Università degli Studi di Ferrara

FACOLTA DI INGEGNERIA

Corso di Laurea in Ingegneria Informatica e dell'Automazione

Implementazione di un classificatore d'immagini per il riconoscimento dei gesti del gioco della morra cinese

Filippini Andrea
Giacomo Polastri

Anno Accademico 2021/2022

Indice

1. Introduzione

Le reti neurali, note anche come reti neurali artificiali (o ANN, artificial neural network) sono una branca del machine learning e sono l'elemento centrale degli algoritmi di deep learning.
Il loro nome e la loro struttura sono ispirati al cervello umano, imitando il modo in cui i neuroni biologici si inviano segnali.
La struttura delle reti neurali artificiali (ANN) è composta da nodi interconnessi e divisi in livelli: un livello di input, uno o più livelli nascosti e un livello di output.
Le reti neurali si affidano ai dati di addestramento per imparare e migliorare la loro accuratezza nel tempo.
Una volta che hanno appreso, questi algoritmi sono dei potenti strumenti nella computer science e nell'AI, consentendoci di classificare e organizzare in cluster i dati ad alta velocità, anche in contesti real-time.
Esistono diverse architetture di reti neurali progettate per differenti contesti, in particolare vengono introdotte delle reti neurali specifiche per lavorare con le immagini chiamate reti neurali convoluzionali.
L'obiettivo finale del progetto consiste nell'addestrare una rete neurale convoluzionale che sia in grado di individuare la mano all'interno dello stream video fornito dalla webcam del proprio dispositivo e di classificarla correttamente in tre classi differenti che rispecchiano le tre posizioni della mano nel famoso gioco orientale morra cinese, che sono rispettivamente sasso, carta e forbice.

2. Reti convoluzionali

In questo capitolo discuteremo la tipologia di rete neurale utilizzata per il progetto e di tutte le tecniche adottate nei vari tentativi per raggiungere il risultato ottimale.
La rete neurale che abbiamo creato viene definita rete neurale convoluzionale ed è una tipologia di rete neurale specializzata nel processare dati che hanno una topologia a griglia come le immagini, che sono viste come griglie di pixel di due dimensioni.
Le immagini hanno una struttura che gode di alcune proprietà intrinseche che si possono sfruttare per la classificazione delle immagini, come la topologia dei pixel, cioè valori simili per pixel vicini, l'invarianza alla traslazione e alla scala.
L'utilizzo delle reti neurali convoluzionali rispetto alle reti neurali fully connected è dovuto ad un minor numero di parametri da gestire, nonostante siano comunque un numero alto.
In aggiunta, l'operazione di convoluzione permette di ottenere informazioni che altrimenti non riuscirei ad ottenere ed elaborare con l'uso di reti fully-connected, come i concetti di "bordo" e "angoli".
Per semplificare ulteriormente il lavoro da parte della rete, non si forniscono come input agli strati fully connected tutti i pixel delle immagini, bensì le feature estratte dai kernel in seguito alla convoluzione.
Il kernel sopra citato è una griglia di pesi sovrapposta ad un'immagine, centrata su un pixel, in modo che ogni peso sia moltiplicato per il pixel sotto di esso e che l'output sul pixel centrale sia dato dalla sommatoria dei prodotti appena calcolati.
L'operazione di convoluzione prevede di far scorrere i kernel sull'immagine orizzontale e poi verticalmente, con l'obbiettivo di rilevare delle feature locali.
Per migliorare l'applicazione dei kernel sulle immagini vengono definite due tecniche: Padding e Stride.
Il Padding è una tecnica che permette di aggiungere dei pixel intorno all'immagine in input per evitare degli effetti di bordo e per fare in modo che ogni pixel dell'immagine sia usato come centro di un kernel.
Con Zero-padding si intende un padding specifico in cui i pixel aggiunti hanno valore 0 e permette di controllare la dimensione del kernel e la dimensione dell'output indipendentemente uno dall'altro.
Invece, lo Stride corrisponde alla dimensione del passo del kernel quando viene applicato ad un'immagine, riducendo l'output se il valore è maggiore di 1.
Un'altra tecnica che si contrappone a quella precedente è il Pooling, la quale permette di ridurre le dimensioni dell'immagine mappando una patch di pixel in un singolo valore, in modo tale da mantenere in memoria un numero di valori minore.

