1 Part II

1.1 Introduction

Federated Learning is a machine learning paradigm aimed at training a high-quality centralized model while keeping the training data distributed across numerous clients. This approach is particularly useful in scenarios where network connections may be unreliable or slow, and it ensures data privacy by allowing models to be trained without the need to access or move the data.

Traditionally, AI applications—such as spam filters, chatbots, and recommendation systems—were trained on massive datasets collected in centralized locations. These datasets often came from user contributions in exchange for free services like email or music streaming.

However, the landscape of AI is shifting towards a decentralized approach. Instead of aggregating data in a single location, modern AI models are being trained collaboratively at the edge—on devices like smartphones, laptops, or private servers. This data never leaves its original source, safeguarding user privacy.

Federated Learning is becoming the standard for addressing growing regulatory demands on data handling and storage. By processing data locally, it also enables efficient use of raw data generated by sensors on satellites, industrial machinery, smart home devices, and wearable technology, all without the need to transfer data to central servers.

As described in Part I, the parameter update rule in a distributed setting can be expressed as:

$$x_{t+1} = x_t - \eta \frac{1}{N} \sum_{i=1}^{N} \nabla f_i(x_t)$$

where:

- N : Number of devices
- $f_i(x_t)$: Loss function for data on device i

In this project, I've implemented various concepts from the paper Randomized Distributed Mean Estimation: Accuracy vs. Communication, which provides a foundational understanding of Federated Learning. This paper also introduces methods to reduce the information exchanged between the master and client nodes using specific encoding and communication protocols, optimizing bandwidth usage.

1.1.1 Project Prerequisites

Before diving into this section of the project, it's recommended to review Part I, as the implementation here builds on the *Proximal SGD* algorithm developed earlier. Additionally, a basic understanding of *Linear Algebra* is sufficient to follow the concepts discussed in this section.

1.1.2 Implementation Overview

In this project, I've utilized the MNIST dataset once again, dividing it into 5 parts for 5 clients. Each client is assigned its own model, which is trained on a subset of the dataset. The setup establishes a Federated Learning framework where:

1. A global model is initialized and distributed to the client nodes.

2. Each client trains its local model on its respective data subset.
3. The locally updated models are sent back to the master node.
4. The master node aggregates the updates and refines the global model.

This process is repeated iteratively until the global model converges.

```
import torch
from torch import nn
from torchvision import datasets
import fastai
from torchvision.transforms import ToTensor
# from fastai.data.core import DataLoader
from torch.utils.data import DataLoader
from fastai.data.core import DataLoaders
from fastai.callback.core import Callback
from fastai.vision.all import Learner, Metric
from fastai import optimizer
import torch.nn.functional as F
from torch.utils.data import Subset
import copy
```

```
training_data = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

test_data = datasets.MNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```

```
batch_size = 256

# Create data loaders.
train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

for X, y in test_dataloader:
    print(f"Shape of X [N, C, H, W]: {X.shape}")
    print(f"Shape of y: {y.shape} {y.dtype}")
    break
```

```
Shape of X [N, C, H, W]: torch.Size([256, 1, 28, 28])
```

```
Shape of y: torch.Size([256]) torch.int64
```

```python
# Get cpu, gpu or mps device for training.
device = (
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.backends.mps.is_available()
    else "cpu"
)
print(f"Using {device} device")
```

```
Using cuda device
```

```python
num_clients = 5
train_size = len(training_data)
# indices = list(range(train_size))

RANDOM_SEED = 42
torch.manual_seed(RANDOM_SEED)
torch.random.manual_seed(RANDOM_SEED)
indices = torch.randperm(train_size).tolist()

subset_size = train_size // num_clients
client_subsets = []
for i in range(num_clients):
    start_idx = i * subset_size
    end_idx = start_idx + subset_size

    if i == num_clients - 1:
        end_idx = train_size

    subset_indices = indices[start_idx:end_idx]
    client_subsets.append(Subset(training_data, subset_indices))

client_loaders = [DataLoader(sub, batch_size=batch_size, shuffle=True) for sub␣
 ↪in client_subsets]
```

