UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Università degli studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di laurea
in Ingegneria Informatica

Machine learning su dispositivi mobili ed edge: il framework PyTorch Mobile

Relatore:
FANTOZZI CARLO

Laureando:
CARRARO EDDIE
2000151

Anno Accademico 2022-2023

Data di laurea 27/09/2023

Indice

Sommario

L'avvento di tecnologie mobili ha rivoluzionato il modo con cui interagiamo con gli strumenti informatici, nonché il nostro modo di interfacciarci col mondo. Dall'invenzione dell' IPhone, avvenuta nel 2007 con la presentazione iconica di Steve Jobs, è stata una guerriglia tra case tech su chi avrebbe sviluppato lo smartphone più potente, ergonomico e "bello" (scontro che si ripropone ogni anno all'uscita di nuovi telefonini). Proprio questa ricerca allo smartphone più potente ha portato i dispositivi mobili ad avere caratteristiche tecniche che farebbero impallidire i PC di medio gamma attuali, portandoli ad avere RAM e processori di ultima generazione. Questa "potenza tascabile", chiaramente, vuole essere sfruttata appieno, perciò vi sono sempre più applicazioni (che siano di editing o videogiochi) pesanti e che fanno "lavorare" a dovere gli smartphone. Con l'avvento del machine learning, si è voluto provare anche ad utilizzare, nei dispositivi mobili, modelli (appunto di machine learning) che fossero utili ad esempio alla segmentazione (per esempio per il cambio sfondo durante le videochiamate), all'object detection, ecc. Purtroppo, nonostante la potenza degli smartphone odierni, è necessario applicare una semplificazione ai modelli, in modo che su mobile funzionino bene e in tempi ridotti (ricordiamo che i tempi di attesa su uno smartphone devono essere molto minori rispetto a quelli su PC, per avere una usabilità decente). Per risolvere ciò, è stato creato PyTorch Mobile, che prende modelli scritti con PyTorch e li rende più leggeri e fruibili su dispositivi mobili. Quello di cui si occuperà la seguente è di comprendere se questa semplificazione comporti (e se si, in quale misura) imperfezioni e inaccuratezze del modello, per capire se l'utilizzo di PyTorch Mobile sia opportuno e utile; in effetti, un modello poco performante risulterebbe poco pratico e non varrebbe la pena usarlo, nonostante l'applicazione in ambito mobile.

Capitolo 1

Introduzione

PyTorch [34] è un framework di machine learning open source basato sulla libreria Torch, che negli ultimi anni ha visto la sua popolarità crescere. Il framework è stato creato da un team specializzato di Meta e il direttore Lin Qiao ne evidenzia le principali caratteristiche [34]:

- Esecuzione ”eager” mediante Python
- Permette di costruire reti neurali dinamiche
- Supporta il training distribuito
- Sfrutta gli acceleratori hardware
- Preferisce la semplicità alla complessità.

PyTorch fa da ”genitore” a **PyTorch Mobile** [21], introdotto alla PyTorch Developer Conference del 2019. Questo framework permette agli sviluppatori di lavorare con modelli PyTorch direttamente su dispositivi mobili (o ancora più interessante su dispositivi quali Oculus Quest e Portal) senza la necessità di dover riscrivere modelli già esistenti in PyTorch. In effetti, per fare in modo che i modelli di PyTorch funzionino su dispositivi mobili è necessario ”semplificarli”, per fare in modo che le limitate risorse del dispositivo riescano a gestire il carico di lavoro impartitogli e che l'operazione richiesta non impieghi troppo tempo per essere eseguita.
Quest'ultimo concetto è molto importante per l'usabilità dell'app in questione: per creare un'app che sia utilizzabile in maniera fluida, è necessario che le operazioni da compiere non impieghino troppo tempo, per non spazientire l'utente (quantomeno è necessario che vi sia la presenza di una barra di caricamento per dare all'utente finale un riferimento a quanto manca al termine di un determinato task)[2].

Al momento, PyTorch Mobile è ancora in beta, perciò alcune funzionalità potrebbero non essere complete.

1.1 Caratteristiche principali di PyTorch Mobile

Le principali caratteristiche di PyTorch mobile sono le seguenti:

- Disponibile per iOS, Android e Linux
- Possiede molteplici API, che coprono vari task di preelaborazione e integrazione
- Supporta la libreria XNNPACK e integra QNNPACK per kernel quantizzati con 8-bit
- Ottimizzazione di modelli PyTorch con optimize_for_mobile
- Fornisce un interprete mobile in Android e iOS
- (beta) Supporto a backend hardware come GPU, DSP e NPU (per la GPU su iOS viene utilizzato Metal, mentre in Android Vulkan).

In particolare, XNNPACK [36] è una libreria che permette di migliorare le performance dell'inferenza su CPU di una rete neurale. Nello specifico, fornisce delle implementazioni ottimizzate di operatori che accelerano il processo. Invece, QNNPACK [24] è una libreria ottimizzata per mobile che fornisce implementazioni di reti neurali comuni a tensori a 8-bit quantizzati.

1.2 Torchvision

Torchvision [32] è una "rappresentazione intermedia" di un modello PyTorch che può essere eseguito in un ambiente ad alte prestazioni, come C++. Esso è composto da dataset popolari, serve per la trasformazione di immagini per computer vision, ecc. Tra i vari pro vi sono i seguenti: supporta la GPU (per cui le esecuzioni sono più veloci), è "pronto all'uso", nel senso che comprende già di default dei modelli pre-allenati e degli esempi di dataset.

1.3 Il principale "rivale": TensorFlow Lite

Uno dei principali framework concorrenti di PyTorch Mobile è **TensorFlow Lite** [22], con il quale condivide alcune caratteristiche e dal quale si differenzia in alcuni ambiti. N.B.: Tensorflow e TensorFlow Lite sono due framework molto diversi, che condividono solo il nome.
La differenza chiave fra i due framework è il workflow utilizzato per usufruire del codice presente rispettivamente in PyTorch e TensorFlow. Se per PyTorch Mobile (come visto nella Sezione 1.4) bastava ottimizzare e (opzionalmente) quantizzare il modello, essendo TensorFlow e TFLite framework separati è necessario stare più attenti (anche perché non tutti i modelli disponibili per TensorFlow sono convertibili per l'utilizzo in TFLite) e modificare i modelli per l'utilizzo mobile/embedded.

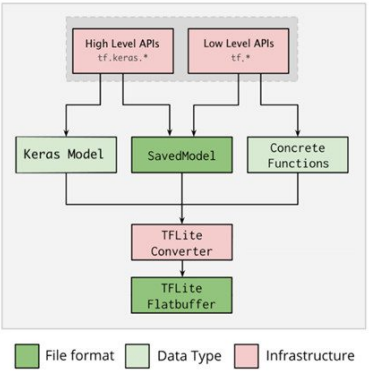


Figura 1.1: Modello di workflow per conversione in TFLite

Anche la lista di operatori disponibile è diversa: in PyTorch Mobile sono disponibili tutti gli operatori (su CPU) presenti anche su PyTorch, mentre TFLite ha una lista di operazioni principali supportate dal suo runtime.

Per quanto riguarda la sintassi di PyTorch (Mobile), questa è molto ”Python friendly”, nel senso che il codice usato con PyTorch è estremamente simile all'utilizzo ”raw” di Python. Al contrario, TensorFlow vi si differenzia di più.

Per la visualizzazione dei modelli, PyTorch Mobile utilizza **TensorBoard** o torchinfo, mentre TFLite fornisce un Model Analyser Tool per l'analisi del modello.

In aggiunta, entrambi i framework supportano vari backend (legati sia alla CPU che alla GPU) e permettono di fare il benchmark dei modelli che utilizzano, oltre a fornire moltissimi modelli pre-allenati (i quali, in PyTorch Mobile, devono essere ri-salvati nel lite interpreter per l'utilizzo nelle piattaforme mobili).

Se TensorFlow è un ambiente più maturo e completo (nonostante le difficoltà maggiori nella conversione di modelli da TensorFlow a quest'ultimo), PyTorch è ancora in stato semi-embrionale (con molte funzionalità ancora in beta) ma con una community forte alle spalle.

1.4 Deployment Workflow

Per utilizzare un modello PyTorch in PyTorch Mobile, si considera come base lo schema rappresentato nella Figura 1.4 [21].

1.4.1 Quantizzazione

La quantizzazione [25] comprende varie tecniche per memorizzare tensori utilizzando una minor precisione rispetto al floating point; questo permette anche di eseguire più velocemente operazioni fra i tensori stessi. Solitamente, il modello viene allenato in FP32 (Floating Point 32 bit) e poi convertito (e perciò quantizzato) in INT8 (Integer 8 bit). Riprendendo ciò che viene detto nella

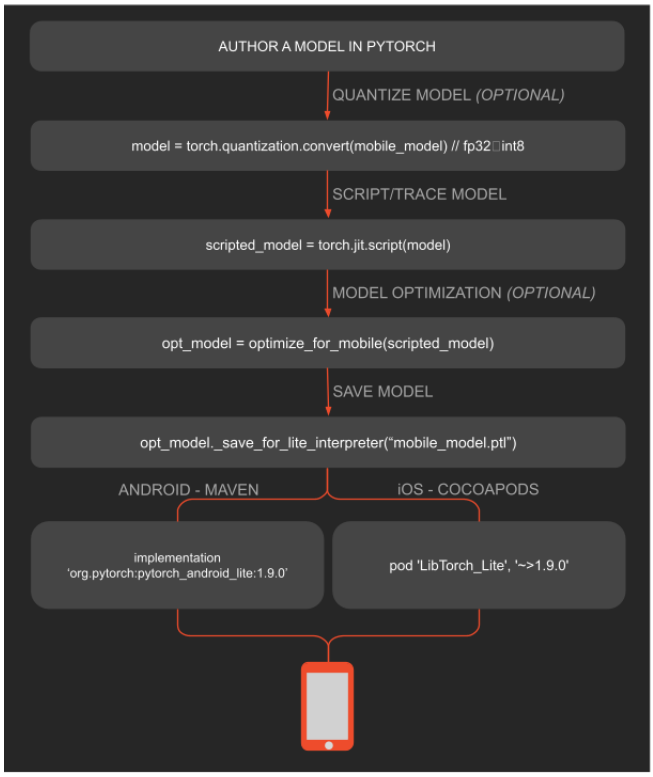


Figura 1.2: Deployment Workflow per modelli in PyTorch Mobile

documentazione [25], "PyTorch supports INT8 quantization compared to typical FP32 models allowing for a 4x reduction in the model size and a 4x reduction in memory bandwidth requirements. Hardware support for INT8 computations is typically 2 to 4 times faster compared to FP32 compute". Nota: gli operatori quantizzati supportano solo il forward pass.