Il pooling aiuta a rendere la rappresentazione più invariante rispetto a traslazioni dell'input e questo torna utile per capire se una feature è presente o meno indipendentemente dalla sua posizione nell'immagine.
Esistono più tipi di pooling e i quali si distinguono per il criterio con cui la patch pixel viene ridotta.
Quello utilizzato in questo progetto è il Max Pooling, dove ogni patch di pixel viene rappresentata con il suo valore massimo.
Si può affiancare la tecnica dello stride al pooling, tipicamente con un valore pari alla finestra di pooling, per ridurre l'immagine senza perdere informazioni durante tale processo.
In generale, le reti neurali convoluzionali che abbiamo implementato per ottenere il risultato desiderano hanno tutte una medesima struttura di base: uno strato di input in cui vengono passate le immagini con una determinata dimensione, degli strati convoluzionali con funzione di attivazione ReLU alternati a strati di pooling, un'operazione di flatten e alcuni strati densi con funzioni di attivazione ReLU e infine uno strato di output con funzione di attivazione softmax.
La scelta delle funzioni di attivazione utilizzate è dovuta al fatto che le conoscenze acquisite fino ad oggi nel campo delle reti neurali definiscono le funzioni di attivazione ReLU e softmax come le migliori funzioni da utilizzare per queste tipologie di progetti.

2.1 Tecniche di Regolarizzazione

Le tecniche di regolarizzazione sono delle metodologie che permettono di modificare il processo di apprendimento con lo scopo di ridurre l'errore di generalizzazione e l'overfitting, portando tuttavia ad un incremento dell'errore di training.
Esistono diverse tecniche di regolarizzazione, ma nel caso specifico di questo progetto abbiamo utilizzato il dropout e l'early stopping.
Entrando nel dettaglio, il dropout è un metodo computazionalmente economico ma potente per regolarizzare, in cui ad ogni iterazione di training vengono rimossi casualmente dei neuroni, questo per far sì che la rete non faccia affidamento su particolari percorsi tra i neuroni e quindi rendere il modello più robusto.
Le probabilità di includere o meno input o unità nascoste viene scelta arbitrariamente prima di iniziare il procedimento, nel nostro caso abbiamo settato a 0,2 per gli strati della rete neurale in cui veniva applicata la convoluzione 0,5 per gli strati densi della rete, ovvero gli strati finali.
Il dropout è una tecnica largamente utilizzata, perché non è strettamente dipendente dall'architettura scelta.
Inoltre, si è visto come la sua adozione porti ad ottimi miglioramenti in presenza di reti abbastanza grandi.
L'Early Stopping è una tecnica di regolarizzazione applicata durante il training, in cui si sceglie un criterio che se verificato termina l'apprendimento, restituendo il modello migliore trovato fino a quel momento.
Come per il dropout, il motivo per cui è stato scelto di applicare l'early stopping è quello di evitare che la rete entri in overfitting, questo monitorando la variazione del valore di loss sul validation set.
Infatti, ciò che viene fatto è controllare che tale valore abbia un andamento complessivo decrescente.
Se il valore di loss relativo al validation set inizia a crescere, significa che la rete non sta generalizzando correttamente e quindi è necessario terminare il training prima che vi sia un ulteriore peggioramento.

2.2 Ottimizzatori

Gli ottimizzatori sono varianti delle tecniche di regolarizzazione e prevedono diversi meccanismi per aggiornare i pesi.
La prima tecnica di ottimizzazione che abbiamo utilizzato è il mini-batch gradient descent che consiste nel calcolare il gradiente su un piccolo sottoinsieme del dataset diverso per ogni iterazione.
Questa tecnica è stata utilizzato in tutti gli esperimenti con una dimensione del sottoinsieme di 32 elementi.
Nel caso dell'algoritmo di ottimizzazione, invece, è stato scelto inizialmente RMSProp, il quale cerca di individuare l'ottimo riducendo progressivamente l'apporto dato dai gradienti più vecchi rispetti a quelli più recenti, seguendo maggiormente la direzione fornita dal gradient descent.
L'abbiamo utilizzato poiché inizialmente abbiamo lavorato con una rete profonda e questa tecnica di ottimizzazione è efficace per queste tipologie di reti.

