Implementation of Image Classification Training Based on FedAvg Algorithm

Baicheng Chen*, Boxuan Wen†, Yeke Zhang‡ and Zixiang Hao†
* *The Chinese University of Hong Kong, Shenzhen*
Email: baichengchen@link.cuhk.edu.cn
†*Harbin Institute of Technology, Shenzhen*
‡*Northeastern University*

***Abstract*—This project mainly focus on how to use Federated Learning to solve the image classification task and find the best accuracy through parameter tuning. Also, we discover the influence of important hyperparameters and heterogeneity. At last, we discuss the application and future works.**

1. Introduction

This study concentrates on image classification because we find that image classification plays important roles in many fields. In the medical field, image classification can be used to save medical resources and improve medical accuracy by classifying images taken by a machine to determine what kind of disease it is. In the field of autonomous driving, image classification can analyze road conditions and vehicle distance to ensure the safety of autonomous driving. In agriculture, image classification can judge the growth of grains or the condition of the soil. In security, image classification can perform face recognition and fingerprint recognition.

However, most of the current image classification methods have greater security risks and waste communication costs. In order to isolate the data to meet security laws. This study aims to train a federated learning model to do the image classification task.

Our group compares some traditional machine learning algorithms to federated learning. LeNet [1] is robust and fast when trained with a simple dataset, but it needs more parameters and has serious security. AlexNet [2] uses Local Response Normalization(LRN) to reduce the influence of noise. However, it oversaturates when data are too big or small. VGG [3] uses 3×3 convolution to enhance the depth of the network and successfully enhance the effectiveness, but it needs more memory usage.

So, we choose Federated Learning (FL) [4]. It can solve privacy problems and have a considerable speed and available memory usage. So it will be suitable to be applied to image classification.

2. Methods

2.1. Dataset Preparation

To make sure that data labelling is completed accurately in the training phase to avoid discrepancies in the data, we choose a typical publicly available dataset CIFAR-10. Compared to MNIST, CIFAR10 is a colour image dataset that is closer to real objects. The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images [5].

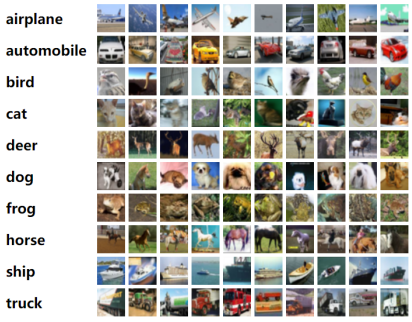


Figure 1: CIFAR-10

We choose CIFAR-10 because compared to handwritten characters, CIFAR-10 contains real objects in the real world, which are not only noisy, but also have different scales and features of the objects, which makes it very difficult for recognition. Direct linear models such as Softmax perform poorly on CIFAR-10.

2.2. Algorithm

In the project, we use FedAvg Algorithm [4]. FedAvg is a commonly used FL algorithm that aggregates model parameters through weighted averaging. This algorithm fits very well with the project because of the low communication cost, support for heterogeneous data, and strong generalization.

The basic procedure of FedAvg is as follows:

Algorithm 1 `FederatedAveraging`

Server executes:
initialize w_0
for each round $t = 1, 2, \dots$ **do**
  $S_t = (\text{random set of } \max(C \cdot K, 1) \text{ clients})$
  for each client $k \in S_t$ **in parallel do**
    $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
  $w_{t+1} \leftarrow \sum_{t=1}^{K} \frac{n_k}{n} w_{t+1}^k$

ClientUpdate(k, w): *// Executed on client k*
for each local epoch i from 1 to E **do**
  batches $\leftarrow$ (data $\mathcal{P}_k$ split into batches of size B)
  for batch b in batches **do**
    $w \leftarrow w - \eta \nabla \ell(w; b)$
return w to server

Figure 2: Structure of FedAvg

1. The server initializes a global model w_0.

2. It chooses a subset of clients to participate in the training round.

3. The server sends w_0 to all the clients.

4. The local training starts. The model is trained using the local data in several iterations on each client device. During each iteration, the clients first calculate the gradient g using w and local data, then update w using the locally calculated gradient g.

5. Then, all the uploaded models w_i are sent back to the server. The server averages all the w_i and gets $\overline{w_i}$.

