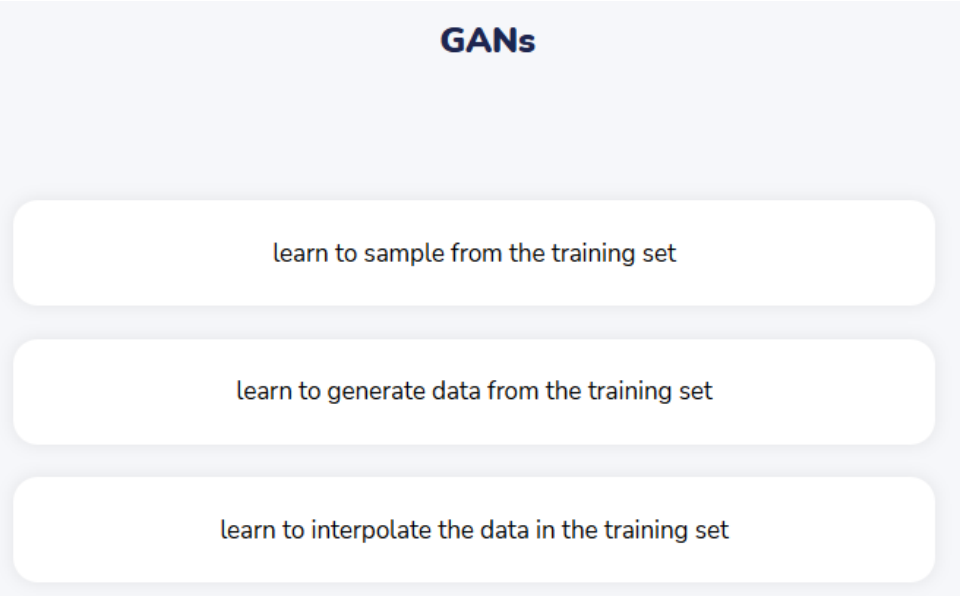
Lezione 12 - Federated learning - 10/12/2024



learn to sample from the training set. Impara a come associare ad ogni random vector, un'immagine che è della stessa distribuzione del training set.

Non sta generando data dal training set, ma sta generando dati che NON sono nel training set, altrimenti staremmo ricreando il training set.

GANs are composed of two models called

Congratulations!

- Interpolator
- Classificator
- Discriminator ✓
- Creator
- Generator ✓
- Inventor

The key layer in the generator is

Congratulations!

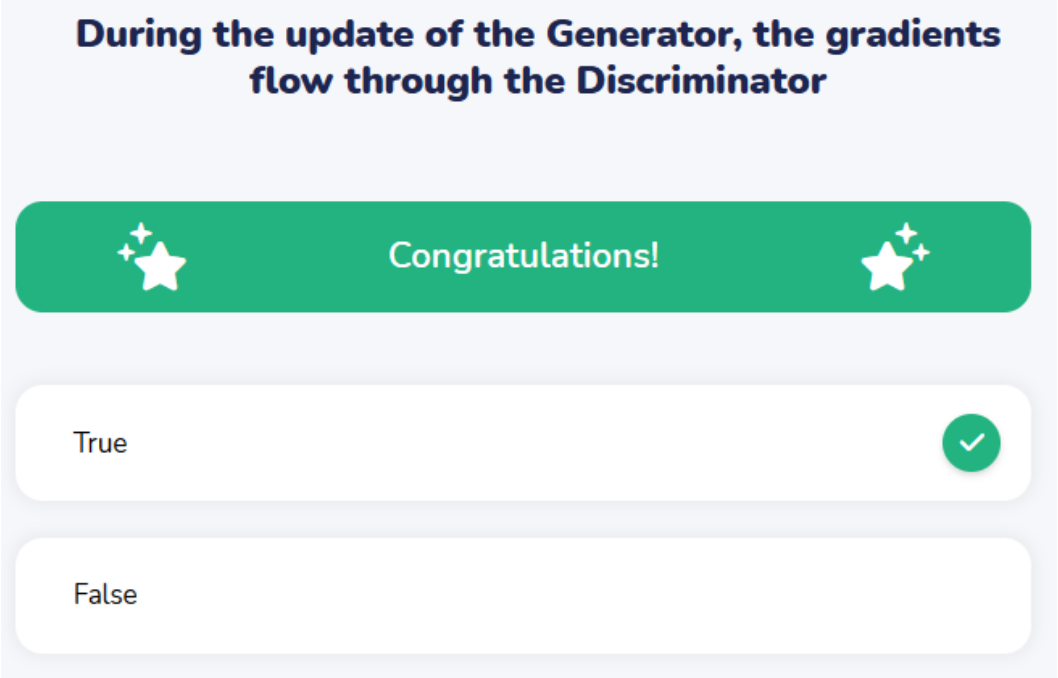
- Canonical convolution
- Inverted convolution
- Interpolated convolution
- Transposed convolution ✓
- Dilated convolution