Nonostante RMSProp funzionasse bene, abbiamo deciso di utilizzare un altro ottimizzatore chiamato Adam e il quale ha fornito risultati migliori rispetto al primo.
Adam migliora l'idea base di RMSProp, perché utilizzando le informazioni delle modifiche del gradiente sia del primo sia secondo ordine si ha una diminuzione delle escursioni del movimento, con l'andamento complessivo che seguirà quello ottimale.

2.3 Data augmentation

La Data Augmentation è una tecnica di regolarizzazione applicata a ciascuna immagine posta in input e che permette di addestrare la rete su più dati rispetto a quelli a disposizione, questo tramite la creazione di nuovi dati ottenuti modificando quelli originali in diversi modi e scelti in base alle necessità.
Per utilizzare implementare la data augmentation, Keras fornisce la classe Image Data Generator [1] in cui è possibile creare immagini alterate rispetto alle originali andando a modificare diversi parametri che verranno elencati in questo paragrafo.
Il primo parametro che abbiamo trattato è stato il *rescale*, ovvero una funzione di ridimensionamento dell'intervallo dei valori che ogni pixel può assumere.
In questo caso abbiamo usato rescale per normalizzare ciascuna immagine, andando a dividere ogni valore per 255, cioè il valore massimo che può assumere ciascun canale, e ottenendo un range di valori tra 0 e 1.
Il secondo parametro che abbiamo utilizzato è il *rotation_range*, il quale applica una rotazione dell'immagine, con un angolo che può variare da 0 al valore che abbiamo definito, ovvero 60 gradi, in maniera randomica.
Altri due parametri su cui abbiamo fatto leva per modificare le immagini sono stati *width_shift_range* e *height_shift_range*, che spostano al massimo di un valore deciso dall'utente la larghezza e la l'altezza dell'immagine, nel nostro caso il massimo valore di spostamento è 0,1, perché con valori più alti abbiamo notato una difficoltà nell'apprendimento da parte della rete.
Il quinto parametro è lo *shear_range*, che permette di andare a modificare l'angolo di taglio dell'immagine di un valore casuale tra 0 e il valore deciso dall'utente che nel nostro è 20 gradi.
Il sesto parametro che abbiamo adoperato per la creazione di nuove immagini è lo zoom_range, il quale ingrandisce o rimpicciolisce l'immagine in maniera casuale.
Per questo parametro è possibile definire due valori specificando limite inferiore e superiore, oppure indicando un solo valore e lo zoom verrà applicato nell'intervallo da 1 meno il valore indicato fino a 1 più il valore indicato, con 1 il valore neutro di tale operazione.
Nel nostro caso abbiamo impostato un solo valore, cioè 0,2, da cui segue un intervallo risultante di [0.8, 1.2].
Un parametro molto importante che siamo andati ad applicare e su cui abbiamo effettuato svariate modifiche è il *brightness_range*, ovvero l'intervallo di luminosità, la cui applicazione va a modificare la luminosità dell'immagine originale sottraendo casualmente al massimo il valore minimo indicato, oppure incrementando la luminosità corrente al massimo del valore scelto.
I valori che abbiamo definito nell'ultimo esperimento effettuato sono stati 0,7 e 1,3, in modo da classificare la posizione della mano anche in condizioni di scarsa o di ricca luminosità.
L'ultimo parametro che abbiamo applicato per la data augmentation è *horizontal_flip*, che semplicemente applica casualmente all'immagine corrente un ribaltamento orizzontale.

***Figura 1**. Alcuni esempi di data augmentation*

3. Tecnologie Utilizzate

In questo capitolo verrà descritto l'ambiente di lavoro utilizzato per raggiungere l'obiettivo del progetto in maniera semplice ed efficace.
Nello specifico, verranno analizzate le tecnologie software che ci hanno permesso di costruire il dataset, di implementare la rete neurale convoluzionale ed infine di allenarla affinché potesse apprendere le feature essenziali per restituire le etichette corrette durante il testing in real time.

