# SplitFed - Federated Learning meets Split Learning

Mobaswirul Islam
mobaswir.shafi@gmail.com
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh

## Abstract

SplitFed Learning (SFL) is a hybrid approach that combines the advantages of both Federated Learning (FL) and Split Learning (SL). Federated Learning enables multiple clients to train a model simultaneously without sharing their data. However, since the entire model is trained on the client side, it may lack robustness, as most clients have limited computational resources. Split Learning, on the other hand, divides the model into two parts: the Client Layer and the Server Layer. This approach allows the use of more complex models and enhances security. However, SL can be slow and requires significant communication overhead. SplitFed Learning leverages the parallel computation capability of Federated Learning and the robust, secure model structure of Split Learning, while also addressing their respective limitations.

In this project, we used a server-client setup to test SFL. Our results showed that SFL is almost as accurate as SL but much faster. This makes SFL useful for areas like healthcare and fraud detection, where data security is important. Future improvements could focus on making SFL even faster and more secure by fine-tuning settings and adding encryption.

***Keywords:*** Distributed Learning, Federated Learning, Split Learning, Split Fed Learning

## 1 Introduction

Machine learning is used in many fields, such as healthcare, finance, and security, to make smart predictions based on large amounts of data. However, sharing data for training machine learning models can lead to privacy and security issues. Traditionally, data is collected in one place and used to train models, but this can expose sensitive information. To solve this problem, researchers have developed new ways to train models without sharing raw data.


*Distributed Computing Systems, BUET, BD*

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

Federated Learning (FL) and Split Learning (SL) are two popular methods that help protect data privacy. FL allows multiple devices or computers to train a model together without sending their data to a central server. Instead, each device trains a copy of the model using its own data and then shares only the model updates. This way, sensitive data stays on the device, reducing privacy risks. However, FL requires a lot of computing power on each device because it trains the full model locally.

On the other hand, Split Learning (SL) divides the model into two parts: one part runs on the client (device), and the other part runs on a central server. This approach reduces the computational load on devices, making it more suitable for resource-limited environments. However, SL has its own problems, such as slower training times because only one device communicates with the server at a time.

To overcome the limitations of both FL and SL, SplitFed Learning (SFL) was introduced. SFL combines the advantages of both methods by allowing devices to train models in parallel while keeping data secure. It helps in speeding up training and ensures privacy by preventing direct access to raw data. In this project, we explore how SFL improves efficiency, privacy, and security while making machine learning more accessible to devices with lower computing power.

## 2 Background

The increasing reliance on machine learning (ML) across industries has raised concerns regarding data privacy, security, and computational efficiency. Traditional machine learning models require centralized data collection for training, which can pose significant risks such as data breaches, compliance violations, and high transmission costs. To address these issues, researchers have developed distributed learning approaches, which allow collaborative model training without requiring data centralization.

Among the notable distributed learning techniques, Federated Learning (FL) and Split Learning (SL) have emerged as effective solutions. These methods aim to balance data privacy, computational efficiency, and model performance. However, each has its own set of advantages and limitations.

### 2.1 Federated Learning (FL)

Federated Learning (FL) is a technique that enables multiple devices (clients) to train a shared machine learning model without transferring raw data to a central server. Instead, clients train their local models and send only the learned

| Approach | Aggregation | Privacy advantage | Client-side training | Distributed | Access to raw data |
|---|---|---|---|---|---|
| SL | No | Yes | Sequential | Yes | No |
| FL | Yes | No | Parallel | Yes | No |
| SFL | Yes | Yes | Parallel | Yes | No |

**Table 1.** Comparison of Learning Approaches

parameters (e.g., gradients) to a central server, where they are aggregated to form a global model.

**Key Characteristics of FL**:

- **Local Training:** Clients independently train their models on their own data.
- **Model Aggregation:** A central server combines model updates from all clients using methods like Federated Averaging (FedAvg).
- **Parallel Processing:** Clients train in parallel, which speeds up the learning process.

**Advantages of FL**:

- Ensures data privacy by keeping raw data on client devices.
- Reduces communication overhead compared to centralized learning.
- Allows training on distributed edge devices such as mobile phones and IoT sensors.

**Challenges of FL**:

- Requires sufficient computational power on client devices to train full models.
- May suffer from non-independent and identically distributed (non-IID) data, leading to biased models.
- The central server still has access to aggregated model updates, which can pose potential security risks.

