Università degli studi di Torino

SCUOLA DI SCIENZE DELLA NATURA

Corso di Informatica

Tesi di Laurea Triennale

Esecuzione di sistemi di assistenza alla guida basati su intelligenza artificiale su hardware embedded di tipo automobilistico

Relatore:
Prof.
Amparore Elvio Gilberto

Candidato:
Antonelli Filippo

Anno Accademico 2021/2022

Abstract

I sistemi ADAS (Advanced Driver Assistance Systems) sono sistemi software di supporto alla guida, che includono sistemi di controllo stato guidatore, assistenza alle manovre orizzontali e laterali, frenata di emergenza e altre funzioni avanzate. Nell'ambito del progetto europeo NextPerception lo studio si propone di testare ed ottimizzare su un elaboratore embedded automotive-grade (la scheda NVIDIA Jetson Nano) le performance di inferenza di una rete neurale convoluzionale volta alla classificazione dello stato di attenzione del guidatore. Lo studio comprende inoltre lo sviluppo di un componente software predisposto per interfacciarsi ad un sistema di report della dinamica veicolare. In una prima fase sono stati riprodotti i risultati di un benchmark open, dai quali si è dedotta la necessità di sperimentare diverse configurazioni del software di classificazione. Tali configurazioni sono state sperimentate attraverso l'uso di diverse architetture di reti neurali, usando strumenti quali TensorRT e CUDA per il calcolo parallelo e la gestione efficiente della memoria, l'uso di floating point a 32 o 16 bit e varie tipologie di elaborazione del flusso dati. Sono state svolte sia prove di accuratezza dei risultati, che di performance raggiunte, assieme allo studio di profilazione dei tempi e della memoria. Il software di classificazione finale permette l'elaborazione in tempo reale di un flusso video a 20Hz, potendo contare su di una velocità di classificazione che raggiunge i 130 frame al secondo.

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

Indice

1 Introduzione **1**
1.1 Il progetto NextPerception 1
1.2 Il monitoraggio dello stato di distrazione degli automobilisti 1
1.3 Obiettivi dello studio 2
1.4 Sistemi embedded 3

2 La classificazione di immagini **5**
2.1 Reti neurali convoluzionali 5
2.2 Tecnologie utilizzate 7
2.2.1 TensorFlow 7
2.2.2 Keras 7
2.2.3 TensorRT 7
2.2.4 Jetson inference 8
2.2.5 ONNX 9
2.2.6 Numpy 9
2.2.7 OpenCV 10
2.2.8 CUDA 10

3 Riproduzione di benchmark di AutoML **11**

4 Valutazione delle performance di TensorFlow su Jetson Nano **15**
4.1 Benchmark per la valutazione di altre reti neurali e di come la batch size influisce sulle prestazioni 15
4.2 Considerazioni finali 18

5 Ottimizzazione della rete neurale **19**
5.1 Conversione delle reti neurali in formato ONNX 19
5.2 Ottimizzazione delle reti neurali 20
5.2.1 Utilizzo di Jetson Inference 20
5.2.2 Utilizzo di TensorRT 21
5.3 Valutazione delle performance a seguito dell'ottimizzazione 27
5.3.1 Velocità di classificazione 27
5.3.2 Accuratezza di classificazione 32

6 Il software prodotto **37**
6.1 La struttura del software 38
6.1.1 Il cuore del software 38
6.1.2 Il simulatore di veicolo 41
6.1.3 La trasmissione di messaggi 42
6.2 Valutazione del software 42
6.2.1 Tempistiche di l'ottimizzazione 42
6.2.2 Profilazione dei tempi 43
6.2.3 Profilazione della memoria 44
6.3 Conclusioni e sviluppi futuri 45

Bibliography **47**

Capitolo 1

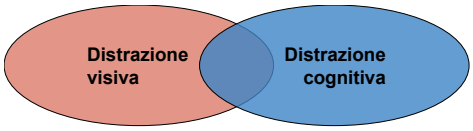
Introduzione

1.1 Il progetto NextPerception

Sempre più frequentemente, affidiamo a dei sistemi intelligenti la responsabilità di prendere delle scelte che influenzano direttamente il nostro benessere. Questo avviene negli ambiti più disparati, tra cui il monitoraggio sanitario e la guida autonoma. I due obiettivi sopra citati sono i punti chiave del progetto NextPerception, che mira a sviluppare tecnologie avanzate per il monitoraggio proattivo dei parametri fisiologici e comportamentali, e sistemi di intelligenza distribuita in grado di prendere decisioni al nostro posto. Con ciò il progetto si focalizza sull'ottimizzazione di questi sistemi, garantendo tempi di risposta rapidi, individuazioni precise e versatilità di applicazione.

1.2 Il monitoraggio dello stato di distrazione degli automobilisti

Uno degli argomenti trattati dal sopra citato progetto è il monitoraggio dello stato di attenzione degli automobilisti.
Possiamo classificare la distrazione secondo due definizioni:



- **distrazione visiva** indica se l'automobilista sembra o meno distratto. Questa condizione può essere rilevata da un flusso di image processing;
- **distrazione cognitiva**, che indica se l'automobilista è mentalmente impegnato in una qualsiasi altra operazione, ad esempio una conversazione.

NextPerception per il riconoscimento della distrazione visiva utilizza un flusso video proveniente da una webcam posizionata all'interno dell'abitacolo. Per il riconoscimento della distrazione cognitiva il progetto si propone di combinare i dati riguardanti la

distrazione visiva con quelli riguardanti la dinamica veicolare. Nella prima versione del software prodotto per il progetto, il componente incaricato al riconoscimento della distrazione visiva elabora il flusso video, calcolando la posizione del viso del conducente e la direzione del suo sguardo. Questa soluzione si è dimostrata inadeguata, in quanto non riesce a rilevare situazioni in cui il conducente ha lo sguardo rivolto verso la strada ma sta svolgendo un compito secondario, ad esempio uno sbadiglio. Inoltre risulta computazionalmente troppo complessa da eseguire sull'hardware a disposizione. Una seconda versione del software, basa il riconoscimento della distrazione su un classificatore in grado di identificare se l'automobilista è concentrato o sta svolgendo un'altra azione durante la guida, classificando ogni fotogramma con una delle seguenti etichette:

- Safe Driving
- Drink
- Hair and makeup
- Look right
- Look left
- Look behind
- Look below
- Operate radio
- Talk cell right
- Talk cell left
- Talk passenger right
- Talk passenger left
- Texting right
- Texting left
- Yawn

1.3 Obiettivi dello studio

Questo studio si propone di verificare su hardware embedded di tipo automotive-grade le performance di esecuzione del sopra citato classificatore, per testarne l'efficienza e per ottimizzarlo. Inoltre si propone di sviluppare un componente software in grado di catturare un flusso video, classificare ogni fotogramma e accoppiare i risultati di classificazione con le informazioni relative alla dinamica veicolare.
Il software sviluppato deve rispettare un protocollo di rete già definito per ricevere la

dinamica veicolare e per trasmettere le informazioni calcolate.
Il riferimento prestazionale da raggiungere per garantire la compatibilità con gli altri componenti software sviluppati da NextPerception è la classificazione di 20 fotogrammi al secondo.

1.4 Sistemi embedded

Si definisce come embedded, un sistema collocato all'interno di una apparato meccanico e/o elettrico, che svolge un lavoro altamente specifico. Dovendo lavorare in sincronia con l'apparato contenitore, è spesso richiesto che il sistema embedded offra una computazione in tempo reale, quindi con ritardi minimi. Nell'ambito dell'automazione automobilistica, sono particolarmente importanti fattori come le dimensioni, il consumo di energia e la latenza di risposta. Nello sviluppo sperimentale di questo progetto, trovano applicazione due schede distinte, la NVIDIA Jetson Nano, utilizzata nella fase di test e di prototipazione del software, e la NVIDIA TX1, utilizzata per valutare le prestazioni del software sviluppato su un sistema hardware diverso.

Figura 1.1: La scheda Jetson Nano

	Jetson Nano	Jetson TX1
GPU	128-core Maxwell	256-core Maxwell
FLOPS	472 GFLOPS	1000 GFLOPS
CPU	Quad-core ARM A57	Quad-core ARM A57
Memory	4 GB 64-bit LPDDR4	4 GB 64-bit LPDDR4
Power	Under 5W	Under 10W
Size	100mm x 80mm x 29mm	170mm x 170mm x16mm

Tabella 1.1: Specifiche delle due schede utilizzate.

Come riportato in tabella 1.1, le due schede hanno la stessa CPU e la stessa quantità di memoria, tuttavia la GPU presente sulla TX1 ha secondo le specifiche il doppio della potenza disponibile sulla Jetson Nano. Le due schede si differenziano inoltre per dimensioni e per consumo energetico. Entrambe le schede si prestano molto bene alle casistiche di "edge computing", per il fattore di forma molto contenuto e per il consumo energetico estremamente basso. In questo studio si verificherà se queste schede offrono prestazioni sufficienti per essere utilizzate nell'ambito della classificazione di immagini in tempo reale.

Capitolo 2

La classificazione di immagini

Per classificare un'immagine si intende un processo in cui un elaboratore attribuisce una o più etichette ad una determinata immagine: in sostanza il computer "riconosce" ciò che è presente nell'immagine.
Un classificatore di immagini potrebbe essere una valida soluzione al problema di valutare se un automobilista è distratto, in quanto offre la possibilità di classificare una serie di azioni che implicano una guida disattenta. Il classificatore su cui questo scritto si concentra è basato sull'intelligenza artificiale, nello specifico basa il suo funzionamento sulle reti neurali convoluzionali.

2.1 Reti neurali convoluzionali

Le reti neurali convoluzionali costituiscono una branca del machine learning e trovano principale applicazione nella classificazione di immagini e nella computer vision. Sono definite come "neurali", in quanto la loro struttura trova fondamento sulla struttura cerebrale umana e ne simula il funzionamento. Queste reti sono composte da una serie di livelli, tra cui un solo livello di ingresso, uno o più livelli intermedi nascosti e un solo livello di uscita. Ogni livello è formato da una serie di neuroni; ogni neurone è connesso ai neuroni dei livelli adiacenti mediante una serie di archi pesati e può essere attivato o meno in base agli output dei neuroni precedenti.

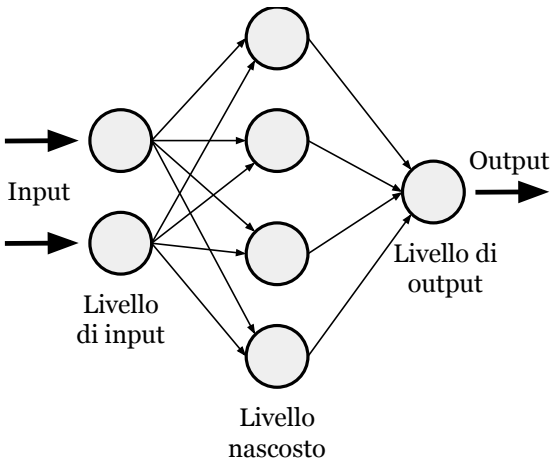


Figura 2.1: Una rappresentazione di rete neurale.

I tre tipi di livello più utilizzati nelle reti neurali convoluzionali sono:

- **Livelli convoluzionali:** sono i livelli computazionalmente più esigenti; si occupano di localizzare caratteristiche rilevanti all'interno dell'immagine in input. Questo processo avviene facendo scorrere un set di filtri (o kernels) lungo l'immagine. Ogni filtro rappresenta una caratteristica da identificare. Questa operazione, comunemente chiamata convoluzione è il cuore pulsante delle reti neurali convoluzionali e produce come risultato una mappa di attivazione per ogni filtro, questa mappa permette di determinare la posizione delle caratteristiche rilevate, al fine di identificare soggetti diversi.

- **Livelli di pooling:** questi livelli conducono il ridimensionamento degli input, applicando un filtro che consente di ridurre le dimensioni dei dati in input, cercando di mantenere il maggior numero possibile di informazioni rilevanti.

- **Livelli FC (fully-connected):** questi livelli si occupano di connettere ogni neurone di un livello con ogni neurone del livello adiacente.

Durante il processo di apprendimento, viene dato alla rete neurale un dataset, contenente molte immagini con le relative etichette. La rete neurale prova quindi a classificare le immagini e successivamente confronta il risultato prodotto con il risultato atteso. I risultati di questo confronto vengono utilizzati dalla rete neurale per modificare i pesi degli archi di connessione e per l'ottimizzazione dei kernels. La rete neurale scorre l'intero dataset più volte; ogni ciclo dell'intero dataset è definito come **epoca di apprendimento**

2.2 Tecnologie utilizzate

2.2.1 TensorFlow

Tensorflow [4] è un framework open source sviluppato da Google, che, tra le svariate funzionalità offerte, si focalizza sul training e sull'inferenza di reti neurali. Tensorflow rappresenta i dati per la computazione sotto forma di matrice multidimensionale, o **tensore**, da cui la libreria prende nome.
Uno dei punti di forza di Tensorflow è la versatilità, infatti è possibile eseguire codice di questa libreria su una ampia gamma di piattaforme, tra cui sistemi embedded, dispositivi mobili, sistemi Cloud ecc. Tensorflow offre la possibilità di salvare una rete neurale durante e dopo la fase di allenamento, sfruttando il formato "Saved model". Così facendo diventa possibile condividere la rete su piattaforme diverse e separare il training, che di solito è una operazione relativamente lunga, in più fasi.

