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]

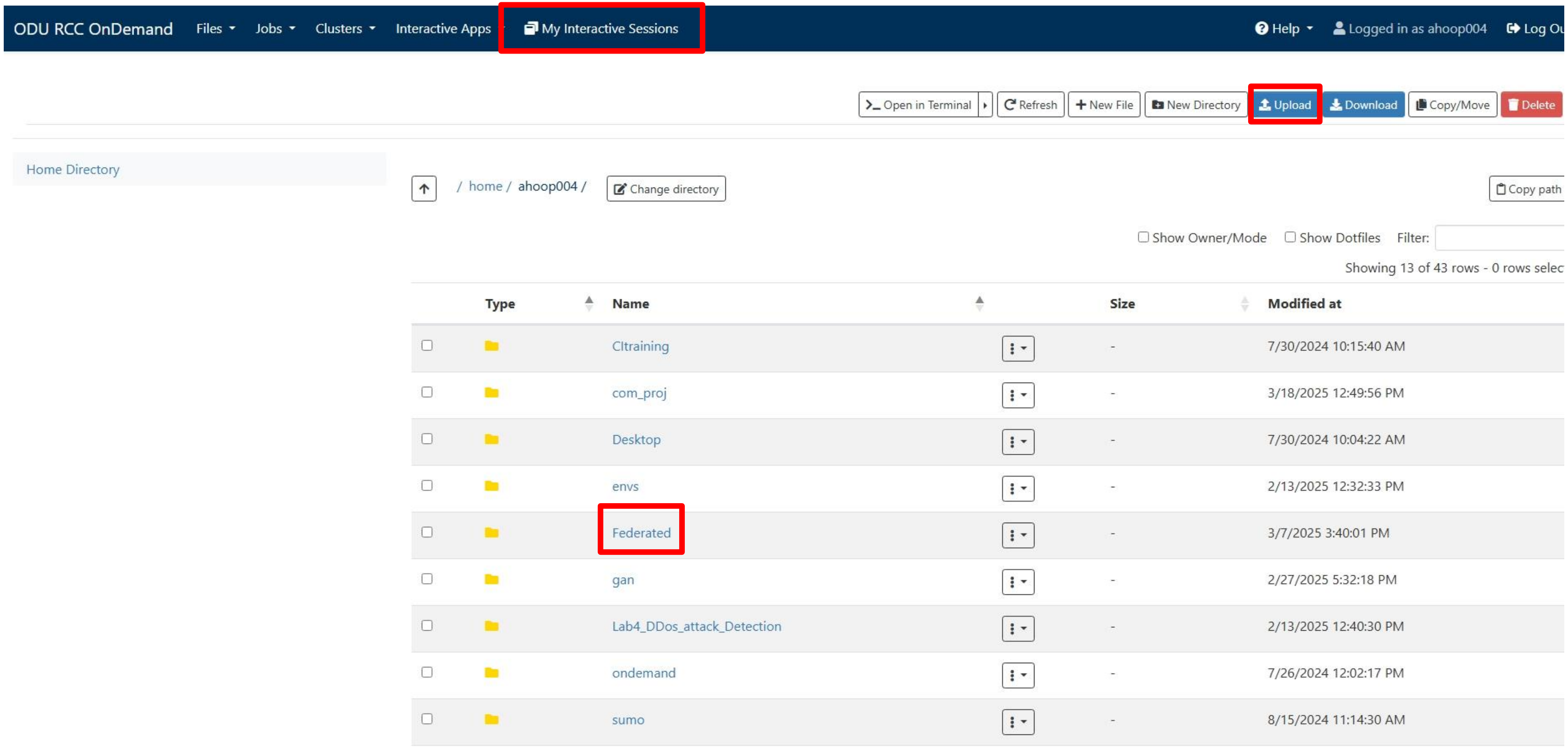
Log in to On Demand

[ODU On Demand HPC]

