fed_avg

October 30, 2025

1 Federated Averaging (FedAvg) baseline on CIFAR-100

- Dirichlet non-IID partitioning
- Partial client participation
- Optional heterogeneity (per-client batch size / epochs / lr) ## Imports and Config values

[1]:
```
import copy
import math
import random
import numpy as np
from collections import defaultdict, OrderedDict
from typing import Dict, List, Tuple, Optional
from dataclasses import dataclass, asdict

import torch
from torch import nn, optim
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms, models
```

[2]:
```
# CONFIGURATION
CONFIG = {
    # Federated Learning
    "num_rounds": 10,
    "num_clients": 20,
    "clients_per_round": 5,  # Partial participation
    "local_epochs": 1,
    "local_batch_size": 32,

    # Data
    "dataset": "CIFAR100",
    "data_root": "./data",
    "alpha": 0.5,  # Dirichlet concentration (lower = more non-IID)
    "num_classes": 100,  # 100 for CIFAR-100, 10 for CIFAR-10 etc..

    # Model & Training
    "model_arch": "resnet18",
    "optimizer": "SGD",
    "learning_rate": 0.01,
```

```python
    "momentum": 0.9,
    "weight_decay": 5e-4,

    # Capture settings
    "max_steps_to_store": None,  # None = store all, or set limit (e.g., 50)
    "return_indices": False,

    # Misc
    "device": "cuda" if torch.cuda.is_available() else "mps",
    "seed": 42,
    "save_prefix": "fedavg_metrics",
}
```

[3]:
```python
# Reproducibility
random.seed(CONFIG["seed"])
np.random.seed(CONFIG["seed"])
torch.manual_seed(CONFIG["seed"])
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(CONFIG["seed"])
```

[4]:
```python
# Helper Functions
def to_cpu_f32(t):
    return t.detach().to("cpu", non_blocking=True).float().clone()

def state_to_cpu_f32(sd: dict):
    return {k: to_cpu_f32(v) for k, v in sd.items()}

def param_order_and_shapes(model: torch.nn.Module):
    return [{"name": n, "shape": list(p.shape), "numel": p.numel()}
            for n, p in model.named_parameters()]

@dataclass
class OptimCfg:
    name: str = "SGD"
    lr: float = 0.01
    momentum: float = 0.9
    weight_decay: float = 5e-4
    nesterov: bool = False

def build_optimizer(model, cfg: OptimCfg):
    if cfg.name.lower() == "sgd":
        return optim.SGD(model.parameters(), lr=cfg.lr, momentum=cfg.momentum,
                         weight_decay=cfg.weight_decay, nesterov=cfg.nesterov)
    elif cfg.name.lower() == "adam":
        return optim.Adam(model.parameters(), lr=cfg.lr, weight_decay=cfg.
 ↪weight_decay)
    else:
```

```
        raise ValueError(f"Unsupported optimizer: {cfg.name}")

def class_histogram_from_loader(loader, num_classes: int):
    counts = torch.zeros(num_classes, dtype=torch.long)
    for batch in loader:
        y = batch[1]
        counts.index_add_(0, y.to(dtype=torch.long), torch.ones_like(y,␣
↪dtype=torch.long))
    return {int(i): int(v) for i, v in enumerate(counts)}
```

1.0.1 Data: CIFAR-100 loaders (train/test)

[5]:
```
def load_cifar100(data_root: str = "./data"):
    mean = (0.5071, 0.4867, 0.4408)
    std = (0.2675, 0.2565, 0.2761)

    train_tf = transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean, std),
    ])
    test_tf = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean, std),
    ])

    train = datasets.CIFAR100(root=data_root, train=True, download=True,␣
↪transform=train_tf)
    test = datasets.CIFAR100(root=data_root, train=False, download=True,␣
↪transform=test_tf)
    return train, test

def _get_targets(dataset) -> np.ndarray:
    targets = getattr(dataset, "targets", None)
    if targets is None:
        targets = getattr(dataset, "labels", None)
    if targets is None:
        raise AttributeError("Dataset has no 'targets' or 'labels'.")
    return np.array(targets)
```

