

Topics

Intro

HPC

Federated Lab

Pytorch

<https://pytorch.org/docs/stable/index.html>

Environment Setup: OnDemand

Use this link to register and use
ODU HPC services

[ODU HPC](#)

Log in to On Demand

[ODU On Demand HPC](#)

ODU HPC Documentation

[Docs](#)

HPC: home/user

ODU RCC OnDemand Files ▾ Jobs ▾ Clusters ▾ Interactive Apps **My Interactive Sessions** Help ▾ Logged in as ahoop004 Log Out

Open in Terminal Refresh New File New Directory **Upload** Download Copy/Move Delete

Home Directory / home / ahoop004 / Change directory Copy path

Show Owner/Mode Show Dotfiles Filter: Showing 13 of 43 rows - 0 rows selected

Type	Name	Size	Modified at
Folder	Cltraining	-	7/30/2024 10:15:40 AM
Folder	com_proj	-	3/18/2025 12:49:56 PM
Folder	Desktop	-	7/30/2024 10:04:22 AM
Folder	envs	-	2/13/2025 12:32:33 PM
Folder	Federated	-	3/7/2025 3:40:01 PM
Folder	gan	-	2/27/2025 5:32:18 PM
Folder	Lab4_DDoS_attack_Detection	-	2/13/2025 12:40:30 PM
Folder	ondemand	-	7/26/2024 12:02:17 PM
Folder	sumo	-	8/15/2024 11:14:30 AM

HPC: Jupyter

ODU RCC OnDemand [Files](#) [Jobs](#) [Clusters](#) [Interactive Apps](#) [My Interactive Sessions](#) [Help](#) [Logged in as ahoop004](#) [Log Out](#)

Home / My Interactive Sessions

You have no active sessions.

Interactive Apps

Desktops
Desktop
Servers
Jupyter
Matlab
RStudio Server
Tensorboard
VS Code

HPC: Launch

Interactive Apps

- Desktops
- Desktop
- Servers
- Jupyter
- Matlab
- RStudio Server
- Tensorboard
- VS Code

Jupyter

This app will launch a Jupyter server using Python on the wahab cluster.

Python Version

Python 3.10

Python Suite

pytorch 2.5.1

Pick a Python suite to start Jupyter. Read more in [HPC Wiki](#).

If you cannot find the module you need, try switch **Python Version** and **Partition**

Additional Module Directory

none

Picks up additional modules from a user-environment directory within [~/envs/](#).

Number of Cores

4

Number of cores on node type (about 8 GB per core unless requesting whole node).

Number of GPU

1

Partition

gpu

GPU device is always highly demanded, therefore you might experience long wait time if use gpu or high-gpu-mem partition.

Authorized user only partitions are not public resource, they are reserved for their device owner, submit to it will result an error.

Number of Hours

8

Launch

* The Jupyter session data for this session can be accessed under the data root

HPC: Navigate to lab

The screenshot shows a Jupyter Notebook interface with the following details:

