

## 1. Introduzione

In un'epoca dove i grandi colossi dell'informatica hanno ormai monopolizzato i dati, raccogliendone enormi quantità dagli utenti delle loro piattaforme e accaparrandosi un abnorme vantaggio su tutti i competitors, sviluppare un prodotto competitivo può risultare quasi impossibile per un'organizzazione con un bacino d'utenza non molto ampio e priva delle quantità di dati di cui dispone la concorrenza.

Proprio per risolvere il problema della carenza di dati, nascono i dataset sintetici: collezioni di dati generati da calcolatori in grado di simulare una raccolta in campo reale. Questi dataset rendono possibile per chiunque perfezionare i propri algoritmi in maniera efficace e veloce: non si devono, infatti, aspettare dati provenienti dal mondo reale per iniziare il training dei propri algoritmi.

Ovviamente, anche i dati sintetici hanno i loro difetti: il rischio, detto reality gap, di un algoritmo allenato con soli dati sintetici è quello di non sapersi adattare alle situazioni reali.

Questo possibile problema è stato per anni un deterrente per l'uso dei synthetic data nelle applicazioni, tuttavia, con il miglioramento nel corso degli anni degli algoritmi di generazione di dati, il gioco ha cominciato a valere la candela grazie all'ormai ottima precisione che i dataset sintetici riescono a raggiungere a fronte di costi e tempi neanche minimamente comparabili a quelli di una raccolta dati sul campo.

Nell'ambito della computer vision, dove si colloca il lavoro di tesi che questa introduzione vuole presentare, un dataset sintetico può definirsi tale se composto da un'immagine, contenente gli oggetti la cui presenza è di interesse dell'algoritmo da allenare, e da un'etichetta che descriva

quanto contenuto nell'immagine con l'obiettivo di "insegnarlo" all'algoritmo.

Questo lavoro di tesi si pone l'obiettivo di costruire, a tal proposito, un generatore di dati sintetici. Tale generatore verrà poi sfruttato per creare un dataset che andrà ad allenare un algoritmo di riconoscimento, con lo scopo di insegnargli ad individuare le mani che vede nelle immagini che andrà ad analizzare.

Per raggiungere tale scopo, sono stati utilizzati gli strumenti descritti nei prossimi paragrafi.

## 1.1 Unity 3D 2019.2.0f1

Unity 3D<sup>1</sup> è un motore grafico progettato principalmente per lo sviluppo in C# di videogiochi ambientati in uno spazio tridimensionale.

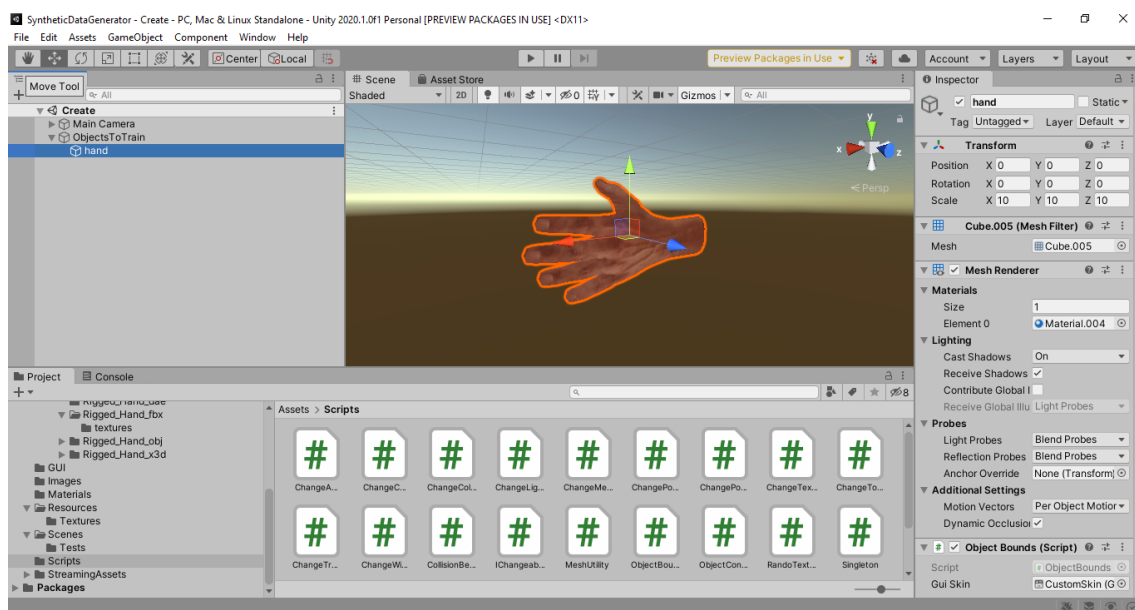


Figura 1.1 - schermata principale di Unity 3D

Essendo lo spazio tridimensionale, detto scena, offerto dall'engine popolabile e modificabile a proprio piacimento, Unity risulta molto versatile e le sue applicazioni toccano

<sup>1</sup> Scaricabile al link <https://store.unity.com/>

svariati settori, tra i quali è presente anche il machine learning.

Gli oggetti 3D presenti nella scena possono essere modificati nelle loro caratteristiche mediante l'azione di scripts: soggetto della modifica possono essere elementi come la posizione dell'oggetto, la sua rotazione o la sua visibilità.

La versione di Unity utilizzata durante il lavoro di tesi supporta modelli 3D di oggetti in formato non proprietario *.fbx*, *.dae*, *.3ds*, *.dxf* e *.obj*. Sono supportati anche formati proprietari, previa conversione in *.fbx* presa in carico da Unity stesso.

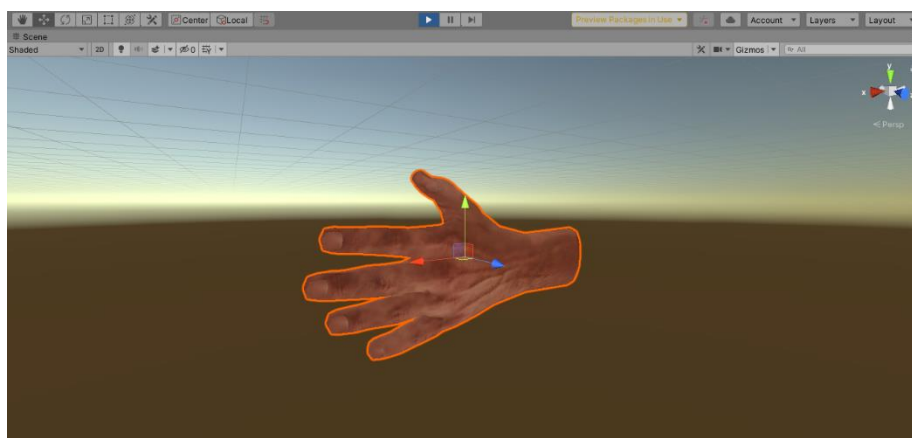


Figura 1.2 - esempio di scena utilizzata durante il lavoro di tesi

Al momento dell'importazione di un file nel suo workspace, Unity compila un documento omonimo in formato *.meta*, contenente informazioni su come l'importazione debba avvenire per quanto riguarda il file in questione.

```
1 fileFormatVersion: 2
2 guid: 9fb6f46b0c4cb604d9d15f991ee26384
3 DefaultImporter:
4   externalObjects: {}
5   userData:
6   assetBundleName:
7   assetBundleVariant:
```

Figura 1.3 - esempio di file *.meta*

È importante precisare che l'importazione di un file all'interno del workspace Unity non implica il suo utilizzo

durante l'esecuzione del software progettato mediante l'engine: per far sì che ciò avvenga, infatti, l'elemento deve essere importato nella scena. Va da sé che un file non può essere importato nella scena senza prima essere stato importato nel workspace.

Pur non essendo il miglior motore grafico in circolazione a livello di qualità dei render o di performance, Unity risulta perfetto per la generazione di dataset sintetici per la computer vision: se configurato correttamente, infatti, è in grado di generare numerose immagini adatte al training molto diverse tra loro e di etichettarle in pochi e non molto complessi passaggi.

## 1.2 TensorFlow 1.15

TensorFlow è una libreria open source sviluppata da Google Brain per lo sviluppo di modelli di machine learning: durante il lavoro di tesi è stata utilizzata la sua versione 1.15<sup>2</sup> per Python 3.5.6 per allenare l'API per l'object detection offerta da TensorFlow stesso a individuare la presenza di mani nelle immagini proposte. Il progresso delle performance del detector è stato monitorato attraverso l'impiego del tool TensorBoard.

Si è tentato di migrare l'intero progetto a una versione di TensorFlow più recente (2.3), ma il training dell'object detection API non risultava essere compatibile con tale release.

```
42 import numpy as np
43 import tensorflow as tf
44 import util
```

Figura 1.4 - importazione di TensorFlow in uno script Python

---

<sup>2</sup> Installabile seguendo le istruzioni al link <https://www.tensorflow.org/install>

### 1.2.1 Object detection API

L'object detection API<sup>3</sup> di TensorFlow è un framework che facilita lo sviluppo e il training di modelli per il riconoscimento di determinati oggetti all'interno di immagini.

