ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Scuola di Ingegneria e Architettura

Corso di Laurea Magistrale in Ingegneria Informatica

Attività Progettuale

in

Mobile Systems M

TensorFlow Lite: analisi e integrazione con dispositivi MQTT virtuali in Android

Simone Pozza

Tutor Accademico

Professor Paolo Bellavista

Anno Accademico 2019/2020

Indice

1 Introduzione

1.1 TensorFlow: machine learning sul cloud Google

TensorFlow è una piattaforma per il machine learning sviluppata dal team Google Brain per uso interno e rilasciato al pubblico come software open source (sotto la licenza Apache 2.0) il 9 novembre 2015. TensorFlow ha raggiunto la versione 1.0 nel febbraio 2017 e la versione 2.0, su cui questo progetto si concentrerà, il 30 settembre 2019.

Il gruppo di ricerca Google Brain fu formato nel 2011 come una collaborazione fra due ricercatori di Google, Jeff Dean e Greg Corrado, e un professore dell'Università di Stanford, Andrew Ng. Da questa collaborazione nacque un sistema software di deep learning su larga scala basato sull'infrastruttura per il cloud computing di Google: DistBelief. DistBelief era un framework che poteva utilizzare cluster composti da migliaia di computer per il training di reti neurali di grandi dimensioni. In questo contesto, Google ha sviluppato anche due algoritmi per il training distribuito su larga scala, Downpour SGD e Sandblaster L-BFGS, i quali aumentano la scalabilità e la velocità del training per il deep learning. DistBelief aveva però delle limitazioni: era strettamente mirato alle reti neurali, era difficile da configurare ed era strettamente accoppiato all'infrastruttura interna di Google, rendendo quasi impossibile condividere esternamente il codice di ricerca. TensorFlow è stato sviluppato per ovviare a tali carenze. TensorFlow è infatti flessibile, facile da usare e completamente open source. Inoltre è più veloce e presenta una migliore la scalabilità rispetto a DistBelief. TensorFlow è quindi il successore di questa infrastruttura per il machine learning e mentre DistBelief era orientato specificamente verso un unico tipo di software per il machine learning, le reti neurali, TensorFlow è stato progettato per poter accomodare diversi approcci.

Il nome TensorFlow deriva dal fatto che il framework consente di definire ed eseguire calcoli che coinvolgono tensori. In matematica, un tensore descrive un mapping lineare da un insieme di oggetti algebrici a un altro; un tensore è quindi una generalizzazione di vettori e matrici di dimensioni più elevate (si veda la figura 1). Internamente, TensorFlow rappresenta i tensori come array multidimensionali.

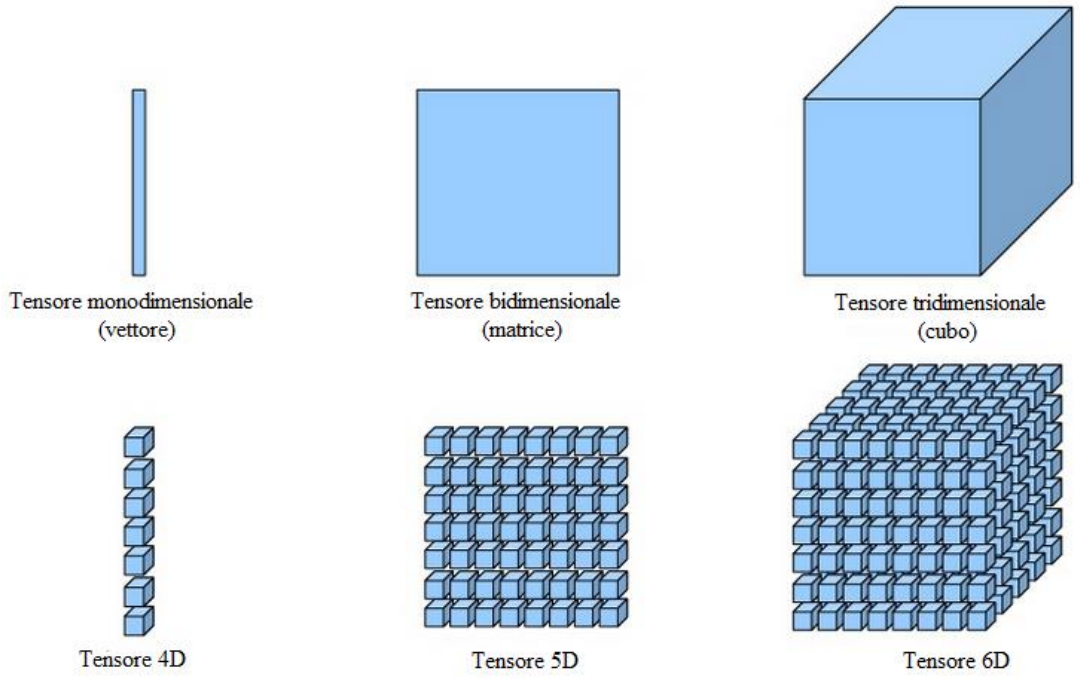

Figura 1- Visualizzazione di tensori di diverse dimensioni.

1.2 TensorFlow Lite: machine learning *at the edge*

Durante lo sviluppo di TensorFlow, Google Brain ha progettato una variante del framework che aveva l'obbiettivo di poter essere eseguito su sistemi eterogenei, inclusi i sistemi mobili, TensorFlow Mobile. Sfruttando la possibilità di eseguire i calcoli sui dispositivi sono stati mitigati i problemi dovuti allo scambio di dati bidirezionale fra dispositivi e data center. TensorFlow Mobile consentiva quindi agli sviluppatori di creare applicazioni interattive che facevano uso di computazioni machine learning, senza incorrere in ritardi dovuti a round-trip. [3]

Nel maggio 2017, Google annunciò uno stack software specifico per lo sviluppo mobile, TensorFlow Lite. TensorFlow Lite è l'evoluzione di TensorFlow Mobile (il quale già supportava il deployment su dispositivi mobili ed embedded) e funziona con una vasta gamma di dispositivi, dai piccoli microcontrollori agli smartphone. TensorFlow Lite consente di eseguire modelli di machine learning sui dispositivi appena citati, utilizzando un formato di modello basato su `FlatBuffers`, un'efficiente libreria di serializzazione inter-piattaforma disponibile in 11 linguaggi. [5]

In TensorFlow e TensorFlow Lite per modello si intende una struttura dati che contiene la logica e la conoscenza di una rete per il machine learning in grado di risolvere un particolare problema. I modelli di TensorFlow lite (file `.tflite`) sono simili ma più veloci e di dimensioni molto inferiori rispetto ai modelli di TensorFlow e ci si può aspettare che vengano

eseguiti con bassa latenza. [2] Un modello *trained* in TensorFlow può essere convertito nel formato TensorFlow Lite (come sarà discusso in dettaglio nella sezione 2.2.2).

TensorFlow Lite è quindi una soluzione *lightweight*, progettata per consentire il machine learning sui dispositivi ai margini (*edge*) della rete, senza ricevere e inviare dati da e verso un server. L'esecuzione di un modello di machine learning sul dispositivo può aiutare a migliorare:

- Tempi di risposta: non è presente la latenza dovuta alla comunicazione con un server.
- Privacy*:* nessun dato ha la necessità di lasciare il dispositivo.
- Consumi energetici: non è richiesta una connessione di rete. [1]

TensorFlow Lite presenta però due importanti limitazioni rispetto a TensorFlow: attualmente, TensorFlow Lite non supporta il training dei modelli *on-device* e supporta solo un sottoinsieme ristretto di operatori TensorFlow che sono stati ottimizzati per l'uso sui dispositivi mobili. Se il modello impiegato utilizza operatori TensorFlow non ancora supportati, è comunque possibile includerli nella build di TensorFlow Lite, tuttavia, questo porterà a un aumento della dimensione del file binario. Entrambe queste restrizioni saranno superate nelle versioni future di TensorFlow Lite: il training dei modelli *on-device* è presente nella roadmap dei miglioramenti previsti [7] e ogni nuova versione supporta sempre più operatori di TensorFlow, mantenendo la latenza bassa e le dimensioni delle applicazioni che fanno uso di modelli TensorFlow Lite ridotte.

1.2.1 Piattaforme alternative a TensorFlow Lite

Attualmente TensorFlow Lite non è l'unica piattaforma che consente di implementare machine learning su dispositivi mobili. Core ML è il framework di Apple che consente a sviluppatori di integrare machine learning nelle applicazioni mobili iOS e inoltre supporta la conversione e l'utilizzo di modelli machine learning classici creati con Scikit Learn nonché dei modelli basati su Keras. Facebook ha annunciato nel novembre 2016 la propria versione di TensorFlowLite chiamata Caffe2Go, una versione di Caffe2 ottimizzata per l'hardware dei dispositivi mobili. Caffe2 è, come TensorFlow, un framework per il machine learning ma mentre TensorFlow è stato progettato per essere utilizzato su server, Caffe2 è nato per essere impiegato su telefoni cellulari e altre piattaforme dalle risorse computazionali relativamente limitate.

