

**ALMA MATER STUDIORUM - UNIVERSITÀ
DI BOLOGNA**

Dipartimento di Informatica — Scienza e Ingegneria

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea
in
Tecnologie Web - T

Training di Reti Neurali da Dati Decentralizzati

Autore:
Samuele Gasbarro

Relatore:
Chiar.mo Prof.
Paolo Bellavista

Correlatore:
Dr. Alessio Mora

III Sessione
Anno Accademico 2023 - 2024

Sommario

Introduzione	2
1 Concetti Generali	3
1.1 Machine Learning	3
1.2 Supervised training	4
1.2.1 Funzionamento del supervised training	4
1.3 Label e classi	5
1.4 Classification task	6
1.4.1 Funzionamento di un classification task	6
1.5 Reti Neurali	7
1.5.1 Perceptron	7
1.5.2 Sigmoid neuron	8
1.5.3 Reti Neurali Feedforward	9
1.6 Loss Function	11
1.6.1 Esempio di loss function	12
1.7 Gradient Descent	13
1.7.1 Stochastic Gradient Descent	14
1.8 Backpropagation	15
1.8.1 Backpropagation algorithm	16
1.9 Convolutional Neural Network	17
2 Progetto	19
2.1 Federated Learning	19
2.2 Federated Averaging Algorithm	19
2.3 Obiettivi	20
2.4 Implementazione	22
2.5 Librerie utilizzate	22
2.6 Partizionamento del dataset	22
2.7 Funzione per mostrare la suddivisione del dataset tra client	24
2.8 Caricamento del dataset	26
2.9 Funzione per l'aggiornamento locale dei client	27
2.10 Avvio della simulazione	28
3 Risultati sperimentali	32
3.1 IID	33
3.2 Distribuzione classi per client	35
3.3 Confronto	37
Conclusioni	40
Bibliografia	43

Introduzione

Negli ultimi anni, l’Intelligenza Artificiale ha rivoluzionato molteplici settori della società, migliorando servizi, prodotti e processi decisionali, quali la medicina, i trasporti, l’industria e la finanza. Al centro di questa trasformazione ci sono le reti neurali, modelli computazionali ispirati alla struttura del cervello umano, capaci di apprendere e generalizzare da grandi volumi di dati. Tuttavia, la necessità di gestire dati su larga scala ha sollevato molte domande riguardo alla privacy, alla sicurezza e alla distribuzione delle informazioni, specialmente quando i dati provengono da dispositivi personali o sensibili, come smartphone e sensori sanitari.

In questo scenario, l’allenamento delle reti neurali utilizzando dati decentralizzati sta emergendo come un nuovo paradigma per sviluppare modelli di IA che rispettano la privacy. Questo approccio si basa su tecniche che consentono di addestrare modelli distribuiti mantenendo i dati sui dispositivi locali, riducendo così la necessità di trasferire informazioni sensibili verso server centrali. Il Federated Averaging Algorithm (FedAvg), sviluppato da Google, è uno dei metodi più innovativi in questo campo. Grazie a questo algoritmo, è possibile combinare i parametri dei modelli addestrati localmente, ottenendo un modello globale aggiornato senza mai centralizzare i dati.

Tale tesi, si propone dunque di esplorare le basi teoriche e pratiche dell’allenamento di reti neurali da dati decentralizzati, approfondendo in particolare le reti neurali e l’algoritmo di Federated Averaging. Saranno analizzati i principali vantaggi e le sfide di questo approccio, con un focus sulla gestione della privacy, sull’efficienza computazionale e sulla robustezza dei modelli risultanti.

1 Concetti Generali

Questo capitolo è focalizzato sull'introduzione generica di alcuni concetti fondamentali per facilitare una piena comprensione degli argomenti trattati nei capitoli successivi. Verranno presentate le teorie e le nozioni essenziali, evitando di entrare in dettagli eccessivamente specifici.

1.1 Machine Learning

Il **Machine Learning** (ML) è una branca dell'Intelligenza Artificiale che si occupa di creare algoritmi e modelli che apprendono o migliorano le performance nel tempo utilizzando dati piuttosto che istruzioni programmate esplicite. In altre parole, il machine learning permette ai computer di riconoscere schemi, fare previsioni o prendere decisioni basandosi su esperienza pregressa, senza richiedere una programmazione manuale per ogni specifico compito. Bisogna sottolineare che il machine learning rappresenta solo una parte dell'AI, nonostante siano termini che vengono spesso utilizzati insieme e in modo interscambiabile. Sebbene il concetto di ML nacque indicativamente intorno agli anni '50, solo nell'ultimo decennio abbiamo visto questa crescita esponenziale del suo utilizzo in concomitanza con l'aumento a dismisura di dati forniti principalmente dai dispositivi mobili (PC, tablet, smartphone). I potenti sensori accessi su questi dispositivi (inclusi fotocamere, microfoni e GPS), combinati con il fatto che vengono trasportati frequentemente, come anche l'uso massivo di social network e strumenti di messaggistica, portano ad avere accesso a una quantità di dati senza precedenti. Una volta reperiti questi dati compito del ML è quello di sfruttare schemi ad hoc per poter ricavare pattern e modelli sui quali fare le nostre valutazioni. Può essere applicato in numerosi ambiti, come riconoscimento di immagini, elaborazione del linguaggio naturale, analisi predittiva, automazione e molto altro. Si può dividere il machine learning in tre categorie:

- **Supervised learning:** definito dall'uso di dataset etichettati per formare algoritmi che classificano i dati o prevedono i risultati in modo accurato. Man mano che i dati di input vengono inseriti nel modello, questo regola i suoi pesi fino a quando non vengono adattati in modo appropriato. Esempi di utilizzo possono essere rilevazioni di email spam/ non spam o per la diagnosi di una malattia.
- **Unsupervised learning:** vengono utilizzati algoritmi di apprendimento automatico per analizzare e raggruppare dataset non etichettati cercando schemi o strutture nascoste. Può essere utilizzato per identificare transazioni bancarie anomale rispetto ai comportamenti standard, senza che un evento di frode sia già etichettato, oppure per operazioni di clustering.
- **Reinforcement learning:** il modello apprende attraverso interazioni con un ambiente, ricevendo ricompense o penalità in base alle azioni intraprese. Si

utilizza spesso in robotica o anche nel campo della guida autonoma.

Concludendo si può dire che il machine learning è una tecnologia chiave di questo periodo che abilita la creazione di sistemi intelligenti in grado di apprendere e adattarsi autonomamente, rendendo possibili innovazioni in svariati settori. Inoltre lo sviluppo di altre applicazioni fondamentali dell'intelligenza artificiale quali le **reti neurali** e il **deep learning** derivano proprio dal machine learning.

1.2 Supervised training

I modelli di machine learning vengono creati addestrando algoritmi con dati etichettati, non etichettati o una combinazione di entrambi. Il **supervised training**, noto anche come *supervised learning*, è una sottocategoria dell'apprendimento automatico e dell'intelligenza artificiale. È definito dall'uso di set di dati etichettati per addestrare algoritmi in grado di classificare i dati o prevedere i risultati in modo accurato [2].

L'apprendimento supervisionato utilizza un set di formazione per insegnare ai modelli a produrre l'output desiderato. Questo set di dati di addestramento include vettori di input x e output y corretti, che consentono al modello di apprendere nel tempo, solitamente valutando $p(y|x)$. L'algoritmo misura la sua precisione attraverso la funzione di perdita, aggiustando l'errore finché non è stato sufficientemente ridotto al minimo [2].

Possiamo suddividere l'apprendimento supervisionato in due campi di applicazione:

1. **Classification:** tipologia di problema in cui l'obiettivo è predire una categoria o classe a cui un'osservazione appartiene, basandosi su dati di input. Entreremo in seguito nel dettaglio in quanto verrà affrontato nella parte progettuale.
2. **Regression:** si occupa di predire un valore numerico continuo in base ai dati di input.

Un modello di rete neurale basato sul supervised learning utilizza dati etichettati (input associati agli output attesi) per apprendere. Durante l'addestramento, il modello confronta le sue previsioni con gli output corretti utilizzando una funzione di perdita (loss function).

1.2.1 Funzionamento del supervised training

Il funzionamento del supervised training può essere schematizzato, in linea di massima, nel seguente modo:

1. **Preparazione del Dataset:** Il primo passo è la raccolta e la preparazione dei dati di addestramento, che devono essere etichettati, cioè ogni input deve avere un output associato.

2. Divisione del Dataset:

- Training set: impiegato per addestrare il modello.
- Validation set (opzionale): utilizzato per sintonizzare i parametri del modello.
- Test set: il cui fine è verificare le prestazioni del modello finale.

3. **Addestramento del Modello:** durante questa fase, il modello analizza il training set per imparare le associazioni tra input e output. Questo avviene ottimizzando i parametri interni del modello (come i pesi in una rete neurale) in modo che l'errore tra la previsione del modello e l'etichetta corretta sia minimizzato. Questa ottimizzazione è spesso fatta utilizzando algoritmi come il Gradient Descent.

4. **Valutazione del Modello:** il modello addestrato viene testato sul validation set (se presente) e poi sul test set. Questo serve a valutare la capacità del modello di generalizzare, cioè di fare previsioni accurate su dati non visti durante l'addestramento.

5. **Tuning e Iterazione:** dopo la prima valutazione, spesso si eseguono iterazioni di hyperparameter tuning (aggiustamento dei parametri del modello) e retraining per migliorare le performance del modello.

6. **Predizione:** una volta addestrato, quest'ultimo risulta pronto per fare previsioni su nuovi dati, ovvero su input non etichettati. In tal modo si ottiene il risultato pratico dell'addestramento supervisionato.

Si è voluto entrare nel dettaglio della trattazione per il supervised training perché sarà la tipologia di machine learning che utilizzeremo in fase sperimentale, evitando la creazione di un validation set ma unendo i vari passaggi sopra descritti con la tecnica delle reti neurali.

1.3 Label e classi

Prima di proseguire è bene dare una definizione precisa a quelli che sono due elementi cardine del problema di supervised training che affronteremo.

La **label** è l'etichetta associata a un dato in input nei dataset. Rappresentano ciò che il modello cerca di imparare durante il supervised training e sono componente fondamentale per questo tipo di apprendimento. Servono come guida durante l'addestramento.

Le **classi** sono raggruppamenti di label, che consentono la classificazione dei dati.

1.4 Classification task

Un **classification task** è un problema di apprendimento supervisionato in cui l’obiettivo è assegnare ogni dato in input a una delle diverse categorie predefinite, chiamate **classi**. Il modello viene addestrato utilizzando un set di dati etichettati (training set) per imparare a prevedere l’etichetta (label) corretta dei dati. Successivamente, viene valutato su un set di dati separato (test set) prima di essere impiegato per fare previsioni su nuovi dati non visti.

Per risolvere un task di classificazione, all’algoritmo di apprendimento viene chiesto di produrre una funzione $f : R^n \rightarrow 1, \dots, k$, con k che rappresenta il numero di classi. Quando $y = f(\mathbf{x})$ il modello assegna l’input descritto dal vettore x alla categoria identificata dal codice numerico y .

Un esempio di classification task è il riconoscimento di oggetti, in cui l’input è un’immagine e l’output è un codice numerico che identifica l’oggetto nell’immagine.

1.4.1 Funzionamento di un classification task

1. **Dataset etichettato:** il modello viene addestrato con un dataset in cui ogni esempio ha una label associata, che rappresenta la classe a cui quell’esempio appartiene.
2. **Estrazione delle caratteristiche:** durante l’addestramento, il modello apprende le caratteristiche dei dati che sono rilevanti per distinguere tra le varie classi.
3. **Assegnazione della classe:** dopo l’addestramento, dato un nuovo input non etichettato, il modello predice la classe a cui l’input probabilmente appartiene.

In seguito verrà realizzato un esempio di classification task sul database *Fashion MNIST* in cui sfruttando le reti neurali come modello saremo in grado di classificare in modo più che soddisfacente le immagini sul test set.

1.5 Reti Neurali

Una rete neurale è un modello computazionale ispirato alla struttura e al funzionamento dei neuroni nel cervello umano, progettato per elaborare informazioni e risolvere problemi complessi.