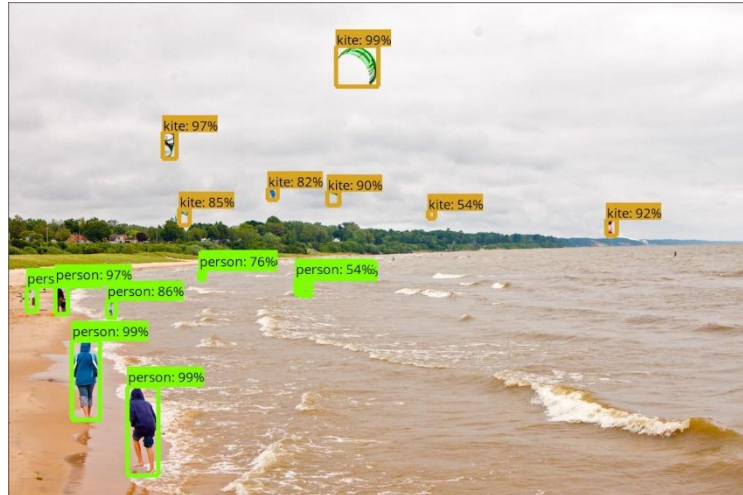


Figura 1.5 - esempio di object detector fondato sull'object detection API di TensorFlow

### 1.2.2 TensorBoard

TensorBoard, autodefinitosi "TensorFlow's visualization toolkit", è un tool che permette di visualizzare, mediante un pannello contenuto in una pagina html all'indirizzo localhost:6006, grafi e grafici relativi al modello di cui si sta effettuando l'allenamento, per poterne osservare e valutare i progressi.

---

<sup>3</sup> Installabile, insieme alle relative dependencies, seguendo le istruzioni al link [https://github.com/tensorflow/models/blob/d530ac540b0103caa194b4824af353f1b073553b/research/object\\_detection/g3doc/installation.md](https://github.com/tensorflow/models/blob/d530ac540b0103caa194b4824af353f1b073553b/research/object_detection/g3doc/installation.md)

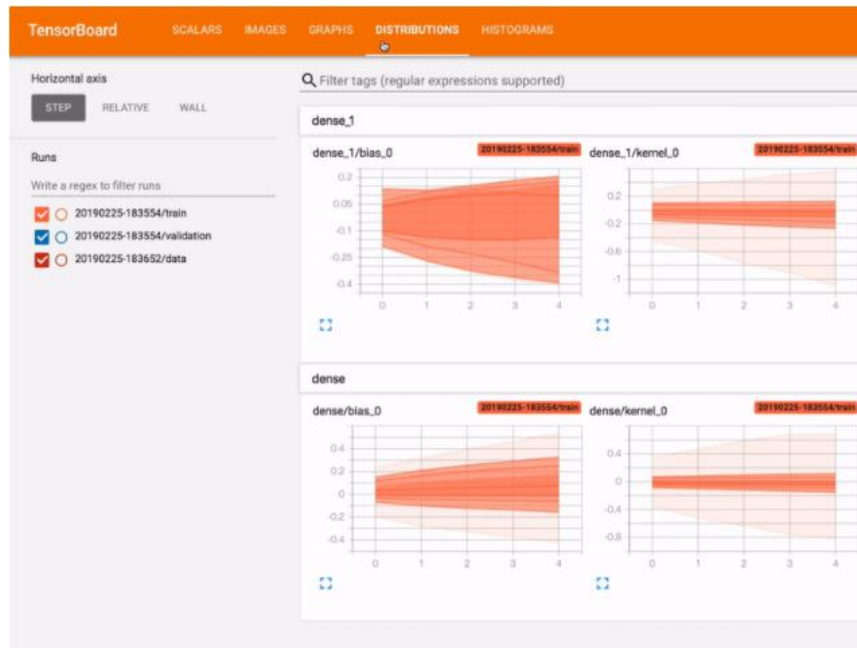


Figura 1.6 - una delle schede della dashboard di TensorBoard

### 1.3 Fatkun Batch Download Image

Per scaricare le immagini dalle quali partire per popolare il dataset, si è scelto di utilizzare Fatkun Batch Download Image<sup>4</sup>, estensione di Google Chrome che permette di scaricare in lotto tutte le immagini presenti nelle schede del browser in un dato momento.

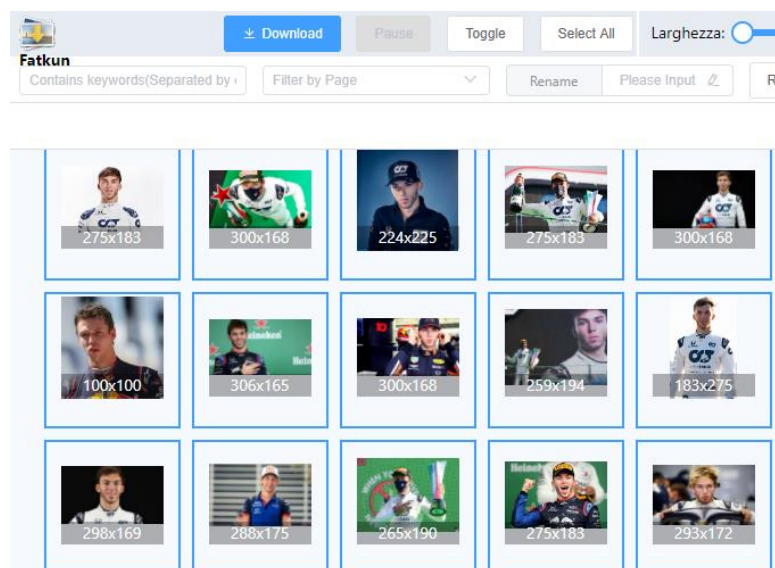


Figura 1.7 - schermata principale di Fatkun

<sup>4</sup> Scaricabile al link <https://chrome.google.com/webstore/detail/fatkun-batch-download-ima/nnjjahlikiabnchcpehckpkdeckfgnohf?hl=it>

## 2. Generazione dataset sintetico

La generazione del dataset sintetico avviene tramite un software, reperito sul web<sup>5</sup> e adattato alle specifiche del progetto, chiamato *SyntheticDataGenerator*, operante sul motore grafico Unity 3D (versione 2019.2.0f1) sotto il controllo di scripts C#.

Il compito di questo software è quello di generare immagini contenenti l'oggetto (o gli oggetti) della detection, effettuandone il labelling in un formato leggibile dallo strumento che si prenderà carico del training del detector, con la possibilità di scegliere se evidenziare visivamente i limiti spaziali di tali oggetti tramite bounding boxes o meno.

Nel caso rappresentato da questo lavoro di tesi, l'oggetto della detection è una mano e il software che si occupa del training è TensorFlow (versione 1.15), che allenerà una sua API per l'object detection.

Il connubio tra immagini e labels delle stesse è ciò che andrà a comporre concretamente il dataset sintetico generato con il software in questione.



Figura 2.1 - esempio di immagine generabile con *SyntheticDataGenerator* e relative informazioni di labelling. L'oggetto della detection, in questo esempio, è una mano.

---

<sup>5</sup> <https://github.com/MatthewHallberg/SyntheticDataGenerator>

## 2.1 La cartella *Assets*

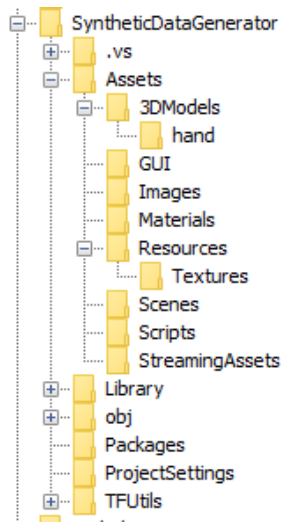


Figura 2.2

Per descrivere i file che compongono il progetto, verrà effettuato un focus sulla cartella *Assets*<sup>6</sup>, contenuta nella folder di progetto *SyntheticDataGenerator*, in particolare sulle sue subdirectories, ospitanti tutto ciò che serve per rendere il progetto funzionante.

Come anticipato nei paragrafi introduttivi, ogni elemento contenuto nella cartella *Assets* implica la presenza di un file omonimo con estensione *.meta*, del quale non verrà analizzato il contenuto.

### 2.1.1 *Assets/3DModels*

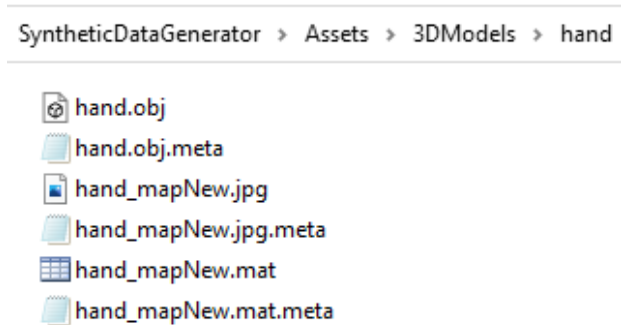


Figura 2.3

La cartella *3DModels*, come intuibile dal nome, è la cartella che ospita i modelli 3D che potranno essere utilizzati per la generazione del dataset tramite la loro importazione nella scena.

Per la generazione di un dataset adatto al training dell'hand detector, è stato utilizzato un modello 3D di mano, reperito sul web<sup>7</sup>, in formato *.obj* e contenuto nella cartella *hand*. Il file *hand.obj* all'interno dell'omonima directory,

<sup>6</sup> Verranno ignorate le altre cartelle visibili all'interno della directory di progetto *SyntheticDataGenerator*, in quanto contenenti file comuni a qualsiasi soluzione Unity, fatta eccezione per *TFUtils*, che verrà però descritta nei paragrafi inerenti alla fase di training

<sup>7</sup> <https://free3d.com/3d-model/freerealsichand-85561.html>



contenente la riproduzione mediante una maglia poligonale (polygon mesh) di una mano, va integrato con il file *hand\_mapNew.mat*, adattamento allo spazio 3D di *hand\_mapNew.jpg*, che fornisce copertura alla mesh, aggiungendole la pelle e dettagli, altrimenti mancanti, come unghie e rughe.