Dirichlet non-IID split (returns dict: client_id -> list of indices)

[6]:
```
def dirichlet_noniid_indices(dataset, num_clients: int, alpha: float,
                             min_per_client: int = 10) -> Dict[int, List[int]]:
    y = _get_targets(dataset)
    num_classes = int(y.max()) + 1
```

```
    idx_by_class = {c: np.where(y == c)[0] for c in range(num_classes)}
    for c in idx_by_class:
        np.random.shuffle(idx_by_class[c])

    client_indices = [[] for _ in range(num_clients)]
    for c in range(num_classes):
        idx_c = idx_by_class[c]
        if len(idx_c) == 0:
            continue
        p = np.random.dirichlet([alpha] * num_clients)
        cuts = (np.cumsum(p) * len(idx_c)).astype(int)[:-1]
        split = np.split(idx_c, cuts)
        for i, shard in enumerate(split):
            client_indices[i].extend(shard.tolist())

    pool = list(range(len(dataset)))
    for i in range(num_clients):
        if len(client_indices[i]) < min_per_client:
            need = min_per_client - len(client_indices[i])
            extra = np.random.choice(pool, size=need, replace=False).tolist()
            client_indices[i].extend(extra)

    for i in range(num_clients):
        random.shuffle(client_indices[i])
    return {i: client_indices[i] for i in range(num_clients)}
```

1.0.2 Model: ResNet18 head for CIFAR-100

[7]:
```
def build_model(num_classes: int = 100) -> nn.Module:
    model = models.resnet18(weights=None)  # no pretrained to avoid download in␣
↪restricted envs
    # CIFAR images are 3x32x32; torchvision ResNet expects 224x224,
    # but it's fine-ResNet is fully conv except FC. It still works on 32x32.
    # Replace final FC layer to match number of classes
    in_feats = model.fc.in_features
    model.fc = nn.Linear(in_feats, num_classes)
    return model
```

[8]:
```
def evaluate(model: nn.Module, loader: DataLoader, device: torch.device) ->␣
↪Tuple[float, float]:
    model.eval()
    correct = 0
    total = 0
    loss_sum = 0.0
    criterion = nn.CrossEntropyLoss()
    with torch.no_grad():
        for x, y in loader:
```

```
            x, y = x.to(device), y.to(device)
            logits = model(x)
            loss = criterion(logits, y)
            loss_sum += loss.item() * x.size(0)
            preds = logits.argmax(dim=1)
            correct += (preds == y).sum().item()
            total += x.size(0)
    return loss_sum / max(1, total), correct / max(1, total)
```

