

ALMA MATER STUDIORUM · UNIVERSITÀ DI  
BOLOGNA CAMPUS DI CESENA

---

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA  
CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

**TECNICHE PARALLELE DI  
MACHINE LEARNING  
APPLICATE A CONTAINER**

**Relatore:**  
Ghini Vittorio

**Presentata da:**  
Stricescu Razvan Ciprian

**Anno Accademico 2021/2022**

*Alla mia famiglia che ha sempre sostenuto le mie passioni ed il mio  
percorso accademico*

# Indice

<b>1</b>	<b>Introduzione</b>	<b>4</b>
1.1	Machine learning, container e dati . . . . .	4
1.2	Obiettivo . . . . .	4
1.3	Virtualizzazione . . . . .	4
1.4	Container . . . . .	5
1.4.1	Reti e container . . . . .	7
1.4.2	Volumi e container . . . . .	8
1.5	Docker compose . . . . .	8
1.6	Docker . . . . .	9
<b>2</b>	<b>Parallelismo dei dati</b>	<b>10</b>
2.1	Discesa del gradiente . . . . .	11
2.2	Data parallelism . . . . .	12
2.3	Model parallelism . . . . .	13
2.4	Quale metodo usare? . . . . .	13
<b>3</b>	<b>Studio pratico sul parallelismo</b>	<b>15</b>
3.1	Introduzione alla metodologia . . . . .	15
3.2	Tecnologie usate . . . . .	16
3.2.1	Tensorflow . . . . .	16
3.2.1.1	Keras . . . . .	17
3.2.2	Mesh Tensorflow . . . . .	17
3.2.3	GitHub Copilot . . . . .	19
3.3	Synchronous data parallelism . . . . .	20
3.3.1	Creazione del dataset . . . . .	20
3.3.2	Creazione del modello . . . . .	22
3.3.3	Creazione della strategia . . . . .	23
3.3.4	Come funziona MultiWorkerMirroredStrategy? . . . . .	23
3.3.5	Gestione dei container . . . . .	25
3.3.6	Conclusioni pratiche . . . . .	26
3.4	Asynchronous data parallelism . . . . .	28
3.4.1	Creazione del dataset . . . . .	28
3.4.2	Creazione della strategia . . . . .	28
3.4.3	Come funziona ParameterServerStrategy? . . . . .	30

3.4.4	Gestione dei container . . . . .	31
3.4.5	Conclusioni pratiche . . . . .	32
3.5	Data parallelism applicato ad un dataset e modello diverso . . .	34
3.6	Model parallelism . . . . .	36
<b>4</b>	<b>Conclusioni e sviluppi futuri</b>	<b>38</b>
4.1	Conclusioni sul data parallelism . . . . .	38
4.2	Sviluppi futuri . . . . .	39

# Capitolo 1

## Introduzione

### 1.1 Machine learning, container e dati

L'obiettivo di questa tesi è l'estensione della conoscenza di un argomento già ampiamente conosciuto e ricercato. Questo lavoro focalizza la propria attenzione su una nicchia dell'ampio mondo della virtualizzazione, del machine learning e delle tecniche di apprendimento parallelo. Nella prima parte verranno spiegati alcuni concetti teorici chiave per la virtualizzazione, ponendo una maggior attenzione verso argomenti di maggior importanza per questo lavoro. La seconda parte si propone di illustrare, in modo teorico, le tecniche usate nelle fasi di training di reti neurali. La terza parte, attraverso una parte progettuale, analizza le diverse tecniche individuate applicandole ad un ambiente containerizzato.

### 1.2 Obiettivo

Una volta introdotti alcuni concetti chiave e la teoria dietro alle tecniche di parallelizzazione nel machine learning, l'obiettivo è quello di trovare possibili soluzioni applicate a container. Più nello specifico lo scopo non è quello di creare nuove soluzioni originali ma bensì quello di implementare soluzioni già esistenti attraverso l'uso di container. Normalmente queste tecniche vengono usate in ambienti molto grandi, molteplici macchine che lavorano insieme, e molto potenti, calcoli su GPU o pod TPU. L'obiettivo è quello di comprendere come poter applicare tali metodi a container e capire i vari i possibili casi d'uso, pregi e difetti, oltre al come creare e gestire sia la soluzione che i container da un punto di vista pratico.

### 1.3 Virtualizzazione

Una delle parole chiave, rappresenta l'astrazione di qualcosa di fisico in qualcosa di intangibile, virtuale. Più nello specifico, in ambito informatico, si riferisce ad

un processo che astrae le componenti fisiche di un computer, l'hardware, al fine di renderle disponibili in forma di risorsa virtuale. Queste risorse possono essere unità di archiviazione, reti, server, applicazioni o semplicemente potere computazionale. Tutto questo è reso possibile dal Hypervisor. L'Hypervisor può essere descritto come un software che si colloca sulla macchina fisica, chiamata anche host. Raggruppa le risorse del host e le alloca alle macchine virtuali.

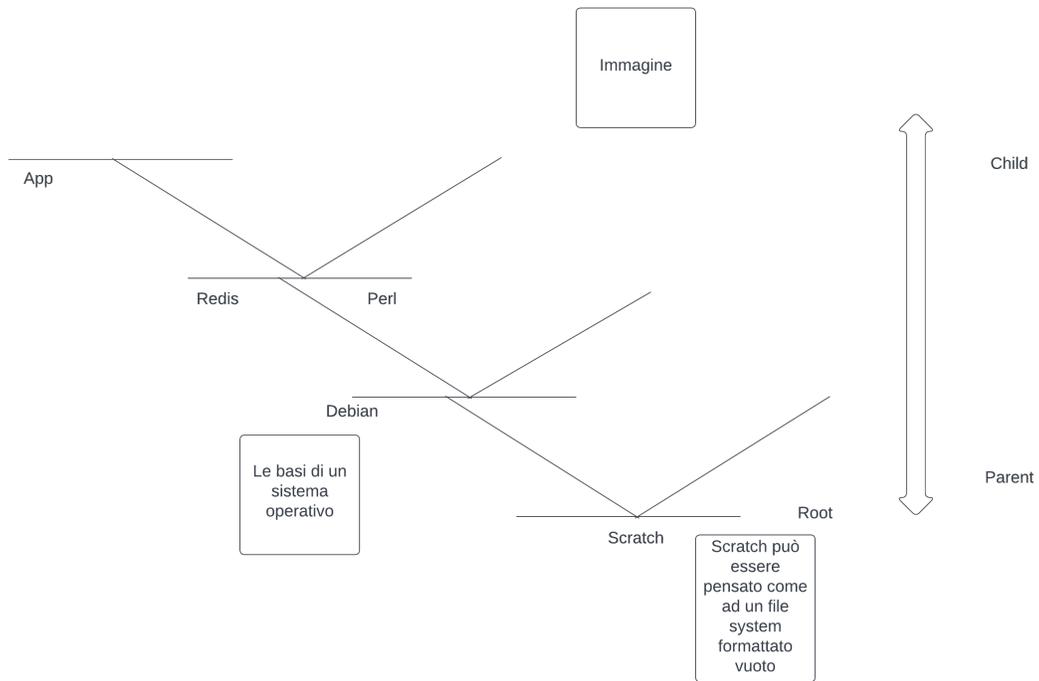
Esistono diverse micro e macro categorie di tipi di virtualizzazione ma la tipologia presa in considerazione in questa tesi è la virtualizzazione a livello sistema operativo. A differenza di un tipo di virtualizzazione a livello hardware in cui viene "costruita" una CPU virtuale, simile alla CPU fisica, nell'OS-level virtualization vengono istanziate delle "partizioni" del sistema operativo in uso sulla macchina fisica.

Generalmente parlando l'OS-level ha un solo kernel e svariate istanze chiamate contenitori mentre nella virtualizzazione hardware i vari OS vengono eseguiti in modo concorrente a quello fisico, anche in maniera dissimile.

## 1.4 Container

Un container nella sua nozione più basilare è un "sandbox", un involucro che viene usato per poter isolare processi all'interno della macchina. Per sandbox si intende un ambiente isolato con il proprio namespace, c-groups e processi. Un container nella definizione classica è completamente legato al tempo di vita del processo isolato, dunque quando un container viene "spento" i suoi processi finiscono anch'essi e viceversa, però non è vincolato ad un solo processo, vi possono essere vari processi concorrenti nello stesso container. In quanto gran parte del mondo container fa riferimento a Docker come piattaforma molti esempi e terminologie faranno riferimento a esso.

Quando si parla di container spesso si confonde la parola stessa con la sua "immagine", una rappresentazione binaria dentro al file system della macchina che contiene alcuni stati. L'immagine di un container è composta da librerie di sistema, strumenti e altre impostazioni necessarie per l'esecuzione di un programma software su una piattaforma di containerizzazione. Inoltre, l'immagine condivide il kernel del sistema operativo della macchina host e ha una struttura gerarchica ad albero molto simile alla nozione di snapshot. Si può pensare all'immagine di un container come una struttura a livelli in cui ogni livello rappresenta una base da cui si può far partire un container, come un template su cui costruirci sopra. Con questo sistema modulare è possibile condividere e riutilizzare immagini in modo consistente e uniforme.



Esempio di struttura gerarchica di un immagine.

Per costruire immagini e dunque container viene spesso usato un Dockerfile, un ambiente sotto forma di file di testo. Nel Dockerfile vengono definiti tutti i comandi che un utente chiamerebbe a linea di comando per assemblare un immagine, costruendo di fatto container in modo semi-automatico.

```

# syntax=docker/dockerfile:1
FROM golang:1.16-alpine AS build

# Install tools required for project
# Run `docker build --no-cache .` to update dependencies
RUN apk add --no-cache git
RUN go get github.com/golang/dep/cmd/dep

# List project dependencies with Gopkg.toml and Gopkg.lock
# These layers are only re-built when Gopkg files are updated
COPY Gopkg.lock Gopkg.toml /go/src/project/
WORKDIR /go/src/project/
# Install library dependencies
RUN dep ensure -vendor-only

# Copy the entire project and build it
# This layer is rebuilt when a file changes in the project directory
COPY . /go/src/project/
RUN go build -o /bin/project

# This results in a single layer image
FROM scratch
COPY --from=build /bin/project /bin/project
ENTRYPOINT ["/bin/project"]
CMD ["--help"]

```

Esempio di un Dockerfile preso direttamente da [https://docs.docker.com/develop/develop-images/dockerfile\\_best-practices/\[2\]](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/[2])

Un container di base contiene già tutte le proprie dipendenze, comportamento insolito per gran parte delle applicazioni. Quando un container viene fatto partire su un sistema operativo di fatto non viene installato niente, ma esiste già sopra il sistema operativo nella propria "bolla". È possibile dunque far funzionare un indefinito numero di versioni differenti di applicazioni che richiedono versioni differenti di librerie, tutto questo sopra un solo sistema operativo. Una volta cancellate le immagini tabula rasa, la macchina è nel suo stato di partenza. Usare senza inquinare.