```python
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
```

```
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork().to(device)
original_shapes = []
for p in model.parameters():
    original_shapes.append(p.shape)
model
```

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

1.1.3 The Three Protocols

For the entire process of training the global model, I've made use of 3 protocols, an encoding protocol, communication protocol and decoding protocol. I implemented the variable size encoder, sparse communication protocol for variable size encoder and functions for rebuilding the model weights and then the same for fixed size encoder. I update the global model with the average of the updates from the clients.

1.1.4 Variable Size Encoder

For each position (i, j) in the weights, we associate a parameter $p_{\{}ij\}, 0 < p_{\{}ij\} <= 1$. This parameter represents the probability of retaining the actual gradient value. The encoding protocol is defined as:

$$Y_i(j) = \begin{cases} \frac{X_i(j)-\mu_i}{p_{ij}(1-p_{ij})} & \text{with probability } p_{ij}, \\ \mu_i & \text{with probability } 1-p_{ij}, \end{cases}$$

where:

- $X_i(j)$: Original gradient value at position (i, j),
- μ_i : Predefined constant (e.g., the mean of the gradient vector),
- p_{ij} : Probability of encoding the gradient.

This probabilistic approach ensures that only a fraction of the data is communicated while retaining the essential information required to reconstruct the original gradient vector.

Key Properties of the Encoding Protocol

`1. Unbiasedness:`

The encoded gradient vector is an unbiased estimate of the original gradient. That is:

$$\mathbb{E}_\alpha[Y_i(j)] = X_i(j),$$

ensuring that the expected value of the encoded vector matches the original vector.

`2. Mean Squared Error (MSE):`

The Mean Squared Error (MSE) of the encoded vector is given by:

$$\mathrm{MSE}_\alpha(X_1, \ldots, X_n) = \frac{1}{n^2} \sum_{i,j} \left(\frac{1 - p_{ij}}{p_{ij}} \right) (X_i(j) - \mu_i)^2.$$

This equation demonstrates that smaller probabilities p_{ij} lead to larger reconstruction errors, highlighting a trade-off between compression and accuracy.

Code Explanation

The Variable Size Encoder is implemented in three main parts:

1. Encoding Gradients (*variable_ size_ encoder*):

- This function encodes gradient vectors using the described protocol.
- A random mask is generated for each position based on the probability matrix.
- Based on the mask, the encoded matrix is computed.

2. Sparse Representation (*sparse_for_ variable_ size_ encoder*):

- This function compresses the encoded vectors by storing only non-μ values along with their indices. This reduces the data size for communication.
- The sparse representation is a list of index-value pairs for each vector.

3. Reconstructing Gradients (*rebuild_from_ protocol_ 1*):

- This function reconstructs the original gradient vectors from the sparse representation.
- The indices and values are used to restore the original positions, while positions not in the sparse representation are set to μ .

1.1.5 Fixed Size Encoder

The *Fixed Size Encoder* is an alternative encoding protocol designed to ensure a deterministic communication cost, irrespective of the data distribution. Unlike the *Variable Size Encoder*, where the number of transmitted values depends on probabilities p_{ij} , the Fixed Size Encoder guarantees that exactly k values are communicated for each gradient vector. This is achieved by randomly selecting k positions in the gradient vector for encoding.

For a gradient vector X_i of size d , the encoding protocol works as follows:

1. Sample a subset D_i of k indices uniformly at random from $\{1, 2, \ldots, d\}$.
2. For each index j , encode the value $Y_i(j)$ as:

$$Y_i(j) = \begin{cases} \frac{d}{k} X_i(j) - \frac{d-k}{k} \mu_i & \text{if } j \in D_i, \\ \mu_i & \text{otherwise,} \end{cases}$$

where:

- $X_i(j)$: Original gradient value at position j ,
- μ_i : A predefined constant (e.g., the mean of the gradient vector),
- d : Total number of elements in the vector,
- k : Fixed number of encoded elements.

This ensures that the size of the support of Y_i is always k , i.e., $|S_i| = k$.

Key Properties of the Encoding Protocol

`1. Unbiasedness:`

The encoded gradient vector is an unbiased estimate of the original gradient. That is:

$$\mathbb{E}_\alpha[Y_i(j)] = X_i(j).$$

`2. Mean Squared Error (MSE):`

The Mean Squared Error (MSE) of the encoded vector is given by:

$$\text{MSE}_\alpha(X_1, \ldots, X_n) = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{d} \left(\frac{d-k}{k} \right) (X_i(j) - \mu_i)^2.$$

Code Explanation The Fixed Size Encoder is implemented in three main parts:

1. Encoding Gradients (*fixed_ size_ encoder*)

This function encodes gradient vectors using the fixed-size encoding protocol:

- A random subset of k indices is sampled from the gradient vector.
- For the sampled indices, the values are transformed using the encoding formula.
- For other indices, the value is set to μ .

2. Sparse Representation (*sparse_for_fixed_ size_ encoder*)

This function compresses the encoded vectors by storing only the non-μ values. It creates a sparse representation that reduces communication overhead.

3. Reconstructing Gradients (*rebuild_from_ protocol_ 2*)

This function reconstructs the original gradient vectors from the sparse representation:

- The indices of selected elements are regenerated using the same random seed.

- The sparse values are placed at the appropriate indices, while other positions are filled with μ .

1.1.6 Averaging Decoder

For updating the global model, we are going to take mean of the model updates from the clients and update the global model with that mean.

Given the model updates $U_1, U_2, \ldots, U_N$ from N clients, the global model update G is computed as:

$$G = \frac{1}{N} \sum_{i=1}^{N} U_i,$$

where:

- U_i represents the update from the i-th client, typically the difference between the client's updated model parameters and the global model parameters,
- N is the total number of participating clients.

The global model is then updated as:

$$W_{\text{global}} \leftarrow W_{\text{global}} + G,$$

where W_{global} represents the parameters of the global model.