1.5.1 Perceptron

Per poter comprendere pienamente il funzionamento delle attuali reti neurali moderne è utile parlare dei *perceptrons*, l'unità di base nelle reti neurali artificiali che simula il comportamento di un neurone biologico.

Il **perceptron** opera in modo semplice: assegnati una serie di input x_1, x_2, \dots, x_n , che hanno valore binario: 0 o 1, viene prodotto un output, anch'esso binario. Ad ogni input viene associato un **peso** w_1, w_2, \dots, w_n , ovvero un numero reale che rappresenta l'importanza dell'input. L'output del neurone, è determinato valutando se la somma ponderata $\sum_j w_j x_j$ sia maggiore o minore del valore di una soglia stabilita.

$$\text{output} = \begin{cases} 0 & \text{se } \sum_j w_j x_j \leq \text{soglia} \\ 1 & \text{se } \sum_j w_j x_j > \text{soglia} \end{cases} \quad (1)$$

È però possibile semplificare la descrizione, pensando alla somma ponderata come un prodotto scalare:

$$w \cdot x = \sum_j w_j x_j \quad (2)$$

dove w e x sono i vettori che rappresentano rispettivamente i pesi e gli input. Inoltre, invece di usare una soglia separata, è possibile introdurre un parametro aggiuntivo chiamato **bias**, definito come $b = -\text{soglia}$. In questo modo, la formula diventa:

$$z = w \cdot x + b = \sum_j w_j x_j + b \quad (3)$$

Il valore z viene poi passato attraverso una funzione di attivazione, in questo caso una funzione a gradino. L'output del perceptron è quindi:

$$\text{output} = \begin{cases} 0 & \text{se } z \leq 0 \\ 1 & \text{se } z > 0 \end{cases} \quad (4)$$

Il bias può essere interpretato come una misura di quanto sia facile per il perceptron attivarsi (cioè produrre un output uguale a 1). Esso permette al modello di adattarsi meglio ai dati, spostando il punto di separazione tra le classi.

I perceptroni possono anche essere utilizzati per calcolare funzioni logiche elementari come AND, OR e NAND, poiché sono in grado di apprendere relazioni lineari tra gli input.

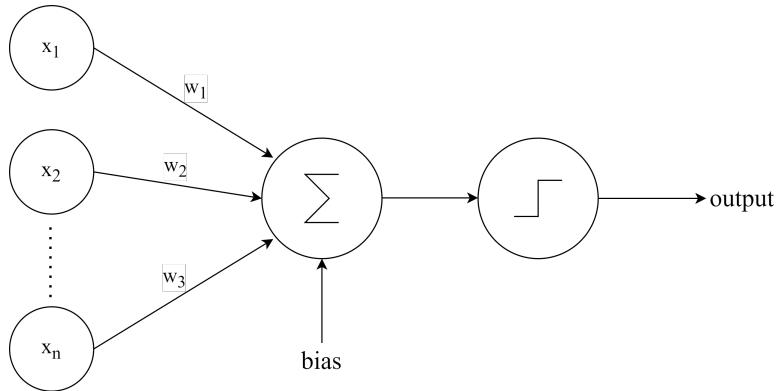


Figure 1: Perceptron

1.5.2 Sigmoid neuron

In molti casi, vorremo poter effettuare piccole modifiche ai pesi e ai bias di un neurone per migliorare gradualmente l'output della rete, permettendo al modello di apprendere attraverso un processo continuo di ottimizzazione. Tuttavia, nel perceptron, piccole variazioni nei pesi o nel bias possono cambiare radicalmente l'output, limitando la sua capacità di apprendere efficacemente. Per risolvere questo problema, è stato introdotto il **sigmoid neuron**, che utilizza una funzione di attivazione più morbida, permettendo di ottenere variazioni graduali nell'output. Il funzionamento di perceptron e sigmoid neuron è simile, ma ci sono differenze significative:

- la prima relativa ai valori degli input, che nel caso del sigmoid neuron sono *compresi* tra 0 e 1, e non hanno più un valore binario;
- la seconda relativa all'*activation function* σ con la quale calcoliamo l'output, non più rappresentata da una funzione a gradino ma adesso definita come:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}} = \frac{1}{1 + \exp(-(\sum_j w_j x_j + b))} \quad (5)$$

Questa funzione approssima un comportamento a gradino per valori estremi di z , ma introduce una risposta graduale per valori intermedi.

La somiglianza tra i due neuroni emerge considerando il comportamento della funzione sigmoide per valori estremi di $\sum_j w_j x_j + b$:

- se tale valore è molto grande (tendente a $+\infty$), l'esponenziale nel denominatore si riduce praticamente a zero, e l'output tende a 1: $z \rightarrow \infty, \sigma(z) \rightarrow 1$.
- Al contrario, se il valore è molto piccolo (tendente a $-\infty$), l'esponenziale domina e l'output si avvicina a 0: $z \rightarrow -\infty, \sigma(z) \rightarrow 0$

In questi casi estremi, il comportamento del sigmoid neuron assomiglia a quello del perceptron.

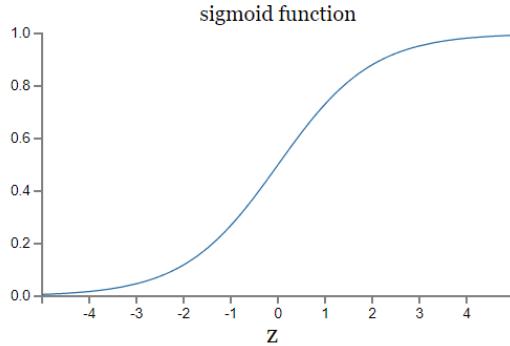


Figure 2: Sigmoid function [7]

Come si può vedere dalla Figura 2 la funzione di output è leggermente "stondata" e non più a gradino. Questo è di fondamentale importanza e ci permette di capire come i piccoli cambiamenti di peso Δw_j e bias Δb portano a piccoli cambiamenti dell'output Δoutput , come mostrato anche dalla seguente equazione:

$$\Delta \text{output} \approx \sum_j \frac{\delta \text{output}}{\delta w_j} \Delta w_j + \frac{\delta \text{output}}{\delta b_j} \Delta b \quad (6)$$

L'uso dei sigmoid neuron garantisce una maggiore flessibilità nell'apprendimento e una migliore interpretazione dell'output, rendendolo una componente chiave nelle moderne reti neurali.

1.5.3 Reti Neurali Feedforward

Dopo aver parlato degli elementi fondamentali delle reti neurali adesso ci concentreremo in particolare sulle reti neurali **feedforward**, ossia un modello che ha come obiettivo quello di approssimare una certa funzione f^* , che descrive la relazione ideale tra un input e un output. In sostanza, le reti neurali feedforward sono costituite da più neuroni organizzati in strati, con le informazioni che si propagano in una sola direzione, dall'input all'output, con quest'ultimo ottenuto dalla composizione delle funzioni calcolate da ciascun strato.

La rete feedforward costruisce una mappatura $y = f(x; \theta)$, dove:

- f rappresenta la funzione calcolata dalla rete;
- x sono gli input;
- θ è l'insieme dei parametri della rete ottimizzati durante l'addestramento. Nel nostro caso rappresentato da pesi w e bias b ;

- y è l'output: risultato della mappatura dell'input attraverso la rete.

L'obiettivo dell'addestramento è trovare i parametri θ che fanno sì che $f(x; \theta)$ si avvicini il più possibile a $f^*(x)$, cioè l'approssimazione della funzione ideale.

Le reti feedforward sono caratterizzate dall'assenza di retroazioni: le informazioni si muovono in un'unica direzione, dagli input agli output, senza feedback tra gli strati.

La struttura di una rete neurale consiste di strati (**layers**) costituiti da nodi o neuroni (come perceptron o sigmoid neuron). La lunghezza della catena di strati determina la profondità (**deep**) del modello. Tipicamente, questi strati sono organizzati come segue:

1. **input layer**: riceve i dati grezzi che la rete neurale deve elaborare;
2. **hidden layers**: elaborano l'informazione tramite una serie di pesi e funzioni di attivazione, che trasformano i dati di input in informazioni elaborate. Sono definiti "nascosti" perché non abbiamo a disposizione informazioni su quale output dovrebbero generare. Pertanto, il compito dell'algoritmo di apprendimento è quello di regolare questi strati in modo che contribuiscano all'output corretto nello strato finale. Ogni strato nascosto è composto da un vettore di valori, e la dimensione di questi vettori è detta **ampiezza del modello**. Tutti i singoli elementi del vettore possono essere considerati come neuroni. Ciascuno strato nascosto può essere visto come l'insieme di diverse **unità** che prendono molti input e restituiscono un solo output, analogamente a una funzione che mappa un vettore in uno scalare.
3. **output layer**: l'ultimo strato della rete fornisce il risultato finale, ossia la risposta della rete in base all'elaborazione degli strati precedenti.

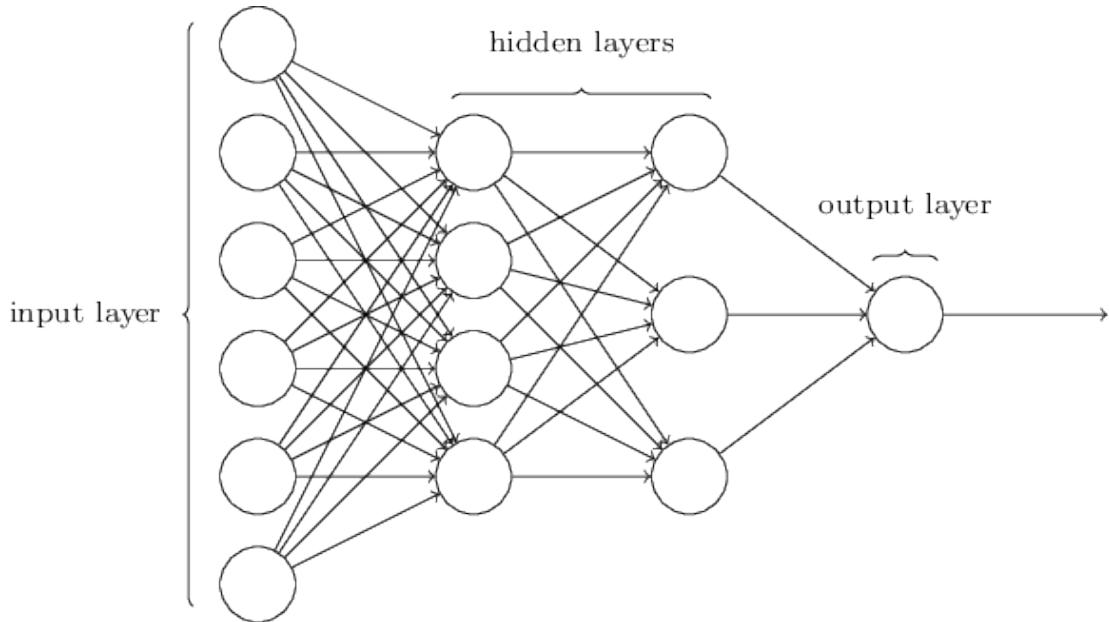


Figure 3: Layers

Un aspetto fondamentale delle reti neurali è che l'output finale è dato dalla composizione di molte funzioni non lineari. Ogni strato nascosto nella rete esegue una trasformazione sui dati in ingresso e passa l'output al successivo strato. Questo processo si traduce nella seguente struttura compositiva:

$$f(x) = f^L(f^{L-1}(\dots(f^1(x)))) \quad (7)$$

dove f^L, f^{L-1} sono le funzioni rappresentate dai diversi strati della rete, e L è il numero totale degli strati. Questa struttura a più livelli consente alla rete di apprendere e rappresentare relazioni altamente complesse e non lineari tra input e output. La capacità di modellare queste relazioni deriva proprio dalla natura compositiva delle funzioni, che consente alla rete di estrarre caratteristiche via via più astratte e rilevanti dai dati man mano che vengono processati attraverso i vari strati.

