Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

FLStudio
Uno studio di
Federated Learning

Relatore:
Prof.
Federico Montori

Presentata da:
Daniele Tarek Iaisy

Correlatore:
Prof.
Alfonso Esposito

II Sessione
Anno Accademico 2023/2024

Abstract

Il federated learning è emerso come un promettente modello distribuito che permette ad un ampio numero di client di partecipare collettivamente al training di un modello di machine learning, senza compromettere la privacy dei dati usati per il training, mantenendoli al sicuro alla fonte. Questo è un modello diametralmente opposto a quello del machine learning tradizionale in cui viene curato un unico dataset globale in cui vengono raccolti tutti i dati. Questa tesi si pone di studiare empiricamente come diversi gradi di ibridazione di approccio da quello completamente federato a quello completamente centralizzato influiscono sulle performance del modello allenato. Lo studio è stato condotto su due dataset di benchmark, il FEMNIST e l'UCI Human Activity Recognition (HAR) e permettendo di analizzare sia una CNN (Convolutional Neural Network) e una MLP (Multi Layer Perceptron) entrambe di dimensioni contenute. Sono state provate diverse combinazioni di percentuali di condivisione dei dataset (da 0%, setting completamente federato al 100%, setting completamente centralizzato), di strategie di condivisione dei dati (scegliendo alcuni client di cui rendere i dataset condivisi o prendendo alcuni elementi dai dataset di tutti i client) e di diversi algoritmi di ottimizzazione. I risultati sperimentali verificano che alte performance sono raggiungibili anche in contesti completamente federati, anche se al costo di un maggiore numero di cicli di training.

Contents

Chapter 1

Introduzione

La rapida proliferazione di dispositivi connessi sulla rete, insieme ad una maggiore sensibilità diffusa nei confronti della privacy hanno fatto sì che si rendessero necessari nuovi modelli di machine learning che tenessero in considerazione i requisiti di contesti privacy oriented. Il Federated learning (FL) è nuovo approccio che risolve questo problema permettendo a diversi client (ad esempio telefoni, dispositivi IoT o server edge) distribuiti per il mondo di allenare collaborativamente uno stesso modello di machine learning. La privacy dei dati è preservata mantenendo i dati all'origine, sui client stessi che li producono, e portando invece il modello da allenare ad essi.

Nonostante le sue potenzialità, il federated learning ha comunque delle difficoltà, prima fra tutte il fatto che tipicamente i dati locali dei client sono estremamente eterogenei tra loro, peggiorando le performance del modello allenato. Una delle assunzioni tipiche per le tecniche dei machine learning più diffuse è che il dataset utilizzato per l'apprendimento sia costituito da sample IID (Indipendenti e Identicamente Distribuiti), fatto che tipicamente non si verifica in contesti federati.

Una delle possibili strategie per migliorare le capacità di un modello allenato in modo federato è quella di costruire un dataset globale, composto da un sottoinsieme dei dataset dei client che viene condiviso e reso pubblico, il federated learning ibrido. Questa tesi studia la variazione di performance di un modello sotto diversi gradi di ibridazione. Vengono presi in considerazione due dataset diversi nel dominio dell'image classification e del human activity recognition -FEMNIST e UCI HAR- per vedere come si comporta il federated learning sotto diverse modalità. Il FEMNIST (Federated Extended MNIST) è un'estensione del celebre MNIST che include immagini sia di cifre che di numeri e i cui caratteri sono partizionati in base allo scrittore originale; l'UCI HAR invece contiene misurazioni fatte da accellerometro e giroscopio di uno smartphone tenuto in vita da 30 persone diverse mentre compiono azioni diverse. Per studiare al meglio gli effetti di diverse strate-

gie di condivisione dei dati, si è allenata la stessa rete neurale con diversi gradi di condisione dei dati (da 0%, federato, a 100%, centralizzato) e diversi metodi di condivisione dei dati. Per il FEMNIST è stata usata una CNN e per UCI HAR un MLP. Entrambi i modelli sono stati allenati sia con SGD che con AdamW, due algoritmi di ottimizzazione tipici nel Deep Learning.

Questo elaborato è strutturato nel modo seguente: il capitolo 2 introduce il federated learning, diverse categorizzazioni e strategie, seguiti da una discussione di vantaggi e limitazioni; il capitolo 3 discute nel dettaglio gli esperimenti, descrivendo i dataset e i modelli usati e spiegando le scelte fatte; il capitolo 4 illustra l'implementazione del progetto in codice python, facendo uso di Flower [1], un framework per fare simulazioni o applicazioni di federated learning che supporta sia PyTorch che Tensorflow e vari ambienti di sviluppo oltre python tra cui Android, iOS e C++; infine il capitolo 5 mostra i risultati degli esperimenti e discute anche possibili metodi per migliorare ulteriormente le performance del modello.

Chapter 2

Stato dell'arte

In questa sezione viene introdotto il Federated Learning, come funziona, i diversi possibili approcci, quali sono i vantaggi e gli svantaggi. Prima di tutto viene spiegato cos'è il Federated Learning e i diversi tipi di partizionamento dei dati, mentre nel secondo paragrafo vengono discussi gli aspetti positivi di questa tecnica, specie nell'area della privacy dei dei dati per contesti ad alta sensibilità. Infine vengono discusse le difficoltà che quest'approccio provoca, principalmente la distribuzione non IID (Indipendent and Identically Distributed) dei dataset locali.

2.1 Federated Learning

Il Machine Learning è la classe di algoritmi di apprendimento in cui ad un modello viene fornito un insieme di dati, il training dataset, da cui imparare una certa funzione obbiettivo su questi dati. Un algoritmo di Machine Learning popolare una decina di anni fa era quello delle SVM (Support Vector Machine), mentre ad oggi il Deep Learning la fa da padrona. Tipicamente, nei problemi di Machine Learning si assume di avere un unico dataset e un unico modello che ha accesso a tutti i sample nel dataset. Tuttavia quest'approccio può porre alcuni problemi: in contesti in cui si lavora con dati sensibili, come i dati medici di privati cittadini, può essere sconsigliato, difficile o vietato (da norme come il GDPR o il più recente AI act) raccogliere tali dati da utilizzare per allenare un modello; inoltre la creazione e gestione di un tale dataset può avere costi non indifferenti dati dal trasporto dei dati in unico data storage centralizzato, la memorizzazione di tali dati e il costo dell'effettivo allenamento del modello.

Il Federated Learning si pone di risolvere questo problema allenando il nostro modello proprio alla fonte dei dati, dove questi sono già presenti o vengono generati. In questo framework possiamo individuare N client ognuno dei quali ha una copia del modello da allenare fornitagli da un server e ognuno dei quali

con il proprio dataset, disgiunto da quello degli altri client. Ad ogni round di apprendimento ognuno degli N client allena il proprio modello con il proprio dataset locale e, finito il passo di apprendimento, invia al server il proprio modello aggiornato. A questo punto il server si occuperà semplicemente di fare una qualche aggregazione dei nuovi N modelli che gli sono stati forniti e redistribuire ai client il nuovo modello prodotto.

In questo studio ci occupiamo di applicare il Federated Learning a 2 reti neurali, un semplice MLP (Multi Layer Perceptron) e una CNN (Convolutional Neural Network), ma questa tecnica può essere usata in tutti gli algoritmi di Machine Learning in cui modello usato permette un qualche meccanismo di aggregazione; le reti neurali non sono uniche in questo e le già citate SVM sono un altro di questi casi.

A seconda di come differiscono i dataset locali di ogni client si possono fare 2 classificazioni di Federated Learning: quello a partizionamento orizzontale e quello a partizionamento verticale.

2.2 Metodi di partizionamento

Uno dei punti fondamentali del federated learning è come sono distribuiti i dati tra i vari client. Si possono individuare almeno due categorie di federated learning in base a come sono distribuiti: l'Horizontal Federated Learning (HFL) e il Vertical Federated Learning (VFL). In questa sezione vengono presentati i due modelli e poi vengono anche discusso un modello ibrido in cui parte dei dati rimane alla fonte e un'altra parte viene condivisa in un unico dataset globale.

2.2.1 Partizionamento orizzontale

Nel partizionamento orizzontale, detto anche sample-based federated learning, i dataset di tutti i client hanno lo stesso schema, le stesse features, e i dataset variano per i sample che contengono. Un esempio di questo caso, con cui tutti abbiamo a che fare ogni giorno, sono i suggeritori di testo nelle tastiere dei nostri smartphone, in cui ogni telefono ha la copia di uno stesso modello di partenza e le feature di input a questo modello sono sempre le stesse, una stringa che abbiamo scritto noi, ma i dataset locali di ogni telefono sono diversi perché includono solo i messaggi scritti su quel telefono. In questo modo si ottiene la personalizzazione del suggerimento di testo.

2.2.2 Partizionamento verticale

Nel partizionamento verticale VFL, conosciuto anche come feature-based federated learning, ogni client utilizza un insieme di feature diverse da quelle degli altri, con un qualche collegamento tra i sample. Un primo studio che ha introdotto il VFL in modo distinto da quello orizzontale è yang et al. [39]. In questo caso i dati non vengono da una fonte comune istanziata più volte (come la stessa applicazione che produce gli stessi dati usata da utenti diversi), ma fonti diverse, eterogenee e tipicamente indipendenti tra loro, che seppur essendo dati diversi fanno riferimento alla stessa "entità" (come ad esempio una persona). Un esempio concreto di questa situazione può essere quello di una banca e una società di e-commerce che hanno informazioni diverse sugli stessi clienti. L'obbiettivo del VFL in questo caso è quello di allenare un modello globale che possa predire o analizzare il comportamento dei clienti, evitando di dover condividere le loro informazioni tra la banca e la società di e-commerce. Per fare ciò, ogni parte coinvolta avrà un proprio modello locale che calcola un embedding delle feature. Questi embeddings vengono poi raccolti e forniti ad un modello globale che finisce di calcolare la funzione obbiettivo.

Una delle principali difficoltà del VFL, assente nel HFL, è quella di allineare le diverse feature tra le varie organizzazioni, in modo che però non avvenga uno scambio di esse. Un altro problema è quello dello scambio degli embeddings in modo sicuro senza che da questi si riesca a risalire alle feature originali, invertendo l'operazione di embedding. Questo tipo di problemi in genere può essere risolto facendo uso di Differential Privacy (DF) e/o di homomorphic encryption; alcuni studio che discutono questi problemi e come risolverli sono Wei et al. [34] e [24]

2.2.3 Ibridazione

In contesti in cui la privacy non è un requisito fondamentale, dove il federated learning può essere applicato per distribuire i calcoli e ottenere personalizzazione sui dati piuttosto che garantire la privacy degli utenti, un'idea interessante è quella seguire un approccio ibrido tra quello del federated learning e il machine learning tradizionale. L'idea è quella di condividere solo una piccola percentuale di dati in modo da migliorare le performance del modello, pur mantenendo lo spostamento di dati al minimo. Quest'interesse è motivato da risultati empirici come quelli di Zhao et al. [40] che hanno visto come condividendo solo il 5% dei dati si può ottenere un miglioramento del 30% sulla precisione del modello.

Se ci si pone in un contesto di federated learning ibrido, anche qua, possiamo avere diversi modelli di condivisione dei dati. Un approccio può essere quello di una condivisione client-based in cui alcuni client condividono interamente i loro

dataset. La selezione dei client può essere fatta sia in modo centralizzato, dallo stesso server che gestisce il modello globale, che su base volontaria in cui un utente può decidere spontaneamente di condividere i propri dati. Un'altra alternativa può essere quella di stabilire un certo quantitativo, come una percentuale fissata o un numero costante di sample, che ogni client deve contribuire al dataset globale, mantenendo il resto dei propri dati privati.

2.3 Strategie di aggregazione

Nel federated learning un punto importante è la strategia di aggregazione usata per aggiornare il modello globale una volta ricevuti gli update dei modelli dei client. In questa sezione viene prima di tutto introdotta la *FedAvg* la prima e più famosa strategia. Notati i problemi di questa strategia, vengono poi discusse altre strategie presenti nella letteratura che puntano a migliorarne i punti deboli, come

2.3.1 FedAvg

La strategia più conosciuta ed usata è la *FedAvg* (Federated Averagin), introdotta in McMahan et al. [28], paper che introduce per la prima volta il federated learning. Nel *FedAvg* il server aggrega i nuovi parametri calcolando una media pesata di tutti quelli ricevuti, dove il peso dato ad ogni modello è la dimensione del dataset locale usato rapporta al numero totale di sample usati nel round di training. In pratica, dati K client, ognuno con la propria copia θ_t dei parametri del modello globale al round t e con il proprio dataset locale di dimensione d^k, per $k = 1, \dots, K$, denotando con $D := \sum_{k=1}^{K} d^k$, il numero totale di sample usati in questo round di training, i parametri del modello globale al round $t + 1$ sono calcolati con

$$\theta_{t+1} := \sum_{k=1}^{K} \frac{d^k}{D} \theta_t^k$$

Questa è una tecnica semplice da capire e da implementare, ottima per eperimenti di benchmark e feasibility studies ed è la strategia utilizzata anche per questo studio. Tuttavia, è una strategia che assenga ad ogni sample di ogni dataset lo stesso peso relativo sul risultato del modello globale e ciò fa si che *FedAvg* possa fare fatica in contesti molto non-IID.

2.3.2 FedProx

Riconoscendo il problema dell'eterogeneità dei dati e delle capacità computazionali di client diversi, Li et al. [23] introducono una nuova strategia di aggregazione chiamata *FedProx*. Tale strategia funziona in modo analogo a *FedAvg* con la sola

differenza che ogni client k, anziché minimizzare la propria normale loss function L^k, minimizza la funzione

$$L^k(\theta^k) + \frac{\mu}{2}||\theta^k - \theta_t||^2$$

dove il termine di prossimità $\frac{\mu}{2}||\theta^k - \theta_t||^2$ penalizza l'allontanarsi molto dal modello globale corrente θ_t analogamente a come funziona la regolarizzazione L_2, introducendo l'iperparametro μ.

Oltre a ridurre l'impatto che l'eterogeneità che ha i dataset locali sulle performance del modello globale, facendo si che i modelli locali non possano allontanarsi eccessivamente da quello globale, permette anche un maggiore grado di eterogeneità delle prestazioni computazionali dei client. Se con la FedAvg, infatti, client che compiono molte epoche di apprendimento velocemente rischiano ritrovarsi con un modello significativamente diverso da uno di un client che ha compiuto pochi cicli di allenamento, forzando quindi round di sincronizzazione che hanno l'effetto di far andare l'intera rete alla velocità del più lento o scartando interamente i nodi più lenti dal sistema federato, il fatto di non potersi allontanare troppo dal modello globale fa si che sia più difficile che modelli locali possano divergere tra loro, rendendo l'intero sistema più efficiente e stabile.

2.3.3 SCAFFOLD

SCAFFOLD (Stocastic Controlled Averaging for Federated Learning) è una strategia introdotta in [21] per affrontare il problema del client drift, ovvero quella situazione in cui i modelli locali tendono a divergere gli uni dagli altri e allontanarsi da un minimo globale, producendo overfitting sui loro dataset locali e rallentando la convergenza del modello globale. In questa strategia viene introdotto il concetto di control variate c, ovvero un ulteriore tensore con la stessa dimensione del modello che sta venendo allenato, utilizzato per ridurre il client drift. Esiste un control variate globale c_t ed uno locale c_t^k per ogni client. Ogni client calcola l'aggiornamento dei suoi parametri come:

$$\Delta\theta_{t+1}^k = \eta(\nabla L^k(\theta_t^k) - c_t^k + c_t)$$

dove η è il learning rate e L^k è la loss function del client k. Dopodiché il server aggiornerà il modello globale con:

$$\theta_{t+1} = \theta_t + \frac{1}{K}\sum_{k=1} K\Delta\theta_{t+1}^k$$

2.3.4 Summary

Come spesso accade non esiste una *one size fits all* e diverse strategie hanno diversi pro e contro. Inoltre la necessità o efficiacia di strategie diverse dipende significativamente dai dati e le prestazioni dei client dello specifico sistema. Si possono però tracciare delle linee generali analizzando le performance di diverse strategie sugli stessi dataset di benchmark. Esistono infatti diversi studi comparativi di diverse strategie come Kairouz et al. [20].

2.4 Vantaggi

Il federated learning offre vantaggi significativi sia in termini di privacy che in termini di costo di infrastruttura. Uno dei primi vantaggi è la possibilità poter allenare un modello di machine learning in modo distribuito, potendo scaricare parte del *compute* sull'edge anziché sull'organizzazione che fornisce il modello, fattore interessante se si considerano i costi elevati che si incorrono nell'allenare modelli allo stato dell'arte. Inoltre in questo setting non c'è bisogno di inviare i dati in unica repository centralizzata, potendo quindi preservare la privacy degli utenti che generano tali dati, eliminando la necessità di un'infrastruttura di database o datawarehouse per l'ente che gestisce il modello ed eliminando il carico sulla rete per trasmettere tali dati, fattori che possono diventare significativi se se il learning è fatto su larga scala come ad esempio sulla miriade di dispositivi IoT.

