

# Relazione tesina Sistemi Embedded

Michele Sulis

Giugno 2025

**Prefazione** La presente relazione ha lo scopo di riportare in maniera organica il lavoro svolto per il completamento della tesina assegnata per il conseguimento dell'esame di Sistemi Embedded.

Il progetto consiste nel design di una rete neurale per il riconoscimento dello stato di cottura di pizze in uscita dal forno, per inviare segnali al sistema di controllo del forno stesso, in maniera da automatizzarlo e non richiedere l'intervento dell'operatore.

In particolare sono state previste 4 classi di riconoscimento:

- pizza cotta (in uscita dal forno);
- pizza cruda (in ingresso nel forno);
- pala vuota (dopo aver scaricato la pizza da cuocere o prima di raccogliere la pizza cotta);
- oggetti non appartenenti a nessuna delle tre precedenti categorie.

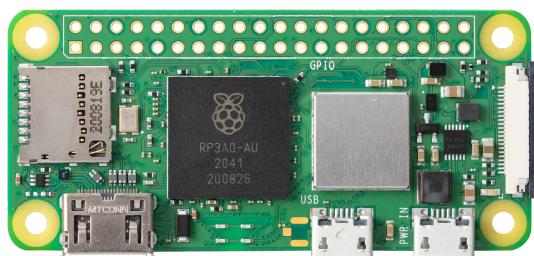
Una volta progettata la rete è stata poi implementata su [Raspberry Pi Zero 2 W](#) con camera CSI (Camera Serial Interface). I principali vincoli progettuali sono stati quelli di avere un sistema che fosse poco ingombrante, con connettività wireless in modo da potersi interfacciare con altri sistemi embedded per la gestione del forno, a relativo basso consumo, ma comunque con una potenza di calcolo necessaria a garantire 2-3 inferenze al secondo.

## 1 Descrizione del sistema

Alla base del funzionamento del sistema vi è la Raspberry Pi Zero 2 W, la cui fotografia frontale è riportata in Figura 1. Essa è una Single Board Computer (SBC) che incorpora l'RP3A0, un System in Package (SiP) costituito da quattro core ARM Cortex-A53 a 64 bit con clock a 1 GHz e da 512MB di SDRAM LPDDR2 su die Broadcom.

La potenza di calcolo data dalle CPU Cortex-A e la memoria sono sufficienti a poter eseguire agevolmente una distribuzione Linux, comunque garantendo il rispetto delle specifiche di progetto per quanto riguarda la velocità di esecuzione, ma al contempo semplificando notevolmente lo sviluppo del software.

La Raspberry Pi Zero 2 W inoltre implementa diverse periferiche tra le quali ragguardevoli nel progetto sono state il connettore CSI-2 per la fotocamera, il lettore microSD e la connettività wireless Wi-Fi a 2.4 GHz con la quale è stato possibile interfacciarsi con la scheda e ricevere i risultati delle inferenze del modello.



**Figura 1.** SBC (Single Board Computer) Raspberry Pi Zero 2 W. Le dimensioni della scheda sono 65mm × 30mm.

Il setup della scheda insieme con la fotocamera è riportato in Figura 2. Il modulo fotocamera si interfaccia alla Zero 2 W tramite CSI-2, che sfrutta due coppie differenziali per i dati e una per il clock, mentre il connettore FFC prevede anche l'alimentazione, le linee di massa, le linee di controllo I2C, i segnali di interrupt e quelli di power-management. Il sensore da 5 MP genera internamente il pixel clock e sincronizza la trasmissione dei frame tramite Vertical SYNChronization (VSYNC). Per trasmettere i dati, ogni pixel viene codificato a 10 bit e inviato sulle due data-lane dell'interfaccia CSI-2, che offre una banda massima aggregata fino a 3Gbit/s. La raspberry Pi, configurato il sensore via I2C all'avvio, rimane in ascolto e riceve i pacchetti via Direct Memory Access (DMA), occupandosi di ricostruire i pixel (RAW o YUV) in memoria non appena arrivano.