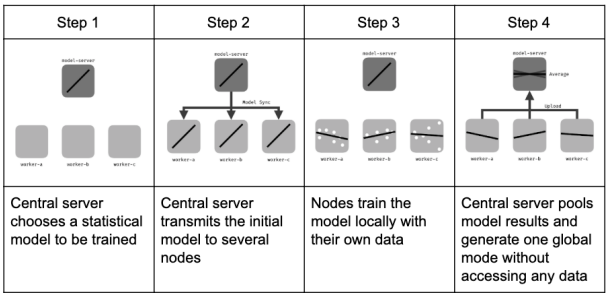


**Figure 1.** Federated Learning Workflow

**Key Characteristics of SL**:

- **Model Partitioning:** The neural network is split between the client and the server.
- **Smashed Data Transmission:** Clients send intermediate activations (smashed data) to the server, which completes the forward and backward pass.
- **Sequential Training:** Unlike FL, clients train in a sequential manner, waiting for server responses.

**Advantages of SL**:

- Reduces computational load on client devices.
- Enhances model privacy, as clients do not store full models.
- Useful for applications with strict privacy requirements, such as healthcare and finance.

**Challenges of SL**:

- Training is slower due to the sequential nature of client-server interactions.
- High communication overhead as clients frequently send and receive smashed data.
- Scaling to multiple clients is challenging, as only one client interacts with the server at a time.

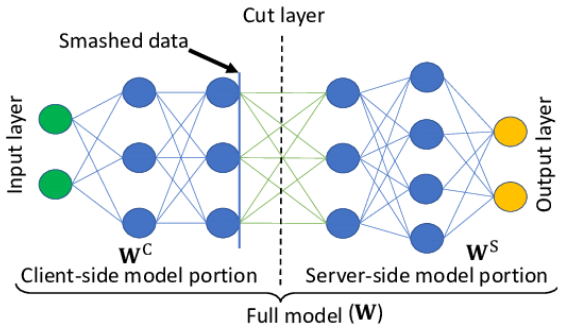


**Figure 2.** Split Learning Workflow

## 2.2 Split Learning (SL)

Split Learning (SL) offers a different approach by dividing the model into two segments: the initial layers are trained on client devices, while the remaining layers are processed on a central server. This approach allows resource-limited devices to participate in model training by offloading a portion of the computation to the server.

## 2.3 Limitations of FL and SL

While FL and SL provide privacy-preserving training mechanisms, they have notable trade-offs. FL ensures faster training through parallelism but demands higher computing power on client devices. Conversely, SL reduces computational requirements on clients but introduces delays due to its sequential nature.

### 2.4 SplitFed Learning (SFL)

To overcome the drawbacks of both FL and SL, researchers introduced SplitFed Learning (SFL), which combines the strengths of both approaches. SFL retains the parallelism of FL and the model privacy of SL, providing an efficient and secure distributed learning solution.

**Key Features of SFL**:

- **Hybrid Approach:** SFL divides the model like SL but allows clients to train in parallel like FL.
- **Federated Model Aggregation:** The client-side updates are aggregated using FedAvg, reducing training time.
- **Improved Scalability:** SFL efficiently handles multiple clients without the sequential bottlenecks of SL.

**Advantages of SFL**:

- Faster training compared to SL due to parallel processing.
- Reduced computational burden on client devices.
- Strong privacy protection as clients do not share raw data.

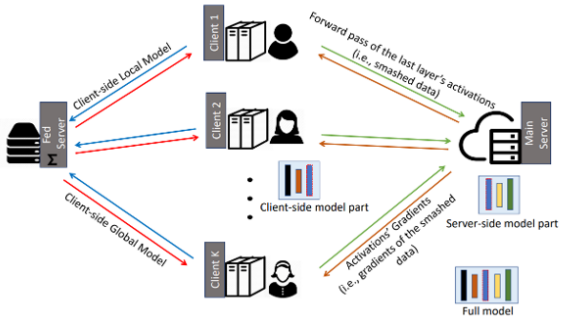


**Figure 3.** SplitFed Learning (SFL) Architecture

### 2.5 Conclusion

Machine learning models trained in a centralized manner pose serious privacy and efficiency concerns. FL and SL address these concerns but introduce their own challenges, making it difficult to achieve an optimal balance between privacy, efficiency, and scalability. SplitFed Learning (SFL) offers a promising alternative by leveraging the strengths of both FL and SL, ensuring privacy while enabling efficient training. The next sections will explore the methodology and experimental validation of SFL in greater detail.

## 3 Methodology

To evaluate the effectiveness of **SplitFed Learning (SFL)**, we designed an experimental setup that integrates Federated Learning (FL) and Split Learning (SL). This section details the architecture, system design, implementation, and evaluation criteria used in our study.