During the update of the Discriminator, the gradients flow through the Generator

Congratulations!

- True
- False ✓

Il discriminatore vuole distinguere i samples reali da quelli generali, quindi prende come dati i samples reali e quelli generati. Quindi i gradienti si fermano, quando raggiungono l'input, non vanno nel generatore.

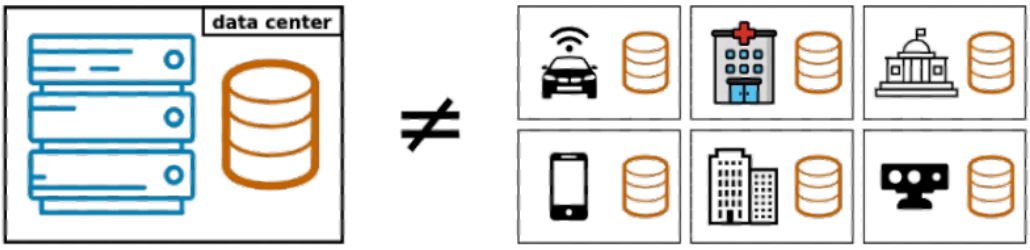


Alla fine del generatore, non abbiamo una ground truth da comparare, quindi mettendo il discriminatore dopo il generatore, abbiamo una ground truth perchè sappiamo quali samples sono reali e quali fake, quindi abbiamo una loss che possiamo backpropagare.

Federated learning

Nel federated learning abbiamo un cambio di paradigma, cambia il modo in cui abbiamo i dati di training. Fin ora abbiamo sempre avuto dati centralizzati, ovvero la macchina che fa il training ha tutti i dati.

Però spesso i dati sono decentralizzati.



Però perchè non centralizziamo i dati, per tornare nella stessa configurazione che abbiamo visto fin ora? Perchè non mandiamo tutto ad un'unità centrale?

Ci sono alcuni problemi:

- Inviare i dati potrebbe essere troppo costoso. Per esempio se abbiamo le videocamere di una macchina autonoma, questi sono molti dati.
- Devices wireless potrebbero avere banda o potenza limitata, e quindi non possono mandare i dati.
- I dati possono essere sensibili e quindi per privacy non possono essere inviati.
- Mantenere il controllo dei dati può dare un vantaggio competitivo per il business o per la ricerca. Per esempio nel problema di face recognition, qualche anno fa i dataset esistenti non erano molto grande, 1k immagini... poi un giorno facebook ha creato il suo dataset usando i dati degli utenti. Questi dati non sono mai stati rilasciati, per privacy e per mantenere un vantaggio competitivo per questi tipi di task.

Perchè non posso allenare su ciascun device sui dati locali?

- Può succedere che i dataset locali siano troppo piccoli, quindi abbiamo performance molto basse, perchè non abbiamo abbastanza dati.
- Potrebbero essere dei risultati non statisticamente significativi, per esempio se devo fare degli studi medici, non posso usare i dati di una persona sola.
- Il dataset locale potrebbe avere un bias, nel senso che non è rappresentativo della distribuzione dei target. Per esempio nelle self driving cars, in milano, se raccolgo dati mentre guido, incontrerò sempre strade di città, edifici... però un giorno voglio andare in campagna, in condizioni di strada molto diverse da quelle che il modello ha finito.

