Topics in Parallel and Distributed Computing: From Concepts to the Classroom * † ‡

Chapter – Applying Federated Learning: A Parallel and Distributed Computing Perspective

Jeremie Ruvunangiza and C. Valderrama

University of Mons, jeremie.ruvunangiza@umons.ac.be,

[2]carlos.valderrama@umons.ac.be

*Call for CDER Book chapter proposal: https://tcpp.cs.gsu.edu/curriculum/?q=call-for-book-chapter-proposals

†Free preprint version of Volume two: https://grid.cs.gsu.edu/~tcpp/curriculum/?q=cder_book_2

‡Free preprint version of volume one: https://grid.cs.gsu.edu/~tcpp/curriculum/?q=cedr_book

TABLE OF CONTENTS

Abstract

This chapter examines how incorporating Federated Learning (FL) into undergraduate computing curricula can serve as an effective framework for teaching the principles of Parallel and Distributed Computing (PDC). Designed to bridge the gap between theoretical knowledge and practical application, the chapter emphasizes hands-on implementation. By focusing on foundational PDC principles, such as data partitioning, communication efficiency, and distributed model training, the chapter equips to address complex challenges related to data privacy, security, and scalability in distributed systems. This chapter aims to prepare students with industry-relevant skills, fostering both technical proficiency and ethical understanding in the rapidly evolving field of distributed computing.

1. Practical Skill Development

   Enable students to implement FL models using industry-standard tools. Foster problem solving skills through real-world applications.

2. Curriculum integration

   Provide educators with practical projects and assignments for existing courses. Align resources with NSF/IEEE-TCPP curriculum guidelines.

Learning outcomes: 1. Understand Federated Learning (FL) and its role in Parallel and Distributed Computing (PDC),

2. Implement FL using frameworks like TensorFlow Federated (TFF) and PySyft,
3. Identify key security and privacy challenges in FL,
4. Optimize FL for handling non-IID data and system heterogeneity,
5. Explore real-world FL applications in AI, healthcare, and IoT

Context for use: FL is essential for privacy-preserving AI and distributed computing. This chapter provides both theoretical insights and practical applications to help students implement and understand FL effectively.

1.1 Introduction

1.1.1 Definition and Overview of Federated Learning

Federated Learning (FL) is a decentralized machine learning paradigm that enables multiple clients to train a shared model collaboratively while keeping their raw data localized. Unlike centralized machine learning, where all data are transferred to a single server for model training, FL distributes computations across multiple devices or nodes. This approach improves data privacy, reduces communication overhead, and improves scalability, making it well suited for privacy-sensitive and resource-constrained environments.

FL can be categorized into two primary architectures:

- Centralized FL: A central aggregator (or server) collects and integrates model updates from clients, ensuring periodic synchronization.
- Decentralized (Peer-to-Peer) FL: Clients communicate directly with each other without relying on a central coordinator, improving robustness, fault tolerance, and scalability.

FL has gained significant attention across various domains, including privacy-preserving AI, edge computing, and large-scale distributed learning systems. Its distributed nature

makes it particularly useful in scenarios where data sharing is restricted due to legal and regulatory constraints (e.g., GDPR, HIPAA) or where network bandwidth is limited. By enabling collaborative model training while maintaining data locality, FL serves as a key enabler for secure, scalable, and privacy-preserving AI.

1.1.2 Core PDC Principles Relevant to Federated Learning

Federated learning (FL) is fundamentally a Parallel and Distributed Computing (PDC) problem, requiring efficient coordination among distributed clients, effective communication strategies, and robust aggregation techniques to ensure model convergence. Unlike traditional distributed computing, FL introduces unique constraints, including the preservation of privacy, system heterogeneity, and communication efficiency, which must be addressed through PDC principles[1][4].

A well-structured FL system leverages multiple PDC concepts, including distributed training, model synchronization, data partitioning, and communication-efficient learning. These principles define how FL operates in decentralized environments and play a crucial role in optimizing performance while maintaining security and scalability[1][3].

Distributed Training and Model Aggregation

In FL, training occurs simultaneously across multiple nodes (clients), where each client performs local computations on its own data. After training, only model updates (e.g., weights, gradients) are sent to a central aggregator or peer nodes for synchronization, without sharing raw data. This distributed training introduces several challenges:

Synchronization of model updates: Since clients operate independently, updates arrive at different times, requiring techniques to handle asynchrony. Efficient aggregation of local updates: Algorithms like Federated Averaging (FedAvg) and Federated Proximal (FedProx) are used to combine local model updates while mitigating issues like stragglers (clients that update slower than others)[1] [2]. Managing resource constraints: Clients have varying computational capacities, leading to potential inefficiencies in training convergence. Adaptive client selection and workload balancing techniques help optimize resource utilization.

Data Partitioning Across Nodes

One of the most significant challenges in FL is handling Non-Independent and Identically Distributed (Non-IID) data across clients. Unlike traditional ML, where the training datasets are balanced and IID, FL often involves highly heterogeneous, biased, and unbalanced datasets. The following issues arise from such distributions:

Data heterogeneity: Clients may have vastly different data distributions based on their local environment (e.g., demographic variations in mobile users, sensor data discrepancies in IoT devices). This leads to skewed model updates, making global model convergence difficult. Bias in Local Data Sets: Some clients may have significantly different feature distributions than others, causing model divergence. Personalized FL approaches aim to mitigate this by adjusting model updates based on client-specific characteristics. System heterogeneity: Clients range from high-performance computing nodes to low-power IoT devices. This disparity necessitates adaptive computation strategies that adjust workload distribution based on client capabilities[3].

To address these challenges, various optimization techniques are applied, including clustering-based aggregation, where clients with similar data distributions are grouped to improve training stability, and transfer learning-based FL, where models trained on similar data subsets share information to enhance performance [1].

Communication-Efficient Learning

Since FL operates in a distributed manner, minimizing communication overhead is essential for scalability. Traditional ML setups involve frequent parameter exchanges between training nodes and a central server, which is impractical in FL due to bandwidth constraints and energy limitations. Some key techniques to improve communication efficiency include [3].

Gradient sparsification: Instead of transmitting full model updates, clients share only the most significant parameter changes, reducing data transfer volume. Model compression: Quantization and pruning techniques help reduce model size while maintaining accuracy and optimizing bandwidth consumption. Asynchronous FL training: Allowing clients to

update the global model at different times without waiting for all clients to finish training can enhance efficiency. By optimizing these communication strategies, FL can be effectively deployed in low-bandwidth environments such as edge computing, IoT networks, and mobile devices, making it a viable solution for decentralized AI[3].

Parallelism and Scalability in FL

FL inherently relies on parallel processing across multiple clients, making it a natural extension of Parallel and Distributed Computing (PDC). However, achieving scalability requires balancing computational efficiency and accuracy.

Client selection strategies: Instead of using all available clients, adaptive client selection ensures that only a subset of high-quality contributors participates in each training round, reducing computational complexity. Hierarchical FL: Multilevel aggregation strategies enable FL to scale to thousands or millions of clients by introducing intermediate aggregators at different network levels (e.g., edge-cloud coordination). Load balancing: The assignment of computational tasks based on the capabilities of the device prevents bottlenecks and improves the speed of the training. By integrating parallelism, efficient data partitioning, and communication-aware optimization techniques, FL provides a practical learning opportunity for students to explore PDC concepts in real-world AI systems. Addressing these challenges allows FL to scale efficiently while ensuring privacy, robustness, and adaptability across diverse computing environments[1].

