



TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

**UNIVERSITÀ DEGLI STUDI DI
ROMA TOR VERGATA**

MACROAREA DI INGEGNERIA

MAGISTRALE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Tesi di laurea magistrale

Deployment di modelli di deep learning distribuito
nell'edge-cloud continuum con requisiti di qualità

2024-2025

Relatori

RELATORE:

Prof.ssa Valeria Cardellini

CANDIDATO:

Simone Nicosanti

RELATORE:

Dr. Gabriele Russo Russo

MATRICOLA:

0334319

Indice

Introduzione	1
1 Background	4
1.1 Inferenza distribuita di un Modello	4
1.1.1 Local Computing	5
1.1.2 Edge Computing	7
1.1.3 Split Computing	8
1.1.3.1 Split Computing senza modifica della rete	9
1.1.3.2 Split Computing con modifica della rete	12
1.2 Quantizzazione di un Modello	14
1.2.1 Concetti generali	14
1.2.2 Strategie di Quantizzazione	15
1.2.2.1 Uniforme VS Non-Uniforme	15
1.2.2.2 Simmetrica VS Asimmetrica	16
1.2.2.3 Statica VS Dinamica	18
1.2.2.4 Granularità di Quantizzazione	19
1.2.3 Quantization Aware Training VS Post Training Quantization .	19
1.2.4 Altri Aspetti	20

1.2.4.1	Quantizzazione Simulata VS Quantizzazione Integer-Only	20
1.2.4.2	Quantizzazione a Precisione Mista	21
1.2.4.3	Supporto Hardware	21
2	Lavori Affini e Frameworks	23
2.1	Lavori Affini	23
2.2	Frameworks	26
2.2.1	Onnx	26
2.2.2	Modelli Yolo	28
2.2.2.1	Esportazione dei Modelli	32
2.2.2.2	Pre elaborazione dell'Input	32
2.2.2.3	Post Elaborazione dell'Output	33
2.2.3	OnnxRuntime	39
2.2.4	Altri strumenti	45
3	Architettura del Sistema	47
3.1	Componenti del Sistema	47
3.1.1	Model Profiler	50
3.1.1.1	Il Problema del Rumore di Quantizzazione	51
3.1.1.2	Modello di Regressione	55
3.1.1.3	Esempio di Uso del Modello	57
3.1.1.4	Misura del Rumore	60
3.1.2	ExecProfiler	62
3.1.3	ServerMonitor	63
3.2	Fasi del Sistema	64

3.2.1	Caricamento del Modello	64
3.2.2	Generazione del Piano	66
3.2.3	Deployment del Piano	68
3.2.4	Inferenza	70
4	Problema di Ottimizzazione	72
4.1	Modello di Base	72
4.2	Le Variabili	75
4.3	Modello di Tempo	76
4.3.1	Tempo di Calcolo	76
4.3.2	Tempo di Trasmissione	77
4.3.3	Tempo Complessivo	80
4.4	Modello di Energia	80
4.4.1	Energia di Calcolo	80
4.4.2	Energia di Trasmissione	80
4.4.3	Energia del Server k-esimo	81
4.4.4	Consumo Energetico Complessivo	81
4.5	Modello di Memoria	81
4.6	Modello di Rumore di Quantizzazione	82
4.7	I Vincoli	84
4.7.1	Vincoli di Quantizzazione	84
4.7.2	Vincoli di Assegnazione	84
4.7.3	Vincoli di Flusso	84
4.7.4	Spazio delle Variabili	86
4.8	Il Problema	86

4.9	Normalizzazione dell'Obiettivo	87
4.10	Post Elaborazione della Soluzione	89
5	Aspetti Implementativi	96
5.1	Profiling del Modello	96
5.2	Profiling dell'Esecuzione	101
5.3	Produzione del Piano	103
5.4	Divisione del Modello	107
5.4.1	Configurazione di Quantizzazione	107
5.5	FrontEnd ed Inferenza	108
6	Esperimenti e Risultati	112
6.1	Profiling del Modello	113
6.2	Set-Up degli Esperimenti	113
6.2.1	Uso dell'affinity	116
6.3	Accuratezza della Predizione	117
6.3.1	Differenza tra tempo di inferenza del modello e somma dei tempi dei livelli	117
6.3.2	Interpolazione delle Misure	120
6.3.3	Confronto: Valore Reale vs Valore Predetto	121
6.3.3.1	Solo Device	121
6.3.3.2	Device + Edge	122
6.3.3.3	Device + Edge + Cloud	124
6.4	Confronto con Baseline	125
6.4.1	Confronti Latenze	127
6.4.2	Confronti Energia	128

6.5	Divisione della Computazione ed Uso della Quantizzazione	132
6.5.1	Device+Edge	133
6.5.2	Device+Edge+Cloud	135
6.5.3	Parallelismo dell’Inferenza	137
6.6	Scalabilità del Problema	139
6.6.1	Costruzione della Rete	141
6.6.2	Configurazione Statica	141
6.6.2.1	Costruzione del Modello	142
6.6.2.2	Risultati	142
6.6.3	Configurazione Randomica	146
6.6.3.1	Costruzione del Modello	146
6.6.3.2	Risultati	147
6.7	Limite sul Consumo	149
6.7.1	Risultati	150
7	Limitazioni e Sviluppi Futuri	152
7.1	Limitazioni	152
7.1.1	Scalabilità del tempo di soluzione	152
7.1.2	Cambio delle Condizioni	152
7.1.3	Tolleranza ai Guasti	153
7.1.4	Aumento del numero di livelli quantizzabili	153
7.1.5	Profiling del Modello	154
7.1.6	Profiling dell’Esecuzione	154
7.1.7	Test del consumo energetico	155
7.2	Sviluppi Futuri	155

7.2.1	Piani adattativi e/o alternativi	155
7.2.2	Rumore di Quantizzazione	156
7.2.3	Classi di Richiesta	156
7.2.4	Integrazione dell'Early Exit	156
7.2.5	Introduzione del parallelismo	157
7.2.6	Integrazione delle parti del sistema	157
Conclusioni		158

Abstract

+ 1-2 frasi per spiegare perché è importante fare inferenza @ edge: es. L'adozione crescente di tecniche di DL in numerosi contesti applicativi motiva l'uso di risorse ai bordi della rete per eseguire task di inferenza in prossimità dei dati/utenti... (come nell'Introduzione)

L'esecuzione di inferenze di modelli di deep learning su device con risorse ridotte o limitate risulta spesso poco efficace, ponendo un limite non indifferente all'effettiva adozione di questi strumenti. Un esempio è l'esecuzione di questi modelli su dispositivi a batteria: l'alta domanda di calcolo di questi modelli può causare un calo molto veloce della carica.

Al contrario, invece, l'esecuzione dell'inferenza su server a livello dell'edge e del cloud permette l'uso di risorse più potenti, ma, spesso, distanti da raggiungere e con un consumo energetico, e quindi un impatto climatico, molto alto.

+1 frase per dire più in generale che è necessario utilizzare meccanismi e strategie per sfruttare in maniera opportuna e dinamica risorse In uno scenario di questo tipo, quindi, l'offloading dell'inferenza nell'edge-cloud nel continuum (spiegando cosa è). Tra queste poi parli di partizionamento e quantizzazione continuum può risultare vantaggioso. A partire dall'input, una parte della computazione può essere eseguita localmente fino a quando diventa conveniente trasmettere il risultato parziale; a quel punto l'elaborazione può proseguire su server edge o cloud, per poi tornare nuovamente sul device quando risultati più efficienti, fino al calcolo del dei modelli" risultato finale.

In questo panorama si inseriscono anche strumenti come la quantizzazione, che, tramite la riduzione della precisione dei valori numerici e la conseguente riduzione della complessità di calcolo, permettono di semplificare il modello e velocizzare l'inferenza. e ridurre l'occupazione di memoria del modello.

In questo lavoro ci proponiamo di individuare, per modelli di deep learning foca-

lizzati sull'elaborazione di immagini, una suddivisione in componenti e il relativo placement su dispositivi diversi nell'edge-cloud continuum, con l'obiettivo di ottimizzare sia il tempo di inferenza sia il consumo energetico, rispettando al contempo un vincolo sul rumore massimo di quantizzazione ammesso e sul consumo energetico massimo del device.

Diversi studi hanno affrontato il problema del deployment di modelli di deep learning nell'edge-cloud continuum, concentrandosi sull'ottimizzazione del tempo di inferenza o del consumo energetico. Il presente lavoro adotta un approccio di modellazione più generale: mentre molte ricerche precedenti hanno considerato modelli lineari per la distribuzione, qui il problema viene formulato in modo più generico come un graph assignment. Inoltre, si propone di combinare in un'unica funzione obiettivo tempo di inferenza ed energia, pesando opportunamente i due termini. Infine, viene invece affrontato il problema della predizione ~~trodotta nel problema la predizione del rumore di quantizzazione tramite regressione polinomiale, aspetto finora non affrontato nelle formulazioni incontrate.~~ *esistenti.*

Lo studio ha tenuto in considerazione diversi aspetti del problema e varie metodologie: al fine di integrare la quantizzazione, si è reso necessario uno studio su come modellare e predire il rumore che questa tecnica induce sul risultato del modello; per prevedere il tempo di esecuzione del modello, è stato necessario studiare un meccanismo di profiling dell'esecuzione del modello sui server disponibili; per ottenere la divisione, è stata modellata l'assegnazione come un problema di ottimizzazione lineare intera.

Al fine di analizzare l'efficacia della modellazione proposta, è stata realizzata un'architettura distribuita che è stata successivamente deployata su un'infrastruttura cloud reale, in cui i nodi presentassero differenti capacità di calcolo, così da simulare un contesto di deployment realistico.

La prestazione della soluzione implementata è stata valutata generando diverse istanze del problema di ottimizzazione, variando i pesi assegnati a tempo ed energia, cosa sono? per poi eseguire l'inferenza con modelli YOLO11, e analizzando il miglioramento nei valori di tempo di inferenza e consumo energetico al variare del rumore di quantizzazione. L'utilizzo iniziale dell'edge, e successivamente della combinazione edge–cloud, ha mostrato un miglioramento significativo nel tempo di inferenza. Parallelamente, è stata condotta un'analisi sulla distribuzione dei livelli e delle componenti tra i server al variare degli stessi parametri: questa ha evidenziato come la quantizzazione giochi un ruolo cruciale non solo nel ridurre il tempo di calcolo dei livelli, ma anche nel diminuire la dimensione dei risultati parziali, semplificandone la trasmissione. Infine, tramite la costruzione di modelli fittizi, è stato svolto uno studio sulla scalabilità del problema di ottimizzazione e sulla sua risoluzione in presenza di un vincolo sul consumo energetico del device.

Nonostante i buoni risultati ottenuti nel miglioramento del tempo di inferenza, la soluzione proposta presenta alcune limitazioni. La principale riguarda la scalabilità del problema di ottimizzazione: la ricerca della soluzione ottima risulta difficilmente estendibile a scenari con un elevato numero di server e apre la strada allo studio di euristiche in grado di fornire alternative praticabili. Inoltre, la mancata modellazione del parallelismo riduce la capacità predittiva rispetto al tempo di inferenza.

In conclusione, lo studio qui riportato si propone di essere un punto di partenza su cui innestare ed integrare soluzioni diverse per l'ottimizzazione dell'inferenza nel contesto dell'edge-cloud continuum, mostrando come alcune di queste soluzioni possano effettivamente essere di beneficio.

CAPITOLO 1 (anche Conclusioni sono numerate)

Introduzione

Oggi, l'uso dei modelli di deep learning sta diventando sempre più pervasivo: sistemi di riconoscimento vocale, di guida automatica e di analisi di immagini stanno rivoluzionando il mondo come lo conosciamo e si mostrano come strumenti promettenti per semplificare e migliorare la vita dell'essere umano.

inferenza (edge) != training (cloud)

Questi benefici, tuttavia, non sono privi di costi: le pipeline di AI si rivelano spesso computazionalmente onerose e dispositivi che si trovano ai bordi della rete, come sensori, smartphone o droni, si dimostrano a volte poco efficienti nello sfruttare a pieno questi strumenti. Da un lato, il peso della computazione e la limitatezza dell'hardware dei device possono rendere la fase di inferenza più lunga di quanto non si vorrebbe; dall'altro, i device che richiedono le inferenze sono spesso dotati di batteria che, con carichi di lavoro molto alti, tende a consumarsi.

In un contesto di questo tipo l'offloading dell'inferenza nell'edge-cloud continuum può rivelarsi vantaggioso: da un lato si riuscirebbe a ridurre i tempi di calcolo attraverso l'uso di risorse più potenti; dall'altro, riducendo la quantità di calcolo fatta in locale, si permetterebbe di ridurre il consumo energetico. Questi benefici devono essere allo stesso tempo bilanciati dai costi relativi alla trasmissione dei dati attraverso la rete.

Tuttavia, l'offloading del carico su server più potenti, non si propone come unica soluzione per la riduzione del tempo di inferenza. Una soluzione, altrettanto valida, è

rappresentata dalla possibilità di ridurre il carico computazionale dei modelli usando delle tecniche di **semplificazione**. Esempi di queste tecniche sono l'early exiting, in cui si interrompe l'inferenza quando si è raggiunto un livello di confidenza del risultato adeguato, e la post-training quantization (PTQ), in cui si riduce la precisione dei pesi per rendere più veloci i calcoli. Queste strategie, sebbene vantaggiose, vengono al costo di una diminuzione dell'accuratezza dei risultati del modello: l'interruzione prematura dell'inferenza con l'early exiting e la riduzione della precisione con la PTQ rendono il modello diverso da quello originariamente addestrato, portando, potenzialmente, ad un'accuratezza minore. Tra le due strategie di semplificazione, si è scelto di analizzare la PTQ e, come si vedrà, la sua integrazione all'interno del sistema richiederà alcuni passaggi non banali.

PTQ "system -level"; early-exit "model-level"

**Espandi
in sez.
"Obiettivi
della tesi"
(riprendi da
abstract cosa
proponi
di fare
esattamente)**

In questo lavoro, ci siamo proposti di trovare, per modelli di deep learning, un partizionamento in componenti e il placement di queste componenti al fine di ottimizzare una funzione obiettivo che combinasse in un'unica metrica tanto il tempo di inferenza quanto il consumo energetico, cercando di garantire il rispetto di alcuni requisiti di qualità. I requisiti su cui si è posta la maggiore attenzione sono stati il limite sul consumo energetico del device e il limite sul rumore di quantizzazione ammesso in fase di inferenza.

Sez. Organizzazione della Tesi

Nei capitoli seguenti si esporranno gradualmente i concetti di partenza da cui il lavoro ha preso le mosse per poi passare all'esposizione concreta del sistema e delle sue parti costituenti. Nello specifico, in Capitolo 1 verranno descritti i concetti di local, edge e split computing, mentre in Capitolo 2 verrà fatta una panoramica sui lavori affini e sugli strumenti usati nello sviluppo del sistema; in Capitolo 4 verrà descritto il problema di ottimizzazione formulato e la post elaborazione della soluzione per poi passare in Capitolo 5 ad una descrizione di alcuni dettagli implementativi; in

Capitolo 6 verranno descritti gli esperimenti fatti sul sistema e i loro risultati; per concludere in Capitolo 7 si valuteranno le limitazioni del sistema e i possibili sviluppi futuri.

Capitolo 1

Background

In questo capitolo vengono introdotti i concetti di base da cui questo lavoro prende le mosse.

Si definiranno e discuteranno le strategie di *Local Computing*, *Edge Computing* e *Split Computing*, insieme alle loro implicazioni nel mondo reale. Si passerà poi ad un approfondimento sulle possibili soluzioni di Split Computing, in particolare quelle con e senza modifica della rete.

Si procederà con una panoramica sulla *Quantizzazione* dei modelli di deep learning e sul loro effetto sull'accuratezza dei modelli, menzionando anche le varie tecniche di quantizzazione e i modi in cui queste possono essere integrate nello split computing al fine di ottimizzare la divisione del modello.

1.1 Inferenza distribuita di un Modello

Come descritto in [1] possiamo menzionare principalmente tre strategie nel contesto di inferenza di un modello di deep learning in contesto distribuito:

- *Local Computing.* In questo caso la computazione non viene delegata ad un altro dispositivo, ma eseguita completamente in locale sul device che genera la richiesta di inferenza.
- *Edge Computing.* In questo approccio la computazione viene completamente delegata ad un dispositivo edge che si suppone vicino al dispositivo sorgente della richiesta.
- *Split Computing.* Secondo questo approccio, il modello viene diviso in parti, tendenzialmente due, una "testa" e una "coda", le quali vengono distribuite sui dispositivi disponibili nel sistema seguendo un certo criterio.

Ad affiancare queste soluzioni per l'inferenza distribuita, possiamo menzionare una tecnica addizionale ovvero l'*Early Exiting*, in cui il modello viene modificato aggiungendo delle uscite anticipate dalla computazione; ogni uscita fornisce, via via che si va avanti nel calcolo, un'accuratezza maggiore del risultato. Questa tecnica non si pone come alternativa, bensì come supporto alle soluzioni di inferenza distribuita.

In Figura 1.1 viene riportata un'immagine esemplificativa dei vari modi di fare inferenza.

1.1.1 Local Computing

Nel caso del Local Computing, la computazione del modello viene eseguita completamente in locale. Se da un lato questo permette di evitare il trasferimento dei dati attraverso la rete, portando ad una riduzione del tempo di risposta, dall'altro la computazione in locale si rivela spesso infattibile: le risorse necessarie, tanto in termini di memoria quanto di capacità di calcolo, spesso eccedono quelle disponibili in locale. Inoltre, qualora il device sia un dispositivo a batteria con autonomia limitata,

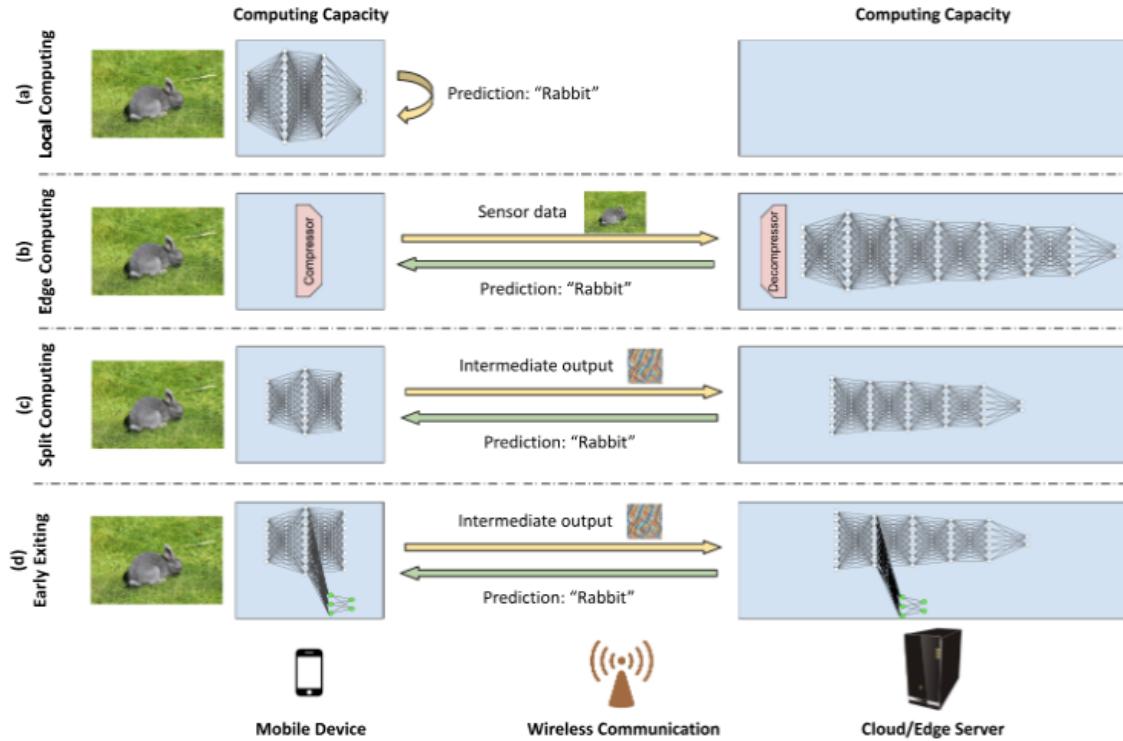


Figura 1.1: Diversi approcci di inferenza per modelli di DL

calcoli onerosi possono comportare un consumo energetico eccessivo, riducendo di conseguenza l'autonomia del dispositivo. Allo stesso tempo, evitare il trasferimento della computazione su un dispositivo edge o cloud permette di assicurare una privacy maggiore; se in alcuni casi questo potrebbe non essere importante, in contesti specifici, come quello medico o di assistenza vocale, la rilevanza è primaria.

Per far fronte ai lati negativi di questo approccio, spesso vengono usati dei modelli con un costo computazionale minore; allo stesso tempo però, l'uso di queste alternative comporta una riduzione dell'accuratezza dei risultati che, in contesti critici, potrebbe non essere accettabile. Le tecniche più usate in questo senso sono:

- Progettazione ed uso di modelli a costo minore. Un esempio ben noto di modello a costo computazionale minore, progettato specificatamente per il riconoscimento di immagine in ambito mobile, è la serie di reti *MobileNet*: alcune varianti

di MobileNet riescono a raggiungere un'accuratezza comparabile con quella di reti più complesse, come ResNet-34, pur riducendo di gran lunga il numero di parametri.

- Uso di modelli complessi appositamente compressi per ridurne il costo. Da questo punto di vista le tecniche più usate sono la model compression, che punta a ridurre il numero di parametri, e la quantizzazione, che punta invece ad usare lo stesso numero di parametri ma rappresentandoli con una precisione inferiore.
- Uso di tecniche di knowledge distillation. In questo approccio vengono usate due reti, una rete "insegnante", più grande e complessa, ed una rete "studente", più semplice e piccola: dato un input, l'output della rete insegnante viene usato per addestrare la rete studente ed aumentare la sua accuratezza. Il principio è proprio quello di trasferire ciò che la rete insegnante ha estratto dall'input alla rete studente in modo che possa apprendere da esso.
perché early exiting prima e non qui?

1.1.2 Edge Computing

Nell'Edge Computing l'input dell'inferenza viene trasferito da **un dispositivo limitato, il device**, ad un dispositivo terzo, l'edge, più potente in termini computazionali rispetto al primo e in cui la computazione viene eseguita in toto. Se da un lato questa soluzione permette di **non avere delle limitazioni in termini di dimensione o complessità del modello**, garantendo quindi il mantenimento della massima accuratezza, il tempo necessario al trasferimento attraverso la rete sia dell'input che dell'output potrebbe impattare negativamente il tempo di risposta.

L'impatto delle condizioni della rete può essere attenuato mediante l'uso di algoritmi di compressione/decompressione; nello specifico, l'input viene compresso attraverso

degli algoritmi eseguiti sul device e, una volta compresso, trasferito sul server edge; il server, una volta ricevuto l'input, lo decomprime ed esegue il modello. Questa soluzione consente di risparmiare tempo in termini di trasmissione, ma, di contro, introduce un carico aggiuntivo tanto sull'edge quanto sul device che, come detto, potrebbe non avere le risorse disponibili. In aggiunta, al variare dell'algoritmo di compressione, la ricostruzione lato edge potrebbe non essere tanto accurata da permettere di ottenere la stessa accuratezza che si sarebbe ottenuta senza compressione. In questo senso, ci sono molti algoritmi di compressione possibili, dal classico JPEG a quelli più , a loro volta, su modelli di ML.

1.1.3 Split Computing

Lo scopo principale dello Split Computing è quello di dividere la computazione del modello tra il dispositivo device e uno o più dispositivi server, cercando di ridurre il tempo di trasferimento dei dati tra i dispositivi partecipanti.

forse a inizio capitolo servirebbe una sezione per introdurre le reti Deep e

Assumendo un modello sequenziale, la soluzione più semplice potrebbe essere quella di cercare un livello l tale da permettere di fare una divisione del modello in un modello *head*, eseguito su device, e in un modello *tail* eseguito su server edge. L'individuazione di questo livello di split deve però essere fatta in maniera intelligente: da un lato è importante individuare un livello tale da avere un output il cui tempo di trasferimento attraverso la rete non sia eccessivamente alto; dall'altro bisogna fare una divisione tale da non sovraccaricare eccessivamente il device, pena ricadere nei contro del local computing.

Chiaramente, a seconda della posizione del livello l nel modello, lo split computing è riconducibile sia al local sia all'edge computing.

Gli approcci di Split Computing possono essere divisi in due categorie, a seconda

che modifichino o meno la struttura della rete: in ?? viene riportata un'immagine esemplificativa dei due casi.

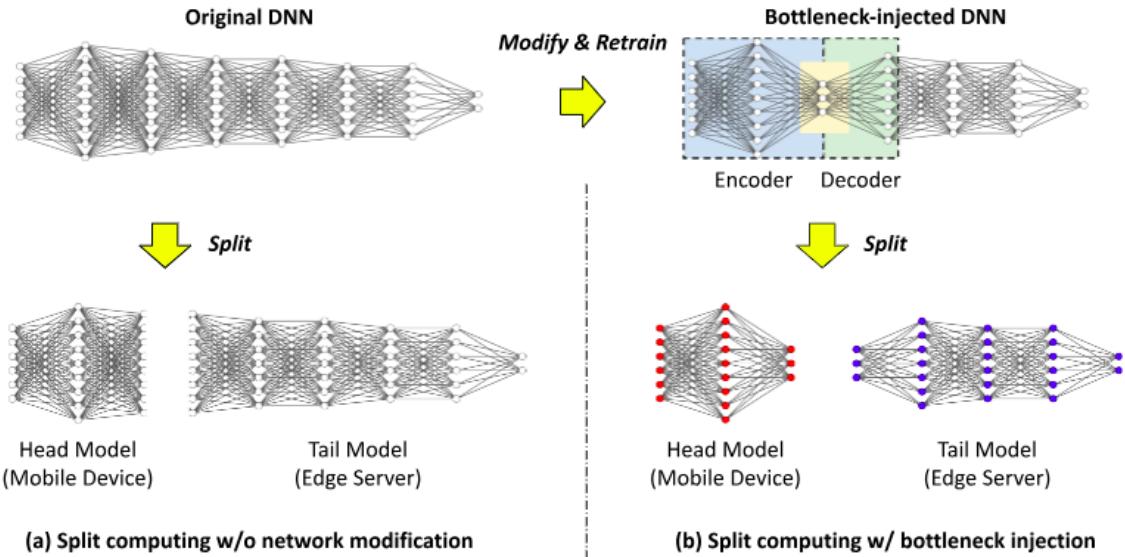


Figura 1.2: Enter Caption

1.1.3.1 Split Computing senza modifica della rete

In questo caso non vengono fatte modifiche della rete, ma viene semplicemente cercato il punto della rete che permette di dividere in maniera ottimale a seconda dei requisiti, che siano essi la latenza, il consumo energetico o altro. Come già evidenziato, uno degli aspetti fondamentali è la ricerca di un punto della rete tale per cui la dimensione dell'output del livello di divisione non comporti un tempo di trasferimento eccessivo; questi punti vengono chiamati *colli di bottiglia* della rete e non sono altro che dei punti presenti naturalmente all'interno del modello in cui le dimensioni dei dati si riducono rispetto al resto della rete. In un contesto semplificato, in cui si vuole ridurre il tempo di trasferimento **dati** dal device al dispositivo edge/cloud, è sufficiente cercare il collo di bottiglia più stretto all'interno del modello.

In questo senso ci sono due cose da sottolineare. In primo luogo non è detto che un collo di bottiglia sia sempre presente nel modello e, anche lo fosse, non è detto che si trovi in un punto significativo: in un modello in cui la dimensione del dato cresce mano mano che si va verso l'output della rete, il collo di bottiglia più significativo è rappresentato proprio dall'input e questo fa ricadere nel caso dell'edge computing. In secondo luogo, la presenza e lo sfruttamento del collo di bottiglia dipendono molto dalla complessità della rete: se un modello è lineare (i.e. non presenta rami di computazione paralleli) l'individuazione del collo di bottiglia risulta molto facile; se il modello invece diventa molto complesso, individuare il collo di bottiglia diventa complesso a sua volta.

Supponiamo di avere un modello $M(x)$, dove x rappresenta il vettore di input; vogliamo dividere il modello in due parti M_H e M_T tale che $M(x) = M_T(M_H(x))$; riconducendoci ai due casi indicati sopra abbiamo che:

- Se il modello è lineare, allora $M_H(x) = \hat{x}$ e quindi avremo $M(x) = M_T(\hat{x})$; la divisione del modello in questo contesto è chiaramente molto semplice.
- Se il modello è complesso, allora in generale $M_H(x) = (\hat{x}_1, \hat{x}_2, \dots, \hat{x}_q)$, ovvero il modello head potrebbe avere un numero maggiore ~~di 1~~ di output, rendendo il sotto modello tail M_T un modello a più input. In questo caso si ha un aumento del numero di modi in cui il modello può essere diviso, portando ad un aumento della complessità del problema. Si noti tra le altre cose che la ricerca di un punto in cui si può dividere il modello in modo che $q = 1$ non è sufficiente per semplificare il problema: il livello in cui $q = 1$ potrebbe avere una dimensione di output molto grande che potrebbe essere sconveniente rispetto ad avere un numero maggiore di output ma più piccoli.

In aggiunta al tempo necessario per il trasferimento dei dati, c'è poi un altro aspetto da considerare, ovvero il tempo che può essere necessario per il calcolo dei livelli prima del raggiungimento del collo di bottiglia. Ponendoci sempre nel contesto semplificato di un modello lineare, supponiamo che vi siano più colli di bottiglia all'interno del modello, non necessariamente della stessa dimensione: siano ad esempio $\{l_1, l_2, \dots, l_b\}$ i possibili livelli che ci danno un restringimento della size e sia l_b il livello che ci dà il restringimento massimo. Prima di ottenere il restringimento dei dati dato da un livello, bisogna **calcolare tutti i livelli** precedenti a quel livello; a seconda delle capacità computazionali del device, potrebbe essere più conveniente fermarsi e trasferire i dati dopo un certo collo l_s con $s < b$ perché il calcolo dei livelli fino a b potrebbe essere controproducente: il sistema potrebbe sì pagare un costo maggiore per il trasferimento dei dati, ma questo sarebbe compensato da un risparmio in termini di velocità di calcolo sul dispositivo edge. Questo ci mostra come il restringimento non solo deve essere presente all'interno della rete, ma deve trovarsi anche in un punto della rete tale da rendere il trasferimento vantaggioso: tendenzialmente, a seconda della capacità di calcolo del device, dovrà trovarsi sufficientemente vicino all'input.

La divisione del modello può essere fatta in due modi principali:

- questo è poco in linea con i ragionamenti appena fatti sopra (o sbaglio?) farei la distinzione
- **Parallelismo di Livello.** In questo caso, ad essere diviso è la computazione al-

prima
l'interno del singolo livello e la divisione viene fatta livello per livello: nello specifico quindi il tensore di input viene diviso orizzontalmente e le varie parti vengono fatte elaborare dai livelli parallelamente. Affinché questa soluzione porti beneficio c'è bisogno di livelli che siano parallelizzabili in modo opportuno, come l'operatore di convoluzione: in questo caso l'input ha (solitamente) tre dimensioni (*numeroCanali, larghezza, altezza*) e la computazione può essere parallelizzata su una qualunque di queste dimensioni; ad esempio si potrebbe

fare in modo che ogni "copia" dell'operatore calcoli un solo canale su tutta la coppia (*larghezza, altezza*) oppure calcoli tutti i canali su una sotto parte di larghezza e altezza. Una soluzione di questo tipo è adottata ad esempio in [2] e [3].

- Parallelismo di Modello. Questo è il caso che è stato discusso fino ad adesso, in cui i livelli del modello vengono raggruppati costruendo dei sotto modelli; l'output di un sotto modello viene usato come input dei sotto modelli successivi. Questa soluzione è in qualche modo riconducibile al problema del *Graph Partitioning* in cui cerchiamo i cut ottimali del grafo al fine di minimizzare la latenza di trasmissione e di computazione.

1.1.3.2 Split Computing con modifica della rete

Come descritto in precedenza, la ricerca di un collo di bottiglia all'interno del modello risulta fondamentale al fine di implementare una strategia di divisione del modello che sia funzionale. La presenza di questi punti all'interno della rete, però, dipende esclusivamente dall'architettura della rete. I casi estremi in questo senso sono:

- Il livello di output è quello a dimensione minore dell'output; questo porta all'esecuzione del modello completamente in locale.
- Il livello di input è quello a dimensione di output minore; questo porta all'esecuzione del modello completamente sull'edge.
- Il livello a dimensione minore è intermedio, ma la sua posizione nel modello porta a ricadere in uno dei casi tra edge e local computing.

Per evitare che le situazioni sopra citate si verifichino, una soluzione adottabile è quella dell'introduzione di colli di bottiglia artificiali all'interno del modello. Questa

tecnica prevede l'introduzione di alcuni livelli all'interno del modello tali per cui la dimensione dell'output di questi livelli renda vantaggiosa la trasmissione e l'offloading della computazione su un dispositivo più potente. Questi livelli aggiuntivi fungono da livelli di encoder-decoder dei dati, in modo simile all'uso che potrebbe essere fatto di un algoritmo di compressione dell'input.

Consideriamo a titolo di esempio un modello lineare $M = \{l_1, l_2, \dots, l_n\}$ formato da n livelli. Questa soluzione prevede l'introduzione di alcuni livelli $\{b_1, b_2, \dots, b_t\}$ in un certo punto del modello, in modo da ottenere un modello

$M' = \{l_1, \dots, l_q, b_1, b_2, \dots, b_t, l_{q+1}, \dots, l_n\}$ tale che:

- La dimensione dell'input di b_1 coincide con la dimensione dell'output di l_q ;
- La dimensione dell'output di b_t coincide con la dimensione dell'input di l_{q+1} (che è anche quella di l_q);
- Esiste un livello b_p all'interno dei livelli aggiuntivi che rende la dimensione di trasmissione vantaggiosa per l'offloading.

Il modello M' verrà quindi diviso in corrispondenza del livello b_p , dove i livelli da b_1 a b_p faranno da livelli di codifica, mentre quelli da b_{p+1} a b_t faranno da livelli di decodifica.

L'uso di una soluzione di questo tipo permette quindi una generalizzazione dell'approccio di base dello split computing. I livelli aggiuntivi, tuttavia, devono essere addestrati in maniera tale da permettere una rappresentazione intermedia che sia sufficientemente buona da non portare a perdita di accuratezza del modello complessivo. In generale, quindi, mentre nel primo caso, sfruttando i colli di bottiglia naturali del modello, non è necessario riaddestrare la rete, nel secondo caso, la tecnica di fine-tuning adottata risulta critica al fine dell'accuratezza del risultato.

Per concludere, non soltanto l'introduzione di un collo di bottiglia è importante per permettere l'offloading, ma il punto in cui questi vengono inseriti fa la differenza. Essendo la parte precedente al collo di bottiglia eseguita sul device e avendo il device dei limiti sotto vari punti di vista (consumo energetico, memoria, ecc.), il restringimento deve essere posto in un punto della rete tale da rendere la parte iniziale del modello sufficientemente leggera da essere eseguibile sul device.

1.2 Quantizzazione di un Modello

Come accennato in precedenza, con quantizzazione si intende la riduzione della precisione con cui i valori numerici del modello sono rappresentati all'interno del calcolatore. Con riferimento a [4] e descriviamo di seguito le principali tecniche di quantizzazione insieme ai loro vantaggi e svantaggi.

1.2.1 Concetti generali

Il concetto di quantizzazione nasce in ambito matematico e ha trovato applicazione in diversi contesti: calcolo approssimato, analisi di dati, elaborazione di segnali ecc. In generale, laddove abbiamo un problema in cui sono coinvolti valori continui, la quantizzazione può essere introdotta.

Se scrivi così ci si aspetta un riferimento al lavoro dove viene introdotto

Shannon introduce il concetto di *variable-rate quantization*: al fine di ottimizzare il rapporto tra memoria usata e perdita di informazione, il numero di bit per rappresentare un evento dovrebbe essere proporzionale alla probabilità con cui l'evento avviene.

Nell'ambito delle reti neurali, tuttavia, la quantizzazione ha cominciato a giocare un ruolo molto importante rispetto al passato. Questo è dovuto principalmente ai seguenti motivi:

- Le reti neurali tendono ad avere un numero di parametri molto alto, a volte anche più alto di quanto non sia strettamente necessario. La quantizzazione può quindi essere usata per ridurre il peso di questi modelli all'interno di un elaboratore.
- Le reti neurali tendono ad essere tolleranti rispetto alla quantizzazione e alla discretizzazione nella produzione del loro risultato.

Assumendo di avere un modello di ML, il target della quantizzazione possono essere tanto i pesi del modello, quanto i valori delle attivazioni, ma in generale lo scopo rimane quello di ridurre la precisione con cui questi valori vengono rappresentati al fine di ridurre il peso del modello in memoria e velocizzare il calcolo in fase di inferenza.

1.2.2 Strategie di Quantizzazione

Le diverse soluzioni di quantizzazione che vengono riassunte di seguito possono essere combinate tra loro in vari modi; ciò rende il problema della quantizzazione un problema di grande interesse e sviluppo per via della varietà con cui soluzioni possono essere trovate.

1.2.2.1 Uniforme VS Non-Uniforme

Detto r il valore di partenza, S un fattore di scala e Z un valore intero, la quantizzazione uniforme è definita come segue:

$$Q(r) = \text{Int}\left(\frac{r}{S}\right) - Z$$

Questo operatore quindi mappa una valore reale r su un certo insieme di interi; si parla di uniforme perché i possibili valori su cui il valore r può essere mappato sono equidistanti tra loro. La quantizzazione uniforme è l'approccio più usato per la

quantizzazione di modelli di ML per via della sua semplicità ed efficacia. Allo stesso tempo è possibile definire l'operazione ~~inversa~~ **di de-quantizzazione** come segue:

$$\hat{r} = S \cdot (Q(r) + Z)$$

Questa operazione non è esattamente l'inversa: a causa dell'arrotondamento, il valore ottenuto \hat{r} sarà diverso rispetto al valore di partenza r .

Approccio opposto a questo è quello di quantizzazione non uniforme, in cui i valori possono non essere equidistanti. La quantizzazione non uniforme può essere definita come segue:

$$Q(r) = X_i \quad \forall r \in [\Delta_i, \Delta_{i+1})$$

Semplicemente, tutti i valori reali che ricadono in un certo intervallo sono mappati su un certo valore intero; la lunghezza dei vari intervalli può, appunto, non essere uguale. Il vantaggio dell'uso di un metodo di questo tipo è che permette di rappresentare in maniera più fitta valori che si trovano in un range più frequente; se pensiamo ad esempio ad una distribuzione di valori gaussiana, allora ha senso rappresentare con accuratezza maggiore i valori attorno alla media e con accuratezza minore i valori sulle code. Questi metodi in generale sono complicati da sfruttare efficacemente su **implementare** GPU e CPU e per questo sono poco usati.

