Implementation of Image Classification Training Based on FedAvg Algorithm

Use CIFAR-10 Dataset

Boxuan Wen[1] Baicheng Chen[2] Zixiang Hao[1] Yeke Zhang[3]

[1]Harbin Institute of Technology, Shenzhen

[2]The Chinese University of Hong Kong, Shenzhen

[3]Northeastern University

March 2, 2024

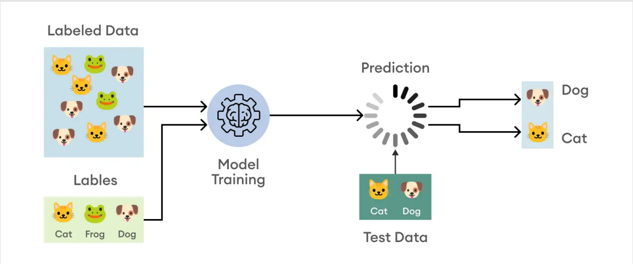
Table of Contents

Table of Contents

Image Classification

Image Classification is a fundamental task in vision recognition. It involves assigning a label or tag to an entire image based on preexisting training data of already labelled images. It is widely used in various fields.

- Medical images
- Autonomous driving
- Agriculture
- Security



Existing Methods

We investigate some of the existing methods and algorithms to do the image classification task.

LeNet

Have a robustness

Training with a simple dataset is fast

More parameters and serious security

AlexNet

Use LRN(Local Response Normalization) to reduce the influence of noise

Oversaturate when data are too big or small

VGG

Use 3×3 convolution to Enhancing the depth of the network and successfully enhance the effectiveness

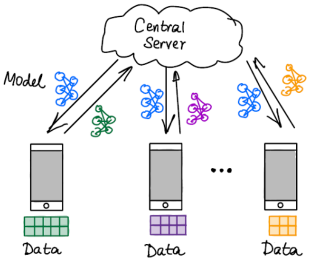
More memory usage

[4, 5, 6]

Algorithm

Most of the current image classification methods adopt Centralized ML or Distributed On-Site Learning, which have greater security risks and waste communication costs.

In order to isolate the data, when the data will not be leaked to the outside, to meet the user's privacy protection and data security requirements. Our group aims to train a federal learning model to do the image classification task.



Comparison

Compare with LeNet —— Better security

Data are isolated and will not be leaked to the outside, to meet the user's privacy protection and data security laws.

Compare with AlexNet —— Ensure that the quality of the model is undamaged

Ensure that there is no negative transfer, and that the federated model is guaranteed to work better than a cut-away independent model

Compare with VGG —— Less memory

Reduce memory usage for the same size of data

Dataset

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.

- 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. [1]

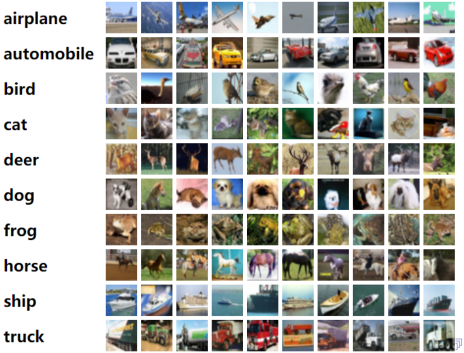


Table of Contents

FedAvg Algorithm

FedAvg is a commonly used FL algorithm that aggregates model parameters through weighted averaging. It has lots of advantages and can solve our target problem very well. [2]

- Low communication cost: Only need to upload local model parameters.
- Support for heterogeneous data: Local device can use different datasets.
- Strong generalization: Train a global model using local data on all devices.

FedAvg Algorithm

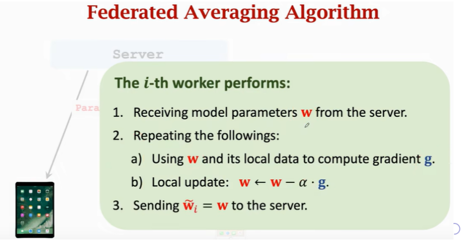
Here are the steps:

1. Initialization: The server initializes a global model w_0.
2. Client selection: To take part in the training round, a subset of clients is chosen.
3. Model distribution: The chosen clients receive the global model w_0. A duplicate of the model is given to each client.
4. Local training: The model is trained using the local data in several iterations on each client device. And get local models w_i.