1.6 Loss Function

La funzione di perdita (**loss function**) è un componente fondamentale nell'apprendimento supervisionato, in quanto guida il processo di ottimizzazione del modello. Si tratta di una funzione matematica che misura quanto bene il modello sta prevedendo i risultati attesi, quindi quantifica la discrepanza tra l'output previsto dal modello e il valore reale fornito nei dati di addestramento. In altre parole, un valore elevato della funzione di perdita indica che il modello sta facendo errori significativi, mentre un valore prossimo a zero indica che il modello sta performing correttamente.

Il nostro obiettivo è trovare pesi e bias ottimali in modo tale che il nostro algoritmo si avvicini il più possibile all'output atteso, quindi vogliamo minimizzare il valore della funzione di perdita durante l'addestramento, affinché il modello diventi sempre più preciso nel fare previsioni. Ciò viene realizzato utilizzando la funzione di perdita più adatta al tipo di modello e problema in questione ma soprattutto tramite tecniche di ottimizzazione, come **discesa del gradiente**, che cercano i parametri del modello che minimizzano questa perdita.

La scelta della funzione di perdita è strettamente legata alla natura dell'unità di output. Nei problemi di classificazione, ad esempio, una delle funzioni di perdita più comuni è la cross-entropy, che misura la differenza tra due distribuzioni di probabilità: quella reale dei dati e quella prevista dal modello. La forma della funzione di cross-entropy dipende dal modo in cui viene rappresentato l'output del modello.

In sintesi, la funzione di perdita è essenziale per l'addestramento dei modelli, poiché fornisce una misura quantitativa dell'accuratezza di questi e consente di guidare il processo di ottimizzazione per migliorarne le performance.

1.6.1 Esempio di loss function

Un esempio di loss function particolarmente diffuso è la *Mean Squared Equation (MSE)*, che rappresenta una forma di errore quadratico medio:

$$C(w, b) = \frac{1}{2n} \sum_x \|y(x) - f_y(w, b, x)\|^2 \quad (8)$$

In questo caso si vuole però presentare la **Sparse Categorical Crossentropy**, ovvero una funzione di perdita utilizzata per problemi di classificazione multiclass, dove le etichette di classe sono rappresentate come numeri interi e che verrà utilizzata nella realizzazione del progetto. La formula che la descrive è la seguente:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_{i=1}^N \log \left(\frac{\exp(\hat{y}_{i,y_i})}{\sum_{c=1}^C \exp(\hat{y}_{i,c})} \right) \quad (9)$$

Dove:

- N è il numero di campioni nel batch,
- C è il numero di classi,
- y_i è l'etichetta corretta per il campione i (un valore intero che rappresenta l'indice della classe corretta),
- $\hat{y}_{i,c}$ è il logit (la previsione non normalizzata) per la classe c per il campione i ,
- $\exp(\hat{y}_{i,c})$ è l'esponenziale del logit per la classe c , e la somma $\sum_{c=1}^C \exp(\hat{y}_{i,c})$ è la normalizzazione per ottenere le probabilità di ciascuna classe.

La perdita totale per il batch di dimensione N è la media di tutte le perdite individuali:

$$\text{Loss} = \frac{1}{N} \sum_{i=1}^N L(\mathbf{y}_i, \hat{\mathbf{y}}_i)$$

La funzione calcola la cross-entropy tra la distribuzione delle probabilità previste e la distribuzione delle probabilità reali. Poiché le etichette sono intere, la cross-entropy si concentra solo sulla probabilità della classe corretta per ogni campione, che è indicata da y_i .

1.7 Gradient Descent

Gradient Descent è un algoritmo di ottimizzazione utilizzato per trovare l'insieme di pesi e bias che minimizzino la funzione di perdita. Come indicato in precedenza, l'obiettivo è ridurre il valore della funzione di perdita, che misura la discrepanza tra l'output previsto dal modello e il valore reale. Il gradiente di una funzione definita in uno spazio n-dimensionale indica la direzione in cui la funzione cresce più rapidamente. Calcolare il gradiente ci consente di individuare la direzione opposta, ovvero quella che porta alla diminuzione più rapida del valore della funzione.

Data una funzione $C(v)$, con $v = v_1, \dots, v_m$ vettore di variabili, il nostro obiettivo è minimizzarla. Utilizziamo v per mostrare che questo metodo può essere applicato su qualsiasi tipo di variabile, ma nel nostro contesto di applicazione v rappresenta i pesi e bias ($C(w, b)$).

Il metodo si basa sull'idea di aggiornare iterativamente i parametri spostandosi gradualmente lungo la direzione opposta al gradiente della funzione, andando verso una diminuzione del valore della funzione.

Il cambiamento ΔC nella funzione di perdita C prodotto da piccole modifiche $\Delta v = (\Delta v_1, \dots, \Delta v_m)^T$ è:

$$\Delta C \approx \nabla C \cdot \Delta v \quad (10)$$

dove il gradiente ∇C è il vettore costituito dalle derivate parziali della funzione rispetto ai parametri:

$$\nabla C = \left(\frac{\delta C}{\delta v_1}, \dots, \frac{\delta C}{\delta v_m} \right)^T \quad (11)$$

Partendo da un punto iniziale nel dominio di C , calcoliamo il gradiente della funzione rispetto ai parametri in quel punto. Nel caso delle reti neurali feedforward è importante inizializzare i pesi con valori casuali molto piccoli e i bias con valori pari a zero o positivi e piccoli. Per ridurre C , scegliamo di aggiornare i parametri muovendoci nella direzione opposta al gradiente, impostando:

$$\Delta v = -\eta \nabla C \quad (12)$$

dove η è un parametro detto **learning rate** che controlla l'entità del passo verso il minimo. In questo modo otteniamo:

$$\Delta C \approx -\eta \|\nabla C\|^2 \quad (13)$$

che è negativo, garantendo che la funzione di perdita diminuisca ad ogni iterazione. L'aggiornamento dei parametri v avviene con la formula:

$$v \rightarrow v' = v - \eta \nabla C \quad (14)$$

Questo processo viene ripetuto iterativamente fino a quando il valore della funzione non converge a un certo valore o viene raggiunto un numero prefissato di iterazioni.

Tuttavia, il calcolo del gradiente può essere complesso e costoso in termini computazionali, soprattutto quando la funzione di perdita è composta da una sequenza di funzioni (come nei modelli di rete neurale profonda). Per questo motivo, è necessario adottare tecniche che ottimizzano l'efficienza del calcolo, come la **back-propagation** e varianti del gradient descent (come il mini-batch gradient descent, **stochastic gradient descent** e momentum).

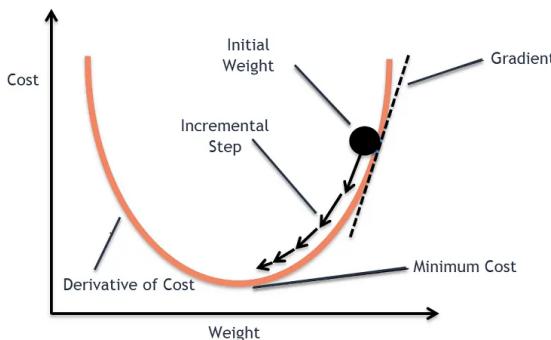


Figure 4: Gradient Descent [3]

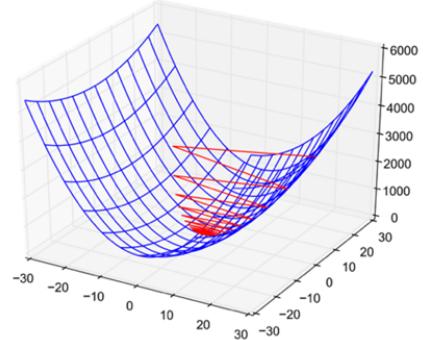


Figure 5: Gradient Descent 3D [1]

1.7.1 Stochastic Gradient Descent

Lo **Stochastic Gradient Descent** è una variante del Gradient Descent che introduce una modifica importante per rendere l'algoritmo più veloce e meno costoso in termini di risorse computazionali, specialmente per problemi di apprendimento su grandi dataset.

Invece di calcolare il gradiente impiegando tutto il dataset, l'algoritmo calcola il gradiente basandosi su un singolo esempio o piccolo *batch* di esempi selezionati casualmente dal dataset. Per ogni esempio o batch, si aggiorna subito il valore dei parametri w e b , spostandosi in una direzione che tenda a ridurre il valore della funzione di costo per quell'esempio o batch specifico.

$$v \rightarrow v' = v - \eta \nabla C_i \quad (15)$$

dove, ∇C_i è il gradiente della loss function calcolato solo sull'i-esimo esempio, invece che su tutto il dataset.

Questo approccio introduce un certo "rumore" negli aggiornamenti dei parametri, ma offre vantaggi significativi in termini di velocità, efficienza computazionale e una migliore esplorazione dello spazio dei parametri. Infatti, calcolando il gradiente su un piccolo numero di esempi, gli aggiornamenti dei parametri avvengono più frequentemente, accelerando il processo di addestramento. Inoltre, questo "rumore" aiuta a evitare che il modello rimanga bloccato in minimi locali della funzione di perdita. Per questi motivi, successivamente in fase sperimentale si preferirà usare lo Stochastic Gradient Descent, lavorando con un dataset di importanti dimensioni.

1.8 Backpropagation

Dopo aver definito il metodo di ottimizzazione per minimizzare la funzione di perdita (ad esempio, il Gradient Descent), il passo successivo è calcolare i gradienti rispetto ai parametri del modello. L'algoritmo di **backpropagation** è il metodo utilizzato per calcolare efficientemente tali gradienti, fornendo informazioni su come la modifica dei pesi e dei bias influenzino il comportamento complessivo della rete. Il gradiente indica come modificare i pesi della rete per minimizzare l'errore, e il backpropagation permette di far "fluire" l'informazione dell'errore all'indietro attraverso i livelli della rete per calcolare i gradienti.

Notazione: usiamo w_{jk}^l per indicare il peso della connessione tra il k^{th} neurone nel $(l-1)^{th}$ strato e il j^{th} neurone nel l^{th} strato. Con la stessa logica indichiamo b_j^l per i bias e a_j^l per gli attivatori dei neuroni.

Per la funzione di attivazione dei neuroni abbiamo:

$$a_j^l = \sigma(\sum_k w_{jk}^l a_j^{l-1} + b_j^l) \quad (16)$$

Possiamo riscrivere l'equazione in notazione vettoriale, definendo la matrice dei pesi w_j^l , il vettore dei bias b^l , il vettore delle attivazioni a^l e immaginando che la funzione σ agisca su tutti gli elementi che gli vengono passati. Definiamo z^l *input pesato* e riscriviamo l'equazione:

$$a_j^l = \sigma(w_j^l a^{l-1} + b^l) = \sigma(z^l) \quad (17)$$

Lo scopo della backpropagation è di calcolare le derivate parziali $\frac{\delta C}{\delta w}$ e $\frac{\delta C}{\delta b}$ della funzione di perdita per ogni peso o bias nella rete. Per rendere applicabile l'algoritmo facciamo due assunzioni:

1. La funzione di perdita può essere espressa come la media dei costi per i singoli esempi nel dataset, ossia $C = \frac{1}{n} \sum_x C_x$. In questo modo, possiamo calcolare i gradienti per ciascun esempio e poi fare una media.

- La funzione di perdita può essere espressa come funzione degli output della rete, rendendo possibile il calcolo dell'errore sullo strato di output.