- File Tree:** On the left, a sidebar titled "Federated /" lists files and their modification times:
 - Data (21h ago)
 - Logs (last mo.)
 - FedLab.ipynb** (5m ago) - This file is highlighted with a red border.
 - load_data_for_clients... (last mo.)
 - model.py (14h ago)
 - util_functions.py (20h ago)
- Code Cells:** The main area contains three code cells labeled [1], [2], and [3].
 - Cell [1]:** Imports os, copy, math, logging, matplotlib.pyplot, and util_functions. It also imports torch and its submodules.

```
[1]: import os
from copy import deepcopy
from math import ceil
import logging
import matplotlib.pyplot as plt
from util_functions import set_logger, save_plt
import torch
```
 - Cell [2]:** Imports numpy, torch.utils.data, DataLoader, model, dist_data_per_client, and util_functions. It defines a #FedAvg block.

```
[2]: import numpy as np
from torch.utils.data import DataLoader
from model import *
from load_data_for_clients import dist_data_per_client
from util_functions import set_seed, evaluate_fn

#FedAvg
```
 - Cell [3]:** Defines a Client class with __init__ and client_update methods.

```
[3]: class Client():

    def __init__(self, client_id, local_data, device, num_epochs, criterion, lr):
        self.id = client_id
        self.data = local_data
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.num_epochs = num_epochs
        self.lr = lr
        self.criterion = criterion
        self.x = None
        self.y = None

    def client_update(self):

        self.y = deepcopy(self.x)
        self.y.to(self.device)

        for epoch in range(self.num_epochs):

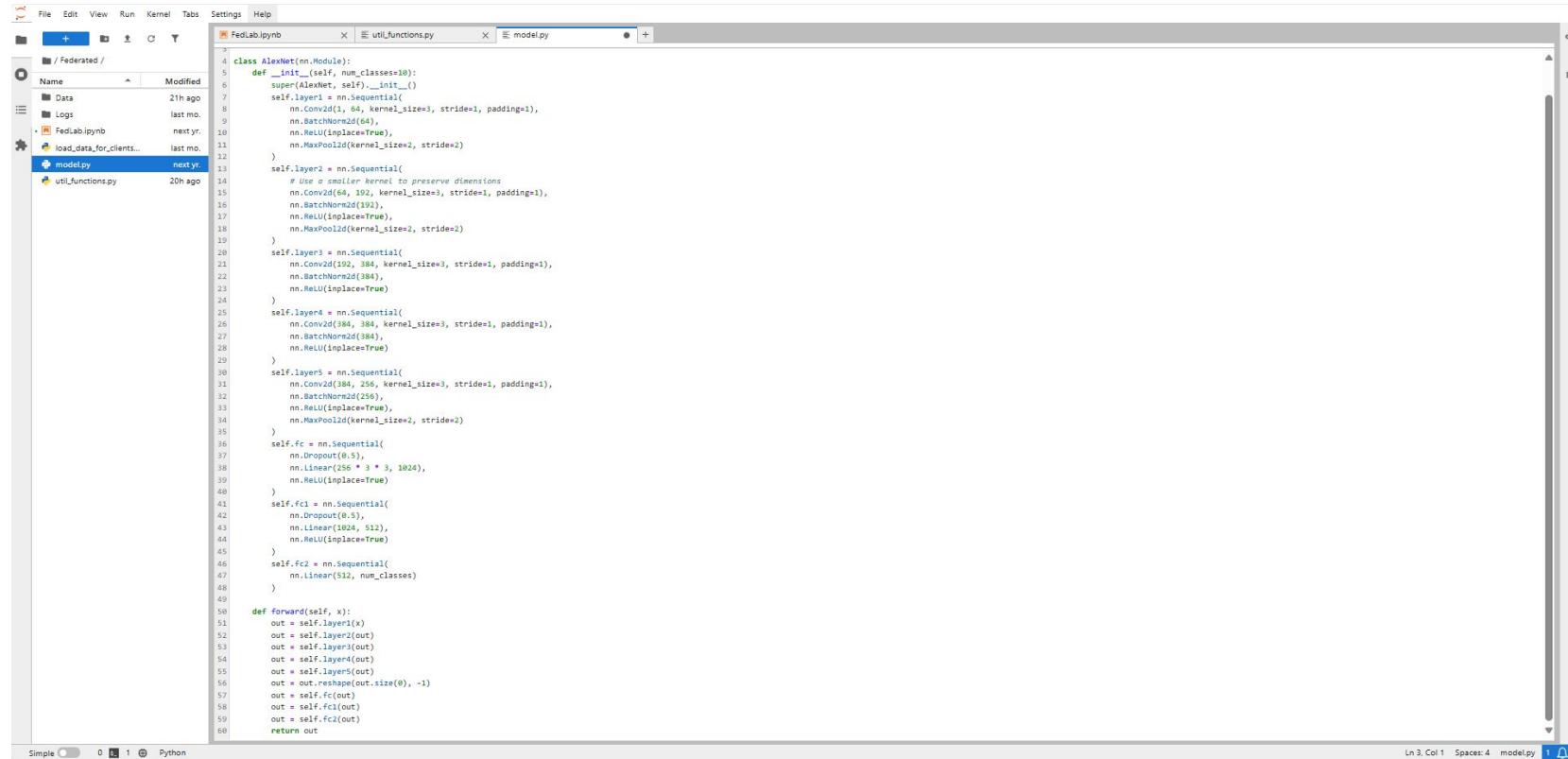
            data_iter = iter(self.data)

            inputs, labels = next(data_iter)
            inputs, labels = inputs.float().to(self.device), labels.long().to(self.device)
            output = self.y(inputs)
            loss = self.criterion(output, labels)
            grads = torch.autograd.grad(loss, self.y.parameters())

            with torch.no_grad():
                for param, grad in zip(self.y.parameters(), grads):
                    param.data = param.data - self.lr * grad.data

            if self.device == "cuda": torch.cuda.empty_cache()
```
- Kernel:** Python 3 (ipykernel) is selected at the top right.
- Status Bar:** At the bottom, it shows "Mode: Command" and the file name "FedLab.ipynb".