2.2.2 Keras

Keras [7] è una libreria Python open source che offre la possibilià di manipolare reti neurali. Keras agisce come intermediario per Tensorflow, rendendo la sperimentazione e lo sviluppo più veloci e user-friendly. Sebbene le funzionalità di Keras siano ridotte rispetto a quelle di Tensorflow, in alcuni ambiti è preferibile, data la semplicità di utilizzo.
Keras si definisce come:

- **Semplice:** riduce il carico cognitivo richiesto per l'utente, permettendo di concentrarsi su ciò che davvero conta.

- **Flessibile:** grazie a una curva di apprendimento molto ripida, rende possibile realizzare progetti semplici in velocità, ma non esclude la possibilità di sviluppare progetti complessi

- **Potente:** Garantisce stabilità e portabilità; è utilizzato da molte famose compagnie, tra cui YouTube e NASA.

2.2.3 TensorRT

TensorRT [2] è un framework sviluppato da NVIDIA e pensato per migliorare le performance di inferenza. La libreria include un ottimizzatore e un runtime, grazie a cui è possibile ottenere una latenza minima e un troughput considerevole. Inoltre, offre miglioramenti anche in termini di memoria e di consumo energetico, risultando una scelta vincente in ambito "edge AI".

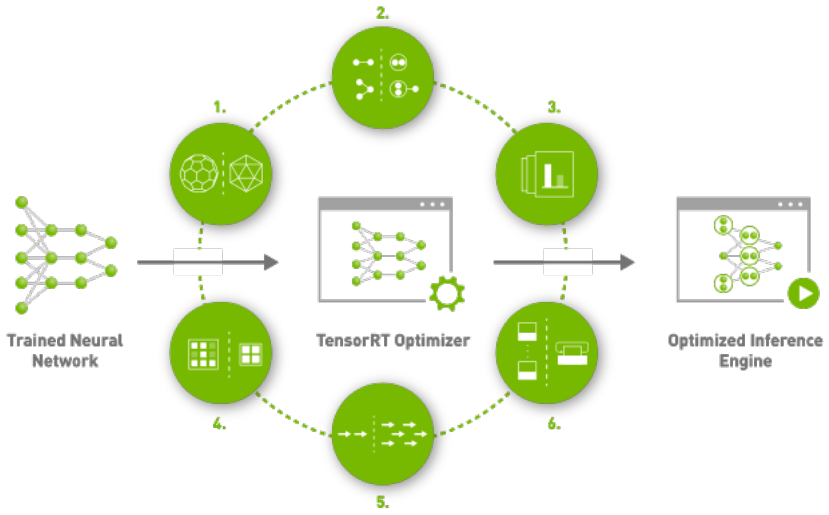


Figura 2.2: Un esempio di workflow con TensorRT

L'ottimizzatore, offre una vasta gamma di funzionalità, tra cui:

- fusione di livelli e tensori;
- eliminazione di livelli ininfluenti;
- calibrazione di precisione (FP32/FP16/INT8);
- auto-tuning dei kernel.

Il risultato di questa ottimizzazione è un motore di inferenza (compatibile con il runtime presente nella libreria), che promette un massiccio salto prestazionale, rendendo la computazione fino a 36 volte più veloce. Grazie a tutte le euristiche fornite, TensorRT guadagna il primo posto in tutti i benchmark MLPERF.[16]

2.2.4 Jetson inference

Jetson inference [1] è un software prodotto da NVIDIA per lo sviluppo efficiente di reti neurali su piattaforme embedded Jetson. Il software include primitive di inferenza tra cui: imageNet per la classificazione di immagini, detectNet per la object detection, poseNet per la pose estimation ecc. Jetson inference sfrutta le funzionalità offerte da TensorRT e le racchiude in uno strumento incredibilmente semplice da utilizzare. Oltre all'inferenza con le reti neurali pre-allenate presenti nella libreria, è possibile effettuare transfer-learning, ri-allenando una rete neurale sfruttando la conoscenza pregressa della rete utilizzata. Queste caratteristiche rendono Jetson inference uno strumento utilissimo nell'approccio al machine learning, altamente funzionale, sebbene limitato all'hardware proprietario.

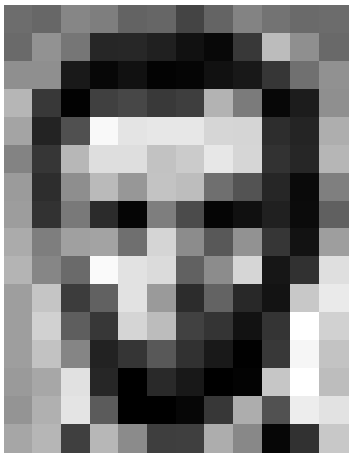
2.2.5 ONNX

ONNX [5] è un formato di rappresentazione per modelli di machine learning, capace di simboleggiare una grande varietà di livelli e architetture. Questo strumento offre una grande funzionalità al mondo del machine learning: un file format universale, che permette di adoperare la stessa rete neurale con una moltitudine di framework, runtimes e compilatori. Tra la vastità di framework compatibili con ONNX, sono presenti anche i sopra citati TensorFlow e Keras. L'utilizzo di questo formato permette l'utilizzo di altri strumenti, tra cui:

- onnxoptimizer [3]: un ottimizzatore che consente di effettuare fine-tuning sulla rete neurale, per ottenere prestazioni migliorate in termini di dimensione, accuratezza, risorse utilizzate e tempistiche.
- strumenti di visualizzazione dell'architettura della rete,ad esempio Zetane[18] che facilitano la comprensione del lavoro svolto dalla rete neurale e permettono di analizzare le caratteristiche intrinseche della rete, ad esempio i pesi degli archi o le strutture dei kernel.
- onnxruntime [8]: Che permette di effettuare inferenze con reti in formato ONNX.

2.2.6 Numpy

Numpy [10] è un progetto open source che introduce le matrici multidimensionali nel linguaggio Python. Tra le funzionalità di Numpy sono presenti strumenti efficienti per effettuare operazioni logico-matematiche, operazioni di ordinamento, manipolazioni di forma e operazioni statistiche. Nell'ambito della classificazione di immagini, numpy rappresenta un potente strumento per rappresentare e pre-processare le immagini da classificare. Infatti risulta estremamente facile e diretto effettuare operazioni di ritaglio e normalizzazione. Numpy risulta uno strumento utile anche in ambito di data-augmentation, rendendo facile lo sviluppo di routine per la rotazione, la traslazione, la distorsione e la modifica cromatica di immagini.



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	76	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	76	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Figura 2.3: Un esempio di immagine rappresentata come matrice

2.2.7 OpenCV

OpenCV [6] (Open source Computer Vision) è una libreria software che offre una infrastruttura per la computer vision e machine perception. Include una moltitudine di algoritmi, grazie a cui sono possibili operazioni come il tracciamento dei volti, l'identificazione di oggetti, l'estrazione di modelli 3D e l'image-stitching. Inoltre, OpenCV rappresenta uno strumento estremamente versatile, rendendo immediato il caricamento e la decodifica di immagini da file e la gestione di flussi video. Questa libreria rappresenta le immagini come matrici multidimensionali, sfruttando la precedentemente citata libreria Numpy.

2.2.8 CUDA

CUDA [13] è un modello di computazione parallela prodotto da NVIDIA, che permette agli sviluppatori di far leva sulla potenza computazionale degli acceleratori GPU. Sono più di 150 le librerie e i framework basati su cuda, tra cui possiamo trovare TensorFlow e ONNX. CUDA trova applicazione in una vastità di scenari distinti, tra cui: medical imaging, finanza computazionale e bioinformatica.

Capitolo 3

Riproduzione di benchmark di AutoML

Una operazione preliminare allo studio è stata la riproduzione dei risultati di un articolo [17] pubblicato da Google AutoML, riguardante EfficientNetV2: una innovativa famiglia di reti neurali che promette una fase di training più rapida e modelli di rete neurale più piccoli, senza sacrificare l'accuratezza. L'articolo prova sperimentalmente grazie ad una serie di benchmark, l'effettiva efficienza delle reti neurali trattate. Particolarmente interessanti sono i test relativi alla velocità di classificazione, utili a capire se una rete neurale sviluppata con questa architettura sarebbe in grado di performare in tempo reale sull'hardware a disposizione. Pertanto, si è riprodotto il relativo benchmark[1] sulla scheda Jetson Nano, utilizzando diverse architetture di rete; i risultati sono successivamente stati confrontati con quelli riportati dall'articolo, ottenuti utilizzando la ben più potente scheda NVIDIA Tesla V100. La scheda utilizzata nell'articolo dispone una potenza di calcolo teorica di 14.13 TFLOPS, circa 28 volte superiore a quella disponile sulla Jetson Nano, che ammonta a circa 472 GFLOPS, inoltre, la scheda V100 possiede 16GB di memoria dedicata, contro i 4GB che la Jetson Nano deve condividere con la CPU. Il benchmark in questione, riportato nel codice 3.1 è scritto utilizzando il linguaggio python e utilizza TensorFlow per gestire la rete neurale. Il flusso del codice, rappresentato nella figura 3.1 può essere descritto come segue: Inizialmente viene caricata in memoria la rete neurale, utilizzando la funzione `build\_tf2\_model()` riportata a riga 3 del codice. Successivamente si valorizza un parametro relativo alla batch size, cioè il numero di immagini da caricare in memoria; ciò avviene con le istruzioni a riga 8. L'utilizzo di batch di immagini permette di ridurre i tempi necessari per spostare le informazioni tra la GPU e la CPU, migliorando il throughput della rete. In seguito viene predisposto il batch di immagini da utilizzare per la classificazione. Avviene poi nelle righe 18 e 19 una fase di "riscaldamento" in cui viene iterata la classificazione; ciò è utile ad evitare oscillazioni dovute a costi one-time (caricamento in memoria centrale, avvio della valutazione della rete neurale in GPU, ecc...). Segue quindi il vero e proprio test: si itera per 10 volte la classificazione del precedentemente composto batch di immagini. Il test è cronometrato e prima della sua terminazione vengono stampate le tempistiche medie di classificazione di un batch di immagini e di

[1] https://github.com/google/automl/blob/master/efficientnetv2/infer.py

un singolo frame. Questa fase è rappresentata dalle righe 23 e 24 del codice. I risultati riportati nell'articolo sono stati prodotti utilizzando batch di 16 immagini, pertanto anche nella riproduzione è stato rispettato questo valore.

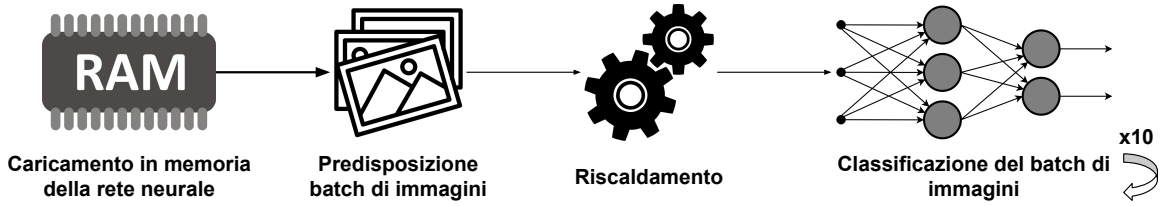


Figura 3.1: Flusso logico del benchmark riprodotto

```
1  def tf2_benchmark():
2    model = build_tf2_model()#funzione che carica in memoria la rete
     neurale e ne stampa l'architettura
3    isize = FLAGS.image_size or model.cfg.eval.isize
4    if FLAGS.export_dir:
5      tf.saved_model.save(model, FLAGS.export_dir)
6      model = tf.saved_model.load(FLAGS.export_dir)
7    #parametro che regola la dimensione del batch di immagini
8    batch_size = FLAGS.batch_size
9    data_dtype = tf.float16 if FLAGS.mixed_precision else tf.float32
10   #predisposizione del batch di immagini
11   imgs = tf.ones((batch_size, isize, isize, 3), dtype=data_dtype)
12
13   @tf.function
14   def f(x): #funzione di classificazione
15     return model(x, training=False)
16
17   print('starting warmup.')
18   for _ in range(10): #fase di riscaldamento
19     f(imgs)
20
21   print('start benchmark.')
22   start = time.perf_counter() #avvio cronometro
23   for _ in range(10): #fase di test
24     f(imgs)
25   end = time.perf_counter() #terminazione cronometro
26   inference_time = (end - start) / 10
27
28   print('Per batch inference time: ', inference_time)
29   print('FPS: ', batch_size / inference_time)
```

Codice 3.1: codice del benchmark AutoML

Nella riproduzione del benchmark sono state utilizzate alcune delle reti appartenenti alla famiglia EfficientNet e EfficientNetV2. Queste reti si basano su variazioni di un modello parametrico e si differenziano in base al numero di parametri nelle matrici della rete; ciò influenza molto il tempo necessario per classificare un'immagine. Nella tabella 3.1 sono riportate le reti neurali utilizzate con le relative dimensioni in termini di numero di parametri.