Definiamo il **prodotto di Hadamard** (indicato con \circ) come un'operazione elemento-per-elemento tra vettori. Definiamo inoltre l'**errore** δ_j^l nel j^{th} neurone nel l^{th} strato: una variazione dell'input del neurone, che causa modifiche all'output:

$$\delta_j^l \equiv \frac{\delta C}{\delta z_j^l} \quad (18)$$

Attraverso le seguenti quattro equazioni sarà possibile calcolare l'errore e il gradiente della funzione di costo:

- Errore nell'output:

$$\delta^L = \nabla_a C \circ \sigma'(z^L) \quad (19)$$

- Propagazione all'indietro dell'errore:

$$\delta^L = ((w^{l+1})^T \delta^{l+1}) \circ \sigma'(z^l) \quad (20)$$

- Gradiente rispetto ai bias:

$$\frac{\delta C}{\delta b_j^l} = \delta_j^l \quad (21)$$

- Gradiente rispetto ai pesi:

$$\frac{\delta C}{\delta w_{jk}^l} = a_k^{l-1} \delta_j^l \quad (22)$$

1.8.1 Backpropagation algorithm

Riassumendo i concetti sopra citati, proviamo a scrivere i passaggi di questo algoritmo:

1. **Input** x : impostare l'attivazione a^1 corrispondente per lo strato di input
2. **Feedforward**: per ogni $l = 2, 3, \dots, L$ strato calcola $z^l = w^l a^{l-1} + b^l$ e $a^l = \sigma(z^l)$
3. **Output error** δ^l : calcola il vettore $\delta^l = \nabla_a C \circ \sigma'(z^L)$
4. **Retroaziona l'errore**: per ogni $l = L-1, L-2, \dots$ calcola $\delta^L = ((w^{l+1})^T \delta^{l+1}) \circ \sigma'(z^l)$
5. **Output**: il gradiente della funzione di costo è dato da $\frac{\delta C}{\delta w_{jk}^l} = a_k^{l-1} \delta_j^l$ e $\frac{\delta C}{\delta b_j^l} = \delta_j^l$

L'algoritmo di backpropagation è fondamentale per l'addestramento delle reti neurali, poiché consente di calcolare efficientemente i gradienti necessari per ottimizzare i pesi e i bias. Grazie a questo algoritmo, è possibile calcolare i gradienti in modo retroattivo, a partire dallo strato di output, per poi propagare l'errore all'indietro attraverso i vari strati della rete. Dato che il calcolo dei gradienti tramite questa tecnica può diventare molto costoso per reti molto grandi, spesso si utilizzano tecniche come lo Stochastic Gradient Descent o le sue varianti (come l'Adam optimizer) al fine di rendere il processo di addestramento più efficiente e gestibile.

1.9 Convolutional Neural Network

Le **Convolutional Neural Networks** (CNN) sono un sottoinsieme delle reti neurali che si distinguono dalle altre per performance migliori con immagini, discorsi e segnali audio, quindi sostanzialmente per l'elaborazione di dati strutturati in griglie.

La struttura di una CNN è caratterizzata da tre tipologie di strati:

1. **Convolutional layer:** rappresenta lo strato o gli strati principali della rete con lo scopo di estrarre caratteristiche locali dai dati attraverso l'uso di filtri. Durante la convoluzione un filtro o kernel viene fatto scorrere sui dati (sull'immagine originale), esaminando una piccola sezione alla volta. Il filtro estrae certe caratteristiche o informazioni creando una serie di *feature maps* (mappe di caratteristiche), che evidenziano pattern rilevanti, come bordi, angoli o texture. Ogni filtro è specializzato nel rilevare una specifica caratteristica, e grazie alla tecnica dei pesi condivisi, lo stesso filtro viene applicato in diverse posizioni del dato d'ingresso.
2. **Pooling layer:** si occupano di ridimensionare le mappe delle caratteristiche, mantenendo le informazioni rilevanti e riducendo la complessità computazionale. Ad esempio, il *max pooling* seleziona il valore massimo in una regione specifica della mappa, conservando le informazioni più salienti.
3. **Fully-connected layer:** rappresenta lo strato finale della rete in cui i neuroni sono completamente connessi. Quest'ultimi ricevono input da tutti i neuroni dello strato precedente e si occupano del compito finale, come la classificazione o la regressione, basandosi sulle caratteristiche estratte dagli strati precedenti.

Sostanzialmente le CNN elaborano e combinano caratteristiche localizzate per riconoscere strutture più complesse. Gli strati convoluzionali e di pooling lavorano in sequenza per interpretare i dati e al termine sono presenti uno o più strati FC che lavorano come una rete neurale tradizionale. Nel caso delle immagini possiamo dire che le CNN scompongono le immagini in piccole parti e combinando le informazioni, comprendo l'immagine nel suo complesso.

Le CNN integrano tre concetti fondamentali che le rendono particolarmente adatte per l'elaborazione di dati visivi e strutturati:

- Campi recettivi locali: ogni neurone elabora solo una piccola porzione dell'input, consentendo di estrarre caratteristiche elementari come bordi o angoli.
- Pesi condivisi: all'interno di una feature map, tutti i neuroni utilizzano lo stesso filtro, permettendo di rilevare le stesse caratteristiche in diverse parti dell'immagine.
- Sottocampionamento spaziale o temporale: riduce la risoluzione spaziale delle mappe delle caratteristiche, diminuendo la precisione della posizione delle caratteristiche rilevate ma conservando la loro relazione relativa, importante per la classificazione.

Un esempio classico di CNN è **LeNet**, progettata per il riconoscimento di cifre scritte a mano. Questa rete combina strati convoluzionali per estrarre caratteristiche locali, strati di pooling per ridurre la complessità e strati completamente connessi per eseguire la classificazione finale. Grazie alla progressiva riduzione della risoluzione spaziale e all'aumento della ricchezza delle rappresentazioni, LeNet è in grado di rilevare caratteristiche distintive, mantenendo robustezza rispetto a trasformazioni geometriche dell'input [4].

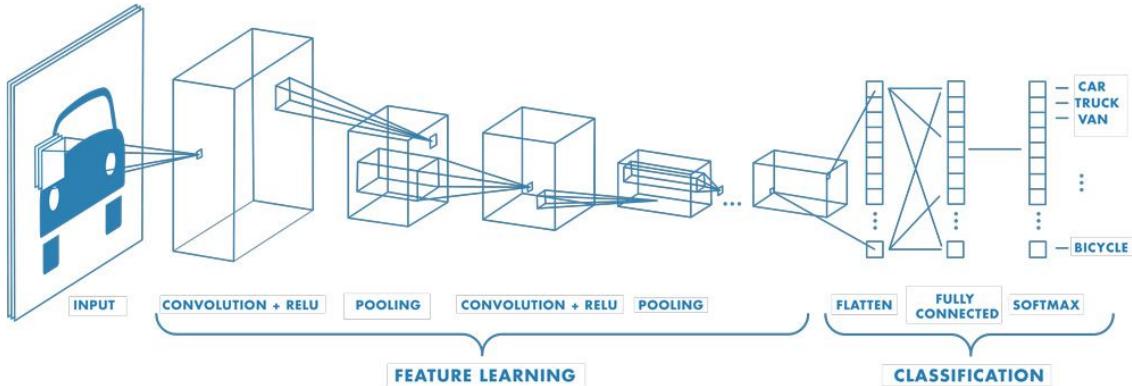


Figure 6: Architettura di una CNN [5]

Le CNN eccellono nel gestire variazioni, spostamenti e distorsioni nei dati. Sono ampiamente utilizzate in applicazioni per riconoscimento di immagini e volti, segmentazione di immagini mediche, riconoscimento vocale e di segnali audio, elaborazione video e guida autonoma.

2 Progetto

2.1 Federated Learning

L'utilizzo costante e globale di dispositivi mobili, oggi, fornisce un'enorme quantità di dati, molti dei quali risultano però essere di natura privata. Sebbene l'utilizzo di questi dati sia essenziale per migliorare le applicazioni, raccoglierli e conservarli in modo centralizzato comporta rischi significativi per la privacy. Per affrontare questo problema, si propone una tecnica di apprendimento che consente di sfruttare tali dati senza centralizzarli, coordinando invece il lavoro di calcolo dei singoli client. Ogni **client**, corrispondente al singolo dispositivo mobile, ha un dataset di allenamento locale che non invia direttamente al **server**, ma lo sfrutta per calcolare un aggiornamento all'attuale modello globale e invierà solo quest'ultimo [6].

Il **Federated Learning** è una tecnica di machine learning che si basa sulle seguenti proprietà:

1. L'addestramento di reti neurali su dati reali reperibili dai dispositivi mobili offre un vantaggio significativo rispetto all'uso di dati di riferimento generalmente disponibili nei data center.
2. Tali dati sono sensibili dal punto di vista della privacy o sono di grandi dimensione, quindi è preferibile non trasferirli al data center solo per addestrare il modello, rispettando il principio della raccolta mirata.
3. Per i compiti supervisionati, le etichette sui dati possono essere ricavate in modo naturale dall'interazione dell'utente.

2.2 Federated Averaging Algorithm

L'algoritmo è diviso in due parti:

1. La parte eseguita sul server centrale.
2. La funzione *ClientUpdate()*, che viene eseguita su ogni client selezionato.

Il server ha il compito di inizializzare i pesi del modello globale w_0 e per ogni round t eseguire le seguenti azione:

- determina quali client utilizzare nel round, scegliendone una frazione C tra i K disponibili;
- per ogni client il server invoca *ClientUpdate()* che aggiorna i pesi del modello per quel client usando i dati locali;
- dopo aver ricevuti i pesi aggiornati dai client selezionati, viene calcolata la media pesata in modo da trovare i pesi globali.

Algorithm 1 Federated Averaging Algorithm

Server executes:

```
1: initialize  $w_0$ 
2: for each round  $t = 1, 2, \dots$  do
3:    $m \leftarrow \max(C \cdot K, 1)$ 
4:    $S_t \leftarrow$  (random set of  $m$  clients)
5:   for each client  $k \in S_t$  in parallel do
6:      $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$ 
7:   end for
8:    $w_{t+1} \leftarrow \sum_{k=1}^K \frac{n_k}{n} w_{t+1}^k$ 
9: end for
```

ClientUpdate(k, w): // Run on client k

```
1:  $\mathcal{B} \leftarrow$  (split  $\mathcal{P}_k$  into batches of size  $B$ )
2: for each local epoch  $i$  from 1 to  $E$  do
3:   for batch  $b \in \mathcal{B}$  do
4:      $w \leftarrow w - \eta \nabla \ell(w; b)$ 
5:   end for
6: end for
7: return  $w$  to server
```

Ogni client si occuperà quindi di aggiornare i pesi sull’allenamento effettuato localmente. Nel dettaglio:

- divide il dataset local in gruppi (batches) di dimensione B ;
- per ogni epoca E esegue il loop sui batches;
- per ogni batch aggiorna i pesi del modello sfruttando il *Gradient Descent* con un dato *learning rate*;
- al termine invia i parametri aggiornati al server.

Questo approccio consente di addestrare modelli altamente performanti su dati distribuiti senza compromettere la sicurezza o la privacy dei dati personali.

L’algoritmo di Federated Averaging (FedAvg) è un esempio chiave di come viene gestito l’addestramento distribuito su client, ed è particolarmente utile per sistemi mobili e dispositivi IoT che non possono permettersi di inviare grandi volumi di dati ai server centralizzati.

2.3 Obiettivi

Scopo del documento è quello di mostrare un’implementazione dell’algoritmo *Federated Averaging* in formato locale, quindi senza la realizzazione dell’effettiva comunicazione tra client e server, e con un numero ridotto di client che lavorano in

modo sequenziale e non in parallelo come vorrebbe l'algoritmo. L'algoritmo verrà testato sul dataset *Fashion MNIST*, in un classico problema di classificazione delle immagini, con l'obiettivo di mettere in luce le differenze di risultati sull'accuratezza del test set del server centrale, in base al partizionamento del dataset tra i client. In questa implementazione, verranno esplorati diversi tipi di partizionamento del dataset tra i client:

- **IID** (Independent and Identically Distributed): i dati sono distribuiti equamente e casualmente tra i client.
- **Non-IID** (Non-Independent and Identically Distributed): i dati sono distribuiti in modo squilibrato, con ciascun client che riceve dati di classi specifiche.

Nonostante non verrà realizzato un protocollo di comunicazione vero e proprio, ma una simulazione in locale, si proverà a realizzare una buona scomposizione modulare in funzioni riutilizzabili per differenti rappresentazioni. L'obiettivo è garantire un codice che possa essere facilmente esteso o modificato, per includere scenari più realistici come la comunicazione asincrona o l'integrazione con dataset differenti. Si vuole mettere in luce sperimentalmente come dati eterogenei peggiorino le performance di un modello di ML allenato tramite Federated Learning rispetto a un caso ideale di dati omogenei.