Model: Alexnet



The screenshot shows a Jupyter Notebook interface with three tabs open: 'FedLab.ipynb', 'util_functions.py', and 'model.py'. The 'model.py' tab is active and displays the Python code for the AlexNet neural network.

```
4 class AlexNet(nn.Module):
5     def __init__(self, num_classes=10):
6         super(AlexNet, self).__init__()
7         self.layer1 = nn.Sequential(
8             nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1),
9             nn.BatchNorm2d(64),
10            nn.ReLU(inplace=True),
11            nn.MaxPool2d(kernel_size=2, stride=2)
12        )
13        self.layer2 = nn.Sequential(
14            # Use a smaller kernel to preserve dimensions
15            nn.Conv2d(64, 192, kernel_size=3, stride=1, padding=1),
16            nn.BatchNorm2d(192),
17            nn.ReLU(inplace=True),
18            nn.MaxPool2d(kernel_size=2, stride=2)
19        )
20        self.layer3 = nn.Sequential(
21            nn.Conv2d(192, 384, kernel_size=3, stride=1, padding=1),
22            nn.BatchNorm2d(384),
23            nn.ReLU(inplace=True)
24        )
25        self.layer4 = nn.Sequential(
26            nn.Conv2d(384, 384, kernel_size=3, stride=1, padding=1),
27            nn.BatchNorm2d(384),
28            nn.ReLU(inplace=True)
29        )
30        self.layer5 = nn.Sequential(
31            nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1),
32            nn.BatchNorm2d(256),
33            nn.ReLU(inplace=True),
34            nn.MaxPool2d(kernel_size=2, stride=2)
35        )
36        self.fc = nn.Sequential(
37            nn.Dropout(0.5),
38            nn.Linear(156 * 3 * 3, 1024),
39            nn.ReLU(inplace=True)
40        )
41        self.fc1 = nn.Sequential(
42            nn.Dropout(0.5),
43            nn.Linear(1024, 512),
44            nn.ReLU(inplace=True)
45        )
46        self.fc2 = nn.Sequential(
47            nn.Linear(512, num_classes)
48        )
49
50    def forward(self, x):
51        out = self.layer1(x)
52        out = self.layer2(out)
53        out = self.layer3(out)
54        out = self.layer4(out)
55        out = self.layer5(out)
56        out = out.reshape(out.size(0), -1)
57        out = self.fc(out)
58        out = self.fc1(out)
59        out = self.fc2(out)
60
61    return out
```

The code defines the AlexNet architecture using PyTorch's nn.Sequential container. It consists of five convolutional layers followed by two fully connected layers. The first four convolutional layers each end with a max pooling layer. The final convolutional layer has a stride of 2. The fully connected layers have 1024 and 512 units respectively, with a dropout of 0.5 after each. The final layer outputs 10 classes. The code uses nn.ReLU(inplace=True) for all activation functions except the final one, which uses nn.ReLU(). The code is annotated with comments explaining its structure and parameters.