**Figura 2.** Setup per i test con la scheda collegata tramite cavo schermato alla camera CSI, su cavalletto.

## 2 Modello di rete neurale

Per quanto riguarda il modello di rete neurale è stato utilizzato EfficientNet-B0 tramite transfer learning [1], partendo dai pesi pre-addestrati su ImageNet [2] e adattando EfficientNet-B0 al dataset specifico.

EfficientNet-B0 è il modello di base della famiglia EfficientNet, sviluppata da Mingxing Tan e Quoc V. Le del team Google Brain nel 2019 [3], basato su blocchi MBConv (Mobile Inverted Bottleneck), che si chiamano così perché prima comprimono temporaneamente le informazioni (riducendo i canali) e poi le espandono nuovamente, ottimizzando l'efficienza computazionale, integrando un meccanismo di attenzione per dare maggiore peso alle feature più importanti. Il modello conta circa 4 milioni di parametri e accetta in input immagini RGB di dimensione  $224 \times 224$  pixel. EfficientNet-B0 è inoltre accessibile “out-of-the-box” tramite `tensorflow.keras.applications`, modulo incluso nella libreria TensorFlow.

```

1  from tensorflow.keras import layers, models
2  from tensorflow.keras.applications.efficientnet import EfficientNetB0
3
4  inputs = layers.Input(shape=(224, 224, 3))
5
6  base_model = EfficientNetB0(
7      include_top=False,
8      weights='imagenet',
9      input_tensor=inputs
10 )
11 base_model.trainable = False
12
13 x = layers.GlobalAveragePooling2D()(base_model.output)
14 x = layers.BatchNormalization()(x)
15 x = layers.Dropout(0.3)(x)
16 x = layers.Dense(256, activation='relu')(x)
17 x = layers.Dropout(0.5)(x)
18 outputs = layers.Dense(4, activation='softmax')(x)
19
20 model = models.Model(inputs, outputs)

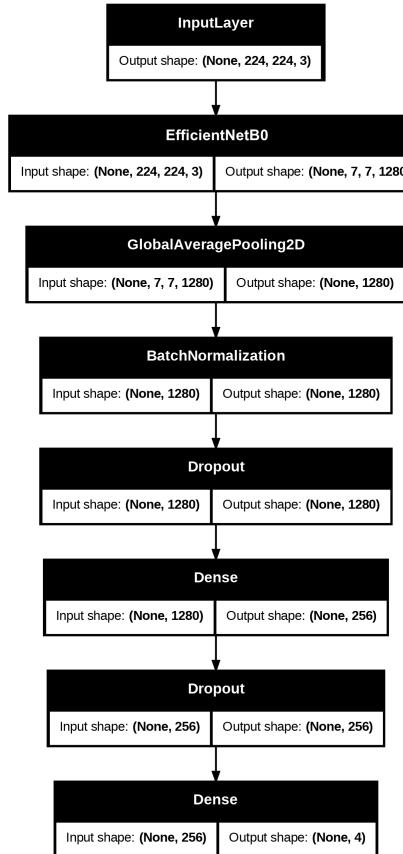
```

**Listato 1.** Codice python per l'inizializzazione del modello.

In particolare, come mostrato nel listato 1, è stato rimosso il cosiddetto *top layer* e aggiunti al suo posto due strati densi, rispettivamente di 256 e 4 neuroni (uno per ogni classe), in modo da sfruttare gli strati più interni per fare estrazione di feature e gli ultimi strati per adattarla all'applicazione.

Il diagramma della rete neurale è riportato in figura 3. Oltre agli strati densi, sono stati aggiunti:

- **GlobalAveragePooling2D**, che riduce ciascuna feature map bidimensionale a un singolo valore medio, concentrando l'informazione più saliente;
- **BatchNormalization**, per normalizzare le attivazioni e rendere l'allenamento più stabile e veloce;
- **Due strati di Dropout** (rate 0.3 e 0.5), che disattivano casualmente porzioni di neuroni durante l'addestramento per limitare l'overfitting.