### 1.4.1 Reti e container

Un aspetto fondamentale del mondo è la comunicazione, il poter scambiare e capire dei messaggi che siano testuali o meno. Nel mondo dei container non è da meno. Principalmente i container possono parlare tra di loro, con l'host o il mondo esterno attraverso l'ausilio di networks, reti.

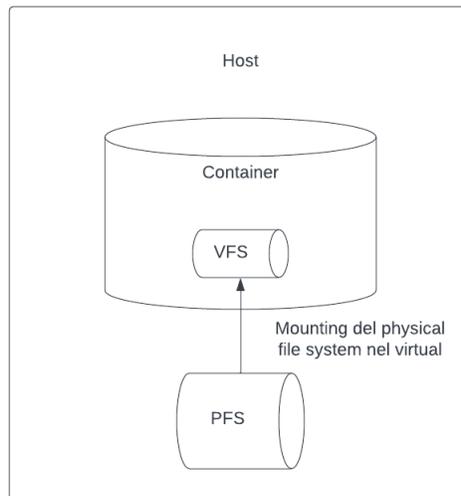
Ogni container creato viene automaticamente aggiunto ad un network di default, più nello specifico un bridge, un interfaccia virtuale che collega tutti i container alla rete del host e al suo subnet. La comunicazione con il mondo esterno è dunque inibita a meno di non esporre una specifica porta, però è possibile per i container comunicare tra di loro semplicemente riferendosi ai loro indirizzi IP.

In Docker il sistema di rete è collegabile usando dei driver. I diversi driver come bridge, host, overlay e altri forniscono delle impostazioni predefinite e forniscono certe funzionalità di base. In base alle necessità progettuali tipologia e dettagli della rete verranno discusse in seguito.

### 1.4.2 Volumi e container

Un volume è una raccolta di dati, un meccanismo per assicurare la persistenza dei dati generati e usati da container. In caso di spegnimento o reset di un container tutto ciò che è contenuto nel suo file system virtuale andrebbe perso, un problema generalmente di grandi dimensioni.

I volumi funzionano collegando un path del file system della macchina host a quello di uno o più container. In pratica si fa il "mounting" del file system.



Rappresentazione di un Docker Volume.

## 1.5 Docker compose

Nel mondo Docker esiste un tool chiamato compose capace di definire e gestire applicazioni Docker multi-container. Compose usa file YAML, formato per la serializzazione di dati, per configurare i servizi delle applicazioni, dalle immagini da usare ai network e volumi. Con un solo comando vengono creati e avviati tutti i servizi indicati nel file e con altrettanta facilità vengono arrestati. Compose è alla sua base un processo in tre fasi:

- Definire un Dockerfile in modo che possa essere riprodotto.
- Definire i servizi necessari nel file docker-compose.yaml.

