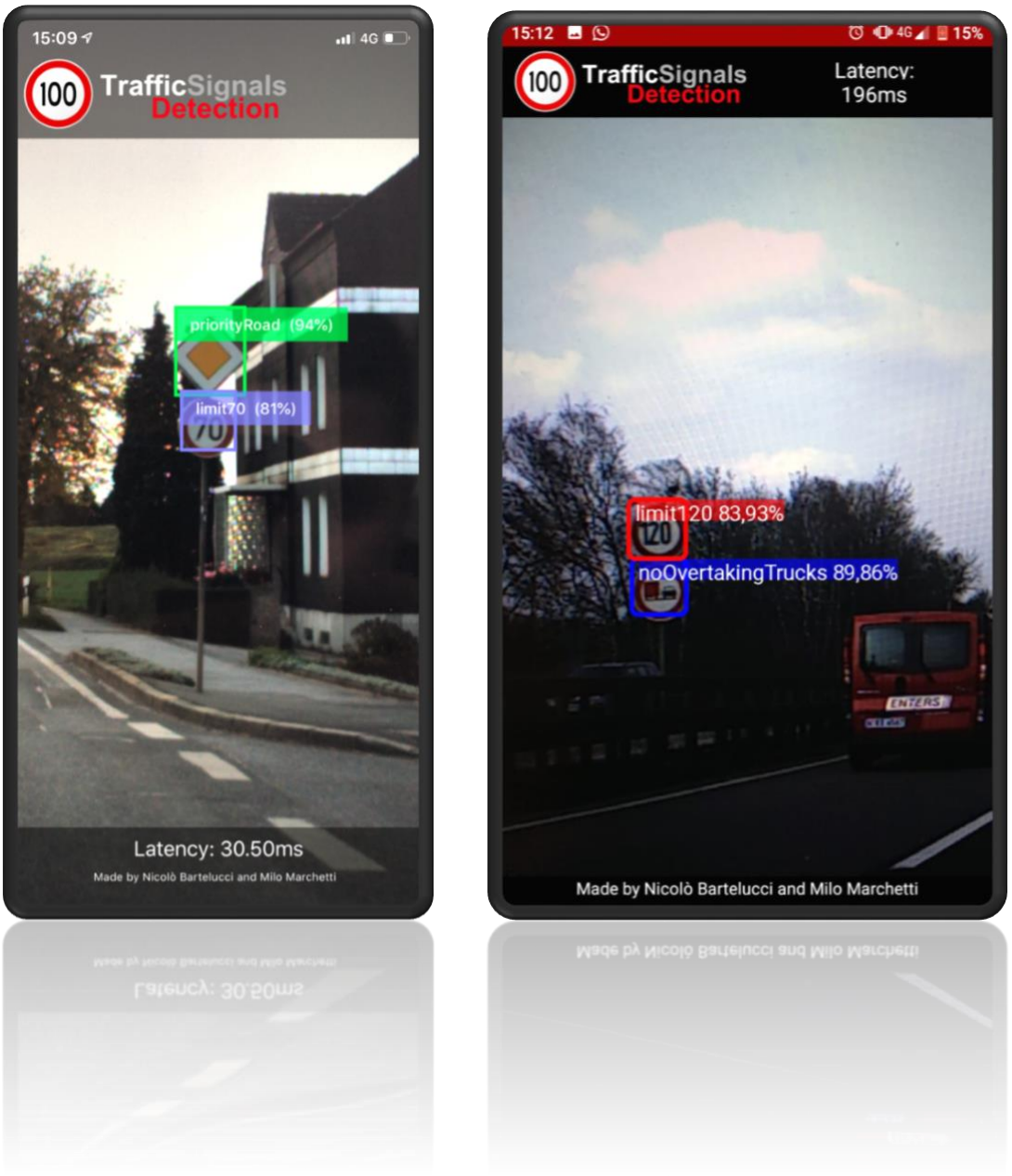
Traffic Signal Detection on Mobile Devices

Progetto di **Sistemi Digitali M**

Nicolò Bartelucci nicolo.bartelucci@studio.unibo.it
Milo Marchetti milo.marchetti@studio.unibo.it



Indice

Introduzione

I contesti applicativi che presentano un connubio tra reti neurali di convoluzione e dispositivi mobili sono molteplici. Uno di quelli che si è più distinto negli ultimi anni è sicuramente quello nel settore stradale.

In questo documento verrà illustrato il percorso effettuato per realizzare una rete neurale capace di riconosce segnali stradali in tempo reale. A tal fine ci approcceremo a tecnologie riguardanti l'ambito della Computer Vision e del Machine Learning, in particolare approfondiremo l'utilizzo di TensorFlow in Python e TensorFlow Lite in Android Studio.

Il processo di sviluppo della rete può essere riassunto in tre fasi. La prima consiste nella ricerca degli strumenti necessari allo sviluppo e nel trovare un dataset adeguato al nostro caso d'uso. La seconda fase tratterà il training della rete neurale, andando a svolgere uno studio sui modelli più utilizzati nell'ambito di Object Detection in portabilità. Nella terza e ultima fase approfondiremo alcune tecniche di ottimizzazione delle reti al fine di migliorare le performance nei dispositivi dall'hardware più datato.