Utility functions: Data

The screenshot shows a Jupyter Notebook interface with three open files: `FedLab.ipynb`, `util_functions.py`, and `model.py`. The `util_functions.py` file is the focus, containing Python code for data loading and transformation.

```
File Edit View Run Kernel Tabs Settings Help
FedLab.ipynb util_functions.py model.py
Federated /
Name Modified
Data 21h ago
Logs last mo.
FedLab.ipynb next yr.
load_data_for_clients... last mo.
model.py next yr.
util_functions.py 20h ago

59     # set transformation differently per dataset
60     if (dataset_name == "CIFAR10"):
61         T = transforms.Compose([
62             # transforms.Resize(244),
63             transforms.ToTensor(),
64             transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
65         ])
66
67     elif (dataset_name == "MNIST"):
68         T = transforms.Compose([
69             transforms.ToTensor(),
70             transforms.Normalize((0.1307,), (0.3081,)),
71         ])
72
73
74     train_data = datasets.__dict__[dataset_name](root=data_path, train=True, download=True,
75                                                 transform=T)
76     test_data = datasets.__dict__[dataset_name](root=data_path, train=False, download=True,
77                                                 transform=T)
78
79 else:
80     # dataset not found exception
81     error_message = f"dataset '{dataset_name}' is not supported or cannot be found in TorchVision Datasets!"
82     raise AttributeError(error_message)
83
84     # unsqueeze channel dimension for grayscale image datasets
85     if train_data.data.ndim == 3: # convert to NxHxW -> NxHxWx1
86         train_data.data.unsqueeze_(3)
87
88     return train_data, test_data
89
90 # This class is used to get data in batches
91 class load_data(Dataset):
92     def __init__(self, x, y):
93         self.length = x.shape[0]
94         self.x = x.permute(0,3,1,2)
95
96         # self.image_transform = transforms.Normalize((127.5, 127.5, 127.5), (127.5, 127.5, 127.5))
97         self.image_transform = transforms.Normalize((0.1307,), (0.3081,))
98
99     def __getitem__(self, index):
100         image, label = self.x[index], self.y[index]
101         image = self.image_transform(image)
102         return image, label
103
104     def __len__(self):
105         return self.length
106
107 # A simple utility function for converting pytorch tensors to numpy
108 def tensor_to_numpy(data, device):
109     if device.type == "cpu":
110         return data.detach().numpy()
111     else:
112         return data.cpu().detach().numpy()
113
114 # A simple utility function for converting numpy to pytorch tensors
115 def numpy_to_tensor(data, device, dtype="float"):
116     if dtype=="float":
117         return torch.tensor(data, dtype=torch.float).to(device)
```

The code defines a `load_data` class that inherits from `Dataset`. It takes `x` and `y` as input. The `x` tensor is permuted to `(0,3,1,2)`. The `image_transform` is set to `transforms.Normalize((0.1307,), (0.3081,))`. The `__len__` method returns the length of `x`. The `__getitem__` method returns a tuple of `image` and `label`.

Below the class definition, there are two utility functions: `tensor_to_numpy` and `numpy_to_tensor`. `tensor_to_numpy` converts a PyTorch tensor to a NumPy array. If the tensor is on the CPU, it uses `detach()` to remove it from the computation graph. If it's on a GPU, it moves it to the CPU first. `numpy_to_tensor` converts a NumPy array to a PyTorch tensor. It checks if the `dtype` is specified as "float" and returns a float tensor if true, otherwise a double tensor.

Line numbers 59 through 116 are highlighted with a red box, indicating the specific code being discussed in the slide.

Federated Learning Lab

Based on the work of
Balaji Varatharajan

Reference code and presentation

[GitHub - BalajiAI/Federated-Learning: Implementation of Federated Learning algorithms such as FedAvg, FedAvgM, SCAFFOLD, FedOpt, Mime using PyTorch.](#)

[Aggregation algorithms in Federated Learning](#)

Papers

FedAvg:[\[1602.05629\] Communication-Efficient Learning of Deep Networks from Decentralized Data](#)

FedOpt:(adam,adagrad,yogi):[\[2003.00295\] Adaptive Federated Optimization](#)

SCAFFOLD:[\[1910.06378\] SCAFFOLD: Stochastic Controlled Averaging for Federated Learning](#)

Base Federated Learning Vocabulary

Federated Learning (FL): A distributed learning paradigm where multiple clients (devices) train a shared model without centralizing data.

Client: The individual devices or nodes that perform local training on private data.

Server: The central coordinator that aggregates updates from clients.

Local Update: The process where each client trains the model on its local data.

Global Model: The aggregated model obtained after combining client updates.

Federated Averaging (FedAvg): The baseline aggregation algorithm that averages client model updates.

Communication Round: A complete cycle of local training and subsequent aggregation on the server.

Data Heterogeneity (Non-IID Data): The variability in data distribution across different clients.