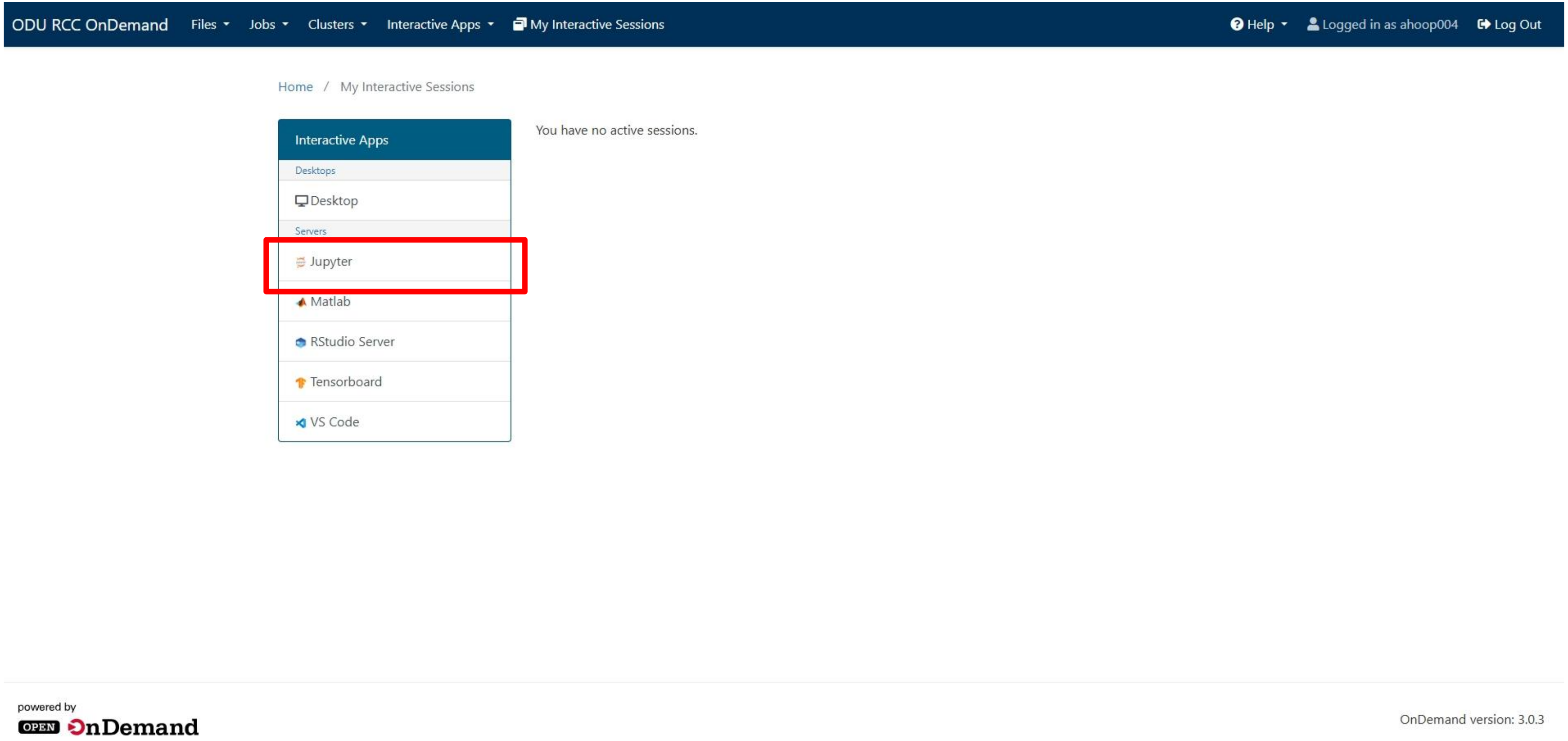
ODU HPC Documentation

[Docs]