Rete neurale	Numero di parametri
EfficientNet-b4	19M
EfficientNet-b5	30M
EfficientNet-b6	43M
EfficientNet-b7	66M
EfficientNetV2-s	22M
EfficientNetV2-m	54M
EfficientNetV2-l	120M

Tabella 3.1: Reti neurali utilizzate nel benchmark. Un numero minore di parametri indica una rete neurale più piccola

É importante valutare le performance di architetture differenti per confrontare l'andamento delle performance da noi misurate con l'andamento delle performance riportate nell'articolo. Avere più campioni rende il confronto affidabile e permette di comprendere se una delle architetture utilizzate potrebbe essere utilizzata per classificare in tempo reale sulla Jetson Nano. Nelle tabelle 3.2 e 3.3 e sono riportati i risultati del benchmark.

	Tempo per la classificazione di un batch di 16 immagini (s)	
Modello	Jetson Nano	Nvidia Tesla V100
EfficientNet-b4	3,62	0.03
EfficientNet-b5	7,36	0.06
EfficientNet-b6	OutOfMemory	0.097
EfficientNet-b7	OutOfMemory	0.17
EfficientNetV2-s	**2,74**	**0.024**
EfficientNetV2-m	7,68	0.057
EfficientNetV2-l	OutOfMemory	0.098

Tabella 3.2: Tempo di classificazione di un batch di 16 immagini con NVIDIA Jetson Nano e NVIDIA Tesla V100; valori minori corrispondono a prestazioni migliori

	Frame classificati al secondo	
Modello	Jetson Nano	Nvidia Tesla V100
EfficientNet-b4	4,42	533,33
EfficientNet-b5	2,17	266,67
EfficientNet-b6	OutOfMemory	164,95
EfficientNet-b7	OutOfMemory	94,118
EfficientNetV2-s	**5,81**	**666,67**
EfficientNetV2-m	2,08	296,30
EfficientNetV2-l	OutOfMemory	163,27

Tabella 3.3: Numero di classificazioni al secondo sulle schede NVIDIA Jetson Nano e NVIDIA Tesla V100; valori maggiori corrispondono a prestazioni maggiori

I risultati mostrano che i modelli dimensionalmente più grandi non sono supportati dalla scheda Jetson Nano, per via del limitato quantitativo di memoria RAM. Si nota che nonostante la scheda Nvidia Tesla V100 sia estremamente più veloce, nell'ordine di 100 volte, il rapporto prestazionale tra le due schede rimane pressochè costante al variare del modello. É evidente dai risultati che non è possibile utilizzare uno dei modelli testati per la classificazione in tempo reale sulla Jetson Nano, in quanto anche il più veloce tra i modelli sfiora le 6 classificazioni al secondo. Ciò fa emergere la necessità di una ottimizzazione, sia in termini di velocità, sia in termini di memoria.

Capitolo 4

Valutazione delle performance di TensorFlow su Jetson Nano

4.1 Benchmark per la valutazione di altre reti neurali e di come la batch size influisce sulle prestazioni

I risultati precedentemente discussi dimostrano che l'architettura EfficientNetV2 non consente la classificazione in real time sulla scheda Jetson Nano. Considerato che una rete neurale è un approssimatore arbitrario di funzioni universali, è possibile utilizzare una architettura di rete diversa, mantenendo prestazioni accettabili per lo stesso compito di classificazione. Si è quindi ritenuto necessario valutare altri modelli di rete neurale, i seguenti:

Rete neurale	Numero di parametri
ResNet50[11]	23,6M
MobileNetV3 [15]	1,5M
EfficientNetB1	6,6M

Tabella 4.1: Reti neurali utilizzate, un numero di parametri maggiore indica una dimensione maggiore della rete

Si è quindi scritto un benchmark molto simile a quello riprodotto nel capitolo precedente, che ha lo scopo di valutare le performance di classificazione delle reti neurali indicate nella tabella 4.1. Il benchmark prodotto consente inoltre di valutare come la dimensione del batch di immagini influisce sulla velocità di classificazione. È obiettivo di questo benchmark dimostrare sperimentalmente che una batch size grande permetta una classificazione più veloce. Il codice del benchmark realizzato è riportato nel codice 4.1 e può essere descritto come segue: Inizialmente, servendosi delle funzionalità della libreria Keras viene caricata in memoria una delle reti neurali,questa operazione avviene nelle righe 28,33,38 del codice. Successivamente viene richiamata la funzione `batch_size_benchmark()` che varia la dimensione del batch e richiama la funzione `benchmark()`, dove avviene il vero e proprio test. Le dimensioni di batch testate sono:

1, 4, 8, 12, 16 immagini. La funzione `benchmark()` è composta da una fase di riscaldamento e dall'effettiva fase di test, in cui viene classificato l'intero batch di immagini per 10 volte. Il tempo totale di classificazione viene cronometrato e viene calcolato il tempo medio di classificazione per frame.

```
def benchmark(model,batch_size,shape):
    imgs = tf.ones((batch_size, shape[0], shape[1], 3), dtype=tf.
   float16) #predisposizione del batch di immagini
    @tf.function
    def f(x):#funzione di classificazione
        return model(x, training=False)
    #fase di riscaldamento
    for _ in range(10):
        f(imgs)
    #fase di test
    start = time.perf_counter()#avvio del cronometro
    for _ in range(10):
        f(imgs)
    end = time.perf_counter()#terminazione del cronometro
    per_frame_inference_time = (end - start) / (10*batch_size)
    return per_frame_inference_time

#funzione che esegue il benchmark variando la dimensione del batch
def batch_size_benchmark(model,model_name,img_shape):
    for batch_size in range(0,17,4): #ciclo che permette di variare la
     dimensione del batch, le dimensioni testate sono: 1,4,8,12,16
        if(batch_size==0):
            #chiamata alla funzione di benchmark e stampa risultati
            print(benchmark(model,1,img_shape))
        else:
            #chiamata alla funzione di benchmark e stampa risultati
            print(benchmark(model,batch_size,img_shape))

#caricamento in memoria della rete ResNet50
resnet = ResNet50(weights='imagenet')
#esecuzione del benchmark
batch_size_benchmark(resnet,'ResNet50',(224,224))
del resnet
#caricamento in memoria della rete MobileNetV3
mobnet = MobileNetV3(weights='imagenet')
#esecuzione del benchmark
batch_size_benchmark(mobnet,'MobileNetV3',(224,224))
del mobnet
#caricamento in memoria della rete EfficientNetB1
efficientnet = EfficientNetB1(weights='imagenet')
#esecuzione del benchmark
batch_size_benchmark(efficientnet,'EfficientNetB1',(240,240))
del efficientnet
```

Codice 4.1: codice del benchmark realizzato

Rete neurale	dimensione del batch di immagini 1	4	8	12	16
ResNet50	0.157	0.081	0.069	0.064	0.059
MobileNetV3	**0.027**	**0.022**	**0.021**	**0.021**	**0.021**
EfficientNetB1	0.067	0.052	0.050	0.049	0.049

Tabella 4.2: Tempo di classificazione per singola immagine misurato in secondi in relazione alla dimensione del batch e alla rete neurale utilizzata; valori minori corrispondono a valutazioni più rapide dell'input.

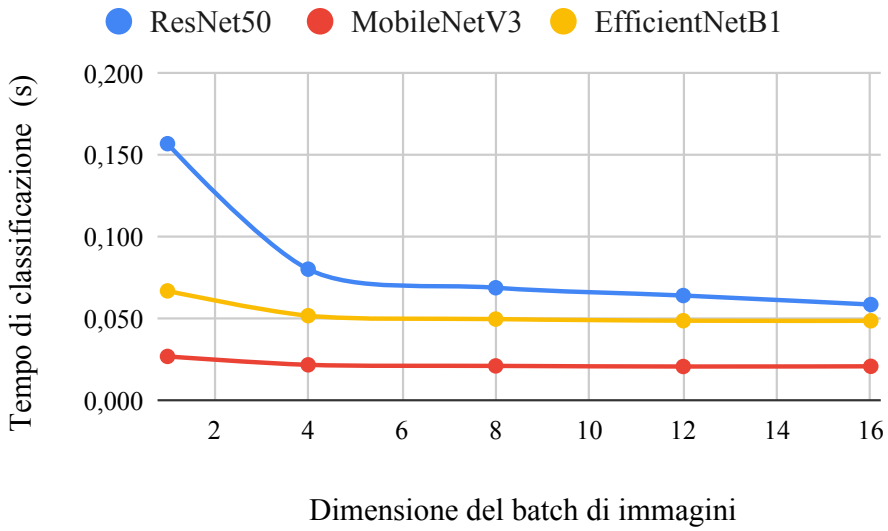


Figura 4.1: Grafico rappresentante il tempo di classificazione per singola immagine, in relazione alla dimensione del batch e alla rete neurale utilizzata.

Rete neurale	dimensione del batch di immagini 1	4	8	12	16
ResNet50	6.357	12.407	14.441	15.506	16.954
MobileNetV3	**36.667**	**45.189**	**46.557**	**47.394**	**47.123**
EfficientNetB1	14.853	19.168	19.963	20.362	20.389

Tabella 4.3: Immagini classificate per secondo in relazione alla dimensione del batch e alla rete neurale utilizzata; valori maggiori corrispondono a prestazioni migliori

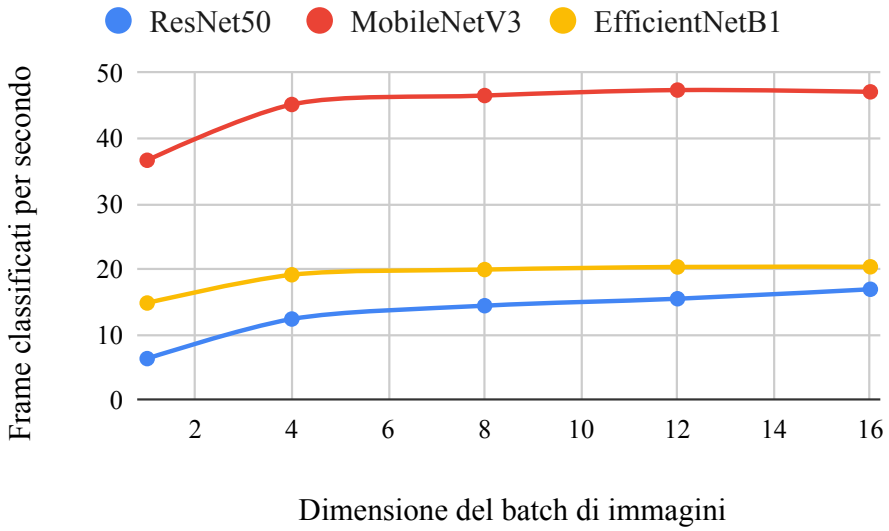


Figura 4.2: Grafico rappresentante il numero di fotogrammi classificati al secondo, in relazione alla dimensione del batch e alla rete neurale utilizzata

I risultati riportati all'interno delle tabelle 4.2 e 4.3 evidenziano che in generale classificare un batch di immagini è più efficiente di classificare una singola immagine. Si nota che utilizzare un batch di immagini rende la classificazione più rapida, in particolare:

- circa 1,6 volte più rapida se si utilizza un batch di 4 immagini
- circa 1,8 volte più rapida se si utilizza un batch di 8 immagini
- circa 1,9 volte più rapida se si utilizza un batch di 12 immagini
- circa 1,95 volte più rapida se si utilizza un batch di 16 immagini

Dal grafico riportato in figura 4.1, si nota che la curva di miglioramento è più ripida nel passaggio tra immagine singola e batch da 4 immagini, mentre per le dimensioni superiori a 4 la pendenza decresce. La rete ResNet50 subisce l'incremento prestazionale maggiore con l'utilizzo di batch di immagini.

4.2 Considerazioni finali

In questo esperimento, si è riusciti a raggiungere la soglia delle 20 classificazioni al secondo, necessarie per analizzare un flusso video in tempo reale, ed essenziali per garantire la compatibilità con gli altri componenti software di NextPerception. Tuttavia, l'utilizzo di batch di immagini introduce una latenza nella classificazione, dovuta al tempo necessario per acquisire e classificare ogni fotogramma. Facendo un esempio, classificando un batch di 4 immagini, prima di ottenere i risultati relativi alla prima immagine, si dovrebbe attendere di acquisire e classificare altre 3 immagini. Un ritardo così importante non è ammissibile nel contesto dell'applicazione da realizzare, per tanto si valuteranno in seguito soluzioni alternative di ottimizzazione.

Capitolo 5

Ottimizzazione della rete neurale

Considerati i risultati degli esperimenti precedentemente svolti, quasi la totalità delle reti neurali testate non permette la classificazione in tempo reale sulla scheda Jetson Nano. Pertanto, in questo capitolo verrà affrontata l'ottimizzazione delle reti neurali, nel tentativo di raggiungere un livello prestazionale sufficiente. Le reti neurali utilizzate in questa fase sono quelle utilizzate nel benchmark trattato nel precedente capitolo. Gli strumenti di ottimizzazione che saranno utilizzati sono quelli forniti da TensorRT e da Jetson Inference.

