12 On-Device Learning

Resources: Slides, Videos, Exercises, Labs

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

On-device Learning represents a significant innovation for embedded and edge IoT devices, enablir models to train and update directly on small local devices. This contrasts with traditional methods, models are trained on expansive cloud computing resources before deployment. With On-Device L devices like smart speakers, wearables, and industrial sensors can refine models in real-time basec data without needing to transmit data externally. For example, a voice-enabled smart speaker coul and adapt to its owner's speech patterns and vocabulary right on the device. However, there is no s thing as a free lunch; therefore, in this chapter, we will discuss both the benefits and the limitations device learning.

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 ✓

Connected

SocratiQ with tinyML

What is SocratiQ?

SocratiQ is an AI Generative Learning Assistant, designed to make learning more efficient, engaging and accessible. We welcome your Feedback. Learn more about SocratiQ.

1 2 3

I feel like this chp should come earlier - maybe cut down chp 2 and put this after ?

Mindfulness break | Examples | Explain | Find related research | Summarize this page | Quiz me

Ask anything...

Information provided here may not always be accurate. Provide feedback

12.1 Introduction

On-device Learning refers to training ML models directly on the device where they are deployed, as opposed to traditional methods where models are trained on powerful servers and then deployed to devices. This method is particularly relevant to TinyML, where ML systems are integrated into tiny, resource-constrained devices.

An example of On-Device Learning can be seen in a smart thermostat that adapts to user behavior over time. Initially, the thermostat may have a generic model that understands basic usage patterns. However, as it is exposed to more data, such as the times the user is home or away, preferred temperatures, and external weather conditions, the thermostat can refine its model directly on the device to provide a personalized experience. This is all done without sending data back to a central server for processing.

Another example is in predictive text on smartphones. As users type, the phone learns from the user's language patterns and suggests words or phrases that are likely to be used next. This learning happens directly on the device, and the model updates in real-time as more data is collected. A widely used real-world example of on-device learning is Gboard. On an Android phone, Gboard learns from typing and dictation patterns to enhance the experience for all users. On-device learning is also called federated learning. Figure 12.1 shows the cycle of federated learning on mobile devices: A. the device learns from user patterns; B. local model updates are communicated to the cloud; C. the cloud server updates the global model and sends the new model to all the devices.

some note-taking functionality ?

could use this background in TinyML section!

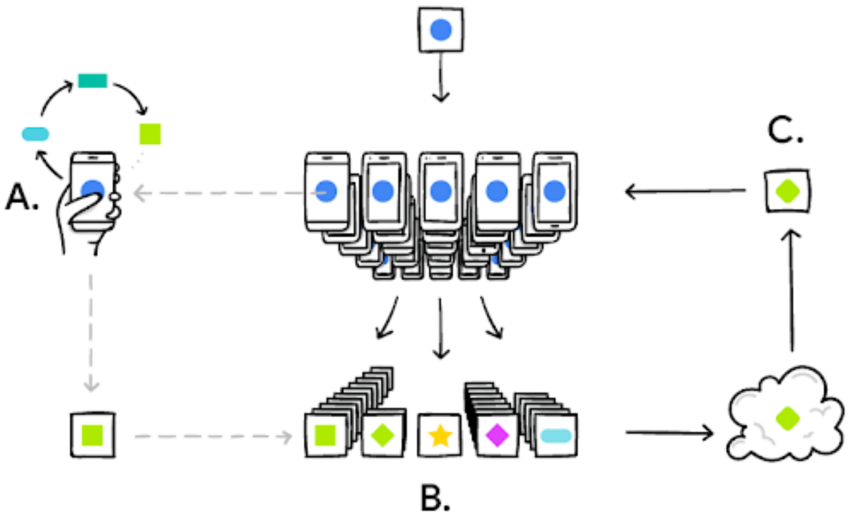


Figure 12.1: Federated learning cycle. Credit: Google Research.

12.2 Advantages and Limitations

On-device learning provides several advantages over traditional cloud-based ML. By keeping data and models on the device, it eliminates the need for costly data transmission and addresses privacy concerns. This allows for more personalized, responsive experiences, as the model can adapt in real-time to user behavior.

However, On-Device Learning also comes with tradeoffs. The limited computing resources on consumer devices can make it challenging to run complex models locally. Datasets are also more restricted since they consist only of user-generated data from a single device. Additionally, updating models requires pushing out new versions rather than seamless cloud updates.

On-device learning opens up new capabilities by enabling offline AI while maintaining user privacy. However, it requires carefully managing model and data complexity within the constraints of consumer devices. Finding the right balance between localization and cloud offloading is key to optimizing on-device experiences.

12.2.1 Benefits

Privacy and Data Security

One of the significant advantages of on-device learning is the enhanced privacy and security of user data. For instance, consider a smartwatch that monitors sensitive health metrics such as heart rate and blood pressure. By processing data and adapting models directly on the device, the biometric data remains localized, circumventing the need to transmit raw data to cloud servers where it could be susceptible to breaches.

table for this?

Server breaches are far from rare, with millions of records compromised annually. For example, the 2017 Equifax breach exposed the personal data of 147 million people. By keeping data on the device, the risk of such exposures is drastically minimized. On-device learning eliminates reliance on centralized cloud storage and safeguards against unauthorized access from various threats, including malicious actors, insider threats, and accidental exposure.

a lot of this was explained in TinyML chp earlier

Regulations like the Health Insurance Portability and Accountability Act (HIPAA) and the General Data Protection Regulation (GDPR) mandate stringent data privacy requirements that on-device learning adeptly addresses. By ensuring data remains localized and is not transferred to other systems, on-device learning facilitates compliance with these regulations.

On-device learning is not just beneficial for individual users; it has significant implications for organizations and sectors dealing with highly sensitive data. For instance, within the military, on-device learning empowers frontline systems to adapt models and function independently of connections to central servers that could potentially be compromised. Critical and sensitive information is staunchly protected by localizing data processing and learning. However, this comes with the tradeoff that individual devices take on more value and may incentivize theft or destruction as they become the sole carriers of specialized AI models. Care must be taken to secure devices themselves when transitioning to on-device learning.

It is also important to preserve the privacy, security, and regulatory compliance of personal and sensitive data. Instead of in the cloud, training and operating models locally substantially augment privacy measures, ensuring that user data is safeguarded from potential threats.

However, this is only partially intuitive because on-device learning could instead open systems up to new privacy attacks. With valuable data summaries and model updates permanently stored on individual devices, it may be much harder to physically and digitally protect them than a large computing cluster. While on-device learning reduces the amount of data compromised in any one breach, it could also introduce new dangers by dispersing sensitive information across many decentralized endpoints. Careful security practices are still essential for on-device systems.

why paranth?

Regulatory Compliance

just went over this?

On-device learning helps address major privacy regulations like (GDPR) and CCPA. These regulations require data localization, restricting cross-border data transfers to approved countries with adequate controls. GDPR also mandates privacy by design and consent requirements for data collection. By keeping data processing and model training localized on-device, sensitive user data is not transferred across borders. This avoids major compliance headaches for organizations.

For example, a healthcare provider monitoring patient vitals with wearables must ensure cross-border data transfers comply with HIPAA and GDPR if using the cloud. Determining which country's laws apply and securing approvals for international data flows introduces legal and engineering burdens. With on-device learning, no data leaves the device, simplifying compliance. The time and resources spent on compliance are reduced significantly.

Industries like healthcare, finance, and government, which have highly regulated data, can benefit greatly from on-device learning. By localizing data and learning, regulatory privacy and data sovereignty

requirements are more easily met. On-device solutions provide an efficient way to build compliant AI applications.

spacing?

Major privacy regulations impose restrictions on cross-border data movement that on-device learning inherently addresses through localized processing. This reduces the compliance burden for organizations working with regulated data.

Reduced Bandwidth, Costs, and Increased Efficiency

One major advantage of on-device learning is the significant reduction in bandwidth usage and associated cloud infrastructure costs. By keeping data localized for model training rather than transmitting raw data to the cloud, on-device learning can result in substantial bandwidth savings. For instance, a network of cameras analyzing video footage can achieve significant reductions in data transfer by training models on-device rather than streaming all video footage to the cloud for processing.

This reduction in data transmission saves bandwidth and translates to lower costs for servers, networking, and data storage in the cloud. Large organizations, which might spend millions on cloud infrastructure to train models on-device data, can experience dramatic cost reductions through on-device learning. In the era of Generative AI, where costs have been escalating significantly, finding ways to keep expenses down has become increasingly important.

Furthermore, the energy and environmental costs of running large server farms are also diminished. Data centers consume vast amounts of energy, contributing to greenhouse gas emissions. By reducing the need for extensive cloud-based infrastructure, on-device learning plays a part in mitigating the environmental impact of data processing (Wu et al. 2022).

Wu, Carole-Jean, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, et al. 2022. "Sustainable Ai: Environmental Implications, Challenges and Opportunities." *Proceedings of Machine Learning and Systems* 4: 795–813.

Specifically for endpoint applications, on-device learning minimizes the number of network API calls needed to run inference through a cloud provider. The cumulative costs associated with bandwidth and API calls can quickly escalate for applications with millions of users. In contrast, performing training and inferences locally is considerably more efficient and cost-effective. Under state-of-the-art optimizations, on-device learning has been shown to reduce training memory requirements, drastically improve memory efficiency, and reduce up to 20% in per-iteration latency (Dhar et al. 2021).

Another key benefit of on-device learning is the potential for IoT devices to continuously adapt their ML model to new data for continuous, lifelong learning. On-device models can quickly become outdated as user behavior, data patterns, and preferences change. Continuous learning enables the model to efficiently adapt to new data and improvements and maintain high model performance over time.

12.2.2 Limitations

went over this in earlier chps?

While traditional cloud-based ML systems have access to nearly endless computing resources, on-device learning is often restricted by the limitations in computational and storage power of the edge device that the model is trained on. By definition, an edge device is a device with restrained computing, memory, and energy resources that cannot be easily increased or decreased. Thus, the reliance on edge devices can restrict the complexity, efficiency, and size of on-device ML models.

Compute resources

Traditional cloud-based ML systems utilize large servers with multiple high-end GPUs or TPUs, providing nearly endless computational power and memory. For example, services like Amazon Web Services (AWS) EC2 allow configuring clusters of GPU instances for massively parallel training.

In contrast, on-device learning is restricted by the hardware limitations of the edge device on which it runs. Edge devices refer to endpoints like smartphones, embedded electronics, and IoT devices. By definition, these devices have highly restrained computing, memory, and energy resources compared to the cloud.

For example, a typical smartphone or Raspberry Pi may only have a few CPU cores, a few GB of RAM, and a small battery. Even more resource-constrained are TinyML microcontroller devices such as the Arduino Nano BLE Sense. The resources are fixed on these devices and can't easily be increased on demand, such as scaling cloud infrastructure. This reliance on edge devices directly restricts the complexity, efficiency, and size of models that can be deployed for on-device training:

- **Complexity:** Limits on memory, computing, and power restrict model architecture design, constraining the number of layers and parameters.
- **Efficiency:** Models must be heavily optimized through methods like quantization and pruning to run faster and consume less energy.
- **Size:** Actual model files must be compressed as much as possible to fit within the storage limitations of edge devices.

Thus, while the cloud offers endless scalability, on-device learning must operate within the tight resource constraints of endpoint hardware. This requires careful codesign of streamlined models, training methods, and optimizations tailored specifically for edge devices.

Dataset Size, Accuracy, and Generalization

In addition to limited computing resources, on-device learning is also constrained by the dataset available for training models.

In the cloud, models are trained on massive, diverse datasets like ImageNet or Common Crawl. For example, ImageNet contains over 14 million images carefully categorized across thousands of classes.

On-device learning instead relies on smaller, decentralized data silos unique to each device. A smartphone camera roll may contain only thousands of photos of users' interests and environments.

This decentralized data leads to a need for IID (independent and identically distributed) data. For instance, two friends may take many photos of the same places and objects, meaning their data distributions are highly correlated rather than independent.

Reasons data may be non-IID in on-device settings:

- **User heterogeneity:** Different users have different interests and environments.
- **Device differences:** Sensors, regions, and demographics affect data.
- **Temporal effects:** time of day, seasonal impacts on data.

