

Scuola di Ingegneria e Architettura  
Corso di Laurea Magistrale in Ingegneria Informatica

# **Progetto e attività progettuale in Sistemi Digitali M**

EYE E PUPIL TRACKING

*Autori*

**Luigi di Nuzzo**

**Daniele Foschi**

**Filippo Veronesi**



# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
<b>2</b>	<b>Rete Neurale</b>	<b>3</b>
2.1	Eye detection . . . . .	3
2.1.1	Dataset . . . . .	4
2.1.2	Trainig . . . . .	4
2.1.3	TfLite . . . . .	6
2.1.4	Metadata . . . . .	6
2.2	Attività progettuale: Pupil detection . . . . .	6
2.2.1	Dataset . . . . .	6
2.2.2	Trainig . . . . .	7
2.2.3	TfLite . . . . .	8
2.2.4	Metadata . . . . .	8
<b>3</b>	<b>Progetto Android</b>	<b>9</b>
3.1	Schermate . . . . .	9
3.2	Nerd Mode . . . . .	10
3.2.1	Preview . . . . .	11
3.2.2	Switch camera . . . . .	11
3.2.3	Analyze . . . . .	11
3.2.4	Attività progettuale: Riconoscimento pupilla . . . . .	12
3.3	Calibration . . . . .	13
3.4	Game . . . . .	13
3.5	Prestazioni . . . . .	14
<b>4</b>	<b>Conclusioni</b>	<b>17</b>



# Elenco delle figure

2.1	Eye Detection labelmap.pbtxt . . . . .	4
2.2	Tensorboard Learning_rate . . . . .	5
2.3	Test Eye-Tracking . . . . .	5
2.4	Pupil Detection labelmap.pbtxt . . . . .	7
2.5	Test Pupil-Tracking . . . . .	7
3.1	Schermata Nerd . . . . .	10
3.2	Schermata Nerd mode con pupilla . . . . .	12
3.3	Schermata Calibration . . . . .	13
3.4	Schermata Game . . . . .	14
3.5	CPU usage . . . . .	15
3.6	Memory usage . . . . .	16



# Capitolo 1

## Introduzione

Questo progetto prevede la realizzazione di un applicativo Android che, sfruttando due reti neurali, sia in grado di riconoscere gli occhi e le pupille dell’utente. Il sistema sfrutterà due modelli di rete neurale addestrati in modo tale da essere in grado di riconoscere gli occhi e le pupille di uno o più utenti utilizzando due dataset: uno avente 10 mila immagini di visi umani ed un altro contenente 5 mila immagini di sole pupille. L’applicativo permetterà di svolgere un filtraggio “live”, cioè apprendendo la camera frontale ed esterna dello smartphone e individuando real-time gli occhi e le pupille dell’utente.

Inoltre, è stato implementato un gioco composto da un Quiz di 4 domande aiutato da un tool di calibrazione. L’utente tramite il riconoscitore di occhi è in grado di rispondere ad una certa domanda spostando, tramite gli occhi e il cellulare, un puntatore verso la risposta giusta posta ai 4 angoli del dispositivo. In caso di risposta corretta, viene posta un’ulteriore domanda, altrimenti si viene avvertiti della risposta sbagliata tramite un alert.

Tale caso di studio è un classico esempio di applicazione di Machine Learning e il software farà ricorso a una rete neurale convoluzionale (CNN). Tale scelta è dovuta al fatto che una rete neurale rappresenta il modo più comodo e pratico per problemi di object detection, come quello di questa attività in cui vengono individuati gli occhi e le pupille.

L’applicativo, inoltre, è pensato per la piattaforma Android e quindi tale progetto pone attenzione anche all’uso di risorse in quanto dovrà funzionare su smartphone, ovvero dispositivi embedded.