Algorithm 1 `FederatedAveraging`

Server executes:
  initialize w_0
  for each round $t = 1, 2, \dots$ **do**
    $S_t =$ (random set of $\max(C \cdot K, 1)$ 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



FedAvg Algorithm

5. Model aggregation: The updated models w_i from each client are sent back to the central server after local training.
6. Model averaging: The central server aggregates the models received from the clients by averaging the model parameters. And finally get $\overline{w_i}$.
7. Repeat: Steps 2 to 6 are repeated for multiple training rounds until convergence or a desired level of performance is achieved.

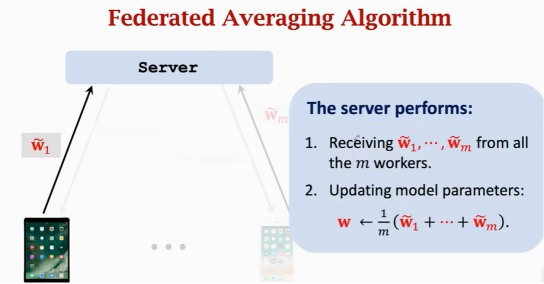
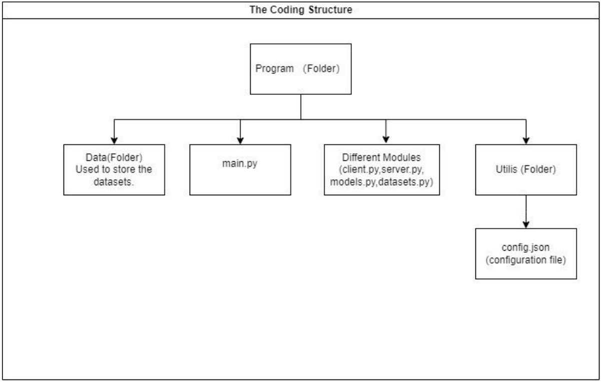
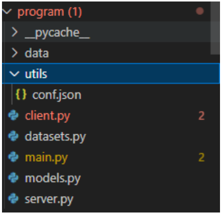


Table of Contents

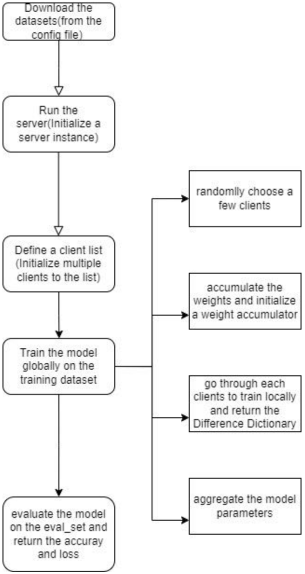
Code Explanation

This is how we store the program.





Code Explanation



I will then present the following important points.

- A. How did we initialize a server instance?
- B. How did we define a client list? What did we do to innovatively distribute the dataset unevenly so as to simulate the heterogeneous inputs?
- C. What are the basic procedures to train the model globally?
- D. What did we do to evaluate the model and help our further parameter tuning?

Code Explanation

```
# 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)
```

Part A: How did we initialize a server?

Basic idea: By defining the data structure to aggregate the parameters and redistribute the updated ones.

Code Explanation

Part B: How did we define a client list?

Basic idea: Define several client instances which can receives the updated model and achieve the local training.

```python
# 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
```

Code Explanation

```python
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]
```

Part B: How did we generate the heterogeneous effects?

Innovations: By defining a client_data_proportions, we successfully stimulated the heterogeneous effects regarding the local data size.

Code Explanation

Part C: What are the basic procedures to conduct the global training?

First of all, select the clients and define the weight_accumulator, which is basically a data structure to transmit the parameters between the central server and the clients.

```python
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)
```

Code Explanation

```
# 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)
```

Part C: What are the basic procedures to conduct the global training?

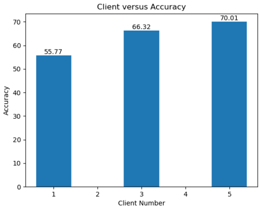
Then, we go through the local clients and train locally. After that, the new parameters are aggregated.

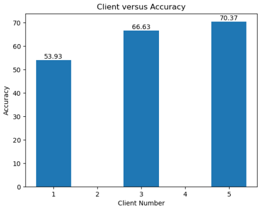
Code Explanation

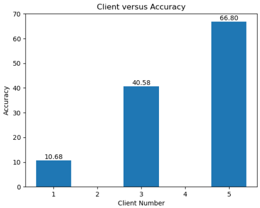
Part D: What did we do to evaluate the model?

Basic idea: plot_based functions

```python
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)
```







Parameters

When we train the model by FedAvg Algorithm, we need to set the hyperparameters as follows:

- Total number of clients: K
- Global epochs: E
- Local epochs
- Number of clients participating in each training or communication round: C
- Batch size
- Learning rate

Final Values of Parameters

After trial and error, we found the best values of the parameters which guarantee both efficiency and model accuracy.