- Attraverso compose eseguire la composizione e l'avvio di tutti i container.

```
version: "3.9" # optional since v1.27.0
services:
  web:
    build: .
    ports:
      - "8000:5000"
    volumes:
      - ./code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Esempio di un file docker-compose.yml da  
<https://docs.docker.com/compose/>[2]

## 1.6 Docker

In quanto molto presente in tutte le parti finora affrontate e quelle ancora da affrontare è opportuno parlare brevemente di Docker. Che cos'è Docker?

Docker è principalmente una piattaforma per sviluppo software e di un particolare tipo di virtualizzazione che rende possibile il "develop and deploy" di applicazioni in container ben confezionati e isolati. Le applicazioni funzionano allo stesso modo a prescindere da dove si trovano o su quale macchina sono. I container Docker possono essere schierati da quasi qualsiasi macchina senza problemi di compatibilità così da permettere di avere un software "system agnostic".

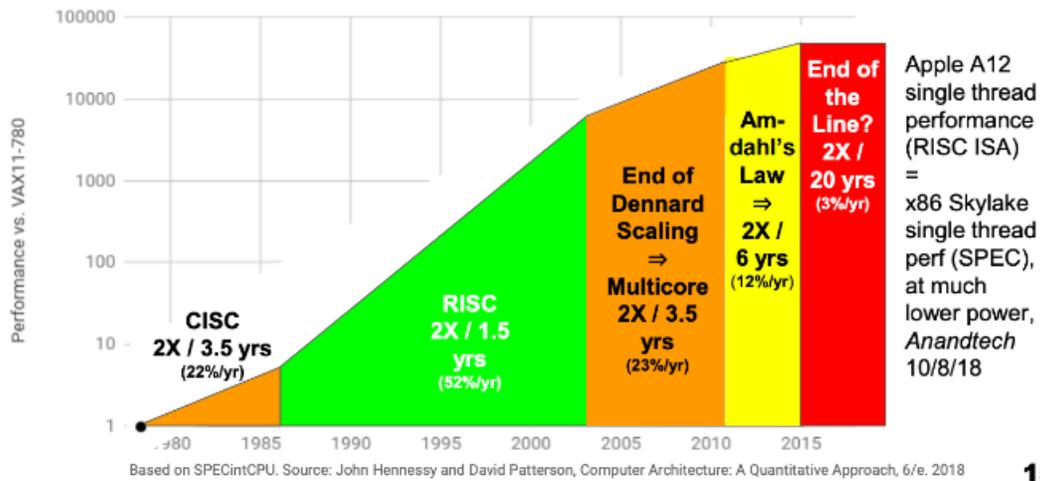
## Capitolo 2

# Parallelismo dei dati

La legge di Moore, che prevede lo sviluppo di sistemi e processori più potenti (e con più transistor), sembra svolgere al suo termine. Da una incapacità fisica a migliorare un sistema come si può continuare ad innovare?

Aumentando l'efficienza di quello che esiste già. Questo può avvenire in molti modi ma l'ambito preso in considerazione da questo lavoro è l'uso delle risorse a disposizione per poter gestire anche applicazioni che richiedono un uso massiccio di potere computazionale e dati.

40 years of Processor Performance



15

Grafico con focus sulla crescita in "performance" dei processori negli ultimi 40 anni. [1]

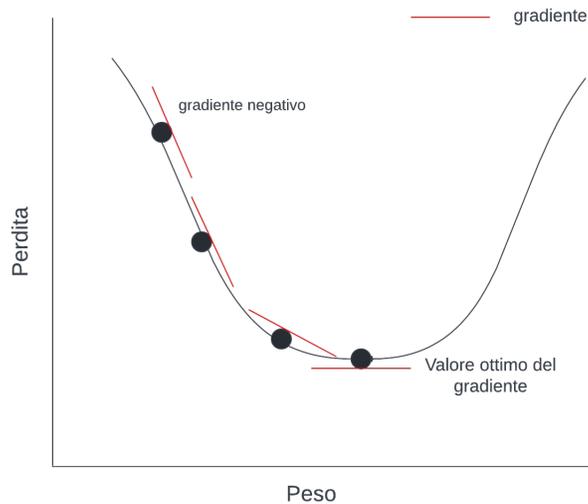
Più nello specifico, nel mondo del machine learning vi sono due fattori fondamentali: dati e potenza. Per poter "allenare" una rete neurale in ambiti come

l'object detection o image classification servono dei dataset molto grandi, nell'ordine dei GB o TB di dati, già ben definiti e classificati con delle label. Ormai però vi sono a disposizione grandi dataset curati e di dominio pubblico per la maggior parte dei problemi.

Il fattore potenza risulta invece molto più dispendioso e difficile da affrontare. Con il generale "rallentamento" della legge di Moore e la volontà di creare modelli basati su algoritmi sempre più complessi, un numero di parametri nell'ordine di miliardi, la possibilità di poter acquisire più potere computazionale rimane in mano ad aziende molto grandi. Dunque per poter affrontare problemi con l'allenamento di una rete neurale per l'identificazione di oggetti servono tecniche in grado di ottimizzare la potenza di calcolo limitata a disposizione.

## 2.1 Discesa del gradiente

L'obiettivo di un generico algoritmo di apprendimento è ridurre al minimo il "loss", la penalità in caso di una predizione incorretta. La perdita come tale è un numero ma viene calcolata attraverso una funzione, separabile e differenziabile, sui parametri dell'algoritmo in fase di allenamento, il modello. Lo scopo dell'algoritmo dunque è l'abbassare, e ottimamente ridurre a zero, questo valore in modo iterativo. L'approccio più usato e facile risulta quello di un algoritmo greedy seguendo l'opposto del gradiente della perdita in quanto indica un ottimo locale.



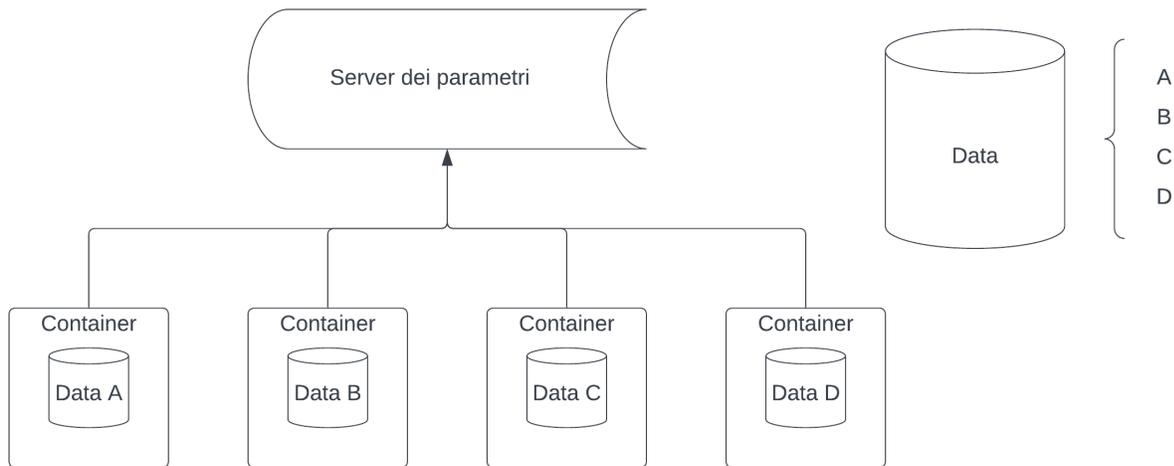
Esempio di una funzione di perdita con ricerca del valore ottimo del gradiente.

Il gradiente negativo indica che bisogna seguire l'opposto, verso un valore positivo.

Di norma nelle reti neurali questa funzione è separabile, calcolata facendo la media delle funzioni di perdita per i singoli dati. Calcolare la media per ogni singolo dato considerando tutti i parametri presenti risulta molto costoso e non ottimale, rendendo necessario l'uso della discesa del gradiente stocastico che ne calcola la media solo su un sottoinsieme significativo dei dati. Seppur usando questo accorgimento risulta lo stesso un'operazione dispendiosa sia in tempo che risorse, pertanto è inevitabile l'uso di metodi atti a parallelizzare il calcolo degli innumerevoli parametri.

## 2.2 Data parallelism

Nel parallelismo di dati vari processi lavorano in ambienti di lavorazione paralleli. Ogni processo, nel nostro caso veri e propri container, hanno un loro proprio dataset da elaborare. Lo stesso modello viene usato per ogni container ma i dati forniti a ciascuno di essi vengono divisi, condividendo solo alcuni parametri chiave (come possono essere dei pesi in una rete neurale). È veloce per quantità di dati piccole però risulta generalmente più lento in situazioni in cui è necessario trasferire tali dati.



Rappresentazione del data parallelism con server dei parametri applicata a container.

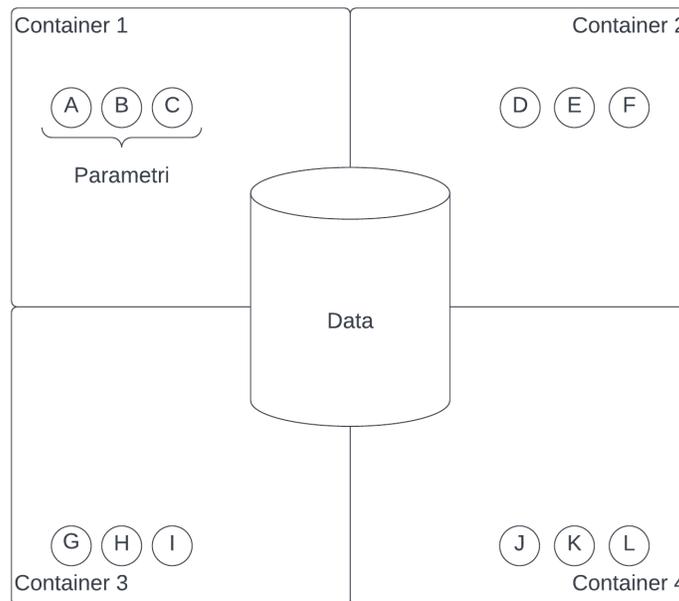
Esistono due macro-varianti di data parallelism:

- **Synchronous data parallelism:** i container rimangono sincronizzati durante il ciclo di training. Tutti gli container devono aspettare che tutti finiscano di calcolare i propri parametri locali prima di poter riprendere un nuovo ciclo.

- **Asynchronous data parallelism:** i container lavorano in modo asincrono senza doversi aspettare a vicenda. Per poter salvare i parametri locali calcolati è necessaria una struttura esterna che comunichi con i container, ad esempio un server dei parametri.

## 2.3 Model parallelism

Il parallelismo di modello invece di condividere un set di parametri e dividere i dati tra container pratica l'esatto opposto, divide i parametri e condivide i dataset. Recentemente usato sempre di più nel deep learning, il model parallelism risulta più veloce per le grandi reti. Risulta più complesso da implementare in quanto la fase di creazione del modello è molto critica per poter permettere la divisione dei parametri.



Rappresentazione del model parallelism applicata a container.

## 2.4 Quale metodo usare?

Entrambi i metodi sono validi, dunque perché spesso nel mondo cloud e deep learning si preferisce il model parallelism?

La risposta è abbastanza complessa e altamente legata al tipo di tecnologia in uso. Ad esempio nei pod TPU (Tensor Processing Unit) della Google[3], circuiti integrati specifici per l'applicazione di machine learning, viene usato il

model parallelism in combinazione con la libreria TensorFlow. Essendo i loro modelli mastodontici necessitano di una potenza di calcolo simile. Inoltre, la parallelizzazione di modello risulta l'unica soluzione nei casi in cui le dimensioni del modello generato sono troppo grandi per una macchina sola.

Al contrario nel mondo delle applicazioni embedded, e non solo, il data parallelism viene regolarmente utilizzata per ridurre il carico su una singola macchina e ottenere informazioni più rapide ed efficienti sui dati.

Seppur la distinzione dei casi d'uso sembri abbastanza netta l'obbiettivo di questa tesi è capire come poter applicare ed usare questi metodi anche in ambienti più piccoli. Una ricerca verso l'efficienza e l'applicabilità anche nel mondo dei piccoli computer e container e non solo nei supercomputer dei cloud provider.

## Capitolo 3

# Studio pratico sul parallelismo

### 3.1 Introduzione alla metodologia

L'approccio usato per poter studiare questi due macro-metodi di parallelismo è uno pratico, piccoli progetti inerenti alla metodologia analizzata. Ogni progetto ha lo scopo di illustrare e spiegare il funzionamento di data e model parallelism integrando soluzioni, già esistenti, all'uso di container per simulare un ambiente multi-device.

Prima di esporre la parte tecnica è opportuno introdurre alcuni concetti base legati al machine learning:

- **Modello:** è l'output di un algoritmo su dei dati in input. Rappresenta ciò che è stato imparato dopo aver eseguito l'algoritmo, l'insieme di regole, numeri, pesi e qualsiasi altra struttura dati specifica dell'algoritmo richiesta per fare previsioni. Spesso usato per indicare l'algoritmo che lo genera.
- **Training:** ciclo di apprendimento, eseguito dall'algoritmo sui dati in input per creare il modello. È possibile specificare vari parametri del ciclo molto importanti quali epoche (epochs) e il numero di passi per epoca (steps per epoch).
- **Batch:** è un iperparametro che indica il numero di dati (già preparati e modificati per l'algoritmo) da elaborare prima di aggiornare i parametri del modello.
- **Epoche:** è un passaggio completo di una batch attraverso l'algoritmo. Questo iperparametro è fondamentale in quanto determina anche la componente temporale (il tempo richiesto) della fase di training.

- **Passi per epoca:** parametro che impone un numero fisso di passi per epoca. Un esempio per illustrare meglio il concetto: se ho a disposizione 1,000 immagini e batch da 10 immagini il numero di step predefinito è

$$1000/10 = 100step$$

Tuttavia impostando un numero di step per epoca fisso si evitano tempi di training molto lunghi in caso di dataset grandi.

Oltre a questi concetti base ve ne sono molti altri di natura molto più specifica, molto spesso esclusivi di un certa libreria o framework.

## 3.2 Tecnologie usate

La parte progettuale è stata realizzata interamente in Python, più precisamente v.3.10.4, su una macchina Linux dalla "potenza" ristretta. Infatti, seppur altamente consigliato per diminuire i tempi di compilazione del modello, non sono stati eseguiti calcoli sulla GPU e i CUDA core ma soltanto sfruttando la CPU della macchina.

Alcune specifiche pertinenti della macchina:

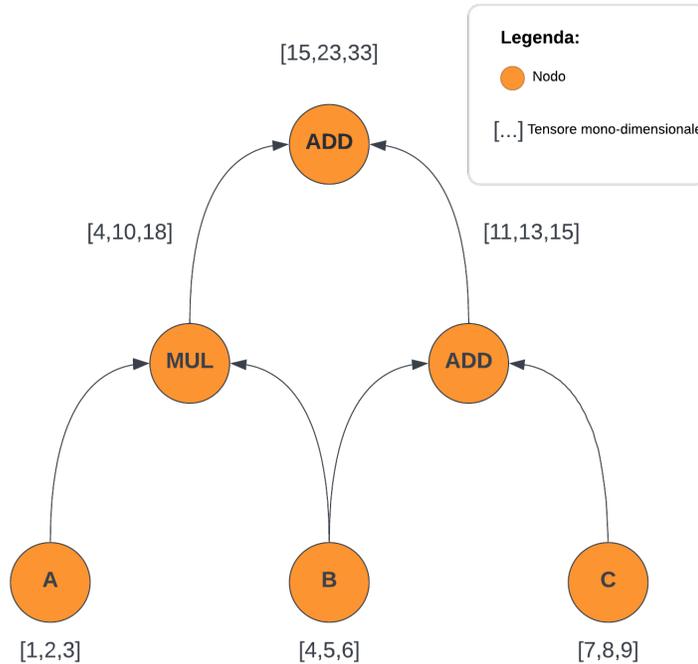
- CPU: Intel Core i5-1021 1.6GHz 4 core
- RAM: 12 GB

Tra le varie librerie usate le più importanti sono Tensorflow e Mesh Tensorflow.

### 3.2.1 Tensorflow

Libreria open-source sviluppata da Google Brain Team nel 2015, facilita tutte le operazioni numeriche e matematiche legate al machine learning. La libreria raggruppa in sé vari modelli e algoritmi e li rende utilizzabili attraverso API. Esegue le proprie funzioni in C++ ma sono disponibili wrapper anche per Python e Javascript.

Permette l'uso di "dataflow graphs", struttura a grafo che indica come i dati si muovono in una serie di nodi. Ogni nodo nel grafo rappresenta un'operazione matematica mentre ogni arco rappresenta un array multi-dimensionale di dati chiamato tensore.



Esempio di un semplice dataflow graph con tensori mono-dimensionali.

Il framework presenta anche la possibilità di definire molti parametri inerenti alla distribuzione di training su macchine diverse sia in una rete che in locale.

### 3.2.1.1 Keras

Keras è un modulo di Tensorflow dedicato a facilitare tutte le operazioni dedicate al machine learning, dalla creazione dell'algoritmo per il modello fino al deploy per il training distribuito. Presenta delle API di alto livello molto facili da usare ed integrare in quanto l'astrazione fornita permette di "saltare" la parte di operazioni puramente matematiche nella descrizione del grafo. Un approccio user friendly a problemi complessi.

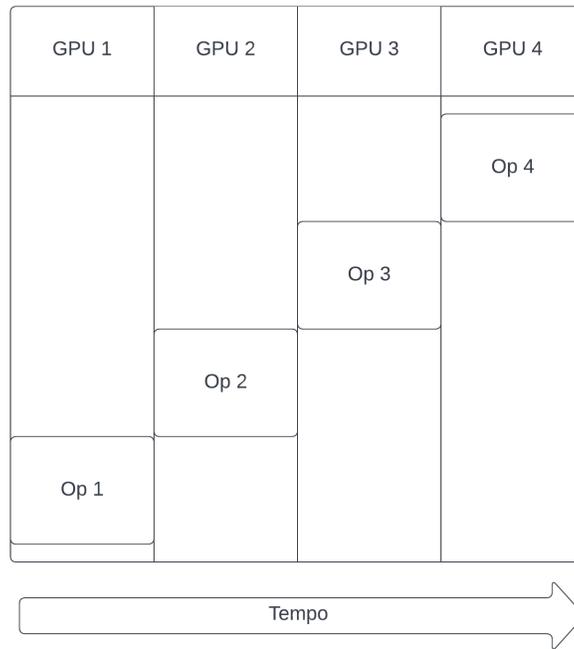
**"Keras è un'API progettata per gli esseri umani, non per le macchine. Keras segue le migliori pratiche per ridurre il carico cognitivo."**[4]

### 3.2.2 Mesh Tensorflow

È una libreria open-source che funziona insieme a Tensorflow (v1.x in quanto la compatibilità con Tensorflow v2.0 non è ancora supportata). Lo scopo di Mesh Tensorflow è semplificare la formalizzazione di strategie di distribuzione di un grafo su architetture multi-CPU, GPU o TPU.

A differenza di Tensorflow, con cui condivide gran parte della struttura base, introduce il concetto di mesh: è un array multi-dimensionale di processori connessi da una rete. Per processori non si intende esclusivamente la CPU ma anche GPU e pod TPU. Ogni tensore viene replicato o diviso su tutti i processori di una mesh dopodiché vengono nominate sia le dimensioni di ogni tensore che della mesh.

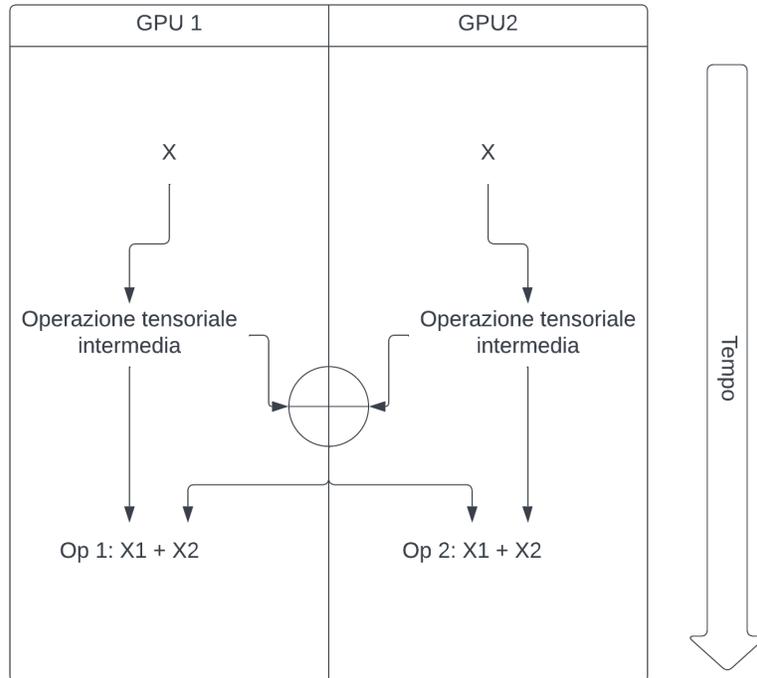
La distribuzione delle dimensioni dei tensori avviene mediante la specifica di un set di regole, un layout, andando ad indicare quali dimensioni tensoriali sono suddivise sulle dimensioni della mesh. Il layout scelto dall'utente non influisce sui risultati ma soltanto sulle prestazioni. In base all'implementazione delle operazioni si determina se l'algoritmo viene eseguito in "pipeline" o in modo parallelo dai vari processori.



Rappresentazione del model parallelism in una rete a quattro GPU senza parallelizzazione delle operazioni.

Se le operazioni avvengono su un algoritmo sequenziale, seppur avendo una struttura multi-processore, uno soltanto alla volta viene sfruttato in quanto l'output di un operazione è l'input della successiva.

Mesh Tensorflow invece permette l'implementazione di operazioni parallelizzate su tutti i processori di una mesh e se possibile un tipo di comunicazione collettiva. Un processore interviene solo sulle dimensioni dei tensori in input già residenti su quel processore e produce la dimensione di output per sé stesso.



Rappresentazione del model parallelism in una rete a due GPU con parallelizzazione delle operazioni.

### 3.2.3 GitHub Copilot

Copilot è un tool creato da GitHub che funge da assistente al programmatore. Viene definito da Github come un'intelligenza artificiale che svolge il lavoro di programmazione in coppia. Copilot è basato su Codex, un modello di linguaggio generativo e già pre-addestrato creato da OpenAI.

L'assistente si basa sui commenti e sul contesto presente sia nel file in utilizzo sia in quelli vicini o correlati nel progetto. Commenti e codice vengono usati per sintetizzare intere righe o funzioni intere. Per addestrare tale modello sono state usate tutte le repository pubbliche presenti su GitHub. Anche nei casi di una piattaforma nuova o linguaggio a mano a mano che esempi e repository diventano pubbliche questi vengono integrati nei training set, migliorando la pertinenza dei suggerimenti.

All'interno dei vari progetti è stato molto utilizzato in quanto ha facilitato molto il lavoro di modellazione e creazione dei dataset, conoscenze precedentemente non possedute. Come tool è un supporto e dunque non può sostituire completamente il programmatore, però rimane comunque un tool valido il cui potenziale è in continuo aumento.

## 3.3 Synchronous data parallelism

In questa prima parte progettuale viene approfondito una sottospecie particolare del data parallelism: synchronous data parallelism. L'idea base dietro ad una distribuzione sincrona dell'allenamento è quella di mantenere regimi di allenamento contemporanei con aggiornamenti ai parametri nello stesso momento, mantenendo l'idea base di dividere i dati, le batch, e non il modello.

### 3.3.1 Creazione del dataset

Il problema affrontato in questa sezione è la generazione di un set di dati conforme alle tipologie prestabilite da Keras e Tensorflow. I dati iniziali provengono da un dataset curato direttamente da Microsoft: Kaggle Cats and Dogs. All'interno di questi file sono presenti circa 24998 immagini divise equamente tra cani e gatti (circa in quanto alcune immagini non sono utilizzabili) già comprese di label in modo da avere sia un dataset di allenamento che di validazione.

Prima di poter compiere alcune trasformazioni sulle immagini in modo da renderle usabili dall'algoritmo bisogna "sanificare" i dati: eliminare tutte le immagini corrotte o che presentano dei byte sbagliati.

**Esempio:** scartare tutte le immagini che non contengono la stringa "JFIF" nel header.

Una volta sanificate le immagini devono essere adeguatamente preparate per la fase di training. Tra le operazioni principali vi sono la definizione di una grandezza standard delle immagini (nel progetto 180x180) e la dimensione della batch (in questo caso 32 immagini). Queste due scelte sono molto importanti in quanto hanno un ruolo significativo nella determinazione della complessità del modello, più i dati sono complessi (immagini più grandi e definite) e le batch grandi (grande quantità di dati da analizzare alla volta) maggiore risulta il tempo impiegato dall'algoritmo nella fase di allenamento, oltre ad un incremento del numero di parametri.

Keras e Tensorflow utilizzano un tipo di dati chiamato Dataset per rappresentare un potenziale set molto grande elementi, inoltre gran parte delle operazioni definite dalle API accettano soltanto questo tipo o simili. La creazione e utilizzo di un Dataset si divide in tre parti principali:

- 1. Creazione del Dataset da dei dati in input (le immagini in dimensione standard).
- 2. Applicare trasformazioni in modo da preprocessare i dati.
- 3. Iterare gli elementi del Dataset per produrre un output (l'algoritmo).

Nel progetto la creazione del Dataset avviene attraverso l'uso di una funzione chiamata `tf.keras.preprocessing.image_dataset_from_directory` che prende tutte le immagini presenti in una directory e compie alcune operazioni di preprocessione, come può essere la suddivisione in batch e la nomenclatura del subset. Così vengono ottenuti due dataset: training e validation. Il dataset di validazione viene usato durante il training per un'approssimazione del modello ad ogni

epoca, in pratica aiuta ad aggiornare i parametri del modello indicando se è la direzione giusta o meno.

Successivamente vengo aumentati attraverso un piccolo algoritmo sequenziale. Preso un elemento ne crea dei simili con dei piccoli cambiamenti, in questo caso una rotazione randomica e un ribaltamento orizzontale sempre randomico.