2.5 Problemi

Nonostante i suoi vantaggi, il federated learning presenta anche delle difficoltà. In primis, c'è il fatto che dato che i dati locali sono tipicamente eterogenei e dotati di bias. In particolare non sono IID (Independent and identically distributed), condizione spesso richiesta nel machine learning. Il risultato è che ogni client quando allena modello gli insegna lo stesso bias presente nel suo dataset. Particolarmente significativo è lo sbilanciamento delle classi in problemi di classificazione [40] [38]. Inoltre, pur non necessitando di dover inviare dati in ad al server è ancora necessario inviargli i modelli aggiornati nel caso del partizionamento orizzontale o l'embedding delle feature nel caso verticale. Se i round di comunicazione tra client e server sono frequenti è possibile che questo risulti in un utlizzo di rente significativamente più alto, fattore che può essere problematico specialmente per i client in caso abbiano risorse limitate come reti mobili per telefoni o dispositivi IoT, oltre che un elevato uso di energia che può rapidamente consumare le batterie. Un altro problema significativo può essere quello di un leaking di dati

sensibili attraverso un *inference attack* a partire dai parametri aggiornati del modello o dal feature embedding. Questi problemi possono essere risolti con tecniche come la differential privacy o l'homomorphic encryption, ma questo comporta un costo computazionale maggiore dell'intero processo e può degradare ancora le performance del modello. In utlimo, il federated learning rimane un modello di computazione distribuita e come tale pone tutte le difficoltà di comunicazione e sincronizzazione tipiche di quest'approccio.

2.6 Sintesi

Conclundendo, il federated learning è un approccio al machine learning che offre svariati vantaggi, dalla distribuzione del calcolo per l'allenamento, alla personalizzazione del comportamento del modello in funzione dell'utente, alla protezione della privacy dei dati in contesti di dati sensibili.

Tuttavia questi benefici si ottengono al costo di dover gestire un sistema distribuito, invece di un unico server cloud, e soprattutto diventa una grande fonte di eterogeneità dei dati, rendendoli non-IID e limitando le performance del modello allenato. Per tale motivo il federated learning rimane una tecnica potente ma da utilizzare con attenzione nei contesti in cui risulta utile o necessario, come ad esempio nel caso in cui esistano regolamentazioni a protezione dei dati.

Chapter 3

Progettazione

In questo capitolo vengono introdotti e descritti i problemi affrontati e le tecniche utilizzate in questo studio. La prima sezione descrive il problema che si vuole studiare: l'ibridazione dall'apprendimento federato a quello centralizzato. Le due sezioni successive invece descrivono i dataset di benchmark che sono stati scelti per studiare il problema.

3.1 Il problema

In questo studio si vuole studiare il federated learning ibrido, analizzando come variano le performance di una rete neurale se è allenata in modo completamente federato, completamente centralizzato o con una qualche via di mezzo ibrida. Per questo sono stati presi in considerazione due dataset di benchmark federati il FEMNIST, un dataset di computer vision, e l'UCI HAR, un dataset di human activity recognition, descritti nelle sezioni successive.

Un punto fondamentale della scelta degli iperparametri delle reti o del processo di training è quello di mantenerli volutamente subottimali per poter vedere e apprezzare il miglioramento di performance dato dalla maggiore condivisione dei dataset locali. Questo motiva la scelta di modelli di dimensione ridotta, la scelta di mantenere una funzione di attivazione, sì famose ed efficace, ma comunque migliorabile come la ReLU o la scelta della classica Creoss Entropy Loss come funzione di costo, che sono sempre state usate in tutti gli esperimenti presentati.

3.2 FEMNIST

3.2.1 Origine

Il famoso dataset MNIST (Modified National Institute of Standards and Technology), l'Hello World del Machine Learning, nasce nel 1994 creato da LeCun et

al. [22] partendo dai un altro dataset del NIST. Il MNIST è stato creato a partire dai dataset SD-1 e SD-3 (Special Dataset). Lo SD-3 veniva usato come training set e conteneva immagini di cifre scritte a mano da impiegati dell'American Census Bureau, mentre lo SD-1, usato come test set, erano cifre scritte a mano da studenti delle scuole superiori americane. Le cifre degli studenti erano però scritte con una calligrafia più difficile da leggere, rendendo più difficile trarre conclusioni sulle capacità del modello alla fine del training. Per questo motivo è stato creato un unico dataset, l'MNIST che contiene un totale di 60'000 immagini di cifre sia degli studenti che degli impiegati dell'American Census Bureau.

L'EMNIST (Extended MNIST) è un'estensione del dataset originale, pubblicata nel 2017 da Cohen et al. [12] nata per rendere il problema più difficile, viste le performance altissime che le CNN ottengono con poche epoche di training sul MNIST originale. Nasce dall'unione di altri dataset gestiti dal NIST, include cifre e anche lettere sia maiuscole che minuscole, per un totale di 62 classi. L'MNIST originale include cifre scritte da circa 250 persone diverse, mentre l'EMNIST contiene dati da 3597 persone diverse e un totale di 814'255 sample diversi.

Per finire, il FEMNIST (Federated EMINST) viene introdotto in Caldas et al. [10] ed è un dataset che include tutti i sample dell'EMNIST partizionati per scrittore originale, in modo da poter essere utilizzato in contesti di apprendimento federato, in cui ogni client esegue il training non su tutti i sample dell'EMNIST, ma solo quelli dello scrittore originale corrispondente.

3.2.2 Struttura

Come tutti i suoi predeccessori, il FEMNIST è un dataset che contiene immagini di 28x28 pixel in greyscale, quindi ad un unico canale. Le cifre e lettere sono tutte centrate nell'immagine in base centro di massa dei pixel del carattere. Il dataset originale è stato costruito da, tra gli altri, dipendenti di google e come tale è stato pensato per funzionare nativamente con Tensorflow. La repository originale per generare il dataset può essere trovata su github [32]. Dato che però il codice per questi esperimenti è stato scritto nel framework di PyTorch, seguendo il dataset loader indicato su Papers with Code [36], per generare il dataset è stata usata la repository di Xiao Chenguang [9], che permette di generare un file `.hdf5` contenente l'intero dataset, potendo anche scegliere se includere tutti i caratteri o solo le cifre come nel MNIST originale.

3.2.3 Rete Neurale

Per questo dataset, essendo un problema di computer vision, è stata usata una CNN (Convolutional Neural Network). Il motivo per cui questa architettura di rete

neurale è così efficace nei problemi di image classification o semantic segmentation è che implementa nativamente un'invarianza rispetto alla posizione relativa che gli oggetti osservati nelle immagini di input. Ad esempio, dato un filtro che delinea il contorno di un viso nella sua feature map, se tale viso viene tralsato di qualche pixel nell'immagine originale, il filtro delineerà un contorno nella featuer map traslato in modo equivalente (equivarianza). Componendo layer convoluzionali e facendoli seguire da layer lineari che calcolano la classificazione delle immagini si ottiene un'effettiva invarianza nella classificazione rispetto alla posizione degli oggetti classificati nell'immagine originale. La rete usata è di dimensioni ridotte con 2 layer convoluzionali seguiti da uno lineare. L'implementazione di questa rete è indicata nel capitolo 4.

3.3 UCI HAR

3.3.1 Struttura

L'UCI HAR (University of California Irvine - Human Activity Recognition) è un dataset diverso dal FEMNIST. In primis non è un dataset di computer vision, ma di human activity recognition. Il dataset è stato introdotto in [5] e la fonte dei dati sono accellerometri e giroscopi di smartphones.

Negli esperimenti condotti 30 volontari di età compresa tra i 19 e i 48 anni hanno compiuto diverse azioni avendo uno smartphone legato in vita usato per registrare l'accelerazione lineare e la velocità angolare 3-assiale. Lo smartphone usato è un Samsung Galaxy SII, le misurazioni sono state fatte ad una frequenza di 50Hz e le azioni compiute sono il camminare, salire o scendere le scale, sedersi, alzarsi e sdraiarsi, per un totale di 6 classi di azioni diverse. I sample del dataset non sono però sequenze di misurazioni; dopo aver misurato velocità e accelerazione, grandezze vettoriali 3-dimensionali, e aver rimosso rumore dai segnali con un filtro Butterworth low-pass [35], è stata fatta un'aggregazione di tutti i valori in una finestra temporale di 2.56s. Tale aggregazione produce 561 valori reali che vengono organizzati in un unico feature vector che funziona da input del modello. La label è l'azione che stava venendo compiuta in quei 2.56 secondi. Scaricati i file del dataset, disponibile sul sito della University of California Irvine [4], sui file `README.txt` e `features_info.txt` si posso trovare informazioni dettagliate sul processing fatto per estrarre le 561 feature, mentre su youtube [29] è disponibile un video che mostra il processo di registrazione delle azioni.

Il dataset include un totale di 10299 vettori di feature che sono stati già divisi randomicamente in un test set del 30% del totale e in un training set che contiene il 70% delle feature.

3.3.2 Reti neurale

Dato che l'UCI HAR è un dataset i cui sample sono feature vector di diverse misurazioni aggregate senza una particolare struttura, per questo problema è stato usata una semplice MLP (Multi Layer Perceptron). Questo dataset è stato trattato sia a partizionamento orizzontale che a verticale e sono stati usati due modelli diversi per i due esperimenti. Nel caso del partizionamento orizzontale si è usato un MLP con un solo layer nascosto da 50 neuroni. In esperimenti preliminari si sono provate diverse dimensioni per questo modello e poi si è provati a ridurlo quanto possibile. Nel caso del partizionamento verticale ogni client è stato dotato di un MLP con un layer nascosto per calcolare l'encoding delle feature e il sever di un altro MLP con un layer nascosto per compiere la classificazione, per un modello totale che comprende 5 layer in tutto (input, hidden layer dei client, encoding layer, hidden layer del server, output layer).

Chapter 4

Implementazione

In questo capitolo viene descritta l'implementazione degli esperimenti e vengono discusse le scelte implementative. La prima sezione è dedicata agli strumenti utilizzati. Per prima cosa viene introdotto PyTorch, il framework usato per definire le reti neurali usate, dopo viene introdotto Flower un framework per implementare e simulare sistemi di federated learning e per ultimo viene introdotto anche Hydra un tool per configurare molteplici esperimenti di datascience in python comodamente.

La seconda sezione invece è dedicata alle implementazioni fatte per questi esperimenti. In primis, viene indicato il codice utilizzato per caricare i dati dalle fonti menzionate nel capitolo precedente. Successivamente vengono descritti i modelli usati per i diversi esperimenti. Infine viene discussa l'ibridazione implementata dal modello di apprendimento federato a quello centralizzato, condividendo acluni sample dei dataset dei client in un unico dataset centralizzato.

4.1 Gli strumenti

4.1.1 PyTorch

PyTorch [3] è una libreria open-source estremamente popolare e flessibile per programmare ed allenare modelli di Deep Learning. Siluppata inizialmente dal FAIR lab (Facebook's AI Reasearch), offre un'API dichiarativa in python per costruire architetture di reti neurali, lasciando totale controllo al programmatore sul passo di feed forward di dati, facilitando la rapida sperimentazione di approci diversi e l'implementazione di pipeline complesse come le GAN (Generative Adversarial Nets) [16], sperimentare tra transformer encoder-decoder [33] o decoder only [30].

PyTorch include una libreria estensiva di diversi tipi di layer, loss functions, funzioni di attivazione e algoritmi di ottimizzazione e la feature di autograd, per

cui facendo uso di un grafo dinamico di computazione PyTorch gestisce in automatico il calcolo del gradiente della loss function rispetto ad ogni parametro, rende l'implementazione di reti neurali estremamente facile, specie per casi particolarmente complicati come CNNs o RNNs.

4.1.2 FlowerAI

Flower (o FlowerAI) [1] è un framework open-soruce progettato semplificare l'implementazione di sistemi di federated learning. Pur essendo una libreria python, supporta anche SDK per Android, iOS e C++, rendendolo ideale anche in ambienti mobile o di IoT. Flower fornisce un'API per definire e implementare comodamente ogni specifico client in un apprendimento federato, configurandone il modello da allenare, l'algoritmo di training e valutazione. Allo stesso modo è controllabile il comportamento del server e la strategia di aggregazione dei parametri aggiornati del modello forniti dai client (i.e. un oggetto di tipo flwr.server.strategy.Strategy).

Flower è stato usato per eseguire le simulazioni di apprendimento federato sui dataset FEMNIST e UCI HAR. In ogni simulazione, ogni client ha allenato il suo modello locale per un numero basso di epoche (5 per l'UCI HAR, 6 per il FEMNIST) e dopodiché manda i parametri aggiornati al server. La strategia utilizzata per aggregare i server è la FedAvg. Questo ciclo di allenamento e aggregazione costituisce un round ed ogni simulazione è durata 20 round.

Va notato che Flower è un framework pensato principalmente per il federated learning orizzontale e non ha un supporto nativo per contesti a partizionamento verticale. La soluzione, suggerita dallo stesso team di Flower [15], è quella di riutilizzare il meccanismo di scambio di parametri da client a server per passare invece l'encoding delle feature calcolato dal modello locale. L'effetto collaterale di questo sistema è che l'intero feed forward delle feature, nei modelli locali dei client ad encodings e poi nel modello del server da encoding a classificazione, richiede l'intero round di training. Questo implica che ogni round può eseguire una sola epoca di training per round, cosa che può essere aggirata moltiplicando il numero di round per il numero di epoche desiderate, e che si è costretti ad un regime di full-batch training, a meno di non voler implementare una gestione delle batches dei dataset in funzione del round.

4.1.3 Hydra

Hydra [2] è una libreria potente e flessibile per configurare ed eseguire molteplici esperimenti di data science implementati con script python. Le configurazioni in Hydra sono gestite tramite file `.yml`, ma permette anche la specifica di parametri dinamicamente via command line. Supporta anche configurazioni gerarchiche, per

cui è possibile specificare configurazioni di default da cui *ereditare* certi parametri e customizzarne altri per l'esperimento desiderato, in modo da permettere il riutilizzo delle configurazioni.

Supporta nativamente anche configurazioni "multi-run" per cui lo stesso script è eseguito più volte con configurazioni diverse, feature estremamente conveniente quando si vogliono provare diversi iperparametri del proprio modello di machine learning, come in questo studio.

4.2 Le implementazioni

4.2.1 Data loading

FEMNIST

Come anticipato il FEMNIST è stato scaricato usando la repository di Xiao Chenguang [9]. Tale repository include uno script per generare due file, `write_digits.hdf5` e `write_all.hdf5`, per la versione dei dataset con e senza le immagini delle lettere. Tale file, una volta letto, contiene un dizionario le cui chiavi sono id degli scrittori originali e gli elementi sono un'ulteriore dizionario con le due chiavi `"images"` e `"labels"` per i dati dell'immagine e il numero della rispettiva classe.

Quella che segue è la funzione utilizzata per aprire il dataset `.hdf5` creato e istanziare i `Dataset` di PyTorch per ognuno dei client usati nella simulazione:

```
    def _get_femnist_datasets(
        num_writers: int,
        val_ratio: float,
        test_ratio: float,
        only_digits: bool = False,
    ) -> tuple[list[Dataset], list[Dataset], list[Dataset]]:
        dataset_file = "write_digits.hdf5" if only_digits
           else "write_all.hdf5"
        full_dataset = h5py.File(f"data/{dataset_file}", "r"
           )
        writers = sorted(full_dataset.keys())[:num_writers]
        train_sets = []
        val_sets = []
        test_sets = []

        for writer in writers:
            images = full_dataset[writer]["images"][:]
            labels = full_dataset[writer]["labels"][:]
            client_dataset = FemnistWriterDataset(
                images, labels, transform=femnist_transform
            )

            train_subset, val_subset, test_subset =
               _split_dataset(
                client_dataset, val_ratio, test_ratio
            )
            train_sets.append(train_subset)
            val_sets.append(val_subset)
            test_sets.append(test_subset)

        full_dataset.close()
        return train_sets, val_sets, test_sets
```

Tutti gli argomenti di questa funzione sono controllabili tramite la configurazione Hydra usata per eseguire lo script. Si noti come l'implementazione supporti anche un test di validazione per una possibile implementazione di early-stopping. I dataset di validazione sono però stati ignorati nel corso degli esperimenti e questa feature è rimasta da esperimenti preliminari.

HAR

Il dataset UCI HAR è distribuito direttamente dalla University of California Irvine [4]. Scaricato e decompresso il dataset, si possono trovare i dati separati nelle due cartelle `test` e `train`. Entrambe le cartelle contengono un file per le feature, un file per l'indice numerico della classe e un file con l'id della persona che ha compiuto l'azione. I sample sono scritti su questi file `.txt` riga per riga in modo corrispondente (la prima riga nel file di labels è la label per il feture vector nella prima riga del file di feature, che è stata compiuta dalla persona con id che compare nella prima riga). Per organizzare queste informazioni in `Dataset`

PyTorch strutturato si è resa necessaria una funzione apposita che riassociasse le informazioni tra questi file.

Quelle che seguono sono le funzioni utilizzate per aprire i file del dataset come sono forniti dall'UCI i Dataset di PyTorch per ognuno dei client 30 e partizionare correttamente i sample:

```python
def _load_har_dataset(train: bool):
    variant = "train" if train else "test"
    full_features = [[] for _ in range(30)]
    full_labels = [[] for _ in range(30)]

    # get subject ids for measurements
    with open(f"data/subject_{variant}.txt", "r") as
       subject_file:
        subject_ids = [int(line.strip()) - 1 for line in
           subject_file]
    # get features and match them with the user
    with open(f"data/X_{variant}.txt", "r") as feature_file:
        for i, line in enumerate(feature_file):
            features = [float(num) for num in line.split()]
            full_features[subject_ids[i]].append(features)
    # get labels and match them with the user
    with open(f"data/y_{variant}.txt", "r") as label_file:
        for i, line in enumerate(label_file):
            label = int(line)
            one_hot = torch.zeros(6, dtype=torch.float32)
            one_hot[label - 1] = label
            full_labels[subject_ids[i]].append(one_hot)

    if train:
        return full_features, full_labels
    # flatten in one dataset for testing
    else:
        return [
            feature for user_features in full_features for
               feature in user_features
        ], [label for user_labels in full_labels for label
           in user_labels]


def _get_har_datasets() -> tuple[list[Dataset], Dataset]:
    train_features, train_labels = _load_har_dataset(True)
    test_features, test_labels = _load_har_dataset(False)

    client_datasets = [
        HarDataset(
            features=torch.tensor(train_features[cid], dtype
               =torch.float32),
            labels=train_labels[cid],
        )
        for cid in range(len(train_features))
        # some clients dont include measurements in the
           trainset
        if len(train_features[cid]) > 0
    ]
    test_dataset = HarDataset(
        torch.tensor(test_features, dtype=torch.float32),
           test_labels
    )
    return client_datasets, test_dataset
```

In questo caso la logica per istanziare il dataset federato è stata divisa in due funzioni in quanto, diversamente dal FEMNIST in cui il dataset originale è già partizionato per client, in questo caso c'è bisogno di (i) aprire 3 file diversi `subject_{variant}.txt`, `X_{variant}.txt` e `y_{variant}.txt` e riassociare i dati tra loro e (ii) instanziare i `Dataset` di PyTorch. La funzione `_load_har_dataset` si occupa di fare la prima tasks, mentre `_get_har_datasets` si occupa di istanziare correttamente le classi `Dataset`. Si noti che i dataset locali sono partrizionati per utente, mentre il dataset di test è un unico dataset con sample di tutti gli utenti.

