TensorFlow Lite - based TinyML implementation in X-HEEP

Special Project per il corso di Architetture dei Sistemi di Elaborazione

Eduard Antonovic Occhipinti

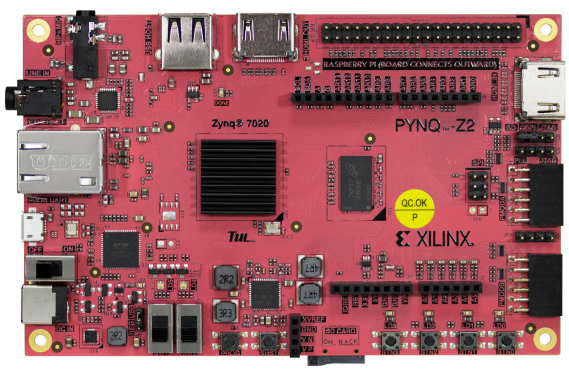
2024-10-04

Politecnico di Torino

Outline

Introduzione

- Il progetto si presuppone di studiare le tecniche disponibili attualmente per la compressione di modelli pre-esistenti in maniera da renderli adatti all'esecuzione su dispositivi embedded.
  - In particolare l'esecuzione su una board `RISC-V`: la `PYNQ`



- Il progetto ha fornito inoltre un'opportunità per imparare a utilizzare una libreria di machine learning (TensorFlow) e mettere mano su teniche più avanzate di manipolazione di modelli.

Outline

TensorFlow

TensorFlow è una libreria open-source per il machine learning sviluppata da Google.

- Fornisce un'interfaccia per la costruzione di modelli ed il training
- Supporta varie tecniche di compressione per ridurre la dimensione dei modelli

Keras

Una `API` di alto livello per la costruzione di modelli in TensorFlow.

TensorFlow Lite

Uno stack software per eseguire modelli TensorFlow su dispositivi embedded (in particolare col sottoinsieme di librerie denominate TensorFlow Lite for Microcontrollers).

TensorFlow Model Optimization Toolkit

Il *TensorFlow Model Optimization Toolkit* è una suite di strumenti utili per la compressione di modelli TensorFlow.

È stato utilizzato in maniera estensiva per il progetto. Prima di arrivare al risultato finale ho deciso di esplorare tutte le tecniche offerte tal toolkit:

- **Quantization**
- **Pruning**
- **Weights Clustering**
- **Collaborative Optimizations**

Quantization

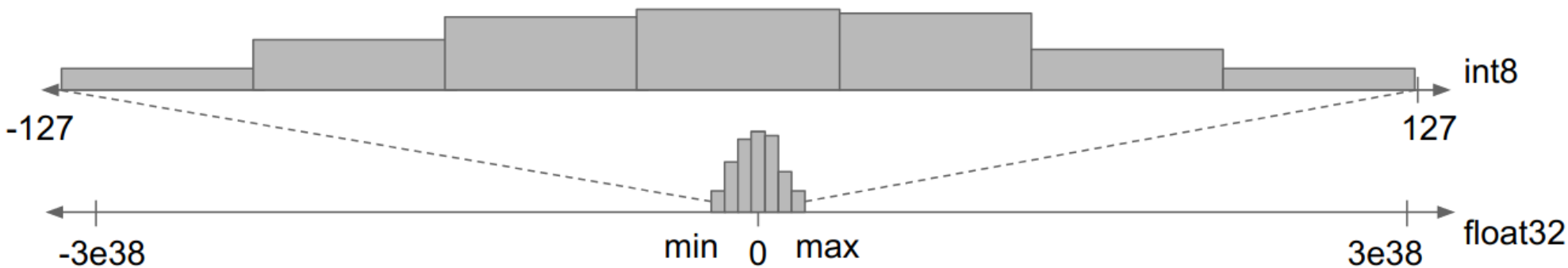


Figura 2: Conversione «lossy» da `FP32` a `INT8`

Quantization

Quantization Aware Training

- Introduce l'errore di quantizzazione come rumore durante il training e come parte della loss, che l'algoritmo di ottimizzazione cerca di minimizzare
- Di conseguenza, il modello apprende parametri più robusti alla quantizzazione
- Sempre opportuno applicarlo ove possibile