### 3.1 System Architecture

SFL follows a hybrid approach where the machine learning model is split between clients and the central server. Clients train the first few layers locally, and the remaining layers are trained on the server. The system consists of the following key components:

- **Clients:** Devices responsible for training the initial layers of the model.
- **Fed Server:** Aggregates updates from multiple clients to form a global model. N. B. For simpli
- **Main Server:** Processes the remaining layers of the model using data received from clients.

### 3.2 Frameworks

The original work from the base paper was not publicly available, requiring us to implement the entire architecture from scratch for our experiments. Initially, we attempted to use the **Flower Framework** for federated learning due to its flexibility in distributed training. However, we found that customizing the Flower framework for our specific experiments was challenging. As a result, we decided to build our own architecture from the ground up.

At first, we experimented with using **SSH-based communication** for secure client-server interactions. However, this approach had limitations in terms of scalability and latency. To overcome these issues, we transitioned to a more traditional **vanilla server-socket communication method**, which provided better control over message passing and synchronization between clients and the server.

**3.2.1 Client-Side Operations.** Each client follows these steps during training:

1. Receives the initial model from the Fed Server.
2. Processes the **first few layers** of the neural network.
3. Sends the **smashed data** (intermediate activations) to the main server.
4. Receives backpropagated gradients from the server.
5. Updates the local model and periodically synchronizes with the Fed Server.

**3.2.2 Server-Side Operations.** The main server and Fed Server coordinate the aggregation of client updates:

1. The main server receives smashed data from clients and processes the final layers.
2. Backpropagation is performed on the server-side layers.
3. Gradients are sent back to clients for local updates.
4. The Fed Server applies **Federated Averaging (FedAvg)** to aggregate client model updates.

### 3.3 Training Strategy

The training process follows an **iterative approach** with multiple communication rounds between clients and the server. The workflow is as follows:

1. Clients train their portion of the model using local data.
2. Smashed data is sent to the server, where final layers are trained.
3. Gradients are returned to clients for local weight updates.
4. The Fed Server aggregates client updates to refine the global model.
5. The updated model is distributed back to clients for the next iteration.

This iterative process continues until the model converges.

## 4 Algorithm and Implementation Details

This section describes the core algorithm behind **SplitFed Learning (SFL)** and provides details of its implementation. The algorithm integrates Federated Learning (FL) and Split Learning (SL) to balance privacy, computational efficiency, and model performance.

### 4.1 SplitFed Learning Algorithm

The training process in SFL consists of multiple communication rounds between clients and the central server. The key steps are outlined in Algorithm 4.

### 4.2 Implementation Details

To implement the SFL architecture, we designed a client-server communication model using socket programming. The implementation consists of the following components:

- **Client-Side Training:** Each client trains the first few layers of the neural network and sends intermediate activations (smashed data) to the server.
- **Server-Side Processing:** The server completes forward and backward propagation for the final layers.
- **Model Aggregation:** The Fed Server averages client updates using Federated Averaging (FedAvg).
- **Communication Protocol:** Implemented using a vanilla server-socket method for efficient client-server message passing.

```python
class ClientModel(nn.Module):
    def __init__(self):
        super(ClientModel, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(784, 128)
        self.activation1 = nn.ReLU()

    def forward(self, x):...
```

**Figure 5.** Client Model

```python
class ServerModel(nn.Module):
    def __init__(self):
        super(ServerModel, self).__init__()
        self.fc1 = nn.Linear(128, 64)
        self.activation1 = nn.ReLU()
        self.fc2 = nn.Linear(64, 10)
        self.activation2 = nn.Softmax(dim=1)

    def forward(self, x):...
```

**Figure 6.** Server Model

### 4.3 Client-Side Implementation

Each client follows these steps:

- Loads the dataset and initializes the model.
- Performs forward propagation on the client-side layers.
- Sends intermediate activations (smashed data) to the server.
- Receives backpropagated gradients from the server.
- Updates the local model and synchronizes with the Fed Server.

```python
def client_train(comm_with_server, comm_with_fed_server = None):
    train_loader, validation_loader = get_train_and_validation_loaders(CLIENT_NO, balanced=True)
    client_model.train()
    for epoch in range(TOTAL_GLOBAL_EPOCH):
        for i, (images, labels) in enumerate(train_loader):
            images, labels = images.to(device), labels.to(device)
            client_optimizer.zero_grad()
            split_layer_tensor = client_model(images)
            split_layer_tensor = split_layer_tensor.detach().requires_grad_(True)

            grads = comm_with_server(labels, split_layer_tensor, epoch, i)

            split_layer_tensor.backward(grads)
            client_optimizer.step()

        print(f"Client Epoch {epoch + 1} completed")
        if comm_with_fed_server:
            comm_with_fed_server(client_model, epoch, validation_loader)

    if comm_with_fed_server:
        comm_with_fed_server(client_model, -1)
    print("Client training done")
```