4.2.2 I modelli

Per tutti e 3 gli esperimenti condotti sono stati utilizzati modelli diversi.

FEMNIST - CNN

Per il dataset FEMNIST, essendo un problema di computer vision, è stata usata una CNN con 2 layer convoluzionali, rispettivamente con 32 e 64 canali di output, con kernel size di 3, padding di 1 e max pooling tra di essi, seguiti da un layer fully connected di 512 neuroni. Il layer finale, di 10 o 62 neuroni a seconda che si usino le immagini delle sole cifre o di tutti i caratteri, usa una log softmax come funzione di attivazione, mentre tutti gli altri layer utilizzato la ReLU. L'implementazione in PyTorch di questa rete neurale è la seguente:

```
class CnnEmnist(nn.Module):
    def __init__(self, num_classes: int):
        super(CnnEmnist, self).__init__()
        # reused pooling layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2,
           padding=0)
        # 2 convolutional layers
        self.conv1 = nn.Conv2d(
            in_channels=1,
            out_channels=32,
            ke
            \begin{figure}[htbp]  % h: here, t: top, b:
               bottom, p: page
                \centering
                \includegraphics[width=0.8\textwidth]{../
                   plots/femnist-horizontal/sgd/results-h0_2
                   -hm_share-disjoint-lr0_01-e6-
                   torch_optim_SGD.png}  % Adjust width as
                   needed
                \caption{Caption describing the image}
                \label{fig:femnists2sgd}
            \end{figure}rnel_size=3,
            padding=1
        )
        self.conv2 = nn.Conv2d(
```

```
            in_channels=32,
            out_channels=64,
            kernel_size=3,
            padding=1,
        )
        # fully connected layer
        self.fc = nn.Linear(64 * 7 * 7, 512)
        # output layer
        self.classification = nn.Linear(512, num_classes)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        # flatten
        x = x.view(-1, 64 * 7 * 7)
        x = F.relu(self.fc(x))
        out = F.log_softmax(self.classification(x), dim=1)
        return out
```

HAR HFL - MLP

Il modello utilizzato per l'UCI HAR a partizionamento orizzontale è una semplice rete neurale composta di soli layer fully connected con un solo hidden layer. Ha un input per le 561 feature, un output di 6 neuroni per le 6 classi e un piccolo layer intermedio di 50 neuroni. Nonostante le dimensioni ridotte del modello (28406 parametri totali) si è riusciti comunque ad ottenere ottime prestazioni ($> 90\%$), dimostrando la validità dell'approccio anche in contesti a risorse ridotte, come telefoni o dispositivi IoT. Questa simulazione è stata anche eseguita interamente in CPU, senza accelerazione GPU, perché dato la maggiore quantità di RAM (32GB di RAM contro 6GB di VRAM sulla GPU) sulla macchina usata per gli esperimenti, si è riusciti a terminare la simulazione più velocemente data la maggiore possibilità di parallelizzazione del training svolto dai client. L'implementazione di questo modello in PyTorch è la seguente:

```
class HarModel(nn.Module):
    def __init__(self, num_classes: int):
        super(HarModel, self).__init__()
        self.fc1 = nn.Linear(561, 50)
        self.fc2 = nn.Linear(50, num_classes)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        out = F.log_softmax(self.fc2(x), dim=1)
        return out
```

HAR VFL - MLP distribuito

In ultimo è stato anche provato un modello a partizionamento verticale delle feature dell'UCI HAR. In questo contesto, dati K client, ogni modello di un client

prende $561/K$ feature e calcola un encoding di queste in un vettore dimensione d. Le dimensioni del vettore provate sono state $d = 10$ e $d = 100$. Calcolati gli encodings, il server raccoglie K di questi encodings in unico vettore di lunghezza Kd. Questo vettore di encodings viene poi passato ad un ultima rete neurale fully connected con un layer nascosto di 50 neuroni e l'output è di nuovo un vettore di dimensione 6 per le 6 attività. Il modello dei client è stato implementato con il codice seguente:

```
class ClientVerticalModel(nn.Module):
    def __init__(self, num_features: int):
        super(ClientVerticalModel, self).__init__()
        self.input = nn.Linear(num_features, 50)
        self.latent = nn.Linear(50, latent_vector_length)

    def forward(self, x):
        x = F.relu(self.input(x))
        latent = F.relu(self.latent(x))
        return latent
```

Mentre il codice per la rete del server che raccoglie gli encodings e calcola la classificazione dell'input è:

```
class ServerVerticalModel(nn.Module):
    def __init__(self, num_clients, num_classes: int):
        super(ServerVerticalModel, self).__init__()
        self.input = nn.Linear(num_clients *
           latent_vector_length, 50)
        self.output = nn.Linear(50, num_classes)

    def forward(self, x):
        x = F.relu(self.input(x))
        out = F.log_softmax(self.output(x), dim=1)
        return out
```

Per comodità in fase di testing del modello è stato anche implementata una versione unica del modello, in grado di prendere le 561 feature e calcolare la classificazione direttamente, facendo uso dei modelli client e del modello server passati nel costruttore. Dato che può facilitare a comprendere meglio il funzionamento del VFL di seguito è riportato anche il codice di questo modello:

```
class FullVerticalModel(nn.Module):
    def __init__(
        self,
        client_models: list[ClientVerticalModel],
        server_model: ServerVerticalModel,
        num_clients: int,
    ):
        super(FullVerticalModel, self).__init__()
        self.client_models = client_models
        self.server_model = server_model
        self.num_clients = num_clients

    def forward(self, x):
```

```
        embeddings = []
        for cid, model in enumerate(self.client_models):
            extracted_features = extract_features(x, self.
               num_clients, cid)
            embeddings.append(model(extracted_features))

        embeddings_aggregated = torch.cat(embeddings, dim=1)
        return self.server_model(embeddings_aggregated)
```

Ognuno di questi modelli è stato allenato con Cross Entropy Loss come loss function e con due diversi algoritmi di ottimizzazione. Il primo provato è il classico SGC (Stochastic Gradient Descent) con learning rate di 0.01 e momentum di 0.9. Esperimenti preliminari hanno provato altri valori come un learning rate più basso come 0.001 o più grande come 0.1 o la rimozione del momentum, ma le performance migliori sono state ottenute con i valori indicati precedentemente; la CNN per il FEMNIST non riusce nemmeno a convergere senza momentum e l'accuracy sul test dataset si ferma intorno al 6%. L'altro algoritmo usato è stato l'AdamW [25], ovvero l'algoritmo Adam (Adaptive Movement Estimation) con weight decay. In esperimenti preliminari è anche stato provato l'Adam classico ma si è notato che sia il loss che la test accuracy erano molto instabili durante il training, suggerendo suggerendo che i parametri del modello fossero altrettanto instabili. Per tale motivo si è scelto di utilizzare la variante AdamW che implementa il weight decay come meccanismo di regolarizzazione per mantere i pesi del modello più stabili. Si noti che si è evitato esplicitamente di utilizzare una L2 regularization come loss function per risolvere l'instabilità di Adam, dato che la L2 regularization e il weight decay sono equivalenti solo sotto SGD [25].

4.2.3 Ibridazione

Per gli esperimenti a partizionamento orizzontale si sono fatti esperimenti per diversi gradi di ibridazione da simulazione completamente federate a simulazioni completamente centralizzate. La configurazione dello script di training ammette due paramerti per controllare il grado di ibridazione da usare: `hybrid_ratio` e `hybrid_method`, che controllano rispettivamente la percentuale dei dataset dei client da condividere in un unico dataset globale e il meccanismo di condivisione dei dataset. Le tecniche di condivisione implementate sono state chiamate `unify` e `share-disjoint`. Nel caso della `share-disjoint` ogni client prende la percentuale indicata di sample dal suo dataset e li sposta nel dataset condiviso, mentre nella `unify` la percentuale indicata di client condividono il loro dataset intero. Ad esempio, supponendo di avere 10 client con 10 dataset che contengono tutti 100 sample con un `hibrid_ratio` di 0.2, con la `share-disjoint` ognuno dei 10 client prende 20 sample dal suo dataset e li mette nel dataset condiviso, ottenento un

dataset condiviso di 200 sample e 10 client con 80 sample ciascuno, mentre nella unify vengono selezionati randomicamente 2 client che condividono tutti i loro 100 sample, ottenendo un dataset condiviso di 200 sampe e 8 client con 100 sample ciascuno.

Il codice che segue è il codice utilizzato per condividere parte dei dataset dei client, una volta che i dataset sono stati istanziati già partizionanti per client:

```
def _centralize_trainsets(
    train_sets: list[Dataset], num_clients: int,
       hybrid_ratio, hybrid_method
):
    if hybrid_method == "unify":
        num_centralized = int(hybrid_ratio * num_clients)
        shared_sets = []
        for _ in range(num_centralized):
            random_index = random.randint(0, len(train_sets)
                - 1)
            shared_sets.append(train_sets.pop(random_index))

        if num_centralized > 0:
            train_sets = [ConcatDataset(shared_sets)] +
               train_sets

    elif hybrid_method == "share-disjoint":
        # this method removes num_shared samples from
           clients and puts them in the collective
        shared_sets = []
        client_sets = []
        for train_set in train_sets:
            set_size = len(train_set)
            num_shared = int(hybrid_ratio * set_size)
            if num_shared > 0:
                shared_set, unshared_set = random_split(
                    train_set, [num_shared, set_size -
                       num_shared]
                )
                shared_sets.append(shared_set)
                client_sets.append(unshared_set)
            else:
                # dont lose clients sets!
                client_sets.append(train_set)
        if len(shared_sets) > 0:
            train_sets = client_sets + [ConcatDataset(
               shared_sets)]

    return train_sets
```

Chapter 5

Risultati

In questo capitolo vengono presentati i risultati degli esperimenti condotti. Il capitolo è diviso in tre sezioni. Le prime due sezioni, una per dataset studiato, hanno la stessa struttura: il primo paragrafo sintetizza il setup dell'esperimentio presentato e discute anche qualche esperimento preliminare condotto che ha portato alla decisione di alcuni parametri. In ultimo, la sezione finale discute possibili miglioramenti al setup utilizzato in questo studio.

5.1 FEMNIST

In questa sezione vengono descritti gli esperimenti fatti con il dataset FEMNIST e vengono descritti i risultati.

5.1.1 Set up

Come già anticipato, gli esperimenti sul dataset FEMNIST sono stati fatti utilizzando una CNN con 2 layer convoluzionali con 32 e 64 canali di output, seguiti da un layer lineare con 512 neuroni. Si sono provate le strategie di ibridazione `unify` e `share-disjoint`, condividendo i dataset selezionando alcuni client i cui dataset vengono interamente condivisi oppure prendendo una parte di sample da ogni dataset locale, mentre come gradi di ibridazione si è andati da 0%, completamente federato, al 100%, completamente centralizzato, in step di 10%. Gli algoritmi di ottimizzazione provati sono SGD con learning rate $\eta = 0.01$ e momentum $\beta = 0.9$ e l'AdamW con learning rate $\eta = 0.01$. Il training ha usato 6 epoche locali ed è durato per 20 round. Sono stati usati i sample di soltanto 100 scrittori del totale presenti nel FEMNIST. I risultati presentati successivamente sono una media su 20 simulazioni.

Oltre ai risultati di questo esperimento, presentati nel paragrafo successivo, sono stati fatte anche simulazioni preliminari con iperparametri diversi. Primo

fra tutti è il numero di client, che negli esperimenti finali è stato mantenuto basso per velocizzare il processo di training, mentre sono state eseguite simulazioni fino a 1000 client utilizzati. Il numero poi stato mantenuto basso perché in tutti gli esperimenti fatti i risultati ottenuti erano sempre gli stessi, indipendentemente dal numero di client che partecipavano. Un'altra risultato interessante è stato l'esperimento utilizzando SGD senza momentum, in cui si è visto che la CNN non è riuscita a imparare, mantenendo un accuracy sul dataset di test intorno al 6%.

5.1.2 Risultati

Il dataset FEMNIST è quello dei due che si è dimostrato più difficile da imparare e su cui è più notevole il miglioramento di performance dato dalla condivisione dei dati (circa il 10% dal setting totalmente federato a quello totalmente centralizzato). La figura 5.3 mostra i risultati del training ai due estremi di ibridazione: **??** mostra i risultati dell'apprendimento federato, mentre **??** quelli dell'apprendimento centralizzato.

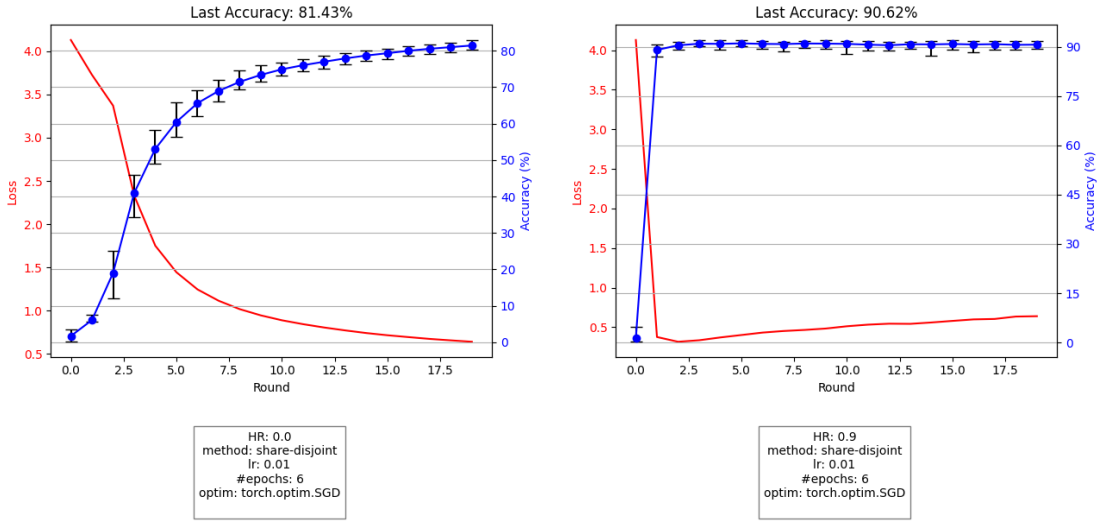


Figure 5.1: Risultati del del training per ogni round. A sinistra (a) si può vedere l'apprendimento federato, a destra (b) quello centralizzato. Algoritmo di ottimizzazione (SGD) e metodo di condivisione (`share-disjoint`) sono tenuti fissati.

Come si può vedere nel caso federato le performance raggiungono una precisione del 81.43% sul testset e si può anche notare che il modello non è ancora arrivato a convergenza. Continuando il training per ulteriori round possiamo quindi aspettarci di poter migliorare le prestazioni ancora. Il modello centralizzato arriva ad un'accuracy maggiore di un 9% e soprattutto bastano 2 round per ottenere risultati sopra al 90%.

Un primo caso di chiara convergenza intorno all'88% lo si vede quando si inizia

a condividere il 20% dei dataset come mostrato in figura **??**, dove si può vedere che l'accuracy del modello smette di migliorare intorno al decimo round.

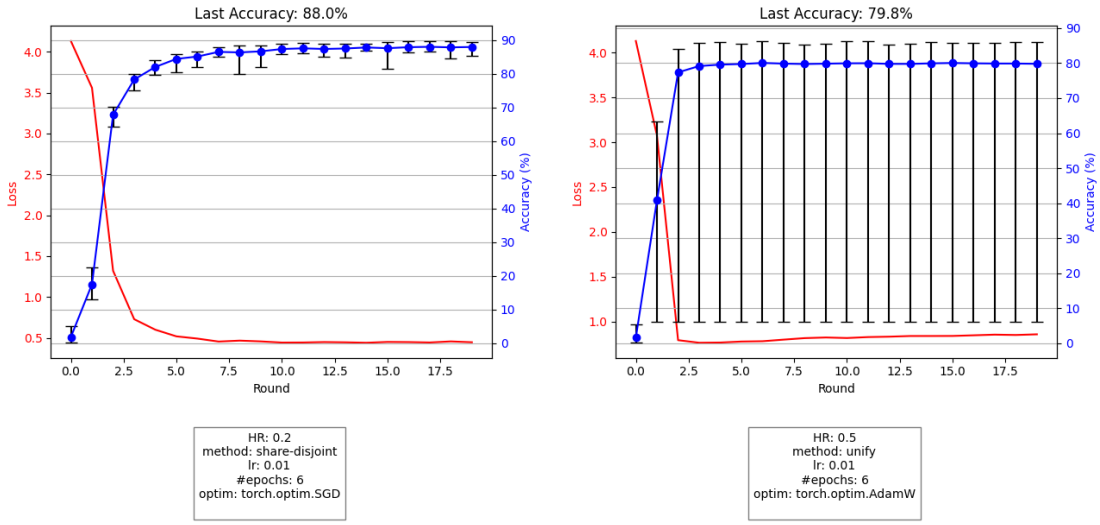
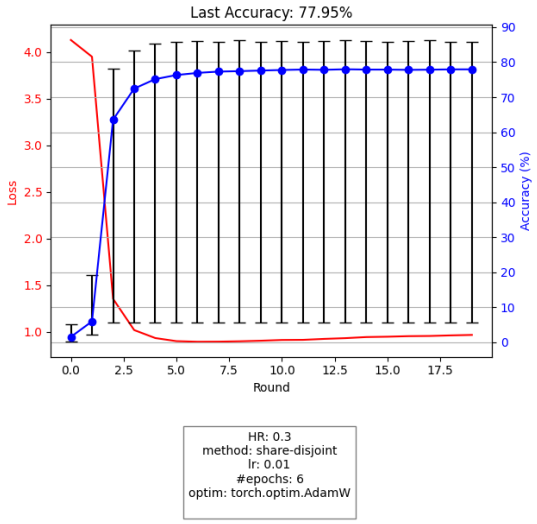


Figure 5.2: Risultati del training diversi. Il grafico a sinitra (a) mostra i risultati di una condivisione `share-disjoint` al 20%, mentre quello a destra (b) usa il metodo `unify` al 50%.