1.2 Objectives of Integrating PDC into FL Education

Federated learning (FL) is rapidly emerging as a foundational component of modern Parallel and Distributed Computing (PDC) systems, bridging the gap between machine learning, privacy preservation, and distributed computing. Its integration into undergraduate curricula provides students with hands-on exposure to real-world challenges in distributed AI, preparing them for careers in AI, cloud computing, cybersecurity, and large-scale systems engineering.

Teaching FL within a PDC-focused curriculum improves students' understanding of

distributed training, system heterogeneity, privacy-preserving mechanisms, and optimization techniques. By addressing key aspects of data decentralization, security constraints, and scalability, students gain both theoretical knowledge and practical problem solving skills applicable to cutting-edge AI applications.

Bridging Theory and Practice A critical objective of integrating FL into PDC education is to help students connect theoretical concepts to practical implementations. Many traditional computing courses focus on abstract parallelism and distributed algorithms without providing real-world case studies demonstrating their impact. FL offers a tangible example of:

Decentralized model training: Understanding how parallelism and distributed model updates impact real-world AI performance. Network Efficiency in Distributed Learning: Exploring the trade-offs between model accuracy and communication constraints in edge and cloud environments. Synchronization and Update Aggregation: Examining the impact of synchronous vs. asynchronous FL models on computational efficiency and training speed. By integrating coding exercises, simulations, and real-world datasets, students go beyond theoretical understanding to practical proficiency, preparing them for industry roles requiring distributed AI expertise.

Developing Technical and Problem-Solving Skills FL provides a unique opportunity to teach students how to optimize parallel computing systems under real-world constraints. Key technical challenges explored in FL-based coursework include:

Handling system and data heterogeneity: Teaching students how to balance computational workloads across high-power cloud nodes and low-power IoT devices. Privacy-preserving learning strategies: Exploring differential privacy, homomorphic encryption, and multiparty secure computation to enhance security in FL systems. Non-IID data and fairness issues: Investigating techniques like clustered federated learning and personalized FL models to mitigate data imbalance and bias. Scalability in large-scale AI systems: Demonstrating strategies for efficient client selection, model compression, and hierarchical FL architectures to optimize training across thousands of distributed nodes. By engaging in project-based

learning, students develop critical problem-solving skills that extend beyond FL, equipping them with the ability to design, implement, and troubleshoot distributed computing solutions.

Preparation for Industry and Research Preparedness

FL is increasingly being adopted in finance, healthcare, cybersecurity, and IoT, making it a valuable area of study for students pursuing careers in AI, data science, and cloud computing. By incorporating FL into PDC coursework, students gain direct exposure to industry-relevant challenges and innovations, such as:

Edge AI and resource-aware distributed computing: Learning how to optimize ML models for low-power devices in smart homes, autonomous vehicles, and mobile applications. Security and compliance in real-world FL deployments: Understanding how FL aligns with GDPR, HIPAA, and financial data protection regulations, preparing students for AI governance and cybersecurity roles. Emerging Research Directions in FL and PDC: Encourage students to explore federated reinforcement learning, decentralized blockchain-based FL, and quantum FL as potential areas for advanced study and innovation. By integrating state-of-the-art FL frameworks (e.g., TensorFlow Federated, PySyft, FedML) into coursework, students gain hands-on experience with real-world FL deployments, making them more competitive in both industry and academic research.

Addressing Ethical and Privacy Considerations Incorporating FL into the education of PDC provides a platform to discuss the ethical implications of distributed AI. Students explore privacy risks, fairness concerns, and adversarial threats in FL-based learning environments, including:

Privacy violations in decentralized AI: Understanding how model inversion attacks and gradient leakage can expose sensitive client data. Bias and Fairness in Federated Models: We discuss how imbalanced training data can lead to biased AI decisions, particularly in healthcare and finance applications. Regulatory compliance and responsible AI: Teaching students about federated learning policies and encouraging them to design AI systems that are both scalable and ethically sound. By addressing these real-world challenges, students

gain a holistic understanding of FL beyond its technical aspects, fostering an appreciation for AI ethics, security, and responsible computing.

Integrating Federated Learning into Parallel and Distributed Computing curricula offers a transformative approach to AI education. By combining theory with hands-on projects, industry-relevant applications, and security-focused discussions, students gain critical skills in distributed computing, privacy-preserving AI, and scalable system design.

This approach ensures that FL is not taught only as an advanced machine learning technique but as a real-world, industry-driven application of PDC, preparing students for cutting-edge roles in AI, research, and cloud computing.

1.3 Security and Privacy in Federated Learning

The inherently decentralized structure of Federated Learning (FL) offers a range of benefits, including data minimization and enhanced user privacy, but it also introduces novel security and privacy challenges that traditional machine learning (ML) frameworks may not adequately address. Malicious actors can target an FL system through various attack vectors, including adversarial manipulations, model poisoning, or inference attacks designed to extract private information from gradient updates. Moreover, the distribution of data and training responsibilities across multiple devices or institutions can lead to system vulnerabilities and coordination complexities.

As FL gains traction in domains with stringent confidentiality requirements, such as healthcare and finance, the ability to safeguard sensitive data and maintain robust security becomes critical. Researchers and practitioners, therefore, emphasize the need for specialized security protocols and privacy-preserving mechanisms tailored to FL's distributed nature.

1.3.1 Secure Aggregation and Privacy-Preserving Techniques

A central objective in Federated Learning is to preserve data confidentiality while facilitating collaborative model training. To achieve this, various cryptographic methods and algorithmic safeguards have been devised:

Secure Multi-Party Computation (SMPC): This approach leverages secret-sharing techniques to encrypt intermediate computations. Each client's model updates remain confidential, preventing other parties or the central aggregator from directly accessing raw gradients.

Homomorphic Encryption (HE): HE enables arithmetic operations to be performed on encrypted data without requiring decryption. In an FL context, model updates can be aggregated securely, reducing the risk of exposing sensitive information even during the aggregation process.

Differential Privacy (DP): By introducing carefully calibrated noise into model updates, DP mechanisms obscure details about individual data points. This significantly mitigates the risk of inference attacks, where adversaries attempt to infer private attributes from the learned model parameters.

Through these privacy-preserving techniques, FL systems can maintain a high level of security without sacrificing the collaborative benefits that distributed training provides.

1.3.2 Mitigating Attacks in Federated Learning

Despite the protective measures outlined above, FL remains susceptible to a variety of threats, with model poisoning attacks being a particularly pressing concern. In such attacks, adversarial clients may contribute malicious updates designed to degrade the global model's performance or bias its outputs.

To defend against these threats, several strategies have emerged:

Anomaly Detection: By monitoring gradient updates for unusual patterns or deviations, the system can flag and isolate suspicious contributions. This proactive approach helps maintain model integrity.

Robust Aggregation Methods (e.g., Krum, Bulyan): Unlike straightforward averaging, these techniques incorporate statistical checks to identify and minimize the influence of outlier gradients, thereby filtering out potentially harmful updates.

Adversarial Training Techniques: Integrating resilience at the model level, adver-

sarial training introduces and accounts for potential malicious examples or updates during the learning process, improving the robustness of the model under adversarial conditions.