**Figure 7.** Client Training

### 4.4 Server-Side Processing

The server-side processing in SplitFed Learning (SFL) involves handling intermediate activations (smashed data) from clients, performing forward and backward propagation on the server-side model, computing gradients, and sending

**Algorithm 1:** Splitfed Learning (SFL)

**Notations:** (1) $S_t$ is a set of $K$ clients at $t$ time instance, (2) $\mathbf{A}_{k,t}$ is the smashed data of client $k$ at $t$, (3) $\mathbf{Y}_k$ and $\hat{\mathbf{Y}}_k$ are the true and predicted labels, respectively, of the client $k$, (4) $\nabla \ell_k$ is the gradient of the loss for the client $k$, (5) $n$ and $n_k$ are the total sample size and sample size at a client $k$, respectively.

/* Runs on Main Server */
**EnsureMainServer executes:**
  **if** time instance $t$=0 **then**
    Initialize $\mathbf{W}_t^{\mathrm{S}}$ (global server-side model)
  **else**
    **for** each client $k \in S_t$ in parallel **do**
      **while** *local epoch* $e \neq E$ **do**
        $(\mathbf{A}_{k,t}, \mathbf{Y}_k) \leftarrow \text{ClientUpdate}(\mathbf{W}_{k,t}^{\mathrm{C}})$
        Forward propagation with $\mathbf{A}_{k,t}$ on $\mathbf{W}_t^{\mathrm{S}}$, compute $\hat{\mathbf{Y}}_k$
        Loss calculation with $\mathbf{Y}_k$ and $\hat{\mathbf{Y}}_k$
        Back-propagation calculate $\nabla \ell_k(\mathbf{W}_t^{\mathrm{S}}; \mathbf{A}_t^{\mathrm{S}})$
        Send $d\mathbf{A}_{k,t} := \nabla \ell_k(\mathbf{A}_t^{\mathrm{S}}; \mathbf{W}_t^{\mathrm{S}})$ (i.e., gradient of the $\mathbf{A}_{k,t}$) to client $k$ for ClientBackprop($d\mathbf{A}_{k,t}$)
      **end**
    **end**
    Server-side model update: $\mathbf{W}_{t+1}^{\mathrm{S}} \leftarrow \mathbf{W}_t^{\mathrm{S}} - \eta \frac{n_k}{n} \sum_{i=1}^{K} \nabla \ell_i(\mathbf{W}_t^{\mathrm{S}}; \mathbf{A}_t^{\mathrm{S}})$
  **end**

/* Runs on Fed Server */
**EnsureFedServer executes:**
  **if** $t$=0 **then**
    Initialize $\mathbf{W}_t^{\mathrm{C}}$ (global client-side model)
    Send $\mathbf{W}_t^{\mathrm{C}}$ to all $K$ clients for ClientUpdate($\mathbf{W}_{k,t}^{\mathrm{C}}$)
  **else**
    **for** each client $k \in S_t$ in parallel **do**
      $\mathbf{W}_{k,t}^{\mathrm{C}} \leftarrow \text{ClientBackprop}(d\mathbf{A}_{k,t})$
    **end**
    Client-side global model updates: $\mathbf{W}_{t+1}^{\mathrm{C}} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} \mathbf{W}_{k,t}^{\mathrm{C}}$
    Send $\mathbf{W}_{t+1}^{\mathrm{C}}$ to all $K$ clients for ClientUpdate($\mathbf{W}_{k,t}^{\mathrm{C}}$)
  **end**

**Figure 4.** Split Fed Algorithm

them back to the clients for local updates. The server-side operations are divided into two components:

- **Main Server:** Processes the smashed data from clients and performs backpropagation.
- **Fed Server:** Aggregates updated models from multiple clients using Federated Averaging (FedAvg).