2.4 Implementazione

Il prototipo è stato realizzato per gestire dataset distribuiti utilizzando tecniche di partizionamento flessibili, permettendo esperimenti su diverse configurazioni (es. distribuzione IID o Non-IID). Ogni client esegue l'addestramento locale sui propri dati e restituisce i pesi aggiornati al server centrale, che li aggredisce per formare un modello globale aggiornato.

Dal punto di vista implementativo, sono stati definiti componenti principali per il caricamento e la suddivisione dei dati, l'addestramento locale e la simulazione del processo di aggregazione. Il codice è strutturato in funzioni riutilizzabili e integrate, semplificando sia la gestione del prototipo sia la possibilità di introdurre nuove funzionalità, come l'aggiunta di protocolli di comunicazione asincrona.

2.5 Librerie utilizzate

Per l'attuazione del progetto verranno utilizzate le seguenti librerie:

```
1 import tensorflow as tf
2 import tensorflow_datasets as tfds
3 import random
4 import matplotlib.pyplot as plt
5 import numpy as np
```

Di particolare interesse è sicuramente *tensorflow*, libreria open-source lanciata da Google, tra le più potenti e versatili per il machine learning e il deep learning. Con l'integrazione di Keras, TensorFlow offre un'API intuitiva per creare, addestrare e ottimizzare modelli di deep learning con poche righe di codice. Ciò consente di passare rapidamente da idee iniziali a prototipi funzionanti.

Sfrutteremo anche la libreria *tensorflow_datasets* progettata per semplificare l'accesso ai dataset di machine learning. I dataset possono essere caricati in formati compatibili con TensorFlow, pronti per l'addestramento, il che riduce il tempo dedicato alla pre-elaborazione. TensorFlowDatasets gestisce automaticamente il download e la memorizzazione in cache dei dataset, riducendo le operazioni manuali e migliorando le prestazioni.

Infine con l'appoggio delle librerie *numpy*, *matplotlib.pyplot* e *random*, avremo modo di gestire array multidimensionali, realizzare grafici e generare numeri casuali.

2.6 Partizionamento del dataset

Fulcro delle riflessioni dei risultati sperimentali sarà la modalità di partizionamento del dataset preso in interesse (in questo caso Fashion MNIST ma modificabile). Per questo motivo in questa sezione si presentano le due diverse funzioni utilizzate per la distribuzione dei dati tra i client: *divide_set()* e *divide_set_uniform()*. Entrambe sono perfettamente integrabili con il resto del programma, con l'unica modifica da

fare relativa alla nomenclatura con le quali vengono chiamate. Infatti, ambedue le funzioni prendono in input il numero di client a cui il dataset verrà suddiviso, i dati di input per l’addestramento (images), le etichette corrispondenti e la dimensione dei batch B e producono in output una lista contenente i dataset per ciascun client.

```

1 def divide_set(num_clients, images, labels, B):
2     ds_train = tf.data.Dataset.from_tensor_slices((images,
3         labels))
4     ds_train = ds_train.shuffle(len(images)).batch(B).
5     prefetch(tf.data.AUTOTUNE)
6
7     # Partizionamento del dataset per N client
8     client_data = []
9     for i in range(num_clients):
10         client_data.append(ds_train.shard(num_clients, i))
11
12     return client_data

```

La funzione *divide_set()* permette di simulare un ambiente di apprendimento federato, in cui ogni client ha accesso a una parte del dataset (IID). L’operazione di **shard** garantisce che i dati siano distribuiti in modo uniforme tra i client, ognuno dei quali riceve un sottoinsieme dei dati, evitando la duplicazione.

```

1 def divide_set_uniform(num_clients, images, labels, B):
2     num_classes = 10
3     images_per_class = len(images) // num_classes # 6000
4     immagini per classe
5     classes_per_client = 5 # Ogni client riceve immagini
6     da 5 classi/1 classe
7     images_per_client_per_class = images_per_class //
8     classes_per_client
9
10    # Creiamo una lista di immagini suddivise per classe
11    class_data = {i: [] for i in range(num_classes)} # 10
12    classi di fashion mnist
13
14    # Suddividiamo le immagini per classe
15    for img, lbl in zip(images, labels):
16        class_data[lbl].append((img, lbl))
17
18    # Creiamo i dati per i client
19    client_data = []

```

```

20         # Ogni client riceve immagini da 5 classi contigue
21         for cl in range(classes_per_client):
22             # La classe di partenza varia a seconda del
23             # client
24             start_class = (i + cl) % num_classes
25             client_class_images = class_data[start_class]
26             client_class_images = client_class_images[:images_per_client_per_class]
27
28             client_images.extend([item[0] for item in
29             client_class_images])
30             client_labels.extend([item[1] for item in
31             client_class_images])
32
33             # Creiamo un dataset da assegnare al client
34             client_dataset = tf.data.Dataset.
35             from_tensor_slices((client_images, client_labels))
36             client_dataset = client_dataset.shuffle(len(
37             client_images)).batch(B).prefetch(tf.data.AUTOTUNE)
38             client_data.append(client_dataset)
39
40     return client_data

```

La funzione `divide_set_uniform()` suddivide il dataset tra un numero specifico di client, in modo tale che ogni client riceva un sottoinsieme di immagini di determinate classi. Nel nostro scenario effettueremo la suddivisione in due modi: il primo che andrà a simulare l'eterogeneità dei dati in un contesto reale, il secondo che estremizzerà tale concetto.

Sostanzialmente, nel primo caso, divideremo il dataset tra dieci client assegnando a ciascuno di essi cinque classi sul quale effettuare l'allenamento. Questa distribuzione risulta utile per simulare uno scenario di apprendimento federato in cui i dati sui dispositivi sono diversi tra loro in termini di distribuzione di classe, rappresentando un setting realistico in applicazioni decentralizzate. Nel secondo caso, ogni client viene allenato su una sola classe, e ciò viene fatto con lo scopo di mostrare cosa comporterebbe uno scenario abbastanza irrealistico realizzato con una suddivisione di questo tipo.

2.7 Funzione per mostrare la suddivisione del dataset tra client

Per garantire una comprensione migliore di cosa accade a seguito del partizionamento, si è voluto realizzare una funzione che crea un grafico a barre impilate con i client sull'asse x e il numero di campioni per classe sull'asse y. In tal modo è possibile avere una visione d'insieme di come sono distribuite le classi tra i client a livello grafico.

```

1 def show_client_class_distribution(client_data,
2     num_classes=10):
3     # Inizializza una lista per memorizzare la
4     # distribuzione delle classi per ogni client
5     class_distribution_per_client = np.zeros((len(
6         client_data), num_classes)) # [client x class]
7
8     # Calcola il numero di campioni per ciascuna classe
9     # per ogni client
10    for i, client_dataset in enumerate(client_data):
11        for batch in client_dataset:
12            images, labels = batch
13            labels = labels.numpy() # Converti in numpy
14            array per lavorare sui singoli valori
15            for label in labels:
16                class_distribution_per_client[i, label] +=
17                    1 # Incrementa il conteggio della classe per il
18                    client
19
20    # Crea un grafico a barre impilate
21    plt.figure(figsize=(10, 6))
22
23    # Crea una lista di colori per le classi (per
24    # differenziarle visivamente)
25    colors = plt.cm.get_cmap("tab10", num_classes)
26
27    # Plot a barre impilate
28    bottom = np.zeros(len(client_data)) # Questo terra'
29    traccia di dove inizia ogni segmento di classe
30
31    for class_id in range(num_classes):
32        # Estraie il numero di campioni per la classe
33        # corrente
34        class_counts = class_distribution_per_client[:, ,
35        class_id]
36
37        # Disegna una barra impilata per questa classe
38        plt.bar(range(len(client_data)), class_counts,
39        bottom=bottom, label=f'Classe {class_id}', color=colors
40        (class_id))
41
42        # Aggiunge l'altezza di questa classe alla
43        # variabile bottom (per impilare correttamente)
44        bottom += class_counts

```

```
31
32     # Impostazioni del grafico
33     plt.title("Distribuzione delle classi per client")
34     plt.xlabel("Client")
35     plt.ylabel("Numero di Campioni")
36     plt.xticks(range(len(client_data)), [f"Client {i}" for
37 i in range(len(client_data))])
38     plt.legend(title="Classi")
39     plt.grid(axis='y')
40
41     plt.legend(title="Classi", bbox_to_anchor=(0.95, 1),
42 loc='upper left')
43     plt.grid(axis='y')
44
45     # Mostra il grafico
46     plt.show()
```

2.8 Caricamento del dataset

```
1 def load_data(B, num_clients):
2     """
3         Carica il dataset specificato e lo prepara per l'
4         addestramento.
5     """
6     (x_train, y_train), (x_test, y_test) = tf.keras.
7         datasets.fashion_mnist.load_data()
8
9     # Normalizza le immagini
10    x_train, x_test = x_train / 255.0, x_test / 255.0
11
12    # Partizionamento del dataset per N client, in base
13    # alle label
14    client_data = divide_set_uniform(num_clients, x_train,
15        y_train, B) #oppure divide_set(num_clients, x_train,
16        y_train, B)
17    show_client_class_distribution(client_data)
18
19    # Preparazione del test set
20    ds_test = tf.data.Dataset.from_tensor_slices((x_test,
21        y_test))
22    ds_test = ds_test.batch(B).cache().prefetch(tf.data.
23        AUTOTUNE)
```

```
19     return client_data, ds_test
```

La funzione *load_data()* carica, normalizza e prepara il dataset Fashion MNIST per l’addestramento, suddividendolo tra più client per simulare un ambiente di apprendimento federato, sfruttando le funzioni presentate in precedenza per il partizionamento.

Ricevuti in input il numero di client e il numero di batch, il dataset Fashion MNIST è caricato tramite l’API Keras e suddiviso in set di addestramento e di test. Segue la normalizzazione delle immagini inserita per facilitare l’addestramento, migliorando stabilità e convergenza. Oltre a partizionare e mostrare il risultato di tale operazione con le funzioni sopra descritte, viene anche preparato il set di test. Vengono restituiti in output la lista dei dataset per ogni client e il dataset di test comune a tutti i client per la valutazione delle prestazioni globali del modello.

2.9 Funzione per l’aggiornamento locale dei client

Per poter simulare l’addestramento locale dei singoli client sul dataset è necessario realizzare la funzione *client_update()*, che avrà come input i pesi iniziali, ovvero i pesi calcolati dal server alla fine di ogni round o nel caso iniziale calcolati sul modello di base, il dataset di allenamento e il numero di epoch; come output verranno restituiti i pesi aggiornati del modello, che saranno poi aggregati dal server centrale per aggiornare il modello globale.