Beyond technical considerations, FL must also adhere to legal and regulatory standards such as the General Data Protection Regulation (GDPR) in Europe and the Health Insurance Portability and Accountability Act (HIPAA) in the United States. Ensuring compliance requires privacy-aware model updates, auditing capabilities, and transparent governance to guarantee that individual rights and institutional responsibilities are upheld throughout the FL lifecycle.

1.4 Communication Efficiency in Federated Learning

A critical bottleneck in Federated Learning (FL) lies in the communication demands imposed by frequent model updates. Because clients typically transmit gradients or model parameters to a central server—or to one another in decentralized setups—network bandwidth can quickly become saturated. As the number of clients or the size of the model grows, communication efficiency becomes paramount, particularly in low-bandwidth or resource-constrained environments.

To address these constraints, researchers and practitioners focus on designing mechanisms that either reduce the amount of data exchanged during each training round or optimize how frequently and to whom updates are sent. Such techniques are integral to improving scalability and performance, especially in large-scale deployments involving heterogeneous devices.

1.4.1 Gradient Compression and Model Sparsification

One core strategy to enhance communication efficiency is to reduce the size and frequency of transmitted updates. Several related methods have emerged:

Gradient Sparsification: Rather than sending a complete gradient vector, only the most salient updates—those above a certain threshold in magnitude—are transmitted. This approach maintains a similar direction of model updates while substantially cutting down

on communication overhead.

Quantization: By lowering the precision of floating-point updates, quantization methods reduce the bit-width of transmitted gradients. Although this step introduces some approximation error, it drastically decreases the volume of data sent per round, enabling faster convergence in bandwidth-limited contexts.

Federated Dropout: Inspired by dropout in neural network training, this technique randomly or selectively omits subsets of parameters from the update. Consequently, only a partial model is transmitted, further reducing communication requirements.

Implementing these compression and sparsification methods can significantly alleviate network load, thereby accelerating training cycles in Federated Learning environments.

1.4.2 Adaptive Client Participation

Another approach to improving communication efficiency is to tailor the participation of clients based on their network conditions or computational capabilities:

Bandwidth-Aware Selection: Systems can prioritize high-bandwidth clients for frequent updates, postpone, or reduce the update frequency of slower or more resource-limited devices. This adaptive scheme prevents stragglers from bottlenecking the entire training process.

Dynamic Communication Scheduling: By adjusting communication intervals—sending updates less frequently or only after a certain amount of local progress—FL systems can strike a balance between convergence speed and bandwidth usage. This flexibility becomes increasingly valuable when clients exhibit varying degrees of connectivity and power constraints.

Through these techniques, Federated Learning frameworks can effectively manage the trade-off between accurate, timely model updates and the practical limitations of real-world network environments, ensuring that collaborative training remains efficient and scalable.

1.5 Scalability and Resource Management in Federated Learning

Federated learning (FL) holds the promise of harnessing vast networks of devices to collaboratively train powerful machine learning models. However, as FL scales to millions or even billions of devices, it faces formidable challenges in distributing workloads, managing resource consumption, and preventing system bottlenecks. These difficulties arise from the inherently heterogeneous nature of large device populations, which vary widely in computational capacity, network bandwidth, and battery life. Balancing load and ensuring energy efficiency therefore become central concerns in designing practical, large-scale FL systems.

The problem of scalability in FL is further compounded by the need to manage intermittent client availability, align training efforts in real time, and accommodate diverse application constraints. As a result, FL frameworks must incorporate sophisticated strategies to handle the workload distribution and energy demands of devices ranging from powerful edge servers to resource-constrained IoT sensors.

1.5.1 Load Balancing Strategies

A primary hurdle in large-scale FL deployments is how to efficiently distribute training responsibilities among a potentially massive set of participating clients:

Adaptive Client Selection: In this approach, the system prioritizes clients with robust computational power or superior connectivity. By selectively including only those devices most capable of contributing meaningful updates, FL frameworks reduce overhead and speed up convergence, while still preserving overall model quality.

Hierarchical FL: To further improve load balancing, hierarchical architectures may be employed. In this design, intermediate nodes—often referred to as edge devices—take on preliminary aggregation tasks before relaying updates to the central server. By offloading part of the aggregation process to intermediate layers, hierarchical FL mitigates the communication burden on the main server and helps cope with uneven device performance across the network.

1.5.2 Energy-Aware Federated Learning

Energy consumption is another pressing concern, especially in battery-powered or resource-limited devices typical of the Internet of Things (IoT):

Federated Knowledge Distillation: This technique involves using large, high-capacity global models as "teacher" networks, while individual client devices train smaller, more efficient "student" models. The knowledge transferred from the teacher to the student can maintain strong performance while drastically reducing computational and energy costs on edge devices.

Model Pruning and Sparsification: By selectively removing weights or reducing the precision of parameters, FL systems can generate models that require fewer computations without overly compromising accuracy. Such techniques are particularly valuable in scenarios where minimizing on-device energy use is critical, and regular battery recharging or continuous power supply cannot be guaranteed.

By integrating these load balancing and energy-aware methods, FL platforms can sustainably scale to massive device networks, optimizing both performance and resource consumption in real-world, large-scale deployments.

1.6 Case Studies: Real-World Applications of Federated Learning

Federated Learning (FL) has expanded far beyond the academic sphere, demonstrating its value across diverse industries and use cases. By enabling collaborative model training without centralizing data, FL techniques allow organizations to harness distributed data resources in a manner that respects privacy and security constraints. Three sectors where FL has gained notable traction—healthcare, finance, and edge computing—illustrate both the benefits and challenges involved in real-world FL deployments.

1.6.1 FL in Healthcare

The healthcare domain stands as a prime example of how FL can unlock the value of distributed data while preserving sensitive patient information. A common use case is the secure training of machine learning models on medical imaging datasets across multiple hospitals or research institutions. Through FL, each hospital can refine a shared model locally, ensuring that patient records and sensitive medical details are never exposed to external entities.

Challenges.Despite its promise, FL in healthcare must navigate a complex regulatory landscape defined by statutes like HIPAA in the United States and GDPR in the European Union. Institutions must ensure that patient data cannot be reconstructed or accidentally leaked through model updates. Additionally, medical data is often non-IID, with each hospital treating different demographics or specialized patient populations, creating convergence and fairness hurdles. Researchers are thus motivated to design FL frameworks that handle these data distribution biases while maintaining compliance and robust performance.

1.6.2 FL in Finance

Another critical area where FL has made significant inroads is the financial sector. Here, it enables multiple banks or financial institutions to jointly train fraud detection or risk assessment models without disclosing sensitive transaction data. By pooling knowledge from varied sources, these collaborative models can detect fraudulent behavior more accurately and promptly than models developed in isolation.

Challenges. Financial data, like healthcare data, is subject to strict privacy and confidentiality regulations. Additionally, adversaries may attempt model inversion attacks, wherein private information about individual transactions or customers is inferred from the trained model. FL practitioners must, therefore, implement robust encryption strategies, differential privacy mechanisms, and careful aggregation techniques to thwart such attacks while maintaining high model fidelity.

1.6.3 FL in Edge Computing and IoT

In the rapidly expanding Internet of Things (IoT), FL offers a means to develop and continuously update machine learning models directly on edge devices such as smart home sensors, wearable technology, or autonomous vehicles. By keeping data at the source, FL maintains user privacy and reduces the burden on central servers or cloud infrastructure. Smart home systems, for example, can employ FL to refine user preference models (e.g., energy usage patterns, appliance scheduling) without transmitting unfiltered household data.

