SVILUPPO DI UNA APPLICAZIONE PER IL RICONOSCIMENTO DI MONETE

Attività Progettuale Sistemi Digitali M

Gabriele Corsi

Luglio 2023

Indice

Introduzione

Il progetto ha come obiettivo quello di creare una applicazione per un dispositivo embedded (in questo caso uno smartphone Android) che sia in grado, attraverso l'utilizzo della fotocamera, di riconoscere una moneta, in particolare classificare le monete commemorative da 2€. Queste monete sono coniate e messe in circolazione a partire dal 2004 dalla Banca Centrale Europea per commemorare anniversari di eventi storici o per porre in risalto eventi attuali di particolare importanza. Di seguito vengono mostrati alcuni esempi [1].

Figura 1: Esempi di monete commemorative

L'idea è quindi di permettere ad un utente di fotografare una di queste monete e restituire le informazioni corrispondenti come l'evento per la quale è stata creata, l'anno di emissione, il numero di monete coniate, ecc.

Il primo capitolo mostra le opzioni provate durante lo svolgimento del progetto e spiega la soluziona trovata.
Il capitolo 2 mostra il codice utilizzato per implementare la soluzione trovata, descrivendo l'intero workflow.
Il terzo capitolo presenta l'applicazione Android, descrivendo le varie interfaccie presenti.
Infine i capitoli Conclusione e Sviluppi Futuri riassumono i risultati ottenuti e mostrano alcuni possibili miglioramenti da implementare.

Capitolo 1

Riconoscimento Moneta

Per il riconoscimento della moneta sono state pensate le seguenti soluzioni:

- utilizzare una rete neurale per l'identificazione degli oggetti (es. YOLOv5)
- utilizzare una rete neurale per la classificazione delle immagini (es. ResNet, EfficientNet, ecc)
- utilizzare una rete neurale di similarità

La prima soluzione è stata presa in considerazione in quanto, in passato, avevo già utilizzato questo tipo di rete, per cui è stato fatto un tentativo. Fin da subito però la rete ha mostrato risultati molto insoddisfacenti in quanto queste rete sono più indicate per identificare classi di oggetti che spesso appartengono a domini molto diversi (per esempio le classi "persona", "macchina", "cane", ecc). In casi come questo dove le monete si differenziano le una dalle altre per alcuni dettagli questo approccio non permette di ottenere buoni risultati.

Per questo motivo la seconda scelta è stata quella di utilizzare una rete neurale convoluzionale in grado di classificare l'immagine di input in ingresso. Queste tipologie di reti sono più adeguate per lo scopo del progetto essendo in grado di distinguere classi molto simili tra di loro. Un problema riscontrato durante la realizzazione del progetto però riguarda il dataset per l'allenamento di queste reti. In rete infatti non sono presenti dataset contenenti tutte le monete commemorative, ma bensì è possibile solo trovare alcuni esempi di ogni moneta cercandoli su Google Immagini. Questo non permette quindi di addestrare in maniera adeguata questo tipo di rete che richiederebbe molte immagini di esempio per ogni moneta commemorativa. Inoltre, se anche si riuscisse a comporre un dataset adeguato, ogni anno vengono emesse nuove monete commemorative, sarebbe quindi necessario ricreare il dataset con le nuove classi e allenare nuovamente la rete.

La scelta finale del progetto è stata quindi quella di utilizzare una rete neurale di similarità, in grado di risolvere le criticità mostrate precedentemente.

1.1 Similarity Learning

Normalmente un classificatore ha l'obiettivo di imparare ad assegnare una classe ad ogni input, nel Similarity Learning la rete invece impara a misurare la similarità tra due input. Queste reti trasformano l'immagini in ingresso in un vettore n-dimensionale chiamato "embedding" (tipicamente n compreso tra 64 e 512). Questo spazio n-dimensionale viene chiamato "representation space" o "embedding space". Utilizzando questo spazio è possibile determinare la distanza tra due input: più questa è piccola più gli input sono simili, più è grande più sono dissimili. Per cui durante l'allenamento la rete dovrà imparare ad assegnare distanze piccole (grandi) a esempi che sono semanticamente simili (dissimili). Durante invece la fase di inferenze su nuovi input la rete calcolerà il nuovo embedding e successivamente si cercherà l'embedding spazialmente più vicini, questo rappresenterà l'immagine più simile all'interno del dataset da cui è poi possibile ricavare la relativa classe.