Privacy Preservation: Techniques used to ensure client data remains private during training.

Scalability: The system's ability to handle a large number of clients efficiently.

FedAvg

Client i

$$y_i = x$$

$$y_i = y_i - \eta \frac{\partial L}{\partial y_i}$$

communicate y_i

Server

$$x = \frac{1}{|S|} \sum_{i \in S} y_i$$

communicate x

FedAvg

Client i

$$y_i = y_i - \eta \frac{\partial L}{\partial y_i}$$

```
grads = torch.autograd.grad(loss, self.y.parameters())
with torch.no_grad():
    for param, grad in zip(self.y.parameters(), grads):
        param.data = param.data - self.lr * grad.data
```

Server

$$x = \frac{1}{|S|} \sum_{i \in S} y_i$$

```
with torch.no_grad():
    for idx in client_ids:
        for a_y, y in zip(avg_y, self.clients[idx].y.parameters()):
            a_y.data.add_(y.data / int(self.fraction * self.num_clients))
    for param, a_y in zip(self.x.parameters(), avg_y):
        param.data = a_y.data
```

Adaptive Federated Optimization Vocab

Adaptive Learning Rate: An approach where the learning rate is automatically adjusted based on historical gradient information.

FedAdam / FedAdagrad / FedYogi: Variants of adaptive optimizers (inspired by [Adam](#), [Adagrad](#), and [Yogi](#)) tailored for federated settings.

Momentum: A technique that incorporates previous updates to smooth and accelerate convergence.

Bias Correction: Adjustments made (e.g., in Adam) to correct the estimates of moment statistics.

Gradient Scaling: Methods to adjust gradients (or learning rates) based on their magnitudes or variance.

Optimizer Hyperparameters: Parameters such as beta coefficients in Adam that control decay rates and other dynamics.

Convergence Stability: The algorithm's ability to reliably reach a minimum despite data heterogeneity and noisy gradients.

Client Drift: The divergence in local updates due to non-IID data that adaptive methods aim to counteract.

FedAdagrad

Client i

$$y_i = x$$

$$y_i = y_i - \eta_l \frac{\partial L}{\partial y_i}$$

$$\Delta y_i = y_i - x$$

communicate Δy_i

Server

$$g = \frac{1}{|S|} \sum_{i \in S} \Delta y_i$$

$$s = s + g^2$$

$$x = x + \frac{\eta_g}{\sqrt{s + \epsilon}} g$$

communicate x

FedAdagrad

$$\text{Client } i \quad y_i = y_i - \eta_l \frac{\partial L}{\partial y_i}$$

```
with torch.no_grad():
    for param, grad in zip(self.y.parameters(), grads):
        param.data = param.data - self.lr * grad.data
```

$$\Delta y_i = y_i - x$$

```
with torch.no_grad():
    delta_y = [torch.zeros_like(param, device=self.device) for param in self.y.parameters()]
    for del_y, param_y, param_x in zip(delta_y, self.y.parameters(), self.x.parameters()):
        del_y.data += param_y.data.detach() - param_x.data.detach()
    self.delta_y = delta_y
```

Server

$$g = \frac{1}{|S|} \sum_{i \in S} \Delta y_i$$

```
with torch.no_grad():
    for idx in client_ids:
        for grad, diff in zip(gradients, self.clients[idx].delta_y):
            grad.data.add_(diff.data / int(self.fraction * self.num_clients))
```

$$s = s + g^2$$

$$x = x + \frac{\eta_g}{\sqrt{s + \epsilon}} g$$

```
for p,g,s in zip(self.x.parameters(), gradients, self.s):
    s.data += torch.square(g.data)
    p.data += self.lr * g.data / torch.sqrt(s.data + self.epsilon)
```