2 Sviluppo di applicazioni basate su TensorFlow Lite

Come esposto nell'introduzione, TensorFlow Lite consente di eseguire inferenza su dispositivi mobili, embedded e IoT, con bassa latenza e modelli di ridotte dimensioni. In questo capitolo saranno approfonditi tutti gli aspetti di TensorFlow Lite relativi alla realizzazione di un'applicazione che ne fa uso.

2.1 Requisiti hardware e software

La documentazione ufficiale non specifica requisiti hardware minimi per i dispositivi Android e iOS o per i dispositivi embedded basati su Linux (e.g. Raspberry Pi) ma siccome il core del framework è implementato (per la maggior parte) in C++ è necessario che il dispositivo abbia un sistema operativo e le librerie standard C++.

2.1.1 TensorFlow Lite per microcontrollori

TensorFlow Lite for Microcontrollers è un port sperimentale di TensorFlow Lite progettato per eseguire modelli di machine learning su microcontrollori e altri dispositivi con pochi kilobyte di memoria. Non richiede il supporto di un sistema operativo, librerie standard C o C++ e nemmeno l'allocazione dinamica della memoria. Il core ha una dimensione di 16 KB su un microprocessore ARM Cortex M3, e con un numero sufficiente di operatori per eseguire un modello per il riconoscimento vocale, occupa un totale di 22 KB. [8]

2.2 Workflow

Per utilizzare un modello TensorFlow lite all'interno di applicazioni mobili è necessario seguire il flusso di lavoro descritto di seguito (e illustrato in figura 2):

1a) Costruire e fare il training di un modello TensorFlow.

1b) Scegliere un modello TensorFlow Lite preconfigurato e saltare lo step 2.

2) Convertire il modello TensorFlow originale in un modello TensorFlow Lite.

3) Effettuare il deployment del modello ed eseguire l'inferenza sul dispositivo.

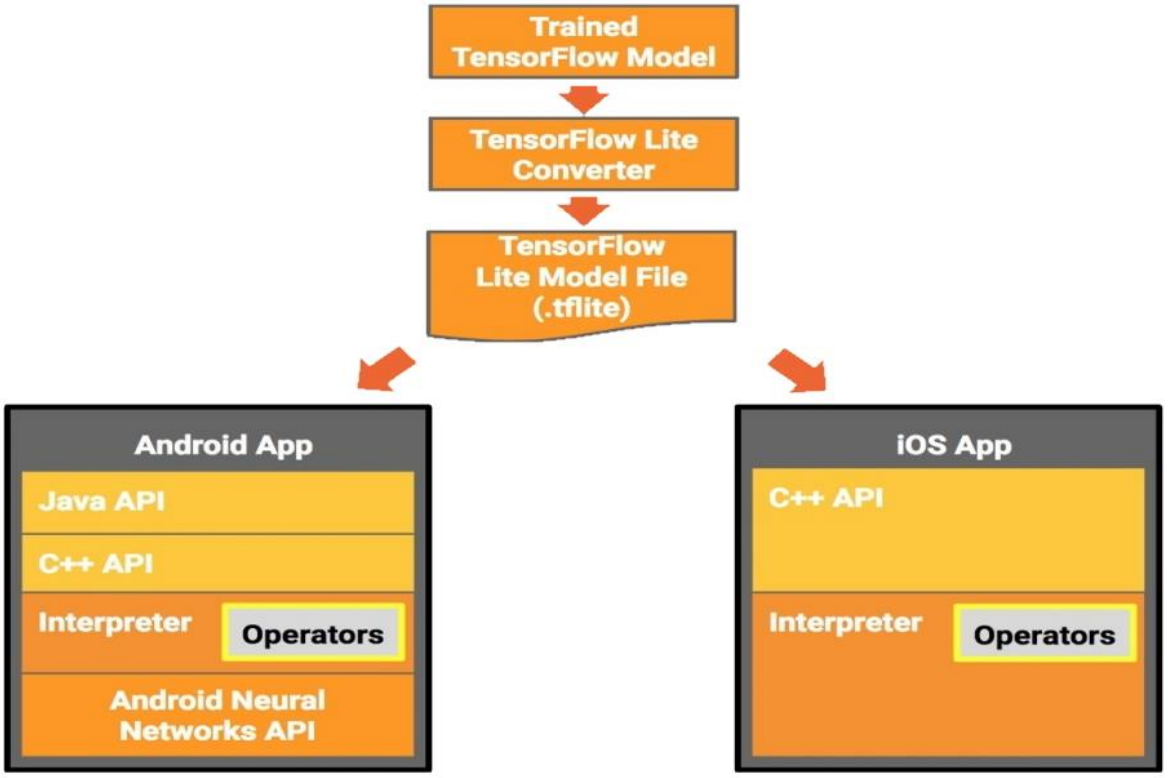

Figura 2- Workflow per l'utilizzo di un modello TensorFlow Lite in un'applicazione mobile.

2.2.1 Scelta del modello

È possibile utilizzare un proprio modello TensorFlow, uno trovato online o scegliere un modello TensorFlow Lite tra i modelli messi a disposizione sul sito ufficiale, i quali possono essere utilizzati direttamente (oppure *re-trained*). Se si sceglie di utilizzare un proprio modello TensorFlow costruito da zero, il training può essere fatto con TensorFlow oppure con Keras. Se, invece, si desidera utilizzare un modello TensorFlow Lite preconfigurato è possibile scegliere uno degli 8 modelli [9] ottimizzati per i casi d'uso più comuni ed eventualmente personalizzarlo in base alle finalità dell'applicazione:

1) *Image classification* (*MobileNet*), per identificare centinaia di classi di oggetti, tra cui persone, attività, animali, piante e luoghi.

2) *Object detection*, per rilevare più di 80 diverse classi di oggetti all'interno di un'immagine.
3) *Pose estimation* (*PoseNet*), utilizzato per stimare la postura di una persona in un'immagine o in un video, in base a dove si trovano le articolazioni del suo corpo.
4) *Smart reply*, per generare suggerimenti di messaggi di risposta basati sul contesto della chat.
5) *Segmentation* (*DeepLab*), per assegnare etichette semantiche (e.g. persona, cane, gatto) a ogni pixel dell'immagine di input. Ciò è differente da quanto avviene per i modelli di *Object detection*, che rileva gli oggetti in regioni rettangolari, e *Image classification*, che classifica l'immagine nel suo complesso.
6) *Artistic Style Transfer*, per creare nuove immagini, note come *pastiche*, basate su due immagini di input: una delle quali rappresenta lo stile e l'altra che rappresenta il contenuto a cui applicare lo stile.
7) *Text Classification*, per classificare un paragrafo in gruppi predefiniti (e.g. opinione positiva o negativa) in base al suo contenuto.
8) *Question and Answer*, per costruire un sistema in grado di rispondere alle domande degli utenti in linguaggio naturale.

Ciascuno dei file compressi, relativi ai modelli appena elencati, che è possibile scaricare contiene un modello TensorFlow Lite. Ad esempio, fra i tanti modelli per il riconoscimento di immagini, l'archivio `MobileNet_v1_1.0_224` contiene un file `mobilenet_v1_1.0_224.tflite` che è possibile utilizzare direttamente su un dispositivo mobile. [2] Se si decide di utilizzare un modello TensorFlow Lite preconfigurato, è possibile saltare i passaggi descritti nella sezione 2.2.2.

2.2.2 Conversione in un modello TensorFlow Lite

Se è stato progettato e *trained* un proprio modello TensorFlow, o è stato fatto il *training* di un modello ottenuto da un'altra fonte, è necessario convertirlo nel formato TensorFlow Lite. La conversione dei modelli riduce le dimensioni dei relativi file e introduce ottimizzazioni che non influiscono sulla precisione. Il convertitore TensorFlow Lite fornisce inoltre opzioni che permettono di ridurre ulteriormente le dimensioni del file e aumentare la velocità di esecuzione, con alcuni compromessi. Queste opzioni saranno discusse nella sezione 2.2.4.