**Main Server Processing.** The Main Server is responsible for training the server-side layers of the model. The process follows these steps:

1. Receives smashed data $A_{k,t}$ from clients.
2. Performs forward propagation on the server-side model $W_t^S$.
3. Computes the predicted labels $\hat{Y}_k$.
4. Calculates loss $L$ using the true labels $Y_k$.
5. Computes gradients $\nabla \ell_k(W_t^S; A_{k,t}^S)$ via backpropagation.
6. Sends gradient $dA_{k,t}$ back to clients for local updates.
7. Updates the global server model based on the aggregated gradients.

```
slt_path = slt_message.content
receive_file(client_socket, specific_client_file_directory_path, slt_path)

labels = torch.load(f"{specific_client_file_directory_path}/{labels_path}", weights_only=False)
split_layer_tensor = torch.load(f"{specific_client_file_directory_path}/{slt_path}", weights_only=False)

server_optimizer.zero_grad()
server_output = server_model(split_layer_tensor)
loss = loss_criteria(server_output, labels)

# print to log file
with open(f"shared_files/server/{server_log_file}", "a") as log_file:
    log_file.write(f"{clients_nos[client_name]},{labels_path},loss,{loss}\n")

loss.backward()
server_optimizer.step()
split_layer_tensor.retain_grad()
torch.save(split_layer_tensor.grad, f"{specific_client_file_directory_path}/grads_{labels_path}")

send_file(client_socket, specific_client_file_directory_path, f"grads_{labels_path}")
```

**Figure 8.** Server Training

### Model Aggregation: Fed Server Processing

The Fed Server plays a crucial role in maintaining the consistency of the global model across multiple clients. It performs two main tasks:

1. **Model Initialization and Distribution:** At the start of training ($t = 0$), the Fed Server initializes the global client-side model $W^C$ and distributes it to all clients for local training.
2. **Model Aggregation and Update:** After each training round, the Fed Server collects updated client models, applies Federated Averaging (FedAvg) to aggregate

them, and distributes the new global model back to the clients.

```python
def send_fed_avg_model_to_clients(client_socket, epoch, send = True):
    global clients, clients_lock, AGGREGATION_DONE
    if not AGGREGATION_DONE[epoch]:
        fed_avg_model = get_fed_avg_model(epoch)
        torch.save(fed_avg_model.state_dict(), f"shared_files/server/fed_avg_model_{epoch}.pt")
        AGGREGATION_DONE[epoch] = True
    if send:
        send_file(client_socket, "shared_files/server", f"fed_avg_model_{epoch}.pt", MessageType.SEND_AGGREGATED_MODEL)

def get_fed_avg_model(epoch):
    global device, CLIENT_WEIGHT, CLIENT_MODEL_PATHS, TOTAL_CLIENTS
    fed_avg_client_model = ClientModel().to(device)
    for i in range(TOTAL_CLIENTS):
        temp_client_model = ClientModel().to(device)
        temp_client_model.load_state_dict(torch.load(CLIENT_MODEL_PATHS[i][epoch]))
        for fed_avg_param, client_param in zip(fed_avg_client_model.parameters(), temp_client_model.parameters()):
            fed_avg_param.data += client_param.data * CLIENT_WEIGHT[i]
    return fed_avg_client_model
```

**Figure 9.** Federated Average

## Acknowledgments

To Robert, for the bagels and explaining CMYK and color spaces.

## References

## 5 Experiment Setup

This section describes the experimental setup used to evaluate the performance of **SplitFed Learning (SFL)**. We outline the dataset, model architecture, number of clients, system configurations, evaluation metrics, and training procedures.

### 5.1 Dataset: Extended MNIST (EMNIST)

We used the **Extended MNIST (EMNIST)** dataset, which is an extension of the MNIST dataset containing handwritten digits and letters.

- **Total Samples:** 814,255 handwritten characters.
- **Classes:** 62 classes (digits 0-9, uppercase and lowercase letters A-Z).
- **Train-Test Split:** 88% training, 12% testing.
- **Preprocessing:** Images were resized to $28 \times 28$ pixels, normalized, and converted to grayscale.

### 5.2 Number of Clients

In our experiments, we simulated a **distributed learning environment** using **3 clients** running in parallel.

- Each client was responsible for training on a unique subset of the dataset.
- Clients trained their **client-side model layers** and communicated with the server.
- Clients followed an iterative learning process where local updates were aggregated at the **Fed Server**.

### 5.3 Model Architecture

We implemented a **LeNet-5 inspired model**, which was split between the clients and the server.

- **Client-Side Layers:** First **two convolutional layers** process input locally.
- **Server-Side Layers:** Remaining layers, including **fully connected layers**, process smashed data.

**Model Summary:**

**Table 2.** LeNet-5 Model Architecture Split

| Layer Type | Client-Side | Server-Side |
|---|---|---|
| Convolutional Layers (2) | Yes | No |
| Batch Normalization | Yes | No |
| Max-Pooling Layers | Yes | No |
| Fully Connected Layers | No | Yes |
| Softmax Activation | No | Yes |

### 5.4 System Configuration

The experiments were conducted using **Google Cloud for client devices** and **a local high-performance PC as the Main Server**.