1.2.2.2 Simmetrica VS Asimmetrica

Considerata la definizione di quantizzazione uniforme riportata in precedenza, uno degli aspetti fondamentali da considerare è la ricerca del valore di S e Z che definiscono la funzione Q .

Partendo dal valore di scala S , questo valore definisce un certo numero di slot; dato un valore reale r questo ha corrispondenza con un unico slot. In generale, il

valore di S può essere definito come:

$$S = \frac{\beta - \alpha}{2^b - 1}$$

Dove:

- α e β sono i valori limite dell'intervallo dei numeri reali che stiamo mappando; questo significa che $\forall r, r \in [\alpha, \beta]$.
Q(r)?
- b è il numero di bit della rappresentazione quantizzata.

A loro volta, i valori di α e β devono essere calcolati. Questo si può fare in modo semplice per quanto riguarda i pesi del modello, in quanto statici. Per i valori delle attivazioni, invece, questo richiede una calibrazione del modello; la calibrazione prevede, tendenzialmente, l'esecuzione del modello su alcune istanze di input al fine di determinare il range di valori che una certa attivazione può avere.

Noti i range in cui i valori target della quantizzazione si trovano, è necessario stabilire i valori di α e β . Considerando il significato di questi valori nella definizione di S , la cosa più semplice è scegliere rispettivamente il primo come valore minimo del range ed il secondo come il valore massimo. Distinguiamo in generale due casi:

- Se $\alpha \neq -\beta$ si parla di *quantizzazione asimmetrica*. Questo tipo di quantizzazione risulta chiaramente più vantaggioso in termini di accuratezza se ci troviamo davanti a livelli che producono risultati in un range di valori asimmetrico, come la ReLU.
- Se $\alpha = -\beta$ si parla di *quantizzazione simmetrica*. Questo tipo di quantizzazione semplifica la definizione della funzione Q perché permette di considerare $Z = 0$, rendendo anche più veloce l'esecuzione del livello quantizzato su alcuni tipi di hardware.

La scelta del min/max per questi valori è la scelta più semplice, ma può anche portare a problemi in caso di outlier, specialmente nelle attivazioni: infatto, la presenza di outlier porterebbe ad un ampliamento del range dei valori, riducendo l'accuratezza della quantizzazione; come principio generale, tanto più si riesce a ridurre la distanza tra α e β , tanto più si riesce a mantenere l'accuratezza del modello.

Due varianti molto comuni per il valore di S sono le seguenti:

- $S = \frac{2\max(|r|)}{2^n - 1} \in [-128, 127]$; si parla in questo caso di *full range*; questa risulta in un'accuratezza maggiore, ma può portare a problemi di overflow in alcuni casi, essendo l'estremo inferiore corrispondente con il minimo intero con segno rappresentabile ad 8 bit.
- $S = \frac{\max(|r|)}{2^{n-1} - 1}$; si parla in questo caso di *restricted range*; questa rappresentazione permette di ridurre la possibilità di overflow, ma può portare ad accuratezza minore.

1.2.2.3 Statica VS Dinamica

Questa differenziazione riguarda il momento in cui i valori dei range vengono determinati. Nel caso dei pesi, essendo questi fissi per il modello, possono essere determinati offline senza problemi. Per le attivazioni, invece, la situazione è leggermente più complessa: infatti, il range del valore di attivazione può essere diverso ad ogni inferenza; quindi, per avere una quantizzazione più accurata, potrebbe essere benefico calcolare i range ad ogni inferenza.

Si distinguono quindi i due casi di:

- *Quantizzazione Statica*. In questo caso, la statistica necessaria al calcolo del range viene calcolata offline. Il calcolo viene fatto attraverso una serie di infe-

renze su un dataset di calibrazione: per ogni istanza del dataset di calibrazione viene calcolata la statistica che poi contribuirà al calcolo del range definitivo.

- *Quantizzazione Dinamica.* In questo caso, la statistica necessaria al calcolo del range viene ricalcolata ad ogni inferenza. Essendo il range ad-hoc per quella specifica inferenza, l'accuratezza del modello quantizzato risulterà maggiore, ma, di contro, vi è un costo non indifferente per il calcolo del range a run time.

1.2.2.4 Granularità di Quantizzazione

La granularità a cui la quantizzazione si può applicare si riferisce ad alcuni tipi specifici di livelli, come i livelli convoluzionali: in questo tipo di livelli, infatti, ogni filtro può avere un range di valori diverso; il concetto di granularità si riferisce a come calcolare il range per questo tipo di livelli.

Distinguiamo due casi principali:

- *Per livello.* In questo caso il range è calcolato considerando tutti i filtri del livello. Chiaramente, in caso di alta variabilità nei pesi dei filtri, l'accuratezza può essere impattata anche molto negativamente.
- *Per canale.* In questo caso il range è calcolato in modo indipendente per ogni filtro.

1.2.3 Quantization Aware Training VS Post Training Quantization

Dato un modello pre addestrato, la quantizzazione del modello, e quindi la modifica dei suoi parametri, può portare il modello ad allontanarsi dal punto di convergenza raggiunto in fase di addestramento con precisione floating point.

Possiamo distinguere due possibili soluzioni per far fronte a questo problema:

la PTQ mi sembra più la causa del problema di cui sopra che non una soluzione

- *Quantization Aware Training.* In questo caso, l’addestramento viene fatto sul modello originario in floating point, ma i parametri del modello sono quantizzati ad ogni passo di addestramento: in questo senso, è come se si introducesse un rumore sui pesi; addestrare il modello in presenza di questo rumore rende il modello più tollerante a quel rumore in fase di inferenza, portando ad una minore penalizzazione dell’accuratezza. Alcuni approcci non si limitano a rendere il modello tollerante alla quantizzazione facendo l’addestramento tenendo in conto la quantizzazione stessa, ma calcolano i parametri stessi della quantizzazione, come ad esempio i range dei valori, come parametri del modello. Lo svantaggio principale di questo approccio è, in generale, il bisogno di (ri-)addestrare il modello per porre rimedio alla perdita di accuratezza dovuta alla quantizzazione; per farlo, possono essere necessarie anche molte epoche.
- *Post Training Quantization.* In questo caso, la quantizzazione viene eseguita senza fine-tuning. Oltre al risparmio in termini di tempo, non abbiamo bisogno di un dataset di riaddestramento per porre in atto questa tecnica; lo svantaggio è chiaramente la perdita maggiore di accuratezza.

1.2.4 Altri Aspetti

1.2.4.1 Quantizzazione Simulata VS Quantizzazione Integer-Only

I due approcci principali per eseguire un modello quantizzato sono quello *simulato* e quello *integer-only*.

Nell’approccio *simulato*, i pesi sono mantenuti come *INT8*, ma le operazioni sono di fatto eseguite in floating point; prima dell’esecuzione dell’operazione effettiva, quindi, per garantire compatibilità, è necessario eseguire la dequantizzazione dei parametri. Questo permette di mantenere una maggiore accuratezza del modello e ridurre il

consumo di memoria, ma riduce i benefici in termini di tempo di inferenza, in quanto è presente un overhead aggiuntivo dato dalla dequantizzazione. Nonostante ciò, questa soluzione può essere utile in contesti in cui l'esecuzione è I/O bound piuttosto che CPU bound: se il collo di bottiglia dell'esecuzione è il caricamento dei dati in memoria, caricare dati a precisione (e quindi dimensione) minore, riduce i tempi anche se le operazioni sono poi eseguite in floating point.

Nella variante *integer-only*, invece, le operazioni sono completamente eseguite in *INT8*, con conseguente beneficio in termini di tempo e/o consumo energetico, ma perdita in termini di accuratezza.

1.2.4.2 Quantizzazione a Precisione Mista

Questa soluzione prevede di non usare un unico tipo di precisione per il modello completo, bensì di usare una precisione diversa per i parametri di ogni livello. Questo permette di trovare la configurazione di precisioni migliore che permette di non scendere sotto una certa soglia di accuratezza pur mantenendo i benefici della computazione a precisione minore. Il problema principale di questo approccio è lo spazio di ricerca molto vasto: questo non solo cresce con il numero di livelli, ma anche con il numero di precisioni possibili che possiamo supportare; detto B l'insieme delle precisioni ammissibili e L l'insieme di livelli del modello, abbiamo un totale di $|L|^{|B|}$ possibili combinazioni di quantizzazione.

1.2.4.3 Supporto Hardware

Il beneficio che si trae dalla quantizzazione non è scontato: costruire un modello quantizzato non è sufficiente per migliorare il tempo di inferenza. Il beneficio che si ottiene dalla quantizzazione del modello dipende soprattutto dal supporto hardware delle operazioni quantizzate: eseguire un modello quantizzato su un hardware che non

supporta le operazioni nella precisione che è stata usata per la quantizzazione non solo potrebbe non dare miglioramenti nel tempo di inferenza, ma essere addirittura deleteria.

In aggiunta, è importante che anche il framework di esecuzione sia in grado di gestire il modello quantizzato; i caso contrario potrebbero essere inserite delle ridondanze per allineare le precisioni dei livelli, portando alla perdita del beneficio.

Capitolo 2

Lavori Affini e Frameworks

Stato dell'Arte?

In questo capitolo verranno prima analizzati lavori affini a questo che hanno affrontato il problema della distribuzione di modelli di DL in ambienti distribuiti.

Successivamente si passerà ad una panoramica riguardante gli strumenti usati ~~nello sviluppo della soluzione~~ ~~qui proposta~~ ~~nell'abstract scritto come YOLO11~~ questo. Si partirà con una discussione riguardante *Onnx*, usato come formato di rappresentazione del modello, per poi passare ad una spiegazione del modello scelto come target, cioè *Yolo11*. Si chiuderà con una spiegazione riguardante *OnnxRuntime* usato come framework per l'inferenza del modello.

Riguardo il modello Yolo, si approfondiranno i dettagli necessari alla pre-elaborazione dell'input e alla post-elaborazione del risultato, ~~aspetti che, come vedremo, non sono gestiti in modo trasparente dall'implementazione del modello che è stata utilizzata ai fini del progetto.~~

~~Forse meglio essere + precisi: Soluzioni per Inferenza tramite Split Computing?~~

2.1 Lavori Affini

In questa sezione vengono analizzate alcune soluzioni proposte per problemi simili a quello in analisi.

In [5] una delle domande che ci si pone è proprio quale sia il costo del trasferimento dei dati attraverso la rete e quando il pagamento di questo costo è giustificato al fine di ottimizzare l'inferenza. Per rispondere a questa domanda vengono prima di tutto costruiti dei modelli per prevedere, dato un dispositivo e un certo tipo di livello, quanto l'esecuzione di quel livello su quel server sia impattante in fase di inferenza: questo permette una generalizzazione ad architetture diverse. Il punto di *cut* possibile è considerato dopo ogni livello. In generale, si considerano dei modelli tendenzialmente lineari e la divisione della computazione in una testa e in una coda.

In [2] viene proposta una soluzione serverless per l'esecuzione del modello in componenti parallele seguendo un paradigma *fork-join*. L'approccio scelto in questa soluzione è quello del *tensor parallelism*, in cui i calcoli che devono essere eseguiti sulle diverse dimensioni del tensore di input vengono parallelizzati su più nodi; per fare questo vengono creati dei layer-group, ovvero gruppi di livelli in cui la computazione sulle parti dei tensori può essere considerata indipendente e quindi parallelizzabile su più funzioni serverless. La definizione dei gruppi permette di ridurre i punti di join e i conseguenti ritardi dovuti alla rete.

In [3] ci si focalizza principalmente sulla divisione della computazione tra più dispositivi edge, ripartendo il calcolo tra i dispositivi in base alle loro capacità di calcolo al fine di ottimizzare il consumo energetico complessivo con limite superiore al tempo di inferenza. Anche in questo caso si sfrutta il *tensor parallelism*: l'immagine di input viene partizionata e le sotto parti distribuite tra i diversi dispositivi, con la computazione che procede in parallelo fino al punto di join finale per l'aggregazione del risultato.

In [6] si punta ad una soluzione dinamica che permetta di cambiare configurazione tra local, edge e split computing in un ambiente dinamico con l'obiettivo di bilanciare

tra l'uso efficiente delle risorse e la precisione del risultato. Il trasferimento di dati in caso di edge e split computing viene effettuato in modo efficiente usando una soluzione di encoding-decoding per la compressione.

In [7] si esplora lo scenario in cui esistono più cammini da un nodo di partenza, il device, e un nodi di arrivo, l'edge: i vari dispositivi su questi cammini possono eseguire una parte della computazione del modello fino ad arrivare alla soluzione, salvata sul dispositivo edge. L'obiettivo è quello di scegliere il cammino migliore che permetta di minimizzare il tempo di inferenza complessivo per diversi modelli, ognuno dei quali serve per portare a termine uno specifico task. La rete di server è considerata un dag, con sorgente il device e destinazione l'edge, senza ammettere ritorni indietro del risultato. Vengono identificati dei punti di cut del modello supponendo che questi punti formino una successione decrescente delle dimensioni di output.

In [8] viene implementata una soluzione per scegliere dinamicamente per ogni task se adottare una divisione del modello di inferenza per tensore o per livello, con l'obiettivo di minimizzare sia il tempo di inferenza sia di massimizzare il rispetto di certi requisiti per il task; il principio è quello per cui ogni inferenza potrebbe avere dei requisiti diversi e quindi si punta a fare un'ottimizzazione per singola richiesta.

In [9] viene analizzata una soluzione di inferenza mista tra device e cloud con l'obiettivo di minimizzare il consumo energetico tenendo il tempo di inferenza sotto una certa soglia massima. Viene assunto un modello come una struttura lineare facendo una divisione in blocchi di layer che vengono mandati in deployment sul device o sul cloud. Oltre alla fase di inferenza, viene modellato il problema anche per la fase di addestramento del modello.

Oltre alla semplice lista di lavori, sarebbe utile avere almeno un cappello iniziale e una conclusione in cui evidenzi punti in comune/differenze tra i lavori e in particolare rispetto alla tesi

All'inizio potresti anche citare brevemente altri paper meno affini ma comunque su inferenza@Edge.
Es. Negli ultimi anni sono state proposte numerose soluzioni per facilitare o ottimizzare l'utilizzo di modelli

2.2 Frameworks

2.2.1 Onnx

Il nome Onnx [10] è l'abbreviazione per *Open Neural Network Exchange* che si propone di essere un formato aperto per la rappresentazione di modelli di ML, definendo un insieme di operatori che permette import/export in e da questo formato in modo semplice.

Da un punto di vista tecnico, ONNX fornisce una definizione estensibile di modello come grafo computazionale, insieme a una serie di operatori predefiniti e tipi di dati standard. Ogni modello è considerato come un insieme di nodi che formano un grafo aciclico; tali nodi hanno uno o più input e uno o più output, e ciascuno di essi rappresenta una chiamata ad un operatore corrispondente. Gli operatori sono implementati esternamente al grafo, ma l'insieme degli operatori predefiniti è portabile tra diversi framework: ogni framework che supporta ONNX fornisce implementazioni di questi operatori compatibili con i tipi di dati previsti.

Una volta che un modello è stato convertito in Onnx, l'ambiente di produzione necessita solo di un runtime che supporti il grafo definito con Onnx. Questo runtime può essere sviluppato in qualsiasi linguaggio valido per l'ambiente di produzione.

Il formato in cui i grafi Onnx sono salvati è Protobuf, usato al fine di ottimizzare quanto più possibile le dimensioni del modello. Gli operatori sono raggruppati in domini: l'esempio principale di dominio è *ai.onnx.ml* che contiene gli operatori di default usati nella maggior parte dei casi. Onnx permette anche la definizione di domini, e quindi operatori, custom, rendendolo, ancora, particolarmente estensibile: in questo senso, un esempio è rappresentato dal dominio di Microsoft usato in *OnnxRuntime* per ottimizzare la fase di inferenza di alcune operazioni.

Internamente, l'input e l'output degli operatori è rappresentato da tensori, ognuno dei quali è dotato di un *type* e *shape*: partendo da queste informazioni è possibile risalire all'occupazione di memoria del tensore, cosa che si rivelerà utile in fase di modellazione del problema ed implementazione della soluzione.

Esiste un gran numero di librerie che hanno come scopo la conversione di un modello al formato Onnx; esempi di queste librerie sono:

- *sklearn-onnx* [11]
- *tensorflow-onnx* [12]
- *torch.onnx* [13]

Onnx fornisce anche un insieme di API utile alla manipolazione del modello in modo semplice e veloce; tra queste API, quelle di maggiore rilevanza nel nostro caso ambito sono due::

- **infer_shapes** [14]. Questa API permette di ricostruire le dimensioni di tutti i tensori intermedi che attraversano il modello in fase di inferenza; partendo da questo quindi si può ricavare la magnitudo dei dati trasportati da questi tensori.
- **extract_model** [15]. Partendo da un modello di base, permette di estrarre un sotto modello specificando i nomi dei tensori di input/output del nuovo modello che vogliamo costruire.

Riassumiamo di seguito i motivi che hanno portato alla scelta di Onnx come formato di rappresentazione del modello:

- Vasto supporto a librerie di ML esistenti. Questo permette ad un utente di costruire e addestrare il suo modello in qualsiasi libreria di ML per poi esportare tale modello in Onnx e usufruire del sistema in modo trasparente.

- Ampia gamma di operatori supportati che permette l'export semplice di molti tipi di modello.
- Esistenza di ambienti di runtime con supporto ad ampio set di architetture hardware (come vedremo in seguito in sottosezione 2.2.3); ciò permette un deployment semplice dei modelli su diversi dispositivi target.
- Supporto nativo all'estrazione di sotto modelli e alla ricostruzione delle dimensioni dei tensori intermedi.
- Export supportato per modelli Yolo di Ultralytics, come vedremo in sottosezione 2.2.2

2.2.2 Modelli Yolo

Con il nome *Yolo*, ci si riferisce di solito ad una famiglia di modelli di computer vision atti a svolgere un insieme variabile di task. Questo lavoro si è focalizzato principalmente su modelli Yolo11 [16]. I motivi che hanno mosso questa scelta sono:

- Vi è un buon compromesso tra efficienza e velocità del modello, come testimoniato in Figura 2.1
- I modelli hanno alte accuratezze con un numero relativamente basso di parametri; questa non è una considerazione universale, ma dipende ampliamente dalla variante di dimensione che si sceglie.
- Adattabile in ambienti diversi e facilmente esportabile in molti formati, tra cui Onnx.
- Diversi task supportati, ovvero classificazione, detection, segmentazione, pose, oriented detection.

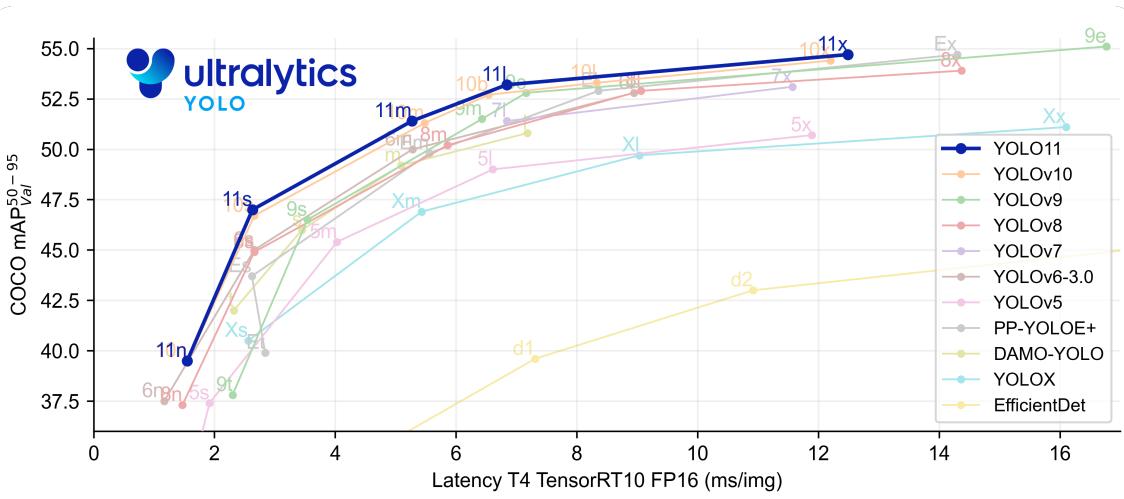


Figura 2.1: mAP dei modelli Yolo al variare della versione

Analizziamo brevemente quali sono i task che Yolo ci permette di svolgere e cosa significa risolverli:

classificazione

- Classification. Si tratta del tipo più classico di task di computer vision: data un'immagine di input con all'interno un oggetto appartenente ad un certo insieme target, si vuole identificare la classe a cui quell'oggetto appartiene. In questo caso quindi l'output del modello potrà essere, a seconda del tipo di modello, o la classe di appartenenza, oppure un insieme di probabilità in cui ogni valore corrisponde alla probabilità di appartenenza ad una certa classe.
- Detection. In questo caso, data un'immagine, si vogliono individuare le varie entità presenti al suo interno. L'output di un modello di questo tipo è un insieme di bounding box, la classe a cui l'oggetto appartiene e la probabilità di appartenenza. La bounding box serve ad incorniciare l'oggetto all'interno dell'immagine ed è tendenzialmente rappresentata in due formati: tramite le coordinate del centro nell'immagine più la coppia altezza e larghezza, oppure

tramite le coordinate di due angoli opposti. Questo task è utile nel momento in cui si vuole sapere se e dove un oggetto è presente nell'immagine. (Figura 2.2b)

Segmentazione

- Segmentation. In questo caso si spinge oltre il task di detection: oltre ad individuare le bounding box che individuano la posizione dell'oggetto nell'immagine di input, si vuole fare una classificazione dell'immagine per pixel; ciò significa che si vuole trovare una maschera della stessa dimensione dell'immagine tale per cui ad ogni pixel dell'immagine originaria sia assegnata la classe dell'entità a cui quel pixel appartiene. Questo permette di individuare delle maschere di classificazione e/o i contorni dell'oggetto all'interno dell'immagine. Si rivela utile quando si vuole sapere la forma di un oggetto, oltre a se e dove l'oggetto è presente nell'immagine. (Figura 2.2a)
- Pose Estimation. In questo caso si vogliono identificare dei punti nell'immagine, detti keypoints, che ci permettono di individuare varie parti di un oggetto, come ad esempio i punti di giunzione. L'output di questi modelli solitamente ci dice la posizione del punto e la confidenza di quel punto. Si rivela un task utile quando bisogna capire la posizione di un oggetto nello spazio e come le parti di quell'oggetto si posizionano le une rispetto alle altre. Esempi di keypoints che Yolo11 individua per il corpo umano sono naso, occhi, orecchie, spalle ecc. (Figura 2.2c)
- Oriented Detection. In questo caso si introduce una dimensione aggiuntiva al problema di detection: oltre ad individuare le bounding box si vuole anche individuare un loro angolo di inclinazione tale da permettere l'individuazione più precisa dell'oggetto all'interno dell'immagine.



Figura 2.2: Esempi di vari task di computer vision

Tutti i modelli Yolo, oltre ad avere delle varianti a seconda del task, hanno delle varianti in dimensioni: all'aumentare della dimensione del modello cresce il numero di parametri, il peso computazionale e anche l'accuratezza del modello; in Tabella 2.1 queste differenze si possono vedere chiaramente per un modello di detection. Osservando la tabella, inoltre, si può già vedere come il carico computazionale non sia affatto irrisorio, a maggior ragione se consideriamo il modello eseguito su un dispositivo a risorse limitate.

Modello	mAP (50-95) - Validation	Params [M]	FLOPS [B]
Yolo11n	39.5	2.6	6.5
Yolo11s	47.0	9.4	21.5
Yolo11m	51.5	20.1	68.0
Yolo11l	53.4	25.3	86.9
Yolo11x	54.7	56.9	194.9

Tabella 2.1: Parametri di modello Yolo al variare della dimensione

Considerato un modello Yolo, fissata la versione, gli aspetti che si possono esplorare sono due: il task che la variante risolve e la dimensione del modello stesso.

A seconda del tipo di task, Ultralytics mette a disposizione dei modelli preaddestrati su diversi dataset; in particolare:

- I modelli di detection, segmentation e pose sono addestrati su Coco;
- I modelli di classificazione sono addestrati su ImageNet;
- I modelli di oriented detection sono preaddestrati su DOTAv1.

Inoltre, l'interfaccia di accesso esposta da Ultralytics per l'inferenza del modello gestisce in modo trasparente la pre e la post elaborazione, sollevando l'utente da questo impegno.

Per quanto riguarda gli scopi di questo lavoro, ci si è concentrati in particolare su task di detection, classificazione e segmentazione.

2.2.2.1 Esportazione dei Modelli

Un altro aspetto che rende i modelli Yolo particolarmente adatti al nostro contesto è il fatto che Ultralytics mette a disposizione delle API di esportazione che permettono di ottenere una versione del modello in diversi formati, tra cui Onnx. L'API di esportazione si rivela molto flessibile, permettendo di personalizzare il modello in vari modi: è possibile, ad esempio, specificare la dimensione dell'immagine di input, la dimensione del batch e altri parametri di personalizzazione.

Tuttavia, mentre le API di Ultralytics gestiscono pre e post elaborazione in modo trasparente, questa gestione non viene riportata nel modello Onnx esportato, obbligando a gestirla a parte.

2.2.2.2 Pre elaborazione dell'Input

Per quanto riguarda l'input, il modello esportato si aspetta di ricevere un'immagine con le seguenti caratteristiche:

- Input di dimensione fissa. Per questo aspetto bisogna fare attenzione al rapporto esistente tra altezza e larghezza dell’immagine: data un’immagine non possiamo semplicemente ridimensionarla, ma è necessario mantenere il rapporto esistente tra le dimensioni, pena la deformazione dell’immagine e la scarsa performance del modello. Per far fronte a questo problema si può ridimensionare l’immagine mantenendo il rapporto tra le dimensioni costante ed aggiungere del padding in larghezza ed altezza per soddisfare le dimensioni che il modello si aspetta di ricevere.
- Formato dell’immagine di tipo channel first. In questo senso possiamo avere due formati: il channel last è un formato di rappresentazione dell’immagine di tipo (W, H, C) dove quindi il canale rappresenta l’ultima dimensione del tensore di input e si tratta del formato adottato in frameworks come Keras e TensorFlow; il formato channel first, in cui l’immagine è rappresentata come un tensore di tipo (C, W, H) con il canale come prima dimensione, è quella usata solitamente in modelli Torch (in cui Yolo è originariamente implementato).
- Formato di colori di tipo BGR e non RGB.
- Tensore dei colori normalizzato.

Tutte queste operazioni possono essere fatte in modo abbastanza semplice combinando operazioni matriciali e operazioni fatte usando la libreria *cv2* [17], che fornisce diverse utils per operazioni di computer vision.

2.2.2.3 Post Elaborazione dell’Output

Nel caso dell’output bisogna prima di tutto tenere in considerazione il task che si sta considerando.

Nel caso di modelli di classificazione, l'output è semplicemente un tensore di dimensione ($batch_size, num_classes$) in cui ogni elemento della seconda dimensione rappresenta la probabilità di appartenenza alla classe associata all'indice. In questo caso quindi, l'unica post-elaborazione necessaria è la ricerca del massimo all'interno del tensore per ogni elemento del batch.

Nel caso di modelli di detection, la situazione risulta leggermente più complessa vista la necessità di individuare delle bounding box che racchiudano l'oggetto nell'immagine. In Figura 2.3 è possibile vedere il formato tipico del tensore di output del modello di Yolo-Detection: nello specifico abbiamo tre dimensioni di size ($batch_size, 4 + num_classes, num_anchors$). Consideriamo un singolo elemento del batch e suppo-

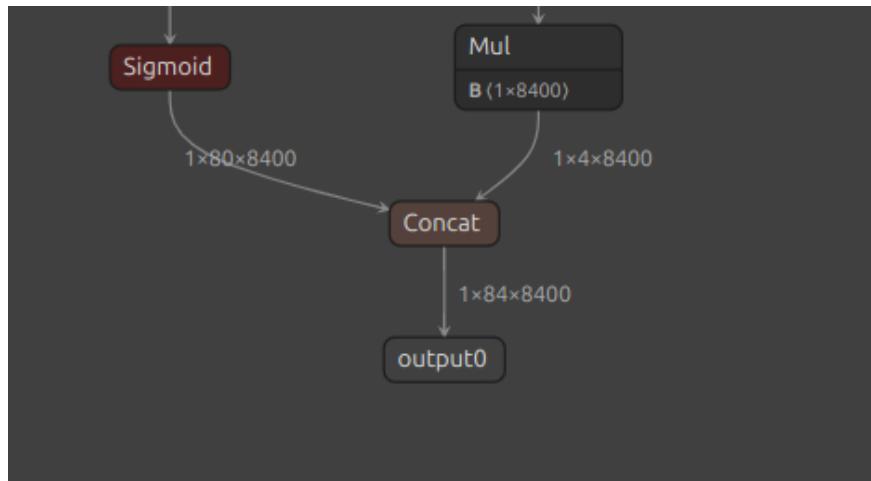


Figura 2.3: Output di modello Yolo-Detection

niamo di trasporre il tensore in formato ($num_anchors, 4 + num_classes$). Un'anchor non è altro che una possibile detection che viene fatta dal modello; ad ogni anchor vengono associati quattro valori seguiti da un numero di valori pari al numero di classi e, in particolare:

- I quattro valori rappresentano le coordinate della bounding box, solitamente rappresentata in formato ($x_center, y_center, bb_width, bb_height$); partendo

da questa rappresentazione si possono individuare le coordinate degli angoli della bounding box.

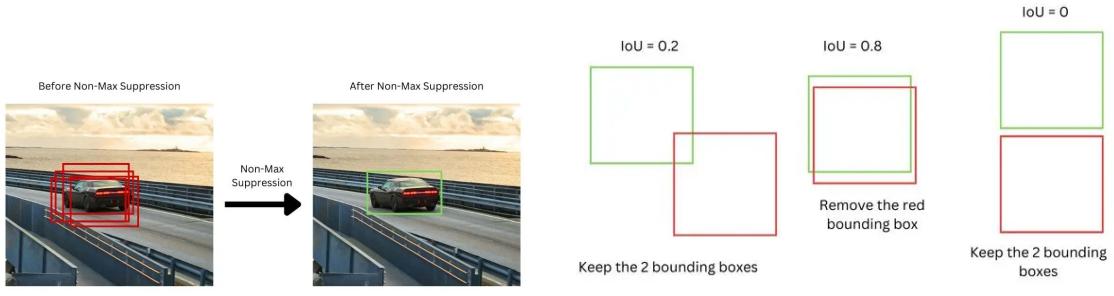
- I valori pari al numero di classi sono un vettore di probabilità; in questo vettore il singolo elemento rappresenta la probabilità che l'oggetto individuato da quella anchor sia un oggetto della classe associata all'elemento.

Vediamo i passi di post elaborazione di un vettore di questo tipo:

1. Estrarre la classe migliore per ogni anchor, riducendoci ad un tensore di size $(num_anchors, 6)$, dove i sei elementi sono $(x_center, y_center, bb_width, bb_height, class_id, class_score)$.
2. Sebbene a questo punto per ogni anchor vi sia lo score massimo per la classe associata, non è detto che questo score sia alto: se ad esempio le classi sono 100 in totale, è possibile che ad ogni classe venga assegnato uno score intorno a 0.01. Ciò rende necessaria l'introduzione di una soglia, detta $score_thr$, tale per cui l'anchor viene scartata se ha uno score inferiore; il tensore che otteniamo è quindi di size $(num_anchors', 6)$, dove $num_anchors'$ rappresenta il numero di anchors che passa l'imposizione della soglia.
3. Nonostante l'uso della soglia di scoring, è possibile che vi siano delle anchor che individuano lo stesso oggetto; tra queste anchor, è bene che ne sia mantenuta solo una per evitare problemi nella rappresentazione dell'output. Lo scarto delle anchor ridondanti viene fatto usando l'operazione di *non maximum suppression* (NMS) che, internamente, usa la metrica di *Intersection-Over-Union* (o IoU), atta a misurare quanto due bounding box si sovrappongono. Nello specifico viene selezionata la bounding box con score maggiore e viene calcolata l'IoU per

ogni altro box; se il valore calcolato è maggiore di una certa soglia, la seconda bounding box viene scartata; si passa poi alla bounding box successiva in ordine di score. L'effetto dell'uso della NMS si può vedere in Figura 2.4a, mentre in Figura 2.4b si può vedere un esempio di calcolo di IoU.

4. Il tensore ottenuto conterrà quindi soltanto le anchor finali che hanno passato entrambe le soglie.
5. L'ultimo passo di post-elaborazione è quello di rescaling delle bounding box alla dimensione dell'immagine originaria: le box così individuate si riferiscono, infatti, alla dimensione dell'immagine di input; tale dimensione però, vista la fase di pre-elaborazione, può però essere diversa rispetto all'immagine originale.



(a) Esempio di Non Maximum Suppression

(b) Esempio di IoU

Nel caso dei modelli di segmentazione, la post elaborazione necessaria è parzialmente simile a quella di modelli di detection. In Figura 2.5 è possibile vedere i formati degli output dei modelli di segmentazione. In questo caso troviamo due tensori di output che devono essere combinati per ottenere l'output finale. Analizziamo i due output, supponendo di trascurare la dimensione del batch:

- In $output0$, troviamo un tensore di size $(num_anchors, 4 + num_classes + mask_weight_size)$. In questo caso, la prima parte della seconda dimensione



Figura 2.5: Yolo Segmentation - Formato Output

ne contiene le stesse informazioni dell'output del task di detection, mentre nella seconda abbiamo delle informazioni aggiuntive relative alla maschera di segmentazione; questi valori non sono altro che dei pesi da combinare con il secondo output per ottenere la maschera di segmentazione finale. In questo esempio, considerando che $num_classes = 80$, troviamo una $mask_size = 32$, che è esattamente la prima dimensione del secondo output (sempre trascurando la dimensione del batch).

- In $output1$, troviamo le cosiddette *maschere prototipo*, che devono essere pesate con i pesi che si trovano nel primo output.

I passi di post elaborazione da fare in questo caso sono quindi:

1. La prima parte di post elaborazione è molto simile a quella della detection e ci permette di trovare dei tensori di size $(num_anchors'', 6 + mask_weight_size)$
2. A questo punto possiamo trovare la maschera di segmentazione, moltiplicando i pesi per i prototipi ottenuti nel secondo output; la maschera finale sarà quindi data da $f = mask_weights * prototypes$, dove l' f ottenuto è un tensore di size $(num_anchors'', mask_width, mask_height)$.

3. Come si vede dalle dimensioni in Figura 2.5, le dimensioni della maschera finale possono essere molto più piccole rispetto a quelle dell'immagine data in input al modello. Si rivela quindi necessario fare upscale della maschera ottenuta perché coincida con la dimensione dell'immagine originaria.
4. In definitiva troviamo due tensori, uno di size $(num_anchors'', 6)$ che individua le bounding box, l'altro di dimensione $(num_anchors'', width, height)$ che definiscono la maschera di segmentazione.

Una precisazione è d'obbligo rispetto al punto 2 della post elaborazione per modelli di segmentazione. Come si può vedere, è previsto un prodotto tra matrici che, in un contesto di limitatezza delle risorse, può essere molto oneroso e richiedere tempo, soprattutto considerando le dimensioni dei tensori coinvolti come evidenziati in Figura 2.5. Tuttavia, bisogna tenere in conto il fatto che questo prodotto viene fatto a valle dell'applicazione sia della soglia di score sia della soglia di IoU; l'uso delle due soglie riduce di molto il numero di anchors considerate (i.e. $num_anchors'' << num_anchors$) riducendo di conseguenza il costo dell'operazione.

Un altro aspetto da considerare è il fatto che queste operazioni, tanto di pre-quanto di post-elaborazione, potrebbero essere integrate, sfruttando Onnx e i suoi operatori, come parte del modello stesso. Sebbene questa soluzione sia stata presa in considerazione, si è ritenuto opportuno non fare un'integrazione diretta, soprattutto per la parte di post elaborazione. In primo luogo, la post elaborazione comprende le soglie per lo score e per la IoU, quindi introdurla nel modello avrebbe comportato l'introduzione di questi parametri come input; ciò avrebbe portato alla necessità di fare più inferenze in caso di analisi del risultato al variare delle soglie. In secondo luogo, l'output grezzo del modello è così più facilmente integrabile in una pipeline di inferenza

più grande in cui, ad esempio, l'output viene usato come input di altri modelli. Per concludere, questa operazione di integrazione, in un contesto di distribuzione che supporti modelli vari, dovrebbe essere fatta dall'utente e ciò potrebbe creare delle difficoltà d'uso. A fronte di queste considerazioni, quindi, il modello Yolo esportato in Onnx è stato mantenuto uguale, al netto del partizionamento che ne verrà poi fatto in fase successiva.

Buona parte della pipeline di pre e post elaborazione usata in questo lavoro è stata adattata partendo da [18]. La libreria *supervision* [19] è stata invece usata per rappresentare le bounding box e le maschere di segmentazione trovate.

2.2.3 OnnxRuntime

OnnxRuntime [20] è stato usato come framework di esecuzione dei modelli esportati in formato Onnx. Si tratta di una libreria multi-piattaforma: supporta una grande varietà di linguaggi, sia di alto che di basso livello, i sistemi operativi più importanti, come Windows e Linux, e, sebbene sviluppato da Microsoft, è disponibile sotto licenza MIT [21]. Supporta una grande varietà di hardware di esecuzione grazie all'implementazione di vari *ExecutionProvider* che lo rendono compatibile ed ottimizzato su hardware diversificati. Sebbene nato inizialmente per l'inferenza, in versioni recenti è stato aggiunto anche supporto alla fase di training.

Come accennato, uno degli elementi chiave di OnnxRuntime è l'*ExecutionProvider* (EP). OnnxRuntime utilizza un'architettura modulare basata su EP, che consente di sfruttare librerie di accelerazione hardware per eseguire in modo ottimale i modelli ONNX sulle diverse piattaforme hardware. Questa interfaccia offre flessibilità agli sviluppatori, permettendo di distribuire i modelli ONNX in ambienti cloud o edge e di ottimizzarne l'esecuzione sfruttando al meglio le capacità di calcolo della piattaforma.