Il **federated learning** cerca di trainare in un modo **collaborativo**, mantenendo i dati decentralizzati.

Qualsiasi modello può essere allenato in questo modo.

Se abbiamo un server centrale e un tot di devices, come possiamo fare?

Innanzitutto, il server deve **inizializzare** il modello, inizializzando i pesi (importante visto che il modello è stocastico, dipende dall'inizializzazione), e poi invia il modello inizializzato alle diverse parti (devices).

Ciascun device fa un **update del modello**, in base al dataset locale.

Quindi l'architettura del modello è la stessa, ma i pesi saranno diversi perché vengono applicati dati diversi.

Le diverse parti quindi manderanno al server il modello che hanno creato. Il server farà un'aggregazione dei diversi modelli, e quando ha computato l'aggregazione, lo invierà ai devices.

E quindi l'aggregazione è iterata. Ci sarà un nuovo update, che verrà rimandato al server, che farà l'aggregazione, e che rimanderà il modello alle parti.

Quindi le diverse parti sono esposte a tutti i dati, non tramite i dati stessi, ma tramite l'effetto che i dati hanno sui pesi.

L'unica cosa che viene comunicata è il modello, non i dati.

Tutto questo ha senso **se non possiamo recuperare i dati**, usando i pesi dei modelli. Perchè se i modelli sono **invertibili**, allora l'invio dei pesi sarebbe la stessa cosa di inviare i dati.

Alla fine, vorremmo che il modello allenato in questo modo, possa raggiungere la performance del modello centralizzato che allena il modello nel modo classico, con tutti i dati.

Vogliamo che il modello sia meglio di quello che potrebbe fare ciascun device usando solamente i dati locali.

Siamo nel mezzo di questi due casi di solito.

Differenze tra distributed learning e federated learning

Nel **distributed learning**, all'inizio i dati sono salvati centralmente, come in un datacenter, e l'obiettivo è solo di fare il training più velocemente. Quindi al posto di fare tutti i batches nel computer centrale, mandiamo diversi batches a diversi computer che li calcolano in modo parallelo.

Di solito abbiamo che i dati sono distribuiti uniformemente e randomicamente ai workers.

Quindi serve per salvare training time.

Nel **federated learning** invece i dati sono naturalmente distribuiti, e generati localmente. I dati non sono indipendenti e non sono identically distributed, e di solito sono anche sbilanciati.

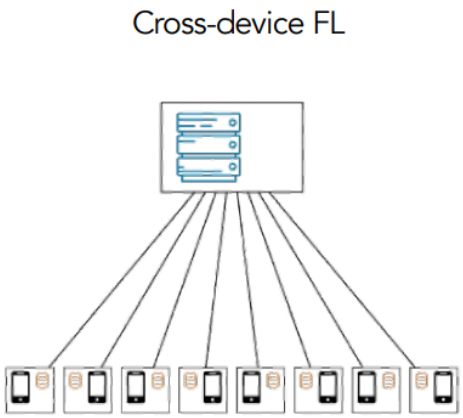
Challenges nel federated learning:

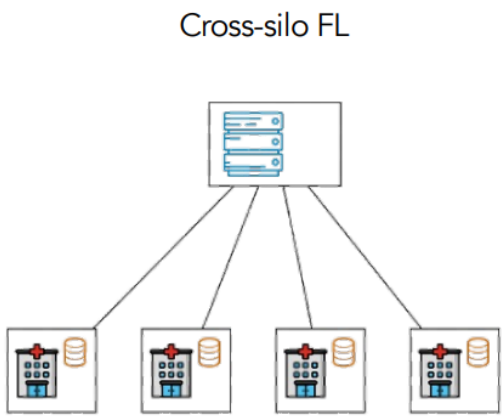
- privacy
- avere a che fare con la possibilità che i partecipanti non siano sempre disponibili
- possono esserci degli attacchi che inviano dati sbagliati, per attaccare il modello

Cross-device vs cross-silo federated learning

In **cross-devices** abbiamo un numero di parti enorme (milioni, miliardi) e dataset molto piccoli per parte (anche 1, 1 selfie). In particolare, ci possiamo aspettare di avere un'availability e una reliability limitata, perchè il device può essere spento, in modalità aereo, irraggiungibile... e ci possono anche essere devices che fanno attacchi sul modello.