PyTorch permette di utilizzare due diversi modi per quantizzare:

- Eager Mode Quantization (beta)
- FX Graph Mode Quantization (prototipo)

La quantizzazione in Eager Mode è prettamente manuale, nel senso che è necessario specificare manualmente in quali fasi avvengono quantizzazione e dequantizzazione; inoltre, supporta solo moduli e non i functional, dove i functional sono le interfacce base per applicare gli operatori di PyTorch ai tensori, mentre i moduli rappresentano un modo diverso per accedere agli operatori forniti dai functional.

Al contrario, la quantizzazione FX Graph Mode aggiunge il supporto ai functional e automatizza il processo di quantizzazione (anche se potrebbe essere necessario fare il refactor del modello per renderlo compatibile con l'FX Graph Mode stessa).

Vi sono poi 3 tipi differenti di quantizzazione supportati:

- quantizzazione dinamica (pesi quantizzati con activations lette/salvate in floating point e quantizzate per i calcoli);
- quantizzazione statica (pesi quantizzati, activations quantizzate, con calibrazione richiesta dopo il training);
- quantizzazione statica ”aware training” (pesi quantizzati, activations quantizzate, dati numerici modellati durante il training).

Nella Figura 1.3 vengono mostrati quali operatori sono compatibili con i diversi tipi di quantizzazione (statica e dinamica).

	Static Quantization	Dynamic Quantization
nn.Linear	Y	Y
nn.Conv1d/2d/3d	Y	N
nn.LSTM	Y (through custom modules)	Y
nn.GRU	N	Y
nn.RNNCell	N	Y
nn.GRUCell	N	Y
nn.LSTMCell	N	Y
nn.EmbeddingBag	Y (activations are in fp32)	Y
nn.Embedding	Y	N
nn.MultiheadAttention	Y (through custom modules)	Not supported
Activations	Broadly supported	Un-changed, computations stay in fp32

Figura 1.3: Compatibilità con i diversi tipi di quantizzazione

1.4.2 Scripting e Tracing

In generale, Tracing e Scripting servono a trasformare un nn.Module in un grafo rappresentabile in formato TorchScript. L'operazione *torch.jit.script(obj, optimize=None, _frames_up=0, _rcb=None, example_inputs=None)* ispeziona il codice sorgente, lo compila come codice TorchScript usando il compilatore TorchScript e ritorna uno ScriptModule o ScriptFunction. L'alternativa è l'utilizzo di *torch.jit.trace()*, che però è limitato al solo utilizzo con operazioni che vengono eseguite con un input *specifico* [23]:

```
dummy_input = torch.rand(1, 3, 224, 224)
torchscript_model = torch.jit.trace(model_quantized, dummy_input)
```

in cui deve essere specificato un esempio di ”dummy input” [11], ovvero di un valore che funge da segnaposto e che, quindi, definisce le dimensioni dell'input passato al modello.

1.4.3 Ottimizzazione

L'operazione *opt_model = optimize_for_mobile(scripted_model)* [19] serve a svolgere un insieme di task utili all'ottimizzazione del modello in modalità **eval**. Le ottimizzazioni applicate sono molteplici, e se non vengono specificate in optimization_blocklist (parametro della funzione) vengono applicate tutte:

- Fusione di Conv2D e BatchNorm.
- Prepacked Insert e Fold, che rimpiazza le convoluzioni 2D e le ottimizzazioni lineari con le controparti pre-impacchettate (prepacked).
- Fusione di ReLU e Hardtanh
- Rimozione del Dropout: rimuove i nodi *dropout* e *dropout_* quando ”training” è false.
- Conv packed params hoisting: sposta i parametri impacchettati (packed) nel modulo radice, in modo tale che le strutture di convoluzione possano essere rimosse. Questo riduce la dimensione del modello.
- Fusione di Add e ReLU: trova istanze di ottimizzazioni di tipo relu che seguono ottimizzazioni di tipo add e le fonde insieme in un singolo add_relu.

La firma completa della funzione è la seguente;

```
torch.utils.mobile_optimizer.optimize_for_mobile(script_module,
    optimization_blocklist=None,vpreserved_methods=None, backend='CPU')
```

e i suoi parametri sono i seguenti:

- script_module: un'istanza di un modello scripted di tipo, appunto, ScriptModule.
- optimization_blocklist (opzionale): un set di tipo MobileOptimizer. Quando non viene passato, il metodo di ottimizzazione esegue tutti i passi di ottimizzazione; altrimenti, verranno eseguiti soltanto quelli che non vengono inclusi nel set.
- preserved_methods: lista opzionale di metodi che devono essere mantenuti quando viene invocato il passo ”freeze_module”.
- backend: tipo di device usato per eseguire il modello in output (di default è la CPU, altrimenti possono essere usati Vulkan o Metal).

1.4.4 Vulkan e Metal

Da PyTorch 1.7 è possibile eseguire l'inferenza dei modelli su GPU che supportano **Vulkan** [33]. Vulkan può essere usato su Windows, Linux, Mac ma principalmente è pensato per device Android. Non è incluso di default in PyTorch e per questo è necessario impostare una variabile d'ambiente "USE_VULKAN".

Da PyTorch 1.12, invece, è possibile sfruttare la GPU Apple Silicon mediante **Metal** [18], per ottenere training di modelli significativamente più veloci usando un dispositivo Mac. Metal può essere attivato usando le Metal Performance Shaders (MPS) di Apple come backend per PyTorch, il quale viene esteso dalle MPS stesse.

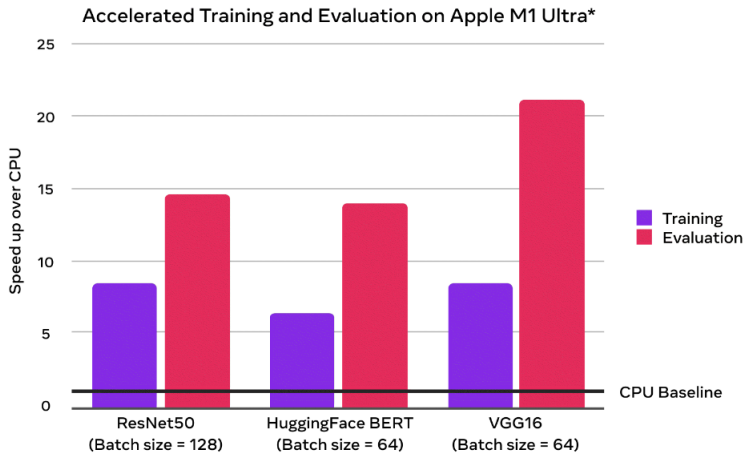


Figura 1.4: Training e evaluation con GPU vs uso di sola CPU (numero di volte più veloce)

In aggiunta, PyTorch (Mobile) supporta l'utilizzo di **CUDA** [6], una piattaforma di parallel computing creata da NVIDIA che permette di velocizzare l'esecuzione di un'applicazione utilizzando la GPU.

1.5 Segmentazione

In questo documento ci si è concentrati maggiormente nella segmentazione di immagini. La **segmentazione** [13] permette di dividere un'immagine in quelli che vengono definiti segmenti, che rappresentano (con la miglior precisione possibile) i vari oggetti/persone/animali presenti.
La segmentazione viene utilizzata in svariati ambiti:

- Guida autonoma (per rilevare altre vetture, pedoni o altro),
- Rimozione dello sfondo (utile ad esempio per programmi di editing),
- In ambito medico, per riconoscere vari organi e svolgere diagnosi (in base al loro aspetto è possibile rilevare malattie o altro),

- Immagini satellitari, per identificare montagne, fiumi e altri terreni,
- Ispezione industriale, ad esempio per trovare difetti nei materiali, per esempio perdite d'acqua, ecc.

