Alma Mater Studiorum – Università di Bologna

Scuola di Ingegneria e Architettura

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea

in

Computer Vision and Image Processing

Barcode Detection by Neural Networks on Android Mobile Platforms

Candidato:

Mattia Michelini

Relatore:

Prof. Luigi Di Stefano

Anno Accademico 2020/2021

Abstract

Lo scopo di questa esperienza di tesi è stato quello di fare un confronto sul tema delle reti neurali e, in particolare, sul modo di portare la potenza inferenziale di questi modelli nel mondo mobile. Il caso di studio interessa i codici a barre e, nello specifico, l'obiettivo è stato quello di riuscire a identificare la zona in cui questi si trovavano, in modo poi da avere una zona minore da indagare con un altro algoritmo specifico per la decodifica (cosa che esula dallo scopo della tesi). Questo è chiaramente un problema di object detection e, per risolverlo, ho esplorato due differenti tipi di approccio: in prima istanza ho indagato la funzione di object detection di ML Kit, per poi passare ad utilizzare direttamente un modello a mia scelta tramite TensorFlow Lite.

Sommario

Introduzione

Nella seguente trattazione ho messo la quasi totalità dell'enfasi sul secondo tipo di approccio che ho indagato, quello basato sul modello in formato TensorFlow Lite, in quanto è stato quello che ha richiesto la maggior quantità di tempo, sia per quanto riguarda lo studio effettivo della soluzione da utilizzare che per quanto riguarda la realizzazione stessa.

In particolare, ho impostato il lavoro di tesi, basato sulla mia esperienza di tirocinio in Datalogic, dividendo la trattazione in due macro-blocchi: uno che possiamo intendere come la parte più teorica, dove vado a descrivere le tecnologie che sono risultate per me nuove e importanti per portare a termine il lavoro, e una più prettamente pratica, dove vado a descrivere i vari passaggi per arrivare al risultato finale. In questa seconda parte ho cercato di organizzare il discorso in maniera tale che rispecchiasse il più possibile la naturale evoluzione del progetto.

Infine, nella parte conclusiva ho tirato le somme di questa esperienza, non prima però di aver fatto un paragone tra le due soluzioni, andando anche a comparare le tempistiche inferenziali. In questa fase ho riassunto anche quelli che potrebbero essere i futuri sviluppi o le modifiche che secondo me potrebbero essere utili per migliorare il risultato finale.

1 - Background teorico

1.1 - Camera X

Camera X [1] è una Jetpack support library introdotta da Google per rendere più facile lo sviluppo di applicazioni Android per la fotocamera. Proprio questa sua dichiarata "semplicità" in confronto alla più utilizzata Camera2 e il fatto di essere un novizio di sviluppo su questa piattaforma, mi hanno portato a provare questo nuovo approccio, caldeggiato anche sui siti di documentazione. Al momento dello sviluppo dell'applicazione finale Camera X si trovava ancora in alfa.

Camera X utilizza un approccio basato su casi d'uso che sono coscienti del lifecycle dell'applicazione; questi sono:

- Preview: caso d'uso che serve per far vedere l'immagine sul display dello smartphone
- Image Capture: caso d'uso (non utilizzato in questo progetto) che permette di scattare foto e di gestirne il contesto
- Image Analysis: questo caso d'uso, centrale nell'applicazione finale, permette di accedere al flusso di dati proveniente dalla camera e di passarlo allo strato sottostante, in maniera tale da fare elaborazione sull'immagine

In particolare, quest'ultimo caso d'uso è ideale per fare processing sull'immagine, computer vision o utilizzare il potere inferenziale del machine learning (utilizzando reti custom oppure ML Kit). Tutto questo viene fatto nel metodo analyze dell'applicazione, che può lavorare in due differenti modi: bloccante o non bloccante. La scelta è molto importante in quanto, nel primo caso, vengono processati tutti i frame catturati e, se il metodo di analisi impiega più tempo rispetto a quello di cattura dell'immagine, questo rimane indietro. Nel secondo caso invece il frame che non risulta più attuale viene scartato a favore del più recente, cosa che nella mia applicazione aiuta sicuramente a far apparire il tutto più responsive in quanto non riesce a lavorare esattamente in tempo reale.

Questa divisione in casi d'uso, che permette di scomporre il problema in differenti task, unito al fatto che utilizzando Camera X non si devono gestire i problemi che riguardano la compatibilità con i vari dispositivi, sono l'essenza per cui questo è definito un approccio facilitato rispetto allo stato dell'arte attuale e per cui ho deciso di utilizzarlo nel mio lavoro di tesi.

1.2 - ML Kit

ML Kit [2] è un SDK per mobile che permette di portare il potere del machine learning di Google su dispositivi Android e iOS in maniera facile e veloce. Non è necessaria infatti nessuna conoscenza specifica di reti neurali, o più in generale di deep learning, e richiedono la scrittura di "poche" righe di codice per l'inclusione di queste funzionalità. Se invece si hanno le conoscenze sopra citate e il proprio caso di studio non viene coperto dalle funzionalità base offerte, si può caricare il proprio modello TensorFlow Lite su Firebase e ML Kit funziona come APIs, permettendo di usare la propria rete custom per fare inferenza all'interno dell'applicazione.

ML Kit sfrutta le principali tecnologie di Google per il machine learning, quali Google Cloud Vision API, TensorFlow Lite e Neural Network API di Android, e le mette insieme in un singolo SDK molto potente e facile da usare. La versatilità insita nello sfruttare questo approccio sta nel fatto che, a seconda del compito che bisogna affrontare, si può decidere se sfruttare il potere computazionale del Cloud, nel caso in cui il task sia molto complesso o se si vogliono livelli di accuratezza molto elevati, oppure se lasciare l'inferenza totalmente sullo smartphone, se quello che si richiede dall'applicazione è il lavoro in real-time. In questo caso si possono utilizzare le funzionalità "base" offerte da ML Kit, che sono basate su modelli altamente ottimizzati e molto allenati. Figura 1 mostra queste funzionalità.

What features are available on device or in the cloud?

Feature	On-device	Cloud
Text recognition	✓	✓
Face detection	✓	
Barcode scanning	✓	
Image labeling	✓	✓
Object detection & tracking	✓	
Landmark recognition		✓
Language identification	✓	
Translation	✓	
Smart Reply	✓	
AutoML model inference	✓	
Custom model inference	✓	

Figura 1. Feature disponibili con ML Kit [2]

Essendo ML Kit tuttora in versione beta, le ultime due funzionalità non erano disponibili al momento della stesura del piano per il mio lavoro. In particolare, custom model inference avrebbe potuto essere significativo per il mio progetto. Per questo motivo nel mio lavoro ML Kit viene sfruttato per fare inferenza solamente in una prima fase di test, dove si usano le funzioni built-in, e non nella versione finale dell'applicazione.

1.3 - TensorFlow Lite

Per questo progetto, dovendo portare il potere inferenziale di una rete neurale su mobile, sono dovuto ricorrere a TensorFlow Lite [3], un set di tool che permette di far funzionare modelli TensorFlow su dispositivi mobile, IoT e sistemi embedded. Il motivo per cui non si possono usare direttamente i modelli TensorFlow sta nel fatto che questi hanno dimensioni eccessive e non sono ottimizzati per lavorare su sistemi a bassa potenza o che comunque devono rispettare dei parametri di consumo di energia. Proprio per questo motivo nasce TensorFlow Lite che, grazie ai suoi componenti, permette di fare inferenza con bassa latenza e dimensioni dei file contenute, in relazione al modello e al contesto scelto.

I componenti principali da cui TensorFlow Lite è composto sono:

- L'interprete: questo ha lo scopo di far girare i modelli ottimizzati sui vari dispositivi quali smartphone, sistemi embedded o microcontrollori;
- Il convertitore: questo ha lo scopo di convertire il modello TensorFlow "originale" in una forma più efficiente in modo tale che l'interprete possa utilizzarlo in maniera ottimizzata. Solitamente in questa conversione vengono sostituite funzioni e modificati alcuni parametri, sia per avere migliori performance dal punto di vista computazionale sia per ridurre la dimensione del file binario stesso.

Proprio questa seconda parte è fondamentale se si vuole fare in modo che il processo di machine learning avvenga sul dispositivo, senza dover quindi comunicare con un server remoto che faccia la parte di elaborazione. In questo modo vengono garantite:

- Performance, sotto forma di basso tempo di latenza
- Privacy, in quanto nessun dato raccolto lascia il dispositivo per avere informazioni
- Basso consumo di batteria, in quanto l'utilizzo dell'hardware è ottimizzato e non sono richiesta connessioni di rete, che spesso risultano essere molto dispendiose in termini di consumo di energia

Ovviamente passando da modello TensorFlow completo a modello TensorFlow Lite non tutte le operazioni sono "ammesse": il set di conversioni di funzioni ottimizzate è infatti limitato a quelle funzioni di maggior uso e diffusione [4]. Per tutte le altre funzioni che non rientrano in questo insieme, che è comunque abbastanza ampio, invece si perdono le agevolazioni citate sopra e l'impatto varia a seconda della complessità dell'operazione specifica che si sta andando ad eseguire, in quanto questa deve essere trasportata per intero nel nostro file.

Oltre alle conversioni fatte in automatico ci sono altri modi per migliorare le prestazioni della rete in termini di tempistiche di risposte e uso di risorse. In particolare, possiamo trovare: quantizzazione, pruning e utilizzo di hardware dedicato [5]. Questo processo di ottimizzazione non è da valutare a posteriori ma è più fruttuoso se viene considerato di pari passo allo sviluppo dell'applicazione, in quanto alcuni passaggi modificano il comportamento dell'intero insieme.

La quantizzazione

Il discorso della quantizzazione è di particolare interesse in quanto può avere un notevole impatto sull'intera rete, andando a ridurre la latenza per fare inferenza ma spesso comportando anche una riduzione nell'accuratezza generale della risposta della rete. Quanto questo processo impatti sul risultato finale non è facile da predire, per questo motivo spesso si procede per tentativi. Nello specifico, quantizzare significa andare a ridurre la precisione dei numeri usati per descrivere i parametri del modello. Fare questa operazione quando si usa un dispositivo potente, come può essere un computer, ha tendenzialmente poco senso in quanto il guadagno in termini di prestazioni è minimo a fronte di un probabile peggioramento non trascurabile dei risultati. Questo ragionamento sul trade-off però è molto più difficile da fare per quanto riguarda il campo mobile, dove i vincoli di potenza e consumo sono molto più stringenti. Utilizzando la quantizzazione, che può essere fatta sia sul risultato finale della rete sia sull'intero processo di allenamento, si possono ottenere delle performance nettamente superiori, non andando ad intaccare più di tanto le performance di inferenza.

Technique	Data requirements	Size reduction	Accuracy	Supported hardware
Post-training float16 quantization	No data	Up to 50%	Insignificant accuracy loss	CPU, GPU
Post-training dynamic range quantization	No data	Up to 75%	Accuracy loss	CPU, GPU (Android)
Post-training integer quantization	Unlabelled representative sample	Up to 75%	Smaller accuracy loss	CPU, GPU (Android), EdgeTPU, Hexagon DSP
Quantization-aware training	Labelled training data	Up to 75%	Smallest accuracy loss	CPU, GPU (Android), EdgeTPU, Hexagon DSP

Tabella 1. Comparazione tra quantizzazioni [5]

Model	Top-1 Accuracy (Original)	Top-1 Accuracy (Post Training Quantized)	Top-1 Accuracy (Quantization Aware Training)	Latency (Original) (ms)	Latency (Post Training Quantized) (ms)	Latency (Quantization Aware Training) (ms)	Size (Original) (MB)	Size (Optimized) (MB)
Mobilenet-v1-1-224	0.709	0.657	0.70	124	112	64	16.9	4.3
Mobilenet-v2-1-224	0.719	0.637	0.709	89	98	54	14	3.6
Inception_v3	0.78	0.772	0.775	1130	845	543	95.7	23.9
Resnet_v2_101	0.770	0.768	N/A	3973	2868	N/A	178.3	44.9

Tabella 2. Comparazione tra diverse reti e confronto tra i tipi di quantizzazioni [5]

Tabella 1 e 2 mostrano gli effetti dell'uso della quantizzazione per quanto riguarda le dimensioni finali della rete in formato TensorFlow Lite e i relativi risultati di inferenza (i numeri sono riferiti ad un Pixel2 facendo uso della CPU) considerando diversi tipi di reti.

Hardware dedicato

L'utilizzo di hardware dedicato è l'altro aspetto di maggior rilevanza per quanto riguarda la questione dell'ottimizzazione delle prestazioni: nel contesto di TensorFlow Lite questi prendono il nome di delegati. Di particolare interesse per il mio progetto sono stati i delegati GPU e quelli NNAPI.

NNAPI

NNAPI sta per Android Neural Network API, API disponibili su tutti i dispositivi Android che utilizzano la versione 8.1 o successive (API level ≥ 27); queste servono per accelerare i modelli TensorFlow Lite che girano su Android e sui cui dispositivi è presente dell'hardware in grado di garantire questa accelerazione, come ad esempio GPU, NPU (Neural Processing Unit) o DSP (Digital Signal Processor).

Prima dell'introduzione di queste API, degli SDK per fare task computazionalmente intensivi erano stati proposti direttamente dai produttori di SoC. Questo aveva dato luogo ad un ecosistema molto variegato, dove una soluzione funzionava solamente su una determinata piattaforma; in questo modo si avevano:

- Lo Snapdragon Neural Processing Engine (SNPE) per quanto riguarda Qualcomm
- La piattaforma HiAI per far funzionare task di relativi alle reti neurali sulle NPU di Kirin
- NeuroPilot SDK per quanto riguarda la piattaforma MediaTek

Le NNAPI ricoprono sostanzialmente un ruolo da intermediario tra un framework di alto livello e le risorse hardware che si trovano su un determinato dispositivo, e hanno la responsabilità di mettere in comunicazione tra loro tutti i vari componenti e task, oltre che a gestirne la corretta esecuzione e scheduling sulla risorsa che è più idonea. Per far sì che queste possano funzionare è comunque richiesto che il vendor del SoC implementi dei driver appositi, dato che il fatto di avere un dispositivo che utilizzi Android 8.1, o una versione successiva di questo, non è garanzia del supporto dell'accelerazione NNAPI.

Ovviamente il boost di performance ottenuto sfruttando questi delegati, se disponibili, varia a seconda della tipologia di hardware presente sul dispositivo mobile. Quindi, utilizzare un NNAPI delegate [6] può favorire le prestazioni della rete ma non sempre questo miglioramento è evidente o scontato, e va quindi verificata la sua efficacia facendo dei test pratici. È possibile anzi che utilizzando questa tipologia di delegati si vada a far peggiorare le performance della rete: questo si può avere in quei casi in cui questo delegato non supporta alcune parti del modello o alcune combinazioni di parametri, e quindi riesce ad elaborare solamente alcune parti del grafo e unicamente su queste effettua l'accelerazione. Le altre vengono eseguite dalla CPU, ottenendo così un'esecuzione separata. Questo tipo di esecuzione, a causa degli elevati costi di sincronizzazione, può portare ad avere prestazioni peggiori rispetto al non utilizzare questa forma di delegato e lasciare tutto il carico direttamente alla CPU. Questo fenomeno non è caratteristico dei soli delegati NNAPI ma può coinvolgere un qualsiasi tipo di delegato.

L'altro tipo di delegato è quello della GPU [7]. Come nel caso dei computer le GPU sono pensate per avere un alto throughput per lavori altamente parallelizzabili, come quando si utilizzano le reti neurali. Le GPU computano con numeri float a 16 o 32 bit e per questo motivo non è necessaria la quantizzazione per avere delle performance ottimali. Le GPU risultano inoltre avere un impatto minore sul consumo della batteria quando si parla di fare inferenza, data la loro efficienza e ottimizzazione per questo tipo di task, andando a generare anche meno calore rispetto allo stesso task eseguito sulla CPU.

Sotto sono riportate due tabelle che mostrano i migliori smartphone e SoC disponibili sul mercato per fare inferenza secondo AI-Benchmark [8], un benchmark composto da diversi test (tutti a tema reti neurali) volti a misurare la potenza computazionale di un dispositivo Android. Come mostrano chiaramente gli score in tabella 3, la presenza di hardware dedicato fa sì che i dispositivi dotati di SoC Kirin la facciano da padrone nelle prestazioni generali testate, anche se questo non risulta essere il miglior SoC tra quelli testati (vedi tabella 4).