Capitolo 1: Background

Il riconoscimento di oggetti in tempo reale è una delle maggiori sfide della Computer Vision in ambito industriale. Molte sono le soluzioni e gli approcci proposti ma non c'è ancora una chiara definizione di cosa sia lo stato dell'arte in questo ambito.

1.1: Computer Vision e Reti Neurali

La Computer Vision è una disciplina inclusa nel campo dell'Artificial Intelligence e studia come dei dispositivi possano dedurre informazioni di alto livello da immagini digitali. Da un lato ingegneristico mira alla replicazione di ciò che la vista dell'uomo è in grado di fare.

Le CNN (Convolutional Neural Network, figura 1), molto utilizzate nell'ambito Computer Vision e Deep Learning, possono essere viste come un insieme di pesi che, collegati tra loro a formare una particolare architettura, trasformano i dati di input nei desiderati dati di output. Ogni peso è solitamente rappresentato da un valore numerico in float32, che può essere ridotto anche a float16 o int8 al fine di velocizzare la rete. Questi pesi sono di solito randomici all'inizio e durante il training vengono modificati al fine di ottenere risultati migliori e più precisi.

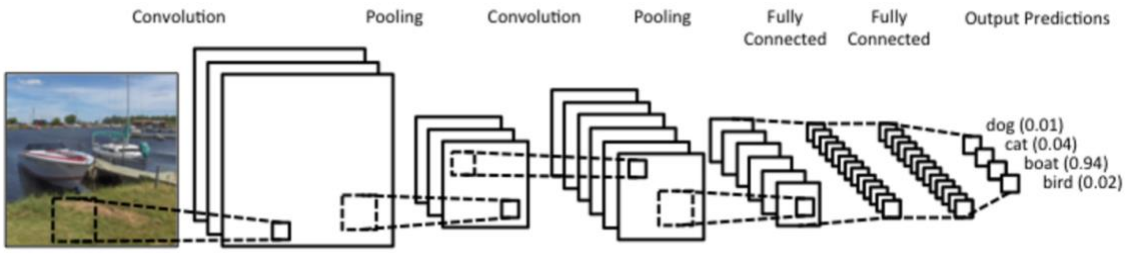


Figura 1: Esempio di Convolutional Neural Network

1.2: Image Classification vs Object Detection

Nell'ambito dell'intelligenza artificiale combinata alla Computer Vision esistono differenti modi di approcciarsi all'elaborazione delle immagini e due di questi sono Image Classification e Object Detection (figura 2). Nonostante vengano spesso confusi sono due cose molto diverse fra loro.

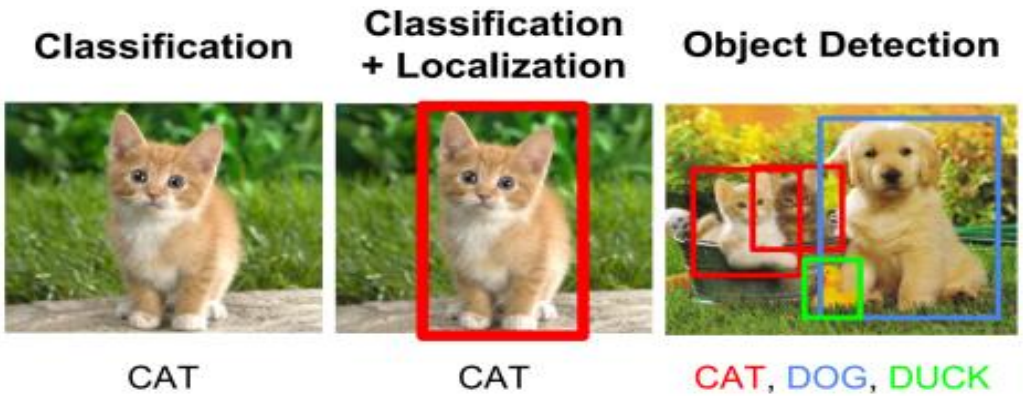


Figura 2: Image Classification, Localization e Object Detection

Per Image Classification si intende il processo che porta un modello, correttamente allenato, a riconosce varie classi di oggetti e ad analizzare un'immagine ed elaborare come risultato la classe dell'oggetto che ritiene essere più probabile all'interno dell'immagine stessa.

Questo processo può essere esteso aggiungendo alla Classification anche la Localization, cioè facendo sì che oltre alla classe il risultato presenti anche le bounding box (cioè l'area dell'immagine che contiene l'oggetto riconosciuto). Questo processo prende il nome di Image Localization e vale sempre per un solo e unico soggetto.

Per Object Detection invece si intende il processo di Image Localization applicato a tutti gli oggetti che sono presenti sull'immagine. Infatti, il risultato della Object Detection è dato da classe e bounding box per ogni oggetto dell'immagine.

Nell'ambito stradale la Object Detection risulta essere la scelta migliore in quanto si ha la necessità di valutare contemporaneamente più oggetti a schermo, come persone, cartelli stradali, automobili, etc…

Capitolo 2: Risorse e Strumenti Utilizzati

Per realizzare una rete con lo scopo di effettuare un riconoscimento di oggetti in un'immagine si sono rivelati essenziali strumenti come Anaconda, per il corretto setup dell'environment Python, e TensorFlow, per il training delle reti sul German Traffic Sign Dataset.