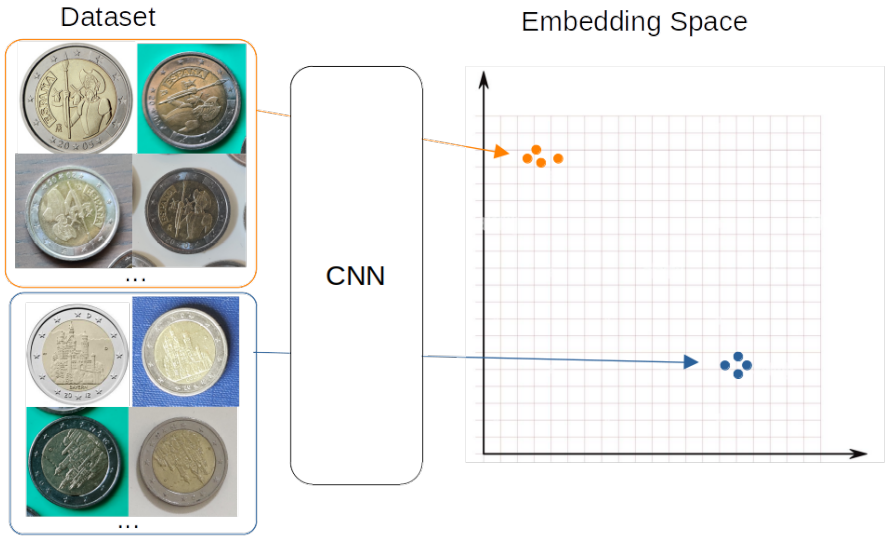


Figura 1.1: In figura viene illustrato un ipotetico embedding space bidimensionale, si noti come l'obiettivo dell'addestramento è quello di fare in modo che embedding di input appartenenti alla stessa classe siano vicini, mentre embedding appartenenti a classi diverse siano lontani

1.2 La Funzione di Loss

Come in ogni rete neurale, anche in questo caso, è necessario definire la funzione di Loss, ovvero la funzione che vogliamo minimizzare per fare in modo che la rete possa aggiornare i pesi in modo corretto e quindi "imparare" dai dati presenti nel dataset. Tipicamente la funzione di Loss viene creata sfruttando l'etichetta

(o "label") dei dati presenti nel dataset, sapendo infatti l'output ideale che la rete dovrebbe ottenere è possibile confrontarlo con l'output reale. Nella rete utilizzata in questo progetto ciò non è possibile, infatti non esiste un output "ideale". Per creare una funzione di Loss in questo caso si utilizzano le "Siamese Networks". In questo tipo di reti i pesi vengono aggiornati nel seguente modo: si calcolano gli embedding di due input del dataset:

- se i due input appartengono alla stessa classe, la funzione di Loss minimizzerà la distanza fra i due embedding e i pesi della rete verranno aggiornati di conseguenza
- se i due input appartengono a classi diverse, la funzione di Loss massimizzerà la distanza fra i due embedding e i pesi della rete verranno aggiornati di conseguenza

Questo tipo di funzione di loss viene chiamata Constrastive Loss.

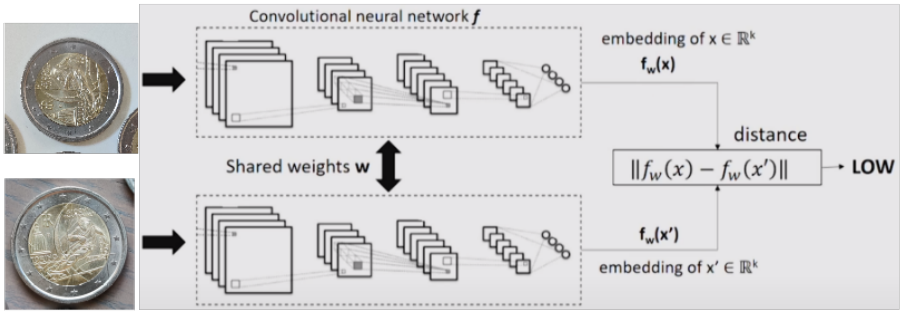


Figura 1.2: In figura viene mostrato il funzionamento di una rete siamese nel caso in cui i due input appartengano alla stessa classe