6. Finally, repeat the steps 2 to 6 until convergence or other desired levels.

2.3. Code Explanation

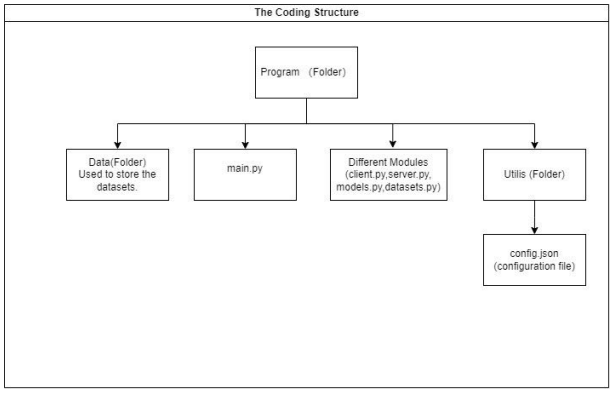


Figure 3: Code Structure

This is an intuitive figure of the structure of our program. Note that nearly all the model parameters are in the *config.json* file, which is important in later parameter tuning.

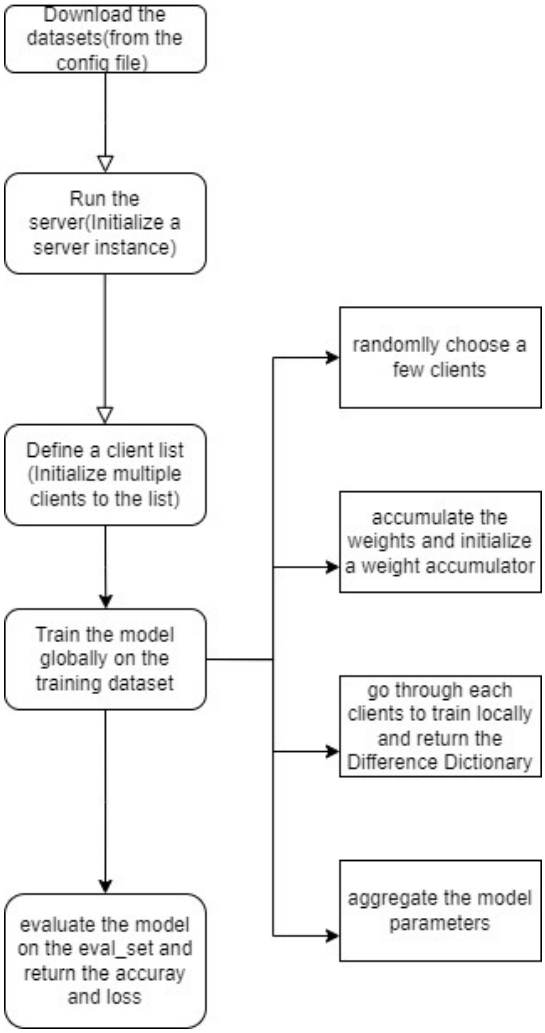


Figure 4: Coding Logic

As this figure shows above, the important parts are the server, client, global training, and some further parameter tuning.

```
# function of aggregation
# weight_accumulator  stores the variance of the parameters uploaded by each clients
def model_aggregate(self, weight_accumulator):
    # a global model that goes through each clients
    for name, data in self.global_model.state_dict().items():
        #  each time updating the parameter, multiply the lamba(in the config file) to t
        update_per_layer = weight_accumulator[name] * self.conf["lambda"]
        # accumulation
        if data.type() != update_per_layer.type():
        # since the updata_per_layer is floattensor type, we should transform it to the
            data.add_(update_per_layer.to(torch.int64))
        else:
            data.add_(update_per_layer)
```

Figure 5: Initialize A Server

According to the FedAvg algorithm, the server is a special type of data structure that can handle the aggregation of the model parameters. After the updating of the model parame-

ters, it should also be able to redistribute the parameters to local clients.

```
# Local model training function
def local_train(self, model):
    # The client receives the server's model and then trains it using a portion of the local dataset
    for name, param in model.state_dict().items():
        # Overwrite the local model with the global model issued by the server
        self.local_model.state_dict()[name].copy_(param.clone())
    # Define the optimizer for local model training
    optimizer = torch.optim.SGD(
        self.local_model.parameters(),
        lr=self.conf['lr'],
        momentum=self.conf['momentum']
    )
    # Train the model locally
    # Set the model to training mode
    self.local_model.train()
    # Start training the model
    for e in range(self.conf["local_epochs"]):
        for batch_id, batch in enumerate(self.train_loader):
            data, target = batch
            # Load to GPU if available
```