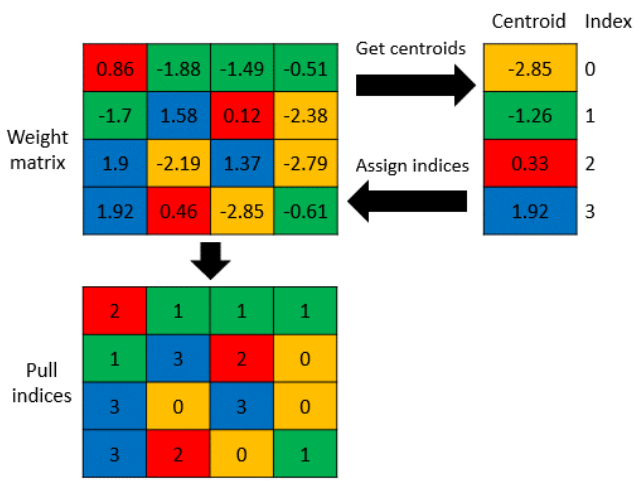
Post-training Quantization

- Classica quantizzazione che comprime i parametri del modello finale per ridurne la dimensione
- Classico caso è la quantizzazione da `FP32` a `INT8` ma un ampio range di possibilità esiste, come `FP16`, `INT4` o addirittura quantizzazione a 1 bit! [1]

[1] S. Ma *et al.*, «The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits». [Online]. Disponibile su: https://arxiv.org/abs/2402.17764

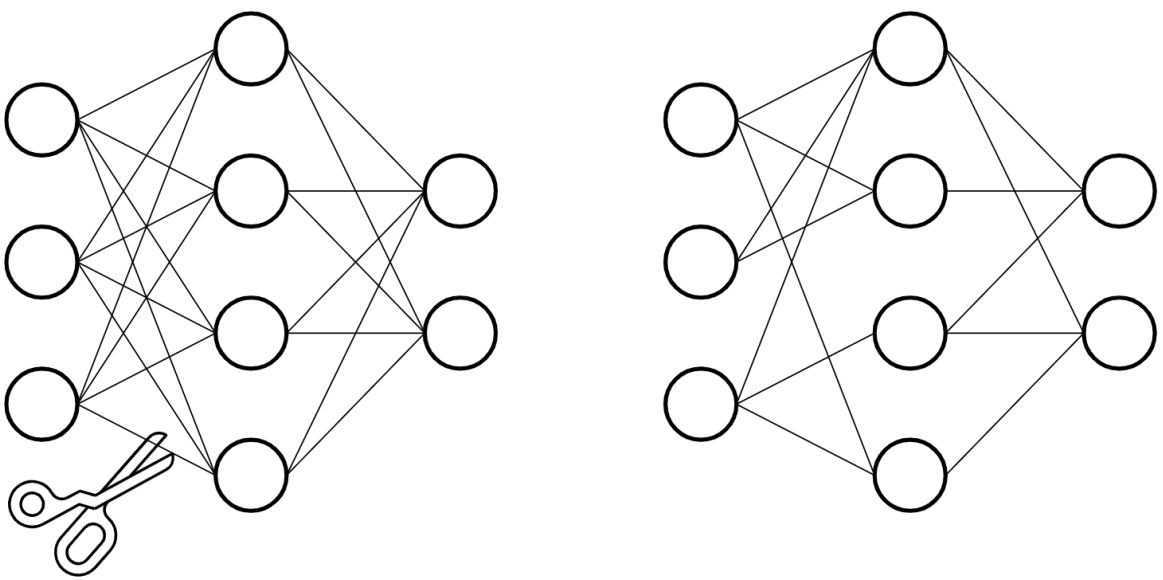
Weights Clustering

Consiste nel raggruppare i pesi in cluster ed approssimarli al centroide del cluster in maniera tale da diminuire il numero di valori distinti e ridurre la dimensione del modello.



Pruning

Consiste nel rimuovere i pesi che sono stati identificati come non importanti.