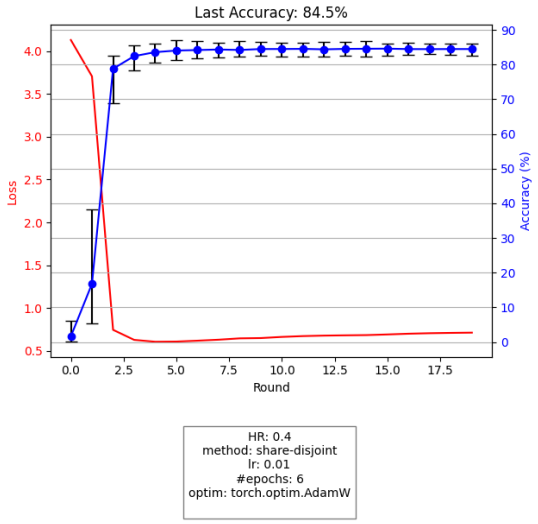
Dei due ottimizzatori provati, SGD è quello che performa meglio: è più veloce nell'apprendimento, ottiene performance migliori alla fine del training ed è molto più stabile. In particolare tutte le run che hanno utilizzato AdamW e il metodo di condivisione `unify` mostrano sempre intervalli di incertezza enormi, come quello riportato in figura **??**. La situazione è analoga anche usando la condivisione `share-disjoint` fino a che il grado di condivisione non raggiunge un minimo del 40%, come mostrato nelle figure 5.3a e 5.3b, e anche dopo oltre il 40% non è garantita una maggiore stabilità, come ha rivelato l'esperimento con 80% di condivisione (figura 5.3c).

Guardando questi grafici si può notare come il comportamento medio non sia necessariamente rappresentativo di tutti i modelli allenati con AdamW, dato l'ampio margine d'incertezza e si potrebbe pensare che, al di là dei problemi di instabilità sia comunque possibile ottenere un modello performante utilizzando quest'algoritmo. Va però notato che comunque anche i modelli meglio performanti allenati con AdamW, non hanno raggiunto performance superiori né uguali a quelle ottenute con SGD, con massimo di 90.69% (mediamente) per SGD, contro un 86% circa di massimo totale.

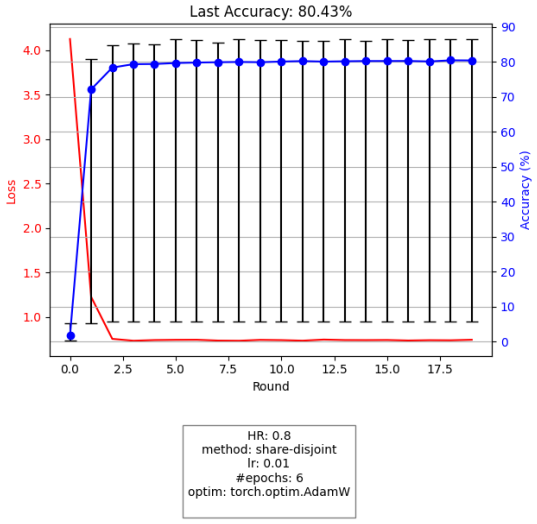
Un'ultima osservazione che può essere fatta è come il metodo di condivisione `share-disjoint` sia generalmente più stabile rispetto al `unify`. La spiegazione è semplice: `unify` scegliendo certi client da scegliere come dataset condivisi crea un dataset globale che conterrà gli stessi bias presenti nei dataset dei client se-



(a) Figura 5.3(a)



(b) Figura 5.3(b)

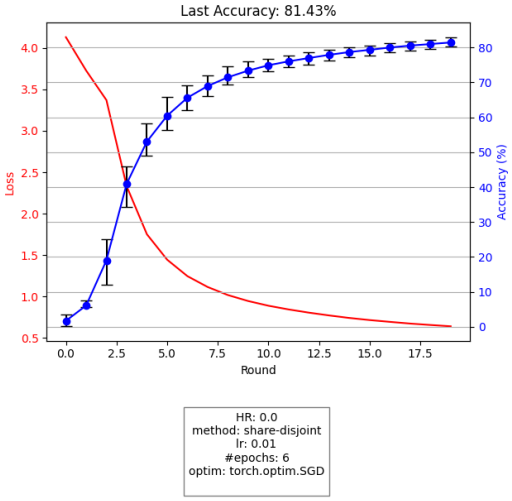


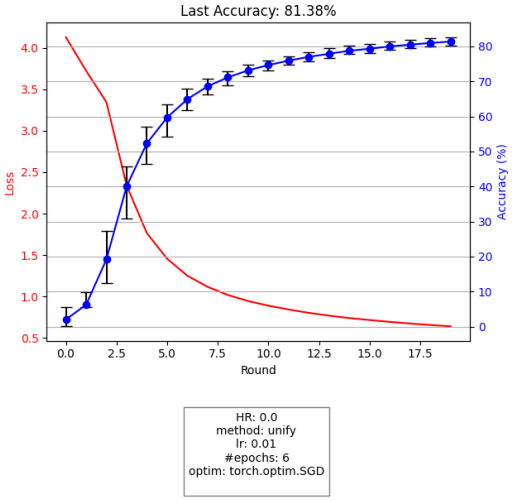
(c) Figura 5.3(c)

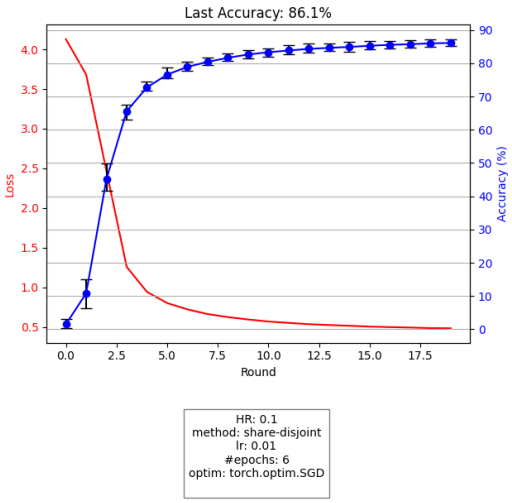
Figure 5.3: Diversi risultati di training con AdamW e metodo `share-disjoint`. Il primo (a) usa una condivisione del 30%, il secondo (b) al 40% il minimo per cui AdamW ha un andamento stabile, l'ultimo (c) mostra un altro training instabile con condivisione all'80%.

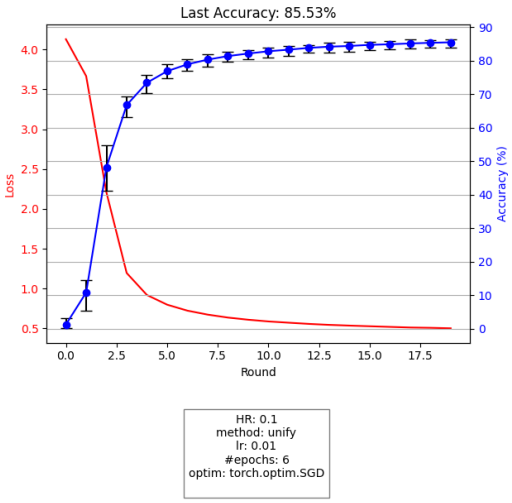
lezionati; `share-disjoint` invece, selezionando alcuni sample da tutti i dataset locali, garantisce di coprire, almeno in parte, tutte le peculiarità nei dati di tutti i client. In questo dataset questo effetto non è particolarmente marcato se non con l'AdamW o nei primissimi round di training con basso gradi di condivisione in SGD.

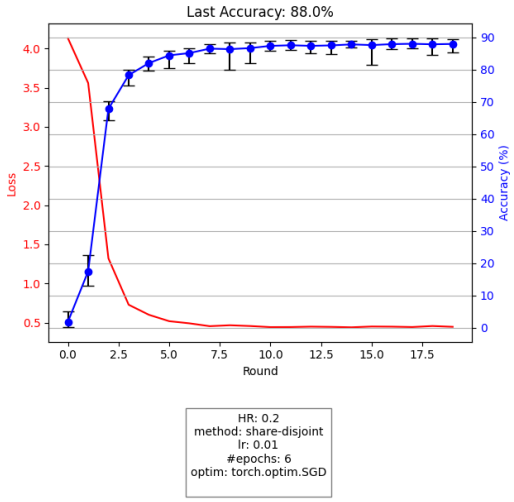
Quella che segue è una lista dei grafici di tutti gli esperimenti che sono stati svolti.

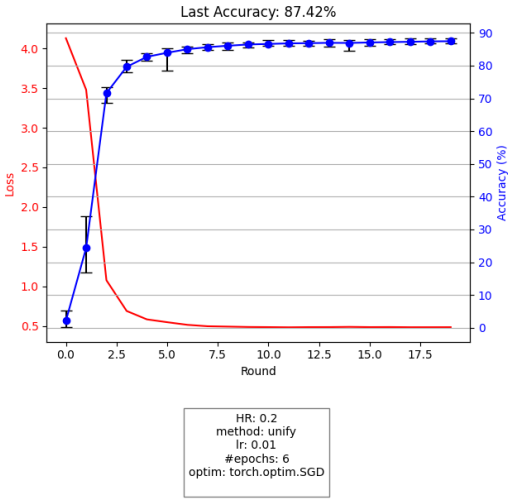




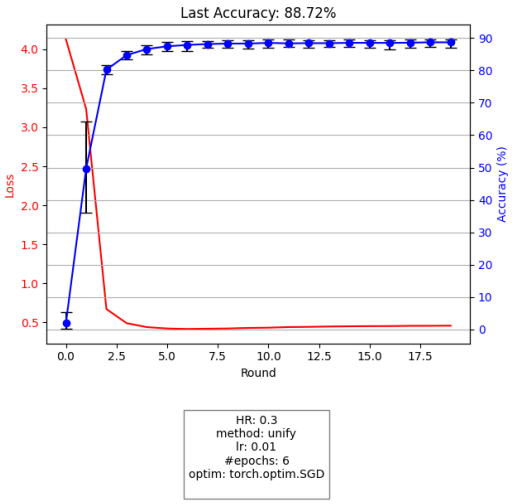




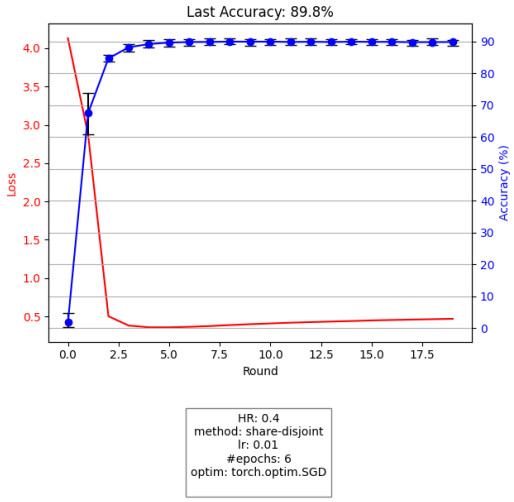


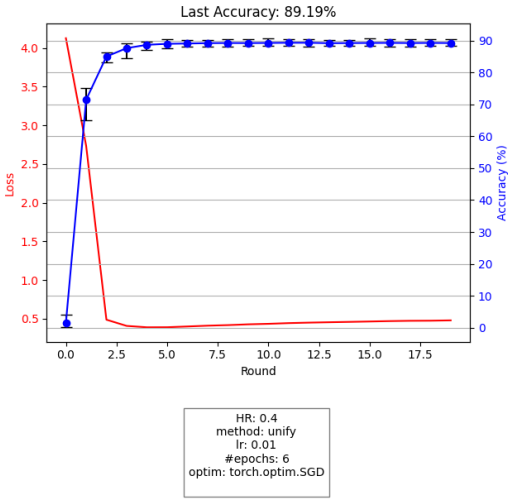


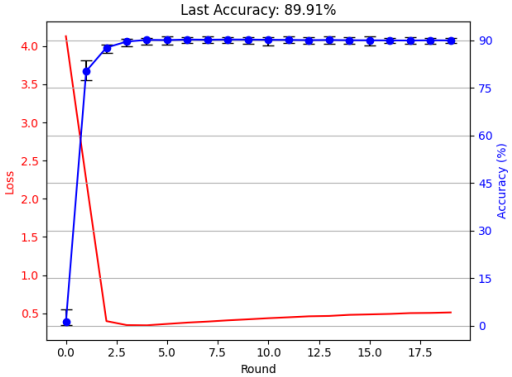
../plots/femnist-horizontal/sgd/res



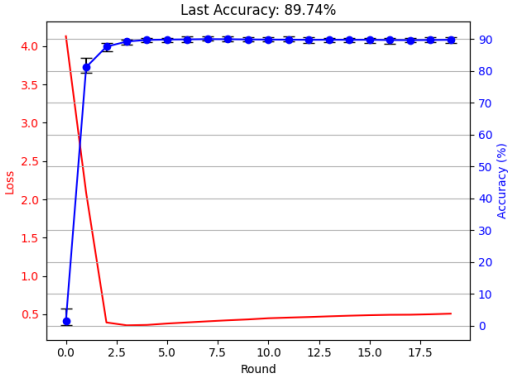
01-e6-torch_



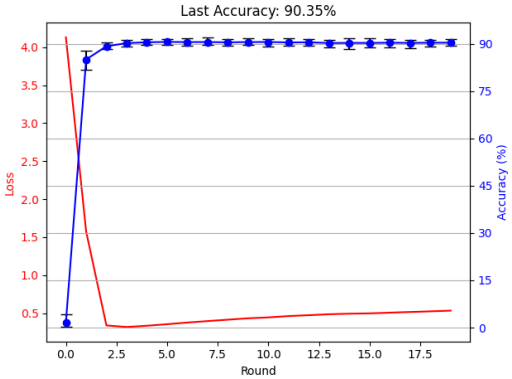




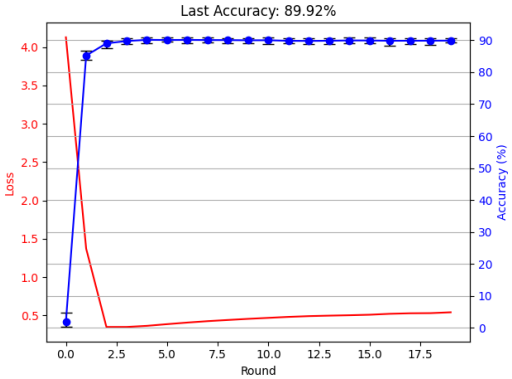
HR: 0.5
method: share-disjoint
lr: 0.01
#epochs: 6
optim: torch.optim.SGD



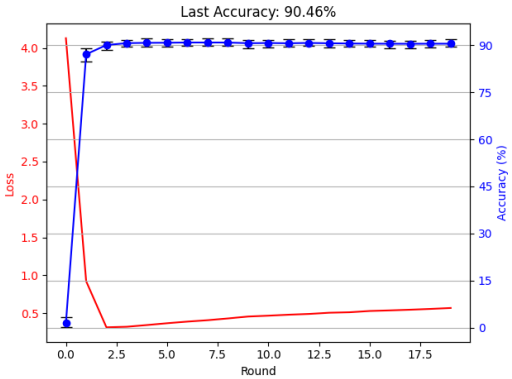
HR: 0.5
method: unify
lr: 0.01
#epochs: 6
optim: torch.optim.SGD



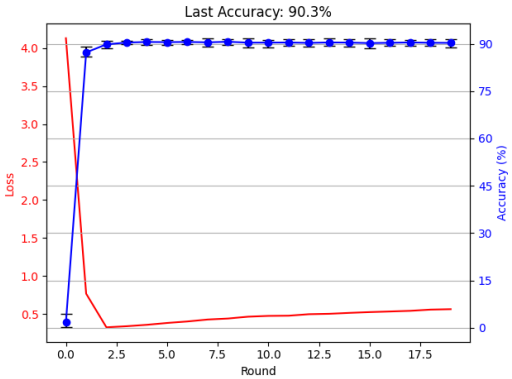
HR: 0.6
method: share-disjoint
lr: 0.01
#epochs: 6
optim: torch.optim.SGD



