Federated Learning Model

Using

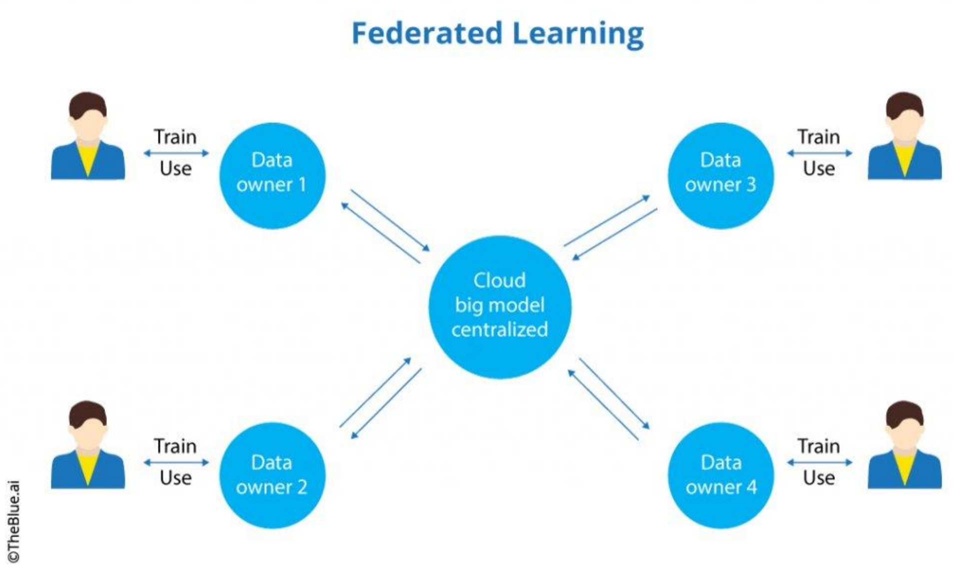
Adaptive Clustering

Problem Statement

To construct a Federated Learning Model using an Adaptive Clustering scheme for client selections.

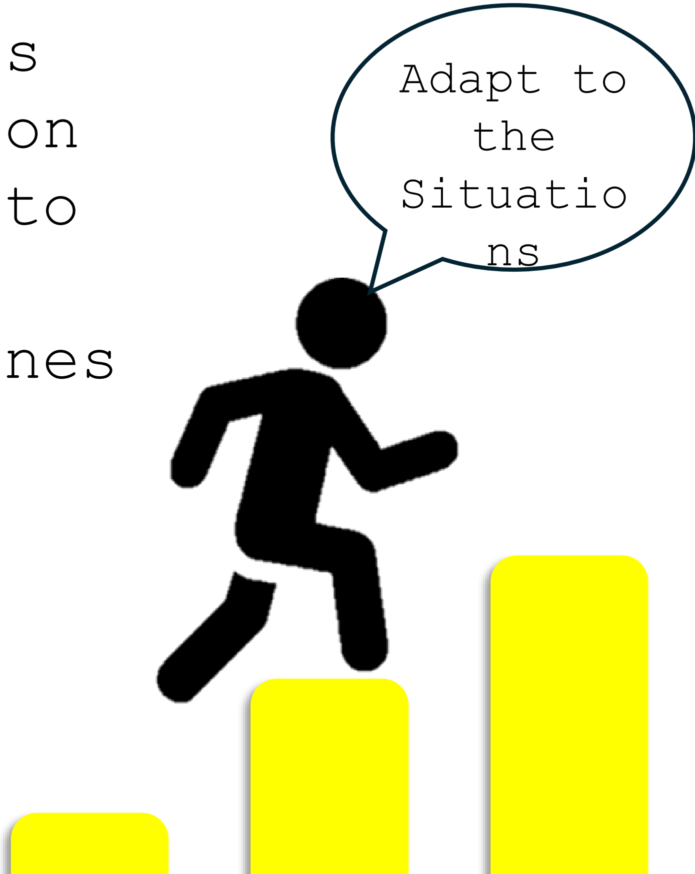
Federated Learning

Federated learning works by training a central model across decentralized devices or servers. Instead of moving all data to a central location, the model is trained locally on each device, and only the model updates are shared. This maintains privacy and allows collaborative learning without sharing raw data.