2.1: TensorFlow e TensorFlow-GPU

TensorFlow è una libreria software di stampo matematico open-source sviluppata dal team Google Brain con l'obiettivo di utilizzare il Machine Learning sfruttando le grandi moli di dati a cui la compagnia ha accesso. Rilasciata al pubblico per la prima volta nel 2015 e nella sua prima versione stabile nel 2017 al giorno d'oggi è una delle librerie più utilizzate per lo sviluppo di reti neurali.

TensorFlow è compatibile con i principali sistemi operativi a 64 bit (Windows, Linux e macOS) e Android. Nonostante la documentazione ufficiale, all'inizio, parlasse di una limitata compatibilità hardware, la libreria può funzionare su numerosi tipi di CPU e anche su GPU, grazie al supporto di linguaggi come CUDA o OpenCL.

Data la complessità delle operazioni svolte da TensorFlow il supporto a CUDA e a cuDNN risulta molto importante. Il primo è un'architettura hardware che consente di l'elaborazione parallela sulle GPU delle schede video NVIDIA, mentre il secondo è una libreria per l'accelerazione delle operazioni base per le reti neurali.

In generale TensorFlow consente agli sviluppatori di svolgere le seguenti operazioni:

- Pre-processamento dei dati
- Realizzazione di un modello
- Addestramento e valutazione del modello designato

2.2: TensorFlow Lite

TensorFlow Lite è un insieme di strumenti per consentire agli sviluppatori di eseguire modelli TensorFlow su dispositivi mobile, embedded e IoT.
TensorFlow Lite è caratterizzato da due componenti principali:

- L'interprete TensorFlow Lite, che consente di eseguire modelli appositamente ottimizzati su diversi tipi di hardware, inclusi telefoni cellulari, sistemi embedded Linux e microcontrollori
- Il convertitore TensorFlow Lite, che permette di convertire modelli TensorFlow in un modulo efficiente per l'uso da parte dell'interprete, e può introdurre ottimizzazioni per migliorare le prestazioni.

2.3: Anaconda

Anaconda è una distribuzione open-source dei linguaggi Pyhton e R mirata per semplificare lo sviluppo e la gestione dei pacchetti per applicazioni in ambito Data Science. La grande popolarità di Anaconda è data dal fatto che riesce pienamente nell'intento di semplificare il processo di setup di un ambiente di sviluppo con Python, racchiudendo assieme nella stessa distribuzione tutto ciò di cui si ha bisogno per iniziare a programmare: l'installer di Python, un package e environment manager dedicato di nome Conda, e circa 300 pacchetti installati come Pandas, NLTK, Numpy, Matplotlib, Jupiter, Requests, SQLAlchemy. Anaconda in combinazione con TensorFlow e vari altri pacchetti risulta essere quindi una piattaforma adatta in ambito Machine Learning e sviluppo di reti neurali.

2.4: The German Traffic Sign Detection Dataset

The German Traffic Sign Detection Dataset (figura 3) è una raccolta di immagini introdotta alla IEEE International Joint Conference on Neural Networks 2013 e destinata ai campi di ricerca della Computer Vision al fine di valutare il rilevamento di segnali stradali in singole immagini.

Figura 3: The German Traffic Sign Detection Dataset

Il Dataset è composto da 900 immagini di 43 classi di segnali diversi, divise in 600 per il training e 300 per l'evaluation. Ogni immagine contiene da 0 a 6 segnali, è disponibile in formato ".ppm" (Portable PixMap) ed è già pre-annotata di bounding-box e classe di interesse. La dimensione dei cartelli varia da una 16x16 a una 128x128 pixels.

Capitolo 3: Training delle Reti

Il training della rete consiste nell'effettuare un ritocco basato su nuovi tipi di classi ai pesi dei layer superiori di modelli precedentemente addestrati rilasciati da TensorFlow (https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md).

La nostra scelta è ricaduta sui modelli che hanno utilizzato il dataset Common Object in Contex (COCO), basato su circa 300.000 immagini raffiguranti 80 categorie di oggetti pre-annotate di bounding-box e labels, pronto per l'utilizzo in ambito Object Detection. Questi modelli vengono differenziati da TensorFlow principalmente in base ai valori di velocità (ms) e precisione di riconoscimento (mAP un'unità di misura per stimare la precisione di una rete). In particolare, sono stati presi in analisi tre modelli: SSD_Inception_v2, SSD_Mobilenet_v2 e SSD_Mobilenet_v3_Small. Sono state effettuate delle prove anche sui modelli Faster_RCNN_Inception_v2 e SSD_Mobilenet_v3_Large che non verranno riportate in quanto computazionalmente troppo dispendiose per dispositivi più datati.

3.1: SSD Inception v2

Modello basato sull'unione di SSD e Inception_v2 che è caratterizzato da una velocità di 42 ms e una precisione di 24 mAP. Le nostre prove ci hanno portato a suddividere il training in due parti utilizzando parametri diversi. Per i primi 50.000 steps si è deciso di impostare un learning rate di 0.001 con batch size 4 mentre per gli steps dal 50.000 al 75.000 abbiamo ridotto il learning rate a 0.0006.
Per comprendere meglio i risultati del training del modello SSD_Inception_v2 riportiamo i grafici relativi alla totalLoss (figura 4), classificationLoss, localizationLoss (figura 5) e i risultati del test post-training (figura 6).