The effectiveness of ML relies heavily on large, diverse training data. With small, localized datasets, on-device models may fail to generalize across different user populations and environments. For example, a disease detection model trained only on images from a single hospital would not generalize well to other patient demographics. Withel's real-world performance would only improve with extensive, diverse medical improvement. Thus, while cloud-based learning leverages massive datasets, on-device learning relies on much smaller, decentralized data silos unique to each user.

The limited data and optimizations required for on-device learning can negatively impact model accuracy and generalization:

- Small datasets increase overfitting risk. For example, a fruit classifier trained on 100 images risks overfitting compared to one trained on 1 million diverse images.
- Noisy user-generated data reduces quality. Sensor noise or improper data labeling by non-experts may degrade training.
- Optimizations like pruning and quantization trade off accuracy for efficiency. An 8-bit quantized model runs faster but less accurately than a 32-bit model.

So while cloud models achieve high accuracy with massive datasets and no constraints, on-device models can struggle to generalize. Some studies show that on-device training matches cloud accuracy on select tasks. However, performance on real-world workloads requires further study (Lin et al. 2022).

For instance, a cloud model can accurately detect pneumonia in chest X-rays from thousands of hospitals. However, an on-device model trained only on a small local patient population may fail to generalize.

Unreliable accuracy limits the real-world applicability of on-device learning for mission-critical uses like disease diagnosis or self-driving vehicles.

On-device training is also slower than the cloud due to limited resources. Even if each iteration is faster, the overall training process takes longer.

For example, a real-time robotics application may require model updates within milliseconds. On-device training on small embedded hardware may take seconds or minutes per update - too slow for real-time use.

Accuracy, generalization, and speed challenges pose hurdles to adopting on-device learning for real-world production systems, especially when reliability and low latency are critical.

12.3 On-device Adaptation

In an ML task, resource consumption mainly comes from three sources:

- The ML model itself;
- The optimization process during model learning
- Storing and processing the dataset used for learning.

Correspondingly, there are three approaches to adapting existing ML algorithms onto resource-constrained devices:

- Reducing the complexity of the ML model
- Modifying optimizations to reduce training resource requirements
- Creating new storage-efficient data representations

In the following section, we will review these on-device learning adaptation methods. The Model Optimizations chapter provides more details on model optimizations.

12.3.1 Reducing Model Complexity

In this section, we will briefly discuss ways to reduce model complexity when adapting ML models on-device. For details on reducing model complexity, please refer to the Model Optimization Chapter.

Traditional ML Algorithms

Due to edge devices' computing and memory limitations, select traditional ML algorithms are great candidates for on-device learning applications due to their lightweight nature. Some example algorithms with low resource footprints include Naive Bayes Classifiers, Support Vector Machines (SVMs), Linear Regression, Logistic Regression, and select Decision Tree algorithms.

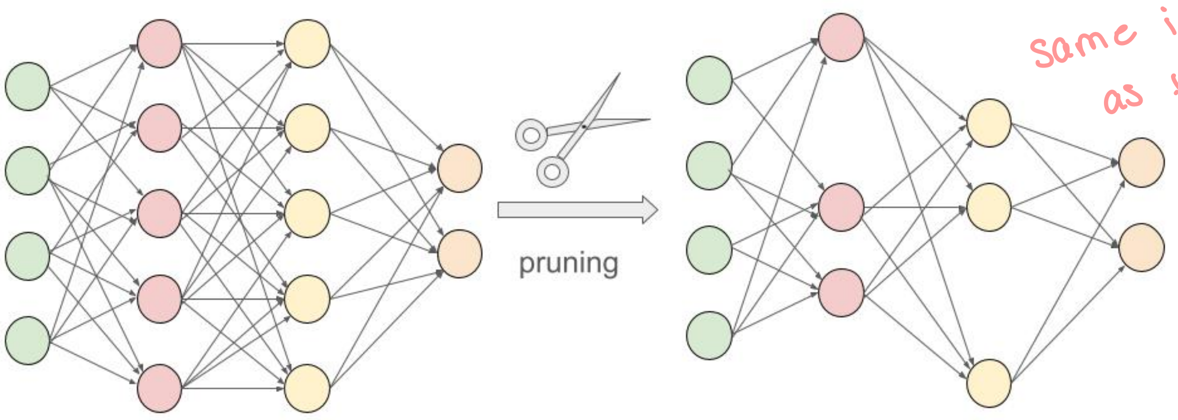
With some refinements, these classical ML algorithms can be adapted to specific hardware architectures and perform simple tasks. Their low-performance requirements make it easy to integrate continuous learning even on edge devices.

Pruning

↳ been defined twice already

Pruning is a technique for reducing the size and complexity of an ML model to improve its efficiency and generalization performance. This is beneficial for training models on edge devices, where we want to minimize resource usage while maintaining competitive accuracy.

The primary goal of pruning is to remove parts of the model that do not contribute significantly to its predictive power while retaining the most informative aspects. In the context of decision trees, pruning involves removing some branches (subtrees) from the tree, leading to a smaller and simpler tree. In the involves removing some branches (subtrees) from the tree, leading to a smaller and simpler tree. In the context of DNN, pruning is used to reduce the number of neurons (units) or connections in the network, as shown in Figure 12.2.



same image as used earlier

Figure 12.2: Network pruning.

Reducing Complexity of Deep Learning Models

Traditional cloud-based DNN frameworks have too much memory overhead to be used on-device. For example, deep learning systems like PyTorch and TensorFlow require hundreds of megabytes of memory overhead when training models such as MobilenetV2, and the overhead scales as the number of training parameters increases.

Traditional cloud-based DNN frameworks have too much memory overhead to be used on-device. For example, deep learning systems like PyTorch and TensorFlow require hundreds of megabytes of memory overhead when training models such as MobilenetV2-w0.35, and the overhead scales as the number of training parameters increases.

Current research for lightweight DNNs mostly explores CNN architectures. Several bare-metal frameworks designed for running Neural Networks on MCUs by keeping computational overhead and memory footprint low also exist. Some examples include MNN, TVM, and TensorFlow Lite. However, they can only perform inference during forward passes and lack support for backpropagation. While these models are designed for edge deployment, their reduction in model weights and architectural connections led to reduced resource requirements for continuous learning.

The tradeoff between performance and model support is clear when adapting the most popular DNN systems. How do we adapt existing DNN models to resource-constrained settings while maintaining support for backpropagation and continuous learning? The latest research suggests algorithm and system codesign techniques that help reduce the resource consumption of ML training on edge devices. Utilizing techniques such as quantization-aware scaling (QAS), sparse updates, and other cutting-edge techniques, on-device learning is possible on embedded systems with a few hundred kilobytes of RAM without additional memory while maintaining high accuracy.

12.3.2 Modifying Optimization Processes

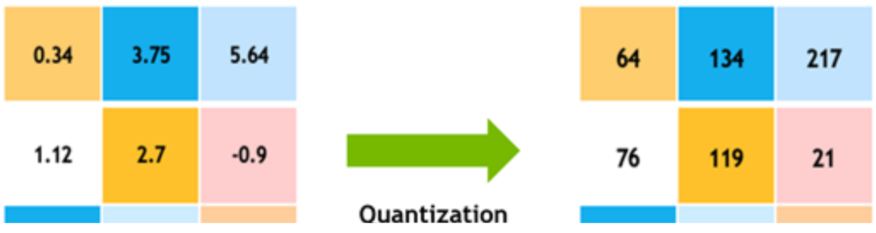
Choosing the right optimization strategy is important for DNN training on a device since this allows for finding a good local minimum. Since training occurs on a device, this strategy must also consider limited memory and power.

Quantization-Aware Scaling

Quantization is a common method for reducing the memory footprint of DNN training. Although this could introduce new errors, these errors can be mitigated by designing a model to characterize this statistical error. For example, models could use stochastic rounding or introduce the quantization error into the gradient updates.

A specific algorithmic technique is Quantization-Aware Scaling (QAS), which improves the performance of neural networks on low-precision hardware, such as edge devices, mobile devices, or TinyML systems, by adjusting the scale factors during the quantization process.

As we discussed in the Model Optimizations chapter, quantization is the process of mapping a continuous range of values to a discrete set of values. In the context of neural networks, quantization often involves reducing the precision of the weights and activations from 32-bit floating point to lower-precision formats such as 8-bit integers. This reduction in precision can significantly reduce the computational cost and memory footprint of the model, making it suitable for deployment on low-precision hardware. Figure 12.3 is an example of float-to-integer quantization.



-4.7
0.68
1.43
FP32
3
81
99
INT8

Figure 12.3: Float to integer qunatization. Credit: Nvidia.

However, the quantization process can also introduce quantization errors that can degrade the model's performance. Quantization-aware scaling is a technique that aims to minimize these errors by adjusting the scale factors used in the quantization process.

The QAS process involves two main steps:

- **Quantization-aware training:** In this step, the neural network is trained with quantization in mind, using simulated quantization to mimic the effects of quantization during the forward and backward passes. This allows the model to learn to compensate for the quantization errors and improve its performance on low-precision hardware. Refer to the QAT section in Model Optimizations for details.
- **Quantization and scaling:** After training, the model is quantized to a low-precision format, and the scale factors are adjusted to minimize the quantization errors. The scale factors are chosen based on the distribution of the weights and activations in the model and are adjusted to ensure that the quantized values are within the range of the low-precision format.

QAS is used to overcome the difficulties of optimizing models on tiny devices without needing hyperparameter tuning; QAS automatically scales tensor gradients with various bit precisions. This stabilizes the training process and matches the accuracy of floating-point precision.

Sparse Updates

Although QAS enables the optimization of a quantized model, it uses a large amount of memory, which is unrealistic for on-device training. So, spare updates are used to reduce the memory footprint of full backward computation. Instead of pruning weights for inference, sparse update prunes the gradient during backward propagation to update the model sparsely. In other words, sparse update skips computing gradients of less important layers and sub-tensors.

However, determining the optimal sparse update scheme given a constraining memory budget can be challenging due to the large search space. For example, the MCUNet model has 43 convolutional layers and a search space of approximately 1030. One technique to address this issue is contribution analysis. Contribution analysis measures the accuracy improvement from biases (updating the last few biases compared to only updating the classifier) and weights (updating the weight of one extra layer compared to only having a bias update). By trying to maximize these improvements, contribution analysis automatically derives an optimal sparse update scheme for enabling on-device training.

Layer-Wise Training

Other methods besides quantization can help optimize routines. One such method is layer-wise training. A significant memory consumer of DNN training is end-to-end backpropagation, which requires all intermediate feature maps to be stored so the model can calculate gradients. An alternative to this approach that reduces the memory footprint of DNN training is sequential layer-by-layer training (T. Chen et al. 2016). Instead of training end-to-end, training a single layer at a time helps avoid having to store intermediate feature maps.

Chen, Tianqi, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. "Training Deep Nets with Sublinear Memory Cost." *ArXiv Preprint* abs/1604.06174. https://arxiv.org/abs/1604.06174.

Trading Computation for Memory

The strategy of trading computation for memory involves releasing some of the memory being used to store intermediate results. Instead, these results can be recomputed as needed. Reducing memory in exchange for more computation is shown to reduce the memory footprint of DNN training to fit into almost any budget while also minimizing computational cost (Gruslys et al. 2016).

Gruslys, Audrunas, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. "Memory-Efficient Backpropagation Through Time." In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, edited by Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, 4125–33. https://proceedings.neurips.cc/paper/2016/hash/a501bebf79d57Abstract.html

12.3.3 Developing New Data Representations

The dimensionality and volume of the training data can significantly impact on-device adaptation. So, another technique for adapting models onto resource-constrained devices is to represent datasets more efficiently.

Data Compression

The goal of data compression is to reach high accuracies while limiting the amount of training data. One method to achieve this is prioritizing sample complexity: the amount of training data required for the algorithm to reach a target accuracy (Dhar et al. 2021).

Dhar, Sauptik, Junyao Guo, Jiayi (Jason) Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. 2021. "A Survey of on-Device Machine Learning: An Algorithms and Learning Theory Perspective." *ACM Transactions on Internet of Things* 2 (3): 1–49. https://doi.org/10.1145/3450494.