Model	CPU	RAM	Year	QUANT Score	QUANT Accuracy	FP16 Score	FP16 Accuracy	Accuracy	AI-Score
Honor 30 Pro+	HiSilicon Kirin 990 5G	8GB	2020	9155	79	53287	93	89	105034 [1.6]
Huawei P40	HiSilicon Kirin 990 5G	8GB	2020	8919	79	51499	93	89	102065 [1.6]
Huawei Mate 30 Pro 5G	HiSilicon Kirin 990 5G	8GB	2019	6899	79	38042	93	88	76206 [1.6]
Honor V30 Pro 5G	HiSilicon Kirin 990 5G	8GB	2019	6658	79	38190	93	88	76120 [1.6]
Huawei Mate 30 5G	HiSilicon Kirin 990 5G	8GB	2019	6792	79	37703	93	88	75524 [1.6]
Huawei Mate 30 Pro	HiSilicon Kirin 990	8GB	2019	6428	79	37268	93	89	75311 [1.6]
Huawei Mate 30	HiSilicon Kirin 990	8GB	2019	6422	79	37025	93	89	74895 [1.6]
Huawei nova 7 5G	HiSilicon Kirin 985	8GB	2020	5977	79	36523	93	89	73866 [1.6]
Honor 30	HiSilicon Kirin 985	8GB	2020	5989	79	36439	93	89	73679 [1.6]
Huawei nova 7 Pro 5G	HiSilicon Kirin 985	8GB	2020	5995	79	36346	93	89	73522 [1.6]
Honor X10	HiSilicon Kirin 820	8GB	2020	4719	79	29678	93	89	60961 [1.6]
Honor 30s	HiSilicon Kirin 820	8GB	2020	4701	79	29420	93	89	60843 [1.6]
Oppo Reno3	Mediatek Dimensity 1000L	12GB	2019	10079	94	20864	92	93	58628 [1]
Samsung S20 Ultra 5G	Exynos 990	12GB	2020	3300	59	14604	50	53	40303 [1.9]

Tabella 3. Migliori smartphone sul mercato per fare inferenza [tabella modificata da [9]]

Processor	CPU Cores	AI Accelerator	AI Score, K
MediaTek Dimensity 1000+	4x2.6 GHz Cortex-A77 & 4x2.0 GHz Cortex-A55	APU 3.0 (6 cores)	88.1
HiSilicon Kirin 990 5G	2x2.86 + 2x2.36 GHz A76 & 4x1.95 GHz A55	NPU (3 cores, Da Vinci series)	77.3 [1]
HiSilicon Kirin 990 5G	2x2.86 + 2x2.36 GHz A76 & 4x1.95 GHz A55	NPU (3 cores, Da Vinci series)	51.5
Snapdragon 865	1x2.84 + 3x2.42 + 4x1.80 GHz Kryo 585	DSP (Hex. 698) + GPU (Adreno 650)	50.5
Snapdragon 855 Plus	1x2.96 + 3x2.42 + 4x1.80 GHz Kryo 485	DSP (Hex. 690) + GPU (Adreno 640)	47.0
Exynos 990	2x2.73 GHz M5 & 2x2.5 GHz A76 & 4x2 GHz A55	GPU (Mali-G77 MP11)	45.5
Snapdragon 855	1x2.84 + 3x2.41 + 4x1.78 GHz Kryo 485	DSP (Hex. 690) + GPU (Adreno 640)	40.7
Exynos 9825 Octa	2x2.7 GHz M4 & 2x2.4 GHz A75 & 4x1.9 GHz A55	GPU (Mali-G76 MP12)	38.3
HiSilicon Kirin 820	1x2.36 + 3x2.22 GHz A76 & 4x1.84GHz A55	NPU (Da Vinci series)	37.7
HiSilicon Kirin 990	2x2.86 + 2x2.09 GHz A76 & 4x1.86 GHz A55	NPU (2 cores, Da Vinci series)	37.5
Exynos 9820 Octa	2x2.7 GHz M4 & 2x2.3 GHz A75 & 4x2 GHz A55	GPU (Mali-G76 MP12)	36.8
Snapdragon 768G	1x2.8 GHz & 1x2.2 GHz & 6x1.8 GHz Kryo 475	DSP (Hex. 696) + GPU (Adreno 620)	31.8 [1]
Q855 + Pixel Neural Core	1x2.84 + 3x2.41 + 4x1.78 GHz Kryo 485	Hex. DSP + Adreno GPU + Google TPU	28.6
Snapdragon 765G	1x2.4 GHz & 1x2.2 GHz & 6x1.8 GHz Kryo 475	DSP (Hex. 696) + GPU (Adreno 620)	25.0
Snapdragon 845	4x2.8 GHz Kryo 385/G & 4x1.7 GHz Kryo 385/S	DSP (Hex. 685) + GPU (Adreno 630)	24.5
Exynos 980	2x2.2 GHz Cortex-A77 & 6x1.8 GHz Cortex-A55	GPU (Mali-G76 MP5)	24.2

Tabella 4. Migliori SoC sul mercato per fare inferenza [tabella modificata da [10]]

È importante a questo punto fare una precisazione: sebbene sia possibile far girare tutto questo anche su dispositivi iOS la compatibilità di tutte le funzionalità è da considerare nella migliore delle ipotesi limitata e, seppur a livello nominale funzioni, tutte le fonti incontrate durante il mio percorso sconsigliano di utilizzare questo approccio sui device mobile Apple.

Ricontrollando le mie fonti per la stesura di questa Tesi ho visto che, dopo la fine del mio lavoro, TensorFlow Lite ha cominciato a sperimentare (versione beta) altri tipi di delegati più legati alla piattaforma, come Hexagon delegate e Core ML delegate. Lo scopo di questi nel primo caso è far funzionare ancora meglio l'approccio con delegati NNAPI sulla piattaforma Snapdragon, mentre per quanto riguarda i secondi l'obiettivo è quello di abilitare l'esecuzione dei modelli TensorFlow Lite sul framework Core ML, in modo da permettere una velocità di inferenza maggiore sui dispositivi iOS.

Questo è il quadro generale per quanto riguarda TensorFlow Lite. Per quanto concerne invece il mio lavoro, ho deciso di non ricorrere all'uso della quantizzazione per un duplice motivo: prima di tutto sono convinto, così come gli autori di [11], che nel prossimo futuro il rapporto potenza/efficienza dell'hardware dei dispositivi mobile andrà a migliorare ulteriormente e che non ci sarà quindi più bisogno di questo tipo di ottimizzazione, così come è stato nel caso dei computer fissi; in secondo luogo, mentre stavo testando delle reti di esempio sul mio dispositivo, ho notato che sfruttando i delegati della GPU ottenevo delle prestazioni migliori in termini di latenza rispetto a quando utilizzavo i delegati NNAPI. Inoltre, rimuovere lo step di quantizzazione della rete si è rivelato essere conveniente al momento della versione definitiva del modello, in quanto questo processo non è descritto in maniera molto chiara sulla documentazione.

Ci tengo a sottolineare che la strada che ho imboccato per questo progetto non è unica, anche se è la sola che ho esplorato. La scelta di utilizzare l'ecosistema di TensorFlow per questo progetto è dovuta ad una duplice motivazione. In primo luogo, utilizzando questo si ha possibilità di lavorare con un qualcosa di totalmente integrato e pensato per essere sfruttato in maniera lineare: la documentazione è abbondante e gli esempi sono tanti, anche se purtroppo la velocità con cui tutto questo si sta sviluppando porta ad avere grandi conflitti tra le varie versioni. In secondo luogo, avevo già utilizzato TensorFlow in un precedente progetto, in quando caldeggiato dai più data la sua forte integrazione con Keras e il suo netto miglioramento con la versione 2.0.

1.4 - Dataset Sintetico

Per dataset sintetico si intende un insieme di immagini annotate che non sono reali ma che vengono generate da un algoritmo o da reti neurali in maniera artificiale.

La scelta di utilizzare un dataset sintetico non è stata volontaria, bensì dettata dalle necessità temporali della mia esperienza di tesi, in quanto in azienda non era disponibile nessun dataset di immagini già predisposto con tutti i dati di labeling che fosse allo stesso tempo compatibile con lo stile delle immagini generate da una fotocamera di un dispositivo Android. Se per allenare la rete si fosse utilizzato un dataset di immagini reali e di dimensione adeguata si sarebbero ottenute plausibilmente, a parità di tutte le altre condizioni, delle abilità di riconoscimento superiori. Detto questo, a mio avviso il risultato è stato comunque più che soddisfacente.

Per questo motivo ho deciso di allenare la rete neurale alla base del processo inferenziale con un dataset sintetico. Secondo le fonti che ho consultato [12], le abilità inferenziali di una rete allenata su un dataset sintetico sono inferiori rispetto all'utilizzo di dataset reale con circa le stesse caratteristiche. Nonostante questo, si riescono ad ottenere comunque degli ottimi risultati dato che il dataset sintetico, non avendo bisogno di tempo uomo per la fase di labelling, può essere di dimensioni praticamente illimitate. Le performance di un dataset sintetico rispetto a quelle di un dataset reale dipendono, oltre che dalla qualità delle immagini utilizzate, anche dal contesto applicativo in cui si sta utilizzando il tutto, dato che alcuni si prestano meglio all'utilizzo di questa tecnica rispetto ad altri.

Per creare questo dataset ho seguito il corso di Adam Kelly su una piattaforma online [13] che mi ha permesso di imparare passo passo come fosse strutturato il formato del dataset che sono andato ad utilizzare e a capire e sfruttare il suo codice per replicare questa struttura in maniera automatizzata per tutte le immagini generate. Questo passo è stato necessario per un duplice motivo: in primo luogo non avevo conoscenze abbastanza approfondite sulla struttura specifica del dataset che mi permettessero di andare a replicare le annotazioni in maniera automatica; in secondo luogo non avevo, e non ho ancora adesso, abbastanza esperienza con la manipolazione di immagini per scrivere del codice da zero in grado di creare dei risultati sufficientemente realistici per realizzare un allenamento efficace di una rete neurale.

Il dataset alla base di questo lavoro è stato quindi generato ricorrendo ad un programma che, partendo da immagini di foreground (figura 2 e 3) e di background (figura 4), è in grado di andare a generare delle immagini artificiali. In questo caso, le immagini di foreground sono i

codici a barre, che ho ricavato partendo da immagini reali ottenute esplorando altri dataset online, dalle quali ho però eliminato ogni qualsivoglia tipologia di sfondo. In questo mio lavoro ho indagato solamente due tipologie di codice a barre: QR e EAN13. Per ognuna di queste due tipologie ho selezionato 30 codici differenti, in maniera tale da avere un minimo di varietà all'interno di ogni categoria, cercando anche di selezionare quelli che presentavano angolazioni particolari o che fossero situati su oggetti sferici, in modo tale da avere anche varietà della forma e di inclinazioni della camera. Per quanto riguarda gli sfondi invece, ho utilizzato un grande numero di scatole poste le une vicino alle altre, in maniera da avere colori differenti, scritte di diverso stile e dimensione.

Figura 2. Esempio di immagine di foreground: QR

Figura 3. Esempio di immagine di foreground: EAN13

Figura 4. Esempio di immagine di background

L'algoritmo di generazione di queste immagini opera nel seguente modo:

- Come operazione preliminare è necessario impostare diversi parametri di lavoro dell'algoritmo, come ad esempio la risoluzione finale delle immagini da generare, il numero massimo di elementi principali presenti nell'immagine o quali effetti si vogliono utilizzare nella generazione di queste. Gli "effetti" che ho utilizzato sono: rotazione, variazioni della luminosità dell'oggetto principale e riduzione delle dimensioni di questo. Queste sono anche la totalità delle feature che si possono andare ad utilizzare e penso che siano molto appropriate ad un contesto di machine learning, in quanto riescono a rendere il riconoscitore molto robusto sotto i più comuni cambiamenti di condizioni di un ambiente non controllato, quali posizione

della camera e condizioni di luce. Una feature che invece avrei voluto avere a disposizione, per questo caso specifico dei codici a barre, sarebbe stata quella della non occlusione degli elementi principali tra di loro, in modo tale da avere una rappresentazione della norma molto più numericamente corposa per poi andare ad aggiungere, nel caso, qualche immagine di questo caso particolare.

- Questo algoritmo genera delle annotazioni in stile COCO (uno dei più grandi dataset per fare object detection e non solo [14], [15]) in maniera automatica durante la generazione delle immagini. Per fare questo però è necessario andare a definire prima dei file in cui si danno delle informazioni di contesto sul dataset. Queste non sono significative nel nostro caso essendo un dataset "giocattolo", ma è comunque essenziale che siano presenti per uniformità di trattazione del dataset. Infatti, la scelta di trovare un algoritmo che mi generasse un dataset con le informazioni COCO-like è stata un perno della mia ricerca: essendo questo una dei dataset generalisti più famosi in circolazione, il supporto per questo è molto ampio ed esistono una moltitudine di funzioni per gestirlo al meglio e operare su questo. Ad esempio, lo script per passare da annotazioni COCO-like al formato TFRecord, necessario per poi andare ad allenare la rete, viene offerto direttamente da Google. Non dico che non fosse possibile andare a salvare le informazioni sulle annotazioni in un qualsiasi altro modo, anche inventato da noi, ma dopo sarebbe stato necessario andare a generare tutti questi script per le traduzioni, cosa possibile ma abbastanza prona ad errori. Infine, c'è da sottolineare come tutte le immagini di foreground debbano essere salvate in una determinata gerarchia di cartelle in maniera tale che le relazioni di categoria-superclasse vengano definite correttamente nello stile COCO-like.
- Una volta settati tutti questi parametri il programma andrà a generare il numero di immagini desiderato con tutte le varie opzioni selezionate. L'algoritmo nella pratica opera nel seguente modo: viene selezionato uno sfondo casuale e di questo viene presa una porzione casuale che corrisponde alla risoluzione settata. Su questo vengono poi inseriti da 1 a N, dove N è il numero massimo di elementi principali che si è deciso di inserire sulla stessa immagine, elementi di foreground con variazioni di luminosità, rotazioni e posizioni del tutto casuali. Una volta generate le immagini (esempio in figura 5) vengono anche creati tutti quei file di annotazioni fondamentali per il corretto allenamento di una rete neurale.

Figura 5. Esempio di immagine generata dall'algoritmo

Una volta completato l'allenamento della rete e riuscito il trasferimento sull'applicazione Android ho notato alcuni comportamenti problematici, che differenti scelte sulle immagini di sfondo per comporre il dataset avrebbero potuto risolvere. Prima di tutto voglio dire che una maggiore varietà all'interno delle immagini di foreground avrebbe sicuramente aiutato a migliorare il riconoscimento, anche se, vedendo come opera l'applicazione, non sono sicuro che sia un punto significativo, dato che questa lavora solamente sulla localizzazione. Invece, quello che avrebbe sicuramente aiutato sarebbe stata una scelta di sfondi più sfidanti per la rete: avendo giocato un po' con l'applicazione, risulta chiaro che la sua astrazione per codice QR sia una forma più o meno trapezoidale con all'interno altre forme simili di colori opposti, mentre gli EAN13 sono pattern di linee alternate in forme oblunghe. In questo caso, sono praticamente certo che andando ad aggiungere al mio pool di sfondi qualche pattern a scacchiera o zebrato avrei reso nettamente meno prona a falsi positivi la mia applicazione. Detto ciò, devo dire che i risultati della rete mi hanno sinceramente sorpreso in senso positivo, in quanto riesce a localizzare la quasi totalità delle due categorie in un contesto reale.

1.5 - Object Detection

In generale, la object detection ha lo scopo di localizzare e classificare gli oggetti di interesse presenti in un'immagine e di associare a questi una bounding box e un intervallo di confidenza. La struttura dell'identificazione degli oggetti può sostanzialmente essere divisa in due macrocategorie: la prima è formata da quei metodi che si affidano ad una struttura a parte per la proposta di regioni e poi vanno a classificarle in un secondo momento, mentre la seconda è composta da quelle che non sfruttano componenti per la proposta di regioni di interesse ma che affrontano il processo come un unico problema di regressione.

Della prima categoria fanno parte tutte quelle reti che utilizzano, a parte oppure al loro interno, una Region Proposal Network per ricevere e identificare le zone di interesse come, ad esempio, la famiglia R-CNN [16]. In un primo momento viene generato dalla RPN un numero di regioni fisso per ogni immagine, quantità che varia a seconda dell'architettura che si sta considerando; ognuna di queste viene poi presa e manipolata in modo che tutte le regioni proposte abbiano una dimensione predeterminata fissa, in modo che queste possano essere gestite dalla rete vera e propria così da estrarre tutte le features necessarie per identificare l'oggetto presente in esse. In generale, il processo di allenamento di questa rete è lungo e molto dispendioso dal punto di vista computazionale, in quanto bisogna condurre due allenamenti separati per le due componenti dell'architettura. Quello appena descritto è il comportamento di una R-CNN: le sue versioni successive sono riuscite a rendere volta per volta l'intero procedimento molto più efficiente, oltre che migliorare la rete stessa, ma non al punto da competere in termini di efficienza computazionale con i membri della seconda categoria. Questi secondi framework sono infatti molto più veloci e leggeri dato che il processo di object detection è realizzato in un unico passaggio: si parte dall'immagine di input e si ottengono le coordinate delle bounding box e la confidenza che la rete ha sulla localizzazione dell'oggetto all'interno di questa. I due principali membri di questa categoria sono le famiglie di reti You Only Look Once (YOLO) [17] e Single Shot multibox Detector (SSD) [18], che sfruttano delle regioni decise a priori all'interno delle quali poi vanno a ricercare le bounding box degli oggetti. Lo svantaggio di queste ultime rispetto alle prime è il fatto che tendenzialmente hanno dei risultati, in termini di riconoscimento, leggermente inferiori, anche se rimangono comunque molto competitive.

