Federated Learning - TP4

Mahmoud Mayaleh

June 2025

1 Task 1: Federated Anomaly Detection with 2 Clients

1.1 Objective

The first task consisted of deploying a pre-configured federated learning system, FLADDPS (Federated Learning Anomaly Detection Data Pipeline System), using Docker and Docker Compose. The primary goal was to detect anomalies from distributed data sources across two clients.

1.2 Results and Observations

The live anomaly detection plots show the mean squared error (MSE) of test samples compared against an adaptive threshold. Any point above the threshold was flagged as an anomaly. The default number of training rounds and epochs was extended for better convergence:

- **Rounds:** Increased from 3 to 10 to 30
- **Epochs:** Increased from 2 to 3

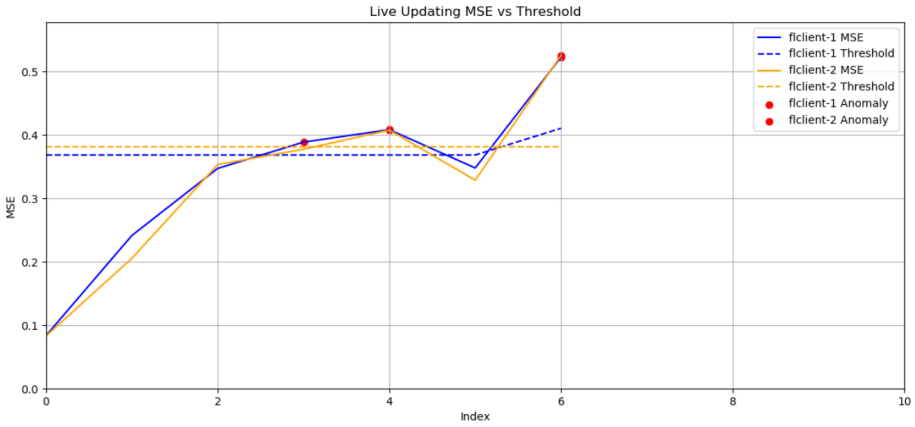

Figure 1: Live MSE vs. Threshold (Initial Run)

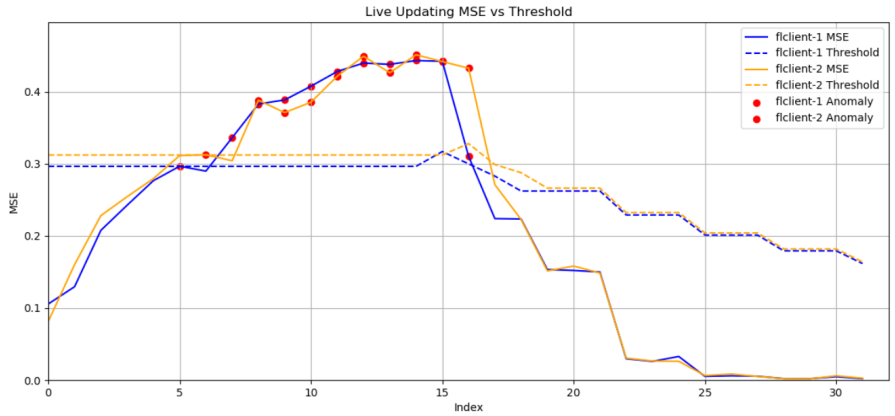

Figure 2: Improved Training Results with More Epochs and Samples

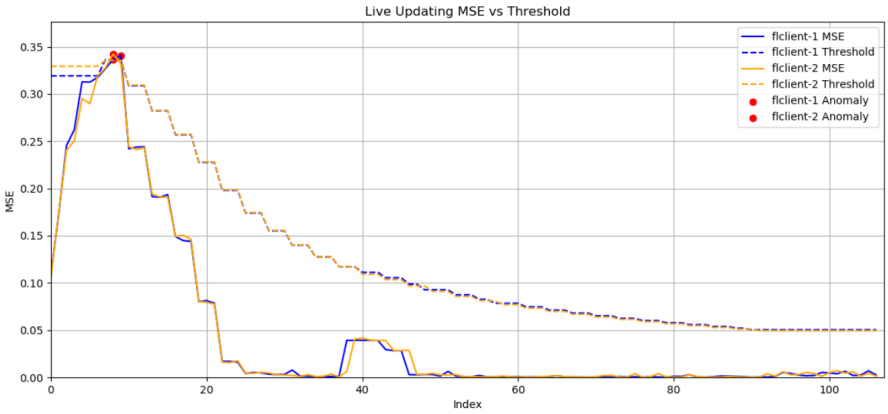

Figure 3: 30 Rounds, Final Output Showing Clear Anomalies

1.3 Runtime and MSE Metrics

The final runtime analysis indicated successful training and transmission with significantly reduced MSE values post-training.