```
def create_dataset():
    os.environ['CUDA_VISIBLE_DEVICES'] = '-1'
    # generate dataset
    image_size = (180, 180)
    batch_size = 32

    train_ds = tf.keras.preprocessing.image_dataset_from_directory(
        "PetImages",
        validation_split=0.2,
        subset="training",
        seed=1337,
        image_size=image_size,
        batch_size=batch_size,
    )
    val_ds = tf.keras.preprocessing.image_dataset_from_directory(
        "PetImages",
        validation_split=0.2,
        subset="validation",
        seed=1337,
        image_size=image_size,
        batch_size=batch_size,
    )
    # print number of images in each dataset
    print("Number of training images before augmentation: " + str(len(train_ds)))
    print("Number of validation images before augmentation: " + str(len(val_ds)))

    # data augmentation
    data_augmentation = keras.Sequential(
        [
            layers.RandomFlip("horizontal"),
            layers.RandomRotation(0.1),
        ]
    )

    # configure dataset for performance
    train_ds = train_ds.prefetch(buffer_size=32)
    val_ds = val_ds.prefetch(buffer_size=32)
    return train_ds, val_ds, data_augmentation, image_size
```

Focus sul codice della creazione dei dataset.

L'ultima operazione presente è il *prefetch* che consiste nel generare già in un buffer la batch per lo step successivo, ottimizzando i tempi tra creazione della batch e training.

### 3.3.2 Creazione del modello

La creazione dell'algoritmo, quello da cui si genera il modello, avviene attraverso la definizione di vari layers, un oggetto richiamabile che accetta come input uno o più tensori e che emette uno o più tensori. In questo caso vi è presente un "entry block" che gestisce alcune operazioni iniziali come il *Rescaling* che prende in input i bit delle immagine, il cui valore va da 0 a 255, e lo ridimensiona in un range da 0 a 1. Essendo un modello sequenziale l'output di un layer diventa l'input del successivo. In questo caso sono state sfruttate le risorse presenti nella documentazione di Keras[4] per la realizzazione del modello in un problema di image classification.

