Riconoscimento efficiente di immagini di animali tramite transfer learning e quantizzazione

ML project a.y. 2024/2025

Stefano Belli, matricola 0350116

Università degli Studi di Roma "Tor Vergata"

Agenda

1. Il problema da affrontare
2. Transfer learning
3. DNNs e dispositivi embedded
4. Quantizzazione
   - Quantizzazione post-training
5. Split del dataset
6. Architettura dei modelli analizzati
7. Training
8. Conversione in modelli TFLite
9. Evaluation dei modelli
   - Metodo evaluate
10. Risultati ottenuti

Il problema da affrontare

Si richiede di progettare un classificatore di immagini di animali tramite modelli preaddestrati, sfruttando il **transfer learning** e la **quantizzazione**.

Per questo progetto, è importante tenere conto sia dell'**accuratezza** del modello che dei **costi computazionali**.

Verranno confrontati due modelli con e senza quantizzazione tenendo conto delle seguenti metriche:

- Loss e accuracy del modello
- Tempo di inferenza del modello
- Grandezza del modello ottenuto

Transfer learning

Addestrare modelli di deep learning in modo efficiente non è affatto semplice:

1. Hardware potente e costoso richiesto
2. Tempi di addestramento del modello elevati
3. Costi per l'energia eccessivi
4. Mancanza di dati per l'addestramento
5. Progettare una rete da zero

Applicazione

E' possibile sfruttare modelli preaddestrati complessi, già testati e perfettamente funzionanti e "trasferirli" al nostro problema congelandone i pesi (*"trained weights"*), lasciando "addestrabili" solo i pesi di una rete neurale densa che è il classificatore rimpiazzato dal nostro.

DNNs e dispositivi embedded

Pensiamo all'IoT e alla diffusione capillare di dispositivi embedded, ad esempio una telecamera: e se volessimo integrare una rete convoluzionale nel device stesso?

- Preserveremmo la privacy dell'utente
- Niente problemi di latenze elevate o legate al trasferimento dati
- Se la telecamera viene disconnessa da internet, la rete può continuare a svolgere il suo compito, rendendo il dispositivo più affidabile

Il problema nell'eseguire le reti neurali su tali dispositivi è ovvia: la poca potenza a disposizione impatta sui tempi di **inferenza** del modello e sulla **memorizzazione** (sia in memoria primaria che secondaria) dei pesi del modello, oltre al fatto che il dispositivo potrebbe non avere capacità di calcolo in virgola mobile (es. non ha una FPU o ISA che supporti operazioni floating point).

Quantizzazione

Una tecnica che consente di ridurre la dimensione dei parametri di un modello:

- Meno **storage** richiesto per mantenere i parametri della rete
- Reappresentare un parametro da `float` $\rightarrow$ `int8_t` significa niente operazioni floating point e quindi **tempi** di inferenza minori

La tecnica non impatta significativamente sull'accuratezza del modello originale: l'alta precisione di un `float` o `double` probabilmente non è necessaria per far si che il modello svolga bene il suo lavoro.

Quantizzazione post-training

In particolare, nel progetto viene utilizzato TFLite/LiteRT e quantizzazione post-training che permette di definire un modello Keras, addestrarlo normalmente, e solo dopo quantizzarlo.

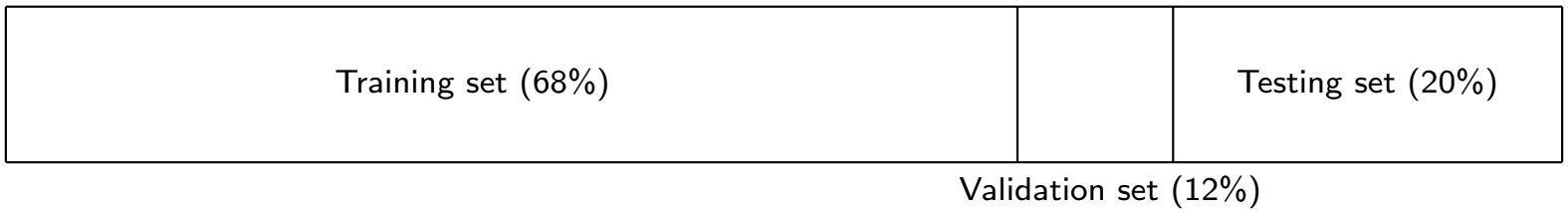
Dopo aver convertito il modello, è possibile applicare 3 livelli di quantizzazione (incrementale)

	Param. fissati	**Variabili**	**Tensori di I/O**
Dynamic range	✓	✗	✗
Float fallback	✓	✓	✗
Integer-only	✓	✓	✓