Considerando quanto appena detto e considerato lo scopo di questo progetto, ho deciso di orientarmi su strutture che utilizzassero la seconda famiglia di reti, essendo queste le uniche in grado di poter lavorare in real time. La tabella qui riportata (tabella 5) tratta da [19], mostra i

ratei di riconoscimento e gli fps che si sono ottenuti dalle varie reti applicandole allo stesso contesto di lavoro. Tutti i numeri qui riportati sono relativi a macchine equipaggiate con hardware molto importante: parliamo di processore Intel i-7 6700k e GPU NVIDIA Titan X, quindi potenze molto distanti dal mondo mobile.

Methods	Trained on	mAP(%)	Test time(sec/img)	Rate(FPS)
SS+R-CNN	07	66.0	32.84	0.03
SS+SPP-net	07	63.1	2.3	0.44
SS+FRCN	07+12	66.9	1.72	0.6
SDP+CRC	07	68.9	0.47	2.1
SS+HyperNet*	07+12	76.3	0.20	5
MR-CNN&S-CNN	07+12	78.2	30	0.03
ION	07+12+S	79.2	1.92	0.5
Faster R-CNN(VGG16)	07+12	73.2	0.11	9.1
Faster R-CNN(ResNet101)	07+12	**83.8**	2.24	0.4
YOLO	07+12	63.4	**0.02**	**45**
SSD300	07+12	74.3	**0.02**	**46**
SSD512	07+12	76.8	0.05	19
R-FCN(ResNet101)	07+12+coco	83.6	0.17	5.9
YOLOv2(544*544)	07+12	78.6	0.03	40
DSSD321(ResNet101)	07+12	78.6	0.07	13.6
DSOD300	07+12+coco	81.7	0.06	17.4
PVANET+	07+12+coco	**83.8**	0.05	21.7
PVANET+(compress)	07+12+coco	82.9	0.03	31.3

Tabella 5. Confronto tra diverse reti in termini di accuratezza e fps [19]

La mia ricerca è partita quindi dalle reti che in questa tabella si sono dimostrate essere le più promettenti, per cui ho indagato le seguenti architetture: PVANET [20], DSSD [21], DSOD [22], SSD [18] e YOLO v2 [23]. Tutte queste però sono state scartate perché decisamente più lente e di dimensioni nettamente superiori rispetto alla rete scelta alla fine per questo progetto. Nel dettaglio, la prima utilizza un approccio che sfrutta una RPN e quindi risulta già una scelta errata se lo scopo è il lavoro in real time; la seconda e la terza sfruttano la stessa logica di SSD, andando però a cambiare la rete che sta alla base di questa (la scelta di questo cambiamento in entrambi i casi è stata fatta per avere una rete più precisa invece che più efficiente e quindi hanno un peggioramento delle prestazioni per quanto riguarda la latenza). SSD e YOLO v2 erano le mie alternative migliori fino a quando non ho scoperto MobileNet [24].

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
GoogleNet	69.8%	1550	6.8
VGG 16	71.5%	15300	138

Tabella 6. Confronto tra diversi modelli in termini di riconoscimento (su sfida di classificazione) e dimensione del modello stesso [20]

In questa tabella (tabella 6) viene sottolineato come le prestazioni di VGG16, rete che sta alla base di SSD, siano solo leggermente superiori rispetto a quelle di MobileNet in un contesto di classificazione, avendo però oltre 30 volte il numero di parametri di questa.

Framework Resolution	Model	mAP	Billion Mult-Adds	Million Parameters
SSD 300	deeplab-VGG	21.1%	34.9	33.1
	Inception V2	22.0%	3.8	13.7
	MobileNet	19.3%	1.2	6.8

Tabella 7. Confronto su reti base che utilizzano SSD [20]

In tabella 7 invece vengono comparate differenti reti base che utilizzano lo stesso framework per fare object detection, e MobileNet risulta essere nettamente la più leggera tra quelle considerate.

Network	mAP	Params	MAdd	CPU
SSD300	23.2	36.1M	35.2B	-
SSD512	26.8	36.1M	99.5B	-
YOLOv2	21.6	50.7M	17.5B	-
MNet V1 + SSDLite	22.2	5.1M	1.3B	270ms
MNet V2 + SSDLite	22.1	**4.3M**	**0.8B**	200ms

Tabella 8. Confronto tra reti per fare object detection che non sfruttano una RPN [21]

Infine, in quest'ultima tabella (tabella 8) vediamo la precisione delle varie reti rapportata al tempo di inferenza, questa volta misurato su un dispositivo mobile.

Non avendo mai lavorato su mobile, la mia ricerca di architetture da utilizzare per questo progetto era partita da metodi generali di object detection per andare a concentrarmi poi su quelli più efficienti, e questo è il motivo per il quale la mia ricerca è partita dalle reti sopra citate e non da MobileNet, che è nettamente più promettente. Se dovessi tornare indietro partirei sicuramente da questa e non indagherei nemmeno le altre soluzioni, dato che le prime non sono state pensate per il mondo di dispositivi mobile o embedded a "basse" prestazioni.

1.6 - MobileNet

Per il progetto ho scelto di utilizzare come rete neurale MobileNet nella sua seconda versione [25]. Questa è una rete che è stata pensata e sviluppata per lavorare in tutti quei contesti in cui i dispositivi hanno delle risorse computazionali limitate, come può essere ad esempio il mondo mobile o quello dei sistemi embedded. Lo scopo principale degli ideatori di questa rete è quello di permettere un lavoro in real time mantenendo comunque dei livelli di accuratezza molto elevati. Un altro pro di questa architettura, per quanto riguarda il mio progetto, è che questa è stata sviluppata da Google e, per questo motivo, si inserisce perfettamente nel workflow di TensorFlow Lite.

La principale novità di questa seconda versione rispetto alla prima è l'introduzione di un nuovo blocco strutturale per la costruzione della rete: inverted residual with linear bottleneck.

Prima di parlare di questo però penso che sia necessario descrivere la novità introdotta dalla precedente versione, in quanto essa è la base per la generazione di questo nuovo elemento. Infatti, nella prima versione di questa rete il layer fondamentale non era la classica convoluzione ma una versione più complessa di essa, che va a dividere in due momenti separati l'intero processo: uno per effettuare un filtraggio dell'input e uno per la creazione di nuove feature da passare allo strato successivo. Questo tipo di convoluzione prende il nome di depthwise separable convolution (vedi figura 6) ed è l'elemento chiave di questa rete neurale. In pratica, viene applicato un filtro di dimensione 3 x 3 ad ogni canale di input e poi il risultato di questa operazione viene elaborato con una convoluzione puntuale (filtro 1 x 1) che va a combinare tutti i canali. Questa operazione separata ha come risultato quello di ridurre drasticamente la computazione e la dimensione del modello, come riporta anche il paper di MobileNet [24]: rispetto alla convoluzione classica, quella impiegata da MobileNet risulta avere un costo computazionale tra le 8 e le 9 volte più piccolo. Proprio per questo motivo questo modello risulta essere così più efficiente, come mostrato in tabella 9, pagando solamente una lieve riduzione percentuale a livello di accuratezza.

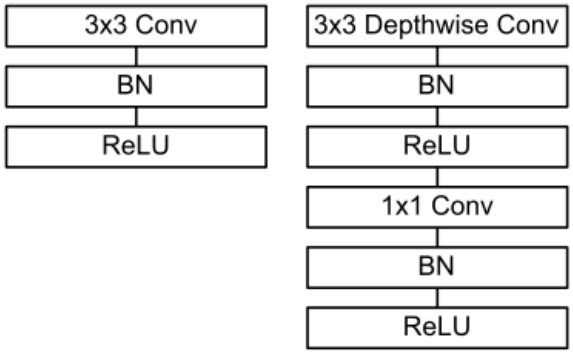

Figura 6. Sinistra: convoluzione standard.
Destra: convoluzione utilizzata in MobileNet [24]

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
Conv MobileNet	71.7%	4866	29.3
MobileNet	70.6%	569	4.2

Tabella 9. Confronto tra due versioni di MobileNet con convoluzione classica e depthwise separable [24]

Per meglio esplicitare quanto sia diversa l'impronta computazionale di questi due tipi di convoluzione facciamo un esempio. Prendiamo un caso in cui si ha uno strato convoluzionale 3 x 3 con in input 16 canali e in output 32. Nel caso di classica convoluzione si ha che ogni canale è attraversato da 32 filtri di dimensione 3 x 3, il che porta ad avere 16 x 32 = 512 feature maps, per un totale di 512 x 3 x 3 = 4608 parametri. Nel caso in cui invece per lo stesso esempio andassimo ad utilizzare la depthwise separable convolution, andremmo ad attraversare i 16 canali con solamente 1 filtro 3 x 3 per ogni canale, ottenendo così 16 feature maps. Queste 16 feature maps devono però ora essere attraversate da 32 filtri convoluzionali 1 x 1 per mixarle. Il computo totale dei parametri utilizzati in questo secondo caso risulta essere molto inferiore, essendo 16 x 3 x 3 = 144 per la prima operazione e 16 x 32 x 1 x 1 = 512 per la seconda, per un conto totale di 656 parametri.

La seconda versione di questa architettura modifica e arricchisce questo elemento, andando ad utilizzare il blocco inverted residual with linear bottleneck. Il motivo per il quale vogliamo introdurre un residual block sta nel voler fare in modo che il gradiente si possa propagare all'intera rete in modo migliore.

Il residual block classico (figura 7) sfrutta la connessione diretta tra blocchi per collegare due strati della rete che hanno un elevato numero di canali, seguendo un pattern largo → stretto → largo.

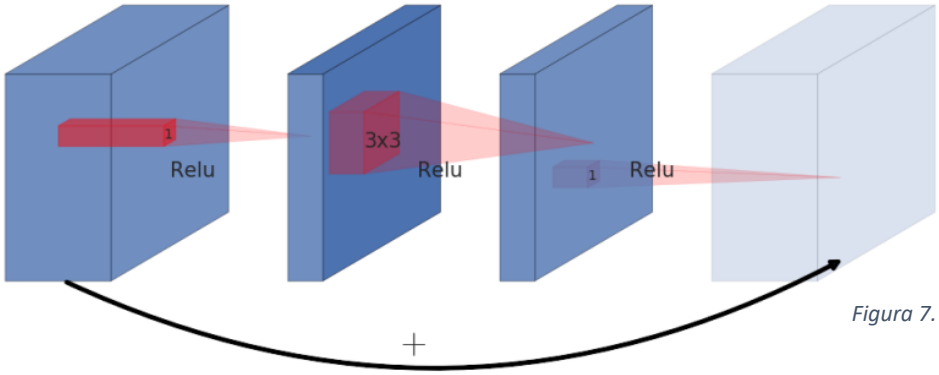

Figura 7. Residual block

MobileNet V2 invece ribalta totalmente questo approccio (figura 8), seguendo un pattern stretto → largo → stretto, il che permette a questo blocco di utilizzare un numero di parametri nettamente inferiore rispetto alla sua versione originale.

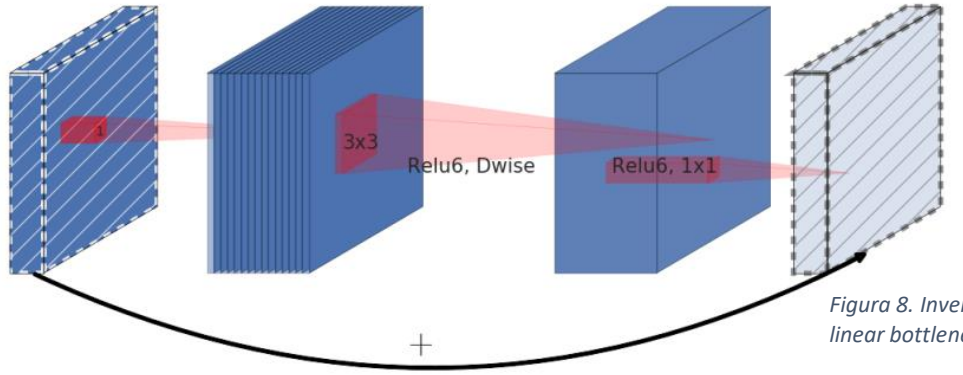

Figura 8. Inverted Residual block with linear bottleneck

Questo è possibile grazie all'introduzione di un collo di bottiglia che ha lo scopo di far calare il numero di canali degli strati connessi direttamente. Questa operazione viene motivata dal paper dicendo che le relazioni di interesse per la reti neurali in verità risiedono solamente in un sottospazio di tutti i canali che vengono considerati. Quando però si lavora su spazi piccoli, l'utilizzo di funzioni di attivazione non lineari provoca una perdita di informazioni, che di solito viene attenuata dall'elevato numero di canali, ma essendo in un caso in cui li vogliamo ridurre,

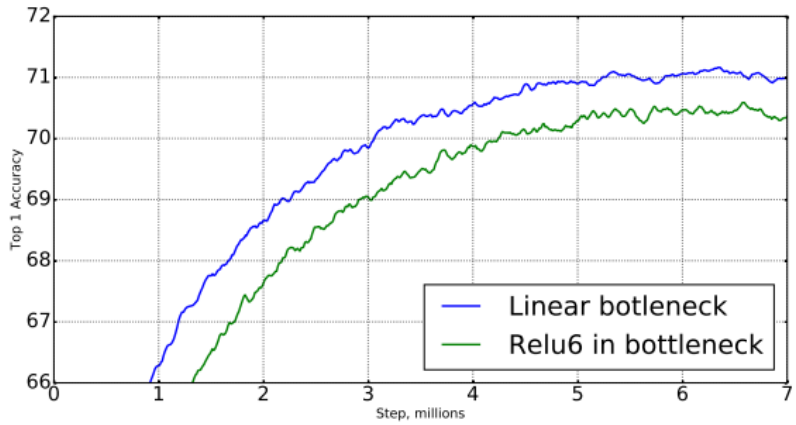

Figura 9. Comparazione tra l'accuratezza della rete nel caso in cui si usi un collo di bottiglia lineare e non lineare

utilizziamo una funzione lineare. I dati del paper (vedi figura 9) dimostrano infatti che l'utilizzo di un collo di bottiglia non lineare peggiora sensibilmente le prestazioni della rete.

Le seguenti immagini (figure 10-13) mostrano l'evoluzione del blocco convoluzionale per porta a quello di MobileNet V2, versione finale mostrata in figura 8.

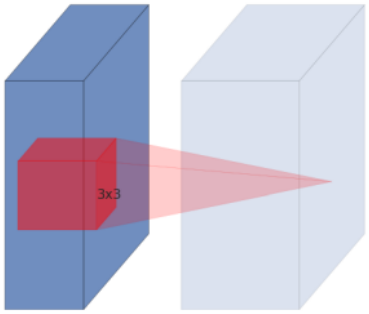

Figura 10. Convolution

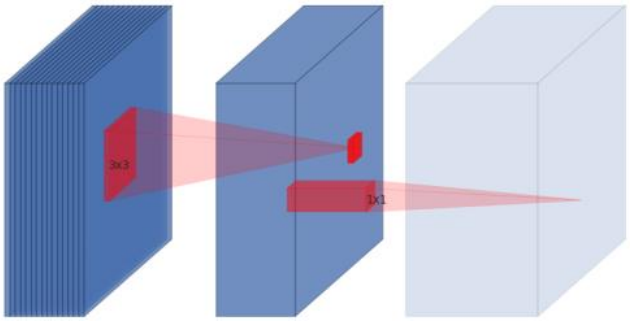

Figura 11. Separable convolution

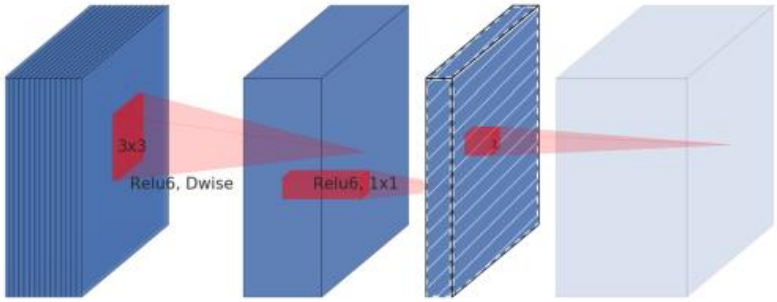

Figura 12. Separable bottleneck convolution

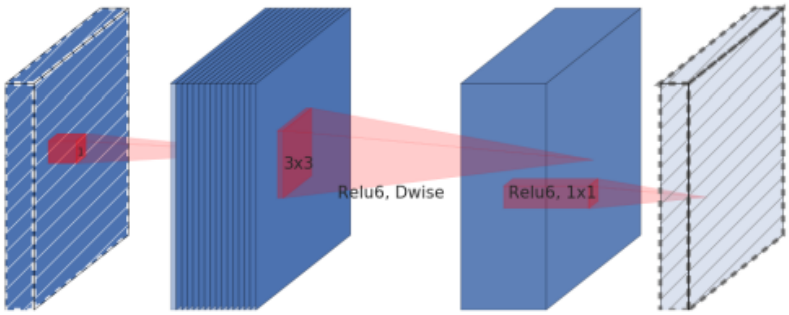

Figura 13. Expasion separable convolution

L'architettura di MobileNet V2 contiene inizialmente un layer totalmente convoluzionale con 32 filtri, seguito da 19 layer di residual bottleneck, descritti in precedenza. Tutti i layer utilizzano kernel di dimensione 3 x 3, cosa standard nelle reti neurali, e Relu6 quando si parla di non linearità, in quanto più robusta rispetto ad altre quando si utilizza in un contesto a bassa precisione. Il rateo di espansione, ovvero quel rateo che sta ad indicare il rapporto tra la dimensione del layer di input/bottleneck e quello interno, viene utilizzato pari a sei nella maggior parte degli esperimenti descritti dagli autori della rete. Se ad esempio l'input presenta 64 canali in input e 128 in output, il numero di canali con cui lavora l'expansion layer è 384.