*Figura 2.4 - hand.obj + hand\_mapNew.mat*



*Figura 2.5 -  
bounds iniziali  
oggetto*



*Figura 2.6 -  
bounds oggetto  
ricalcolati in  
funzione della  
posizione della  
telecamera*

Qualsiasi modello 3D inserito all'interno della scena Unity presenta inizialmente bounds statici, indipendenti dalla prospettiva dalla quale l'oggetto viene osservato: questi limiti spaziali verranno ricalcolati in base alla posizione della telecamera per ogni oggetto presente nella scena ad ogni spostamento dello stesso - spostamento che coinciderà con la generazione di una nuova immagine, di modo da garantire un'informazione precisa sulla porzione di immagine nella quale l'oggetto si trova, predisponendo un dataset accuratamente

labellato.

Da ora in poi, quando si parlerà di limiti spaziali o di bounds di un oggetto, si farà riferimento a quelli calcolati in funzione della posizione della telecamera.

### 2.1.2 *Assets/GUI*

SyntheticDataGenerator > Assets > GUI

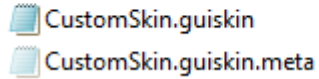


Figura 2.7

All'interno della cartella *GUI* (abbreviazione di Graphical User Interface) è presente un unico file contenente tutte le direttive necessarie alla creazione di uno

spazio, all'interno dell'interfaccia che mostrerà in real-time le immagini generate, pronto ad accogliere bounding boxes, riquadri colorati che cingeranno tra i loro lati gli oggetti della detection.

La presenza delle bounding boxes è opzionale e sarà l'utente finale a consentirla o meno: durante il lavoro di tesi, ad esempio, sono state utilizzate bounding boxes per verificare se i bounds della mano venissero calcolati correttamente o meno, ma si è deciso di escluderne la presenza nel dataset finale.

Il file *CustomSkin.guiskin* è strettamente dipendente dal contenuto di *Assets/Images*.

### 2.1.3 *Assets/Images*

SyntheticDataGenerator > Assets > Images

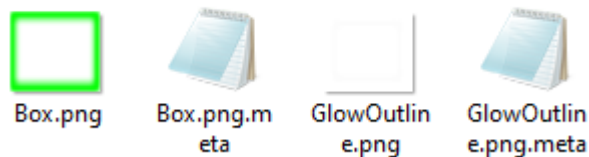


Figura 2.8

La cartella analizzata in questo paragrafo contiene tutto il materiale necessario per la rappresentazione di bounding boxes: queste saranno infatti formate

dal connubio tra i files *Box.png* (riquadro verde) e *GlowOutline.png* (illuminazione bordi del riquadro) e saranno ridimensionate in funzione dei limiti spaziali dell'oggetto a cui fanno riferimento, andando a fornire un'informazione visiva sulla porzione di immagine all'interno della quale è

presente ciò che ci interessa riconoscere. Le bounding boxes - nel caso l'utente lo desiderasse - appariranno sull'interfaccia che riporta, durante il ciclo di esecuzione, le immagini generate (predisposta ad accoglierle dal contenuto di *Assets/GUI*); la loro presenza, tuttavia, si estenderà anche alle immagini contenute nel dataset in quanto le fotografie contenute in quest'ultimo non sono altro che una cattura dell'interfaccia sopracitata.



Figura 2.9 - esempio di immagine generata con aggiunta di bounding box

#### 2.1.4 *Assets/Materials*

SyntheticDataGenerator > Assets > Materials

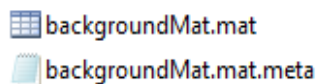


Figura 2.10

Questa subdirectory contiene il file *backgroundMat.mat*, materiale che ricoprirà il piano frontale alla telecamera operante da sfondo per il dataset.

L'immagine che il file "spalma" sul piano di background, adattandola allo spazio 3D mediante file sopracitato, cambierà ad ogni iterazione del processo di generazione e sarà pescata dalla cartella *Assets/Resources/Textures*.

### 2.1.5 *Assets/Resources*

SyntheticDataGenerator > Assets > Resources

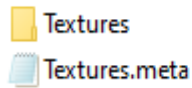


Figura 2.11

La cartella *Assets/Resources* contiene la directory *Textures*, all'interno della quale sono poste tutte le fotografie che fungeranno da background per le immagini del dataset che verrà generato.



Figura 2.12 - una delle immagini di background utilizzate durante la generazione

Le 1020 immagini presenti in *Textures* sono state scaricate mediante la già citata estensione Chrome chiamata Fatkun Batch Downloader e raffigurano, perlopiù, l'interno di abitazioni.

La scelta di utilizzare immagini con questo specifico soggetto è dettata dal fatto che l'efficacia del detector allenato con questo dataset verrà verificata mediante le immagini provenienti da una webcam posta all'interno di un'abitazione: si è quindi cercato di rendere il dataset il più rappresentativo possibile della situazione reale all'interno della quale si sarebbe effettuata, almeno durante la fase di testing, la detection.

### 2.1.6 *Assets/StreamingAssets*

SyntheticDataGenerator > Assets > StreamingAssets

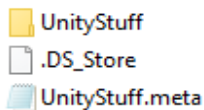


Figura 2.13

Questa subdirectory, inizialmente vuota, una volta terminato il ciclo di generazione immagini andrà a contenere la cartella

*UnityStuff*, che ospiterà il dataset sintetico e quanto necessario per trainare l'object detection API di TensorFlow con esso.

SyntheticDataGenerator > Assets > StreamingAssets > UnityStuff

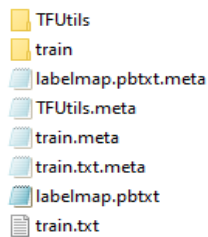


Figura 2.14

All'interno di questa cartella compariranno i file *train.txt* e *labelmap.pbtxt*, contenenti rispettivamente le informazioni di labelling del dataset e le direttive per aiutare TensorFlow a distinguere tra loro gli oggetti della detection. Saranno inoltre presenti le

cartelle *train*, contenente le immagini generate, e *TFUtils* - copia della cartella omonima contenuta in *SyntheticDataGenerator* - il cui contenuto sarà approfondito nella parte di relazione relativa al training, essendo strettamente correlato a suddetta fase.

### 2.1.7 Assets/Scripts

SyntheticDataGenerator > Assets > Scripts

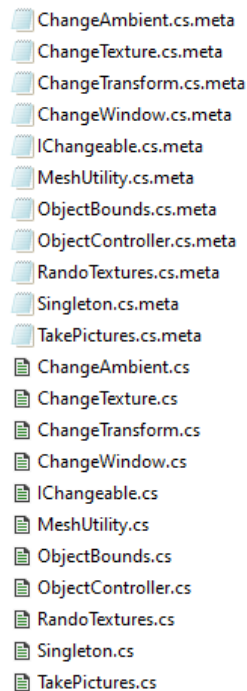


Figura 2.15

La cartella analizzata in questo paragrafo contiene tutto ciò che regola e automatizza il processo di generazione del dataset. Per capire perfettamente come questo avviene, verrà fatta una breve descrizione di ogni script C# presente in questa cartella. Ogni script rappresenta un'interfaccia oppure una classe, astratta o meno.

## Interfacce:

- *IChangeable.cs*:  
interfaccia implementata da tutte le classi il cui nome contiene la parola "*Change*", impone a queste ultime di realizzare una funzione senza ritorni chiamata *ChangeRandom()*;

## Classi astratte:

- *Singleton.cs*:  
classe che implementa il pattern Singleton<sup>8</sup>, garantendo a tutte le classi che la ereditano un'unica istanza e un singolo punto d'accesso ad essa;

Classi (eccetto *RandoTextures.cs* e *ObjectController.cs*, sono tutte classe figlie di *MonoBehaviour*<sup>9</sup>):

- *ChangeAmbient.cs*:  
cambia la luce ambientale (non proveniente da una fonte specifica, ma diffusa su tutto lo spazio 3D) della scena Unity, assegnandole una tonalità randomica di bianco, tramite la funzione *ChangeRandom()* e riportandola alla tonalità predefinita a fine ciclo di generazione;
- *RandoTextures.cs*:  
classe utility, quando chiamata in causa mescola le immagini presenti in *Assets/Resources/Textures* e ne passa una all'oggetto invocante ogni volta che viene invocata *GetRandomTexture()*, seguendo l'ordine in cui le immagini si trovano dopo lo shuffling;
- *ChangeTexture.cs*:  
cambia la fotografia di background per l'immagine che verrà generata tramite l'implementazione di

---

<sup>8</sup> Pattern che fa parte della "Gang of Four", descritta nel libro "Design patterns" che fornisce patterns utilizzabili nell'OOP

<sup>9</sup> Classe dalla quale devono derivare gli scripts assegnati a un oggetto della scena Unity e operanti direttamente su di esso

*ChangeRandom()*, ottenendola sfruttando l'accesso singleton di *RandoTextures.cs*;

- *ChangeTransform.cs*:

cambia posizione e rotazione dell'oggetto al quale è assegnata tramite l'implementazione di *ChangeRandom()*. Le coordinate spaziali sono fatte variare randomicamente all'interno di un range di valori che assicurino la visibilità dell'oggetto alla telecamera e quindi la sua presenza all'interno dell'immagine che si andrà a generare. Per quanto riguarda la rotazione, sebbene idealmente ci si vorrebbe assicurare una detection a 360 gradi dell'oggetto, si è scelto di imporre una rotazione massima di +/-20 gradi in quanto, con valori più ampi, il dataset risultava troppo dispersivo, visto l'elevato numero di posizioni assumibili dall'oggetto, spesso molto diverse tra loro;

- *ChangeWindow.cs*:

questa classe mette a disposizione un'implementazione di *ChangeRandom()* che ridimensiona l'interfaccia utente di Unity, mantenendo le proporzioni tra la sua altezza e la sua larghezza e ridimensionando di pari passo l'immagine che andrà a finire nel dataset, essendo essa una cattura della schermata in questione, contenente quanto visto dalla Main Camera con l'eventuale aggiunta di bounding boxes;

- *MeshUtility.cs*:

classe utility la cui unica funzione, esigente come parametro un oggetto presente nella scena Unity, ritorna i riferimenti alle mesh degli oggetti figli di quello passato come parametro. Non dovrà essere assegnata a nessun oggetto della scena proprio a causa di questa capacità di lavorare sul parametro passato;

- *ObjectBounds.cs*:

classe che, prelevando le mesh collegate all'oggetto a cui è assegnata sfruttando *MeshUtility.cs*, ridefinisce i limiti inizialmente statici dell'oggetto facendoli coincidere con i vertici della mesh visibili aventi x e y minime e massime: si configureranno, dunque, quattro coordinate rappresentanti bounds non più relativi all'oggetto, bensì alla sola parte visibile di esso. Nel caso l'utente lo richiedesse tramite il passaggio di un parametro booleano, i bounds saranno evidenziati da una bounding box, che sarà presente anche nell'immagine del dataset oltre che nell'interfaccia utente.