```
def make_model(input_shape, num_classes, data_augmentation):
    inputs = keras.Input(shape=input_shape)
    # Image augmentation block
    x = data_augmentation(inputs)

    # Entry block
    x = layers.Rescaling(1.0 / 255)(x)
    x = layers.Conv2D(32, 3, strides=2, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    x = layers.Conv2D(64, 3, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
```

Focus sul entry block nella funzione di creazione del modello.

Altri layer come *Conv2D*, crea un kernel convoluzionale, o il *MaxPolling2D*, esegue il downsampling dell'input lungo le sue dimensioni spaziali, definiscono operazioni complesse su tensori.

Una volta definiti tutti i layer la funzione *keras.Model()* li raggruppa in un oggetto con funzioni di addestramento e inferenza.

```
=====
Total params: 2,782,649
Total params: 2,782,649
Trainable params: 2,773,913
Non-trainable params: 8,736

-----
Trainable params: 2,773,913
Non-trainable params: 8,736
-----
```

Focus sul numero di parametri del modello a fine compilazione.

### 3.3.3 Creazione della strategia

Definita la creazione del modello e dataset per poter imporre una tecnica di parallelizzazione al cluster di container è necessario definire una strategia.

*MultiWorkerMirroredStrategy* è una strategia già presente nel modulo Keras e predispone le regole per un ambiente di parallelizzazione dei dati sincrono distribuito. Creato un oggetto strategia, all'interno del suo scope avviene la creazione del modello, la definizione di callback, ad esempio il salvataggio di un checkpoint ad ogni fine epoca, e la compilazione del modello. Fuori dallo scope è presente la creazione dei dataset mediante la funzione *create\_dataset()* e il passaggio da un tipo Dataset a DistributedDataset per i dati di training. Questa conversione avviene in modo da poter suddividere le batch in fase successiva. La parte finale è la fase di training attraverso la funzione *fit()* applicata al modello.

```
strategy = tf.distribute.MultiWorkerMirroredStrategy()
print("Number of devices: {}".format(strategy.num_replicas_in_sync))

#create distributed dataset

# create dataset
train_ds, val_ds, data_augmentation, image_size = create_dataset()
distributed_train_ds = strategy.experimental_distribute_dataset(train_ds)

#open a strategy scope
with strategy.scope():
    model = make_model(input_shape=image_size + (3,), num_classes=2, data_augmentation=data_augmentation)
    # keras.utils.plot_model(model, show_shapes=True)
    # get model summary
    epochs = 2
    steps_per_epoch = 20
    model.summary()
    # callbacks
    callbacks = [
        keras.callbacks.ModelCheckpoint("save_at_{epoch}.h5"),
    ]
    model.compile(
        optimizer=keras.optimizers.Adam(1e-3),
        loss="binary_crossentropy",
        metrics=["acc"],
    )
history = model.fit(
    distributed_train_ds, validation_split=0.1, epochs=epochs, steps_per_epoch=steps_per_epoch, callbacks=callbacks, validation_data=val_ds,
)
```

Focus sulla creazione della strategia, lo scope e la funzione *fit()*.

### 3.3.4 Come funziona MultiWorkerMirroredStrategy?

Questa strategia fornita da Tensorflow implementa l'allenamento sincrono distribuito tra più lavoratori, i worker. Applicando una tipologia di data parallelism tutto il modello viene replicato in ogni singolo worker mentre le batch di allenamento vengono divise per ognuno. In questo modo da una batch iniziale di 32 elementi vengono create ( $n = \text{numero di worker}$ )  $32/n$  batch locali, una per ogni worker. Durante la fase di training viene eseguito l'algoritmo per la creazione del modello su ogni lavoratore usando la propria batch locale. Essendo il trai-

ning diviso in epoche ad ogni fine le repliche vengono sincronizzate, mantenendo il comportamento di convergenza del modello.

La sincronizzazione avviene nel seguente modo:

- 1. I worker eseguono un'epoca di training sulla batch locale, dunque viene generato il gradiente dei pesi rispetto alla perdita del modello.
- 2. A fine epoca i parametri calcolati su ogni batch locale vengono confrontati in modo da poter creare una nuova "base" di partenza uguale per tutte le repliche.
- 3. Viene fatta partire una nuova epoca di training sulle batch locali a partire dalla base di partenza generata precedentemente.

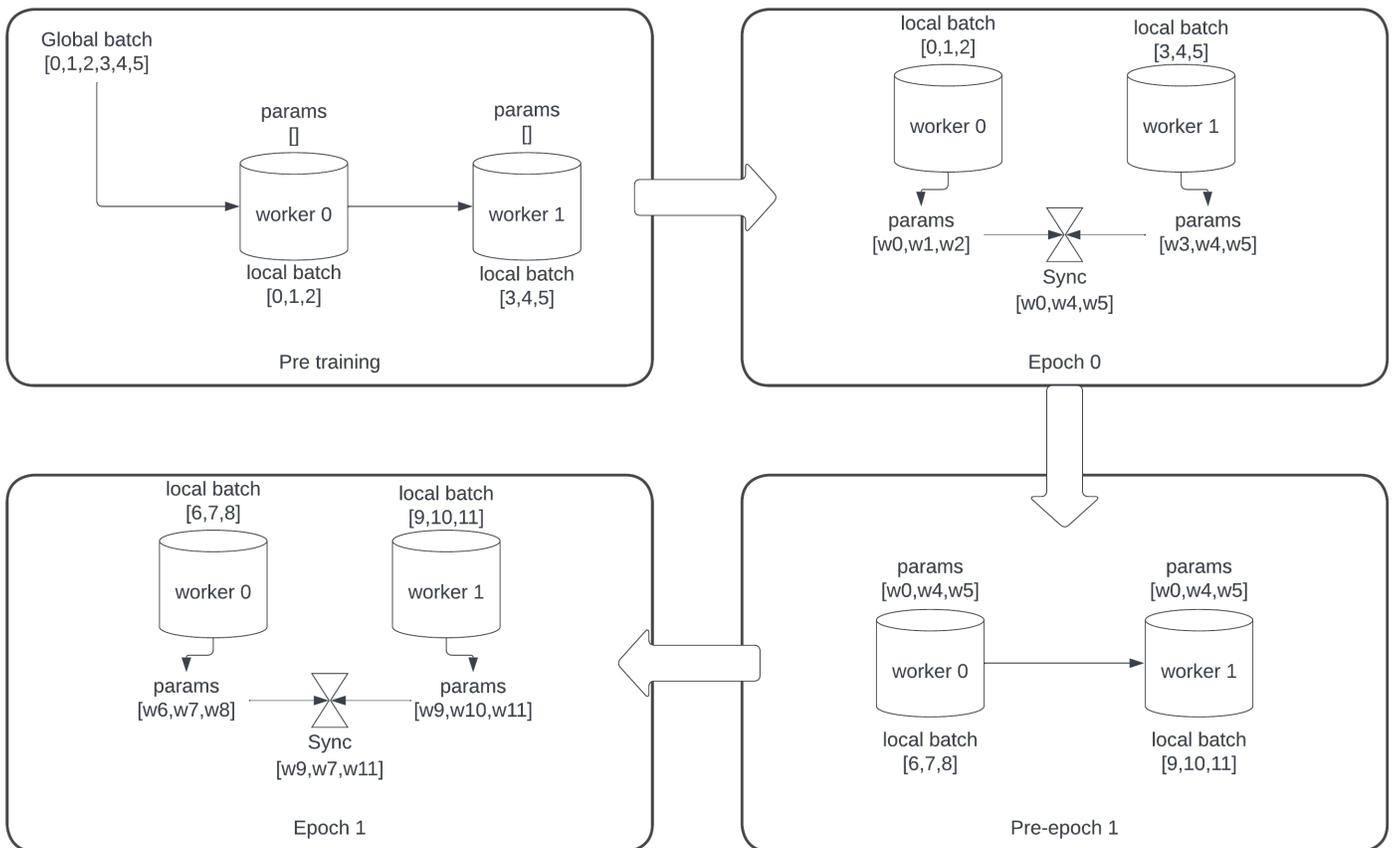


Diagramma che mostra il funzionamento di MultiWorkerMirroredStrategy su due worker.

I worker possono essere macchine dotate di una o più GPU oppure dei pod TPU in cloud, però a fini pratici in questo caso sono due container Docker costruiti sopra un'immagine ubuntu.

Per poter usare questa strategia multi-device è necessario configurare un cluster, collettivo di macchine, attraverso l'uso di alcune direttive come il `TF_CONFIG`, variabile d'ambiente che descrive ruolo e come comunicare con il resto delle macchine. Essendo che tutte i container usano lo stesso codice l'unica differenza tra di essi è questa variabile d'ambiente.

```
os.environ['TF_CONFIG'] = json.dumps({
    'cluster': {
        'worker': ["localhost:12345", "localhost:23456"]
    },
    'task': {'type': 'worker', 'index': 0}
})
```

Focus sul codice della variabile d'ambiente `TF_CONFIG` nel worker 0.

Il worker che si occupa di attuare e gestire le operazioni inerenti alla strategia è il worker con indice zero, chiamato anche "chief". Questo lavoratore coordina il resto del cluster in tutti gli aspetti, come ad esempio la divisione delle batch, ma assume anche il ruolo di lavoratore. La comunicazione, essendo in locale, avviene mediante porte in localhost come indicato nella variabile d'ambiente. Il protocollo usato da Tensorflow e Keras è gRPC (Google Remote Procedure Call), un framework RPC multiplatforma ed open-source. Usa HTTP/2 per il trasporto e Protocol Buffer come linguaggio di descrizione dell'interfaccia.

Più nello specifico in questo caso vengono creati due server:

- `grpc://localhost:12345`
- `grpc://localhost:23456`

La parte numerica finale è la porta esposta dai container.

### 3.3.5 Gestione dei container

Come già accennato i due container presi in considerazione sono `host-worker` (chiamato così perché ha anche il ruolo di chief) e `worker`. Entrambi questi container sono costruiti sopra un'immagine, la cui base è ubuntu, in cui sono stati installati tutti i framework e tool necessari, da pip fino a Tensorflow, e integrate ad altri comandi attraverso un Dockerfile.

Tra i comandi utilizzati vi sono: *FROM* che indica l'immagine di partenza, *WORKDIR* che indica la directory in cui si trova il container alla partenza, *CMD* che esegue un comando (in questo caso di fare partire uno script python) ed infine *EXPOSE* che espone la porta indicata al host.