# Capitolo 2

## Rete Neurale

L'obiettivo primario di questo task è quello di produrre un modello di rete neurale addestrata in grado di riconoscere gli occhi di una o più persone. Successivamente, si è prodotto un modello di rete neurale addestrata in grado di riconoscere le pupille degli esseri umani.

A tali scopi si è utilizzato un modello messo a disposizione dalla libreria *TensorFlow 2 Detection Model Zoo* [4] ottimizzato per il training di modelli specifici per object detection di immagini. Visto l'ambito dei sistemi embedded nel quale il progetto complessivo si colloca si è optato per un modello SSD MobileNet<sup>1</sup>[3], una CNN pensata per dispositivi mobile. MobileNet è il primo modello di computer vision pensato per dispositivi embedded basato su TensorFlow. MobileNet è sufficientemente leggera e veloce da essere eseguita su smartphone senza consumo di risorse eccessivo mantenendo comunque una precisione adeguata.

### 2.1 Eye detection

La prima rete neurale riguarda il riconoscimento degli occhi di esseri umani, la quale risulterà essere il primo passo per eseguire anche un corretto riconoscimento delle pupille. Infatti, durante l'implementazione Android, l'output di questa rete, verrà usata come input per la seconda rete neurale incaricata del riconoscimento delle pupille. In questa fase ci limitiamo a creare una rete neurale in grado di riconoscere gli occhi.

---

<sup>1</sup>SSD MobileNet V2 FPNLite 640x640

### 2.1.1 Dataset

Per il training del modello si è utilizzato un dataset pubblico rilasciato da Google<sup>2</sup> contenente 10 mila immagini di visi umani, dove ogni foto conteneva da uno a più visi umani.

Oltre alle immagini il dataset conteneva un utile file *Excel* che indicava le coordinate della posizione degli occhi delle persone all'interno dell'immagine.

Prima dell'addestramento della rete era però necessario convertire questi dati di input in file *\*.xml* in modo tale da poterli poi usare per generare i **TFRecord**, utili per TensorFlow. Per questo motivo è stato implementato uno script *Java* in grado di creare un file *.xml* per ogni foto del dataset estraendo le informazioni dal file *Excel*. Le informazioni più importanti per il nostro progetto che sono state estratte sono: i dati geografici delle posizioni degli occhi (x e y, sia min che max) e il nome dell'oggetto da riconoscere.

Inoltre, per la corretta generazione dei **TFRecord**, si è reso necessario compilare il file *labelmap.pbtxt* con al suo interno i nomi e gli id dei nostri oggetti da riconoscere: nel nostro caso un solo item di id pari a 1 e con nome "Human eye".

```
item {
  id: 1
  name: 'Human eye'
}
```

Figura 2.1: Eye Detection labelmap.pbtxt

### 2.1.2 Trainig

Si è quindi proceduto all'addestramento della rete tramite Python usando TensorFlow e le API di Keras. Si è reso necessario modificare il file *pipeline.config* adattandolo alle nostre esigenze:

- *numclasses* che corrisponde al numero di item da identificare;
- *path* vari da adattare al nostro workspace, tra cui il path per i checkpoint, per i record di input e per il labelmap.

Durante il procedimento di training si sono tenuti controllati i valori della nostra rete. In particolare, abbiamo usato **Tensorboard**: un framework grafico utile a capire l'andamento della nostra rete, infatti abbiamo avuto modo di controllare ad ogni training le performance della nostra rete di machine learning.

---

<sup>2</sup>Open Image V6, <https://storage.googleapis.com/openimages/web/index.html?v6>

TensorBoard fornisce la visualizzazione e gli strumenti necessari per la sperimentazione del machine learning:

- Monitoraggio e visualizzazione di metriche come perdita e precisione;
- Visualizzazione del grafico del modello (operazioni e livelli);
- Visualizzazione degli histogrammi di pesi, distorsioni o altri tensori man mano che cambiano nel tempo.