Il convertitore TensorFlow Lite è uno strumento disponibile sotto forma di API Python (e alternativamente come strumento a riga di comando) che converte i modelli TensorFlow nel formato TensorFlow Lite. Consiste nella classe Python `tf.lite.TFLiteConverter`, che

fornisce i seguenti metodi per convertire rispettivamente modelli TensorFlow (directory `SavedModel`), modelli creati tramite l'API `tf.keras` e concrete functions:

- `TFLiteConverter.from_saved_model()`
- `TFLiteConverter.from_keras_model()`
- `TFLiteConverter.from_concrete_functions()`

`SavedModel` è il formato di serializzazione universale per i modelli TensorFlow e fornisce un modo indipendente dal linguaggio per salvare modelli di machine learning, consentendone il *recovery* e rendendoli ermetici. Grazie a questo formato strumenti di più alto livello (fra cui TensorFlow.js, TensorFlow Python e TensorFlow Lite) possono produrre, "consumare" e trasformare modelli TensorFlow. [10]
`tf.keras` è l'implementazione in TensorFlow della API specificata da Keras, una libreria per reti neurali open-source e scritta in Python compatibile con diversi framework per il machine learning (incluso proprio TensorFlow).
Una concrete function definisce un grafo che può essere convertito in un modello TensorFlow Lite o esportato in un modello `SavedModel`; consente quindi di salvare i modelli in forma di grafi, il che è indispensabile per eseguire l'inferenza in TensorFlow Lite 2.0.

La conversione inversa, da un modello `.tflite` a uno TensorFlow o Keras, non è attualmente supportata e non ne è nemmeno prevista l'implementazione. [14] Seppure le ottimizzazioni introdotte durante la conversione non abbiano effetti sulla precisione del modello, queste rendono il processo irreversibile. In particolare, la topologia del modello originale viene ottimizzata durante la conversione e ciò comporta una perdita di informazioni. Inoltre, per modelli `tf.keras` alcune configurazioni vengono completamente scartate poiché non sono richieste per l'inferenza. Tuttavia, analizzando il file `.tflite` con uno strumento come Netron [13] è possibile recuperare informazioni (quali i pesi) che possono aiutare a ripristinare il modello originale.

Come accennato nell'introduzione, l'interprete TensorFlow Lite attualmente supporta un sottoinsieme limitato degli operatori di TensorFlow; ciò significa che alcuni modelli richiedono ulteriori passaggi per essere compatibili con TensorFlow Lite. Se il modello utilizza operatori non ancora supportati dall'interprete TensorFlow Lite, è possibile includerli al costo di un aumento considerevole (fino a 30 volte!) della dimensione del file binario relativo all'interprete di TensorFlow Lite (descritto nella sezione seguente). Per convertire un modello TensorFlow in un modello TensorFlow Lite con operazioni TensorFlow non ancora supportate, queste devono

essere specificate tramite l'argomento `target_spec.supported_ops` del convertitore, dopo il caricamento del modello e prima della conversione.

2.2.3 Deployment: utilizzo del modello

Utilizzare il modello sul dispositivo significa avvalersi dell'interprete TensorFlow Lite per fare inferenza. L'inferenza è il processo che esegue dei dati mediante il modello per ottenere delle previsioni. L'interprete richiede quindi un modello e dei dati di input. L'interprete TensorFlow Lite consiste in una libreria che consente di caricare un modello, fornire gli input, eseguire le operazioni che il modello definisce su questi e prelevare gli output dell'inferenza. Le API di inferenza TensorFlow sono disponibili per le più comuni piattaforme mobili/integrate come Android, iOS e Linux, in diversi linguaggi di programmazione (Java, Swift, Objective-C, C++ e Python). Su Android, l'inferenza può essere eseguita utilizzando API Java o C++. Le API Java possono essere utilizzate direttamente all'interno delle classi di Activity di Android. Le API invece C++ offrono maggiore flessibilità e velocità, ma possono richiedere la scrittura di *wrapper* JNI per spostare i dati fra i livelli Java e C++. Su iOS, sono disponibili librerie native scritte in Swift e Objective-C per l'inferenza. Su piattaforme Linux, è possibile eseguire inferenze utilizzando API in C++ e Python. Per gli ambienti Android esiste inoltre una libreria addizionale a disposizione degli sviluppatori, la *TensorFlow Lite Android Support Library*. Questa rende più facile integrare i modelli nelle applicazioni, grazie ad API che aiutano a trasformare i dati di input (quali bitmap e tipi primitivi) nel formato richiesto dall'interprete (tensori sotto forma di `ByteBuffer`) che può essere difficile da manipolare.

Figura 3 riassume i processi di conversione e deployment dei modelli TensorFlow lite descritti nelle ultime due sezioni: 2.2.2 e 2.2.3.

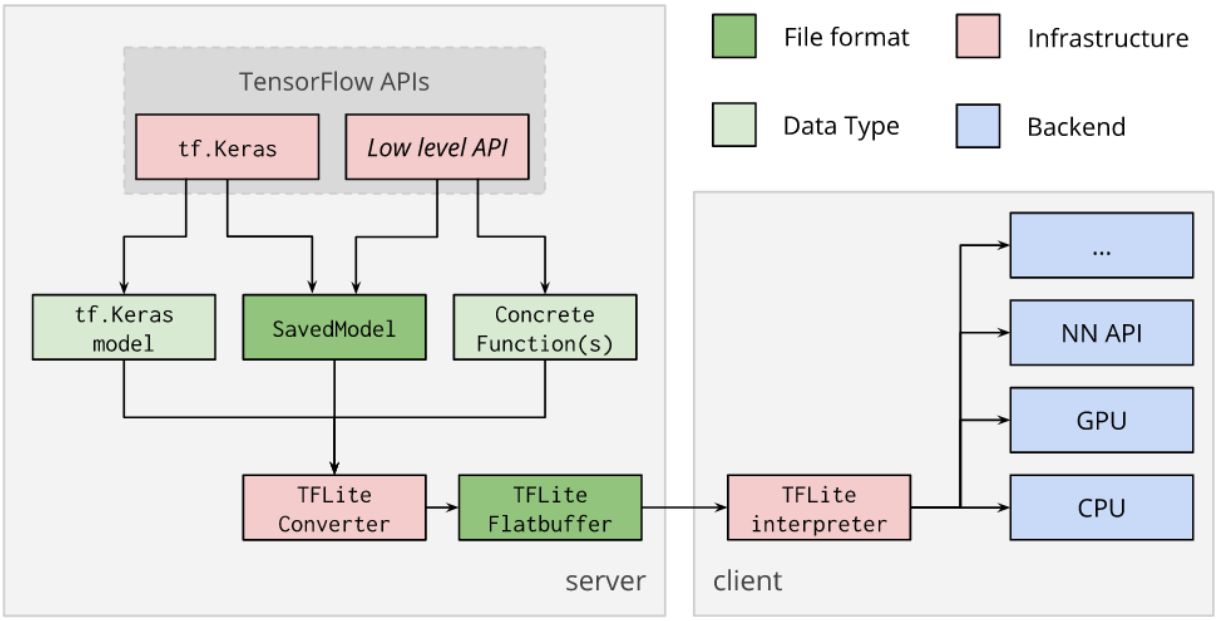

Figura 3 - Diagramma dei processi di conversione e deployment di un modello TensorFlow Lite [11]

2.2.4 Performance e ottimizzazione del modello

L'efficienza dell'inferenza è particolarmente importante per dispositivi mobili e IoT. Tali dispositivi hanno infatti molte restrizioni dal punto di vista della potenza di elaborazione, della memoria e del consumo energetico e poiché le attività di machine learning sono costose dal punto di vista computazionale, l'ottimizzazione del modello diventa estremamente importante. Seppure i requisiti hardware minimi di TensorFlow Lite siano molto bassi, siccome è desiderabile una bassa latenza per applicazioni mobili, la velocità di calcolo rimane il principale collo di bottiglia per le applicazioni. Ad esempio, un dispositivo mobile con hardware in grado di eseguire 10 Giga Floating Point Operations al secondo (FLOPS) può eseguire un modello che richiede 5 GFLOPS a 2 frame al secondo, il che potrebbe impedire all'applicazione che sfrutta il modello di raggiungere le prestazioni desiderate. [3]

Sono possibili diversi approcci per ottimizzare applicazioni basate su modelli TensorFlow Lite:

- Implementare le *best practices* suggerite dal team di sviluppo di TensorFlow lite. [12]

  Si tratta di strategie eterogenee quali scegliere un modello al cui precisione è commisurata alla funzione da svolgere, "comprimere" il modello applicando la quantizzazione e aumentare il numero di thread dedicati all'interprete.

  Per applicazioni che richiedono una ridotta precisione è meglio usare modelli più piccoli perché non solo utilizzano meno spazio su disco e in memoria, ma sono anche generalmente più veloci e più efficienti dal punto di vista energetico.

  Se un modello utilizza numeri in virgola mobile, riducendo la precisione dei valori e

delle operazioni all'interno del modello stesso, la quantizzazione può ridurne sia le dimensioni che il tempo necessario per l'inferenza, semplicemente trasformando i float in numeri a 8 bit. Esistono due varianti di quantizzazione: quantizzazione post-training e training quantizzato. Il primo tipo non richiede il *re-training* del modello ma, in rari casi, può comportare una perdita di precisione. Quando la perdita di accuratezza non è accettabile, è preferibile utilizzare il training quantizzato.