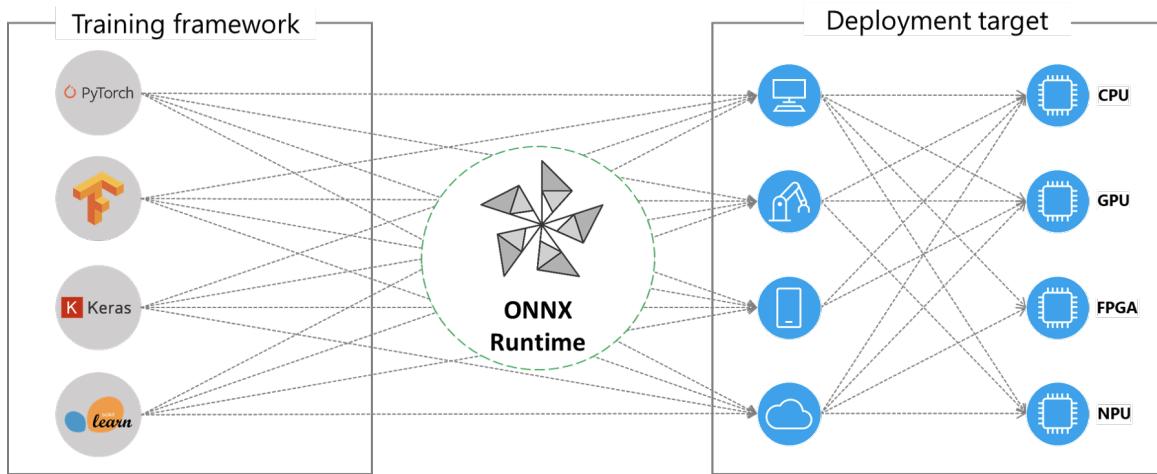


Figura 2.6: Esecuzione su Diverse Piattaforme

Le librerie EP preinstallate nell’ambiente di esecuzione si occupano poi dell’elaborazione e dell’esecuzione del sotto-grafo ONNX sull’hardware. Questa architettura consente di astrarre i dettagli delle librerie hardware-specifiche, ottimizzando così l’esecuzione del modello su diverse piattaforme, come CPU, GPU o NPU specializzate. L’interazione di OnnxRuntime è esemplificata in Figura 2.7:

1. Partendo da un modello Onnx, viene costruita una rappresentazione in-memory del modello;
2. Vengono eseguite delle ottimizzazioni indipendenti dal provider su cui il modello viene mandato in esecuzione;
3. Il grafo viene partizionato in sottografi;
4. I sottografi sono assegnati agli ExecutionProviders sulla base della capacità dell’hardware sottostante di eseguire i nodi di quel sottografo; il provider della CPU è il provider di default e, in caso di impossibilità di assegnare ad altri, quello su cui si fa fallback della computazione. Il partizionamento in sottografi non è da intendersi come un partizionamento fatto al fine di ottimizzare ulteriormente

il modello (ad esempio tramite parallelismo), bensì al fine di stabilire dove i nodi possano essere mandati in esecuzione in base alla capacità dell'hardware sottostante.

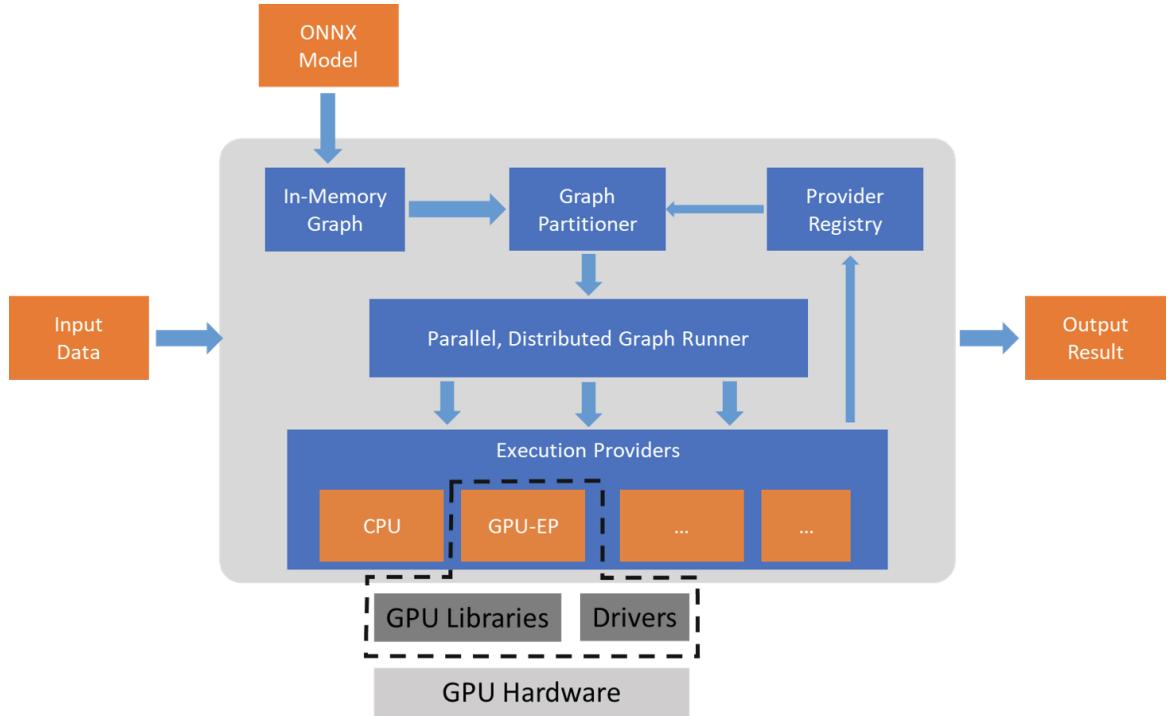


Figura 2.7: OnnxRuntime: Interazione con gli EP

Prima di distribuire il grafo, questo viene ottimizzato. OnnxRuntime offre diversi livelli di ottimizzazione [22], che si differenziano per l'intensità dell'ottimizzazione stessa. Stabilito il livello di ottimizzazione voluto, le ottimizzazioni di tutti i livelli precedenti sono eseguite in ordine. Alcune ottimizzazioni sono indipendenti dall'hardware su cui si manda in esecuzione il grafo, mentre altre non lo sono. I livelli principali sono:

1. Basic. Si cerca di rimuovere computazioni ridondanti ed eseguire delle fusioni che preservano la semantica del modello. Questa ottimizzazione è indipendente dal provider sottostante.

2. Extended. Vengono eseguite delle fusioni di livelli più complesse; sono disponibili solo per alcuni provider.
3. Layout. Si tratta di ottimizzazioni che cambiano il layout dei dati per ottenere delle performance migliori.

Un'altra funzionalità significativa offerta da OnnxRuntime è la possibilità di quantizzare il modello in modo relativamente semplice [23]. In OnnxRuntime ci sono due modi di rappresentare un modello quantizzato:

- Operator-Oriented. In questo caso l'operatore viene sostituito nel modello con una versione quantizzata
- Tensor-Oriented. Vengono inseriti nel modello, prima e dopo il livello target della quantizzazione, degli operatori di `QuantizeLinear` e `DequantizeLinear` che simulano il processo di quantizzazione.

Si noti che si sta parlando qui solo della rappresentazione della quantizzazione nel modello: in fase di esecuzione intervengono le ottimizzazioni del grafo che fonderanno ad esempio i livelli aggiuntivi di `QuantizeLinear` e `DequantizeLinear`. In Figura 2.8 è possibile vedere la differenza nella rappresentazione. Il processo di quantizzazione si compone di due fasi:

1. Pre-Processing. Vengono eseguite alcune operazioni di ottimizzazione del modello e di shape inference per ottimizzare la fase successiva.
2. Quantizzazione. Nel nostro caso ci si è focalizzati sulla quantizzazione statica, sebbene la libreria offra anche supporto alla quantizzazione dinamica. In questo caso è necessario fornire un dataset di calibrazione che permetta di trovare i

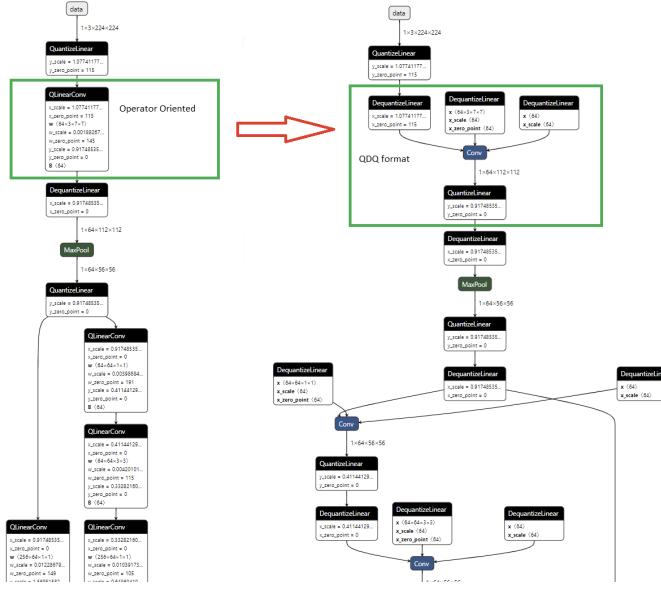


Figura 2.8: OnnxRuntime: QOperator vs QDQ

parametri di quantizzazione ottimali per il modello; la quantizzazione statica usata di default è stata la quantizzazione MinMax.

L’interfaccia di quantizzazione statica di OnnxRuntime è estremamente ricca e permette, ad esempio, di:

- Specificare i tipi di livelli del modello che si vogliono quantizzare, quantizzando tutti e soli i nodi di quei tipi;
- Specificare i nomi dei livelli da quantizzare, quantizzando solo quei livelli;
- Specificare i nomi dei livelli da escludere dal processo di quantizzazione.

In Figura 2.9 viene riportato un esempio del guadagno fornito dalla quantizzazione in termini di velocità di inferenza; il test è stato eseguito con il seguente setup:

- Macchina GCP c4-standard-4;
- 200 run;

- Modello Yolo11x-seg;
- Quantizzazione con QOperator;
- Solo CPU;
- Tempo in millisecondi.

Come si può vedere abbiamo uno speed-up della velocità di inferenza di circa 3.75 volte.

```
(myenv) google@quantization:~$ python3 YoloTestQuant_1.py
Quant Time >> 290.33177825499996
Not Quant Time >> 1092.61298542
(myenv) google@quantization:~$
```

Figura 2.9: OnnxRuntime: Speed-Up di Quantizzazione

Per riassumere, i motivi che hanno portato alla scelta di OnnxRuntime come framework di esecuzione sono stati:

- Semplicità di supporto e di utilizzo.
- Support multi-linguaggio e multi-piattaforma. La possibilità di eseguire su hardware diversi è essenziale in un contesto di edge-cloud continuum, in cui si può spaziare da device fortemente limitati a server molto potenti e dotati di GPU. In questo ricade anche la possibilità di avere una libreria multilinguaggio che sia supportata sia dai linguaggi di basso livello (ad esempio, per un contesto IoT) fino ai linguaggi di alto livello (come Java o Kotlin in ambito Android).
- Possibilità di quantizzare in modo semplice e flessibile e, nello specifico, la possibilità di quantizzare per singoli livelli o per tipo di livello.

2.2.4 Altri strumenti

Altri strumenti usati e degni di nota sono:

- gRPC [24]. Considerando che lo scopo del progetto è quello di essere adatto ad un'esecuzione in ambiente di edge-cloud continuum, gRPC è stato usato per garantire l'interoperabilità nell'uso di linguaggi diversi in ambiente di esecuzione diversi. Altri vantaggio offerto da gRPC è la compressione dell'header dei messaggi inviati. Il contro principale nell'uso gRPC sta nella limitata dimensione dei messaggi, soprattutto in un contesto come questo in cui è necessario il trasferimento di modelli di ML di dimensioni varie: in questi casi è necessaria la divisione in chunks di questi messaggi e la ricostruzione dell'intero lato ricevente.
- Networkx [25]. NetworkX è una libreria Python che permette di costruire ed eseguire algoritmi su grafi. In questo contesto è stato usato per costruire delle astrazioni del modello considerato e della rete di dispositivi su cui questo modello deve essere mandato in esecuzione. A rivelarsi particolarmente utile in NetworkX è stata la possibilità di poter associare ad ogni nodo e ad ogni arco delle informazioni in modo semplice e veloce: vedremo ad esempio come ad un arco di rete sono associate informazioni relative a banda e latenza, oppure come ad un nodo del modello sono associate informazioni relative alla dimensione dei tensori di output.
- PuLP [26]. PuLP è una libreria Python per la costruzione di problemi di ottimizzazione lineare ad obiettivo unico, astraendoli dal risolutore usato al livello inferiore. Consente di costruire problemi continui, interi o misti.

- CPLEX [27]. Usato per la risoluzione del problema di ottimizzazione che vedremo nelle sezioni successive.

Capitolo 3

Architettura del Sistema

In questo capitolo verranno analizzate le diverse componenti che costituiscono il sistema, il loro ruolo al suo interno e il modo in cui interagiscono.

Si partirà con una panoramica veloce su tutte le componenti, per poi passare ad uno studio più dettagliato di tre componenti: *ModelProfiler*, *ExecProfiler* e *ServerMonitor*. Le informazioni raccolte da queste componenti saranno quelle che rientrano poi nella modellazione del problema nei capitoli successivi.

Si concluderà con un’analisi delle interazioni tra le componenti del sistema descrivendo le quattro situazioni di *Caricamento del Modello*, *Generazione del Piano*, *Deployment del Piano* ed *Inferenza*.

3.1 Componenti del Sistema

Nota: Figura maiuscolo, sezione minuscolo ?

In Figura 3.1 sono riportate le componenti principali del sistema insieme alle loro interazioni fondamentali, approfondite ulteriormente in sezione 3.2.

Di seguito diamo una breve descrizione delle componenti del sistema; per le componenti che richiedono un approfondimento maggiore saranno riservate delle sottosezioni

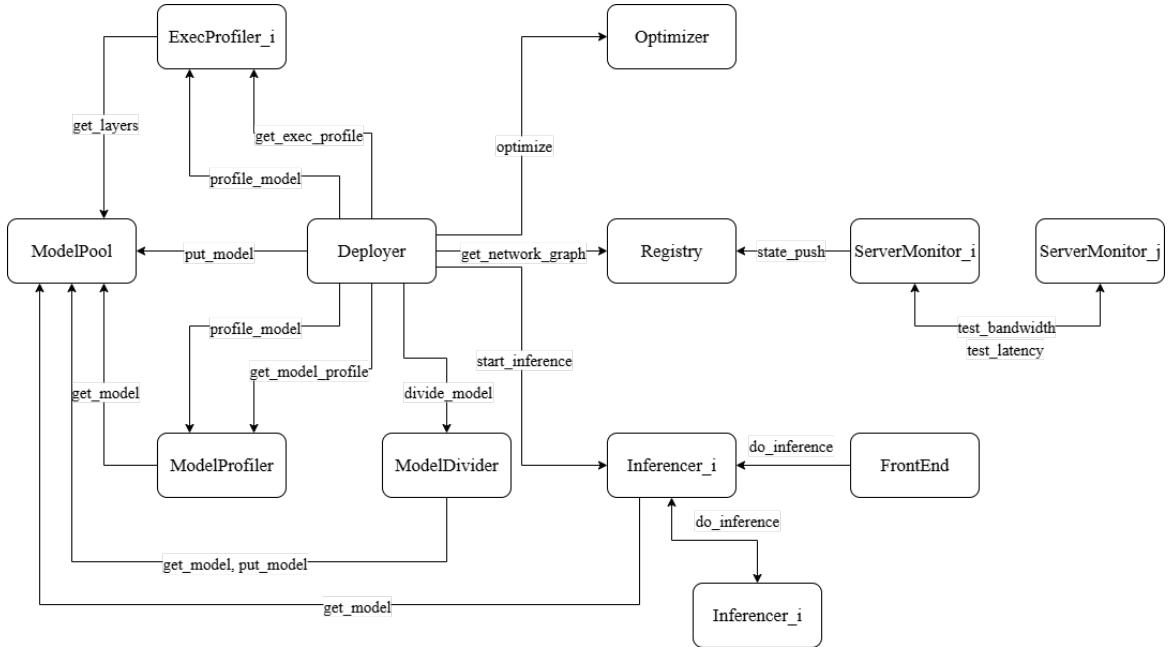


Figura 3.1: Componenti del Sistema

apposite:

- *Deployer*. Si tratta della componente di front-end a cui vengono fatte le richieste di caricamento del modello, generazione e deployment del piano. Funge da controller per la fase preliminare all'inferenza.
- *ModelPool*. Si tratta della componente atta a gestire il salvataggio e il recupero dei modelli usati all'interno del sistema. Ad esso si rivolgeranno le altre componenti nel momento in cui hanno bisogno di fare il salvataggio o il recupero di un modello o di un sotto modello. Dato un modello completo, questo è univocamente identificato dal suo nome; dato un sotto modello questo è identificato univocamente dal nome del modello originale, dall'identificativo del server su cui il sotto modello deve essere eseguito e da un valore numerico usato per distinguere il sotto modello nel server di destinazione.

- *ModelProfiler*. Si tratta della componente con il compito di costruire un’astrazione del modello, estraendo da esso le informazioni fondamentali da usare poi in fase di ottimizzazione. Vedere sottosezione 3.1.1 per una trattazione più approfondita sul profiling del modello.
- *ExecProfiler*. Si tratta di una componente in esecuzione su ogni server atto a fare inferenza; la componente si occupa di fare il profiling del tempo di esecuzione del modello su quello specifico server. Vedere sottosezione 3.1.2 per una trattazione più approfondita sul profiling dell’esecuzione.
- *ServerMonitor*. Si tratta di una componente in esecuzione su ogni server atto a fare inferenza; la componente si occupa di valutare lo stato del server su cui è in esecuzione, nonché informazioni relative all’interazione con altri server (e.g. banda di trasmissione, latenza, ecc.). Vedere sottosezione 3.1.3 per una trattazione più approfondita sul monitoraggio del server.
- *Registry*. Si tratta della componente del sistema che registra i server disponibili nel sistema, nonché informazioni relative al loro stato al fine di costruire un grafo di rete aggiornato con lo stato corrente. Quando un server viene attivato, invia una richiesta di registrazione al Registry, il quale assegna un identificativo univoco al server.
- *Optimizer*. Si tratta della componente del sistema atta a definire il problema di ottimizzazione e a risolverlo. Il risultato dalla fase di ottimizzazione è un piano di deployment che viene poi usato nelle fasi successive. Vedere Capitolo 4 per la definizione del problema e per la post elaborazione della soluzione. In generale il piano descrive la struttura di un grafo di sotto modelli, indicando dove il modello originale debba essere tagliato per ottenere questi sotto modelli

e dove questi debbano essere mandati in esecuzione; descrive anche quali livelli debbano essere quantizzati.

- *ModelDivider*. Si tratta della componente con il compito di quantizzare e dividere il modello sulla base del piano per cui è stata fatta la richiesta di deployment. Vedere sezione 5.4 per una trattazione più approfondita sulla divisione del modello.
- *FrontEnd*. Si tratta della componente del sistema in esecuzione sul device che fa partire l'inferenza. La componente, quando ricevuto l'input del modello, si occupa di far partire l'inferenza seguendo il piano prodotto; quando ricevuto l'output del modello, invece, si occupa di restituirlo al chiamante. Vedere sottosezione 3.2.4 per una descrizione più dettagliata del ruolo del FrontEnd nel sistema.
- *Inferencer*. Si tratta di una componente in esecuzione su ogni server atto a fare inferenza. La componente si occupa della fase di inferenza di un sotto modello dopo che questo è stato estratto dal modello originario e assegnato al server. Vedere sottosezione 3.2.4 per una descrizione più dettagliata del ruolo dell'Inferencer nel sistema.

3.1.1 Model Profiler

Il *ModelProfiler* è la componente del sistema che si occupa del profiling di una DNN. Per profiling della DNN si intende l'estrazione delle informazioni salienti riguardanti il modello e la costruzione di un'astrazione contenente queste informazioni.

Le informazioni che vengono estratte dal modello sono le seguenti:

- Per i Layer:

- Numero di FLOPS;
 - Dimensione dei pesi;
 - Dimensione dell'output;
 - Se si tratta di un livello quantizzabile, in base ad una certa definizione di quantizzabile che vedremo a breve.
- Per gli archi tra Layer:
 - Dimensione totale dei tensori passati attraverso l'arco; *o usi i punti e le maiuscole, oppure ; e le minuscole a inizio riga*
 - Nomi dei tensori passati attraverso l'arco.
 - Per i tensori che sono trasmessi tra i livelli nel modello:
 - Il nome del livello sorgente del tensore;
 - La dimensione del tensore.

Anche se queste ultime informazioni sono ridondanti perché ottenibili combinando le informazioni dei livelli e quelle degli archi, sono comunque mantenute a parte nel profiling per semplificare l'implementazioni delle fasi successive.

Oltre alle informazioni sopra riportate, vengono anche profilate le informazioni relative al rumore di quantizzazione, come descritto in sottosottosezione 3.1.1.1. Come vedremo nelle sottosezioni successive, dati i livelli del modello, gli x livelli con il numero maggiore di FLOPS vengono marcati come *quantizable*.

Sposterei nel capitolo successivo fino a ExecProfiler
3.1.1.1 Il Problema del Rumore di Quantizzazione

Per rumore di quantizzazione si intende l'effetto che la quantizzazione di uno o più livelli può avere sull'output del modello. L'analisi di questo rumore è fondamentale per stabilire quali livelli quantizzare: l'uso indiscriminato della quantizzazione potrebbe

essere deleterio per l'accuratezza del modello e c'è dunque bisogno di porre un limite al suo uso. Un esempio di questo effetto negativo è riportato in [28, 29, 30]: in questo caso, la quantizzazione dei livelli di `Concat` più "bassi" del modello (i.e. vicino all'output del modello) porta il modello *Yolo8* a dare in output delle probabilità di appartenenza alle classi nulle. Il motivo di questo è stato introdotto in sezione 1.2: questo problema è da attribuire al fatto che nella quantizzazione *Min-Max* (usata di default da OnnxRuntime) vengono stabiliti i parametri di quantizzazione sulla base del range dei tensori intermedi; i range dei tensori in input/output a queste `Concat` sono talmente ampi da portare a zero tutte le classificazioni.

Un modo per valutare il rumore di quantizzazione è valutare la differenza tra i risultati del modello quantizzato e quelli del modello originale, considerando quest'ultimo come baseline. Supponiamo di avere n input di calibrazione per valutare il rumore; indicato con o il tensore di output del modello originale e con o_q il tensore di output del modello quantizzato, possiamo definire il rumore nel seguente modo:

$$\rho = \frac{1}{n} \sum_{i=1}^n \text{mean}(|o_i - o_{q,i}|) \quad (3.1.1)$$

Questa misura è indipendente dal tipo di task e risulta pertanto più flessibile e utilizzabile con modelli diversi.

In OnnxRuntime abbiamo la possibilità di scegliere, per ogni livello, se attivare o meno la quantizzazione. Ciò porta quindi a un numero di combinazioni di quantizzazioni che cresce esponenzialmente con il numero di nodi: per un modello ad x nodi, ci sono 2^x possibili combinazioni di quantizzazione; questo è naturalmente uno spazio di combinazioni estremamente grande da analizzare in maniera esaustiva.

Per ridurre lo spazio di combinazioni da analizzare si può quindi prendere un restringimento dei livelli; vi sono infatti alcuni livelli per i quali la quantizzazione può

apportare un beneficio maggiore; questi livelli sono:

- Livelli che hanno un carico computazionale elevato, individuabili sulla base del numero di *FLOPS*;
- Livelli che hanno un output di dimensione elevata. Quantizzando questi livelli si riduce la dimensione dell'output e quindi l'eventuale tempo di trasmissione di questi dati tra due server.

Nel nostro lavoro sono stati considerati soltanto i livelli a carico computazionale maggiore: la scelta è stata guidata principalmente da motivi pratici in quanto molto spesso in modelli Yolo ci sono livelli vicini che hanno output di stesse dimensioni; di conseguenza, quantizzare in base alla dimensione dell'output non avrebbe apportato un grande beneficio, in quanto è probabile che livelli vicini vengano partizionati in modo da stare sullo stesso server.

Indichiamo quindi con V_Q l'insieme di livelli quantizzabili del modello. In termini di profiling, questi livelli saranno marcati con un attributo **quantizable** impostato a `True` che permette di riconoscerli. A questo punto, il problema si riduce a trovare il rumore di quantizzazione per le rimanenti $2^{|V_Q|}$ combinazioni di quantizzazioni. Notiamo che, anche se il problema si è ridotto in dimensione, il numero di possibilità rimane comunque piuttosto alto:

- Con $|V_Q| = 10$, abbiamo 1024 combinazioni;
- Con $|V_Q| = 15$, abbiamo più di 32000 combinazioni;
- Con $|V_Q| = 20$, abbiamo più di 10^6 combinazioni.

Supposto di conoscere per ognuna delle combinazioni il rumore che essa genera sull'output del modello, la sua modellazione in un problema di ottimizzazione non è affatto banale. Indicati con:

- η_C il rumore indotto dalla combinazione C ;
- $q_i \in \{0, 1\}$ la variabile binaria che ci dice se un livello (quantizzabile) viene quantizzato o meno;
- $\mathcal{C}(V_Q)$ l'insieme degli insiemi combinazione di V_Q ;

Il rumore finale espresso in funzione delle variabili q_i sarà dato da:

$$\eta = \sum_{C \in \mathcal{C}(V_Q)} \left(\eta_C \cdot \prod_{i \in C} q_i \cdot \prod_{i \notin C} (1 - q_i) \right) \quad (3.1.2)$$

Ognuno di questi prodotti, per poter essere espresso in forma lineare richiederà una variabile che lo rappresenti e che sia soggetta a vincoli che descrivono il prodotto. Nello specifico, avremo bisogno di $\sum_{k=1}^{|V_Q|} \binom{|V_Q|}{k}$ variabili prodotto $q_C \in \{0, 1\}$, ognuna soggetta ai seguenti vincoli:

$$\begin{cases} q_C \leq q_i & \forall i \in C \\ q_C \leq (1 - q_i) & \forall i \notin C \\ q_C \geq \sum_{i \in C} q_i + \sum_{i \notin C} (1 - q_i) - (|V_Q| - 1) \end{cases} \quad (3.1.3)$$

Una modellazione di questo tipo porta chiaramente all'esplosione del problema anche per un numero piccolo di livelli quantizzabili.

In conclusione, i problemi relativi alla modellazione del rumore di quantizzazione sono i seguenti:

- Alto numero di combinazioni di quantizzazione;
- Esplosione del problema in caso di modellazione esatta anche per un numero piccolo di livelli per cui è ammessa la quantizzazione;

- Tempi di valutazione del rumore abbastanza alti: per ogni combinazione sono necessari i seguenti passi:
 1. Costruzione del modello quantizzato;
 2. Inferenza su un insieme di input per valutare il rumore indotto;
 3. Confronto con il modello non quantizzato.

3.1.1.2 Modello di Regressione

Alla luce di quanto detto, c'è bisogno di un modo di rappresentare il rumore di quantizzazione senza un'esplosione del problema.

La soluzione adottata nel contesto di questo studio è basata sull'uso di un regressore polinomiale addestrato sui dati relativi al rumore. La scelta dell'uso del regressore polinomiale è stata dettata da:

- Possibilità di limitare l'esplosione del problema attraverso il grado massimo del polinomio di regressione;
- Semplicità di integrazione all'interno di un problema di ottimizzazione;
- Buoni risultati in fase di train e test sui dati raccolti.

Il dataset per addestrare e testare il regressore polinomiale sarà formato da feature binarie, ognuna delle quali indica se il livello corrispondente è quantizzato o meno, e target continuo a rappresentare il rumore corrispondente a quella combinazione. Al fine di addestrare il modello, saranno necessari i seguenti passi:

1. Costruire un certo numero di combinazioni di quantizzazione per i $|V_Q|$ livelli quantizzabili;
2. Per ogni combinazione, costruire il modello quantizzato corrispondente;

3. Valutare il rumore di quantizzazione del modello così costruito.

Si noti che, a differenza del caso precedente, non è necessario costruire $2^{|V_Q|}$ punti del dataset: trattandosi di un regressore è possibile costruire un sottoinsieme delle combinazioni e addestrare il modello su quelle; chiaramente tanti più saranno i punti del dataset, tanto maggiore sarà l'accuratezza del modello così costruito.

Supponiamo di addestrare un regressore polinomiale di grado d . Siano:

- $q_i \in \{0, 1\}$ la variabile binaria che ci dice se un livello (quantizzabile) viene quantizzato;
- ρ_C il coefficiente del regressore associato al prodotto delle variabili binarie q_i dei livelli i nella combinazione C ;
- $\mathcal{C}_{\leq d}(V_Q)$ l'insieme degli insiemi combinazione di V_Q di cardinalità minore o uguale a d

Il rumore finale approssimato con il regressore in funzione delle variabili q_i sarà dato da:

$$\eta = \sum_{C \in \mathcal{C}_{\leq d}(V_Q)} \left(\rho_C \cdot \prod_{i \in C} q_i \right) \quad (3.1.4)$$

Ognuno di questi prodotti, per poter essere espresso in forma lineare, richiederà una variabile che lo rappresenti e che sia soggetta a determinati vincoli. Nello specifico, avremo bisogno di $\sum_{k=1}^d \binom{|V_Q|}{k}$ variabili prodotto $q_C \in \{0, 1\}$, ognuna soggetta ai seguenti vincoli:

$$\begin{cases} q_C \leq q_i & \forall i \in C \\ q_C \geq \sum_{i \in C} q_i - (|C| - 1) \end{cases} \quad (3.1.5)$$

Si noti i seguenti due aspetti:

- Mentre nella formulazione in Equazione 3.1.2 soltanto uno dei termini η_C è attivo per una certa combinazione di quantizzazione, nella formulazione in Equazione 3.1.4 possono esserci più termini ρ_C attivi contemporaneamente; sarà la combinazione di questi termini a dare il risultato finale del rumore.
- Usando una formulazione di questo tipo, il numero di variabili è molto minore rispetto alla formulazione iniziale: basti considerare che, fissato $d = 3$, per $|V_Q| = 10$ si passa da 1023 variabili a 175 variabili (quasi 6 volte in meno), mentre per $|V_Q| = 15$ si passa da 32767 variabili a 575 (quasi 57 volte in meno).
- Per ogni variabile q_C anche il numero di vincoli diminuisce; per ogni q_C passiamo da $|V_C| + 1$ vincoli a $|C| + 1$ vincoli, dove $|C| \leq d$.

3.1.1.3 Esempio di Uso del Modello

Il test che descriviamo in seguito è stato eseguito con i seguenti parametri:

- Modello: Yolo11n-det (modello nano di detection);
- Numero di livelli quantizzabili: 10;
- Dimensione del Train Set per il regressore: 500 elementi;
- Dimensione del Test Set per il regressore: 50 elementi;
- Dimensione insieme per calibrazione del modello: 100 elementi;
- Dimensione insieme per valutazione del rumore: 10 elementi.

In Figura 3.2 sono riportati i nomi dei livelli a maggior numero di FLOPS per il modello Yolo11n-det, insieme ai FLOPS corrispondenti.

```
/model.3/conv/Conv 472678400.0
/model.23/cv2.0/cv2.0.0/conv/Conv 472678400.0
/model.23/cv2.0/cv2.0.1/conv/Conv 472678400.0
/model.5/conv/Conv 472268800.0
/model.1/conv/Conv 237568000.0
/model.23/cv2.1/cv2.1.0/conv/Conv 236134400.0
/model.7/conv/Conv 236134400.0
/model.16/cv1/conv/Conv 210534400.0
/model.2/cv2/conv/Conv 160563200.0
/model.4/cv2/conv/Conv 158924800.0
```

Figura 3.2: Livelli a Massimo numero di FLOPS per Yolo11n

In Figura 3.3 sono riportate le curve di train e di test per il regressore al variare del grado del polinomio; sono stati considerati gradi tra 1 e 5. Per comodità, i valori di R^2 -score al variare del grado sono riportati in Tabella 3.1.

Come possiamo vedere dalla Tabella 3.1:

- Il fit del modello migliora all'aumentare del grado. Questo è prevedibile considerando che aumentare il grado implica considerare un maggior numero di interazioni tra i rumori indotti dalla quantizzazione dei diversi livelli;
- Il test score del modello aumenta fino al massimo raggiunto al grado 4, oltre il quale comincia a manifestarsi overfitting: ciò mostra come le interazioni da considerare per valutare il rumore sia limitato.
- Il valore di test score per grado del polinomio pari a 3 si mostra comunque alto quindi, al fine di ridurre il numero di variabili e vincoli necessari alla rappresentazione del polinomio nel problema di ottimizzazione, possiamo considerare questo come grado massimo (il grado $d = 3$ si è rivelato valido ai fini del test score anche al variare dei parametri sopra citati).

Un'altra cosa interessante da notare in Figura 3.3 è che il rumore indotto sull'output del modello quando tutti i livelli possibili sono quantizzati non è massimo:

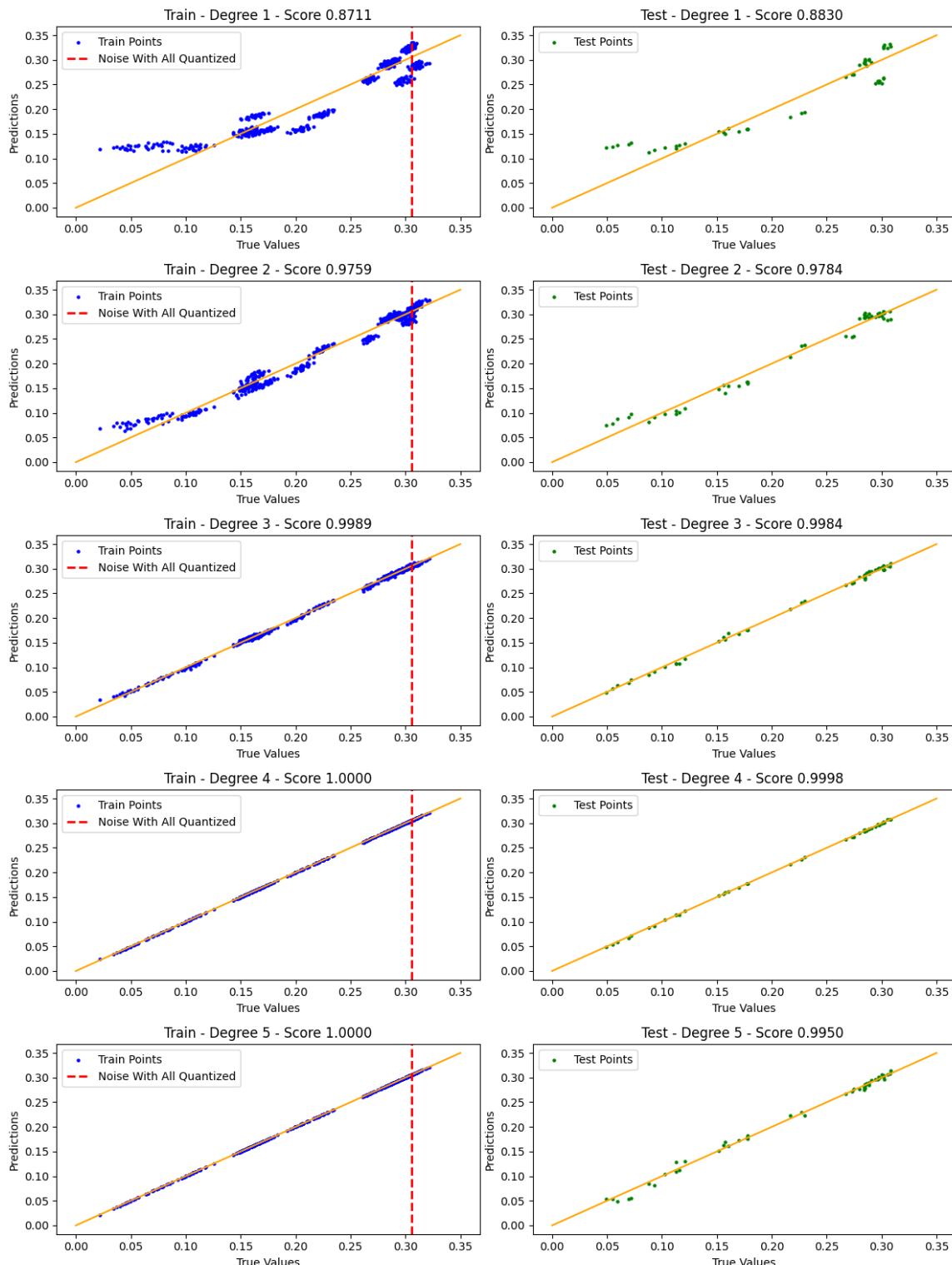


Figura 3.3: Curve di Train e di Test per il Regressore al variare del Grado

	Grado 1	Grado 2	Grado 3	Grado 4	Grado 5
Train Score	0.8711	0.9759	0.9989	1.0	1.0
Test Score	0.8830	0.9784	0.9984	0.9998	0.9950

Tabella 3.1: R^2 -scores al variare del grado

ci sono delle combinazioni di quantizzazioni dei livelli che danno valori del rumore più alti. In generale, possiamo dire che gli effetti della quantizzazione dei livelli non si combinano semplicemente in maniera lineare, ma in modo non lineare con effetti additivi e sottrattivi sul rumore complessivo.

3.1.1.4 Misura del Rumore

Un possibile problema da considerare è legato alla validità di una misura del rumore definita come in Equazione 3.1.1: in particolare, si potrebbero usare altre misure per valutare la differenza tra i due tensori di output, come ad esempio la norma infinito; di fatto, la norma infinito permetterebbe di individuare la differenza massima (in valore assoluto) tra i due tensori, dando quindi una misura della differenza diversa, ma altrettanto sensata. Ci sono alcuni aspetti da considerare in questo senso:

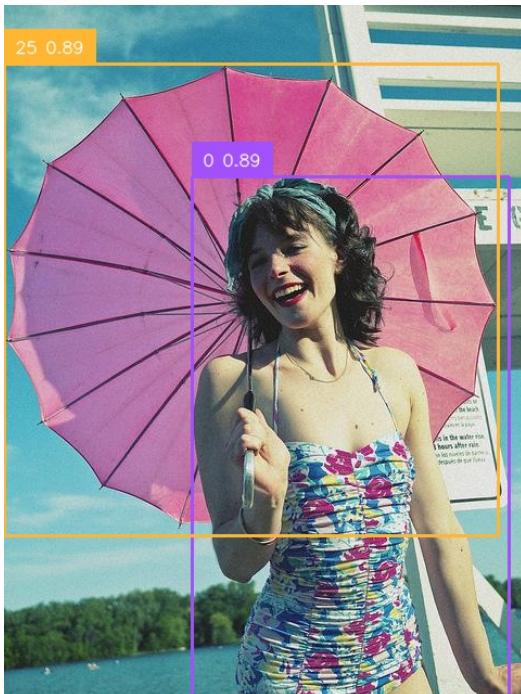
- Nel caso di modelli Yolo di segmentazione o detection, non tutti gli output del modello contribuiscono al risultato finale, ma alcuni vengono scartati in fase di post elaborazione: usare quindi una metrica che valuta la differenza in termine assoluto, potrebbe far dipendere la differenza tra gli output da un elemento dei tensori che viene poi scartato in fase successiva;
- L'uso di una norma infinito creerebbe una maggiore instabilità, rendendo la valutazione della differenza più complessa.