Un ulteriore metodo è l'utilizzo delle Triplet Networks che prevede di aggiornare i pesi utilizzando sempre 3 input: un input chiamato "anchor", un input della stessa classe dell'anchor e un input di classe diversa dell'anchor. In questo caso la funzione di Loss conterrà sia una distanza da minimizzare che una da massimizzare.

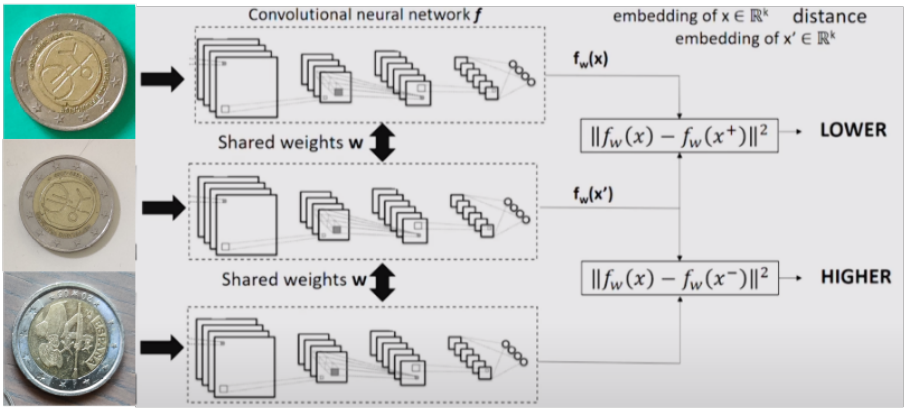


Figura 1.3: In figura il funzionamento di una "triplet network"

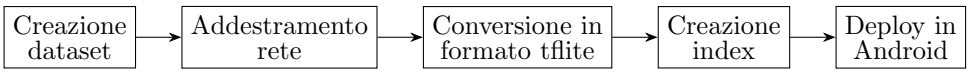
1.3 Vantaggi Rete di Similarità

Il vantaggio di utilizzare questi modelli di similarità è che permette, una volta allenata la rete con un determinato dataset, di poter allargare il numero di classi riconosciute soltanto con l'aggiunta di poche (o anche una) immagini di esempio. Sarà sufficiente infatti utilizzare l'immagine di una nuova classe per calcolare il relativo embedding e sfruttare questo nuova informazione per le successive inferenze.

Capitolo 2

Implementazione

L'implementazione dell'architettura è stata fatta utilizzando il framework Tensorflow [2], il grafico sotto mostra l'intero processo dalla creazione del dataset fino al deploy in ambiente Android.



2.1 Creazione dataset

In rete non sono stati trovati dataset di monete commemorative percui si è deciso di comporne uno utilizzando una piccola collazione personale. Per velocizzare la creazione si sono effettuate foto ritraenti più monete contemporaneamente e successivamente, utilizzando una rete YOLOv5 allenata per riconoscere monete generiche, sono state ritagliate in modo automatico. Successivamente sono state divise manualmente nelle rispettive classi. Ciò ha permesso di ottenere con uno sforzo ridotto un dataset di 560 immagini divisi in 14 classi.