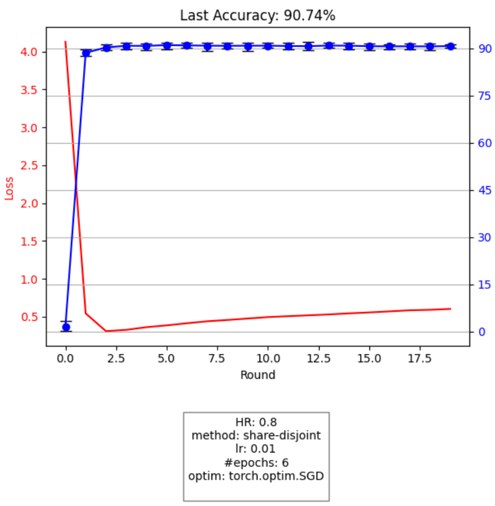
HR: 0.6
method: unify
lr: 0.01
#epochs: 6
optim: torch.optim.SGD

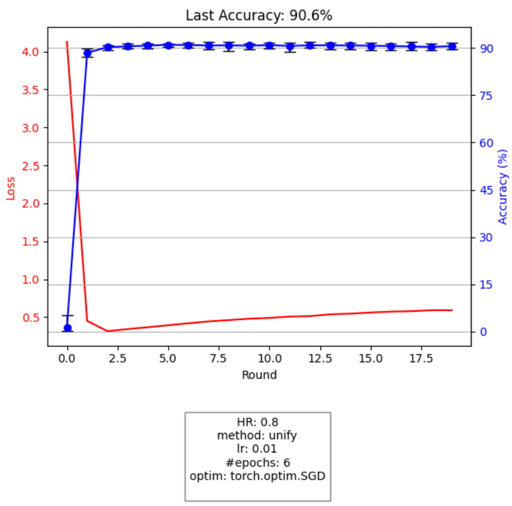


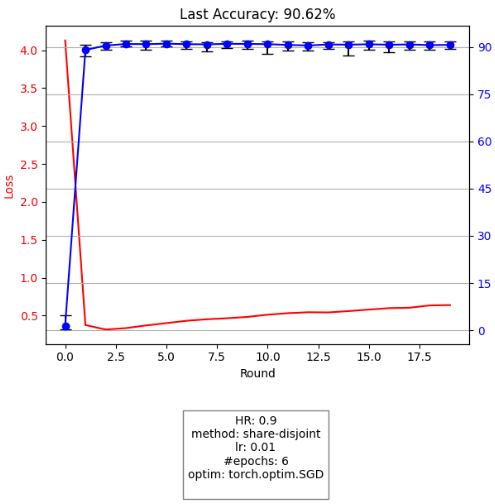
HR: 0.7
method: share-disjoint
lr: 0.01
#epochs: 6
optim: torch.optim.SGD

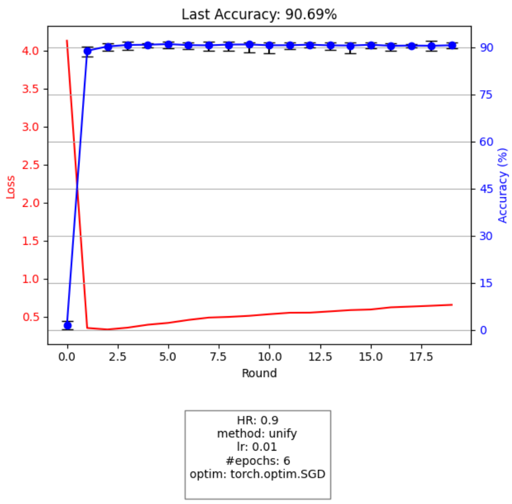


HR: 0.7
method: unify
lr: 0.01
#epochs: 6
optim: torch.optim.SGD





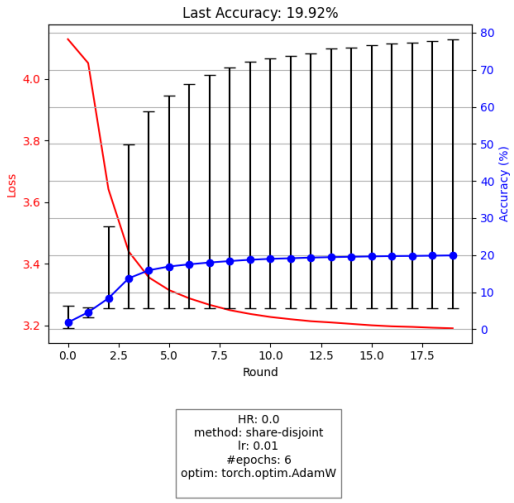


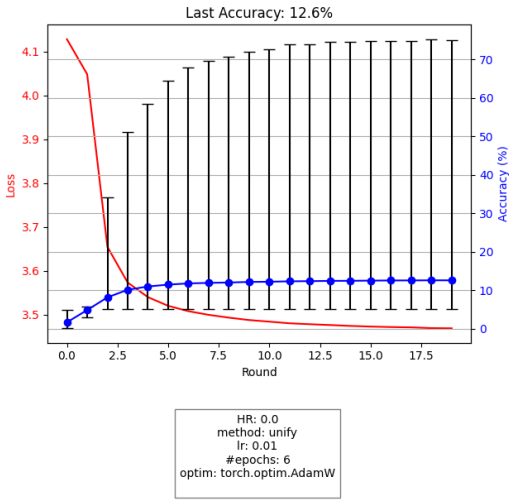


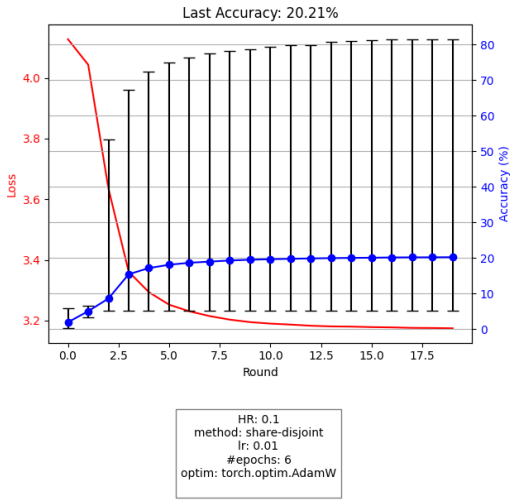
../plots/femnist-horizontal/sgd/result[illegible]

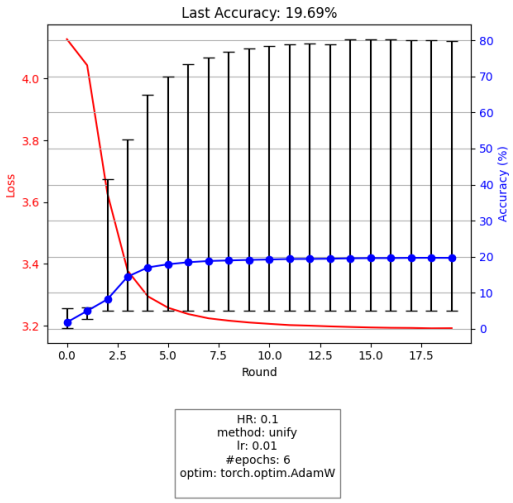
../plots/femnist-horizontal/sgd/[illegible]

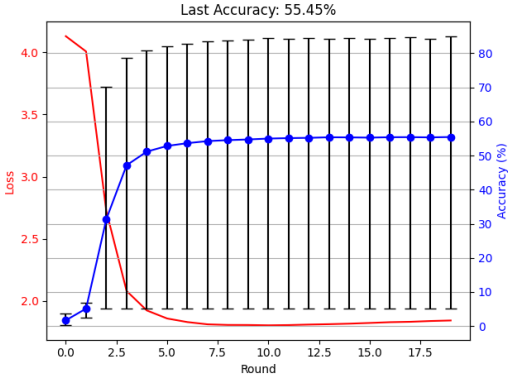
Ora sono riportati i risultati di AdamW.



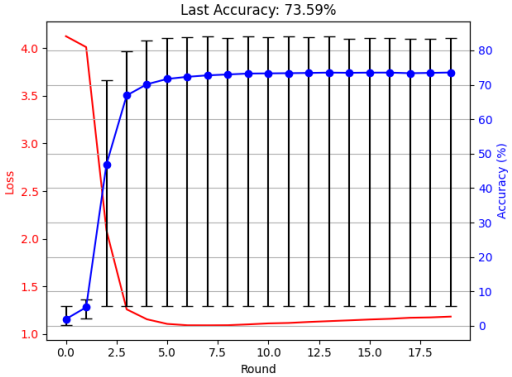




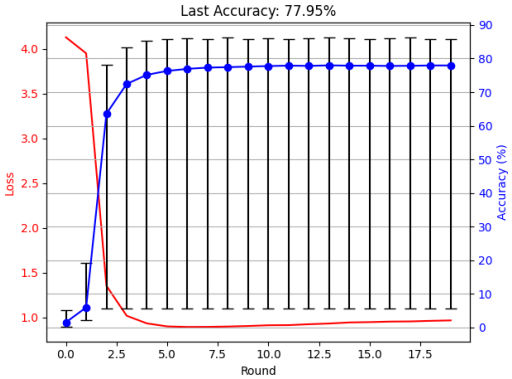




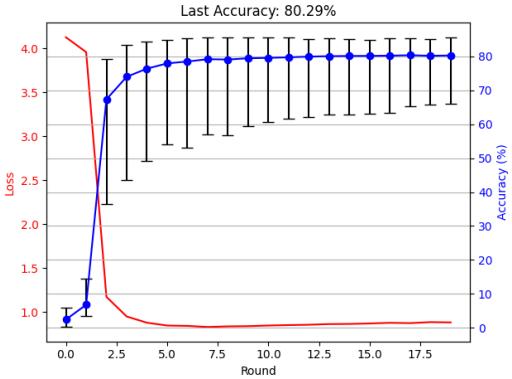
HR: 0.2
method: share-disjoint
lr: 0.01
#epochs: 6
optim: torch.optim.AdamW



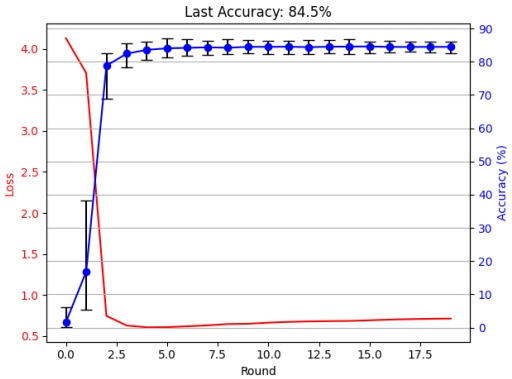
HR: 0.2
method: unify
lr: 0.01
#epochs: 6
optim: torch.optim.AdamW



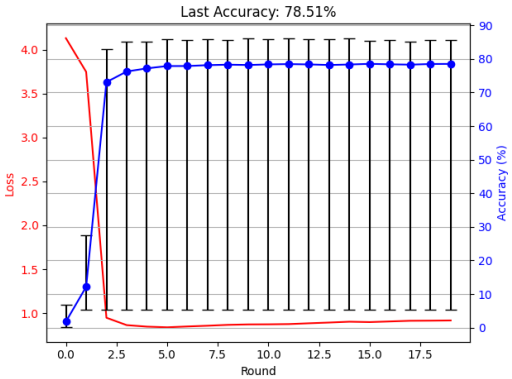
HR: 0.3
method: share-disjoint
lr: 0.01
#epochs: 6
optim: torch.optim.AdamW



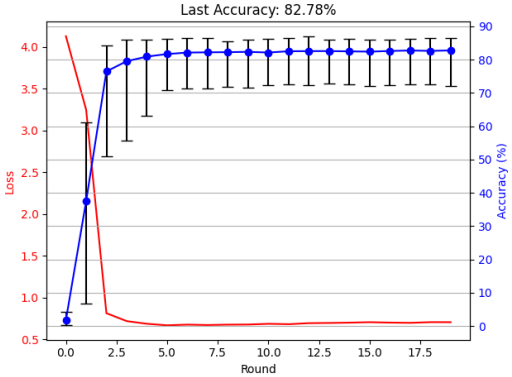
HR: 0.3
method: unify
lr: 0.01
#epochs: 6
optim: torch.optim.AdamW



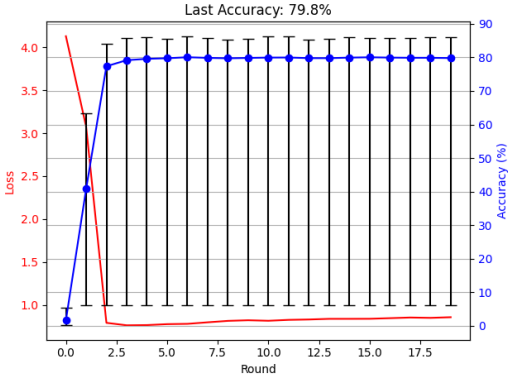
HR: 0.4
method: share-disjoint
lr: 0.01
#epochs: 6
optim: torch.optim.AdamW



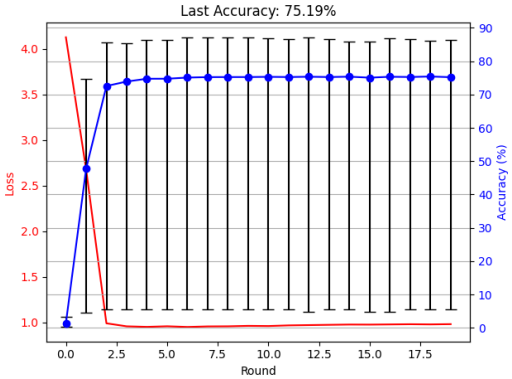
HR: 0.4
method: unify
lr: 0.01
#epochs: 6
optim: torch.optim.AdamW



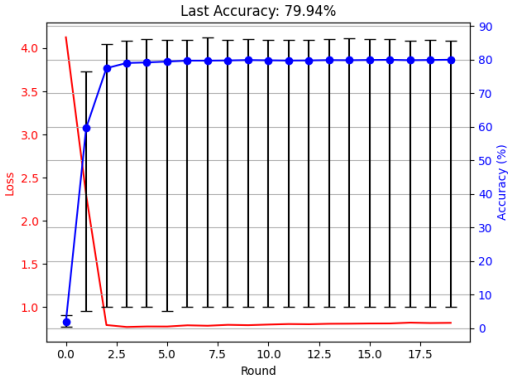
HR: 0.5
method: share-disjoint
lr: 0.01
#epochs: 6
optim: torch.optim.AdamW



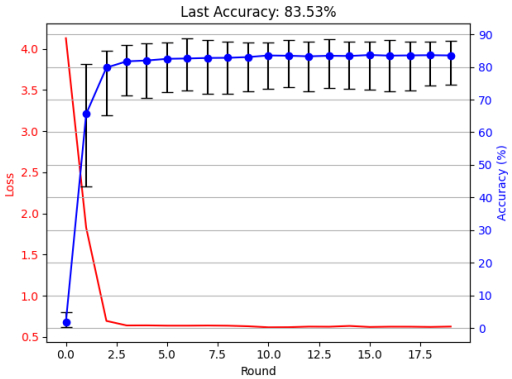
HR: 0.5
method: unify
lr: 0.01
#epochs: 6
optim: torch.optim.AdamW



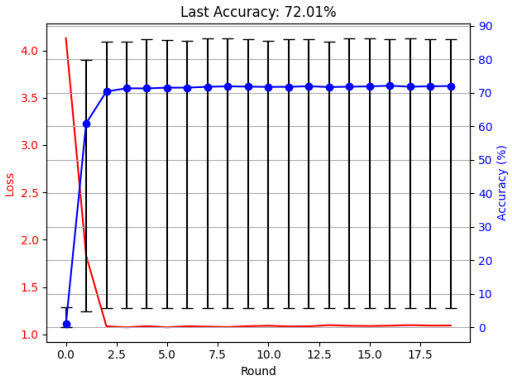
HR: 0.6
method: share-disjoint
lr: 0.01
#epochs: 6
optim: torch.optim.AdamW



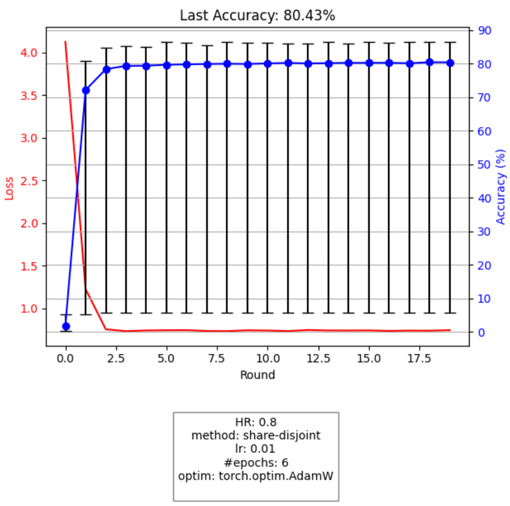
HR: 0.6
method: unify
lr: 0.01
#epochs: 6
optim: torch.optim.AdamW

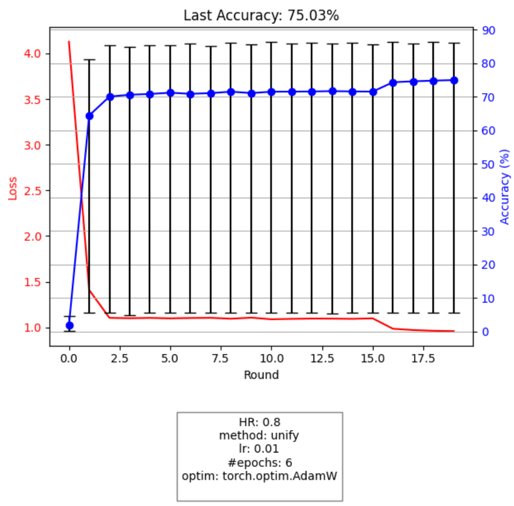


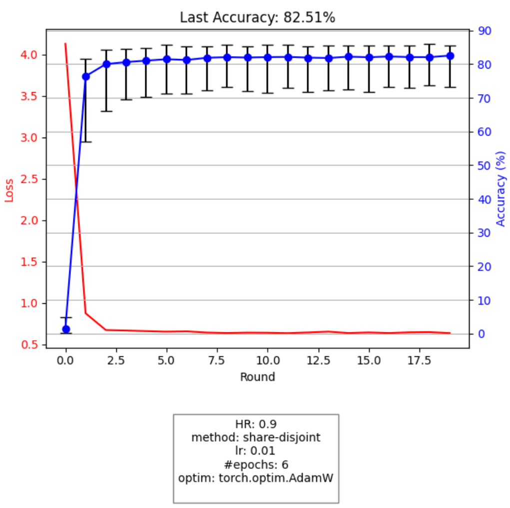
HR: 0.7
method: share-disjoint
lr: 0.01
#epochs: 6
optim: torch.optim.AdamW

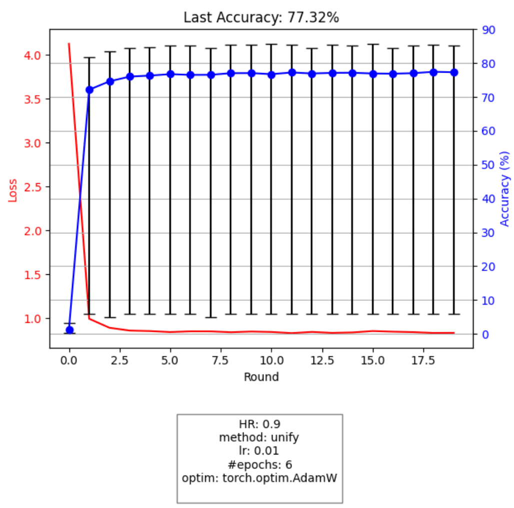


HR: 0.7
method: unify
lr: 0.01
#epochs: 6
optim: torch.optim.AdamW









../plots/femnist-horizontal/adamw/resu

5.2 HAR - HFL

In questa sezione vengono descritti gli esperimenti fatti con il dataset UCI HAR con il normale partizionamento orizzontale e vengono descritti i risultati.