Nel **cross-silo** abbiamo un numero di parti limitato (2-100), abbiamo devices reliable, che sono quasi sempre available, e tipicamente onesti (arrivano dati buoni).

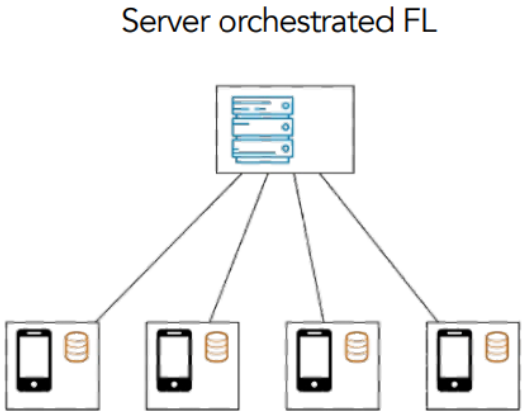


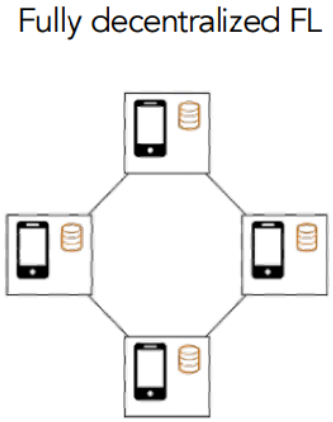


Server orchestrated vs fully decentralized federated learning

Nel **server orchestrated federated learning** abbiamo la comunicazione server-client tipica, abbiamo il server che fa la coordinazione e l'aggregazione globale, e può succedere che il server sia il bottleneck, è anche il single point of failure.

Nel **fully decentralized federated learning** abbiamo comunicazione device to device, quindi facciamo aggregazioni locali. Questo scala in modo migliore con un numero di devices alto.





Aspetti storici

Il termine federated learning è stato introdotto nel 2016 da ricercatori google, nel 2020 c'erano già più di 1000 papers solo nella prima metà dell'anno... c'è una crescita rapida su questo topic.

Diversi framework open-source sono in fase di sviluppo: PySyft, TensorFlow Federated, FATE, Flower, Substra...

Non è solo machine learning, è anche numerical optimization, privacy & security, networks, systems, hardware... ci sono molti aspetti in questo approccio.

FedAvg

Innanzitutto, consideriamo un insieme di K parti.

Ogni parte k del dataset D_k ha una cardinalità di n_k.

Let $\mathcal{D} = \mathcal{D}_1 \cup \cdots \cup \mathcal{D}_K$ be the joint dataset and $n = \sum_k n_k$ the total number of points

D è l'unione di tutti i dataset e la cardinalità totale è la somma della cardinalità dei dataset locali n.

Vogliamo trovare una configurazione dei pesi che minimizza la loss su tutti i dati. Però non abbiamo un modello che vede tutti i dati, ma invece abbiamo la somma tra i diversi subsets, che sono combinati... la somma è pesata, le parti che hanno più dati avranno un peso più grande.

- We want to solve problems of the form $\min_{\theta \in \mathbb{R}^p} F(\theta; \mathcal{D})$ where:

  $F(\theta; \mathcal{D}) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(\theta; \mathcal{D}_k)$

  and

  $F_k(\theta; \mathcal{D}_k) = \sum_{d \in \mathcal{D}_k} f(\theta; d)$

In generale teta sono i parametri del modello.

$\theta \in \mathbb{R}^p$ are model parameters (e.g., weights of a logistic regression or neural network)

Questo tipo di notazione copre un'ampia classe di problemi di apprendimento automatico formulati come minimizzazione del rischio empirico.

Pseudocode FedAvg

Abbiamo il server side a sinistra e client side a destra.

Partendo dal **server side**. Stiamo creando la forma più generale di FedAvg, come se avessimo migliaia di parti. Per non perdere tempo nelle comunicazione, può essere che ogni volta non stiamo combinando tutti i modelli delle migliaia di parti, ma invece selezioniamo in modo random un parte di loro. Questo dipende dal client sampling rate 'ro'.