Infine, se la latenza dell'applicazione è più importante dell'efficienza energetica e del consumo di risorse, è possibile aumentare il numero di thread per velocizzare l'esecuzione.

- Sfruttare l'accelerazione hardware facendo uso di processori grafici e hardware dedicato per le reti neurali. Questo tipo di ottimizzazione è strettamente dipendente dall'hardware installato sul dispositivo e dal suo sistema operativo. Se si esegue un modello TensorFlow Lite su un sistema Android 8.1 (livello API 27) o superiore che supporta l'accelerazione hardware con hardware dedicato per reti neurali o una GPU, allora l'interprete utilizzerà l'*Android Neural Networks API* per velocizzare l'esecuzione del modello. Per esempio, il telefono Google Pixel 2 dispone di un chip ottimizzato per l'elaborazione delle immagini, che può essere abilitato con Android 8.1 e supporta quindi l'accelerazione hardware. [2]
- Utilizzare il *Model Optimization Toolkit*: un insieme di strumenti progettati per facilitare agli sviluppatori l'ottimizzazione dei modelli, consentendo la riduzione delle dimensioni di questi e l'aumento dell'efficienza con minimo impatto sulla precisione. Molti di questi strumenti possono essere applicati a tutti i modelli TensorFlow e non sono specifici di TensorFlow Lite, ma sono particolarmente utili quando si eseguono inferenze su dispositivi con risorse limitate. Questo toolkit include strumenti per l'implementazione dei due tipi di quantizzazione consigliati dalle *best practices*: la *Quantization Aware Training (QAT) API* consente di implementare il training quantizzato di un intero modello o solo parti di esso con poche linee di codice [15] mentre l'*Integer Quantization Tool* consente quantizzare un modello in floating-point per utilizzare solo interi con segno a 8 bit. Rispetto a QAT, questo strumento è molto più semplice da usare e comporta una perdita di precisione simile sulla maggior parte dei modelli, come mostrato in figura 4.

Model	Float baseline	Quantization during training	Quantization after training
MobileNet v1 (1.0, 224)	70.95%	69.97%	69.568%
ResNet v2	76.8%	76.7%	76.652%
Inception v3	77.9%	77.5%	77.782%

Figura 4 – Tabella di confronto relativa alla perdita di precisione dovuta a training quantizzato e quantizzazione post-training [16]

Il *Model Optimization Toolkit* include anche la W*eight Pruning API* che consente di ottimizzare un modello riducendo il numero di operazioni coinvolte nel calcolo rimuovendo i parametri, e quindi le connessioni, tra gli strati della rete neurale. Fare weight pruning significa impostare a zero parametri nei tensori dei pesi per eliminare connessioni non necessarie tra gli strati della una rete neurale. Ciò viene fatto durante il training per permettere alla rete neurale di adattarsi ai cambiamenti. Il beneficio di questo processo è la compressione su disco: i tensori "potati" possono essere compressi riducendo le dimensioni del modello. Inoltre, il weight pruning è compatibile con la quantizzazione; è stato sperimentato che combinando questi metodi è possibile ridurre la dimensione di un modello di 20 volte. [17]

3 Integrazione di TensorFlow Lite con dispositivi MQTT virtuali

In questo capitolo viene descritta un'applicazione Android, da me sviluppata, che prevede:

- un client MQTT (detto *publisher*) che pubblica le immagini classificate da un modello preconfigurato TensorFlow Lite,
- un broker MQTT in esecuzione sullo stesso dispositivo Android,
- un altro client MQTT (detto *subscriber*) che si iscrive presso il broker e salva le immagini ricevute sul disco.

Per la parte di acquisizione e classificazione delle immagini, quest'applicazione si basa sulle classi Java fornite nell'esempio presente nel repository ufficiale di TensorFlow Lite e descritto brevemente nella sezione successiva. [18]

3.1 *TensorFlow Lite image classification Android example*

Questo esempio consiste in un'applicazione Android che utilizza i frame catturati dalla fotocamera posteriore del dispositivo come input per un modello di classificazione, il quale restituisce come output una lista di valori che corrispondono alle probabilità che un'immagine riporti un oggetto descritto da un *label*. Ogni label corrisponde a un oggetto che il modello è in grado di riconoscere; la lista di tutti i label è contenuta in un file di testo `labels.txt` di cui ho riportato un estratto di seguito.

```
soap dispenser
soccer ball
sock
solar dish
sombrero
soup bowl
space bar
```

Questo file include 1001 entità fra cui animali, piante, edifici, strumenti musicali e capi di vestiario.

Per ottenere i frame della fotocamera e gestire le preferenze che l'utente specifica tramite l'interfaccia grafica, vengono usate le funzioni della classe `CameraActivity` definita nel file `CameraActivity.java`. La logica per l'elaborazione delle immagini e l'esecuzione dell'inferenza è invece implementata nel file `Classifier.java`. Nel costruttore della classe `Classifier` viene: caricato un modello `.tflite`, istanziato un nuovo `Interpreter` e caricato il file `labels.txt`. In tutte le applicazioni-esempio di TensorFlow Lite, i file relativi al modello e alla lista degli oggetti riconosciuti si trovano in una directory separata chiamata

`assets` (ma non è necessario che siano lì). Sempre nel costruttore `Classifier` le immagini catturate dalla fotocamera vengono trasformate da bitmap al formato `TensorImage` per rendere più efficiente la loro elaborazione e infine viene inizializzato l'output del modello (`TensorBuffer`).
L'inferenza avviene nel metodo pubblico `recognizeImage` il quale invoca il metodo `run` dell'`Interpreter` istanziato nel costruttore e restituisce una lista di istanze `Recognition` ognuna delle quali corrisponde a un label. `Recognition` è una classe privata definita nel file `Classifier.java` che contiene informazioni specifiche per un risultato dell'inferenza fra cui il titolo e la *confidence* (compresa fra 0 e 1) che l'immagine rappresenti l'oggetto descritto dal titolo. Il metodo `recognizeImage` restituisce un numero di risultati specificato da una costante che è pari a 3 di default.

I risultati dell'inferenza sono mostrati nell'interfaccia grafica grazie al metodo `processImage` della classe `ClassifierActivity` (sottoclasse di `CameraActivity`). Questo metodo esegue la classificazione in un thread in background per evitare di creare latenza. La classe `ClassifierActivity` si occupa inoltre di istanziare e configurare il modello per classificazione specificato dall'utente tramite l'interfaccia grafica. I modelli che è possibile utilizzare in quest'applicazione sono 4: MobileNetV1, MobileNetV quantizzato, EfficientNetLite ed EfficientNetLite quantizzato. MobileNetV1 è uno dei classici modelli TensorFlow Lite preconfigurati (descritti nella sezione 2.2.1) mentre EfficientNetLite è più recente e più accurato.

3.2 Architettura dell'applicazione e componenti principali

Il codice sorgente del progetto è reperibile qui [21]; le classi discusse in questa sezione sono contenute nel package org.tensorflow.lite.examples.classification.mqtt.
Il client MQTT che pubblica le immagini è implementato nella stessa classe in cui avviene la classificazione delle immagini stesse: `MainActivity`. Il broker e il client MQTT che esegue sottoscrizioni e salva le immagini ricevute sono implementati in classi separate che estendono `Thread` chiamate `Broker` e `Subscriber` rispettivamente.
Il broker è realizzato facendo uso della libreria Moquette per Android. Originariamente sviluppata per integrare broker MQTT *lightweight* in progetti relativi all'Internet of Things, Moquette è stata adattata all'ambiente Android da uno sviluppatore esterno al team originale come progetto indipendente. [19] I client MQTT sono invece realizzati sfruttando la libreria open-source Eclipse Paho, creata per fornire implementazioni affidabili dei protocolli di

messaggistica standard dell'ambiente Machine-to-Machine e Internet of Things. [20]

3.2.1 File `AndroidManifest.xml` e `build.gradle`

Per poter utilizzare la libreria Eclipse Paho è necessario definire i seguenti permessi nel manifest dell'applicazione:

```
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Il permesso WAKE_LOCK è necessario per impedire al dispositivo di entrare in modalità *sleep*, il che farebbe cadere la connessione con il broker. ACCESS_NETWORK_STATE è necessario perché l'applicazione rilevi cambiamenti nello stato della rete. INTERNET è necessario per consentire la connessione dei client MQTT verso il broker. READ_PHONE_STATE è necessario per conoscere informazioni quali l'identificatore Imei del dispositivo. READ_EXTERNAL_STORAGE è necessario per poter leggere i messaggi MQTT eventualmente resi persistenti e WRITE_EXTERNAL_STORAGE consente non solo di leggere questo tipo di messaggi ma anche di salvare le immagini ricevute da subscriber.
Inoltre, per associare l'applicazione al servizio Android Paho, è necessario aggiungere la riga seguente:

```
<application
    […]
    <service android:name="org.eclipse.paho.android.service.MqttService" />
    […]
</application>
```

Per evitare di replicare le informazioni relative al broker, ho deciso di includere nel manifest l'indirizzo e la porta sotto forma di metadati:

```
<application
    […]
    <meta-data
        android:name="BROKER_ADDRESS"
        android:value="localhost" />
    <meta-data
        android:name="BROKER_PORT"
        android:value="1883" />
    […]
</application>
```

Se si usa Android Studio e/o Gradle per gestire le dipendenze dell'applicazione allora si deve aggiungere il repository di Eclipse Paho nel file `build.gradle` del progetto:

```
allprojects {
    repositories {
        […]
```

```
        maven {
            url "https://repo.eclipse.org/content/repositories/paho-releases/"
        }
    }
}
```

Per quanto riguarda il file `build.gradle` dell'applicazione (diverso da quello del progetto), oltre alle librerie TensorFlow Lite devono essere incluse le dipendenze relative a Eclipse Paho e Moquette:

```
dependencies {
    […]
    implementation 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.1.1'
    implementation 'org.eclipse.paho:org.eclipse.paho.android.service:1.1.1'
    implementation 'io.moquette:moquette-netty-parser:0.8.1'
    implementation 'io.moquette:moquette-broker:0.8.1'
    implementation 'io.moquette:moquette-parser-commons:0.8.1'
}
```

3.2.2 Classe `Broker`

Come accennato il funzionamento del Broker è definito nell'omonima classe, nella quale viene fatto *override* del metodo `run`. In questo metodo vengono letti dal manifest indirizzo e porta, vengono configurate alcune proprietà e infine viene avviato il server che funge da broker.

```
import io.moquette.BrokerConstants;
import io.moquette.server.Server;
import io.moquette.server.config.MemoryConfig;
[…]
public class Broker extends Thread {
    private static final String TAG = "Broker";

    @Override
    public void run() {
        ApplicationInfo ai = null;
        try {
            ai =
MainActivity.getAppContext().getPackageManager().getApplicationInfo(MainActivity.get
AppContext().getPackageName(), PackageManager.GET_META_DATA);
        } catch (PackageManager.NameNotFoundException e) {
            e.printStackTrace();
        }
        String brokerAddress = (String) ai.metaData.get("BROKER_ADDRESS");
        int brokerPort = (int) ai.metaData.get("BROKER_PORT");

        Properties properties = new Properties();
        properties.setProperty(BrokerConstants.HOST_PROPERTY_NAME, brokerAddress);
        properties.setProperty(BrokerConstants.PORT_PROPERTY_NAME,
String.valueOf(brokerPort));
        properties.setProperty(BrokerConstants.PERSISTENT_STORE_PROPERTY_NAME,
Environment.getExternalStorageDirectory().getAbsolutePath() + File.separator +
BrokerConstants.DEFAULT_MOQUETTE_STORE_MAP_DB_FILENAME);
        MemoryConfig memoryConfig = new MemoryConfig(properties);
        Server mqttBroker = new Server();
        try {
            mqttBroker.startServer(memoryConfig);
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
        Log.i(TAG, "Server Started");
    }
}
```

3.2.3 Classe `MainActivity`

Questa classe contiene la maggior parte della logica dell'applicazione. Questa estende `CameraActivity` come `ClassifierActivity` e include gli stessi metodi di per l'aggiornamento dell'interfaccia grafica ma differisce da quest'ultima nei metodi `processImage` e `onCreate` e per il fatto che implementa l'interfaccia `MqttCallback`.

Nel metodo `processImage` dopo l'invocazione di `recognizeImage` (in cui avviene l'inferenza) avviene quella del metodo `publishImage` al quale viene passato come primo argomento il label con confidence più alta per l'immagine appena classificata e l'immagine stessa come secondo argomento.

```
protected void processImage() {
    […]

    runInBackground(
            new Runnable() {
                @Override
                public void run() {
                    if (classifier != null) {
                        […]
                        final List<Classifier.Recognition> results =
                                classifier.recognizeImage(rgbFrameBitmap,
sensorOrientation);
                        […]
                        publishImage(results.get(0).getTitle(), rgbFrameBitmap);
                        […]
                    }
                    readyForNextImage();
                }
            });
}
```

Il metodo `publishImage` converte l'immagine ricevuta come secondo argomento in un array di byte, quindi crea un nuovo messaggio MQTT che contiene l'immagine come payload e infine lo pubblica presso il broker assegnandogli come topic il primo argomento.

```
protected void publishImage(String topic, Bitmap bitmap) {
    try {
        if (!mqttAndroidClient.isConnected()) {
            mqttAndroidClient.connect();
        }
        // conversione da bitmap a byte[]
        ByteArrayOutputStream stream = new ByteArrayOutputStream();
        bitmap.compress(Bitmap.CompressFormat.PNG, 100, stream);
        byte[] byteArray = stream.toByteArray();
        // configurazione e pubblicazione del messaggio presso il broker
```

```
        MqttMessage message = new MqttMessage();
        message.setPayload(byteArray);
        message.setQos(0);
        mqttAndroidClient.publish(topic, message, null, new IMqttActionListener() {
            @Override
            public void onSuccess(IMqttToken asyncActionToken) {
                Log.i(TAG, "Image published (topic=" + topic + ")");
            }

            @Override
            public void onFailure(IMqttToken asyncActionToken, Throwable exception)
{
                Log.i(TAG, "Image publish failed!");
            }
        });
    } catch (MqttException e) {
        Log.e(TAG, e.toString());
        e.printStackTrace();
    }
}
```

Nel metodo `onCreate` per prima cosa viene invocato lo stesso metodo della superclasse (per inizializzare l'interfaccia grafica) poi vengono gestite le richieste relative ai cosiddetti *dangerous permissions*: permessi che potrebbero potenzialmente incidere sulla privacy dell'utente o sul normale funzionamento del dispositivo. Da Android 6.0 in poi se un'applicazione include nel proprio manifest permessi di questo tipo, l'utente deve approvarli esplicitamente. Siccome i permessi READ_PHONE_STATE, READ_EXTERNAL_STORAGE e WRITE_EXTERNAL_STORAGE fanno parte del gruppo dangerous permissions viene invocato il metodo `requestPermissions`, il quale mostra una finestra di dialogo dove l'utente può esprimere la propria decisione, nel caso uno di questi non sia già stato concesso. In caso contrario vengono avviati il broker, subscriber e publisher tramite l'invocazione di `startBrokerAndClients`.

```
@Override
protected void onCreate(final Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    MainActivity.context = getApplicationContext();
    if (ContextCompat.checkSelfPermission(context,
Manifest.permission.READ_PHONE_STATE) != PackageManager.PERMISSION_GRANTED
            || ContextCompat.checkSelfPermission(context,
Manifest.permission.READ_EXTERNAL_STORAGE) != PackageManager.PERMISSION_GRANTED
            || ContextCompat.checkSelfPermission(context,
Manifest.permission.WRITE_EXTERNAL_STORAGE) != PackageManager.PERMISSION_GRANTED) {
        requestPermissions(new String[]{Manifest.permission.READ_PHONE_STATE,
Manifest.permission.READ_EXTERNAL_STORAGE,
Manifest.permission.WRITE_EXTERNAL_STORAGE}, MQTT_PERMISSIONS_REQUEST_CODE);
    } else {
        Toast.makeText(MainActivity.this, "Permissions were already granted!",
Toast.LENGTH_SHORT).show();
        startBrokerAndClients();
    }
}
```

Siccome `requestPermissions` è un metodo asincrono nel caso i permessi non siano stati garantiti in precedenza, `startBrokerAndClients` viene invocato solo se l'utente decide di concederli sul momento. Quando l'utente risponde alla richiesta di autorizzazione, il sistema invoca l'implementazione dell'app del metodo `onRequestPermissionsResult` che gestisce anche le richieste relative ai permessi sulla fotocamera fatte nella classe `CameraActivity`.

```
@Override
public void onRequestPermissionsResult(int requestCode, final String[] permissions,
int[] grantResults) {

    switch (requestCode) {
        case PERMISSIONS_REQUEST:
            […] gestione delle richieste relative ai permessi sulla fotocamera

        case MQTT_PERMISSIONS_REQUEST_CODE: // REQUEST_PERMISSION_READ_PHONE_STATE,
REQUEST_PERMISSION_READ_EXTERNAL_STORAGE e REQUEST_PERMISSION_WRITE_EXTERNAL_STORAGE
            if (grantResults.length > 0 && grantResults[0] ==
PackageManager.PERMISSION_GRANTED) {
                Toast.makeText(MainActivity.this, "Permissions Granted!",
Toast.LENGTH_SHORT).show();
                startBrokerAndClients();

            } else {
                Toast.makeText(MainActivity.this, "Permissions denied - can't run
application!", Toast.LENGTH_SHORT).show();
            }
            return;
    }
}
```

Il metodo `startBrokerAndClients` avvia prima il thread relativo al broker e attende la sua terminazione poi avvia il thread relativo al subscriber e infine costruisce un nuovo `MqttAndroidClient` che si connette al broker tramite l'invocazione di connect – il pusblihser. `MqttAndroidClient` è una delle classi della libreria Eclipse Paho e consente a un'applicazione Android di comunicare con un server MQTT tramite metodi non bloccanti.
Per consentire al thread `Subscriber` di notificare lo UI thread delle sottoscrizioni effettuate, viene costruito un oggetto `Handler` e passato come argomento al costruttore di `Subscriber`.