Adaptive Clustering

The Adaptive clustering approach dynamically clusters clients, starting with individual clusters and merging them based on training progress. Simulated Annealing-like randomness helps escape local minima, while stabilization rounds promote convergence. This adaptation to data distribution and local performance can improve overall model performance.It determines optimal cluster numbers, based on the Loss reduction ratio, and updates the number of clusters, enhancing accuracy and robustness.

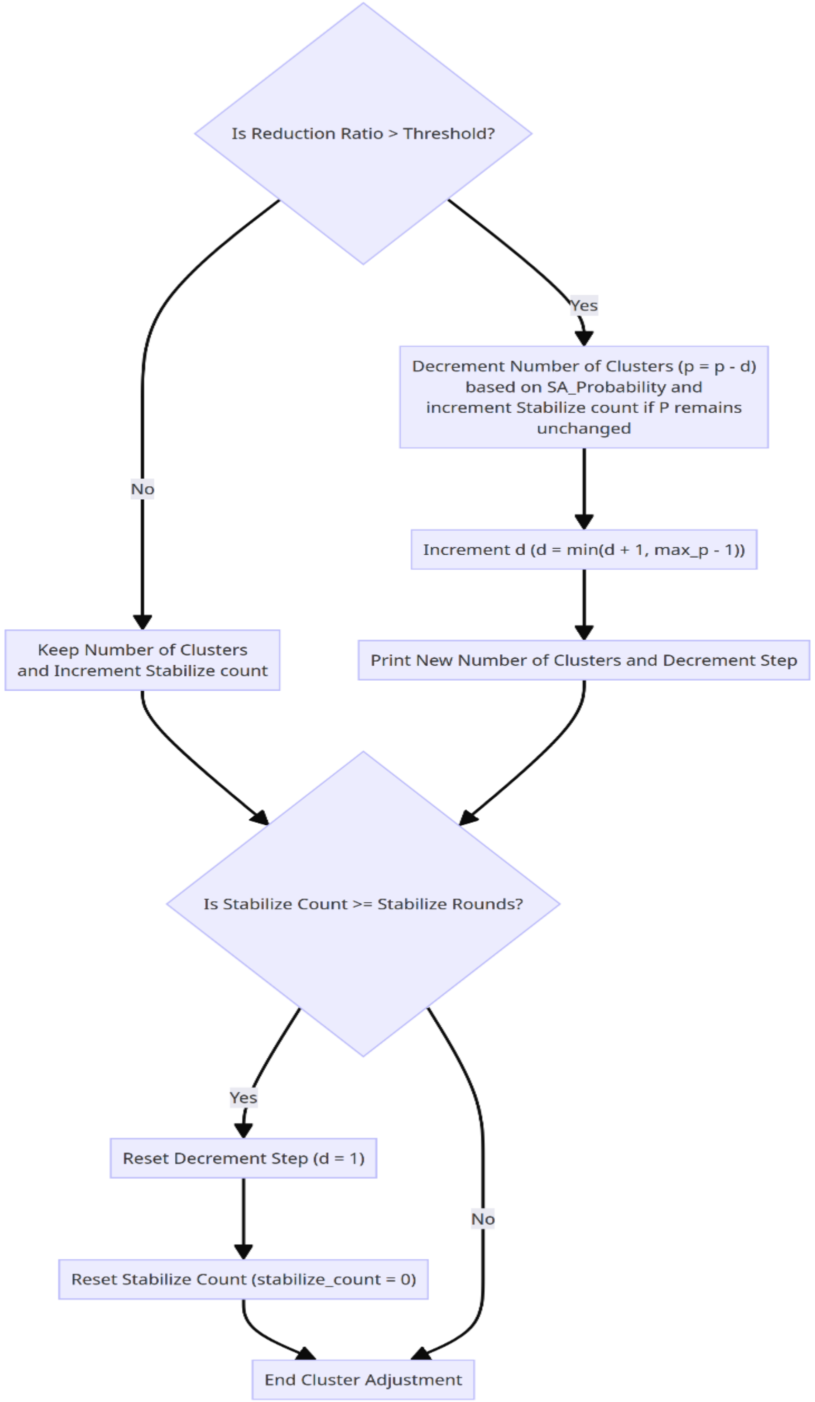


Stabilization Techniques:

SA - like:

-> The SA-like stabilization technique uses a Probability – based approach to decide whether to keep the current number of clusters or decrease it.

