such as personalization layers, importance weighting, and adaptive aggregation schemes attempt to mitigate these issues, but there is no universally optimal solution. The degree and nature of non-IID data varies widely across applications, making this one of the most persistent and fundamental challenges in decentralized learning.

A common example of data fragmentation arises in speech recognition systems deployed on personal assistants. Each user exhibits a unique voice profile, accent, and speaking style, which results in significant differences across local datasets. Some users may issue frequent, clearly enunciated commands, while others speak infrequently or in noisy environments. These variations cause device-specific gradients to diverge, especially when training wake-word detectors or adapting language models locally.

In federated learning deployments for virtual keyboards, the problem is further amplified. One user might primarily type in English, another in Hindi, and a third may switch fluidly between multiple languages. The resulting training data is highly non-IID—not only in language but also in vocabulary, phrasing, and typing cadence. A global model trained on aggregated updates may degrade if it fails to capture these localized differences, highlighting the need for adaptive, data-aware strategies that accommodate heterogeneity without sacrificing collective performance.

10.8.0.3 Update Monitoring and Validation

Unlike centralized machine learning systems, where model updates can be continuously evaluated against a held-out validation set, on-device learning

introduces a fundamental shift in visibility and observability. Once deployed, models operate in highly diverse and often disconnected environments, where internal updates may proceed without external monitoring. This creates significant challenges for ensuring that model adaptation is both beneficial and safe.

A core difficulty lies in the absence of centralized validation data. In traditional workflows, models are trained and evaluated using curated datasets that serve as proxies for deployment conditions. On-device learners, by contrast, adapt in response to local inputs, which are rarely labeled and may not be systematically collected. As a result, the quality and direction of updates—whether they improve generalization or induce drift—are difficult to assess without interfering with the user experience or violating privacy constraints.

The risk of model drift is especially pronounced in streaming settings, where continual adaptation may cause a slow degradation in performance. For instance, a voice recognition model that adapts too aggressively to background noise may eventually overfit to transient acoustic conditions, reducing accuracy on the target task. Without visibility into the evolution of model parameters or outputs, such degradations can remain undetected until they become severe.

Mitigating this problem requires mechanisms for on-device validation and update gating. One approach is to interleave adaptation steps with lightweight performance checks—using proxy objectives or self-supervised signals to approximate model confidence (Y. Deng, Mokhtari, and Ozdaglar 2021). For example, a keyword spotting system might track detection confidence across recent utterances and suspend updates if confidence consistently drops below a threshold. Alternatively, shadow evaluation can be employed, where multiple model variants are maintained on the device and evaluated in parallel on incoming data streams, allowing the system to compare the adapted model's behavior against a stable baseline.

Another strategy involves periodic checkpointing and rollback, where snapshots of the model state are saved before adaptation. If subsequent performance degrades—based on downstream metrics or user feedback—the system can revert to a known good state. This approach has been used in health monitoring devices, where incorrect predictions could lead to user distrust or safety concerns. However, it introduces storage and compute overhead, especially in memory-constrained environments.

In some cases, federated validation offers a partial solution. Devices can share anonymized model updates or summary statistics with a central server, which aggregates them across users to identify global patterns of drift or failure. While this preserves some degree of privacy, it introduces communication overhead and may not capture rare or user-specific failures.

Ultimately, update monitoring and validation in on-device learning require a rethinking of traditional evaluation practices. Instead of centralized test sets, systems must rely on implicit signals, runtime feedback, and conservative adaptation policies to ensure robustness. The absence of global observability is not merely a technical limitation—it reflects a deeper systems challenge in aligning local adaptation with global reliability.

10.8.0.4 Resource Competition and Cost

On-device learning introduces new modes of resource contention that are not present in conventional inference-only deployments. While many edge devices are provisioned to run pretrained models efficiently, they are rarely designed with training workloads in mind. Local adaptation therefore competes for scarce resources—compute cycles, memory bandwidth, energy, and thermal headroom—with other system processes and user-facing applications.

The most direct constraint is compute availability. Training involves additional forward and backward passes through the model, which can significantly exceed the cost of inference. Even when only a small subset of parameters is updated—such as in bias-only or head-only adaptation—backpropagation must still traverse the relevant layers, triggering increased instruction counts and memory traffic. On devices with shared compute units (e.g., mobile SoCs or embedded CPUs), this demand can delay interactive tasks, reduce frame rates, or impair sensor processing.