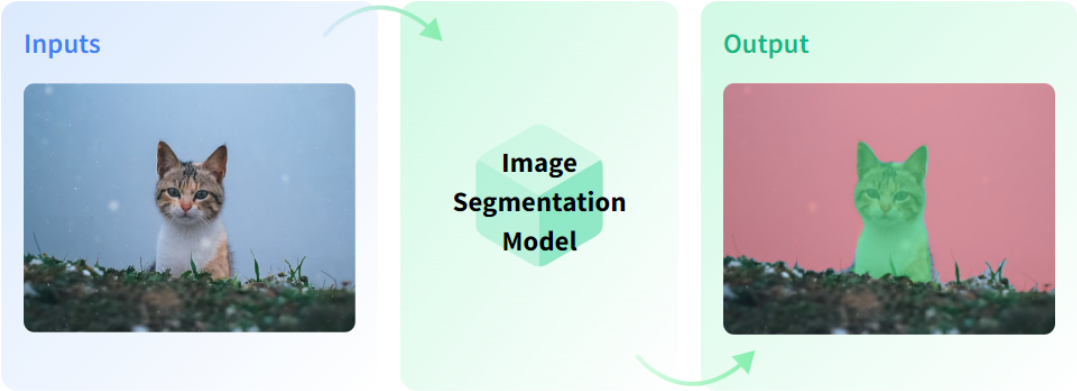


Figura 1.5: Un esempio di segmentazione di immagini

Vi sono 3 principali tipi [28] di segmentazione:

1. Segmentazione semantica
2. Segmentazione d'istanza
3. Segmentazione panottica

La prima è quella utilizzata nel nostro caso. Per poter allenare una rete per la segmentazione semantica bisogna innanzitutto analizzare una collezione di immagini aventi i pixel "etichettati" (labeled), e dopo aver creato la rete bisogna allenarla per farle imparare le varie categorie da riconoscere. Infine, bisogna valutare l'accuratezza della rete.
Per quanto riguarda la segmentazione **d'istanza**, questa consiste nel creare un contorno che racchiude i vari oggetti rilevati. Contrariamente alla segmentazione semantica, in questo caso non tutti i pixel vengono etichettati.
Infine, la segmentazione **panottica** rappresenta un mix delle due precedenti: etichetta ogni pixel, ma distingue varie istanze di una stessa classe (se ad esempio ci sono due persone vicine, le distingue con colori diversi per distinguere i pixel di una da quelli dell'altra).

1.5.1 Segmentazione semantica: come funziona

Come già detto poc'anzi, l'obiettivo della segmentazione semantica [29] è quello di etichettare ogni pixel dell'immagine presa in considerazione (che essa sia un'immagine a colori (altezza*lunghezza*3) o in scala di grigi (altezza*lunghezza*3)) con una label in modo da ottenere, in

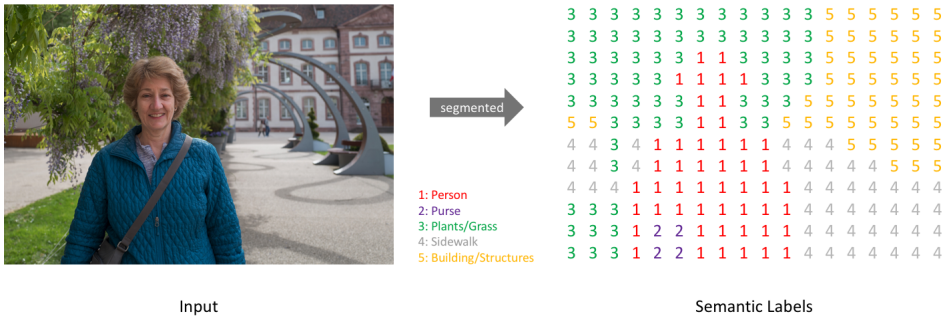


Figura 1.6: Esempio di segmentation map

output, una **segmentation map** in cui ogni pixel contiene la label corrispondente (la segmentation map è grande altezza*lunghezza*1).

Così facendo, creiamo una cosiddetta **maschera**.

N.B.: nella Figura 1.6, per chiarezza, la segmentation map è stata creata usando l'immagine a sinistra, ma con una risoluzione minore. In effetti, come già ribadito più volte, il numero di label deve essere uguale al numero di pixel presenti nell'immagine.

Utilizzando poi le label ricavate dalla segmentazione, possiamo distinguere i vari "soggetti" presenti nell'immagine.

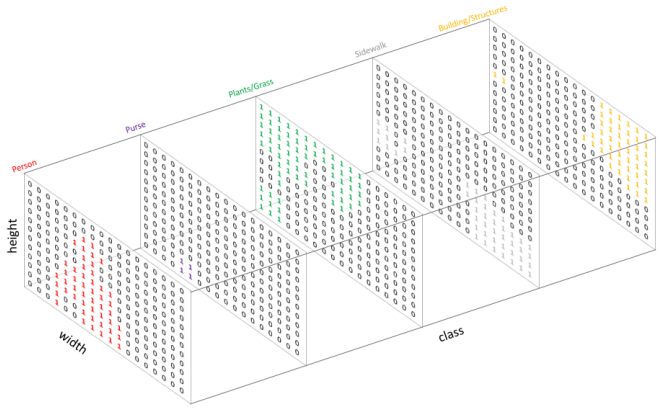


Figura 1.7: Distinzione delle varie classi di soggetto

Per costruire un'architettura che riesca ad applicare la segmentazione, l'approccio più semplice che potrebbe venire in mente è quello di creare una rete neurale collegando fra loro vari layer convoluzionali, con lo stesso padding per mantenere le dimensioni dell'immagine iniziale. Il problema principale è che ciò sarebbe computazionalmente molto costoso, in quanto sarebbe necessario mantenere la risoluzione massima dell'immagine attraverso tutta la rete. Il punto è che, per la segmentazione, è necessario disporre dell'immagine a risoluzione alta, in quanto è necessario sia riconoscere i vari soggetti ma anche capire dove questi siano posizionati precisamente (questa necessità non è presente, ad esempio, per la Classificazione d'Immagine, in quanto in

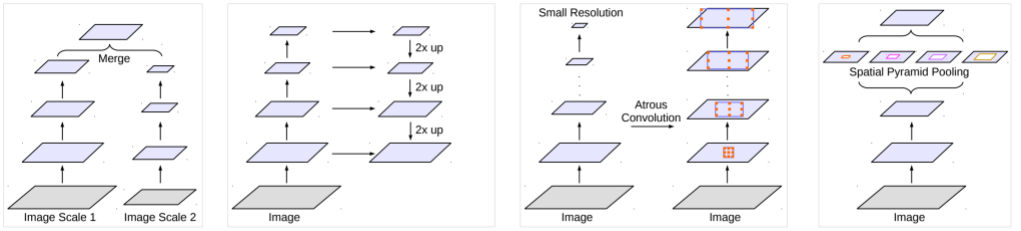
quel caso è sufficiente capire *cosa* ci sia nell'immagine, più che *dove* questo sia posizionato). Uno degli approcci più popolari per risolvere questo problema è quello di seguire una **Struttura Encoder/Decoder** (che troviamo anche per quanto riguarda i modelli DeepLabv3, vedi Sezione 1.5.2), in cui viene inizialmente fatto un *downsample* alla risoluzione dell'input, permettendo un minor costo computazionale passando attraverso la rete e creando delle feature map a bassa risoluzione, per poi fare un *upsample* di queste in una segmentation map a massima risoluzione.

1.5.2 DeepLabv3

Durante gli esperimenti svolti utilizzando la segmentazione è stato usato DeepLabv3 [7] [8], un'architettura di **segmentazione semantica** [30]. DeepLabv3 è stato lo Stato dell'Arte in materia per un po' di tempo (ed è ancora un'architettura molto utilizzata), e nel nostro caso è stato usato con **Resnet-50** [26], ovvero una rete neurale convoluzionale avente 50 livelli. DeepLabv3 è stato sviluppato dal team di Google per affrontare il problema, per l'appunto, della segmentazione semantica. Rappresenta un aggiornamento delle sue due precedenti versioni (v1 e v2), le quali sono decisamente meno potenti. Inoltre, rispetto alla versione 3, le precedenti due utilizzavano "*DensCRF*", ovvero un modello non allenabile. Al contrario, DeepLabv3 presenta un sistema allenabile e si distingue per le seguenti caratteristiche:

- Alta velocità
- Miglior precisione
- Semplicità architetturale
- Generalizzazione nel personalizzare i task.

DeepLabv3 vuole risolvere due problemi frequenti nella segmentazione: in primo luogo, la *riduzione di feature* andando in profondità nella rete (ciò permette il risparmio di risorse, ma può diventare un problema in quanto le feature che denotano la posizione degli oggetti nell'immagine possono essere perse); in secondo luogo, il fatto che lo stesso oggetto può essere rappresentato in **molteplici scale** (una rete potrebbe riconoscere un essere umano solamente se "grande" rispetto all'immagine, mentre potrebbe non riconoscerlo se troppo piccolo o in secondo piano). N.B.: "andare in profondità" in una rete significa, di fatto, progredire attraverso i livelli della rete stessa, "allontanandosi" dall'immagine di input (in effetti, una rete è composta da molteplici livelli, i quali svolgono delle azioni specifiche).
La soluzione al primo problema si chiama "**Atrous Convolution**", un modulo comunemente conosciuto come "dilated convolution", mentre al secondo problema pone rimedio l'"**Atrous Spatial Pyramid Pooling**", usato per l'estrazione di feaure su più livelli. Sempre per quanto riguarda il problema della molteplicità delle scale, sono stati pensati 4 differenti approcci [27] che furono poi raggruppati in diverse categorie.



(a) Image Pyramid (b) Encoder-Decoder (c) Deeper w. Atrous Convolution (d) Spatial Pyramid Pooling

Figura 1.8: Differenti architetture per risolvere il multi-scale problem

1. Image Pyramids: l'immagine di input viene ridimensionata a varie grandezze; le nuove immagini vengono poi passate alla rete. Successivamente, la rete neurale estrae varie feature per ogni diverso scalamento.

2. Encoder-Decoder structure: l'encoder cattura le informazioni essenziali dell'immagine, gradualmente riducendo le dimensioni spaziali della mappa di feature; dopodiché il decodificatore recupera i dettagli cruciali degli oggetti (come i bordi).