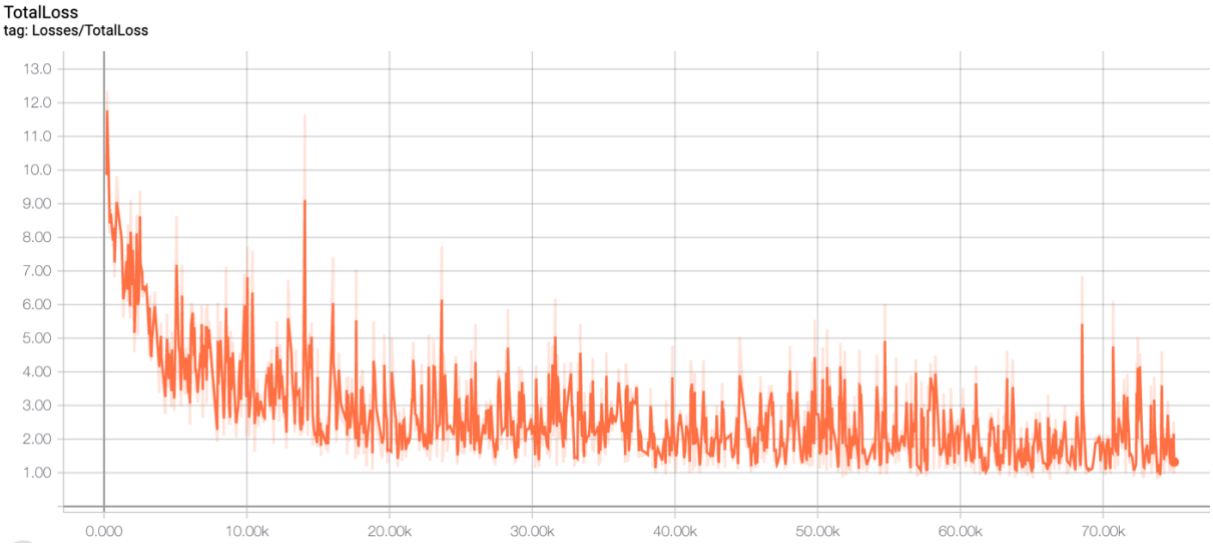


Figura 4: TotalLoss caso SSD_Inception_v2 75.000 steps

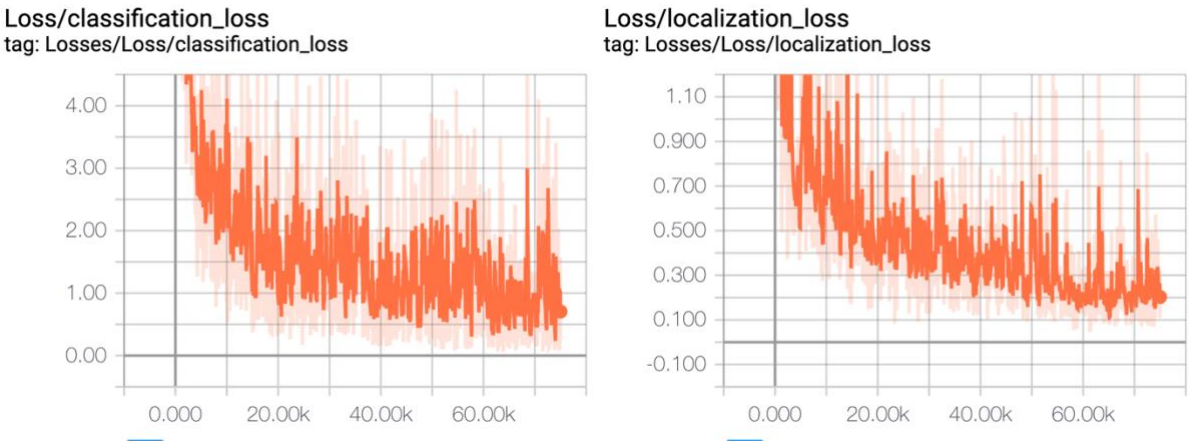


Figura 5: classificationLoss e localizationLoss caso SSD_Inception_v2 75.000 steps