Challenges.Deploying FL in edge computing environments demands particular care in managing the low-power and real-time requirements typical of IoT devices. Energy consumption must be minimized to accommodate battery limitations, and model latency must remain low to enable responsive control or decision-making. Coordination can also be challenging when devices intermittently connect to the network or possess widely varying computational capabilities.

Overall, these real-world case studies highlight both the versatility and the complexity of implementing Federated Learning at scale. Across healthcare, finance, and the IoT ecosystem, FL solutions must strike a delicate balance between leveraging distributed data resources and safeguarding the confidentiality, robustness, and performance standards demanded by mission-critical applications.

1.7 Practical Implementation of PDC in Federated Learning

Practical deployment of Federated Learning (FL) necessitates a careful alignment of privacy, debugging, and communication (PDC) considerations to ensure robust and efficient model training. Because FL operates over distributed and often heterogeneous datasets, implementing real-world systems entails not only choosing appropriate software frameworks but also establishing rigorous testing, debugging, and validation protocols. In addition, FL developers must navigate unique constraints—such as the inability to directly inspect raw data and the need to uphold stringent privacy requirements—when refining and debugging

their models. The following subsections discuss the principal components of a practical FL setup, using TensorFlow Federated (TFF) as a leading example, and highlight potential project ideas for hands-on experience.

1.7.1 Implementing FL with TensorFlow Federated (TFF)

TensorFlow Federated (TFF) is a widely adopted framework for building FL systems, providing high-level abstractions that streamline the process of deploying models across multiple clients. The following is a broad outline of steps that practitioners typically follow.

1. **Installation and setup**: Begin by installing TensorFlow Federated via

   “‘ pip install tensorflow-federated “‘

   This environment includes the libraries necessary to simulate and test federated workflows.

2. **Defining the Global Model**: Specify an initial global model, typically using

   TensorFlow's Keras API. This model will be distributed to individual clients for local training.

3. **Simulating FL Clients**: Partition a centralized dataset into multiple subsets, each representing a unique client. TFF's simulation API makes it possible to emulate the diversity of real-world client data within a controlled environment.

4. **Implementing FedAvg Aggregation**: Employ Federated Averaging (FedAvg)—or a related approach—to aggregate client updates. This step involves combining client-generated gradients or weights into a refined global model, which is then redistributed for subsequent training rounds.

5. **Optimizing Communication Efficiency**: Enhance performance in bandwidth-constrained settings through techniques such as gradient sparsification or quantization. These optimizations limit the size and frequency of updates without unduly sacrificing model accuracy.

By following these steps within TFF, developers and researchers can iteratively improve their FL configurations, adapting to project-specific requirements such as dataset size, device heterogeneity, or privacy needs.

1.7.2 Debugging and testing strategies

Debugging an FL system poses unique hurdles, as developers are typically restricted from accessing the raw data held by participating clients. Consequently, traditional ML debugging practices must be adapted to respect these privacy constraints.

Real-Time Logging for Communication Latency: Monitoring and logging communication metrics can help identify bottlenecks or connectivity issues, especially crucial in large-scale or heterogeneous FL deployments.

Gradient Anomaly Detection: Since malicious clients or system errors may produce aberrant gradients, anomaly detection mechanisms can flag suspicious updates, helping developers diagnose and isolate potential attacks or misconfigurations early.

Simulated Edge Cases: Testing for adversarial conditions—such as intermittent network connections, faulty devices, or partial participation—helps to ensure the FL system remains robust. Synthetic scenarios also offer a way to explore the resilience of the system without risking production data or violating privacy rules.

These strategies enable systematic problem solving while respecting client confidentiality, ensuring that FL solutions can be fine-tuned to handle real-world complexities.

1.7.3 Hands-on FL Projects

For students and professionals seeking practical experience with Federated Learning, several project avenues stand out:

Privacy-Preserving FL for Healthcare: Investigate how differential privacy can be integrated into medical imaging tasks to protect patient identities while enabling collaborative model training across multiple hospitals.

Low-Bandwidth FL Optimization: Focus on gradient sparsification or model prun-

ing techniques to reduce communication overhead in IoT contexts. Such projects are particularly relevant for smart devices with limited power or connectivity.

Adversarial Robustness in FL: Design experiments to expose FL models to poisoning attacks and evaluate how countermeasures—such as robust aggregation or adversarial training—can be employed to defend against malicious updates.

By pursuing these project ideas, learners gain exposure to core challenges in FL—ranging from safeguarding sensitive data to optimizing performance under constrained resources—and solidify their understanding of privacy, debugging, and communication strategies.

1.8 Obstacles and Future Trends

Federated Learning (FL) has demonstrated significant potential in privacy-preserving, decentralized machine learning, but its scalability, efficiency, and security challenges remain active research areas in **Parallel and Distributed Computing (PDC)**. The complexity of coordinating learning across heterogeneous devices, optimizing communication efficiency, and ensuring robust security presents key technical hurdles that require innovative solutions.

This section explores the open challenges in PDC-driven FL and highlights **opportunities for innovation**, particularly in privacy-preserving algorithms and large-scale energy-efficient architectures.

1.9 Open Challenges in PDC-Focused Federated Learning

1.9.1 Handling Heterogeneity in FL: System, Computational, and Network Variability

A fundamental challenge in FL is the heterogeneity across client devices, which significantly impacts computational efficiency, convergence rates, and overall system performance. Unlike traditional centralized machine learning, where model training occurs in controlled environments, FL involves:

Diverse hardware capabilities: Clients range from high-end GPUs to low-power mobile or IoT devices, resulting in variable processing speeds and memory constraints. **Un-**

reliable network connectivity: Clients may experience differing bandwidth capacities, causing communication bottlenecks. **Asynchronous updates**: Stragglers (slow clients) may delay global model aggregation, while fast clients risk stale updates.

1.9.2 Potential Solutions

Adaptive Client Selection: Dynamically choosing a subset of clients per training round based on computational power, network speed, and data diversity to balance efficiency and representativeness. Hierarchical FL (HFL): Introducing intermediate aggregators at different network levels (e.g., edge devices) to reduce communication delays and improve system scalability. **Personalized FL Models**: Implementing meta-learning or clustered FL approaches, where clients with similar data distributions train specialized models to improve convergence.

1.9.3 Managing Non-IID and Skewed Data Distributions

Federated Learning often operates in environments where clients possess Non-Independent and Identically Distributed (Non-IID) data, meaning datasets are highly imbalanced and diverse across devices. This leads to:

Global model instability: If client updates are overly biased toward their local distributions, the global model may fail to generalize across different users. **Slow convergence and fairness issues**: Models trained on underrepresented data distributions may disproportionately benefit certain clients, leading to ethical concerns.

1.9.4 Potential Solutions

Federated Transfer Learning (FTL): Leveraging knowledge transfer techniques to improve model adaptation across clients with different data distributions. **Gradient Clipping and Regularization Techniques**: Enforcing bounds on weight updates to prevent dominant clients from skewing the global model. **Bayesian Aggregation Methods**: Adjusting model aggregation weights based on the statistical distribution of client datasets to

improve generalization.

1.9.5 Opportunities for Innovation in PDC and FL

- Advancing Privacy-Preserving Distributed Algorithms.