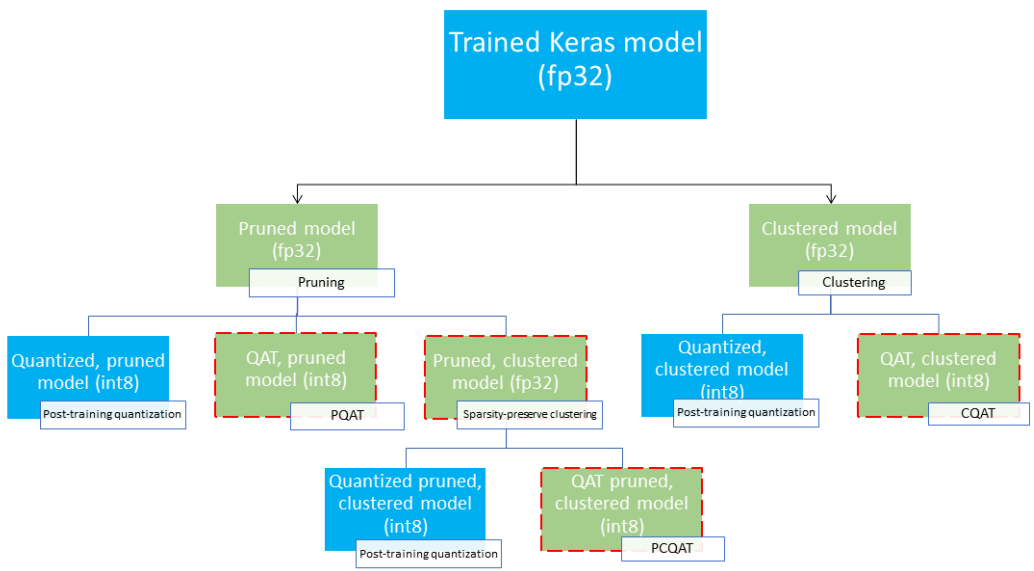


Before pruning

After pruning

Collaborative Optimizations

Consiste nel combinare le varie tecniche di ottimizzazione in maniera tale che non interferiscano le une con le altre.



Outline

LeNet-5

Per iniziare a prendere mano con TensorFlow, è stato scelto di implementare un modello classico per architettura `x86`: `LeNet-5`

Per prendere la mano anche con i diversi tipi di modelli supportati da TensorFlow e rendersi meglio conto delle limitazioni e vantaggi di ciascuno di essi, questi è stato implementato in tre versioni:

- Un modello **sequenziale**
- Un modello **funzionale**
- Un modello **sottoclassato**

Modello sequenziale

È il modello più semplice da implementare, meno flessibile ma meglio supportato.

I modelli sequenziali permettono una più facile manipolazione dei layer e sono più adatti ad ottimizzazioni iterative.

- Modello scelto per le studiare le varie ottimizzazioni disponibili

```python
model = keras.models.Sequential(
    [
        keras.layers.Flatten(input_shape=(32, 32, 1)),
        keras.layers.Dense(128, activation="relu"),
        keras.layers.Dropout(0.2),
        keras.layers.Dense(10),
    ]
)
```

Modello funzionale

I modelli funzionali sono più flessibili e permettono di creare modelli più complessi, con possibilità di creare topology non lineari, multipli input e output, etc.

```python
inputs = keras.Input(shape=(32, 32, 1))

x = layers.Conv2D(6, 5, activation="tanh")(inputs)
x = layers.AveragePooling2D(2)(x)
x = layers.Activation("sigmoid")(x)
x = layers.Conv2D(16, 5, activation="tanh")(x)
x = layers.AveragePooling2D(2)(x)
x = layers.Activation("sigmoid")(x)
x = layers.Conv2D(120, 5, activation="tanh")(x)
x = layers.Flatten()(x)
x = layers.Dense(84, activation="tanh")(x)

outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Modello sottoclassato

Permette alta flessibilità e controllo, ma è più complesso da implementare e meno supportato.