3. Context Module: usato in DeepLabv1 e v2, utilizza dei moduli extra in cascata sulla rete originale per cogliere le informazioni essenziali dell'immagine.

4. Spatial Pyramid Pooling(SPP): esplora la mappa di feature in ingresso con filtri o operazioni di pooling, catturando oggetti a diverse scale. Viene poi eseguita l'aggregazione delle informazioni.

Parlando delle componenti utilizzate da DeepLabv3, le due principali sono l'Atrous Convolution e l'Atrous Spatial Pyramid Pooling. L'**Atrous Convolution** è una tecnica che permette di controllare la risoluzione con cui le features sono calcolate dalla rete neurale (in particolare da una DCNN, come nel nostro caso) e modifica il cosiddetto "Field of View" (FOV) del filtro (per **Field of View** [12] si intende la massima area di un soggetto che una fotocamera(o qualsiasi altro strumento) può "cogliere" e dipende dal numero di pixel e dalla loro grandezza, ergo dalla densità). Quello che si fa, di fatto, è di aggiungere dei "buchi" ("trous" in Francese) tra i pesi del filtro (come si vede nella Figura 1.9), permettendo appunto un ingrandimento del field of view.

Ciò che rende questo componente vantaggioso è che permette l'ingrandimento del FOV senza che questo aumenti il numero di parametri o il numero di componenti da calcolare.

Vediamo due esempi di DCNN, una utilizzante l'Atrous Convolution e una no.

Nella Figura 1.10 (a) la rete è costruita in modo che la feature map in output sia progressivamente sempre più piccola; questa tipologia di rete è molto utile per cogliere informazioni a "lungo raggio" (di fatto l'immagine in output, pur essendo piccola, riassume le feature di tutta l'immagine; tuttavia,

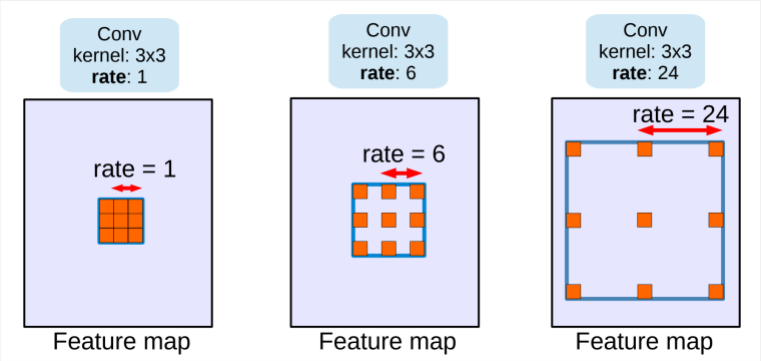
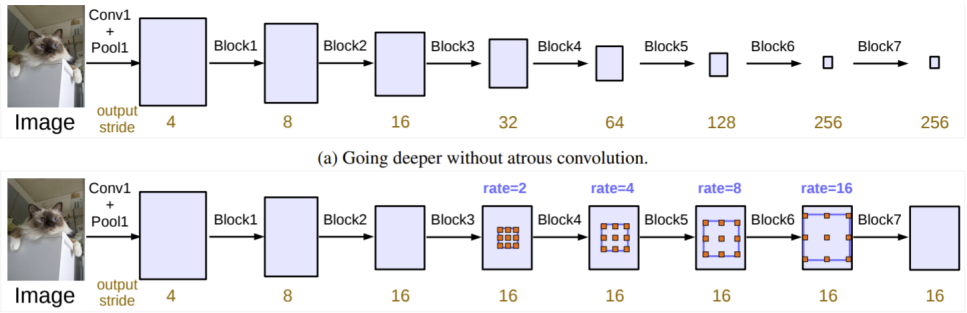


Figura 1.9: Esempio di Atrous Convolution, con i pesi a diverse distanze



(a) Going deeper without atrous convolution.

(b) Going deeper with atrous convolution. Atrous convolution with $rate > 1$ is applied after block3 when $output\_stride = 16$.

Figura 1.10: DCNN senza Atrous Convolution (a) e con Atrous Convolution (b)

una feature map così piccola non permette di compiere task di segmentazione che richiedono informazioni spaziali dettagliate). Nella parte (b), invece, viene preservata la risoluzione spaziale, grazie all'aumento del "dilation rate" per ogni blocco. Di conseguenza, il field of view viene aumentato, in modo tale da ottenere risultati migliori. La peculiarità di tutto ciò è che, utilizzando diversi valori per il dilation rate, è possibile avere diversi field of view e, di conseguenza, è possibile calcolare le feature di un'immagine per differenti scale di soggetti, tutto ciò senza ridurre la grandezza delle feature map.

Per quanto riguarda invece l' **Atrous Spatial Pyramid Pooling** (ASPP), questo è praticamente identico allo Spatial Pyramid Pooling (visto poc'anzi), con l'aggiunta dell'Atrous Convolution. L'ASPP impiega più layer di atrous convolution in parallelo, i quali permettono di "testare" l'immagine originale con filtri diversi (e complementari) aventi diverse modifiche per l'FOV, in modo da riconoscere i soggetti in scale diverse. Infine, le feature estratte da ogni iterazione parallela vengono processati in branch separati e fusi per generare il risultato finale. Questo, però, porta ad un problema, colto dagli autori dell'ASPP: "As the sampling rate becomes larger, the number of valid filter weights (i.e., the weights applied to the valid feature region, instead of padded zeros) becomes smaller." Ciò viene mostrato nella Figura 1.11; nell'estremo caso in cui il rate value è simile alla grandezza della feature map, il filtro 3*3, invece di cogliere il contesto dell'immagine

intera, degenera in un semplicissimo filtro 1*1, in quanto soltanto il peso centrale del filtro è efficace.

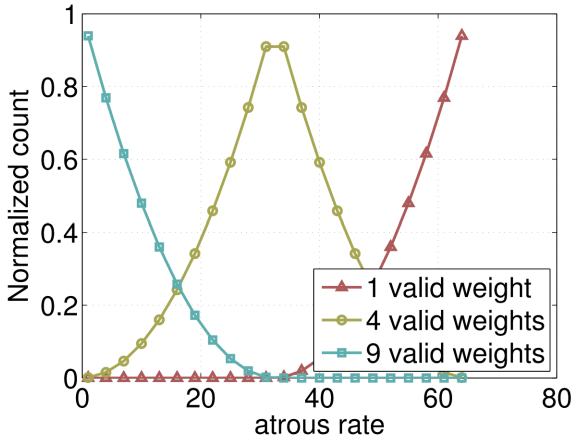


Figura 1.11: Filtro (3*3) applicato ad una feature map (65*65) con diversi atrous rates

Per ovviare al problema, viene aggiunto un modulo ”image pooling”, che usa un'operazione di global average pooling per incorporare informazioni di contesto globali. Le feature risultanti vengono passati a una convoluzione 1*1 con 256 filtri, per poi farne l'upsample alle dimensioni spaziali desiderate. Tutte le feature risultanti da tutti i branch vengono poi concatenate e passate ad un'altra convoluzione 1*1, per poi passarle ad un'altra convoluzione 1*1 che genera i logit finali.

1.6 Dataset utilizzato

Il dataset [17] utilizzato per la parte sperimentale consiste in circa tre centinaia di immagini di volti di esseri umani, delle più disparate età e caratteristiche fisiche (si va da neonati ad anziani, da uomini a donne). Oltre alle foto dei suddetti, vi sono anche le corrispettive **ground truth**, ovvero le segmentazioni delle immagini che dovrebbero essere corrette. In realtà si può vedere facilmente che ciò non vale per tutte le immagini. Come si può notare nella Figura 1.12, per esempio, una parte dell'occhio non viene segmentata a dovere.

Come si può vedere dalla Figura 1.13, inoltre, notiamo come nel database siano presenti ritratti di uomini,donne e bambini delle più disparate età, carnagioni, e con diversi tipi di luminosità (si va da immagini più definite e nitide ad altre con luci pessime o, in altri casi, poco luminose).

Figura 1.12: Immagine del database (sinistra) e relativa ground truth (destra)

Figura 1.13: Esempi (tra loro diversificati) di immagini nel database utilizzato

Capitolo 2

Codice Python

Oltre al codice per ”importare” il modello necessario alla segmentazione, è necessario applicare il modello e valutare la precisione della segmentazione attuata rispetto alla ground truth: in questo modo avremo una misura quantitativa di quanto buono è il modello e dell'imprecisione aggiunta dal modello ottimizzato per dispositivi mobili.

2.1 Utilizzo di un modello

In questo documento verrà utilizzato un modello già esistente e pre-allenato, che utilizza l'architettura DeepLabv3 [7].

```
import torch
from torch.utils.mobile_optimizer import optimize_for_mobile

model = torch.hub.load('pytorch/vision:v0.7.0', 'deeplabv3_resnet50', pretrained=True)
model.eval()
scripted_module = torch.jit.script(model)
# Export full jit version model (not compatible mobile interpreter), leave it here
# for comparison scripted_module.save("deeplabv3_scripted.pt")
# Export mobile interpreter version model (compatible with mobile interpreter)
optimized_scripted_module = optimize_for_mobile(scripted_module)
optimized_scripted_module._save_for_lite_interpreter("deeplabv3_scripted.ptl")
```

N.B.: i pacchetti PyThon necessari, ovvero torch, os e numpy, possono essere installati utilizzando il comando

```
pip install packagename
```

Nel codice sopra riportato, notiamo innanzitutto il caricamento del modello preallenato DeepLabv3 mediante l'utilizzo di una ResNet50 (più dettagli nella Sezione 1.5.2), che viene messo in modalità ”eval” (di cui abbiamo già parlato). Successivamente viene usata la funzione

”torch.jit.script() per compilare il codice sorgente del modello come codice TorchScript e ritornare uno ScriptModule o ScriptFunction; questo serve a salvare il modello su disco e caricarlo in un altro ambiente, magari utilizzando anche un altro linguaggio di programmazione [35]. In alternativa all'utilizzo di *script*, può essere usata la funzione *torch.jit.trace()*, come già visto nella Sezione 1.4.2. L'ottimizzazione è già stata trattata nella Sezione 1.4.3, mentre *save_for_lite_interpreter* serve, per l'appunto, a salvare il modello in formato ”.ptl”, ovvero compatibile con il lite interpreter.