3.1 Google Colaboratory

Google Collaboratory [2] è un ambiente di Jupyter Notebook gratuito offerto da Google che gira interamente nel cloud, non richiede alcuna configurazione ed è possibile effettuare modifiche contemporaneamente dagli utenti autorizzati ad applicare modifiche al notebook, similmente a Google Docs.
Le pagine web non sono statiche, ma rappresentano un ambiente interattivo chiamato Notebook Colab che permette di scrivere ed eseguire codice scritto in Python direttamente nel Browser e consente un accesso gratuito ma temporalmente limitato alle GPU e TPU dei server Google.
Inoltre, Colab supporta molteplici librerie di machine learning popolari e facilmente caricabili nei Notebook.
I Notebook Colab una volta creati vengono memorizzati direttamente su Google Drive, inoltre sono file facilmente condivisibili con diversi livelli di accesso, ad esempio la sola visualizzazione del lavoro, la possibilità di commentare o la possibilità di modificare direttamente il documento.
In più, è possibile importare i dati da Google Drive o Github, inclusi fogli di calcolo e altri tipi di dati.
Per i vantaggi che offre, Google Collaboratory è stato l'unico ambiente di lavoro che abbiamo utilizzato.
Gli addestramenti delle reti sono stati fatto sfruttando la GPU di Google, caricando preventivamente il dataset delle immagini su Google Drive.

3.2 Jupyter

Jupyter [3], chiamato così dai tre linguaggi di programmazione iniziali Julia, Python e R, un'applicazione fondata nel 2015 basata sul modello client-server e creata dell'organizzazione no profit Progetto Jupyter.
Presenta un software gratuito, standard aperti e servizi web per l'elaborazione interattiva in tutti i linguaggi di programmazione.
I due componenti centrali di Notebook Jupyter sono un set di diversi kernel (interpreti) e la dashboard.
I kernel sono piccoli programmi che elaborano richieste ("request") specifiche nel linguaggio e reagiscono con relative risposte.
Un kernel standard è IPython, un interprete da riga di comando che permette di lavorare con Python.
La dashboard serve da una parte come interfaccia di gestione per i singoli kernel, mentre dall'altra come strumento per la creazione di nuovi documenti Notebook o per aprire progetti già esistenti.
In una sola istanza gli utenti possono scrivere, documentare ed eseguire codici, visualizzare dati, eseguire calcoli ed esaminare i risultati corrispondenti.
In particolare, durante la fase di prototipo possono trarre beneficio dal fatto che ciascun codice può essere ospitato in celle indipendenti, permettendo così di testare individualmente specifici blocchi di codici.
Uno degli scopi d'utilizzo più importanti è la creazione e addestramento di modelli di machine learning.
Nella nostra esperienza abbiamo utilizzato Jupyter Notebook come ambiente locale in cui utilizzare le nostre GPU per addestrare le reti che abbiamo individuato per il raggiungimento dell'obiettivo.

3.3 Tensorflow & Keras

TensorFlow [4] è una piattaforma open source end-to-end per l'apprendimento automatico.
Dispone di un ecosistema completo e flessibile di strumenti, librerie e risorse della comunità che consente agli sviluppatori di creare e distribuire facilmente applicazioni basate su ML.
Si ha la possibilità di creare in maniera semplice modelli di machine learning utilizzando API intuitive di alto livello come Keras.
Tensorflow è compatibile con i principali sistemi operativi a 64 bit come Windows, Linux, Mac OS X e Android.
Inoltre, la libreria è in grado di funzionare su numerose tipologie di CPU e GPU, grazie al supporto di diverse API come CUDA, OpenCL e DirectML.
Nella nostra esperienza abbiamo lavorato con Tensorflow sul sistema operativo Windows ed eseguendo l'addestramento su GPU di differenti marche, rispettivamente NVIDIA e AMD, abbiamo constatato che la piattaforma Tensorflow è in grado di funzionare con entrambe utilizzando API CUDA per quanto riguarda NVIDIA e DirectML per quanto riguarda AMD.
Keras è una libreria open source scritta in Python che permette l'implementazione rapida di reti neurali.
Come è stato citato in precedenza, esso non funge da framework, ma è un application program interface per l'accesso e la programmazione a diversi framework di apprendimento automatico.
Per questo progetto abbiamo importato diverse librerie essenziali di keras per la costruzione della rete neurale come:

- tensorflow.keras.preprocessing.image, da cui abbiamo importato ImageDataGenerator, utile per la data augmentation del dataset come citato nel capitolo precedente;
- tensorflow.keras.models, da cui abbiamo importato "Sequential" per implementare un modello strutturato a livelli per la rete neurale convoluzionale;
- tensorflow.keras.layers, importando tutte le classi che costituiscono il modello della rete neurale come Input, Convolution2D, MaxPooling, Flatten, Dropout, Dense e altre classi;
- tensorflow.keras.optimizers, grazie alla quale abbiamo importato gli ottimizzatori utilizzati durante i vari esperimenti come, ad esempio, Adam;
- tensorflow.keras.callbacks: libreria che riguarda i diversi callbacks, nella nostra esperienza abbiamo utilizzato EarlyStopping.