Nella tabella 10 viene descritta l'intera struttura della rete: *n* rappresenta il numero di volte che quel determinato layer è presente in sequenza, *c* rappresenta il numero di canali di output di quel determinato layer (layer ripetuti hanno lo stesso numero di canali di output), *t* rappresenta il rateo di espansione utilizzato in quel layer e *s* rappresenta lo stride del primo layer della sequenza (mentre tutti gli altri appartenenti a questa hanno stride 1).

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Tabella 10. Struttura di MobileNet V2

Per effettuare della object detection è poi necessario arricchire questa architettura con una versione modificata di Single Shot Detector (SSD). La variante che qui viene introdotta prende il nome di SSDLite, e risulta essere molto più mobile-friendly rispetto alla sua versione originale per quanto riguarda il conto dei parametri (tabella 11) e il costo computazionale. L'idea dietro a questa nuova versione è la stessa presente alla base di MobileNet nella sua versione originale: sostituire la convoluzione classica con una depthwise separable. Come viene riportato da [25], il primo strato di SSDLite viene collegato al layer di espansione di MobileNet numero 15, mentre il secondo e il resto della rete sono collegati all'ultimo layer di MobileNet. Nelle tabelle sotto riportate possiamo vedere una comparazione tra le due versioni di SSD in termini di dimensione e di numero operazioni, oltre ad un confronto tra le varie reti in termini di precisione e di tempi di latenza (tabella 12).

	Params	MAdds
SSD	14.8M	1.25B
SSDLite	**2.1M**	**0.35B**

Tabella 11. Comparazione SSD vs SSDLite

Network	mAP	Params	MAdd	CPU
SSD300	23.2	36.1M	35.2B	-
SSD512	26.8	36.1M	99.5B	-
YOLOv2	21.6	50.7M	17.5B	-
MNet V1 + SSDLite	22.2	5.1M	1.3B	270ms
MNet V2 + SSDLite	22.1	**4.3M**	**0.8B**	200ms

Tabella 12. Comparazione delle prestazioni sul dataset COCO. I numeri riportati sono calcolati su un Google Pixel1

Nel caso in cui anche questo modello non dovesse rientrare nei parametri prestazionali richiesti dallo specifico caso, sia per la versione base che per la V2 utilizzata in questo progetto, è possibile andare ad intervenire su due parametri di design della rete: width multiplier e resolution multiplier.

Lo scopo del width multiplier α è quello di assottigliare la rete in maniera uniforme in ogni layer. Per uno specifico layer con uno specifico α settato, il numero di canali di input M diventa αM e il numero di canali di output N diventa αN, con α che può variare tra 0 e 1, dove 1 è il modello standard di MobileNet. L'effetto pratico (tabella 13) di questo parametro è di ridurre il costo computazionale della rete e il numero dei parametri considerati di un fattore α^2. Il nuovo modello così ottenuto però deve essere ri-allenato dall'inizio.

Il secondo iper-parametro ρ invece serve per controllare la risoluzione dell'immagine di input utilizzata dalla rete e, di conseguenza, tutte le rappresentazioni interne di questa. Questo parametro viene settato implicitamente quando si sceglie la risoluzione di ingresso della rete. Le risoluzioni classiche di questa rete sono 224, 192, 160 o 128 e l'effetto di questo parametro, come nel caso precedente, è quello di ridurre il costo computazionale della rete di un fattore ρ^2 (tabella 14).

L'utilizzo di questi due parametri ha lo scopo di rendere possibile per l'utente la selezione del trade-off accuratezza/latenza che più si addice al proprio contesto. Nelle tabelle riportate si possono vedere i differenti risultati dell'utilizzo di questi iper-parametri.

Width Multiplier	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
0.75 MobileNet-224	68.4%	325	2.6
0.5 MobileNet-224	63.7%	149	1.3
0.25 MobileNet-224	50.6%	41	0.5

Tabella 13. Esempi di width multiplier

Resolution	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
1.0 MobileNet-192	69.1%	418	4.2
1.0 MobileNet-160	67.2%	290	4.2
1.0 MobileNet-128	64.4%	186	4.2

Tabella 14. Esempi di resolution multiplier

Tutte le immagini (figura 7-13) e le tabelle (tabella 10-14) sopra riportate provengono dal paper di MobileNet V2 [25].

Voglio fare una piccola parentesi per quanto riguarda la scelta di questa architettura per il mio lavoro. Al momento della mia ricerca su quale fosse la migliore architettura da utilizzare per questo progetto mi sono imbattuto nella versione 3 di MobileNet [26] che, stando a quanto scritto dal suo paper, dovrebbe essere generalmente migliore della versione qui utilizzata, sia dal punto di vista delle capacità di riconoscimento che dal punto di vista della latenza. Detto questo, ho comunque utilizzato la versione 2 in quanto per la 3, versione molto più recente, al momento dell'allenamento della rete non erano disponibili dei modelli già allenati sui principali dataset generici, e quindi non era possibile applicare la tecnica del transfer learning, decisiva per ottenere ottimi risultati di riconoscimento. Dovessi ricominciare ora il mio lavoro, MobileNet V3 sarebbe decisamente la mia scelta.

2 - Configurazione del sistema

Prima di parlare di come sia stata allenata effettivamente la rete, è necessario descrivere le tecnologie e le risorse usate.

Per questo progetto ho usato la mia macchina personale con sistema operativo Windows 10, equipaggiata con: un processore AMD Ryzen 2700X, una scheda grafica NVIDIA GeForce RTX 2070, 32 GB di RAM DDR4 2933MHz e un SSD Nvme Samsung 970 Evo Pro. Tutti i tempi e i riferimenti di questa parte sono relativi a questa configurazione.

2.1 - Installazione di CUDA

Il primo step fondamentale se si vuole andare ad allenare una rete neurale, anche se solo per fare del fine tuning di questa, è l'installazione del toolkit CUDA [27] e di tutto ciò che lo circonda. Questo è essenziale se si vuole sfruttare la grande potenza computazionale che può esercitare una GPU dato che, di default, tutto l'allenamento verrebbe svolto sulla CPU, nettamente più lenta (in media un 20x).
Nonostante questa operazione sia ben descritta in documentazione e risulti quasi banale in teoria, la compatibilità tra le varie componenti è una cosa che si scopre solo una volta installato il tutto, e non è mai un processo lineare, come potrebbe dire chiunque abbia affrontato questo percorso.
Prima di installare il toolkit è necessario fare in modo che i driver della scheda grafica NVIDIA siano aggiornati oltre una data versione, soglia minima che dipende dalla versione del toolkit stesso che si vuole utilizzare. Allo stesso modo, perché questo possa essere installato correttamente, è necessario che sul sistema sia presente Visual Studio, anche in una sua versione Community. Questa parte nello specifico è stata abbastanza complicata sotto l'aspetto della compatibilità anche se, in teoria, perché l'intero procedimento vada a buon fine è sufficiente installare una qualunque versione di Visual Studio dalla 2015 in avanti. Non so quale fosse la causa che nel mio caso portava al fallimento dell'intera operazione fatto sta che, per fare in modo che questo processo di installazione andasse a buon fine, ho dovuto fare diversi tentativi con tre differenti versioni di Visual Studio in quanto l'intero procedimento falliva non trovando dei file che in realtà erano presenti e linkati correttamente nel Path della mia macchina. Dopo svariate prove con le diverse versioni di Visual Studio, quella che per me ha fatto terminare correttamente l'installazione di CUDA è stata quella Express del 2017.
Una volta installato correttamente questo toolkit è necessario installare anche cuDNN, una libreria di Deep Neural Network che lavora insieme a CUDA. Questa operazione è risultata

essere abbastanza intuitiva e, per vedere che il tutto funzioni correttamente, cosa che data la mia esperienza precedente non è scontata, basta provare un semplice comando di TensorFlow e vedere come a riga di comando vengano individuate, prima dell'esecuzione del comando stesso, le GPU disponibili nel sistema (nel mio caso solamente una).
Il codice sotto riportato può essere utilizzato per fare un check sulla corretta installazione di CUDA, da eseguire ovviamente dopo aver installato TensorFlow. Se l'installazione è terminata correttamente, un output simile a figura 14 e 15 dovrebbe apparire a terminale, descrivendo i device disponibili sul sistema.

from tensorflow.python.client import device_lib
print(device_lib.list_local_devices())

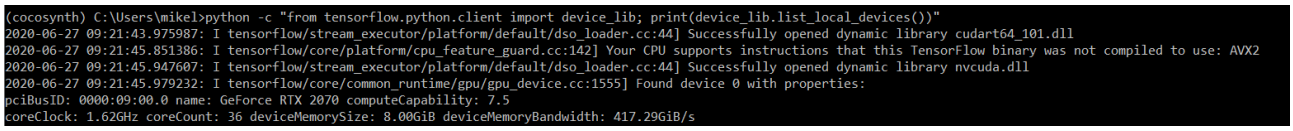

Figura 14. Parte dell'output del comando che compare sempre quando si lavora con TensorFlow

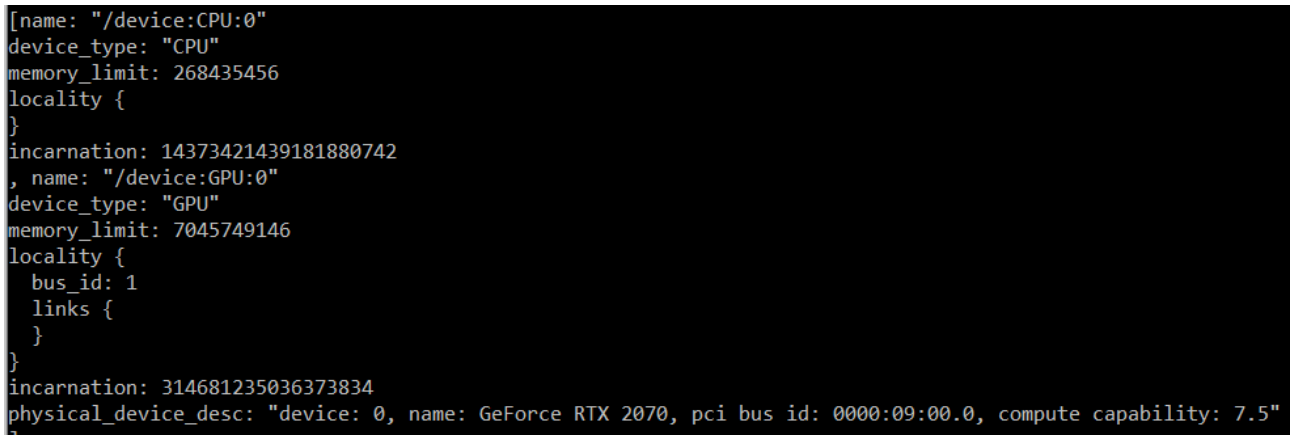

Figura 15. Parte dell'output che mostra i device disponibili sul sistema

Per decidere quale versione del toolkit andare ad installare è fondamentale consultare il sito di TensorFlow, in quanto solitamente l'ultima versione del toolkit non è supportata da nessuna versione di questo framework (figura 16). Nel mio caso ho installato la 10.1, in quanto la più recente che era possibile utilizzare con le versioni di TensorFlow impiegate nel progetto.

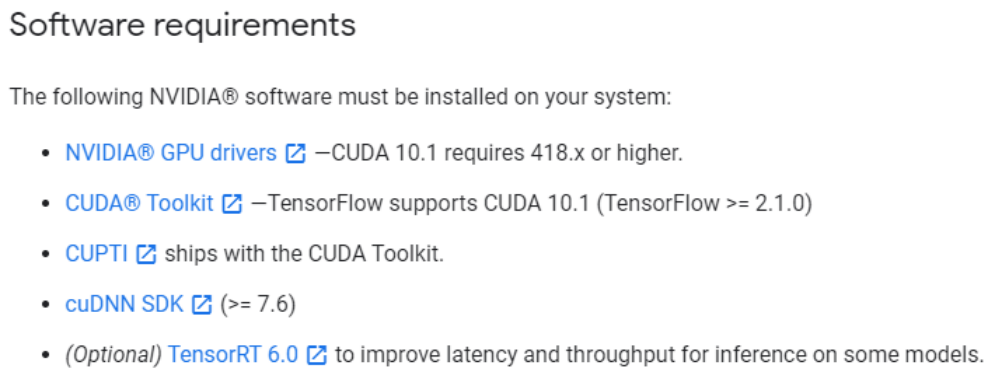

Figura 16. Lista dei requisiti per TensorFlow [28]

2.2 - Python

Un ragionamento simile è necessario quando si pensa a quale versione di Python utilizzare. In questo caso, sia per quanto riguarda la mia esperienza personale sia per quello che dicono in rete, la compatibilità è un po' più flessibile ma, per evitare problemi, ho sempre utilizzato la 3.6.8, anche se non ho mai riscontrato problemi con la versione 3.7 [29] utilizzata per errore durante il progetto.

2.3 - Protobuf

Dopo aver installato Python è necessario anche fare in modo che sul sistema sia presente una versione di Protobuf e tipicamente in questo caso la più recente va bene, dato che né io né nessuna delle fonti che ho consultato ha evidenziato dei problemi di compatibilità; la versione da me utilizzata in questo caso è la 3.11.2. Protobuf sta per Protocol Buffers ed è il meccanismo estendibile, indipendente sia dalla piattaforma che dal linguaggio, che permette di serializzare i dati strutturati. Le API di TensorFlow utilizzate per fare Object Detection sfruttano Protobuf per configurare il modello e allenare i parametri della rete.

Durante tutto questo progetto il lavoro è stato effettuato all'interno di un ambiente virtuale realizzato tramite Anaconda, suite ideale per la gestione di tutti i possibili conflitti tra le varie versioni di TensorFlow e Python. In verità ho dovuto utilizzare due environment distinti che, seppur praticamente identici, differivano solamente per la versione di TensorFlow utilizzata: su quello principale è installata la 2.1.0 mentre sul secondario è presente la versione 1.15. Questo si è reso necessario perché alcune parti del processo non erano compatibili con la versione 2.1.0 come, ad esempio, la gestione dei file in formato TFRecord.