A supporto di questi due punti, si consideri l'esempio in Figura 3.4. Come si può vedere in Figura 3.4a la norma infinito della differenza ha un valore piuttosto alto;

osservando le coordinate dell'elemento a valore maggiore della differenza, si può vedere come si tratti della larghezza di una bounding box. Tuttavia, osservando le differenze tra Figura 3.4b e Figura 3.4c, si può notare come considerare un valore così alto sia ingiustificato: di fatto, le differenze tra i due output finali sono minime.

```
customuser@470c1e476f50:/workspaces/Dissertation/Analysis/Quantization/Output$ python3 Test_Output.py
Inf Norm >> 254.42572
Inf Norm Elem Coord >> (0, 3, 8101)
Mean Diff >> 0.23271148
```

(a) Norma Infinito della Differenza



(b) Con Modello Standard



(c) Con Modello Quantizzato

Figura 3.4: Analisi delle Differenze tra Output

In conclusione, l'uso della media permette di attenuare l'effetto di predizioni che verrebbero poi scartate in fase di post-elaborazione.

Ovviamente, la post elaborazione potrebbe essere introdotta a valle della predizione anche in fase di profiling, ma questo renderebbe la profilazione del modello eccessivamente dipendente dal modello specifico.

3.1.2 ExecProfiler

L'ExecProfiler è volto a valutare, per ogni livello del modello, il tempo di esecuzione del livello su una specifica macchina: ogni server ha quindi il suo ExecProfiler in esecuzione che profila il modello in questione.

Poiché in fase di ottimizzazione la decisione sul placement verrà presa livello per livello, è necessario avere per ogni macchina una stima del tempo di esecuzione di ogni livello ~~su quella specifica macchina~~. Inoltre, poiché in fase di ottimizzazione verrà stabilito se quantizzare un livello o meno e l'uso della quantizzazione cambia il tempo di esecuzione del livello, la valutazione dovrà essere fatta sia per livello quantizzato che per livello non quantizzato.

Per la valutazione del tempo di esecuzione livello per livello abbiamo due alternative:

- Recuperare il modello intero, fare l'inferenza sul modello completo e valutare, usando strumenti di profiling, il tempo di esecuzione per ogni livello. Questa soluzione, sebbene semplice, presenta una criticità non trascurabile, cioè il fatto che a basso livello possono essere eseguite delle fusioni tra livelli dipendenti dal tipo di hardware su cui il modello viene mandato in esecuzione: ciò rende estremamente difficile ricollegare il tempo di esecuzione del livello fuso a quello dei singoli livelli originali.
- Tagliare il modello livello per livello ed eseguire il sotto modello contenente il singolo livello per il quale si sta valutando il tempo di inferenza. In questo caso il problema che si pone riguarda la presenza di overhead aggiuntivo che entra nella stima del tempo di esecuzione di quel modello.

Nel nostro caso, è stata scelta la seconda alternativa.

Il secondo aspetto da considerare è quello della quantizzazione: nel nostro caso, il modello viene quantizzato interamente per poi essere tagliato in modo opportuno in corrispondenza dei livelli quantizzati.

La responsabilità del taglio e della quantizzazione eseguita in fase di profilazione viene affidata al ModelPool, che quindi permette di recuperare i singoli livelli di un modello tanto nella loro variante quantizzata che in quella non quantizzata. Si noti che in questo caso non siamo interessati ~~solo ad avere~~ ~~ad una quantizzazione volta all'accuratezza del modello finale bensì ad una versione quantizzata del modello da cui poter estrarre i~~ livelli quantizzati: di conseguenza, non abbiamo bisogno di un dataset di calibrazione che sia esteso o verosimile.

3.1.3 ServerMonitor

Il ServerMonitor è la componente in esecuzione sul singolo server che si occupa di valutare lo stato corrente del server, le informazioni ad esso relative e lo stato delle connessioni verso gli altri server.

Le informazioni raccolte relative al server su cui il monitor è in esecuzione sono:

- Stato della memoria;
- Statistiche energetiche del server e, in particolare:
 - Potenza media di trasmissione;
 - Potenza media di calcolo.

Le informazioni che sono raccolte relative alla connessione verso un altro server sono:

- Il throughput di trasmissione. Questo viene valutato attraverso l'invio di una certa quantità di dati verso ogni altro server del sistema, dividendo quindi la quantità di dati per il tempo totale di invio.
- Il round trip time di trasmissione.

Queste informazioni vengono poi comunicate al registry, che le usa per costruire un grafo di rete contenente le informazioni dei server attivi e le connessioni esistenti tra di loro.

3.2 Fasi del Sistema

3.2.1 Caricamento del Modello

In Figura 3.5 sono descritte le interazioni in fase di caricamento del modello. Il supposto client invia il modello e altre informazioni ad esso correlate alla componente di Deployer. Il modello viene quindi salvato sul ModelPool, che si occuperà di gestire il recupero del modello quando necessario alle altre componenti. Una volta caricato, vengono fatte partire le operazioni di profiling del modello; il modello è sottoposto a due tipi di profiling:

- Il profiling fatto dal ModelProfiler, descritto in sottosezione 3.1.1.
- Il profiling fatto dall'ExecProfiler, descritto in sottosezione 3.1.2. Questo profiling, come spiegato in precedenza, viene eseguito in modo indipendente su tutti quanti i server.

Questa fase iniziale rappresenta forse la fase più lunga del sistema; al netto del trasferimento del modello, le fasi più lunghe sono rappresentate proprio dal profiling del modello e dal profiling dell'esecuzione. Nel primo caso, infatti, viene incluso il

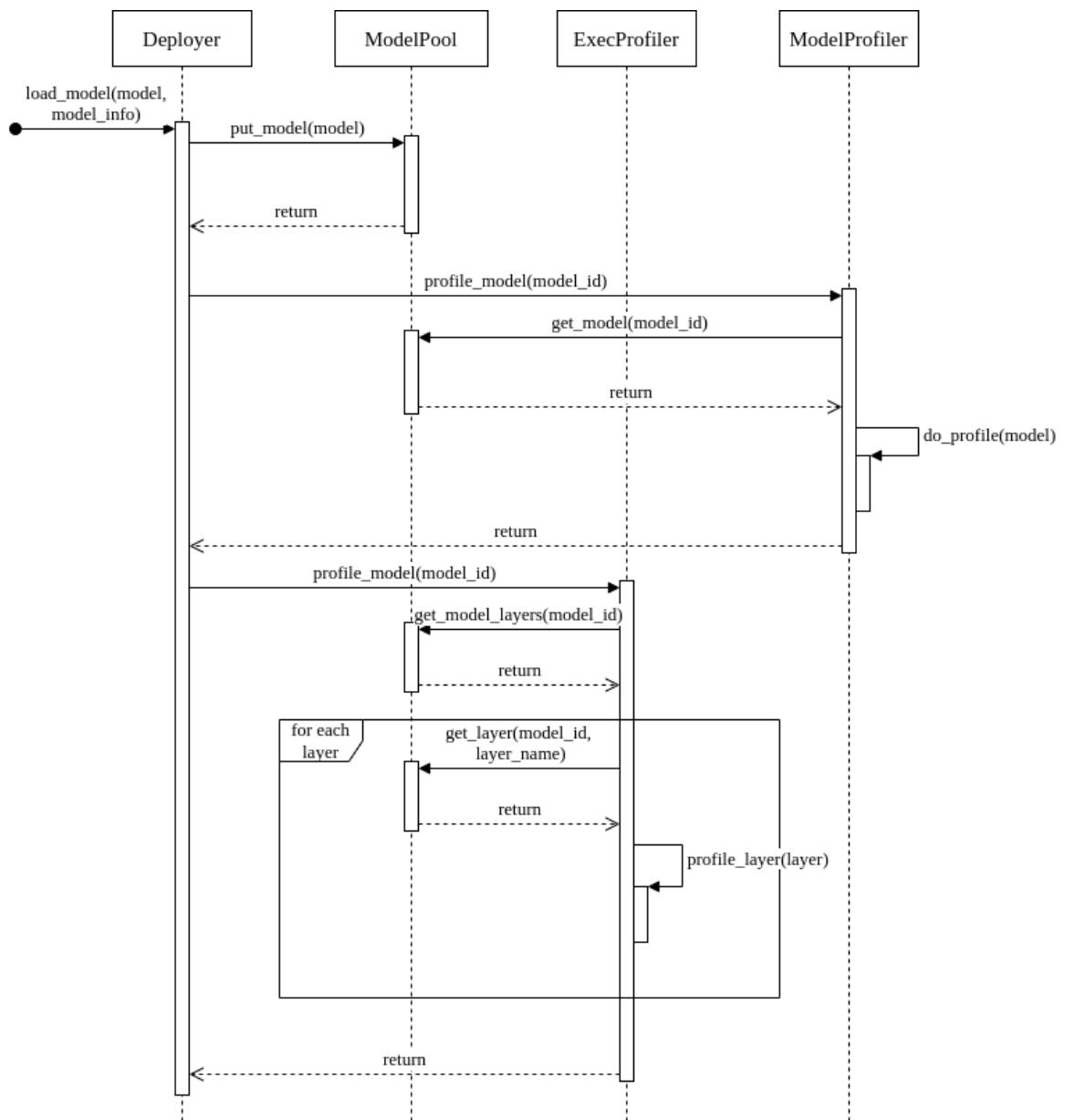


Figura 3.5: Sequence Diagram: Caricamento del Modello

profiling del rumore di quantizzazione che, come spiegato, non è banale; nel secondo caso, invece, la velocità del profiling dipenderà tendenzialmente della capacità di calcolo dell'hardware su cui si sta facendo il profiling: un dispositivo dotato di GPU sarà sicuramente più veloce nel profiling rispetto ad uno che non la ha.

3.2.2 Generazione del Piano

In Figura 3.8 sono descritte le interazioni in fase di generazione del piano. Anche in questo caso, la richiesta di generazione del piano viene mandata alla componente di Deployer insieme ai parametri di ottimizzazione. La componente si occupa di raccogliere le informazioni necessarie alla fase di ottimizzazione:

- Dal ModelProfiler, viene ottenuto il profilo del modello;
- Dal Registry, viene recuperato il profilo della rete, con le relative informazioni relative allo stato corrente della rete dei server (banda tra dispositivi, latenza ecc.);
- Dall'ExecProfiler, viene recuperato, per ogni server attivo nel sistema, il suo profilo di esecuzione del modello per il quale si è chiesto di generare il piano ottimizzato;

Raccolte tutte le informazioni necessarie, la richiesta di ottimizzazione viene inviata all'Optimizer il quale procede in tre passi successivi:

1. Viene risolto il problema di ottimizzazione (vedere Capitolo 4);
2. Viene post elaborata la soluzione del problema (vedere sezione 4.10);
3. Viene costruito il piano effettivo da attuare per la fase di inferenza (vedere sezione 5.3).

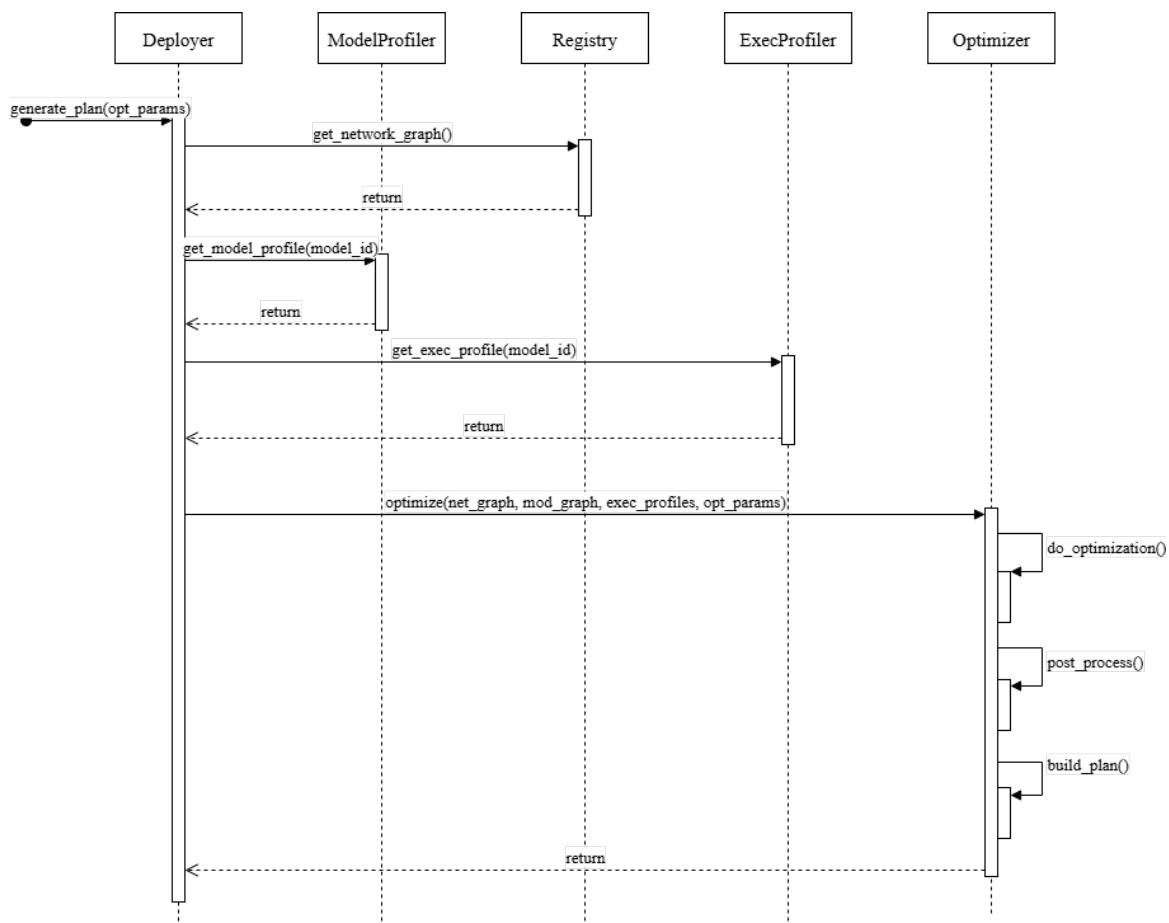


Figura 3.6: Sequence Diagram: Generazione del Piano

3.2.3 Deployment del Piano

In Figura 3.7 sono mostrate le interazioni in fase di deployment del piano, ovvero quando viene richiesta l'attivazione di un piano precedentemente generato. La divisione tra generazione del piano e sua messa in atto rende più flessibile l'uso del sistema: in questo modo l'utente ha la possibilità di generare diversi piani al variare al variare di alcuni parametri, analizzare la soluzione e valutare quella che è più adatta alle sue esigenze in un certo momento.

Partendo dal Deployer, la richiesta viene inviata al ModelDivider che, recuperato il modello dal ModelPool, si occupa di due aspetti:

- Della quantizzazione dei livelli del modello, se in fase di ottimizzazione è stato scelto di quantizzare il modello;
- Della divisione del modello in parti secondo quanto stabilito dal piano: vengono quindi estratti dei sotto modelli dal modello originale.

Indichiamo questi sottomodelli (che sono a loro volta dei modelli) con il nome di *componenti*; una volta prodotta, la componente viene salvata anch'essa a livello del ModelPool. Completate le fasi di quantizzazione e di taglio del modello, il Deployer invia a ciascun Inferencer la richiesta di attivazione del piano. Ricevuta la richiesta, l'Inferencer, dopo aver consultato il piano, recupera le componenti a lui assegnate e avvia la gestione dell'inferenza per questi sottomodelli.

Si noti che tutti gli Inferencer conoscono il piano globale per un certo modello: questo permette a ciascun Inferencer di sapere, dato un certo output di una componente, quali sono le componenti successive per ogni componente che ha in gestione.

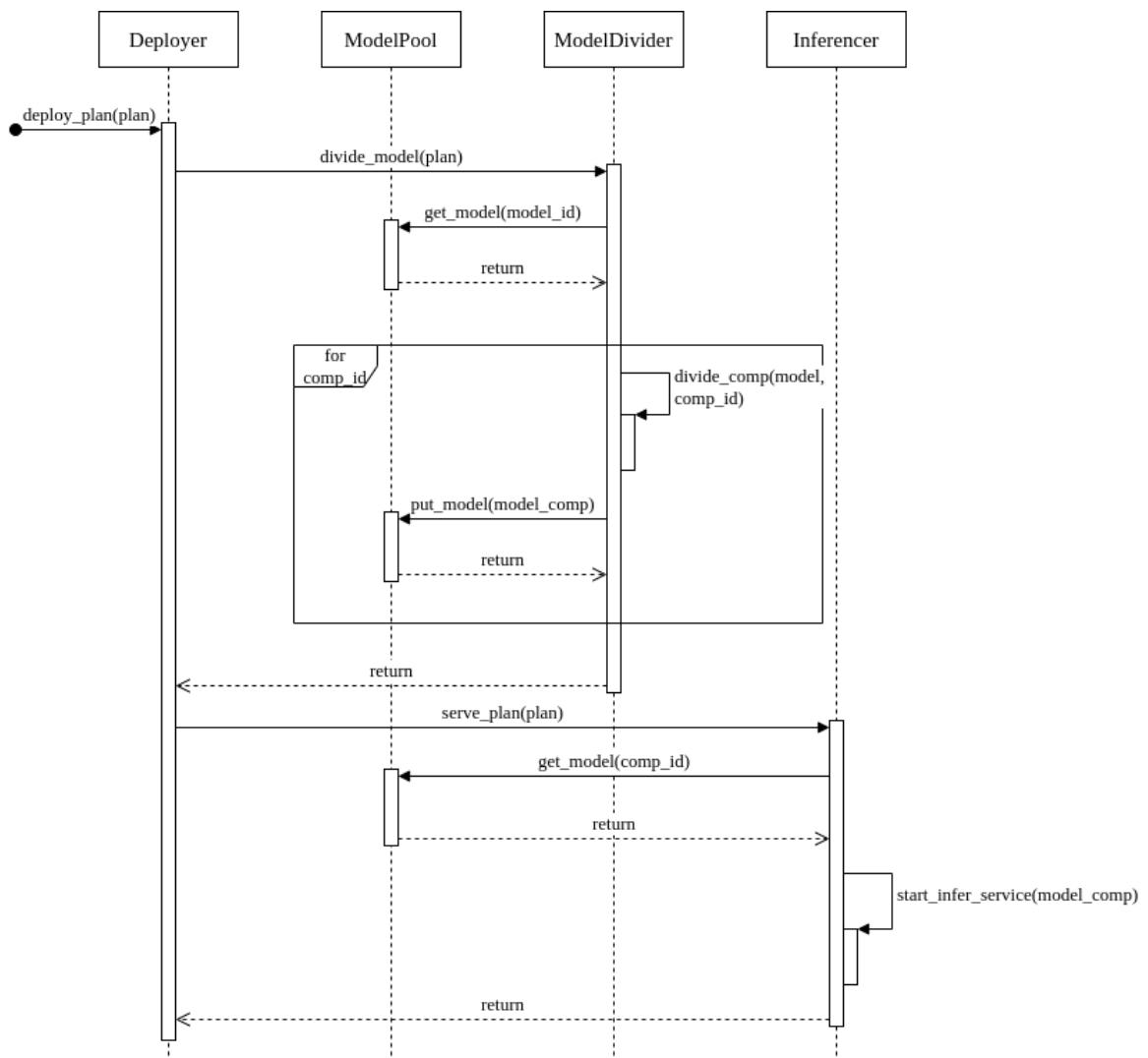


Figura 3.7: Sequence Diagram: Deployment del Piano

3.2.4 Inferenza

In Figura 3.8 sono descritte le interazioni in fase di inferenza. La richiesta di inferenza, con il relativo input ed altri metadati ad esso relativi (e.g. l'identificativo del modello con il quale si vuole fare inferenza), viene per prima cosa inviata alla componente di FrontEnd, la quale, seguendo il piano che è stato mandato in deployment, la inoltra al primo Inferencer del piano. A questo punto, il singolo Inferencer esegue l'inferenza della componente ed invia l'output all'Inferencer successivo nel piano. Si noti che un Inferencer può avere in gestione più di una componente, quindi l'inferenza può essere fatta più volte dallo stesso Inferencer, ma per componenti diverse a parità di richiesta. L'output dell'inferenza viene ritornato alla componente di FrontEnd tramite una callback: nel periodo intermedio, il thread del FrontEnd che si è occupato della gestione dell'input originale rimane in stato di blocco (stato rappresentato con la notazione non standard dell'azzurro sull'activation bar di questa componente).

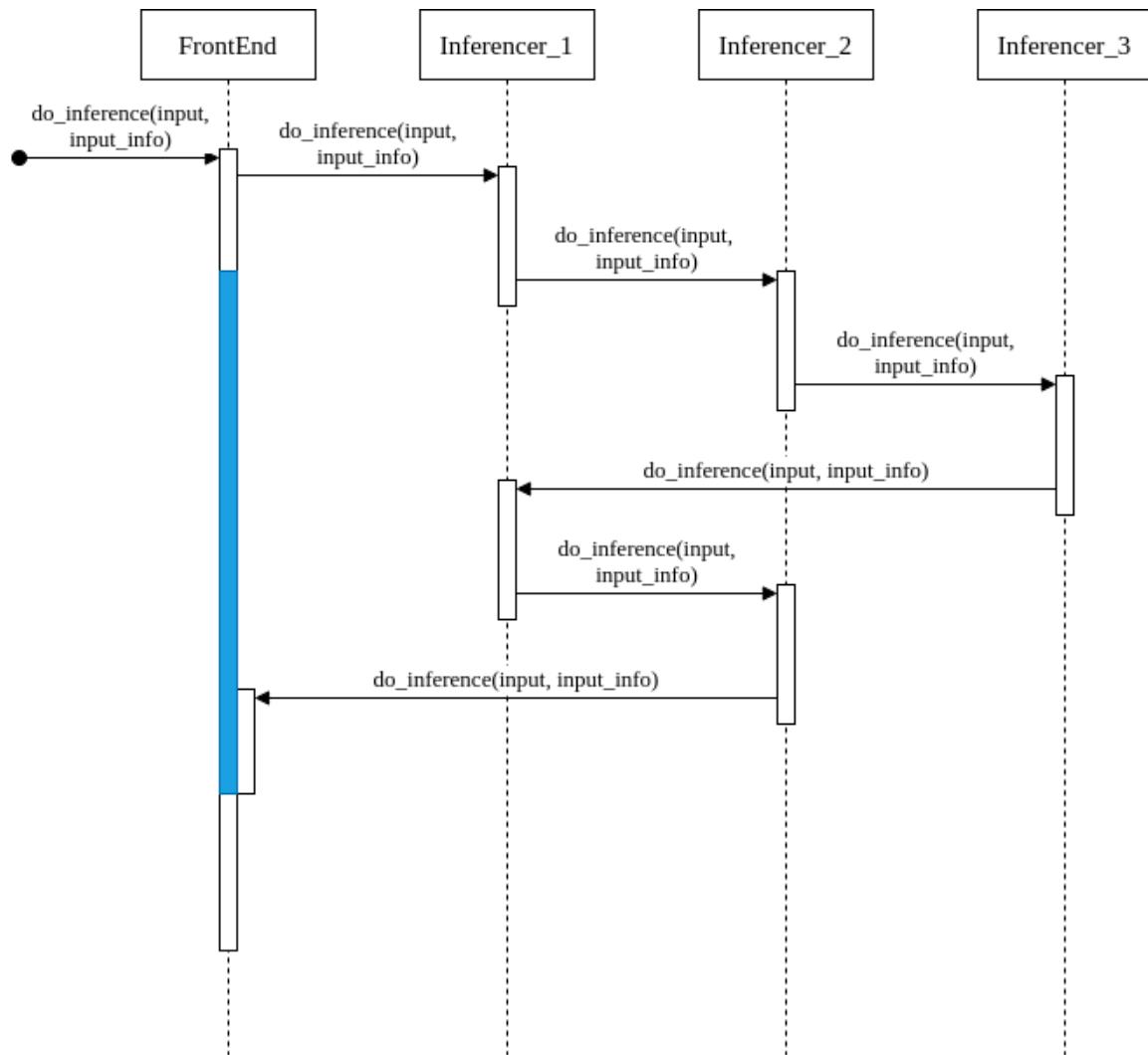


Figura 3.8: Sequence Diagram: Inferenza

Capitolo 4

Problema di Ottimizzazione

descritto

In questo capitolo verrà spiegato ~~il problema di ottimizzazione che sarà risolto e del problema in esame~~ la ~~sua~~ modellazione come problema di ottimizzazione lineare intera.

Verrà introdotta l'astrazione dei grafi usata per la modellazione, per poi definire le variabili di base che sono coinvolte nel problema. Si passerà quindi alla modellazione del tempo di inferenza, del consumo energetico e della memoria.

Verrà quindi introdotta la modellazione del rumore di quantizzazione e spiegato il modo in cui è stato integrato nella modellazione del problema.

Si continuerà poi con l'integrazione delle parti costituenti il problema e con la spiegazione della strategia di normalizzazione adottata.

Si concluderà con una spiegazione relativa alla post elaborazione della soluzione, in particolare la sua necessità e il modo in cui è stata sviluppata.

4.1 Modello di Base

Modelliamo una DNN D come un grafo logico, rappresentabile mediante una coppia $G_D = (V_D, E_D)$, dove:

- V_D rappresenta l'insieme dei nodi del grafo; ogni nodo è associato univocamente ad un livello all'interno della DNN. Dato un $i \in V_D$ indichiamo con:
 - $i.quantizable$ un attributo che indica se il livello è quantizzabile o meno.
- E_D rappresenta l'insieme degli archi orientati del grafo; ogni arco è associato univocamente ad uno spostamento di dati che avviene tra i livelli del modello. Dato un $m \in E_D$ indichiamo con:
 - $m.tensorsName$ una lista con i nomi dei tensori che vengono trasportati tramite quell'arco.

Indichiamo con:

- $V_I \subset V_D$ i nodi di input del modello: questi nodi sono tali per cui non esistono archi di ingresso a questi nodi; in simboli $\forall i \in V_I \quad \nexists u \in V_D \text{ tc } \exists(u, i) \in E_D$;
- $V_O \subset V_D$ i nodi di output del modello: questi nodi sono tali per cui non esistono archi di uscita da questi nodi; in simboli $\forall i \in V_O \quad \nexists u \in V_D \text{ tc } \exists(i, u) \in E_D$

Definiamo anche T_D l'insieme dei tensori che vengono scambiati tra i livelli all'interno del modello. Dato un tensore $t \in T_D$, definiamo le seguenti caratteristiche:

- $t.tensorName$ è il nome del tensore;
- $t.tensorSize$ è la dimensione del tensore;
- $t.tensorSource$ è il nodo $i \in V_D$ che genera il tensore.

Sia invece G_N il grafo della rete. G_N può essere rappresentato come un grafo diretto mediante una coppia $G_N = (V_N, E_N)$, dove:

- V_N rappresenta l'insieme dei nodi del grafo, ognuno dei quali è associato univocamente ad un server della rete. Dato un $k \in V_N$ indichiamo con:
 - $k.computePower$ un valore che indica la potenza consumata dal dispositivo in fase di calcolo.
 - $k.selfTxPower$ un valore che indica la potenza consumata dal dispositivo in fase di trasmissione a se stesso.
 - $k.otherTxPower$ un valore che indica la potenza consumata dal dispositivo in fase di trasmissione ad altri dispositivi.
- E_N rappresenta l'insieme degli archi del grafo, ognuno dei quali è associato univocamente ad un collegamento di rete presente tra i server di rete. Dato un $n \in E_N$ indichiamo con:
 - $n.bandWidth$ un valore che indica la banda disponibile sul collegamento fisico;
 - $n.rtt$ il valore che indica il round-trip-time sul collegamento fisico.

Si notino i seguenti due aspetti:

- Non essendoci limitazioni sulla presenza di cicli, all'interno di E_N è ammessa la presenza dei così detti *cappi*: in simboli abbiamo che $\forall u \in V_N \exists (u, u) \in E_N$; questo permette di modellare il caso in cui un nodo passa dati a se stesso. Questa modellazione richiederà un post-processing aggiuntivo che sarà meglio descritto in sezione 4.10.
- Assumiamo che tutte le possibili coppie di archi siano presenti in E_N : il nodi di rete quindi sono assunti come organizzati in una mesh.

Vogliamo dunque costruire un mapping del grafo logico (i.e. la DNN) sul grafo fisico (i.e. la rete di server). Ci riconduciamo, quindi, ad un problema di graph assignment e graph partitioning simili a quelli descritti in [31, 32]. Si noti, tuttavia, che non siamo interessati ad una corrispondenza 1:1 tra gli archi logici di E_D sugli archi fisici in E_N : nel nostro caso, infatti, risulta di interesse maggiore il mapping di un tensore su un arco fisico, a rappresentare la trasmissione di quel tensore attraverso quell'arco. Supponiamo, ad esempio, di trovarci in un caso come quello in Figura 4.1

💡 di avere due possibili server indicati con k ed h e che il nodo a sia assegnato al server k ; se sia il nodo b che il nodo c vengono mappati sul server h , allora sarà sufficiente che la trasmissione di t avvenga una sola sull'arco fisico (k, h) al fine di permettere al server h di calcolare il risultato dei nodi che gli competono.

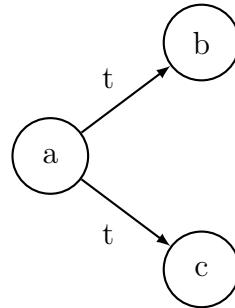


Figura 4.1: Esempio Mapping di Tensore

4.2 Le Variabili

Definiamo in prima istanza le seguenti variabili:

- $x_{ik} \in \{0, 1\} \quad \forall i \in V_D, \forall k \in V_N$; in particolare $x_{ik} = 1$ se e solo se il layer $i \in V_D$ viene mappato su server $k \in V_N$;
- $y_{tn} \in \{0, 1\} \quad \forall t \in T_D, \forall n \in E_N$; in particolare $y_{tn} = 1$ se e solo se il tensore $t \in T_D$ viene trasmesso sull'arco fisico $n \in E_N$

- $q_i \in \{0, 1\} \forall i \in V_D$; in particolare $q_i = 1$ se e solo se il layer $i \in V_D$ viene quantizzato. Notiamo che nel nostro caso stiamo considerando i soli due casi *quantizzato* e *non quantizzato*, quindi la variabile binaria è sufficiente a rappresentare la nostra situazione.
- $x_{ik}^q \in \{0, 1\} \forall i \in V_D, \forall k \in V_N$; in particolare, $x_{ik}^q = 1$ se e solo se il livello i viene assegnato al server k e per il livello i viene attivata la quantizzazione.
- $y_{tn}^q \in \{0, 1\} \forall t \in T_D, \forall n \in E_N$; in particolare, $y_{tn}^q = 1$ se e solo se il tensore t viene trasmesso sull'arco fisico n e per il nodo logico generatore del tensore viene attivata la quantizzazione.

4.3 Modello di Tempo

4.3.1 Tempo di Calcolo

Indichiamo con $f_k(i, q_i)$ una funzione che ci dice per il server $k \in V_N$, il tempo di calcolo per il layer $i \in V_D$, quando il suo stato di quantizzazione è q_i (i.e. quantizzato vs non-quantizzato). Partendo da questo, possiamo definire la seguente espressione:

$$T_{ik}^c = [f_k(i, 0) \cdot (1 - q_i) + f_k(i, 1) \cdot q_i] \cdot x_{ik} \quad (4.3.1)$$

Nello specifico, il tempo impiegato per eseguire il livello i sul server k sarà nullo quando $x_{ik} = 0$; altrimenti sarà pari a $f_k(i, 0)$, se il livello non è quantizzato, oppure pari a $f_k(i, 1)$, se il livello è quantizzato. Sviluppando il prodotto nell'espressione precedente, troviamo la seguente formulazione:

$$\begin{aligned} T_{ik}^c &= f_k(i, 0) \cdot x_{ik} - (f_k(i, 0) - f_k(i, 1)) \cdot x_{ik} \cdot q_i \\ &= f_k(i, 0) \cdot x_{ik} - (f_k(i, 0) - f_k(i, 1)) \cdot x_{ik}^q \end{aligned} \quad (4.3.2)$$

Notiamo che in Equazione 4.3.2 è stata usata la variabile x_{ik}^q . Questa variabile, dunque, modella il prodotto tra le due variabili x_{ik} e q_i ; affinché questa modellazione sia valida, la variabile deve essere soggetta ai seguenti vincoli:

$$\begin{cases} x_{ik}^q \leq x_{ik} \\ x_{ik}^q \leq q_i \\ x_{ik}^q \geq x_{ik} + q_i - 1 \end{cases} \quad (4.3.3)$$

Analizziamo in dettaglio Equazione 4.3.2. Possiamo notare come il primo termine rappresenti il tempo di calcolo del livello i sul server k : questo tempo c'è sempre, a prescindere dallo stato di quantizzazione del livello; al contrario, il termine tra parentesi lo possiamo vedere come un *guadagno di quantizzazione*, che viene sottratto al tempo di esecuzione del livello quando si sceglie di attivare la quantizzazione per quel livello su quel server.

A questo punto, quindi, abbiamo che il tempo di calcolo per il server k è dato da:

$$T_k^c = \sum_{i \in V_D} T_{ik}^c \quad (4.3.4)$$

Per concludere, il tempo di calcolo complessivo del modello all'interno del sistema è dato da:

$$T^c = \sum_{k \in V_N} T_k^c \quad (4.3.5)$$

4.3.2 Tempo di Trasmissione

Indichiamo con $g(t, n)$ la funzione che stima il tempo di trasmissione del tensore $t \in T_D$ attraverso l'arco fisico $n \in V_N$. Nel nostro caso possiamo definirla come:

$$g(t, n) = \frac{t.tensorSize}{n.bandWidth}$$

Da ciò, indicati con:

- $t \in T_D$ un tensore;
- $i = t.tensorSource \in V_D$ il nodo sorgente del tensore;
- $n \in E_N$ l'arco fisico su cui mappiamo il tensore;
- $n[0] = k$ la sorgente dell'arco fisico;

definiamo il tempo di trasmissione del tensore t sull'arco n a carico del server k come:

$$T_{tk}^x = \left[g(t, n) \cdot (1 - q_i) + \frac{g(t, n)}{\alpha} \cdot q_i \right] \cdot y_{tn} + n.rtt \cdot y_{tn} \quad (4.3.6)$$

Nello specifico, quindi, il tempo impiegato per la trasmissione del tensore t attraverso l'arco fisico n sarà nullo se $y_{tn} = 0$; altrimenti, sarà pari al valore della latenza dell'arco fisico sommato con un valore dipendente da $g(t, n)$. Notiamo che quando $q_i = 1$, il tempo di trasmissione per il tensore è scalato di α : questo permette di modellare il fatto che i dati del tensore hanno una dimensione minore quando viene attivata la quantizzazione per il livello sorgente di quel tensore. Nel nostro caso, assumendo quantizzazione *INT8* e dati di tipo *FLOAT32*, avremo $\alpha = 4$. Eseguendo il prodotto nella relazione precedente, troviamo la seguente formulazione:

$$\begin{aligned} T_{tk}^x &= g(t, n) \cdot y_{tn} - \left(g(t, n) - \frac{g(t, n)}{\alpha} \right) \cdot y_{tn} \cdot q_i + n.rtt \cdot y_{tn} \\ &= g(t, n) \cdot y_{tn} - \left(g(t, n) - \frac{g(t, n)}{\alpha} \right) \cdot y_{tn}^q + n.rtt \cdot y_{tn} \end{aligned} \quad (4.3.7)$$

Notiamo che in Equazione 4.3.7 è stata usata la variabile y_{tn}^q . Questa variabile, dunque, modella il prodotto tra le due variabili y_{tn} e q_i ; affinché questa modellazione sia valida, la variabile deve essere soggetta ai seguenti vincoli:

$$\begin{cases} y_{tn}^q \leq y_{tn} \\ y_{tn}^q \leq q_i \\ y_{tn}^q \geq y_{tn} + q_i - 1 \end{cases} \quad (4.3.8)$$

Analizziamo in dettaglio Equazione 4.3.7. Possiamo notare come il primo termine rappresenti il tempo di trasmissione del tensore sull'arco fisico: questo tempo c'è sempre, a prescindere dallo stato di quantizzazione del livello sorgente; al contrario, il termine tra parentesi lo possiamo vedere come un *guadagno di quantizzazione*, che viene sottratto al tempo di trasmissione base quando si sceglie di quantizzare il livello sorgente dei dati. Per concludere, notiamo come il termine relativo al round-trip-time sia sempre presente: questo è infatti un costo che viene sempre pagato e che non può essere scontato in quanto dipende dalla distanza tra i server che comunicano. Si noti che è stato scelto l'rtt e non la latenza dell'arco in quanto, terminata la trasmissione del tensore, sarà necessario aspettare l'arrivo del messaggio alla controparte e la conseguente trasmissione di un ACK dal ricevente.

Partendo da Equazione 4.3.7 possiamo definire il tempo complessivo di trasmissione di un server nel seguente modo:

$$\begin{aligned} T_k^x &= \sum_{t \in T_D} \left(\sum_{n \in E_N \wedge k=n[0]=n[1]} T_{tk}^x + \sum_{n \in E_N \wedge k=n[0] \wedge k \neq n[1]} T_{tk}^x \right) \\ &= \sum_{t \in T_D} (T_{tk}^{x-self} + T_{tk}^{x-other}) = T_k^{x-self} + T_k^{x-other} \end{aligned}$$

Dove T_{tk}^{x-self} rappresenta il tempo che il server k impiega a trasmettere il tensore $t \in T_D$ attraverso l'arco di rete $(k, k) \in E_N$, ovvero a "trasmettere" i dati a se stesso; viceversa, $T_{tk}^{x-other}$ rappresenta il tempo che il server k impiega a trasmettere il tensore $t \in T_D$ attraverso l'arco di rete $(k, h) \in E_N$ con $h \neq k$.