SA - Like Flow



Is Reduction Ratio > Threshold?
Yes
Decrement Number of Clusters (p = p - d) based on SA_Probability and increment Stabilize count if P remains unchanged
Increment d (d = min(d + 1, max_p - 1))
Print New Number of Clusters and Decrement Step
No
Keep Number of Clusters and Increment Stabilize count
Is Stabilize Count >= Stabilize Rounds?
Yes
Reset Decrement Step (d = 1)
Reset Stabilize Count (stabilize_count = 0)
No
End Cluster Adjustment

Implementation:

1. Simple MLP and Data split

```
class SimpleMLP(nn.Module):
    def __init__(self):
        super(SimpleMLP, self).__init__()
        self.fc1 = nn.Linear(73, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 14)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return torch.log_softmax(x, dim=1)
```

```
num_clients = 8
train_loaders = [DataLoader(TensorDataset(X_train_tensor, y_train_tensor), batch_size=64, shuffle=True) for _ in range(num_clients)]
test_loader = DataLoader(TensorDataset(X_test_tensor, y_test_tensor), batch_size=64, shuffle=False)
```

->The SimpleMLP class defines a neural network with three fully connected layers
->ReLU activation is applied after the first two layers (fc1 and fc2), introducing non-linearity to the model
->The final layer (fc3) produces logits which are normalized using log_softmax for multi-class classification tasks.
->The data split is performed using multiple data loaders (train_loaders), where each data loader represents a subset of the training data assigned to a specific client.

2. Global Model initialization and clients training:

```
global_model.train()
total_loss = 0
num_batches = 0
local_params_list = []

for train_loader in train_loaders:
    local_model = SimpleMLP()
    local_model.load_state_dict(global_model.state_dict())
    local_optimizer = optim.Adam(local_model.parameters(), lr=lr)

    for data, target in train_loader:
        local_optimizer.zero_grad()
        output = local_model(data)
        loss = nn.functional.nll_loss(output, target)
        loss.backward()
        local_optimizer.step()
        total_loss += loss.item()
        num_batches += 1

    local_params = torch.cat([param.data.view(-1) for param in local_model.parameters()])
    local_params_list.append(local_params.unsqueeze(0))
    stacked_params = torch.cat(local_params_list, dim=0)

avg_loss = total_loss / num_batches
reduction_ratio = prev_loss / avg_loss if prev_loss > 0 else 0
```

->Set the global model in training mode (global_model.train()).
->Initialize `total_loss` and `num_batches` for calculating the average loss.
->Train each client after initializing it with the global model on its respective `train_loader` in `train_loaders`.
->Store the client model weights in `stacked_params`.
->For each epoch i (i = 1..m), compute the average losses Li from all clients. Then calculate the reduction ratio between Li-1 and Li

3. Cluster Adjustment and Client Selection:

```
if reduction_ratio > threshold_w:
    # SA-like algorithm: Decide whether to keep p unchanged based on sa_prob
    if random.random() < sa_prob:
        # Keep the number of clusters unchanged
        stabilize_count += 1
    else:
        # Decrease the number of clusters
        stabilize_count = 0
        p = max(p - d, 1)
        d = min(d + 1, max_p - 1)  # Increment d
        print(f"Adjusting clusters DOWN. New number of clusters: {p}, d: {d}")
else:
    stabilize_count += 1

if stabilize_count >= stabilize_rounds:
    # Reset d and stabilize count if we've stabilized for enough rounds
    d = 1
    stabilize_count = 0

prev_loss = avg_loss

cluster_labels = perform_clustering(stacked_params, n_clusters=p)
print(f"Cluster Labels: {cluster_labels}")

selected_clients = select_clients(train_loaders, cluster_labels)
print(f"Selected Clients: {[train_loaders.index(client) for client in selected_clients]}")
```

->Check if the reduction ratio meets the threshold (threshold_w) for potential cluster adjustment.
->If yes, decide whether to keep the number of clusters unchanged based on the simulated annealing probability (sa_prob).
->Adjust the number of clusters (p) and the increment factor (d) accordingly.
->Perform clustering (perform_clustering) using the global model and stacked parameters (stacked_params) with the updated number of clusters (p).
->Select clients (selected_clients) based on the cluster labels using select_clients.