Other more common methods of data compression focus on reducing the dimensionality and the volume of the training data. For example, an approach could take advantage of matrix sparsity to reduce the memory footprint of storing training data. Training data can be transformed into a lower-dimensional embedding and factorized into a dictionary matrix multiplied by a block-sparse coefficient matrix (Darvish Rouhani, Mirhoseini, and Koushanfar 2017). Another example could involve representing words from a large language training dataset in a more compressed vector format (Li et al. 2016).

Darvish Rouhani, Bita, Azalia Mirhoseini, and Farinaz Koushanfar. 2017. "TinyDL: Just-in-time Deep Learning Solution for Constrained Embedded Systems." In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 1–4. IEEE. https://doi.org/10.1109/iscas.2017.8050343.

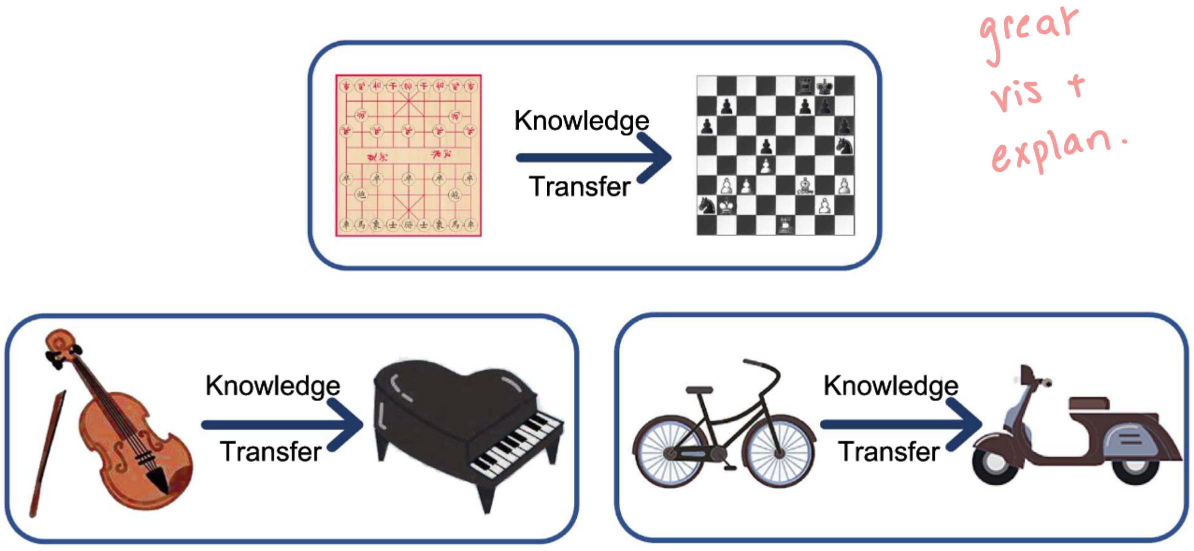
Li, Xiang, Tao Qin, Jian Yang, and Tie-Yan Liu. 2016. "LightRNN: Memory and Computation-Efficient Recurrent Neural Networks." In *Advances in Neural Information Processing Systems 29:*

12.4 Transfer Learning

Transfer learning is an ML technique in which a model developed for a particular task is reused as the starting point for a model on a second task. In the context of on-device AI, transfer learning allows us to leverage pre-trained models that have already learned useful representations from large datasets and finetune them for specific tasks using smaller datasets directly on the device. This can significantly reduce the computational resources and time required for training models from scratch.

Figure 12.4 includes some intuitive examples of transfer learning from the real world. For instance, if you can ride a bicycle, you know how to balance yourself on two-wheel vehicles. Then, it would be easier for you to learn how to ride a motorcycle than it would be for someone who cannot ride a bicycle.

Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain, edited by Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, 4385–93. https://proceedings.neurips.cc/paper/2016/hash/c3e4035af2a1cc Abstract.html.



great
vis +
explan.

Figure 12.4: Transferring knowledge between tasks. Credit: Zhuang et al. (2021).

Zhuang, Fuzhen, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2021. "A Comprehensive Survey on Transfer Learning." *Proc. IEEE* 109 (1): 43–76. https://doi.org/10.1109/jproc.2020.3004555.

Let's take the example of a smart sensor application that uses on-device AI to recognize objects in images captured by the device. Traditionally, this would require sending the image data to a server, where a large neural network model processes the data and sends back the results. With on-device AI, the model is stored and runs directly on-device, eliminating the need to send data to a server.

If we want to customize the model for the on-device characteristics, training a neural network model from scratch on the device would be impractical due to the limited computational resources and battery life. This is where transfer learning comes in. Instead of training a model from scratch, we can take a pre-trained model, such as a convolutional neural network (CNN) or a transformer network trained on a large dataset of images, and finetune it for our specific object recognition task. This finetuning can be done directly on the device using a smaller dataset of images relevant to the task. By leveraging the pre-trained model, we can reduce the computational resources and time required for training while still achieving high accuracy for the object recognition task.

Transfer learning is important in making on-device AI practical by allowing us to leverage pre-trained models and finetune them for specific tasks, thereby reducing the computational resources and time required for training. The combination of on-device AI and transfer learning opens up new possibilities for AI applications that are more privacy-conscious and responsive to user needs.

Transfer learning has revolutionized the way models are developed and deployed, both in the cloud and at the edge. Transfer learning is being used in the real world. One such example is the use of transfer learning to develop AI models that can detect and diagnose diseases from medical images, such as X-rays, MRI scans, and CT scans. For example, researchers at Stanford University developed a transfer learning model that can detect cancer in skin images with an accuracy of 97% (Esteva et al. 2017). This model was pre-trained on 1.28 million images to classify a broad range of objects and then specialized for cancer detection by training on a dermatologist-curated dataset of skin images.

Esteva, Andre, Brett Kuprel, Roberto A. Novoa, Justin Ko, Susan M. Swetter, Helen M. Blau, and Sebastian Thrun. 2017. "Dermatologist-Level Classification of Skin Cancer with Deep Neural Networks." *Nature* 542 (7639): 115–18. https://doi.org/10.1038/nature21056.

Implementation in production scenarios can be broadly categorized into two stages: pre-deployment and post-deployment.

12.4.1 Pre-Deployment Specialization

In the pre-deployment stage, transfer learning acts as a catalyst to expedite the development process. Here's how it typically works: Imagine we are creating a system to recognize different breeds of dogs. Rather than starting from scratch, we can utilize a pre-trained model that has already mastered the broader task of recognizing animals in images.

This pre-trained model serves as a solid foundation and contains a wealth of knowledge acquired from extensive data. We then finetune this model using a specialized dataset containing images of various dog breeds. This finetuning process tailors the model to our specific need — precisely identifying dog breeds. Once finetuned and validated to meet performance criteria, this specialized model is then ready for deployment.

Here's how it works in practice:

- **Start with a Pre-Trained Model:** Begin by selecting a model that has already been trained on a comprehensive dataset, usually related to a general task. This model serves as the foundation for the task at hand.
- **Finetuning:** The pre-trained model is then finetuned on a smaller, more specialized dataset specific to the desired task. This step allows the model to adapt and specialize its knowledge to the specific requirements of the application.
- **Validation:** After finetuning, the model is validated to ensure it meets the performance criteria for the specialized task.
- **Deployment:** Once validated, the specialized model is then deployed into the production environment.

This method significantly reduces the time and computational resources required to train a model from scratch (Pan and Yang 2010). By adopting transfer learning, embedded systems can achieve high accuracy on specialized tasks without the need to gather extensive data or expend significant computational resources on training from the ground up.

Pan, Sinno Jialin, and Qiang Yang. 2010. "A Survey on Transfer Learning." *IEEE Trans. Knowl. Data Eng.* 22 (10): 1345–59. https://doi.org/10.1109/tkde.2009.191.

12.4.2 Post-Deployment Adaptation

Deployment to a device need not mark the culmination of an ML model's educational trajectory. With the advent of transfer learning, we open the doors to the deployment of adaptive ML models in real-world scenarios, catering to users' personalized needs.

Consider a real-world application where a parent wishes to identify their child in a collection of images from a school event on their smartphone. In this scenario, the parent is faced with the challenge of locating their child amidst images of many other children. Transfer learning can be employed here to finetune an embedded system's model to this unique and specialized task. Initially, the system might use a generic model trained to recognize faces in images. However, with transfer learning, the system can adapt this model to recognize the specific features of the user's child.

Here's how it works:

1. **Data Collection:** The embedded system gathers images that include the child, ideally with the parent's input to ensure accuracy and relevance. This can be done directly on the device, maintaining the user's data privacy.
2. **Model Finetuning:** The pre-existing face recognition model, which has been trained on a large and diverse dataset, is then finetuned using the newly collected images of the child. This process adapts the model to recognize the child's specific facial features, distinguishing them from other children in the images.
3. **Validation:** The refined model is then validated to ensure it accurately recognizes the child in various images. This can involve the parent verifying the model's performance and providing feedback for further improvements.
4. **Deployment:** Once validated, the adapted model is deployed on the device, enabling the parent to easily identify their child in images without having to sift through them manually.

This on-the-fly customization enhances the model's efficacy for the individual user, ensuring that they benefit from ML personalization. This is, in part, how iPhotos or Google Photos works when they ask us to recognize a face, and then, based on that information, they index all the photos by that face. Because the learning and adaptation occur on the device itself, there are no risks to personal privacy. The parent's images are not uploaded to a cloud server or shared with third parties, protecting the family's privacy while still reaping the benefits of a personalized ML model. This approach represents a significant step forward in the quest to provide users with tailored ML solutions that respect and uphold their privacy.

this bullet-pt system is great!

12.4.3 Benefits

deficits section?

Transfer learning has become an important technique in ML and artificial intelligence, and it is particularly valuable for several reasons.

1. **Data Scarcity:** In many real-world scenarios, acquiring a sufficiently large labeled dataset to train an ML model from scratch is challenging. Transfer learning mitigates this issue by allowing the use of pre-trained models that have already learned valuable features from a vast dataset.
2. **Computational Expense:** Training a model from scratch requires significant computational resources and time, especially for complex models like deep neural networks. By using transfer learning, we can leverage the computation that has already been done during the training of the source model, thereby saving both time and computational power.
3. **Limited Annotated Data:** For some specific tasks, there might be ample raw data available, but the process of labeling that data for supervised learning can be costly and time-consuming. Transfer learning enables us to utilize pre-trained models that have been trained on a related task with labeled data, hence requiring less annotated data for the new task.

There are advantages to reusing the features:

1. **Hierarchical Feature Learning:** Deep learning models, particularly Convolutional Neural Networks (CNNs), can learn hierarchical features. Lower layers typically learn generic features like edges and shapes, while higher layers learn more complex and task-specific features. Transfer learning allows us to reuse the generic features learned by a model and finetune the higher layers for our specific task.
2. **Boosting Performance:** Transfer learning has been proven to boost the performance of models on tasks with limited data. The knowledge gained from the source task can provide a valuable starting point and lead to faster convergence and improved accuracy on the target task.

Exercise 12.1: Transfer Learning

12.4.4 Core Concepts

Understanding the core concepts of transfer learning is essential for effectively utilizing this powerful approach in ML. Here, we'll break down some of the main principles and components that underlie the process of transfer learning.

Source and Target Tasks

In transfer learning, there are two main tasks involved: the source task and the target task. The source task is the task for which the model has already been trained and has learned valuable information. The target task is the new task we want the model to perform. The goal of transfer learning is to leverage the knowledge gained from the source task to improve performance on the target task.

Suppose we have a model trained to recognize various fruits in images (source task), and we want to create a new model to recognize different vegetables in images (target task). In that case, we can use transfer learning to leverage the knowledge gained during the fruit recognition task to improve the performance of the vegetable recognition model.

Representation Transfer

Representation transfer is about transferring the learned representations (features) from the source task to the target task. There are three main types of representation transfer:

- **Instance Transfer:** This involves reusing the data instances from the source task in the target task.
- **Feature-Representation Transfer:** This involves transferring the learned feature representations from the source task to the target task.
- **Parameter Transfer:** This involves transferring the model's learned parameters (weights) from the source task to the target task.