5.1 Conversione delle reti neurali in formato ONNX

Le reti neurali utilizzate in precedenza sono salvate nel formato utilizzato da Keras: hdf5. Questo formato non è compatibile con gli ottimizzatori. Pertanto si è valutato di utilizzare il formato ONNX, compatibile con entrambi gli strumenti di ottimizzazione. La conversione di formato è stata realizzata servendosi della libreria tf2onnx [14], che offre la funzione `convert.from_keras()`. Questa funzione, dato in input un modello Keras e un percorso di output, salva su disco il modello convertito. Successivamente, il modello ONNX così prodotto viene modificato, nello specifico, si altera il livello di input, escludendo la possibilità di utilizzare i batch di immagini, che come già dimostrato non sono utili al fine del progetto e necessitano di profili di ottimizzazione personalizzati. I modelli così convertiti sono quindi pronti ad essere sottoposti al processo di ottimizzazione.

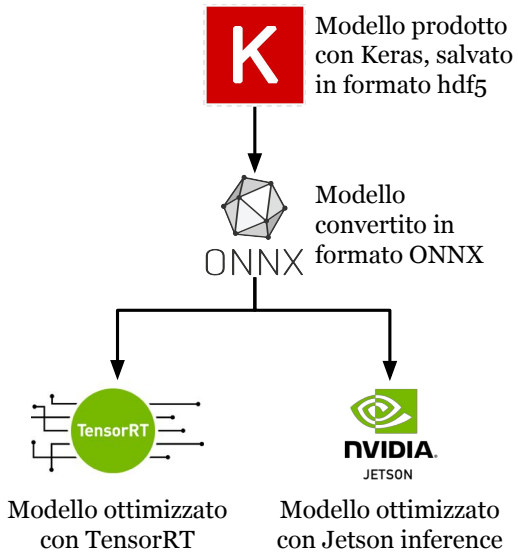


Figura 5.1: Flusso del lavoro nella fase di ottimizzazione

5.2 Ottimizzazione delle reti neurali

Il modello convertito viene quindi sottoposto all'ottimizzazione tramite due strumenti distinti: Jetson.inference e TensorRT. Questi strumenti semplificano la struttura della rete in base ad una serie di euristiche che considerano sia l'hardware utilizzato, sia il tipo di livelli presenti nella rete.

5.2.1 Utilizzo di Jetson Inference

Ottimizzazione e caricamento in memoria della rete

Il processo di ottimizzazione è estremamente semplice da svolgere se si utilizza Jetson Inference e viene avviato in automatico quando viene richiamata la funzione `jetson.inference.imageNet()`; questa funzione ottimizza e carica in memoria la rete, richiedendo i seguenti parametri:

- Il percorso relativo al file ONNX contenente la **rete neurale**
- Il percorso relativo al file TXT contenente le **etichette delle classi**
- Il nome del **livello di input** della rete neurale
- Il nome del **livello di output** della rete neurale

A questi parametri è stato aggiunto modificando il codice sorgente della libreria un parametro che controlla la **precisione di calcolo** della GPU, che altrimenti è impostata di default sulla più veloce tra le precisioni disponibili. Il processo di ottimizzazione produce il motore di inferenza con estensione `.engine` che viene salvato all'interno della cartella contenente la rete neurale. L'ottimizzazione viene avviata solo se non esiste un file `.engine` precedentemente creato per la rete neurale da ottimizzare, altrimenti viene semplicemente caricato in memoria il motore di inferenza. Questo meccanismo permette di ridurre notevolmente il tempo di avvio del programma.

Classificazione di immagini

Il procedimento per effettuare la classificazione può invece essere descritto come segue:

- Viene caricata in memoria GPU l'immagine da classificare; questo passaggio può essere svolto in diverse modalità:
  - **caricamento da disco**, servendosi della funzione `jetson.utils.loadImage()` che richiede il percorso dell'immagine da caricare e un descrittore di formato (ad esempio `rgb32f` per caricare un'immagine con i canali rosso giallo e blu e i valori interpretati in floating point a 32 bit);
  - **caricamento da numpy**, servendosi della funzione `jetson.utils.cudaFromNumpy()`, che richiede un array multidimensionale numpy contenente l'immagine. Questa modalità di caricamento è utile se si intende applicare una pre-elaborazione all'immagine utilizzando numpy;

  - **caricamento da uno stream video**, servendosi della classe `jetson.utils.videoSource`, che offre la funzione `Capture()`. Questa funzione cattura e carica in memoria un fotogramma dal flusso video. Lo stream video può essere un file residente su disco o un flusso proveniente da una webcam.

- Se necessario viene applicata una **pre-elaborazione** all'immagine; la libreria offre molte funzionalità di pre-elaborazione; l'unica utilizzata nel progetto è `jetson.utils.cudaCrop()` che consente di ritagliare l'immagine.

- Viene eseguita la vera e propria classificazione, servendosi della funzione `jetson.inference.Classify()`, che richiede un'immagine precedentemente caricata e le sue dimensioni, questa funzione restituisce una tupla contenente gli indici delle classi e le relative confidenze di classificazione.

Come si può notare, la libreria fornisce tutte le funzionalità necessarie per classificare e manipolare immagini. Pertanto si dimostra estremamente intuitiva da utilizzare.

5.2.2 Utilizzo di TensorRT

TensorRT si è dimostrato decisamente meno semplice da utilizzare, per via della altissima personalizzazione, che alza molto la complessità del codice. In questo caso è stato necessario definire una classe python chiamata `npClassifier` che sfruttando la libreria TensorRT, offre tutte le funzionalità per caricare, ottimizzare ed utilizzare una rete neurale. Questo approccio rende l'utilizzo della libreria molto più diretto e sostenibile. L'utilizzo della classe può essere descritto come segue:

Creazione di un'istanza di npClassifier

Alla creazione di un oggetto `npClassifier`, viene ottimizzata e caricata in memoria una rete neurale salvata in formato ONNX. Il costruttore di `npClassifier`, riportato nel codice 5.1, richiede due parametri: il percorso in cui la rete neurale è salvata e un flag `FP_16` che permette di scegliere la precisione di calcolo. Il parametro `FP_16` è opzionale e il suo valore di default è `False`. Le operazioni che avvengono durante l'esecuzione del costruttore sono le seguenti:

- viene definito un logger: uno strumento utilizzato per stampare a console gli eventi relativi allo stato della rete. (ottimizzazione, caricamento, classificazione, etc.);

- vengono valorizzate delle variabili contenenti i percorsi relativi alla rete da caricare e alla rete ottimizzata;

- viene eseguito il metodo `load_model()` che se necessario ottimizza la rete e la carica in memoria;

- viene eseguito il metodo `alloc_buf()` che crea dei buffer di memoria necessari al funzionamento del classificatore.

```
def __init__(self,onnx_path,FP_16=False):#costruttore della classe
    #definizione di un logger
    self.trtLogger= trt.Logger(trt.Logger.WARNING)
    self.explicit_batch = 1 << (int)(trt.NetworkDefinitionCreationFlag
   .EXPLICIT_BATCH)
    self.onnx_path=onnx_path #percorso del modello ONNX
    if(FP_16):#scelta della precisione
            self.engine_path=onnx_path+'FP_16.engine'
        else:
            self.engine_path=onnx_path+'FP_32.engine'
    #funzione che carica in memoria l'engine
    self.engine=self.__load_model(FP_16)
    self.context=self.engine.create_execution_context()
    #definizione dei buffer di memoria GPU
    self.in_cpu, self.out_cpu, self.in_gpu, self.out_gpu, self.stream,
   self.cropped = self.__alloc_buf()
```

Codice 5.1: costruttore della classe npClassifier

Caricamento e ottimizzazione della rete

Il caricamento e l'ottimizzazione avvengono nel costruttore, tramite il metodo `load_model()`. Questo metodo si appoggia su due ulteriori metodi: `load_engine()` e `build_and_save_engine()`; il codice relativo al caricamento e all'ottimizzazione della rete è riportato nel codice 5.2 ed così descrivibile:

- si controlla che il file contenente la rete neurale sia un file ONNX, a riga 3 del codice, in caso negativo si restituisce un messaggio di errore e si termina l'esecuzione;

- si controlla se esiste una versione ottimizzata della rete neurale e che la sua data di creazione sia successiva a quella della rete neurale di partenza:

  - nel caso in cui ci sia una versione ottimizzata si procede a caricarla in memoria, grazie al metodo `load_engine()`

  - altrimenti si procede all'ottimizzazione con il metodo `build_and_save_engine()`, che carica la rete salvata nel file ONNX, svolge la sua ottimizzazione, salva su file la rete ottimizzata e la carica in memoria.

La precisione di calcolo della rete ottimizzata è determinata dal valore del parametro `FP_16`; nel caso in cui questo parametro sia valorizzato a `True`, la rete ottimizzata avrà precisione floating point a 16 bit, altrimenti avrà precisione floating point a 32 bit. Il meccanismo di caricamento realizzato permette di svolgere l'ottimizzazione solo se necessario e non ad ogni esecuzione del codice.

```
#funzione per l'ottimizzazione e per il caricamento della rete neurale
def __load_model(self,FP_16):
        if not (self.onnx_path.endswith('.onnx')):
            print('Unsupported model, provide an onnx path')
            exit()
        #controlla se e' necessario svolgere l'ottimizzazione
        if ((not os.path.exists(self.engine_path)) or os.path.getmtime
    (self.engine_path)<os.path.getmtime(self.onnx_path)):
            print('Generating engine')
            #in questo caso si ottimizza e carica in memoria la rete
            return self.__build_and_save_engine(FP_16)
        else:
            print('Existing engine found! Loading from file')
            #in questo caso si carica in memoria la rete gia'
    ottimizzata
            return self.__load_engine()
#funzione che ottimizza la rete e la carica in memoria
def __build_and_save_engine(self,FP_16):
    with trt.Builder(self.trtLogger) as builder, \
        builder.create_network(self.explicit_batch) as network, \
        trt.OnnxParser(network, self.trtLogger) as parser:
        builder.max_batch_size = 1 #
        print("loading model")
        parser.parse_from_file(self.onnx_path) #caricamento dell' ONNX
        print("network.num_layers", network.num_layers)
        print("building engine")
        config = builder.create_builder_config()
        profile = builder.create_optimization_profile()
        profile.set_shape("input_1", shape, shape, shape)
        config.add_optimization_profile(profile)
        config.max_workspace_size = 1 << 20
        if FP_16: #scelta della precisione di calcolo
                config.set_flag(trt.BuilderFlag.FP16)
        serialized_engine = builder.build_serialized_network(network,
    config) #ottimizzazione della rete neurale
        with open(self.onnx_path+'.engine', 'wb') as f:
            #salvataggio dell'engine ottimizzato
            f.write(serialized_engine)
        runtime = trt.Runtime(self.trtLogger)
        #caricamento in memoria della rete neurale
        engine = runtime.deserialize_cuda_engine(serialized_engine)
        return engine
#funzione per il caricamento in memoria di una rete ottimizzata
def __load_engine(self):
    runtime=trt.Runtime(self.trtLogger)
    with open(self.engine_path,'rb') as f:
        serialized_engine=f.read()#lettura della rete neurale salvata
    #caricamento in memoria della rete ottimizzata
    engine=runtime.deserialize_cuda_engine(serialized_engine)
    return engine
```

Codice 5.2: metodi per l'ottimizzazione e il caricamento in memoria della rete

Preallocazione dei buffer di memoria

Durante l'esecuzione del costruttore di npClassifier è necessario allocare dei buffer di memoria, essenziali per il funzionamento del classificatore. L'allocazione avviene all'interno del metodo `alloc_buf()`, riportato nel codice 5.3; i buffer servono per trasferire informazioni tra la CPU e la GPU, in particolare:

- **out_cpu**: per trasferire le immagini da classificare dalla CPU alla GPU;
- **in_gpu**: per accogliere le immagini da classificare, provenienti dalla CPU e dirette alla GPU;
- **out_gpu**: per trasferire i risultati di classificazione dalla GPU alla CPU;
- **crop**: per accogliere le immagini da classificare opportunamente tagliate, provenienti dalla CPU e dirette alla GPU.

```
1 #funzione che crea dei buffer di memoria CPU e GPU utili per caricare
    le immagini e scaricare i risultati di classificazione
2 def __alloc_buf(self):
3      # buffer CPU
4      h_in_size = trt.volume(self.engine.get_binding_shape(0))
5     h_out_size = trt.volume(self.engine.get_binding_shape(1))
6     h_in_dtype = trt.nptype(self.engine.get_binding_dtype(0))
7     h_out_dtype = trt.nptype(self.engine.get_binding_dtype(1))
8     in_cpu = cuda.pagelocked_empty(h_in_size, h_in_dtype)
9     out_cpu = cuda.pagelocked_empty(h_out_size, h_out_dtype)
10    # buffer GPU
11    in_gpu = cuda.mem_alloc(in_cpu.nbytes)
12    out_gpu = cuda.mem_alloc(out_cpu.nbytes)
13    stream = cuda.Stream()
14    cropped=gpuarray.empty((240,240,3),np.dtype(np.float32))
15    return out_cpu, in_gpu, out_gpu, stream,cropped
```

Codice 5.3: metodo che alloca i buffer di memoria

Caricamento di immagini in memoria GPU

Per il caricamento in memoria GPU delle immagini, la classe npClassifier utilizza due modalità:

- sfrutta la funzione `cuda.memcpy_htod()` per copiare un' immagine in forma di tensore numpy all'interno del buffer `in_gpu`;
- sfrutta la funzione `copy_and_crop_centered()` presente nel package della classe, per copiare all'interno del buffer `in_gpu` solo la porzione di immagine da classificare. La porzione da classificare viene estratta centralmente dall'immagine di input e ha le dimensioni previste dall'input della rete neurale. Questa funzione prevede inoltre un parametro opzionale `stream`, che può essere utilizzato per effettuare il taglio e il caricamento in memoria in maniera asincrona. Questa funzione è riportata nel codice 5.4 e basa il suo funzionamento su `copy_and_crop()`.