FedAdam

Client i

$$y_i = x$$

$$y_i = y_i - \eta_l \frac{\partial L}{\partial y_i}$$

$$\Delta y_i = y_i - x$$

communicate Δy_i

Server

$$g = \frac{1}{|S|} \sum_{i \in S} \Delta y_i$$

$$m = \beta_1 m + (1 - \beta_1) g$$

$$v = \beta_2 v + (1 - \beta_2) g^2$$

$$\hat{m} = \frac{m}{1 - \beta_1^t}$$

$$\hat{v} = \frac{v}{1 - \beta_2^t}$$

$$x = x + \eta_g \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

communicate x

FedAdam

Server

$$g = \frac{1}{|S|} \sum_{i \in S} \Delta y_i$$

$$m = \beta_1 m + (1 - \beta_1) g$$

$$v = \beta_2 v + (1 - \beta_2) g^2$$

$$\hat{m} = \frac{m}{1 - \beta_1^t}$$

$$\hat{v} = \frac{v}{1 - \beta_2^t}$$

$$x = x + \eta_g \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

```
with torch.no_grad():
    for idx in client_ids:
        for grad, diff in zip(gradients, self.clients[idx].delta_y):
            grad.data.add_(diff.data / int(self.fraction * self.num_clients))

for p,g,m,v in zip(self.x.parameters(), gradients, self.m, self.v):
    m.data = self.beta1 * m.data + (1 - self.beta1) * g.data
    v.data = self.beta2 * v.data + (1 - self.beta2) * torch.square(g.data)
    m_bias_corr = #####m#####
    v_bias_corr = #####v#####
    p.data += self.lr * m_bias_corr / (torch.sqrt(v_bias_corr) + self.epsilon)
```

FedYogi

Client i

$$y_i = x$$

$$y_i = y_i - \eta_l \frac{\partial L}{\partial y_i}$$

$$\Delta y_i = y_i - x$$

communicate Δy_i

Server

$$g = \frac{1}{|S|} \sum_{i \in S} \Delta y_i$$

$$m = \beta_1 m + (1 - \beta_1) g$$

$$v = v + (1 - \beta_2) g^2 \odot \text{sgn}(g^2 - v)$$

$$\hat{m} = \frac{m}{1 - \beta_1^t}$$

$$\hat{v} = \frac{v}{1 - \beta_2^t}$$

$$x = x + \eta_g \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

communicate x

FedYogi

Server

$$g = \frac{1}{|S|} \sum_{i \in S} \Delta y_i$$

```
with torch.no_grad():
    for idx in client_ids:
        for grad, diff in zip(gradients, self.clients[idx].delta_y):
            grad.data.add_(diff.data / int(self.fraction * self.num_clients))
```

$$m = \beta_1 m + (1 - \beta_1) g$$

$$v = v + (1 - \beta_2) g^2 \odot sgn(g^2 - v)$$

$$\hat{m} = \frac{m}{1 - \beta_1^t}$$

$$\hat{v} = \frac{v}{1 - \beta_2^t}$$

$$x = x + \eta_g \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

```
for p,g,m,v in zip(self.x.parameters(), gradients, self.m, self.v):
    m.data = self.beta1 * m.data + (1 - self.beta1) * g.data
    v.data = v.data + (1 - self.beta2) * torch.sign(torch.square(g.data) - v.data) * torch.square(g.data)
    m_bias_corr = #####m_bias_corr#####
    v_bias_corr = #####v_bias_corr#####
    p.data += self.lr * m_bias_corr / (torch.sqrt(v_bias_corr) + self.epsilon)
```

SCAFFOLD Vocab

SCAFFOLD: Stochastic Controlled Averaging for Federated Learning; a method to correct client drift.

Control Variates: Auxiliary variables used to reduce variance in local updates and correct for client drift.

Client Control Variate: A variable maintained at each client to adjust local updates based on estimated drift.

Server Control Variate: The aggregate control variable maintained by the server to guide correction across clients.

Drift Correction: The process of adjusting updates to counter the bias introduced by heterogeneous data distributions.

Variance Reduction: Techniques used to decrease the variability in gradient estimates, enhancing convergence.

Local Gradient Correction: Specific adjustments made to local gradients using control variates.

Stochastic Optimization: The broader framework that underpins methods like SCAFFOLD, dealing with randomness in gradient updates.

Update Correction: The mechanism to adjust the direction and magnitude of client updates based on control variates.