- *ObjectController.cs*:

classe derivante da Singleton, funge da centro di controllo delle sovrapposizioni tra oggetti, andando a disattivare gli oggetti le cui bounding boxes si sovrappongono a quelle di altri oggetti per una percentuale maggiore di quella definita dall'utente. La presenza di questo script all'interno del progetto è pensata per rendere il generatore più scalabile, aprendo alla generazione di immagini con disruptors o semplicemente con più oggetti da identificare;

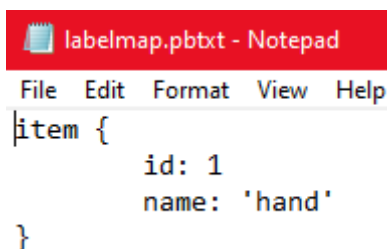
- *TakePictures.cs*:

script principe del progetto, si occupa innanzitutto di predisporre *Assets/StreamingAssets/UnityStuff* a contenere il risultato della generazione creando i file *train.txt* e *labelmap.txt* e la cartella *train*, copiando nel contempo la cartella *TFUtils* contenuta in *SyntheticDataGenerator*.

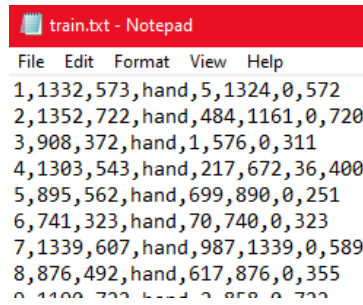
Successivamente, viene compilato - in un formato adatto ad essere interpretato da TensorFlow - *labelmap.pbtxt* con tutte le informazioni relative agli oggetti della detection presenti nella scena - anche quelli non



visibili: ad ognuno di loro viene assegnato un numero identificativo e un nome.



andare ad allenare il suo modello di detection.



```
train.txt - Notepad
File Edit Format View Help
1,1332,573,hand,5,1324,0,572
2,1352,722,hand,484,1161,0,720
3,908,372,hand,1,576,0,311
4,1303,543,hand,217,672,36,400
5,895,562,hand,699,890,0,251
6,741,323,hand,70,740,0,323
7,1339,607,hand,987,1339,0,589
8,876,492,hand,617,876,0,355
```

Figura 2.17 - esempio di compilazione *train.txt*

### 2.1.8 Assets/Scenes

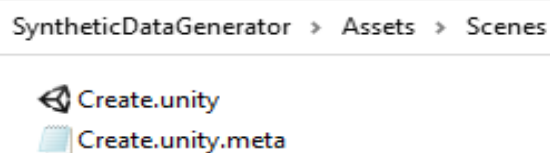


Figura 2.18

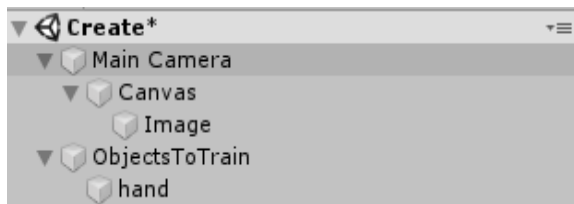


Figura 2.19 - oggetti della scena Unity utilizzati durante il lavoro di tesi

La cartella analizzata in questo paragrafo contiene quanto necessario per concretizzare il processo di generazione immagini: il file *Create.unity*, infatti, contiene una scena Unity appositamente predisposta per il fine sopracitato. La scena in questione si

compone di due oggetti principali:

- *Main Camera*: telecamera o, dato il suo impiego, fotocamera di scena; quanto catturato da essa sarà la base delle immagini del dataset. Presenta un oggetto figlio, *Canvas*, piano frontale alla telecamera, che presenta a sua volta un erede, *Image*, che andrà a contenere l'immagine di sfondo. A *Image* sono assegnati gli scripts *RandoTextures.cs*, che fungerà da utility, e *ChangeTexture.cs*, che si occuperà di cambiare il background, contenuto in *backgroundMat*;

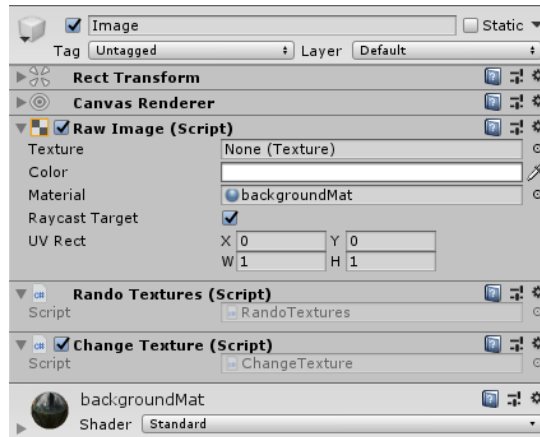


Figura 2.20 - scripts e materiali assegnati a *Image*

- ***ObjectsToTrain:***

oggetto astratto, ad esso sono assegnati gli scripts *ObjectController.cs*, *ChangeWindow.cs*, *ChangeAmbient.cs* e *TakePictures.cs*.

Qualsiasi oggetto che si vuole far apparire nel dataset dovrà essere aggiunto alla scena come oggetto figlio di *ObjectsToTrain*, di modo da ereditare gli scripts sopracitati.

Tuttavia, per il corretto funzionamento del software, è necessario assegnare ad ogni oggetto figlio gli scripts *ObjectBounds.cs* e *ChangeTransform.cs*, di modo da poter spostare ogni oggetto e calcolarne i bounds indipendentemente dagli altri oggetti e ottenere il risultato cercato.

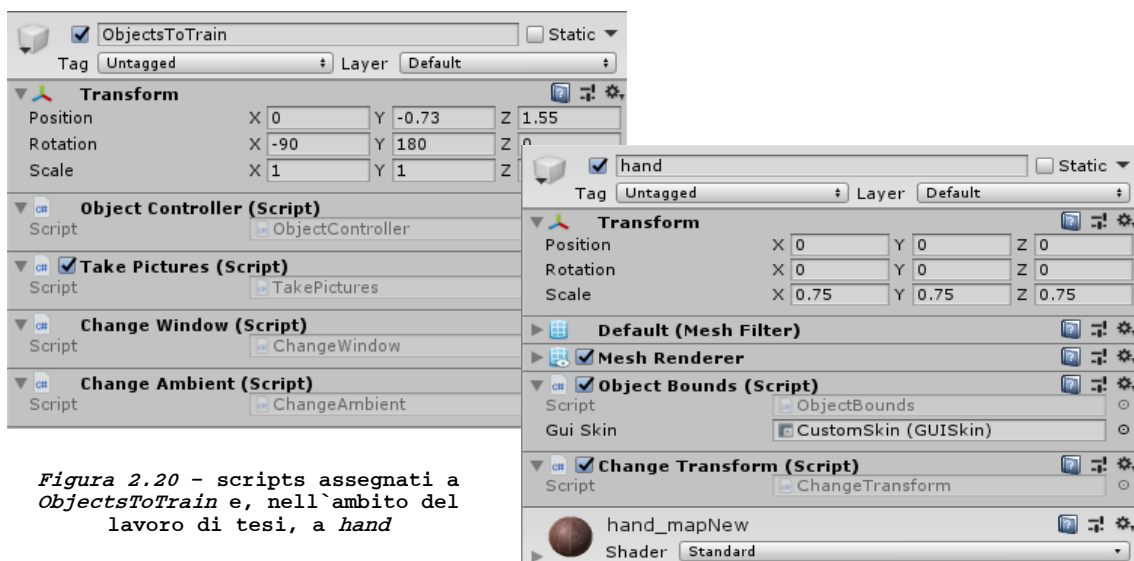


Figura 2.20 - scripts assegnati a *ObjectsToTrain* e, nell'ambito del lavoro di tesi, a *hand*

Una volta aggiunti gli oggetti di cui si vuole allenare la detection a *ObjectsToTrain*, sarà sufficiente premere il tasto play dell'editor Unity per generare il numero di immagini specificato in *TakePictures.cs*, script che viene fatto partire alla pressione del tasto play. Nel lavoro di tesi l'unico oggetto che appare sotto *ObjectsToTrain* è l'oggetto *hand*, la cui rappresentazione grafica discende dal file *hand.obj* contenuto in *Assest/3DModels*.