In conclusione, il tempo di trasmissione complessivo per il modello è dato da:

$$T^x = \sum_{k \in V_N} T_k^x \quad (4.3.9)$$

4.3.3 Tempo Complessivo

Il tempo complessivo lo possiamo definire sommando Equazione 4.3.5 e Equazione 4.3.9, trovando:

$$T = T^c + T^x \quad (4.3.10)$$

4.4 Modello di Energia

4.4.1 Energia di Calcolo

Indichiamo con $h_k^c(t)$ la funzione che stima l'energia usata per il calcolo di durata t dal dispositivo k . Nel nostro caso assumiamo una funzione lineare del tipo:

$$h_k^c(t) = k.computePower \cdot t \quad (4.4.1)$$

L'energia consumata per il calcolo sul server k sarà data da:

$$E_k^c = h_k^c(T_k^c)$$

Il consumo energetico complessivo dato dal calcolo del modello è dato da:

$$E^c = \sum_{k \in V_N} E_k^c$$

4.4.2 Energia di Trasmissione

Siano $h_k^{x-self}(t)$ e $h_k^{x-other}(t)$ rispettivamente le funzioni che stimano l'energia usata per la trasmissione di durata t dal dispositivo k a se stesso e dal dispositivo k ad un dispositivo h dove $k \neq h$. Assumiamo anche in questo caso delle funzioni lineari di tipo:

$$\begin{aligned} h_k^{x-self}(t) &= k.selfTxPower \cdot t \\ h_k^{x-other}(t) &= k.otherTxPower \cdot t \end{aligned} \quad (4.4.2)$$

L'energia consumata sul server k per la trasmissione è definita come:

$$E_k^x = l_k^{x-self}(T_k^{x-self}) + l_k^{x-other}(T_k^{x-other})$$

In questo caso, quindi, come nel caso del tempo di trasmissione, il contributo al consumo energetico è dato dal contributo di due termini, di cui il primo rappresenta il consumo energetico per trasmettere verso se stessi, che dipenderà tendenzialmente dal consumo dell'interfaccia di loopback, e il secondo rappresenta il consumo energetico per trasmettere verso altri nodi della rete.

Il consumo energetico complessivo per la trasmissione è dato da:

$$E^x = \sum_{k \in V_N} E_k^x$$

4.4.3 Energia del Server k-esimo

Il consumo energetico sul server k lo possiamo definire attraverso la somma seguente:

$$E_k = E_k^c + E_k^x \quad (4.4.3)$$

4.4.4 Consumo Energetico Complessivo

Il consumo energetico complessivo sarà dato dalla seguente relazione:

$$E = E^c + E^x \quad (4.4.4)$$

4.5 Modello di Memoria

La memoria può essere vista come composta da tre parti principali:

- Parte in cui sono mantenuti i pesi del modello;
- Parte in cui sono mantenuti gli input dei nodi del modello;

- Parte in cui sono mantenuti gli output dei nodi del modello.

Di conseguenza, detta m_k la memoria occupata sul server k , possiamo esprimere m_k nel seguente modo:

$$m_k = \mu_k^{weig} + \mu_k^{inp} + \mu_k^{out} \quad (4.5.1)$$

Assumendo che l'engine di inferenza faccia riuso della memoria in modo efficiente, possiamo considerare il secondo e il terzo termine trascurabili, arrivando a scrivere:

$$m_k = \sum_{i \in V_D} \left(i.weightSize \cdot (1 - q_i) + \frac{i.weightSize}{\alpha} \cdot q_i \right) \cdot x_{ik} \quad (4.5.2)$$

Sviluppando il prodotto, come nei casi precedenti troviamo la seguente relazione:

$$m_k = \sum_{i \in V_D} \left[i.weightSize \cdot x_{ik} - \left(i.weightSize - \frac{i.weightSize}{\alpha} \right) \cdot x_{ik}^q \right] \quad (4.5.3)$$

4.6 Modello di Rumore di Quantizzazione

Come discusso in sottosottosezione 3.1.1.2 consideriamo per la quantizzazione soltanto un sottoinsieme di livelli; sia quindi $V_Q = \{i \in V_D : i.quantizable = True\} \subseteq V_D$. Indichiamo poi con $\mathbf{q} = \{q_i\}_{i \in V_Q}$ il vettore delle variabili q_i di quantizzazione dei livelli quantizzabili: questo vettore definisce uno schema di quantizzazione del modello, ovvero una certa combinazione di quantizzazioni/non-quantizzazioni per i livelli quantizzabili.

Supponiamo di avere una regressione polinomiale $\eta(\mathbf{q})$ di grado d che, dato in input il vettore \mathbf{q} sopra definito, restituisce come risultato il valore del rumore di quantizzazione quando viene applicato lo schema di quantizzazione definito da \mathbf{q} .

Trattandosi di una regressione polinomiale di grado d , al suo interno potremo avere prodotti di al massimo d variabili $\{q_i\}_{i \in V_Q}$. Indicato con $\mathcal{C}_{\leq h}(V_Q)$ l'insieme dei sottoinsiemi di V_Q di cardinalità minore o uguale ad h , sia $\hat{q}_k \in \{0, 1\}$ una variabile

che rappresenta il prodotto logico delle variabili q_i per $i \in V_{Q_k}$ e per $V_{Q_k} \in \mathcal{C}_{\leq d}(V_Q)$.

Ognuna di queste variabili è soggetta ai seguenti vincoli:

$$\begin{cases} \hat{q}_k \leq q_i & \forall i \in V_{Q_k} \\ \hat{q}_k \geq \sum_{i \in V_{Q_k}} q_i - (|V_{Q_k}| - 1) \end{cases} \quad (4.6.1)$$

Definiamo una variabile di esistenza $p \in \{0, 1\}$ tale che $p = 1$ se e solo se $\exists i \in V_Q$ tale che $q_i = 1$. Questa variabile ci dice se esiste almeno un livello quantizzato e sarà dunque soggetta ai vincoli seguenti:

$$\begin{cases} p \geq q_i & \forall i \in V_Q \\ p \leq \sum_{i \in V_Q} q_i \end{cases} \quad (4.6.2)$$

Possiamo quindi ridefinire il regressore come $\hat{\eta}(\hat{\mathbf{q}}, p)$, dove $\hat{\mathbf{q}} = \{\hat{q}_k\}_{V_{Q_k} \in \mathcal{C}_{\leq d}(V_Q)}$, ottenendo in definitiva, indicati con \mathbf{w}^T i pesi del regressore e con c la sua intercetta:

$$\hat{\eta}(\hat{\mathbf{q}}, p) = \mathbf{w}^T \hat{\mathbf{q}} + c \cdot p \quad (4.6.3)$$

Che è rappresentabile, posti i vincoli lineari definiti fino a questo punto, in modo lineare.

Si noti che la variabile binaria p è stata definita per garantire che in assenza di livelli quantizzati il risultato del regressore sia sempre nullo (i.e. $\hat{\eta}(\mathbf{0}, 0) = 0$). Potrebbe essere possibile, infatti, che in fase di addestramento del regressore l'intercetta sia calcolata come non nulla per avere un fitting migliore; tuttavia, considerando come baseline per il rumore il modello originale, un valore del rumore non nullo in assenza di quantizzazione sarebbe paradossale, in quanto l'assenza di livelli quantizzati implica l'uguaglianza con il modello originale.

4.7 I Vincoli

4.7.1 Vincoli di Quantizzazione

Ricordando che soltanto alcuni livelli possono essere quantizzati, possiamo imporre il vincolo:

$$q_i = 0 \quad \forall i \notin V_Q = \{i \in V_D : i.quantizable = True\} \quad (4.7.1)$$

Questo assicura che i livelli non quantizzabili non saranno scelti per la quantizzazione in fase di ottimizzazione.

4.7.2 Vincoli di Assegnazione

I vincoli di assegnazione che dobbiamo definire per il modello sono i seguenti:

$$\begin{aligned} \sum_{k \in V_N} x_{ik} &= 1 & \forall i \in V_D \\ x_{i0}^a &= 1 & \forall i \in V_I \\ x_{j0}^a &= 1 & \forall j \in V_O \end{aligned} \quad (4.7.2)$$

Dove:

- Il primo vincolo garantisce assegnazione dei nodi logici ai nodi fisici.
- Il secondo e il terzo vincolo garantiscono che, detto $k = 0$ il server da cui viene fatta partire l'inferenza, i nodi di input e di output saranno mappati su quel server.

4.7.3 Vincoli di Flusso

Dato un tensore $t \in T_D$, indichiamo con $V_D^t \subset V_D$ l'insieme dei nodi logici che ricevono in input t e con $i = t.tensorSource \in V_D$ il nodo sorgente di t . I vincoli di flusso che

dobbiamo definire per il modello sono i seguenti.

$$\forall t \in T_D, \forall n = (k, h) \in E_N \quad \begin{cases} y_{tn} \leq x_{ik} \\ y_{tn} \leq \sum_{j \in V_D^t} x_{jh} \\ y_{tn} \geq x_{ik} + \frac{1}{|V_D^t|} \sum_{j \in V_D^t} x_{jh} - 1 \end{cases} \quad (4.7.3)$$

Dove:

- Il primo vincolo della terna forza la variabile a 0 se il nodo sorgente del tensore non è mappato sul server k sorgente dell'arco fisico.
- Il secondo vincolo della terna forza la variabile a 0 se non ci sono nodi logici che ricevono in input il tensore che sono mappati sul server h destinazione dell'arco fisico.
- Il terzo vincolo della terna forza la variabile ad 1 nel momento in cui entrambe le condizioni precedenti sono soddisfatte.

Si noti che non è necessario il vincolo per il quale un tensore deve essere mappato su almeno un arco di rete: infatti, poiché i nodi di rete sono organizzati in una mesh e poiché ogni livello è assegnato esattamente ad un server, dato un tensore t che collega due livelli del modello, la variabile y_{tn} avrà valore 1 almeno per quell'arco di rete ai cui estremi sono mappati i livelli sorgente e destinazione del tensore.

4.7.4 Spazio delle Variabili

Gli spazi delle variabili sono definiti nella seguente relazione:

$$\begin{aligned}
 x_{ik} &\in \{0, 1\} & \forall i \in V_D, \forall k \in V_N \\
 y_{tn} &\in \{0, 1\} & \forall t \in T_D, \forall n \in E_N \\
 q_i &\in \{0, 1\} & \forall i \in V_D \\
 x_{ik}^q &\in \{0, 1\} & \forall i \in V_D, \forall k \in V_N \\
 y_{tn}^q &\in \{0, 1\} & \forall t \in T_D, \forall n \in E_N \\
 \hat{q}_k &\in \{0, 1\} & \forall V_{Q_k} \in \mathcal{C}_{\leq d}(V_Q) \\
 p &\in \{0, 1\}
 \end{aligned} \tag{4.7.4}$$

4.8 Il Problema

Siano:

- ω_t il peso associato al tempo di inferenza;
- ω_e il peso associato all'energia di inferenza;
- J_0 l'energia massima consumabile sul device $k = 0 \in V_N$, cioè il server che fa partire l'inferenza;
- η_{max} il rumore di quantizzazione massimo accettato;
- m_k^{max} la memoria massima disponibile sul server $k \in V_N$.

Supponiamo che i pesi siano normalizzati: $\omega_t + \omega_e = 1$.

Definiamo quindi il problema di minimizzazione nel seguente modo:

$$\begin{aligned}
 & \min \quad o(\omega_t \cdot T, \omega_e \cdot E) \\
 & \text{subject to} \quad E_0 \leq J_0 \\
 & \quad \hat{\eta}(\hat{\mathbf{q}}, p) \leq \eta_{max} \\
 & \quad m_k \leq m_k^{max} \quad \forall k \in V_N
 \end{aligned} \tag{4.8.1}$$

Oltre a questi vincoli, abbiamo anche: (4.3.3), (4.3.8), (4.6.1), (4.6.2), (4.7.1), (4.7.2), (4.7.3) e (4.7.4).

4.9 Normalizzazione dell’Obiettivo

Notiamo che il problema definito in questo modo è un problema di ottimizzazione multi-obiettivo che, in generale, risulta complesso da risolvere.

Una delle prime complicazioni da affrontare riguarda la scala dei valori: nel nostro caso, infatti, gli obiettivi sono misurati secondo unità diverse (nello specifico, secondi per il tempo e Joule per l’energia) e ciò può comportare a delle scale diverse che possono impattare negativamente la fase di ottimizzazione. Il modo più semplice di risolvere questo problema è normalizzare le due funzioni nell’intervallo $[0, 1]$ in modo che il loro peso nella funzione obiettivo sia equiparabile. Per farlo, possiamo seguire l’approccio descritto in [33] per l’ottimizzazione pesata.

Nello specifico, procediamo nel modo descritto di seguito:

1. Risolviamo il seguente problema:

- Obiettivo: $\min T$
- Vincoli: come in Equazione 4.8.1
- Risultato: il valore T_{min} sul punto $x_{T_{min}}$

2. Risolviamo il seguente problema:

- Obiettivo: $\min E$
- Vincoli: come in Equazione 4.8.1
- Risultato: il valore E_{min} sul punto $x_{E_{min}}$

3. Calcoliamo i seguenti valori:

- $T_{max} = \max(T(x_{T_{min}}), T(x_{E_{min}}))$
- $E_{max} = \max(E(x_{T_{min}}), E(x_{E_{min}}))$

4. Calcoliamo le seguenti funzioni:

- $T^{norm} = \frac{T - T_{min}}{T_{max} - T_{min}}$
- $E^{norm} = \frac{E - E_{min}}{E_{max} - E_{min}}$

5. Risolviamo il seguente problema di ottimizzazione:

- Obiettivo: $\min (\omega_T \cdot T^{norm} + \omega_E \cdot E^{norm})$
- Vincoli: come in Equazione 4.8.1

Definite le funzioni T^{norm} e E^{norm} , notiamo che si tratta di funzioni adimensionali entrambe nell'intervallo $[0, 1]$, quindi possono essere combinate senza temere che la diversa scala di valori o le diverse unità di misura impattino negativamente l'ottimizzazione.

Al variare dei pesi nell'intervallo $[0, 1]$ si otterranno i vari punti della frontiera di Pareto. Notiamo che i valori estremi dei pesi coincidono con i casi estremi di ottimizzazione: nello specifico, quando $\omega_T = 1$ (e quindi $\omega_E = 0$) viene ottimizzata

soltanto la funzione tempo; viceversa, l'unica funzione che impatta l'ottimizzazione è la funzione di energia.

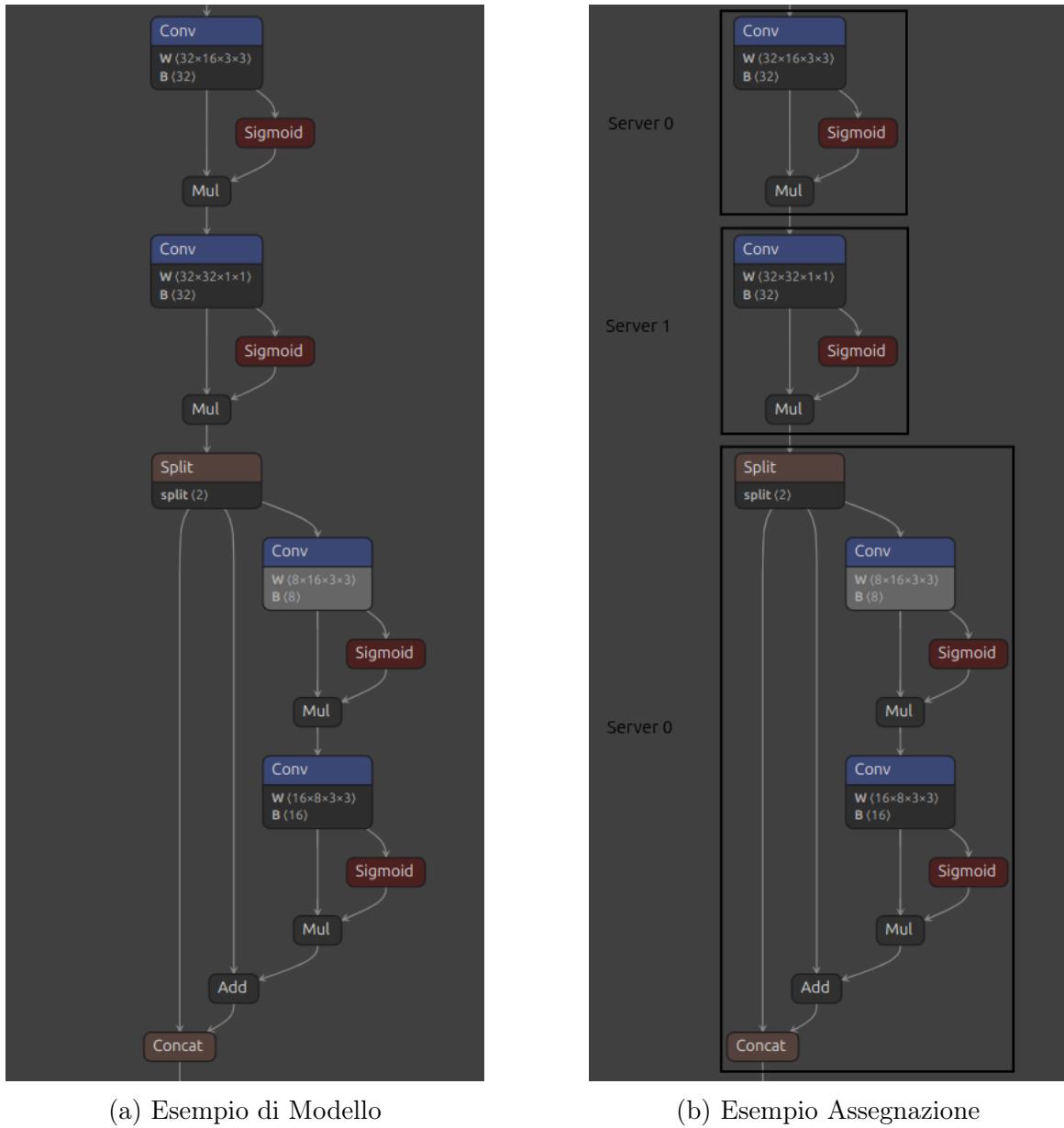
Di fatto, questo è stato ritenuto l'approccio più semplice per normalizzare il problema; in [33] viene riportato l'approccio di ottimizzazione gerarchica come alternativa che, sebbene interessante, non è stato ulteriormente approfondito ai fini di questo studio.

4.10 Post Elaborazione della Soluzione

Risolto il problema di ottimizzazione, possiamo raccogliere il risultato. Le informazioni fondamentali da ricavare dalla soluzione sono:

- Valori delle variabili $x_{ik} = 1$, per sapere, dato un livello, a quale server è stato assegnato;
- Valori delle variabili $q_i = 1$, per sapere, dato un livello (quantizzabile), se la quantizzazione per quel livello è stata attivata o meno.

Indichiamo con il termone *componente* un insieme di livelli del modello originale che sono interconnessi tra di loro e che vengono assegnati allo stesso server. Supposto che il modello originale sia quello rappresentato in Figura 4.2a, una possibile divisione in componenti del modello è quella rappresentata in Figura 4.2b. Da questo esempio si può vedere come non sia scontato che ad un server corrisponda una singola componente: è infatti possibile che l'insieme di livelli assegnati ad un server sia tale per cui non vi siano dei collegamenti tra questi livelli. Dato un server, quindi, ci sarà un certo numero di componenti ad esso assegnate e questo numero dipenderà da come è stata fatta la suddivisione in fase di ottimizzazione.



(a) Esempio di Modello

(b) Esempio Assegnazione

Figura 4.2: Esempio di Costruzione del Grafo delle Componenti

Stabilita la divisione in componenti, è possibile definire un grafo avente per nodi queste componenti e per archi i collegamenti tra livelli adiacenti assegnati a server diversi; mantenendoci sull'esempio in Figura 4.2b avremo un totale di tre nodi e due archi, di cui il primo è rappresentato dall'arco che collega la prima **Mul** al secondo **Conv** e il secondo dall'arco che collega la seconda **Mul** allo **Split**. Questo grafo di componenti non sarà altro che un'astrazione costruita sul grafo originale del modello sulla base delle assegnazioni dei livelli ai server e sulla base delle connessioni tra i livelli assegnati allo stesso server.

Tuttavia, la costruzione delle componenti sulla base delle sole assegnazioni e connessioni tra i livelli del modello originale potrebbe non essere sufficiente ad evitare problemi di ciclicità. Essendo il grafo originale un DAG è necessario che il grafo delle componenti sia a sua volta un DAG, pena la possibilità di incorrere in deadlock in fase di inferenza. L'aciclicità del grafo delle componenti non è garantita per due motivi: in primo luogo stiamo assegnando nodi di un DAG ai nodi di un grafo (i.e. il grafo di rete) che non è aciclico; in secondo luogo, all'interno del problema non ci sono vincoli che garantiscono l'assegnazione aciclica. Una possibile soluzione potrebbe essere l'introduzione di un vincolo di aciclicità all'interno del problema, come le soluzioni descritte in [34]: ciò però porterebbe ad un'ulteriore complicazione del problema, oltre all'impossibilità di avere dei ritorni indietro nel flusso dei dati tra i server (a meno di non definire dei server fittizi che lo permettano).

Si è quindi preferito adottare un approccio di post-elaborazione della soluzione, in cui eventuali cicli tra le componenti vengano risolti. Un esempio di assegnamento ciclico è presentato in Figura 4.3a: in questo caso la componente assegnata al server 0 riceve il suo input da quella assegnata al server 1 e, allo stesso tempo, invia il suo output a questa medesima componente. Questo assegnamento ciclico si può risolvere,

ad esempio, come mostrato in Figura 4.3b, dividendo la componente assegnata al server 1 in due componenti separate. In Algoritmo 1 e Algoritmo 2 viene riportato lo pseudo codice degli algoritmi usati per la costruzione delle componenti e la costruzione di un grafo di componenti aciclico.

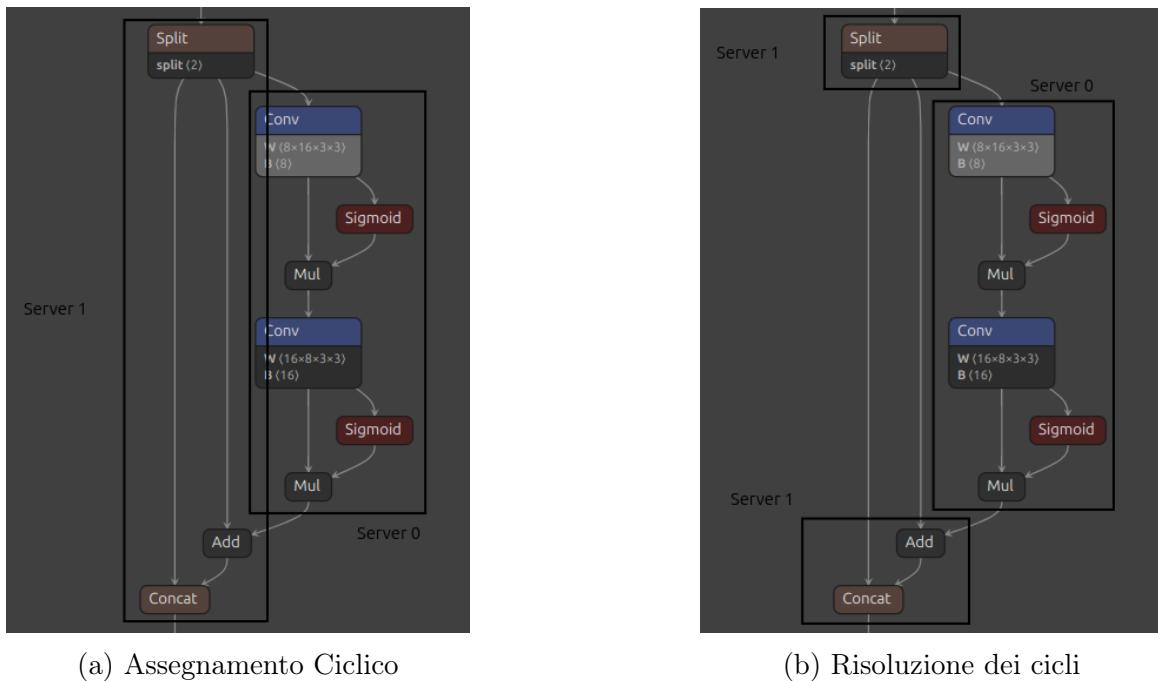


Figura 4.3: Esempio di assegnazione ciclica e risoluzione

Analizziamo alcuni aspetti significativi di Algoritmo 1:

- L’attraversamento del grafo viene fatto seguendo un ordinamento topologico: questo permette una migliore gestione e analisi delle dipendenze.
- L’insieme `excludeSet` rappresenta l’insieme di componenti che dobbiamo escludere dalle possibili perché l’aggiunta del livello in una di queste componenti introdurrebbe una dipendenza circolare: nello specifico, queste componenti da escludere sono quelle possibili p tali per cui esiste una componente d da cui il livello l dipende e per cui c’è dipendenza tra p e d ; inserire il livello nella com-

Algoritmo 1 Assignment of Nodes to Components

```
1: Function ASSIGNNODESTOCOMPONENTS(modelGraph, assignmentMap)
2: Initialize an empty table nodeComponentAssignment
3: Initialize an empty table nodeDepDict
4: Initialize an empty table nodePosDict
5: Initialize an empty table compDepDict In italiano sarebbe meglio per consistenza  
:-)
6: for each node n in modelGraph.getNodes() do
7:   Assign the empty set to entry n in nodeDepDict
8:   Assign the empty set to entry n in nodePosDict
9: end for
10: for each node n in the topological ordering of modelGraph do
11:   dependencySet  $\leftarrow$  the set of dependencies of n from nodeDepDict
12:   possibleSet  $\leftarrow$  the candidate components of n from nodePosDict
13:   Initialize excludeSet as an empty set
14:   for each component cdep in dependencySet do
15:     for each candidate component cpos in possibleSet do
16:       if cpos belongs to the dependencies of cdep in compDepDict then
17:         Add cpos to excludeSet
18:       end if
19:     end for
20:   end for
21:   differenceSet  $\leftarrow$  possibleSet \ excludeSet
22:   if differenceSet is empty or n is a generator or receiver node then
23:     nodeCompId  $\leftarrow$  generate a new component identifier based on
      assignmentMap[n]
24:   else
25:     nodeCompId  $\leftarrow$  an element chosen arbitrarily from differenceSet
26:   end if
27:   Record in nodeComponentAssignment the association between n and
      nodeCompId
28:   Update the structures nodeDepDict, nodePosDict, and compDepDict by
      invoking procedure UPDATESETS with the current parameters
29: end for
=0
```

ponente p creerebbe un ciclo perché, indicata con $(x \rightarrow y)$ la dipendenza di y da x , $(p \rightarrow d)$ di base, ma inserire il livello l in p introdurrebbe la dipendenza $(d \rightarrow p)$ perché il livello dipende già da d .

- Si noti che, per motivi di semplicità di gestione, i nodi di input (i.e. generatori) e i nodi di output (i.e. receiver) sono gestiti in componenti separate.

Algoritmo 2 Update of Dependency Sets

```
1: Function UPDATESETS(modelGraph, assignmentMap, n, nodeCompId, nodeDepDict, nodePosDict, compDepDict)
2: if n is not a generator node in modelGraph then
3:   for each successor node s of n in modelGraph do
4:     if assignmentMap[n] equals assignmentMap[s] then
5:       Add nodeCompId to nodePosDict[s]
6:     end if
7:   end for
8:   for each parallel node p of n in modelGraph do
9:     if assignmentMap[n] equals assignmentMap[p] then
10:      Add nodeCompId to nodePosDict[p]
11:    end if
12:   end for
13: end if
14: Extend compDepDict[nodeCompId] with the contents of nodeDepDict[n], excluding nodeCompId itself
15: for each descendant node d of n in modelGraph do
16:   Add nodeCompId to nodeDepDict[d]
17:   Extend nodeDepDict[d] with the contents of compDepDict[nodeCompId]
18: end for
19: for each component identifier cother in the keys of compDepDict do
20:   if nodeCompId belongs to compDepDict[cother] then
21:     Extend compDepDict[cother] with the contents of nodeDepDict[n], excluding nodeCompId
22:   end if
23: end for
=0
```

Analizziamo alcuni aspetti significativi di Algoritmo 2:

- Il primo ciclo **for** aggiorna le componenti possibili dei nodi successori: se un nodo successore è assegnato allo stesso server del nodo corrente, allora la componente assegnata al nodo corrente può essere anche la componente del successore. Questo permette di accorpare livelli successivi assegnati allo stesso server nella stessa componente.
- Il secondo ciclo **for** aggiorna le componenti possibili dei nodi paralleli, dove con nodi paralleli si intendono livelli che non sono né discendenti né antenati del nodo corrente (si trovano appunto su un ramo del grafo parallelo a quello del nodo corrente): se un nodo parallelo è assegnato allo stesso server del nodo corrente, allora la componente assegnata al nodo corrente può essere anche la componente del parallelo. Questa soluzione permette di ridurre il numero totale di componenti generate dall'algoritmo e di avere uno sfruttamento migliore delle ottimizzazioni in fase di inferenza: avremo infatti componenti, quindi sotto modelli, più grandi su cui sarà più facile applicare ottimizzazioni.
- Il terzo ciclo **for** aggiorna le dipendenze dei livelli dalle componenti: una volta che un nodo è stato assegnato ad una componente, tutti i nodi discendenti, cioè tutti i nodi che si trovano dopo quel livello nel grafo e che quindi dipendono direttamente o indirettamente dal suo output, saranno dipendenti sia da questa componente, sia da tutte le componenti da cui il nodo aggiunto dipendeva a sua volta.
- Il quarto ciclo **for** serve invece a garantire che, se ci sono altre componenti dipendenti dalla componente cui il nodo è stato assegnato, allora le loro dipendenze devono essere a loro volta aggiornate con le dipendenze del livello che è stato aggiunto alla componente.

Capitolo 5

Aspetti Implementativi

In questo capitolo verranno descritti ~~alcuni~~ gli aspetti implementativi ~~più~~ più significativi ~~ed interessanti.~~

Verranno affrontati in particolare aspetti relativi al profiling del modello e dell'esecuzione.

Si procederà con una descrizione del modo in cui il piano viene prodotto e di come questo viene usato in fase di divisione del modello e di inferenza.

Si concluderà con una descrizione più dettagliata del flusso di inferenza e del modo in cui le componenti sono gestite in questa fase.

5.1 Profiling del Modello

In fase di profilazione del modello, come descritto in sottosezione 3.1.1 viene fatto il profiling del numero di FLOPS per ogni livello; questo viene fatto usando la libreria *onnx-tool* [35].

Il profilo che viene costruito per il modello non è altro che un grafo *networkx*: la libreria è molto flessibile e permette di assegnare ai nodi e agli archi vari attributi e

di accedervi in stile dizionario Python; questa caratteristica è usata per associare a nodi ed archi le informazioni dei livelli e degli archi del modello. Inoltre, astrarre il profilo da Onnx permette di rendere il sistema estensibile all'uso di altri framework.

Per motivi di semplicità di implementazione, modellazione ed estensione, in fase di profilazione del modello vengono introdotti due nodi aggiuntivi al modello, chiamati rispettivamente *Generator* e *Receiver*. Come esemplificato dai nomi, il nodo *Generator* rappresenta il nodo di input del modello, mentre il nodo *Receiver* il nodo che riceve tutti gli output del modello: da un punto di vista computazionale, potrebbero essere considerati al pari di livelli identità che restituiscono in output il loro stesso input. Da un punto di vista implementativo, l'uso di questi livelli fintizi semplifica la definizione del problema, in quanto ci sarà un unico nodo di input ed un unico nodo di output; inoltre permette di gestire in maniera più semplice la componente di FrontEnd, come descritto in sezione 5.5.

Per quanto concerne il profiling delle dimensioni dei pesi per i livelli, in Onnx l'accesso ai pesi è possibile attraverso l'attributo `initializer` del grafo Onnx. Per la raccolta delle dimensioni dei tensori intermedi, invece, si può usare la `infer_shapes` di Onnx che ricostruisce le dimensioni dei tensori intermedi dell'inferenza e salva le informazioni come attributi del grafo Onnx. Le dimensioni sono calcolate in MB.

Per quanto riguarda il profiling del rumore di quantizzazione, una considerazione importante da fare è che, nelle prove fatte, l'unico tipo di livello che è stato preso come possibile per la quantizzazione è quello convoluzionale: il motivo è che, in fase di profilazione dei FLOPS, questo tipo di livello è sempre risultato quello a carico computazionale maggiore. Non sono stati studiati gli impatti della considerazione di altri tipi di livello sulle performance predittive del regressore.

In Figura 5.1 vengono riportati i dettagli di un profilo:

- In Figura 5.1a viene mostrato l'arco tra il nodo *InputGenerator* e il nodo */model.0/conv/Conv*. Mentre nel modello originale era il livello convoluzionale a ricevere l'input, qui l'input viene passato dall'*InputReceiver*, come mostrato anche in Figura 5.1b nel primo arco.
- In Figura 5.1c viene mostrato l'arco tra l'ultimo nodo del modello, e cioè il livello di *Concat*, e il nodo di *OutputReceiver*. Mentre nel modello originale era il livello di *Concat* a fornire l'output, nella profilo del modello l'output viene restituito dal nodo *OutputReceiver*.
- In Figura 5.1d vengono mostrati alcuni dettagli del regressore del rumore di quantizzazione. Possiamo vedere in particolare i coefficienti associati alla quantizzazione di più nodi, il valore dell'intercetta e il grado, oltre agli score di train e di test.

Per quanto riguarda la costruzione del regressore del rumore di quantizzazione, il dataset considerato è *coco128*, un sottoinsieme del più grande *coco* contenente 128 immagini. In fase di valutazione del rumore di quantizzazione, il primo passo è la costruzione di un modello quantizzato secondo un certo schema di quantizzazione, ovvero per una certa combinazione di livelli; questa costruzione viene fatta seguendo i seguenti passi:

1. Viene costruito un oggetto OnnxRuntime di tipo *Calibrator*, che si occupa di raccogliere le statistiche relative al dataset di calibrazione sulla base del quale verrà fatta la quantizzazione. Il calibrator di default, usato anche da noi, è di tipo min-max. Il costruttore dell'oggetto ha come parametro opzionale anche il provider OnnxRuntime da usare: ciò permette di eseguire la calibrazione con inferenze accelerate su GPU.



Figura 5.1: Dettagli del Profilo del Modello

2. Partendo dall'oggetto calibrator, viene fatta l'inferenza su un *CalibrationDataset*, ottenendo un oggetto di tipo *TensorRange*, contenente le statistiche dei tensori; poiché nel nostro caso usiamo una quantizzazione min-max, le statistiche raccolte sono relative ai range dei tensori del modello.
3. Usando i *TensorRange* raccolti dal calibrator viene costruito un oggetto di tipo *QDQQuantizer*, che quantizza il modello in formato QDQ. Partendo da questo oggetto, tramite il metodo `quantize`, viene costruita una versione quantizzata del modello, specificando quali sono i livelli da quantizzare e gli altri parametri di quantizzazione.

Questa implementazione della quantizzazione è ad un livello più basso rispetto all'API standard offerta da OnnxRuntime. La libreria, infatti, offre un'API diretta di `quantize_model` che però esegue tutte le volte le inferenze per calcolare i tensor range; poiché il calcolo dei range è fatto a monte della quantizzazione effettiva, questo risulta **indipendente** dai livelli che stiamo quantizzando. Di conseguenza, dovendo valutare il rumore di quantizzazione per molte configurazioni, è bene calcolare i range una sola volta, per poi riusarli al variare dei livelli che stiamo quantizzando.

Una volta quantizzato il modello secondo un certo schema (i.e. per certi livelli da quantizzare), deve essere valutato il rumore per lo schema usato; questa valutazione viene fatta su un *NoiseDataset*. Il confronto viene fatto considerando il modello non quantizzato come baseline e, poiché il modello originale non cambia, l'inferenza dell'originale sul *NoiseDataset* viene eseguita una sola volta e il risultato riutilizzato per il confronto con il risultato dell'inferenza di tutte le versioni quantizzate.

Poiché dobbiamo costruire un dataset di addestramento per il regressore che tenga in considerazione diversi schemi di quantizzazione, abbiamo bisogno **di più valori** di

valori del rumore per più schemi di quantizzazione. Questi schemi vengono costruiti in modo casuale e senza ripetizione.

La configurazione per la costruzione del regressore di quantizzazione è stata fissata alla seguente:

- Massimo numero di livelli quantizzabili: 12;
- Tipo di calibrazione: min-max;
- Dimensione *CalibrationDataset* per quantizzazione modello: 100;
 - Estratto casualmente da *coco128*.
- Dimensione *NoiseDataset* del rumore: 20;
 - Estratto casualmente da *coco128*, ma senza sovrapposizione con il *CalibrationDataset*.
- Dimensione dataset addestramento per il regressore: 1000;
- Dimensione dataset testing per il regressore: 100.

5.2 Profiling dell'Esecuzione

In fase di profilazione del modello, come spiegato in sottosezione 3.1.2, viene stimato il tempo di esecuzione di ogni livello del modello.

Un primo aspetto che è importante evidenziare è che la somma dei tempi medi di esecuzione di ogni livello potrebbe non dare come risultato il tempo medio di esecuzione del modello complessivo. In primo luogo abbiamo un tempo di start-up dell'inferenza dei sottomodelli dei livelli che può impattare negativamente il tempo stimato per l'esecuzione di quel livello: al fine di evitare un'influenza eccessiva del

tempo di cold-start, la prime 10 run vengono fatte senza considerare il loro tempo di inferenza. Il secondo motivo è che in fase di inferenza del modello completo, come indicato anche in sottosezione 2.2.3, il provider sottostante può mettere in atto delle ottimizzazioni avanzate (i.e. fusioni, parallelismi, ecc.) che questa modellazione non prende in considerazione. Questo aspetto e il modo di affrontarlo saranno ulteriormente approfonditi in sottosezione 6.3.1.

Per ciò che riguarda il profiling del tempo di esecuzione del layer, come spiegato in sottosezione 3.1.2, questa viene fatta per ogni livello e, per ognuno, viene valutato sia il caso in cui il livello è quantizzato sia quello in cui non lo è. Un aspetto importante da sottolineare è il fatto che, considerando noi soltanto un piccolo sottoinsieme di livelli ai fini della quantizzazione, il profiling viene fatto solo su questi livelli e non su tutti: questo permette un profiling più veloce rispetto. Inoltre, come giustificato in sottosezione 6.3.2 viene calcolato anche il tempo medio di esecuzione del modello intero non quantizzato e del modello in precisione mista; quest'ultimo in particolare è il modello in cui tutti i livelli quantizzabili sono effettivamente quantizzati.