**Client Machines (Google Cloud VMs):**

- **Number of Clients:** 3 clients, each running on a separate VM.
- **Instance Type:** Google Cloud **e2-standard-4** (4 vCPUs, 16GB RAM).
- **Operating System:** Ubuntu 20.04 LTS.
- **Frameworks:** PyTorch, NumPy.

**Main Server (Local PC):**

- **Processor:** Intel Core i7 (12th Gen) with 8 cores.
- **GPU:** NVIDIA **RTX 3050 (4GB VRAM)**.
- **RAM:** 16GB DDR4.
- **Storage:** 1TB SSD.
- **OS:** Windows 11

### 5.5 Evaluation Metrics

To evaluate the effectiveness of SFL, we used the following metrics:

1. **Validation Accuracy**: Measures model performance on test data.
2. **Training Loss**: Tracks reduction in error over training rounds.
3. **Overall Accuracy**: Measures final aggregated model's accuracy.

### 5.6 Training Procedure

The training process consists of multiple communication rounds between clients and the server.

1. Clients receive an initial model from the Fed Server.
2. Each client trains its **client-side model** on its local dataset.
3. Clients send **smashed data** to the **Main Server** (local PC).
4. The **Main Server** processes smashed data and performs backpropagation.
5. Gradients are sent back to clients for local updates.

6. Clients update local models and send updates to the **Fed Server**.
7. The **Fed Server** aggregates updates using **Federated Averaging (FedAvg)**.

## 6 Evaluation

This section presents the evaluation of **SplitFed Learning (SFL)** and its comparison with **Split Learning (SL)** in both balanced and imbalanced dataset scenarios. The evaluation is based on two key performance metrics: **validation accuracy** and **training loss**.

### 6.1 Validation Accuracy

The validation accuracy was measured across different dataset distributions and weight configurations to evaluate model performance.

- **Balanced Dataset:** SFL achieves consistently higher accuracy than SL, demonstrating its ability to maintain better model convergence.
- **Imbalanced Dataset:** SFL retains a competitive accuracy level, showing its robustness against data imbalance.
- **Performance Across Clients:** Clients in SFL experience more stable and higher validation accuracy than SL.

These results indicate that **SFL consistently outperforms SL**, particularly in balanced data settings, while still maintaining competitive accuracy in imbalanced data scenarios.

### 6.2 Training Loss

The training loss was analyzed to evaluate model convergence stability and optimization efficiency.

- **SL:** Displays more fluctuations, indicating less stable convergence across clients.
- **SFL:** Shows a steady decline in loss, ensuring better optimization and training efficiency.
- **Balanced vs Imbalanced Dataset:** While both settings converge, the balanced dataset results in smoother loss reduction.

The findings show that **SFL provides a more stable training process with lower loss fluctuations compared to SL**, which is especially useful in real-world federated learning applications.

### 6.3 Overall Performance

To summarize, **SFL consistently outperforms SL** across different dataset settings, providing higher accuracy, lower loss, and better model stability.

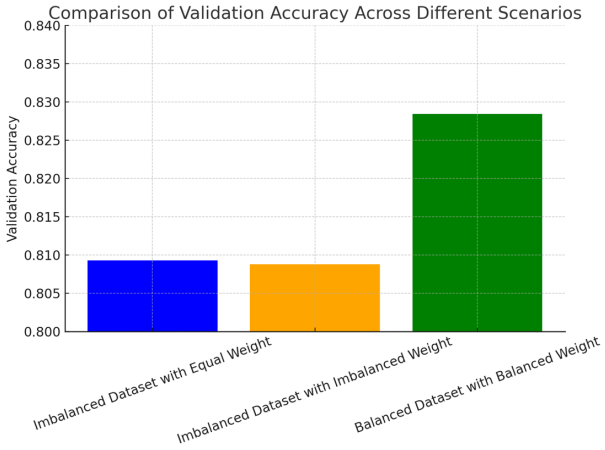


**Figure 14.** Final Comparison of Validation Accuracy Across Different Scenarios

### 6.4 Key Takeaways

- **SFL consistently achieves higher validation accuracy** across multiple scenarios.
- **SFL reduces training loss fluctuations**, leading to more stable model convergence.
- **SFL performs best in balanced datasets**, making it ideal for real-world applications.
- **SFL adapts well to imbalanced datasets**, outperforming SL in all cases.

### 6.5 Conclusion

The evaluation results confirm that SplitFed Learning (SFL) is a more effective alternative to traditional Split Learning (SL), offering higher accuracy, faster convergence, and better model stability. These advantages make SFL a **promising solution for distributed machine learning applications**.