In natural language processing, a model trained to understand the syntax and grammar of a language (source task) can have its learned representations transferred to a new model designed to perform (source task) can have its learned representations transferred to a new model designed to perform sentiment analysis (target task).

Finetuning

Finetuning is the process of adjusting the parameters of a pre-trained model to adapt it to the target task. This typically involves updating the weights of the model's layers, especially the last few layers, to make the model more relevant for the new task. In image classification, a model pre-trained on a general dataset like ImageNet (source task) can be finetuned by adjusting the weights of its layers to perform well on a specific classification task, like recognizing specific animal species (target task).

Feature Extractions

spacing?

Feature extraction involves using a pre-trained model as a fixed feature extractor, where the output of the model's intermediate layers is used as features for the target task. This approach is particularly useful when the target task has a small dataset, as the pre-trained model's learned features can significantly enhance performance. In medical image analysis, a model pre-trained on a large dataset of general medical images (source task) can be used as a feature extractor to provide valuable features for a new model designed to recognize specific types of tumors in X-ray images (target task).

12.4.5 Types of Transfer Learning

Transfer learning can be classified into three main types based on the nature of the source and target tasks and data. Let's explore each type in detail:

Inductive Transfer Learning

In inductive transfer learning, the goal is to learn the target predictive function with the help of source data. It typically involves finetuning a pre-trained model on the target task with available labeled data. A common example of inductive transfer learning is image classification tasks. For instance, a model pre-trained on the ImageNet dataset (source task) can be finetuned to classify specific types of birds (target task) using a smaller labeled dataset of bird images.

Transductive Transfer Learning

Transductive transfer learning involves using source and target data, but only the source task. The main aim is to transfer knowledge from the source domain to the target domain, even though the tasks remain the same. Sentiment analysis for different languages can serve as an example of transductive transfer learning. A model trained to perform sentiment analysis in English (source task) can be adapted to perform sentiment analysis in another language, like French (target task), by leveraging parallel datasets of English and French sentences with the same sentiments.

Unsupervised Transfer Learning

Unsupervised transfer learning is used when the source and target tasks are related, but there is no labeled data available for the target task. The goal is to leverage the knowledge gained from the source task to improve performance on the target task, even without labeled data. An example of unsupervised transfer learning is topic modeling in text data. A model trained to extract topics from news articles (source task) can be adapted to extract topics from social media posts (target task) without needing labeled data for the social media posts.

Comparison and Tradeoffs

By leveraging these different types of transfer learning, practitioners can choose the approach that best fits the nature of their tasks and available data, ultimately leading to more effective and efficient ML models. So, in summary:

- **Inductive:** different source and target tasks, different domains
- **Transductive:** different source and target tasks, same domain
- **Unsupervised:** unlabeled source data, transfers feature representations

great!

Table 12.1 presents a matrix that outlines in a bit more detail the similarities and differences between the types of transfer learning:

Table 12.1: Comparison of transfer learning types.

	Inductive Transfer Learning	Transductive Transfer Learning	Unsupervised Transfer Learning
Labeled Data for Target Task	Required	Not Required	Not Required
Source Task	Can be different	Same	Same or Different
Target Task	Can be different	Same	Can be different
Objective	Improve target task performance with source data	Transfer knowledge from source to target domain	Leverage source task to improve target task performance without labeled data
Example	ImageNet to bird classification	Sentiment analysis in different languages	Topic modeling for different text data

amazing!

12.4.6 Constraints and Considerations

When engaging in transfer learning, there are several factors that must be considered to ensure successful knowledge transfer and model performance. Here's a breakdown of some key factors:

Domain Similarity

Domain similarity refers to how closely related the source and target domains are. The more similar the domains, the more likely the transfer learning will be successful. Transferring knowledge from a model trained on images of outdoor scenes (source domain) to a new task that involves recognizing objects in indoor scenes (target domain) might be more successful than transferring knowledge from outdoor scenes to a task involving text analysis, as the domains (images vs. text) are quite different.

Task Similarity

Task similarity refers to how closely related the source and target tasks are. Similar tasks are likely to benefit more from transfer learning. A model trained to recognize different breeds of dogs (source task) can be more easily adapted to recognize different breeds of cats (target task) than it can be adapted to perform a completely different task like language translation.

Data Quality and Quantity

The quality and quantity of data available for the target task can significantly impact the success of transfer learning. More high-quality data can result in better model performance. Suppose we have a large dataset with clear, well-labeled images to recognize specific bird species. In that case, the transfer learning process will likely be more successful than if we have a small, noisy dataset.

Feature Space Overlap

Feature space overlap refers to how well the features learned by the source model align with the features needed for the target task. Greater overlap can lead to more successful transfer learning. A model trained on high-resolution images (source task) may not transfer well to a target task that involves low-resolution images, as the feature space (high-res vs. low-res) is different.

Model Complexity

The complexity of the source model can also impact the success of transfer learning. Sometimes, a simpler model might transfer better than a complex one, as it is less likely to overfit the source task. For example, a simple convolutional neural network (CNN) model trained on image data (source task) may transfer more successfully to a new image classification task (target task) than a complex CNN with many layers, as the simpler model is less likely to overfit the source task.

By considering these factors, ML practitioners can make informed decisions about when and how to utilize transfer learning, ultimately leading to more successful model performance on the target task. The success of transfer learning hinges on the degree of similarity between the source and target domains. Overfitting is risky, especially when finetuning occurs on a limited dataset. On the computational front, certain pre-trained models, owing to their size, might not comfortably fit into the memory constraints of some devices or may run prohibitively slowly. Over time, as data evolves, there is potential for model drift, indicating the need for periodic re-training or ongoing adaptation.

Learn more about transfer learning in Video 12.1 below.

Video 12.1: Transfer Learning

12.5 Federated Machine Learning

Federated Learning Overview → make header?

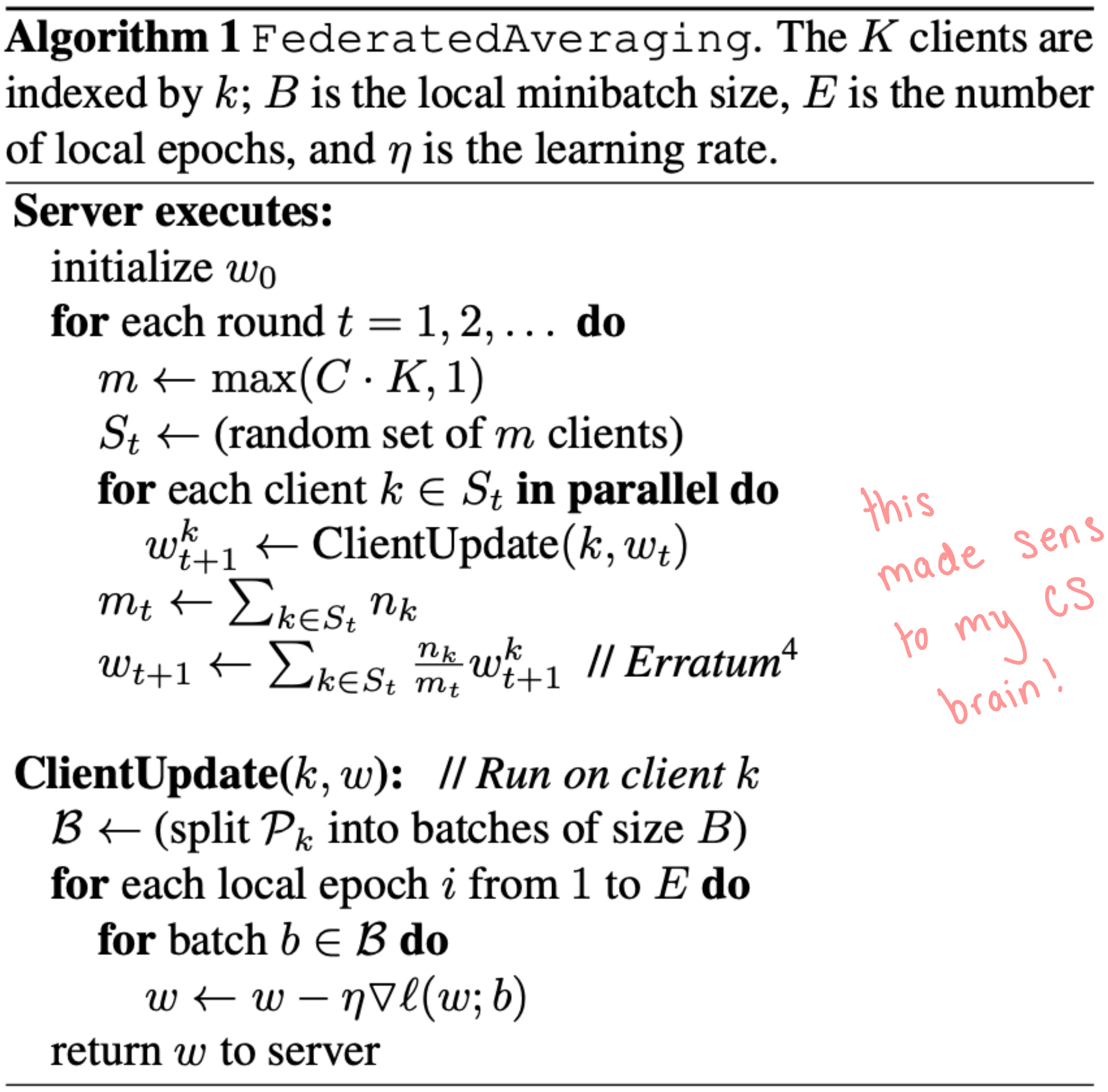
The modern internet is full of large networks of connected devices. Whether it’s cell phones, thermostats, smart speakers, or other IOT products, countless edge devices are a goldmine for hyper-personalized, rich data. However, with that rich data comes an assortment of problems with information transfer and privacy. Constructing a training dataset in the cloud from these devices would involve high volumes of bandwidth, cost-efficient data transfer, and violation of users’ privacy.

Federated learning offers a solution to these problems: train models partially on the edge devices and only communicate model updates to the cloud. In 2016, a team from Google designed architecture for federated learning that attempts to address these problems.

In their initial paper, Google outlines a principle federated learning algorithm called FederatedAveraging, which is shown in Figure 12.5. Specifically, FederatedAveraging performs stochastic gradient descent (SGD) over several different edge devices. In this process, each device calculates a gradient $g_k = \nabla F_k(w_t)$ which is then applied to update the server-side weights as (with η as learning rate across k clients):

$$w_{t+1} \rightarrow w_t - \eta \sum_{k=1}^{K} \frac{n_k}{n} g_k$$

This summarizes the basic algorithm for federated learning on the right. For each round of training, the server takes a random set of client devices and calls each client to train on its local batch using the most server takes a random set of client devices and calls each client to train on its local batch using the most recent server-side weights. Those weights are then returned to the server, where they are collected individually and averaged to update the global model weights.



Algorithm 1 FederatedAveraging. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

```
Server executes:
  initialize $w_0$
  for each round $t = 1, 2, \dots$ do
    $m \leftarrow \max(C \cdot K, 1)$
    $S_t \leftarrow$ (random set of $m$ clients)
    for each client $k \in S_t$ in parallel do
      $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
    $m_t \leftarrow \sum_{k \in S_t} n_k$
    $w_{t+1} \leftarrow \sum_{k \in S_t} \frac{n_k}{m_t} w_{t+1}^k$   // Erratum[4]

ClientUpdate($k, w$):   // Run on client k
  $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size $B$)
  for each local epoch $i$ from 1 to $E$ do
    for batch $b \in \mathcal{B}$ do
      $w \leftarrow w - \eta \nabla \ell(w; b)$
  return $w$ to server
```

this made sense to my CS brain!

Figure 12.5: Google's Proposed FederatedAverage Algorithm. Credit: McMahan et al. (2017).

With this proposed structure, there are a few key vectors for further optimizing federated learning. We will outline each in the following subsections.

Video 12.2 gives an overview of federated learning.