3 - Parte sperimentale

3.1 - Generazione del dataset

Come detto in precedenza, per questo progetto ho utilizzato un dataset sintetico. Nonostante il principale vantaggio di un dataset sintetico sia quello di avere un numero praticamente illimitato di elementi, per questo lavoro, dato anche il numero non troppo elevato di elementi principali differenti, ho scelto di non esagerare con le dimensioni del dataset. La scelta è stata quindi quella di generare 5000 immagini per fare training e 1000 per fare la validazione delle performance. La dimensione delle immagini in entrambi i casi è di 800 x 800, scelta fatta empiricamente dopo aver valutato diverse misure. Infatti, in questo caso volevo che la dimensione dei codici a barre, in relazione al campo di inquadramento, fosse il più simile il possibile rispetto a quella che si ha nella realtà utilizzando una fotocamera. Andando invece a selezionare dimensioni più grandi per l'immagine i codici risultavano essere troppo piccoli (per ovviare a questo avrei dovuto ricominciare la procedura per la selezione delle immagini di foreground da capo senza avere un reale vantaggio); viceversa se le misure dell'immagine venivano ridotte, le dimensioni degli elementi principali diventavano talmente preponderanti da coprire la quasi totalità dell'immagine e, inoltre, davano luogo a molte sovrapposizioni, comportamento non normale e non desiderato. Per questo motivo 800 x 800 si è rivelato essere la scelta ideale per la dimensione del dataset.

```
python ./python/image_composition.py --input_dir C:/Users/mikel/Desktop/cocosynth-master/datasets/barcode_synthetic/input
--output_dir C:/Users/mikel/Desktop/t_job/models/research/object_detection/images/train --count 5000 --width 800 --height 800

python ./python/coco_json_utils.py -md C:/Users/mikel/Desktop/t_job/models/research/object_detection/images/train/
mask_definitions.json -di C:/Users/mikel/Desktop/t_job/models/research/object_detection/images/train/dataset_info.json

python ./python/image_composition.py --input_dir C:/Users/mikel/Desktop/cocosynth-master/datasets/barcode_synthetic/input
--output_dir C:/Users/mikel/Desktop/t_job/models/research/object_detection/images/val --count 1000 --width 800 --height 800

python ./python/coco_json_utils.py -md C:/Users/mikel/Desktop/t_job/models/research/object_detection/images/val/
mask_definitions.json -di C:/Users/mikel/Desktop/t_job/models/research/object_detection/images/val/dataset_info.json
```

Figura 17. Comandi per la generazione delle immagini del dataset e delle descrizioni

Sopra (figura 17) sono riportati i comandi usati per la generazione delle immagini e delle annotazioni COCO-like, sia per quanto riguarda la parte di training del dataset sia per quella di validation. Questa operazione ha richiesto all'incirca quattro ore sul mio sistema, e ho notato che questo tempo non è influenzato da tutti i parametri che si usano nell'algoritmo alla stessa maniera: se infatti si aumenta il numero di elementi di foreground o la dimensione delle immagini finali, a parità di numero di immagini generate, il tempo aumenta considerevolmente, mentre l'inclusione o l'esclusione di una particolare feature non ha praticamente impatto.

Essendo poi questo un dataset generato automaticamente, non è stato necessario andare ad annotare a mano ogni singola immagine, cosa che risulta essere la più costosa e time-consuming quando si parla di generazione di dataset tradizionali. Le annotazioni generate dall'algoritmo utilizzato sono in stile COCO, uno dei più famosi dataset per la object detection.

Un particolare che ci tengo a sottolineare è che con questo algoritmo per la generazione di immagini è possibile costruire una gerarchia superclasse-classe per gli elementi di foreground, proprio come avviene nella versione ufficiale di COCO. Infatti, è sufficiente che al momento della generazione delle immagini di foreground queste vengano salvate in una gerarchia di cartelle con nomi significativi (in quanto questi andranno poi a rappresentare la superclasse) che rispecchino la tassonomia voluta dalla famiglia di immagini, e sarà poi l'algoritmo ad occuparsi di trasporre questa organizzazione nel dataset (figura 18).

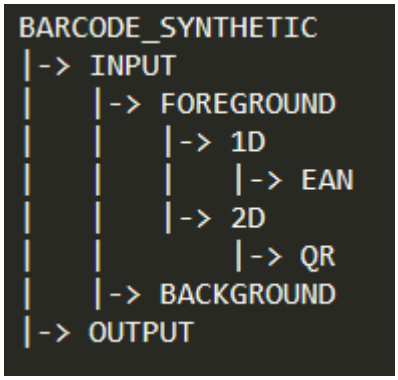

Figura 18. Tassonomia delle cartelle contenenti le immagini base del dataset

Per allenare la rete selezionata però questo formato non va bene. MobileNet, così come ogni altro modello che viene allenato utilizzando le API di TensorFlow per la Object Detection, necessita di un formato particolare, ovvero il TFRecord. Questo formato di file è necessario in quanto permette di leggere efficientemente il dataset durante la fase di allenamento poiché racchiude tutte le informazioni del dataset serializzate e in formato binario (nel repository di TensoFlow è presente un notebook che permette di visualizzare che cosa rappresentano questi file). TFRecord e Protobuf lavorano a stretto contatto per garantire efficienza all'intero processo di allenamento e gestione della rete neurale.

Per generare questi tipi di file è possibile utilizzare due diversi metodi, a seconda di come si sono generate le annotazioni e in generale il dataset. Se il formato di annotazione delle immagini è abbastanza standard, ad esempio CSV, o segue quello di uno dei grandi dataset, come può essere ad esempio COCO o Pascal, allora TensorFlow, all'interno del suo repository GitHub, offre degli script per questa traduzione. Se invece le annotazioni sono state salvate in un formato

custom dell'utente, è necessario andare a scrivere questo script di traduzione seguendo le linee guida di [30]. Nel mio caso il formato delle annotazioni era uguale a quello di COCO, e ho quindi potuto usare direttamente uno script offerto dal gruppo di TensorFlow. In figura 19 è riportato il comando per la generazione dei TFRecord relativi alle varie sotto parti del dataset.

```
python create_coco_tf_record.py --logtostderr --train_image_dir C:/Users/mikel/Desktop/t_job/models/research/
object_detection/images/train/images --val_image_dir C:/Users/mikel/Desktop/t_job/models/research/object_detection/images/
val/images --test_image_dir C:/Users/mikel/Desktop/t_job/models/research/object_detection/images/test/images
--train_annotations_file C:/Users/mikel/Desktop/t_job/models/research/object_detection/images/train/coco_instances.json
--val_annotations_file C:/Users/mikel/Desktop/t_job/models/research/object_detection/images/val/coco_instances.json
--testdev_annotations_file C:/Users/mikel/Desktop/t_job/models/research/object_detection/images/test/coco_instances.json
--output_dir C:/Users/mikel/Desktop/t_job/models/research/object_detection/TFRecords/temp
```

Figura 19. Comando per la generazione dei file TFRecord a partire dal dataset generato dall'algoritmo

Prima però di fare questa operazione sono anche andato a modificare il file labelmap.pbtxt, file dove sono contenute le etichette utilizzate per descrivere il dataset. Non è necessario farlo prima di generare i file in formato TFRecord, ma è utile per fare un controllo sulla coerenza di queste prima di proseguire. Originariamente questo file era settato come quello del dataset di COCO, con tutte e 90 le etichette. Nel nostro caso, essendo queste solamente due, ho eliminato l'intero contenuto e inserito quelle dei due codici a barre come mostrato in figura 20, facendo attenzione al fatto che rispettassero la struttura data in precedenza alle immagini.

```
item{
    id:1
    name:'QR'
}
item{
    id:2
    name:'EAN'
}
```

Figura 20. Contenuto del file labelmap.pbtxt

In questo caso non si nota la struttura sottesa al dataset, in quanto lo scopo è solo quello di fare un mapping uno a uno univoco tra un ID ed una determinata classe, ma la struttura del dataset, descritta in precedenza, rimane.

Quando si modifica questo file è fondamentale che i nomi utilizzati per descrivere le classi siano identici a quelli usati nel dataset, e di conseguenza nelle cartelle, in quanto anche solo maiuscole o minuscole o l'inversione dell'ID porta il sistema a fare confusione e, a seconda del caso, ad avere dei pessimi risultati o a non riuscire a proseguire con l'allenamento. Infatti, quando si va a creare il dataset in formato COCO, questi ID vengono già assegnati alle immagini, come si può vedere in figura 21.

```
"categories": [
    {"supercategory": "2D", "id": 1, "name": "QR"},
    {"supercategory": "1D", "id": 2, "name": "EAN"}
]
```


Figura 21. Categorie COCO utilizzate nel progetto

Quella riportata sopra è la parte conclusiva della descrizione del dataset in formato COCO ed è la trasposizione gerarchica dell'organizzazione descritta in precedenza (figura 18). In figura 22 invece ho messo in evidenza come viene rappresentato un oggetto nel formato COCO.


```
{
    "segmentation": [[]],
    "iscrowd": 0,
    "image_id": 190,
    "category_id": 2,
    "id": 368,
    "bbox": [66.5, 227.5, 249.0, 295.0],
    "area": 49309.0
}
```


Figura 22. Descrizione di un oggetto di interesse, in questo caso un codice EAN, contenuto dell'immagine #190 del dataset. Da questa immagine sono stati rimossi i dati di segmentazione in quanto non rilevanti, ma nella descrizione sono presenti.

3.2 - Fine tuning del modello

A questo punto siamo davvero pronti per allenare la rete. Questa è stata già implementata dal gruppo di TensorFlow, quindi non è necessario andare a scrivere a mano nulla della sua struttura. Oltre a questo, Google fornisce già anche dei checkpoint di tutte le sue reti, che sono state precedentemente allenate su diversi dataset. In questo caso il punto di partenza per sfruttare il transfer learning è stato il checkpoint di MobileNet V2 allenata sul dataset COCO. Qui [31] è possibile trovare tutti i modelli che Google ha già allenato per noi. Una volta scaricato ed estratto questo file, l'ho inserito nella cartella di lavoro, ovvero un clone del repository GitHub di TensorFlow, con però tutte le modifiche e le aggiunte descritte fino a qui.

Per procedere ora è necessario andare a prendere il file di configurazione della rete, presente sempre all'interno di questa repository, e copiarlo nella cartella di lavoro, in modo da poterlo opportunamente modificare senza corrompere il file originario. Questo file di configurazione, che si può trovare in *models\research\object_detection\samples\configs* insieme a molti altri file di configurazione di altre reti, rappresenta in sostanza la rete stessa: in questo sono infatti presenti le descrizioni di tutti i layer utilizzati dalla rete, con tutti i parametri esposti e potenzialmente configurabili. Di default questi sono impostati come definiti dal paper [25] ma in questo file c'è la possibilità di vedere anche quei parametri che rimangono "nascosti", come ad esempio il rateo delle ancore di proposta di box predittive.
Ci tengo ad evidenziare il fatto che, al momento della stesura della tesi, mi sono accorto che nei modelli preallenati era presente una versione della rete che sfruttava SSDLite invece della normale SSD, con i vantaggi sui tempi di risposta a parità di livello di riconoscimento di cui ho parlato in precedenza; se avessi usato quella plausibilmente il risultato finale sarebbe stato migliore in termini di latenza.

Per fare in modo che il fine tuning della rete vada a buon fine (ricordo che in questo progetto è stata utilizzata la tecnica del transfer learning cosa che viene generalmente caldeggiata in letteratura rispetto ad un allenamento che parte da zero) è necessario andare ad apportare le seguenti modifiche al nostro file di configurazione:

- Prima di tutto è necessario andare a modificare il numero delle classi che si vogliono individuare grazie a questa rete neurale. Per fare questo è necessario modificare il parametro *num_classes*, di default settato a 90, in modo che sia uguale a 2;
- In secondo luogo, dato che stiamo facendo del fine tuning, è necessario impostare il valore di *fine_tune_checkpoint* in modo tale che questo punti all'indirizzo in cui è

possibile trovare il modello già allenato. Questo sarà uno dei file all'interno della cartella compressa che abbiamo scaricato da [31], in particolare quello chiamato *model.ckpt* con formato di file *data-00000-of-00001*. Nel caso in cui invece non si volesse sfruttare un checkpoint per fare l'allenamento della rete è sufficiente lasciare il parametro a default, dato che questo non punta a nulla;

- È poi necessario modificare alcuni valori, sia nella sezione *train_input_reader* sia in quella *eval_input_reader*, in maniera tale che *input_path* di *tf_record_input_reader* punti alla posizione corretta in cui sono stati salvati i file TFRecord che rappresentano rispettivamente la parte di training e di evaluation del dataset. In queste sezioni è anche necessario andare ad indicare la posizione del file *labelmap.pbtxt* generato in precedenza, dove sono elencate le classi del problema;
- Infine, nella sezione *eval_config*, bisogna andare ad inserire il numero di esempi che si utilizzano per fare la fase di valutazione, in questo caso 1000.

In questo file di configurazione si può anche andare a specificare il numero di steps di allenamento per la rete. Nonostante in rete svariati esempi suggeriscano che non sono necessari molti step per avere un riconoscitore efficace dato che si tratta di fine tuning (alcune fonti parlavamo anche solamente di mille steps, numero che francamente mi sembra un po' basso), ho deciso di lasciare il parametro al valore di default, ovvero 200 mila. Questo valore è suggerito dal team di TensorFlow in maniera empirica, in quanto dicono che con questo valore sono riusciti ad allenare la rete sul dataset degli animali domestici. Lasciando questo e gli altri valori non citati sopra a default però non si avrà mai una modifica nel valore del learning rate della nostra rete. Se avessi dovuto allenare questo modello su un problema dotato di più classi plausibilmente avrei rimosso questo limite a priori per osservare solamente la curva di apprendimento, fino ad arrivare ad una stabilizzazione.

Una volta modificato in questa maniera il file di configurazione, eseguendo il comando in figura 23 è possibile avviare il processo di allenamento della rete.

```
python legacy/train.py --logtostderr --train_dir=training/ --pipeline_config_path=training/ssd_mobilenet_v2_coco.config
```

Figura 23. Comando utilizzato per allenare la rete

L'intero processo di allenamento della rete neurale è durato all'incirca 22 ore sul mio sistema, fermandosi intorno al 192 millesimo step, in quanto in un'altra console ho eseguito un comando che veniva dichiarato compatibile con l'esecuzione dell'allenamento ma che evidentemente non lo era.

Durante questo processo è stato possibile osservare l'avanzamento dello stato di allenamento della rete utilizzando TensorBoard. Questo è un tool che è in grado di fornire tutte le misurazioni e le visualizzazioni necessarie per monitorare l'evolvere della rete man mano che l'allenamento prosegue. In particolare, le seguenti figure (figura 24-27) mostrano questa evoluzione sotto diversi punti di vista.

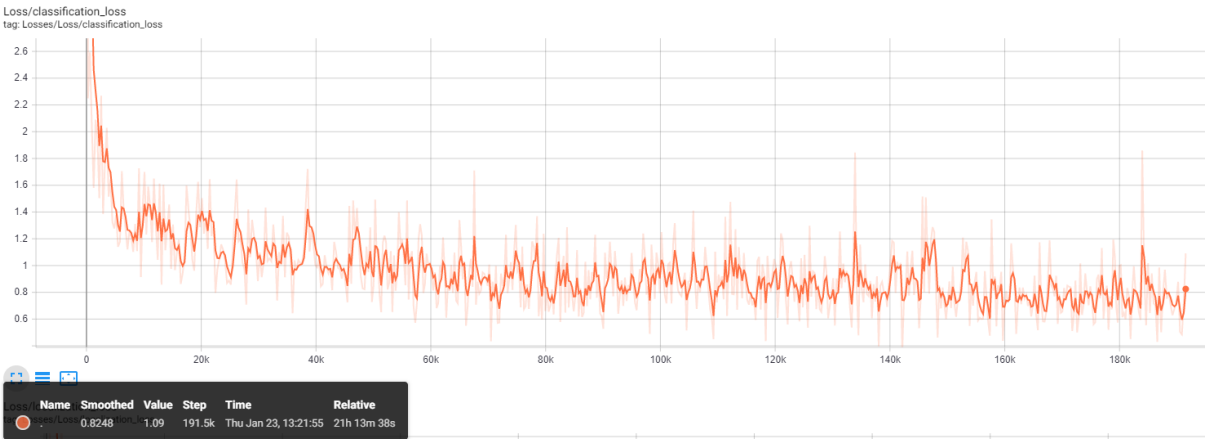

Figura 24. Evoluzione della classification loss sull'intera durata dell'allenamento

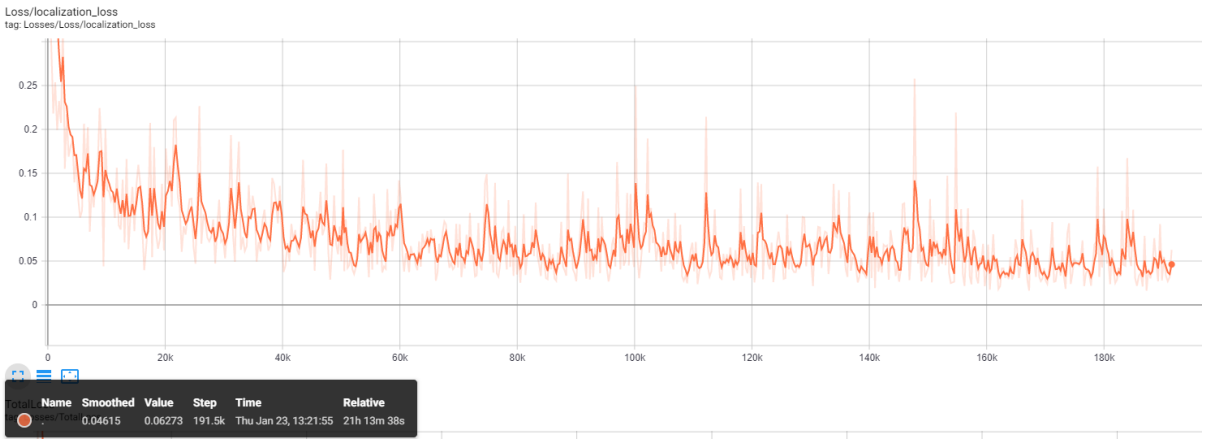

Figura 25. Evoluzione della localization loss sull'intera durata dell'allenamento