**Figura 3.** Diagramma della rete neurale a progettata a partire da EfficientNetB0, con i top layer custom, per il transfer learning.

## 2.1 Dataset e addestramento

Per poter addestrare la rete è stato assemblato un dataset di 18000 immagini equamente suddivise nelle quattro classi. Le prime tre classi sono state ottenute da [Google Immagini](#), controllate e ridimensionate: le immagini originali sono state scalate e adattate a rapporto d'aspetto 1:1 tramite padding, dopodiché si è applicata una data augmentation per equilibrare il numero di immagini tra le classi. In particolare, dalla versione maggiore ( $256 \times 256$ ) sono state estratte n sottoimmagini di  $224 \times 224$  pixel con centro casuale e ruotate di  $90^\circ$ ,  $180^\circ$  e  $270^\circ$ . La quarta classe, invece, è stata creata selezionando casualmente 4 500 immagini dal dataset ImageNet.

Il dataset è quindi stato suddiviso in 80% training set, 10% validation set e 10% test set, e l'addestramento è stato eseguito su piattaforma [Kaggle](#) con GPU ad alte prestazioni. Dopo vari esperimenti, la configurazione ottimale — batch size 32 per 2 epoche — ha raggiunto un'accuratezza del 98,6% sul test set.

## 2.2 Conversione a TensorFlowLite e quantizzazione

```

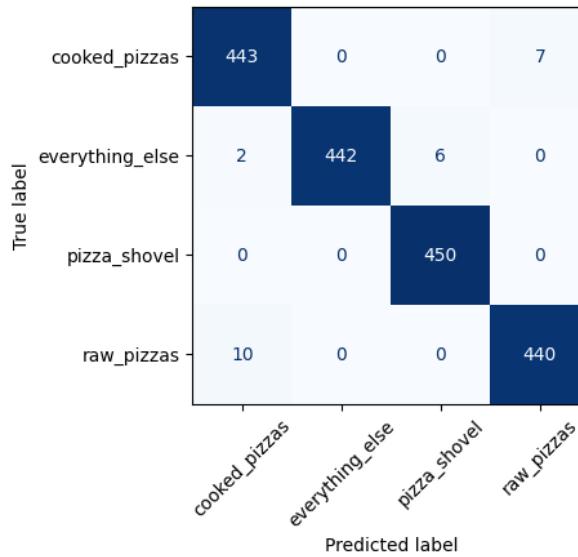
1 #takes keras model and turn it in a converter TFLite
2 converter = tf.lite.TFLiteConverter.from_keras_model(model)
3
4 converter.representative_dataset = representative_dataset_gen
5 converter.optimizations = [tf.lite.Optimize.DEFAULT]
6
7 #conversion to int16
8 converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS]
9 converter.target_spec.supported_types = [tf.int16]
10 #do not quantize in & out
11
12 tflite_quant_model = converter.convert()
13
14 open(name_modelTFLite, "wb").write(tflite_quant_model)

```

**Listato 2.** Codice python per la conversione in TFLite e la quantizzazione del modello a partire da un modello Keras.

Nel Listato 2 è riportato il processo di conversione del modello Keras in formato TensorFlow Lite (TFLite). In primo luogo, si istanzia un oggetto `TFLiteConverter` a partire dal modello Keras, in modo che il convertitore possa accedere all'architettura e ai pesi del modello. Impostando un `representative_dataset`, è poi possibile eseguire la quantizzazione ottimizzata, riducendo la precisione interna (in questo caso a 16 bit) senza alterare float input/output. Il convertitore infatti può osservare come il modello si comporta su dati reali, e scegliere valori di scala e offset adatti per rappresentare i numeri a precisione ridotta. In questo modo si ottiene una quantizzazione più efficace, con minori perdite in termini di accuratezza. È stato riscontrato durante la fase di test che questo tipo di quantizzazione è sostenibile dall'hardware sul quale è stato fatto girare. L'operazione `converter.convert()` genera infine un file FlatBuffer `.tflite`, che contiene il grafo e i pesi per l'esecuzione su dispositivi embedded. L'operazione `converter.convert()` genera infine un file FlatBuffer `.tflite`, che contiene il grafo e i pesi per l'esecuzione su dispositivi embedded.