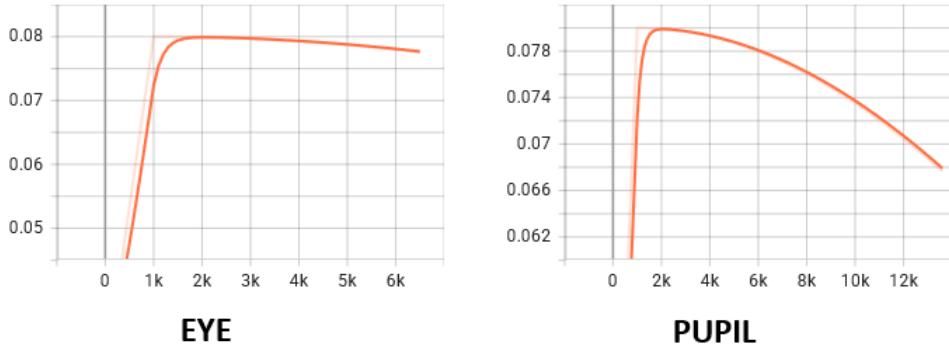


Figura 2.2: Tensorboard Learning\_rate

Si è ottenuto in output un modello addestrato e pronto all’uso.



Figura 2.3: Test Eye-Tracking

Prima di testare direttamente su Android, abbiamo deciso prima di testare e analizzare la nostra nuova rete utilizzando la libreria Matplotlib, utile per visualizzare graficamente via shell i nostri risultati. In particolare abbiamo usato **Tkinter**, l’unico framework GUI incluso nella libreria standard di Python.

A fini di testing si è preso ad esempio il volto di un personaggio pubblico, quello di Albert Einstein, e anche un’immagine contenente più persone in modo tale da poter verificare la correttezza e la precisione della rete anche in condizioni più difficili.

### 2.1.3 TfLite

Dopo esserci accertati che la rete funzionasse correttamente e avesse dei livelli di precisione sopra una certa soglia, si è ottenuto in output un modello addestrato e pronto all’uso, che poi è stato convertito e quantizzato in un formato adatto ai sistemi embedded, quello di TensorFlow Lite.

### 2.1.4 Metadata

Per un corretto funzionamento della rete neurale all’interno di Android, TfLite richiede che venga generato anche un file Metadata che contiene le informazioni necessarie per pre-processare le immagini. Questo file si rende necessario in quanto i nostri tensori di input sono di tipo kTfLiteFloat32. È necessario creare un nuovo *label\_map.txt* con al suo interno nella prima riga il nostro item per la detection: nel nostro caso *”Human eye”*.

Una volta generato questo nuovo file metadata.tflite siamo pronti ad importarlo all’interno della nostra applicazione Android come file di Machine Learning *”ml”*.

## 2.2 Attività progettuale: Pupil detection

Per quanto riguarda invece la rete neurale incaricata al riconoscimento delle pupille, viene creata e addestrata in maniera simile alla nostra prima rete neurale. In questa fase ci limitiamo a crearla, in un secondo momento, durante l’implementazione Android, essa userà come input l’output della prima rete neurale dedicata agli occhi.

### 2.2.1 Dataset

Per il training del modello si è utilizzato un dataset pubblico rilasciato da Kaggle<sup>3</sup> contenente 5 mila immagini di sole pupille umane, comodo per il raggiungimento del nostro obiettivo di pupil detection.

Questa volta, per ottenere i file *\*.xml* di ogni immagine, abbiamo usufruito di un tool grafico chiamato **LabelImage**, scaricabile gratuitamente su Github<sup>4</sup>.

---

<sup>3</sup><https://www.kaggle.com/datasets/pavelbiz/eyes-rtte>

<sup>4</sup><https://github.com/tzutalin/labelImg>

Questo tool ci permetteva di costruire i boxes attorno alla pupilla ottenendo in output per ogni immagine un file `*.xml` contenente le informazioni necessarie: come il nome dell'oggetto su cui si è costruito attorno il box ("pupil"), e le relative coordinate geografiche dei 4 punti del box (xmin, ymin, xmax, ymax).