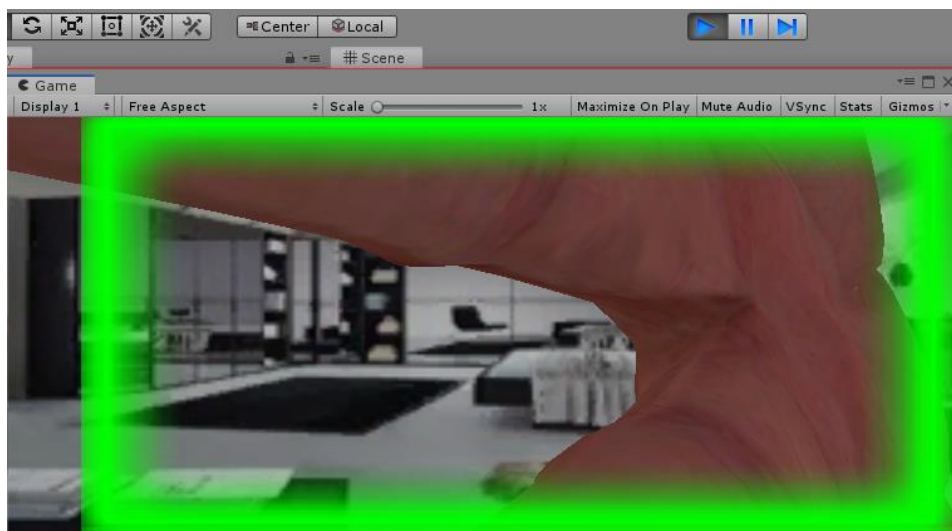


Figura 2.21 - interfaccia utente, contenente un'immagine in via di salvataggio con sovrapposta la relativa bounding box. In alto si può osservare il tasto play, premuto per far partire la generazione

### 3. Training dell'object detection API

#### 3.1 Introduzione generica al training

La computer vision è l'insieme di tutti quei task che consentono di automatizzare i comportamenti di un computer, permettendogli di analizzare direttamente un'immagine.

Considerando l'andamento delle performance degli algoritmi che operano nell'ambito della computer vision, è impossibile non considerare come negli ultimi anni, questo strumento abbia fatto passi da gigante nel campo dell'object detection.

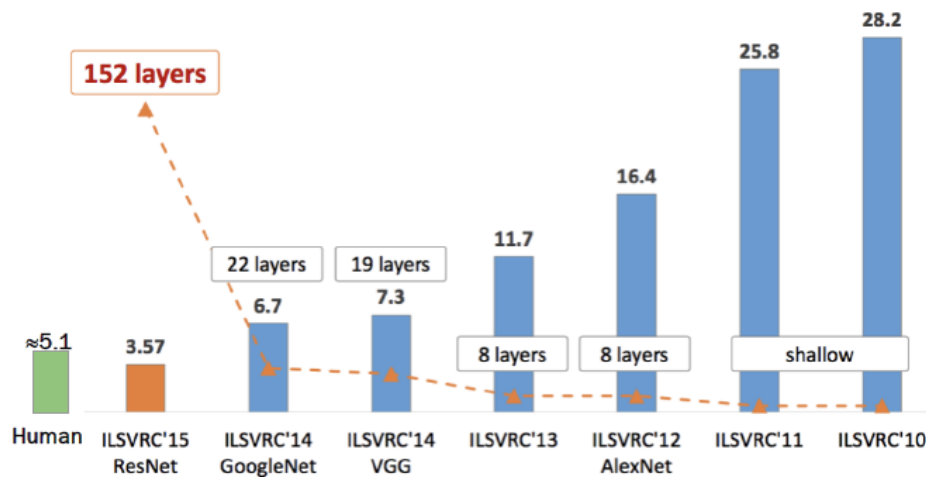


Figura 3.1 - immagine relativa alla progressione della percentuale di errore all'interno dei sistemi di computer vision.

Già dal 2015, possiamo notare come la percentuale relativa ad errori di classificazione, sia diminuita drasticamente, arrivando addirittura a superare l'errore umano.

Uno dei principali fattori, responsabili di questo successo e soprattutto senza il quale il concetto di computer vision non avrebbe ragione di esistere, è il training.

La fase di training dipende fortemente dalla capacità computazionale della macchina su cui viene eseguita, non è quindi un caso se la straordinaria evoluzione della computer vision coincida con l'introduzione, all'interno del

training, delle GPU che permettono di velocizzare i tempi di allenamento della rete.

Il training è la fase del processo di object detection, nella quale gruppi di dati vengono utilizzati per addestrare un sistema a riconoscere determinati oggetti, nel caso specifico si tratta di una mano.

Ancora prima, però, di poter parlare di training è necessario citare due attività ad esso correlate: la creazione del dataset<sup>11</sup> di immagini contenenti l'oggetto da rilevare e il concetto di classificazione delle immagini.

Quest'ultima viene svolta grazie ad un classificatore, il cui scopo primario è quello di analizzare un'immagine e restituire la probabilità che contenga una mano.

Per fare ciò, il classificatore deve essere stato addestrato su un insieme di dati contenente immagini etichettate come mani ed è proprio a questo punto che entra in gioco il training.

Tramite il training, si va ad allenare il classificatore, su un set di dati sintetici, al fine di rilevare la posizione ma soprattutto la presenza di una mano nell'immagine.

Il training è basato su una tecnica di apprendimento supervisionato, ovvero un metodo che, tramite una serie di step di allenamento, permette al sistema di elaborare automaticamente previsioni dell'output sulla base dell'input che viene fornito.

Il sistema composto da algoritmo addestrato, dati e parametri operativi costituisce il modello.

---

<sup>11</sup> La creazione del dataset è stata ampiamente analizzata nel capitolo 2

Un aspetto strettamente connesso con il training è la loss function, ovvero una quantità che mette in relazione il valore predetto dal modello e il valore corretto, del dataset di training.

Ovviamente più la loss è piccola più il modello è accurato e tendenzialmente il suo valore tende a diminuire progressivamente con l'aumentare degli step effettuati.

Tornando al concetto di training, tutti i dati utilizzati per l'addestramento, prendono il nome di training set e consistono in un vettore di immagini in input a cui viene successivamente associata una risposta o una determinata classificazione.

Una volta eseguito, l'algoritmo di training apprende, in base alla risposta o alla classificazione, quali sono le caratteristiche fondamentali che permettono di individuare gli elementi appartenenti alla categoria d'interesse (la mano) all'interno dell'immagine.

Successivamente alla fase di apprendimento, è necessario verificare la correttezza e le prestazioni del modello, eseguendolo su un insieme di dati, chiamato test set, per rilevare eventuale overfitting o underfitting rispetto al training set.

In base alle performance del modello si può parlare di:

- Underfitting: il modello ha prestazioni scarse sul training set perché non è in grado di ridurre l'errore tra i dati in input e i valori di uscita.
- Overfitting: il modello funziona bene con il training set, ma non con il test set, perché memorizza i dati che ha osservato, ma non è in grado di generalizzare il modello con dati mai osservati.
- Well-fitted: il modello restituisce buoni risultati sia con il training set che con il test set.

Per il progetto, sono stati utilizzati per il training set, dati sintetici creati con Unity, mentre come test set, le immagini delle nostre mani riprese tramite webcam del computer.

### 3.2 Basi per il training

Il punto di partenza per effettuare il training è la cartella di progetto *tesi*, nella quale si va ad effettuare la clonazione della directory *models*, fornita da GitHub per TensorFlow 1.8.0, tramite il comando: `git clone https://github.com/tensorflow/models.git`.

*Models* non è altro che una repository contenente una serie di diverse implementazioni di modelli e soluzioni di modellizzazione per gli utilizzatori di TensorFlow.

In particolare, la raccolta di modelli di nostro interesse è *research* che contiene implementazioni di codice e modelli pre-addestrati.

In questa cartella si trovano le due subdirectories che verranno utilizzate per effettuare la fase di training: *UnityStuff* e *object\_detection*.

### 3.3 *UnityStuff*

*UnityStuff* è la cartella, creata in ambiente Unity, che contiene la maggior parte delle informazioni necessarie per procedere con il training.

Questa directory non è presente in *research*, già dal momento in cui si effettua la clonazione di *models*, ma viene aggiunta in un secondo momento, a seguito della generazione del training set.

All'interno di *UnityStuff* troviamo alcune cartelle indispensabili per il training: *trainOutput*, *train*, *finalOutput* e *TFUtils*



Nome	Ultima modifica	Tipo	Dimensione
finalOutput	12/09/2020 16:58	Cartella di file	
TFUtils	08/09/2020 13:40	Cartella di file	
train	08/09/2020 13:41	Cartella di file	
trainOutput	12/09/2020 16:34	Cartella di file	
labelmap.pbtxt	25/08/2020 21:41	File PBTEXT	1 KB
labelmap.pbtxt.meta	25/08/2020 21:41	File META	1 KB
TFUtils.meta	25/08/2020 21:41	File META	1 KB
train.record	06/09/2020 17:33	File RECORD	65.167 KB
train.txt	06/09/2020 17:27	Documento di testo	52 KB

Figura 3.2 - contenuto della cartella UnityStuff.

### 3.3.1 Cartella *trainOutput*

Sostanzialmente il training è costituito da una serie di step, ognuno dei quali ha associato un valore, relativo alla loss, e un numero che lo identifica univocamente.

Nel momento in cui si esegue il comando relativo al training viene creata la cartella *trainOutput*.

All'interno in *trainOutput* si trovano una serie di file con una denominazione del tipo `model.ckpt-XXXX` dove "XXXX" corrisponde al un valore numerico che, come appena descritto, identifica lo step.

Il training non va a salvare in *trainOutput* un file `model.ckpt-XXXX` per ogni step effettuato ma, dopo ogni esecuzione, salva il valore dell'ultimo step effettuato, detto checkpoint.

Per ogni esecuzione, il training realizza un certo numero di step, valore che può essere modificato nel file `.config` del modello.

Quando il training viene riavviato non riparte quindi da zero ma dall'ultimo checkpoint memorizzato.

Infine, sono presenti una serie di file `.LAPTOP-PAPUPM4Q` che fanno riferimento ai grafici riguardanti l'andamento della loss al procedere degli step.