È stato poi valutato il modello TFLite sul test set, attraverso la matrice di confusione mostrata in Figura 4.



**Figura 4.** Matrice di confusione del modello TFLite con pesi quantizzati a 16 bit, identica alla versione a precisione completa.

## 3 Implementazione su Raspberry

### 3.1 Toolchain e librerie

Una volta progettato e testato il modello e convertito in TensorFlow Lite, è stato necessario predisporre la toolchain e le librerie per il deployment. Su Raspberry Pi Zero 2 W è stata installata Raspbian Lite a 64 bit (basata su Debian), che include già diversi pacchetti utili allo sviluppo, come il compilatore `g++`. In un secondo momento sono state aggiunte:

- `libcamera`[4], libreria C++ per la gestione dello streaming video dalla fotocamera, che sfrutta un thread dedicato e invoca una callback a ogni nuovo frame;
- `OpenCV`[5], utilizzata per convertire i frame da YUV a RGB, ridimensionare le immagini e preparare il formato di input per l'interprete TFLite.

Infine, TensorFlow Lite è stato cross-compilato con Bazel [6], producendo delle librerie dinamiche che - insieme a FlatBuffers - formano il motore di inferenza TFLite. Questo motore carica il modello `.tflite` e converte i dati di ingresso (immagini → tensori) prima di eseguire l'inferenza. Il progetto integra queste librerie, organizzandole nella struttura dei file descritta nelle sottosezioni successive.

### 3.2 CameraHandler.cpp

`CameraHandler.cpp` è una classe wrapper attorno a `libcamera` che gestisce l'inizializzazione, la configurazione e l'avvio/arresto di una videocamera. La funzione `init(width,height)` avvia il `CameraManager`, seleziona la prima camera disponibile e configura un flusso NV12 (YUV semi-planare) alla risoluzione richiesta, allocando automaticamente i buffer via `FrameBufferAllocator`. Chiamando `start()` la classe mette in coda le richieste di acquisizione e avvia lo streaming, mentre `stop()` lo interrompe. Ogni volta che un frame è pronto, la callback interna `requestComplete(Request*)` mappa il buffer in memoria con `mmap`, converte i dati YUV in BGR tramite OpenCV e incapsula il risultato in un oggetto `CameraFrame`, che viene passato all'interprete TensorFlow Lite per l'inferenza. Il distruttore `CameraHandler()` si occupa infine di fermare la camera, liberare i buffer e arrestare il `CameraManager`.

### 3.3 ModelInterpreter.cpp

`ModelInterpreter.cpp` è una classe wrapper attorno a TensorFlow Lite che crea il motore di inferenza, caricando il file `.tflite` e le etichette da disco, costruendo l'interprete con `InterpreterBuilder` e allocando i tensori all'interno di `init()`. Il metodo `runInference(const uint8_t* image_data)` copia i dati in ingresso nel tensore (supportando sia `uint8` sia `float32`), invoca l'interprete con `Invoke()` e infine estrae le classi e le confidenze nel vettore di `Detection`.

### 3.4 main.cpp

`main.cpp` è il punto di ingresso del programma. Inizializza i due sottosistemi principali: `ModelInterpreter` per l'inferenza e `CameraHandler` per l'acquisizione video. Quando un nuovo frame è disponibile, la callback `processFrameAndInfer` viene invocata, ridimensiona l'immagine alla risoluzione del modello, converte i colori da BGR a RGB ed esegue l'inferenza. I risultati (classi e confidence) vengono stampati su console e per la classe con confidenza massima viene riportato il nome. Infine, il frame originale viene visualizzato a schermo tramite OpenCV.