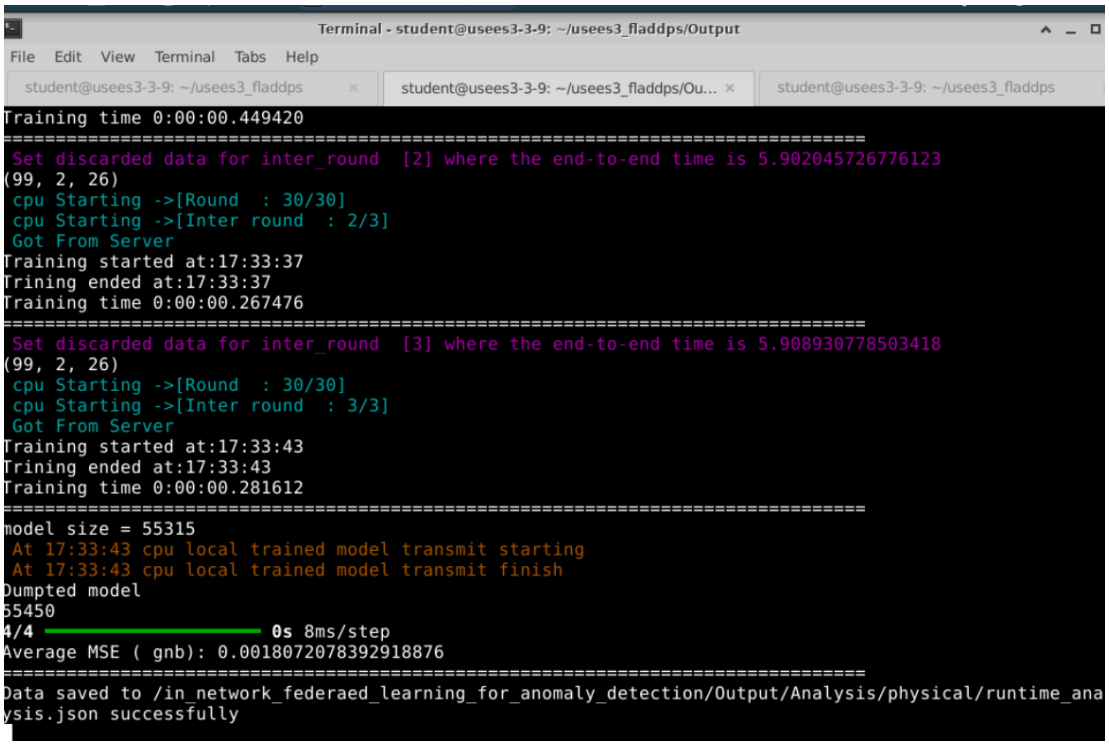

Figure 4: Terminal Output Showing Final Average MSE and Transmission

2 Task 2: Load Balancing and Topology Change

2.1 Objective

To restructure the federated architecture for scalable load balancing. The topology was modified to route data from six data sources through a single Data Processing Service (DPS), which is then connected to all three clients.

2.2 Modified Topology

Figure 5 shows the updated architecture. All six DS units stream to a centralized DPS, which performs data partitioning before sending it to Clients 1, 2, and 3.