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.595
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.956
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.680
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.411
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.652
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.835
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.545
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.632
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.632
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.451
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.687
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.840
```

Figura 6: evaluation SSD_Inception_v2 75.000 steps

3.2: SSD Mobilenet v2

Modello basato sull'unione di SSD e Mobilenet_v2 che è caratterizzato da una velocità di 31 ms e una precisione di 22 mAP. Come per il precedente modello, abbiamo deciso di suddividere il training in due parti utilizzando parametri diversi. Per i primi 50.000 steps si è deciso di impostare un learning rate di 0.001 con batch size 4 mentre per gli steps dal 50.000 al 75.000 abbiamo ridotto il learning rate a 0.0006.

Per comprendere meglio i risultati del training del modello SSD_Mobilenet_v2 riportiamo i grafici relativi alla totalLoss (figura 7), classificationLoss, localizationLoss (figura 8) e i risultati del test post-training (figura 8).

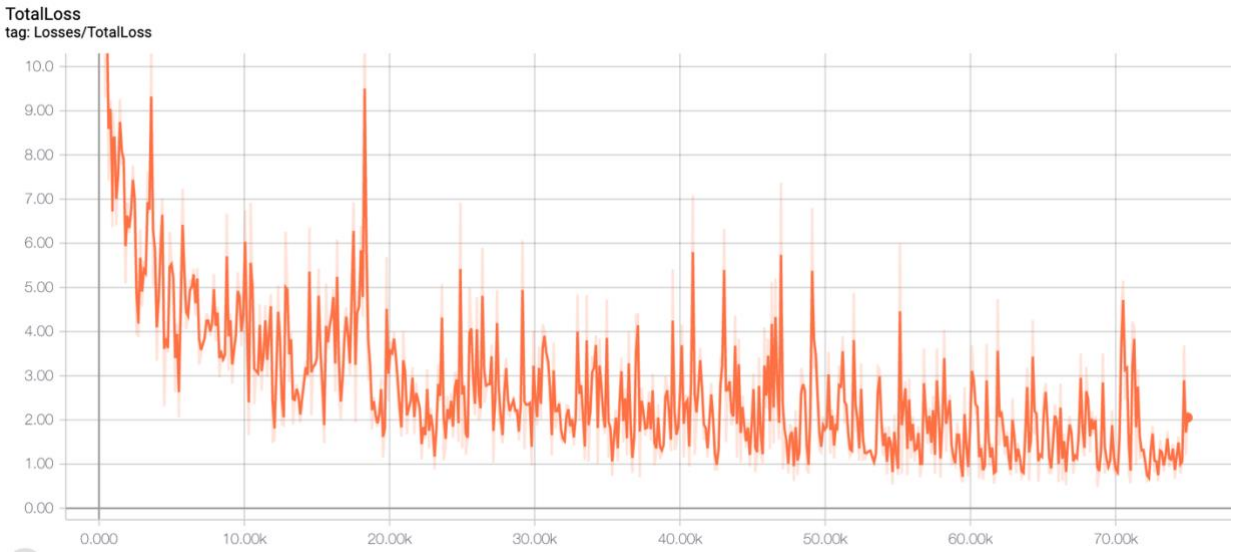


Figura 7: TotalLoss caso SSD_Mobilenet_v2 75.000 steps

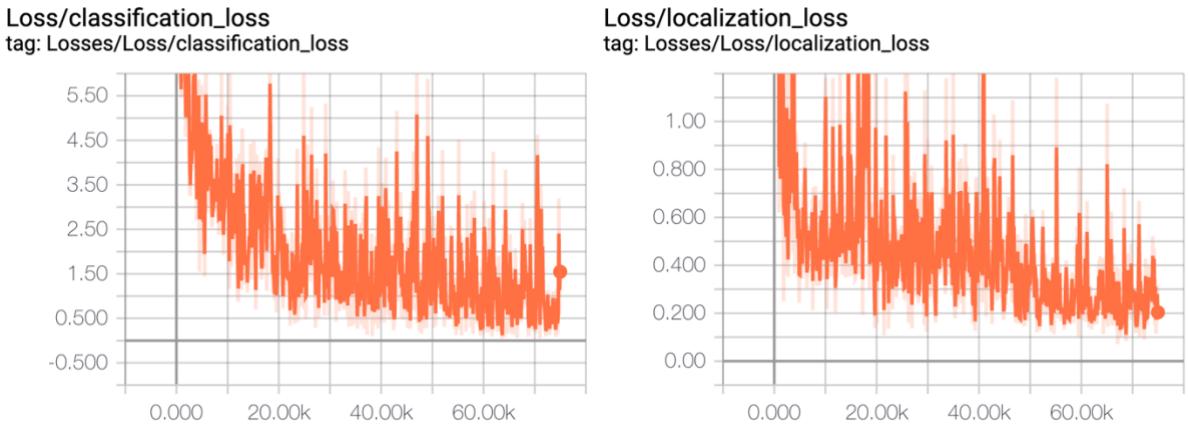


Figura 8: classificationLoss e localizationLoss caso SSD_Mobilenet_v2 75.000 steps

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.512
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.914
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.522
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.320
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.601
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.802
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.485
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.565
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.565
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.365
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.638
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.808
```

Figura 9: evaluation SSD_Mobilenet_v2 75.000 steps

3.3: SSD Mobilenet v3 Small

Modello basato sull'unione di SSD e Mobilenet_v3 che è caratterizzato da una velocità di 43 ms (su un dispositivo Pixel 1) e una precisione di 15.6 mAP. Il training di questo modello è stato eseguito impostando un learning rate base e un warmup learning rate di 0.001 con batch size 128 per un totale di 25.000 steps. Il numero di step differisce dai modelli precedenti a causa della batch size, ma corrisponde all'incirca al tempo di training impiegato per i modelli precedenti.

Per comprendere meglio i risultati del training del modello SSD_Mobilenet_v3_Small riportiamo i grafici relativi alla totalLoss (figura 10), classificationLoss/localizationLoss (figura 11) e i risultati del test post-training (figura 12).