La funzione si occupa di creare un modello a tre strati, impostare i pesi che gli vengono passati, compilare il modello con l’utilizzo dell’ottimizzatore **Stochastic Gradient Descent** con learning rate 0.01 e infine addestrare il modello per *E* epoches sui dati locali (*ds_train*), che rappresentano il dataset del client. Le motivazioni sulla scelta di questo ottimizzatore possono essere comprese dalle argomentazioni mostrate nella Sezione 1.7.1.

```
1 def client_update(initial_weights, ds_train, E):
2     # Crea il modello e applica l'aggiornamento sui dati
3     # del client
4     model = tf.keras.models.Sequential([
5         tf.keras.layers.Flatten(input_shape=(28, 28)),
6         tf.keras.layers.Dense(128, activation='relu'),
7         tf.keras.layers.Dense(10)
8     ])
9     model.set_weights(initial_weights)
10    model.compile(optimizer=tf.keras.optimizers.SGD(0.01),
11                   loss=tf.keras.losses.
12                     SparseCategoricalCrossentropy(from_logits=True),
13                     metrics=[tf.keras.metrics.
14                         SparseCategoricalAccuracy()])
```

```

13     # Addestra il modello
14     model.fit(ds_train, epochs=E)
15
16     return model.get_weights()

```

2.10 Avvio della simulazione

Come parte centrale del programma e simulazione del comportamento del server avremo la funzione *start_simulation()* che avvia la simulazione di Federated Learning. Riceve come parametri: K (numero di client), rounds (numero di round), C (percentuale di client selezionati), B (dimensione del minibatch) ed E (epoche).

```

1 def start_simulation(K, rounds, C, B, E):
2     # Carica e partiziona i dati e ottieni il test_set
3     client_data, ds_test = load_data(B, K)
4
5     client_ids = list(range(K))
6
7     # Definisci modello di base
8     model = tf.keras.models.Sequential([
9         tf.keras.layers.Flatten(input_shape=(28, 28)),
10        tf.keras.layers.Dense(128, activation='relu'),
11        tf.keras.layers.Dense(10)
12    ])
13    # Ottieni pesi iniziali
14    w = model.get_weights()
15
16    # Compila il modello prima di eseguire la valutazione
17    model.compile(optimizer=tf.keras.optimizers.SGD(0.01),
18                   loss=tf.keras.losses.
19                   SparseCategoricalCrossentropy(from_logits=True),
20                   metrics=[tf.keras.metrics.
21                   SparseCategoricalAccuracy()])
22
23    # Fase di evaluation del modello globale del server
24    # prima del primo round
25    print("Valutando il modello prima del primo round...")
26    loss, accuracy = model.evaluate(ds_test)
27    print(f"Modello iniziale - Loss: {loss:.4f}, Accuracy: {accuracy:.4f}")
28
29    # Inizializza le liste per registrare l'accuratezza e
30    # i round
31    accuracy_per_round = [accuracy]
32    rounds_list = [0]

```

```

29
30     for t in range(rounds):
31         print(f"Round {t + 1} iniziato")
32
33         m = max(int(C * K), 1)
34         St = random.sample(client_ids, m)
35
36         pesi_cliente = []
37         for i in St:
38             print(f"\nChiamando il Client({i + 1})...")
39
40             # Passa i pesi al client per l'aggiornamento
41             updated_weights = client_update(w, client_data
42             [i], E) # Aggiornamento dei pesi
43
44             # Raccogli i pesi aggiornati dal client
45             pesi_cliente.append(updated_weights)
46
47             # Calcola i nuovi pesi mediando quelli ricevuti
48             # dai client
49             w = [sum(client_weights[layer] for client_weights
50             in pesi_cliente) / len(pesi_cliente) for layer in range
51             (len(w))]
52             print(f"Round {t + 1} completato, pesi aggiornati
53             calcolati.")
54
55             # Fase di valutazione del modello alla fine di ogni
56             # round
57             print(f"Valutando il modello alla fine del round {t+1}...")
58             # Aggiornamento dei pesi nel modello globale
59             model.set_weights(w)
60
61             loss, accuracy = model.evaluate(ds_test)
62             print(f"Round {t+1} - Loss: {loss:.4f}, Accuracy:
63             {accuracy:.4f}")
64
65             # Aggiungi l'accuratezza e il round alla lista per
66             # il grafico
67             accuracy_per_round.append(accuracy)
68             rounds_list.append(t + 1)
69
70             # Visualizza il grafico dell'accuratezza
71             plt.figure(figsize=(10, 6))
72             plt.plot(rounds_list, accuracy_per_round, marker='o',

```

```

    color='b')
65     plt.title("Federated Learning - 5 classi per client")
66     plt.xlabel("Round")
67     plt.ylabel("Test Accuracy")
68     plt.grid()
69     plt.show()

```

Questa funzione si appoggia su quelle definite in precedenza per realizzare le operazioni più importanti.

Inizialmente vengono caricati e partizionati i dati con la funzione `load_data()`. Segue la definizione del modello di base dal quale ottenere i pesi iniziali e una valutazione iniziale con calcolo dell'accuratezza iniziale del modello sul dataset di test.

Si entra nel ciclo principale del Federated Learning, in cui per ogni round, viene selezionato un sottoinsieme casuale di client per l'addestramento locale. Per ogni client viene chiamata la funzione `client_update()` che restituisce i pesi aggiornati, i quali vengono raccolti.

Si calcolano quindi i nuovi pesi globali come media dei pesi ricevuti dai client selezionati, che serviranno per aggiornare il modello globale e valutarne l'accuratezza, la quale viene mostrata al termine dei round tramite rappresentazione grafica.

La funzione `main()` definisce e avvia la simulazione di Federated Learning, basandosi sulla funzione `start_simulation()` importata dal modulo `fed_avg`. Si è deciso di utilizzare un file separato per gestire l'avvio della simulazione, in modo da aumentare la scalabilità e poterne controllare la configurazione. I risultati che verranno mostrati nella prossima sezione sono basati sulla configurazione dei parametri come mostrato nel codice.

```

1 import fed_avg
2
3 def main():
4     # Parametri personalizzabili per la simulazione
5     K = 10      # Numero di client
6     rounds = 10  # Numero di round di Federated Learning
7     C = 1       # Percentuale di client da selezionare in
8     ogni round
9     B = 50      # Dimensione del minibatch
10    E = 20      # Numero di epoche di addestramento
11    # Avvia la simulazione con i parametri definiti
12    fed_avg.start_simulation(K, rounds, C, B, E)
13
14 if __name__ == "__main__":
15     main()

```

- K (10): definisce il numero totale di client coinvolti nel processo.
- rounds (10): specifica il numero di round di Federated Learning, ossia le iterazioni durante le quali i client aggiornano e inviano i loro pesi per la mediazione.
- C (1): rappresenta la percentuale di client selezionati per ogni round. Un valore di 1 indica che verranno selezionati tutti i client a ogni round.
- B (50): dimensione del minibatch utilizzato per addestrare il modello localmente su ciascun client.
- E (20): numero di epoche di addestramento locale che ogni client effettuerà sui propri dati per aggiornare i pesi del modello.

3 Risultati sperimentali

Una volta terminata l'implementazione del programma, il passaggio successivo è stato quello di testare l'algoritmo di *Federated Averaging* in locale e analizzarne i risultati, per poter simulare e comprendere quello che potrebbe essere il funzionamento in scala globale. La fase di test prevede tre diverse configurazioni delle modalità di distribuzione delle classi tra client del dataset analizzato. Le modalità di partizionamento sono:

- **IID (Identically and Independently Distributed)**: ogni client ha una distribuzione uniforme delle classi e quindi tutte le classi sono rappresentate in modo equilibrato.
- **5 classi per client**: ogni client riceve immagini da 5 classi contigue, quindi la distribuzione delle classi tra i client non è uniforme e ogni client è specializzato in un sottoinsieme di classi.
- **1 classe per client**: ogni client riceve immagini da una sola classe, il che significa che ogni client è specializzato in una classe specifica.

L'analisi verrà effettuata sui valori delle accuracy del test set del server al termine di ogni round.

L'**accuracy** è una delle misure principali di performance nel machine learning che rappresenta il numero di predizioni corrette sul totale delle predizioni effettuate. Nel contesto di classificazione, l'accuracy indica quanto spesso il modello ha classificato correttamente i campioni di un dataset, sia esso di addestramento, validazione o test. Nel nostro esperimento sfrutteremo questo indicatore per capire quanto bene il modello globale classifica correttamente le immagini di Fashion MNIST (test set) al termine di ciascun round di addestramento federato. L'accuracy rifletterà quindi l'efficacia del modello globale nel generalizzare sulle classi delle immagini.

3.1 IID

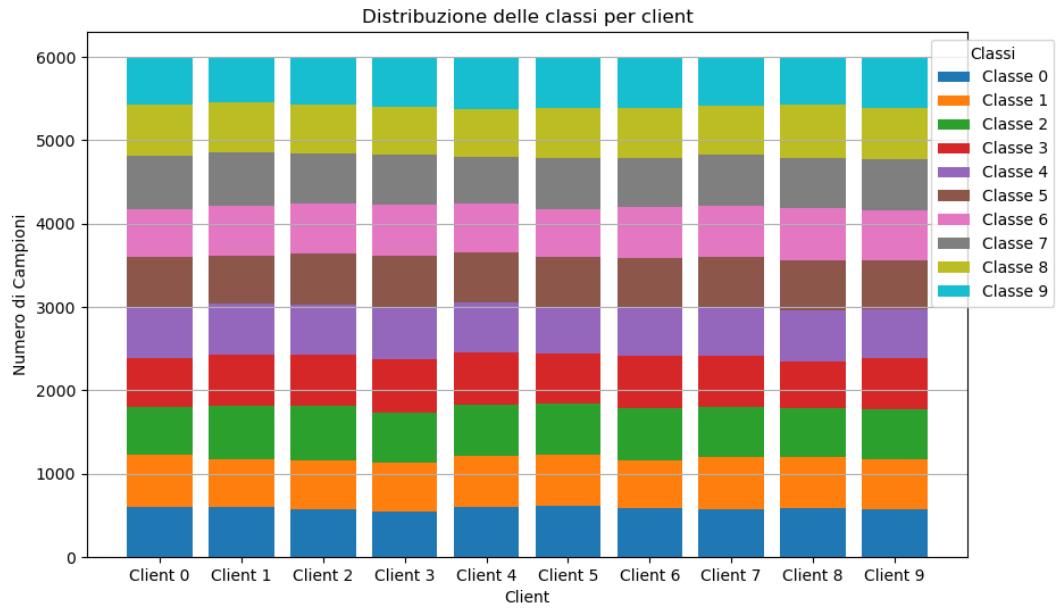


Figure 7: Divisione delle classi IID

La Figura 7 mostra come i dati vengono distribuiti tra i client tramite l'utilizzo della funzione `divide_set()` in precedenza descritta. I dati vengono suddivisi tra i client tramite l'operazione di "shard" che assegna ai client i dati in modo casuale e uniforme su tutte le classi, rendendo la distribuzione dei dati IID.

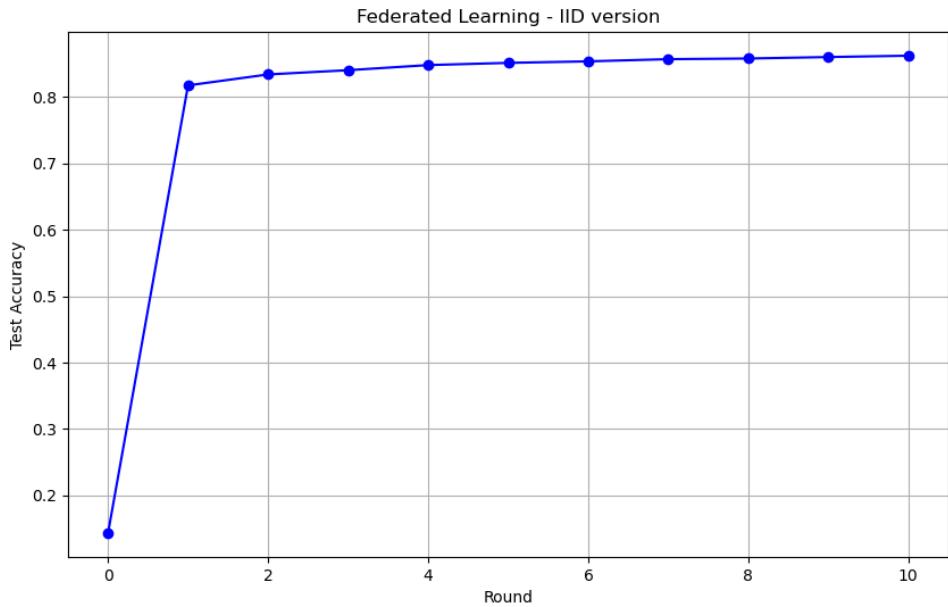


Figure 8: Grafico accuracy - round per distribuzione IID

La Figura 8 mostra i risultati dell’accuracy ottenuta ogni round sul test set globale mantenuto dal server. Il modello raggiunge un’accuracy elevata già a partire dal primo round (81,74 %) e continua a migliorare costantemente, con un progresso modesto ma costante. Questa configurazione si dimostra efficiente, consentendo al modello di raggiungere rapidamente una buona performance poiché ogni client dispone di dati rappresentativi di tutte le classi. Avere una distribuzione dei dati omogenea rappresenta una situazione ideale e questo si riflette negli ottimi risultati ottenuti.