Figure 6: Define The Client List

By definition of the FedAvg algorithm, client should receive the parameters from the central server and use them to train multiple rounds locally. So we define the *local_train* function in the client class. Then we create a list containing several clients.

```
class Client(object):
    def __init__(self, conf, model, train_dataset, id=-1):
        # Read the configuration file
        self.conf = conf
        # Get the local model for the client (usually transmitted by the server)
        self.local_model = models.get_model(self.conf["model_name"])
        # Client ID
        self.client_id = id
        # Client's local dataset
        self.train_dataset = train_dataset
        # Customize the dataset splitting logic based on client ID
        all_range = list(range(len(self.train_dataset)))
        data_len = len(self.train_dataset)

        # Define the proportion of data each client receives (uneven distribution)
        # You can customize this distribution as needed
        client_data_proportions = [0.01,0.03,0.05,0.07,0.09,0.11,0.13,0.15,0.17,0.19]  # Example: 60%, 30%, 10% distribution

        # Ensure that client_data_proportions has enough elements to cover all possible client IDs
        while len(client_data_proportions) < conf["no_models"]:
            # Extend the list with random proportions
            client_data_proportions.append(random.uniform(0.05, 0.5))  # Random proportion between 5% and 50%

        # Calculate the number of samples each client receives based on proportions
        total_proportion = sum(client_data_proportions)
        client_data_sizes = [int(data_len * proportion / total_proportion) for proportion in client_data_proportions]

        # Assign data indices to clients based on their ID
        start_index = sum(client_data_sizes[:id])
        end_index = start_index + client_data_sizes[id]
        train_indices = all_range[start_index:end_index]
```

Figure 7: Generating The Heterogeneous Effects

In order to achieve the heterogenous effects on different clients in perspective of the local dataset size, we innovatively defined the proportion array to split the total dataset in certain proportions and distribute them. This will help us figure out the model's robustness against the dataset heterogeneity.

```
for e in range(conf["global_epochs"]):
    print("Global Epoch %d" % e)

    # Randomly choose k clients to train
    candidates = random.sample(clients, conf["k"])
    print("Selected clients are: ")
    for c in candidates:
        print(c.client_id)

    # Accumulate the weights
    weight_accumulator = {}

    # Initialize the empty model parameters weight_accumulator
    for name, params in server.global_model.state_dict().items():
        # Generate a zero matrix that has the same size as the parameter matrix
        weight_accumulator[name] = torch.zeros_like(params)
```

Figure 8: Globally Training 1

When we conducted the global training, this was the first step. Select the clients and define the *weights_accumulator*, a special type of data structure that can transmit the parameters based on certain weights between the server and the clients.

```
    # Go through the chosen clients and train locally
    for c in candidates:
        diff = c.local_train(server.global_model)
        # Update the global weights based on the returned parameter of the
        for name, params in server.global_model.state_dict().items():
            weight_accumulator[name].add_(diff[name])

    # Aggregate the model parameters
    server.model_aggregate(weight_accumulator)
```

Figure 9: Globally Training 2

Then go through each local client and train locally. At the end of this part, the model parameters will be aggregated.

```
client_numbers = [1,3,5]  # Client numbers
bars = plt.bar(client_numbers, client_accuracies[-1])  # Plot the bar chart for the last epoch
plt.xlabel('Client Number')
plt.ylabel('Accuracy')
plt.title('Client versus Accuracy')

# Annotate each bar with its height
for bar in bars:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width() / 2, height, '%.2f' % height, ha='center', va='bottom')

# Save the plot to a specific location
plt.savefig('C:/Users/lenovo/Desktop/pic/homo.png')

# Show the plot
plt.show()
```

Figure 10: Evaluate The Model

We use the matplotlib-based functions to draw some charts to evaluate the model's performance. Specifically, we compare the accuracy versus the number of clients under different levels of heterogeneity. Note that we define the following level of heterogeneity:

1. Homogeneity, or in other words, No Heterogeneity. In this scenario, we split the whole dataset evenly and distributed them to the local clients.

2. Lower Level Of Heterogeneity. We split the dataset following the proportions [0.01, 0.03, 005, 0.07, 0.09, 0.11, 0.13, 0.15, 0.17, 0.19]. Obviously, as an arithmetic progression with the common difference of 0.02, it's heterogenous. But compared to some other proportions, it's relatively even.