1.0.3 Local Training with Gradient Capture

[9]:
```
@torch.no_grad()
def clone_state(model):
    return {k: v.detach().clone() for k, v in model.state_dict().items()}

def train_one_client_with_capture(
    global_model: nn.Module,
    client_loader: DataLoader,
    loss_fn: nn.Module,
    opt_cfg: OptimCfg,
    epochs: int = 1,
    device: torch.device = torch.device("cpu"),
    max_steps_to_store: int = None,
    return_indices: bool = False,
    num_classes: int = None,
    client_seed: int = None
):
    model = copy.deepcopy(global_model).to(device)
    model.train()

    if client_seed is not None:
        torch.manual_seed(client_seed)
        random.seed(client_seed)
        np.random.seed(client_seed)

    global_before = clone_state(model)
    opt = build_optimizer(model, opt_cfg)

    grads_per_step_raw = []
    grads_per_step_wd = []
    batch_sizes = []
    step_losses = []
    step_batch_indices = []

    steps_stored = 0
    for _ in range(epochs):
        for batch in client_loader:
```

```python
            if return_indices and len(batch) == 3:
                x, y, idxs = batch
            else:
                x, y = batch[0], batch[1]
                idxs = None

            x, y = x.to(device, non_blocking=True), y.to(device,␣
↪non_blocking=True)

            opt.zero_grad(set_to_none=True)
            logits = model(x)
            loss = loss_fn(logits, y)
            loss.backward()

            # Capture gradients before optimizer step
            if max_steps_to_store is None or steps_stored < max_steps_to_store:
                raw_dict = {}
                wd_dict = {}
                for name, p in model.named_parameters():
                    if p.grad is None:
                        continue
                    g = p.grad
                    raw_dict[name] = to_cpu_f32(g)
                    if opt_cfg.weight_decay and opt_cfg.weight_decay > 0:
                        wd_dict[name] = to_cpu_f32(g + opt_cfg.weight_decay * p.
↪data)
                    else:
                        wd_dict[name] = to_cpu_f32(g)

                grads_per_step_raw.append(raw_dict)
                grads_per_step_wd.append(wd_dict)
                batch_sizes.append(int(x.shape[0]))
                step_losses.append(float(loss.detach().item()))
                if return_indices and idxs is not None:
                    step_batch_indices.append([int(i) for i in idxs])

                steps_stored += 1

            opt.step()

    local_after = clone_state(model)

    # Compute delta
    delta = OrderedDict()
    for k in local_after.keys():
        delta[k] = to_cpu_f32(local_after[k]) - to_cpu_f32(global_before[k])
```

```python
    # Diagnostics
    if len(grads_per_step_raw) > 0:
        first_step = grads_per_step_raw[0]
        per_layer_norms = {k: float(v.view(-1).norm().item()) for k, v in␣
↪first_step.items()}
        grad_norm_total = float(torch.sqrt(sum(v.pow(2).sum() for v in␣
↪first_step.values())).item())
    else:
        per_layer_norms, grad_norm_total = {}, 0.0

    class_dist = None
    if num_classes is not None:
        class_dist = class_histogram_from_loader(client_loader,␣
↪num_classes=num_classes)

    telemetry = {
        "per_layer_norms": per_layer_norms,
        "gradient_norm": grad_norm_total,
        "loss_history": step_losses,
        "batch_sizes": batch_sizes,
        "num_steps_captured": len(grads_per_step_raw),
        "num_samples": sum(batch_sizes),
        "class_distribution": class_dist,
    }
    if return_indices and len(step_batch_indices) > 0:
        telemetry["batch_indices"] = step_batch_indices

    return {
        "local_state_after": state_to_cpu_f32(local_after),
        "delta": delta,
        "grads_per_step_raw": grads_per_step_raw,
        "grads_per_step_wd": grads_per_step_wd,
        "telemetry": telemetry,
    }
```

1.0.4 FedAvg Aggregation

[10]:
```python
def average_weights(weight_list, sizes):
    if not weight_list:
        raise ValueError("No client weights provided.")
    if len(weight_list) != len(sizes):
        raise ValueError("weights and sizes mismatch")

    total = float(sum(sizes))
    avg = {k: torch.zeros_like(v) for k, v in weight_list[0].items()}

    for wi, si in zip(weight_list, sizes):
```

```
        w = si / total
        for k in avg.keys():
            if avg[k].dtype.is_floating_point:
                avg[k] += wi[k].float() * w
            else:
                avg[k] = wi[k].clone()
    return avg
```