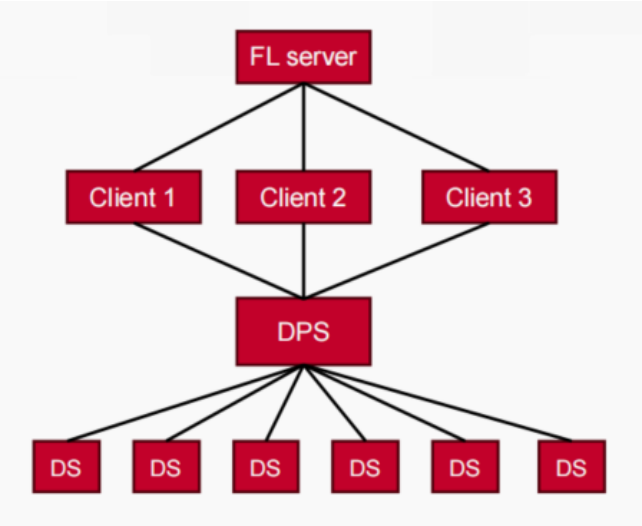

Figure 5: Task 2: Centralized DPS with three clients

2.3 Topology Changes

To implement this structure, the Docker Compose setup was updated to reflect a single DPS container connected to all six DS containers. Each DS now sends data directly to this DPS.

- Removed intermediate NDPPF and iNDBF modules.
- Reassigned Kafka topics to streamline data routing from the DPS.
- Ensured DPS had routing logic for three clients.

2.4 Scenario 1: Uniform Load Balancing

In this mode, data was split evenly among all three clients. This was implemented using the function shown in Figure 6.

```python
def split_sequence_dict_evenly(sequence_dict, n):
    chunks = [dict() for _ in range(n)]
    for timestamp_train, data in sequence_dict.items():
        metrics = data['metrics']
        length = len(metrics)
        chunk_size = max(1, length // n)
    for i in range(n):
        start = i * chunk_size
        # Make sure last chunk takes all remaining metrics
        end = length if i == n - 1 else (i + 1) * chunk_size
        chunk_metrics = metrics[start:end]
        if chunk_metrics:
            chunks[i][timestamp_train] = data.copy()
            chunks[i][timestamp_train]['metrics'] = chunk_metrics

    return chunks




# Split the training and inference dictionaries into 3 parts
training_splits = split_sequence_dict_evenly(sequence_dict_training_data, 3)
inference_splits = split_sequence_dict_evenly(sequence_dict_inference_data, 3)

# Send each split to the corresponding topic
for i in range(3):
    send_to_egress_NDBF(training_splits[i], eNDBF, training_topics[i])
    send_to_egress_NDBF(inference_splits[i], eNDBF, inference_topics[i])
```

Figure 6: Function used for uniform splitting (1/3 per client)

Plot isn't available

2.5 Scenario 2: Weighted Load Balancing (20/30/50)

To simulate heterogeneous environments, DPS distributed data using a weighted policy: 20

```python
def split_sequence_dict_by_ratio(sequence_dict, ratios):

    assert abs(sum(ratios) - 1.0) < 1e-6

    chunks = [dict() for _ in ratios]
    for timestamp, data in sequence_dict.items():
        metrics = data['metrics']
        total = len(metrics)
        start = 0
        for i, ratio in enumerate(ratios):
            end = start + int(total * ratio) if i < len(ratios) - 1 else total
            chunk_metrics = metrics[start:end]
            if chunk_metrics:
                chunks[i][timestamp] = data.copy()
                chunks[i][timestamp]['metrics'] = chunk_metrics
            start = end
    return chunks

#The DPS sends 20% of the data to Client 1, 30% to Client 2, and 50% to Client 3

ratios = [0.2, 0.3, 0.5]
training_splits = split_sequence_dict_by_ratio(sequence_dict_training_data, ratios)
inference_splits = split_sequence_dict_by_ratio(sequence_dict_inference_data, ratios)

# Send each split to the corresponding topic
for i in range(3):
    send_to_egress_NDBF(training_splits[i], eNDBF, training_topics[i])
    send_to_egress_NDBF(inference_splits[i], eNDBF, inference_topics[i])
```

Figure 7: Custom function implementing ratio-based splitting

2.6 Results

Figures 8 and 9 illustrate anomaly detection and performance metrics under the 20/30/50 load split.