Un altro aspetto da considerare è il punto in cui viene fatto il taglio per il livello quantizzato. Ricordando che nella rappresentazione QDQ di OnnxRuntime vengono aggiunti dei livelli di `QuantizeLinear` e `DequantizeLinear` a circondare il livello, il livello dopo la quantizzazione è rappresentabile come in Figura 5.2a. Poiché noi stiamo considerando un livello quantizzato *INT8*, è lecito assumere che il livello quantizzato dovrà ricevere e dare in output dei tensori di tipo *INT8*, e questo avviene solo se il taglio viene fatto dopo i tensori di `QuantizeLinear`, come mostrato in Figura 5.2b. Questo permette anche di garantire l'invio di una quantità di dati minore nel momento in cui il taglio del modello viene fatto, appunto, dopo un livello quantizzato, cosa che, come vedremo, succede spesso.

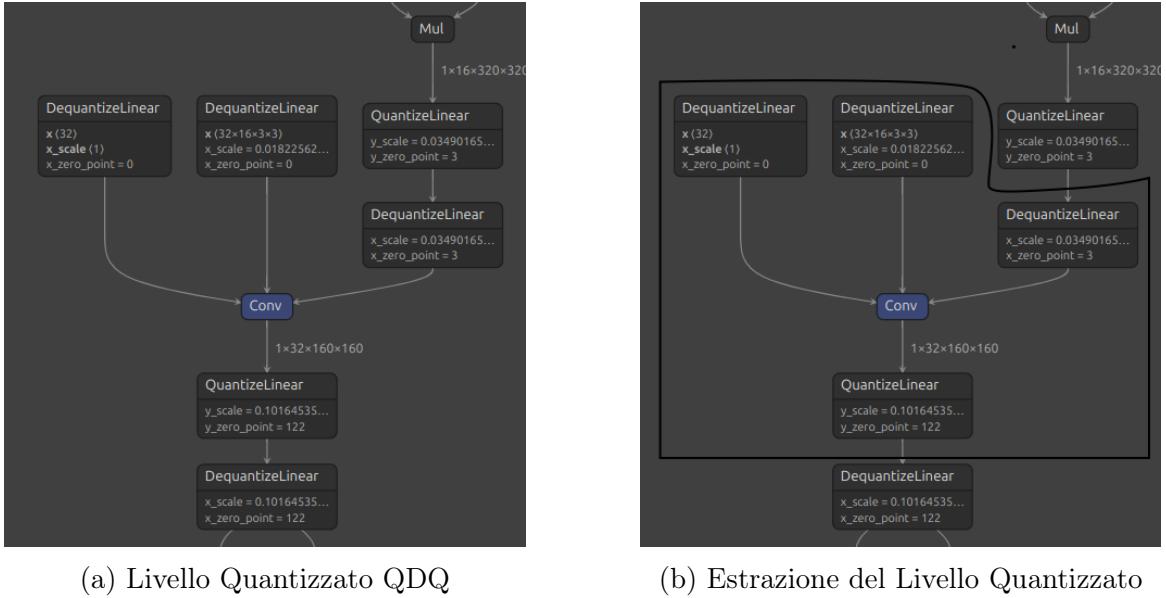


Figura 5.2: Livello Quantizzato QDQ e sua Estrazione

Di seguito descriviamo i dettagli dei profili di esecuzione:

- In Figura 5.3a sono mostrati i tempi profilati per tre layer; per ogni tempo viene calcolato sia il tempo medio di esecuzione sia la sua mediana. Si noti come nel caso del livello quantizzabile vi siano quattro valori, due per ogni versione del livello.
- In Figura 5.3b sono mostrati i tempi profilati per la versione completa del modello e la versione mista.

5.3 Produzione del Piano

Una volta che il problema è stato risolto, viene prodotto un nuovo grafo, detto *Solved-Graph*, che contiene informazioni prese in parte dal profilo del modello e in parte dalla soluzione del problema. Il *SolvedGraph* contiene gli stessi nodi ed archi del profilo del modello, ma con informazioni associate diverse; in particolare:



Figura 5.3: Esempi di profili di esecuzione

- Le informazioni associate al nodo (i.e. livello del modello) sono:
 - L'id del server di rete a cui il livello è stato assegnato;
 - Due booleani per indicare se si tratta del nodo generatore o del nodo ricevente;
 - L'id della componente a cui il nodo verrà assegnato in fase di post elaborazione;
 - Un booleano per indicare se è stato scelto di quantizzare il livello;
- Le informazioni associate all'arco sono:
 - I nomi dei tensori che vengono trasmessi tramite quell'arco del modello.

Prodotto il *SolvedGraph*, questo viene elaborato usando un algoritmo simile a quello descritto in Algoritmo 1, assegnando un valore valido all'attributo *component* del nodo del *SolvedGraph*. Come anticipato in precedenza e come verrà chiarito in sezione 5.5, il nodo generatore e il nodo ricevente si trovano ognuno in una componente propria, separati da altri nodi.

Partendo dalle assegnazioni dei nodi alle componenti e dai nomi dei tensori trasmessi con gli archi, è possibile costruire il piano di deployment, che altro non è che una rappresentazione del grafo delle componenti con l'aggiunta di alcune informazioni. In Figura 5.4, è possibile vedere un esempio di piano minimale (in cui non sono applicate divisioni del modello) le informazioni salienti necessarie per la definizione del piano sono le seguenti:

- Per ogni componente:
 - I nomi dei tensori di input;
 - I nomi dei tensori di output;
 - Il numero di livelli quantizzati in quella componente;
- Per ogni tensore di output, le componenti successive a cui devono essere inviati.
- Sono aggiunti anche i risultati di ottimizzazione del modello; i valori di *cost* si riferisce ai valori di tempo ed energia normalizzati, mentre i valori di *value* si riferiscono ai valori di tempo ed energia denormalizzati.

Conoscere queste informazioni permette l'estrazione del sotto modello tramite le API di Onnx. Si noti che le componenti con in gestione generatore e ricevente sono marcate come tali, in modo da riconoscerle più semplicemente e che la componente che contiene il solo nodo ricevente non ha componenti successive, trattandosi della componente di output.

Un'altra informazione significativa che viene allegata al piano è l'elenco dei livelli per cui è stata attivata la quantizzazione: questa informazione sarà importante in fase di divisione del modello e generazione dei sotto modelli corrispondenti alle componenti, come descritto in sezione 5.4.

```

"model_name": "yolol1x-seg",
"plan_dict": {
    "('yolol1x-seg', '0', 0)": {
        "component_size": 1,
        "quantized_nodes_num": 0,
        "is_only_input": true,
        "is_only_output": false,
        "input_names": [
            "images"
        ],
        "output_connections": {
            "images": [
                "('yolol1x-seg', '0', 1)"
            ]
        }
    },
    "('yolol1x-seg', '0', 1)": {
        "component_size": 652,
        "quantized_nodes_num": 9,
        "is_only_input": false,
        "is_only_output": false,
        "input_names": [
            "images"
        ],
        "output_connections": {
            "output1": [
                "('yolol1x-seg', '0', 2)"
            ],
            "output0": [
                "('yolol1x-seg', '0', 2)"
            ]
        }
    },
    "('yolol1x-seg', '0', 2)": {
        "component_size": 1,
        "quantized_nodes_num": 0,
        "is_only_input": false,
        "is_only_output": true,
        "input_names": [
            "output0",
            "output1"
        ],
        "output_connections": {
3198
4200
4201
4202
4203
4204
4205
4206
4207
4208
4209
4210
4211
4212
4213
4214
4215
4216
4217
4218
4219
4220
4221
4222
4223
4224
4225
4226
4227
4228
4229
4230
4231
4232
4233
4234
4235
4236
4237
4238
4239
4240
4241
            "input_names": [
            ],
            "output_connections": [
                "output1": [
                    "('yolol1x-seg', '0', 2)"
                ],
                "output0": [
                    "('yolol1x-seg', '0', 2)"
                ]
            ],
            "quantized_nodes": [
                "/model.3/conv/Conv",
                "/model.4/cv2/conv/Conv",
                "/model.5/conv/Conv",
                "/model.16/cv1/conv/Conv",
                "/model.17/conv/Conv",
                "/model.23/proto/cv1/conv/Conv",
                "/model.23/cv4.0/cv4.0.0/conv/Conv",
                "/model.23/cv2.0/cv2.0.0/conv/Conv",
                "/model.23/proto/cv2/conv/Conv"
            ],
            "latency_value": 2.898525651587422,
            "energy_value": 8.453550062854715,
            "device_energy": 8.453550062854712,
            "latency_cost": 0.9999999988130297,
            "energy_cost": 3.346112449620353e-13
        }
    }
}

```

Figura 5.4: Esempio minimale di piano

5.4 Divisione del Modello

In fase di divisione del modello per il deployment del piano, viene eseguita in primo luogo la quantizzazione seguendo lo schema deciso in fase di ottimizzazione: i nomi dei livelli che si devono quantizzare possono essere facilmente estratti dal piano, come visibile in Figura 5.4. Se non ci sono livelli da quantizzare, chiaramente questa fase viene saltata.

Al momento della divisione vera e propria, viene usata l'API `extract_model` offerta da Onnx; questa API chiede che siano dati come parametri, oltre al modello iniziale, i nomi dei tensori di input e i nomi dei tensori di output del sotto modello che si vuole estrarre. Anche in questo caso, l'informazione è estraibile facilmente dal piano. Chiaramente, la componente di input e quella di output non saranno estratte, in quanto non c'è nessun nodo del modello reale che sia assegnato loro.

5.4.1 Configurazione di Quantizzazione

Come introdotto in sezione 1.2, ci sono molti modi in cui la quantizzazione può essere applicata al modello; inoltre, il numero di possibili configurazioni di quantizzazione è estremamente grande, come descritto in sottosottosezione 3.1.1.1. Non potendo esplorare tutte le possibili varianti, ci si è concentrati sulla configurazione di quantizzazione seguente:

- Metrica per stabilire i livelli quantizzabili: Numero FLOPS;
- Numero massimo di livelli quantizzabili: 12;
- Quantizzazione per canale: No;
- Precisione di quantizzazione pesi: *INT8*;

- Precisione di quantizzazione attivazioni: *INT8*;
- Simmetria pesi: Sì;
- Simmetria attivazioni: Sì.

Questa configurazione è stata usata nello stesso modo sia in fase di profilazione che in fase di divisione del modello.

La quantizzazione simmetrica fa sì che, in fase di calcolo dei parametri dei livelli di `QuantizeLinear` e `DequantizeLinear`, il valore `zero_point` venga messo a 0; questo permette di rendere più efficiente il calcolo per i livelli quantizzati in alcuni provider, come quello per *OpenVINO* e *Cuda*.

5.5 FrontEnd ed Inferenza

Come accennato in sottosezione 3.2.4, vi sono due componenti che intervengono in fase di inferenza, ovvero il *FrontEnd* e *Inferencer*. Quando il piano viene ricevuto, il server lo analizza e, a seconda del tipo di componente, può seguire due comportamenti:

- Se si tratta di una componente reale, ovvero di un sotto modello del modello originale, la componente viene recuperata accedendo al ModelPool e viene attivata la gestione di questo sotto modello da parte dell'*Inferencer*.
- Se si tratta della componente di input o di output, e quindi non c'è un sotto modello associato, questa viene gestita dal *FrontEnd*.

Quando riceve un input per la componente di input, il *FrontEnd* procede nel seguente modo:

1. Verifica che tutti gli input siano stati ricevuti. Questo permette di gestire casi di modelli con più tensori di input.

2. Quando gli input sono tutti, invia gli input alle componenti successive seguendo il piano. Inviare gli input alle componenti successive equivale ad inviarlo agli *Inferencer* che hanno in gestione queste componenti.
3. Blocca il thread da cui la richiesta è stata ricevuta.

L'*Inferencer*, invece, quando ricevuto un input procede nel seguente modo:

1. Verifica che tutti gli input per la componente corrente siano stati ricevuti.
2. Se ricevuti tutti gli input della componente corrente, allora viene eseguita l'inferenza per la componente.
3. Invia l'output della componente alle componenti successive. L'invio dell'output alle componenti successive è fatto usando un thread separato: questo permette di evitare il blocco del thread che ha gestito la richiesta che sarebbe altrimenti causato dalla sincronia di gRPC.

Quando riceve un input per la componente di output, il *FrontEnd* procede nel seguente modo:

1. Verifica che gli output ricevuti siano tutti quelli del modello;
2. Se ricevuti tutti gli output del modello, allora il risultato viene salvato in una struttura condivisa a cui il thread originale che ha gestito la richiesta può accedere.
3. Il thread originale viene sbloccato; a questo punto può leggere il risultato nella struttura condivisa e ritornarlo al client.

Di fatto quindi *FrontEnd* ed *Inferencer* hanno comportamenti molto simili, ma la separazione delle componenti permette di gestire in modo più semplice e flessibile la ricezione dell'input e il ritorno dell'output.

Quando viene ricevuta un sotto modello reale, OnnxRuntime richiede la creazione di una *InferenceSession*, ovvero di un oggetto che astrae l'inferenza del modello per l'architettura sottostante. Uno dei parametri fondamentali della *InferenceSession* è il *Provider* di esecuzione, che ci permette di interfacciarsi con l'architettura sottostante per ottimizzare l'inferenza in base alle sue capacità di calcolo. Prima di creare una sessione quindi si verifica quali provider sono disponibili; nello specifico:

1. Si verifica se è disponibile un provider per *CUDA* che permetta di accelerare l'inferenza su GPU; se non disponibile allora
2. Si verifica se è disponibile un provider per *OpenVINO* che permetta di accelerare l'inferenza su architetture *Intel*; se non disponibile allora
3. Si utilizza il *CPUExecutionProvider*, ovvero il provider di default.

Stiamo quindi considerando solo tre possibili casi dei vari che OnnxRuntime offre.

Un'altra considerazione importante da fare è relativa all'invio dei tensori. Poiché stiamo usando gRPC, il quale permette una dimensione massima del messaggio di 4 MB, e poiché non conosciamo la dimensione massima del tensore nel modello, i chunk vengono divisi in parti quando inviati tra le componenti. Di fatto la dimensione massima del modello potrebbe anche essere calcolata analizzando il modello, per poi modificare i parametri del canale gRPC per permettere la trasmissione di messaggi più grandi; tuttavia, l'uso di messaggi troppo grandi è sconsigliato in gRPC per motivi di efficienza. Nella trasmissione del (chunk) tensore tra le componenti, vengono scambiate le seguenti informazioni:

- Una tupla contenente le dimensioni del tensore;
- Il tipo di dato del tensore;
- Bytes del (chunk) tensore.

Si noti che la conoscenza delle prime due informazioni permette l'allocazione di un buffer di memoria lato ricevente in cui inserire il tensore può essere salvato pezzo per pezzo durante la ricezione.

Per concludere, una considerazione importante deve essere fatta rispetto all'uso dei canali gRPC e in generale di TCP. Quando usiamo i canali gRPC su TCP, le connessioni verso gli altri Inferencer vengono aperte una sola volta e riutilizzate di volta in volta. Tuttavia, poiché le inferenze possono richiedere molto tempo (anche secondi) è possibile che si verifichi il cosiddetto *slow start after idle* di TCP, per il quale, se il canale TCP rimane inutilizzato per troppo tempo, si riparte da una finestra di trasmissione piccola: questo è descritto, ad esempio, in [36]. Questo, chiaramente, può danneggiare fortemente le performance del sistema perché, a livello di prestazione di trasmissione, equivarrebbe a riaprire tutte le volte la connessione. Per ovviare a questo problema, vengono periodicamente inviati dei messaggi sui canali tra inference, in modo che il canale non sia inutilizzato per un periodo di tempo tale da portare al restart.

Capitolo 6

Esperimenti e Risultati

In questo capitolo verranno riportati i risultati sperimentali ottenuti usando il sistema.

Si partirà con una descrizione del set-up degli esperimenti e del modo in cui il sistema è stato deployato, così come delle configurazioni considerate.

A seguire, vi sarà una descrizione del modo in cui è stato affrontato il rumore introdotto dalla misurazione dei tempi medi di esecuzione per livello.

Si continuerà con un esperimento relativo all'accuratezza della predizione del modello costruito, per poi passare ad una valutazione del sistema, confrontando il beneficio da esso portato rispetto al caso di base, sia in termini di tempo di inferenza sia in termini energetici.

Si analizzerà poi il modo i cui i livelli del modello originale sono distribuiti tra i diversi server e come la computazione del modello è stata divisa.

6.1 Profiling del Modello

In questa sottosezione vengono mostrati brevemente i tempi di profiling del modello per tre varianti di task e dimensioni del modello *yolo11*. Il profiling è stato eseguito nelle seguenti condizioni:

- GPU Nvidia Tesla T4; **altre info su tipo istanza usata, CPU etc**
- Condizioni di quantizzazione: riportate in sezione 5.1 e sottosezione 5.4.1.

In Tabella 6.1 vengono mostrati questi tempi. Come prevedibile, al crescere della complessità del modello cresce anche il tempo di profilazione: questo incremento è da imputarsi principalmente al tempo di costruzione del dataset per l'addestramento del regressore usato per la predizione del rumore indotto dalla quantizzazione. Sebbene

	yolo11n-cls	yolo11m-det	yolo11x-seg
Profile Time [s]	246.93	1380.23	3677.14

Tabella 6.1: Tempi di profiling per modello

il tempo di profiling cresca all'aumentare della complessità del modello, è importante tenere in considerazione il fatto che si tratta di un costo pagato una sola volta: una volta che il modello è stato caricato e il profiling eseguito, posto che le condizioni di quantizzazione rimangano le stesse, questo può essere salvato e riutilizzato.

6.2 Set-Up degli Esperimenti

A meno che non diversamente specificato, il setup usato per gli esperimenti è da considerarsi quello specificato in questa sezione.

Gli esperimenti sono stati eseguiti su macchine virtuali messe a disposizione da **emulare** Google Cloud Provider. Per simulare diverse bande dei dispositivi e diverse latenze

tra questi, è stato usato lo strumento `tc` di Linux, mentre per simulare la diversa capacità di calcolo è stato usato *Docker* combinato con la `-cpu-affinity`.

Le macchine considerate e le loro configurazioni in termini di capacità di calcolo sono riportate in Tabella 6.2. Nella tabella, il valore di `-cpu-affinity` indica il numero di CPUs a cui il container ha affinità impostata a prescindere da quale CPU si tratti. Tanto i tipi di macchine quanto i valori per le affinity sono stati scelti al fine di emulare un contesto di esecuzione reale: come si vedrà nei confronti successivi, la macchina scelta per il device ha delle prestazioni inferiori rispetto a quella scelta per l'edge anche a parità di numero di CPUs affini. Il motivo della scelta dell'uso di `-cpu-affinity` viene chiarito in sottosezione 6.2.1.

	Tipo di Macchina	-cpus-affinity	GPU
Device	e2-standard-4	1	-
Edge	c3-standard-4	1	-
Cloud	n1-standard-4	-	Nvidia Tesla T4

Tabella 6.2: Tipi di Macchine e Configurazioni di Calcolo

Per quanto riguarda le configurazioni di rete, le configurazioni di banda massima di trasmissione e latenza vengono riportate rispettivamente in Tabella 6.3 e Tabella 6.4. In generale, si è assunto che il device operasse con una connessione 4G, l'edge con una connessione Wi-Fi e il cloud con una connessione ethernet. Per quanto

	Banda Massima MB/s
Device	5 MB/s
Edge	20 MB/s
Cloud	100 MB/s

Tabella 6.3: Banda Massima di Trasmissione per Dispositivo in MB/s

riguarda i valori della banda, è opportuno specificare che i valori riportati sono, come indicato, i valori massimi della banda di trasmissione configurati tramite `tc`; tuttavia,

	Device	Edge	Cloud
Device	-	5 ms	55 ms
Edge	5 ms	-	50 ms
Cloud	55 ms	50 ms	-

Tabella 6.4: Latenza tra dispositivi in ms

questi valori possono differire rispetto a quelli che verranno effettivamente misurati in fase di esecuzione del sistema: infatti, in fase di monitoring, viene valutato il valore del throughput TCP che, a seconda del valore della latenza, potrà essere anche significativamente più basso rispetto a quello specificato in fase di configurazione.

Le configurazioni energetiche vengono riportate in Tabella 6.5. Queste configurazioni sono state scelte in base a diversi parametri:

- Il consumo energetico dovuto alla trasmissione di un dispositivo verso se stesso è stato considerato trascurabile rispetto agli altri consumi, quindi impostato a zero;
- Il consumo energetico relativo al calcolo è stato valutato considerando le specifiche tecniche dei dispositivi:
 - Nel caso del Cloud, il suo consumo è stato considerato quello dell'esecuzione in GPU. Di fatto la GPU montata dal cloud ha un consumo massimo di 70 W; tuttavia, considerando che l'esecuzione del modello potrebbe non implicare un carico massimo sul device, è stato considerato un valore medio di 35 W.
 - Nel caso dell'Edge, il consumo della CPU montata è disponibile, quindi è stato usato quello relativo alla singola CPU.
 - Nel caso del Device, poiché queste macchine vengono virtualizzate su architetture diverse, non è stato possibile fare riferimento ad una specifica

tecnica particolare; è stato quindi scelto un consumo pari alla metà di quello del nodo Edge.

	Calcolo [W]	Self-Tx [W]	Other-Tx [W]
Device	2.9165	0	3.507
Edge	5.833	0	2.265
Cloud	35	0	0.014

Tabella 6.5: Potenze misurate in W

Poiché, purtroppo, Google Cloud Provider non fornisce delle API per la misura del consumo energetico su una macchina virtuale, il confronto tra i valori predetti e i valori reali per il consumo energetico non sarà possibile nelle sezioni successive.

6.2.1 Uso dell'affinity

L'uso del `-cpu-affinity` di Docker è stato preferito all'uso del `-cpus`. Il motivo è dovuto a un'interferenza introdotta da Docker in fase di profilazione del modello e che viene mostrata in Figura 6.1: come si vede nella sezione sottolineata nell'immagine, nella misurazione dei tempi per il livello, all'incirca ogni due tempi vicini a 0.091 ve ne è uno di 0.019. Il motivo di questo comportamento è dovuto al modo in cui Docker regola l'uso delle CPUs quando viene usato il `-cpus`: viene assegnato un certo tempo su tutte le CPUs della macchina proporzionale al valore del parametro. Quando il livello è computazionalmente più oneroso quindi, la limitazione sul tempo di esecuzione viene cattura meglio, perché sarà più probabile che il calcolo del livello vada oltre il tempo assegnato e che il calcolo venga messo in blocco per rispettare l'assegnazione del limite. Quando il livello, invece, non è computazionalmente oneroso, quindi è, ad esempio, un'operazione in memoria oppure un livello quantizzato, il calcolo è molto più veloce e il limite di tempo non viene catturato dalla profilazione.

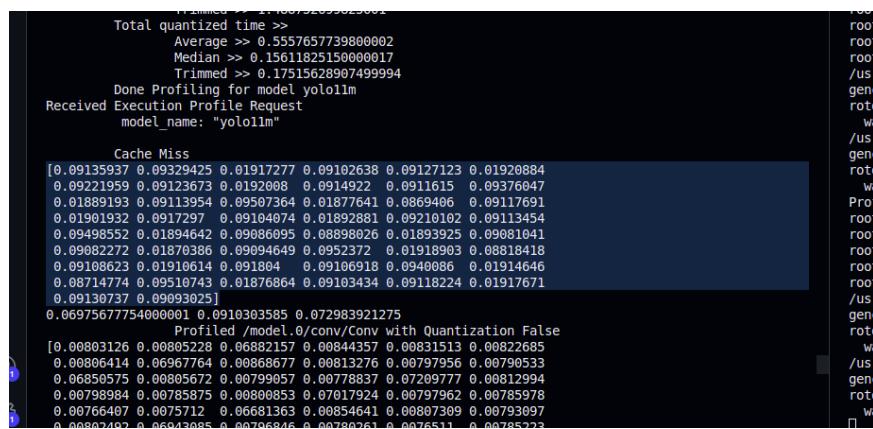


Figura 6.1: Interferenza -cpus

Quindi, al fine di garantire un profiling accurato e in cui non vi fossero interferenze aggiuntive dovute all'uso di Docker si è scelto di usare il `-cpu-affinity`. In questo caso il container potrà eseguire solo sulle CPUs per le quali è stata impostata affinità, ma senza limiti temporali: ciò permette di evitare l'introduzione di una correlazione simile a quella che vediamo nell'immagine, perché per ogni misurazione si avrà accesso alla risorsa in modo completo. Di contro, l'uso della cpu affinity riduce il grado di libertà nella sperimentazione, perché ci obbliga a discretizzare la quantità di risorse che il dispositivo device ed edge hanno a disposizione; motivo per il quale sono stati scelti hardware diversi per le due macchine.

6.3 Accuratezza della Predizione

6.3.1 Differenza tra tempo di inferenza del modello e somma dei tempi dei livelli

Come spiegato in precedenza, il profiling del modello su un server viene fatto per livello, viene fatto calcolando il tempo di esecuzione del sotto modello contenente il singolo livello. Questa approssimazione, purtroppo, porta ad una perdita di accuratezza nelle misure delle prestazioni in confronto all'esecuzione del modello complessivo; questa

perdita è data principalmente da due fattori:

- Non vengono tenute in conto eventuali fusioni di livelli che vengono fatte a basso livello per accelerare l'inferenza;
- Non vengono tenuti in conto gli overhead aggiuntivi dati dalla misura livello per livello; nello specifico, quando viene fatto partire un modello abbiamo un tempo aggiuntivo per far partire l'esecuzione del livello. Nel momento in cui eseguiamo livello per livello questo overhead viene tenuto in considerazione più volte.

Questo comportamento è mostrato in Figura 6.2 e Tabella 6.6, in cui possiamo vedere come come questo errore sia particolarmente alto nel nodo device e nel nodo edge, mentre è praticamente nullo su cloud con esecuzione GPU. Questa differenza tra esecuzione su CPU ed esecuzione su GPU può essere dovuta ai seguenti aspetti:

- In primo luogo, quando misurato il tempo di inferenza per livello su GPU, il trasferimento dei dati non viene sulla e dalla GPU non viene incluso; poiché vogliamo misurare il tempo di esecuzione del livello, includere questo tempo aumenterebbe il rumore delle misure.
- In secondo luogo, il provider OpenVINO non permette una misura del tempo di esecuzione che non includa l'overhead aggiuntivo dovuto al provider stesso: mentre con CUDA è possibile fare il trasferimento dei tensori in anticipo, permettendo la riduzione dell'overhead, in OpenVINO questa cosa non è possibile, portando a peggiorare la qualità delle misure.
- In conclusione, il provider OpenVINO attua delle ottimizzazioni aggiuntive a quelle eseguite a priori da Onnx e, anche in questo caso, non è possibile stabilire l'effetto di queste ottimizzazioni sul tempo di inferenza totale.

in italiano. Esporta in svg/PDF/eps per migliore risoluzione

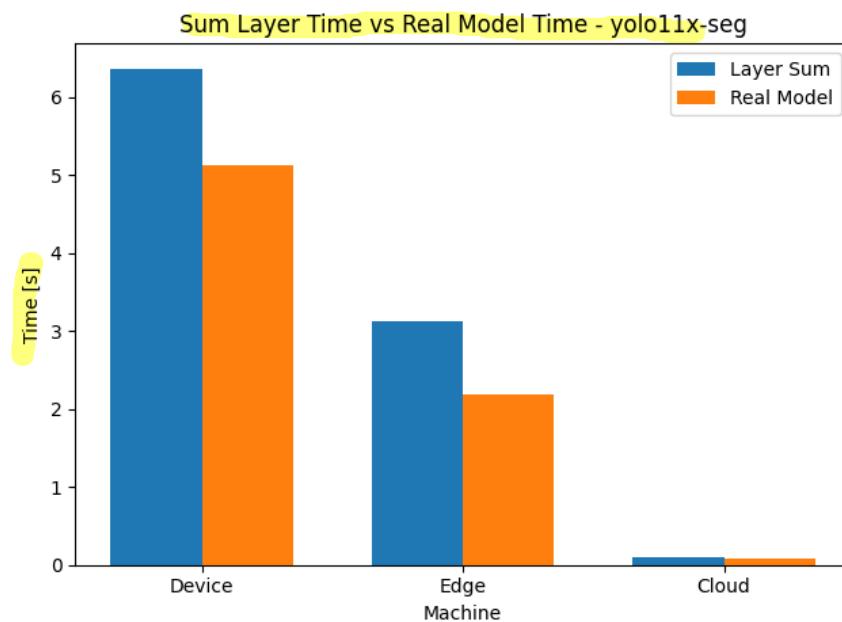


Figura 6.2: Differenza tra somma dei tempi per livello e tempo del modello complessivo

Machine	Sum	Real	Difference
Device	6.3715	5.1313	1.2402
Edge	3.1195	2.1930	0.9265
Cloud	0.1031	0.0760	0.0271

Tabella 6.6: Differenza tra somma dei tempi per livello e tempo del modello complessivo

Risulta interessante notare anche come in Tabella 6.6 le differenze tra somma e reale per il device e l'edge siano, anche se non uguali, vicine: questo è perché a parità di modello, e quindi di livelli, e di provider, cioè OpenVINO, l'overhead introdotto sull'inferenza del singolo livello sarà simile, al netto, ovviamente, delle differenze introdotte dagli hardware differenti.

Chiaramente, una discrepanza eccessiva tra il valore reale del tempo di esecuzione e il valore predetto in fase di ottimizzazione può essere deleteria ai fini sperimentali: se il tempo predetto come somma dei tempi per layer per l'esecuzione su solo device è troppo inaccurato, si potrebbero prendere delle decisioni non ottimali; ad esempio, si potrebbe decidere di fare l'offloading dell'intero carico computazionale a prescindere. Ciò rende necessaria una correzione delle misure, spiegata in sottosezione 6.3.2

6.3.2 Interpolazione delle Misure

Per far fronte a questi overhead aggiuntivi, è stata implementata un'interpolazione dei valori misurati livello per livello con il tempo di esecuzione del modello complessivo.

Nello specifico, sono state applicate le seguenti correzioni. Detto $\hat{f}_k(i, 0)$ il tempo di esecuzione del sotto modello contenente il solo livello i non quantizzato sul server k dopo la correzione il suo valore è il seguente:

$$\hat{f}_k(i, 0) = f_k(i, 0) \cdot \frac{T_k^{whole}}{\sum_{j \in V_D} f_k(j, 0)}$$

Dove:

- $f_k(i, 0)$ è il tempo di esecuzione misurato sul server per il sotto modello contenente il solo livello i non quantizzato;
- T_k^{whole} è il tempo di esecuzione del modello complessivo non quantizzato misurato sul server.

Allo stesso modo, detto $\hat{g}_k(i)$ il guadagno di quantizzazione del sotto modello contenente il solo livello i sul server k , il suo valore è il seguente:

$$\hat{g}_k(i) = (f_k(i, 0) - f_k(i, 1)) \cdot \frac{T_k^{whole} - T_k^{mixed}}{\sum_{j \in V_Q} (f_k(j, 0) - f_k(j, 1))}$$

Dove:

- $f_k(i, 1)$ è il tempo di esecuzione misurato sul server per il sotto modello contenente il solo livello i quantizzato;
- T_k^{mixed} è il tempo di esecuzione del modello complessivo sul server quando tutti i livelli quantizzabili sono stati quantizzati.

Validazione del Modello

6.3.3 Confronto: Valore Reale vs Valore Predetto

6.3.3.1 Solo Device

In questo esperimento è stato considerato quanto segue:

- Modello yolo11x-seg;
- Rumore di quantizzazione crescente;
- In fase di ottimizzazione, i pesi considerati sono stati 1 per la latenza e 0 per l'energia: avendo infatti un unico dispositivo e considerando la formulazione del problema di ottimizzazione, analizzare più configurazioni dei pesi risulta superfluo.

In Figura 6.3 e nello specifico nel primo grafico, viene riportato il confronto tra il valore della latenza ottenuto in fase di ottimizzazione e il valore ottenuto in fase sperimentale. Come la figura rende chiaro, il valore ottenuto risulta molto vicino a quello predetto in fase di ottimizzazione: possibili discrepanze in questo caso possono essere

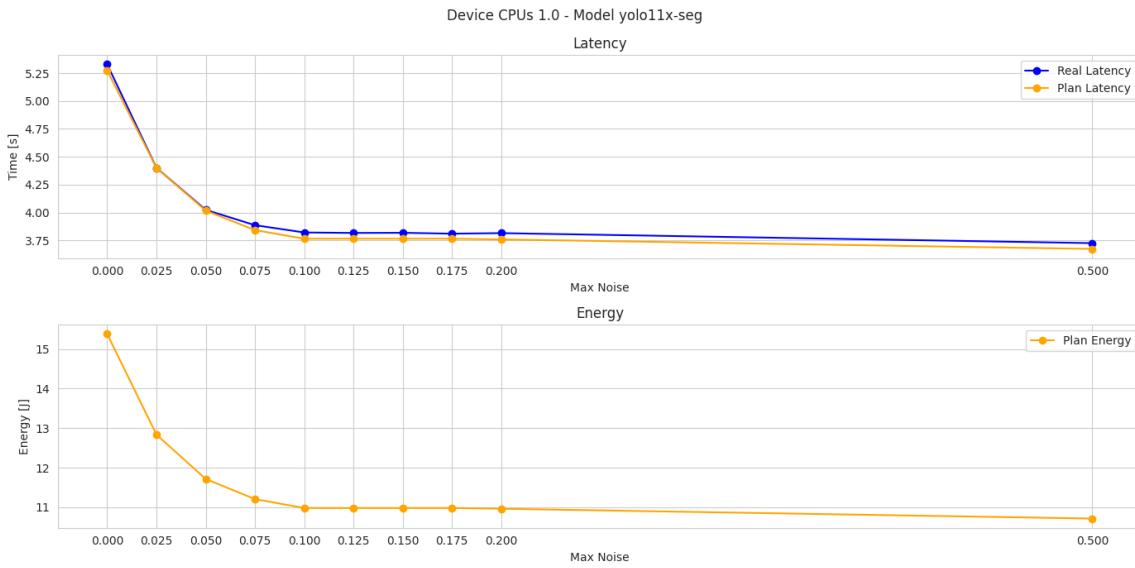


Figura 6.3: Solo Device: Reale vs Predetto

imputate ad overhead aggiuntivi non considerati in fase di ottimizzazione. Partendo da Figura 6.3 è già possibile apprezzare quanto la quantizzazione porti a un beneficio delle prestazioni in termini di tempo di inferenza. In Tabella 6.7 è possibile vedere i valori a confronto all'aumentare del rumore di quantizzazione. La tabella mostra chiaramente come la differenza tra il valore reale e quello predetto sia piuttosto vicina, con una differenza circa di 60 ms nella predizione. Inoltre, è possibile vedere come il tempo di inferenza decresca da circa 5.32 s fino a 3.72 s all'aumentare del rumore di quantizzazione, permettendo un risparmio medio di 2 s. Questi risultati saranno comunque oggetto di discussione nelle sezioni successive.

6.3.3.2 Device + Edge

In questo esperimento è stato considerato quanto segue:

- Modello yolo11x-seg;
- Rumore di quantizzazione crescente;

Max Noise	Run Time	Plan Latency	Latency Diff
0.0000	5.3285	5.2728	0.0558
0.0250	4.4011	4.4010	0.0001
0.0500	4.0243	4.0169	0.0074
0.0750	3.8866	3.8426	0.0440
0.1000	3.8214	3.7653	0.0560
0.1250	3.8171	3.7653	0.0518
0.1500	3.8184	3.7653	0.0531
0.1750	3.8105	3.7653	0.0451
0.2000	3.8154	3.7590	0.0565
0.5000	3.7257	3.6738	0.0519

Tabella 6.7: Solo Device: Reale vs Predetto

- Pesi di latenza ed energia variabili nell’insieme $\{0.0, 0.5, 1.0\}$, con somma dei pesi ad 1.0.

In Figura 6.4 nella prima riga è possibile vedere il confronto grafico tra i valori predetti e quelli reali al crescere del peso della latenza nella risoluzione del problema di ottimizzazione. Anche in questo caso si può vedere come vi sia una buona aderenza tra i valori teorici e i valori reali della latenza, ad eccezione di alcuni picchi che si presentano nei casi di peso della latenza pari a 0.0 e 0.5 quando il rumore di quantizzazione è pari a 0.15: è possibile che in questo caso vi sia una combinazione di quantizzazione che l’interpolazione del guadagno non riesce a catturare completamente.

Consideriamo i casi di $w_L = 1$ e $w_L = 0$ (e i rispettivi valori di w_E). In questi casi è possibile notare come la predizione sia effettivamente coerente: gli andamenti delle predizioni per la latenza, nel primo caso, e per l’energia, nel secondo, sono non crescenti; chiaramente, all’aumentare del rumore di quantizzazione, se non si trova una configurazione di quantizzazione migliore si rimane su quella del caso precedente.

In Tabella 6.8 sono riportati i valori medi reali, predetti e la loro differenza in valore assoluto nel caso di $w_L = 0$. Si vede qui di quanto è mediamente errata

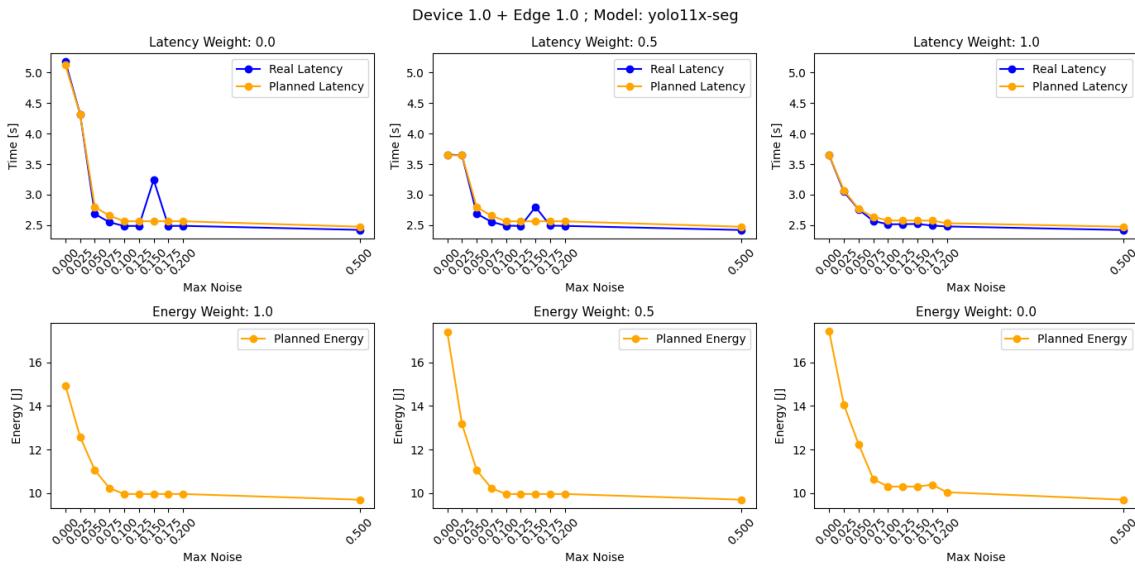


Figura 6.4: Device ed Edge: Reale vs Predetto - Grafico

la predizione: nella maggior parte dei casi la differenza è inferiore ai 100 ms, fatta eccezione per il caso in cui il rumore di quantizzazione è impostato a 0.15, in cui la differenza raggiunge i 600 ms.