```python
class Lenet(keras.Model):
  def __init__(self):
    super().__init__()

    self.conv1 = self._make_conv_layer(6, 5, input_shape=[32, 32, 1])
    self.conv2 = self._make_conv_layer(16, 5)
    self.conv3 = self._make_conv_layer(
      120, 5, pooling=False, flatten=True
    )
    self.dense1 = layers.Dense(84, activation="tanh")
    self.dense2 = layers.Dense(10, activation="softmax")

  def call(self, inputs):
    x = self.conv1(inputs)
    x = self.conv2(x)
    x = self.conv3(x)
    x = self.dense1(x)
    x = self.dense2(x)

    return x
```

```python
  def _make_conv_layer(
    self, filters, kernel_size, ish=None, pooling=True, flatten=False
):
    l = keras.Sequential()

    if input_shape is not None:
      l.add(
        layers.Conv2D(
          filters, kernel_size, activation="tanh", ish=input_shape
        )
      )
    else:
      l.add(layers.Conv2D(filters, kernel_size, activation="tanh"))
    if pooling:
      l.add(layers.AveragePooling2D(2))
      l.add(layers.Activation("sigmoid"))
    if flatten:
      l.add(layers.Flatten())

    return l
```

Outline

YOLOv8

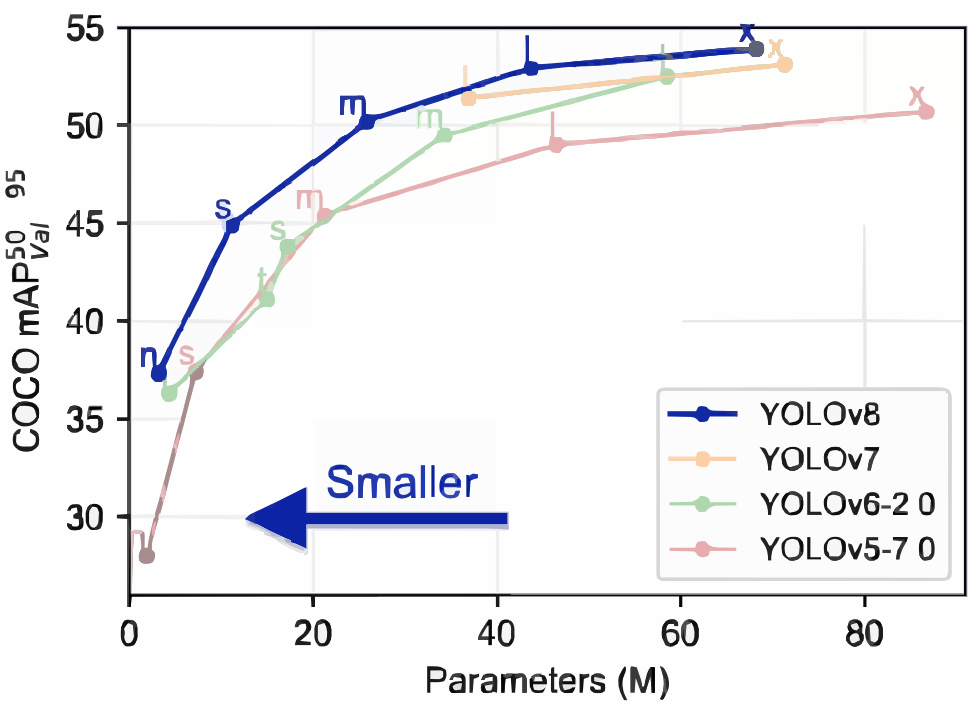