### 3.3.2 Cartella *train*

*Train* è la cartella nella quale vengono salvate tutti i file *.jpg*, relativi alle immagini che andranno a comporre il dataset sintetico, e i corrispettivi file *.meta* generati automaticamente da Unity.

Questo aspetto è stato già ampiamente approfondito nei paragrafi precedenti.

### 3.3.3 Cartella *finalOutput*

La directory *finalOutput* è la cartella che viene creata quando, una volta ultimato il training, si esegue l'inference del modello addestrato.

Il concetto di inference verrà approfondito in seguito, durante la definizione di tutti i comandi utilizzati per il training.

File:

- *frozen\_inference\_graph.pb*: è un file che fa riferimento ad un grafico congelato, ovvero un grafico sul quale non è possibile eseguire nuovamente un training poiché le sue variabili sono diventate costanti.  
Inoltre è serializzato, cioè la sua struttura è stata trasformata in un set di dati binari al fine di renderla facilmente trasmissibile e memorizzabile.  
Questo file costituisce il modello usato per l'object detection.
- *saved\_model.pb*: è un modello che deve essere importato nella sessione, contiene il grafico completo con tutti i pesi relativi all'allenamento, proprio come il grafico congelato, ma a differenza di quest'ultimo può essere allenato più volte perché le sue variabili non vengono memorizzate all'interno del file, inoltre non è serializzato.

- *model.ckpt*: questi file sono i checkpoint, generati durante l'allenamento, che vengono utilizzati per riprendere il training o per avere una copia di backup nel momento in cui qualcosa vada storto dopo l'allenamento.

Cartelle:

- *saved\_model*: è la cartella all'interno della quale troviamo il file *saved\_model.pb* descritto precedentemente.

### 3.3.4 Cartella *TFUtils*

Questa cartella è, sotto certi aspetti, il cuore del progetto, ovvero quella che ci permette di realizzare il training.

È proprio all'interno di questa directory che, tramite CMD, vengono eseguiti i comandi necessari ad avviare, non solo il training, ma anche l'inference e il test effettivo per verificare la bontà del progetto.

File:

- *ssdlite\_mobilenet\_v2\_coco*: questo file contiene principalmente un modello di rilevamento rapido degli oggetti, addestrato sul set di dati coco.  
SSD è un rilevatore di oggetti abbastanza veloce da poter essere utilizzato, come nel nostro caso, su video in tempo reale.  
Esistono molte varianti di SSD, in particolare quella utilizzata si avvale di MobileNet\_V2 come spina dorsale e prende il nome di SSDLite perché è dotata di convoluzioni separabili in profondità per i livelli SSD.

MobileNet\_V2<sup>12</sup> è un'architettura di rete neurale che funziona in modo molto efficiente su dispositivi mobili.

Coco<sup>13</sup> è un set di dati di rilevamento, segmentazione e didascalia di oggetti su larga scala.

- *CreateTFRecord.py*: durante un processo di object detection può essere utile serializzare i dati e archivarli in una serie di file che possono essere letti in modo lineare.

TFRecord è un formato semplice per memorizzare una sequenza di record binari che utilizza buffer di protocollo come libreria per una serializzazione efficiente dei dati strutturati.

I messaggi di protocollo, detti Protobuf, sono definiti da file *.proto* e sono un tipo di messaggio flessibile che rappresenta una mappatura del tipo {"string": value}.

I Protobuf sono progettati per essere utilizzati con TensorFlow e in tutte le API di livello superiore.

Il file TFRecord può essere letto solo in sequenza e contiene una sequenza di record, dove ogni record è costituito da una stringa di byte, per il payload dei dati, più la lunghezza dei dati e hash CRC32C per il controllo dell'integrità.

Infine i record vengono concatenati assieme per produrre il file.

- *testDetection.py*: è il file che contiene gli script di python che verranno eseguiti durante la fase di inference.

---

<sup>12</sup> Fonte:

<https://github.com/tensorflow/models/tree/master/research/slim/nets/mobilenet>

<sup>13</sup> Fonte: <https://cocodataset.org>

Questi script contengono, innanzitutto, il numero di classi da utilizzare, ovvero le tipologie di oggetti che si vogliono rilevare.

Ci sono inoltre il percorso relativo al file *frozen\_inference\_graph.pb*, che costituisce il modello usato per l'object detection, e la lista delle stringhe da utilizzare per aggiungere la label corretta ad ogni box contenente la mano rilevata.

Nel progetto troviamo un numero di classi pari a 1 perché si deve identificare solo un oggetto, ossia la mano, mentre la lista di stringhe consiste, in realtà, nel solo valore "hand".

Infine una parte del codice di questo file è riservata all'attivazione della videocamera USB, necessaria per effettuare il test finale di rilevamento della mano.

#### Cartelle:

- *ssdlite\_mobilenet\_v2\_coco*: in questa cartella troviamo gli stessi elementi descritti nel paragrafo precedente, riguardante la directory *finalOutput*, perché è proprio da *ssdlite\_mobilenet\_v2\_coco* che il comando di inference preleva i dati necessari alla creazione di *finalOutput* per verificare il corretto funzionamento del modello addestrato.

Nome	Ultima modifica	Tipo	Dimensione
ssdlite_mobilenet_v2_coco	08/09/2020 13:40	Cartella di file	
CreateTFRecord.py	02/09/2020 15:08	File PY	4 KB
CreateTFRecord.py.meta	25/08/2020 21:41	File META	1 KB
ObjectDetectNotes.txt	12/09/2020 18:05	Documento di testo	1 KB
ObjectDetectNotes.txt.meta	25/08/2020 21:41	File META	1 KB
ssdlite_mobilenet_v2_coco.config	25/08/2020 21:41	XML Configuration...	5 KB
ssdlite_mobilenet_v2_coco.config.meta	25/08/2020 21:41	File META	1 KB
ssdlite_mobilenet_v2_coco.meta	25/08/2020 21:41	File META	1 KB
testDetection.py	25/08/2020 21:41	File PY	3 KB
testDetection.py.meta	25/08/2020 21:41	File META	1 KB

Figura 3.3 - contenuto della cartella TFUtils.

### 3.4 *Object\_detection*

*Object\_detection* è la cartella, già presente in *research* al momento della clonazione di *models*, che racchiude al suo interno l'API TensorFlow Object Detection<sup>14</sup>, ovvero un framework open source, basato su TensorFlow, che semplifica la creazione, l'addestramento e la distribuzione di modelli di rilevamento degli oggetti.

Fondamentalmente, questa cartella, viene utilizzata nel momento in cui si vanno ad eseguire i comandi relativi a training e inference e prende in considerazione il file *export\_inference\_graph.py* e le cartelle *protos* e *legacy*<sup>15</sup>.

### 3.5 Training

Terminata la carrellata di file e cartelle che fanno da requisiti fondamentali, si procede col descrivere la parte pratica della fase di training.

Il training, di per sé, inizia nel momento in cui si ha a disposizione il training set di Unity.

La prima cosa da fare, partendo dalla cartella *tesi*, è spostarsi, tramite linea di comando, all'interno di *models\research* ed eseguire il comando  
*protoc object\_detection/protos/\*.proto --python\_out=.*

All'interno della cartella *protos* si trova l'API Tensorflow Object Detection che utilizza Protobuf<sup>16</sup> per configurare i parametri del modello e di addestramento.

---

<sup>14</sup> Fonte:

[https://github.com/tensorflow/models/tree/master/research/object\\_detection](https://github.com/tensorflow/models/tree/master/research/object_detection)

<sup>15</sup> Questi tre elementi verranno descritti nel prossimo paragrafo dedicato alla realizzazione del training e ai comandi utilizzati.

<sup>16</sup>

Fonte: [https://github.com/tensorflow/models/blob/d530ac540b0103caa194b4824af353f1b073553b/research/object\\_detection/g3doc/installation.md](https://github.com/tensorflow/models/blob/d530ac540b0103caa194b4824af353f1b073553b/research/object_detection/g3doc/installation.md)

Per poter utilizzare il framework, è necessario compilare le librerie Protobuf ed è possibile farlo tramite il comando di cui sopra.

A questo punto, spostandosi all'interno di *UnityStuff\TFUtils*, si procede con il comando *python createTFrecord.py* che va ad eseguire, tramite python, il file *CreateTFrecord.py*<sup>17</sup>.

Una volta creato il TFrecord inizia il training vero e proprio.

```
Sempre all'interno di TFUtils si digita python  
../../object_detection/legacy/train.py --  
pipeline_config_path=ssdlite_mobilenet_v2_coco.config --  
train_dir=../trainOutput/ --logtostderr.
```

Tramite questo comando viene eseguito il file *train.py*, presente in *object\_detection\legacy*, che contiene gli script di python che permettono di allenare il modello.

Il file *train.py* non fa altro che andare a settare come path, per la configurazione della pipeline, il file *ssdlite\_mobilenet\_v2\_coco.config* che, come visto in precedenza, è un modello in grado di rilevare oggetti basandosi su una rete neurale e un dataset coco.

Nel momento in cui il training ha inizio viene creata la cartella *TrainOutput*, nella quale vengono memorizzati i checkpoint step, con relativo valore di loss, e inviati i log al file standard STDERR tramite la porzione di codice *logtostderr*.

Nel momento in cui si dovesse omettere questo parametro, i log verrebbero inviati a STDOUT che non registrerebbe più

---

<sup>17</sup> La descrizione del file e i relativi effetti sono stati definiti nel paragrafo 3.3.4

nulla e i log stessi apparirebbero sullo schermo, poiché STDOUT non viene reindirizzato a nessun file.

Una volta completato il training, è necessario testare il modello per verificare che funzioni nella maniera desiderata.