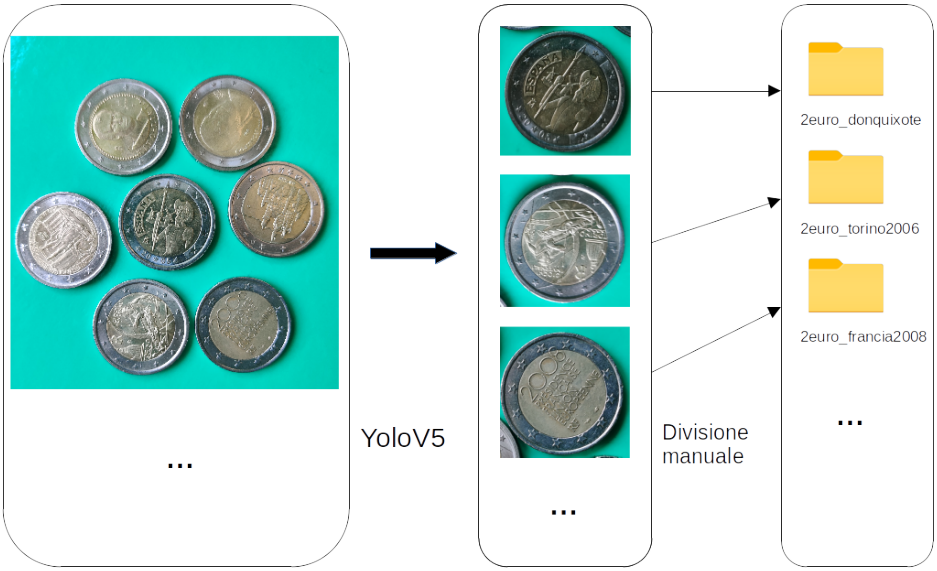


Figura 2.1: Creazione dataset

2.2 Addestramento rete

Per creare ed allenare un modello di similarità è stata utilizzata la libreria tensorflow_similarity [3], il codice python descritto in seguito è presente nel file train_model.py. La rete scelta nel progetto è la EfficientNet allenando solamente gli ultimi tre livelli specificando il parametro trainable="partial". É necessario inoltre decidere la dimensionalità dell'embedding, in questo caso 128. Successivamente si specificano i seguenti iperparametri:

- epochs = 25, numero di epoche per le quali la rete viene allenata
- LR = 0.00003, Learning Rate, parametro con il quale si imposta la velocità di apprendimento della rete
- steps_per_epoch = 100, il numero batch forniti alla rete per ogni epoca
- distance = cosine, la funzione di distanza da utilizzare per calcolare lo spazio tra due embedding

Si compila il modello e si avvia l'addestramento con il dataset importato precedentemente. Alla conclusione dell'addestramento si salva il modello in formato SavedModel. Di seguito il codice descritto.

```
embedding_size = 128
model = tfsim.architectures.EfficientNetSim(
    train_ds.example_shape,
    embedding_size,
    variant="B3",
    trainable="partial",
    pooling="gem",
    gem_p=1.0,
)
epochs = 25
LR = 0.00003
gamma = 80
steps_per_epoch = 100
val_steps = 40
distance = "cosine"
loss = tfsim.losses.MultiSimilarityLoss(distance=distance)

model.compile(
    optimizer=tf.keras.optimizers.Adam(LR),
    loss=loss,
    distance=distance,
    search="nmslib",
)
history = model.fit(
    train_ds,
    epochs=epochs,
    steps_per_epoch=steps_per_epoch,
    validation_data=test_ds,
    validation_steps=val_steps,
    callbacks=callbacks,
)
```

2.3 Conversione in formato tflite

Successivamente sono stati utilizzati i due file convert_to_tflite.py e write_metadata.py per convertire il modello da tipo SavedModel a tipo tflite, utilizzato nei dispositivi embedded. La scrittura dei metadati è necessaria per permettere alla libreria Model Maker di Tensorflow [4] di utilizzare delle interfaccie ad alto livello per la gestione delle inferenze in ambiente Android.

