Progetto Sistemi Digitali

Andrea Borghesi - andrea.borghesi4@studio.unibo.it

July 2023

Abstract

L'obiettivo di questo progetto è quello di creare un'applicazione per dispositivi Android che fosse in grado di riconoscere le razze dei cani tramite la fotocamera del dispositivo. Per raggiungere questo scopo, si sono analizzate diverse tecniche di Computer Vision che hanno portato alla scelta del modello YOLOv5. Quest'ultimo è un modello pre-addestrato per l'*object detection* e l'*image classification* con più di 7 milioni di parametri. Tramite una fase di *transfer learning*, basata su un dataset individuato su Kaggle, si è insegnato al modello ad individuare le classi di nostro interesse. Tuttavia, la complessità del modello e il grande numero di classi da predire si sono verificati essere troppo complessi per la piattaforma Colab utilizzata per il training. Per cui, ci si è limitati a 25 classi, numero che permetteva di sfruttare la GPU di Colab e ha portato a un'accuratezza più che ragionevole.

1 Dataset

Il dataset utilizzato è denominato "*Stanford Dogs Dataset*" ed è disponibile su Kaggle. Esso contiene 20580 immagini di cani di 120 razze diverse. A ogni immagine è associato un file in formato PASCAL VOC XML che specifica le coordinate del *bounding box* dove si trova il cane e la sua razza. Per esempio:

```
<annotation>
	<folder>02085620</folder>
	<filename>n02085620_10074</filename>
	<source>
		<database>ImageNet database</database>
	</source>
	<size>
		<width>333</width>
		<height>500</height>
		<depth>3</depth>
	</size>
	<segment>0</segment>
	<object>
```

```
                <name>Chihuahua</name>
                <pose>Unspecified</pose>
                <truncated>0</truncated>
                <difficult>0</difficult>
                <bndbox>
                        <xmin>25</xmin>
                        <ymin>10</ymin>
                        <xmax>276</xmax>
                        <ymax>498</ymax>
                </bndbox>
        </object>
</annotation>
```

I cani presenti nelle foto, anche se della stessa razza, possono presentare colori e taglie diverse, ma soprattutto possono essere cuccioli o adulti, ciò può rappresentare una sfida notevole per il modello.

Dopo alcuni test di addestramento del modello, si è subito capito che non sarebbe stato possibile procedere con un numero di classi così elevato sulla piattaforma Colab, che mediamente offre delle GPU tra i 12 e i 15 GB e un tempo di esecuzione limitato (circa 5 ore con la GPU). Dunque, si sono filtrate le foto per trattenere solo quelle relative a 25 razze selezionate (tendezialmente le più celebri e comuni). L'applicazione finale sarà quindi in grado di riconoscere solamente le seguenti razze:

- Standard Poodle (barboncino standard)
- Miniature Poodle (barboncino nano)
- Toy Poodle (barboncino toy)
- Samoyed (samoyedo)
- Great Pyrenees (pastore dei Pirenei)
- Pug (carlino)
- Siberian Husky (husky siberiano)
- Saint Bernard (san Bernardo)
- Great Dane (alano)
- French Bulldog (bouledogue francese)
- Boxer
- Bernese Mountain Dog (bovaro del Bernese)
- Doberman

- German Shepherd (pastore tedesco)
- Rottweiler
- Border Collie
- Cocker Spaniel
- Labrador Retriever
- Golden Retriever
- Scottish Terrier (terrier scozzese)
- Standard Schnauzer (schnauzer medio)
- Beagle
- Basset (bassotto)
- Pekinese (pekinese)
- Chihuahua

Per preparare il dataset per la fase di addestramento bisogna suddividere le immagini e il *labels* in cartelle diverse, per poi spartirle casualmente in ulteriori cartelle per creare due insiemi: uno per il training e l'altro per la fase di validazione.

```
data_path = './data'
dataset_path = './data_for_training'
# build directory tree
if os.path.exists(dataset_path):
    shutil.rmtree(dataset_path)
os.mkdir(dataset_path)

os.mkdir(f'{dataset_path}/images')
os.mkdir(f'{dataset_path}/labels')
os.mkdir(f'{dataset_path}/images/train')
os.mkdir(f'{dataset_path}/labels/train')
os.mkdir(f'{dataset_path}/images/test')
os.mkdir(f'{dataset_path}/labels/test')

annotation_path = f'{data_path}/annotations/Annotation'
images_path = f'{data_path}/images/Images'

train_images_path = f'{dataset_path}/images/train'
train_labels_path = f'{dataset_path}/labels/train'
```