2.2 Segmentazione delle immagini

Dopo aver aperto l'immagine a cui applicare la segmentazione utilizzando

```
input_image = Image.open("nome_img.jpg")
```

è necessario preprocessarla e normalizzarla. La parte di **preprocessing** [31] prevede un facoltativo passaggio di ridimensionamento e di ”centercrop” (ovvero l'accentramento dell'immagine in modo da ”occupare” interamente entrambe le dimensioni, vedi un esempio in Figura 2.1) seguito dalla conversione dell'immagine in un Tensor (che convertirà i valori da un intervallo di [0, 255] ad uno di [0, 1], per poi infine applicare la normalizzazione, utilizzando i valori specifici di Imagenet ”mean = [0.485, 0.456, 0.406]” e ”std = [0.229, 0.224, 0.225]”. Dopo aver applicato il preprocessing all'immagine, questa passa per l'unsqueeze(), una funzione che permette di passare da un'immagine di dimensione [1 x C x H x W] ad una di dimensione [C x H x W] (in cui C è il numero di canali, H l'altezza dell'immagine e W la larghezza. Il valore ”1” invece è la batch size [3], ovvero il numero di campioni processati prima che il modello aggiorni i parametri interni (la batch size ci serviva mentre passavamo l'immagine al modello, quindi ora possiamo rimuoverla).

Figura 2.1: Immagine ”centered” (sinistra) e immagine ”centerCropped” (destra). Da [5]

Analizziamo ora, quindi, la funzione ”segment(model, path)”, creata ad hoc per la segmentazione:

```
def segment(model, path):#,
show_orig=True):
```

```
img = Image.open(path)
# Comment the Resize and CenterCrop for better inference results
  preprocess = transforms.Compose([
                  #transforms.Resize(640),
                  #transforms.CenterCrop(224),
                transforms.ToTensor(),
                transforms.Normalize(mean = [0.485, 0.456, 0.406],
                                     std = [0.229, 0.224, 0.225])])
  input_tensor = preprocess(img)
  input_bach = input_tensor.unsqueeze(0)

  # move the input and model to GPU for speed if available
  if torch.cuda.is_available():
    input_batch = input_batch.to('cuda')
    model.to('cuda')

  with torch.no_grad():
    output = model(input_batch)['out'][0]

  om = torch.argmax(output, dim=0).squeeze().detach().cpu().numpy()

  return om
```

Nella seconda parte del codice, invece, per prima cosa (se "cuda" è disponibile) spostiamo gli input e il modello nella GPU per velocizzare il processo, dopodiché passiamo l'immagine preprocessata al modello, il quale ritornerà un oggetto di tipo OrderedDict, e per ottenerne l'output è necessario usare la chiave "out". Dopo aver fatto ciò noteremo, stampando la dimensione dell'output del modello, che il numero di canali dell'immagine è 21. Ciò significa che il modello è stato allenato su 21 classi.
Ottenuto l'output, dobbiamo trasformarlo in un'immagine 2D con 1 solo canale, in cui ogni pixel corrisponde ad una classe. Di fatto, bisogna associare ad ogni pixel un valore, che deve corrispondere ad uno dei valori dei 21 pixel dei rispettivi canali alla stessa posizione di quello in output. Ciò viene risolto prendendo l'indice massimo per ogni posizione dei pixel, utilizzando "argmax()".
Per applicare la funzione all'immagine desiderata (e utilizzando il modello che ci serve) basti fare

```
output_predictions = segment(model, input_image_path)
```

in cui "model" è il modello utilizzato e "input_image_path" è il path dell'immagine da utilizzare.
Abbiamo applicato il modello all'immagine e ottenuto una nuova immagine, ovvero una segmentazione a partire dall'immagine di partenza. Possiamo applicare questa funzione per tutte le

immagini del database e per i diversi modelli generati in precedenza. Possiamo poi passare al confronto prestazionale usando l'IoU e Dice.

2.3 Intersection over Union (IoU) e Dice Score

Per verificare le prestazioni del modello utilizzato per la segmentazioni e per valutare le differenze fra modello non-ottimizzato e quello ottimizzato, sono stati usati due metriche: l'**Intersection over Union** (**IoU**) e il **Dice Score**.
L'**IoU** [14] viene utilizzato come metro di giudizio quando si vogliono "sovrapporre" due maschere e si vuole valutare la loro somiglianza, calcolando uno score che si sostanzia nel dividere (come dice il nome stesso) l'intersezione per l'unione delle suddette maschere. La formula per il calcolo dell'IoU è la seguente:

$$IoU(P,T) = \frac{|P \cap T|}{|P \cup T|}$$

dove P rappresenta la segmentation maschera predetta, mentre T è la ground truth mask. Se la predizione è totalmente corretta, allora lo score sarà uguale a 1, mentre questo diminuirà in base all'imprecisione tra la predizione e la ground truth.
Il **Dice Score** [14], noto anche col nome di Dice Similarity Coefficient, misura la sovrapposizione tra la maschera predetta e la ground truth. Similmente all'IoU, il valore massimo è 1 (il quale corrisponde ad una uguaglianza totale), mentre il valore scende tanto più le due maschere sono diverse. Il Dice è definito come:

$$Dice(P,T) = \frac{2|P \cap T|}{|P| + |T|}$$

Nella Figura 2.2 sono riportate le rappresentazioni grafiche dei due coefficienti.

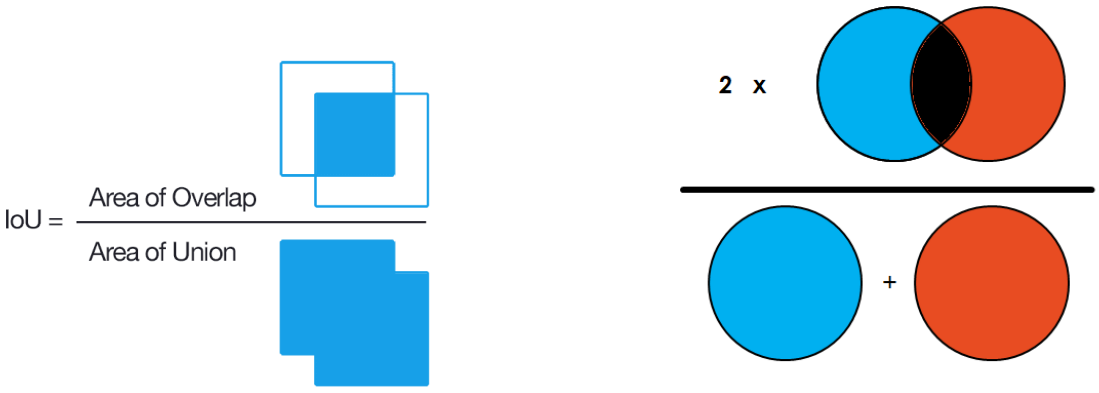


Figura 2.2: Rappresentazione grafica IoU (sinistra) e Dice (destra). Da [15] e [10]

Nel codice utilizzato per la segmentazione, vengono implementate le due funzioni atte a calcolare l'IoU e il Dice [16]:

```
def get_iou(pred, gt, beta=1):
    y_pred_bool = pred.astype(bool)
    y_true_bool = gt.astype(bool)
    tp = np.logical_and(y_true_bool, y_pred_bool).sum()
    tn = np.logical_and(~y_true_bool, ~y_pred_bool).sum()
    fp = np.logical_and(~y_true_bool, y_pred_bool).sum()
    fn = np.logical_and(y_true_bool, ~y_pred_bool).sum()
    iou = tp / (tp + fn + fp)

    return iou

def get_dice(pred, gt):
    y_pred_bool = pred.astype(bool)
    y_true_bool = gt.astype(bool)

    tp = np.logical_and(y_true_bool, y_pred_bool).sum()
    tn = np.logical_and(~y_true_bool, ~y_pred_bool).sum()
    fp = np.logical_and(~y_true_bool, y_pred_bool).sum()
    fn = np.logical_and(y_true_bool, ~y_pred_bool).sum()

    dice = 2*tp / (2*tp + fn + fp)

    return dice
```

Prima di svolgere effettivamente il calcolo sopra riportato, vengono presi gli array bidimensionali ”pred” e ”gt” (rispettivamente le maschere predetta e ground truth) e convertiti in boolean. Dopodiché si può passare alla realizzazione della formula.