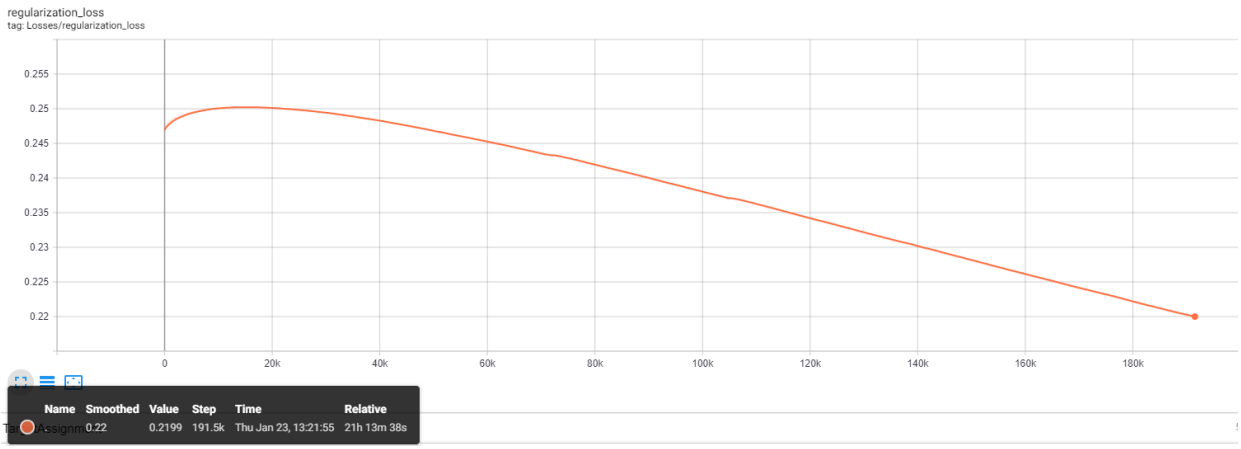

Figura 26. Evoluzione della regularization loss sull'intera durata dell'allenamento

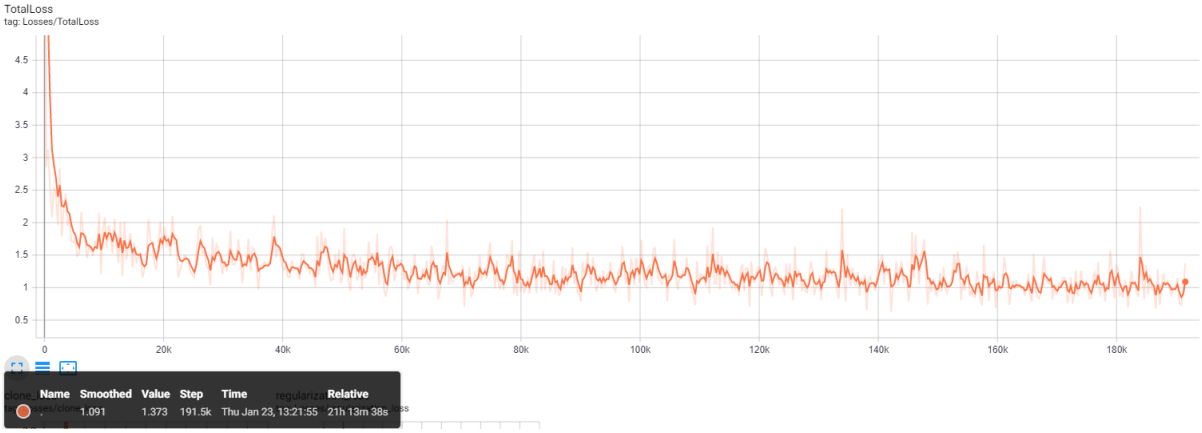

Figura 27. Evoluzione della total loss sull'intera durata dell'allenamento

Stando a quanto dice la documentazione, è possibile monitorare il processo di allenamento anche sfruttando il comando riportato in figura 28, in alternativa o insieme a TensorBoard ma, come dicevo prima, quando ho avviato questo comando l'intero processo di allenamento della rete si è interrotto.

```
python legacy/eval.py --logtostderr --pipeline_config_path=training/ssd_mobilenet_v2_coco.config --checkpoint_dir=training --eval_dir=eval
```

Figura 28. Comando per fare una eval sull'ultimo checkpoint salvato dal processo di allenamento

```
I0123 13:42:42.287973 11692 eval_util.py:371] # success: 1000
INFO:tensorflow:# skipped: 0
I0123 13:42:42.287973 11692 eval_util.py:372] # skipped: 0
I0123 13:42:42.449028 11692 object_detection_evaluation.py:1311] average_precision: 0.989771
I0123 13:42:42.603080 11692 object_detection_evaluation.py:1311] average_precision: 0.991102
INFO:tensorflow:Writing metrics to tf summary.
I0123 13:42:42.833158 11692 eval_util.py:80] Writing metrics to tf summary.
INFO:tensorflow:Losses/Loss/classification_loss: 0.738531
I0123 13:42:42.834158 11692 eval_util.py:87] Losses/Loss/classification_loss: 0.738531
INFO:tensorflow:Losses/Loss/localization_loss: 0.069007
I0123 13:42:42.835159 11692 eval_util.py:87] Losses/Loss/localization_loss: 0.069007
INFO:tensorflow:PascalBoxes_PerformanceByCategory/AP@0.5IOU/EAN: 0.991102
I0123 13:42:42.835159 11692 eval_util.py:87] PascalBoxes_PerformanceByCategory/AP@0.5IOU/EAN: 0.991102
INFO:tensorflow:PascalBoxes_PerformanceByCategory/AP@0.5IOU/QR: 0.989771
I0123 13:42:42.835159 11692 eval_util.py:87] PascalBoxes_PerformanceByCategory/AP@0.5IOU/QR: 0.989771
INFO:tensorflow:PascalBoxes_Precision/mAP@0.5IOU: 0.990436
I0123 13:42:42.835159 11692 eval_util.py:87] PascalBoxes_Precision/mAP@0.5IOU: 0.990436
INFO:tensorflow:Metrics written to tf summary.
I0123 13:42:42.835159 11692 eval_util.py:88] Metrics written to tf summary.
```

Figura 29. Risultato del comando di figura 28

Fortunatamente però, durante tutto il processo di allenamento, ogni circa 2000 step, TensorFlow salva un checkpoint temporaneo in modo che, in caso di problemi, il lavoro effettuato fino a quel momento non vada perso. A mano a mano che questi si accumulano i più vecchi vengono cancellati, essendo questi obsoleti, mantenendo alla fine solamente i cinque più recenti oltre a quello finale.

3.3 - Testing della rete

Avendo appena finito di allenare la rete neurale sul nostro dataset, lo step successivo è quello di trasferire la rete sul dispositivo mobile in modo da poterne sfruttare le capacità inferenziali. Per fare questo però non possiamo utilizzare nessuno dei file che sono stati generati fino ad ora, ma è necessario andare a prendere il checkpoint finale della fase di allenamento e utilizzarlo in maniera tale da generare un file in formato TensorFlow Lite. Il punto di partenza di questa operazione sarà il grafo di inferenza/checkpoint dell'ultimo momento di allenamento della rete, che teoricamente dovrebbe essere quello che porta con sé la maggior conoscenza del contesto di applicazione e le migliori percentuali di riconoscimento.

Prima di procedere oltre però ho voluto accertarmi che la rete si comportasse a dovere e che individuasse effettivamente i codici a barre se presenti nell'immagine. Per fare questo ho sfruttato un notebook già implementato dal gruppo di TensorFlow che si può trovare nel loro repository GitHub (*models/research/object_detection/object_detection_tutorial.ipynb*). Per far funzionare questo notebook, oltre ad andare a inserire i path corretti per le varie risorse richieste, è necessario modificare la variabile *model_dir* in modo che punti al path dove è possibile trovare il modello salvato. Per ottenere questo è necessario prima di tutto utilizzare il comando mostrato in figura 30.

```
python export_inference_graph.py --input_type image_tensor --pipeline_config_path training/ssd_mobilenet_v2_coco.config
--trained_checkpoint_prefix  training/model.ckpt-190865 --output_directory fine_tuned_model
```

Figura 30. Comando per esportare il grafo di inferenza

Una volta fatto ciò, è necessario che il modello venga caricato utilizzando questo comando/formato: *model = tf.compat.v2.saved_model.load(str(model_dir))*. Infatti, lasciando il notebook come era al momento della clonazione del repository, questo file non funzionava per un duplice motivo. Prima di tutto c'era un'incompatibilità con la versione di TensorFlow utilizzata fino ad ora: mentre per eseguire l'allenamento e il comando sopra ho utilizzato la versione 1.15, questo notebook richiede invece la versione 2.* per riuscire ad eseguire dei comandi di traduzione, e per farla aggiornare è necessario riavviare il kernel del notebook, dato che installare semplicemente una versione diversa all'inizio del notebook stesso non è sufficiente a renderla quella in uso. In secondo luogo, è risultato necessario modificare la riga di codice scritta sopra, che serve a caricare il modello, in quanto, a seconda di come si voglia vedere il problema, o non era compatibile con la versione 2 di TensorFlow oppure con il modo

in cui andavo ad indicare il modello, dato che in teoria questo notebook è pensato per scaricare un modello da [31]. Una volta fatte queste modifiche il tutto funziona.

Quello che stiamo controllando qui sono le abilità di riconoscimento della rete e non le sue prestazioni in termini di latenza, in quanto questo test è effettuato sulla mia macchina. Di seguito riporto tre immagini: le prime due (figura 31 e 32) sono state scattate da posizione più ravvicinata con il mio OnePlus 5, dotato di una doppia fotocamera, e risultano avere una risoluzione maggiore (1379 x 2031) e una dimensione doppia rispetto alla terza (1060 x 1885), scattata da un prototipo di Memor20 e da una distanza maggiore (figura 33). Questi fattori messi insieme sono a mio avviso sufficienti per giustificare la differenza dei risultati in termini di riconoscimento, dato che il soggetto è il medesimo.

Figura 31. Scattata con OnePlus 5

Figura 32. Scattate con OnePlus 5

Figura 33. Scattata con Memor 20

Oltre ad aver utilizzato queste immagini reali per testare il risultato dell'allenamento della rete, sono anche ricorso ad un insieme di cento immagini sintetiche, che sono andato a generare utilizzando lo stesso algoritmo utilizzato per la creazione del dataset, come sorta di test set. I

risultati sono stati molto confortanti (figura 34 e 35), in quanto la rete riusciva a identificare la quasi totalità dei codici con soglie di sicurezza sulla classe dell'oggetto molto elevate. Ricordo che queste immagini hanno una risoluzione di 800 x 800 ma la loro dimensione è nettamente maggiore rispetto a quella delle foto riportate sopra, quasi il doppio rispetto alle immagini scattate con il mio OnePlus 5.

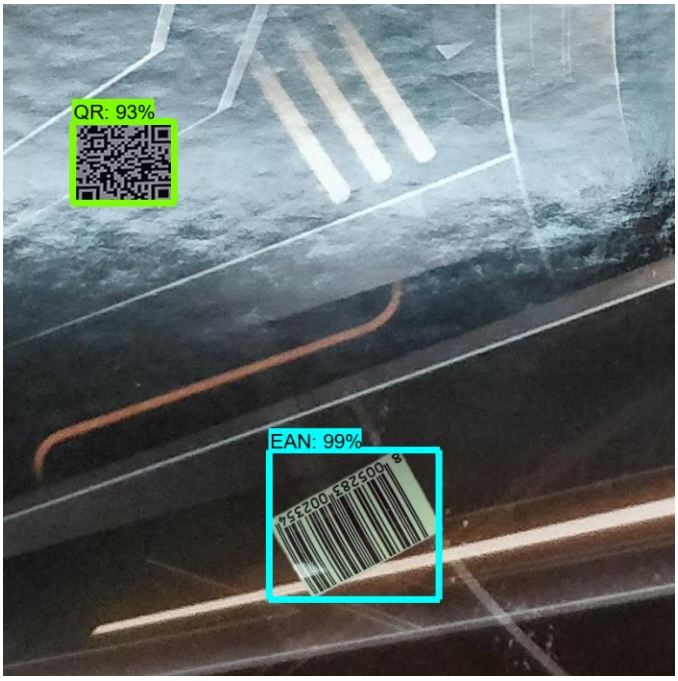

Figura 34. Immagine di test generata sinteticamente

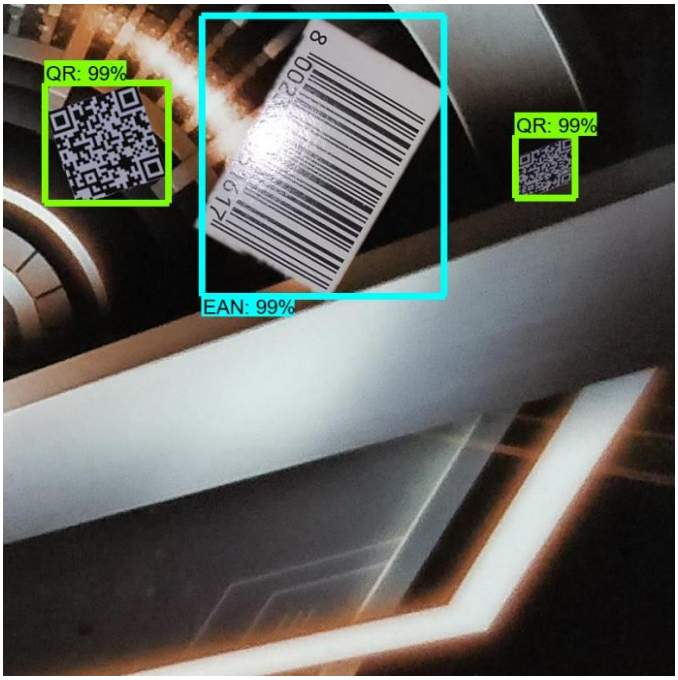

Figura 35. Immagine di test generata sinteticamente

Una volta appurato che il tutto funzionava correttamente e con risultati più che soddisfacenti, ho deciso di proseguire con il progetto e di non sottoporre la rete ad un ulteriore fase di allenamento.

3.4 - Passaggio a TensorFlow Lite

Il prossimo passo ora è quello di trasformare il checkpoint della rete in un qualcosa che possa essere utilizzato su un dispositivo mobile. Questo processo viene scomposto in due passi, di cui la figura 36 rappresenta il primo.

```
python export_tflite_ssd_graph.py --pipeline_config_path=C:/Users/mikel/Desktop/t_job/models/research/object_detection/fine_tuned_model/
pipeline.config --trained_checkpoint_prefix=C:/Users/mikel/Desktop/t_job/models/research/object_detection/fine_tuned_model/model.ckpt
--output_directory=C:/Users/mikel/Desktop/t_job/models/research/object_detection/PROVA_TFLITE --add_postprocessing_op=true
```

Figura 36. Comando utilizzato per ottenere il grafo TensorFlow Lite

Utilizzando questo script Python, che si può trovare in *models\research\object_detection* all'interno del repository di TensorFlow, andiamo a generare un frozen graph della rete. Anche se il comando è simile a quello usato precedentemente bisogna fare attenzione a non confonderli: il grafo generato qui è infatti totalmente diverso da quello generato in precedenza, in quanto questo può essere dato direttamente come input a TensorFlow Lite mentre l'altro no.

Il prossimo passo consiste nell'andare ad utilizzare TensorFlow Lite per ottenere, a partire dal file appena generato *tflite_graph*, il modello della rete ottimizzato per l'ambiente mobile, andando a sfruttare il tool TOCO (TensorFlow Lite Optimizing Converter). Il risultato finale di questa operazione sarà il file *detect.tflite*, in formato flatbuffer utilizzato da TensorFlow Lite per fare inferenza. Per avere un'idea più chiara del flusso di trasformazione ho riportato un diagramma di flusso per l'intera operazione (figura 37).