```
private void startBrokerAndClients() {
    // avvio Broker
    Broker broker = new Broker();
    broker.start();
    try {
        broker.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    // definizione di un oggetto Handler legato al main thread
    Handler handler = new Handler(Looper.getMainLooper()) {
        public void handleMessage(Message topicMessage) {
            if (topicMessage.what == 99) {
                String subscriberTopic = (String) topicMessage.obj;
                Toast.makeText(MainActivity.this, "Subscriber is waiting for images
```

```
about " + subscriberTopic, Toast.LENGTH_SHORT).show();
                }
            }
        };
        // avvio Subscriber
        Subscriber subscriber = new Subscriber(handler);
        subscriber.start();
        // lettura di indirizzo e porta del server dai metadati del manifest
        ApplicationInfo ai = null;
        try {
            ai =
MainActivity.getAppContext().getPackageManager().getApplicationInfo(MainActivity.get
AppContext().getPackageName(), PackageManager.GET_META_DATA);
        } catch (PackageManager.NameNotFoundException e) {
            e.printStackTrace();
        }
        String brokerAddress = (String) ai.metaData.get("BROKER_ADDRESS");
        int brokerPort = (int) ai.metaData.get("BROKER_PORT");
        //avvio client publisher
        try {
            String serverURI = "tcp://" + brokerAddress + ":" + brokerPort;
            mqttAndroidClient = new MqttAndroidClient(getApplicationContext(),
serverURI, TAG, new MemoryPersistence());
            mqttAndroidClient.setCallback(this);
            mqttAndroidClient.connect();
        } catch (MqttException e) {
            e.printStackTrace();
        }
}
```

Il thread `Broker` viene avviato e immediatamente si attende la sua terminazione tramite `join` anziché eseguire direttamente il codice nel metodo `startBrokerAndClients`, perché quando un'applicazione tenta di eseguire un'operazione di rete sul suo UI thread viene lanciata una `NetworkOnMainThreadException`.

L'interfaccia `MqttCallback` permette a un'applicazione di essere notificata quando si verificano eventi asincroni relativi al client. Le classi che implementano questa interfaccia devono fare override di tre metodi:

1. `connectionLost(Throwable cause)`
2. `messageArrived (String topic, MqttMessage message)`
3. `deliveryComplete(IMqttDeliveryToken token)`

Nella classe `MainActivity` l'implementazione di questi metodi consiste in semplici invocazioni di metodi per il logging.

3.2.4 Classe `Subscriber`

Questa classe estende `Thread` (come `Broker`) e implementa l'interfaccia `MqttCallback` (come `MainActivity`). Nel suo costruttore viene: salvato il riferimento all'`Handler` del UI thread passato come argomento, istanziato un nuovo `MqttAndroidClient` (configurandolo

con indirizzo e porta del broker letti dal manifest), scelto il topic per l'iscrizione e infine inizializzato il campo relativo al percorso dove verranno salvate le immagini ricevute dal broker.

```
Subscriber(Handler handler) {
    this.mainThreadHandler = handler;
    // lettura di indirizzo e porta del server dai metadati del manifest
    ApplicationInfo ai = null;
    try {
        ai =
MainActivity.getAppContext().getPackageManager().getApplicationInfo(MainActivity.get
AppContext().getPackageName(), PackageManager.GET_META_DATA);
    } catch (PackageManager.NameNotFoundException e) {
        e.printStackTrace();
    }
    String brokerAddress = (String) ai.metaData.get("BROKER_ADDRESS");
    int brokerPort = (int) ai.metaData.get("BROKER_PORT");
    String brokerURI = "tcp://" + brokerAddress + ":" + brokerPort;

    this.mqttAndroidClient = new MqttAndroidClient(MainActivity.getAppContext(),
brokerURI, TAG, new MemoryPersistence());
    this.pickNewRandomTopic();
    this.imagesFolder = Environment.getExternalStorageDirectory().toString() +
File.separator + "MobileSystemsM";
}
```

Nel metodo `run` avviene la connessione al broker e l'iscrizione al topic.

```
@Override
public void run() {
    try {
        mqttAndroidClient.setCallback(this);
        do {
            SystemClock.sleep(300);
            mqttAndroidClient.connect();
        }
        while (!mqttAndroidClient.isConnected() && Log.i(TAG, "Falied to connect to
MQTT Broker - attempting reconnection") != 0);
        Log.i(TAG, "Connected successfully  to MQTT Broker");

        this.subscribeToTopic();

    } catch (MqttException e) {
        e.printStackTrace();
    }
}
```

Nel metodo `subscribeToTopic` non viene solamente realizzata l'iscrizione ma viene anche inviato un messaggio informativo allo UI thread.

```
private void subscribeToTopic() {
    try {
        mqttAndroidClient.subscribe(this.topic, 0, null, new IMqttActionListener() {
            @Override
            public void onSuccess(IMqttToken asyncActionToken) {
                Log.i(TAG, "Subscribed to topic");
            }

            @Override
```

```
            public void onFailure(IMqttToken asyncActionToken, Throwable exception)
{
                Log.i(TAG, "Failed to subscribe");
            }
        });
    } catch (MqttException e) {
        e.printStackTrace();
    }
    Message message = Message.obtain(this.mainThreadHandler, 99, this.topic);
    mainThreadHandler.sendMessage(message);
}
```

A differenza di `MainActivity` l'implementazione di `messageArrived` non prevede un semplice logging. Alla ricezione di un messaggio dal broker il subscriber annulla l'iscrizione al topic corrente, converte l'immagine contenuta nel payload del messaggio in bitmap, la salva sul disco quindi sceglie un nuovo topic ed effettua una nuova iscrizione presso il broker.