Tabella: ✓ indica che avviene la quantizzazione, ✗ indica che non avviene

Split del dataset

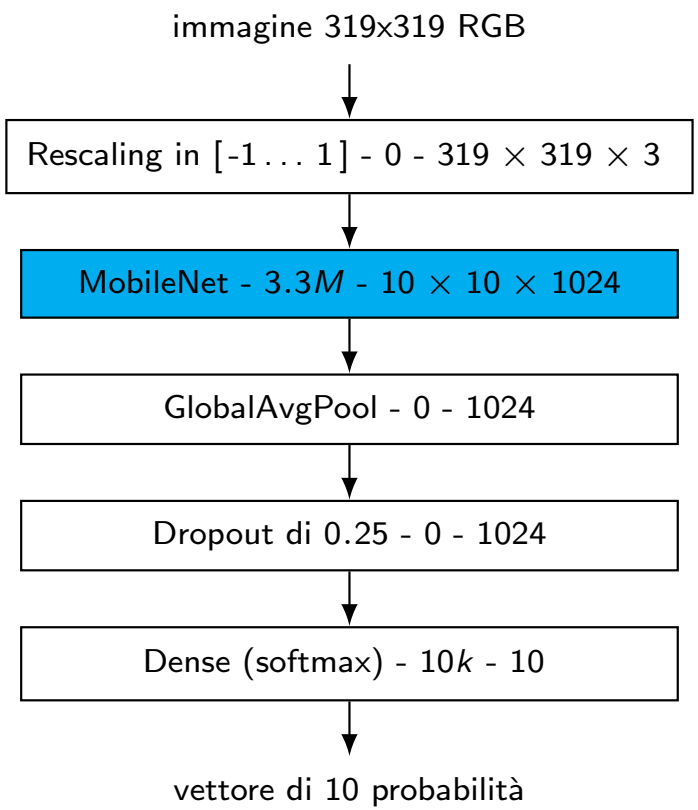
Il dataset fornito consiste in immagini di animali con etichette associate (10 classi), pronto per essere letto e splittato da `keras.utils.image_dataset_from_directory`.

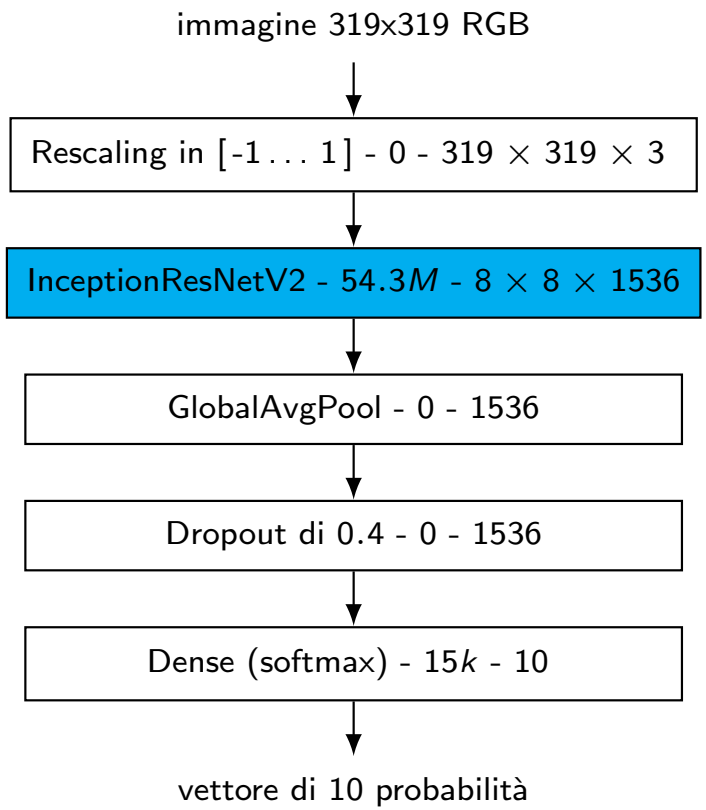


- Il **training set** viene utilizzato per l'addestramento dei modelli
- Il **validation set** viene usato ogni 3 epoche di addestramento e mostrare esempi di predizione
- Il **testing set** viene usato per effettuare le misurazioni finali

Architettura dei modelli analizzati

I due modelli pretrained scelti sono `MobileNet` e `InceptionResNetV2`, facilmente istanziati con i trained weights `imagenet` grazie a `keras.applications`





Training

```
# you may override this
def compile_model(self, *args, **kwargs):
  self.model.compile(
    optimizer=keras.optimizers.Adam(),
    loss=keras.losses.
     CategoricalCrossentropy(),
    metrics=[
      keras.metrics.CategoricalAccuracy()
    ])

# you may override this
def fit_model(self, *args, **kwargs):
  return self.model.fit(
    args[0],
    validation_data=args[1],
    validation_freq=3,
    epochs=40,
    callbacks=[
      keras.callbacks.EarlyStopping(
        start_from_epoch=3,
        patience=2,
        restore_best_weights=True),
      keras.callbacks.LearningRateScheduler(
        lambda e, l: l if e < 1 else l * np.
     exp(-0.15)
    )])
```