Energy consumption compounds this problem. Adaptation typically involves sustained computation over multiple input samples, which taxes battery-powered systems and may lead to rapid energy depletion. For instance, performing a single epoch of adaptation on a microcontroller-class device can consume several millijoules—an appreciable fraction of the energy budget for a duty-cycled system operating on harvested power. This necessitates careful scheduling, such that learning occurs only during idle periods, when energy reserves are high and user latency constraints are relaxed.

From a memory perspective, training incurs higher peak usage than inference, due to the need to cache intermediate activations, gradients, and optimizer state (Lin et al. 2020). These requirements may exceed the static memory footprint anticipated during model deployment, particularly when adaptation involves multiple layers or gradient accumulation. In highly constrained systems—such as those with under 512 KB of RAM—this may preclude certain types of adaptation altogether, unless additional optimization techniques (e.g., checkpointing or low-rank updates) are employed.

These resource demands must also be balanced against quality of service (QoS) goals. Users expect edge devices to respond reliably and consistently, regardless of whether learning is occurring in the background. Any observable degradation—such as dropped audio in a wake-word detector or lag in a wearable display—can erode user trust. As such, many systems adopt opportunistic learning policies, where adaptation is suspended during foreground activity and resumed only when system load is low.

In some deployments, adaptation is further gated by cost constraints imposed by networked infrastructure. For instance, devices may offload portions of the learning workload to nearby gateways or cloudlets¹⁶, introducing bandwidth and communication trade-offs. These hybrid models raise additional questions of task placement and scheduling: should the update occur locally, or be deferred until a high-throughput link is available?

In summary, the cost of on-device learning is not solely measured in FLOPs or memory usage. It manifests as a complex interplay of system load, user experience, energy availability, and infrastructure capacity. Addressing these

¹⁶ Cloudlets: Smaller-scale cloud datacenters located at the edge of the internet to decrease latency for mobile and wearable devices.

challenges requires co-design across algorithmic, runtime, and hardware layers, ensuring that adaptation remains unobtrusive, efficient, and sustainable under real-world constraints.

10.8.0.5 Deployment and Compliance Risks

The deployment of adaptive models on edge devices introduces challenges that extend beyond technical feasibility. In domains where compliance, auditability, and regulatory approval are necessary—such as healthcare, finance, or safety-critical systems—on-device learning poses a fundamental tension between system autonomy and control.

In traditional machine learning pipelines, all model updates are centrally managed, versioned, and validated. The training data, model checkpoints, and evaluation metrics are typically recorded in reproducible workflows that support traceability. When learning occurs on the device itself, however, this visibility is lost. Each device may independently evolve its model parameters, influenced by unique local data streams that are never observed by the developer or system maintainer.

This autonomy creates a validation gap. Without access to the input data or the exact update trajectory, it becomes difficult to verify that the learned model still adheres to its original specification or performance guarantees. This is especially problematic in regulated industries, where certification depends on demonstrating that a system behaves consistently across defined operational boundaries. A device that updates itself in response to real-world usage may drift outside those bounds, triggering compliance violations without any external signal.

Moreover, the lack of centralized oversight complicates rollback and failure recovery. If a model update degrades performance, it may not be immediately detectable—particularly in offline scenarios or systems without telemetry. By the time failure is observed, the system’s internal state may have diverged significantly from any known checkpoint, making diagnosis and recovery more complex than in static deployments. This necessitates robust safety mechanisms, such as conservative update thresholds, rollback caches, or dual-model architectures¹⁷ that retain a verified baseline.

In addition to compliance challenges, on-device learning introduces new security vulnerabilities. Because model adaptation occurs locally and relies on device-specific, potentially untrusted data streams, adversaries may attempt to manipulate the learning process—by tampering with stored data such as replay buffers, or by injecting poisoned examples during adaptation—to degrade model performance or introduce vulnerabilities. Furthermore, any locally stored adaptation data, such as feature embeddings or few-shot examples, must be secured against unauthorized access to prevent unintended information leakage.