  Although FL mitigates some privacy risks by keeping data decentralized, recent research has shown that model updates can still leak sensitive information through inference attacks. Enhancing privacy in FL requires innovations in privacy-preserving distributed computing techniques such as:

  A. Secure Aggregation Techniques - Differential Privacy (DP) in FL: Adding adaptive noise to model updates before aggregation to prevent membership inference attacks while maintaining model utility. - Homomorphic Encryption (HE): Allowing computations on encrypted gradients without exposing individual client updates. - Secure Multi-Party Computation (SMPC): Splitting model updates into encrypted shares distributed across multiple parties for privacy-preserving aggregation.

  B. Blockchain-Assisted Federated Learning - Decentralized FL Aggregation: Leveraging blockchain networks to maintain immutable and verifiable records of model updates without requiring a central aggregator. - Smart Contract-Based Model Validation: Implementing trustless FL environments where secure aggregation occurs via programmable cryptographic contracts.

  These privacy-preserving strategies will enhance security in highly sensitive applications, including healthcare, finance, and edge AI deployments.

- Designing Energy-Efficient, Large-Scale FL Architectures

  FL is often deployed in resource-constrained environments, such as IoT devices, smartphones, and edge computing platforms. Training large-scale models across millions of nodes requires efficiency-optimized architectures to minimize computation, communication, and energy consumption.

A. Communication-Efficient FL Protocols - Gradient Sparsification and Quantization: Transmitting only the top-k significant gradients instead of full updates reduces communication overhead. - Adaptive FL Training Rounds: Dynamically adjusting client update frequencies based on network conditions and computation resources.

B. Energy-Aware FL Optimization - Federated Knowledge Distillation (FKD): Training smaller models on resource-constrained devices while distilling knowledge from larger, more powerful models. - Distributed Model Compression Techniques: Applying model pruning, tensor decomposition, and federated dropout to reduce computational burden.

These approaches will be essential for scaling FL to millions of connected devices, supporting applications in 5G, smart cities, and autonomous AI systems.

1.10 Conclusion and Future Research Directions

The integration of **Federated Learning with Parallel and Distributed Computing** presents exciting challenges and opportunities for the next generation of decentralized AI systems. Future research should focus on:

Optimizing FL for Non-IID data using adaptive aggregation and client selection techniques Enhancing privacy in FL through secure aggregation, homomorphic encryption, and blockchain-based trust models Reducing FL communication overhead via gradient sparsification, quantization, and federated dropout strategies Scaling FL to energy-constrained environments with federated knowledge distillation and model compression techniques

By addressing these challenges, FL can evolve into a scalable, secure, and energy-efficient framework for real-world AI applications, bridging the gap between privacy-preserving learning and large-scale distributed intelligence.

Bibliography

[1] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *Foundations and trends® in machine learning*, 14(1–2):1–210, 2021.

[2] Sandeep Kumar, Ketan Rajawat, and Daniel P Palomar. Distributed inexact successive convex approximation admm: Analysis-part i. *arXiv preprint arXiv:1907.08969*, 2019.

[3] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

[4] Naoya Yoshida, Takayuki Nishio, Masahiro Morikura, Koji Yamamoto, and Ryo Yonetani. Hybrid-fl for wireless networks: Cooperative learning mechanism using non-iid data. In *ICC 2020-2020 IEEE International Conference On Communications (ICC)*, pages 1–7. IEEE, 2020.

APPENDIX

.1 Tutorial

Federated learning (FL) is a decentralized machine learning approach in which multiple clients (devices or nodes) collaboratively train a model without sharing their raw data.

This is in contrast to traditional centralized learning, where all training data is aggregated on a central server or in the cloud to train a single model. FL was introduced to address privacy and data governance concerns: clients send only model updates (e.g. learned parameters or gradients) to a central server, which aggregates them, rather than sending raw data. This appendix provides a step-by-step tutorial comparing federated learning and centralized learning on the MNIST dataset using PyTorch for model implementation and Flower for the federated learning framework.

We will first implement a centralized training baseline in MNIST (all data on one server). Then, we will set up a simulated federated learning experiment using Flower, where multiple virtual clients train on non-IID partitions of MNIST (each client will have a different subset of the data distribution). By comparing the two approaches, you will see how federated learning performs relative to a conventional centralized approach, and understand the trade-offs involved. The tutorial assumes an undergraduate-level understanding of Python and basic machine learning (e.g., training a simple neural network). All code is written in Python using PyTorch for the model and training, and the Flower framework to handle federated learning orchestration. The authors appreciate the Flower team for developing the open source Flower framework and for providing inspiration for this tutorial.

.2 Installation and Setup

To follow along, you will need to install Python and several libraries (PyTorch, Flower, etc.). The following instructions are applicable to Windows, macOS, and Linux:

1. **Install Python 3.8+**: If not already installed, download Python from the official

website (python.org) and follow the installer for your operating system. On Windows and macOS, make sure that you check the option to add Python to your PATH. On Linux, you can use your package manager (e.g., sudo apt-get install python3).

2. **Create a virtual environment (optional)**: It is good practice to use a virtual environment for projects. You can create one using Python's venv module or Conda. For example, in a terminal:

```
python3 -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate

```

This step is optional, but helps to isolate the dependencies of the project.

3. **Install PyTorch and TorchVision:** PyTorch is the deep learning library that we use to build and train the neural network, and TorchVision will provide the MNIST dataset. Install them using pip:

```
pip install torch torchvision
```

This installs the CPU version of PyTorch by default, which is sufficient for this tutorial. (If you have a GPU and want to use it, visit the PyTorch website and follow the instructions to install a version with CUDA support, as the pip command might need a specific wheel or use Conda.)

4. **Install Flower and other libraries**: Flower is the federated learning framework. We also need matplotlib and scikit-learn for visualization and evaluation. Install them with:

```
pip install flwr matplotlib scikit-learn
```

This will pull in Flower (for federated learning simulation), Matplotlib (for plotting graphs), and scikit-learn (for generating a confusion matrix and other metrics).

After these steps, you should have all necessary packages installed. You can verify by launching a Python shell and importing torch and flwr (import torch, flwr) - if there are no errors, you are set up. Now we can proceed to implement the centralized and federated learning experiments.

.3 Centralized Learning on MNIST

First, we implement the centralized learning baseline. In centralized learning, we have a single server that holds all the training data and trains a model on it. This is the traditional approach to machine learning. We will use the MNIST dataset, which consists of 60,000 training images of handwritten digits (0 through 9) and 10,000 test images. Each image is 28x28 pixels in grayscale and the task is to classify the digit. We will build a simple convolutional neural network (CNN) in PyTorch, train it on the entire MNIST training set, and evaluate it on the test set.

1. **Data Loading and Preparation**: We will use torchvision to load the MNIST dataset. TorchVision will download MNIST if it is not already available. We will normalize the images and set up DataLoaders for training and testing. The code below loads MNIST and prepares data loaders:

```
    import torch
    from torchvision import datasets, transforms
    from torch.utils.data import DataLoader

    # Device configuration: use GPU if available, else CPU
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print("Using device:", device)

    # Define transform: convert images to tensor and normalize pixel
    values
    transform = transforms.Compose([
        transforms.ToTensor(),  # convert image to PyTorch tensor
```

```
        transforms.Normalize((0.1307,), (0.3081,))  # normalize with MNIST mean and std
    ])

    # Download and load the MNIST training and test sets
    trainset = datasets.MNIST(root="./data", train=True, download=True, transform=transform)
    testset = datasets.MNIST(root="./data", train=False, download=True, transform=transform)

    # Create DataLoader for batching
    trainloader = DataLoader(trainset, batch_size=64, shuffle=True)
    testloader = DataLoader(testset, batch_size=1000, shuffle=False)

    print(f"Loaded MNIST dataset with {len(trainset)} training examples and {len(testset)} test examples.")