Compilazione del modello:

- Optimizer: adam
- Loss: categorical crossentropy
- Metrica d'interesse: categorical accuracy

E il suo addestramento:

- 40 epoche totali
- Validazione ogni 3 epoche
- Early stopping
  - Pazienza 2
  - Partenza da epoca 3
  - Ripristino pesi migliori
  - `val_loss` monitorata
- Decaying learning rate di $exp(-0.15)$ dopo la prima epoca

Conversione in modelli TFLite

- Dopo la definizione del modello e il suo addestramento, è possibile *convertire* il modello in tflite e avviare la *quantizzazione*.
- Nel progetto vengono provati tutti e 3 i livelli di quantizzazione offerti.
- Il modello viene salvato sul filesystem e quindi sarà possibile effettuare la valutazione in seguito usando le utils di `tf.lite` per caricare il modello in memoria e istanzizare un `tf.lite.Interpreter` su cui chiamare il metodo `invoke` per fare inferenza.
- Data l'immediatezza di utilizzo, non è necessario effettuare nessuna azione sul modello originale, pre o post quantizzazione, si può direttamente procedere con l'evaluation che ne risalterà i vantaggi

Evaluation dei modelli

Dopo la validazione dei modelli (cambiando l'input shape, il batch size, il decaying lr, il dropout e/o l'architettura di rete in generale) l'evaluation finale (sul testing set) viene effettuata tenendo conto sia del **costo computazionale** che dell'**accuracy** del modello.

Dato l'obiettivo dell'evaluation e:

- il fatto che vogliamo una stima dei costi che rifletta l'utilizzo reale
- il fatto che l'evaluate di tf/keras non fornisce informazioni affidabili sul tempo di inferenza
- l'evaluate di tf/keras processa elementi in batch
- l'utilizzo del profiler di tf non è necessario per il task
- il fatto che per tflite non esiste un metodo/funzione che misuri le prestazioni del modello, sia per quanto riguarda l'accuratezza che i costi computazionali...
- ...e quindi bisogna scrivere una funzione di evaluate da zero per tflite

Metodo evaluate

E' risultato conveniente scrivere un metodo evaluate univoco che misurasse **loss**, **accuracy**, **inference time** allo stesso modo sia per modelli tf/keras che tflite.

Per la loss e l'accuracy si utilizzano le classi:

- `keras.losses.CategoricalCrossentropy`
- `keras.metrics.CategoricalAccuracy`

in modalità standalone: aggregando e calcolando i risultati al termine (o in iterazione intermedia per logging).

Per riflettere l'utilizzo reale del modello, ai modelli tf viene passato un singolo elemento del testing set, invocato il metodo `keras.Model.__call__` e quindi misurato il tempo di inferenza con un timer ad alta risoluzione (`time.perf_counter`). Stessa cosa per i modelli tflite, solo che viene invocato il metodo `tf.lite.Interpreter.invoke` invece che la `__call__`. Dall'accumulo dei tempi di inferenza, si ricava facilmente il tempo di inferenza medio.

Risultati ottenuti

Loss e accuracy

	MN	IRNV2
NQ	0.1141, 0.9650	0.0804, 0.9805
TL	0.1141, 0.9650	0.0804, 0.9805
DR	0.1195, 0.9617	0.0807, 0.9803
FF	0.1492, 0.9608	0.1028, 0.9803
IO	0.1473, 0.9604	0.1034, 0.9795

Avg. inference time

	MN	IRNV2
NQ	60 ms	581 ms
TL	28 ms	314 ms
DR	35 ms	308 ms
FF	28 ms	278 ms
IO	28 ms	279 ms

Size [`get_file_size(weights_file)`]

	MN	IRNV2
NQ	12.67 MB	209.34 MB
TL	12.25 MB	207.24 MB
DR	3.24 MB	53.01 MB
FF	3.36 MB	53.90 MB
IO	3.36 MB	53.90 MB

Per entrambi i casi:

- Dimezzamento dei tempi di inferenza
  - Anche solo convertendo i modelli keras in tflite
- $\frac{1}{4}$ della size originale
- Prestazioni di accuratezza quasi intatte
  - `InceptionResNetV2` subisce meno l'effetto della quantizzazione sull'accuratezza

In conclusione, `MobileNet` è la scelta migliore per tempi di inferenza, grandezza del modello e accuratezza (quel $< 2\%$ in più non vale la pena), in particolare con quantizzazione integer-only.

Grazie per l'attenzione!