## 7 Future Works

Future improvements and research directions for **SplitFed Learning (SFL)** include but are not limited to:

- Scaling with more clients and sophisticated models.
- Adaptive weight assignment based on dataset characteristics instead of just dataset size.
- Handling straggler clients to avoid laggy participants in the training process.
- Enhancing security and privacy with techniques such as homomorphic encryption and differential privacy.
- Optimizing communication overhead through model compression and gradient sparsification.
- Adapting to heterogeneous devices with varying computational capabilities.
- Automated hyperparameter tuning for optimal training efficiency.

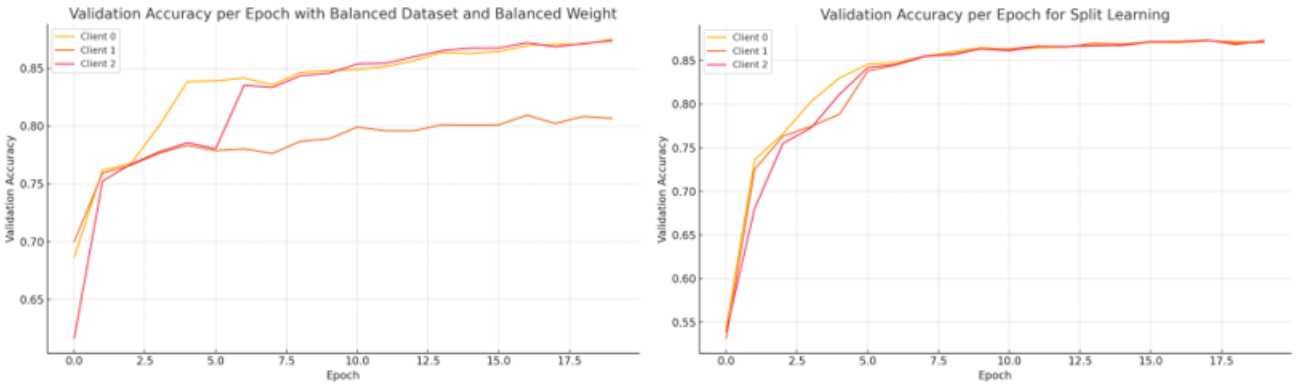


**Figure 10.** Comparison of Validation Accuracy for SFL (Left) vs SL (Right) on **Balanced Dataset**

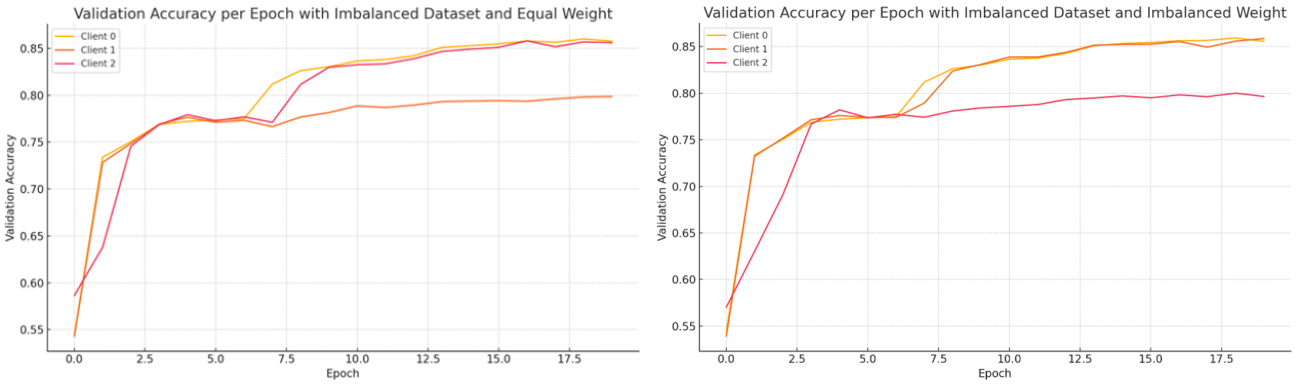


**Figure 11.** Comparison of Validation Accuracy for SFL (Left) vs SL (Right) on **Imbalanced Dataset**

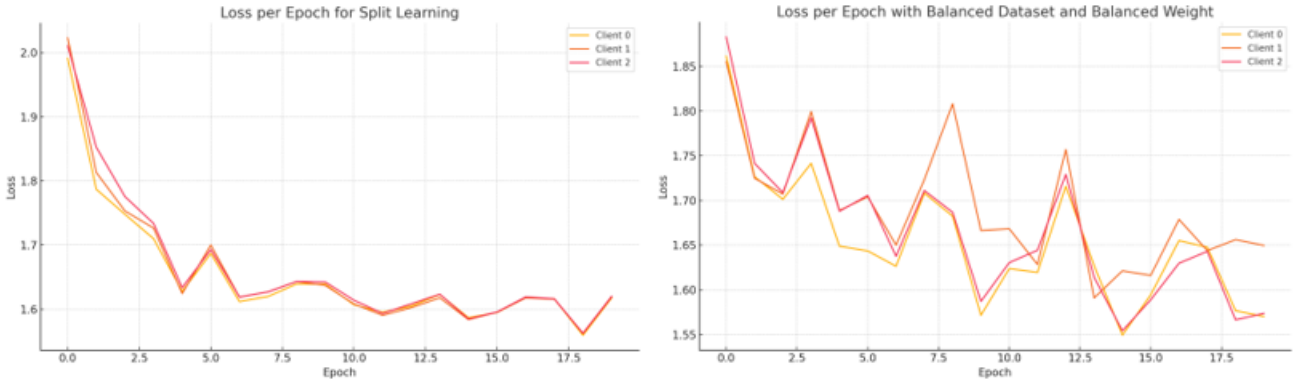


**Figure 12.** Comparison of Training Loss for SFL (Left) vs SL (Right) on **Balanced Dataset**

## 8 Conclusion

This experiment validates **SplitFed Learning (SFL)** as an improved hybrid approach that integrates the strengths of **Federated Learning (FL)** and **Split Learning (SL)** to enhance privacy, computational efficiency, and scalability in distributed machine learning. The experimental results demonstrated that SFL consistently achieves higher validation accuracy and lower training loss compared to SL, particularly in both balanced and imbalanced dataset scenarios. The improved convergence stability of SFL reduces training fluctuations, making it a more reliable approach for decentralized learning environments.