```
FROM keras-container:v1.0
WORKDIR /keras_detect
CMD python3 image_detect.py
EXPOSE 12345

FROM keras-container:worker
WORKDIR /keras_detect
CMD python3 image_detect.py
EXPOSE 23456
```

Focus sul Dockerfile per l'host-worker e worker

Per poter gestire un sistema multi-container come questo viene usato Docker compose. Attraverso il file docker\_compose.yml sono indicate alcune direttive per il deployment dei due container.

```
version: "3.9"
services:
  host_worker:
    image: keras-host:cats
    network_mode: "host"
  worker:
    image: keras-worker:cats
    network_mode: "host"
```

Focus sul file docker\_compose.yml.

In particolare vengono definiti due servizi con le proprie immagini ed entrambi appartenenti alla rete "host", la stessa rete della macchina su cui girano.

### 3.3.6 Conclusioni pratiche

Grazie all'uso della strategia fornita da Tensorflow l'overhead della sincronizzazione risulta leggero ed in gran parte auto-coordinato. L'esistenza della variabile d'ambiente TF\_CONFIG facilita enormemente l'uso di un cluster multi-device, o meglio dire multi-container.

Tra le difficoltà incontrate la maggiore è quella di riuscire ad eseguire il training di un modello abbastanza grande su una macchina tutt'altro che potente. Il non usare calcoli sulla GPU presenta un grande handicap nel fattore tempo. A fini sperimentali, dunque, ho deciso di diminuire il numero di immagini in una singola batch, da 32 a 8, senza modificare il modello. Di seguito alcuni grafici ottenuti alla fine di un ciclo di training di 20 epoche su due container.

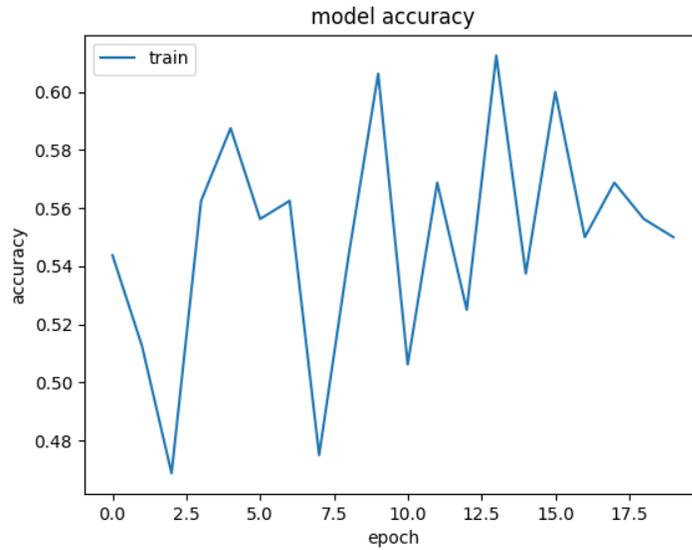


Grafico che mostra l'accuracy.

La precisione è il numero di punti dati correttamente previsti su tutti i punti dati. In questo caso è mostrato come percentuale. Si può notare come con il progredire delle epoche tenda a diventare sempre più lineare verso una crescita poco ripida.

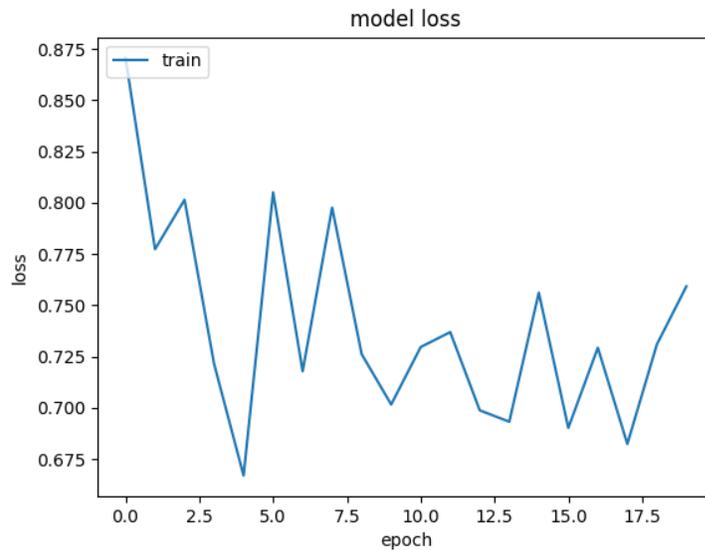


Grafico che mostra il loss.

La perdita è la penalità per una cattiva previsione dei punti dati. Anche in questo caso è rappresentata come percentuale. È notevole la progressione verso un loss sempre minore e con una disparità tra alti e bassi altrettanto minore.

Purtroppo in questo caso il modello generato risulta comunque impreciso e pronò ad errori in quanto il numero di epoche compiute è decisamente troppo basso per permettere un apprendimento su un modello di queste dimensioni.

## 3.4 Asynchronous data parallelism

Nella seconda parte riguardante il data parallelism viene affrontato una metodologia simile alla prima ma con alcune sostanziali differenze pratiche e teoriche. Nel data parallelism asincrono non vi è la necessità di sincronizzare i vari lavoratori ad ogni fine epoca di allenamento, ma bensì subentrano altre modalità e ruoli.

### 3.4.1 Creazione del dataset

Per quanto riguarda la creazione del dataset e il modello non vi sono particolari differenze rispetto alla tipologia sincrona. Durante la creazione del dataset viene posta una maggiore enfasi sull'eseguire le operazioni di *shuffle()* e *repeat()*.

Lo shuffle del dataset serve in quanto di base l'addestramento del server dei parametri presuppone che ogni lavoratore riceva lo stesso set di dati, tranne quando viene mischiato. Dunque, utilizzando questo metodo vengono garantite più iterazioni sui dati.

```
train_ds = train_ds.shuffle(4).repeat().prefetch(buffer_size=8)
val_ds = val_ds.repeat().prefetch(buffer_size=8)
```

Focus sul codice nella parte finale di creazione del dataset.

In quanto i lavoratori non si sincronizzano potrebbero terminare l'elaborazione dei propri set di dati in momenti diversi. È dunque necessario definire una ripetizione del dataset indefinitamente finché non finiscono le epoche di training.

La creazione del modello avviene allo stesso modo del synchronous data parallelism.

### 3.4.2 Creazione della strategia

ParameterServerStrategy è una strategia già presente in Tensorflow e predisponde le regole per un ambiente di parallelizzazione dei dati asincrono distribuito. Anche qui è necessario creare un oggetto strategia al cui interno avviene la creazione del modello, la definizione di callback, ad esempio il salvataggio di un checkpoint ad ogni fine epoca, e la compilazione del modello, mentre fuori viene collocata la creazione dei dataset.

Una delle differenze principali è la necessità di usare un "cluster resolver", un oggetto che permette di identificare i vari ruoli dei container presenti nella rete. I container possono assumere 3 ruoli (4 contando l'evaluator non presente):

- **Chief:** è il coordinatore ed è il ruolo analogo allo chief in synchronous data parallelism con la differenza che va dichiarato esplicitamente. Non esegue il lavoro di un worker. Crea risorse come i dataset, salva checkpoint ed altre operazioni di coordinamento.
- **Worker:** a differenza del worker sincrono a fine epoca non aspetta i suoi simili ma bensì comunica con il parameter server.
- **Parameter Server:** è un server in cui vengono salvati il valore dei parametri. Comunica con i worker accettando o meno i parametri inviati a fine epoca ed ha lo scopo di "base comune dei parametri" per la successiva epoca dei worker.

A livello di codice vengono definiti due blocchi diversi, uno per il chief e l'altro per worker e parameter server (ps).

I worker e i ps prendono i dati necessari forniti da un cluster resolver, creato in precedenza, e creano un proprio server mettendosi in attesa dell'inizializzazione del coordinatore. La comunicazione avviene sempre tramite gRPC.

```
# resolve cluster
cluster_resolver = tf.distribute.cluster_resolver.TFConfigClusterResolver()
if cluster_resolver.task_type in ("worker", "ps"):
    # wait for chief to finish
    # Start a TensorFlow server and wait.
    # Set the environment variable to allow reporting worker and ps failure to
    # coordinator. This is a workaround and won't be necessary in the future.
    os.environ["GRPC_FAIL_FAST"] = "use_caller"

    server = tf.distribute.Server(
        cluster_resolver.cluster_spec(),
        job_name=cluster_resolver.task_type,
        task_index=cluster_resolver.task_id,
        protocol=cluster_resolver.rpc_layer or "grpc",
        start=True)
    # print server status and port
    print("Server address: " + server.target)
    print("Server status: " + str(server.server_def))
    print("Server port: " + str(server.target.split(":")[-1]))
    server.join()
```

Focus sulla creazione dei server per worker e ps

Il coordinatore si occupa di creare l'oggetto strategia e di eseguire tutte le operazioni inerenti alla creazione e all'allenamento del modello.