Prima dell'addestramento della rete era però necessario, usando questi nuovi dati `*.xml`, generare i **TFRecord**, utili per TensorFlow. Inoltre, per la corretta generazione dei **TFRecord**, si è reso necessario compilare il file `labelmap.pbtxt` con al suo interno i nomi e gli id dei nostri oggetti da riconoscere: nel nostro caso un solo item di id pari a 1 e con nome "pupil".

```
item {
  id: 1
  name: 'gaze'
}
```

Figura 2.4: Pupil Detection labelmap.pbtxt

## 2.2.2 Trainig

Per quanto riguarda il Training, abbiamo effettuato le stesse scelte effettuate per l'eye detection.

Prima di testare direttamente su Android, anche in questo caso abbiamo deciso prima di testare e analizzare la nostra nuova rete utilizzando la libreria Matplotlib e usando **Tkinter**.

A fini di testing si è preso ad esempio l'occhio qui sotto:

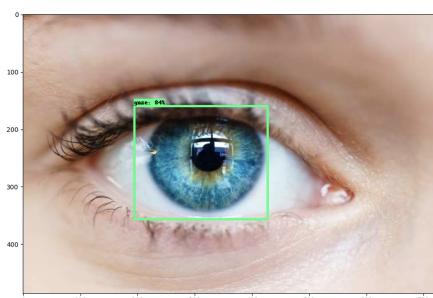


Figura 2.5: Test Pupil-Tracking

### 2.2.3 TfLite

Seguendo gli stessi identici procedimenti eseguiti durante la conversione dell'eye detection, anche qui dopo esserci accertati che la rete funzionasse correttamente e avesse dei livelli di precisione sopra una certa soglia, si è ottenuto in output un modello addestrato e pronto all'uso, che poi è stato convertito e quantizzato in un formato adatto ai sistemi embedded, quello di TensorFlow Lite.

### 2.2.4 Metadata

Per un corretto funzionamento della rete neurale all'interno di Android, TfLite richiede che venga generato anche un file Metadata che contiene le informazioni necessarie per pre-processare le immagini. Questo file si rende necessario in quanto i nostri tensori di input sono di tipo kTfLiteFloat32. È necessario creare un nuovo *label\_map.txt* con al suo interno nella prima riga il nostro item per la detection: nel nostro caso *"pupil"*.

Una volta generato questo nuovo file metadata.tflite siamo pronti ad importarlo all'interno della nostra applicazione Android come file di Machine Learning *"ml"*.

# Capitolo 3

## Progetto Android

Per la dimostrazione pratica degli argomenti fino a qui presentati si è scelto di progettare e implementare un applicativo software per dispositivi mobili.

L'analisi della gamma di dispositivi utilizzabili e reperibili dal team di questo progetto ha evidenziato due architetture su cui sviluppare l'applicativo: IOS e Android.

Entrambe le architetture sono parimente valide ai fini del progetto ma dovensi confrontare con il tempo a disposizione e le conoscenze già in possesso dagli elementi del gruppo di sviluppo si è deciso di convogliare interamente le energie sullo sviluppo su piattaforma Android.

Le motivazioni di tale scelta sono le seguenti:

1. La rete neurale è stata sviluppata in Tensorflow (vedi 2) che ha nativamente grande interoperabilità con l'ambiente Android.
2. Parte del gruppo di lavoro ha esperienza con lo sviluppo di applicazioni in Android nativo.

Tutti i file e le risorse progettate e realizzate per questo progetto sono reperibili nel seguente repository *GitHub*:

[https://github.com/DaniDF/eye\\_pupil\\_tracker](https://github.com/DaniDF/eye_pupil_tracker)

### 3.1 Schermate

L'applicazione per dimostrare l'utilizzo e il funzionamento della rete neurale implementa 3 sezioni:

- ”**Nerd**” (vedi 3.2) dove vengono mostrate tutte le informazioni che è in grado di acquisire il dispositivo e che la rete neurale processa.