1.1 Federated Round with Capture

[11]:
```
def run_fed_round_with_capture(
    round_num: int,
    global_model: nn.Module,
    clients: dict,
    loss_fn: nn.Module,
    opt_cfg: OptimCfg,
    local_epochs: int,
    device: torch.device,
    num_classes: int = None,
    max_steps_to_store: int = None,
    return_indices: bool = False,
    server_seed: int = None,
    client_seeds: dict = None,
    training_meta: dict = None,
    global_eval_fn = None
):
    if server_seed is not None:
        torch.manual_seed(server_seed)
        random.seed(server_seed)
        np.random.seed(server_seed)

    participating_clients = list(clients.keys())
    global_state_cpu = state_to_cpu_f32(global_model.state_dict())

    client_metrics = {}
    raw_gradients = {}
    model_updates = {}

    for cid, loader in clients.items():
        cseed = (client_seeds or {}).get(cid)
        result = train_one_client_with_capture(
            global_model=global_model,
            client_loader=loader,
            loss_fn=loss_fn,
            opt_cfg=opt_cfg,
            epochs=local_epochs,
            device=device,
```

```
            max_steps_to_store=max_steps_to_store,
            return_indices=return_indices,
            num_classes=num_classes,
            client_seed=cseed
        )

        model_updates[cid] = result["delta"]
        raw_gradients[cid] = {
            "grads_per_step_raw": result["grads_per_step_raw"],
            "grads_per_step_wd": result["grads_per_step_wd"],
        }

        tele = result["telemetry"]
        client_metrics[cid] = {
            "gradient_norm": tele["gradient_norm"],
            "per_layer_norms": tele["per_layer_norms"],
            "local_epochs": local_epochs,
            "learning_rate": opt_cfg.lr,
            "num_samples": tele["num_samples"],
            "class_distribution": tele["class_distribution"],
            "local_loss": float(tele["loss_history"][-1]) if␣
↪tele["loss_history"] else None,
            "loss_history": tele["loss_history"],
            "batch_sizes": tele["batch_sizes"],
        }
        if return_indices and ("batch_indices" in tele):
            client_metrics[cid]["batch_indices"] = tele["batch_indices"]

    # Server aggregate delta
    agg_delta = {}
    if len(model_updates) > 0:
        keys = next(iter(model_updates.values())).keys()
        sizes = [client_metrics[cid]["num_samples"] for cid in␣
↪participating_clients]
        for k in keys:
            stacked = torch.stack([model_updates[cid][k] for cid in␣
↪participating_clients])
            weights = torch.tensor([s / sum(sizes) for s in sizes])
            agg_delta[k] = (stacked.T @ weights).T

    # Global evaluation
    global_accuracy = None
    global_loss = None
    if callable(global_eval_fn):
        global_loss, global_accuracy = global_eval_fn(global_model)

    config_snapshot = {
```

```python
        "arch": type(global_model).__name__,
        "optimizer": asdict(opt_cfg),
        "loss": type(loss_fn).__name__,
        "num_classes": num_classes,
        "param_meta": param_order_and_shapes(global_model),
        "seeds": {"server_seed": server_seed, "client_seeds": client_seeds},
        "device": str(device),
    }
    if training_meta:
        config_snapshot.update({"training_meta": training_meta})

    return {
        "round": int(round_num),
        "participating_clients": participating_clients,
        "client_metrics": client_metrics,
        "global_model_state": global_state_cpu,
        "global_accuracy": global_accuracy,
        "global_loss": global_loss,
        "raw_gradients": raw_gradients,
        "model_updates": model_updates,
        "server_aggregate_delta": agg_delta,
        "config_snapshot": config_snapshot,
    }
```

1.2 Save Exports

[12]:
```python
import json
import pickle

def save_round_export(metrics_to_export, prefix: str = "fed_round"):
    r = metrics_to_export["round"]
    tensor_blob = {
        "global_model_state": metrics_to_export["global_model_state"],
        "raw_gradients": metrics_to_export["raw_gradients"],
        "model_updates": metrics_to_export["model_updates"],
        "server_aggregate_delta": metrics_to_export.
 ↪get("server_aggregate_delta"),
    }
    meta_blob = {k: v for k, v in metrics_to_export.items()
                 if k not in tensor_blob.keys()}

    torch.save(tensor_blob, f"{prefix}_{r:02d}_tensors.pt")

    try:
        with open(f"{prefix}_{r:02d}_meta.json", "w") as f:
            json.dump(meta_blob, f, indent=2)
    except TypeError:
```

```
        with open(f"{prefix}_{r:02d}_meta.pkl", "wb") as f:
            pickle.dump(meta_blob, f)
```