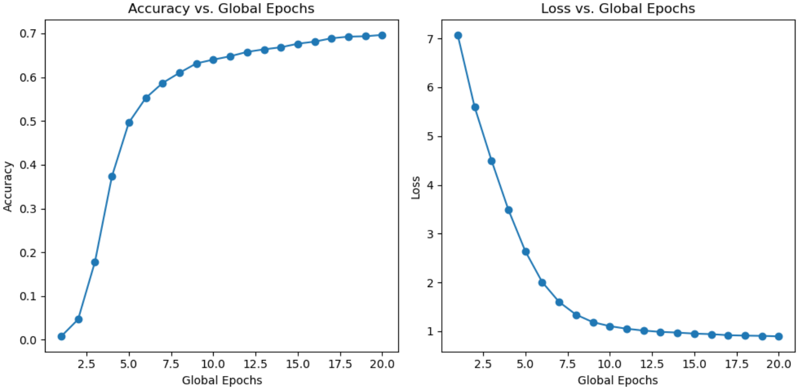
- Total number of clients $= 5$
- Global epochs $= 20$
- Local epochs $= 3$
- Number of clients participating in each training or communication round $= 3$
- Batch size $= 32$
- Learning rate $= 0.001$

Final Accuracy $= 71.51\%$

```
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
```

Global Epochs

When we did parameter tuning, we first chose global epochs because it has a deep influence on the convergence of the trained model. Finally, when K=5 and C=3, we found the best E=20, which guarantees both model convergence and high test accuracy.



Clients

If we change the number of clients, we can see the difference of Centralized and Distributed ML. [3]

Client $= 1$

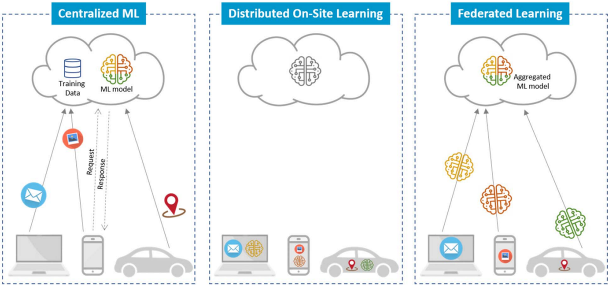
Centralized ML

- Data uploaded to the server
- Waste the computility

Client > 1

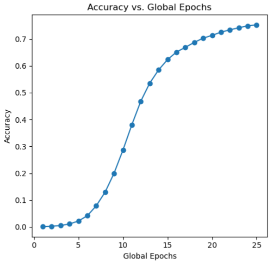
Federated Learning

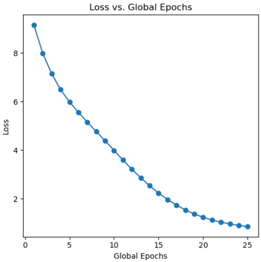
- Models uploaded to the server
- Safe and efficient



Clients

We did some experiments to verify these features.
When client=1, namely Centralized ML, really takes a long time to train and needs more global epochs to converge.
When client>1, namely FL, is more efficient and uses less global epochs to converge.
The following figure shows that when C=1 and E=25, the loss curve is just about to converge.





Clients

Except for security and efficiency, we found the number of clients also influences Accuracy.

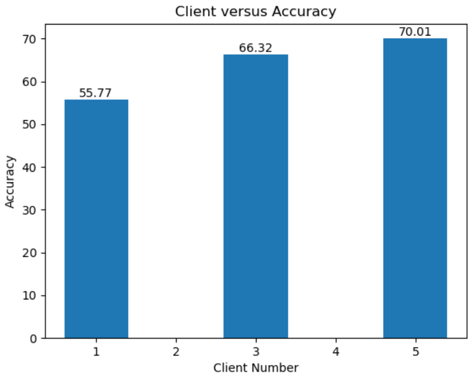


Figure: Testing Accuracy

Heterogeneity

In Federated Learning, Heterogeneity is a big problem. [2]

Federated Optimization We refer to the optimization problem implicit in federated learning as federated optimization, drawing a connection (and contrast) to distributed optimization. Federated optimization has several key properties that differentiate it from a typical distributed optimization problem:

- **Non-IID** The training data on a given client is typically based on the usage of the mobile device by a particular user, and hence any particular user's local dataset will not be representative of the population distribution.
- **Unbalanced** Similarly, some users will make much heavier use of the service or app than others, leading to varying amounts of local training data.
- **Massively distributed** We expect the number of clients participating in an optimization to be much larger than the average number of examples per client.
- **Limited communication** Mobile devices are frequently offline or on slow or expensive connections.