5.2.1 Set up

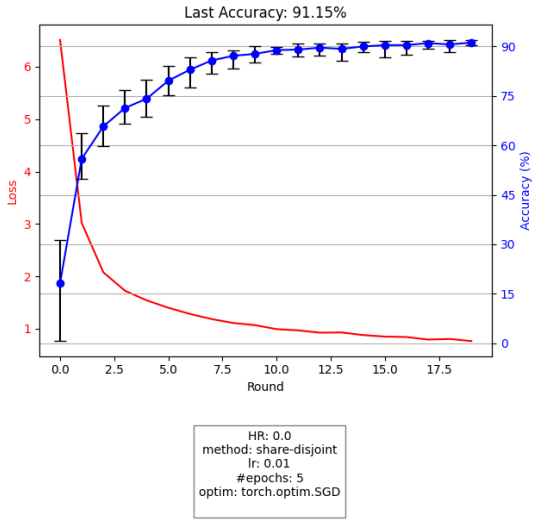
Per l'UCI HAR invece è stata utilizzata una semplice MLP con un solo hidden layer di 50 neuroni. Di nuovo, si sono provate le strategie di ibridazione `unify` e `share-disjoint` con gradi di ibridazione da 0% a 100% in passi di 10%. Anche qui sono stati testati sia l'ottimizzatore SGD che AdamW con learning rate $\eta = 0.01$ e momentum $\beta = 0.9$ nel caso di SGD. Il training ha usato 5 epoche locali, è durato 20 round e i risultati sono una media su 20 simulazioni consecutive. In questo caso è stato utilizzato l'intero dataset con 30 diversi client.

In esperimenti preliminari anche con questo dataset si è provato ad utilizzare SGD senza momentum e in questo caso il modello è stato in grado di convergere lo stesso, seppur rallentando prevedibilmente l'apprendimento. Sono stati provati anche valori diversi di learning rate come $\eta = 0.1$ o $\eta = 0.001$ ma non si sono viste differenze interessanti nell'apprendimento, se non una leggera instabilità maggiore o una velocità di apprendimento minore. Inoltre i primi modelli provati erano di dimensioni più grandi. Tuttavia, viste fin dall'inizio le ottime prestazioni ottenute su questo dataset si è ridotto il modello ad un unico layer nascosto di soli 50 neuroni.

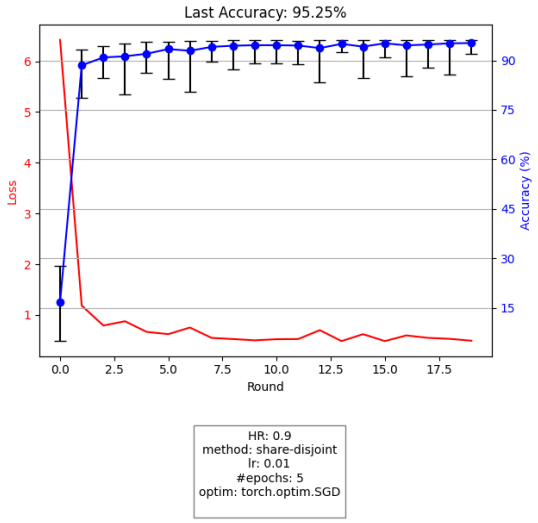
5.2.2 Risultati

L'UCI HAR è il dataset che ha mostrato le performance migliori. Innanzitutto nonostante la dimensione piuttosto contenuta del modello utilizzato si è riusciuti ad avere ottime prestazioni lo stesso, non scendendo mai al di sotto del 90% di accuracy e avvicinandosi al 95% per gli esperimenti con maggior grado di ibridazione. Le figure 5.26a e 5.26b mostrano i risultati ai due estremi di ibridazione. Come si può vedere anche in questo caso il modello centralizzato richiede solo 2 round per superare il 50% di accuracy e termina l'apprendimento intorno al quinto round. Il modello federato invece richiede ben più round di training, superando il 90% intorno al quindicesimo round e termina con un 91.15%.

Una differenza interessante rispetto al FEMNIST è che in questo caso AdamW si comporta sensibilmente meglio. Tanto per cominciare le performance dei modelli allenati con SGD sono perfettamente comparabili a quelle dei modelli allenati con AdamW, anziché performare peggio. Inoltre AdamW mostra una maggiore resistenza all'instabilità dell'apprendimento provocata dalla condivisione dei dataset



(a) 0% Condivisione

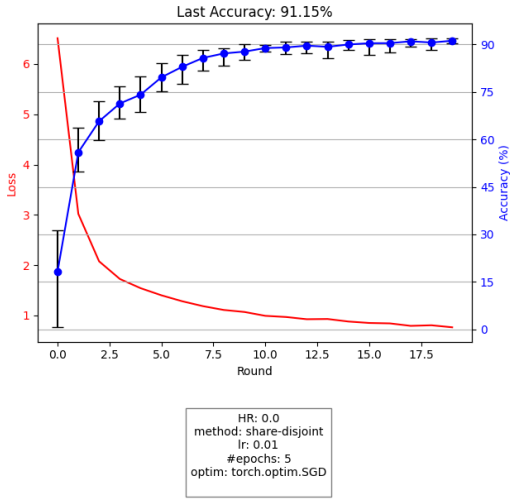


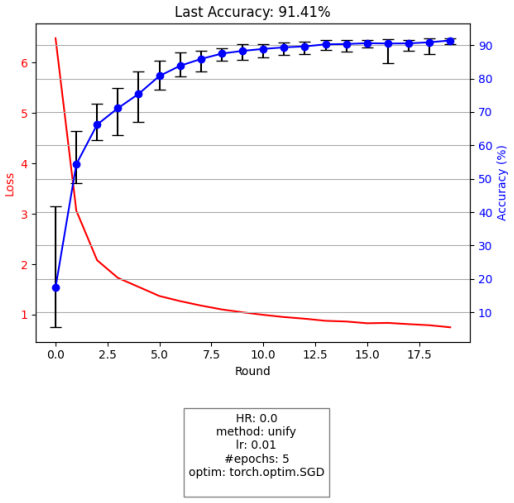
(b) 100% Condivisione

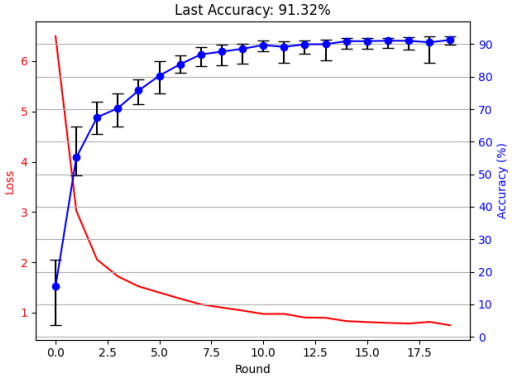
Figure 5.26: Risultati del training per ogni round. A sinistra (a) si può vedere l'apprendimento federato, a destra (b) quello centralizzato. Algoritmo di ottimizzazione (SGD) e metodo di condivisione (`share-disjoint`) sono tenuti fissati.

con il metodo `unify`. In figura **??** si può vedere il risultato del training con SGD, metodo `unify` al 60% di condivisione, mentre in figura **??** si vede il risultato del training sotto le stesse condizioni usando l'algoritmo AdamW. Come si può vedere dagli intervalli di incertezza AdamW risulta più stabile in queste condizioni.

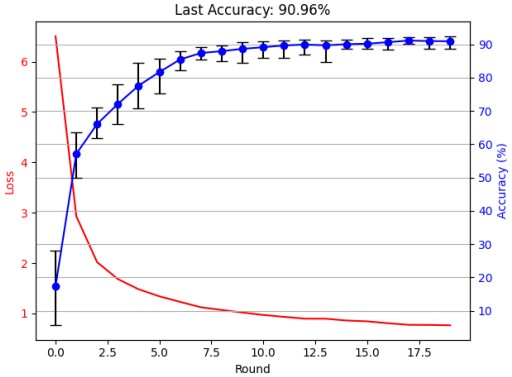
Anche per questo dataset, qui di seguito si trovano i grafici dei risultati di tutti gli esperimenti condotti. La colonna di sinistra riporta i risultati ottenuti con il metodo `share-disjoint`, quella di destra `unify`. Sono prima riportati i risultati di SGD.



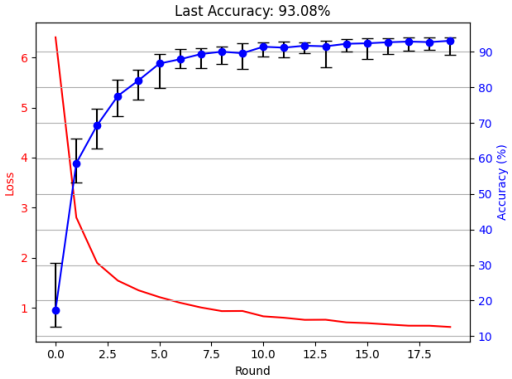




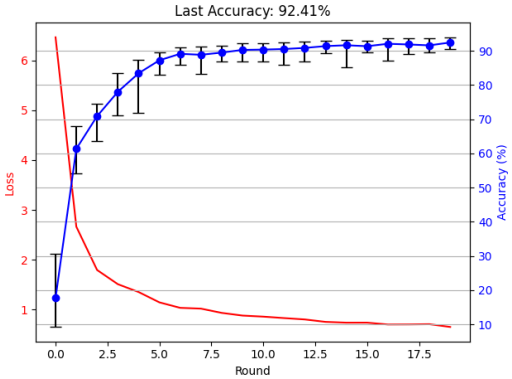
HR: 0.1
method: share-disjoint
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



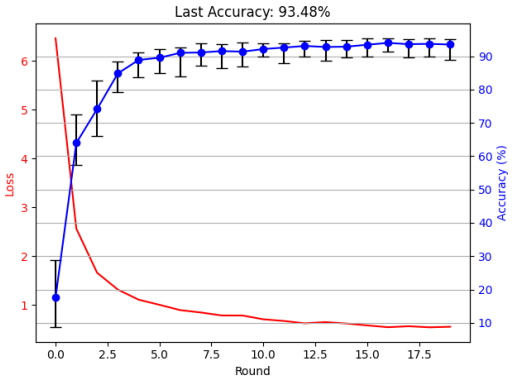
HR: 0.1
method: unify
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



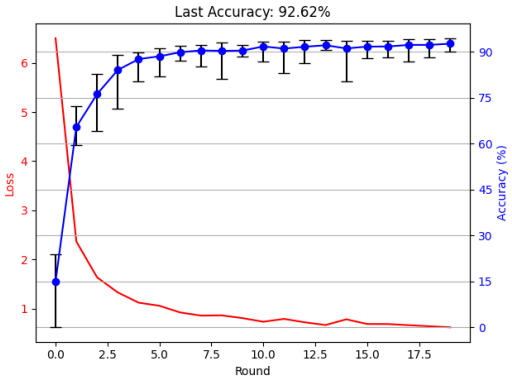
HR: 0.2
method: share-disjoint
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



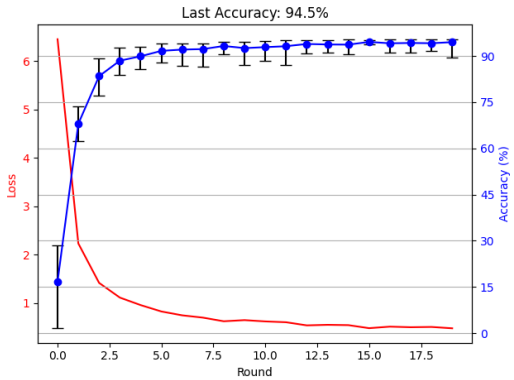
HR: 0.2
method: unify
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



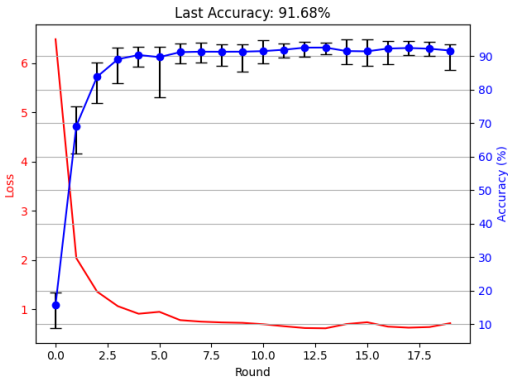
HR: 0.3
method: share-disjoint
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



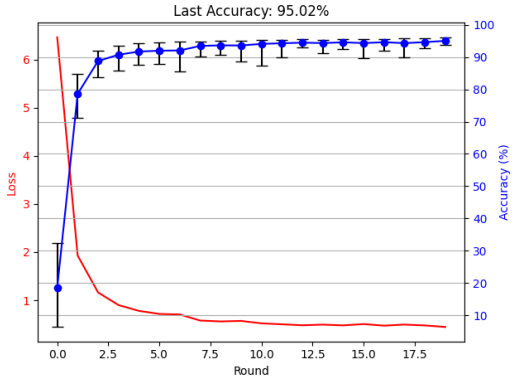
HR: 0.3
method: unify
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



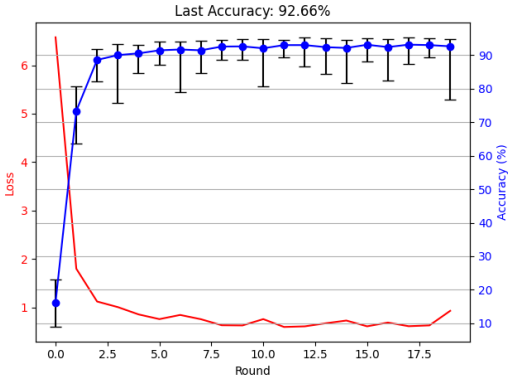
HR: 0.4
method: share-disjoint
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



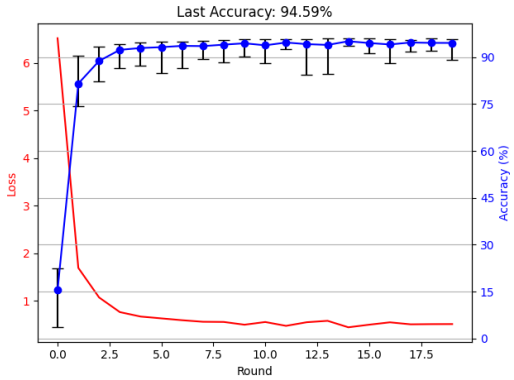
HR: 0.4
method: unify
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



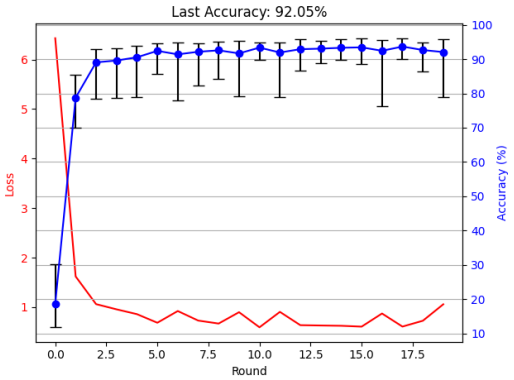
HR: 0.5
method: share-disjoint
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



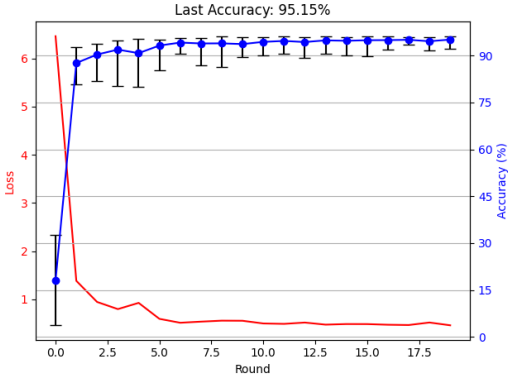
HR: 0.5
method: unify
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