Un primo tentativo di applicare le tecniche studiate è stato fatto su un modello di object detection: YOLOv8.

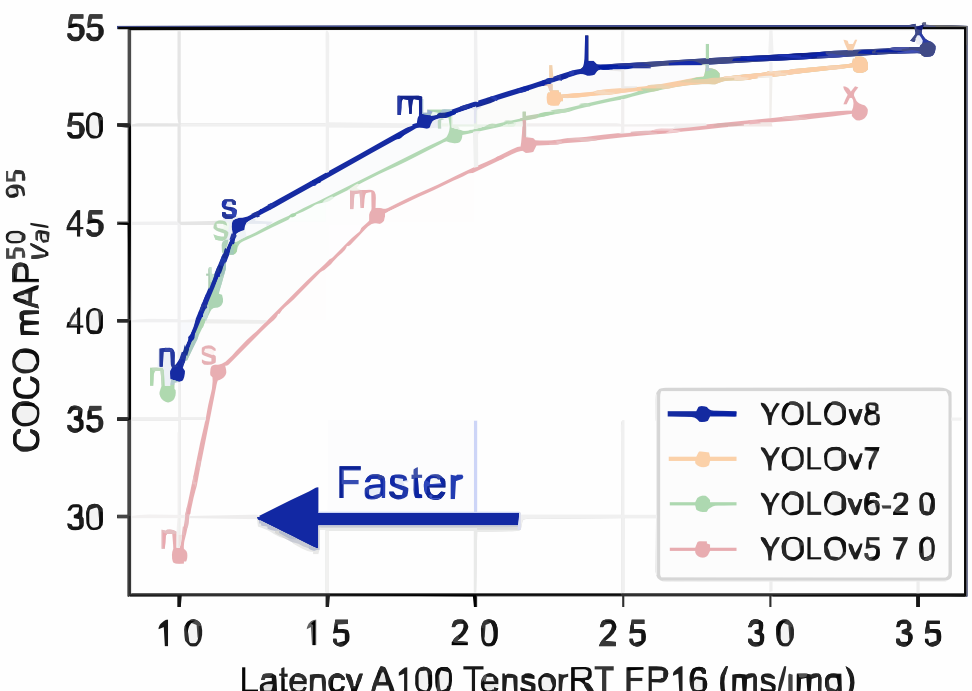
- Al momento dei miei esperimenti questi corrispondeva allo stato dell'arte in termini di performance e dimensione ma nel frattempo è stato superato con due nuove versioni [1] [2] (la ricerca in questo campo è molto attiva).
- Mentre scrivevo questa slide un'altra versione, YOLO11 è stata rilasciata.

[1] C.-Y. Wang, I.-H. Yeh, e H.-Y. M. Liao, «YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information». [Online]. Disponibile su: https://arxiv.org/abs/2402.13616

[2] A. Wang *et al.*, «YOLOv10: Real-Time End-to-End Object Detection». [Online]. Disponibile su: https://arxiv.org/abs/2405.14458

In particolare con il checkpoint `yolov8n`, il più piccolo tra quelli dispoinibili come possiamo vedere nel seguente grafico.





Quantizzazione

Una prima prova è stata fatta cercando di quantizzare a `INT8` il modello `torch` (post-training)

Risultati

- Riduzione della dimensione del modello dal checkpoint da 6.2 MB (o 12.2 MB se consideriamo il modello `.onnx` usato come step intermedio per la conversione) a 3.34 MB
- Perdita di performance `mAP50` e `mAP50-95` molto bassa