```
# We now have 5 different datasets, each with some sort of representation of the␣
 ↪data that is unknown, ie, we have no
# statistical information on the data that each of these clients would have
# We now need to implement variations of the 3 protocols, namely, the encoding␣
 ↪protocol, the communication protocol and the decoding protocol

# For communication protocol for fixed size encoder, we set the seed. So the␣
 ↪seed is communicated with the values.
SEED = 42
torch.manual_seed(SEED)

# Encoders
def variable_size_encoder(grad_vectors, mu, p=0.1):
    # Lets take p = 0.1
    new_grad_vectors = []
    with torch.no_grad():
        for i in range(len(grad_vectors)):
            mask = torch.rand_like(grad_vectors[i], device=grad_vectors[i].
 ↪device) < p
            Y = torch.empty_like(grad_vectors[i], device=grad_vectors[i].device)
            Y[mask] = (grad_vectors[i][mask] - mu[i] * (1-p))/p
            Y[~mask] = mu[i]
```

```
            new_grad_vectors.append(Y)
    return new_grad_vectors

def fixed_size_encoder(grad_vectors, mu, seed, k=40):
    # k can vary
    orig = k
    new_grad_vectors = []

    with torch.no_grad():
        for i in range(len(grad_vectors)):
            k = orig
            shape = grad_vectors[i].shape
            # Flattening the parameters to permutate over them
            flat_grad = grad_vectors[i].view(-1)
            d = flat_grad.numel()
            k = min(k, d)
            torch.manual_seed(seed)
            # Shuffle the list [1, 2, ... d] and get the first k elements
            indices = torch.randperm(d, device=flat_grad.device)[:k]
            mask = torch.zeros(d, dtype=torch.bool, device=flat_grad.device)
            mask[indices] = True

            Y = torch.empty_like(flat_grad)
            # Encode the parameters
            chosen_vals = (d/k)*flat_grad[mask] - ((d-k)/k)*mu[i]
            Y[mask] = flat_grad[mask]
            Y[~mask] = mu[i]
            Y = Y.view(shape)
            new_grad_vectors.append(Y)
    return new_grad_vectors


# Decoders : I wont be making use of this later on
def averaging_decoder(grad_vectors_list):
    if isinstance(grad_vectors_list, list):
        grad_vectors_list = torch.stack(grad_vectors_list, dim=0)
    return torch.mean(grad_vectors_list, dim=0)