Scaffold(Stochastic Controlled Averaging for Federated Learning)

Client i

$$y_i = x, c = c$$

$$y_i = y_i - \eta_l \left(\frac{\partial L}{\partial y_i} - c_i + c \right)$$

$$c_i^+ = \begin{cases} \text{(i)} \frac{\partial L}{\partial x} & \text{or (ii)} \\ c_i - c_i - c + \frac{1}{K_{\eta_l}}(x - y_i) \end{cases}$$

$$\Delta y_i = y_i - x$$

$$\Delta c_i = c_i^+ - c_i$$

$$c_i = c_i^+$$

communicate $(\Delta y_i, \Delta c_i)$

Server

$$g = \frac{1}{|S|} \sum_{i \in S} \Delta y_i$$

$$\Delta c = \frac{1}{|S|} \sum_{i \in S} \Delta c_i$$

$$x = x + \eta_g g$$

$$c = c + \frac{|S|}{N} \Delta c$$

communicate (x, c)

Scaffold(Stochastic Controlled Averaging for Federated Learning)

Client i

$$y_i = x, c = c$$

$$y_i = y_i - \eta_l \left(\frac{\partial L}{\partial y_i} - c_i + c \right)$$

$$c_i^+ = \begin{cases} \text{(i)} \frac{\partial L}{\partial x} \text{ or (ii)} c_i - c_i - c + \frac{1}{K_{\eta_l}}(x - y_i) \end{cases}$$

$$\Delta y_i = y_i - x$$

$$\Delta c_i = c_i^+ - c_i$$

$$c_i = c_i^+$$

communicate $(\Delta y_i, \Delta c_i)$

```
def client_update(self):
    self.x.to(self.device)
    self.y = deepcopy(self.x) #Initialize local model
    self.y.to(self.device)

    for epoch in range(self.num_epochs):
        data_iter = iter(self.data)
        inputs, labels = next(data_iter)
        inputs, labels = inputs.float().to(self.device), labels.long().to(self.device)
        output = self.y(inputs)
        loss = self.criterion(output, labels)
        grads = torch.autograd.grad(loss, self.y.parameters())

        with torch.no_grad():
            for param, grad, s_c, c_c in zip(self.y.parameters(), grads, self.server_c, self.client_c):
                s_c, c_c = s_c.to(self.device), c_c.to(self.device)
                param.data = param.data - self.lr * (grad.data + (s_c.data - c_c.data))

        if self.device == "cuda": torch.cuda.empty_cache()

    with torch.no_grad():
        delta_y = [torch.zeros_like(param, device=self.device) for param in self.y.parameters()]
        delta_c = deepcopy(delta_y)
        new_client_c = deepcopy(delta_y)

        for del_y, param_y, param_x in zip(delta_y, self.y.parameters(), self.x.parameters()):
            del_y.data += param_y.data.detach() - param_x.data.detach()
        a = (ceil(len(self.data.dataset)) / self.data.batch_size)*self.num_epochs*self.lr
        for n_c, c_l, c_g, diff in zip(new_client_c, self.client_c, self.server_c, delta_y):
            n_c.data += c_l.data - c_g.data - diff.data / a

        for d_c, n_c_l, c_l in zip(delta_c, new_client_c, self.client_c):
            d_c.data.add_(n_c_l.data - c_l.data)

    self.client_c = deepcopy(new_client_c) #Update client_c with new_client_c
    self.delta_y = delta_y
    self.delta_c = delta_c
```

Scaffold(Stochastic Controlled Averaging for Federated Learning)

Server

$$g = \frac{1}{|S|} \sum_{i \in S} \Delta y_i$$

$$\Delta c = \frac{1}{|S|} \sum_{i \in S} \Delta c_i$$

$$x = x + \eta_g g$$

$$c = c + \frac{|S|}{N} \Delta c$$

communicate (x, c)

```
def server_update(self, client_ids):
    self.x.to(self.device)
    for idx in client_ids:
        with torch.no_grad():
            for param, diff in zip(self.x.parameters(), self.clients[idx].delta_y):
                param.data.add_(diff.data * self.lr / int(self.fraction * self.num_clients))
            for c_g, c_d in zip(self.server_c, self.clients[idx].delta_c):
                c_g.data.add_(c_d.data * self.fraction)
```

780 goals

Familiarize yourself with the HPC environment

Complete code for Fed Adagrad and Fed Yogi

Choose one to run

Different dataset

Compare to baseline and Report

880

Beat FedAvg MNIST score: 84.07%
with any algorithm/model combo

Hyperparameter tuning and report
parameter impact

Apply iid and non-iid compare
algorithms