1.3 Final / main executionn

[13]:
```
if __name__ == "__main__":
    print("FedAvg Training with Gradient Capture : ")
    print(f"Device: {CONFIG['device']}")
    print(f"Rounds: {CONFIG['num_rounds']}")
    print(f"Clients: {CONFIG['num_clients']} (sampling␣
 ↪{CONFIG['clients_per_round']}/round)")
    print(f"Local epochs: {CONFIG['local_epochs']}")
    print(f"Alpha (non-IID): {CONFIG['alpha']}")
    print("_" * 100)

    # Load data
    print("\n[1/5] Loading data...")
    train_dataset, test_dataset = load_cifar100(CONFIG["data_root"])
    test_loader = DataLoader(test_dataset, batch_size=256, shuffle=False,␣
 ↪num_workers=2)

    # Partition data
    print("[2/5] Partitioning data (Dirichlet non-IID)...")
    client_indices = dirichlet_noniid_indices(
        train_dataset,
        CONFIG["num_clients"],
        CONFIG["alpha"]
    )

    client_loaders = {}
    for cid, indices in client_indices.items():
        subset = Subset(train_dataset, indices)
        client_loaders[cid] = DataLoader(
            subset,
            batch_size=CONFIG["local_batch_size"],
            shuffle=True,
            num_workers=2
        )

    print(f"   Client data sizes: {[len(idx) for idx in client_indices.
 ↪values()]}")

    # Initialize model
    print("[3/5] Building model...")
    device = torch.device(CONFIG["device"])
    global_model = build_model(CONFIG["num_classes"]).to(device)
    loss_fn = nn.CrossEntropyLoss()
```

```
    opt_cfg = OptimCfg(
        name=CONFIG["optimizer"],
        lr=CONFIG["learning_rate"],
        momentum=CONFIG["momentum"],
        weight_decay=CONFIG["weight_decay"]
    )

    # Training loop
    print("[4/5] Starting federated training...")
    for round_num in range(CONFIG["num_rounds"]):
        print(f"\n--- Round {round_num + 1}/{CONFIG['num_rounds']} ---")

        # Sample clients
        participating = random.sample(
            list(client_loaders.keys()),
            CONFIG["clients_per_round"]
        )
        selected_loaders = {cid: client_loaders[cid] for cid in participating}

        # Run round with capture
        metrics = run_fed_round_with_capture(
            round_num=round_num,
            global_model=global_model,
            clients=selected_loaders,
            loss_fn=loss_fn,
            opt_cfg=opt_cfg,
            local_epochs=CONFIG["local_epochs"],
            device=device,
            num_classes=CONFIG["num_classes"],
            max_steps_to_store=CONFIG["max_steps_to_store"],
            return_indices=CONFIG["return_indices"],
            server_seed=CONFIG["seed"] + round_num,
            client_seeds={cid: CONFIG["seed"] + round_num + int(cid) for cid in␣
↪participating},
            training_meta={"dataset": CONFIG["dataset"], "alpha":␣
↪CONFIG["alpha"]},
            global_eval_fn=lambda m: evaluate(m, test_loader, device)
        )

        # Aggregate and update global model
        client_states = [metrics["model_updates"][cid] for cid in participating]
        client_sizes = [metrics["client_metrics"][cid]["num_samples"] for cid␣
↪in participating]

        # Apply aggregated update to global model
        aggregated_state = average_weights(
```

```python
            [global_model.state_dict() for _ in participating],  # Start from global
            client_sizes
        )

        # Actually update global model with deltas
        current_state = global_model.state_dict()
        new_state = {}
        for k in current_state.keys():
            if k in metrics["server_aggregate_delta"]:
                new_state[k] = current_state[k] + metrics["server_aggregate_delta"][k].to(device)