# Communication protocols
def sparse_for_variable_size_encoder(encoded_vectors, mu):
    final_vectors = []
    with torch.no_grad():
        for i in range(len(encoded_vectors)):
            flat_vector = encoded_vectors[i].view(-1)
            mask = flat_vector != mu[i]
            # vals = encoded_vectors[i][mask]
            indices = torch.nonzero(mask, as_tuple=False).view(-1)
```

```
            values = flat_vector[mask]
            final_vectors.append(list(zip(indices, values)))


    return final_vectors, mu

def sparse_for_fixed_size_encoder(encoded_vectors, mu, seed):
    final_vectors = []
    with torch.no_grad():
        for i in range(len(encoded_vectors)):
            flat_vector = encoded_vectors[i].view(-1)
            mask = torch.zeros(len(flat_vector), dtype=torch.bool,␣
 ↪device=flat_vector.device)
            mask[flat_vector != mu[i]] = True
            values = flat_vector[mask]
            final_vectors.append(values)

    return final_vectors, mu, seed

def rebuild_from_protocol_1(final_vectors, mu, original_shapes):
    rebuilt_vectors = []
    with torch.no_grad():
        for i, vec_data in enumerate(final_vectors):
            num_elements = 1

            for dim_size in original_shapes[i]:
                num_elements *= dim_size


            Y_flat = torch.full((num_elements,), mu[i], dtype=torch.float32,␣
 ↪device=mu[i].device)

            indices = torch.tensor([pair[0] for pair in vec_data], dtype=torch.
 ↪long, device=Y_flat.device)
            values = torch.tensor([pair[1] for pair in vec_data], dtype=Y_flat.
 ↪dtype, device=Y_flat.device)
            Y_flat[indices] = values
            Y = Y_flat.view(original_shapes[i])
            rebuilt_vectors.append(Y)
    return rebuilt_vectors

def rebuild_from_protocol_2(final_vectors, mu, seed, original_shapes):
    rebuilt_vectors = []
    with torch.no_grad():
        for i, values in enumerate(final_vectors):

            num_elements = 1
```

```
            for dim_size in original_shapes[i]:
                num_elements *= dim_size


            k = len(values)  # number of chosen elements
            d = num_elements
            torch.manual_seed(seed)
            indices = torch.randperm(d, device=device)[:k]
            Y_flat = torch.full((num_elements,), float(mu[i]), dtype=torch.
 ↪float32, device=mu[i].device)
            sorted_indices, sorted_pos = torch.sort(indices)
            sorted_values = values[sorted_pos]
            Y_flat[sorted_indices] = values
            chosen_mask = (Y_flat != mu[i])
            rebuilt_vectors.append(Y_flat.view(original_shapes[i]))

    return rebuilt_vectors


parameters = list(model.parameters())
mu_1 = []
with torch.no_grad():
    for p in parameters:
        mu_1.append(torch.mean(p))
encoded_vectors = fixed_size_encoder(parameters, mu_1, SEED, k=1000)
final_vectors, mu, SEED = sparse_for_fixed_size_encoder(encoded_vectors, mu_1,␣
 ↪SEED)
rebuilt_vectors = rebuild_from_protocol_2(final_vectors, mu_1, SEED,␣
 ↪original_shapes)
```

```
class ProxSGDWithLinearSearch:
    def __init__(self, params, lr):
        self.params, self.lr = list(params), lr
        self.state = {p: {} for p in self.params}
        self.hypers = [{'lr': lr}]
        self.max_iter = 5
        self.eta = 1e-5

    def soft_threshold(self, x, eta):
        # Apply the soft-thresholding operator
        return F.softshrink(x, lambd=eta)

    def prox_operator(self, x):
        # Use the soft-thresholding operator as the proximal step
        return self.soft_threshold(x, self.eta)

    def Gt(self, x, step_size, x_grad):
```

```
        return (1/step_size) * (x - self.prox_operator(x - step_size * x_grad))

    def step(self, *args, **kwargs):
        model = kwargs.get("model")
        loss_fn = kwargs.get("loss_fn")
        X = kwargs.get("X")
        y = kwargs.get("y")