Per fare ciò si deve esportare il grafico di inferenza tramite lo script *export\_inference\_graph.py*, contenuto nella cartella *object\_detection*, eseguendo il comando:

```
python ../../object_detection/export_inference_graph.py --  
input_type          image_tensor          --pipeline_config_path  
ssdlite_mobilenet_v2_coco.config          --  
trained_checkpoint_prefix ../trainOutput/model.ckpt-????? -  
-output_directory ../finalOutput.
```

I parametri da passare sono sostanzialmente 3:

- il valore del checkpoint più alto tra i file che si trovano nella cartella *trainOutput*, considerando che per il modello sono stati eseguiti circa 75.000 step di allenamento.
- il path per la configurazione della pipeline che, come già descritto per il comando di training, è sempre *ssdlite\_mobilenet\_v2\_coco.config*.
- La posizione in cui si desidera che venga posizionato il grafico di inferenza, ovvero *finalOutput*.

A questo punto non rimane altro da fare che testare realmente il funzionamento del modello digitando il comando *python testDetection.py* che va ad eseguire il file *testDetection.py*<sup>18</sup>.

---

<sup>18</sup> La descrizione del file e i relativi effetti sono stati definiti nel paragrafo 3.3.4



## 4. Risultati ottenuti

Un fattore importante, per il corretto funzionamento del progetto, in termini di accuratezza nel rilevamento della mano, è la loss function.

All'inizio del training il valore della loss, relativo ai primi step di addestramento, è risultato essere abbastanza elevato, variando continuamente tra 8 e 10.

Il motivo di ciò è dovuto al fatto che, avendo iniziato da veramente poco l'allenamento, il rilevatore non si era ancora adattato al dataset sintetico fornito.

Già dopo un centinaio di step, la loss è calata notevolmente assestandosi su un valore di circa 1 e, continuando con il training, è arrivata a mantenersi costantemente attorno a 0.4/0.5 come valore ultimo.

Di seguito sono riportati i grafici dell'andamento della loss function, generati da TensorBoard<sup>19</sup>, tramite comando `python C:\Users\Matte\anaconda3\envs\tesi\Lib\site-packages\tensorboard\main.py --logdir=../trainOutput`.

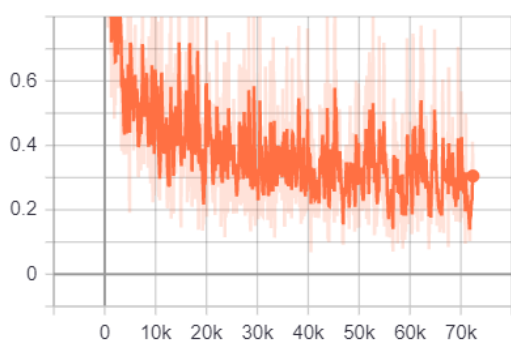


Figura 3.4 - immagine del grafico relativo a Loss/classification\_loss.  
Tag: Losses/Loss/ classification\_loss

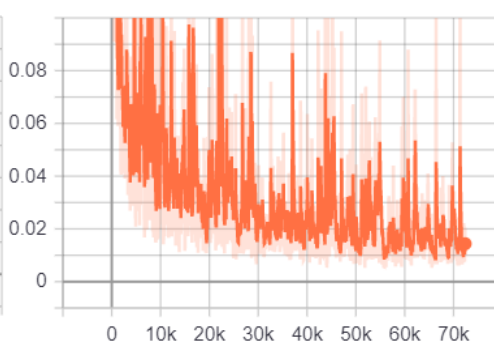
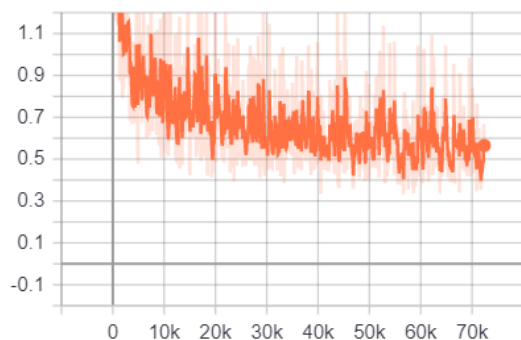


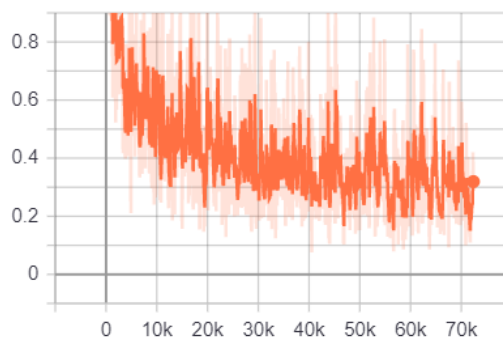
Figura 3.5 - immagine del grafico relativo a Loss/localization\_loss.  
Tag: Losses/Loss/ localization\_loss

---

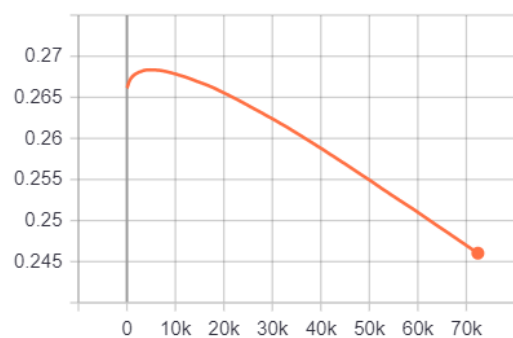
<sup>19</sup> TensorBoard è un toolkit di visualizzazione di TensorFlow per il tracciamento e la visualizzazione di metriche come perdita e accuratezza, fonte: <https://www.tensorflow.org/tensorboard>



*Figura 3.6* - immagine del grafico  
relativo a TotalLoss.  
Tag: Losses/ TotalLoss.



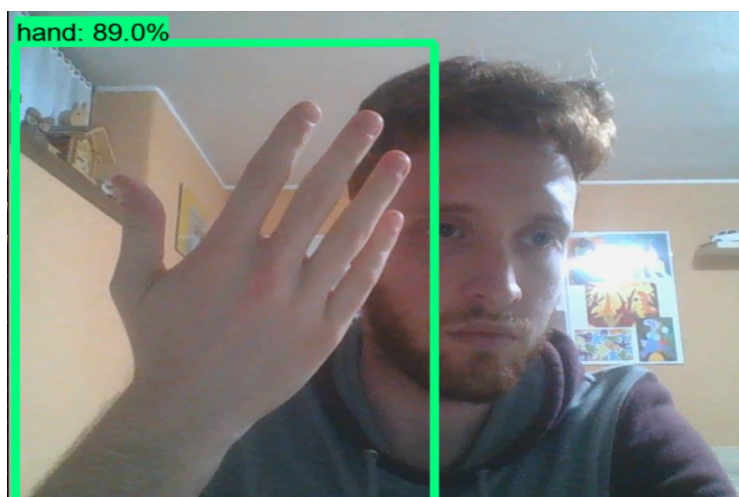
*Figura 3.7* - immagine del grafico  
relativo a clone\_loss.  
Tag: Losses/ clone loss.



*Figura 3.7* - immagine del grafico  
relativo a regularization\_loss.  
Tag: Losses/ regularization loss.

Il testing si conclude con l'attivazione della webcam USB del pc, nella quale vengono inquadrare le mani utilizzate come test set.

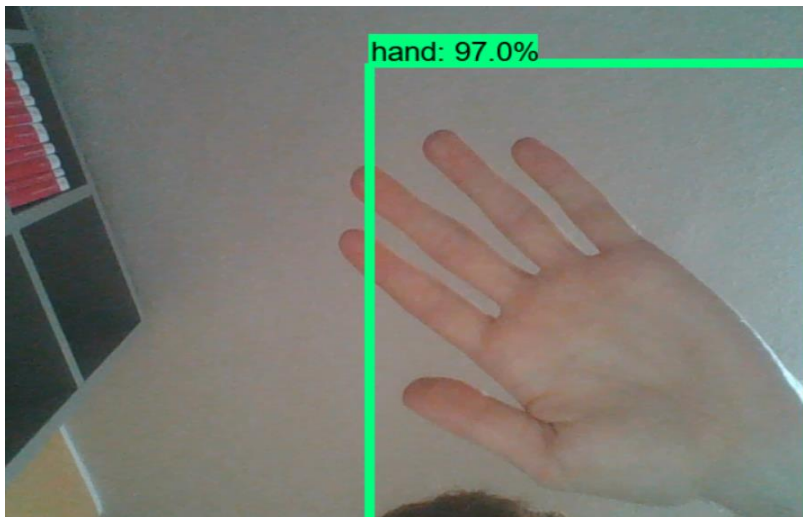
I risultati sono i seguenti:



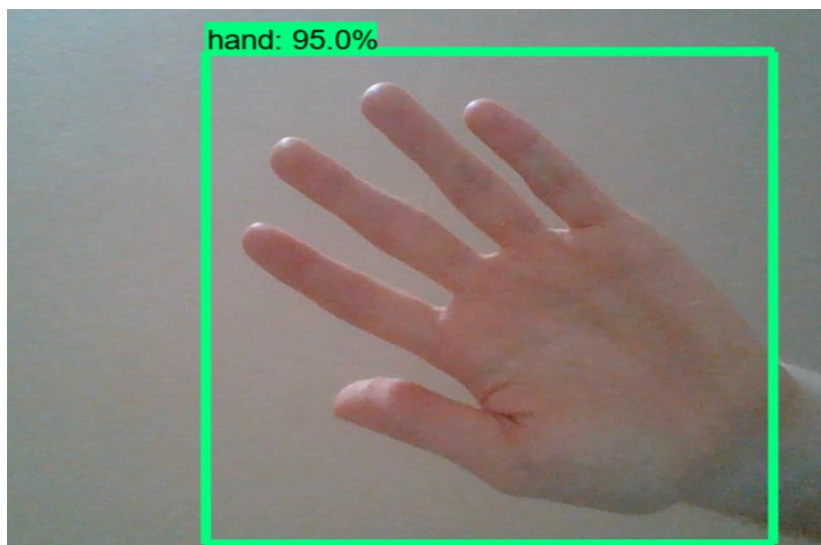
*Figura 3.8* - rilevamento del dorso della mano.