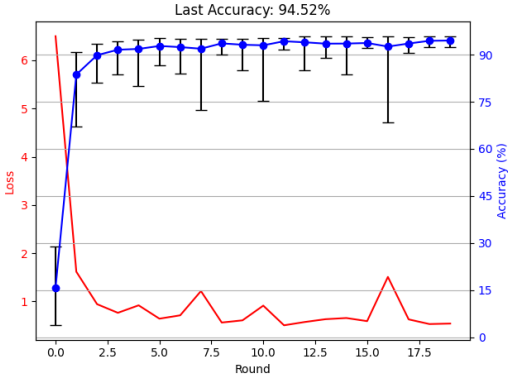
HR: 0.6
method: share-disjoint
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



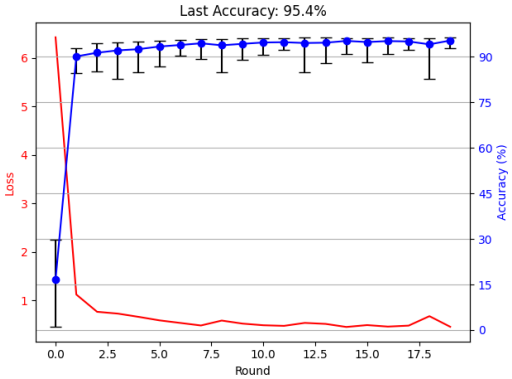
HR: 0.6
method: unify
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



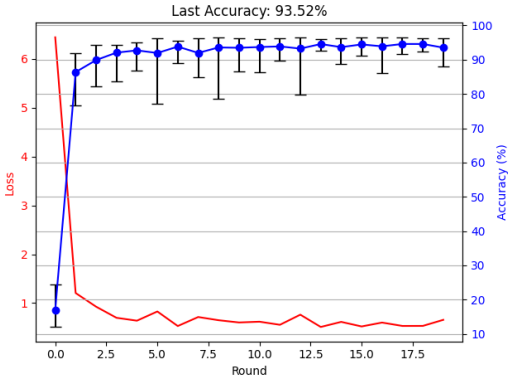
HR: 0.7
method: share-disjoint
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



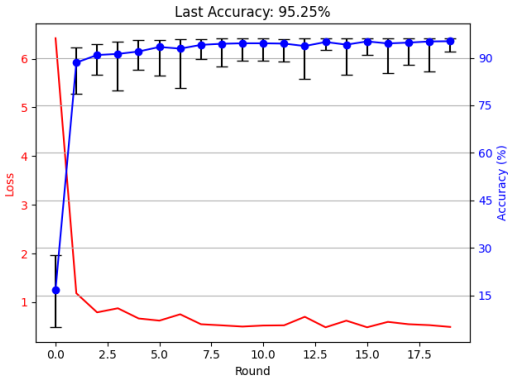
HR: 0.7
method: unify
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



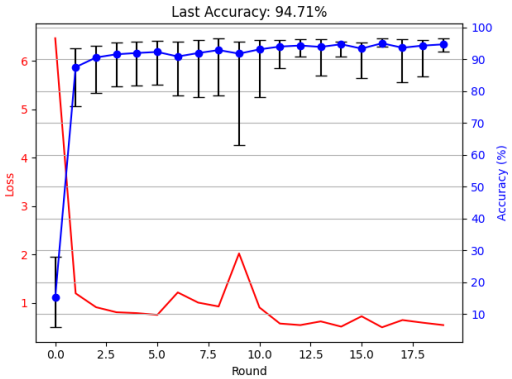
HR: 0.8
method: share-disjoint
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



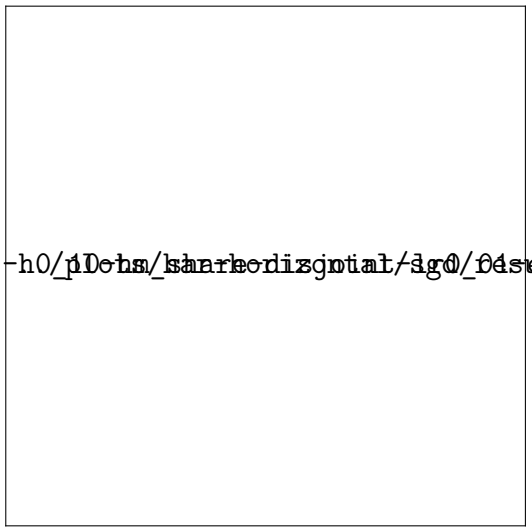
HR: 0.8
method: unify
lr: 0.01
#epochs: 5
optim: torch.optim.SGD



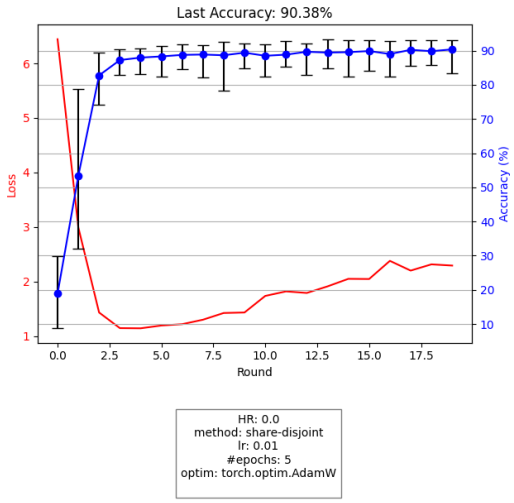
HR: 0.9
method: share-disjoint
lr: 0.01
#epochs: 5
optim: torch.optim.SGD

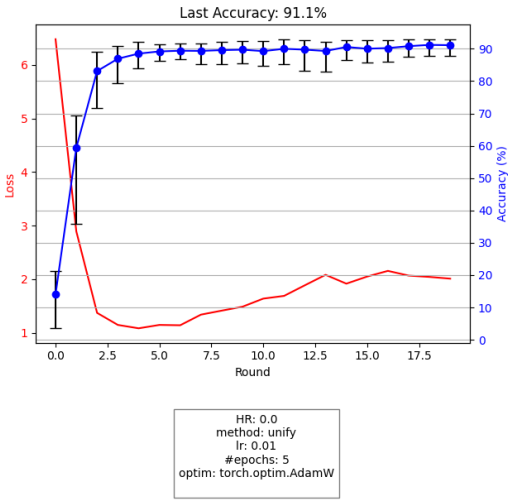


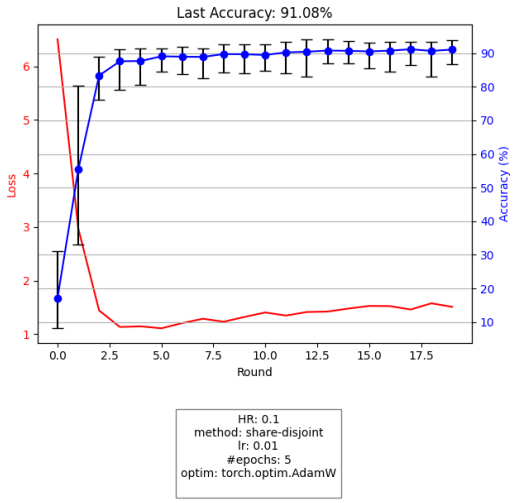
HR: 0.9
method: unify
lr: 0.01
#epochs: 5
optim: torch.optim.SGD

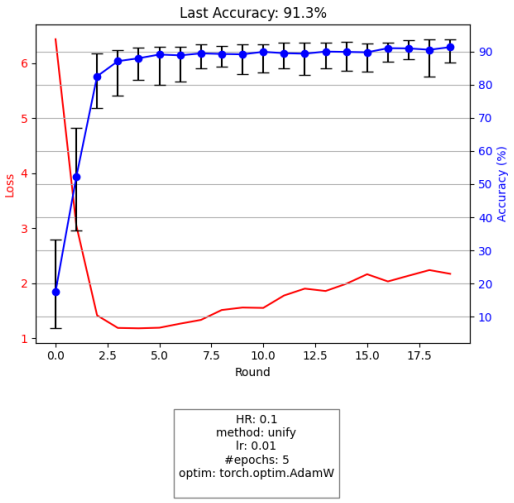

../plots/har-horizontal/sgd/results-h0/


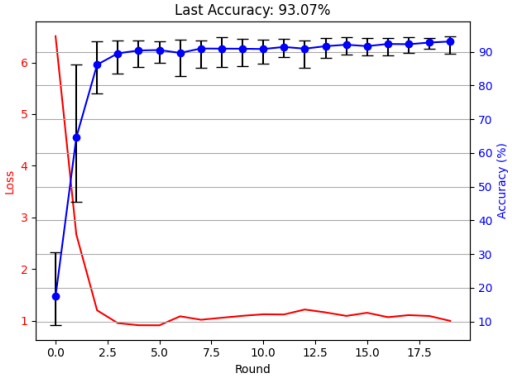
I seguenti sono i risultati di AdamW.



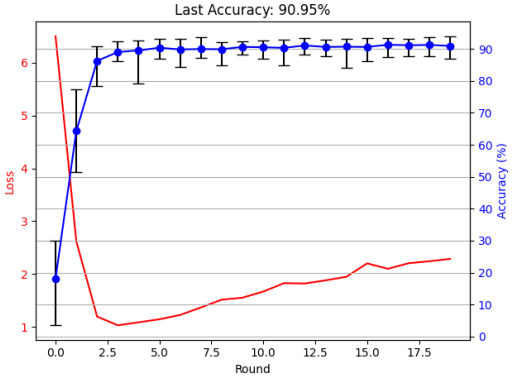




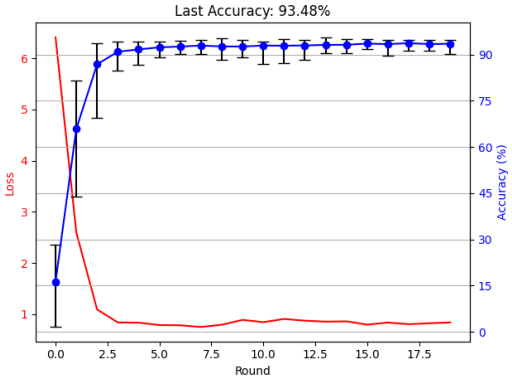




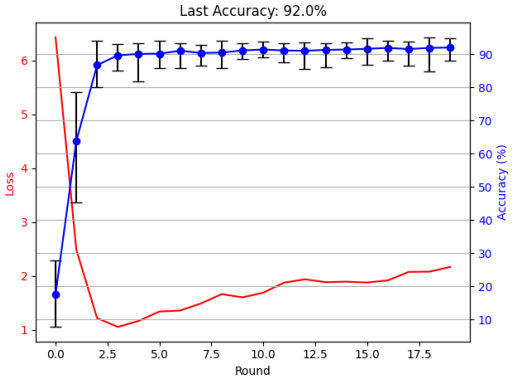
HR: 0.2
method: share-disjoint
lr: 0.01
#epochs: 5
optim: torch.optim.AdamW



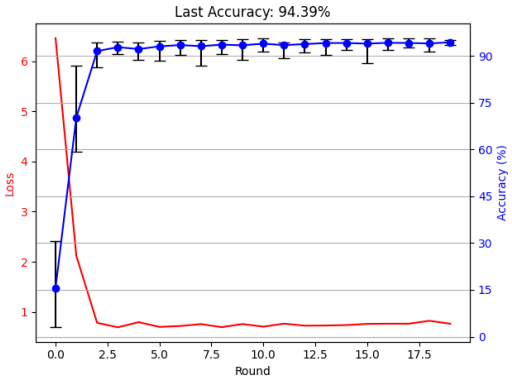
HR: 0.2
method: unify
lr: 0.01
#epochs: 5
optim: torch.optim.AdamW



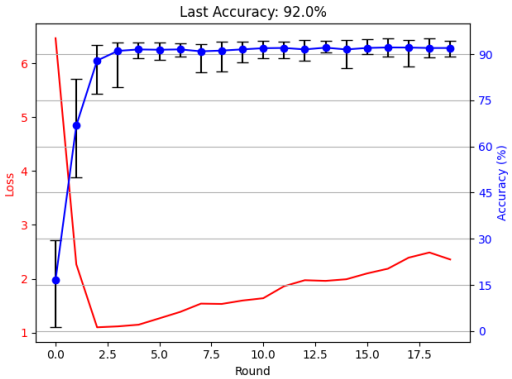
HR: 0.3
method: share-disjoint
lr: 0.01
#epochs: 5
optim: torch.optim.AdamW



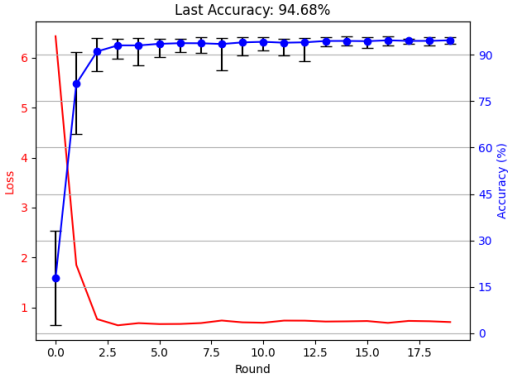
HR: 0.3
method: unify
lr: 0.01
#epochs: 5
optim: torch.optim.AdamW



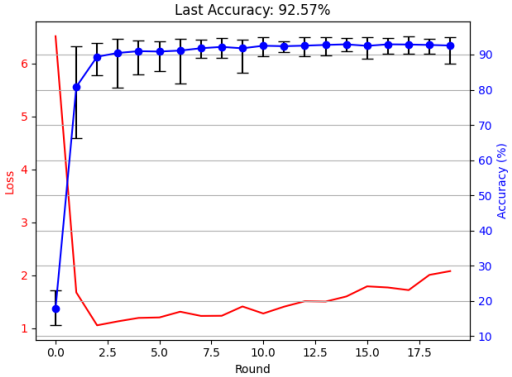
HR: 0.4
method: share-disjoint
lr: 0.01
#epochs: 5
optim: torch.optim.AdamW



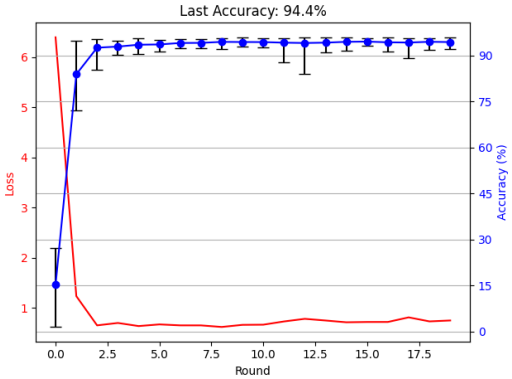
HR: 0.4
method: unify
lr: 0.01
#epochs: 5
optim: torch.optim.AdamW



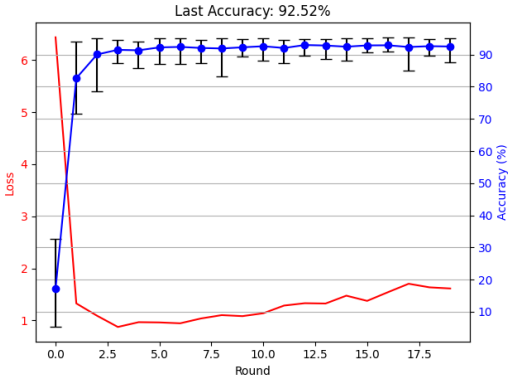
HR: 0.5
method: share-disjoint
lr: 0.01
#epochs: 5
optim: torch.optim.AdamW



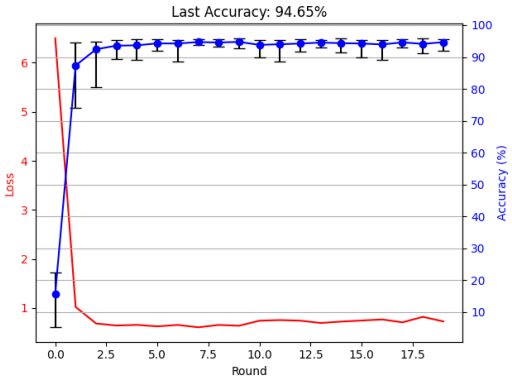
HR: 0.5
method: unify
lr: 0.01
#epochs: 5
optim: torch.optim.AdamW



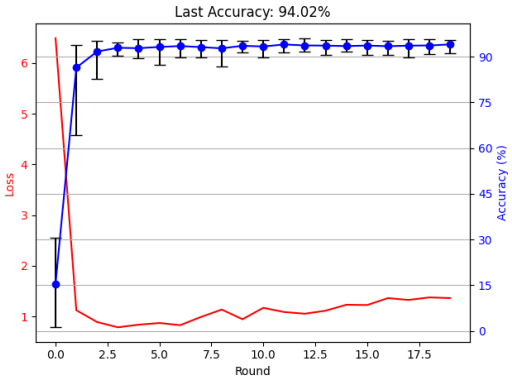
HR: 0.6
method: share-disjoint
lr: 0.01
#epochs: 5
optim: torch.optim.AdamW



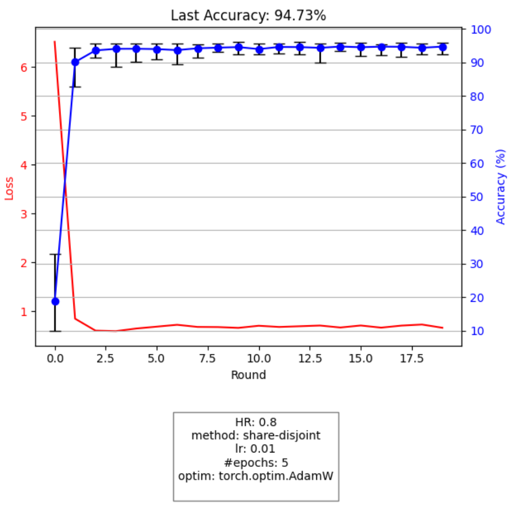
HR: 0.6
method: unify
lr: 0.01
#epochs: 5
optim: torch.optim.AdamW