Il primo step è l'inizializzazione dei pesi del modello. Poi selezioniamo quanti rounds di aggregazione vogliamo fare. Per ogni round, selezioniamo dal numero totale di clients K una parte random, e questo è il set di clients per il round t. Quindi ogni client di questo set, in parallelo, performerà l'update del modello (mandiamo il modello al client che fa l'update, e ci rimanda i pesi aggiornati).

Alla fine facciamo l'aggregazione, facendo una media dei parametri che abbiamo su ciascun client, usando la frazione, che sono quei pesi che danno più importanza ai pesi che sono computati dalle parti che hanno più dati.

Quindi il modello è aggiornato, si va al prossimo round e si ripete.

Algorithm FedAvg (server-side)

Parameters: client sampling rate ρ

initialize θ
for each round $t = 0, 1, \dots$ **do**
  $\mathcal{S}_t \leftarrow$ random set of $m = \lceil \rho K \rceil$ clients
  for each client $k \in \mathcal{S}_t$ in parallel **do**
    $\theta_k \leftarrow \text{ClientUpdate}(k, \theta)$
  $\theta \leftarrow \sum_{k \in \mathcal{S}_t} \frac{n_k}{n} \theta_k$

Algorithm ClientUpdate(k, θ)

Parameters: batch size B, number of local steps L, learning rate η

for each local step $1, \dots, L$ **do**
  $\mathcal{B} \leftarrow$ mini-batch of B examples from $\mathcal{D}_k$
  $\theta \leftarrow \theta - \frac{n_k}{B} \eta \sum_{d \in \mathcal{B}} \nabla f(\theta; d)$
send θ to server

A destra invece vediamo il **client side**.

Ciascun client ha potenzialmente il proprio batch size B. Abbiamo un numero di steps locali che vengono fatti, e il learning rate.

In ciascun update del client, possiamo fare multiple iterazioni, possiamo scegliere quante (anche solo 1 volendo, o multiple, prima di ritornare i pesi).

Per ogni step andiamo a prendere un sample di mini batch dal dataset locale, e andiamo ad aggiornare i pesi del modello con la classica gradient descent. Dopo che abbiamo finito tutti gli steps locali, rimandiamo i pesi aggiornati al server.

- For L = 1 and **ρ** = 1, it is equivalent to classic parallel SGD: updates are aggregated and the model synchronized at each step
- For L > 1: each client performs multiple local SGD steps before communicating

Se consideriamo colo 1 local step, e usiamo ro=1 quindi usiamo tutti i devices, questo è la stessa cosa del SGD parallelo. Questo è molto inteso dal punto di vista delle comunicazioni, perchè tutte le parti devono essere reliable, e facciamo tante comunicazioni.

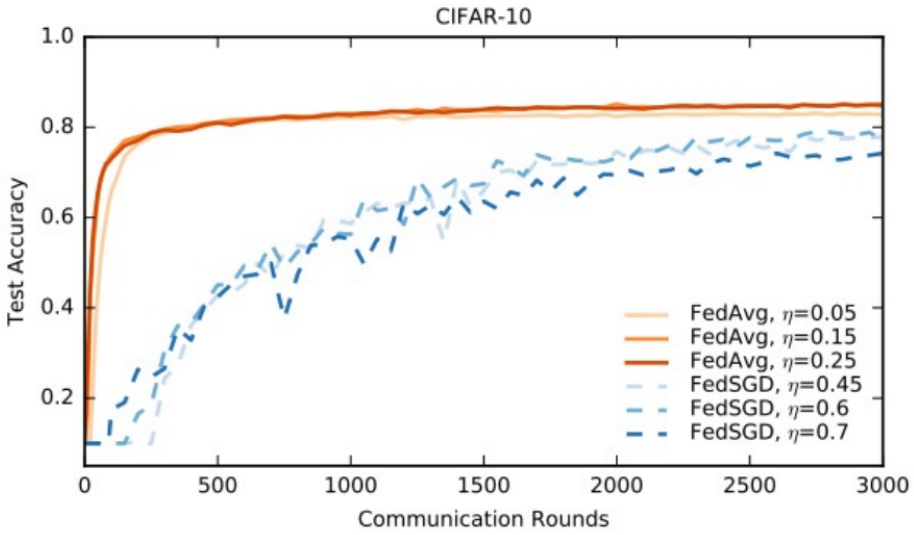
Quando invece aumentiamo il numero di local steps, abbiamo che ciascun client fa più steps di local stochastic gradient descent, prima di rimandare i pesi al server. In questo modo possiamo ridurre le comunicazioni tra client e sever.