`copy_and_crop()` contiene una chiamata alla funzione `cuda.Memcpy2D` [12], specializzata nello spostare dati con un layout bitmap. Questa funzione dato un numero di pixel da scartare in verticale ed in orizzontale, scorre l'immagine di input a livello di byte ed effettua la copia nel buffer di destinazione.

Per confrontare le performance temporali delle due funzioni, si è scritto un test in cui vengono caricate in memoria GPU 1000 immagini con entrambe le funzioni e viene calcolato il tempo medio di caricamento; i risultati sono riportati nella tabella 5.1.

	tempo impiegato(s)
copy_and_crop()	**1.41e-4**
cuda.memcpy_htod()	3.40e-4

Tabella 5.1: Tempistiche medie per il caricamento in memoria GPU di una immagine. Un tempo inferiore corrisponde a prestazioni migliori

Dai risultati del test, la funzione `copy_and_crop_centered()` risulta preferibile alla funzione `cuda.memcpy_htod()`, in quanto fornisce prestazioni temporali nettamente migliori (carica un'immagine circa 2,4 volte più velocemente) e consente di copiare e tagliare l'immagine di input con una sola funzione.

```
1     def copy_and_crop_centered(dst, src,stream=None):
2     #dimensioni dell'immagine da caricare
3     src_h = src.shape[0]
4     src_w = src.shape[1]
5     #dimensioni del buffer su cui copiare l'immagine
6     dst_h = dst.shape[0]
7     dst_w = dst.shape[1]
8     copy_and_crop(dst,src,(src_w-dst_w)/2,(src_h-dst_h)/2,stream)
9
10 def copy_and_crop(dst, src, w_offset, h_offset, stream=None):
11     assert src.dtype == dst.dtype,\
12         "src ({}) and dst ({}) must have the same datatype.".format(
    str(src.dtype), str(dst.dtype))
13     itemsize = np.dtype(src.dtype).itemsize
14     copy = cuda.Memcpy2D()
15     src_on_gpu = isinstance(src, pycuda.gpuarray.GPUArray)
16     dst_on_gpu = isinstance(dst, pycuda.gpuarray.GPUArray)
17     cuda.Memcpy2D()
18     #definizione dei flag che indicano la posizione di sorgente e
    destinazione
19     if src_on_gpu:
20         copy.set_src_device(src.gpudata)
21     else:
22         copy.set_src_host(src)
23     if dst_on_gpu:
24         copy.set_dst_device(dst.gpudata)
25     else:
26         copy.set_dst_host(dst)
27     #definizione della dimensione in byte di ogni elemento dei tensori
28     copy.src_pitch =src.strides[0]
```

```
    copy.dst_pitch =dst.strides[0]
    copy.width_in_bytes = dst.strides[0]
    copy.height = dst.shape[0]
    #definizione degli offset di taglio
    copy.src_x_in_bytes= w_offset*src.strides[1]
    copy.src_y=h_offset
    if stream is None:
        copy(False)#memcpy sincrona
    else:
        copy(stream)#memcpy asincrona

```

Codice 5.4: funzioni per il caricamento di immagini in GPU

Classificazione di immagini

Il processo di classificazione prevede che sia stata caricata in memoria GPU l'immagine da classificare con uno dei metodi precedentemente esposti, prevede inoltre che dopo la classificazione i risultati vengano trasferiti dalla memoria GPU alla memoria CPU. Questo meccanismo è racchiuso nei metodi riportati nel codice 5.5, che permettono di svolgere tutte le operazioni previste per la classificazione con una sola chiamata di funzione. I metodi possono essere descritti come segue:

- `inference()`, utilizzato dagli altri metodi di classificazione.
  Questo metodo sfrutta `copy_and_crop_centered()` per il caricamento in memoria dell'immagine, lancia il meccanismo di classificazione richiamando la funzione `context.execute()` e scarica in memoria centrale i risultati prodotti, utilizzando la funzione `memcpy_dtoh()` offerta dalla libreria pycuda;

- `predict_from_img()`, che permette di classificare un'immagine fornita in forma di tensore numpy;

- `predict_from_path()`, che permette di classificare un' immagine salvata su disco;

- `predict_async()`, che permette di svolgere il caricamento in memoria e la classificazione in maniera asincrona, sincronizzando GPU e CPU solo alla fine della classificazione.

```
#funzione di classificazione
def __inference(self,inputs):
    #caricamento in memoria gpu dell'immagine da classificare
    copy_and_crop_centered(self.cropped_buf,inputs.astype(np.float32))
    #classificazione
    self.context.execute(1, [int(self.in_gpu), int(self.out_gpu)])
    #caricamento in memoria CPU dei risultati di classificazione
    cuda.memcpy_dtoh(self.out_cpu, self.out_gpu)
    return self.out_cpu
#funzione che classifica un'immagine caricata in un array numpy
def predict_from_img(self,img):
    h, w, c = img.shape
```

```
13     if (h<240 or w<240):
14         print("ERR: img is too small")
15         exit()
16     input = img.astype(np.float32)
17     return self.__inference(input)
18 #funzione che classifica un'immagine salvata su disco
19 def predict_from_path(self,img_path):
20     img = cv2.imread(img_filename, cv2.IMREAD_COLOR)
21     return self.predict_from_img(img)
22 #funzione di classificazione asincrona
23 def predict_async(self,img):
24     copy_and_crop(self.cropped_buf, np.float32(img), 40, 0, self.
    stream)
25     self.context.execute_async(1, [int(self.cropped_buf.gpudata), int(
    self.out_gpu)],self.stream.handle, None)
26     cuda.memcpy_dtoh_async(self.out_cpu, self.out_gpu, self.stream)
27     self.stream.synchronize()
28     return self.out_cpu
```

Codice 5.5: metodi per la classificazione

Riscaldamento del classificatore

La classe include un metodo `warmup()` riportato nel codice 5.6, che effettua il riscaldamento del classificatore, classificando un'immagine di prova per 20 volte e stabilizzando le tempistiche di classificazione.

```
1 def warmup(self):
2     size = (w, h, channels) = (240, 240, 3)
3     img = np.zeros(size, np.int8)
4     input = img.astype(np.float32)
5     for i in range(20):
6         _ = self.__inference(input.reshape(-1))
7     del size,img,input
```

Codice 5.6: metodo per il riscaldamento del classificatore

5.3 Valutazione delle performance a seguito dell'ottimizzazione

Per valutare le performance di classificazione delle reti ottimizzate con i metodi precedentemente illustrati, si è scritta una serie di benchmark pensati per misurare le variazioni in termini di velocità e di accuratezza di classificazione.

5.3.1 Velocità di classificazione

Lo scopo principale dell'ottimizzazione svolta è quello di ridurre la tempistica necessaria per la classificazione. Per quantificare i miglioramenti introdotti dall'ottimizzazione si è composto un benchmark riportato nel codice 5.7. La struttura del benchmark è molto simile a quelle viste in precedenza, comprende l'ottimizzazione della rete neurale e il suo

caricamento in memoria, una fase di riscaldamento e l'effettiva fase di test. Nella fase di test, per 1000 iterazioni, un'immagine viene caricata in memoria GPU e classificata. Al termine del benchmark viene stampata una media del tempo di classificazione. Con ogni strumento di ottimizzazione sono state prodotte due ottimizzazioni distinte, che differiscono per la precisione di calcolo. Sono state valutate entrambe le precisioni disponibili sulla GPU della scheda Jetson Nano: floating point a 16 bit (FP_16) e floating point a 32 bit (FP_32). Le reti neurali di partenza sono state utilizzate all'interno del runtime offerto da Keras. Le reti neurali ottimizzate sono state utilizzate all'interno dei runtime dei relativi ottimizzatori. Nella tabella 5.2 sono riportate le tempistiche di classificazione misurate.

```
#BENCHMARK KERAS
@tf.function
def f(x,model):
  return model(x)
def newClassification(frame,model): #funzione di classificazione
  return f(np.array([frame]),model).numpy()
nrep=1000 #numero di iterazioni da svolgere
def benchmark_keras(model_path):
    print('keras load model')
    #caricamento rete neurale
    model = keras.models.load_model(model_path)
    frame=np.zeros((240,240,3),np.float32)
    print('keras warmup')
    #fase di riscaldamento
    for _ in range(20):
        _=newClassification(frame,model)
    #fase di test
    exec_time=time.time()
    print('keras benchmark')
    for _ in range(nrep):
        _=newClassification(frame,model)
    exec_time=time.time()-exec_time
    del model
    return(exec_time) #output dei risultati
print (benchmark_keras(percorso_rete_neurale))

#BENCHMARK TENSORRT
nrep=1000
def benchmark_trt(classifier):
    img=np.zeros((240,240,3),np.float32) #generazione immagine da
   classificare
    classifier.warmup() #fase di riscaldamento
    exec_time=time.time() #avvio cronometro
    #fase di test
    for _ in range(nrep):
        _=classifier.predict_from_img(img)#classificazione
    exec_time=time.time()-exec_time
    return(exec_time) #output dei risultati
#ottimizzazione della rete con precisione FP_16
classifier = npClassifier(percorso_rete_neurale,FP_16=True)
print(benchmark_trt(classifier))
del classifier
```

```
42 #ottimizzazione della rete con precisione FP32
43 classifier = npClassifier(percorso_rete_neurale,FP_16=False)
44 print(benchmark_trt(classifier))
45 del classifier
46
47 #BENCHMARK JETSON INFERENCE
48 #funzione per il caricamento dei parametri necessari a ottimizzare e
     caricare la rete
49 def parseArguments(model_path,precision):
50   return ['./imagenet_benchmark.py',input_source,'--model='+model_path
     ,'--labels='+labels,'--input_blob='+input_blob,'--output_blob='+
     output_blob,'--precision='+precision]
51 #funzione di ottimizzazione e caricamento
52 def initializeNetwork(model_path,precision):
53   arguments = parseArguments(model_path,precision)
54   network = jetson.inference.imageNet(model_path,arguments)
55   return network
56
57 def benchmark_jetson(model_path,precision): #funzione di benchmark
58     #inizializzazione rete
59     network=initializeNetwork(model_path,precision)
60     nrep=1000
61     frame=np.zeros((240,240,3),np.float32)
62     #fase di riscaldamento
63     for _ in range(20):
64         #caricamento immagine in memoria GPU
65         frame=jetson.utils.cudaFromNumpy(frame)
66         _=network.Classify(frame) #classificazione
67     exec_time=time.time()
68     #fase di test
69     for _ in range(nrep):
70         #caricamento immagine in memoria GPU
71         frame=jetson.utils.cudaFromNumpy(frame)
72         _=network.Classify(frame) #classificazione
73     exec_time=time.time()-exec_time
74     del network
75     return(exec_time)#output dei risultati
76
77 print(benchmark_jetson (percorso_rete_neurale,'FP32'))
78 print(benchmark_jetson (percorso_rete_neurale,'FP16'))
```

Codice 5.7: benchmark per la misurazione della velocità di classificazione pre e post ottimizzazione

Rete Neurale	Keras	TensorRT 32FP	TensorRT 16FP	Jetson.inference 32FP	Jetson.inference 16FP
EfficientNetB1	0.08939	0.04361	0.03561	0.04336	0.03537
ResNet50	0.17084	0.05385	0.03039	0.05369	0.03023
MobileNetV3	**0.03128**	**0.00807**	**0.00675**	**0.00775**	**0.00639**

Tabella 5.2: Tempi di classificazione misurati in secondi,relativi alla rete neurale e al runtime utilizzato. Valori inferiori corrispondono a prestazioni migliori.