- **Calibration** (vedi 3.3) schermata in cui vengono mostrate elementi grafici in modo da capire il range di movimento degli occhi nello spazio.
- **Game** (vedi 3.4) viene implementato un semplice gioco che sfrutta il movimento degli occhi per eseguire comandi e azioni che facciano evolvere l'utente nel gioco.

## 3.2 Nerd Mode

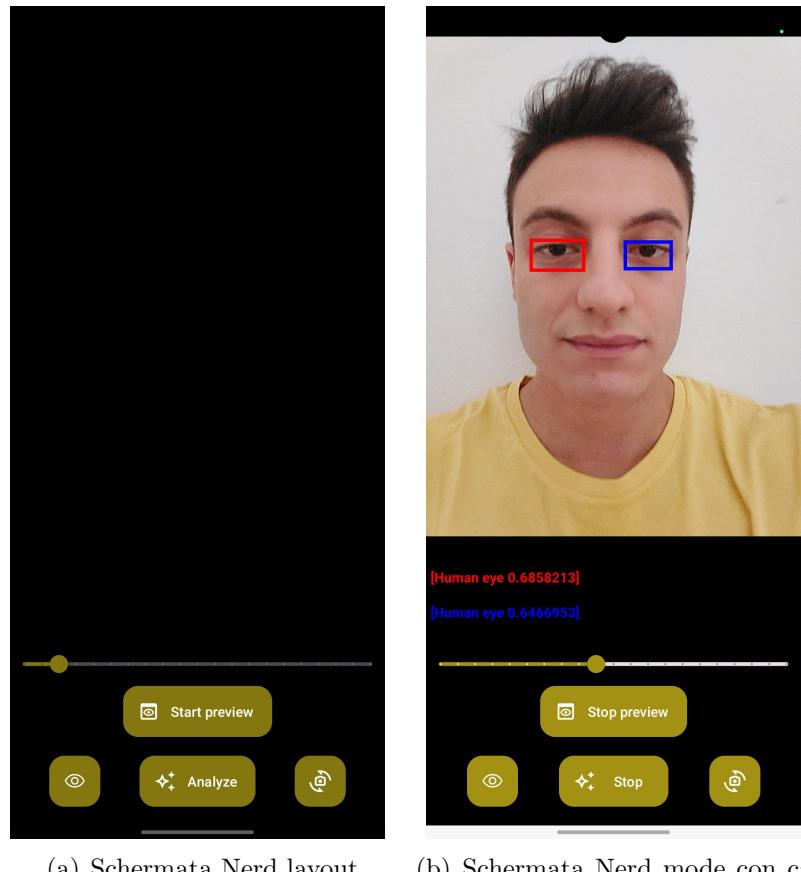


Figura 3.1: Schermata Nerd

La schermata ”Nerd” è stata pensata per mostrare in modo grafico e con la maggior facilità possibile tutte le informazioni che la telecamera del dispositivo è in grado di acquisire.

Nella parte più alta dello schermo viene mostrato ciò che la telecamera acquisisce e sovrapposto l’analisi della stessa immagine tramite la rete neurale.

Per ogni occhio rilevato dalla rete neurale appare a schermo un rettangolo di colori diversi per evidenziare in che posizione è stata effettuato il tracciamento. Ulteriormente viene indicato per ogni identificazione qual è la sua accuratezza con un valore numerico decimale nell’intervallo  $[0, 1]$ .

### 3.2.1 Preview

Il bottone ”*Start preview*” (vedi figura 3.1(a)) avvia l’acquisizione dalla fotocamera del dispositivo, come fotocamera predefinita viene utilizzata quella ”interna” ovvero quella rivolta verso l’utente. È possibile anche usare la fotocamera ”esterna” mediante il bottone ”*Switch Camera*” (vedi 3.2.2).