```

else:
    strategy = tf.distribute.experimental.ParameterServerStrategy(cluster_resolver)
    train_ds, val_ds, data_augmentation, image_size = create_dataset()

    # create dataset
    # train_ds, val_ds, data_augmentation, image_size = create_dataset()
    # open a strategy scope
    with strategy.scope():
        model = make_model(input_shape=image_size + (3,), num_classes=2, data_augmentation=data_augmentation)
        # keras.utils.plot_model(model, show_shapes=True)
        # get model summary
        epochs = 20
        steps_per_epoch = 20
        validation_steps = 20
        model.summary()

```

Focus su una parte di codice inerente al coordinatore, il resto è molto simile al chief nella versione sincrona.

### 3.4.3 Come funziona ParameterServerStrategy?

Questa strategia fornita da Tensorflow fa parte del modulo `experimental` in quanto presenta ancora delle parti in via di sviluppo. È un metodo data parallel asincrono pensato per la scalabilità del machine learning su un cluster di macchine anche diverse tra loro in ambito di potenza.

Un cluster è formato da lavoratori e server dei parametri. I ps creano, salvano e distribuiscono i parametri e questi vengono letti e aggiornati dai lavoratori in modo indipendente e asincrono. Ogni worker e ps eseguono un *tf.distribute.Server* ed è il coordinatore, o chief, a gestire la distribuzione di dati e parametri su questi. Ciascun worker elabora solo le richieste del coordinatore e comunica con i ps senza avere interazioni con altri simili, così permettendo un ritmo di lavoro individuale non affetto da errori a tempo di esecuzione di altri lavoratori. Gli unici a dover essere sempre disponibili sono i ps e il chief. L'esecuzione tipica di questa strategia risulta:

- Il coordinatore distribuisce parametri e risorse a lavoratori e server dei parametri.
- I lavoratori eseguono un'epoca di training sulla propria batch locale. A fine epoca cercano di comunicare con i rispettivi server dei parametri.
- Il server dei parametri accetta o meno l'aggiornamento dei parametri e ridistribuisce i parametri al worker prima della nuova epoca.
- A fine training il chief produce il modello.

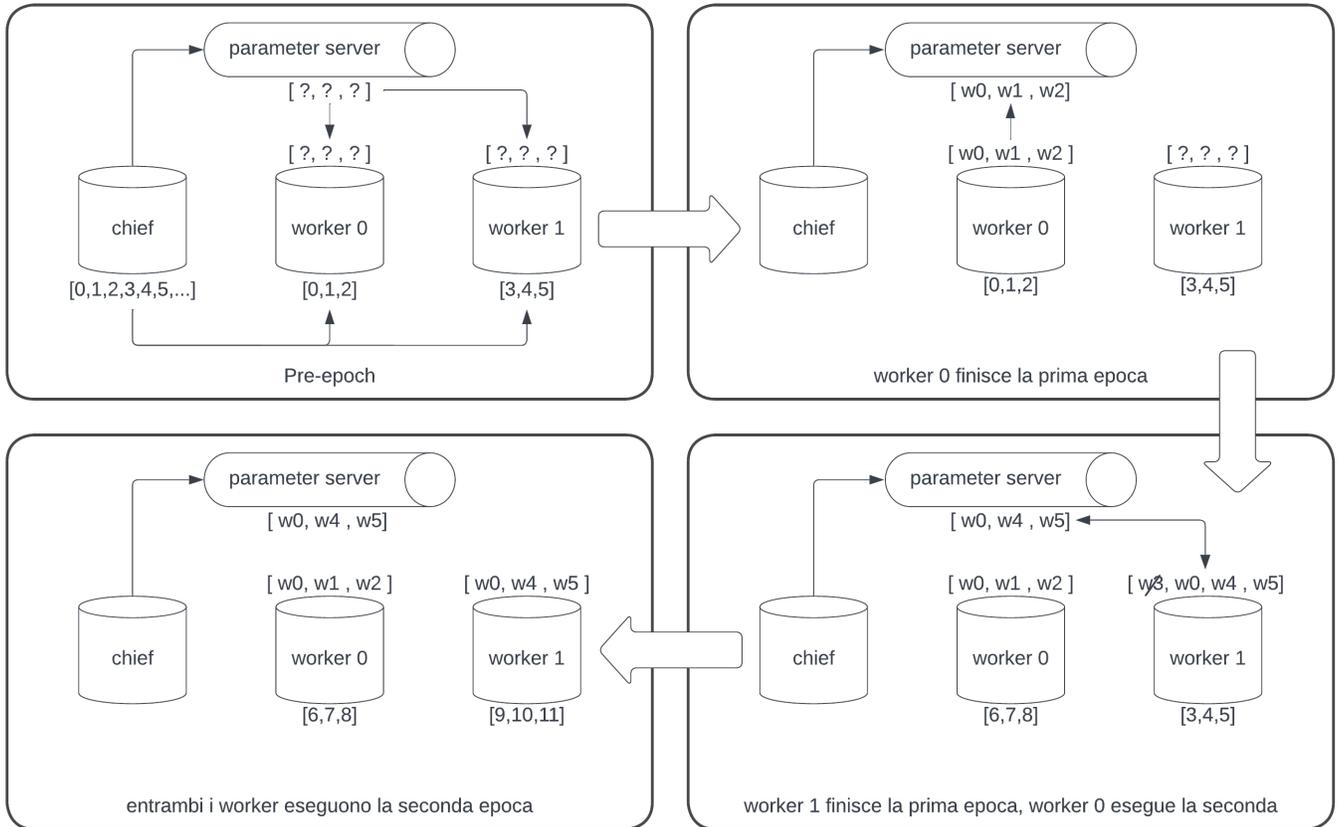


Diagramma che mostra il funzionamento di ParameterServerStrategy usando due worker, uno chief e un parameter server.

### 3.4.4 Gestione dei container

Per questa strategia i container usati sono cinque: un chief, due worker e due parameter server. La scelta di usare due parameter server è dovuta ad una curiosità tecnica e come preventivo visto il grande numero di parametri del modello. I due ps si dividono i parametri in modo quasi equo, per una vera divisione è consigliato un numero dispari. Come per la versione sincrona tutti questi container si basano su immagini proprie basate su ubuntu, ognuna con il proprio Dockerfile, la cui struttura è del tutto uguale alla versione sincrona. La struttura del docker\_compose.yml è la seguente:

```
version: "3.9"
services:
  chief:
    image: keras-async:chief
    network_mode: "host"
  ps_0:
    image: keras-async:ps
    network_mode: "host"
  ps_1:
    image: keras-async:ps-1
    network_mode: "host"
  worker_0:
    image: keras-async:worker-0
    network_mode: "host"
  worker_1:
    image: keras-async:worker-1
    network_mode: "host"
```

File docker\_compose.yml.

### 3.4.5 Conclusioni pratiche

Questo tipo di tipologia presenta varie difficoltà implementative, soprattutto in un ambiente così "piccolo" e ristretto in ambito potenza. L'overhead per quanto riguarda tutta la struttura è sicuramente maggiore e fonte di errori e ritardi. Purtroppo non sono riuscito ad usare alcune funzioni del modulo `experimental` come `DatasetCreator` che ottimizzano di certo la creazione dei dataset. Per quanto riguarda l'uso di variabili "shard" ho preferito non usarle in modo da mantenere una struttura più semplice, sia dal punto di vista pratico che teorico. Anche qui per eseguire il regime di allenamento il numero di immagini in una batch è stato abbassato a 8, permettendo così di avere dei risultati iniziali in tempi gestibili. Sicuramente con una maggior conoscenza, una documentazione migliore e un modello più piccolo i risultati ottenuti sarebbero molto più simili a quelli ottenuti nella versione sincrona, invece di leggermente peggiori.

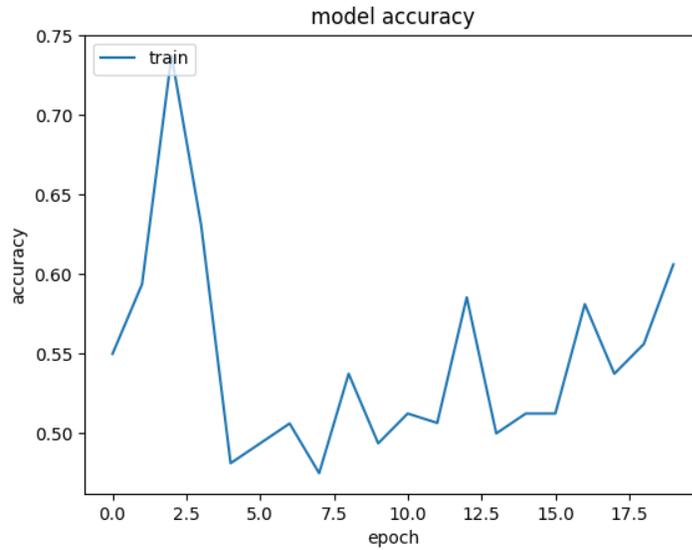


Grafico che mostra l'accuracy.

Per l'accuracy si possono notare le stesse caratteristiche già individuate, soltanto in maniera più fiavole. La precisione dopo un picco iniziale molto alto scende ad un valore medio risalendo in maniera lenta con il progredire delle epoche.

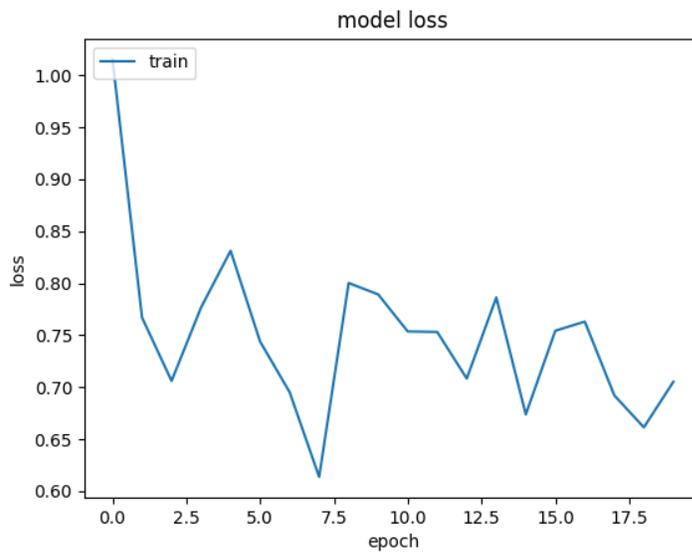


Grafico che mostra il loss.

Similmente alla metodologia sincrona si può notare la lenta discesa della perdita nelle epoche.

### 3.5 Data parallelism applicato ad un dataset e modello diverso

A causa delle grandi dimensioni del modello, e dei dataset usati, il risultato finale non è ben allenato, non è un modello pronto a predire qualcosa. Dunque, per poter testare anche questo aspetto è stata presa la decisione di svolgere un lavoro molto simile ma su modello molto meno complesso.

MNIST (Modified National Institute of Standards and Technology) è un database contenente numeri scritti a mano da usare in regimi di allenamento per il problema dell'immagine classification. È presente come uno dei dataset base già presenti in Tensorflow, dunque molto facile da usare attraverso API specifiche. (`tf.keras.datasets.mnist.load_data`)

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 784)]	0
dense (Dense)	(None, 256)	200960
dense_1 (Dense)	(None, 256)	65792
dense_2 (Dense)	(None, 10)	2570

=====  
Total params: 269,322  
Trainable params: 269,322  
Non-trainable params: 0

Struttura del modello ottenuta attraverso la funzione `summary()` sul modello.

Il modello creato risulta così molto più piccolo e leggero, quasi una riduzione del 90% del numero di parametri, consentendo, attraverso minimi cambiamenti al parco container già definito, di allenare a sufficienza una rete neurale. Tra le differenze maggiori vi è la decisione di usare un parameter server in meno nella versione asincrona in quanto non necessario. Struttura del codice e container rimangono molto simili al modello precedente.

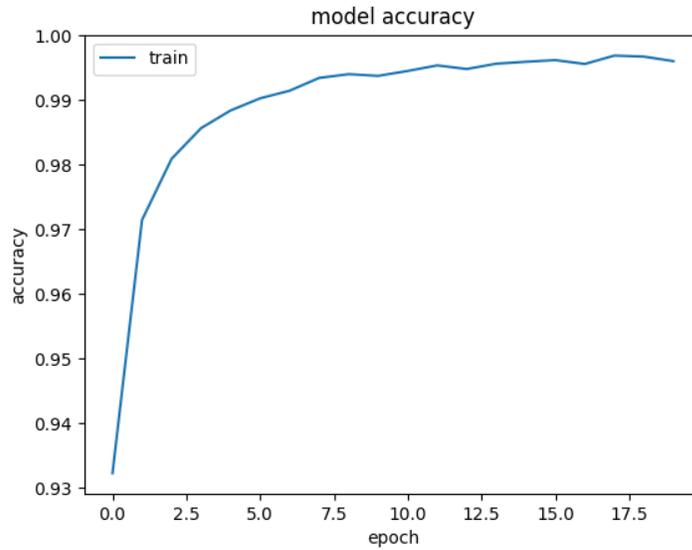


Grafico che mostra l'accuracy nella variante sincrona.

In questo caso si può ben notare la crescita della precisione verso valori molto alti. Contrariamente si può notare la perdita.

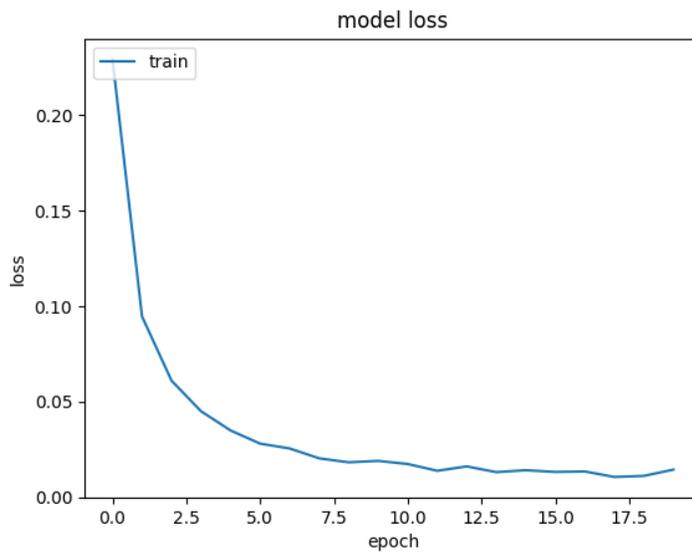


Grafico che mostra il loss nella variante sincrona.

Nella variante asincrona i risultati sono simili ma presentano una crescita e decrescita dotate di picchi e baratri più marcati.

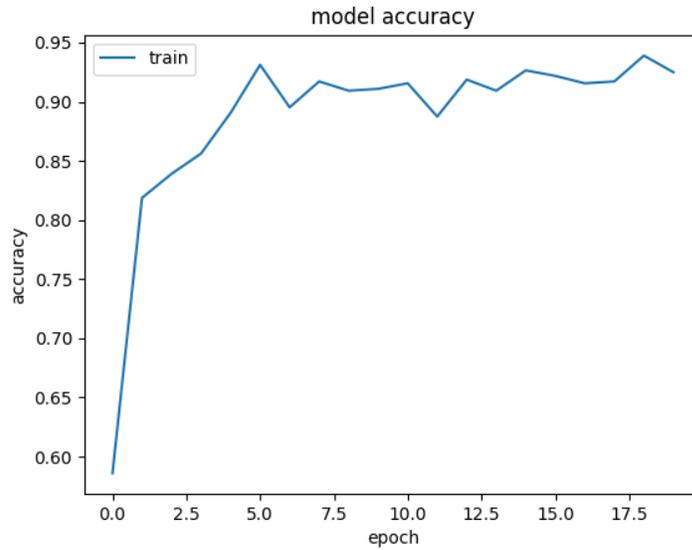


Grafico che mostra l'accuracy nella variante asincrona.

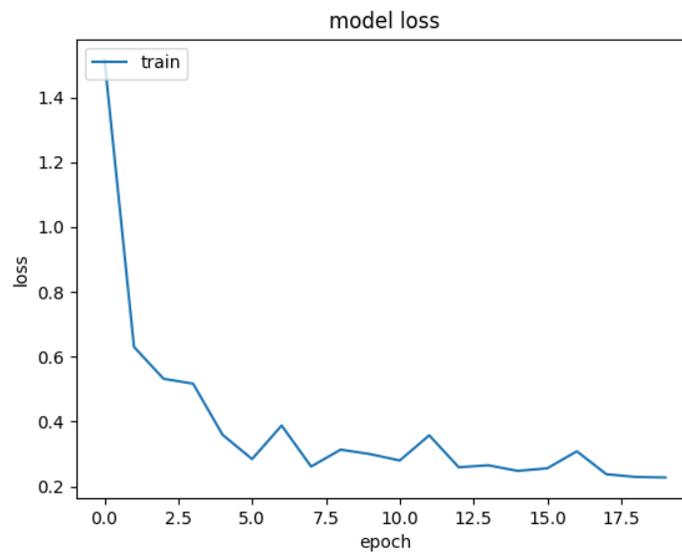


Grafico che mostra il loss nella variante asincrona.

### 3.6 Model parallelism

Per poter analizzare il problema del model parallelism l'utilizzo di Tensorflow e Keras è risultato molto difficile in quanto non è una modalità supportata

di base. Per ovviare a questo problema esiste la libreria Mesh Tensorflow, precedentemente introdotta.

A causa di problemi tecnici dovuti sia ad una scarsa documentazione e sia ad una preparazione non adeguata nel complesso ambito del machine learning non è stato possibile conseguire una prova pratica sui container. A causa di tempistiche ristrette ulteriori tentativi non sono stati possibili.

## Capitolo 4

# Conclusioni e sviluppi futuri

### 4.1 Conclusioni sul data parallelism

Per entrambe le versioni di data parallelism sono state trovate delle soluzioni già esistenti, fornite da Tensorflow e Keras, integrandole ad un cluster di container costruito appositamente per poter utilizzare la strategia proposta. Per il synchronous data parallelism si è individuata la strategia MultiWorkerMirroredStrategy, integrando un cluster di container worker in localhost. Per l'asynchronous data parallelism è stata usata la strategia ParameterServerStrategy assieme alla creazione di vari container dai ruoli diversi anch'essi in localhost. Lo studio sulla fattibilità risulta così compiuto, in quanto per entrambe le strategie l'integrazione dei container è risultata possibile.

Il data parallelism è una tecnica che ben si presta ad un'implementazione su container. La divisione dei dati risulta veramente importante nel caso di dataset molto grandi e attraverso l'uso dei volumi dei container, non usato per semplificare la struttura, è possibile ridurre lo spazio occupato dai dati. Anche la comunicazione tra vari agenti risulta molto semplice da implementare grazie alle funzioni di Docker e la possibilità di usare il localhost. Dai dati finali estratti da entrambi i metodi, synchronous e asynchronous, si nota molto bene come l'approccio asincrono produce un gradiente più "rumoroso"; quindi, è sicuramente necessario completare più epoche rispetto all'approccio sincrono.

Il parallelismo asincrono risulta davvero efficace quando vi è una disparità di potenza tra le macchine lavoratrici. Nel caso dei container è un problema inesistente, rendendo i tempi di training molto simili. L'approccio sincrono risulta dunque il migliore nel complessivo, sia in termini di risultati che tempi, inoltre a differenza della modalità asincrona non vi sono altri ruoli da gestire oltre al lavoratore (il chief non viene contato in quanto è anch'esso un lavoratore). Senza un overhead, dato dal dover gestire uno o più server dei parametri e un chief, la comunicazione risulta molto più semplice ed anche la gestione dei container rispecchia tale semplicità.

L'applicazione di tecniche parallele di machine learning su un cluster di con-

tainer risulta un approccio realizzabile e utilizzabile. L'astrazione e la modularità dei container permettono di creare dei cluster, sia in locale che su macchine diverse, in maniera semplice e veloce.

## 4.2 Sviluppi futuri

A causa di problematiche dovute a tempistiche e complessità il model parallelism risulta il prossimo passo da compiere per permettere un vero confronto di tutte le tecniche di parallelizzazione applicate a container. Una futura soluzione potrebbe basarsi su PlacementMeshImpl, una strategia fornita da Mesh Tensorflow. Tale strategia è costruita per poter usare delle mesh multi GPU/CPU dunque capire come funziona e come poterla integrare ad un cluster di container è uno degli sviluppi futuri.

Dopo aver analizzato anche il model parallelism e la libreria Mesh Tensorflow, un'ottimizzazione delle soluzioni già affrontate potrebbe portare a dei risultati migliori. pur mantenendo lo stesso cluster di container e numero di epoche. Tra le ottimizzazioni possibili vi è una migliore struttura del modello ed una creazione dei dataset più efficace. Una possibile soluzione futura potrebbe anche fare uso di metodi e funzioni presenti nei moduli "experimental" di Tensorflow e Mesh Tensorflow.

# Bibliografia

- [1]<https://noahgift.github.io/cloud-data-analysis-at-scale/topics/end-of-moores-law.html>
- [2]<https://docs.docker.com/>
- [3]<https://cloud.google.com/tpu?hl=it>
- [4]<https://keras.io/>

## **Ringraziamenti**

Gli anni passati al campus di Cesena, oltre ad essere formativi dal punto di vista accademico, hanno rappresentato un momento di maturazione. Questa tesi in particolare mi ha aiutato a scegliere il percorso per il futuro.

Innanzitutto, vorrei ringraziare il professor Ghini che mi ha accompagnato in questo percorso di tesi. La sua passione per tutto ciò che riguarda l'informatica è semplicemente contagiosa.

Un grande ringraziamento va anche ai miei amici e colleghi che mi hanno accompagnato in questa avventura, sia nei momenti migliori che nei peggiori. In particolare, vorrei ringraziare Lorenzo, Thomas e Francesco.

Un ultimo ringraziamento va alla mia famiglia e ai miei amici da una vita, la mia seconda famiglia.