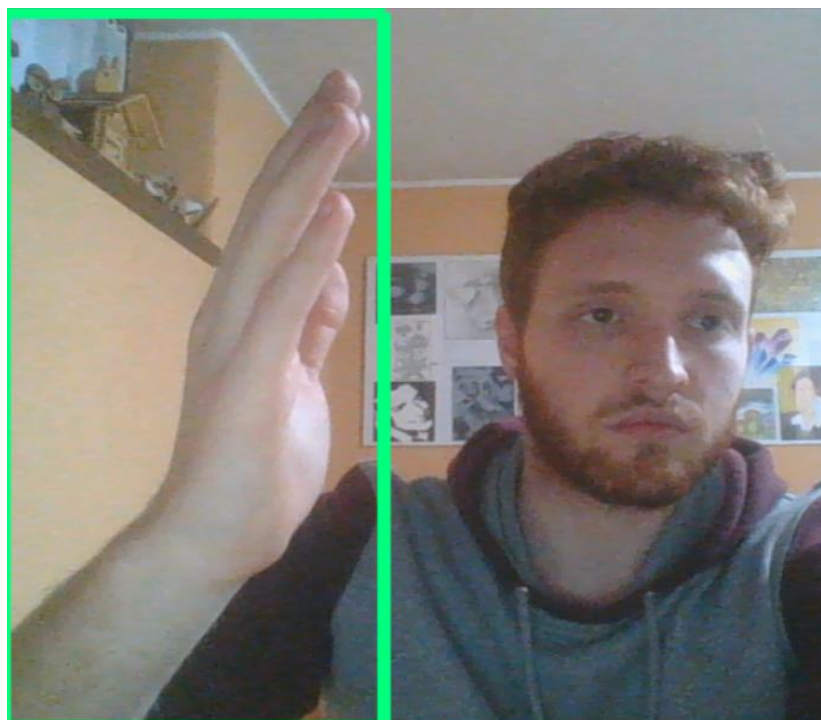
*Figura 3.9 - rilevamento del palmo della mano.*



*Figura 3.10 - rilevamento del palmo della mano.*



*Figura 3.11 - rilevamento del palmo della mano.*



*Figura 3.12 - rilevamento del profilo della mano.*

## 5. Sviluppi futuri

Ogni immagine è costituita da un insieme di caratteristiche che sono in grado di rappresentare al meglio il file originale. Queste caratteristiche vengono utilizzate come punto di partenza per parecchi sistemi di visione artificiale. Lo scopo di questo sviluppo futuro è quindi quello di estrarre dall'immagine alcune features che la rendono unica in modo da poterle poi confrontare con un dataset che fa riferimento ad una competenza specifica.

La Key Point Detection può essere fatta tramite i metodi tradizionali, oppure tramite metodi automatizzati. Per quanto riguarda i primi, il riconoscimento dei punti chiave di un'immagine avviene manualmente, cosa che porta ad avere tempi lunghi e bassa precisione. Al contrario, i metodi automatizzati, ossia quelli che rilevano automaticamente i punti, aiutano a migliorare le performance rispetto ai precedenti. A loro volta, questi ultimi hanno due diversi approcci. Il primo viene chiamato *Area Based Matching (ABM)*, si basa sulla registrazione pixel per pixel e segnala la presenza delle funzionalità senza la loro rilevazione. Il secondo è *Feature Based Matching (FBM)*.

Gli elementi caratteristici di un'immagine possono essere angoli, bordi o colori. Anche le caratteristiche si suddividono in due sottogruppi che sono le caratteristiche globali, utilizzate per rappresentare il contenuto dell'immagine, ma hanno lo svantaggio di restituire immagini confuse e quindi hanno un'utilità limitata. Effettuando però operazioni come la segmentazione delle immagini e il campionamento delle stesse, si possono ridurre tali limiti. E le caratteristiche locali per il rilevamento dei key point. Ovviamente, per poter trovare una corrispondenza tra le immagini si devono riconoscere una serie di punti. Per poterli rilevare occorrono quindi dei rilevatori, le cui

caratteristiche comprendono la robustezza, la ripetibilità, l'accuratezza, l'efficienza e la quantità.

Un secondo sviluppo da apportare al progetto è quello di realizzare un dataset sintetico basato su una visione a 360 dell'oggetto da rilevare.

Nel progetto attuale la mano viene importata all'interno dell'immagine di background mostrando il palmo alla telecamera oppure con una rotazione massima di 20 per permettere di identificarne anche il profilo.

La visione del dorso non è stata presa in considerazione perché, per forma e dimensioni, è molto simile al palmo e quindi viene rilavato comunque.

Ovviamente andando a creare un dataset contenente anche l'immagine del dorso della mano si riesce ad ottenere una detection, di questa caratteristica, ancora più accurata.

Rimanendo sempre nell'ambito della forma della mano, un'ulteriore miglioramento potrebbe riguardare l'introduzione della detection dei gesti tramite modelli di mano che li riproducono.

Un esempio banale può essere quello dell'identificazione dei numeri andando ad eseguire il training su un dataset sintetico composto da più modelli che riproducono la forma assunta dalla mano per realizzare un determinato valore numerico.

Oppure si potrebbe procedere allo stesso modo ma modificando la tipologia di gesto e realizzare, ad esempio, una sorta di controllore per il gioco "carta-sasso-forbice" che, riconoscendo le tipologie di gesto, va a decretare il vincitore.

In ambito più tecnologico, la detection delle mani e, più precisamente, dei gesti di quest'ultime potrebbe fungere anche da pannello di controllo per varie apparecchiature, ad

esempio rilevare un dito verso l'alto potrebbe significare, per un televisore oppure per un forno, rispettivamente "volume su" e "temperatura su".

Considerando invece, l'approccio con cui si realizza il rilevamento della mano, è possibile sostituire la detection delle mani, tramite video, per puntare sulla detection di oggetti stampati in 3D che rappresentano semplicemente la stampa in tre dimensioni del modello con cui è stata trainata la rete.

Un altro sviluppo, molto simile a quello appena affrontato, consiste nel modificare il dataset sintetico e non l'oggetto da rilevare, ovvero si potrebbe procedere con la scannerizzazione 3D di oggetti reali e si esegue il training utilizzando, come modello, quel preciso scan.

Tramite questo approccio, nel momento in cui si effettua il rilevamento dell'oggetto, si ottiene un risultato eccellente, in termini di accuratezza del detector, ma è possibile imbattersi in un grosso rischio.

Utilizzando come training set il modello scannerizzato di un oggetto è probabile che effettuando la detection su un altro elemento, della stessa categoria di quello usato come modello, quest'ultimo non venga riconosciuto.

Succede questo perché, la scannerizzazione di un oggetto restituisce una rappresentazione 3D molto precisa, quindi effettuando il training su un dataset composto da immagini basate su quel determinato scan, si allena la rete a riconoscere quel particolare oggetto e quindi c'è la probabilità che il detector sia talmente accurato da non riconoscere altri oggetti ad eccezione di quello scannerizzato.

Un ulteriore sviluppo futuro è quello riguardante l'ampliamento del dataset tramite l'inserimento di più modelli di mani 3D diversi tra loro per dimensioni, lunghezza delle dita o colore della pelle. Questa parte è già stata provata durante lo svolgimento del progetto, ma è stato riscontrato un problema durante la fase di training che non permette il corretto funzionamento della rete. Quindi si tratterebbe di risolvere il problema legato a tale errore e effettuare l'allenamento, in modo da avere un riconoscimento delle mani generalizzato e non fortemente influenzato dall'unico modello presente.

Infine, durante il lavoro, è stata utilizzata una versione non aggiornata di TensorFlow, ovvero la 1.15. Sarebbe opportuno migrare il tutto all'ultima versione 2.0 perché sono state rimosse le API ridondanti e inoltre si integra meglio con il runtime di Python. La migrazione è già stata provata durante la realizzazione del progetto riscontrando però un problema legato a `ssd_mobilenet_v2` che non veniva supportato dalla versione TensorFlow utilizzata.



## 6. BIBLIOGRAFIA

<https://www.tensorflow.org/>

[https://it.wikipedia.org/wiki/Unity\\_\(motore\\_grafico\)](https://it.wikipedia.org/wiki/Unity_(motore_grafico))

[https://github.com/tensorflow/models/tree/master/research/object\\_detection](https://github.com/tensorflow/models/tree/master/research/object_detection)

<https://www.tensorflow.org/tensorboard>

<https://docs.unity3d.com/2019.2/Documentation/Manual/>

[https://it.wikipedia.org/wiki/Mesh\\_poligonale](https://it.wikipedia.org/wiki/Mesh_poligonale)

[https://it.wikipedia.org/wiki/Singleton\\_\(informatica\)](https://it.wikipedia.org/wiki/Singleton_(informatica))

[https://it.wikipedia.org/wiki/Design\\_Patterns](https://it.wikipedia.org/wiki/Design_Patterns)

<https://docs.microsoft.com/it-it/dotnet/api/system.collections?view=netcore-3.1>

[https://it.wikipedia.org/wiki/Apprendimento\\_supervisionato](https://it.wikipedia.org/wiki/Apprendimento_supervisionato)

<https://machinethink.net/blog/mobilenet-ssdlite-coreml>

[https://link.springer.com/chapter/10.1007/978-3-030-31129-2\\_82](https://link.springer.com/chapter/10.1007/978-3-030-31129-2_82)

Corso di Ingegneria dei Sistemi di Controllo - Ermidoro Michele