Max Noise	Run Time	Plan Latency	Latency Diff	in italiano
0.0000	5.1731	5.1218	0.0513	la differenza sarebbe forse meglio riportarla in % rispetto a quella reale
0.0250	4.3179	4.3138	0.0041	
0.0500	2.6833	2.7960	0.1127	
0.0750	2.5472	2.6550	0.1078	
0.1000	2.4862	2.5639	0.0778	
0.1250	2.4877	2.5639	0.0763	
0.1500	3.2433	2.5639	0.6794	
0.1750	2.4870	2.5639	0.0769	
0.2000	2.4886	2.5639	0.0753	
0.5000	2.4229	2.4749	0.0520	

Tabella 6.8: Device ed Edge: Reale vs Predetto - $w_L = 0$

6.3.3.3 Device + Edge + Cloud

In questo esperimento è stato considerato quanto segue:

- Modello yolo11x-seg;
- Rumore di quantizzazione crescente;
- Pesi di latenza ed energia variabili nell'insieme $\{0.0, 0.5, 1.0\}$, con somma dei pesi ad 1.0.

In Figura 6.5, nella prima riga, è possibile vedere il confronto tra i valori reali della latenza e quelli predetti in fase di ottimizzazione. Come è possibile osservare, nel caso in cui $w_L = 1$, la predizione risulta piuttosto accurata mentre, negli altri due casi, le differenze arrivano ad essere anche significative. I possibili motivi di questa discrepanza verranno chiariti nelle sezioni successive.

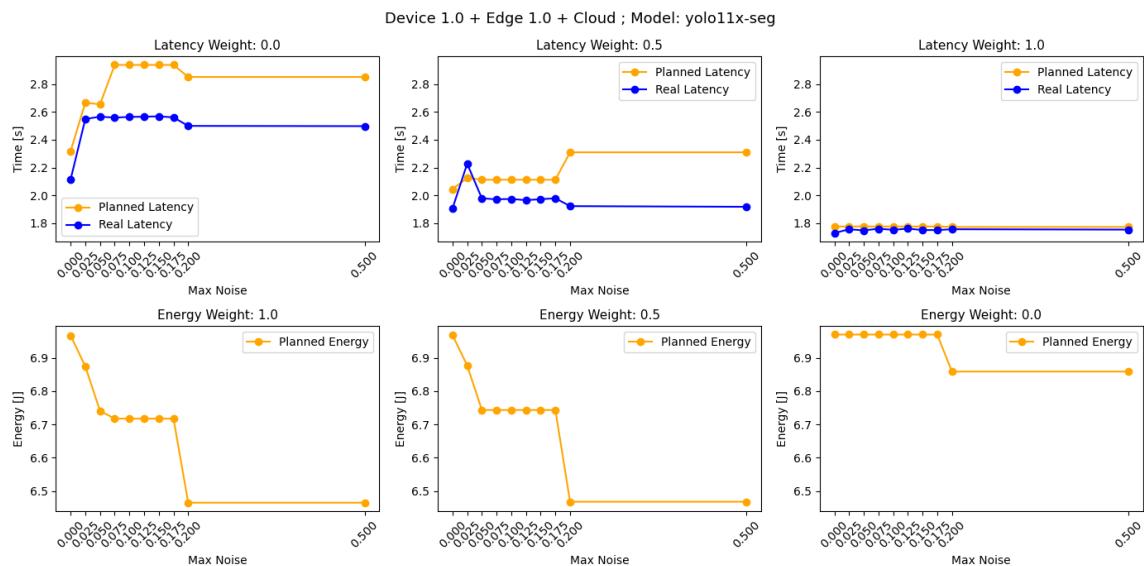


Figura 6.5: Device, Edge, Cloud: Reale vs Predetto

6.4 Confronto con Baseline

In questa sezione ci proponiamo di analizzare, se presente, l'effettivo miglioramento apportato dal sistema in termini tanto di tempo di inferenza quanto di consumo energetico. Nello specifico, è stata considerata la seguente configurazione:

- Modello yolo11x-seg;
- Rumore di quantizzazione crescente;
- Pesi di latenza ed energia variabili nell'insieme $\{0.0, 0.5, 1.0\}$, con somma dei pesi ad 1.0.

In Figura 6.6 vengono riportati i confronti tra le diverse configurazioni; considerata come situazione base il caso in cui è presente il solo device, nella prima riga sono riportate le rappresentazioni grafiche del tempo di inferenza al variare del rumore di quantizzazione e al crescere del peso della latenza nella risoluzione del problema di ottimizzazione, mentre nella seconda riga i valori predetti per l'energia al diminuire del peso dell'energia nel problema di ottimizzazione.

Le osservazioni che verranno fatte di seguito sono complementari a quelle che verranno fatte in sezione 6.5.

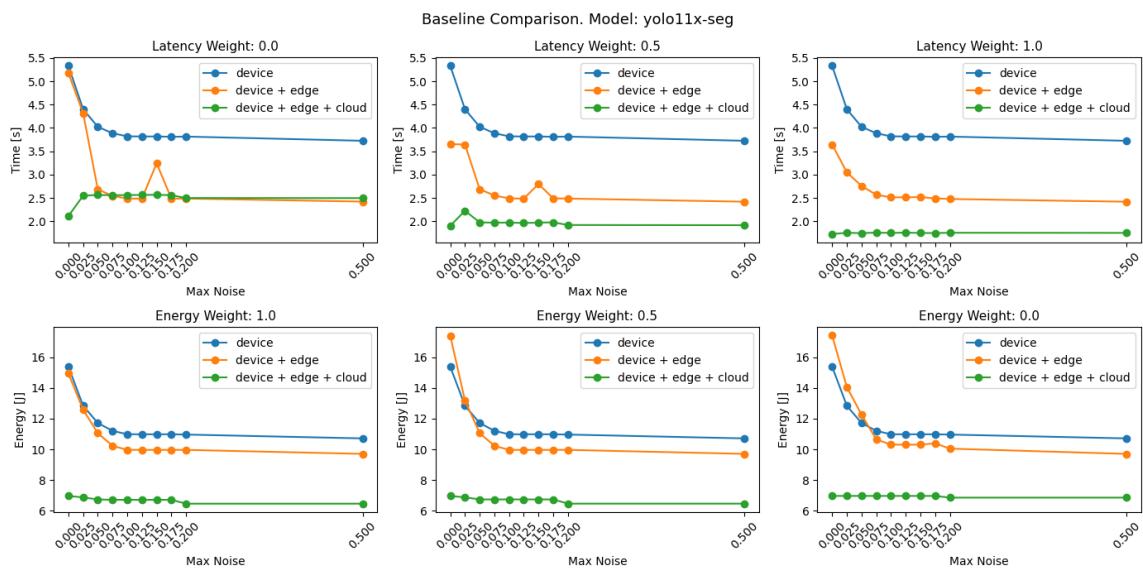


Figura 6.6: Confronto con Baseline

6.4.1 Confronti Latenze

Vengono riportati in Tabella 6.9, Tabella 6.10 e Tabella 6.11 i valori reali della latenza al crescere del rumore di quantizzazione ammesso nei tre casi di w_L rispettivamente pari a 1.0, 0.5 e 0.0.

Come poteva essere prevedibile, nel caso in cui $w_L = 1.0$, le configurazioni più vantaggiose risultano essere quelle che presentano dispositivi più potenti, sia che si tratti dell'aggiunta del solo nodo edge, sia che si tratti dell'aggiunta di edge e cloud.

In generale, è opportuno notare come l'uso della quantizzazione sia benefico in termini di latenza in tutte le configurazioni tranne in quella con il cloud, cosa che sarà analizzata più in dettaglio successivamente. Nel caso della configurazione *device*, all'incremento del rumore di quantizzazione, si passa da una latenza di 5.32 s ad una latenza di 3.72 s, con un risparmio di 1.6 s (un miglioramento di circa il 30%). Per la configurazione *device+edge* si passa invece da 3.64 s a 2.42 s (un guadagno di circa il 33%). Nell'ultimo caso, invece, la latenza si mantiene abbastanza stabile intorno a 1.75 s.

Confrontando invece le tre configurazioni tra di loro, si vede chiaramente come la configurazione a *device+edge+cloud*, pur non traendo beneficio dalla quantizzazione, sia la migliore in assoluto. Allo stesso modo, è interessante osservare come la configurazione *device* a rumore massimo 0.5, raggiunga una latenza molto vicina a quella *device+edge*: chiaramente il costo da pagare in questi termini è l'accuratezza del risultato. Il risultato tuttavia è sorprendente se si considera che il massimo numero di livelli quantizzabili che è stato considerato in questo esperimento è 12; questo significa che, a parità di dispositivo, 12 livelli di 652, e cioè l' 1.85% dei livelli, contribuisce al 30% del tempo di inferenza.

Nel caso in cui il $w_L = w_E = 0.5$, notiamo che gli andamenti della configurazione *device+edge* sono piuttosto simili a quelli del caso $w_L = 1.0$. Nella configurazione *device+edge+cloud*, invece, come si può apprezzare meglio dalla tabella, i valori di latenza sono sensibilmente più alti; vedremo in seguito che la causa di ciò è l'esecuzione di una parte molto piccola di computazione sugli altri dispositivi.

Risulta interessante notare come nella configurazione a $w_L = 0.0$, a partire da un certo rumore di quantizzazione ammesso (nello specifico 0.05), i valori di latenza convergano allo stesso valore nelle due configurazioni *device+edge* e *device+edge+cloud*.

Max Noise	Device	Device+Edge	Device+Edge+Cloud
0.0000	5.3285	3.6455	1.7286
0.0250	4.4011	3.0483	1.7555
0.0500	4.0243	2.7579	1.7477
0.0750	3.8866	2.5696	1.7594
0.1000	3.8214	2.5146	1.7519
0.1250	3.8171	2.5156	1.7600
0.1500	3.8184	2.5203	1.7504
0.1750	3.8105	2.4927	1.7500
0.2000	3.8154	2.4790	1.7558
0.5000	3.7257	2.4204	1.7531

Tabella 6.9: Confronto Latenza $w_L = 1.0$

6.4.2 Confronti Energia

In termini di confronto tra i valori predetti per l'energia, è fondamentale tenere in considerazione quanto segue: nella formulazione del problema il consumo energetico che viene considerato è il consumo energetico complessivo di tutti i dispositivi nel sistema, non di un dispositivo specifico; l'unico dispositivo per il quale viene riservato un trattamento diverso è il device, per il quale si può eventualmente attivare un vincolo

Max Noise	Device	Device+Edge	Device+Edge+Cloud
0.0000	5.3285	3.6551	1.9050
0.0250	4.4011	3.6448	2.2272
0.0500	4.0243	2.6873	1.9779
0.0750	3.8866	2.5546	1.9714
0.1000	3.8214	2.4910	1.9737
0.1250	3.8171	2.4877	1.9653
0.1500	3.8184	2.8000	1.9720
0.1750	3.8105	2.4915	1.9788
0.2000	3.8154	2.4882	1.9222
0.5000	3.7257	2.4208	1.9175

Tabella 6.10: Confronto Latenza $w_L = 0.5$

Max Noise	Device	Device+Edge	Device+Edge+Cloud
0.0000	5.3285	5.1731	2.1144
0.0250	4.4011	4.3179	2.5490
0.0500	4.0243	2.6833	2.5658
0.0750	3.8866	2.5472	2.5592
0.1000	3.8214	2.4862	2.5640
0.1250	3.8171	2.4877	2.5655
0.1500	3.8184	3.2433	2.5686
0.1750	3.8105	2.4870	2.5606
0.2000	3.8154	2.4886	2.4995
0.5000	3.7257	2.4229	2.4977

Tabella 6.11: Confronto Latenza $w_L = 0.0$

relativo al consumo energetico massimo voluto, ma che, in questo esperimento, non è stato considerato.

Riportiamo in Tabella 6.12, Tabella 6.13 e Tabella 6.14 i valori predetti per il consumo energetico nei casi rispettivamente di w_E pari a 0.0, 0.5 e 1.0.

Come è possibile osservare sia graficamente sia dai valori numerici, la configurazione che permette il consumo energetico complessivo minore è quella con tutti i dispositivi attivi. Il motivo è da attribuire al fatto che, sebbene il cloud sia il dispositivo a consumo energetico maggiore, risulta anche quello a maggior velocità di calcolo: in termini energetici complessivi, quindi, il cloud permette il minore impatto. In generale, vi sono delle variazioni minime al variare del peso dato al consumo energetico in fase di ottimizzazione.

Osservazioni interessanti si possono fare invece tra il caso *device* e *device+edge* nelle tre configurazioni energetiche. Osserviamo che quando il peso dato al consumo energetico è massimo, la configurazione con *device+edge* risulta la più conveniente a prescindere dal valore della quantizzazione; similmente al caso del cloud, il device ha un impatto minore in termini di consumo, ma ha anche un tempo di elaborazione più lungo, per cui risulta più conveniente in termini energetici delegare gran parte del lavoro al nodo edge traendo vantaggio dalla sua maggiore capacità di calcolo.

Nel caso bilanciato, osserviamo che il solo *device* si comporta leggermente meglio della configurazione *device+edge* per i primi due valori di quantizzazione, ma, all'aumentare del rumore ammesso, la seconda configurazione continua ad essere la migliore. In questo caso, il motivo è da attribuire proprio al bilanciamento che viene fatto in fase di ottimizzazione: ricordando le configurazioni di calcolo e di consumo dei due dispositivi, il device potrebbe trasmettere immediatamente l'input all'edge, consumando energia per la trasmissione, ma velocizzando il calcolo, ma potrebbe an-

che scegliere di tenere la computazione in locale, non delegando la computazione e traendo, se e dove possibile, vantaggio dalla quantizzazione. Riconducendoci ai valori in Tabella 6.7, come a rumore consentito massimo pari a 0.025 corrisponda un decremento del tempo di inferenza (predetto) di quasi 900 ms e quindi, in termini energetici, un decremento di quasi 3 J in termini energetici.

Max Noise	Device	Device+Edge	Device+Edge+Cloud
0.0000	15.3780	17.4059	6.9698
0.0250	12.8355	14.0559	6.9698
0.0500	11.7153	12.2528	6.9698
0.0750	11.2069	10.6509	6.9698
0.1000	10.9816	10.3059	6.9698
0.1250	10.9816	10.3059	6.9698
0.1500	10.9816	10.3059	6.9698
0.1750	10.9816	10.3955	6.9698
0.2000	10.9630	10.0475	6.8587
0.5000	10.7146	9.7062	6.8587

Tabella 6.12: Confronto Energia $w_E = 0.0$

Max Noise	Device	Device+Edge	Device+Edge+Cloud
0.0000	15.3780	17.3855	6.9683
0.0250	12.8355	13.1690	6.8770
0.0500	11.7153	11.0539	6.7433
0.0750	11.2069	10.2316	6.7433
0.1000	10.9816	9.9660	6.7433
0.1250	10.9816	9.9660	6.7433
0.1500	10.9816	9.9660	6.7433
0.1750	10.9816	9.9660	6.7433
0.2000	10.9630	9.9660	6.4677
0.5000	10.7146	9.7062	6.4677

Tabella 6.13: Confronto Energia $w_E = 0.5$

Max Noise	Device	Device+Edge	Device+Edge+Cloud
0.0000	15.3780	14.9377	6.9670
0.0250	12.8355	12.5812	6.8742
0.0500	11.7153	11.0539	6.7404
0.0750	11.2069	10.2315	6.7174
0.1000	10.9816	9.9659	6.7174
0.1250	10.9816	9.9659	6.7174
0.1500	10.9816	9.9659	6.7174
0.1750	10.9816	9.9659	6.7174
0.2000	10.9630	9.9659	6.4649
0.5000	10.7146	9.7061	6.4649

Tabella 6.14: Confronto Energia $w_E = 1.0$

6.5 Divisione della Computazione ed Uso della Quantizzazione

In questa sezione ci proponiamo di analizzare in maniera più dettagliata il modo in cui i livelli del modello vengono distribuiti tra i nodi del sistema e quanto e se i nodi vengono quantizzati. Nello specifico, è stata considerata la seguente configurazione:

- Modello yolo11x-seg;
- Rumore di quantizzazione crescente;
- Pesi di latenza ed energia variabili nell'insieme $\{0.0, 0.5, 1.0\}$, con somma dei pesi ad 1.0.

Prima di commentare i risultati è importante ricordare una cosa. Come si potrà osservare dai grafici, al device è sempre assegnato un numero di nodi pari almeno a 2; il motivo è che, nella formulazione del problema, il nodo di input e di output sono stati sempre assunti sul device, in modo da modellare in modo coerente la generazione dell'istanza di input e la ricezione del risultato dell'inferenza. Di fatto, questi nodi

possono essere visti semplicemente come livelli identità, che si limitano a dare in output il loro input. La stessa cosa si dica per l'assegnazione a livello di componenti: al nodo device sono sempre assegnate due componenti, una di input ed una di output, che sono usate per gestire meglio l'inferenza a livello di implementazione, come spiegato nelle sezioni precedenti; motivo per il quale al device sono sempre assegnate almeno due componenti.

Risulta importante sottolineare, ai fini dell'analisi dei risultati seguenti, che quanto viene riportato riguardo i livelli quantizzati è il numero di livelli che viene quantizzato, non quali livelli lo sono: a parità di numero di livelli quantizzati che verrà mostrato, è possibile che i livelli quantizzati siano diversi. Ricordiamo infatti, come mostrato in Figura 3.3, che la quantizzazione agisce tendenzialmente in modo non lineare e che quindi, anche a parità di numero di livelli quantizzati, il valore del rumore di quantizzazione può essere diverso.

6.5.1 Device+Edge

In Figura 6.7 vengono riportati i grafici a barre in cui è rappresentata la distribuzione dei nodi. Sulla prima riga abbiamo i grafici riguardanti la distribuzione dei nodi, mentre nella seconda riga quanti di questi nodi assegnati sono stati quantizzati in quella distribuzione; la rappresentazione è per w_L decrescente. Nel caso in cui w_L è pari a 1.0, vediamo come all'aumentare del rumore di quantizzazione la computazione sia delegata completamente al nodo edge, fino ad arrivare al rumore di quantizzazione pari a 0.075. Prima di questo valore, eseguire parte della computazione sul device è evidentemente considerato deleterio e si preferisce piuttosto sfruttare la quantizzazione sull'edge per ottimizzare il tempo di inferenza.

Ricordiamo che i dati devono essere trasferiti dal device all'edge e che il tempo

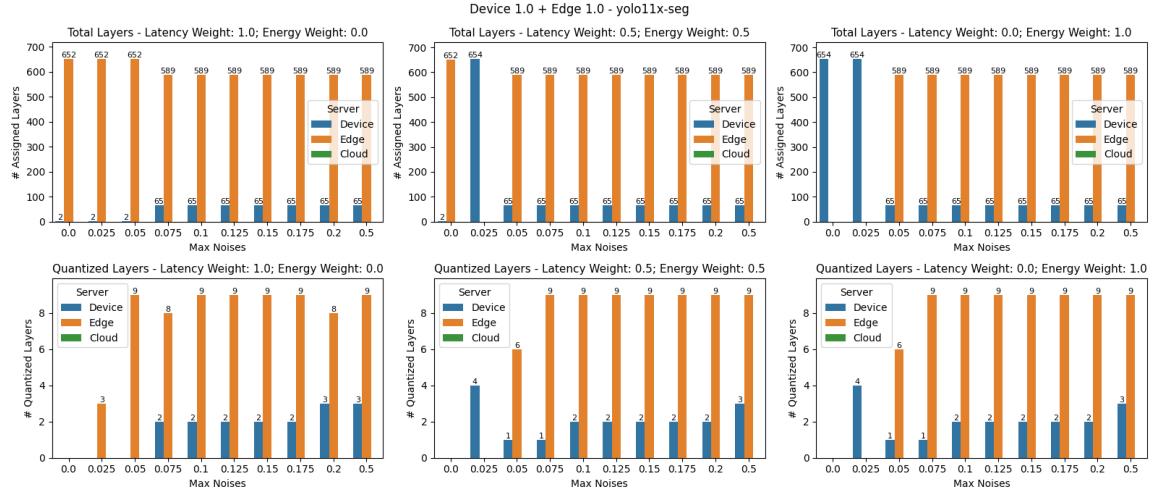


Figura 6.7: Device+Edge: Distribuzione dei nodi e livelli quenatizzati

di trasmissione è parte del problema di ottimizzazione; considerando la dimensione del tensore di input pari a $(1, 3, 640, 640)$ e un input in *FLOAT32*, abbiamo una dimensione del tensore di input pari a circa 4.68 MB. I termini di ottimizzazione, non trasmettere immediatamente il tensore di input significa pagare il costo di computazione sul device, aspettando che all'interno del modello vi sia un punto di restringimento della dimensione dei dati che renda più vantaggiosa la trasmissione; se però questo punto non è presente o è molto in là nel modello, si perde il vantaggio dell'offloading. Questo è ciò che succede in questa situazione: fino al valore del rumore di quantizzazione 0.05 non ci sono punti nella fase iniziale del modello che permettano una trasmissione vantaggiosa e l'unico punto risulta proprio l'input; dopo quel valore, invece, la quantizzazione di un livello permetterà una diminuzione della dimensione dei dati (passiamo infatti da *FLOAT32* a *INT8*) rendendo l'attesa della trasmissione più vantaggiosa. La stessa valutazione si può fare per quanto riguarda la ricezione dell'output. Per valori del rumore di quantizzazione superiori a 0.075, notiamo che il numero di livelli assegnati rimane lo stesso sia per il device che per l'edge, ma

cambia il numero di livelli quantizzati su ciascun dispositivo, fino ad arrivare ad un numero totale di livelli quantizzati pari a 12, appunto il massimo numero di livelli quantizzabili.

Nel caso $w_L = w_E = 0.5$, è interessante vedere come il bilanciamento tra latenza ed energia porti a due risultati opposti nei casi di rumore di quantizzazione 0.0 e 0.025: nel primo caso la computazione è completamente delegata al nodo edge, mentre nel secondo caso la computazione è completamente mantenuta in locale, pur con la quantizzazione di 4 livelli. Qui c'è da considerare infatti non soltanto il tempo di trasmissione, ma anche l'energia di trasmissione sul device che, come abbiamo visto, è abbastanza alta. A partire dal rumore 0.05, invece, le configurazioni rimangono abbastanza stabili.

Nel caso $w_L = 0.0$, per gli stessi valori del rumore di quantizzazione precedenti, vediamo come la computazione venga mantenuta completamente in locale e come, raggiunto il valore 0.05, venga fatto l'offloading. Questo conferma quanto detto prima, e cioè che fino al raggiungimento di un valore di quantizzazione adeguato, non ci sono livelli che rendano la trasmissione vantaggiosa in termini energetici, quindi è preferibile piuttosto mantenere il calcolo in locale.

6.5.2 Device+Edge+Cloud

In Figura 6.8 vengono riportati i grafici a barre in cui è rappresentata la distribuzione dei nodi. Sulla prima riga abbiamo i grafici riguardanti la distribuzione dei nodi, mentre nella seconda riga quanti di questi nodi assegnati sono stati quantizzati in quella distribuzione. La rappresentazione è per w_L decrescente.

Considerando Figura 6.8, possiamo notare come:

- Quanto $w_L = 1.0$, l'ottimizzazione porta all'offloading completo della compu-

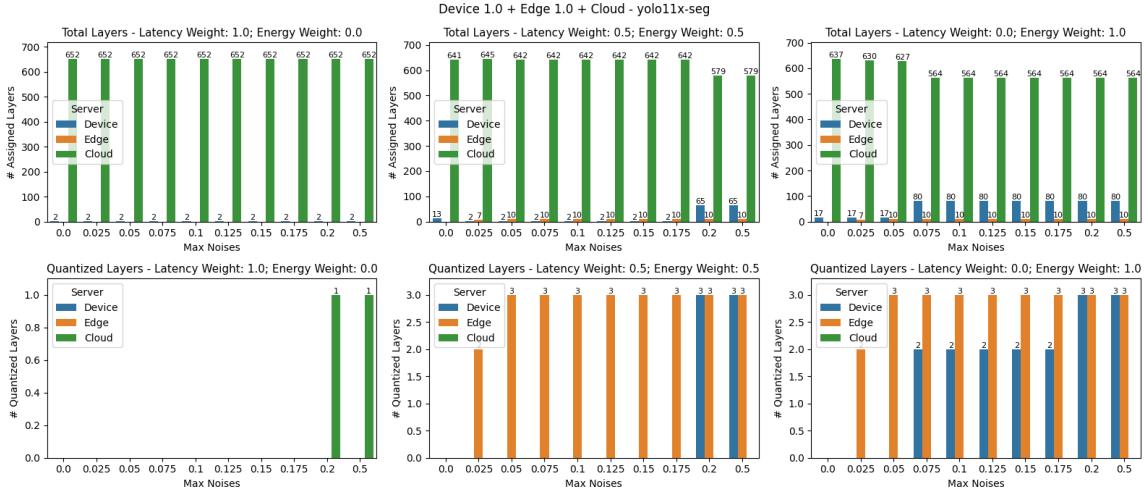


Figura 6.8: Device+Edge+Cloud: Distribuzione dei nodi e livelli quenatizzati

tazione a livello del cloud. Come osservato in precedenza, l'uso della quantizzazione non sembra portare benefici a livello cloud, quindi non si sceglie di quantizzare livelli se non per gli ultimi due valori del rumore.

- Quando $w_L = w_E = 0.5$, vi sia una distribuzione maggiore della computazione tra i tre dispositivi. Significativo è il caso con rumore di quantizzazione nullo, in cui il calcolo viene diviso tra device e cloud anche senza l'uso della quantizzazione, a differenza della configurazione *device+edge*. Questo è dovuto al fatto che il consumo energetico in trasmissione per il cloud è molto basso, quindi la parte finale della computazione viene delegata al device, scegliendo di pagare un costo aggiuntivo di tempo per risparmiare in termini energetici. Prova di questo sia il fatto che, all'attivazione della quantizzazione, la computazione viene fatta solo su edge e cloud, mentre il device viene omesso: non solo il nodo edge si trova più vicino al cloud, ma è anche più veloce rispetto al device. All'aumentare del rumore di quantizzazione la divisione rimane pressoché stabile, fino al valore 0.2, a partire dal quale il device torna ad essere utilizzato. Possiamo vedere che

da questo punto in poi vi sono dei livelli che vengono quantizzati sul device e, confrontando anche con la distribuzione dei nodi in Figura 6.7, vediamo che si tratta della stessa distribuzione di nodi: in questo caso quello che sta succedendo è che la quantizzazione permette una diminuzione del carico di trasmissione per cui la prima parte della computazione viene fatta in locale.

- Quando $w_L = 0.0$, ci troviamo davanti ai casi in cui il calcolo viene distribuito sui tre dispositivi per il maggior numero dei valori del rumore di quantizzazione. Le configurazioni dei consumi infatti sono quelle più varie, per cui la distribuzione del calcolo risulta più vantaggiosa.

6.5.3 Parallelismo dell’Inferenza

In Figura 6.9 sono riportati i grafici a barre rappresentanti il numero di componenti assegnate a ciascun dispositivo. La rappresentazione è per w_L decrescente.

Dal grafico in Figura 6.9 si può vedere come i casi in cui le componenti sono maggiormente distribuite tra i dispositivi corrispondano ai casi in cui in Figura 6.5 la differenza tra predizione e valore reale è maggiore: si veda come esempio la configurazione a $w_L = 0.0$ e $\max_noise = 0.5$.

Il motivo di questa discrepanza è da ricercarsi nella mancata modellazione in fase di ottimizzazione del parallelismo tra le componenti: tanto la computazione dei livelli quanto gli invii dei tensori sono stati modellati in modo sequenziale. Questo chiaramente può portare ad una differenza significativa nel momento in cui vi sono molte componenti: ad esempio, mentre una componente sta finendo di inviare il suo output ad una componente successiva, un’altra può cominciare ad eseguire. Questo è reso chiaro in Figura 6.10 in cui viene riportato il grafo delle componenti per la configurazione di interesse: in rosso sono indicate le componenti di input e di output;

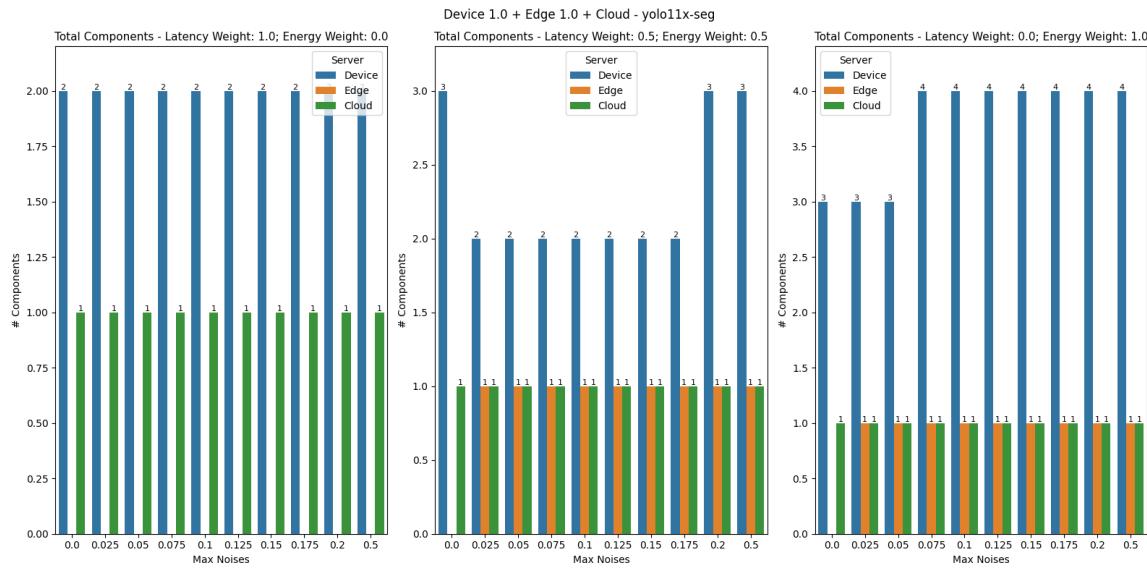


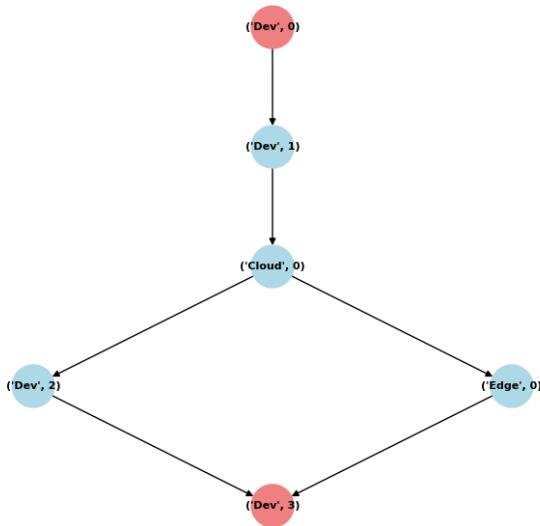
Figura 6.9: Device+Edge+Cloud: Distribuzione delle componenti

per ogni componente viene indicato il dispositivo di esecuzione e un indice che identifica la componente sul dispositivo. In questo caso, non solo la componente del cloud, terminata la sua esecuzione, invia ad un successore alla volta, ma il device e l'edge a loro volta, ricevuto l'input necessario calcolano ed inviano l'output alla componente finale in parallelo.

Evidenziata questa mancanza del modello, si mette comunque in luce il fatto che la modellazione del parallelismo all'interno del problema di ottimizzazione avrebbe reso la risoluzione di quest'ultimo estremamente complessa. Intuitivamente, sarebbe stato necessario:

- Introdurre le componenti all'interno del problema di ottimizzazione;
- Definire un primo mapping tra il grafo del modello e le componenti, costruendo un grafo delle componenti che fosse anche aciclico; allo stesso tempo sarebbe stato necessario tracciare le dipendenze tra le componenti così costruite in modo da definire la latenza come l'istante di inizio dell'ultima componente sul device.

Forse queste considerazioni le sposterei al capitolo finale con gli sviluppi futuri/limitazioni

Figura 6.10: Grafo delle Componenti per $w_L = 0.0$ e $noise = 0.5$

- Definire un secondo mapping tra il grafo delle componenti e il grafo della rete.

Questo avrebbe definito quindi un problema di doppio graph assignment, di cui uno aciclico, portando ad un problema estremamente complesso da risolvere in modo esatto e non semplice da affrontare in modo euristico.

6.6 Scalabilità del Problema

Il problema di ottimizzazione formulato è riconducibile ad un problema di graph assignment e, come tale, risulta NP-Hard. Di conseguenza, è opportuno valutare la scalabilità della soluzione.

Risulta chiaro come il tempo di risoluzione del problema dipenda da vari fattori; di seguito viene riportato un elenco non completo di alcune di queste variabili:

- Numero di livelli nel modello;
- Numero di tensori nel modello;

- Numero di nodi di rete;
- Configurazioni energetiche dei server.

Un aspetto non trascurabile da considerare poi è la struttura del modello: sebbene il parallelismo non sia tenuto in considerazione nella risoluzione del problema, un modello con una struttura particolarmente complessa può portare ad uno scambio di tensori tra i livelli che potrebbe rendere il problema complicato da risolvere.

Al fine di testare la risoluzione del problema con varie configurazioni del modello, verranno costruiti dei modelli *fittizi*: questi modelli non hanno lo scopo di essere realistici, ma vogliono essere degli strumenti che permettono il test di scalabilità in varie condizioni. Poiché i modelli sono *fittizi*, nella risoluzione del problema non verrà considerato il rumore di quantizzazione. Per finire, è stato considerato il caso intuitivamente più complesso in termini di risoluzione, ovvero il caso di pesi di latenza ed energia bilanciati ($w_L = w_E = 0.5$).

Ricordiamo che nella risoluzione del problema, sono tre i problemi di ottimizzazione che vengono risolti:

- Problema di ottimizzazione per trovare il tempo minimo: indicato di seguito con *Latency Time*.
- Problema di ottimizzazione per trovare l'energia minima: indicato di seguito con *Energy Time*.
- Problema di ottimizzazione finale con obiettivo la somma pesata normalizzata dei due termini: indicato di seguito con *WholeTime*.

Oltre a questi tempi, ci sono anche altri due valori che possiamo considerare:

- Tempo di costruzione del problema: indicato di seguito con *Build Time*.

- Tempo di post elaborazione della soluzione: indicato di seguito con *Post Time*.

Chiaramente, il tempo complessivo sarà dato dalla somma di tutti questi termini; di seguito abbiamo preferito concentrarci sui singoli contributi, quindi il tempo complessivo non è stato riportato.

6.6.1 Costruzione della Rete

Supposto di indicare ogni server con un indice crescente che parte da 0, la configurazione della rete è stata fatta nel seguente modo:

- Numero di FLOPS per server: partendo da un valore iniziale x e moltiplicando per $10^{3 \cdot idx}$.
- Configurazione energetica: crescente linearmente con l'indice del server partendo da un valore di base.
- Trasmissione tra server x ed y :
 - Latenza tra due server: crescente linearmente all'aumentare di $|x - y|$;
 - Banda tra due server: decrescente linearmente all'aumentare di $|x - y|$.

Per quanto riguarda il profilo di esecuzione di un server per il modello fittizio, questo è stato calcolato prendendo semplicemente come tempo di esecuzione di un livello il rapporto tra i FLOPS del livello e i FLOPS calcolabili dal server.

Sono stati considerati un massimo 6 nodi di rete.

6.6.2 Configurazione Statica

In questo caso, lo scopo è analizzare la variazione del tempo di risoluzione del problema al crescere della dimensione del modello.

6.6.2.1 Costruzione del Modello

Poiché, come detto, il modello può crescere in vari modi, consideriamo in questo caso un modello estremamente semplice, costruito come segue:

- Modello con una struttura lineare, formato da un unico branch;
- Nessuna skip connection tra i livelli;
- Tutti i layer con stesse caratteristiche.

Si parte da un numero iniziale di livelli pari a 500 e si incrementa il numero di livelli sull'unico branch di 150 livelli ad ogni iterazione.

6.6.2.2 Risultati

Di seguito vengono riportati i risultati del test, sia al variare del numero di livelli sia al variare del numero di nodi di rete.

In Figura 6.11 troviamo gli andamenti del tempo di risoluzione: ogni grafico è relativo ad un certo numero di livelli; sull'asse x viene riportato il numero di nodi di rete per cui il problema è stato risolto, mentre sull'asse y il tempo di risoluzione in scala logaritmica. Come possiamo vedere dai grafici, a parità di numero di livelli:

- I tempi di risoluzione dei problemi ai minimi crescono abbastanza linearmente in scala logaritmica e, quindi, esponenzialmente in scala lineare: questo andamento era più o meno prevedibile; l'NP-Hardness del problema di assegnazione, fa sì che il tempo di risoluzione del problema cresca facilmente all'aumentare del numero di nodi di rete.
- In generale, il problema di risoluzione più complicato all'aumentare del numero di nodi di rete sembra essere quello complessivo, seguito dal problema di

minimizzazione del tempo e da quello di minimizzazione dell'energia. Interessante notare come, per pochi server (minore o uguale a 3) il problema più complicato sembri quello della latenza, mentre problema complessivo e problema dell'energia si comportino più o meno allo stesso modo.

- Il tempo di post elaborazione della soluzione sembra costante: l'algoritmo di risoluzione delle dipendenze rappresenta, infatti, la parte più onerosa della post elaborazione e questo algoritmo non dipende dal numero di server considerati.
- Il tempo di costruzione del problema cresce, sebbene non linearmente in scala logaritmica e, quindi, non esponenzialmente in scala lineare.

In Figura 6.12 troviamo gli andamenti del tempo di risoluzione fissato il numero di nodi di rete: ogni grafico è relativo al numero di nodi di rete usato in quella soluzione; sull'asse x troviamo il numero di livelli del modello, mentre sull'asse y il tempo di risoluzione in scala logaritmica. Come possiamo vedere dai grafici, a parità di numero di server:

- Con pochi server, il tempo di costruzione e di post elaborazione risultano predominanti sui singoli tempi di risoluzione del problema, anche se non lo saranno sulla somma dei tre tempi di risoluzione.
- I tempi di risoluzione crescono con andamento più o meno lineare, quindi esponenziale in scala lineare, all'aumentare del numero di livelli.
- Con pochi server, i tempi di risoluzione tendono ad essere instabili, mentre dai 5 server in poi gli andamenti si stabilizzano.