Successivamente, si sono quindi popolate le varie cartelle con le immagini e le informazioni relative alle annotazioni.

```
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
from PIL import Image, ImageDraw
import xml.etree.ElementTree as ET

label = 0
for breed_dir in os.listdir(annotation_path):
    breed = breed_dir[breed_dir.index('-')+1:]
    if breed not in breeds_to_use: continue
    breed_label[label] = breed
    # Loop images and annotations by breed
    for file in os.listdir(annotation_path + '/' + breed_dir):
      img = cv.imread(f'{images_path}/{breed_dir}/{file}.jpg')
      # annotation in PASCAL VOC format
      tree = ET.parse(f'{annotation_path}/{breed_dir}/{file}')
      # parse bounding box coordinates
      xmin = int(tree.getroot().findall('object')[0].find('
          ↪ bndbox').find('xmin').text)
      xmax = int(tree.getroot().findall('object')[0].find('
          ↪ bndbox').find('xmax').text)
      ymin = int(tree.getroot().findall('object')[0].find('
          ↪ bndbox').find('ymin').text)
      ymax = int(tree.getroot().findall('object')[0].find('
          ↪ bndbox').find('ymax').text)
      h, w, _ = img.shape
      b_center_x = ((xmin + xmax) / 2) / w
      b_center_y = ((ymin + ymax) / 2) / h
      b_width = (xmax - xmin) / w
      b_height = (ymax - ymin) / h
      cv.imwrite(f'{train_images_path}/{file}.jpg', img)
      with open(f'{train_labels_path}/{file}.txt', 'w+') as f:
          f.write(f'{label} {b_center_x} {b_center_y} {b_width
              ↪ } {b_height}')
          f.close()
    label += 1
```

Infine, si è optato per spostare casualmente il 15% delle istanze nelle cartella utilizzare per la fase di validazione.

```
import random
test_images_path = f'{dataset_path}/images/test'
test_labels_path = f'{dataset_path}/labels/test'

unique_filename = set()
for file in os.listdir(train_images_path):
    unique_filename.add(file.split('.')[0])

unique_filename = list(unique_filename)
random.shuffle(unique_filename)

test_set_size = 0.15
test_filename = unique_filename[ -round(test_set_size*len(
    ↪ unique_filename)): ]

for filename in test_filename:
    shutil.move(f'{train_labels_path}/{filename}.txt', f'{
        ↪ test_labels_path}/{filename}.txt')
    shutil.move(f'{train_images_path}/{filename}.jpg', f'{
        ↪ test_images_path}/{filename}.jpg')
```

2 YOLOv5

L'architettura **YOLO** (*You Only Look Once*) è un algoritmo di rilevamento degli oggetti in tempo reale. A differenza dei metodi tradizionali, YOLO affronta il problema della *object detection* come un problema di regressione, prevedendo direttamente le *bounding box* e le probabilità di appartenza alla classi da un'immagine di input.

Nei modelli tradizionali, si utilizza comunemente un approccio a due fasi: la prima fase prevede la generazione di potenziali candidati a bounding box per gli oggetti presenti nell'immagine, i quali vengono successivamente inviati alla seconda fase, in cui gli oggetti vengono classificati e raffinati.

Formulando il rilevamento degli oggetti come un problema di regressione, YOLO elimina la necessità di fasi separate, ottenendo una pipeline più semplice ed efficiente. Ciò consente a YOLO di elaborare le immagini in maniera più rapida ed efficiente.

YOLOv5 è una versione sviluppata da Ultralytics nel 2019. Rappresenta un'implementazione snella e leggera, mirata all'efficienza e alla velocità.
La rete neurale di YOLOv5 è basata sull'architettura dell'encoder-decoder, che sfrutta una rete neurale convoluzionale (CNN) pre-addestrata come estrattore di features. Questo permette a YOLOv5 di apprendere rappresentazioni di alto

livello dalle immagini di input.
L'encoder utilizza diversi livelli di convoluzione per estrarre progressivamente le caratteristiche dell'immagine. Mentre la parte finale della rete neurale è composta da layer di convoluzione e fully connected che generano le predizioni delle *bounding box* e delle probabilità di classe.