Video 12.2: Transfer Learning

getting playback error on this video

12.5.1 Communication Efficiency

One of the key bottlenecks in federated learning is communication. Every time a client trains the model, they must communicate their updates back to the server. Similarly, once the server has averaged all the updates, it must send them back to the client. This incurs huge bandwidth and resource costs on large networks of millions of devices. As the field of federated learning advances, a few optimizations have been developed to minimize this communication. To address the footprint of the model, researchers have developed model compression techniques. In the client-server protocol, federated learning can also minimize communication through the selective sharing of updates on clients. Finally, efficient aggregation techniques can also streamline the communication process.

12.5.2 Model Compression

In standard federated learning, the server communicates the entire model to each client, and then the client sends back all of the updated weights. This means that the easiest way to reduce the client's memory and communication footprint is to minimize the size of the model needed to be communicated. We can employ all of the previously discussed model optimization strategies to do this.

In 2022, another team at Google proposed that each client communicates via a compressed format and decompresses the model on the fly for training (Yang et al. 2023), allocating and deallocating the full memory for the model only for a short period while training. The model is compressed through a range of

Yang, Tien-Ju, Yonghui Xiao, Giovanni Motta, Françoise Beaufays, Rajiv Mathews, and Mingqing Chen. 2023.

memory for the model only for a short period while training. The model is compressed through a range of various quantization strategies elaborated upon in their paper. Meanwhile, the server can update the uncompressed model by decompressing and applying updates as they come in.

"Online Model Compression for Federated Learning with Large Models." In *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 1–5. IEEE; IEEE. https://doi.org/10.1109/icassp49357.2023.10097124.

12.5.3 Selective Update Sharing

There are many methods for selectively sharing updates. The general principle is that reducing the portion of the model that the clients are training on the edge reduces the memory necessary for training and the size of communication to the server. In basic federated learning, the client trains the entire model. This means that when a client sends an update to the server, it has gradients for every weight in the network.

However, we cannot just reduce communication by sending pieces of those gradients from each client to the server because the gradients are part of an entire update required to improve the model. Instead, you need to architecturally design the model such that each client trains only a small portion of the broader model, reducing the total communication while still gaining the benefit of training on client data. A paper (Shi and Radu 2022) from the University of Sheffield applies this concept to a CNN by splitting the global model into two parts: an upper and a lower part, as shown in Z. Chen and Xu (2023).

Shi, Hongrui, and Valentin Radu. 2022. "Data Selection for Efficient Model Update in Federated Learning." In *Proceedings of the 2nd European Workshop on Machine Learning and Systems*, 72–78. ACM. https://doi.org/10.1145/3517207.3526980.

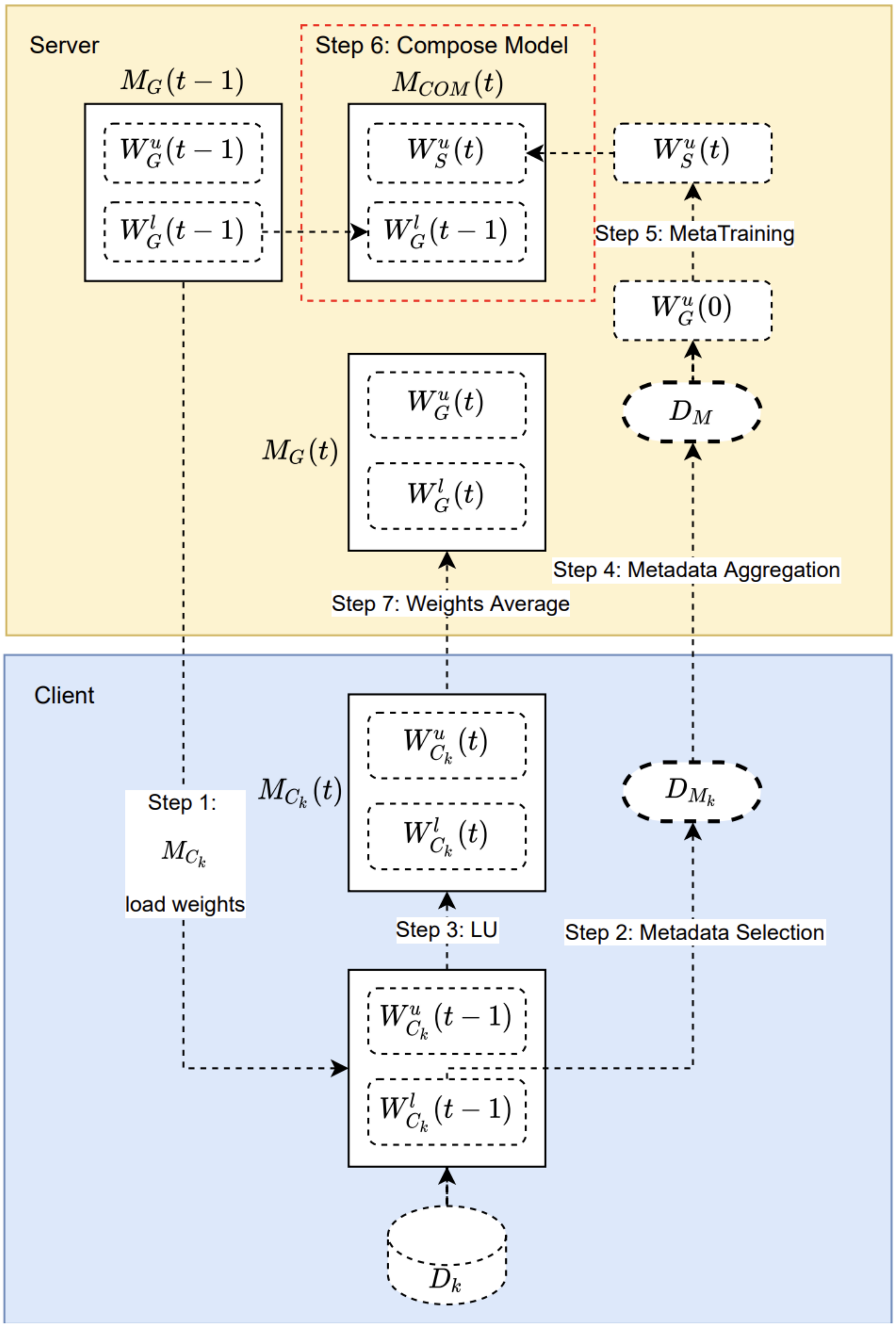


Figure 12.6: Split model architecture for selective sharing. Credit: Shi et al., (2022).

this did not make sense to my brain ... or cut?
to explain

The lower part is designed to focus on generic features in the dataset, while the upper part, trained on those generic features, is designed to be more sensitive to the activation maps. This means that the lower part of the model is trained through standard federated averaging across all of the clients. Meanwhile, the upper part of the model is trained entirely on the server side from the activation maps generated by the clients. This approach drastically reduces communication for the model while still making the network robust to various types of input found in the data on the client devices.

12.5.4 Optimized Aggregation

In addition to reducing the communication overhead, optimizing the aggregation function can improve model training speed and accuracy in certain federated learning use cases. While the standard for aggregation is just averaging, various other approaches can improve model efficiency, accuracy, and security. One alternative is clipped averaging, which clips the model updates within a specific range. Another strategy to preserve security is differential privacy average aggregation. This approach integrates differential privacy into the aggregations tep to protect client identities. Each client adds a layer of random noise to their updates before communicating to the server. The server then updates the server with the noisy updates, meaning that the amount of noise needs to be tuned carefully to balance privacy and accuracy.

In addition to security-enhancing aggregation methods, there are several modifications to the aggregation methods that can improve training speed and performance by adding client metadata along with the weight updates. Momentum aggregation is a technique that helps address the convergence problem. In federated learning, client data can be extremely heterogeneous depending on the different environments in which the devices are used. That means that many models with heterogeneous data may need help to converge. Each client stores a momentum term locally, which tracks the pace of change over several updates. With clients communicating this momentum, the server can factor in the rate of change of each update when changing the global model to accelerate convergence. Similarly, weighted aggregation can factor in the client performance or other parameters like device type or network connection strength to adjust the weight with which the server should incorporate the model updates. Further description of specific aggregation algorithms is described by Moshawrab et al. (2023).

Moshawrab, Mohammad, Mehdi Adda, Abdenour Bouzouane, Hussein Ibrahim, and Ali Raad. 2023. "Reviewing Federated Learning Aggregation Algorithms; Strategies, Contributions, Limitations and Future Perspectives." *Electronics* 12 (10): 2287. https://doi.org/10.3390/electronics12102287.

12.5.5 Handling non-IID Data

explain more?

When using federated learning to train a model across many client devices, it is convenient to consider the data to be independent and identically distributed (IID) across all clients. When data is IID, the model will converge faster and perform better because each local update on any given client is more representative of the broader dataset. This makes aggregation straightforward, as you can directly average all clients. However, this differs from how data often appears in the real world. Consider a few of the following ways in which data may be non-IID:

- If you are learning on a set of health-monitor devices, different device models could mean different sensor qualities and properties. This means that low-quality sensors and devices may produce data, and therefore, model updates distinctly different than high-quality ones
- A smart keyboard trained to perform autocorrect. If you have a disproportionate amount of devices from a certain region, the slang, sentence structure, or even language they were using could skew more model updates towards a certain style of typing
- If you have wildlife sensors in remote areas, connectivity may not be equally distributed, causing some clients in certain regions to be unable to send more model updates than others. If those regions have different wildlife activity from certain species, that could skew the updates toward those animals

not a full sentence + not gram. aligned w/ other 2 bullets

There are a few approaches to addressing non-IID data in federated learning. One approach would be to change the aggregation algorithm. If you use a weighted aggregation algorithm, you can adjust based on different client properties like region, sensor properties, or connectivity (Zhao et al. 2018).

Zhao, Yue, Meng Li, Liangzhen Lai, Naveen Suda, Damon Civin, and Vikas Chandra. 2018. "Federated Learning with Non-Iid Data." *ArXiv Preprint* abs/1806.00582. https://arxiv.org/abs/1806.00582.

12.5.6 Client Selection

Considering all of the factors influencing the efficacy of federated learning, like IID data and communication, client selection is a key component to ensuring a system trains well. Selecting the wrong clients can skew the dataset, resulting in non-IID data. Similarly, choosing clients randomly with bad network connections can slow down communication. Therefore, several key characteristics must be considered when selecting the right subset of clients.

When selecting clients, there are three main components to consider: data heterogeneity, resource allocation, and communication cost. We can select clients on the previously proposed metrics in the non-IID section to address data heterogeneity. In federated learning, all devices may have different amounts of computing, resulting in some being more inefficient at training than others. When selecting a subset of clients for training, one must consider a balance of data heterogeneity and available resources. In an ideal scenario, you can always select the subset of clients with the greatest resources. However, this may skew your dataset, so a balance must be struck. Communication differences add another layer; you want to avoid being bottlenecked by waiting for devices with poor connections to transmit all their updates. Therefore, you must also consider choosing a subset of diverse yet well-connected devices.

12.5.7 An Example of Deployed Federated Learning: G board

A primary example of a deployed federated learning system is Google's Keyboard, Gboard, for Android devices. In implementing federated learning for the keyboard, Google focused on employing differential privacy techniques to protect the user's data and identity. Gboard leverages language models for several key features, such as Next Word Prediction (NWP), Smart Compose (SC), and On-The-Fly rescoring (OTF) (Xu et al. 2023), as shown in Figure 12.7.

Xu, Zheng, Yanxiang Zhang, Galen Andrew, Christopher A Choquette-Choo, Peter Kairouz, H Brendan McMahan, Jesse Rosenstock, and Yuanbo Zhang. 2023. "Federated Learning of Gboard Language Models with Differential Privacy." *ArXiv Preprint* abs/2305.18465. https://arxiv.org/abs/2305.18465.

NWP will anticipate the next word the user tries to type based on the previous one. SC gives inline suggestions to speed up the typing based on each character. OTF will re-rank the proposed next words based on the active typing process. All three of these models need to run quickly on the edge, and federated learning can accelerate training on the users' data. However, uploading every word a user typed to the cloud for training would be a massive privacy violation. Therefore, federated learning emphasizes differential privacy, which protects the user while enabling a better user experience.