Maintaining model integrity over time is particularly difficult in decentralized settings, where central monitoring and validation are limited. Autonomous updates could, without external visibility, cause models to drift into unsafe or biased states. These risks are compounded by compliance obligations such as the GDPR’s right to erasure: if user data subtly influences a model through adaptation, tracking and reversing that influence becomes complex.

¹⁷ | System designs that employ two separate models, typically to enhance reliability and safety by providing a fallback.

The security and integrity of self-adapting models—especially at the edge—pose critical open challenges. A comprehensive treatment of these threats and corresponding mitigation strategies, including attack models and edge-specific defenses, is presented in Chapter 15: Security and Privacy.

Privacy regulations also interact with on-device learning in nontrivial ways. While local adaptation can reduce the need to transmit sensitive data, it may still require storage and processing of personal information—such as sensor traces or behavioral logs—on the device itself. Depending on jurisdiction, this may invoke additional requirements for data retention, user consent, and auditability. Systems must be designed to satisfy these requirements without compromising adaptation effectiveness, which often involves encrypting stored data, enforcing retention limits, or implementing user-controlled reset mechanisms.

Lastly, the emergence of edge learning raises open questions about accountability and liability (Brakerski et al. 2022). When a model adapts autonomously, who is responsible for its behavior? If an adapted model makes a faulty decision—such as misdiagnosing a health condition or misinterpreting a voice command—the root cause may lie in local data drift, poor initialization, or insufficient safeguards. Without standardized mechanisms for capturing and analyzing these failure modes, responsibility may be difficult to assign, and regulatory approval harder to obtain.

Addressing these deployment and compliance risks requires new tooling, protocols, and design practices that support auditable autonomy—the ability of a system to adapt in place while still satisfying external requirements for traceability, reproducibility, and user protection. As on-device learning becomes more prevalent, these challenges will become central to both system architecture and governance frameworks.

10.8.0.6 Summary

Designing on-device learning systems involves navigating a complex landscape of technical and practical constraints. While localized adaptation enables personalization, privacy, and responsiveness, it also introduces a range of challenges that span hardware heterogeneity, data fragmentation, observability, and regulatory compliance.

System heterogeneity complicates deployment and optimization by introducing variation in compute, memory, and runtime environments. Non-IID data distributions challenge learning stability and generalization, especially when models are trained on-device without access to global context. The absence of centralized monitoring makes it difficult to validate updates or detect performance regressions, and training activity must often compete with core device functionality for energy and compute. Finally, post-deployment learning introduces complications in model governance, from auditability and rollback to privacy assurance.

These challenges are not isolated—they interact in ways that influence the viability of different adaptation strategies. Table 10.4 summarizes the primary challenges and their implications for ML systems deployed at the edge.

Table 10.4: Challenges in on-device learning and their implications for system design and deployment.

Challenge	Root Cause	System-Level Implications
System Heterogeneity	Diverse hardware, software, and toolchains	Limits portability; requires platform-specific tuning
Non-IID and Fragmented Data	Localized, user-specific data distributions	Hinders generalization; increases risk of drift
Limited Observability and Feedback	No centralized testing or logging	Makes update validation and debugging difficult
Resource Contention and Scheduling	Competing demands for memory, compute, and battery	Requires dynamic scheduling and budget-aware learning
Deployment and Compliance Risk	Learning continues post-deployment	Complicates model versioning, auditing, and rollback

10.9 Conclusion

On-device learning is a major shift in the design and operation of machine learning systems. Rather than relying exclusively on centralized training and static model deployment, this paradigm enables systems to adapt dynamically to local data and usage conditions. This shift is motivated by a confluence of factors—ranging from the need for personalization and privacy preservation to latency constraints and infrastructure efficiency. However, it also introduces a new set of challenges tied to the constrained nature of edge computing platforms.

Throughout this chapter, we explored the architectural and algorithmic strategies that make on-device learning feasible under tight compute, memory, energy, and data constraints. We began by establishing the motivation for moving learning to the edge, followed by a discussion of the system-level limitations that shape practical design choices. A core insight is that no single solution suffices across all use cases. Instead, effective on-device learning systems combine multiple techniques: minimizing the number of trainable parameters, reducing runtime costs, leveraging memory-based adaptation, and compressing data representations for efficient supervision.