```
@Override
public void messageArrived(String topic, MqttMessage message) throws Exception {
    Log.i(TAG, "Recevied image for topic " + this.topic);
    mqttAndroidClient.unsubscribe(this.topic);
    //conversione da byte[] a bitamp
    BitmapFactory.Options options = new BitmapFactory.Options();
    options.inMutable = true;
    Bitmap bitmap = BitmapFactory.decodeByteArray(message.getPayload(), 0,
message.getPayload().length, options);
    // salvataggio immagine su disco
    String imageName = topic + ".png";
    File imageFile = new File(imagesFolder, imageName);
    OutputStream outputStream = new FileOutputStream(imageFile);
    bitmap.compress(Bitmap.CompressFormat.PNG, 80, outputStream);
    outputStream.close();
    Log.i(TAG, "Saved image " + imageName);
    // nuova sottoscrizione
    this.pickNewRandomTopic();
    this.subscribeToTopic();
}
```

3.3 Risultati

È opportuno menzionare che per poter eseguire quest'applicazione sul mio dispositivo Android è stato necessario:

- Installare la versione appropriata del Software Development Kit in Android Studio.
- Abilitare il debug tramite USB nel dispositivo.
- Scaricare e installare i driver relativi al produttore del dispositivo sulla macchina su cui esegue Android Studio.
- Riavviare il protocollo di connessione alla macchina (Media Transfer Protocol o Picture Transfer Protocol) nel dispositivo.

Il funzionamento dell'applicazione è quello atteso: l'interfaccia grafica dell'esempio TensorFlow Lite viene avviata per prima e se l'utente ha concesso i permessi necessari vengono avviati anche il broker MQTT e i due client. Se il client publisher è creato con successo ogni immagine classificata viene anche pubblicata presso il broker che invia al subscriber le immagini il cui label corrisponde al topic specificato durante la sua sottoscrizione. Ricevuta un'immagine, il subscriber la salva su disco e notificata il publisher del cambiamento del topic di interesse come mostrato in figura 5.

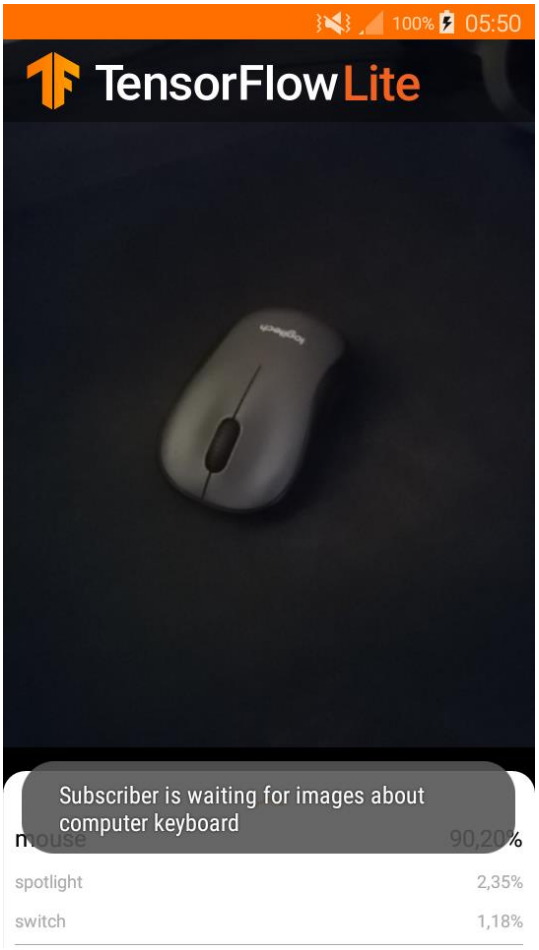

Figura 5 – Ricevuta un'immagine relativa a un topic (mouse), subscriber cambia topic di interesse (computer keyboard) e notifica publisher.

3.3.1 Utilizzo delle risorse del dispositivo

Tramite Android Studio è possibile monitorare l'uso delle risorse del dispositivo da parte dell'applicazione. Android Profiler (introdotto in Android Studio 3.0) sostituisce Android Monitor e fornisce in tempo reale dati relativi a: attività della CPU, allocazione della memoria, traffico di rete e consumo energetico. Siccome il mio smartphone è un dispositivo Android 6.0 (API 23) e la visualizzazione del profilo energetico è possibile solo su dispositivi Android 8.0 (API 26), questo profilo non è incluso nei risultati descritti di seguito.

In figura 6 è riportato il profilo dell'applicazione in un intervallo di tempo di 30 secondi durante il quale viene usato per la classificazione il modello EfficientNet quantizzato. Come si può vedere non si hanno transizioni fra activity e l'utilizzo della memoria resta costante. Il profilo della CPU presenta invece picchi di utilizzo (da 30% a 50%) in corrispondenza della ricezione di messaggi MQTT nel thread Subscriber. Non è presente alcun traffico in rete perché, come già discusso, TensorFlow Lite non prevede scambio di informazioni con un server e il broker MQTT è in esecuzione sullo stesso dispositivo dei client. Se l'applicazione facesse affidamento a un server esterno allora sarebbe necessaria una connessione a Internet.

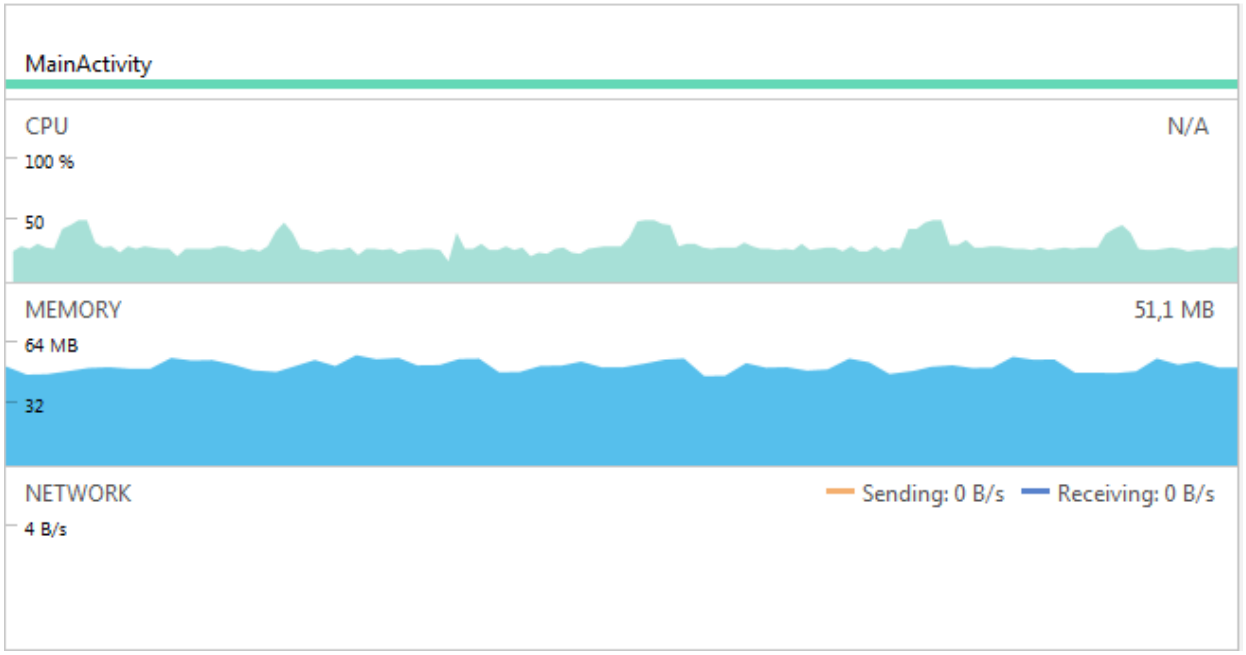

Figura 6 – Profilo dell'applicazione durante l'uso del modello EfficientNet quantizzato.

Il profilo dell'applicazione non varia significativamente se si utilizza l'altro modello quantizzato (MobileNet). Invece se l'utente seleziona uno dei modelli floating point, tramite l'interfaccia grafica, si ha un incremento di circa il 40% nell'utilizzo della memoria, come mostrato in figura 7. I pallini rossi rappresentano l'interazione dell'utente con il touch screen durante la scelta del nuovo modello.

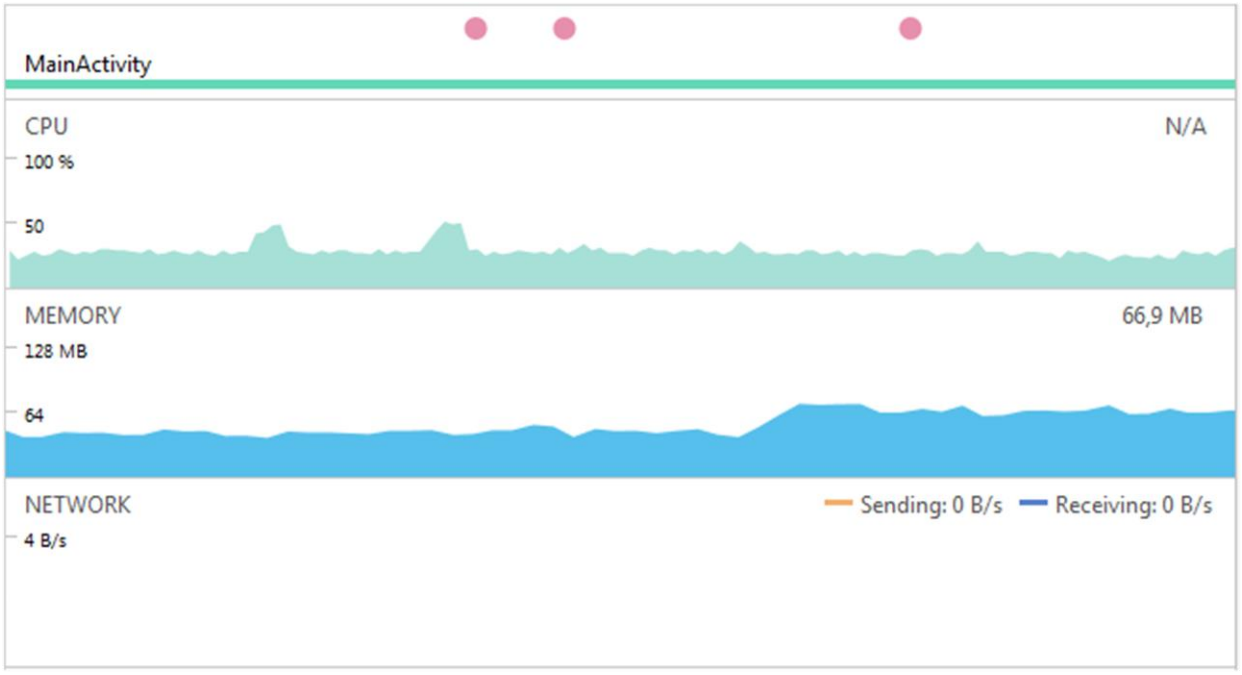

Figura 7 - Profilo dell'applicazione durante la transizione fra EfficientNet quantizzato e EfficientNet floating point.

Analizzando il profilo della memoria singolarmente è possibile vedere che questo aumento nell'utilizzo della memoria è dovuto principalmente all'allocazione di più codice nell'heap, come evidenziato in figura 8.

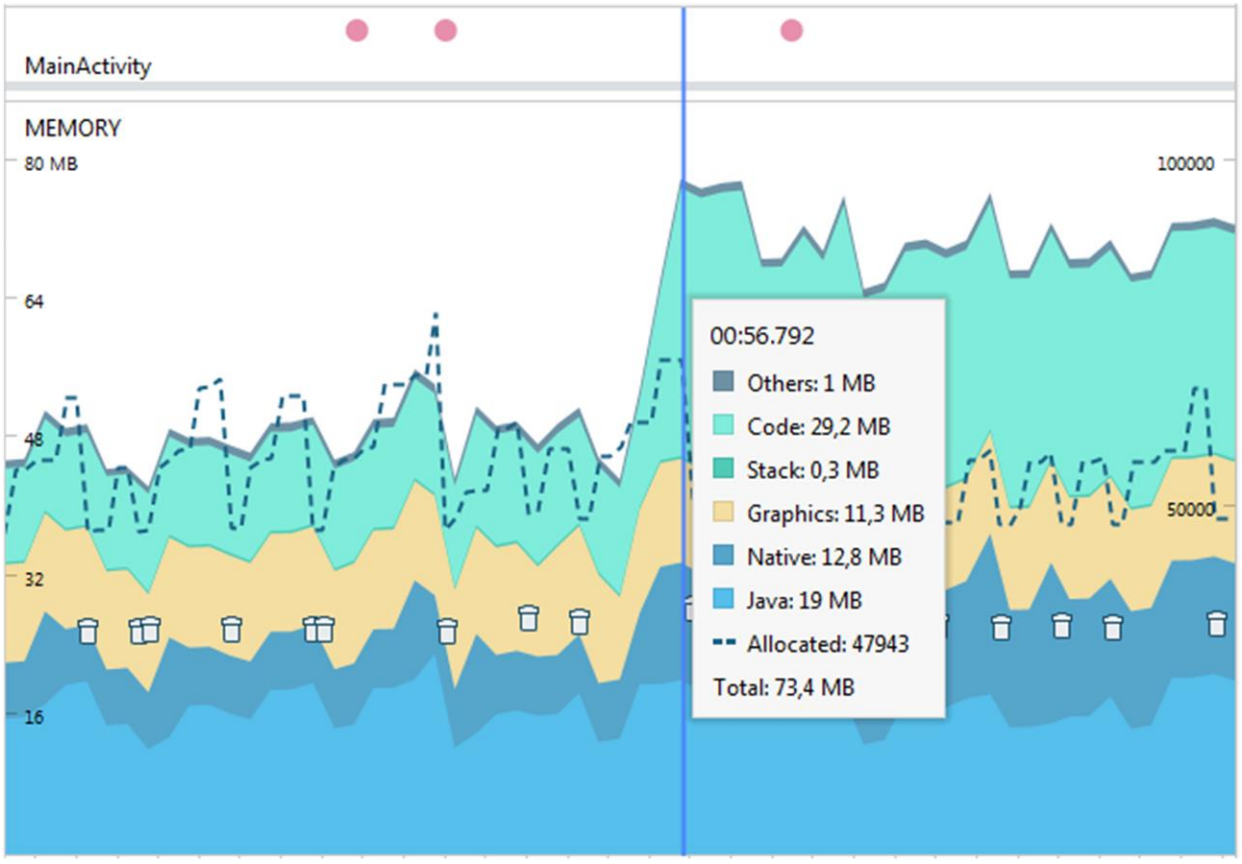

Figura 8 - Profilo della memoria durante la transizione fra modelli.

3.3.2 Precisione e tempi di risposta

La piccola differenza in accuratezza fra i modelli MobileNet e EfficientNet (4% secondo i benchmark di Google) non incide sul funzionamento dell'applicazione: tutte le immagini vengono inviate al broker MQTT una volta classificate a prescindere dalla confidence.