4. Cluster Adjustment and Client Selection:

```
def perform_clustering(stacked_params, n_clusters):
    clustering = AgglomerativeClustering(n_clusters=n_clusters)
    cluster_labels = clustering.fit_predict(stacked_params.detach().numpy())
    return cluster_labels

def select_clients(train_loaders, cluster_labels):
    selected_clusters = set()
    selected_clients = []
    for loader, label in zip(train_loaders, cluster_labels):
        if label not in selected_clusters:
            selected_clusters.add(label)
            selected_clients.append(loader)
    return selected_clients
```

->Utilizes Agglomerative Clustering to cluster clients based on weights similarity.

->Picking one client from each cluster at random.

5. Model Aggregation:

```
for train_loader in selected_clients:
    local_model = SimpleMLP()
    local_model.load_state_dict(global_model.state_dict())
    local_optimizer = optim.Adam(local_model.parameters(), lr=lr)

    for data, target in train_loader:
        local_optimizer.zero_grad()
        output = local_model(data)
        loss = nn.functional.nll_loss(output, target)
        loss.backward()
        local_optimizer.step()

    # Aggregate local model updates to the global model
    global_params = global_model.state_dict()
    local_params = local_model.state_dict()
    for key in global_params.keys():
        global_params[key] += local_params[key] / len(selected_clients)

global_model.load_state_dict(global_params)
```

->Iterate over selected clients (selected_clients) for training.
->Initialize local models for each of the selected clients and load the state of the global model.
->Train the local models using its respective data (train_loader).
->Aggregate the local model updates to the global model (global_model) by averaging the parameters across selected clients.
->Update the global model's parameters (global_model) with the aggregated updates.

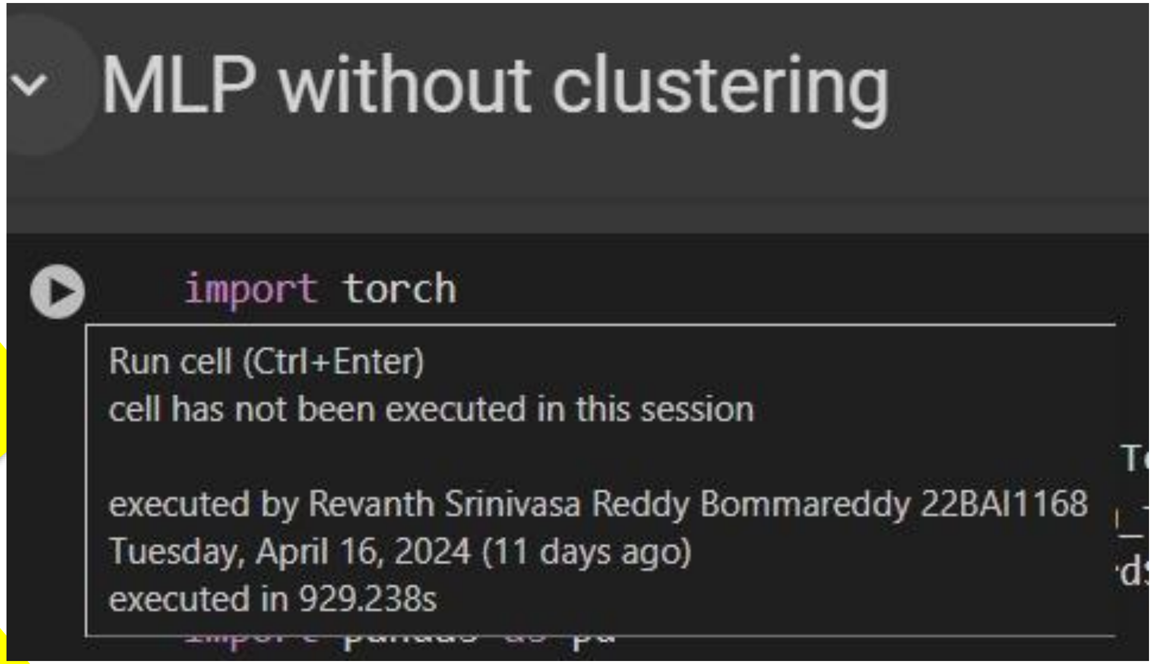
6. Evaluation:

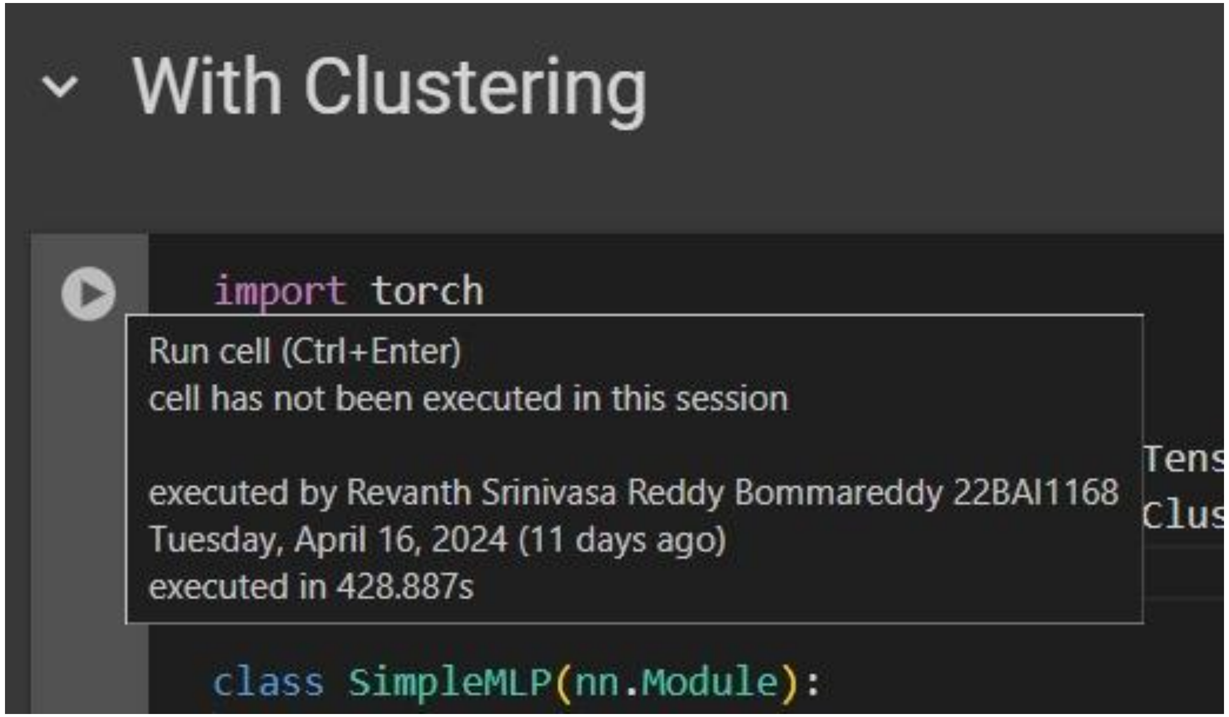
```
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        outputs = global_model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total
print(f"Epoch {epoch + 1}/{num_epochs} completed. Accuracy: {accuracy:.2f}%")
```

Explanation:

->Evaluate the performance of the trained global model (global_model) on the test dataset (test_loader).
->Compute the accuracy of the model by comparing predicted labels with ground truth labels.
->Print the accuracy after each epoch to monitor the model's progress during training.

Execution Time





Output

```
Epoch 1/7: Avg Loss: 0.0888, Reduction Ratio: inf, Threshold: 0.5, p: 8, d: 1
Adjusting clusters DOWN. New number of clusters: 7, d: 2
Epoch 1/7 completed. Accuracy: 97.50%
Epoch 2/7: Avg Loss: 0.0309, Reduction Ratio: 2.8753, Threshold: 0.5, p: 7, d: 2
Adjusting clusters DOWN. New number of clusters: 5, d: 3
Epoch 2/7 completed. Accuracy: 98.63%
Epoch 3/7: Avg Loss: 0.0394, Reduction Ratio: 0.7833, Threshold: 0.5, p: 5, d: 3
Epoch 3/7 completed. Accuracy: 99.22%
Epoch 4/7: Avg Loss: 0.2021, Reduction Ratio: 0.1951, Threshold: 0.5, p: 5, d: 3
Epoch 4/7 completed. Accuracy: 99.53%
Epoch 5/7: Avg Loss: 1.5653, Reduction Ratio: 0.1291, Threshold: 0.5, p: 5, d: 3
Epoch 5/7 completed. Accuracy: 99.68%
Epoch 6/7: Avg Loss: 15.9998, Reduction Ratio: 0.0978, Threshold: 0.5, p: 5, d: 1
Epoch 6/7 completed. Accuracy: 99.71%
Epoch 7/7: Avg Loss: 200.8382, Reduction Ratio: 0.0797, Threshold: 0.5, p: 5, d: 1
Epoch 7/7 completed. Accuracy: 99.72%
```

Thank You