Le comunicazioni sono di solito il bottleneck del federated learning, si perde tanto tempo a fare le comunicazioni.

FedAvg con L > 1 raggiunge una generalizzazione migliore empiricamente, rispetto alla SGD parallela con mini-batch.

- Convergence to the optimal model can be guaranteed for i.i.d. data [Stich, 2019] [Woodworth et al., 2020] but issues arise in strongly non-i.i.d. cases

Questo è il plot dopo rounds di comunicazione. In rosso vediamo FedAvg con parametri diversi, e in blu FedSGD.



Se immaginiamo che alexnet a 16 milioni di parametri... che possono essere compressi a 10mb (model compression), se facciamo 500 rounds, usiamo molti dati.

Abbiamo detto che i dati potrebbero essere molto non i.i.d. (**indipendent and identically distributed**), i dati possono avere bias, per come sono generati. Imparare da questo tipo di distribuzione dati è difficile e lento, perché ogni parte vuole che il modello vada in una direzione diversa.

Se le distribuzioni sono molto diverse tra le diverse parti, imparare il modello può essere fatto, però abbiamo bisogno di moltissimi parametri, perchè il modello deve avere la capacità per imparare che diverse parti possono avere diversi dati.

Un altro approccio per **affrontare i dati non i.i.d.** consiste nel rimuovere il vincolo che il modello appreso debba essere lo stesso per tutte le parti ("una soluzione unica per tutti"). L'idea è quella che, per esempio se abbiamo uno smartwatch, quando è nuovo riconosce dei patterns per la camminata, per la corsa... però può essere che il modo in cui io corro sia diversa rispetto a come corre qualcun altro e quindi potrebbe non essere rilevata in modo corretto. é per questo che io posso selezionare "sto correndo", di modo che i dati vengano utilizzati per customizzare il modello. Però non voglio usare solo i miei dati, voglio collaborare con dati che arrivano da altri devices. Ci sono **2 modi** famosi per fare questo.

Proponiamo di **regolarizzare i modelli personalizzati**, sulla media di tutti i modelli. Cioè vogliamo che i modelli personalizzati sono siano molto lontano rispetto alla media dei parametri sugli altri modelli. Quindi si aggiunge un penalty term al modello, in base a quanto sono distanti i nostri pesi rispetto alla media degli altri.

$$F(\theta_1, \ldots, \theta_K; \mathcal{D}) = \frac{1}{K} \sum_{k=1}^{K} F_k(\theta_k; \mathcal{D}_k) + \frac{\lambda}{2K} \sum_{k=1}^{K} \left\| \theta_k - \frac{1}{K} \sum_{l=1}^{K} \theta_l \right\|^2$$

Il secondo approccio è ispirato **dal meta-learning**. L'idea è di imparare un modello globale che può essere adattato facilmente a ciascuna parte (device). Ovvero, che in poche iterazioni si può adattare il modello. Quindi stiamo imparando una parametrizzazione del modello generale (meta model) che può essere aggiornata facilmente per specializzare la task.

$$F(\theta; \mathcal{D}) = \frac{1}{K} \sum_{k=1}^{K} F_k(\theta - \alpha \nabla F_k(\theta); \mathcal{D}_k)$$

Può essere provato che queste formulazioni sono relazionate (e al FedAvg).

Una modifica di questi metodi potrebbe essere una che riduce il peso degli outliers. Per esempio se al posto di fare una media (nell'ultimo step del server-side), facessimo una mediana... o comunque una statistica più robusta agli outliers, saremmo più resistenti agli attacchi.