### 3.5 Makefile

Il `Makefile` automatizza la compilazione del progetto. Definisce le opzioni del compilatore `g++`, include le directory di intestazione necessarie (OpenCV, TensorFlow Lite, libcamera), e collega le rispettive librerie. I file sorgente `.cpp` vengono compilati in oggetti `.o`, infine uniti nell'eseguibile `my_interpreter`. Il target `clean` consente di rimuovere i file generati. È incluso anche il path runtime per le librerie dinamiche tramite l'opzione `-Wl,-rpath`.

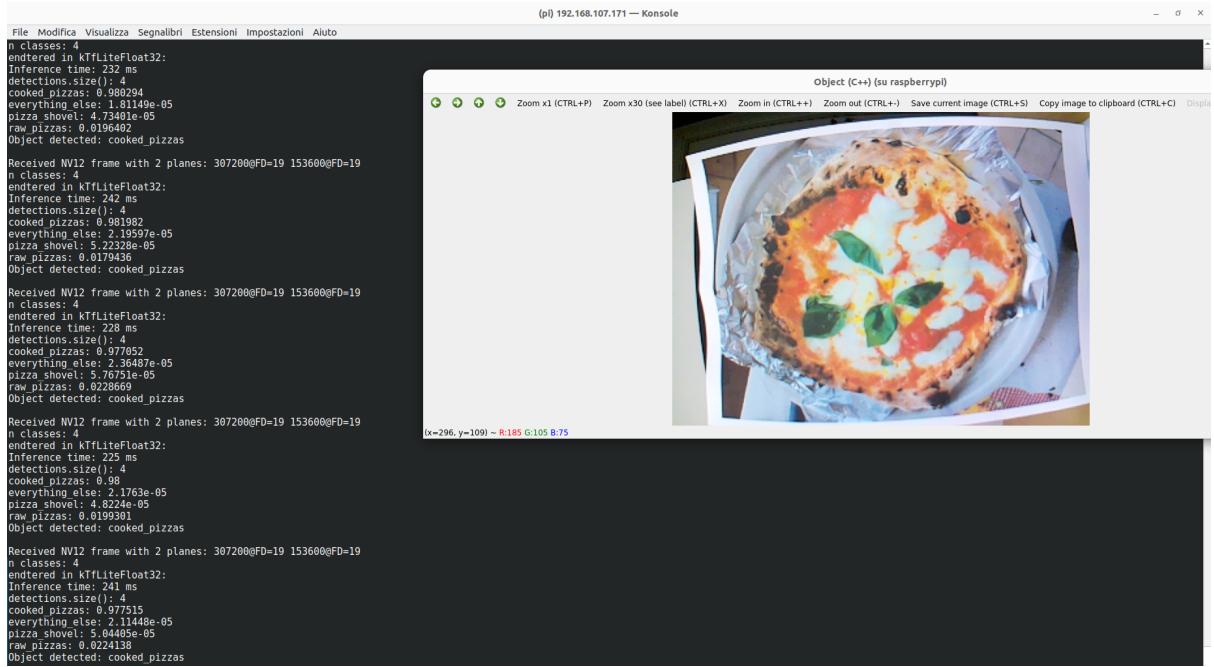
## 4 Risultati

Una volta compilato correttamente il progetto direttamente su Raspberry e aver ottenuto l'eseguibile, per verificare il corretto funzionamento del sistema è stato allestito il seguente setup: la scheda con fotocamera è stata montata su un supporto, il quale a sua volta è stato agganciato a un cavalletto. Le pizze da riconoscere sono state fotografate sia direttamente in loco sia scaricate da Google Immagini, stampate su fogli A4 o visualizzate su tablet. La scheda è stata connessa al ricevitore tramite Wi-Fi e controllata via SSH. Un'immagine del set per il testing è riportata in Figura 5.



**Figura 5.** Rispettivamente, vista laterale destra **a**, e dall'alto **b** del setup di test. Il righello è segnato fino ad una lunghezza di 20 cm e rappresenta un'indicazione qualitativa della distanza a cui deve trovarsi la fotocamera dall'oggetto, per il riconoscimento corretto.