3. Higher Level Of Heterogeneity. We split the dataset following the proportions [0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.91]. This is a highly uneven array which will easily lead to high heterogeneity.

3. Results and Discussion

3.1. Running Results

When we trained the model by the FedAvg Algorithm, we needed to set the hyperparameters as follows: Total number of clients, global epochs, local epochs, number of

clients selected to participate in each round, batch size, and learning rate.

After parameter tuning, we found the best values of the parameters which guarantee both efficiency and model accuracy. We let the Total number of clients = 5, Global epochs = 20, Local epochs = 3, Number of clients participating in each training or communication round = 3, Batch size = 32, and Learning rate = 0.001. And our final testing accuracy is **71.51%**. Here is the running result's screenshot.

```
Client 4 local train done
Client 4 local train done
Client 4 local train done
Client 4 local train done
Epoch 19, acc: 71.510000, loss: 0.828486
```

Figure 11: Running result

3.2. Discussion

Among the hyperparameters, we think the number of clients and global epochs are worth exploring. So, we set the total number of clients as K, global epochs as E, number of clients participating in each training round as C.

3.2.1. Global Epochs.

In the parameter tuning process, we first choose global epochs because it has a deep influence on the convergence of the trained model. We try it use loops and draw the accuracy and loss curve. Finally, when K = 5 and C = 3, we found the best E = 20, which has the best performance.

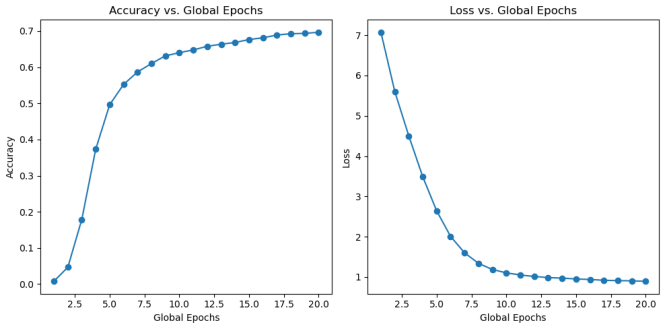


Figure 12: Curves of FL

3.2.2. Clients.

If we change the number of clients, we can see the difference between Centralized and Distributed ML.

When K = C = 1, we get a centralized ML model. In Centralized ML, the data is uploaded to the server and it may waste the computility. When K and C are more than 1, we get a FL model. In FL, the models are uploaded from the clients to the server and it is safe and efficient.

So, based on these, we did some experiments to verify. This figure shows that when C = 1, namely centralized ML, needs more than 25 global epochs to converge. And when we experimented, it took a very long time to run, longer than 20 epochs of FL. So compared with the figure of FL before, we verified FL is more efficient and uses less global epochs to converge.

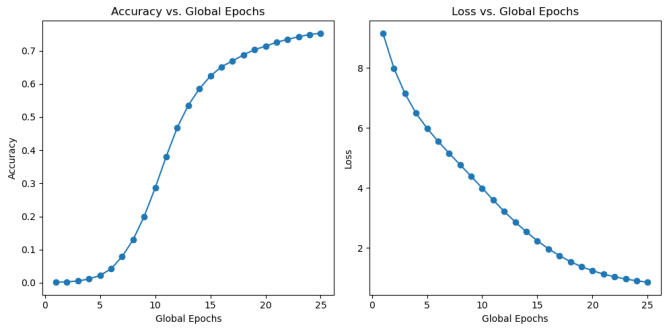


Figure 13: Curves of Centralized ML

Except for security and efficiency, we also found the number of clients also influences accuracy. In this bar chart, the accuracy increases as the number of clients increases.

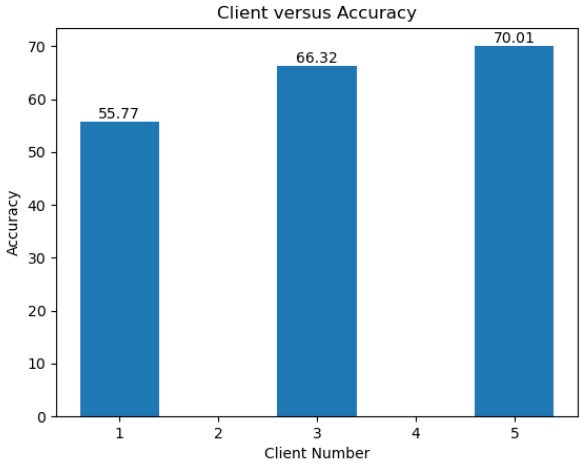


Figure 14: Clients vs. Accuracy Bar Chart

3.2.3. Heterogeneity.

The last work we did was the heterogeneity analysis. In FL, heterogeneity is a big problem. In the original paper on FL, Google's researchers claimed that heterogeneity problems of FL contain Non-IID data, unbalanced proportions, massively distributed clients, and limited communication.