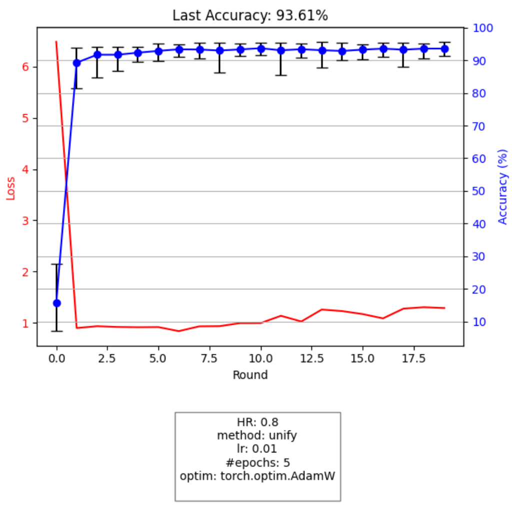


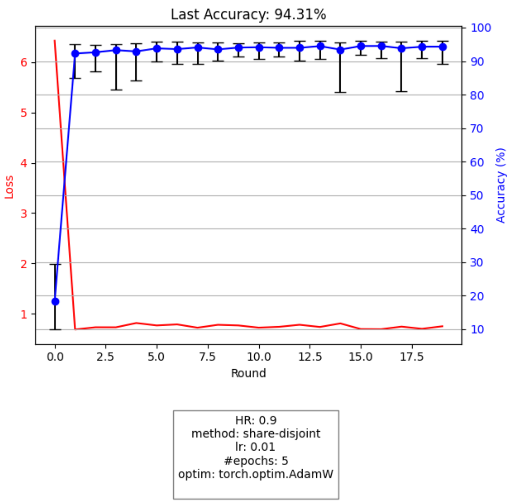
HR: 0.7
method: share-disjoint
lr: 0.01
#epochs: 5
optim: torch.optim.AdamW

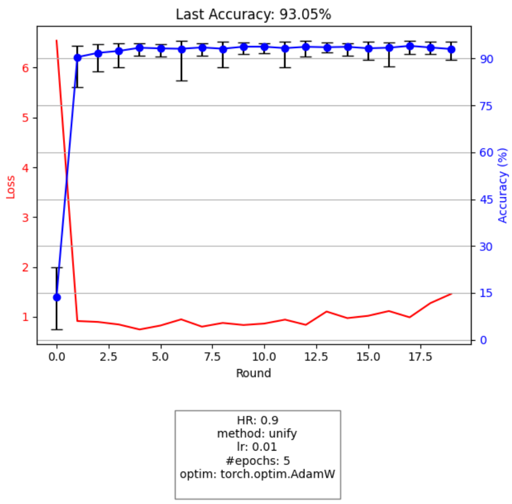


HR: 0.7
method: unify
lr: 0.01
#epochs: 5
optim: torch.optim.AdamW









../plots/har-horizontal/adam/results-h0_10-hr1_share-disjoint-adam-0.4-ef5-lr0.1 ../plots/har-horizontal/adam/results-h0_10-hr1_unify ...

5.3 HAR - VFL

L'ultimo esperimento fatto è quello del UCI HAR con partizionamento verticale delle feature.

In questo caso la rete neurale che compie la classificazione è distribuita tra client e server. Dato un numero di client K configurabile ogni ogni client ha un MLP che accetta $561/K$ feature (dove l'utlimo client prende le feature rimanenti in caso di resto diverso da 0) e ne calcola un encoding. Sono state provate due dimensioni diverse di encoding: su un vettore lungo 10 e lungo 100. Data la lunghezza l del vettore di encoding, il server ha un'altra MLP che accetta un vettore lungo Kl e ne calcola la classificazione. Sia il modello dei client che quello del server hanno un hidden layer di 50 neuroni.

I risultati di questo esperimento sono stati però deludenti: a discapito di quale ottimizzatore venisse utilizzato tra SGD e AdamW, del numero di client da 3 a 50 e della dimensione dell'encoding non si è mai riusciti ad ottenere un'accuracy superiore al 18%, rimanendo quindi appena sopra 1 / 6 = 16.7% cioè la probabilità di prenderci tirando a caso ogni volta su 6 classi. Si può quindi concludere che questo partizionamento delle feature su questo dataset non è un approccio funzionante.

5.4 Miglioramenti

Come già accennato, questo studio non ha l'obbiettivo di ottenere le migliori prestazioni possibili e come tale offre un ampio margine di miglioramento, specie nel caso in cui queste stesse tecniche vengano applicate a problemi più complessi.

Funzioni di attivazione

Un primo possibile miglioramento è quello di cambiare alcuni parametri della rete neurale. La funzione di attivazione ReLU, ad esempio, è estremamente popolare ed efficace nel mitigare il problema del vanishing gradient, ma anch'essa ha le sue limitazioni come il problema del *dying ReLU*. Alternative che possono essere esplorate sono la Leaky ReLU (LReLU) [26], definita come

$$LReLU(z) = \begin{cases} z, & \text{se } z > 0 \\ \alpha z, & \text{altrimenti} \end{cases}$$

che ammette valori diversi da 0 per argomenti negativi e introduce un altro iperparametro α, tipicamente piccolo, o la Parametric ReLU (PReLU) che rende

l'iperparametro α un parametro a_i da imparare per ogni neurone:

$$PReLU(z_i) = \begin{cases} z_i, & \text{se } z_i > 0 \\ a_i z_i, & \text{altrimenti} \end{cases}$$

Altre varianti della ReLU interessanti, in quanto funzioni liscie e differenziabili anche su 0, sono la Gaussian Error Linear Units (GELU) [17]

$$GELU(z) = zP(Z \leq z) = z\Phi(z)$$

o la Exponential Linear Unit (ELU) [11], che assume anche valori negativi fino a -1

$$ELU(z) = \begin{cases} z, & \text{se } z > 0 \\ \alpha(e^z - 1), & \text{altrimenti} \end{cases}$$

per un qualche iperparametro $\alpha < 0$.

Regolarizzazione

In questi esperimenti non si è fatto particolare uso di tecniche di normalizzazione o regolarizzazione, se non per una normalizzazione applicata alla immagini del FEMNIST ottenuta dalle trasformazioni built-in fornite da PyTorch. Si potrebbe allora provare alcune di queste tecniche come l'utilizzo della normalizzazione L_2 in caso si usi SGD (L_2 e weight decay sono equivalenti solo sotto SGD [25]) oppure una delle varie tecniche di normalizzazione che velocizzano il training riducendo l'effetto del *covariate shift*, come la batch normalization [19], la layer normalization [6] o la group normalization [37]

Scaling Up

Un altro metodo efficace per migliorare le performace di una rete neurale è quello di scalare le dimensioni del modello, dei dataset o del training. Operare con un'ampia mole di utenti e quindi con molte fonti di dati, far produrre più dati per il training ove possibile o rendere i modelli più complessi possono essere opzioni percorribili. Bisogna tenere in considerazione il fatto che modelli più grandi possono incappare nel problema dell'overfitting, richiedere più risorse computazionali, rischiando di tagliare fuori alcuni dispositivi meno performati, e aumenta l'utilizzo della banda di rete per la comunicazione di un maggior numero di parametri.

Inductive Bias migliori

Un'altra recente area di studio nel campo del deep learning è quella della ricerca di architetture che per come sono costruite implementano strutturalmente un'invarianza desiderata a qualche trasformazione dell'input. Un esempio concreto è l'invarianza traslazionale che implementano i layer convoluzionali attraverso la condivisione dei pesi. Il geometric deep learning [8] è lo studio di simmetrie (i.e. invarianti o equivarianti) geometriche che possono essere sfruttate nella progettazione di architetture neurali. Il FEMNIST è un dataset particolare in cui tutti i caratteri sono stati centrati e riscalati per ricoprire tutti la stessa area, cosa che in generale nella scrittura naturale è lontana dalla realtà. Può essere interessante esplorare architetture convoluzionali che siano invarianti anche a rotazioni o rescaling [31, 13, 27, 14].

Rimane però il fatto che le simmetrie desiderate dipendono fortemente dal problema trattato e per tale ragione vanno analizzate per lo specifico problema preso in considerazione.

Data Augmentation

Un altro metodo per *insegnare* ad un modello ad essere invariante a certe trasformazioni come la rotazione è quello del data augmentation. In questo caso quello che si può fare è produrre nuovi dati da fornire al modello nel training, facendo delle trasformazioni sui dati originali (come appunto rotare le immagini, aggiungere noise artificiale, invertire le immagini...) o anche producendo dati sintetici del tutto nuovi. Nell'era dei modelli generativi infatti, un'ulteriore possibilità è quella di far generare dei dati del tutto sintetici ad un modello generativo che abbiano la stessa distribuzione dei dati naturali.

Bisogna però tenere presente che questo approccio non è efficace nel risolvere il problema della *curse of dimensionality* [7, 18, 8], per cui il numero di data points nel training set scala esponenzialmente in funzione della dimensionalità dell'input della funzione che cerchiamo di apprendere.

Chapter 6

Conclusione

In questo studio si è potuta verificare l'efficacia del federated learning ibrido. La condivisione di una piccola parte dei dataset locali può essere una strategia efficacie per affrontare le difficoltà poste dal federated learning.

I risultati dimostrano che condividere una parte dei dataset ha un effetto positivo apprezzabile sulle performance del modello. Sono stati considerati due dataset di due problemi fondamentalmente diversi -UCI HAR e FEMNIST- a confermare il fatto che i risultati non siano una particolarità dello specifico problema o modello utilizzato. Il miglioramento di performance è stato particolarmente notevole nel FEMNIST che si è rivelato il dataset più complicato dei due.

Tuttavia rimane importante bilanciare in modo appropriato la condivisione di dati, con la protezione della privacy ove necessario e informare l'utente della condivisione in atto, comportamento virtuoso anche in assenza di legislazione specifica come il GDPR.

I due dataset diversi sono stati provati con diverse configurazioni: il modello è rimasto lo stesso, ma è stato allenato con diversi algoritmi di ottimizzazione, diversi gradi di condivisione dei dataset e diverse tecniche per la condivisione. I risultati mostrano che AdamW è un algoritmo tendenzialmente più instabile, che miglioramenti delle performance apprezzabili possono essere ottenuti con una condivisione di non più del 20% e che condividere pochi dati di ogni client è permette un miglioramento più sicuro sulle performance rispetto al selezionare alcuni client specifici.

In conclusione, questa tesi contribuisce alla crescente mole di letteratura nel machine learning, in generale, e federated learning, nello specifico, evidenziando il potenziale promettente di questo modello di apprendimento automatico. Lavori futuri potrebbero esplorare iperparametri diversi e più ottimali per provare ad allenare modelli *compute optimal* oppure studiare benchmark più complessi per performance *state of the art*.

Bibliography

[1] Flower. https://flower.ai/. Accessed: 2024-09-29.

[2] Hydra. https://hydra.cc/. Accessed: 2024-10-12.

[3] Pytorch. https://pytorch.org/. Accessed: 2024-09-29.

[4] D. Anguita, Alessandro Ghio, L. Oneto, Xavier Parra, and Jorge Luis Reyes-Ortiz. Human activity recognition using smartphones. https://archive.ics.uci.edu/dataset/240/human+activity+recognition+using+smartphones. Accessed: 2024-09-29.

[5] D. Anguita, Alessandro Ghio, L. Oneto, Xavier Parra, and Jorge Luis Reyes-Ortiz. A public domain dataset for human activity recognition using smartphones. In *The European Symposium on Artificial Neural Networks*, 2013. URL https://api.semanticscholar.org/CorpusID:6975432.

[6] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer Normalization. *arXiv: Learning*, 2016. URL https://doi.org/10.48550/arXiv.1607.06450.

[7] R. Bellman, Rand Corporation, and Karreman Mathematics Research Collection. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957. ISBN 9780691079516. URL https://books.google.it/books?id=wdtoPwAACAAJ.

[8] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges. *arXiv preprint arXiv:2104.13478*, 2021.

[9] Xiao Chenguang. Hdf5-femnist. https://github.com/Xiao-Chenguang/HDF5-FEMNIST, 2023. Accessed: 2024-09-29.

[10] Sebastian cladas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečný, H. Brendan McMahan, Virginia Smith, and Ameet Talwalkar. LEAF: A Benchmark for Federated Settings. 2019.

[11] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv: Learning*, 2016. URL https://doi.org/10.48550/arXiv.1511.07289.

[12] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre van Schaik. EMNIST: Extending MNIST to handwritten letters. 2017.

[13] Taco S. Cohen and Max Welling. Group Equivariant Convolutional Networks. *arXiv: Learning*, 2016. URL https://doi.org/10.48550/arXiv.1602.07576.

[14] Taco S. Cohen and Max Welling. Steerable CNNs. *arXiv: Learning*, 2016. URL https://doi.org/10.48550/arXiv.1612.08498.

[15] Flower. Flower tutorial | vertical federated learning using flower. https://www.youtube.com/watch?v=56-GvUaXKXo, 2023. Accessed: 2024-09-29.

[16] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Nets. 2014.

[17] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv: Learning*, 2016. URL https://doi.org/10.48550/arXiv.1606.08415.

[18] Gordon P. Hughes. On the mean accuracy of statistical pattern recognizers. *IEEE Trans. Inf. Theory*, 14:55–63, 1968. URL https://api.semanticscholar.org/CorpusID:206729491.

[19] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv: Learning*, 2015. URL https://doi.org/10.48550/arXiv.1502.03167.

[20] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G.L. D'Oliveira, Hubert Eichner, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaid Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konečný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Mariana Raykova, Hang Qi, Daniel Ramage, Ramesh Raskar, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha

Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Federated optimization in heterogeneous networks. *arXiv: Learning*, 2018. URL https://doi.org/10.48550/arXiv.1812.06127.

[21] Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank J. Reddi, Sebastian U. Stich, and Ananda Theertha Suresh. SCAFFOLD: Stochastic controlled averaging for federated learning. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5132–5143, 2020. URL https://proceedings.mlr.press/v119/karimireddy20a.html.

[22] Yann LeCun. The mnist database of handwritten digits. https://yann.lecun.com/exdb/mnist/, 1998. Accessed: 2024-09-28.

[23] Tian Li, Anit Kumar Sahu, Maziar Sanjabi, Manzil Zaheer, Ameet Talwalkar, and Virginia Smith. Federated optimization in heterogeneous networks. *arXiv: Learning*, 2018. URL https://doi.org/10.48550/arXiv.1812.06127.

[24] Yang Liu, Yan Kang, Tianyuan Zou, Yanhong Pu, Yuanqin He, Xiaozhou Ye, Ye Ouyang, Ya-Qin Zhang, and Qiang Yang. Vertical federated learning: Concepts, advances and challenges. 2022. URL https://doi.org/10.48550/arXiv.2211.12814.

[25] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. 2017. URL https://doi.org/10.48550/arXiv.1711.05101.

[26] Andrew L. Maas. Rectifier nonlinearities improve neural network acoustic models. 2013. URL https://api.semanticscholar.org/CorpusID:16489696.

[27] Diego Marcos, Michele Volpi, Nikos Komodakis, and Devis Tuia. Rotation equivariant vector field networks. *arXiv: Learning*, 2016. URL https://doi.org/10.48550/arXiv.1612.09346.

[28] H. B. McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *International Conference on Artificial Intelligence and Statistics*, 2016. URL https://api.semanticscholar.org/CorpusID:14955348.

[29] Jorge Luis Reyes Ortiz. Activity recognition experiment using smartphone sensors. https://www.youtube.com/watch?v=XOEN9W05_4A, 2012. Accessed: 2024-09-29.

[30] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018. URL https://api.semanticscholar.org/CorpusID:49313245.

[31] I. Sosnovik. Symmetry-based learning from limited data. *University of Amsterdam*, 2023. URL https://hdl.handle.net/11245.1/fb28dcd6-bfa6-4afd-b35e-97d1b156bfbc.

[32] TalwalkarLab. Leaf: A benchmark for federated settings. https://github.com/TalwalkarLab/leaf, 2019. Accessed: 2024-09-29.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. 2017.

[34] Kang Wei, Jun Li, Chuan Ma, Ming Ding, Sha Wei, Fan Wu, Guihai Chen, and Thilina Ranbaduge. Vertical federated learning: Challenges, methodologies and experiments. 2022. URL https://doi.org/10.48550/arXiv.2202.04309.

[35] Wikipedia. Filtro butterworth. https://it.wikipedia.org/wiki/Filtro_Butterworth. Accessed: 2024-09-29.

[36] Papers with Code. Femnist (federated extended mnist). https://paperswithcode.com/dataset/femnist, 2024. Accessed: 2024-09-29.

[37] Yuxin Wu and Kaiming He. Group Normalization. *arXiv: Learning*, 2018. URL https://doi.org/10.48550/arXiv.1803.08494.

[38] C. Xiao and S. Wang. An experimental study of class imbalance in federated learning. 2021. URL https://doi.org/10.48550/arXiv.2109.04094.

[39] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. 2019. URL https://doi.org/10.48550/arXiv.1902.04885.

[40] Yue Zhao, Meng Li, Liangzhen Lai, Naveen Suda, Damon Civin, and Vikas Chandra. Federated learning with non-iid data. 2018. URL https://doi.org/10.48550/arXiv.1806.00582.