```python
from ultralytics import YOLO

results_quant = YOLO(baseline_quantized).val(data="coco128.yaml")
```

ORIGINAL_FP32

```
{
|   'metrics/precision(B)': 0.6401136562982888,
|   'metrics/recall(B)': 0.5371329744029286,
|   'metrics/mAP50(B)': 0.6049606058248067,
|   'metrics/mAP50-95(B)': 0.4455822678794395,
|   'fitness': 0.4615201016739762
}
```

QUANTIZED_INT8

```
{
|   'metrics/precision(B)': 0.6693744628674021,
|   'metrics/recall(B)': 0.5357106358126033,
|   'metrics/mAP50(B)': 0.6089456673483312,
|   'metrics/mAP50-95(B)': 0.4540750785376153,
|   'fitness': 0.4695621374186869
}
```

Pruning

Un tentativo di ridurre ulteriormente la dimensione finale del modello è stata fatta utilizzando il pruning di torch.

- Non essendo tutti i layer compatibili ho isolato solo i layer di tipo `Conv2d`.

```python
import torch
import torch.nn as nn
import torch.nn.utils.prune as prune
import YOLO

model = YOLO("yolov8n.pt")
PRUNING_AMOUNT = 0.1

for name, m in model.named_modules():
  if isinstance(m, nn.Conv2d):
      prune.l1_unstructured(m, name="weight", amount=PRUNING_AMOUNT)
      prune.remove(m, "weight")  # make permanent
```

- Purtroppo i risultati rendono il modello molto meno efficace dell'originale.

PRUNED

```
{
|   'metrics/precision(B)': 0.48268779643490495,
|   'metrics/recall(B)': 0.42881404068150936,
|   'metrics/mAP50(B)': 0.462184408704583,
|   'metrics/mAP50-95(B)': 0.3170316451700596,
|   'fitness': 0.33154692152351195
}
```

Altri appunti

- Il metodo consigliato da `ultralytics` per il pruning del modello al momento dei miei test non funzionava correttamente (https://github.com/ultralytics/ultralytics/issues/3507)
- Il metodo descritto da https://github.com/VainF/Torch-Pruning non era compatibile con tutti i layer del modello

Outline

FastestDet

Dato gli scarsi risultati nel cercare di comprimere un modello già estremamente efficiente quale YOLOv8 senza comprometterne in maniera significativa le performance, ho esplorato altre opzioni finendo per trovare un modello con appena 250K parametri, FastestDet

Retraining

Una delle strategie adottate per ridurre la dimensione è stata quella di provare a re-trainare il modello con immagini più piccole e ad andare di conseguenza anche a modificare i vari layer a livello di codice.

Diff

```
 DATASET:
-  TRAIN: "/home/qiuqiu/Desktop/coco2017/train2017.txt"
-  VAL: "/home/qiuqiu/Desktop/coco2017/val2017.txt"
+  TRAIN: "-"
+  VAL: "-"
   NAMES: "configs/coco.names"
 MODEL:
   NC: 80
-  INPUT_WIDTH: 352
-  INPUT_HEIGHT: 352
+  INPUT_WIDTH: 128
+  INPUT_HEIGHT: 128
...
```

Retraining

Problemi nell'adattare il codice

- I `requirements.txt` non erano scitti in maniera corretta e causavano conflitti tra le dipendenze, questo ha comportato il dover adattare il codice per far sì che funzionasse con le nuove versioni di `torch`, linguaggio in cui il modello è implementato.
- Purtroppo anche dopo questo il training non è andato a buon fine in quanto il dataset originale (che come abbiamo visto in precedenza era hardcodato) non era in formato `coco` standard.

Manipolazione modello originale

Ho quindi deciso di riutilizzare il checkpoint in formato `.onnx` fornito dall'autore.

```python
import tempfile
import onnx2tf

saved_model_fastest_det = tempfile.mkdtemp()

onnx2tf.convert(
    ONNX_PATH,
    output_integer_quantized_tflite=True,
    output_h5=True,
    output_folder_path=saved_model_fastest_det,
)
```

- La decisione di usare `.h5` come formato rende la conversione più robusta
- È possibile effettuare quantizzazione già in questo step di conversione

```python
import numpy as np

def representative_dataset():
    for _ in range(100):
        data = np.random.rand(1, 352, 352, 3)
        yield [data.astype(np.float32)]
```

```python
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model(
    saved_model_fastest_det
)

converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset
converter.target_spec.supported_ops = [
    tf.lite.OpsSet.TFLITE_BUILTINS_INT8
]
converter.inference_input_type = tf.uint8
converter.inference_output_type = tf.uint8
```

Risultati

Quantizzando abbiamo ridotto la dimensione del modello da 1.2 MB a 0.42 MB, riuscendo con successo a scendere sotto il nostro target di 0.5 MB.

- Il modello riesce ad operare anche su immagini scalate, senza bisogno di re-training

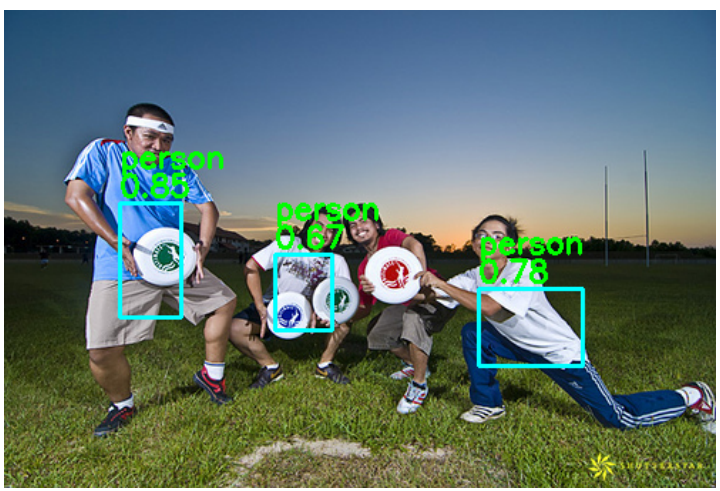


Figura 7: Inference su immagine originale

Figura 8: Inference su immagine 128x86

Conversione a `header file`

Andiamo adesso a convertire il modello in un `header file` in maniera tale da poterlo usare direttamente in un progetto `C`/`C++` per effettuare inference. Per fare ciò ho scritto due piccole funzioni, un' alternativa poteva essere quella di usare `xxd -i [file]` ma fare la conversione in maniera programmatica permette di avere più controllo sul risultato finale.

```python
def c_style_hexdump(input, ouput, name):
  with open(input, "rb") as f:
    file = f.read()

  file = bytearray(file)
  _bytes = [f"0x{x:02x}" for x in file]
  file = ",".join(_bytes)

  with open(ouput, "w") as f:
    f.write("#pragma once\n")
    f.write("#include <stdalign.h>\n")
    f.write(f"alignas(16) const unsigned char {name}[] = {{{file}}};")

  return len(_bytes)
```

```python
def build_header(output, names_with_sizes):
  with open(output, "w") as f:
    f.write('#pragma once\n#ifdef __cplusplus\nextern "C"\n{\n#endif\n')
    f.write("#include <stdalign.h>\n\n")

    for name, size in names_with_sizes:
      f.write(
        f"alignas(16) extern const unsigned char {name}[{size}];\n"
      )

    f.write("\n#ifdef __cplusplus\n}\n#endif\n")
```

Conversione a `header file`

```python
MODEL = "models/fastest_det_rom.c"
INPUT = "models/fastest_det_input.c"
TEST_IMAGE = "fastest_det/data/3.jpg"
HEADER = "models/fastest_det.h"
```

```python
model_size = c_style_hexdump(
  fastest_det_quant_file, MODEL, "model_data"
)
```

Codice 1: Generiamo il dump del modello, noteremo che la dimensione del file `C` generato è di ben 2.1 MB, ciò è causato da byte a zero di padding, che non influiscono sulla dimensione del modello caricato in memeoria

```python
from PIL import Image

img = Image.open(TEST_IMAGE)
img = img.convert("RGB")

img_array = np.array(img)
img_array = img_array.astype(np.uint8)

img.thumbnail((128, 128))
img_array_compressed = np.array(img)
img_array_compressed = img_array_compressed.astype(np.uint8)

_, imagebin = tempfile.mkstemp(".bin")

with open(imagebin, "wb") as f:
    f.write(bytearray(img_array_compressed))  # type: ignore

input_size = c_style_hexdump(imagebin, INPUT, "input_data")
```

Codice 2: Comprimiamo l'immagine e la salviamo come file `C`

Conversione a `header file`

Generiamo infine l'header file che verrà incluso nel codice sorgente con `build_header(HEADER, [("input_data", input_size), ("model_data", model_size)])`.