2.4 Lettura delle immagini

Per leggere la maschera (che rappresenta la ground truth per la segmentazione dell'immagine correlata) e mostrare a schermo le immagini segmentate vengono utilizzate le seguenti funzioni [16]. La prima:

```
def read_bmask(path:str) -> np.ndarray:
    return cv2.imread(path, cv2.IMREAD_GRAYSCALE) / 255.0
```

serve a leggere la ground truth mask (la funzione considera il fatto che la maschera sia in scala di grigi, quindi con valori che vanno da 0 a 1 (ecco spiegata la divisione per 255)); usiamo poi la funzione:

```
def show_mask(mask: np.ndarray, ax, random_color:bool=False):
if random_color:
    color = np.concatenate([np.random.random(3), np.array([0.6])], axis=0)
else:
    color = np.array([30/255, 144/255, 255/255, 0.6])
h, w = mask.shape[-2:]
mask_image = mask.reshape(h, w, 1) * color.reshape(1, 1, -1)
ax.imshow(mask_image)
```

che serve a mostrare a schermo la maschera desiderata. Per mostrare l'immagine presa in considerazione per la segmentazione, invece, basta utilizzare

```
img = Image.open(input_image_path)
plt.imshow(img); plt.show()
```

Capitolo 3

PyTorch Mobile in Android

Per questo documento è stato utilizzato Android [20] come sistema operativo nel quale utilizzare i modelli di PyTorch. In particolar modo, si è voluto confrontare qualitativamente le prestazioni di un modello PyTorch nativo con uno quantizzato e trattato per l'uso mobile (ergo, PyTorch Mobile).

3.1 Gradle dependencies

Per utilizzare un modello PyTorch utilizzando Android Studio, è necessario aggiungere le seguenti al file build.gradle:

```
    repositories {
    jcenter()
}

dependencies {
    implementation 'org.pytorch:pytorch_android_lite:1.12.2'
    implementation 'org.pytorch:pytorch_android_torchvision_lite:1.12.2'
}
```

Chiaramente il numero della versione potrebbe variare; in effetti, le versioni nella repository GitHub presente nella pagina ufficiale di PyTorch Mobile presenta una versione troppo vecchia, che probabilmente entrava in conflitto con la versione corrente del modello utilizzato.

3.2 Caricamento del modulo e preparazione degli input

Per caricare il modello nell'applicazione Android, viene utilizzato

```
mModule = LiteModuleLoader.load(MainActivity.assetFilePath(
getApplicationContext(),"deeplabv3_scripted_optimized.ptl"));
```

Per quanto riguarda invece la preparazione degli input, viene utilizzato

```
final Tensor inputTensor = TensorImageUtils.bitmapToFloat32Tensor(mBitmap,
                        TensorImageUtils.TORCHVISION_NORM_MEAN_RGB,
                        TensorImageUtils.TORCHVISION_NORM_STD_RGB);
```

Il metodo *TensorImageUtils.bitmapToFloat32Tensor* crea tensori in un formato supportato da *torchvision* usando, come sorgente, *android.graphics.Bitmap*.
N.B.: quasi tutti i modelli pre-allenati sono normalizzati alla stessa maniera (ad esempio, in immagini RGB di forma (3 * H * W), dove H e W sono almeno 224). L'immmagine deve essere caricata nel range [0, 1] e poi normalizzata usando *mean = [0.485, 0.456, 0.406]* e *std = [0.229, 0.224, 0.225]*, come già mostrato nel Capitolo 2. "inputTensor" ha una forma di 1*3*H*W, dove H e W corrispondono all'altezza e alla larghezza del bitmap. I file **bitmap** [4] sono immagini lossless e a cui,quindi, non vengono applicate compressioni. Sono molto utilizzati per mantenere la risoluzione delle immagini digitali su schermi e dispositivi diversi.

3.3 Inferenza ed elaborazione dei risultati

Il prossimo passo è applicare il modello e ottenere la predizione della maschera dell'immagine considerata da quest'ultimo:

```
Tensor outputTensor = module.forward(IValue.from(inputTensor)).toTensor();
float[] scores = outputTensor.getDataAsFloatArray();
```

Il metodo *org.pytorch.Module.forward* esegue il metodo "forward" del modello caricato e i risultati vengono salvati nel Tensor *org.pytorch.Tensor*. Successivamente, con l'ausilio di *org.pytorch.Tensor.getData* il contenuto viene preso e convertito in un array di float.
Dopodiché, troviamo lo score massimo (nello specifico, maxi si riferisce all'indice della classe di soggetti a cui il determinato pixel appartiene con la più alta probabilita):

```
int width = mBitmap.getWidth();
    int height = mBitmap.getHeight();
    int[] intValues = new int[width * height];
    for (int j = 0; j < height; j++) {
        for (int k = 0; k < width; k++) {
            int maxi = 0, maxj = 0, maxk = 0;
            double maxnum = -Double.MAX_VALUE;
            for (int i = 0; i < CLASSNUM; i++) {
                float score = scores[i * (width * height) + j * width + k];
                if (score > maxnum) {
                    maxnum = score;
```

```
                    maxi = i; maxj = j; maxk = k;
                }
            }
        }
```

Nel codice di cui sopra [9], **maxi** è il valore intero che identifica il soggetto segmentato (essere umano, pecora, cane, etc) e può essere usato per distinguere i vari soggetti dell'immagine, per esempio così:

```
if (maxi == PERSON)
    intValues[maxj * width + maxk] = 0xFFFF0000;
else if (maxi == DOG)
    intValues[maxj * width + maxk] = 0xFF00FF00;
else if (maxi == SHEEP)
    intValues[maxj * width + maxk] = 0xFF0000FF;
else
    intValues[maxj * width + maxk] = 0xFF000000;
```

N.B.: l'ultimo "else" distinguerà qualsiasi cosa non sia una persona, un cane o una pecora dal resto, assegnandole un colore diverso. Le costanti sono definite all'inizio della MainActivity:

```
private static final int CLASSNUM = 21;
private static final int DOG = 12;
private static final int PERSON = 15;
private static final int SHEEP = 17;
```

I valori sono definiti dal caso particolare dell'aver utilizzato un modello DeepLabv3: questo infatti da in output un tensore di dimensione [21, larghezza, altezza] data in input un'immagine di dimensioni "larghezza*altezza". Ogni valore dell'array di output sopracitato è compreso fra 0 e 20 (per un totale di 21 label semantiche citate poc'anzi) e il valore viene quindi sfruttato per diversificare le differenti classi semantiche. In effetti, nel nostro caso sarebbe necessario solamente un valore: il valore 15 (che rappresenta la classe "persona"), mentre tutti gli altri potrebbero essere unificati in un unico colore, avendo a che fare con un database contenente solo volti umani; nell'esempio considerato in [20] l'immagine di prova contiene, oltre che un pastore (quindi una persona), anche delle pecore e un cane, perciò era necessario distinguere il tutto.

3.4 UI dell'app

È possibile mostrare l'immagine a schermo mediante una ImageView, nonché utilizzare dei Button per la gestione delle immagini e il salvataggio in memoria dell'immagine segmentata.

In particolare, per mostrare anche l'immagine segmentata (oltre all'immagine di partenza) mediante l'ImageView, è necessario il seguente pezzo di codice:

```
Bitmap bmpSegmentation = Bitmap.createScaledBitmap(bitmap, width, height, true);
Bitmap outputBitmap = bmpSegmentation.copy(bmpSegmentation.getConfig(), true);
outputBitmap.setPixels(intValues, 0, outputBitmap.getWidth(), 0, 0,
            outputBitmap.getWidth(), outputBitmap.getHeight());
imageView.setImageBitmap(outputBitmap);
```

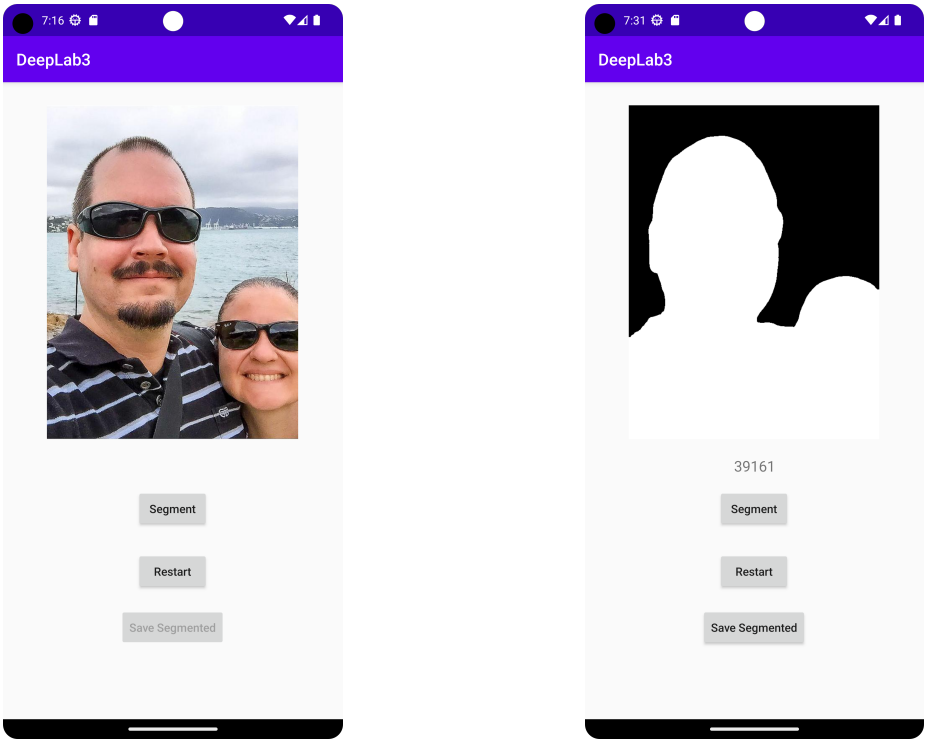


Figura 3.1: A sinistra la home dell'applicazione con l'immagine di partenza, a destra con l'immagine segmentata