```

Running the above will download MNIST (if not present) and print out the number of examples. We use a batch size of 64 for training and 1000 for testing (larger batch for evaluation to speed up). We also check for a GPU and use it if it is available to accelerate training.

2. **Defining the Model** Next, we define a simple neural network for classifying MNIST digits. We'll use a Convolutional Neural Network (CNN) since it performs well on images. Our model will have two convolutional layers (with ReLU activations and pooling) followed by two fully-connected (Linear) layers. This is a lightweight network suitable for demonstration:

```
        import torch.nn as nn
import torch.nn.functional as F

# Define a simple CNN model for MNIST
class Net(nn.Module):
```

```
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, padding=1)    # 1
    input channel (grayscale), 16 filters
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)  # 32
    filters
        self.pool = nn.MaxPool2d(2, 2)  # 2x2 max pooling
        self.fc1 = nn.Linear(32 * 7 * 7, 128)  # 32 feature maps * 7x7
    after pooling
        self.fc2 = nn.Linear(128, 10)         # 10 output classes (digits
     0-9)

    def forward(self, x):
        # Two convolutional layers with ReLU and pooling
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 32 * 7 * 7)      # flatten
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Initialize the network and move to device
model = Net().to(device)
```

This defines our CNN. The output of model(x) will be a vector of length 10 with raw scores for each class. We will use the Cross-Entropy loss in training, which internally applies a softmax on these scores and computes the error against the true label.

3. **Training the Model (Centralized)** Now, we train the CNN on the full MNIST training set. We'll use Stochastic Gradient Descent (SGD) optimizer. For simplicity, we train for a few epochs (e.g., 5) and monitor the training loss and accuracy.

```
        import torch.optim as optim

# Loss function and optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Training loop for a certain number of epochs
epochs = 5
for epoch in range(1, epochs+1):
    model.train()  # set model to training mode
    running_loss = 0.0
    correct = 0
    total = 0
    for batch_idx, (images, labels) in enumerate(trainloader):
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()             # reset gradients
        outputs = model(images)           # forward pass
        loss = loss_fn(outputs, labels)   # compute loss
        loss.backward()                   # backpropagate
        optimizer.step()                  # update weights

        running_loss += loss.item()
        # compute training accuracy for this batch (optional)
        preds = outputs.argmax(dim=1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)
    avg_loss = running_loss / len(trainloader)
    accuracy = correct / total
    print(f"Epoch {epoch}: Loss = {avg_loss:.4f}, Training Accuracy = {
   accuracy*100:.2f}%")


```

In each epoch, we iterate over the training DataLoader. We accumulate the loss to

compute an average, and also calculate how many predictions were correct to get the training accuracy. After running this loop, the model's parameters have been updated through SGD. You should see the loss decreasing and training accuracy increasing over epochs (e.g., it might reach around 98 pourcents training accuracy after 5 epochs).

4. **Evaluation on Test Set** After training, we evaluate the model on the unseen test set of 10,000 images to measure how well the model generalizes. We will calculate the overall test accuracy and also display a confusion matrix to see the breakdown of correct and incorrect classifications for each digit.

```
        from sklearn.metrics import accuracy_score,
   ConfusionMatrixDisplay

model.eval()  # set model to evaluation mode
y_true = []
y_pred = []
with torch.no_grad():
    for images, labels in testloader:
        images = images.to(device)
        outputs = model(images)
        preds = outputs.argmax(dim=1).cpu().numpy()
        y_pred.extend(preds)
        y_true.extend(labels.numpy())

# Calculate accuracy
test_accuracy = accuracy_score(y_true, y_pred)
print(f"Test Accuracy (Centralized model): {test_accuracy*100:.2f}%")

# Plot confusion matrix
ConfusionMatrixDisplay.from_predictions(y_true, y_pred, cmap="Blues",
   display_labels=list(range(10)))

```

This will print the test accuracy and display a confusion matrix for the 10 digit classes.

The confusion matrix is a 10x10 grid where the cell in row i and column j shows how many examples of true class i were predicted as class j. The diagonal cells (where i=j) are the correctly classified counts for each digit. Ideally, we want large numbers on the diagonal and small numbers off-diagonal. For example, you might observe that the centralized model achieves around 98-99 percent test accuracy after full training (since MNIST is a relatively easy dataset for a CNN). The confusion matrix would mostly have high counts on the diagonal, meaning most digits are classified correctly, with only a few misclassifications (e.g., a few 5's mistaken for 3's, etc.). This centralized result will serve as a reference when we later evaluate the federated learning approach. Now that we have a strong centralized baseline, let us move on to federated learning and see how to simulate it using Flower.

.4 Federated Learning with Flower

In a federated learning scenario, the training data is not all in one place. Instead, it is distributed across multiple clients. Each client will train the model on its local data and only send model updates (not raw data) to the central server. The server will aggregate these updates (for example, by averaging them, as in the FedAvg algorithm.

) to update the global model. This process repeats for multiple rounds. In the end, the global model has learned from all data from the clients, without the data ever leaving the clients. In this tutorial, we simulate the federated setting using Flower's simulation capabilities. We will create a number of MNIST client datasets (each representing a client's local data). To make the scenario challenging, we use a non-IID data split: each client will only have examples of a few-digit classes, rather than a perfectly balanced mix. Non-IID (nonidentically and independently distributed) means that the clients' data distributions differ – a realistic scenario in federated learning (e.g., different users might have different types of data on their devices).

We will then define a Flower client class that can train our PyTorch model on local data and communicate with the Flower server. Finally, we will run the federated simulation for

a few rounds and evaluate the global model.

Simulating Non-IID Data Partitioning We want to partition the MNIST training set among (simulated) clients in a non-iid way. Let us say we simulate 5 clients. We will split the 10 digit classes among these 5 clients so that each client sees only a subset of the classes. For example: Client 0 gets images of digits 0 and 1. Client 1 gets images of digits 2 and 3. Client 2 gets images of digits 4 and 5. Client 3 gets images of digits 6 and 7. Client 4 gets images of digits 8 and 9. This means that the data of each client is not representative of the overall distribution (each has only two out of 10 classes, which is a highly noni.i.d. scenario). In real-world FL, non-iid could be less extreme, but this set-up will clearly illustrate the challenges. We will use the original trainset we loaded and split its indices according to the labels.

```
    import numpy as np
from torch.utils.data import Subset

# Number of clients in our federated simulation
NUM_CLIENTS = 5

# Create a dictionary to hold indices for each client
clients_indices = {i: [] for i in range(NUM_CLIENTS)}
for idx, (_, label) in enumerate(trainset):
    # Assign index to a client based on the label
    if label in [0, 1]:
        clients_indices[0].append(idx)
    elif label in [2, 3]:
        clients_indices[1].append(idx)
    elif label in [4, 5]:
        clients_indices[2].append(idx)
    elif label in [6, 7]:
        clients_indices[3].append(idx)
    else:
        clients_indices[4].append(idx)