2.1 Data Augmentation

Il modello pubblicato da Ultralytics esegue in automatico il processo di *data augmentation* per aumentare il numero di istanze di training modificandone alcune caratteristiche. Questo dovrebbe portare il modello a generalizzare meglio. Lo script che controlla questa fase si può trovare in yolov5/utils/augmentations.py nella loro libreria su GitHub. Per tentare di migliorare l'addestramento del modello, si sono eseguiti vari test andando a modificare il seguente array, che contiene le operazioni di data augmentation e gli iperparametri associati a ognuna di esse.

```
T = [
    A.RandomResizedCrop(
        height=size, width=size, scale=(0.8, 1.0),
        ratio=(0.9, 1.11), p=0.0),
    A.Blur(p=0.01),
    A.MedianBlur(p=0.01),
    A.ToGray(p=0.01),
    A.CLAHE(p=0.01),
    A.RandomBrightnessContrast(p=0.0),
    A.RandomGamma(p=0.0),
    A.ImageCompression(quality_lower=75, p=0.0)]
```

2.2 Transfer learning

Il **transfer learning** è un metodo utile per riaddestrare rapidamente un modello su nuovi dati senza dover riaddestrare l'intera rete. Parte dei pesi iniziali viene bloccata (*freeze*) e il resto dei pesi viene utilizzato per calcolare la loss. Ciò richiede meno risorse rispetto all'addestramento normale e consente tempi di addestramento più veloci, sebbene possa comportare una riduzione dell'accuratezza finale del modello addestrato.

I parametri che si sono impiegati sono i seguenti:

```
model_pretrained = 'yolov5s' # or 'yolov5m'
name = 'yolo_dogs_breeds'
yaml_file_name = f'yolov5/{name}.yaml'
image_size = 640
train_file_path = 'yolov5/train.py'
cfg_file_path = 'yolov5/models/yolov5s.yaml'
hyp_file_path = 'yolov5/data/hyps/hyp.scratch-med.yaml'
batch_size = 32 if image_size == 640 else 64
epochs = 100
data_file_path = yaml_file_name
weight_file_path = f'{model_pretrained}.pt'
workers = 20
```

Sempre considerando le limitazioni dovute a Colab, come modello pre-addestrato si è impiegato "yolov5s", che è la versione *small*. Analogamente, la *batch_size* dipende dalle dimensioni delle immagini utilizzate come input.

Infine, bisogna creare il file che indica al modello dove sono le nostre cartelle e quali sono le nostre classi da predire.

```
yaml_file_content = {
    'train': f'../{train_images_path}',
    'val': f'../{test_images_path}',
    'nc': len(breed_label.keys()), # number of classes
    'names': list(breed_label.values())
}
with open(yaml_file_name, 'w+') as f:
  yaml.dump(yaml_file_content, f)
```

Ora è possibile eseguire il comando di training.

```
!python $train_file_path --hyp $hyp_file_path --name $name
    --cfg $cfg_file_path --img $image_size --batch $batch_size
    --epochs $epochs --data $data_file_path
    --weights $weight_file_path --device 0
```

Con questa configurazione la rete ha 214 layer e circa 7 milioni di parametri.

2.3 Metriche

Come era prevedibile alcune classi vengono predette con un'accuratezza migliore. Si può prendere come esempio la classe relativa ai chihuahua e quella relativa ai barbonici nani (miniature_poodle).
Ora possiamo testare il nostro modello addestrato da terminale.

Class	**Images**	**Instances**	**P**	**R**
All	632	632	0.782	0.815
Chihuahua	632	24	0.951	0.817
Labrador_retriever	632	28	0.733	0.714
Rottweiler	632	16	0.623	1.000
toy_poodle	632	33	0.731	0.697
Scotch_terrier	632	21	0.89	0.773
standard_schnauzer	632	26	0.742	0.808
German_shepherd	632	13	0.772	1.000
beagle	632	30	0.902	0.925
Border_collie	632	21	0.887	0.952
Pekinese	632	15	0.804	0.867
basset	632	29	0.8	0.862
boxer	632	14	0.548	0.786
Siberian_husky	632	32	0.903	0.871
cocker_spaniel	632	20	0.816	0.886
Bernese_mountain_dog	632	38	0.946	0.92
standard_poodle	632	24	0.713	0.621
Saint_Bernard	632	22	0.885	0.864
Samoyed	632	36	0.734	0.922
Great_Dane	632	32	0.865	0.781
Great_Pyrenees	632	33	0.734	0.818
pug	632	31	0.748	0.767
golden_retriever	632	19	0.869	0.701
miniature_poodle	632	23	0.419	0.565
Doberman	632	30	0.776	0.693
French_bulldog	632	22	0.767	0.773

Table 1: Object detection performance metrics.

```
trained_model = f'yolov5/runs/train/{name}/weights/best.pt'

!python yolov5/detect.py --source $image_path
    --weights $trained_model
```

L'output è un'immagine con il bounding box e la relativa classe (se viene trovata con successo) affiancata dal livello di confidenza dell'inferenza.