We also examined federated learning as a key enabler of decentralized model refinement, particularly when coordination across many heterogeneous devices is required. While federated approaches provide strong privacy guarantees and infrastructure scalability, they introduce new concerns around client scheduling, communication efficiency, and personalization—all of which must be addressed to ensure robust real-world deployments.

Finally, we turned a critical eye toward the limitations of on-device learning, including system heterogeneity, non-IID data distributions, and the absence of reliable evaluation mechanisms in the field. These challenges underscore the importance of co-designing learning algorithms with hardware, runtime, and privacy constraints in mind.

As machine learning continues to expand into mobile, embedded, and wearable environments, the ability to adapt locally—and do so responsibly, efficiently, and reliably—will be essential to the next generation of intelligent systems.

REFERENCES

References

- Arivazhagan, Manoj Ghuhun, Vinay Aggarwal, Aaditya Kumar Singh, and Sunav Choudhary. 2019. “Federated Learning with Personalization Layers.” *CoRR* abs/1912.00818 (December). <http://arxiv.org/abs/1912.00818v1>.
- Banbury, Colby, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, et al. 2021. “MLPerf Tiny Benchmark.” *arXiv Preprint arXiv:2106.07597*, June. <http://arxiv.org/abs/2106.07597v4>.
- Bommasani, Rishi, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, et al. 2021. “On the Opportunities and Risks of Foundation Models.” *arXiv Preprint arXiv:2108.07258*, August. <http://arxiv.org/abs/2108.07258v3>.
- Brakerski, Zvika et al. 2022. “Federated Learning and the Rise of Edge Intelligence: Challenges and Opportunities.” *Communications of the ACM* 65 (8): 54–63.
- Chen, Wei-Yu, Yen-Cheng Liu, Zsolt Kira, Yu-Chiang Frank Wang, and Jia-Bin Huang. 2019. “A Closer Look at Few-Shot Classification.” In *International Conference on Learning Representations (ICLR)*.
- Dean, Jeffrey, and Sanjay Ghemawat. 2008. “MapReduce: Simplified Data Processing on Large Clusters.” *Communications of the ACM* 51 (1): 107–13. <https://doi.org/10.1145/1327452.1327492>.
- Deng, Chulin, Yujun Zhang, and Yanzhi Wu. 2022. “TinyTrain: Learning to Train Compact Neural Networks on the Edge.” In *Proceedings of the 39th International Conference on Machine Learning (ICML)*.
- Deng, Yuzhe, Aryan Mokhtari, and Asuman Ozdaglar. 2021. “Adaptive Federated Optimization.” In *Proceedings of the 38th International Conference on Machine Learning (ICML)*.
- Finn, Chelsea, Pieter Abbeel, and Sergey Levine. 2017. “Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks.” In *Proceedings of the 34th International Conference on Machine Learning (ICML)*.
- Gama, João, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. “A Survey on Concept Drift Adaptation.” *ACM Computing Surveys* 46 (4): 1–37. <https://doi.org/10.1145/2523813>.
- Hard, Andrew, Kanishka Rao, Rajiv Mathews, Saurabh Ramaswamy, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. 2018. “Federated Learning for Mobile Keyboard Prediction.” In *International Conference on Learning Representations (ICLR)*.