2.4 Creazione index e inserimento nel modello

Per poter eseguire le inferenze sul dispositivo mobile, oltre alla rete addestrata con i relativi pesi è necessario creare l'index. Questo index non è altro che la raccolta degli embedding di alcuni (o anche uno soltanto) esempi per ogni classe che si vuole riconoscere. Quando si esegue l'inferenza di un nuovo input è si andrà a cercare all'interno dell'index l'embedding più vicino a quello appena calcolato, individuando quindi la classe corrispondente. Se, come nel caso del progetto, in futuro sarà necessario aggiungere ulteriori classi da identificare, basterà ricreare questo index aggiungendo alcuni esempi della nuova classe. La libreria

Model Maker di Tensorflow utilizza il metodo denominato ScaNN [5] (Scalable Nearest Neighbors) per ricercare in modo efficiente gli embeddings più vicini a quello calcolato durante l'inferenza. L'index viene poi integrato all'interno del file .tflite in modo da avere all'interno di questo file sia l'architettura della rete con i relativi pesi sia i dati necessari per la ricerca dell'embedding più vicino.

```
data_loader = searcher.ImageDataLoader.create("./model_efficientNet
                                    .tflite")
train_folder ="./index_folder/"
for folder in sorted(listdir(train_folder)):
    abs_folder = join(train_folder, folder)
    data_loader.load_from_folder(abs_folder, folder)
scann_options = searcher.ScaNNOptions(
      distance_measure="squared_l2",
      tree=searcher.Tree(num_leaves=math.ceil(math.sqrt(tot_db)),
                                        num_leaves_to_search=15),
      score_ah=searcher.ScoreAH(dimensions_per_block=1,
            anisotropic_quantization_threshold=0.2))
model = searcher.Searcher.create_from_data(data_loader,
                                    scann_options)
model.export(
      export_filename="searcher.tflite",
      userinfo="",
      export_format=searcher.ExportFormat.TFLITE)
```

2.5 Deploy in Android

Il file viene quindi importato nella cartella assests del progetto Android ed è possibile caricarlo come mostrato nel codice seguente:

```
val options = ImageSearcher.ImageSearcherOptions.builder()
.setBaseOptions(BaseOptions.builder().build())
.setSearcherOptions(
SearcherOptions.builder().setL2Normalize(true).build())
.build()
imageSearcher =
ImageSearcher.createFromFileAndOptions(this, "searcher.tflite",
                                    options)
```

Le inferenze sono possibili utilizzando il seguente codice:

```
var results =imageSearcher.search(TensorImage.fromBitmap(photo))
```

2.6 Visualizzazione dell'Embedding Space

Utilizzando strumenti che convertono la similarità tra punti n-dimensionali è possibile visualizzare gli embedding prodotti dalla rete in uno spazio tridimensionale. In particolare utilizzando la libreria TSNE (T-distributed Stochastic Neighbor Embedding) presente in scikit-learn [6] è stato possibile visualizzare il dataset utilizzato nell'allenamento e verificare anche visualmente la bontà della rete come visibile in Figura 2.2.

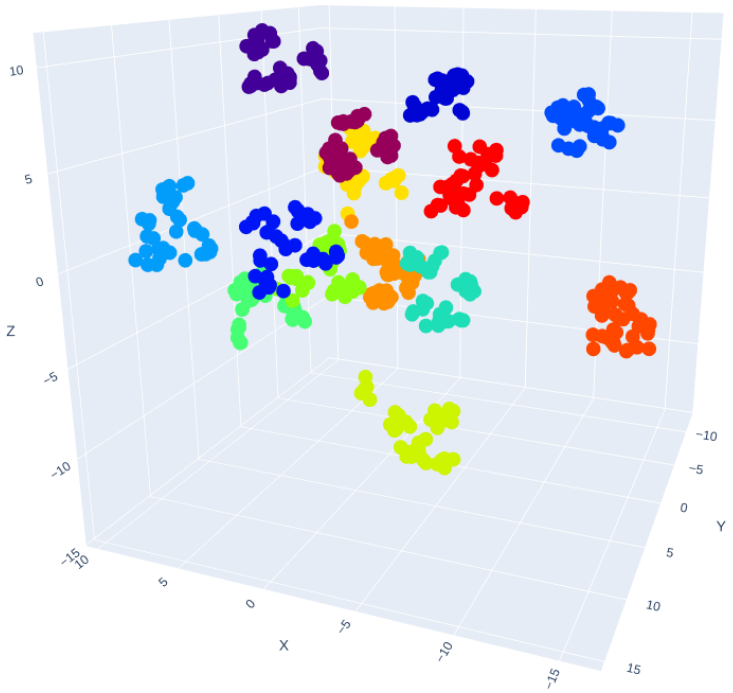


Figura 2.2: Embedding del dataset visualizzati in uno spazio tridimensionale, ogni colore è associato ad una classe diversa

Capitolo 3

Applicazione Android

La soluzione trovata è stata implementata all'interno di una applicazione Android. Lo sviluppo e i test sono stati eseguiti utilizzando lo smartphone OnePlus 7 (GM1900).