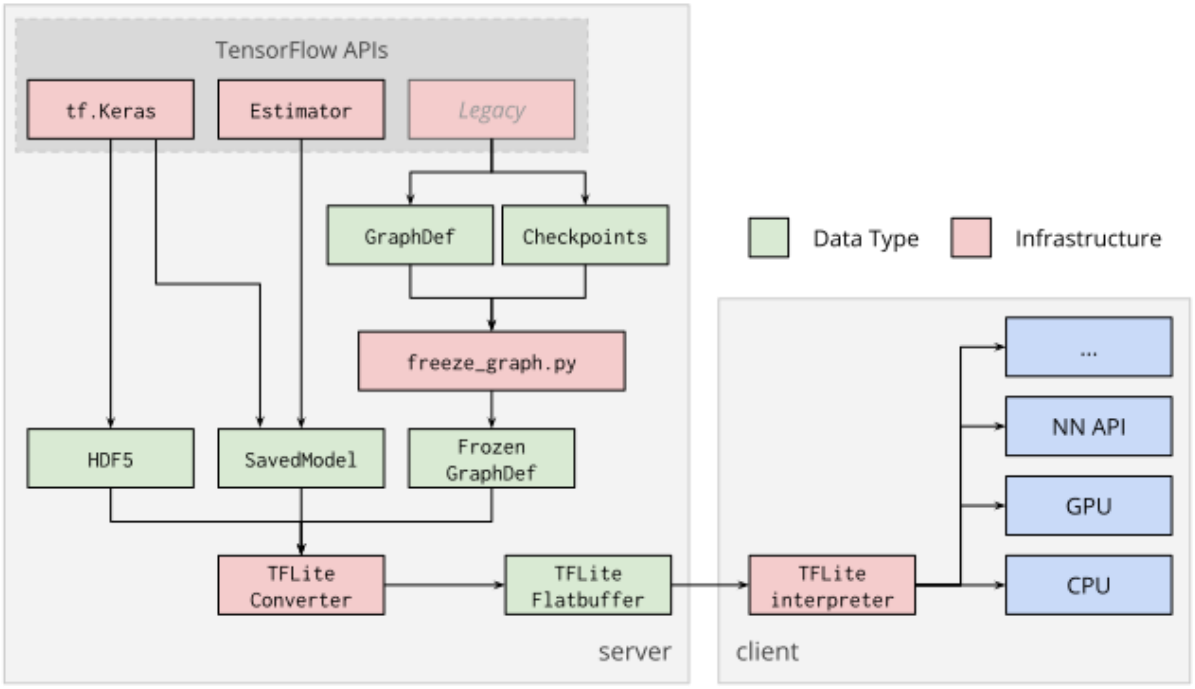

Figura 37. Flusso della traduzione [32]

Installazione di Bazel

Prima di poter fare il secondo step di questo passaggio è necessario andare ad installare Bazel, il principale sistema di costruzione per TensorFlow. Seguendo le istruzioni di [33], [34] e [35] sono riuscito ad installare correttamente Bazel ma, al momento di eseguire il comando necessario per la traduzione in formato TensorFlow Lite, ho avuto molti problemi di compatibilità tra questo e Visual Studio, cosa abbastanza risaputa quando si lavora su Windows. In particolare ricordo che, al momento dell'installazione di Bazel, sulla mia macchina erano installate quattro versioni differenti di Visual Studio, tutte linkate correttamente nel path di sistema. Non sto a riportare tutta la sequenza di tentativi ed errori in cui mi sono imbattuto per far funzionare il comando descritto in figura 38 ma riassumo il tutto dicendo che, quando questo falliva, compariva a terminale prima il messaggio *Bazel couldn't find a valid Visual C++ build tools installation on your machine* e poi *Visual C++ build tools seems to be installed*; oltre a ciò, veniva mostrato anche il percorso in cui questi build tools erano situati (rimandava ad una versione di Visual Studio), specificando che un insieme di file non erano presenti quando in realtà erano tutti presenti e i percorsi delle cartelle in cui questi si trovavano erano correttamente linkati. Andando per tentativi, ossia provando a cancellare e reinstallare svariate versioni di Visual Studio nelle quali spuntavo tutte le caselle che iniziavano per MFC e quelle comuni per tutti i tipi di piattaforma, alla fine la versione che ha fatto sì che il comando di traduzione riuscisse a terminare correttamente è stata quella del 2017 nella versione Community.

Infine, ci tengo a precisare che il comando finale utilizzato per la generazione del file *detect.tflite* che ha funzionato per me, riportato in figura 38, è leggermente diverso rispetto a quello mostrato sulla guida ufficiale fornita dal gruppo di TensorFlow per la trasformazione di un modello float [36]. Oltre all'aggiunta delle prime tre variabili che iniziano per BAZEL, essenziali dato che qui vengono impostati dei parametri di ambiente senza i quali il comando non funziona, ho dovuto rimuovere gli apici singoli dai nomi delle variabili di output. Infatti, utilizzando la versione del comando proposta dalla guida (con in più le variabili di ambiente), il file generato risultava essere vuoto: in questo caso il comando, seppur terminando correttamente, alla fine restituiva un warning riguardante un ipotetico typo nella definizione delle variabili di output, in quanto queste non facevano match con nessun elemento generato dal grafo selezionato. Questi quattro output rappresentano la conoscenza che ci ritorna dal processo di inferenza e rappresentano rispettivamente *detection_boxes*, *detection_classes*, *detection_scores*, e *num_detections.*

```
set BAZEL_VS=C:\Program Files (x86)\Microsoft Visual Studio\2017\
set BAZEL_VC=C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC
set BAZEL_VC_FULL_VERSION=14.16.27023
bazel run -c opt tensorflow/lite/toco:toco -- --input_file=C:/Users/mikel/Desktop/t_job/models/research/object_detection/
PROVA_TFLITE/tflite_graph.pb --output_file=C:/Users/mikel/Desktop/t_job/models/research/object_detection/
TFLITE_p2/detect.tflite --input_shapes=1,300,300,3 --input_arrays=normalized_input_image_tensor --output_arrays=TFLite_Detection
_PostProcess,TFLite_Detection_PostProcess:1,TFLite_Detection_PostProcess:2,TFLite_Detection_PostProcess:3
--inference_type=FLOAT --allow_custom_ops
```

Figura 38. Comando per ottenere il file detect.tflite

Tutti i comandi che ho descritto in questa sezione hanno funzionato con la versione 2.0.0 di TensorFlow. Non so onestamente se questo sia entrato in gioco in qualche maniera nel caso del comando riportato in figura 38, ma per quanto riguarda quello di figura 36, utilizzando la versione 2.1.0, si originavano degli errori legati plausibilmente alla compatibilità tra il comando e questa versione che era uscita da poco tempo.

3.5 - Applicazione finale

A questo punto non resta altro da fare se non trasportare la rete su un dispositivo Android e sfruttarla per fare inferenza: per fare ciò sarà sufficiente spostare nella cartella degli assets dell'applicazione il file tflite e la labelmap. L'applicazione all'interno della quale il file *detect.tflite* verrà inserito è un app di fotocamera, sviluppata seguendo l'approccio descritto da Camera X, con l'obiettivo di catturare il flusso di dati proveniente dalla camera in modo da localizzare in real time la posizione dei codici a barre presenti nell'inquadratura.

MainActivity

Come detto in precedenza, la versione di Camera X che ho utilizzato per sviluppare questa applicazione è la 5 della alfa. Ci tengo a precisare inoltre che questa applicazione è nata dall'unione di svariate parti di applicazioni di esempio e tutorial, riadattate ed elaborate in maniera tale che il tutto potesse funzionare a dovere facendo quanto richiesto.

All'avvio dell'applicazione di camera devono essere create in prima battuta le configurazioni dei casi d'uso che vogliamo utilizzare: preview (figura 39), ovvero quello che viene mostrato a schermo, e analisi (figura 40), luogo dove andiamo a inserire il nostro processo inferenziale. Le figure sotto riportate (figura 39-44) rappresentano le parti chiave della *MainActivity*.

```
val previewConfig = PreviewConfig.Builder().apply {  this: PreviewConfig.Builder
    setLensFacing(CameraX.LensFacing.BACK)
    if (textureView.display.rotation == 0) {
        setTargetAspectRatio(screenAspectRatio0)
    } else {
        setTargetAspectRatio(screenAspectRatio13)
    }
    setTargetRotation(textureView.display.rotation)
}.build()

preview = AutoFitPreviewBuilder.build(previewConfig, textureView)
```

Figura 39. Configurazione del caso d'uso preview

```
val analyzerConfig = ImageAnalysisConfig.Builder().apply { this: ImageAnalysisConfig.Builder
    setLensFacing(CameraX.LensFacing.BACK)
    setImageReaderMode(ImageAnalysis.ImageReaderMode.ACQUIRE_LATEST_IMAGE)
    //setTargetResolution(Size(metrics.widthPixels/2,metrics.heightPixels/2))
    if (textureView.display.rotation == 0) {
        setTargetAspectRatio(screenAspectRatio0)
    } else {
        setTargetAspectRatio(screenAspectRatio13)
    }
    val analysisThread = HandlerThread( name: "CustomThread").apply { this: HandlerThread
        start()
    }
    setCallbackHandler(Handler(analysisThread.looper))
}.build()

imageAnalyzer = ImageAnalysis(analyzerConfig)
```

Figura 40. Configurazione del caso d'uso analisi

In figura 40 si può anche notare come sia possibile selezionare l'immagine che deve essere analizzata in questo caso d'uso in un determinato momento: *ACQUIRE_LATEST_IMAGE* indica proprio che ad ogni ciclo l'immagine su cui si deve fare inferenza è quella più recente e non la successiva rispetto a quella appena analizzata. Come dicevo anche in un capitolo precedente, tutte le immagini tra l'ultima analizzata e quella più recente vengono scartate.

```
val customAnalyzer = CustomAnalyzer {r ->

    tracker.trackResults(r)
    trackingOverlay.postInvalidate()
    / graphicOverlay.clear() .../
}



imageAnalyzer!!.analyzer = customAnalyzer
classifier = Classifier.create(this)
```

Figura 41. Aggiunta di un gestore di quello che mi restituisce la fase di analisi

In figura 41 invece si può notare come il risultato dell'analisi (*r*), che racchiude tutti i 4 parametri della rete, venga utilizzato da una funzione in grado di mostrare a schermo le bounding box intorno ai codici a barre, le soglie di confidenza e la classe. Infine, le configurazioni vengono collegate al lifecycle dell'applicazione (figura 42).

```
CameraX.bindToLifecycle(this as LifecycleOwner, preview,imageAnalyzer)
```

Figura 42. Collegamento dei due casi d'uso al lifecycle dell'applicazione

```
class CustomAnalyzer(
    private val r : (res: List<Classifier.Recognition>) -> Unit
) : ImageAnalysis.Analyzer {

    private var lastAnalyzedTimestamp = 0L
    private var decoded = 0L
    private var cont = -1L

    override fun analyze(image: ImageProxy?, rotationDegrees: Int) {
```

Figura 43. Custom analyzer

```
override fun analyze(image: ImageProxy?, rotationDegrees: Int) {

    val currentTimestamp = System.currentTimeMillis()
    if (currentTimestamp - lastAnalyzedTimestamp >= TimeUnit.MILLISECONDS.toMillis( duration: 1)) {
        val cameraImage = image!!.image ?: return

        rettangolo = Rect(image.cropRect)

        val supp0 = image.image!!.planes[0].buffer.duplicate()
        val supp1 = image.image!!.planes[1].buffer.duplicate()
        val supp2 = image.image!!.planes[2].buffer.duplicate()
        val bitmap = ConvertiImmagine.imageToBitmap3(
            supp0, supp1,supp2,cameraImage.height, cameraImage.width,
            rotationDegrees
        )

        val croppedBitmap = bitmap.scale(classifier!!.getImageSizeX(),
            classifier!!.getImageSizeY(), filter: true)

        val results = classifier!!.recognizeImage(croppedBitmap,
            preview_x, preview_y)
        r(results)

        lastAnalyzedTimestamp = currentTimestamp
    }

}
```

Figura 44. Funzione analyze interna a custom analyzer

In figura 44 è mostrata la funzione *analyze*, parte chiave della *MainActivity*: qui si ottengono le informazioni derivanti dal processo inferenziale a partire dall'immagine proveniente dalla fotocamera. Dopo un primo controllo sull'integrità dell'immagine, è presente un giro di scomposizione e ricomposizione della stessa, necessario in quanto per la valutazione avevo bisogno di un bitmap (la funzione *imageToBitmap3* di *ConvertiImmagine* infatti ha lo scopo di tradurre l'immagine ricevuta dalla fotocamera dal formato YUV420 in bitmap). Il motivo per cui la scomposizione si è resa necessaria sta nel fatto che la funzione *analyze* è un mondo chiuso e non si può passare l'immagine senza che questa venga distrutta mentre, con questo trucchetto, si aggira il problema: provando a tradurre direttamente l'immagine in bitmap infatti la funzione *analyze* andava in crash poiché perdeva il riferimento all'immagine. All'inizio della funzione

analyze ho anche inserito un controllo sul timestamp dell'ultima analisi così da poter avere un tempo minimo tra un'analisi e l'altra: in questa maniera si può risparmiare un po' di computazione nel caso fosse necessario o, motivo per cui è stato introdotto, vedere meglio i risultati a schermo. Chiaramente così impostato ad un millisecondo non ha nessun effetto.

Backend

Il vero lavoro di classificazione però viene fatto dalla funzione *recognizeImage* del classificatore. Questo è descritto nell'immagine sotto (figura 45), ma ci sono due aspetti di particolare interesse: in primis il fatto che si utilizzi, tra le opzioni di questo, un delegato della GPU in modo tale da accelerare le operazioni di inferenza e, in secondo luogo, il fatto che, partendo da modello e opzioni, venga creato l'interprete, ovvero quel componente sul quale si andrà poi ad invocare il comando per fare inferenza.

```
protected Classifier(Activity activity) throws IOException {

  Log.d( tag: "Classifier",  msg: "Stato: dentro CLASSIFIER");
  tfliteModel = loadModelFile(activity);
  Log.d( tag: "Classifier",  msg: "Stato: caricato il modello");
  gpuDelegate = new GpuDelegate();
  Log.d( tag: "Classifier",  msg: "Stato: fatto il delegato");
  tfliteOptions.addDelegate(gpuDelegate);
  Log.d( tag: "Classifier",  msg: "Stato: aggiunto delegato alle opzioni");
  tflite = new Interpreter(tfliteModel, tfliteOptions);
  Log.d( tag: "Classifier",  msg: "fatto il tflite");
  labels = loadLabelList(activity);
  Log.d( tag: "Classifier",  msg: "Stato: caricate le labels");
  imgData =
          ByteBuffer.allocateDirect(
                  DIM_BATCH_SIZE
                          * getImageSizeX()
                          * getImageSizeY()
                          * DIM_PIXEL_SIZE
                          * getNumBytesPerChannel());
  Log.d( tag: "Classifier",  msg: "Stato: fatto imageData");
  imgData.order(ByteOrder.nativeOrder());
  Log.d( tag: "Classifier",  msg: "Stato: settato ordine imageData");
}
```

Figura 45. Il classificatore

Analizzando la funzione *recognizeImage* nello specifico (figura 46), si vede che è qui che viene invocata la *runInference*, funzione che ci restituisce i risultati del processo di inferenza che poi vengono opportunamente organizzati (figura 47). Qui viene anche controllato che le proposte individuate dalla rete rispettino i parametri di confidenza minima voluti dall'utente per essere mostrate a schermo (in questo caso ho deciso di restituire tutte quelle con una confidenza

maggiore del 50%); di queste vengono poi anche impostate correttamente le dimensioni della bounding box, ovvero viene generato il rettangolo float chiamato *detection* (figura 46). Il tutto viene impacchettato in una Array List che è poi restituita alla funzione *analyze* della *MainActivity*.

```
public List<Recognition> recognizeImage(final Bitmap bitmap, final int x, final int y) {
    Trace.beginSection( sectionName: "RECOGNIZEIMAGE");
    Trace.beginSection( sectionName: "preprocessBitmap");
    convertBitmapToByteBuffer(bitmap);
    Trace.endSection();

    Trace.beginSection( sectionName: "feed");
    outputLocations = new float[1][NUM_DETECTIONS][4];
    outputClasses = new float[1][NUM_DETECTIONS];
    outputScores = new float[1][NUM_DETECTIONS];
    numDetections = new float[1];

    Object[] inputArray = {imgData};
    Map<Integer, Object> outputMap = new HashMap<>();
    outputMap.put(0, outputLocations);
    outputMap.put(1, outputClasses);
    outputMap.put(2, outputScores);
    outputMap.put(3, numDetections);
    Trace.endSection();

    runInference(inputArray, outputMap);
    Trace.endSection();

    final ArrayList<Recognition> recognitions = new ArrayList<>(NUM_DETECTIONS);
    for (int i = 0; i < NUM_DETECTIONS; ++i) {
        if ( outputScores[0][i] > MINIMUM_CONFIDENCE){
            final RectF detection =
                    new RectF(
                            left: outputLocations[0][i][1] * x,
                            top: outputLocations[0][i][0] * y,
                            right: outputLocations[0][i][3] * x,
                            bottom: outputLocations[0][i][2] * y);
            int labelOffset = 1;
            recognitions.add(
                    new Recognition(
                            id: "" + i,
                            labels.get((int) outputClasses[0][i] + labelOffset),
                            outputScores[0][i],
                            detection));
        }
    }
    Trace.endSection(); // "recognizeImage"
    return recognitions;
}
```

Figura 46. La funzione di riconoscimento dell'immagine

```
protected void runInference(Object[] inputArray, Map<Integer, Object> outputMap){
    tflite.runForMultipleInputsOutputs(inputArray, outputMap);
}
```

Figura 47. La funzione runInference

Infine, per quanto riguarda la funzione che effettivamente va a disegnare le box intorno agli oggetti, in una prima versione avevo scritto una classe a mano che funzionava ma era di un livello molto semplice. Continuando poi a consultare esempi fatti dal gruppo di ML Kit e di TensorFlow, mi sono imbattuto in una versione nettamente superiore dal punto di vista estetico e ho quindi deciso di abbandonare la mia versione a favore di questa.