✓

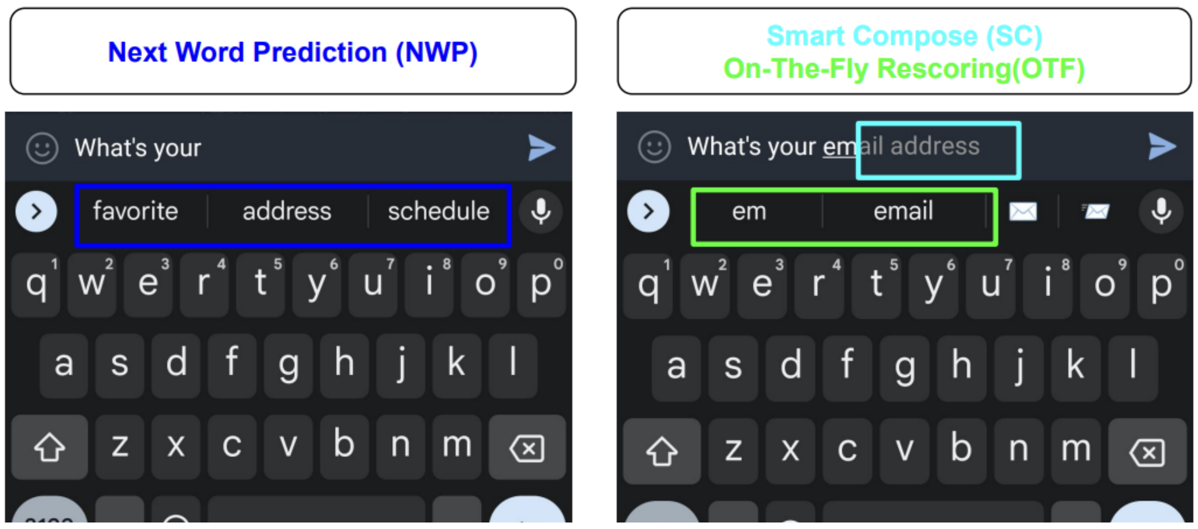


Figure 12.7: Google G Board Features. Credit: Zheng et al., (2023).

To accomplish this goal, Google employed its algorithm DP-FTRL, which provides a formal guarantee that trained models will not memorize specific user data or identities. The algorithm system design is shown in Figure 12.8. DP-FTRL, combined with secure aggregation, encrypts model updates and provides an optimal balance of privacy and utility. Furthermore, adaptive clipping is applied in the aggregation process to limit the impact of individual users on the global model (step 3 in Figure 12.8). By combining all these techniques, Google can continuously refine its keyboard while preserving user privacy in a formally provable way.

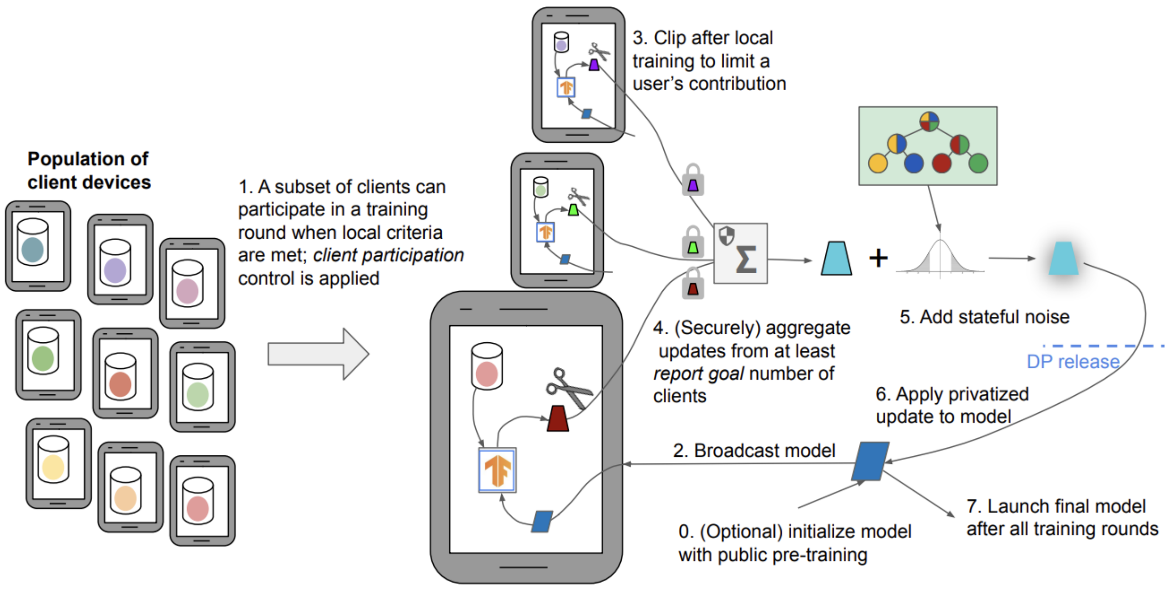

Population of client devices
1. A subset of clients can participate in a training round when local criteria are met; *client participation* control is applied
3. Clip after local training to limit a user's contribution
4. (Securely) aggregate updates from at least *report goal* number of clients
5. Add stateful noise
DP release
6. Apply privatized update to model
2. Broadcast model
0. (Optional) initialize model with public pre-training
7. Launch final model after all training rounds

Figure 12.8: Differential Privacy in G Board. Credit: Zheng et al., (2023).

Exercise 12.2: Federated Learning - Text Generation

Exercise 12.3: Federated Learning - Image Classification

12.5.8 Benchmarking for Federated Learning: MedPerf

One of the richest examples of data on the edge is medical devices. These devices store some of the most personal data on users but offer huge advances in personalized treatment and better accuracy in medical AI. Given these two factors, medical devices are the perfect use case for federated learning. MedPerf is an open-source platform used to benchmark models using federated evaluation (Karargyris et al. 2023). Instead of just training models via federated learning, MedPerf takes the model to edge devices to test it against personalized data while preserving privacy. In this way, a benchmark committee can evaluate various models in the real world on edge devices while still preserving patient anonymity.

Karargyris, Alexandros, Renato Umeton, Micah J Sheller, Alejandro Aristizabal, Johnu George, Anna Wuest, Sarthak Pati, et al. 2023. "Federated Benchmarking of Medical Artificial Intelligence with MedPerf." *Nature Machine Intelligence* 5 (7): 799–810. https://doi.org/10.1038/s42256-023-00652-2.

12.6 Security Concerns

Performing ML model training and adaptation on end-user devices also introduces security risks that must be addressed. Some key security concerns include:

- **Exposure of private data:** Training data may be leaked or stolen from devices
- **Data poisoning:** Adversaries can manipulate training data to degrade model performance
- **Model extraction:** Attackers may attempt to steal trained model parameters
- **Membership inference:** Models may reveal the participation of specific users' data
- **Evasion attacks:** Specially crafted inputs can cause misclassification

Any system that performs learning on-device introduces security concerns, as it may expose vulnerabilities in larger-scale models. Numerous security risks are associated with any ML model, but these risks have specific consequences for on-device learning. Fortunately, there are methods to mitigate these risks and improve the real-world performance of on-device learning.

12.6.1 Data Poisoning

On-device ML introduces unique data security challenges compared to traditional cloud-based training. In particular, data poisoning attacks pose a serious threat during on-device learning. Adversaries can manipulate training data to degrade model performance when deployed.

Several data poisoning attack techniques exist:

- **Label Flipping:** It involves applying incorrect labels to samples. For instance, in image classification, cat photos may be labeled as dogs to confuse the model. Flipping even 10% of labels can have significant consequences on the model.
- **Data Insertion:** It introduces fake or distorted inputs into the training set. This could include pixelated images, noisy audio, or garbled text.
- **Logic Corruption:** This alters the underlying [patterns](https://www.worldscientific.com/doi/10.1142/S0218001414600027) in data to mislead the model. In sentiment analysis, highly negative reviews may be marked positive through this technique. For this reason, recent surveys have shown that many companies are more afraid of data poisoning than other adversarial ML concerns.

What makes data poisoning alarming is how it exploits the discrepancy between curated datasets and live training data. Consider a cat photo dataset collected from the internet. In the weeks later, when this data trains a model on-device, new cat photos on the web differ significantly.

With data poisoning, attackers purchase domains and upload content that influences a portion of the training data. Even small data changes significantly impact the model's learned behavior. Consequently, poisoning can instill racist, sexist, or other harmful biases if unchecked.

Microsoft Tay was a chatbot launched by Microsoft in 2016. It was designed to learn from its interactions with users on social media platforms like Twitter. Unfortunately, Microsoft Tay became a prime example of data poisoning in ML models. Within 24 hours of its launch, Microsoft had to take Tay offline because it had started producing offensive and inappropriate messages, including hate speech and racist comments. This occurred because some users on social media intentionally fed Tay with harmful and offensive input, which the chatbot then learned from and incorporated into its responses.

This incident is a clear example of data poisoning because malicious actors intentionally manipulated the data used to train and inform the chatbot's responses. The data poisoning resulted in the chatbot adopting harmful biases and producing output that its developers did not intend. It demonstrates how even small amounts of maliciously crafted data can significantly impact the behavior of ML models and highlights the importance of implementing robust data filtering and validation mechanisms to prevent such incidents from occurring.

Such biases could have dangerous real-world impacts. Rigorous data validation, anomaly detection, and tracking of data provenance are critical defensive measures. Adopting frameworks like Five Safes ensures models are trained on high-quality, representative data (Desai et al. 2016).

Desai, Tanvi, Felix Ritchie, Richard Welpton, et al. 2016. "Five Safes: Designing Data Access for Research." *Economics Working Paper Series* 1601: 28.

Data poisoning is a pressing concern for secure on-device learning since data at the endpoint cannot be easily monitored in real-time. If models are allowed to adapt on their own, then we run the risk of the device acting maliciously. However, continued research in adversarial ML aims to develop robust solutions to detect and mitigate such data attacks.

12.6.2 Adversarial Attacks

During the training phase, attackers might inject malicious data into the training dataset, which can subtly alter the model's behavior. For example, an attacker could add images of cats labeled as dogs to a dataset used to train an image classification model. If done cleverly, the model's accuracy might not significantly drop, and the attack could be noticed. The model would then incorrectly classify some cats as dogs, which could have consequences depending on the application.

In an embedded security camera system, for instance, this could allow an intruder to avoid detection by wearing a specific pattern that the model has been tricked into classifying as non-threatening.

During the inference phase, attackers can use adversarial examples to fool the model. Adversarial examples are inputs that have been slightly altered in a way that causes the model to make incorrect predictions. For instance, an attacker might add a small amount of noise to an image in a way that causes a face recognition system to misidentify a person. These attacks can be particularly concerning in applications where safety is at stake, such as autonomous vehicles. In the example you mentioned, the researchers were able to cause a traffic sign recognition system to misclassify a stop sign as a speed sign. This type of misclassification could lead to accidents if it occurred in a real-world autonomous driving system.

To mitigate these risks, several defenses can be employed:

- **Data Validation and Sanitization:** Before incorporating new data into the training dataset, it should be thoroughly validated and sanitized to ensure it is not malicious.
- **Adversarial Training:** The model can be trained on adversarial examples to make it more robust to these types of attacks.
- **Input Validation:** During inference, inputs should be validated to ensure they have not been manipulated to create adversarial examples.
- **Regular Auditing and Monitoring:** Regularly auditing and monitoring the model's behavior can help detect and mitigate adversarial attacks. However, this is easier said than done in the context of tiny ML systems. It is often hard to monitor embedded ML systems at the endpoint due to communication bandwidth limitations, which we will discuss in the MLOps chapter.

By understanding the potential risks and implementing these defenses, we can help secure on-device training at the endpoint/edge and mitigate the impact of adversarial attacks. Most people easily confuse data poisoning and adversarial attacks. So Table 12.2 compares data poisoning and adversarial attacks:

Table 12.2: Comparison of data poisoning and adversarial attacks.