In fase di test sono state valutate diverse distanze fra l'obiettivo della fotocamera e i fogli con gli oggetti da riconoscere, regolando manualmente la messa a fuoco ogni volta. Un'immagine dell'interfaccia del ricevitore è riportata in Figura 6.



**Figura 6**

**Figura 7.** Interfaccia lato ricevitore composta da un terminale connesso via ssh -X, che consente di inoltrare l'interfaccia grafica della Raspberry sul client remoto. Via SSH si esegue il programma compilato e, nella figura, a sinistra è visibile l'output testuale, mentre a destra viene mostrata l'immagine catturata dalla fotocamera.

Una volta trovata la distanza ottimale, i risultati sono stati molto buoni, con un'accuratezza delle classificazioni praticamente unitaria. Criticità e limiti del setup verranno discussi nella sezione successiva.

## 5 Conclusioni e considerazioni finali

Il sistema realizzato soddisfa le specifiche prefissate nella fase di design. Come descritto nella [Prefazione](#), l'obiettivo era progettare una rete neurale capace di riconoscere lo stato di cottura delle pizze in uscita dal forno e di inviare i corrispondenti segnali al sistema di controllo, automatizzando l'intero processo senza intervento dell'operatore.

L'intero flusso — dall'acquisizione del fotogramma con la camera CSI sul Raspberry Pi Zero 2 W, al preprocessing, all'inferenza con il motore TFLite, fino alla trasmissione del risultato via wireless — garantisce un ingombro ridotto, un consumo contenuto e una velocità di elaborazione di almeno 3 inferenze al secondo (latency media di circa 230 ms per inferenza), in linea con i vincoli di sistema.

**Limiti e possibili miglioramenti** Durante la validazione pratica è emerso che il riconoscimento ottimale richiede che la pizza si trovi a distanza fissa (nei test su immagini stampate su fogli A4, circa 20cm) e in primo piano rispetto alla camera. Questo vincolo può risultare restrittivo in ambienti reali. Per superare questa limitazione, si propongono due possibili strategie:

- **Estensione del dataset:** includere nella fase di raccolta immagini di pizze riprese a diverse distanze e inserite in contesti più ampi (scene di forno complete).
- **Aggiornamento dell'ottica:** montare un obiettivo con nitidezza superiori, in grado di fornire immagini più definite anche a distanze variabili. Ciò faciliterebbe l'estrazione delle feature dalla rete e ridurrebbe ulteriormente eventuali errori di classificazione.

Infine, si potrebbe sviluppare un'architettura CNN personalizzata, più leggera di EfficientNet-B0, con l'obiettivo di ridurre la latenza di inferenza e il consumo energetico. La sfida consisterebbe nel bilanciare l'efficienza computazionale con un livello di accuratezza ancora accettabile.

## Riferimenti bibliografici

- [1] Keras Team. *Keras Applications - Usage examples for image classification models*. 2024. URL: <https://keras.io/api/applications/#usage-examples-for-image-classification-models>.
- [2] Jia Deng et al. «ImageNet: A Large-Scale Hierarchical Image Database». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2009, pp. 248–255. DOI: [10.1109/CVPR.2009.5206848](https://doi.org/10.1109/CVPR.2009.5206848). URL: <https://image-net.org>.
- [3] Mingxing Tan e Quoc V. Le. «EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks». In: *Proceedings of the 36th International Conference on Machine Learning (ICML)*. A cura di Kamalika Chaudhuri e Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, giu. 2019, pp. 6105–6114. URL: <https://proceedings.mlr.press/v97/tan19a.html>.
- [4] libcamera Project. *libcamera - A Complex Camera Support Library for Linux*. 2024. URL: <https://libcamera.org>.
- [5] Gary Bradski. «The OpenCV Library». In: *Dr. Dobb's Journal of Software Tools*. 2000. URL: <https://opencv.org>.
- [6] Google Developers. *Build TensorFlow Lite for Arm Cortex*. 2024. URL: <https://ai.google.dev/edge/litet/build/arm>.