        orig_params = [p.data.clone() for p in self.params]
        step_size = self.lr
        with torch.no_grad():
            pred = model(X)
            old_loss = loss_fn(pred, y)
        flag = True
        for _ in range(self.max_iter):
            for p in self.params:
                if p.grad is not None:
                    Gt_val = self.Gt(p.data, step_size, p.grad.data)
                    p.data = p.data - step_size * Gt_val
            with torch.no_grad():
                pred = model(X)
                new_loss = loss_fn(pred, y)
            if new_loss < old_loss:
                flag = False
                break
            else:
                for i, j in zip(self.params, orig_params):
                    i.data.copy_(j)
                step_size *= 0.5
        if flag:
            for p in self.params:
                if p.grad is not None:
                    Gt_val = self.Gt(p.data, step_size, p.grad.data)
                    p.data = p.data - step_size * Gt_val
        else:
            self.lr = step_size
        # print(self.lr)
    def zero_grad(self, *args, **kwargs):
        for p in self.params:
            p.grad = None

    def set_hypers(self, **kwargs):
        if 'lr' in kwargs:
            self.lr = kwargs['lr']
            self.hypers[0]['lr'] = kwargs['lr']
```

```
```

```
loss_fn = nn.CrossEntropyLoss()
```

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step(model=model, loss_fn=loss_fn, X=X, y=y)
        optimizer.zero_grad()

        if batch % 100 == 0:
            print(f"BATCH: {batch} of {size/batch_size} batches")
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

```
def test(dataloader, model, loss_fn, number="main"):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error for client {number}: \n Accuracy: {(100*correct):>0.1f}%,␣
 ↪Avg loss: {test_loss:>8f} \n")
```

Code Explanation

I developed two classes to simplify debugging and ensure reproducibility of results. The *Client* class includes methods for training and testing the model at the client, setting model parameters, and retrieving encoded parameters. The *Master* class contains methods to update parameters based on the encoding protocol used, along with a method for testing the model.

```
class Client:
    def __init__(self, model, train_dataloader, loss_fn, mu, max_iter=2):
        self.model = model
        # self.optimizer = ProxSGDWithLinearSearch(model.parameters(), 20)
        self.train_dataloader = train_dataloader
        self.max_iter = max_iter
        self.loss_fn = loss_fn
        self.mu = mu

    def train(self):
        for i in range(0, self.max_iter):
            train(self.train_dataloader, self.model, self.loss_fn,␣
↪ProxSGDWithLinearSearch(self.model.parameters(), 20))

    def test(self, test_dataloader, number):
        test(test_dataloader, self.model, self.loss_fn, number)

    def set_parameters(self, model):
        self.model = copy.deepcopy(model)

    def get_encoded_1(self, p):
        encoded = variable_size_encoder(list(self.model.parameters()), self.mu,␣
↪p)
        final_vectors, mu = sparse_for_variable_size_encoder(encoded, self.mu)
        return final_vectors, mu

    def get_encoded_2(self, k, seed):
        encoded = fixed_size_encoder(list(self.model.parameters()), self.mu,␣
↪seed, k)
        self.encoded = encoded
        final_vectors, mu, seed = sparse_for_fixed_size_encoder(encoded, self.
↪mu, seed)
        # encoded_vectors = encoded
        return final_vectors, mu, seed

    # Built the member function to test if continuous update of the mean is␣
↪helpful or not
    def update_mean(self):
        with torch.no_grad():
            params = list(self.model.parameters())
            self.mu = [torch.mean(p) for p in params]
```

```
class Master:
    def __init__(self, model, mu, loss_fn):
        self.model = model
        self.original_shapes = [p.shape for p in model.parameters()]
```

```
        self.mu = mu
        self.loss_fn = loss_fn

    def set_mean(self, mu):
        self.mu = copy.deepcopy(mu)

    def update_global_model_from_protocol_1(self, clients_data):
        # clients_data is a list of tuples (final_vectors, mu, p) from each␣
↪client
        # Decode each client's parameters and then compute updates
        decoded_params_list = []
        for (final_vectors, mu) in clients_data:
            decoded = rebuild_from_protocol_1(final_vectors, mu, self.
↪original_shapes)
            decoded_params_list.append(decoded)

        master_params = list(self.model.parameters())
        all_updates = []
        with torch.no_grad():
            for decoded_params in decoded_params_list:
                updates = [(dp - mp) for dp, mp in zip(decoded_params,␣
↪master_params)]
                all_updates.append(updates)