Nel caso si volesse **salvare l'immagine** segmentata in memoria, è stato aggiunto un Button che si occupa di fare ciò (inoltre, questo è cliccabile soltanto mentre l'ImageView mostra effettivamente l'immagine segmentata, mentre si disattiva quando quest'ultima mostra l'immagine di input). Per salvare l'immagine, è necessario specificare il permesso nel manifest:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Il codice che permette di salvare l'immagine è il seguente:

```
ImageView imageView = findViewById(R.id.imageView);
BitmapDrawable drawable = (BitmapDrawable) imageView.getDrawable();
Bitmap bitmap = drawable.getBitmap();
```

```
// For Android 10 and above
ContentValues values = new ContentValues();
values.put(MediaStore.Images.Media.DISPLAY_NAME, "image_name");
values.put(MediaStore.Images.Media.MIME_TYPE, "image/png");
Uri uri = getContentResolver().insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
                                      values);

try {
    OutputStream outputStream = getContentResolver().openOutputStream(uri);
    bitmap.compress(Bitmap.CompressFormat.PNG, 100, outputStream);
    outputStream.close();
    // Notify the gallery about the new image
    Intent mediaScanIntent = new Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE);
    mediaScanIntent.setData(uri);
    sendBroadcast(mediaScanIntent);
} catch (IOException e) {
    e.printStackTrace();
    Toast.makeText(this, "Failed to save image", Toast.LENGTH_SHORT).show();
    }
```

Il codice di cui sopra viene inserito nel setOnClickListener del Button correlato (nel nostro caso, il terzo).
Inoltre, per misurare le differenze prestazionali dei vari modelli, è stata aggiunta una porzione di codice che misura il tempo impiegato dal modello in questione ad applicare la segmentazione. A questo scopo, è stato utilizzato il clock *elapsedRealTime()* [1], che ritorna il tempo (in millisecondi) da quando il sistema è stato avviato, includendo i deep sleep. Questo clock è stato preferito agli altri (uptimeMillis() e System.currentTimeMillis()) in quanto elapsedRealTime continua sempre a ”tenere il tempo”, perciò ci permette di avere una stima del tempo utilizzato dal modello leggermente più veritiera. La misura del tempo impiegato per la segmentazione, nell’app Android, compare subito sotto la textView che mostra l’immagine segmentata (viene mostrata, chiaramente, solamente dopo aver effettuato la segmentazione) e lo si può notare dalla Figura 3.1. Il codice utilizzato è stato posto all’interno della funzione ”run”:

```
    long inferenceTime = 0;

    @Override
    public void run() {
        final long startTime = SystemClock.elapsedRealtime();

        [...]
```

```
        inferenceTime = SystemClock.elapsedRealtime() - startTime;
```

Per mostrare il valore a schermo, è stata usata una textView a cui è stata passata la variabile ”inferenceTime”.

3.5 Interprete mobile efficiente

N.B.: questa funzione, al momento della pubblicazione, è ancora in fase beta, come altre funzioni di PyTorch.
Nel progetto, è stato utilizzato l'interprete efficiente di PyTorch. Sia in Android che in iOS è possibile usufruirne (nel nostro caso utilizzando delle librerie pre-built). Per utilizzarlo, è necessario innanzitutto aggiornare il *build.gradle* del nostro progetto:

```
    repositories {
    maven {
        url "https://oss.sonatype.org/content/repositories/snapshots"
    }
}

dependencies {
    implementation 'androidx.appcompat:appcompat:1.2.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.0.2'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'
    implementation 'org.pytorch:pytorch_android_lite:1.9.0'
    implementation 'org.pytorch:pytorch_android_torchvision:1.9.0'

    implementation 'com.facebook.fbjni:fbjni-java-only:0.0.3'
}
```

Successivamente, è necessario sostituire

```
mModule = Module.load(MainActivity.assetFilePath(getApplicationContext(),
                "deeplabv3_scripted.pt"));
```

con

```
mModule = LiteModuleLoader.load(MainActivity.assetFilePath(getApplicationContext(),
                "deeplabv3_scripted.ptl"));
```

Capitolo 4

Esperimenti

Dopo aver chiarito i concetti base riguardanti PyTorch e PyTorch Mobile e dopo aver presentato il codice necessario alla segmentazione delle immagini nel dataset prescelto, arriviamo al cardine della questione: mostrare (con l'utilizzo di metriche ogettive quali IoU e il Dice) le differenze prestazionali e nel tempo di esecuzione tra modelli PyTorch e modelli quantizzati e ottimizzati per dispositivi mobili PyTorch Mobile. Nota: al seguito, si farà riferimento con modello "scripted" al modello base a cui è stato solamente applicato il metodo *torch.jit.script()*, mentre con modello "optimized" si farà riferimento al modello scripted a cui è stato applicato il metodo *torch.utils.mobile_optimizer.optimize_for_mobile*.

Per avere una visione d'insieme degli effetti generati da ogni passaggio effettuato, è necessario prendere in esame 6 modelli da ambiente Python e 4 da ambiente Android. In particolare, in ambiente Python consideriamo:

- modello di partenza
- modello scripted
- modello quantized
- modello scripted_optimized
- modello quantized_scripted
- modello quantized_scripted_optimized

Invece, in ambiente Android:

- modello scripted
- modello scripted_optimized
- modello quantized_scripted
- modello quantized_scripted_optimized

4.1 Differenze prestazionali

I risultati trovati coi modelli su PC (ambiente Python) sono molto simili e di seguito ne indico le medie (calcolate sui parametri IoU e Dice di ogni immagine del database sopra citato):

```
IoU normale: 0.9479431220764858
Dice normale: 0.9722676375047619

IoU scripted quantized: 0.9479431220764858
Dice scripted quantized: 0.9722676375047619

IoU scripted not quantized: 0.9479431220764858
Dice scripted not quantized: 0.9722676375047619

IoU quantized: 0.9479431220764858
Dice quantized: 0.9722676375047619

IoU scripted optimized quantized: 0.9479430788844698
Dice scripted optimized quantized: 0.9722676120317809

IoU scripted optimized not quantized: 0.9479430788844698
Dice scripted optimized not quantized: 0.9722676120317809
```

Come già detto, la differenza risulta essere minima e condizionata solo da alcune singole immagini del database piuttosto che dalla maggior parte di queste. Basti infatti confrontare i valori trovati con la segmentazione per accorgersi che le immagini responsabili della diversificazione dei valori trovati sono poche (circa 5 o 6). Inoltre, le differenze tra le maschere in questione sono impercettibili e lo si nota dalla piccola differenza fra gli score; un esempio riguarda l'immagine numero 41 del database, i cui score del modello scripted,optimized,quantized sono IoU = 0.7430177861842028 e Dice = 0.8525647782525615, mentre quelli del modello scripted sono IoU = 0.7430195363474049 e Dice = 0.8525659303904809 (le variazioni iniziano alla sesta cifra dopo la virgola).

Per quanto riguarda, invece, i risultati trovati coi modelli su Android, sono i seguenti:

```
IoU scripted quantized: 0.9360477994484384
Dice scripted quantized: 0.9655800831027452

IoU scripted not quantized: 0.9346661879719783
Dice scripted not quantized: 0.9643451447595628
```

```
IoU scripted optimized quantized: 0.9525977620132241
Dice scripted optimized quantized: 0.974832603956036

IoU scripted optimized not quantized: 0.9489895752411988
Dice scripted optimized not quantized: 0.9711772825981562
```

Come si può facilmente notare, in ambiente Android sia la quantizzazione che l'ottimizzazione contribuiscono ad aumentare la qualità delle maschere generate, nonostante le differenze non siano esorbitanti. Inoltre, i modelli non ottimizzati presentano in generale problemi di esecuzione (ad esempio maschere generate malamente e tempi di esecuzione molto elevati) e causano talvolta l'interruzione dell'app. Nella Figura 4.1 vediamo un esempio di maschere generate in Android da due modelli diversi (più precisamente, uno ottimizzato per mobile e l'altro no), evidenziando le differenze che queste presentano.

Figura 4.1: A partire da sinistra: l'immagine originale, la maschera generata dal modello ottimizzato e quella generata dal modello non ottimizzato.

Per quanto riguarda le differenze fra le maschere generate in Android e da PC con lo stesso modello, non notiamo differenze sostanziali, eccezion fatta per quelle generate dal modello non ottimizzato per mobile (che applicato in ambiente Android, come già visto, genera delle maschere meno precise in molti casi).

4.2 Differenze nel tempo di esecuzione

L'utilizzo dell'app in ambiente Android ci permette di confrontare le prestazioni temporali dei modelli utilizzati (ergo, modello DeepLabv3 con ResNet50 "base" e la versione "ottimizzata" per mobile e quantizzata (o meno)). Chiaramente, più l'immagine utilizzata è grande, più i tempi necessari alla segmentazione risulteranno lunghi (non dimentichiamo che la segmentazione che vogliamo utilizzare, ovvero quella semantica, vuole assegnare ad ogni pixel una label, perciò più

pixel ci sono e più il processo sarà lungo). Le segmentazioni effettuate con il modello ”scripted” (perciò il modello normale trattato per l'utilizzo con lite interpreter) e ”optimized” (ovvero il modello (ottenuto a partire dallo scripted) ottimizzato per prestazioni migliori su mobile) mostrano una differenza temporale decisamente netta: si parla, nella maggior parte dei casi, di più del doppio (fino a toccare alle volte anche il triplo) di tempo impiegato dal modello ”scripted” rispetto al modello ottimizzato per mobile. Come si vede nella Figura 4.2, il rapporto tra i due tempi è di circa 1 a 2. Chiaramente questa non è una regola scritta: in effetti, ad ogni iterazione il tempo impiegato per la segmentazione **può variare** a causa di vari fattori, sia per un modello che per l'altro; ciò che rimane, tuttavia, è questa differenza netta nelle tempistiche (anch'essa variabile da campione a campione e da un'iterazione all'altra).