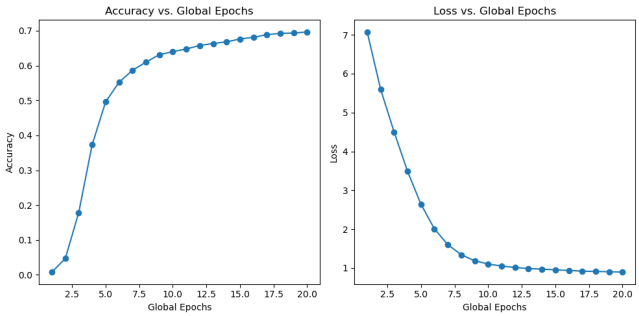
Our group did some experiments to change the proportions of data on each client as mentioned before and explore the influence of heterogeneity (Mentioned in **Code**

Explanation part).

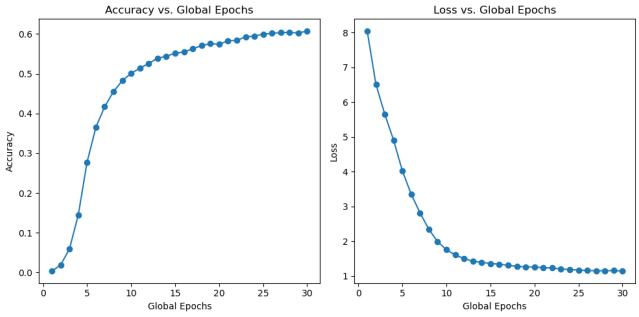
Based on the partitioned method, we finished the experiment and got some results. This figure shows the running result's screenshot of the experiment. The accuracy dropped obviously and the loss increased.

```
Client 1 local train done
Client 1 local train done
Client 1 local train done
Client 1 local train done
Epoch 29, acc: 60.760000, loss: 1.144015
```

Figure 15: Heterogeneity Experiment Result



(a) Homogeneous

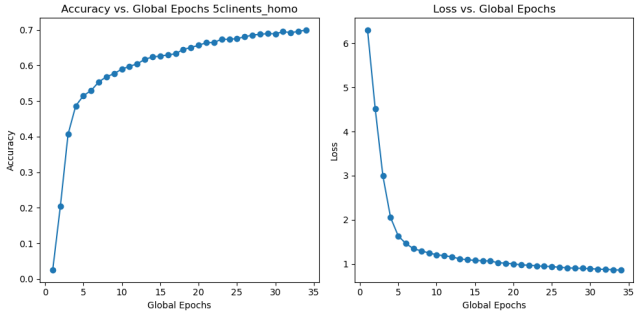


(b) Heterogeneous

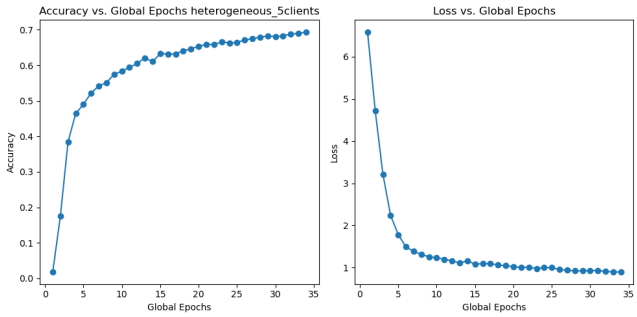
Figure 16: Experiment with K=5 & C=3

When we compared the accuracy and loss curves between the homogeneous experiment (a) and the heterogeneous experiment (b), we found the differences. First is the accuracy. The accuracy of (a) is almost 10 percent higher than (b). And for the loss curve, (b) takes more global epochs to converge than (a). Also, the curve in (a) is smoother than (b).

We also do another experiment to show the features more clearly. We set K = 10 and C = 5, then the features in the curves are zoomed in.