3.1 Interfaccia Utente

L'interfaccia principale si presente in questo modo: lo schermo mostra ciò che viene inquadrato dalla fotocamera dello smartphone, al centro dello schermo è presente un cerchio nero che mostra dove nella foto la moneta si deve trovare quando viene scattata la foto. In basso il pulsante "FOTO" permette di eseguire l'inferenza e avviare la schermata successiva.

Figura 3.1: Schermata principale

L'interfaccia successiva mostra nella parte superiore la foto scattata ritagliata e nella parte inferiore la lista delle classi di monete più simili in ordine decrescente.

Figura 3.2: Schermata con le classi simili individuate

Cliccando su un elemento di questa lista è possibile aprire una schermata

dove vengono mostrati i dettagli della relativa moneta:

- il titolo
- l'immagine di riferimento
- la descrizione della moneta
- il numero di monete coniate
- la data di emissione

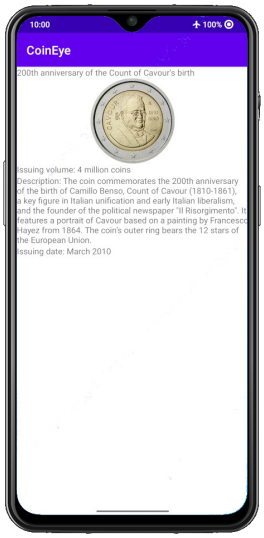


Figura 3.3: Schermata con i dettagli di una moneta

Capitolo 4

Conclusione

In questo progetto è stata realizzata una applicazione per un sistema embedded in grado di fornire i dettagli di una moneta commemorativa attraverso l'individuazione della stessa mediante una foto. In particolare la soluzione implementata prevede l'utilizzo di un modello di similarità che utilizza la trasformazione di una immagine in un vettore n-dimensionale e sfrutta le distanze in questo spazio per determinare la similarità fra due input.

Il modello è stato addestrato utilizzando 14 classi differenti ma, durante la creazione dell'index, sono state aggiunte:

- 4 classi utilizzando 4 immagine di riferimento per classe
- 6 classi utilizzando 1 immagine di riferimento per classe

L'applicazione ha dato ottimi risultati sulle 14 classi che hanno contribuito all'addestramento, individuando in maniera consistente la classe di appartenenza. Per le classi che non hanno partecipato all'addestramento la rete spesso è in grado di individuarle e quasi sempre vengono inserite tra i primi 5 risultati.

Capitolo 5

Sviluppi Futuri

Alcuni possibili sviluppi futuri sono:

- ingrandire il dataset, sia nel numero di foto per ogni classe, sia nel numero di classi, con l'idea di ingrandire il numero di feature che la rete è in grado di riconoscere e quindi migliorando la sua capacità di generalizzare
- verificare che il sistema sia in grado di scalare adeguatamente, per esempio verificare se con l'aggiunta di un numero maggiori di classi che non hanno contribuito all'addestramento il sistema mantenga la precisione mostrata nel progetto
- provare ad addestrare la rete utilizzando diverse combinazioni di iper-parametri come:
  - epoche
  - step per epoche
  - dimensionalità dell'embedding
  - funzione di loss
  - utilizzare più esempi per ogni classe durante la creazione dell'index
  - ecc
- effettuare il porting su sistema iOS

Bibliografia

[1] https://www.ecb.europa.eu/euro/coins/comm/html/index.en.html

[2] https://www.tensorflow.org/api_docs/python/tf

[3] https://blog.tensorflow.org/2021/09/introducing-tensorflow-similarity.html

[4] https://www.tensorflow.org/lite/models/modify/model_maker

[5] https://ai.googleblog.com/2020/07/announcing-scann-efficient-vector.html

[6] https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html