3.2 Distribuzione classi per client

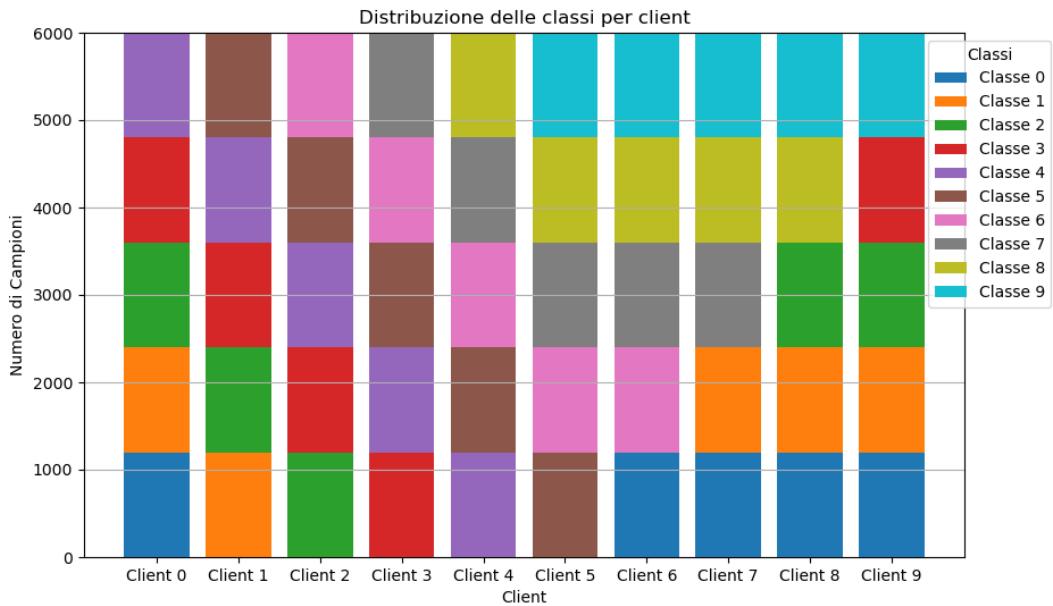


Figure 9: Distribuzione 5 classi per client

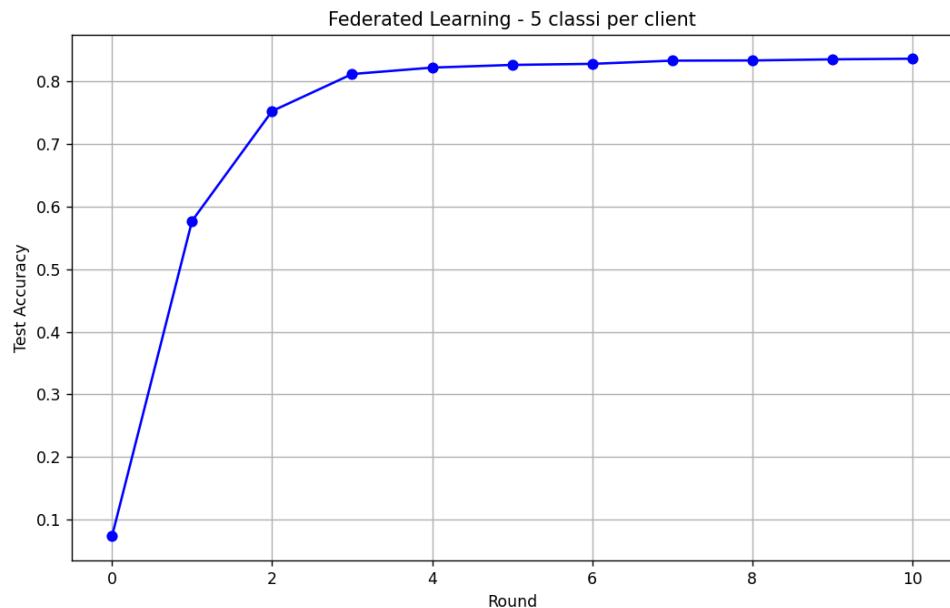


Figure 10: Grafico accuracy - round per distribuzione 5 classi per client

Questa configurazione si ottiene attraverso la funzione `divide_set_uniform()` e impostando `class_per_client = 5` al suo interno, che distribuisce a ciascun client dati di 5 classi contigue, limitando la varietà ma mantenendo una certa diversità di dati, con l'obiettivo di simulare una disposizione eterogenea dei dati, simile a quanto potrebbe accadere nella realtà (10). L'accuracy parte leggermente a rilento nei primissimi round (57,73%) ma migliora rapidamente, raggiungendo ottimi livelli, simili alla prima versione con l'aumentare dei cicli. La limitata diversità iniziale dei dati rallenta l'apprendimento del modello rispetto alla distribuzione IID, pur permettendo comunque un buon livello di generalizzazione nel tempo.

Per estremizzare il concetto dell'eterogeneità dei dati si è pensato di creare una configurazione in cui ciascun client riceve dati di una sola classe, come si può vedere dalla Figura 11.

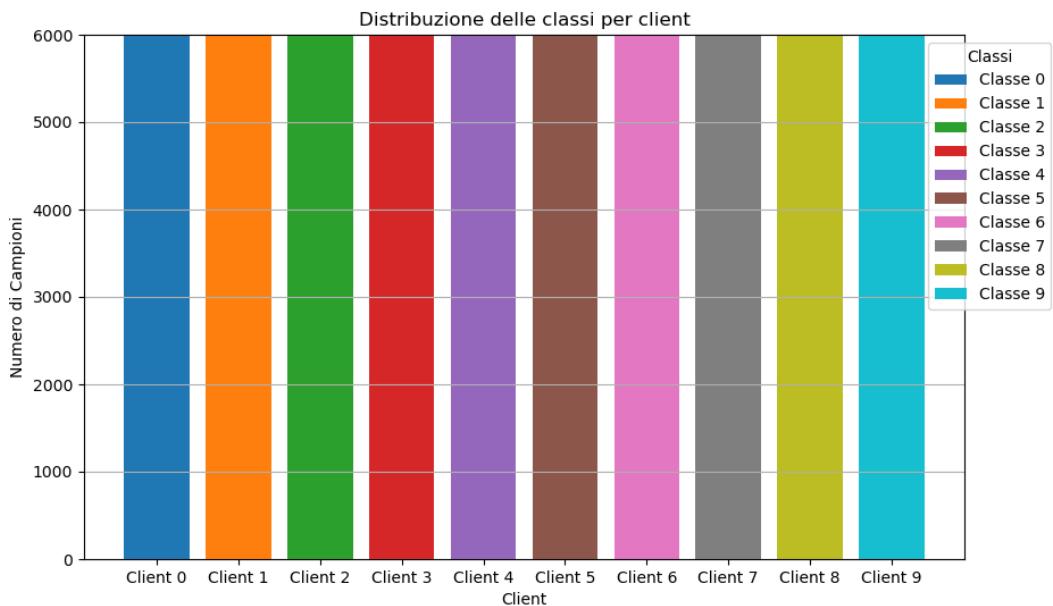


Figure 11: Distribuzione 1 classe per client

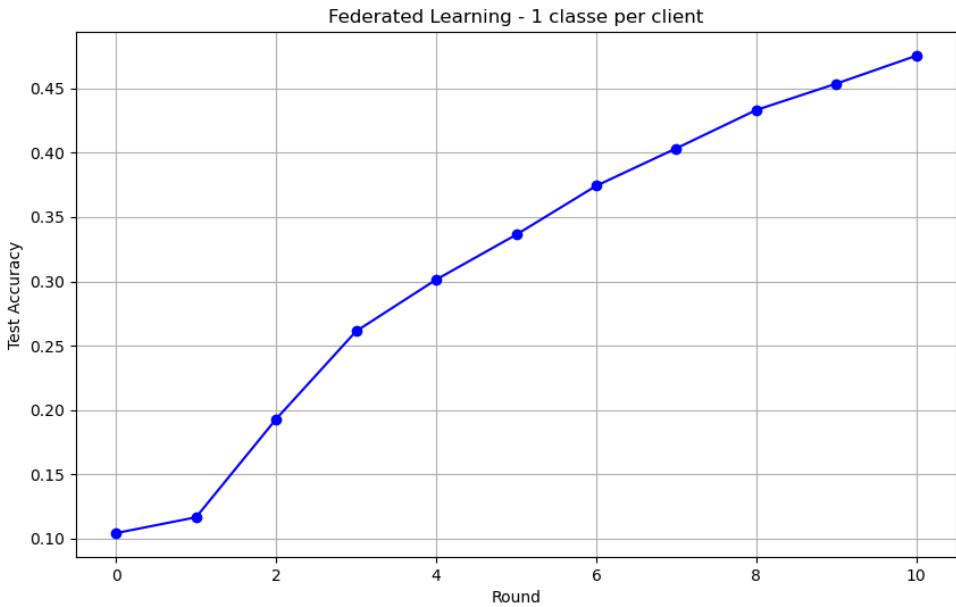


Figure 12: Grafico accuracy - round per distribuzione 1 classe per client

Guardando la Figura 12 e senza soffermarsi sui valori della test accuracy sull’asse delle ordinate, la prima cosa che salta all’occhio è la crescita lenta e progressiva della “curva di apprendimento”, molto diversa dai grafici presentati in precedenza. All’inizio dell’addestramento, l’accuracy è molto bassa perché il modello globale riceve aggiornamenti da client che lavorano solo su specifiche classi. Il modello tende quindi a “specializzarsi” in quelle poche classi, risultando meno accurato sulle altre classi non viste. Al passare dei round il modello inizia gradualmente a imparare dalle informazioni dei client però, poiché i client non hanno esempi di tutte le classi, il modello necessita di più round per migliorare la sua performance.

3.3 Confronto

Analizzati nel dettaglio i risultati ottenuti dalle tre diverse implementazioni, possiamo procedere con un confronto tra i risultati relativi alle tre distribuzioni dati prese in esame.

Table 1: Risultati sperimentali ottenuti dal calcolo dell’accuracy sul test set del server nei 10 rounds

	Round 0	Round 1	Round 2	Round 3	Round 4	Round 5	Round 6	Round 7	Round 8	Round 9	Round 10
IID	14,33	81,74	83,39	84,03	84,8	85,13	85,36	85,69	85,79	86,01	86,21
5 classi	7,36	57,73	75,27	81,19	82,22	82,65	82,82	83,33	83,36	83,54	83,64
1 classe	10,44	11,67	19,3	26,13	30,14	33,63	37,43	40,34	43,32	45,37	47,54

La Tabella 1 mostra i risultati dell’accuracy su 10 round di addestramento in un contesto di Federated Learning, con le tre diverse configurazioni di distribuzione dei dati per i client:

- **IID:** Il modello raggiunge un’accuracy elevata già al primo round (81,74%) e continua a migliorare costantemente, con un progresso modesto ma costante fino a stabilizzarsi attorno a un’accuracy di circa 86,21% al decimo round. Come si può vedere dai risultati e dal grafico di confronto in Figura 13 si dimostra la più efficiente, consentendo al modello di raggiungere rapidamente alte performance poiché ogni client dispone di dati rappresentativi di tutte le classi distribuiti in modo omogeneo.
- **5 classi per client:** L’accuracy parte più bassa rispetto a IID (57,73%) e migliora rapidamente, raggiungendo 83,64% al decimo round. Rispetto alla configurazione IID, la crescita iniziale è più lenta, ma il modello si avvicina all’accuracy di IID dopo diversi round. La limitata diversità iniziale dei dati rallenta l’apprendimento del modello rispetto alla distribuzione IID, pur permettendo comunque un buon livello di generalizzazione nel tempo.
- **1 classe per client:** L’accuracy iniziale è molto bassa (10,44%) e aumenta lentamente ad ogni round, raggiungendo solo il 47,54% al decimo round. La crescita è progressiva ma nettamente inferiore rispetto alle altre due configurazioni. Questa configurazione illustra l’impatto negativo di una distribuzione altamente non-IID, in cui i client non riescono a fornire informazioni adeguate su tutte le classi al modello globale, causando un addestramento non bilanciato che limita fortemente la generalizzazione del modello.