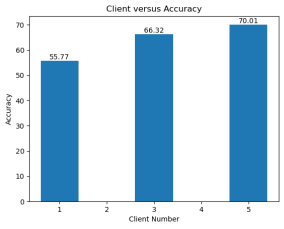
(a) Homogeneous



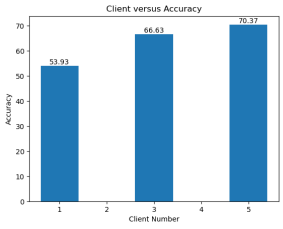
(b) Heterogeneous

Figure 17: Experiment with K=10 & C=5

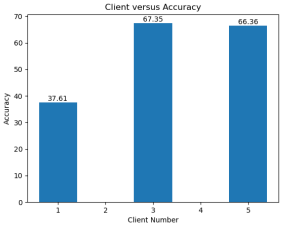
Furthermore, we also discovered the relationship among the number of clients, the extent of heterogeneity and the accuracy. We drew three bar charts to show the results, representing the situation of homogeneity (a), lower level of heterogeneity (b) and higher level of heterogeneity (c) respectively.



(a) Homogeneity



(b) Low



(c) High

Figure 18: Clients, Heterogeneity vs. Accuracy

By analyzing these figures, we can draw the following conclusions:

1. The Lower Level Of Heterogeneity will not make the model perform worse in a noticeable manner, which indicates that our model has some sort of robustness against the Heterogeneity.

2. When dealing with the Higher Level Of Heterogeneity, if the number of clients who take part in each round is set to be 1 or 5, the performance becomes worse in some way. But when it is set to 3, the performance is very good, which tells us maybe we can improve the robustness by carefully adjusting the number of clients.

4. Conclusion

FedAvg is a distributed Machine Learning Algorithm that enables model training across multiple devices while preserving data privacy. In this article, we described our approach and discussed the performance of our model throughout the training iterations. We analyzed key metrics such as accuracy and loss to gauge the effectiveness of our training strategy. Next, we dived into model evaluation and parameter tuning, elaborating on the evaluation metrics employed to assess our model's performance and outlining strategies for parameter tuning to optimize our model further. Finally, we explored the challenges posed by data heterogeneity, discussing the implications of varying data distributions across different clients and proposing potential solutions to address this issue. In conclusion, our project offers insights into the FedAvg algorithm's effectiveness in training models on decentralized data sources like CIFAR-10. Through analysis and discussion, we aim to contribute to the ongoing research in Federated Learning and its applications in the future.

Given the features of Federated Learning, it has broad application prospects. For example, it could be used in privacy protection, handling data heterogeneity, continuous optimization, and improved service quality. One specific field for Federated Learning to apply is the medical field, which could protect the privacy of patients and promote model generalization.

Algorithm 2 `FedProx` (Proposed Framework)

Input: $K, T, \mu, \gamma, w^0, N, p_k, k = 1, \cdots, N$
for $t = 0, \cdots, T-1$ **do**
  Server selects a subset S_t of K devices at random (each device k is chosen with probability p_k)
  Server sends w^t to all chosen devices
  Each chosen device $k \in S_t$ finds a w_k^{t+1} which is a γ_k^t-inexact minimizer of: $w_k^{t+1} \approx \arg\min_w \ h_k(w;\ w^t) = F_k(w) + \frac{\mu}{2}\|w - w^t\|^2$
  Each device $k \in S_t$ sends w_k^{t+1} back to the server
  Server aggregates the w's as $w^{t+1} = \frac{1}{K}\sum_{k \in S_t} w_k^{t+1}$
end for

Figure 19: Structure of FedProx

To improve our project further, we got inspiration from the FedProx algorithm [6]. As a Federated Learning method, FedProx offers a solution to the challenge of data heterogeneity by introducing a regularization term. This adjustment helps reduce the disparity in data distributions among different clients, ultimately enhancing the model's generalization performance.

In our project, which utilized the FedAvg algorithm to train CIFAR-10, we did not initially consider regularization. However, we recognized the potential of this approach to mitigate the impact of data heterogeneity. By incorporating regularization techniques inspired by FedProx, we aim to make our project more adept at handling diverse datasets.

Acknowledgments

The authors would like to express the heartfelt gratitude for the guidance of Professor Soummya Kar.

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[4] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*. PMLR, 2017, pp. 1273–1282.

[5] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[6] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," *Proceedings of Machine learning and systems*, vol. 2, pp. 429–450, 2020.