Rete Neurale	Keras	TensorRT 32FP	TensorRT 16FP	Jetson.inference 32FP	Jetson.inference 16FP
EfficientNet	11.19	22.93	28.08	23.06	28.27
ResNet50	5.85	18.57	32.90	18.63	33.08
MobileNetV3	**31.97**	**123.97**	**148.13**	**129.01**	**156.51**

Tabella 5.3: Immagini classificate al secondo, in relazione alla rete neurale e al runtime utilizzato. Valori maggiori corrispondono a prestazioni migliori

Dai risultati, si può notare un sostanziale incremento delle prestazioni: si raggiunge un throughput sufficiente alla classificazione in tempo reale con le reti EfficientNetB1 e MobileNetV3, mentre i risultati relativi alla rete ResNet50 sono insoddisfacenti se si utilizza la precisione FP32. Le tempistiche risultanti dall'utilizzo di TensorRT e Jetson.inference sono simili, un risultato atteso in quanto Jetson.inference basa il funzionamento dell'ottimizzazione su TensorRT. La precisione di computazione ridotta porta un sensibile miglioramento, nello specifico:

- L'incremento prestazionale medio a seguito dell'ottimizzazione con TensorRT è del 303% nel caso di precisione FP32, del 425% nel caso di precisione FP16.

- L'incremento prestazionale medio a seguito dell'ottimizzazione con Jetson inference è del 309% nel caso di precisione FP32, del 435% nel caso di precisione FP16.

L'architettura di rete più performante in termini di velocità è MobileNetV3, che a seguito dell'ottimizzazione supera abbondantemente la soglia delle 30 classificazioni al secondo. Questa è anche la rete che ha subito il salto prestazionale più importante, con un aumento del throughput di oltre 400%.

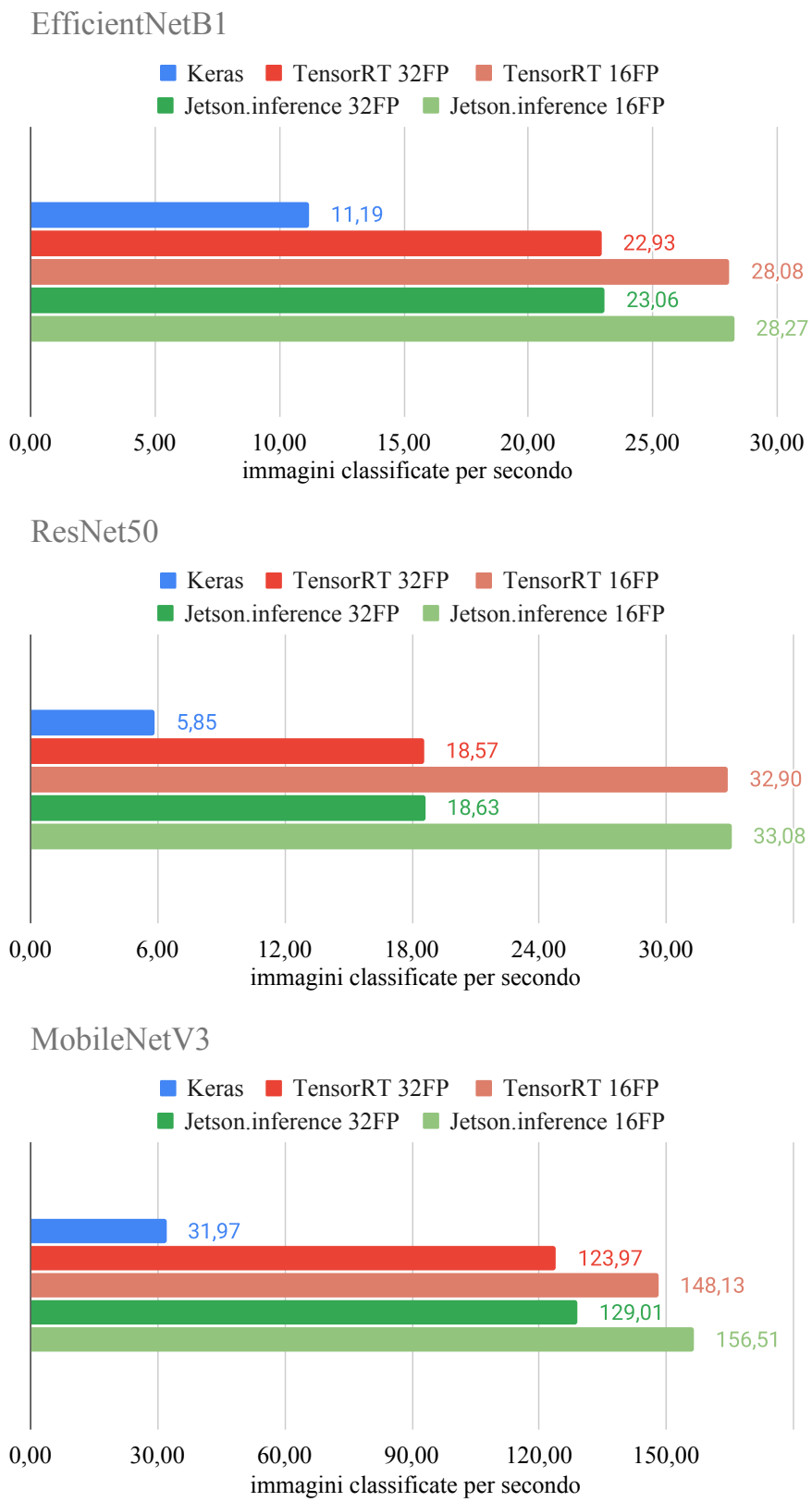


Figura 5.2: Immagini classificate al secondo per le reti neurali testate. I risultati sono relativi alla rete neurale di partenza e alle sue versioni ottimizzate. Valori più grandi corrispondono a prestazioni migliori.

5.3.2 Accuratezza di classificazione

Il processo di ottimizzazione modifica l'architettura delle strutture neurali e potrebbe introdurre errori nella classificazione. È importante che l'ottimizzazione non degradi eccessivamente le performance di classificazione.

Test di classificazione

Un primo test svolto per valutare i risultati dell'ottimizzazione è stata la classificazione di alcune immagini con le reti neurali di partenza, con le reti in formato ONNX e con le reti ottimizzate dai due strumenti. Da questo test è emerso che la rete ottimizzata utilizzando Jetson.inference non classifica correttamente, attribuendo lo stesso risultato ad ogni immagine. Le reti in formato ONNX e le reti ottimizzate con TensorRT producono i risultati attesi. Il risultato relativo all'ottimizzazione con Jetson.inference porta a pensare che questa libreria non riesca a lavorare correttamente con dei modelli personalizzati e supporti esclusivamente i modelli prodotti con gli strumenti offerti dalla libreria stessa. Ciò esclude Jetson.inference dagli sviluppi futuri.

Misurazione dell'accuratezza

Per misurare l'accuratezza di classificazione, si è realizzato un ulteriore test, riportato nel codice 5.8, che consiste nella classificazione del dataset di training della rete e del dataset di valutazione. La struttura del dataset può essere illustrata come segue:

```
Dataset
├── C00
│   ├── user00-f00.png
│   ├── ...
│   ├── user00-f09.png
│   ├── ...
│   └── user29.f09.png
├── ...
└── C14
```

All'interno del dataset sono presenti 15 cartelle, ognuna delle quali contiene le immagini appartenenti alla relativa classe; Dentro ogni cartella sono presenti le immagini di 30 utenti, per ogni utente sono presenti 10 immagini. Le immagini raffiguranti lo user00 costituiscono il dataset di valutazione, le immagini raffiguranti tutti gli altri user costituiscono il dataset di training.

```
model = load_network() #caricamento in memoria della rete neurale
expectedPredicted=np.array([],[])
expected=[]
predicted=[]
expected00=[]
predicted00=[]
#istruzione che permette di scorrere il dataset
for (root, dirs, files) in os.walk(dataset_path, topdown=True):
  print(root)
  className = root[-2:] #estrazione della classe attesa
  for file in files:
    if(file.endswith('.png')):
      if(file.startswith('._')):
        file=file[2:]
      print('PATH: ',os.path.join(root,file))
      #caricamento in memoria dell'immagine
      img=loadImage(os.path.join(root,file))
      results=model.classify(np.array([img])) #classificazione
      exp=from_ITSC_to_CM(int(className))
      pred=np.argmax(results)
      print('expected: ',int(exp),' predicted: ',int(pred),' results: ',results)
      #i risultati relativi allo user00 fanno parte del dataset di valutazione
      if('user00' in file):
        print('User00')
        expected00.append(int(exp))
        predicted00.append(int(pred))
      else: #gli altri risultati fanno parte del dataset di training
        print('Not User00')
        expected.append(int(exp))
        predicted.append(int(pred))
      print('--------------------------------')
accuracy_training_set=np.sum(np.equal(expected, predicted)) / len(expected) #calcolo dell'accuratezza per il dataset di training
accuracy_eval_set=np.sum(np.equal(expected00, predicted00)) / len(expected00) #calcolo dell'accuratezza per il dataset di valutazione
```

Codice 5.8: Benchmark per la valutazione dell'accuratezza delle reti neurali

Il codice utilizza la funzione python `os.walk()` per percorrere la cartella contenente i dataset. Tutte le immagini del dataset vengono classificate con la funzione di classificazione relativa alla rete neurale utilizzata; i risultati vegono inseriti in un vettore. Viene valorizzato un altro vettore che contiene le etichette di classe attese per ogni immagine. L'accuratezza di ogni rete viene calcolata come il rapporto tra il numero di classificazioni effettuate correttamente e il numero totale di classificazioni effettuate. Nelle tabelle 5.4 e 5.5 sono riportati i risultati relativi ai valori di accuratezza delle reti neurali utilizzate.

Rete Neurale	Keras	OnnxRuntime	TensorRT FP32	TensorRT FP16
EfficientNetB1	0.7214	0.7214	0.7214	0.7218
ResNet50	0.6837	0.6837	0.6837	0.6828
MobileNetV3	**0.9336**	**0.9336**	**0.9336**	**0.9336**

Tabella 5.4: Valori di accuratezza delle reti neurali sul dataset di training; un valore più alto corrisponde a prestazioni di classificazione migliori

Rete Neurale	Keras	OnnxRuntime	TensorRT FP32	TensorRT FP16
EfficientNetB1	0.3533	0.3533	0.3533	0.3467
ResNet50	0.4267	0.4267	0.4267	0.4267
MobileNetV3	**0.5400**	**0.5400**	**0.5400**	**0.5400**

Tabella 5.5: Valori di accuratezza delle reti neurali sul dataset di valutazione; un valore più alto corrisponde a prestazioni di classificazione migliori

I punteggi di accuratezza relativi alle reti originali e a quelle convertite in ONNX sono sovrapponibili, un risultato che dimostra che la conversione in ONNX non altera le performance di classificazione. I risultati prodotti dalla rete ottimizzata con TensorRT sono eccellenti, riportando la stessa accuratezza del modello originale se la computazione avviene a 32 bit. La semplificazione della computazione a 16 bit comporta un lieve discostamento dell'accuratezza, nell'ordine di 0,1 punti percentuali. La rete che riporta i punteggi di accuratezza massimi risulta essere MobileNetV3, che è oltretutto la rete più veloce nella classificazione, ciò la porta ad essere la rete favorita ai fini del progetto.

Valutazione dell'errore introdotto

Un ulteriore benchmark prodotto riguarda la valutazione dell'errore introdotto con la semplificazione della precisione di computazione. Per misurare questo errore è stato utilizzato il benchmark riportato nel codice 5.9, che classifica l'intero dataset con la rete ottimizzata a precisione completa (FP_32) e con la rete ottimizzata a precisione ridotta (FP_16). I risultati di classificazione ottenuti vengono utilizzati per il calcolo dell'errore assoluto e dell'errore relativo. L'errore assoluto è calcolato come il valore assoluto della differenza tra i valori prodotti dalle due reti. L'errore relativo è calcolato come il rapporto tra l'errore assoluto e il valore prodotto dalla rete a precisione completa. Questi errori vengono calcolati per ogni risultato di classificazione; sono particolarmente rilevanti i valori massimi e medi degli errori, riportati all'interno della tabella 5.6