```c
#pragma once
#include <stdalign.h>
alignas(16) const unsigned char input_data[] = {
  0x19, 0x47, 0x6b, 0x1c, 0x4a, 0x6e, 0x1d,
  ...
};
```

Codice 3: `models/fastest_det_input.c`

```c
#pragma once
#include <stdalign.h>
alignas(16) const unsigned char model_data[] = {
  0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c,
  ...
};
```

Codice 4: `models/fastest_det_rom.c`

```c
#pragma once

#ifdef __cplusplus
extern "C"
{
#endif

#include <stdalign.h>

alignas(16) extern const unsigned char input_data[33024];
alignas(16) extern const unsigned char model_data[420000];

#ifdef __cplusplus
}
#endif
```

Codice 5: `models/fastest_det.h`

Utilizzo

Per utilizzare il modello, dobbiamo scrivere il codice `C++` relativo (che utilizzerà le librerie di `TensorFlow Lite (Micro)` in maniera appropriata).

```cpp
#pragma message "hello_world_test.cc"
extern "C"
{
#include "fastest_det_test.h"
#include <math.h>
#include <stdio.h>
}

#include "models/fastest_det.h"
#include "tensorflow/lite/core/c/common.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/micro_log.h"
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "tensorflow/lite/micro/micro_profiler.h"
#include "tensorflow/lite/micro/recording_micro_interpreter.h"
#include "tensorflow/lite/micro/system_setup.h"
#include "tensorflow/lite/schema/schema_generated.h"

namespace
{
constexpr int kTensorArenaSize = 0x70000;
uint8_t tensor_arena[kTensorArenaSize];
const tflite::Model *model = nullptr;

using FastestDetOpResolver = tflite::MicroMutableOpResolver<11>;

TfLiteStatus RegisterOps(FastestDetOpResolver &op_resolver)
{
    TF_LITE_ENSURE_STATUS(op_resolver.AddQuantize());
    TF_LITE_ENSURE_STATUS(op_resolver.AddPad());
    TF_LITE_ENSURE_STATUS(op_resolver.AddDepthwiseConv2D());
    TF_LITE_ENSURE_STATUS(op_resolver.AddConv2D());
    TF_LITE_ENSURE_STATUS(op_resolver.AddConcatenation());
    TF_LITE_ENSURE_STATUS(op_resolver.AddTranspose());
    TF_LITE_ENSURE_STATUS(op_resolver.AddReshape());
    TF_LITE_ENSURE_STATUS(op_resolver.AddGather());
```

```
37      TF_LITE_ENSURE_STATUS(op_resolver.AddAveragePool2D());
38      TF_LITE_ENSURE_STATUS(op_resolver.AddLogistic());
39      TF_LITE_ENSURE_STATUS(op_resolver.AddSoftmax());
40      return kTfLiteOk;
41  }
42  } // namespace
43
44  TfLiteStatus load_model()
45  {
46      if (model != nullptr) { return kTfLiteOk; }
47      model = ::tflite::GetModel(model_data);
48      TFLITE_CHECK_EQ(model->version(), TFLITE_SCHEMA_VERSION);
49      return kTfLiteOk;
50  }
51
52  TfLiteStatus Infer(const char *data, size_t len, int8_t **out, size_t
    *out_len)
53  {
54      if (model == nullptr) { return kTfLiteError; }
55      FastestDetOpResolver op_resolver;
56      TF_LITE_ENSURE_STATUS(RegisterOps(op_resolver));
57
58      tflite::MicroInterpreter interpreter(
59          model, op_resolver, tensor_arena, kTensorArenaSize);
60      TF_LITE_ENSURE_STATUS(interpreter.AllocateTensors());
```

```
61      TfLiteTensor *input = interpreter.input(0);
62      TFLITE_CHECK_NE(input, nullptr);
63      TfLiteTensor *output = interpreter.output(0);
64      TFLITE_CHECK_NE(output, nullptr);
65      memcpy(input->data.int8, data, len);
66      TF_LITE_ENSURE_STATUS(interpreter.Invoke());
67      *out = output->data.int8;
68      *out_len = output->bytes;
69
70      return kTfLiteOk;
71  }
72
73  extern "C" int init_tflite()
74  {
75      tflite::InitializeTarget();
76      TF_LITE_ENSURE_STATUS(load_model());
77      return kTfLiteOk;
78  }
79
80  extern "C" int
81  infer(const char *data, size_t len, int8_t **out, size_t *out_len)
82  {
83      return Infer(data, len, out, out_len);
84  }
```

Grazie dell’attenzione