```

```
# Use the indices to create subset datasets for each client
client_datasets = []
for i in range(NUM_CLIENTS):
    client_datasets.append(Subset(trainset, clients_indices[i]))
    print(f"Client {i} has {len(clients_indices[i])} samples, labels: {set(
    trainset.targets[clients_indices[i]].numpy())}")
```

In the loop above, we iterate through every example in the MNIST training set and assign it to one of the 5 clients based on its label. The result will be five subsets of the training data. The printout will show the number of samples each client has and the labels that are present (e.g., client 0 might have 12,000 samples of labels 0,1, Client 1 has 12,000 of 2,3, etc., since MNIST has about 6000 images per digit). The total across clients is still 60,000 (we have just partitioned them). Non-IID check: The set of labels for each client is disjoint from the others (none has all digits). This will make federated training more challenging, as no single client can by itself learn to recognize all 10 digits – they must collaborate via the central server. **Defining a Flower Client Class** Flower provides an API to define custom clients. We will create a class MNISTClient that extends Flower's flwr.client.NumPyClient. The NumPyClient is a convenient flower client abstraction where we can send/receive model parameters as NumPy arrays. Our client will need to do three main things:

- **get-parameters**: return the current local model parameters (so that the server can initialize or update models).
- **fit**: train the model on the local data of the client for one round (one or more epochs) and return the updated parameters.
- **evaluate**: (optional in simulation) evaluate the model on the client's data and return metrics (not strictly necessary if we perform a centralized evaluation on the server).

Each client will have its own instance of the CNN model (the same architecture as defined above). At the start of each round, the Flower server will send the latest global model parameters to each client (Flower will call client.fit(parameters) with those parameters).

The client will load those into its model, train on its local client dataset, and then return the new weights. Flower will aggregate these from all clients. We also pass each client a reference to the global test set for evaluation purposes (or we could have the server handle evaluation centrally — we will do both to illustrate). Let us implement the client class:

```
    import flwr as fl

class MNISTClient(fl.client.NumPyClient):
    def __init__(self, cid, train_data, test_data=None):
        self.cid = cid  # client ID (string in Flower)
        self.trainloader = DataLoader(train_data, batch_size=32, shuffle=True)
        self.testloader = DataLoader(test_data, batch_size=1000, shuffle=False
    ) if test_data else None
        # Each client has its own model and optimizer
        self.model = Net().to(device)
        self.optimizer = optim.SGD(self.model.parameters(), lr=0.02)
        self.loss_fn = nn.CrossEntropyLoss()

    def get_parameters(self, config=None):
        """Return current model parameters as a list of numpy arrays."""
        self.model.eval()
        params = [param.cpu().numpy() for param in self.model.parameters()]
        return params

    def fit(self, parameters, config=None):
        """Receive global model parameters, train on local data, return
    updated parameters."""
        # Load global weights into the local model
        for param, new_val in zip(self.model.parameters(), parameters):
            param.data = torch.tensor(new_val, device=device)
        # Train for 1 epoch on local dataset
        self.model.train()
        for images, labels in self.trainloader:
            images, labels = images.to(device), labels.to(device)
```

```
            self.optimizer.zero_grad()
            outputs = self.model(images)
            loss = self.loss_fn(outputs, labels)
            loss.backward()
            self.optimizer.step()
        # After training, get updated weights
        updated_params = [param.cpu().numpy() for param in self.model.
    parameters()]
        # Optionally, we can return the number of examples used (for weighting
    ) and any metrics
        num_samples = len(self.trainloader.dataset)
        return updated_params, num_samples, {}

    def evaluate(self, parameters, config=None):
        """Evaluate the current model on local test data (if provided)."""
        # Load global weights into model
        for param, new_val in zip(self.model.parameters(), parameters):
            param.data = torch.tensor(new_val, device=device)
        self.model.eval()
        if not self.testloader:
            return 0.0, 0, {"accuracy": 0.0}
        correct = 0
        total = 0
        loss = 0.0
        with torch.no_grad():
            for images, labels in self.testloader:
                images, labels = images.to(device), labels.to(device)
                outputs = self.model(images)
                loss += self.loss_fn(outputs, labels).item()
                preds = outputs.argmax(dim=1)
                correct += (preds == labels).sum().item()
                total += labels.size(0)
        accuracy = correct / total
        loss = loss / len(self.testloader)
```

```
60         return loss, total, {"accuracy": accuracy}
```

- Each MNISTClient is initialized with a training dataset (train-data) and optionally a test-data. In our case, we will pass the global test set to each client for simplicity in evaluation (so self.testloader is the entire MNIST test set for all clients – this is just for convenient evaluation in simulation; in real FL, clients would likely only have local validation data or none at all).

- fit: We load the model weights provided by the server (parameters argument) into the local model. We then do one epoch of training (for images, labels in self.trainloader). Here we chose local epochs = 1 for each round to keep it quick; this can be adjusted (clients could train for multiple local epochs per round). We use a small batch size (32) for local training. After training, we gather the updated model parameters and return them along with the number of samples used (num-samples). Flower will use num-samples to perform weighted aggregation (FedAvg weighs each client's update by the number of training samples, so clients with more data have a proportional influence.

- evaluate: Flower can use this method to obtain evaluation metrics from clients. We implemented it to compute the local accuracy on the test set. However, in a typical federated setup, one might instead do a server-side evaluation on a global test set (since each client might not have a representative test set). Flower allows centralized evaluation via an evaluate-fn on the server, which we will use as well. We include an evaluation here for completeness, but later we will primarily rely on a server-side evaluation for the global model.

.5 Running the Federated Training Simulation

Now we have the client data sets and the client class. We can use Flower to run a simulation of the federated learning process. Flower's simulation will create multiple client instances and coordinate the training rounds. We need to define:

- A function to instantiate clients (given a client ID, return an MNISTClient for that ID).
- A strategy for the server, e.g., FedAvg (Federated Averaging), which Flower provides by default.
- (Optionally) an evaluation function on the server to evaluate the global model on the test set each round.

We will use flwr.simulation.start-simulation for simplicity, which runs the whole federated training loop in the process (suitable for notebooks or single-machine simulations). We will configure it to use all 5 clients each round (since we have only 5 and we want them all to participate in every round in this demo). We also set up a server-side evaluation function to compute the global accuracy after each round using the aggregated model. This will help us track performance over rounds and eventually plot accuracy vs. rounds.

```
    # Create a global model and test DataLoader for server-side evaluation
global_model = Net().to(device)
global_testloader = DataLoader(testset, batch_size=1000, shuffle=False)

# Define a server-side evaluation function
acc_history = []  # to record accuracy each round

def evaluate_global(parameters: fl.common.NDArrays, config):
    # Load parameters into the global model
    param_tensors = [torch.tensor(np.array(p), device=device) for p in
   parameters]
    for param, new_val in zip(global_model.parameters(), param_tensors):
        param.data = new_val
    # Compute test accuracy and loss on the global test set
    global_model.eval()
    total, correct = 0, 0
    loss = 0.0
    with torch.no_grad():
```

```
        for images, labels in global_testloader:
            images, labels = images.to(device), labels.to(device)
            outputs = global_model(images)
            loss += loss_fn(outputs, labels).item()
            preds = outputs.argmax(dim=1)
            correct += (preds == labels).sum().item()
            total += labels.size(0)
    accuracy = correct / total
    loss = loss / len(global_testloader)
    acc_history.append(accuracy)
    print(f"Server-side evaluation - Round accuracy: {accuracy*100:.2f}%")
    # Return (loss, metrics) as required by Flower
    return loss, {"accuracy": accuracy}

# Define client creation function
def client_fn(cid: str):
    cid = int(cid)
    return MNISTClient(cid, train_data=client_datasets[cid], test_data=testset
    )

# Configure the federated learning strategy (FedAvg) and run simulation
strategy = fl.server.strategy.FedAvg(
    fraction_fit=1.0,  # use all clients each round
    fraction_evaluate=0.0,  # we'll do centralized evaluation
    min_fit_clients=NUM_CLIENTS,
    min_available_clients=NUM_CLIENTS,
    evaluate_fn=evaluate_global  # use our server-side evaluation function
)

# Run the simulation for a few rounds
NUM_ROUNDS = 5
history = fl.simulation.start_simulation(
    client_fn=client_fn,
    num_clients=NUM_CLIENTS,
```

```
    config=fl.server.ServerConfig(num_rounds=NUM_ROUNDS),
    strategy=strategy
)
```