Il tempo necessario per l'inferenza invece varia significativamente fra i 4 modelli e costituisce il componente principale della latenza dell'applicazione. È possibile aumentare il numero di thread dedicati all'inferenza dall'interfaccia grafica dell'applicazione per migliorarne le prestazioni. La tabella seguente riassume i tempi per l'inferenza minimi e massimi dei vari modelli in esecuzione sulla CPU del mio smartphone (Samsung A300 Galaxy A3) relativamente al numero di thread. Per ogni configurazione di modello e numero di thread, i valori riportati sono il risultato di un minuto di utilizzo dell'applicazione.

Modello	**1 thread**	**2 thread**	**4 thread**
EfficientNet quantizzato	385-412 ms	240-266 ms	185-224 ms
EfficientNet floating point	584-630 ms	418-445 ms	339-391 ms
MobileNet quantizzato	392-428 ms	241-270 ms	223-262 ms
MobileNet floating point	799-854 ms	411-513 ms	354-423 ms

Come prevedibile, il modello più veloce è EfficientNet quantizzato mentre quello più lento è MobileNet floating point con un tempo per l'inferenza medio quasi doppio! La differenza in prestazioni fra i modelli quantizzati diventa più evidente aumentando il numero thread. Ciò risulta efficace nella riduzione dei tempi per l'inferenza ma fornisce rendimenti decrescenti per tutti i modelli.
Smartphone più recenti sono in grado di garantire tempi di risposta un ordine di grandezza minori, come dimostrato dai benchmark di Google relativi a Pixel 3 e Pixel 4. [22] Dispositivi che possono impiegare per l'inferenza la propria GPU o l'Android Neural Networks API (NNAPI) assicurano prestazioni ancora migliori senza dover ricorrere all'utilizzo di molteplici thread.

4 Bibliografia

[1] TensorFlow Lite, documentazione ufficiale, https://www.tensorflow.org/lite/guide, ultimo accesso: dicembre 2019.

[2] Tang Jeff, *Intelligent mobile projects with TensorFlow*, UK, Packt Publishing Ltd., maggio 2018.

[3] Jeffrey Dean, Greg S. Corrado et al., *Large Scale Distributed Deep Networks*, Mountain View, USA, 2012.

[4] Paper DistBelief Alsing Oscar, *Mobile Object Detection using TensorFlow Lite and Transfer Learning*, KTH royal institute of technology, Svezia, agosto 2018.

[5] FlatBuffers, documentazione ufficiale, https://google.github.io/flatbuffers , ultimo accesso: maggio 2020.

[6] TensorFlow Developer summit, slideshow, https://www.slideshare.net/modulabs/machine-learning-on-your-hand-introduction-to-tensorflow-lite-preview , ultimo accesso: maggio 2020.

[7] TensorFlow Lite, documentazione ufficiale, https://www.tensorflow.org/lite/guide/roadmap, ultimo accesso: maggio 2020.

[8] TensorFlow Lite, documentazione ufficiale, https://www.tensorflow.org/lite/microcontrollers, ultimo accesso: maggio 2020.

[9] TensorFlow Lite, documentazione ufficiale, https://www.tensorflow.org/lite/models, ultimo accesso: maggio 2020.

[10] TensorFlow Lite, documentazione ufficiale, https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/saved_model/README.md, ultimo accesso: maggio 2020.

[11] TensorFlow Lite, documentazione ufficiale, https://www.tensorflow.org/lite/convert, ultimo accesso: maggio 2020.

[12] TensorFlow Lite, documentazione ufficiale, https://www.tensorflow.org/lite/performance/best_practices, ultimo accesso: maggio 2020.

[13] Netron, repository ufficiale, https://github.com/lutzroeder/Netron, ultimo accesso: maggio 2020.

[14] TensorFlow, repository ufficiale, https://github.com/tensorflow/tensorflow/issues/25085, ultimo accesso: maggio 2020.

[15] TensorFlow, documentazione ufficiale, https://blog.tensorflow.org/2020/04/quantization-aware-training-with-tensorflow-model-optimization-toolkit.html, ultimo accesso: maggio 2020.

[16] TensorFlow, documentazione ufficiale, https://blog.tensorflow.org/2019/06/tensorflow-integer-quantization.html, ultimo accesso: maggio 2020.

[17] TensorFlow, documentazione ufficiale, https://blog.tensorflow.org/2019/05/tf-model-optimization-toolkit-pruning-API.html, ultimo accesso: maggio 2020.

[18] TensorFlow Lite, esempio di classificazione di immagini in Android, https://github.com/tensorflow/examples/tree/master/lite/examples/image_classification/android, ultimo accesso: maggio 2020.

[19] Moquette, repository ufficiale, https://github.com/technocreatives/moquette, ultimo accesso: maggio 2020.

[20] Libreria Eclipse Paho per Android, https://github.com/eclipse/paho.mqtt.android, ultimo accesso: maggio 2020.

[21] Repository GitHub del progetto, https://github.com/pozzasimone/ProgettoMobileSystems, ultimo accesso: maggio 2020.

[22] Benchmark modelli TensorFlow Lite, https://www.tensorflow.org/lite/performance/benchmarks, ultimo accesso: maggio 2020.