Aspect	Data Poisoning	Adversarial Attacks
Timing	Training phase	Inference phase
Target	Training data	Input data
Goal	Negatively affect model's performance	Cause incorrect predictions
Method	Insert malicious examples into training data, often with incorrect labels	Add carefully crafted noise to input data
Example	Adding images of cats labeled as dogs to a dataset used for training an image classification model	Adding a small amount of noise to an image in a way that causes a face recognition system to misidentify a person
Potential Effects	Model learns incorrect patterns and makes incorrect predictions	Immediate and potentially dangerous incorrect predictions
Applications Affected	Any ML model	Autonomous vehicles, security systems, etc

12.6.3 Model Inversion

Model inversion attacks are a privacy threat to on-device machine learning models trained on sensitive user data (Nguyen et al. 2023). Understanding this attack vector and mitigation strategies will be important for building secure and ethical on-device AI. For example, imagine an iPhone app that uses on-device learning to categorize photos in your camera roll into groups like "beach," "food," or "selfies" for easier searching.

Nguyen, Ngoc-Bao, Keshigeyan Chandrasegaran, Milad Abdollahzadeh, and Ngai-Man Cheung. 2023. "Re-Thinking Model Inversion Attacks Against Deep Neural Networks." In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 16384–93. IEEE. https://doi.org/10.1109/cvpr52729.2023.01572.

The on-device model may be trained by Apple on a dataset of iCloud photos from consenting users. A malicious attacker could attempt to extract parts of those original iCloud training photos using model inversion. Specifically, the attacker feeds crafted synthetic inputs into the on-device photo classifier. By tweaking the synthetic inputs and observing how the model categorizes them, they can refine the inputs until they reconstruct copies of the original training data - like a beach photo from a user's iCloud. Now, the attacker has breached that user's privacy by obtaining one of their photos without consent. This demonstrates why model inversion is dangerous - it can potentially leak highly sensitive training data.

spacing

Photos are an especially high-risk data type because they often contain identifiable people, location information, and private moments. However, the same attack methodology could apply to other personal data, such as audio recordings, text messages, or users' health data.

To defend against model inversion, one would need to take precautions like adding noise to the model outputs or using privacy-preserving machine learning techniques like federated learning to train the on-device model. The goal is to prevent attackers from being able to reconstruct the original training data.

12.6.4 On-Device Learning Security Concerns

While data poisoning and adversarial attacks are common concerns for ML models in general, on-device learning introduces unique security risks. When on-device variants of large-scale models are published, adversaries can exploit these smaller models to attack their larger counterparts. Research has demonstrated that as on-device models and full-scale models become more similar, the vulnerability of the original large-scale models increases significantly. For instance, evaluations across 19 Deep Neural Networks (DNNs) revealed that exploiting on-device models could increase the vulnerability of the original large-scale models by up to 100 times.

There are three primary types of security risks specific to on-device learning:

- **Transfer-Based Attacks:** These attacks exploit the transferability property between a surrogate model (an approximation of the target model, similar to an on-device model) and a remote target model (the original full-scale model). Attackers generate adversarial examples using the surrogate model, which can then be used to deceive the target model. For example, imagine an on-device model designed to identify spam emails. An attacker could use this model to generate a spam email that is not detected by the larger, full-scale filtering system.
- **Optimization-Based Attacks:** These attacks generate adversarial examples for transfer-based attacks using some form of the objective function and iteratively modify inputs to achieve the desired outcome. Gradient estimation attacks, for example, approximate the model's gradient using query outputs (such as softmax confidence scores), while gradient-free attacks use the model's final decision (the predicted class) to approximate the gradient, albeit requiring many more queries.
- **Query Attacks with Transfer Priors:** These attacks combine elements of transfer-based and optimization-based attacks. They reverse engineer on-device models to serve as surrogates for the target full-scale model. In other words, attackers use the smaller on-device model to understand how the larger model works and then use this knowledge to attack the full-scale model.

By understanding these specific risks associated with on-device learning, we can develop more robust security protocols to protect both on-device and full-scale models from potential attacks.

12.6.5 Mitigation of On-Device Learning Risks

Various methods can be employed to mitigate the numerous security risks associated with on-device learning. These methods may be specific to the type of attack or serve as a general tool to bolster security.

One strategy to reduce security risks is to diminish the similarity between on-device models and full-scale models, thereby reducing transferability by up to 90%. This method, known as similarity-unpairing, addresses the problem that arises when adversaries exploit the input-gradient similarity between the two models. By finetuning the full-scale model to create a new version with similar accuracy but different input gradients, we can construct the on-device model by quantizing this updated full-scale model. This unpairing reduces the vulnerability of on-device models by limiting the exposure of the original full-scale model. Importantly, the order of finetuning and quantization can be varied while still achieving risk mitigation (Hong, Carlini, and Kurakin 2023).

To tackle data poisoning, it is imperative to source datasets from trusted and reliable vendors.

Hong, Sanghyun, Nicholas Carlini, and Alexey Kurakin. 2023. "Publishing Efficient on-Device Models Increases Adversarial Vulnerability." In *2023 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, 271–90. IEEE; IEEE. https://doi.org/10.1109/satml54575.2023.00026.

Several strategies can be employed to combat adversarial attacks. A proactive approach involves generating adversarial examples and incorporating them into the model's training dataset, thereby fortifying the model against such attacks. Tools like CleverHans, an open-source training library, are instrumental in creating adversarial examples. Defense distillation is another effective strategy, wherein the on-device model outputs probabilities of different classifications rather than definitive decisions (Hong, Carlini, and Kurakin 2023), making it more challenging for adversarial examples to exploit the model.

The theft of intellectual property is another significant concern when deploying on-device models. Intellectual property theft is a concern when deploying on-device models, as adversaries may attempt to reverse-engineer the model to steal the underlying technology. To safeguard against intellectual property theft, the binary executable of the trained model should be stored on a microcontroller unit with encrypted software and secured physical interfaces of the chip. Furthermore, the final dataset used for training the model should be kept private.

Furthermore, on-device models often utilize well-known or open-source datasets, such as MobileNet's Visual Wake Words. As such, it is important to maintain the privacy of the final dataset used for training the model. Additionally, protecting the data augmentation process and incorporating specific use cases can minimize the risk of reverse-engineering an on-device model.

Lastly, the Adversarial Threat Landscape for Artificial Intelligence Systems (ATLAS) serves as a valuable

Lastly, the Adversarial Threat Landscape for Artificial-Intelligence Systems (ATLAS) serves as a valuable matrix tool that helps assess the risk profile of on-device models, empowering developers to identify and mitigate potential risks proactively.

12.6.6 Securing Training Data

There are various ways to secure on-device training data. Each concept is really deep and could be worth a class by itself. So here, we'll briefly allude to those concepts so you're aware of what to learn further.

great!

Encryption

Encryption serves as the first line of defense for training data. This involves implementing end-to-end encryption for local storage on devices and communication channels to prevent unauthorized access to raw training data. Trusted execution environments, such as Intel SGX and ARM TrustZone, are essential for facilitating secure training on encrypted data.

Additionally, when aggregating updates from multiple devices, secure multi-party computation protocols can be employed to enhance security (Kairouz, Oh, and Viswanath 2015); a practical application of this is in collaborative on-device learning, where cryptographic privacy-preserving aggregation of user model updates can be implemented. This technique effectively hides individual user data even during the aggregation phase.

Kairouz, Peter, Sewoong Oh, and Pramod Viswanath. 2015. "Secure Multi-Party Differential Privacy." In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, edited by Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, 2008–16. https://proceedings.neurips.cc/paper/2015/hash/a01610228fe99 Abstract.html

Differential Privacy

Differential privacy is another crucial strategy for protecting training data. By injecting calibrated statistical noise into the data, we can mask individual records while still extracting valuable population patterns (Dwork and Roth 2013). Managing the privacy budget across multiple training iterations and reducing noise as the model converges is also vital (Abadi et al. 2016). Methods such as formally provable differential privacy, which may include adding Laplace or Gaussian noise scaled to the dataset's sensitivity, can be employed.

Dwork, Cynthia, and Aaron Roth. 2013. "The Algorithmic Foundations of Differential Privacy." *Foundations and Trends in Theoretical Computer Science* 9 (3-4): 211–407. https://doi.org/10.1561/0400000042

Abadi, Martin, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. "Deep Learning with Differential Privacy." In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 308–18. CCS '16. New York, NY, USA: ACM. https://doi.org/10.1145/2976749.2978318

Anomaly Detection

Anomaly detection plays an important role in identifying and mitigating potential data poisoning attacks. This can be achieved through statistical analyses like Principal Component Analysis (PCA) and clustering, which help to detect deviations in aggregated training data. Time-series methods such as Cumulative Sum (CUSUM) charts are useful for identifying shifts indicative of potential poisoning. Comparing current data distributions with previously seen clean data distributions can also help to flag anomalies. Moreover, suspected poisoned batches should be removed from the training update aggregation process. For example, spot checks on subsets of training images on devices can be conducted using photoDNA hashes to identify poisoned inputs.

Input Data Validation

Lastly, input data validation is essential for ensuring the integrity and validity of input data before it is fed into the training model, thereby protecting against adversarial payloads. Similarity measures, such as cosine distance, can be employed to catch inputs that deviate significantly from the expected distribution. Suspicious inputs that may contain adversarial payloads should be quarantined and sanitized. Furthermore, parser access to training data should be restricted to validated code paths only. Leveraging hardware security features, such as ARM Pointer Authentication, can prevent memory corruption (ARM Limited, 2023). An example of this is implementing input integrity checks on audio training data used by smart speakers before processing by the speech recognition model (Z. Chen and Xu 2023).

Chen, Zhiyong, and Shugong Xu. 2023. "Learning Domain-Heterogeneous Speaker Recognition Systems with Personalized Continual Federated Learning." *EURASIP Journal on Audio, Speech, and Music Processing* 2023 (1): 33. https://doi.org/10.1186/s13636-023-00299-2

12.7 On-Device Training Frameworks

Embedded inference frameworks like TF-Lite Micro (David et al. 2021), TVM (T. Chen et al. 2018), and MCUNet (Lin et al. 2020) provide a slim runtime for running neural network models on microcontrollers and other resource-constrained devices. However, they don't support on-device training. Training requires its own set of specialized tools due to the impact of quantization on gradient calculation and the memory footprint of backpropagation (Lin et al. 2022).

David, Robert, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, et al. 2021. "Tensorflow Lite Micro: Embedded Machine Learning for Tinyml Systems." *Proceedings of Machine Learning and Systems* 3: 800–811.

Chen, Tianqi, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, et al. 2018. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning." In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 578–94.

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/86c51678350f6 Abstract.html

In recent years, a handful of tools and frameworks have started to emerge that enable on-device training. These include Tiny Training Engine (Lin et al. 2022), TinyTL (Cai et al. 2020), and TinyTrain (Kwon et al. 2023).

12.7.1 Tiny Training Engine

Tiny Training Engine (TTE) uses several techniques to optimize memory usage and speed up the training process. An overview of the TTE workflow is shown in Figure 12.9. First, TTE offloads the automatic differentiation to compile time instead of runtime, significantly reducing overhead during training. Second, TTE performs graph optimization like pruning and sparse updates to reduce memory requirements and accelerate computations.

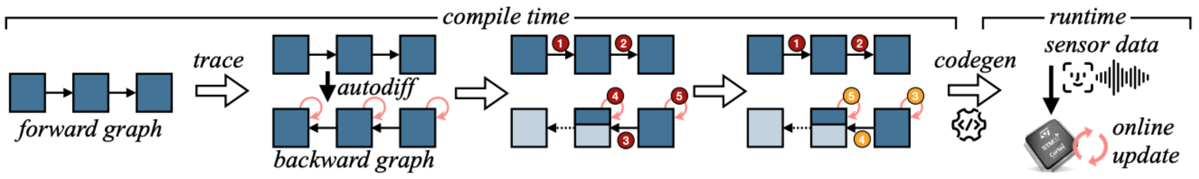


Figure 12.9: TTE workflow.

Specifically, TTE follows four main steps:

- During compile time, TTE traces the forward propagation graph and derives the corresponding backward graph for backpropagation. This allows differentiation to happen at compile time rather than runtime.