So, we classify the situations of Heterogeneity into Data Heterogeneity and Computational Heterogeneity.

Heterogeneity may be caused by the dataset, computing resources, or models' local parameters, etc.
Our group did some experiments to explore the Data Heterogeneity caused by the dataset, namely the amount of data used to train on each client.

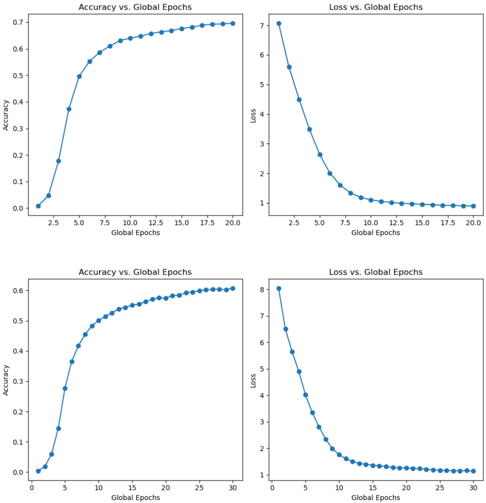
Heterogeneity

We set different proportions of data for each client (Which have already been introduced in the Code Explanation part). Then, we get some figures.

```
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
```

Heterogeneity

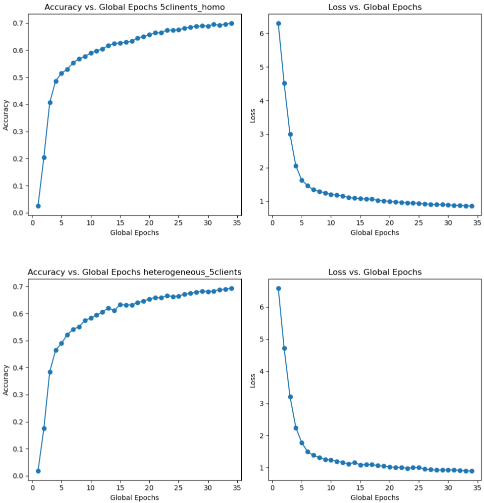
Compare the convergence curve with the homogeneous experiment.



- Accuracy
- Loss
- Epochs to converge
- Curve Smoothness

Heterogeneity

Compare the convergence curve with the homogeneous experiment.



- Accuracy
- Loss
- Epochs to converge
- Curve Smoothness

Table of Contents

Conclusion of the Project

- Introduction to the FedAvg algorithm and the CIFAR-10 dataset
- Analysis of the training process and results
    - Detailed description of the training process
    - Analysis of the model performance and accuracy during the training iterations
- Discussion on model evaluation and parameter tuning
    - Explanation of the evaluation metrics used to assess the model's performance
    - Strategies for parameter tuning and optimizing the model for better results
- Exploration of heterogeneity problems

Expectations for Future Research

Future prospects of federated learning applied in the medical field:

- Privacy Protection
- Handling Data Heterogeneity
- Continuous Optimization
- Improved Service Quality

Expectations for Future Research

Inspiration from FedProx:
FedProx is a federated learning algorithm that improves upon FedAvg by addressing data heterogeneity. It introduces a regularization term to adjust the weights of client updates, which helps to reduce the divergence in data distributions among different clients and enhances the model's generalization performance. [7]

As for our project, we did not consider the solution of regularization. Through this idea, we may be able to make our project more suitable for handling heterogeneous data.

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

References

[1] Alex Krizhevsky, Learning Multiple Layers of Features from Tiny Images. (2009)

[2] McMahan, Brendan, et al. Communication-efficient learning of deep networks from decentralized data. (2023)

[3] Sawsan AbdulRahman, Hanine Tout, et.al. A Survey on Federated Learning: The Journey From Centralized to Distributed On-Site Learning and Beyond. (2020)

[4] Cesare Alippi, Simone Disabato, Manuel Roveri, Moving Convolutional Neural Networks to Embedded Systems: The AlexNet and VGG-16 Case. (2018)

[5] Rashad Al-Jawfi, Handwriting Arabic Character Recognition LeNet Using Neural Network. (2009)

[6] M. Bath and L.G. Mansson, Visual grading characteristics (VGC) analysis: a non-parametric rank-invariant statistical method for image quality evaluation. (2006)

[7] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, Virginia Smith, Federated Optimization in Heterogeneous Networks. (2020)

Thank you!

Let's move on Q&A!