2.4 Esportazione in Torchscript

Per eseguire il nostro modello su un dispositivo Android, si è clonata la repository nominata "Object Detection with YOLOv5 on Android". L'autore suggerisce come esportare il modello da PyTorch a TorchScript per poterlo eseguire su un dispositivo mobile. Nello specifico, nella sezione 4 del file README.md spiega come modificare lo script originale export.py della repository di YOLOv5. Dopo averlo fatto è possibile eseguire il seguente comando:

```
!python yolov5/export.py --optimize --weights $trained_model
    --include torchscript --data $data_path --imgsz $imgsz
```

3 Android Project

Per utilizzare il modello di TorchScript su un dispositivo Android ci si è basati su una libreria disponibile gratuitamente su GitHub (citata nel paragrafo precedente). Al suo interno, vengono fornite delle classi basate sulla libreria *org.pytorch*

per eseguire il nostro modello pre-addestrato di YOLOv5 e mostrare il risultato a video tramite i bounding box.
Il refactoring maggiore è stato eseguito su *MainActivity.java* e sulla gestione della camera.

3.1 Camera

Al progetto è stata aggiunta la classe *CameraController.java* basata sulla **CameraX API**, la quale si occupa di aprire e chiudere la fotocamera con dei metodi specifici e di analizzarne l'immagine catturata.

3.2 TorchScript su Android

I pesi che abbiamo esportato precedentemente da Colab dovranno essere posizionati nella cartella *assets* insieme a un file .txt contente i nomi associati alle classi. Bisogna fare attenzione a due cose molto importanti che non sono spiegate dall'autore su GitHub e sono fondamentali per il corretto funzionamento del codice:

- La lista dei nomi delle classi in *assets*, che nel mio caso è il file "dogs_breeds.txt" deve riportare le classi nello stesso ordine presente nel file del parametro "- - data" utilizzato in fase di training.
- Tutti i file .ptl e .yaml personalizzati dovrebbero avere lo stesso nome.

Il file *PrePostProcessor.java* è da personalizzare per adattarlo al nostro problema specifico, modificando i parametri che verranno utilizzati durante l'esecuzione di una inferenza.
Infine, tramite il file *ObjectDetectionActivity.java* e il metodo *run()* nella MainActivity, si può controllare il comportamento del modello e la visualizzazione dei risultati sullo schermo.

4 CanineCam

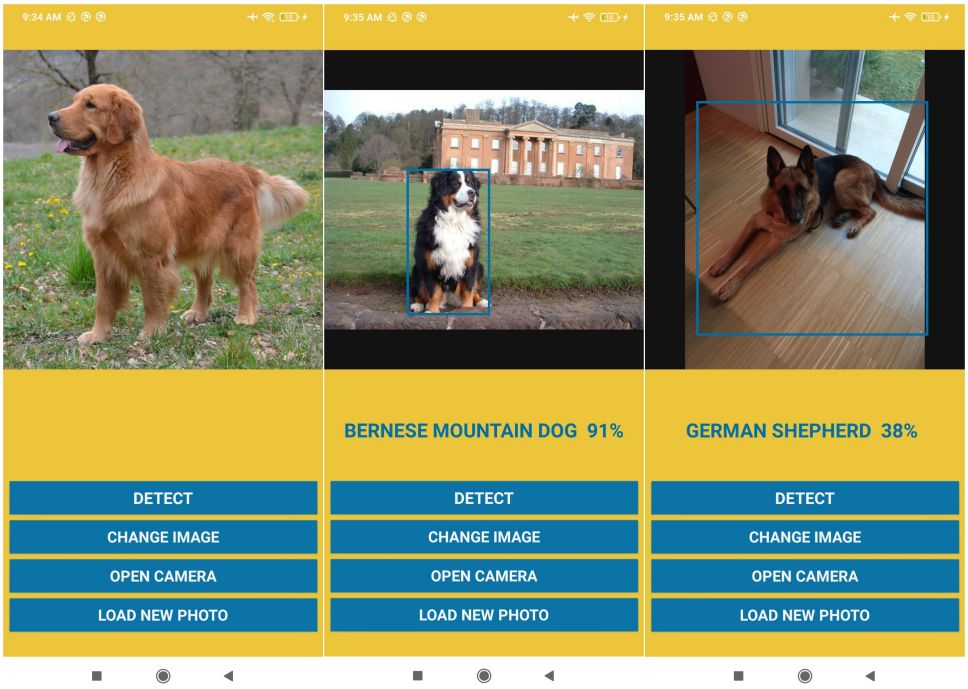
L'app prodotta è stata denominata "CanineCam". Tutto le funzioni possono essere eseguite dalla schermata principale e si possono riassumere nei seguenti punti:

1. Homepage. La schermata principale è definita da una *view* che si occupa di mostrare l'immagine caricata o della fotocamera aperta.
2. Detect. Il primo pulsante esegue la *detection* sull'immagine correntemente presente nella view e mostrare il risultato sopra di essa.
3. Change image. Carica un'altra immagine tra quelle presenti nel boundle dell'app da usare come test.