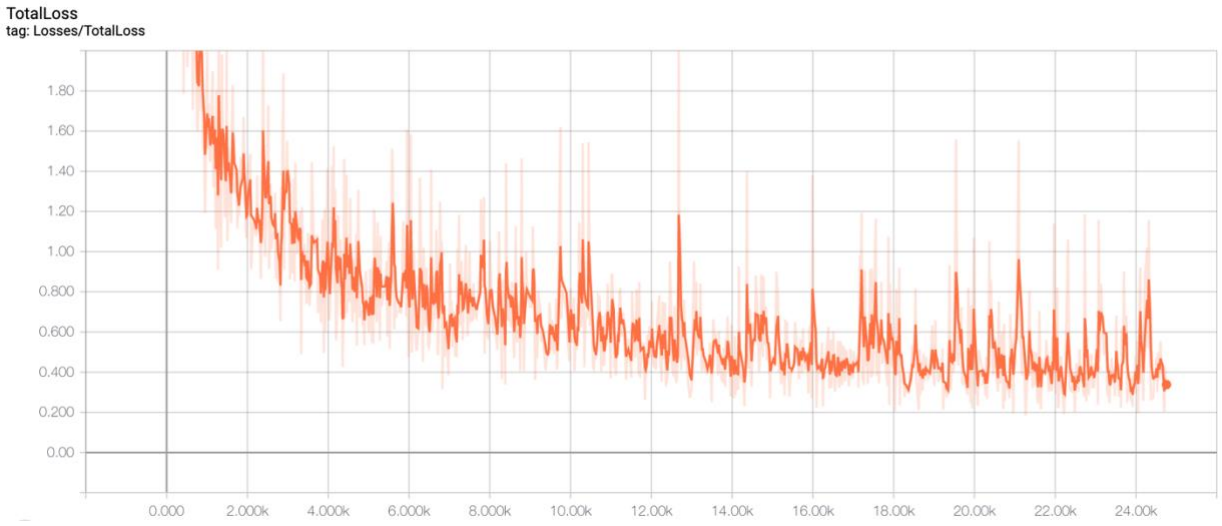


Figura 10: TotalLoss caso SSD_Mobilenet_v3_Small 25.000 steps

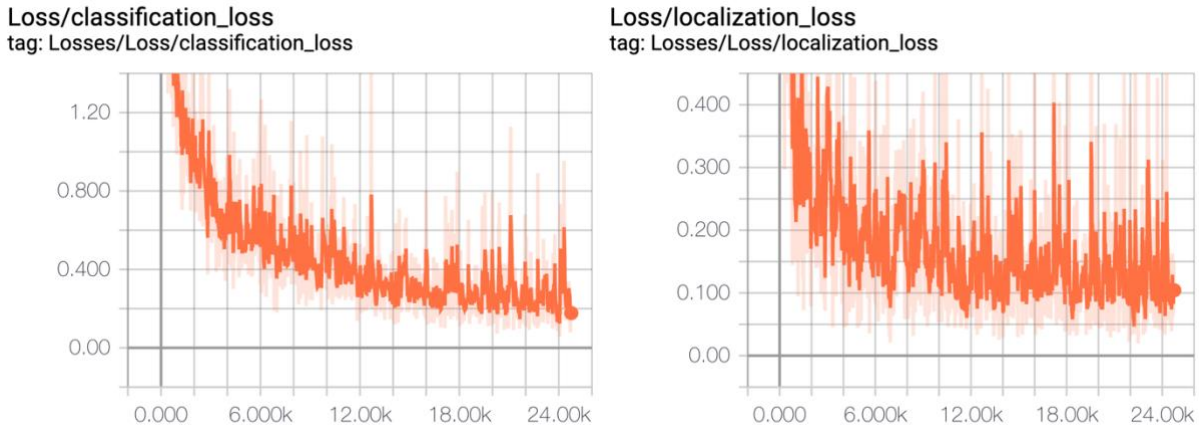


Figura 11: classificationLoss e localizationLoss caso SSD_Mobilenet_v3_Small 25.000 steps