- Hayes, Tyler L., Kushal Kafle, Robik Shrestha, Manoj Acharya, and Christopher Kanan. 2020. "REMIND Your Neural Network to Prevent Catastrophic Forgetting." In *Computer Vision – ECCV 2020*, 466–83. Springer International Publishing. https://doi.org/10.1007/978-3-030-58598-3/_28.
- Houlsby, Neil, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Chloé de Laroussilhe, Andrea Gesmundo, Mohammad Attariyan, and Sylvain Gelly. 2019. "Parameter-Efficient Transfer Learning for NLP." In *International Conference on Machine Learning*, 2790–99. PMLR.
- Howard, Andrew G., Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." *ArXiv Preprint abs/1704.04861* (April). <http://arxiv.org/abs/1704.04861v1>.
- Hu, Edward J., Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. "LoRA: Low-Rank Adaptation of Large Language Models." *arXiv Preprint arXiv:2106.09685*, June. <http://arxiv.org/abs/2106.09685v2>.
- Konečný, Jakub, H. Brendan McMahan, Daniel Ramage, and Peter Richtárik. 2016. "Federated Optimization: Distributed Machine Learning for on-Device Intelligence." *CoRR*. <http://arxiv.org/abs/1610.02527>.
- Lai, Pete Warden Daniel Situnayake. 2020. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media.
- Levy, Orin, Alon Cohen, Asaf Cassel, and Yishay Mansour. 2023. "Efficient Rate Optimal Regret for Adversarial Contextual MDPs Using Online Function Approximation." *arXiv Preprint arXiv:2303.01464*, March. <http://arxiv.org/abs/2303.01464v2>.
- Li, Tian, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. 2020. "Federated Learning: Challenges, Methods, and Future Directions." *IEEE Signal Processing Magazine* 37 (3): 50–60. <https://doi.org/10.1109/msp.2020.2975749>.
- Lin, Ji, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. 2020. "MCUNet: Tiny Deep Learning on IoT Devices." In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, Virtual*, edited by Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin. <https://proceedings.neurips.cc/paper/2020/hash/86c51678350f656dccc7f490a43946ee5-Abstract.html>.
- Ma, Jeffrey, Alan Tu, Yiling Chen, and Vijay Janapa Reddi. 2024. "FedStaleWeight: Buffered Asynchronous Federated Learning with Fair Aggregation via Staleness Reweighting," June. <http://arxiv.org/abs/2406.02877v1>.
- McMahan, H Brendan, Eider Moore, Daniel Ramage, Seth Hampson, et al. 2017. "Communication-Efficient Learning of Deep Networks from Decentralized Data." In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 1273–82.
- Nguyen, John, Kshitiz Malik, Hongyuan Zhan, Ashkan Yousefpour, Michael Rabbat, Mani Malek, and Dzmitry Huba. 2021. "Federated Learning with

- Buffered Asynchronous Aggregation,” June. <http://arxiv.org/abs/2106.06639v4>.
- Quiñonero-Candela, Joaquin, Masashi Sugiyama, Anton Schwaighofer, and Neil D. Lawrence. 2008. “Dataset Shift in Machine Learning.” *The MIT Press*. The MIT Press. <https://doi.org/10.7551/mitpress/7921.003.0002>.
- Rebuffi, Sylvestre-Alvise, Hakan Bilen, and Andrea Vedaldi. 2017. “Learning Multiple Visual Domains with Residual Adapters.” In *Advances in Neural Information Processing Systems*. Vol. 30.
- Rodio, Angelo, and Giovanni Neglia. 2024. “FedStale: Leveraging Stale Client Updates in Federated Learning,” May. <http://arxiv.org/abs/2405.04171v1>.
- Rolnick, David, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Greg Wayne. 2019. “Experience Replay for Continual Learning.” In *Advances in Neural Information Processing Systems (NeurIPS)*.
- Sanh, Victor, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. “DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter.” *arXiv Preprint arXiv:1910.01108*, October. <http://arxiv.org/abs/1910.01108v4>.
- Wang, Shaofeng, Xiangyu Li, Wanli Ouyang, and Xiaogang Wang. 2020. “Pick and Choose: Selective Backpropagation for Efficient Network Training.” In *European Conference on Computer Vision (ECCV)*.
- Wang, Yaqing, Quanming Yao, James T. Kwok, and Lionel M. Ni. 2020. “Generalizing from a Few Examples: A Survey on Few-Shot Learning.” *ACM Computing Surveys* 53 (3): 1–34. <https://doi.org/10.1145/3386252>.
- Warden, Pete. 2018. “Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition.” *arXiv Preprint arXiv:1804.03209*, April. <http://arxiv.org/abs/1804.03209v1>.
- Zhang, Xitong, Jialin Song, and Dacheng Tao. 2020. “Efficient Task-Specific Adaptation for Deep Models.” In *International Conference on Learning Representations (ICLR)*.
- Zhao, Yue, Meng Li, Liangzhen Lai, Naveen Suda, Damon Civin, and Vikas Chandra. 2018. “Federated Learning with Non-IID Data.” *CoRR* abs/1806.00582 (June). <http://arxiv.org/abs/1806.00582v2>.