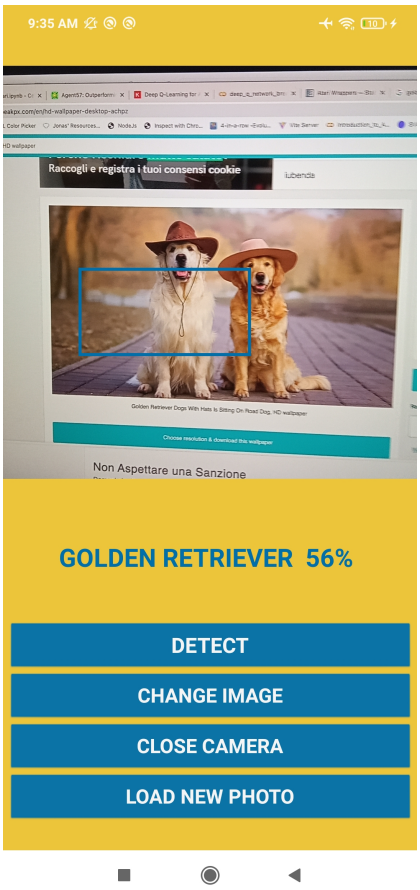
4. Open Camera. Aprire la fotocamera ed eseguire in tempo reale, senza bisogno di premere il pulsante "detect", il rilevamento di cani e dell'eventuale razza.

5. Load New Photo. Permette di caricare una nuova fotografia salvata in precedenza sul proprio dispositivo fisico.

Di seguito vengono mostrate alcune schermate dell'applicazione.



La terza schermata mostra la predizione di una foto caricata da cellulare. Si può notare una minore confidenza da parte del modello dovuta probabilmente alla qualità dell'immagine.
Per ultima, la sfida più complicata risulta essere quella della *live-detection* (predizione in tempo reale). Nell'immagine sotto si può notare che il modello è riuscito a identificare correttamente la razza del cane, tuttavia da alcuni test sembra risulti essere molto suscettibile ai movimenti della fotocamera e della sua qualità.



5 Conclusioni

La parte più complicata è stata l'integrazione del modello YOLOv5 sul dispositivo Android. La repository di GitHub impiegata non è ben documentata e si sono presentati molti bug che si sono dovuti risolvere in autonomia.
L'applicazione funziona correttamente in buona parte dei casi che sono stati testati, anche se non con confidenze molto alte.
Per migliorare questo progetto bisognerebbe eseguire un training più lungo provando anche a testare diversi iperparametri. Una volta ottenuto un risultato più affidabile si può estendere molto facilmente l'elenco delle classi riconoscibili dal modello.
Quello che si è notato è che il modello eseguito su un computer con PyTorch ottiene dei risultati migliori. Indagare ulteriormente su questo aspetto potrebbe migliorare la qualità delle predizioni effettuate su uno smartphone.

6 Risorse

Su YOLOv5:

- Documentazione YOLOv5: https://docs.ultralytics.com/yolov5/
- Repository ufficiale YOLOv5: https://github.com/ultralytics/yolov5
- Tutorial YOLOv5:
  https://colab.research.google.com/github/ultralytics/yolov5/blob/master/tutorial.ipynb
- Tutorial YOLOv5 training:
  https://docs.ultralytics.com/yolov5/tutorials/train_custom_data/#3-train

Su Android ObjectDetection:

- ObjectDetection repository:
  https://github.com/pytorch/android-demo-app/blob/master/ObjectDetection/
- Utilizzare custom weights:
  https://github.com/pytorch/android-demo-app/tree/master/ObjectDetection#4-update-the-demo-app

Dataset:

- Stanford Dataset: https://www.kaggle.com/datasets/jessicali9530/stanford-dogs-dataset

Materiale utile:

- *Image Preprocessing for Efficient Training of YOLO Deep Learning Networks*:
  H. -J. Jeong, K. -S. Park and Y. -G. Ha, "Image Preprocessing for Efficient Training of YOLO Deep Learning Networks," 2018 IEEE International Conference on Big Data and Smart Computing (BigComp), Shanghai, China, 2018, pp. 635-637, doi: 10.1109/BigComp.2018.00113.