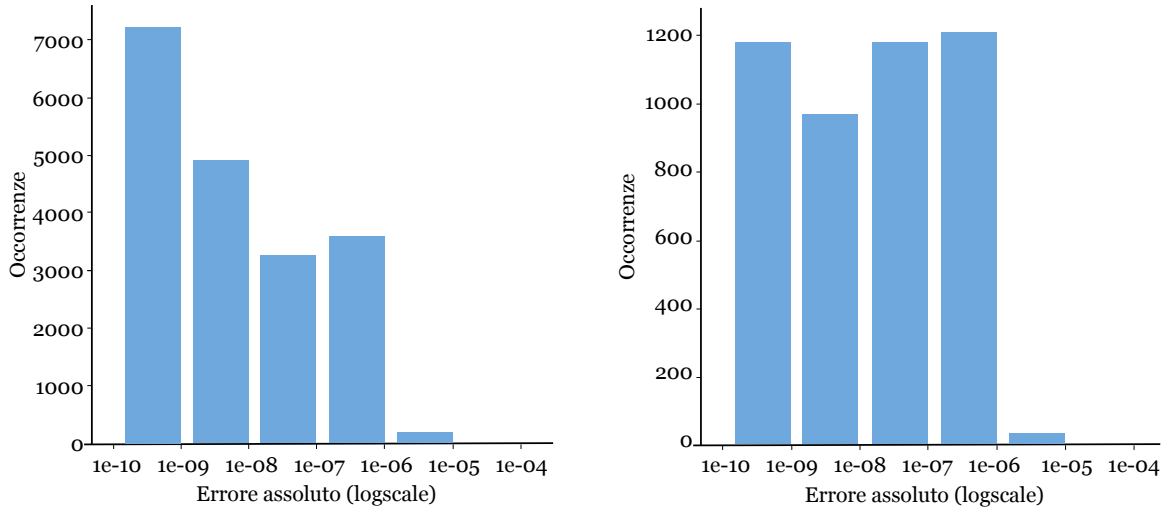
```
#CALCOLO DEI VALORI CON LA RETE FP32
#caricamento della rete neurale ottimizzata FP32
classifier = npClassifier(model_path)
for(root, dirs, files) in sorted(os.walk(dataset_path, topdown=True)):
    print(root)
    for file in files:
        if (file.endswith('.png')):
            if (file.startswith('._')):
                file = file[2:]
            #caricamento immagine da classificare
            img = loadImage(os.path.join(root, file))
            results=classifier.predict_from_img(img) #classificazione
            #aggiunta dei valori calcolati in un array
            FP32_results= [*FP32_results,*results]
            print('--------------------------------')
del classifier

#CALCOLO DEI VALORI CON LA RETE FP16
#caricamento della rete neurale ottimizzata FP16
classifier = npClassifier(model_path,FP_16=True)
for(root, dirs, files) in sorted(os.walk(dataset_path, topdown=True)):
    print(root)
    for file in files:
        if (file.endswith('.png')):
            if (file.startswith('._')):
                file = file[2:]
            #caricamento immagine da classificare
            img = loadImage(os.path.join(root, file))
            results=classifier.predict_from_img(img) #classificazione
            #aggiunta dei valori calcolati in un array
            FP16_results= [*FP16_results,*results]
            print('--------------------------------')
del classifier

np_FP16_results=np.array(FP16_results)
np_FP32_results=np.array(FP32_results)
#calcolo dell'errore assoluto
abs_error=abs(np_FP32_results - np_FP16_results)
# calcolo dell'errore relativo
rel_error=abs_error/np_FP32_results
```

Codice 5.9: benchmark per la misurazione dell'errore dovuto alla semplificazione in Floating point a 16 bit

Rete Neurale	errore assoluto massimo	errore assoluto medio	errore relativo massimo	errore relativo medio
EfficientNet	2.98e-6	2.38e-8	5.29e-5	2.03e-6
ResNet50	**0**	**0**	**0**	**0**
MobileNetV3	2.26e-6	6.70e-9	2.64e-5	2.15e-6

Tabella 5.6: Errore introdotto usando FP16 invece di FP32

Si nota che l'errore introdotto è sufficientemente ridotto da risultare ininfluente ai fini della classificazione. La rete ResNet50 non genera errore con la conversione, un comportamento probabilmente dovuto alla struttura della rete.



EfficientNetB1 MobileNetV3

Figura 5.3: Distribuzione dell'errore introdotto utilizzando FP16 invece di FP32

Si nota inoltre che gli errori sono distribuiti attorno a valori molto piccoli, la distribuzione dell'errore è asimmetrica e ha una coda lunga a destra. Si osserva quindi che l'errore introdotto è ammissibile e che è ragionevole preferire la versione semplificata della rete, considerati i miglioramenti riscontrati nelle tempistiche di inferenza e le prestazioni simili in termini di accuratezza.

Capitolo 6

Il software prodotto

I risultati dei benchmark precedentemente svolti dimostrano che è possibile classificare immagini in tempo reale sulla scheda Jetson Nano. Ai fini del progetto NextPerception, è necessario includere il processo di classificazione in un software che si occupa di catturare i fotogrammi di un flusso video, classificarli e accoppiare i risultati con le informazioni della dinamica veicolare. Si è quindi prodotto un software, schematizzato in figura 6.1 che realizza quanto descritto.

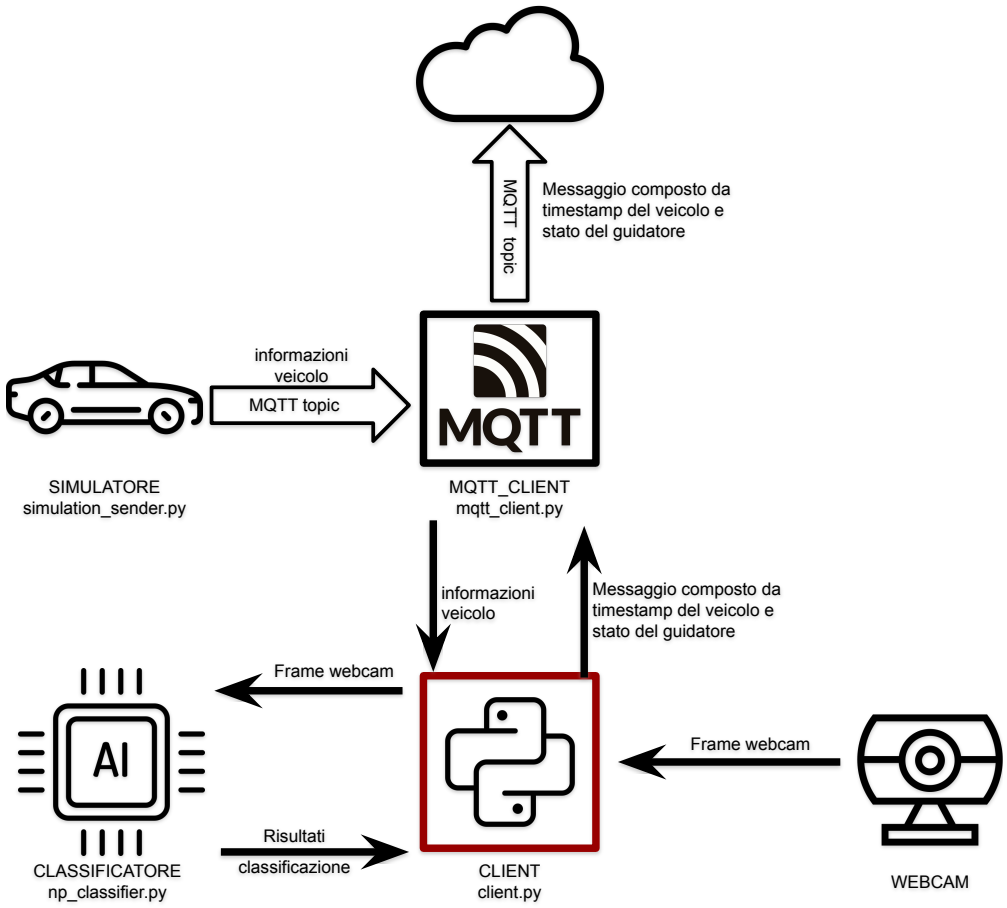


Figura 6.1: Rappresentazione schematica del software realizzato

6.1 La struttura del software

Il software in questione suddivide le funzionalità in più codici python, per garantire la sostenibilità del codice. I codici su cui il software fa affidamento sono i seguenti:

- `np_classifier.py`: la porzione di del software che si occupa della classificazione di immagini; contiene la classe `npClassifier` precedentemente descritta, che permette di sfruttare le potenzialità della libreria `TensorRT` in modo semplice ed intuitivo;

- `simulation_sender.py`: che si occupa di simulare un veicolo e trasmettere le informazioni relative alla dinamica veicolare alle altre componenti software;

- `client.py`: che si occupa di acquisire i fotogrammi dalla webcam, classificarli utilizzando `npClassfier` e accoppiare e trasmettere i dati provenienti dal simulatore con i risultati di classificazione.

- `mqtt_client.py`: che utilizzando il protocollo MQTT mette in comunicazione i componenti sopra descritti.

6.1.1 Il cuore del software

La componente principale del software è all'interno del codice 6.1, contenuto nel file `client.py`. Inizialmente viene istanziato un classificatore, sfruttando la classe `npClassifier` precedentemente discussa. Successivamente viene istanziato un oggetto `MQTTClient`, che si occuperà di popolare una coda con le informazioni provenienti dal simulatore e di trasmettere i risultati di classificazione. Viene poi inizializzato l'oggetto `capture` di classe `cv2.VideoCapture`; questo oggetto permette di operare con il flusso video proveniente dalla webcam e di catturare quindi i fotogrammi da classificare. Terminata questa fase di inizializzazione, ha inizio un ciclo in cui avviene il corpo del codice. Ad ogni iterazione del ciclo, viene catturato un fotogramma con la funzione `capture.read()`, questo fotogramma viene classificato con la funzione `classifier.predict_from_image()`. Dal risultato di classificazione viene estratta la classe che ha confidenza maggiore e le viene associata la relativa etichetta. Inoltre viene valorizzata una variabile booleana `safe_driving`, che assume valore `True` se la classe predetta è la classe 0, `False` altrimenti. Utilizzando la funzione `get_last_element()` viene estratto dalla coda `message_queue` un messaggio contenente la dinamica veicolare. In questa fase di sviluppo l'unica informazione rilevante della dinamica veicolare è il timestamp, utile ad accoppiare i risultati di classificazione con lo stato del veicolo. La funzione `get_last_element()` restituisce l'ultimo messaggio della coda, se la coda ha almeno un messaggio, restituisce None altrimenti; questo comportamento non bloccante permette al software di eseguire tutte le classificazioni possibili, senza attendere i dati dal simulatore. Si è scelto di utilizzare una coda per permettere al software di essere indipendente dal simulatore e di poter continuare a funzionare al massimo delle proprie potenzialità sia nel caso in cui il simulatore invii troppi messaggi, sia nel caso in cui il simulatore non invii messaggi o risulti spento. Viene quindi composto il

messaggio JSON contenente tutte le informazioni calcolate; il messaggio ha la seguente struttura:

- **timestamp**: l'ultimo timestamp ricevuto dal simulatore;
- **dcdc_timestamp**: timestamp della scheda Jetson Nano;
- **safe_driving**: valorizzato secondo l'omonima variabile;
- **classification_class**: il numero relativo alla classe predetta dal classificatore;
- **classification_classname**: l'etichetta di classe predetta dal classificatore;
- **classification_confidence**: la confidenza di classificazione.

Questo messaggio viene quindi pubblicato sul topic MQTT `TOPIC_RESULTS`, utilizzando la funzione `publish()` dell'oggetto `client`.
La condizione di terminazione del ciclo è che l'utente invochi una `KeyboardInterrupt`, ad esempio premendo la combinazione di tasti `CTRL+C`. Dopo la terminazione del ciclo viene stampato il tempo medio per iterazione, per avere un'idea di come il software ha performato. Vengono infine rilasciati il flusso video della webcam e la connessione MQTT

```
def get_last_element(queue):
    element = None
    while True:
        try:
            element = queue.get(False) #get non bloccante
        except Empty as ee:
            break
    return element
MOBILENET = mobilenet_path #percorso per la rete ONNX
TOPIC_SIMULATOR_SENDER = "Simulator_Data" #topic MQTT del simulatore
TOPIC_RESULTS = "NP_UNITO_DCDC" #topic MQTT per pubblicare i risultati
ACTIONS_CM15 = ['Baseline', 'Drink', 'HairAndMakeup', 'LookRight', '
    LookLeft', 'LookBehind', 'LookBelow', 'OperateRadio', '
    TalkCellRight', 'TalkCellLeft', 'TalkPassengerRight', '
    TalkPassengerLeft', 'TextingRight', 'TextingLeft', 'Yawn']

iterations = 0 #variabile che tiene il conteggio delle classificazioni
ttime=0 #variabile per cronometrare il tempo totale di classificazione
message_queue = Queue() #coda da cui prelevare la dinamica veicolare
timestamp='unknown' #ultimo timestamp noto del simulatore
#istanziamento del classificatore
classifier = npClassifier(MOBILENET,FP_16=True)
#instanziamento del client MQTT
client=MQTTClient(message_queue,TOPIC_SIMULATOR_SENDER,TOPIC_RESULTS)
client.connect()
capture = cv2.VideoCapture(0) #flusso video della webcam
#setup della webcam
capture.set(cv2.CAP_PROP_FRAME_WIDTH,320)
capture.set(cv2.CAP_PROP_FRAME_HEIGHT,240)
classifier.warmup()#riscaldamento del classificatore
```

```
28 print("Starting loop, press CTRL+C to stop")
29 try:
30     frame = None
31     while True:
32         exectime=time.perf_counter()
33         iterations=iterations+1
34         #cattura del fotogramma da classificare
35         ret, frame = capture.read(frame)
36         if  not ret:
37             break
38         results = classifier.predict_from_img(frame) #classificazione
39         #estrazione della classe predetta
40         classification_class=np.argmax(results)
41         #associazione della classe alla relativa etichetta
42         classification_classname=ACTIONS_CM15[classification_class]
43         classification_confidence=results[classification_class]
44         safe_driving= classification_class==0 #guida sicura?
45         #estrazione della dinamica veicolare dalla coda
46         simulator_data=get_last_element(message_queue)
47         if(simulator_data is not None):
48              #aggiornamento del timestamp
49             timestamp=json.loads(simulator_data)['time']
50         json_to_send=r""" #composizione del messaggio JSON da inviare
51         {
52         "timestamp": %s,
53         "dcdc_timestamp": %s
54         "safe_driving": %s,
55         "classification_class": %i
56         "classification_classname": %s
57         "classification_confidence": %f
58         }
59         """%(str(timestamp),datetime.utcnow(),safe_driving,
    classification_class,classification_classname,
    classification_confidence)
60         client.publish(json_to_send) #inoltro del messaggio JSON
61         exectime=time.perf_counter()-exectime
62         ttime=ttime+exectime #tempo totale di esecuzione
63 except KeyboardInterrupt: #cattura di CTRL+C per terminare il codice
64     pass
65 print('\nLoop stopped')
66 #stampa del tempo totale e del tempo medio per iterazione
67 print('ttime: ',ttime,'\nsingleactiontime: ',ttime/iterations)
68 capture.release()
69 client.disconnect()
```