            # Average updates across clients
            # Stack each parameter across clients and mean
            averaged_updates = []
            num_clients = len(all_updates)
            for param_idx in range(len(master_params)):
                stack = torch.stack([all_updates[c][param_idx] for c in␣
↪range(num_clients)], dim=0)
                avg = torch.mean(stack, dim=0)
                averaged_updates.append(avg)

            # Apply averaged updates to master model
            for mp, au in zip(master_params, averaged_updates):
                mp.data.add_(au)

    def update_global_model_from_protocol_2(self, clients_data):
        # clients_data is a list of tuples (final_vectors, mu, seed, k) from␣
↪each client
        decoded_params_list = []
        for (final_vectors, mu, seed) in clients_data:
            decoded = rebuild_from_protocol_2(final_vectors, mu, seed, self.
↪original_shapes)
            decoded_params_list.append(decoded)
```

```
        self.decoded_params = decoded_params_list
        self.clients_data = clients_data
        master_params = list(self.model.parameters())
        all_updates = []
        with torch.no_grad():
            for decoded_params in decoded_params_list:
                updates = [(dp - mp) for dp, mp in zip(decoded_params,␣
↪master_params)]
                all_updates.append(updates)

            num_clients = len(all_updates)
            averaged_updates = []
            for param_idx in range(len(master_params)):
                stack = torch.stack([all_updates[c][param_idx] for c in␣
↪range(num_clients)], dim=0)
                avg = torch.mean(stack, dim=0)
                averaged_updates.append(avg)

            # Apply averaged updates to master model

            for mp, au in zip(master_params, averaged_updates):
                mp.data.add_(au)

    def test(self, test_dataloader):
        test(test_dataloader, self.model, self.loss_fn, "master")
```

```
model = NeuralNetwork().to(device)

parameters = list(model.parameters())
mu_1 = []
with torch.no_grad():
    for p in parameters:
        mu_1.append(torch.mean(p))
# mu_1 = torch.zeros(len(parameters), device=device)
master = Master(model, mu_1, loss_fn)
clients = [Client(NeuralNetwork().to(device), client_loaders[i], loss_fn, mu_1,␣
↪5) for i in range(num_clients)]
```

```
# Encoder 1

for i in range(1):
    # Master sends global parameters to the client
    for client in clients:
        client.set_parameters(master.model)
        # client.update_mean()

    # Clients are trained locally
```

```
    for client in clients:
        client.train()

    clients_data_protocol_1 = []
    for client in clients:
        final_vectors, mu = client.get_encoded_1(p=0.5)
        clients_data_protocol_1.append((final_vectors, mu))

    master.update_global_model_from_protocol_1(clients_data_protocol_1)

    master.test(test_dataloader)
```

```
# Encoder 2
base_seed = 42

for i in range(5):
    # Master sends global parameters to the client
    for client in clients:
        client.set_parameters(master.model)
        client.update_mean()

    # Clients are trained locally
    for client in clients:
        client.train()

    clients_data_protocol_2 = []
    for client_id, client in enumerate(clients):
        current_seed = base_seed + i * 1000 + client_id
        torch.manual_seed(current_seed)
        final_vectors, mu, seed = client.get_encoded_2(k=5000000,␣
 ↪seed=current_seed)
        clients_data_protocol_2.append((final_vectors, mu, seed))

    master.update_global_model_from_protocol_2(clients_data_protocol_2)

    master.test(test_dataloader)
```

```
master.test(test_dataloader)
```

```
Test Error for client master:
 Accuracy: 97.8%, Avg loss: 0.068344
```

```
for i in range(len(clients)):
    clients[i].test(test_dataloader, i)
```

```
Test Error for client 0:
 Accuracy: 93.8%, Avg loss: 0.199948
```

```
Test Error for client 1:
 Accuracy: 94.3%, Avg loss: 0.184125

Test Error for client 2:
 Accuracy: 94.2%, Avg loss: 0.184488

Test Error for client 3:
 Accuracy: 94.6%, Avg loss: 0.170945

Test Error for client 4:
 Accuracy: 94.0%, Avg loss: 0.196233
```