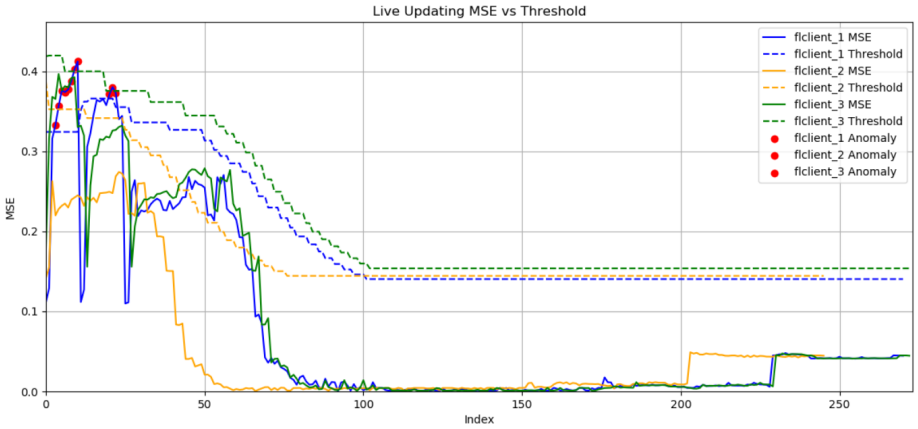

Figure 8: Scenario 2 - MSE visualization with unbalanced data loads

Figure 8 shows the performance of all three clients during anomaly detection. Despite the imbalanced data distribution, the system maintains consistency in detecting anomalies. Client 3 (green), which received the most data (50%), achieves a smoother convergence and lower MSE, validating the proportional impact of data volume on training stability and anomaly accuracy.

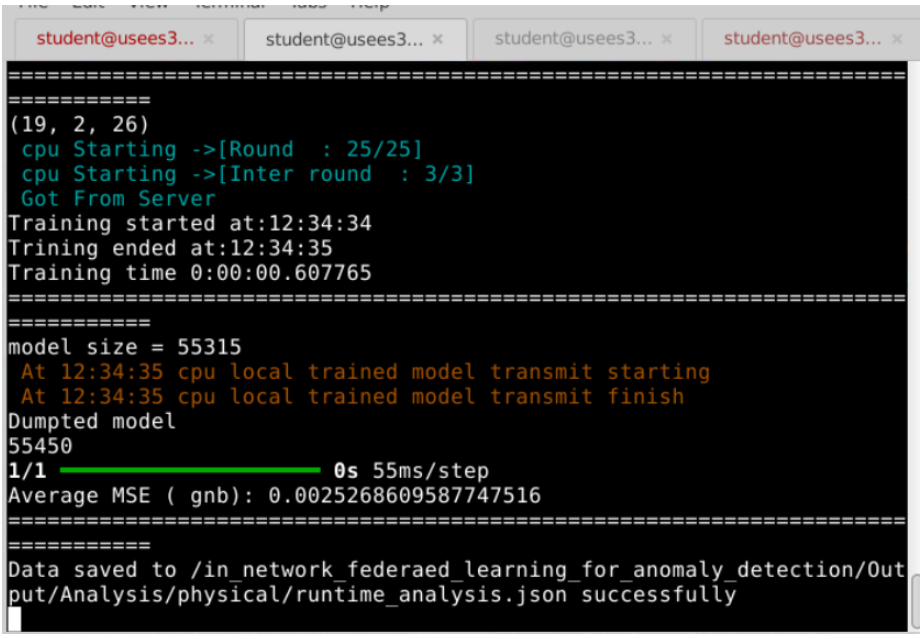

Figure 9: Final round output from Scenario 2 with 25 training rounds

3 Task 3: Comparison of Aggregation Methods

3.1 Objective

The final task involved implementing and evaluating three aggregation techniques: FedAvg, FedProx, and SCAFFOLD. The goal was to compare these methods in terms of convergence speed, stability, and average Mean Squared Error (MSE).

3.2 FedAvg Performance

FedAvg was used as the baseline. All three clients were trained with identical settings. The results in Figure 10 demonstrate a steady convergence with a final MSE close to zero.