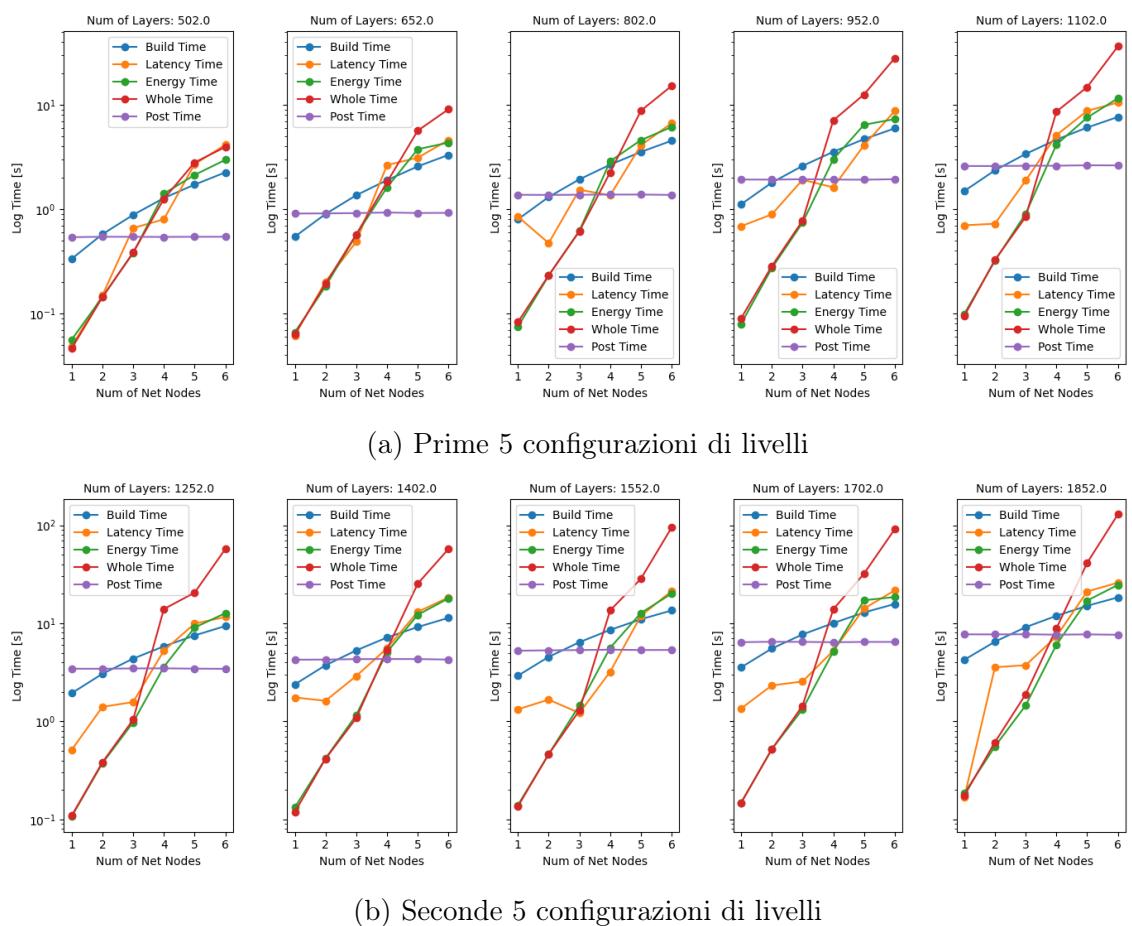


Figura 6.11: Caso Statico - Andamento del tempo di risoluzione all'aumentare del numero di livelli

- Raggiunto un certo numero di server, 6 in questo caso, il tempo di costruzione e di post elaborazione sono la parte meno significativa della risoluzione complessiva.
- Il tempo di post elaborazione della soluzione cresce, sebbene non linearmente e, quindi, non esponenzialmente in scala lineare.
- Il tempo di costruzione del problema cresce, sebbene non linearmente e, quindi, non esponenzialmente in scala lineare.

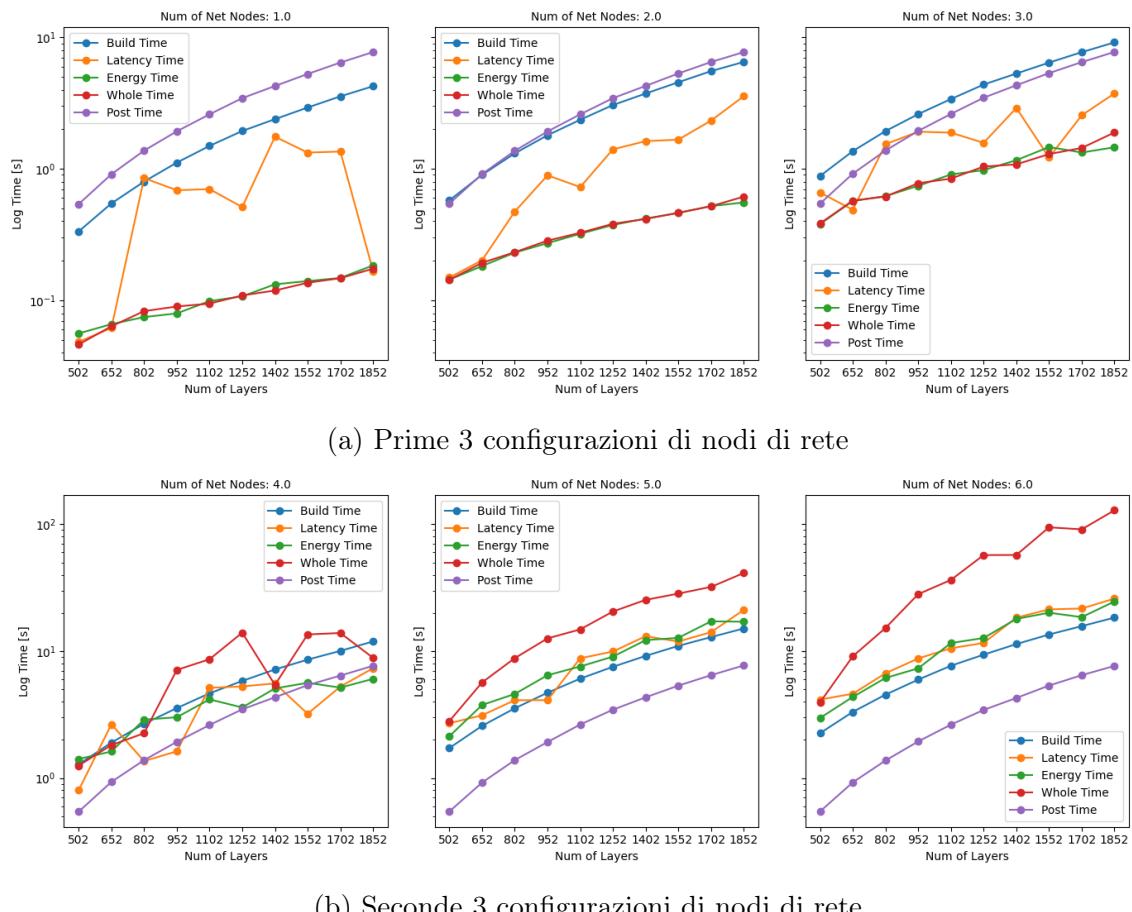


Figura 6.12: Caso Statico - Andamento del tempo di risoluzione all'aumentare del numero di nodi di rete

6.6.3 Configurazione Randomica

Lo scopo di questo test è vedere se e come la risoluzione del problema è influenzata dalla struttura del modello. Chiaramente, non sarà possibile fare un'analisi di tutte le strutture possibili per un modello, in quanto queste strutture possono essere complesse a piacere; di conseguenza, verranno analizzate delle strutture randomiche che partono da una base comune.

6.6.3.1 Costruzione del Modello

I modelli sono stati costruiti partendo da un'analisi statistica di un modello di riferimento; nello specifico, per il modello di riferimento:

- Per ogni coppia di tipi di livelli è stato fatto un conteggio del numero di archi che connettevano quei tipi di livelli; da questo conteggio è stata definita una distribuzione di probabilità di transizione da un livello di tipo x ad un livello di tipo y .
- Per ogni tipo di livello, sono stati raccolti:
 - I FLOPS dei livelli di quel tipo;
 - Le dimensioni dei tensori di output dei livelli di quel tipo;
 - Le dimensioni dei pesi dei livelli di quel tipo;

Partendo da questi dati, è stato costruito un modello fittizio. Definito un certo numero di branch, di cui uno principale, e il numero di livelli in ogni branch, si è proceduto nel seguente modo:

1. Ogni branch viene costruito randomicamente, considerando come primo livello un livello convoluzionale, per poi transitare al livello successivo secondo le pro-

babilità di transizione del modello di riferimento. Notare che in questo modo il branch è semplicemente una concatenazione lineare di livelli;

- Per ogni livello del branch, il numero di FLOPS, le dimensioni dei pesi e le dimensioni dei tensori di output sono estratti dai dati relativi al tipo di livello usando una distribuzione uniforme;
2. Una volta creato un branch, in base ad una certa probabilità di skip, per ogni livello si valuta se aggiungere una skip connection verso un livello successivo nel branch.
 3. Una volta creato un branch parallelo, si scelgono casualmente due punti del branch primario a cui connetterlo.
 4. Il modello finale sarà dato dal branch primario a cui sono connessi i diversi branch secondari.

Nel nostro esperimento sono stati considerati:

- 5 branch totali, di cui il principale di 500 livelli ed i secondari di 50.
- Ad ogni iterazione, la dimensione del branch principale viene incrementata di 50 livelli, mentre le dimensioni dei branch secondari vengono incrementate di 15 livelli.
- La probabilità di skip è stata impostata a 0.15.

6.6.3.2 Risultati

In Figura 6.13 sono mostrati gli andamenti dei tempi di risoluzione fissato il numero di livelli e all'aumentare del numero di nodi di rete: ogni grafico fa riferimento ad un

numero di livelli; sull'asse x è riportato il numero di nodi di rete, mentre sull'asse y il tempo di risoluzione in scala logaritmica. Possiamo notare come:

- Gli andamenti dei tempi di post elaborazione e di costruzione del problema siano coincidenti con quelli in Figura 6.11.
- Per quanto riguarda i tempi di risoluzione dei problemi di ottimizzazione, troviamo degli andamenti tendenzialmente lineari in scala logaritmica (quindi esponenziali in scala lineare), ma con natura diversa rispetto al caso statico. Se nel caso statico, il problema che risultava più complesso da risolvere risultava quello di minimizzazione complessiva, in questo caso il problema che richiede più tempo sembra quello di minimizzazione dell'energia. L'unica eccezione è quella con 1142 livelli, in cui il problema a complessità maggiore è quello del tempo.
- Il numero di livelli sembra giocare un ruolo inferiore nel tempo di risoluzione: mentre nel caso statico l'andamento crescente con il numero di livelli era abbastanza evidente, qui ci sono dei casi in cui non c'è molta differenza, come ad esempio tra le coppie 702 e 812 o 922 e 1032.
- Gli andamenti non sono costanti al crescere del numero di livelli: ad esempio, il tempo per risolvere il problema di minimizzazione complessivo è inferiore con 1142 livelli che con 1032.

Questo mostra non solo come la complessità del problema aumenti con il numero di livelli e con il numero di server, ma anche come la struttura del modello impatti quale dei tre problemi risulti più complesso.

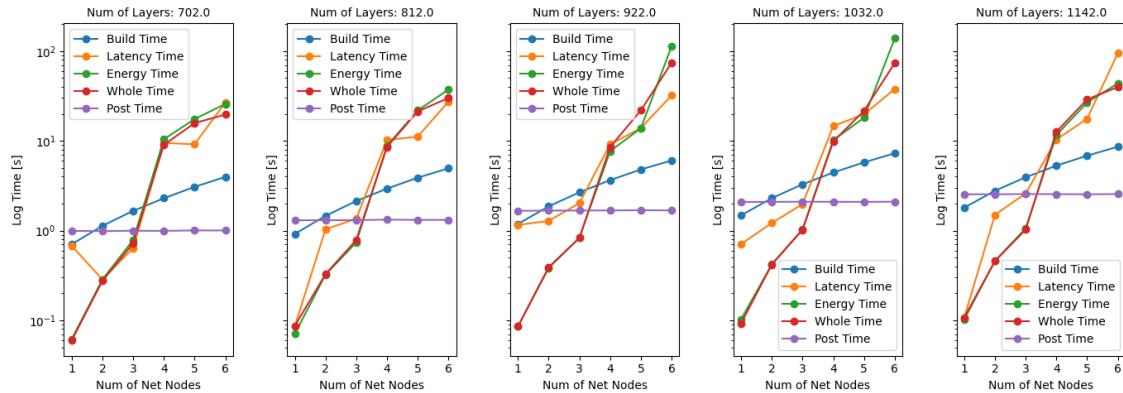


Figura 6.13: Caso Random - Andamento del tempo di risoluzione all'aumentare del numero di livelli

6.7 Limite sul Consumo Energetico

Lo scopo di questo esperimento è verificare la capacità del sistema di dividere il modello in presenza di un limite sul consumo energetico del device.

Per farlo, ci siamo messi in una configurazione che fosse semplice da analizzare. In Tabella 6.15 viene riportata la configurazione usata in questo esperimento: in tabella, la banda e la latenza si intendono bidirezionali; il modello considerato è stato yolo11x-seg. Sebbene questa configurazione possa sembrare poco realistica per alcuni aspetti,

	Device	Edge
FLOPS	10^{12}	10^{11}
Potenza Calcolo [W]	5	2.5
Potenza Trasmissione [W]	2	2
Bandwidth [MB/s]	20	20
Latenza [ms]	5	5

Tabella 6.15: Limite Consumo - Configurazione Esperimento

come per il fatto che il device sia più potente dell'edge, è importante considerare che questo esperimento vuole essere più un *proof of concept* piuttosto che un test fatto, come nei casi precedenti, per valutare l'accuratezza della predizione sul consumo; non

disponendo di strumenti per verificare l'accuratezza della predizione, ci accontentiamo quanto meno di verificare che il sistema si comporti come dovrebbe.

Intuitivamente, risolvendo il problema con i pesi $w_L = 1.0$ e $w_E = 0.0$, poiché il dispositivo più veloce è il device, l'ottimizzatore sarà portato a posizionare il modello interamente sul device; tuttavia, imponendo un limite al consumo energetico del dispositivo, si troverà obbligato a dividere il modello, posizionando dei livelli anche sul nodo edge.

6.7.1 Risultati

In Figura 6.14 è possibile osservare l'andamento del valore del consumo energetico predetto per il device in fase di ottimizzazione. Sull'asse delle x viene rappresentato il limite al consumo energetico che viene chiesto in fase di ottimizzazione, mentre sull'asse delle y il valore predetto; entrambi i consumi sono misurati in Joule. Come è possibile osservare, il problema non ha soluzione fino a quando non si raggiunge un certo limite di consumo: infatti, quando il tetto al consumo è molto basso, il modello tenderà ad essere spostato interamente sul nodo edge; tuttavia, trovandosi l'input sul device, è necessario spendere almeno l'energia per trasferire l'input e questo potrebbe essere impossibile a seconda del limite massimo. Superato un certo valore al limite, in questo caso 0.45 J, il problema ha soluzione e il modello viene diviso tra i due dispositivi, come vedremo in seguito. All'aumentare del limite, il consumo predetto tende ad aumentare, fino a raggiungere il valore massimo, ottenuto quando il modello è eseguito completamente in locale. In Figura 6.15 è possibile vedere il modo in cui i livelli sono posizionati nella risoluzione del problema. Sull'asse x troviamo il limite all'energia misurato in Joule, mentre sull'asse y troviamo il numero di livelli assegnati a ciascun server. Come potevamo aspettarci, quando il limite

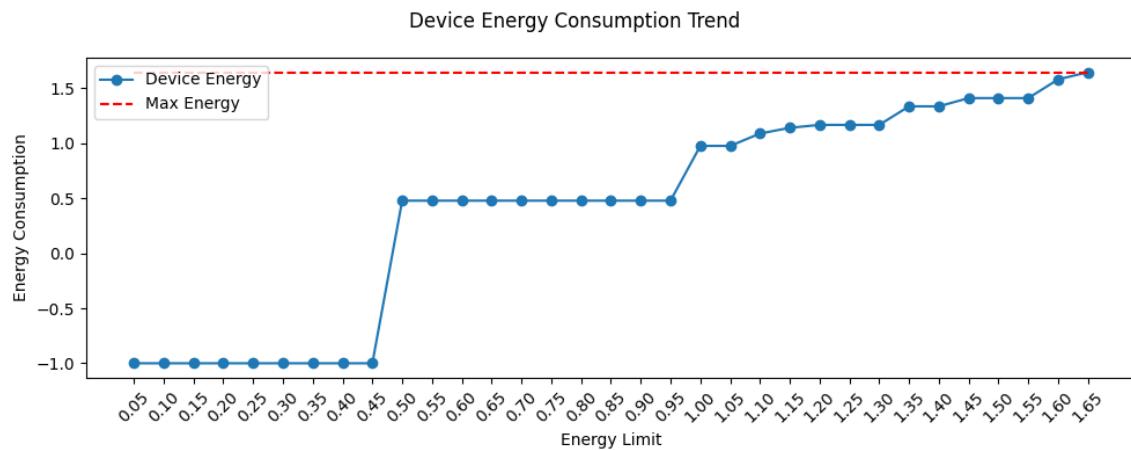


Figura 6.14: Limite Consumo - Andamento del Valore Predetto

è molto basso, la computazione viene eseguita quasi completamente sul nodo edge, anche se c'è un piccolissimo sottoinsieme di nodi eseguito anche sul device. Andando avanti, i livelli vengono sempre più posizionati sul device, fino a quando il modello è eseguito interamente in locale.

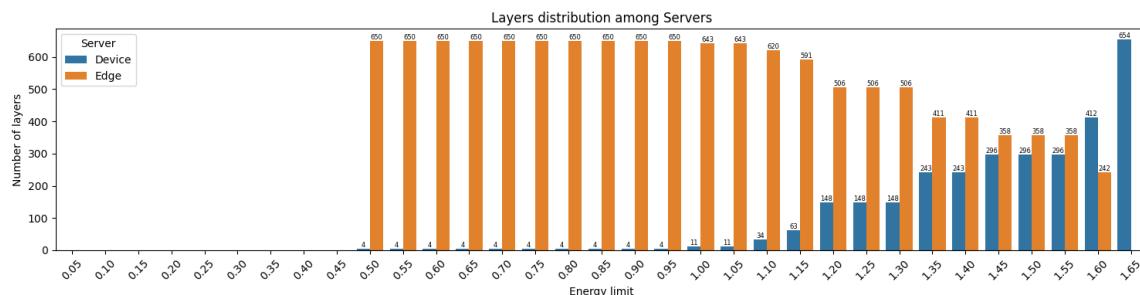


Figura 6.15: Limite Consumo - Assegnazione dei Livelli

Capitolo 7

Limitazioni e Sviluppi Futuri

In questo capitolo verrà prima fatta un'analisi delle mancanze e delle limitazioni del sistema presentato, per poi passare a delle proposte di sviluppi futuri.

7.1 Limitazioni

7.1.1 Scalabilità del tempo di soluzione

Come mostrato in sezione 6.6 il tempo di risoluzione del problema all'ottimo cresce notevolmente, tanto all'aumentare del numero di server quanto all'aumentare della dimensione del modello; inoltre, alcuni modelli si mostrano come più complicati da risolvere rispetto ad altri, anche a parità di condizioni e di requisiti. Tutto ciò porta chiaramente ad un limite di utilizzo in contesti in cui la soluzione deve essere generata in modo dinamico e/o real time.

7.1.2 Cambio delle Condizioni

Il sistema è stato costruito considerando una situazione che fosse stabile sia in termini di rete sia in termini di numero di server: il throughput e la latenza tra due dispositivi

sono stati considerati come costanti ed affidabili, cosa non necessariamente vera soprattutto considerando reti mobili, come 4G e 5G, spesso influenzate dalle condizioni ambientali. Allo stesso tempo, non sono stati considerati scenari di aggiunta di server. Tutte queste possibili variazioni nelle condizioni del sistema richiederebbero inevitabilmente la generazione di un nuovo piano e la riconfigurazione della distribuzione delle parti del modello.

7.1.3 Tolleranza ai Guasti

Così come non è stata considerata la possibilità di aggiunta di server, non sono stati considerati nemmeno scenari di fallimento dei server. Tuttavia, se nel caso dell'aumento del numero di server la peggiore delle ipotesi è che il piano generato in precedenza non sia più quello ottimo, nel caso di fallimento del server l'inferenza non sarà possibile in generale. In questo senso, quindi, si renderebbe necessaria l'introduzione di fallback ad un caso base, oppure la possibilità di riconfigurare velocemente il sistema in caso di fallimento. La riconfigurazione veloce del sistema, chiaramente, va a scontrarsi con la limitazione relativa al tempo di risoluzione del problema.

7.1.4 Aumento del numero di livelli quantizzabili

Nonostante i risultati ottenuti introducendo la quantizzazione siano promettenti sia nel caso di deployment sul device sia nel caso di deployment nel continuum, il numero di livelli quantizzabili risulta fortemente limitato dalla possibilità di costruire un regressore accurato all'aumentare del numero di livelli quantizzabili: un aumento del numero di livelli quantizzabili richiederebbe, infatti, l'analisi di più configurazioni e, per un alto numero di livelli, l'esplorazione sarebbe infattibile. La limitazione principale in questo senso è data dal fatto che l'uso di un numero maggiore di li-

velli quantizzabili potrebbe aprire la possibilità a degli splitting più vantaggiosi e interessanti.

7.1.5 Profiling del Modello

Anche se in sezione 6.1 non si è andati nel dettaglio riguardo la profilazione del modello e la variazione al variare della configurazione di quantizzazione, questo aspetto è un punto abbastanza critico in termini di tempo. Quando il modello viene caricato, viene eseguito un profiling iniziale, estraendo un grafo che ne definisca la struttura e le caratteristiche significative. Insieme a queste informazioni, viene anche costruito il regressore per la predizione riguardante la perdita di accuratezza al variare delle configurazioni di quantizzazione. Quest'ultima è la parte più lunga del profiling in quanto richiede, per varie configurazioni di quantizzazione, di costruire il modello quantizzato e di valutare la perdita di accuratezza su un dataset di calibrazione.

7.1.6 Profiling dell'Esecuzione

Come mostrato in Capitolo 6, l'uso di interpolazione permette di riuscire a trascurare l'overhead del profiling livello per livello e di avvicinarsi maggiormente al tempo reale di inferenza del modello. Tuttavia, una limitazione di questa strategia di profiling è il fatto che non sono tenute in conto le ottimizzazioni che vengono fatte a basso livello e questo potrebbe, in alcuni casi, introdurre degli errori nella predizione, come anche parzialmente mostrato negli esperimenti.

Un'altra limitazione relativa al profiling dell'esecuzione è la variabilità di hardware su cui il sistema è stato testato: l'uso di Onnx e OnnxRuntime permette di avere supporto per un'ampia gamma di hardware, ma, nel contesto di questo studio soltanto, sono state considerate soltanto architetture x86 e GPU NVIDIA.

7.1.7 Test del consumo energetico

Come segnalato varie volte nel corso dello studio, uno studio dell'efficacia del sistema rispetto al consumo energetico non è stato possibile a causa della mancanza di strumenti di monitoraggio e di profilazione sulle macchine di test.

7.2 Sviluppi Futuri

7.2.1 Piani adattativi e/o alternativi

Come spiegato nelle limitazioni, variazioni nella rete di server, sia che si tratti di cambiamenti nel throughput e nella latenza, sia che si tratti di variazioni nel numero di server, non sono state considerate. Per far fronte alla tolleranza ai guasti si potrebbero generare dei piani alternativi, risolvendo il problema tenendo conto di varie configurazioni della rete. Questo richiederebbe un sistema di gestione globale in stile MAPE in grado di stabilire, a seconda dei server attivi, quale piano alternativo è il migliore. Nel caso di inserimento di un nuovo server o di modifica dei parametri del collegamento tra server, l'inferenza è ancora possibile ma il piano sub ottimo. In questo senso, il sistema di controllo dovrebbe essere in grado, partendo dall'ultimo stato di generazione del piano, di calcolare un valore indicante il guadagno ottenuto dalla riconfigurazione e, in base a qualche criterio, avviare la generazione del nuovo piano. Situazioni particolarmente variabili, inoltre, potrebbero portare la creazione di molte richieste: ciò rende necessario un sistema di attuazione che tenga conto delle molte richieste ed interrompa quelle vecchie. Ovviamente, tutto questo si scontra con il problema di scalabilità della soluzione; in questo senso si potrebbe valutare, considerando condizioni stabili in intervalli di tempo medio-lunghi, l'uso primario di una euristica, per poi avere un perfezionamento del piano con la soluzione ottima.

7.2.2 Rumore di Quantizzazione

Il rumore di quantizzazione è stato descritto attraverso un'analisi della differenza tra il risultato del modello quantizzato e il risultato del modello originale, assunto come baseline. Una direzione sicuramente interessante potrebbe essere l'estensione del sistema a delle misure del rumore diverse specificabili, ad esempio, dall'utente: mean average precision o valore della loss-function del modello quantizzato sono tutte possibili alternative non esplorate. Ovviamente, l'introduzione di misure diverse per il rumore potrebbe anche richiedere la definizione di modelli diversi per meglio rappresentare questo rumore: polinomi di grado superiore o alberi di decisione potrebbero essere modelli da studiare a questo proposito.

7.2.3 Classi di Richiesta

Nei casi considerati, il limite al rumore di quantizzazione è stato considerato fisso: in fase di ottimizzazione, viene indicato quale sia il rumore massimo che si è disposti ad accettare per tutte le richieste di inferenza del modello. Una possibile direzione per uno studio futuro potrebbe essere l'introduzione di classi di richiesta (e.g. *Critical*, *Normal* e *Low*) ognuna associata ad un rumore massimo ammesso e ad ognuna delle quali corrisponde un piano di inferenza diverso.

7.2.4 Integrazione dell'Early Exit

Come brevemente descritto in Capitolo 1, un'altra direzione per la semplificazione del modello è l'uso dell'Early Exit. Mentre allo stato corrente la risposta ricevuta dalla componente di output coincide con la risposta del modello, l'introduzione dell'early exit richiederebbe l'introduzione di classi di risposta per permettere di distinguere tra le risposte ottenute dalle varie uscite del modello. Al netto di ciò, l'uso dell'early

exit combinato con classi di richiesta e con diverse configurazioni di quantizzazione permetterebbe di creare delle famiglie di piani che potrebbero rispondere a diverse esigenze in base al tipo di richiesta.

7.2.5 Introduzione del parallelismo

Come mostrato in Capitolo 6, la mancata considerazione del parallelismo della fase di inferenza può portare ad una differenza molto alta tra ciò che è predetto e il tempo della fase di inferenza. Questo, chiaramente, può portare alla generazione di piani sub-ottimi. Si rivela dunque necessaria la considerazione del parallelismo in fase di ottimizzazione. Fare ciò, richiederà sicuramente l'introduzione del concetto di componente all'interno del problema di ottimizzazione, così come la modellazione delle loro dipendenze e dell'ordine di esecuzione. Tuttavia, la modellazione di queste dipendenze nel problema di ottimizzazione porterà sicuramente ad un'ulteriore complicazione della risoluzione, accentuando i problemi di scalabilità. Pertanto, si rivelerà ancora più importante la ricerca e lo studio di un'euristica capace di risolvere il problema, in modo da tenere conto di questo aspetto e di ottenere beneficio dalla parallelizzazione delle componenti.

7.2.6 Integrazione delle parti del sistema

Sebbene le varie parti del sistema siano state progettate al fine di rendere il flusso teorico chiaro, ci sono alcune parti che non sono ancora completamente integrate tra di loro. Una possibile direzione di sviluppo futura è sicuramente l'integrazione delle varie parti della pipeline al fine di ottenere un prodotto che sia più vicino ad uno stato stabile per la produzione.

**Sposterei qui le Limitazioni e sviluppi futuri.
Puoi chiamare il capitolo "Conclusioni e Sviluppi Futuri"**

Conclusioni

L'obiettivo dello studio è stato quello di cercare, per modelli di deep learning, un partizionamento in componenti e il placement di queste componenti nell'edge-cloud continuum al fine di ottimizzare un obiettivo che combinasse, in un'unica metrica, tanto il tempo di inferenza quanto il consumo energetico, focalizzandosi in particolare sul massimo consumo energetico del dispositivo richiedente l'inferenza e sul massimo rumore di quantizzazione accettato.

Usando OnnxRuntime come framework di inferenza, è stato possibile garantire il supporto a diverse architetture hardware e anche l'estensione a tipi di architettura non testati nel contesto di questo studio.

Come studio preliminare, è stato affrontato il problema della modellazione del rumore di quantizzazione e della sua introduzione nel problema di ottimizzazione. Trattandosi di un problema combinatorio, lo spazio di ricerca delle possibili configurazioni di quantizzazione è stato ridotto e, in questo spazio ridotto, il rumore di quantizzazione è stato modellato attraverso una regressione polinomiale. La regressione polinomiale, addestrata con i valori del rumore di varie configurazioni di quantizzazione, ha prodotto buonissimi risultati in termini di accuratezza.

L'ottimizzazione del deployment è stata modellata come un problema di ottimizzazione lineare intera in cui i nodi di un grafo logico, cioè il modello di deep learning, vengono assegnati ai nodi di un grafo fisico, cioè la rete di server, rispettando un

rumore massimo di quantizzazione e un consumo massimo sul device che fa partire l’inferenza. Nella modellazione del problema di ottimizzazione sono stati tenuti in considerazione il tempo di calcolo del modello su un server, la trasmissione tra i server e gli eventuali guadagni ottenuti dalla quantizzazione. Risolto il problema, è stato fatto un post processing della soluzione al fine di costruire un grafo di componenti, cioè di sotto modelli, che fosse aciclico e che permettesse la costruzione di un piano per l’inferenza distribuita nell’edge-cloud continuum.

Per poter testare il tutto, è stata costruita un’architettura di sistema comprendente, tra le altre cose, una componente per il profiling generico del modello e per il profiling dell’esecuzione del modello sul server, così come delle componenti per l’esecuzione dei sotto modelli ottenuti dalla generazione del piano di inferenza.

I risultati sperimentali hanno mostrato come l’ottimizzazione riesca spesso a predire bene il tempo di inferenza del modello oppure a sovrastimarla. L’uso della quantizzazione si è rivelato fondamentale non solo per ridurre il tempo di esecuzione di alcuni livelli, ma anche per ridurre la dimensione dei tensori intermedi e permettere l’offloading della computazione su server diversi. I risultati simulati del consumo energetico mostrano come il sistema riesca a ridurre il consumo energetico complessivo per l’inferenza.

Tuttavia, il sistema presenta alcune limitazioni, come il problema di scalabilità nel calcolo della soluzione ottima, la mancata gestione del cambio delle condizioni e il basso numero di livelli considerati per la quantizzazione.

A partire da questo lavoro, sono molteplici le strade che si possono esplorare per estenderlo. L’implementazione di strategie per la costruzione di piani adattativi e/o alternativi permetterebbe di gestire il cambiamento delle condizioni di rete e del numero di server, mentre uno studio più esteso della modellazione del rumore di

quantizzazione permetterebbe di aumentare il numero di livelli quantizzabili, aprendo la strada a nuove configurazioni di ottimizzazione. Parallelamente a quest'ultimo, l'integrazione di classi di richiesta e di strategie di early-exit potrebbe risultare nella costruzione di un ventaglio di piani che potrebbero essere usati nelle situazioni più disparate. Da ultimo, trovare un'euristica veloce per la risoluzione del problema permetterebbe di introdurre il parallelismo nella formulazione di quest'ultimo, portando ad una riduzione dell'errore nei casi in cui la predizione non si rivela molto precisa.

In conclusione, la soluzione sviluppata ha dimostrato l'efficacia di strategie di approssimazione di modelli di deep learning nel contesto del deployment nell'edge-cloud continuum e si propone come un punto di partenza per l'estensione ad altre strategie di approssimazione e a situazioni di variabilità delle condizioni di deployment.

Bibliografia

Il doi non è indicato in maniera consistente per tutti i riferimenti. Puoi anche toglierlo se preferisci, oppure aggiungerlo dove manca

- [1] Y. Matsubara, M. Levorato, and F. Restuccia, “Split computing and early exiting for deep learning applications: Survey and research challenges,” *ACM Computing Surveys*, vol. 55, no. 5, p. 1–30, Dec. 2022. [Online]. Available: <http://dx.doi.org/10.1145/3527155>
- Non è indicata la sede della pubblicazione (la conferenza ICDCS)**
- [2] M. Yu, Z. Jiang, H. C. Ng, W. Wang, R. Chen, and B. Li, “Gillis: Serving large neural networks in serverless functions with automatic model partitioning,” Jul. 2021. [Online]. Available: <http://dx.doi.org/10.1109/ICDCS51616.2021.00022>
- [3] L. Zeng, X. Chen, Z. Zhou, L. Yang, and J. Zhang, “Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices,” *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, p. 595–608, Apr. 2021. [Online]. Available: <http://dx.doi.org/10.1109/TNET.2020.3042320>
- [4] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, “A survey of quantization methods for efficient neural network inference,” in *Low-power computer vision*. Chapman and Hall/CRC, 2022, pp. 291–326.
- [5] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, “Neurosurgeon: Collaborative intelligence between the cloud and mobile edge,” *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

- [6] M. Mendula, P. Bellavista, M. Levorato, and S. L. de Guevara Contreras, “Furcifer: a context adaptive middleware for real-world object detection exploiting local, edge, and split computing in the cloud continuum,” in *2024 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 2024, pp. 47–56.
- Mancano volume e numero pagine (se non disponibili perché recente, aggiungi come nota "to appear")**
- [7] Y. Huang, L. Zhang, and J. Xu, “Learning the optimal path and dnn partition for collaborative edge inference,” *IEEE Transactions on Mobile Computing*, 2025.
- [8] S. Tuli, G. Casale, and N. R. Jennings, “Splitplace: Ai augmented splitting and placement of large-scale neural networks in mobile edge environments,” *IEEE Transactions on Mobile Computing*, vol. 22, no. 9, pp. 5539–5554, 2022.
- [9] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, “Jointdnn: An efficient training and inference engine for intelligent mobile cloud computing services,” *IEEE transactions on mobile computing*, vol. 20, no. 2, pp. 565–576, 2019.
- [10] Open Neural Network Exchange. [Online]. Available: <https://onnx.ai/>
- [11] sklearn-onnx: Convert your scikit-learn model into ONNX. [Online]. Available: <https://onnx.ai/sklearn-onnx/>
- [12] tf2onnx - Convert TensorFlow, Keras, Tensorflow.js and Tflite models to ONNX. [Online]. Available: <https://github.com/onnx/tensorflow-onnx>
- [13] torch.onnx. [Online]. Available: <https://docs.pytorch.org/docs/main/onnx.html>
- [14] onnx.shape_inference. [Online]. Available: https://onnx.ai/onnx/api/shape_inference.html
- [15] onnx.utils. [Online]. Available: <https://onnx.ai/onnx/api/utils.html>

- [16] G. Jocher and J. Qiu, “Ultralytics yolo11,” 2024. [Online]. Available: <https://github.com/ultralytics/ultralytics>
- [17] G. Bradski, “The OpenCV Library,” *Dr. Dobb’s Journal of Software Tools*, 2000.
- [18] YoloV8 Pre e Post Processing. [Online]. Available: <https://github.com/levipereira/ultralytics/blob/main/examples/YOLOv8-Segmentation-ONNXRuntime-Python/main.py>
- [19] “Roboflow supervision.” [Online]. Available: <https://supervision.roboflow.com/latest/>
- [20] O. R. developers, “Onnx runtime,” <https://onnxruntime.ai/>, 2021, version: x.y.z.
- [21] onnxruntime. [Online]. Available: <https://github.com/microsoft/onnxruntime>
- [22] Graph Optimizations in ONNX Runtime . [Online]. Available: <https://onnxruntime.ai/docs/performance/model-optimizations/graph-optimizations.html>
- [23] Quantize ONNX Models. [Online]. Available: <https://onnxruntime.ai/docs/performance/model-optimizations/quantization.html>
- [24] gRPC. [Online]. Available: <https://grpc.io/>
- [25] A. Hagberg, P. J. Swart, and D. A. Schult, “Exploring network structure, dynamics, and function using networkx,” Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), Tech. Rep., 2008.
- [26] S. Mitchell, M. OSullivan, and I. Dunning, “Pulp: a linear programming toolkit for python,” *The University of Auckland, Auckland, New Zealand*, vol. 65, p. 25, 2011.

- [27] IBM, *IBM ILOG CPLEX Optimization Studio CPLEX User's Manual*, International Business Machines Corporation, 2025, version 22.1.2. [Online]. Available: <https://www.ibm.com/docs/en/icos/22.1.2?topic=optimizers-users-manual-cplex>
- [28] ONNX Runtime Contributors, “Qdq format quantized model is not supported by tensorrt ep,” <https://github.com/microsoft/onnxruntime/issues/14233>, 2023, GitHub issue #14233, Microsoft/onnxruntime.
- [29] ——, “Add support for dynamic input shape for int8 qdq model in tensorrt ep,” <https://github.com/microsoft/onnxruntime/issues/17410>, 2024, GitHub issue #17410, Microsoft/onnxruntime.
- [30] S. Stha. (2023) Quantizing yolov8 models. Medium article. [Online]. Available: <https://medium.com/@sulavstha007/quantizing-yolo-v8-models-34c39a2c10e2>
- [31] G. Russo Russo, V. Cardellini, F. L. Presti, and M. Nardelli, “Towards a security-aware deployment of data streaming applications in fog computing,” *Fog/Edge Computing For Security, Privacy, and Applications*, pp. 355–385, 2021.
- [32] C. E. Ferreira, A. Martin, C. C. de Souza, R. Weismantel, and L. A. Wolsey, “The node capacitated graph partitioning problem: a computational study,” *Mathematical programming*, vol. 81, pp. 229–256, 1998.
- [33] O. Grodzevich and O. Romanko, “Normalization and other topics in multi-objective optimization,” 2006.
- [34] M. Y. Özkaya and Ü. V. Çatalyürek, “A simple and elegant mathematical formulation for the acyclic dag partitioning problem,” *arXiv preprint arXiv:2207.13638*, 2022.

- [35] S. Thanatos, “onnx-tool: Tools for parsing and analyzing onnx models,” <https://github.com/ThanatosShinji/onnx-tool>, accessed: 2025-06-26.
- [36] eivanov89, “The surprising grpc client bottleneck in low-latency networks,” Hacker News, Jul. 2025, discussione su Hacker News. [Online]. Available: <https://news.ycombinator.com/item?id=44658973>