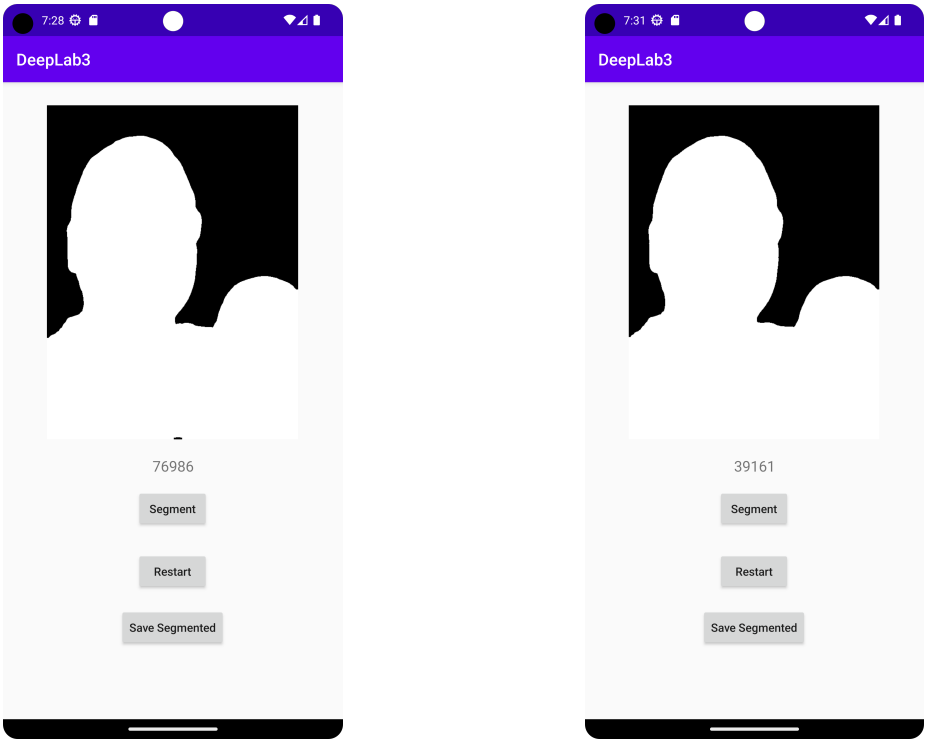


Figura 4.2: Differenza temporale utilizzando il modello ”scripted”(a sx) e ”optimized” (a dx)

4.3 Differenze di score (IoU e Dice)

In generale, gli score ottenuti utilizzando il database mostrato in precedenza (Sezione 1.6) superano quasi tutti il valore 0.90, per cui effettivamente il modello utilizzato è abbastanza preciso nella segmentazione di ”portraits”.
Per parlare ulteriormente degli score prodotti dalle maschere ottenuta dal processo di segmentazione, tuttavia, è necessario distinguere le maschere provenienti da ambiente Android e quelle provenienti dal codice Python. In effetti, quello che si nota è che in ambiente Python gli score ottenuti

utilizzando sia il modello normale che quello ottimizzato per mobile sono identici, nonostante le varie ottimizzazioni applicate, mentre in ambiente Android, effettivamente (e come già mostrato), gli score appaiono differenti usando i due modelli (sia in ambito temporale che in ambito prestazionale). Quello che si nota usando i modelli in Android è che gli score ottenuti utilizzando il modello ”non ottimizzato” sono identici a quelli ottenuti in Python; al contrario, gli score ottenuti in Android usando il modello ottimizzato appaiono migliori rispetto a quelli ottenuti in Python e, quindi, anche a quelli ottenuti col modello non ottimizzato. Quello che si può dedurre è che il modello non ottimizzato (.ptl, perciò compatibile con il lite interpreter) funzioni peggio in Android, generando maschere più imprecise e impiegando più tempo per l'esecuzione. Al contrario, il modello ottimizzato è più veloce e funziona meglio, creando maschere più precise.

Capitolo 5

Conclusioni

Dopo una doverosa parte di approfondimento teorico circa la segmentazione e il suo funzionamento, ci siamo cimentati in una serie di esperimenti pratici che ci hanno permesso di comprendere (almeno in parte) come funziona nella pratica la segmentazione e cosa possiamo applicare ad un normale modello per permettere il suo funzionamento in un dispositivo mobile. Arrivati al termine, abbiam o quindi raggiunto il nostro obiettivo. Chiaramente, come sottolineato varie volte, c'è stata la volontà di concentrarsi sulla segmentazione nonostante ci siano numerosi altri task di machine learning che potrebbero servire in un dispositivo mobile e che sono utilizzabili mediante l'utilizzo di PyTorch Mobile; per questo motivo, è possibile usare il seguente documento come punto di partenza per l'utilizzo di altri task utili (con le dovute modifiche e considerazioni da applicare).

Sitografia

[1] *Android SystemClock*. URL: https://developer.android.com/reference/android/os/SystemClock.

[2] *App Usability*. URL: https://www.interaction-design.org/literature/article/it-ain-t-what-you-do-it-s-the-way-that-you-do-it-mobile-app-usability-best-practices.

[3] *Batch size*. URL: https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/#:~:text=The%20batch%20size%20is%20a%20number%20of%20samples%20processed%20before,samples%20in%20the%20training%20dataset..

[4] *Bitmap*. URL: https://www.adobe.com/it/creativecloud/file-types/image/raster/bmp-file.html#i-vantaggi-dei-file-bmp.

[5] *Center vs CenterCrop*. URL: https://abhiandroid.com/ui/scaletype-imageview-example.html.

[6] *CUDA*. URL: https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/.

[7] *DeepLabv3*. URL: https://paperswithcode.com/method/deeplabv3#:~:text=DeepLabv3%20is%20a%20semantic%20segmentation,by%20adopting%20multiple%20atrous%20rates..

[8] *DeepLabv3 2*. URL: https://learnopencv.com/deeplabv3-ultimate-guide/.

[9] *DeepLabv3 in Android*. URL: https://pytorch.org/tutorials/beginner/deeplabv3_on_android.html.

[10] *Dice img*. URL: https://www.kaggle.com/code/yerramvarun/understanding-dice-coefficient.

[11] *Dummy input*. URL: https://www.webopedia.com/definitions/dummy/.

[12] *FOV: definizione*. URL: https://www.princetoninstruments.com/learn/camera-fundamentals/field-of-view-and-angular-field-of-view#:~:text=Field%20of%20view%20(FOV)%20is,lens%20and%20the%20sensor%20size..

[13] *Image Segmentation*. URL: https://huggingface.co/tasks/image-segmentation.

[14] *Iou Dice*. URL: https://www.researchgate.net/publication/372660191_Improving_Existing_Segmentators_Performance_with_Zero-Shot_Segmentators#pf13.

[15] *Iou img*. URL: https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/.

[16] *IoU, Dice and view img/mask Functions*. URL: https://github.com/LorisNanni/Improving-existing-segmentators-performance-with-zero-shot-segmentators/tree/main.

[17] *Kim, Y.W.; Byun, Y.C.; Krishna, A.V.N. Portrait Segmentation Using Ensemble of Heterogeneous*
Deep-Learning Models. Entropy 2021, 23. URL: https://doi.org/10.3390/e23020197.

[18] *Metal*. URL: https://pytorch.org/blog/introducing-accelerated-pytorch-training-on-mac/.

[19] *Ottimizzazione Modello*. URL: https://pytorch.org/docs/stable/mobile_optimizer.html.

[20] *PyTorch in Android*. URL: https://pytorch.org/mobile/android/.

[21] *PyTorch Mobile*. URL: https://pytorch.org/mobile/home/.

[22] *PyTorch Mobile vs TensorFlow Lite*. URL: https://www.kdnuggets.com/2021/11/on-device-deep-learning-pytorch-mobile-tensorflow-lite.html.

[23] *PyTorch to TorchScript: tracing*. URL: https://coremltools.readme.io/docs/model-tracing.

[24] *QNNPACK*. URL: https://github.com/pytorch/QNNPACK.

[25] *Quantizzazione*. URL: https://pytorch.org/docs/stable/quantization.html.

[26] *ResNet-50*. URL: https://datagen.tech/guides/computer-vision/resnet-50/.

[27] *Rethinking Atrous Convolution for Semantic Image Segmentation*. URL: https://arxiv.org/pdf/1706.05587v3.pdf.

[28] *Segmentation*. URL: https://learnopencv.com/image-segmentation/.

[29] *Segmentazione semantica: come funziona*. URL: https://www.jeremyjordan.me/semantic-segmentation/.

[30] *Semantic Segmentation*. URL: https://www.mathworks.com/solutions/image-video-processing/semantic-segmentation.html#:~:text=What%20Is%20Semantic%20Segmentation%3F,pixels%20that%20form%20distinct%20categories..

[31] *Semantic Segmentation using TorchVision*. URL: https://learnopencv.com/pytorch-for-beginners-semantic-segmentation-using-torchvision/.

[32] *Torchvision*. URL: https://medium.com/swlh/understanding-torchvision-functionalities-for-pytorch-391273299dc9.

[33] *Vulkan*. URL: https://pytorch.org/tutorials/prototype/vulkan_workflow.html.

[34] *What is PyTorch Mobile*. URL: https://analyticsindiamag.com/what-is-pytorch-mobile/.

[35] *Why TorchScript*. URL: https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html#:~:text=This%20format%20allows%20us%20to,to%20provide%20more%20efficient%20execution.

[36] *XNNPACK*. URL: https://blog.tensorflow.org/2020/07/accelerating-tensorflow-lite-xnnpack-integration.html.