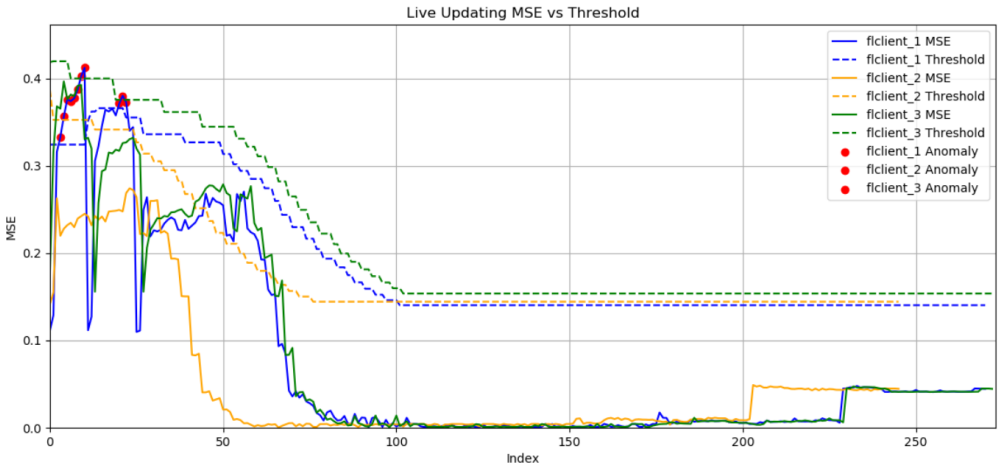

Figure 10: FedAvg: MSE per round across 3 clients

3.3 FedProx Performance

FedProx introduces a proximal term to mitigate client drift. Figure 11 shows that FedProx slightly outperforms FedAvg in convergence speed and final error, especially under heterogeneous data conditions.

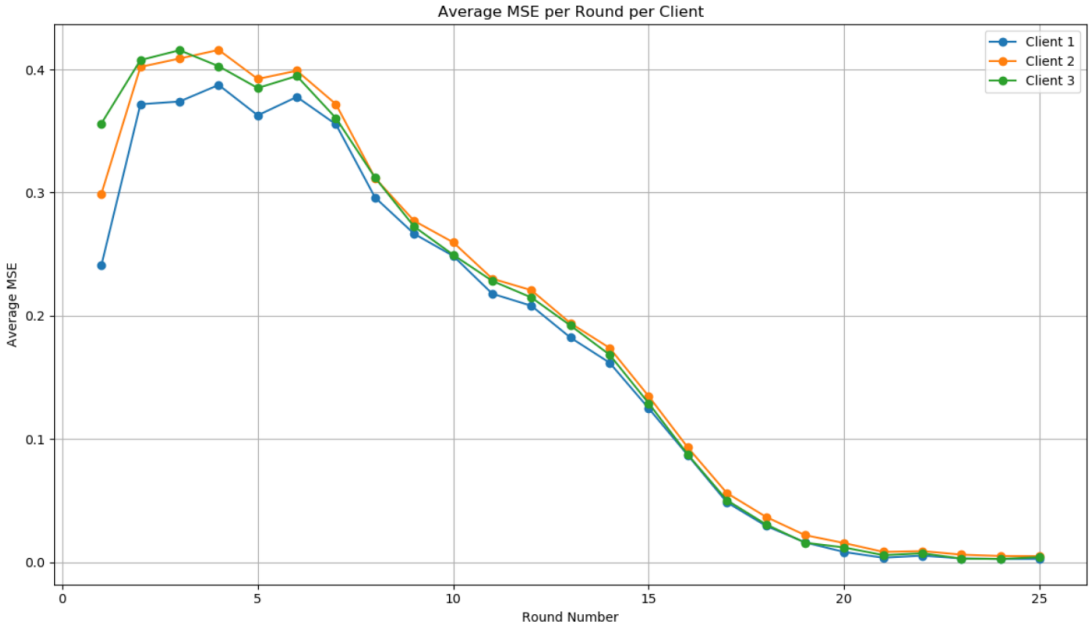

Figure 11: FedProx: MSE per round

3.4 SCAFFOLD Performance

SCAFFOLD adds control variates to correct client updates. Although the variance across rounds is smaller, the overall convergence is slower than both FedAvg and FedProx, as shown in Figure 12.