Sicurezza del modello

Ci tengo a fare una nota finale su questa applicazione: al momento dell'esposizione del mio lavoro al gruppo di Mobile Android di Datalogic, essi hanno fatto sorgere una questione che fino a quel momento, non essendo io molto esperto di sviluppo Android, non avevo preso in considerazione, ovvero la sicurezza del modello stesso. Infatti, per come l'ho inserito io all'interno dell'applicazione, il file *detect.tflite*, punto cardine dell'intera applicazione e reale valore di questa, risulta essere una semplice risorsa, localizzata all'interno della cartella contenente tutti gli assets utilizzati dall'applicazione. Questo lo rende facilmente prelevabile e riutilizzabile dai concorrenti qualora questa soluzione risultasse essere davvero applicabile commercialmente. Non avendo pensato a questa evenienza, ho deciso di fare una ricerca in merito, indagando eventuali modi proposti da Google per ovviare a questa possibilità. La soluzione che viene caldeggiata dal gruppo di TensorFlow per la distribuzione del modello è quella di ricorrere a Firebase anche se questo, come detto dalla stessa fonte, non rimuove in alcun modo la possibilità che un concorrente possa prenderlo e impacchettarlo nella sua versione dell'applicazione. Viene anche detto che solitamente questi modelli sono talmente tanto ottimizzati su un contesto specifico da non poter essere praticamente riutilizzati, ma in questo caso l'affermazione non mi sembra essere valida.

4 - Conclusioni

Il lavoro da me descritto in questa tesi è stata la parte cardine del mio tirocinio. Prima di sviluppare la versione finale dell'applicazione che ho descritto ho fatto svariati test con ML Kit, dato che lo scopo del lavoro era appunto quello di confrontare le prestazioni di un progetto custom sull'individuazione dei codici a barre con la funzione ML Kit di barcode detection.

Infatti di questa applicazione, costruita con Camera X, è presente anche una versione che per fare inferenza sui dati che provengono dalla fotocamera sfrutta, invece del modello tflite, direttamente ML Kit tramite appunto la funzionalità di barcode detection. Dato che la versione ML Kit non si occupa solamente dell'individuazione dei codici a barre ma li decodifica anche e dato che possiede una classe di riconoscimento molto più ampia (può infatti distinguere tredici differenti classi di codici a barre), paragonare queste due soluzioni non è facile. Nonostante questi fattori, questa versione risulta essere molto più performante rispetto a quella da me sviluppata in termini di latenza. Inoltre, la versione ML Kit difficilmente mostra a schermo dei falsi positivi, cosa che, come ho sottolineato nella parte di creazione del dataset, la mia versione invece fa su determinati pattern. I punti di forza principali della mia applicazione invece sono:

- Il fatto di essere totalmente personalizzabile: infatti in questa soluzione ogni parametro (architettura della rete, costruzione del dataset, oggetti da indagare, ecc) può essere configurato, mentre ML Kit deve essere preso come scatola nera e ci si può operare solamente ai morsetti;
- Risulta essere molto più robusta: questa è infatti in grado di tollerare meglio la distanza dal soggetto riuscendo comunque ad individuarlo (velocità di riconoscimento e distanza di riconoscimento sono due parametri che non vanno molto d'accordo dato che entrambi sono legati, l'uno l'inverso rispetto all'altro, alle dimensioni originali dell'immagine) e soprattutto è molto più robusta per quanto riguarda condizioni di luce e rotazione.

Per quanto appena esposto non so quanto senso abbia parlare di numeri e rapportarli tra di loro, ma comunque ci proverò. Prima di tutto è importante dire che le prestazioni di entrambe sono fortemente influenzate sia dal SoC del telefono su cui stanno girando che dalla tipologia della fotocamera presente sul dispositivo. Cosa da cui invece non sono influenzate le prestazioni in termini di latenza è il numero di elementi principali presenti sulla scena: che sia uno o che siano cinque il tempo di inferenza non varia.

Nell'immagine sottostante (figura 48 e 49) riporto un paio di test fatti con la versione ML Kit dell'applicazione.

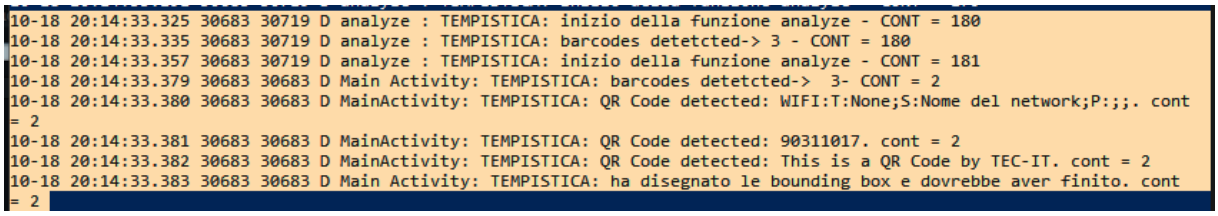

Figura 48. Risultato test su OnePlus 5 dell'applicazione versione "ML Kit"

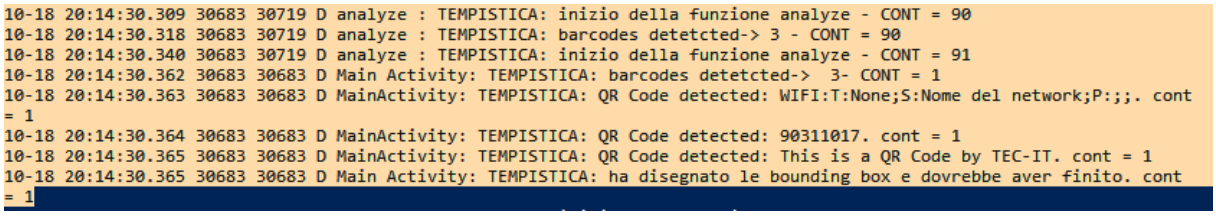

Figura 49. Risultato test su OnePlus 5 dell'applicazione versione "ML Kit"

Le tempistiche medie su svariati cicli di analisi fatti con il mio OnePlus 5, dotato di Snapdragon 835, mostrano un tempo medio di 53ms per quanto riguarda la versione ML Kit contro un tempo medio di 91ms per quanto riguarda la versione con il modello allenato da me. Questa differenza si fa ancora più ampia se si considera l'altro dispositivo utilizzato per fare i test, ovvero il Memor 20 di Datalogic, equipaggiato di uno Snapdragon 660: qui la versione ML Kit ha prestazioni nell'intorno degli 80ms, mentre con la mia versione i tempi oscillano molto dopo il primo avvio, ma hanno una media intorno ai 165ms.

La differenza tra le due versioni diventa tanto più marcata quanto più l'hardware del dispositivo mobile è di fascia bassa: questo deriva dal fatto che la mia implementazione sfrutta solamente la GPU (cosa che nei dispositivi di bassa fascia solitamente risulta essere poco potente) mentre ML Kit plausibilmente sfrutta numerosi metodi di ottimizzazione quali, ad esempio, quantizzazione e NNAPI.

Infine, riporto anche dei numeri che riguardano la mia versione dell'applicazione e che non hanno un corrispettivo per la versione di ML Kit. L'applicazione testata su Huawei P30 Pro mostra dei tempi nettamente inferiori rispetto a quelli evidenziati nel test con il mio daily driver: una media di 34ms di latenza tra un'analisi e l'altra, che coincide con il perfetto lavoro in real time.

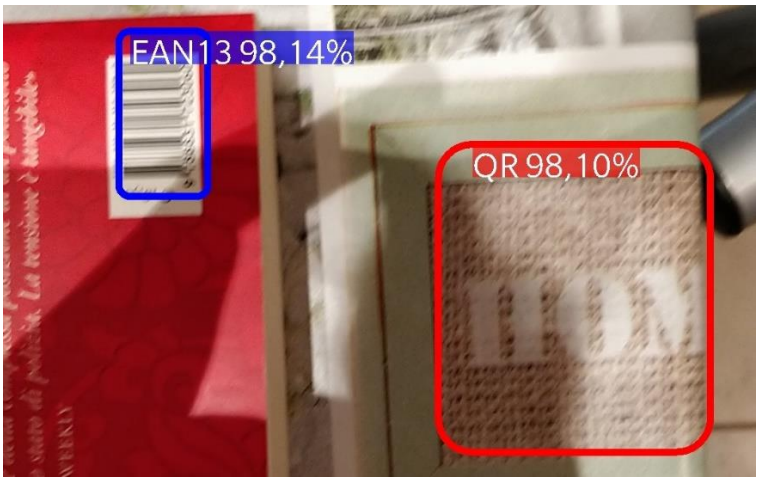

Figura 50. Screenshot dell'applicazione finale (catturato su OnePlus5). In evidenza un pattern problematico

Figura 51. Screenshot dell'applicazione finale (catturato su OnePlus5). In evidenza l'abilità di individuare diversi codici su diversi piani e sotto diverse condizioni di luminosità

Figura 52. Screenshot dell'applicazione finale (catturato su OnePlus5). In evidenza l'abilità di individuare codici a barre con angolazioni particolari su differenti piani

Quelli sopra riportati (figura 50-52) sono degli screenshots fatti sul mio dispositivo mentre l'applicazione era in esecuzione. Come si può vedere, se la distanza non è eccessiva, i codici a barre vengono riconosciuti con successo ma, a causa di come è costruito il dataset sintetico, ci sono alcuni pattern che fanno scattare il riconoscimento, anche con una dose di sicurezza abbastanza alta.

Sviluppi futuri

Alcuni sviluppi futuri volti a migliorare le performance dell'applicazione esistono già ora e potremmo dividerli in due categorie a seconda dello scopo: migliorare la latenza e migliorare le prestazioni di riconoscimento. Per quanto riguarda la prima parte, come già detto in precedenza, si potrebbe passare dalla versione due alla versione tre di MobileNet, in quanto questa, secondo [26], dovrebbe avere prestazioni nettamente superiori; altrimenti si potrebbe pensare di passare ad un modello totalmente quantizzato ed eseguito utilizzando le NNAPI, soluzione che, a seconda del SoC utilizzato, potrebbe avere un grosso impatto. Per quanto riguarda invece l'aumento delle capacità di riconoscimento, questo è legato all'architettura che si usa per il progetto, quindi utilizzare la nuova versione di MobileNet dovrebbe portare a dei

leggeri miglioramenti (principalmente legati al fatto che questa versione analizza immagini più grandi rispetto alla versione due).

Se invece vogliamo ragionare su modifiche che potrebbero migliorare la qualità della risposta della rete attuale, di sicuro il primo punto sarebbe quello di modificare il dataset: rendere molto più vasto il pool di immagini (sia foreground che background) per la generazione del dataset sintetico con elementi più sfidanti per la rete, risulterebbe sicuramente in un minor numero di falsi positivi. Altrimenti sarebbe ancora meglio costruire un dataset reale di dimensione consistente e, al contempo, applicare regole di augmenting in fase di allenamento.

Per concludere ci tengo a fare una considerazione personale sullo stato dell'arte della machine vision: penso che tutto questo sia fenomenale. Fino all'anno scorso, prima di cominciare questo percorso, non pensavo che le tematiche da me affrontate fossero ad uno stadio così avanzato e performante, né tantomeno che tutto questo potesse essere utilizzato a questo livello su piattaforme mobile che fanno della loro efficienza un selling point chiave.

Bibliografia

[1] «CameraX overview», *Android Developers*. https://developer.android.com/training/camerax

[2] «ML Kit», *Google Developers*. https://developers.google.com/ml-kit/guides

[3] «TensorFlow Lite guide», *TensorFlow*. https://www.tensorflow.org/lite/guide

[4] «TensorFlow Lite and TensorFlow operator compatibility», *TensorFlow*. https://www.tensorflow.org/lite/guide/ops_compatibility

[5] «Model optimization | TensorFlow Lite», *TensorFlow*. https://www.tensorflow.org/lite/performance/model_optimization

[6] «TensorFlow Lite NNAPI delegate», *TensorFlow*. https://www.tensorflow.org/lite/performance/nnapi

[7] «TensorFlow Lite GPU delegate», *TensorFlow*. https://www.tensorflow.org/lite/performance/gpu

[8] «AI-Benchmark». http://ai-benchmark.com/

[9] «AI-Benchmark-rank». http://ai-benchmark.com/ranking.html

[10] «AI-Benchmark-proc». http://ai-benchmark.com/ranking_processors.html

[11] A. Ignatov *et al.*, «AI Benchmark: Running Deep Neural Networks on Android Smartphones», *arXiv:1810.01109 [cs]*, ott. 2018. Available at: http://arxiv.org/abs/1810.01109.

[12] Y. Toda *et al.*, «Training instance segmentation neural network with synthetic datasets for crop seed phenotyping», *Communications Biology*, vol. 3, n. 1, Art. n. 1, apr. 2020, doi: 10.1038/s42003-020-0905-5.

[13] «Complete Guide to Creating COCO Datasets», *Udemy*. https://www.udemy.com/course/creating-coco-datasets/

[14] T.-Y. Lin *et al.*, «Microsoft COCO: Common Objects in Context», *arXiv:1405.0312 [cs]*, feb. 2015. Available at: http://arxiv.org/abs/1405.0312.

[15] «COCO - Common Objects in Context». https://cocodataset.org/#home

[16] R. Girshick, J. Donahue, T. Darrell, e J. Malik, «Rich feature hierarchies for accurate object detection and semantic segmentation», *arXiv:1311.2524 [cs]*, ott. 2014. Available at: http://arxiv.org/abs/1311.2524.

[17] J. Redmon, S. Divvala, R. Girshick, e A. Farhadi, «You Only Look Once: Unified, Real-Time Object Detection», *arXiv:1506.02640 [cs]*, mag. 2016. Available at: http://arxiv.org/abs/1506.02640.

[18] W. Liu *et al.*, «SSD: Single Shot MultiBox Detector», *arXiv:1512.02325 [cs]*, vol. 9905, pagg. 21–37, 2016, doi: 10.1007/978-3-319-46448-0_2.

[19] Z.-Q. Zhao, P. Zheng, S. Xu, e X. Wu, «Object Detection with Deep Learning: A Review», *arXiv:1807.05511 [cs]*, apr. 2019. Available at: http://arxiv.org/abs/1807.05511.

[20] K.-H. Kim, S. Hong, B. Roh, Y. Cheon, e M. Park, «PVANET: Deep but Lightweight Neural Networks for Real-time Object Detection», *arXiv:1608.08021 [cs]*, set. 2016. Available at: http://arxiv.org/abs/1608.08021.

[21] C.-Y. Fu, W. Liu, A. Ranga, A. Tyagi, e A. C. Berg, «DSSD : Deconvolutional Single Shot Detector», *arXiv:1701.06659 [cs]*, gen. 2017. Available at: http://arxiv.org/abs/1701.06659.

[22] Z. Shen, Z. Liu, J. Li, Y.-G. Jiang, Y. Chen, e X. Xue, «DSOD: Learning Deeply Supervised Object Detectors from Scratch», *arXiv:1708.01241 [cs]*, apr. 2018. Available at: http://arxiv.org/abs/1708.01241.

[23] J. Redmon e A. Farhadi, «YOLO9000: Better, Faster, Stronger», *arXiv:1612.08242 [cs]*, dic. 2016. Available at: http://arxiv.org/abs/1612.08242.

[24] A. G. Howard *et al.*, «MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications», *arXiv:1704.04861 [cs]*, apr. 2017. Available at: http://arxiv.org/abs/1704.04861.

[25] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, e L.-C. Chen, «MobileNetV2: Inverted Residuals and Linear Bottlenecks», *arXiv:1801.04381 [cs]*, mar. 2019. Available at: http://arxiv.org/abs/1801.04381.

[26] A. Howard *et al.*, «Searching for MobileNetV3», *arXiv:1905.02244 [cs]*, nov. 2019. Available at: http://arxiv.org/abs/1905.02244.

[27] «CUDA Toolkit 11.0 RC Download», *NVIDIA Developer*, ott. 06, 2015. https://developer.nvidia.com/cuda-downloads

[28] «GPU support», *TensorFlow*. https://www.tensorflow.org/install/gpu

[29] «Install TensorFlow with pip», *TensorFlow*. https://www.tensorflow.org/install/pip

[30] «TFRecord and tf.Example | TensorFlow Core», *TensorFlow*. https://www.tensorflow.org/tutorials/load_data/tfrecord

[31] «tensorflow/models», *GitHub*. https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

[32] «TOCO», *GitHub*. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/toco

[33] «TensorFlow/Bazel», *GitHub*. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android#bazel

[34] «Installing Bazel on Windows». /versions/master/install-windows.html

[35] «Build Bazel on Windows». https://docs.bazel.build/versions/master/windows.html#using

[36] «tensorflowlite/running mobile», *GitHub*. https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/running_on_mobile_tensorflowlite.md