La dimensione della *preview view* deve essere adattata alla dimensione del sensore della camera e quindi dell’immagine risultante. Queste dimensioni dipendono sia dal dispositivo su cui viene eseguita l’applicazione sia da quale fotocamera viene selezionata, pertanto dinamicamente l’applicazione deve adattare le dimensioni della *preview view* e coerentemente il layer su cui vengono disegnati i rettangoli di tracking degli occhi. Se questa operazione non venisse effettuata o non eseguita dinamicamente la rete opererebbe comunque in maniera corretta ma la visualizzazione dei risultati risulterebbe completamente errata mostrando l’occhio in in punto dello schermo e il corrispondente rettangolo in una posizione diversa.

### 3.2.2 Switch camera

Il bottone ”*Switch camera*” (vedi figura 3.1(a)) permette di scegliere quale camera utilizzare per la visualizzazione e per la successiva analisi. La scelta è tra la fotocamera ”interna” ovvero quella rivolta verso l’utente, lato schermo, e quella ”esterna” nell’altro lato del dispositivo.

Per l’acquisizione dalla camera viene utilizzato in framework **CameraX**[1] e per costruzione, nell’attuale stato dell’arte, non permettere di scegliere, se disponibili, quale delle fotocamere esterne utilizzare, verrà dunque utilizzata in modo automatico la fotocamera esterna principale.

### 3.2.3 Analyze

Il bottone ”*Analyze*” (vedi figura 3.1(a)) è abilitato ad essere premuto solo quando è contemporaneamente abilitata la funzione ”*Preview*”.

Alla pressione del bottone viene innescata l’analisi in tempo reale delle immagini acquisite e passate alla rete neurale per il processamento *Eye-Tracking* (vedi 2.1).

Al termine dell'analisi i risultati del tracciamento vengono mostrati a schermo su un livello grafico sovrapposto a quello di *preview* della camera. Si ottiene una visualizzazione del contorno degli occhi trovati tramite rettangolo colorato e la corrispettiva accuratezza (vedi 3.1(b)).

È stato fondamentale gestire correttamente l'orientamento dell'immagine qualora l'utente cambi fotocamera. La fotocamera interna infatti produrrà un risultato specchiato rispetto alla visualizzazione, pertanto prima di mostrare a video i risultati ottenuti andranno orientati in base alla camera d'origine di tali dati.

Posizionato immediatamente sotto la parte di *preview* è stato aggiunto uno *slider* che permette di filtrare i risultati ottenuti tramite valore minimo di accuratezza.

### 3.2.4 Attività progettuale: Riconoscimento pupilla

Il bottone ”*Pupil-Tracking*” (vedi figura 3.1(a)) è abilitato solo qualora sia già in esecuzione l'analisi dell'immagine.

La funzionalità di ”*Pupil-Tracking*” acquisisce per ogni occhio rilevato dalla fase di analisi dell'immagine un occhio e su questa porzione di immagine avvia il rilevamento della pupilla.

Al termine del processamento della rete per il riconoscimento della pupilla (vedi 2.2) i risultati vengono mostrati a video nelle stesse modalità di quelle descritte per ”*Eye-Tracking*”.

Qui di seguito un test effettuato direttamente su Android Studio:

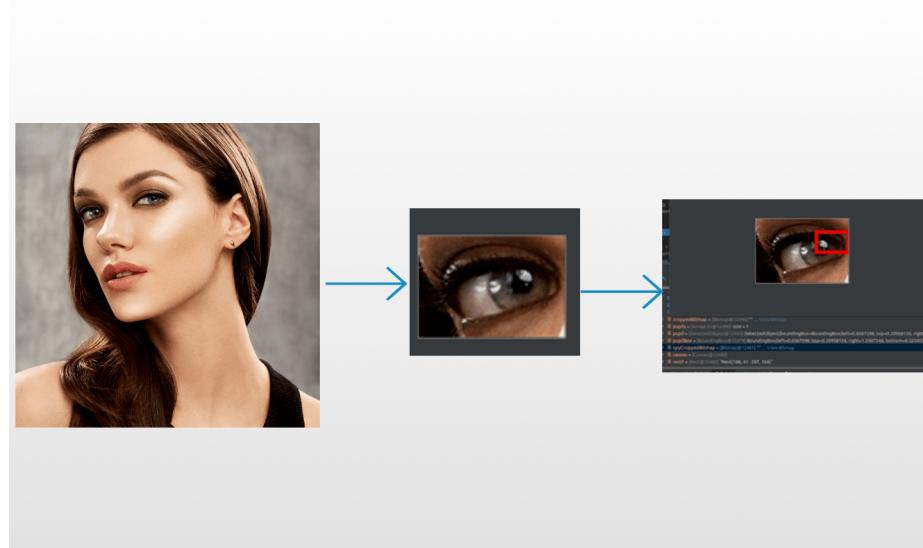


Figura 3.2: Schermata Nerd mode con pupilla

### 3.3 Calibration

La schermata ”Calibration” è usata per calibrare l’applicazione e apprendere lo spazio di movimento massimo dall’occhio.

Come è mostrato in figura 3.3 sullo schermo sono mostrati dei punti che alternativamente lampeggiano, l’utente è così portato a guardare il punto lampeggiante.

In questo modo è stato possibile apprendere lo spazio di movimento dell’occhio sullo schermo.



Figura 3.3: Schermata Calibration

### 3.4 Game

Nella fase di progettazione del gioco è stato deciso di implementare un gioco a quiz, in cui l’utente muovendo gli occhi avrebbe selezionato la risposta desiderata per la domanda sottoposta.

A tal proposito si è deciso di far riferimento al servizio OpenDB[2] (*Open Trivia DB*) da cui tramite richieste *HTTP* vengono acquisite 10 domande per

iterazione in formato *JSON*. Successivamente vengono estratte solo le domande con quattro risposte possibili e mostrate a video i quesiti in modo sequenziale.

Il gioco in background abilita la fotocamera frontale e in tempo reale inizia il tracciamento degli occhi. Quando il tracciamento ha successo a video viene mostrato un cerchio rosso che rappresenta la posizione degli occhi. Muovendo gli occhi, e quindi il cerchio, su una risposta la si conferma. Se la risposta data è corretta il gioco mostrerà un'altra domanda fino a quando l'utente non sceglie di terminare il gioco.

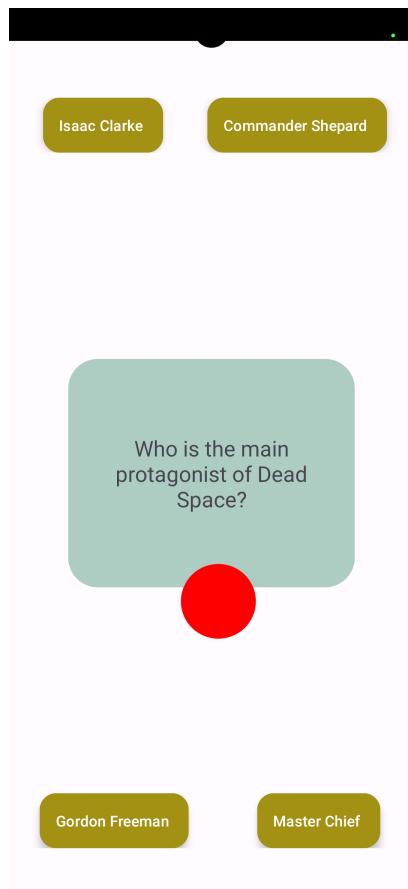
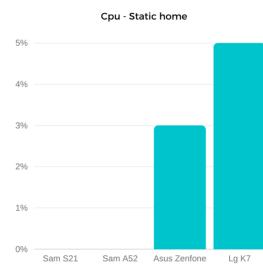


Figura 3.4: Schermata Game

## 3.5 Prestazioni

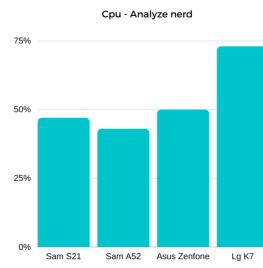
Di seguito vengono mostrati alcuni dati (l'utilizzo della CPU e della memoria) riguardanti le prestazioni lato Android utilizzando 4 differenti dispositivi.