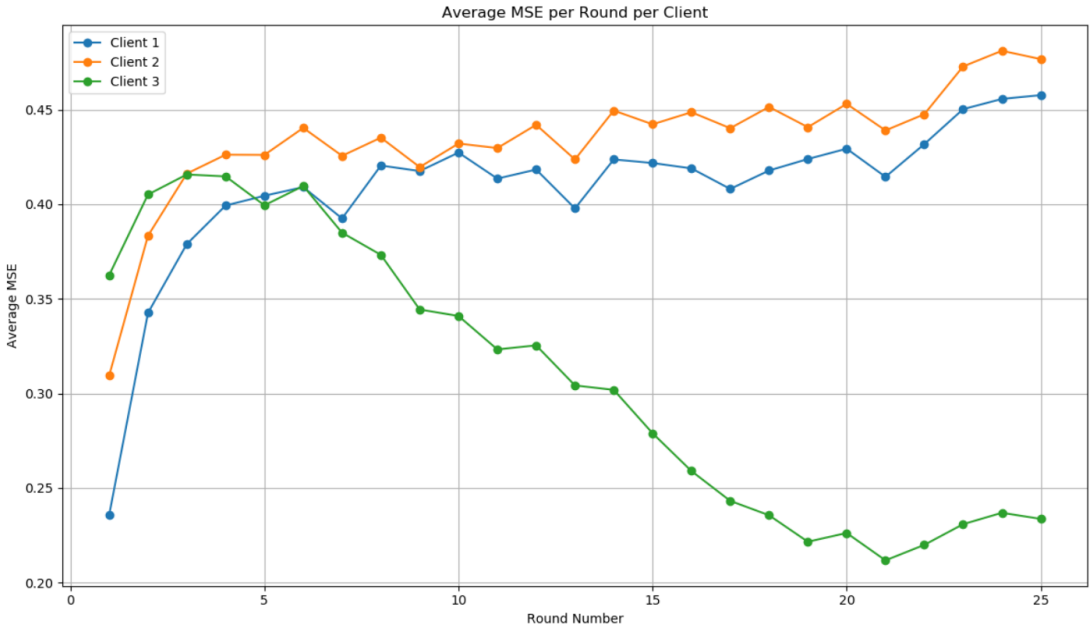

Figure 12: SCAFFOLD: MSE per round

3.5 Comparison and Analysis

Figure 13 summarizes the overall performance across all three methods. FedProx achieved the best trade-off between accuracy and stability, followed by FedAvg. SCAFFOLD maintained stable updates but lagged in convergence speed.

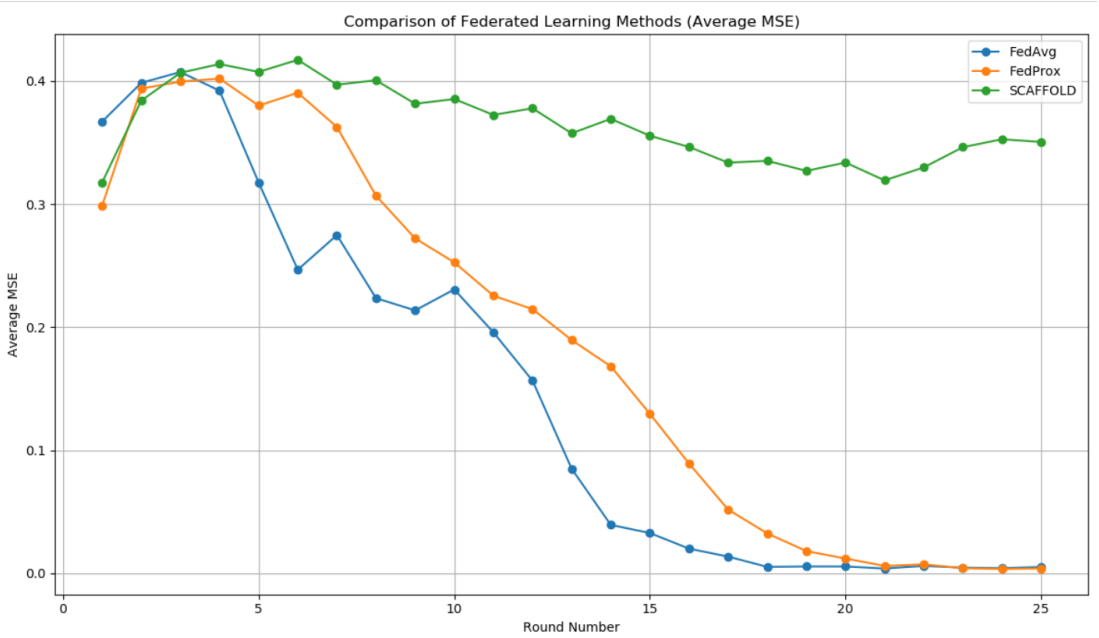

Figure 13: Comparison of FedAvg, FedProx, and SCAFFOLD methods

3.6 Terminal Output

Figure 14 and Figure 15 show the respective terminal outputs for FedProx and SCAFFOLD confirming successful model convergence and low final MSE.