Oltre a Tensorflow e Keras abbiamo utilizzato altre librerie come pyplot, pandas e numpy.

3.4 OpenCV & CVZone

OpenCV [5], abbreviazione di *Open Source Computer Vision*, è una libreria open-source scritta inizialmente in C++ e successivamente estesa ad altri linguaggi che comprende numerosi algoritmi di computer vision.
Per permette la lettura e scrittura di una sequenza d'immagini, OpenCV predispone la classe VideoCapture.
Sono presenti diversi overloading del costruttore della classe, ad esempio *cv2.VideoCapture(int index)* per acquisire uno stream video dal dispositivo identificato dall'index specificato come primo argomento.
In alternativa, si può usare *cv2.VideoCapture(const String)* per operare su tutti i frame di un filmato, specificando come primo argomento il percorso del sistema operativo in cui si trova il file.
Un'altra funzione importante che verrà utilizzata in seguito è *cv2.set(int propId, double value)*, la quale assegna il valore contenuto in *value* alla proprietà associata all'index contenuta in *propId*.
Il valore di ritorno della funzione è "True" se la proprietà è supportata dall'istanza fornita da VideoCapture.
Tra le proprietà su cui si può agire troviamo l'altezza e la larghezza, rispettivamente con propId=3 e prodId=4 e i cui valori di default sono 640 e 480, oppure i frame per secondo (FPS) corrispondenti a propId=5.
Un'altra funzione usata all'interno di questo progetto è *cv2.read()*, la quale permette di decodificare e ottenere un frame dallo stream video, restituendo "False" nel caso in cui l'acquisizione fallisca.
Per semplificare l'uso degli strumenti offerti da OpenCV è stata sviluppata *CVZone* [6], una libreria che predispone numerose funzioni per effettuare operazioni di detection e image processing.
Quest'ultima libreria si basa anche su un secondo modulo open-source sviluppato da Google chiamato *mediapipe* [7], il cui obbiettivo è fornire strumenti per effettuare operazioni di computer vision.
La classe più importante su cui è basato l'intero progetto è HandDetector, la quale è contenuta in CVZone e il cui costruttore *HandDetector(int maxHands, double detectionCon)* prevede due argomenti:

- *maxHands* è il numero massimo di mani identificabili all'interno di un'immagine
- *detectionCon* è il valore di confidenza durante il tracking della mano

Tra i metodi invocabili sull'istanza fornita dal costruttore troviamo *findposition(img, Bool draw)*, il cui primo argomento è l'immagine acquisita tramite la webcam, tipicamente un array numpy, mentre il secondo è un booleano che se posto a True permette di disegnare un riquadro intorno alla mano.
Tale funzione restituisce una tupla di quattro valori corrispondenti ai vertici della "bounding box" che racchiude la mano e un array di coordinate (x, y) corrispondenti ai 21 punti visibili in figura 1.