Furthermore, SFL effectively manages data imbalance across clients, ensuring that model performance remains robust

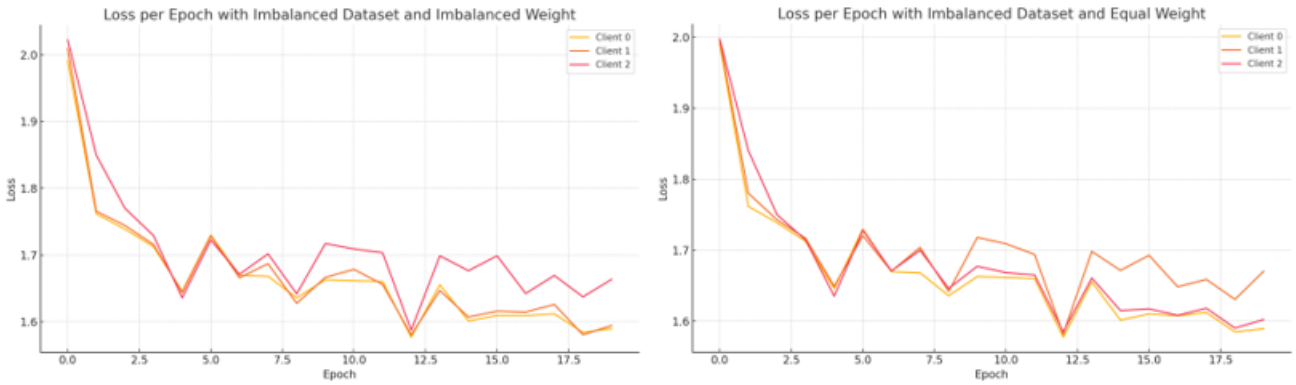


**Figure 13.** Comparison of Training Loss for SFL (Left) vs SL (Right) on **Imbalanced Dataset**

even when dataset distributions vary. By reducing the computational burden on client devices while maintaining high accuracy, SFL proves to be well-suited for resource-constrained environments where privacy and efficiency are critical considerations.

Despite these advantages, there remain several areas for future exploration. Expanding SFL to support a larger number of clients and more complex deep learning architectures can provide further insights into its scalability. Additionally, optimizing weight assignment strategies based on dataset characteristics rather than solely relying on dataset size can further improve model performance. Addressing the issue of straggler clients by incorporating dynamic scheduling and reducing communication overhead will enhance overall training efficiency. Strengthening privacy-preserving mechanisms through advanced encryption and secure aggregation techniques can provide better protection for sensitive client data.

In conclusion, **SFL presents a scalable, privacy-aware, and computationally efficient alternative to traditional Split Learning, making it a promising framework for decentralized artificial intelligence systems.** By addressing the remaining challenges, SFL has the potential to further advance the field of distributed deep learning and enable more robust, secure, and efficient collaborative learning paradigms.