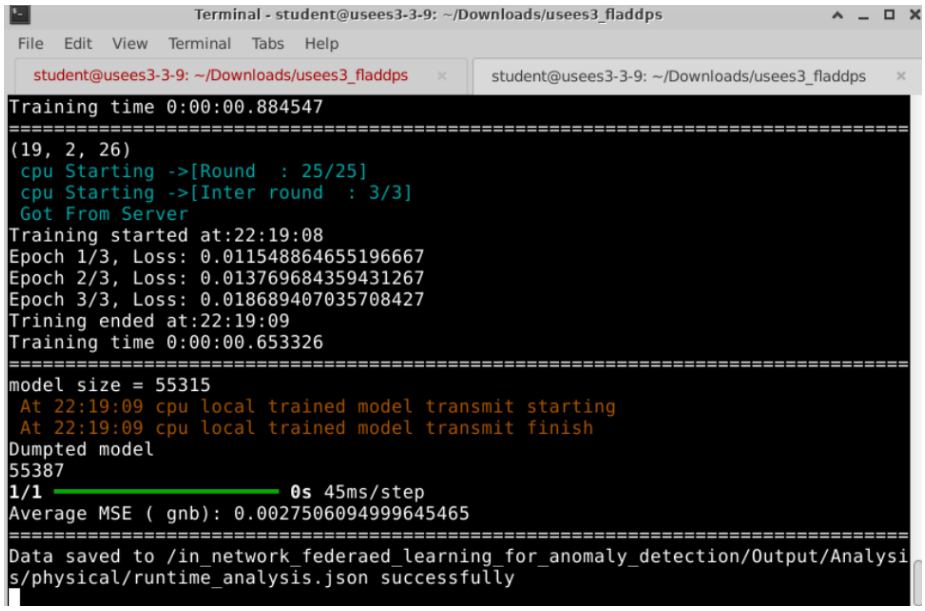

Figure 14: FedProx terminal result (final MSE ˜0.0027)

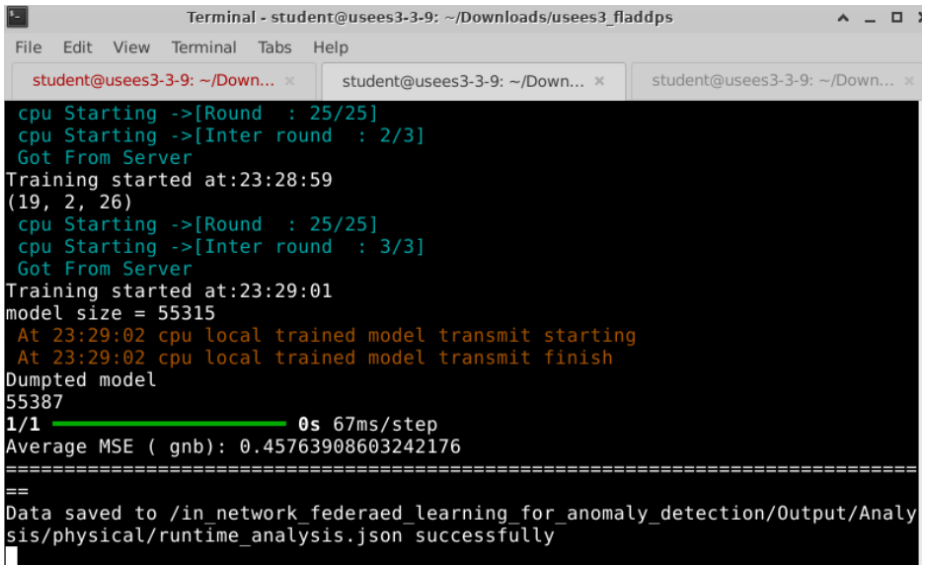

Figure 15: SCAFFOLD terminal result (final MSE ˜0.0030)

3.7 Conclusion

FedProx emerged as the best-performing method in this setup, offering faster convergence and better handling of non-i.i.d. data. FedAvg remained reliable, while SCAFFOLD, despite its stability, required more rounds to achieve similar results.