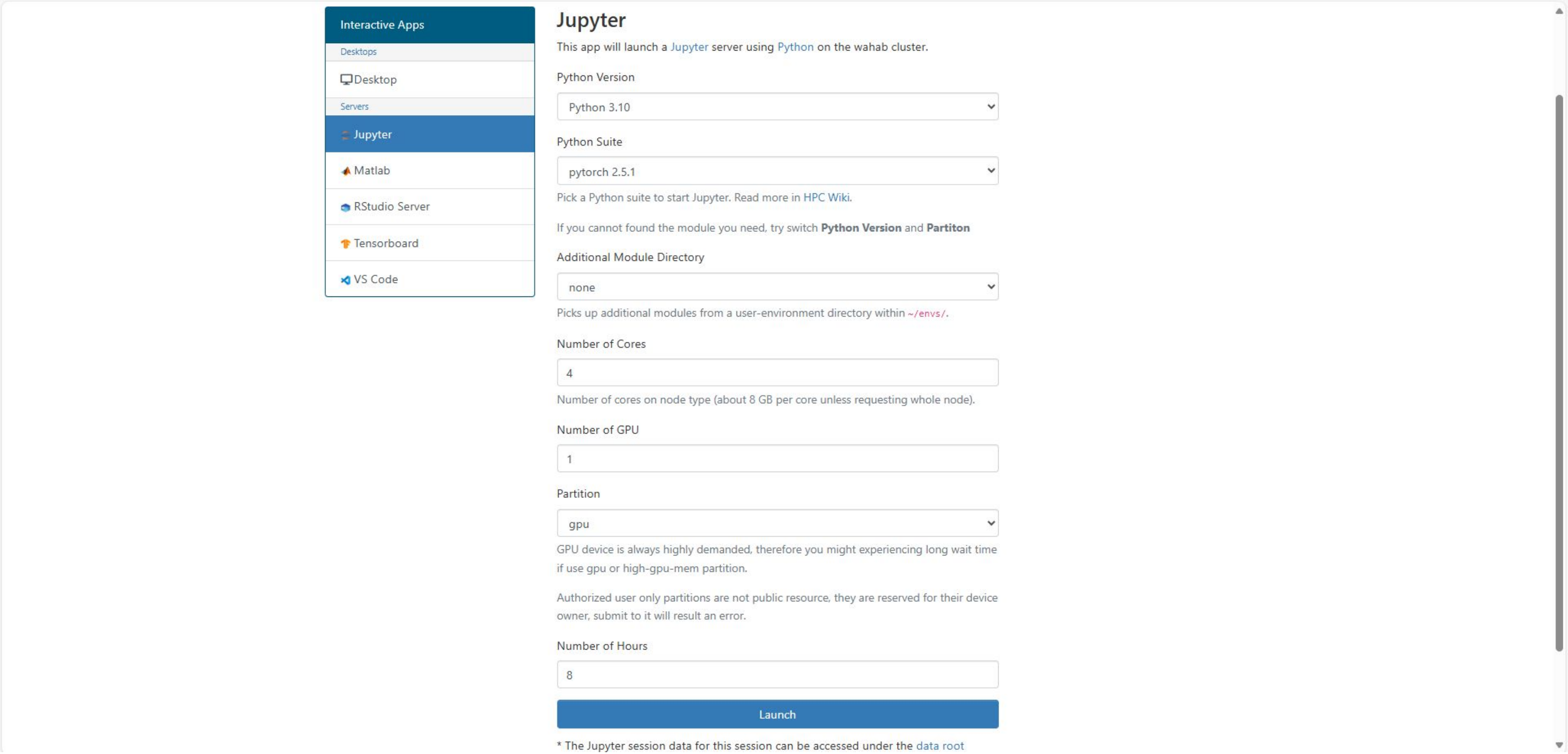
HPC: home/user



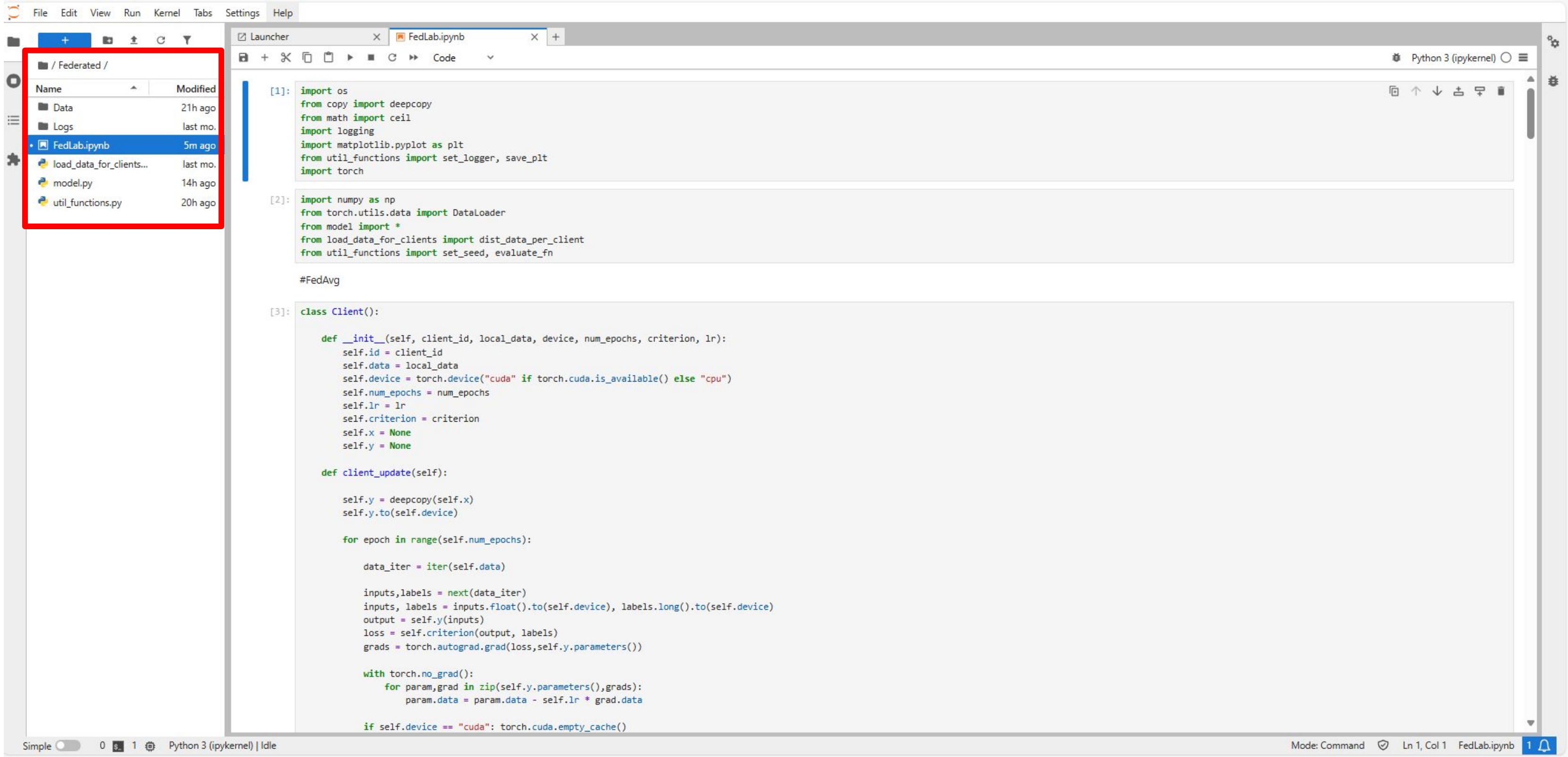
HPC: Jupyter


ODU RCC OnDemand
Files
Jobs
Clusters
Interactive Apps
My Interactive Sessions
Help
Logged in as ahoop004
Log Out
Home / My Interactive Sessions
Interactive Apps
Desktops
Desktop
Servers
Jupyter
Matlab
RStudio Server
Tensorboard
VS Code
You have no active sessions.
powered by
OPEN OnDemand
OnDemand version: 3.0.3

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 found the module you need, try switch Python Version and Partiton
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 experiencing 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