            else:
                new_state[k] = current_state[k]
        global_model.load_state_dict(new_state)

        # Print metrics
        print(f"   Clients: {participating}")
        print(f"   Global Loss: {metrics['global_loss']:.4f}")
        print(f"   Global Acc: {metrics['global_accuracy']:.4f}")

        # Save metrics
        save_round_export(metrics, prefix=CONFIG["save_prefix"])
        print(f"   Saved: {CONFIG['save_prefix']}_{round_num:02d}_*.pt/json")

    # Final evaluation
    print("\n[5/5] Final evaluation...")
    final_loss, final_acc = evaluate(global_model, test_loader, device)
    print(f"   Final Test Loss: {final_loss:.4f}")
    print(f"   Final Test Accuracy: {final_acc:.4f}")

    print("\n" + "-" * 100)
    print("Training complete!")
```

```
FedAvg Training with Gradient Capture :
Device: cuda
Rounds: 10
Clients: 20 (sampling 5/round)
Local epochs: 1
Alpha (non-IID): 0.5
--------------------------------------------------------------------------------
--------------------

[1/5] Loading data…
[2/5] Partitioning data (Dirichlet non-IID)…
   Client data sizes: [2309, 2647, 2243, 2492, 2185, 3039, 2303, 2767, 3159,
2351, 2476, 2130, 2687, 2019, 2317, 2234, 2780, 1822, 3040, 3000]
```

```
[3/5] Building model…
[4/5] Starting federated training…

--- Round 1/10 ---
/run/nvme/job_30378818/tmp/ipykernel_892135/644794527.py:73: UserWarning: The
use of `x.T` on tensors of dimension other than 2 to reverse their shape is
deprecated and it will throw an error in a future release. Consider `x.mT` to
transpose batches of matrices or `x.permute(*torch.arange(x.ndim - 1, -1, -1))`
to reverse the dimensions of a tensor. (Triggered internally at
/pytorch/aten/src/ATen/native/TensorShape.cpp:4413.)
  agg_delta[k] = (stacked.T @ weights).T
/run/nvme/job_30378818/tmp/ipykernel_892135/644794527.py:73: UserWarning:
Tensor.T is deprecated on 0-D tensors. This function is the identity in these
cases. (Triggered internally at
/pytorch/aten/src/ATen/native/TensorShape.cpp:4420.)
  agg_delta[k] = (stacked.T @ weights).T
   Clients: [9, 7, 2, 0, 6]
   Global Loss: 4.6169
   Global Acc: 0.0095
   Saved: fedavg_metrics_00_*.pt/json

--- Round 2/10 ---
   Clients: [17, 10, 4, 9, 6]
   Global Loss: 4.7198
   Global Acc: 0.0100
   Saved: fedavg_metrics_01_*.pt/json

--- Round 3/10 ---
   Clients: [2, 11, 13, 3, 10]
   Global Loss: 4.7732
   Global Acc: 0.0100
   Saved: fedavg_metrics_02_*.pt/json

--- Round 4/10 ---
   Clients: [4, 14, 17, 9, 15]
   Global Loss: nan
   Global Acc: 0.0100
   Saved: fedavg_metrics_03_*.pt/json

--- Round 5/10 ---
   Clients: [9, 19, 4, 8, 7]
   Global Loss: nan
   Global Acc: 0.0100
   Saved: fedavg_metrics_04_*.pt/json

--- Round 6/10 ---
   Clients: [19, 6, 14, 16, 15]
```

```
   Global Loss: 6.2686
   Global Acc: 0.0100
   Saved: fedavg_metrics_05_*.pt/json

--- Round 7/10 ---
   Clients: [18, 5, 2, 7, 14]
   Global Loss: 49.5099
   Global Acc: 0.0098
   Saved: fedavg_metrics_06_*.pt/json

--- Round 8/10 ---
   Clients: [18, 5, 2, 7, 14]
   Global Loss: 98156.3394
   Global Acc: 0.0100
   Saved: fedavg_metrics_07_*.pt/json

--- Round 9/10 ---
   Clients: [14, 19, 9, 8, 15]
   Global Loss: 196964833028488768.0000
   Global Acc: 0.0100
   Saved: fedavg_metrics_08_*.pt/json

--- Round 10/10 ---
   Clients: [13, 9, 18, 16, 7]
   Global Loss: nan
   Global Acc: 0.0100
   Saved: fedavg_metrics_09_*.pt/json

[5/5] Final evaluation…
   Final Test Loss: nan
   Final Test Accuracy: 0.0100

--------------------------------------------------------------------------------
--------------------
Training complete!
```