Codice 6.1: codice di client.py, che acquisisce e classifica i fotogrammi, e accoppia i risultati con la dinamica veicolare

6.1.2 Il simulatore di veicolo

Per verificare il funzionamento del codice, si è sviluppato un simulatore, non avendo a disposizione un veicolo da cui prelevare le informazioni della dinamica veicolare. Il simulatore, riportato nel codice 6.2, legge uno per volta i file presenti nella cartella `UserProcessed`. Questi file contengono le informazioni relative alla dinamica veicolare in formato CSV. Dal CSV letto vengono rimosse le informazioni non rilevanti, utilizzando la funzione `loc()`. Successivamento ogni riga del file viene convertita in formato JSON e trasmessa al client, sfruttando la classe `SimulationSender` presente in `mqtt_client.py`. La trasmissione avviene ad una frequenza di 20Hz, per rendere la simulazione coerente con le altre componenti del software NextPerception.

```
#topic MQTT su cui trasmettere la dinamica veicolare
TOPIC_SIMULATOR_SENDER = "Simulator_Data"
#istanziazione del client MQTT
simulatorSender = SimulatorSender(TOPIC_SIMULATOR_SENDER)
    simulatorSender.connect()
    #vengono percorsi i file presenti nella directory UserProcessed
    for root, _, files in os.walk('UserProcessed', topdown=False):
        for filename in files: #per ogni file
            print('Sending simulation of '+ filename )
            #lettura file contenente la dinamica veicolare
            data=pd.read_csv(os.path.join(root, filename))
            #vengono eliminate le informazioni non rilevanti
            data=data.loc[:,~data.columns.str.startswith(to_exclude)]
            for i in range(len(data)): #per ogni riga del file
                #conversione in JSON
                json_output = json.dumps(json.loads(data.iloc[i]
                                            .to_json(orient='index')),
                                           indent=4,sort_keys=False)
                #trasmissione via MQTT della dinamica veicolare
                simulatorSender.publish(json_output)
                #ritardo necessario per trasmettere a 20 Hz
                time.sleep(0.05)
    simulatorSender.disconnect()
```

Codice 6.2: simulatore di un veicolo, che trasmette la dinamica veicolare ad una frequenza di 20Hz

6.1.3 La trasmissione di messaggi

La comunicazione delle componenti del software è basata sul protocollo MQTT, che viene implementato grazie alle classi `MQTTClient` e `SimulationSender`, contenute nel file mqtt_client.py. Queste classi sfruttano la libreria `paho.mqtt.client` [9] per l'utilizzo del protocollo di trasmissione MQTT. La struttura dei metodi delle due classi è simile, ed è così rappresentabile:

- **init**: costruttore della classe;
- **connect**: metodo da richiamare per avviare il client MQTT
- **disconnect**: metodo da richiamare per terminare il client MQTT
- **publish**: metodo per pubblicare su un topic MQTT
- **on_message**: metodo che viene richiamato quando arriva un messaggio MQTT.

La classe `MQTTClient` è utilizzata nel file `client.py` e utilizza il metodo `on_message` per ricevere i messaggi dal simulatore e caricarli nella coda condivisa con `client.py`. Questa classe utilizza inoltre il metodo `publish` per pubblicare i risultati di classificazione sul topic MQTT `TOPIC_RESULTS`.
La classe `SimulatorSender` è utilizzata nel file `simulator_sender.py` per pubblicare la dinamica veicolare sul topic MQTT `TOPIC_SIMULATOR_SENDER`

6.2 Valutazione del software

Dopo lo sviluppo, ha avuto luogo una fase di profilazione del software, utile a verificare la bontà di quanto prodotto ed a valutare possibili modifiche che porterebbero miglioramenti.

6.2.1 Tempistiche di l'ottimizzazione

Il processo di ottimizzazione è un'operazione onerosa in termini di tempo. La classe `npClassifier` minimizza le tempistiche di avvio del software eseguendo l'ottimizzazione solo se necessario e salvando su un file l'engine ottimizzato. Il tempo di ottimizzazione è quindi un costo one-shot, necessario per produrre l'engine di classificazione. Si è misurato il tempo necessario per l'ottimizzazione delle diverse reti utilizzate, i risultati sono riportati nella tabella 6.1.

Precisione	Rete neurale		
	EfficientNetB1	ResNet50	MobileNetV3
FP_16	331.69s	52.60s	206.79s
FP_32	200.38s	37.62s	110.97s

Tabella 6.1: Tempistiche di ottimizzazione per reti neurali utilizzate, al variare della precisione di calcolo.

6.2.2 Profilazione dei tempi

Si è svolta una misurazione dei tempi di avvio del software, per ricercare le componenti che impattano maggiormente; i risultati riportati in tabella 6.2, riguardano i tempi di avvio utilizzando la rete MobileNetV3 e sono stati catturati con la funzione python `time.perf_counter()`. I risultati contengono esclusivamente le operazioni rilevanti al tempo di avvio.

Operazione	Tempo necessario (s)
Importazione librerie python	1.257
Inizializzazione classificatore	3.58507
Inizializzazione stream video	0.35411
Setup stream video	0.45757
Riscaldamento classificatore	1.9092
Tempo di avvio totale	7.56295

Tabella 6.2: Tempi necessari per l'avvio del software con la rete MobileNetV3

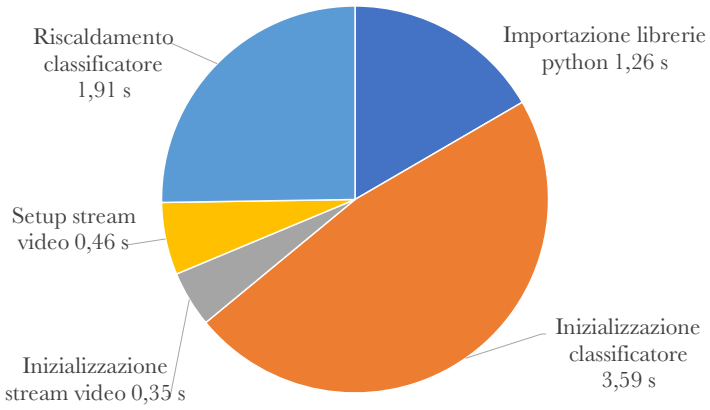


Figura 6.2: Grafico rappresentante i tempi di avvio delle componenti del software.

Si può notare che l'operazione più onerosa in termini di tempo è l'inizializzazione del classificatore, che utilizza circa il 50% del tempo totale di avvio. Anche l'operazione di riscaldamento del classificatore impatta particolarmente, utilizzando il 25% del tempo totale; questa operazione non è essenziale per il funzionamento del software, ma ne garantisce la stabilità temporale. Il tempo totale di avvio ammonta a circa 7,5 secondi se si utilizza la rete MobileNetV3. È stato misurato il tempo di avvio per le altre architetture di rete testate, i risultati risultano molto simili; il software impiega approssimativamente 7,7s per l'avvio con la rete ResNet50 e circa 8,1s per l'avvio con la rete EfficientNetB1.

6.2.3 Profilazione della memoria

Per via del quantitativo molto limitato di memoria RAM presente sulla scheda NVIDIA Jetson Nano è essenziale verificare che il software utilizza solo la memoria necessaria. Si è quindi svolta una profilazione della memoria allocata durante l'esecuzione del software, per valutare il quantitativo di memoria ancora utilizzabile e le possibili migliorie applicabili. La profilazione è stata effettuata utilizzando la libreria python `memory_profiler`; i risultati riportati in tabella 6.3, sono relativi al software descritto, che utilizza la rete MobileNetV3 e includono solo le operazioni più onerose in termini di memoria.

Nome simbolico dell'operazione	Utilizzo memoria (MB)
Importazione librerie python	142.559
Inizializzazione classificatore	871,707
Inizializzazione dello stream video	5.891
Classificazione	341.637
Totale	1361,41

Tabella 6.3: Memoria RAM utilizzata durante l'esecuzione del software

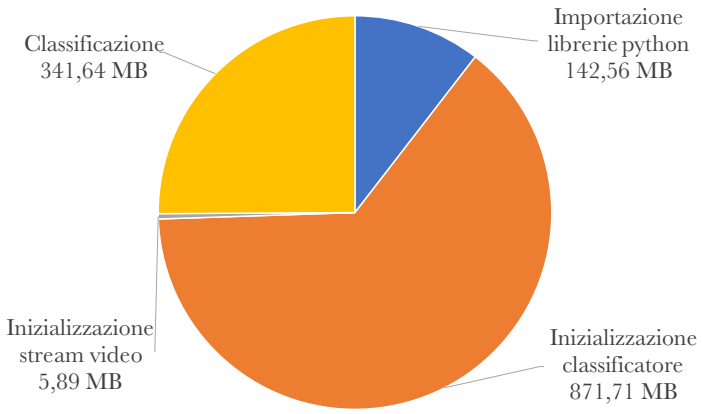


Figura 6.3: Utilizzo di memoria RAM durante l'esecuzione del software

Dai risultati si nota che le operazioni che richiedono il maggior quantitativo di memoria sono quelle relative al classificatore. Il quantitativo di memoria utilizzata in totale ammonta a circa 1300 MB, ben inferiore alla quantità di memoria disponibile sulla scheda Jetson Nano: 4 GB. I risultati non evidenziano margini di miglioramento evidenti, in quanto la maggior parte della memoria è utilizzata dal classificatore e non risulta possibile ridurre questo utilizzo senza modificare il codice sorgente della liberia TensorRT. I test relativi alle altre reti neurali trattate dimostrano un leggero discostamento in termini di memoria utilizzata, nello specifico, se si utilizza la rete EfficientNetB1 il software richiede circa 984MB, se si utilizza la rete ResNet50 il software richiede circa 1380MB.

6.3 Conclusioni e sviluppi futuri

Si è dimostrato sperimentalmente che la scheda Jetson Nano risulta sufficientemente prestante da permettere la classificazione di immagini in tempo reale rispettando i vincoli progettuali. Il processo di ottimizzazione utilizzato ha portato un incremento prestazionale medio del 400%, senza degradare l'accuratezza di classificazione. Dai test risulta che durante l'esecuzione del software sviluppato, con un flusso video a 30 frames per secondo, l'utilizzo medio della CPU ammonta al 45% e l'utilizzo medio della GPU ammonta al 25%. Risulta quindi che la scheda utilizzata dispone di ulteriore potenza di calcolo, utilizzabile in sviluppi futuri.
Una possibile iterazione futura del software potrebbe riguardare il riconoscimento della distrazione cognitiva dell'automobilista; ciò potrebbe essere realizzato implementando un'ulteriore rete neurale che elabora i dati della dinamica veicolare e lo stato di attenzione del guidatore, determinando se è necessario inviare un segnale di allerta.

Bibliografia

[1] NVIDIA Hello AI World. https://github.com/dusty-nv/jetson-inference. Accessed: 2022-06-26.

[2] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt. Accessed: 2022-06-26.

[3] ONNX optimizer. https://github.com/onnx/optimizer. Accessed: 2022-06-26.

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[5] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

[6] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[7] Francois Chollet et al. Keras, 2015.

[8] ONNX Runtime developers. Onnx runtime. https://onnxruntime.ai/, 2021. Version: x.y.z.

[9] Eclipse. Eclipse Paho™ MQTT Python Client. https://github.com/eclipse/paho.mqtt.python.

[10] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[12] NVIDIA. Copies memory for 2d arrays. `https://nvidia.github.io/cuda-python/module/cuda.html`.

[13] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. CUDA, release: 10.2.89, 2020.

[14] ONNX. tensorflow-onnx: Convert tensorflow, keras, tensorflow.js and tflite models to onnx. `https://github.com/onnx/tensorflow-onnx`, 2022. Accessed: 2022-06-26.

[15] Siying Qian, Chenran Ning, and Yuepeng Hu. Mobilenetv3 for image classification. In *2021 IEEE 2nd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*, pages 490–497, 2021.

[16] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark, 2019.

[17] Mingxing Tan and Quoc V. Le. EfficientNetV2: Smaller Models and Faster Training. 2021.

[18] Zetane. ML models and internal tensors 3D visualizer. `https://github.com/zetane/viewer`. Accessed: 2022-06-26.