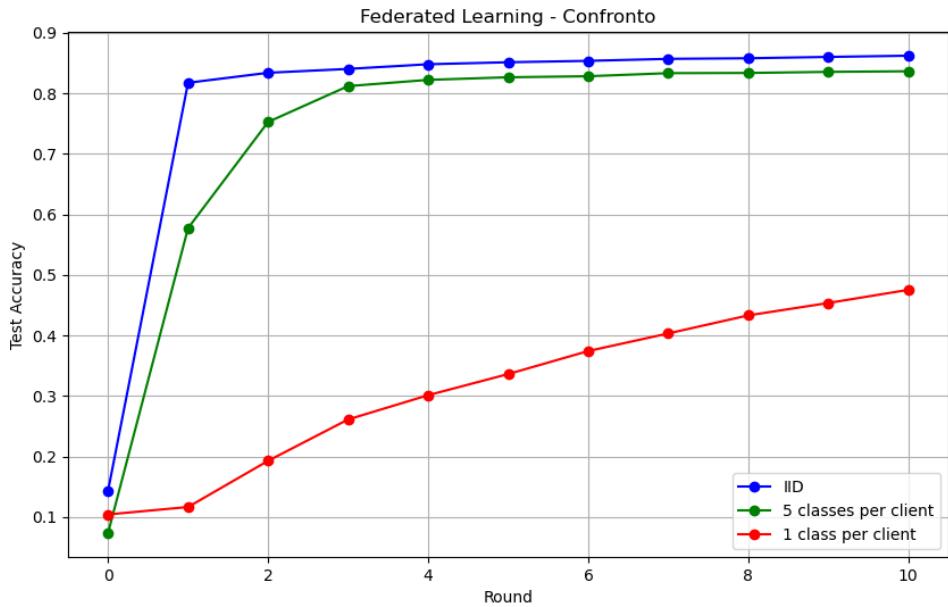


Figure 13: Grafico di confronto dei risultati sperimentali tra le tre diverse tipologie di partizionamento del dataset

Come si può vedere dalla Figura 13 la distribuzione dei dati tra i client incida significativamente sull'efficienza e la performance finale dell'addestramento federato. Se la configurazione **IID** risulta la più performante, grazie alla distribuzione equilibrata e completa di tutte le classi per ogni client, la versione con **5 classi per client** rappresenta un buon compresso: sebbene più lenta nei brevi periodi, permette al modello di raggiungere un'accuracy simile a IID nei round successivi. Infine, La configurazione che presenta una distribuzione con **1 classe per client** penalizza pesantemente l'addestramento e mostra chiaramente come una distribuzione non-IID estrema comprometta la capacità del modello di generalizzare.

Come pronosticato nella Sezione 2.3 i risultati sperimentali mostrato come una distribuzione dei dati omogenea sia superiore ad una eterogenea nell'ambito di un allenamento di un modello di ML allenato tramite Federated Learning.

Conclusioni

La tecnologia del **Federated Learning** può ricoprire un ruolo importante nella decentralizzazione dei dati e ell'apprendimento supervisionato del mondo del machine learning tramite l'utilizzo delle reti neurali. Ha dimostrato di essere una tecnica promettente per preservare la privacy degli utenti durante l'addestramento di modelli su larga scala, poiché i dati non vengono mai centralizzati ma rimangono sui dispositivi degli utenti. Tuttavia, mentre questo approccio riduce i rischi di esposizione dei dati, non garantisce una protezione totale contro potenziali attacchi come il reconstruction attack o il membership inference attack, che possono dedurre informazioni sui dati degli utenti a partire dai modelli condivisi. Inoltre come mostrato dai risultati sperimentali le limitazioni relative all'eterogenità dei dati sono evidenti.

Differential Privacy e Secure Multi-Party Computation

Per rafforzare ulteriormente la privacy, l'integrazione di tecniche avanzate come la Differential Privacy (DP) e la Secure Multi-Party Computation (SMPC) offre un interessante orizzonte di ricerca. La Differential Privacy aggiunge rumore ai parametri del modello per rendere più difficile identificare i dati di un singolo utente, mentre la Secure Multi-Party Computation consente ai client di collaborare senza mai rivelare i propri dati grezzi al server centrale, grazie a tecniche di cifratura e aggregazione sicura. La combinazione di queste due metodologie può migliorare significativamente la protezione dei dati, garantendo che le informazioni personali degli utenti siano protette durante il processo di apprendimento federato.

Le tecniche di Differential Privacy e Secure Multi-Party Computation si applicano particolarmente bene agli algoritmi sincroni, come FedAvg, dove i client aggiornano il modello in contemporanea e i parametri vengono aggregati dal server. Questo approccio sincrono facilita l'integrazione di rumore differenziale e la crittografia, poiché consente un controllo centralizzato sul processo di aggregazione. Tuttavia, in scenari asincroni, dove gli aggiornamenti arrivano in momenti diversi, l'applicazione di queste tecniche diventa più complessa, richiedendo nuove soluzioni per gestire la sincronizzazione e mantenere alte le garanzie di privacy.

Nonostante i progressi, l'uso di Federated Learning in combinazione con tecniche di Differential Privacy e SMPC deve essere ulteriormente esplorato per ottimizzare il trade-off tra privacy, accuratezza del modello e costo computazionale. In particolare, l'efficacia di queste tecniche in scenari globali e distribuiti su larga scala rimane una sfida aperta. Le direzioni future della ricerca si concentreranno sull'integrazione di esse per ottenere un modello di apprendimento federato che possa bilanciare sicurezza, scalabilità e performance, portando il FL a un livello di protezione più robusto per le applicazioni future, soprattutto in contesti in cui la privacy e la riservatezza dei dati sono prioritarie.

Scalabilità del Federated Average Algorithm

Quanto mostrato in precedenza nei capitoli di Implementazione e Risultati sperimentali non rispecchia chiaramente quello che sarebbe una distriuzione del *Federated Average Algorithm* a livello globale per diversi fattori come: numero di client utilizzati, database di addestramento preso in considerazione, semplice simulazione della comunicazione, sequenzialità e non parallelismo del client. Ci fornisce però la base per poter trarre le giuste osservazioni in caso in cui il modello volesse essere allargato:

1. **Scenari Bilanciati (IID):** Se il FL fosse eseguito in contesti globali dove i dati sono più o meno bilanciati (come nel caso IID), i benefici di avere un dataset distribuito uniformemente e rappresentativo per ciascun client sarebbero evidenti. I modelli globali avrebbero la possibilità di convergere più rapidamente e con maggiore precisione. Ad esempio, in scenari come la raccolta di dati da diverse regioni geografiche con una varietà di comportamenti umani, sarebbe vantaggioso avere un bilanciamento delle classi.
2. **Scenari Sbilanciati (Non-IID):** Quando i dati sono sbilanciati, come nei casi del 5 classi per client e 1 classe per client, la simulazione mostra una convergenza più lenta e una precisione inferiore. In scenari globali reali, questo potrebbe essere un riflesso delle situazioni in cui i dispositivi dei clienti (ad esempio, smartphone o sensori) raccolgono dati molto specifici e rari, rendendo difficile per il modello globale fare progressi rapidi. In un contesto globale, la diversità dei dati dei client potrebbe richiedere approcci di aggregazione più sofisticati (come l'uso di tecniche di federated learning avanzate, ad esempio basate su clusterizzazione dei dati o tecniche di personalizzazione locale).
3. **Scalabilità e Diversità dei Dati:** su scala globale, la distribuzione dei dati tra i dispositivi (client) sarebbe inevitabilmente estremamente non uniforme (non IID), amplificando il problema emerso nei casi con distribuzione di una sola classe per client: Il modello globale faticherà inizialmente a generalizzare, tuttavia, la collaborazione progressiva tra client permetterà un apprendimento incrementale, specialmente se la partecipazione di diversi tipi di client aumenta round dopo round. Quindi in un contesto generale, sarebbe cruciale implementare strategie che favoriscano l'equità, garantendo che i dati di ogni client contribuiscano proporzionalmente al modello.
4. **Disparità tra i Client:** a livello globale i client hanno capacità eterogenee in termini di potenza computazionale e connettività. Per affrontare questo problema su larga scala, si potrebbero utilizzare tecniche di ponderazione o adattamento dei contributi, come: Federated Averaging Ponderato con lo scopo di dare più peso ai client con dati più rappresentativi o Personalizzazione dei Modelli, creando versioni locali adattate ai dati di ogni client.

5. Efficienza e Comunicazione: la crescita progressiva osservata suggerisce l'importanza di tecniche per accelerare l'apprendimento, ottimizzando il bilancio tra accuratezza e costi di comunicazione.

Considerazioni finali

In conclusione, il Federated Learning rappresenta una tecnologia chiave per affrontare le sfide legate alla privacy e alla distribuzione dei dati nel contesto dell'apprendimento automatico. I vantaggi derivanti dal mantenere i dati localmente sui dispositivi degli utenti sono significativi, specialmente in settori dove la riservatezza delle informazioni è una priorità, come la sanità, le telecomunicazioni e i servizi finanziari. Tuttavia, il percorso per rendere il Federated Learning una soluzione largamente adottabile su scala globale è ancora ricco di sfide.

Tra le aree che richiedono ulteriore approfondimento, l'integrazione di tecniche avanzate come la Differential Privacy e la Secure Multi-Party Computation emerge come un passaggio cruciale per garantire la sicurezza dei dati durante l'addestramento distribuito. Al tempo stesso, è essenziale trovare un equilibrio ottimale tra protezione della privacy, accuratezza dei modelli e costi computazionali, specialmente in scenari caratterizzati da eterogeneità nei dispositivi client e dalla non uniformità dei dati (non-IID).

La scalabilità del Federated Learning rimane un punto di attenzione, poiché i risultati sperimentali indicano che l'efficienza e l'efficacia degli algoritmi di aggregazione, come il Federated Average Algorithm, possono variare significativamente in base alla distribuzione dei dati e alle condizioni operative. Il futuro del FL richiederà lo sviluppo di metodologie capaci di gestire la diversità delle condizioni globali, comprese le disparità tra client in termini di risorse computazionali e qualità della connessione.

Infine, è necessario continuare a esplorare tecniche innovative per migliorare l'equità e l'efficienza della collaborazione tra client, favorendo una maggiore inclusione di dispositivi eterogenei e garantendo che ogni client possa contribuire in modo significativo al modello globale. Questo è particolarmente importante in applicazioni critiche, dove l'accuratezza del modello deve essere mantenuta elevata nonostante la complessità del contesto operativo.

Il Federated Learning ha già dimostrato un potenziale significativo, ma è chiaro che ulteriori progressi tecnici e sperimentali saranno necessari per affrontare le sfide ancora aperte. La combinazione di ricerca accademica e innovazione industriale sarà determinante per portare questa tecnologia a maturità, aprendo la strada a nuovi scenari di applicazione e contribuendo a creare un ecosistema di apprendimento automatico più sicuro, distribuito ed efficiente.

Bibliografia

References

- [1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016.
- [2] IBM. *ibm*. URL: <https://www.ibm.com/topics/supervised-learning>.
- [3] İlyurek Kılıç. *The Mathematics Behind Optimization: Gradient Descent*. URL: <https://medium.com/@ilyurek/the-mathematics-behind-optimization-gradient-descent-26413cadb7d5>.
- [4] Yann LeCun et al. “Gradient-Based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE* (1998).
- [5] MathWorks. *Esempio di una rete con numerosi layer convoluzionali. A ciascuna immagine di addestramento vengono applicati dei filtri a diverse risoluzioni e l'output di ogni immagine convoluta viene utilizzato come input per il layer successivo*. URL: <https://it.mathworks.com/discovery/convolutional-neural-network.html>.
- [6] H. Brendan McMahan et al. “Communication-Efficient Learning of Deep Networks from Decentralized Data”. In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2017.
- [7] Michael A. Nielsen. *Neural Networks and Deep Learning*. 2015. URL: <http://neuralnetworksanddeeplearning.com>.