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.421
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.842
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.377
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.250
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.512
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.889
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.427
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.500
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.500
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.337
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.568
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.892
```

Figura 12: evaluation SSD_Mobilenet_v3_Small 25.000 steps

Capitolo 4: Ottimizzazione delle Reti Post Training

Come già accennato nei capitoli precedenti, esistono varie tecniche di ottimizzazione delle reti neurali che permettono di migliorare i tempi di esecuzione dei modelli andandone a ridurre la dimensione in byte. Questo al fine di migliorare le performance anche in dispositivi dall'hardware più datato o a basso consumo. Due delle tecniche di maggior rilievo per ottimizzare le reti dopo averle addestrate sono il Pruning e la Quantization.

4.1: Pruning della Rete

Il Pruning di una rete neurale è una tecnica che permette di eliminare delle operazioni di scarso valore da una rete (figura 13) al fine di ridurre il numero dei layers cercando di avere un impatto minimo sull'accuracy.

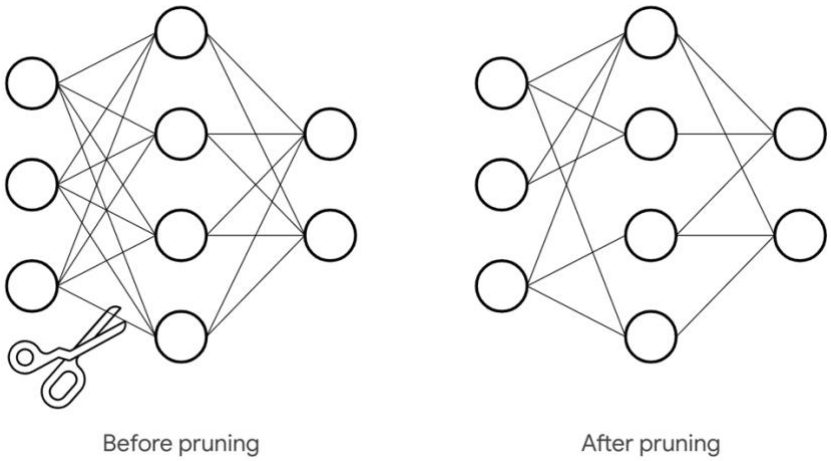


Figura 13: Pruning di una rete neurale

Per effettuare ciò viene calcolata la media di tutti i pesi dei singoli tensori e li si ordinano in modo crescente. Quindi si eliminano i layer "più in alto", cioè quelli con una media di pesi più bassa, che saranno i layer di convoluzione che avranno l'impatto più basso nella rete rispetto a tutti gli altri. Eliminando la parte meno influente della rete si ottiene un modello meno pesante del precedente e di conseguenza tempi di esecuzione più bassi.

Le API per effettuare il Pruning dei pesi di una rete sono basate su Keras, quindi effettuare il prune di una rete Keras addestrata è molto facile per gli sviluppatori.

4.2: Quantization della Rete

La quantizzazione di una rete neurale post-training è una tecnica molto efficace per ridurre il peso complessivo (in byte) del modello ".tflite". Questa tecnica si basa sulla trasformazione del formato dei pesi dei tensori, dei dati in input e di quelli in output dal

valore nativo float32 a un altro formato di rappresentazione che, nel caso di TensorFlow Lite, sono uint8, int8, float16.

Pesi, input e output possono essere quantizzati contemporaneamente o singolarmente in base alle proprie esigenze. Esperimenti riportano che con il passare da un tipo float32 a un uint8 si è in grado di ridurre la dimensione del modello esportato di circa 4 volte, senza andare ad intaccare eccessivamente il corretto funzionamento della rete.

La quantizzazione post-training è stata la tecnica scelta per l'ottimizzazione della nostra rete in quanto adatta ad essere applicata ai nosti modelli. Un problema della quantizzazione di modelli tflite è il mancato supporto alle custom operations: delle operazioni non supportate ufficialmente da TensorFlow Lite. Nel grafo del modello TensorFlow esportato della nostra rete è presente la custom operation "TFLite_PostProcessing" collocata esattamente come ultima operazione. Tale operazione è strettamente necessaria per ottenere i dati riguardanti le classi e le bounding-box a partire dalle anchor-box.

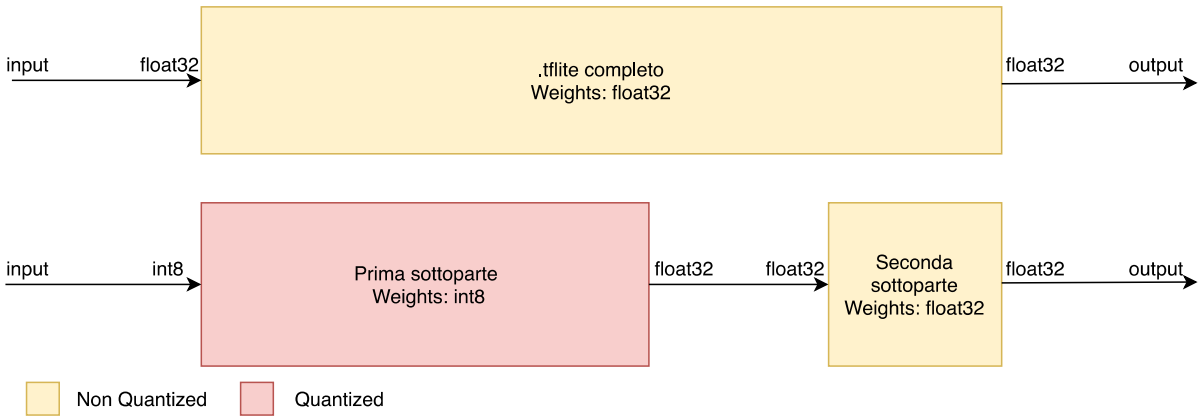


Figura 14: Split modello per Quantizzazione

Per sopperire questa mancanza di supporto si è deciso di dividere (figura 14), fin quando mancherà il supporto alle custom operation, il grafo completo in due sottografi, al fine di quantizzare la maggior parte della rete. La prima sottoparte conterrà tutta la rete meno la "TFLite_PostProcessing" e verrà quantizzata negli ingressi e nei pesi a int8 lasciando gli output a float32. La seconda sottoparte includerà solo la custom operation, non quantizzandola poiché non supportata, con ingressi, pesi e uscite in float32.

Capitolo 5: Analisi dei Risultati

A fronte dei dati ottenuti dai training delle reti e dalla loro quantizzazione post training sono stati effettuati per questo progetto dei test su una serie di dispositivi (Android e iOS). I test sono stati effettuati sfruttando come base un'applicazione di sample fornita da TensorFlow che abbiamo modificato per poter applicare la quantizzazione (https://github.com/tensorflow/examples/tree/master/lite/examples/object_detection).