Now,let us break down what is happening in the code above:

- We created a global model (the same architecture) that we use for evaluation. This model will be updated with the latest global parameters each round in evaluate-global.
- evaluate-global takes the global model parameters (as a list of NumPy arrays) from Flower and evaluates the model on the entire test set. It prints the accuracy for each round and stores it in acc-history for later plotting. Returns the loss and accuracy metrics to Flower (Flower can log or use them, though in this simulation it is mainly for our own observation).
- client-fn is a factory function that Flower uses to initialize clients by ID. Flower will call this for each client it needs (IDs will be "0", "1", ... as strings). We convert the ID to int and return an MNISTClient with the corresponding partition of data.
- We configure the FedAvg strategy:
  - fraction-fit=1.0 means 100 percents of available clients are used for training each round (so all 5 clients train each round).
  - fraction-evaluate=0.0 means that we are not sampling clients for evaluation; instead, we rely on evaluate-fn for centralized evaluation. We set min-available-clients=5 to ensure that all 5 must be present.
  - We provide our evaluate-fn (evaluate-global) to compute metrics on the server side.
- Finally, we start the simulation for NUM-ROUNDS (e.g., 5 rounds). Flower will output logs for each round, showing which clients were selected (0-4 each time, in our case) and the results of training and evaluation.

During each round:

- Flower will send the current global model parameters to all clients (initially the model is randomly initialized).
- Each client runs a fit for one epoch on its local data and returns updated parameters.
- The server (Flower) aggregates these updates (FedAvg: effectively a weighted average by data set size).
- The server then evaluates the aggregated model using evaluate-global, printing the accuracy.
- That completes one round; the updated global model is then sent out again for the next round.

After 5 rounds, the simulation ends. The history object returned can contain details (e.g., metrics history), but we already stored accuracy in acc-history and printed it each round.

.6 Evaluating the Federated Model

Let's examine the results of federated training. After running the above simulation, you will see output for each round such as:

```
    Round 1: fit progress...
Server-side evaluation - Round accuracy: XX.XX%
Round 2: fit progress...
Server-side evaluation - Round accuracy: YY.YY%
...
```

Due to the non-IID split, the initial rounds may yield a lower accuracy. For example, after Round 1, the global model might only reach modest accuracy (since each client trained on only two classes, the aggregated model has seen all classes but only one epoch of data from each). As the rounds progress, the accuracy should improve as the model refines.

Performance visualization:

```
    # After training, plot accuracy vs. rounds
import matplotlib.pyplot as plt

rounds = range(1, len(acc_history) + 1)
plt.figure()
plt.plot(rounds, [a*100 for a in acc_history], marker='o')
plt.title("Federated Learning: Accuracy vs Rounds")
plt.xlabel("Round")
plt.ylabel("Test Accuracy (%)")
plt.grid(True)
plt.show()

# Compute final confusion matrix for federated model
y_true_fed, y_pred_fed = [], []
global_model.eval()
with torch.no_grad():
    for images, labels in testloader:
        images, labels = images.to(device), labels.to(device)
        outputs = global_model(images)
        preds = outputs.argmax(dim=1).cpu().numpy()
        y_pred_fed.extend(preds)
        y_true_fed.extend(labels.cpu().numpy())
ConfusionMatrixDisplay.from_predictions(y_true_fed, y_pred_fed, cmap="Blues",
   display_labels=list(range(10)))
plt.title("Federated Model Confusion Matrix")
plt.show()

print(f"Final Federated Model Test Accuracy: {acc_history[-1]*100:.2f}%")
```

This will produce a plot of test accuracy over the rounds and the confusion matrix for the final federated model. With our configuration (5 clients, each with 2 classes, 1 local epoch per round, 5 rounds), you might find that the final accuracy is a bit lower than the

centralized model. For instance, the federated model might reach around 90+ accuracy after 5 rounds in this non-IID scenario (the exact number may vary). If you increase the number of rounds or local epochs, the accuracy can be further improved. Similarly, if the data split were IID (each client having a mix of all classes), federated learning would reach centralized performance more easily.

The **confusion matrix** for the federated model will probably show more confusion between certain classes compared to the confusion matrix for the centralized model. For example, because some clients never saw certain classes, the global model needed a few rounds to start recognizing them. You might notice more mistakes for those classes early on, gradually improving by the final round. The trend of accuracy vs. rounds should show an upward trajectory, illustrating that as we perform more federated rounds, the global model gets better.

Summary: Federated learning, as shown by our Flower simulation, achieves a comparable model without centralized data collection. However, it required careful orchestration (multiple rounds) and had slightly lower accuracy in this limited-round experiment due to the challenging non-IID partition. With additional rounds or techniques to handle non-IID data, federated models can approach centralized performance,all while preserving user data privacy. This tutorial illustrated the end-to-end process using Flower – from setting up clients to aggregating updates. The Flower framework made it straightforward to simulate FL: we were able to reuse the same PyTorch model and training code inside a federated setting with relatively few changes, primarily by wrapping it in the Flower client API.

Quiz: Knowledge Check

To ensure an understanding of the key concepts of this tutorial, try answering the following multiple-choice questions:

1. **Where is the training data stored in a federated learning system?**

1. **Where is the training data stored in a federated learning system?**

A. All training data are sent to a central server.

B. Training data remains distributed on client devices.

C. Training data is stored in a shared cloud database accessible by all clients.

D. Half of the data are on the server and half on the clients.

Answer: B

2. What is a primary advantage of federated learning compared to centralized learning?

A. Requires more communication overhead.

B. Allows model training without sharing user raw data, improving privacy.

C. Reduces the need for robust security measures.

D. Centralizes the data for easier management.

Answer: B

3. In a cross-silo federated learning setup:

A. Data comes from multiple large, diverse organizations.

B. Data is often from individual mobile or IoT devices.

C. Data is stored in a single data center.

D. None of the above.

Answer: A

4. Which of the following is an example of a state-of-the-art optimization method in federated learning?

A. Federated Averaging (FedAvg)

B. Stochastic Gradient Descent (SGD)

C. Adam

D. FedProx

Answer: D

5. Compared to centralized learning, federated learning generally:

A. Has lower communication overhead.

B. Has no need for data security.

C. Often needs a carefully weighted average of client model updates, ignoring dataset sizes.

D. Requires more data communication and a more complex training setup.

Answer: B