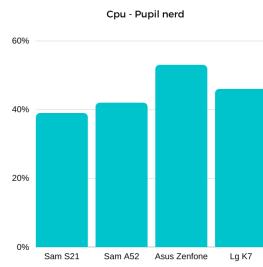
(a) Static home



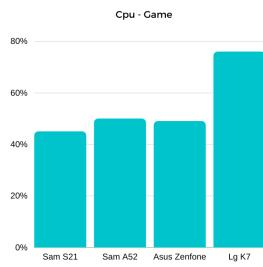
(b) Preview nerd



(c) Analyze nerd

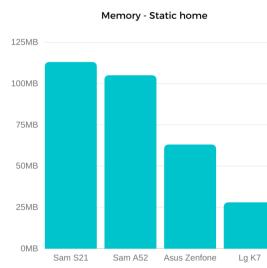


(d) Pupil nerd

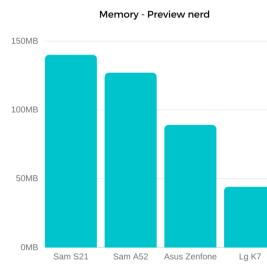


(e) Game

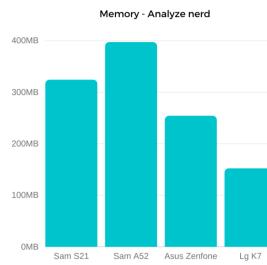
Figura 3.5: CPU usage



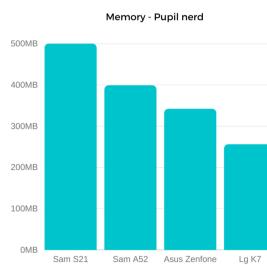
(a) Static home



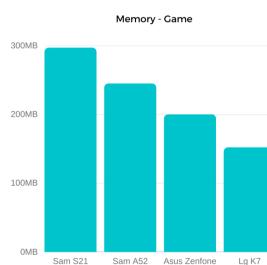
(b) Preview nerd



(c) Analyze nerd



(d) Pupil nerd



(e) Game

Figura 3.6: Memory usage

# **Capitolo 4**

## **Conclusioni**

Gli obiettivi del progetto sono stati raggiunti a pieno e con risultati soddisfacenti.

L'applicativo è conforme alle aspettative e svolge i compiti adeguatamente. Le parti realizzate in Android risultano con prestazione adeguate, nonostante i cali di precisione dovuti alla conversione in modello .tflite.

Il modello addestrato restituisce un output corretto la maggior parte delle volte con una precisione di eye tracking superiore al 85% e di pupil tracking superiore al 95%, precisione che una volta importato su Android cala, ma mantenendo sempre risultati soddisfacenti.



# Bibliografia

- [1] Camerax - a jetpack library, built to help make camera app development easier. for new apps, we recommend starting with camerax. it provides a consistent, easy-to-use api that works across the vast majority of android devices, with backward-compatibility to android 5.0 (api level 21).
- [2] Opentdb - free to use, user-contributed trivia question database.
- [3] Ssdmobilenet - ssd mobilenet v2 object detection model with fpn-lite feature extractor, shared box predictor and focal loss, trained on coco 2017 dataset with trainning images scaled to 640x640.
- [4] Tensorflow - we provide a collection of detection models pre-trained on the coco 2017 dataset.