L'app Android è stata sviluppata in Android Studio in linguaggio Java mentre per l'app iOS è stato utilizzato xCode per macOS in linguaggio Swift. Il principio di funzionamento è molto semplice: sono effettuate continue chiamate alle api TensorFlow Lite fornendo in input un'immagine acquisita, al momento della chiamata, dalla fotocamera del dispositivo in esecuzione. Dopo che gli output sono stati elaborati, questi vengono mostrati a display come bounding-box, classe di previsione, precisione percentuale e tempo di latenza. In questo modo l'effetto risultante è un effetto real-time (o quasi), che migliora al migliorare delle prestazioni del modello o dispositivo utilizzato. L'applicazione è stata eseguita nei seguenti dispositivi: iPhone X per iOS e Xiaomi Mi A1, Huawei P9 Lite per la controparte Android.

Per quanto riguarda la riguarda la capacità dei modelli nel riconoscere le varie classi di segnali stradali, il modello SSD_Inception_v2 si è rivelato essere il più preciso con il minor numero di falsi positivi e negativi. A seguire, il modello SSD_Mobilenet_v2 risulta essere anch'esso valido nonostante presenti alcuni falsi positivi nel caso dei segnali di pericolo. Infine, SSD_Mobilenet_v3_Small è il modello che presenta la peggiore qualità di riconoscimento.

Per quanto riguarda le prestazioni, in termini di latenza e frame per secondo (fps), i dati risultano essere diametralmente opposti come possiamo vedere nella tabella 1.

	Non-Quantized Model		
	SSD Inception v2	**SSD Mobilenet v2**	**SSD Mobilenet v3**
iPhone X	135 ~ 138 ms ~ 7.5 fps	~ 27 ms ~ 37 fps	~ 17 ms ~ 58 fps
Xiaomi Mi A1	650 ~ 720 ms ~ 1.4 fps	160 ~ 210 ms ~ 5 fps	80 ~ 110 ms ~ 10 fps
Huawei P9 Lite	730 ~ 790 ms ~ 1.3 fps	260 ~ 290 ms ~ 3.5 fps	100 ~ 120 ms ~ 9 fps

Tabella 1: Risultati Modelli senza con Quantizzazione Post-Training

Le latenze più basse sono state ottenute dal modello SSD_Mobilenet_v3_Small, sono poi raddoppiate con SSD_Mobilenet_v2 e ulteriormente aumentate con SSD_Inception_v2. I dati sono coerenti con la dimensione dei singoli modelli e infatti, come affermato in

precedenza, maggiore è la dimensione del modello peggiori sono i tempi di latenza. Al fine di cercare di ridurre le dimensioni del modello, e quindi di ridurre i tempi di latenza, abbiamo utilizzato la tecnica di quantizzazione descritta nel capitolo precedente. Il dispositivo iPhone X è stato in questo caso omesso dai test in quanto le reti risultavano avere comportamenti real-time senza quantizzazione.

Come possiamo vedere nella tabella 2, l'approccio proposto di quantizzazione mirato a sorvolare il problema delle custom operation non sempre ha portato a una riduzione dei tempi di latenza.

	Quantized Model		
	SSD Inception v2	**SSD Mobilenet v2**	**SSD Mobilenet v3**
Xiaomi Mi A1	620 ~ 680 ms ~ 1.5 fps	200 ~ 220 ms ~ 4.5 fps	100 ~ 140 ms ~ 8 fps
Huawei P9 Lite	640 ~ 690 ms ~ 1.5 fps	190 ~ 220 ms ~ 5 fps	140 ~ 150 ms ~ 7 fps

Tabella 2: Risultati Modelli con Quantizzazione Post-Training

In dispostivi più datati la riduzione della dimensione del modello (di circa quattro volte rispetto a quello iniziale) ha portato a un miglioramento delle performance nonostante la presenza dell'overhead dovuto alla seconda chiamata alla classe Interpreter, la classe responsabile all'esecuzione del modello TensorFlow Lite. In dispositivi più performanti invece, il guadagno dovuto alla quantizzazione non è stato sufficiente a superare la perdita dovuta all'overhead.

Sviluppi Futuri e Conclusioni

Dopo aver analizzato i risultati ottenuti possiamo concludere che sulla maggior parte dei dispositivi più recenti, di fascia medio-alta, la rete ottenuta con il modello SSD_Mobilenet_v2 è quella che ha portato i migliori compromessi di velocità di esecuzione e precisione media.

Il progetto è aperto a sviluppi futuri come ad esempio la ricerca di un dataset più ricco ed esteso che potrebbe portare un sensibile miglioramento all'accuracy ed anche di aumentare il numero di classi di segnali stradali riconosciuti. Inoltre, in attesa di un supporto ufficiale da parte di TensorFlow Lite alle custom operations, in particolare alla "TFLite_PostProcessing" operation, la soluzione proposta di split dei modelli si è rivelata utile a migliorare i tempi di esecuzione in diversi dispositivi (meno recenti e di fascia medio-bassa).

Quando arriverà il supporto ufficiale sarà possibile applicare la quantizzazione post training in modo molto più efficiente andando ad evitare l'overhead dovuto all'esecuzione di due modelli ".tflite" in cascata anziché uno.

I files del progetto sono disponibili al link github.com/nicobargit/TrafficSignalDetection.