Lin, Ji, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. "On-Device Training Under 256kb Memory." *Adv. Neur. In.*

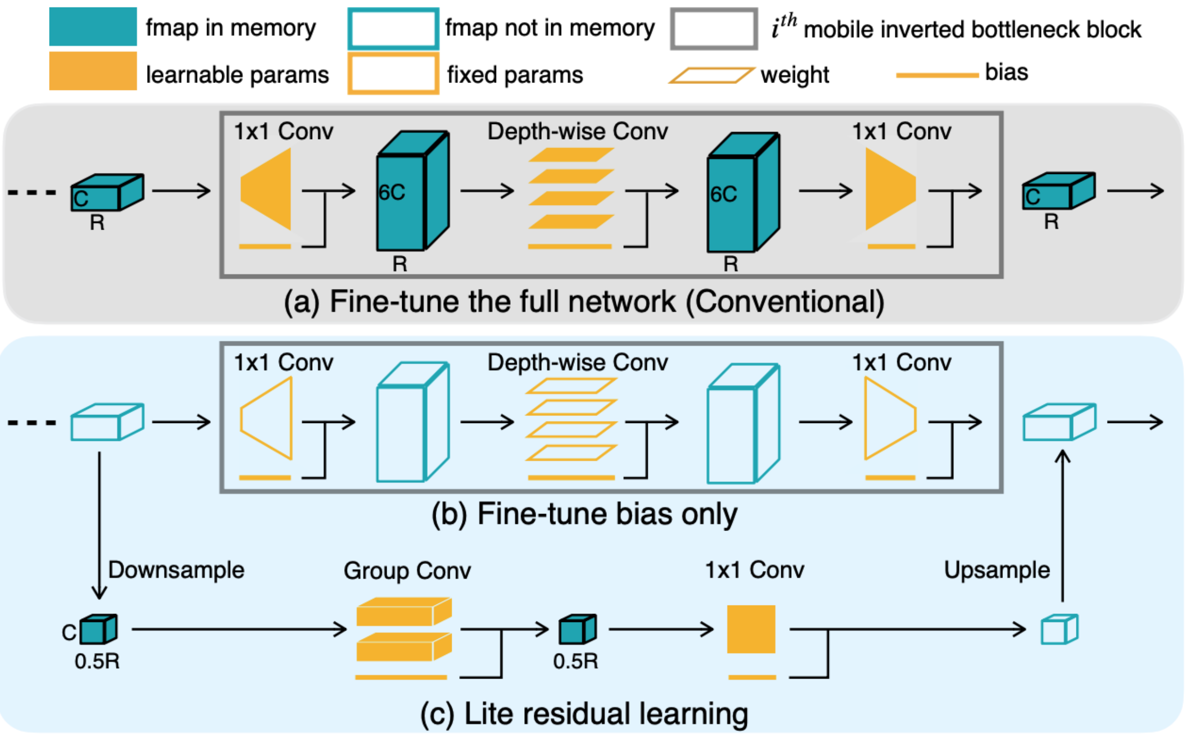
35: 22941–54.

- TTE prunes any nodes representing frozen weights from the backward graph. Frozen weights are weights that are not updated during training to reduce certain neurons' impact. Pruning their nodes saves memory.
- TTE reorders the gradient descent operators to interleave them with the backward pass computations. This scheduling minimizes memory footprints.
- TTE uses code generation to compile the optimized forward and backward graphs, which are then deployed for on-device training.

12.7.2 Tiny Transfer Learning

Tiny Transfer Learning (TinyTL) enables memory-efficient on-device training through a technique called weight freezing. During training, much of the memory bottleneck comes from storing intermediate activations and updating the weights in the neural network.

To reduce this memory overhead, TinyTL freezes the majority of the weights so they do not need to be updated during training. This eliminates the need to store intermediate activations for frozen parts of the network. TinyTL only finetunes the bias terms, which are much smaller than the weights. An overview of TinyTL workflow is shown in Figure 12.10.



confused on this image

Figure 12.10: TinyTL workflow. Credit: Cai et al. (2020).)

Cai, Han, Chuang Gan, Ligeng Zhu, and Song Han. 2020. "TinyTL: Reduce Memory, Not Parameters for Efficient on-Device Learning." 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/81f7acabd4112 Abstract.html.

Freezing weights apply to fully connected layers as well as convolutional and normalization layers. However, only adapting the biases limits the model's ability to learn and adapt to new data.

To increase adaptability without much additional memory, TinyTL uses a small residual learning model. This refines the intermediate feature maps to produce better outputs, even with fixed weights. The residual model introduces minimal overhead - less than 3.8% on top of the base model.

By freezing most weights, TinyTL significantly reduces memory usage during on-device training. The residual model then allows it to adapt and learn effectively for the task. The combined approach provides memory-efficient on-device training with minimal impact on model accuracy.

12.7.3 Tiny Train

TinyTrain significantly reduces the time required for on-device training by selectively updating only certain parts of the model. It does this using a technique called task-adaptive sparse updating, as shown in Figure 12.11.

Based on the user data, memory, and computing available on the device, TinyTrain dynamically chooses which neural network layers to update during training. This layer selection is optimized to reduce computation and memory usage while maintaining high accuracy.

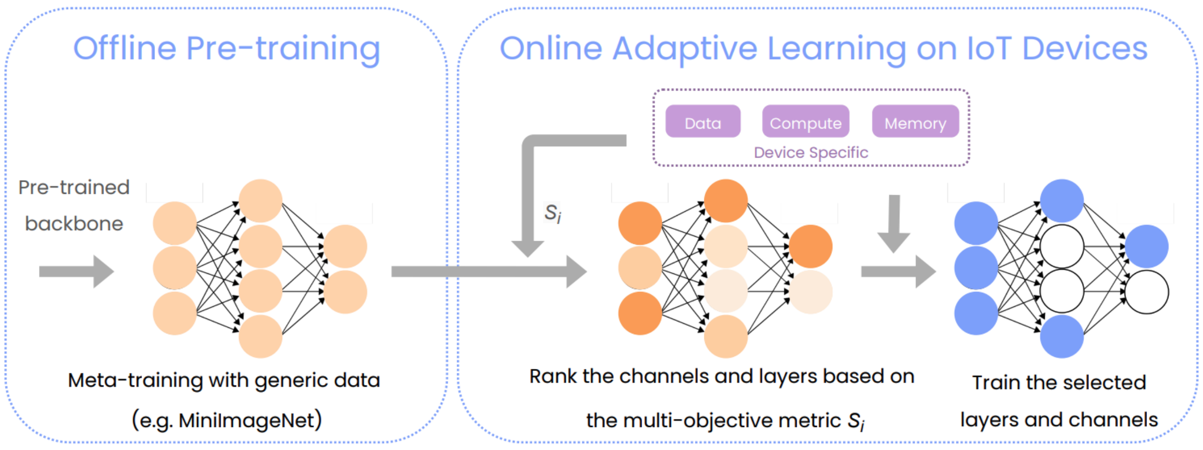

Offline Pre-training
Online Adaptive Learning on IoT Devices
Pre-trained
backbone
Meta-training with generic data
(e.g. MiniImageNet)
Data
Compute
Memory
Device Specific
S_i
Rank the channels and layers based on
the multi-objective metric S_i
Train the selected
layers and channels

Figure 12.11: TinyTrain workflow. Credit: Kwon et al. (2023).

Kwon, Young D, Rui Li, Stylianos I Venieris, Jagmohan Chauhan, Nicholas D Lane, and Cecilia Mascolo. 2023. "TinyTrain: Deep Neural Network Training at the Extreme Edge." *ArXiv Preprint* abs/2307.09988. https://arxiv.org/abs/2307.09988.

More specifically, TinyTrain first does offline pretraining of the model. During pretraining, it not only trains the model on the task data but also meta-trains the model. Meta-training means training the model on metadata about the training process itself. This meta-learning improves the model's ability to adapt accurately even when limited data is available for the target task.

Then, during the online adaptation stage, when the model is being customized on the device, TinyTrain performs task-adaptive sparse updates. Using the criteria around the device's capabilities, it selects only certain layers to update through backpropagation. The layers are chosen to balance accuracy, memory usage, and computation time.

By sparsely updating layers tailored to the device and task, TinyTrain significantly reduces on-device training time and resource usage. The offline meta-training also improves accuracy when adapting to limited data. Together, these methods enable fast, efficient, and accurate on-device training.

12.7.4 Comparison

Here is a table summarizing the key similarities and differences between the Tiny Training Engine, TinyTL, and TinyTrain frameworks:

Framework	Similarities	Differences
Tiny Training Engine	On-device training Optimize memory & computation Leverage pruning, sparsity, etc	Traces forward & backward graphs Prunes frozen weights Interleaves backprop & gradients Code generation
TinyTL	On-device training Optimize memory & computation Leverage freezing, sparsity, etc	Freezes most weights Only adapts biases Uses residual model
TinyTrain	On-device training Optimize memory & computation Leverage sparsity, etc	Meta-training in pretraining Task-adaptive sparse updating Selective layer updating

✓

12.8 Conclusion

The concept of on-device learning is increasingly important for increasing the usability and scalability of TinyML. This chapter explored the intricacies of on-device learning, exploring its advantages and limitations, adaptation strategies, key related algorithms and techniques, security implications, and existing and emerging on-device training frameworks.

On-device learning is, undoubtedly, a groundbreaking paradigm that brings forth numerous advantages for embedded and edge ML deployments. By performing training directly on the endpoint devices, on-device learning obviates the need for continuous cloud connectivity, making it particularly well-suited for IoT and edge computing applications. It comes with benefits such as improved privacy, ease of compliance, and resource efficiency. At the same time, on-device learning faces limitations related to hardware constraints, limited data size, and reduced model accuracy and generalization.

Mechanisms such as reduced model complexity, optimization and data compression techniques, and related learning methods such as transfer learning and federated learning allow models to adapt to learn and evolve under resource constraints, thus serving as the bedrock for effective ML on edge devices.

The critical security concerns in on-device learning highlighted in this chapter, ranging from data poisoning and adversarial attacks to specific risks introduced by on-device learning, must be addressed in real workloads for on-device learning to be a viable paradigm. Effective mitigation strategies, such as data validation, encryption, differential privacy, anomaly detection, and input data validation, are crucial to safeguard on-device learning systems from these threats.

The emergence of specialized on-device training frameworks like Tiny Training Engine, Tiny Transfer Learning, and Tiny Train presents practical tools to enable efficient on-device training. These frameworks employ various techniques to optimize memory usage, reduce computational overhead, and streamline the on-device training process.

In conclusion, on-device learning stands at the forefront of TinyML, promising a future where models can autonomously acquire knowledge and adapt to changing environments on edge devices. The application of on-device learning has the potential to revolutionize various domains, including healthcare, industrial IoT, and smart cities. However, the transformative potential of on-device learning must be balanced with robust security measures to protect against data breaches and adversarial threats. Embracing innovative on-device training frameworks and implementing stringent security protocols are key steps in unlocking the full potential of on-device learning. As this technology continues to evolve, it holds the promise of making our devices smarter, more responsive, and better integrated into our daily lives.

12.9 Resources

great chp!

Here is a curated list of resources to support students and instructors in their learning and teaching journeys. We are continuously working on expanding this collection and will add new exercises soon.

Slides

These slides serve as a valuable tool for instructors to deliver lectures and for students to review the material at their own pace. We encourage both students and instructors to leverage these slides to enhance their understanding and facilitate effective knowledge transfer.

- Intro to TensorFlow Lite (TFLite).
- TFLite Optimization and Quantization.
- TFLite Quantization-Aware Training.
- Transfer Learning:
  - Transfer Learning: with Visual Wake Words example.
  - On-device Training and Transfer Learning.
- Distributed Training:
  - Distributed Training.

    - Distributed Training.
- Continuous Monitoring:
    - Continuous Evaluation Challenges for TinyML.
    - Federated Learning Challenges.
    - Continuous Monitoring with Federated ML.
    - Continuous Monitoring Impact on MLOps.

Videos

- Video 12.1
- Video 12.2

Exercises

To reinforce the concepts covered in this chapter, we have curated a set of exercises that challenge students to apply their knowledge and deepen their understanding.

- Exercise 12.1
- Exercise 12.2
- Exercise 12.3

Labs

In addition to exercises, we also offer a series of hands-on labs that allow students to gain practical experience with embedded AI technologies. These labs provide step-by-step guidance, enabling students to develop their skills in a structured and supportive environment. We are excited to announce that new labs will be available soon, further enriching the learning experience.

- *Coming soon.*

Written, edited and curated by Prof. Vijay Janapa Reddi (Harvard University)

This book was built with Quarto.