Model: Alexnet

```python
class AlexNet(nn.Module):
    def __init__(self, num_classes=10):
        super(AlexNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.layer2 = nn.Sequential(
            # Use a smaller kernel to preserve dimensions
            nn.Conv2d(64, 192, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(192),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(192, 384, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(384),
            nn.ReLU(inplace=True)
        )
        self.layer4 = nn.Sequential(
            nn.Conv2d(384, 384, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(384),
            nn.ReLU(inplace=True)
        )
        self.layer5 = nn.Sequential(
            nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.fc = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(256 * 3 * 3, 1024),
            nn.ReLU(inplace=True)
        )
        self.fc1 = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(1024, 512),
            nn.ReLU(inplace=True)
        )
        self.fc2 = nn.Sequential(
            nn.Linear(512, num_classes)
        )

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.layer5(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc(out)
        out = self.fc1(out)
        out = self.fc2(out)
        return out
```

Utility functions: Data

```python
        # set transformation differently per dataset
        if (dataset_name == "CIFAR10"):
            T = transforms.Compose([
                        # transforms.Resize(244),
                        transforms.ToTensor(),
                        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
                    ])
        elif (dataset_name == "MNIST"):
            T = transforms.Compose([
                transforms.ToTensor(),
                transforms.Normalize((0.1307,), (0.3081,))
            ])


        train_data = datasets.__dict__[dataset_name](root=data_path, train=True, download=True,
                                                     transform=T)
        test_data = datasets.__dict__[dataset_name](root=data_path, train=False, download=True,
                                                   transform=T)
    else:
        # dataset not found exception
        error_message = f"...dataset \"{dataset_name}\" is not supported or cannot be found in TorchVision Datasets!"
        raise AttributeError(error_message)

    # unsqueeze channel dimension for grayscale image datasets
    if train_data.data.ndim == 3: # convert to NxHxW -> NxHxWx1
        train_data.data.unsqueeze_(3)

    return train_data,test_data

# This class is used to get data in batches
class load_data(Dataset):
    def __init__(self, x, y):
        self.length = x.shape[0]
        self.x = x.permute(0,3,1,2)

        # self.image_transform = transforms.Normalize((127.5, 127.5, 127.5),(127.5, 127.5, 127.5))
        self.image_transform = transforms.Normalize((0.1307,), (0.3081,))

    def __getitem__(self, index):
        image,label = self.x[index],self.y[index]
        image = self.image_transform(image)
        return image,label

    def __len__(self):
        return self.length

# A simple utility function for converting pytorch tensors to numpy
def tensor_to_numpy(data, device):
    if device.type == "cpu":
        return data.detach().numpy()
    else:
        return data.cpu().detach().numpy()

# A simple utility function for converting numpy to pytorch tensors
def numpy_to_tensor(data, device, dtype="float"):
    if dtype=="float":
        return torch.tensor(data, dtype=torch.float).to(device)
```

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
```

$Server$

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

```
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

Client i

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

```
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

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

```
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

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

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

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

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

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

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

communicate x

FedAdam

$Server$

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

```
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))
```

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

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

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

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

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

```
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 = ################################################################
                v_bias_corr = ################################################################
                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

$Server$

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

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

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

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

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

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

communicate x

FedYogi

$Server$

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

```python
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))
```

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

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

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

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

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

```python
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 = ################################################################
                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)

$\text{Client } i$

$y_i = x,\ c = c$

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

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

$\Delta y_i = y_i - x$

$\Delta c_i = c_i^+ - c_i$

$c_i = c_i^+$

$\text{communicate } (\Delta y_i, \Delta c_i)$

$Server$

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

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

$x = x + \eta_g g$

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

$\text{communicate } (x, c)$

Scaffold(Stochastic Controlled Averaging for Federated Learning)

Client i

$$y_i = x,\ c = c$$

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

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

$$\Delta y_i = y_i - x$$

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

$$c_i = c_i^+$$

$$\text{communicate } (\Delta y_i, \Delta c_i)$$

```python
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)

$Server$

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

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

$$x = x + \eta_g g$$

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

$$\text{communicate }(x, c)$$

```
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