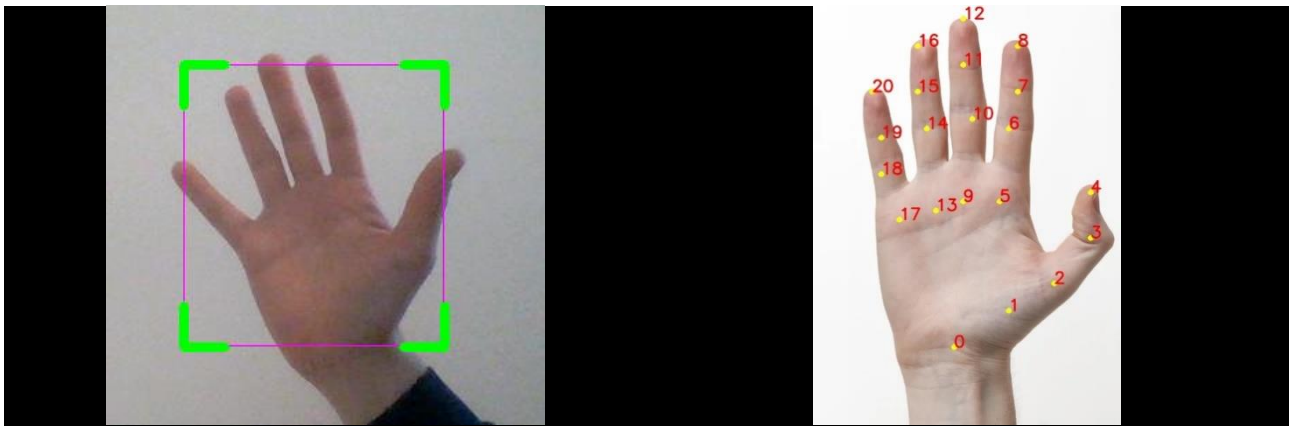

***Figura 2**. Bounding box e punti di riferimento restituiti dalla funzione findpostion*

Per implementare tutte le funzionalità di questo progetto sono state utilizzate le versioni dei moduli OpenCV 4.5.5, CVZone 1.4.1 e mediapipe 0.8.8.

4. Implementazione delle funzionalità

In questo capitolo verrà discusso brevemente la creazione di un dataset contenente delle immagini di gesti di mani provenienti dal gioco conosciuto comunemente come “sasso, carta forbice”.
Successivamente, verranno illustrate tutte le metodologie adottate per la creazione delle diverse architetture di rete neurali convoluzionali e i rispettivi risultati ottenuti al termine dell’addestramento.
Segue poi la spiegazione di come il funzionamento delle reti neurali sia stato esteso ad un contesto “real time”, cioè come queste, dopo averle collegate ad una webcam, siano in grado di classificare in tempo reale i gesti di una mano.

4.1 Creazione del dataset

Affinché una rete sia in grado di apprendere correttamente come classificare i gesti della morra cinese, è necessario predisporre un dataset contenente numerose immagini raffiguranti le mani
A tal proposito, tensorflow mette a disposizione il dataset chiamato “rock_paper_scissors”, contenente 2925 immagini di mani in CGI su sfondo bianco, di cui 2520 di training, 372 di test e 33 per la validazione.
Di seguito, sono presenti alcune immagini appartenenti al dataset:

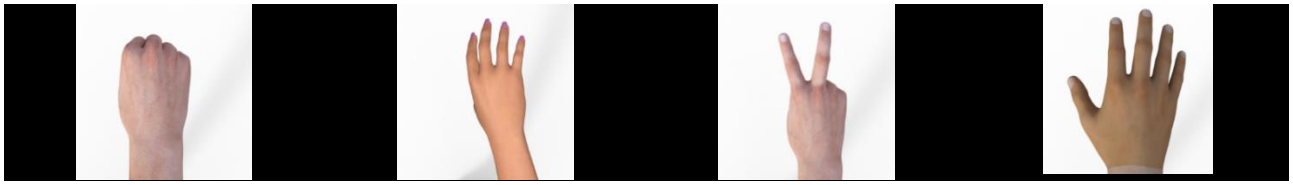

***Figura 3**. Immagini provenienti dal dataset 1*

A causa della presenza del medesimo sfondo per ciascuna immagine, tale dataset non è adatto per applicazioni di “image recognition” in cui la mano abbia uno sfondo diverso da quello bianco.
Infatti, affinché la rete sviluppi una maggiore capacità di generalizzazione, è necessario creare un dataset le cui immagini abbiano sfondi diversi tra loro.
Per sopperire a questo problema, il primo approccio è stato la creazione di un dataset più ampio rispetto a quello precedente, con cinque sfondi differenti per ciascuna classe:

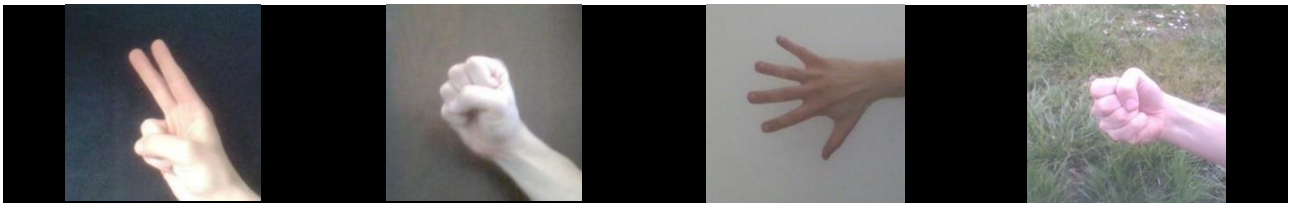

***Figura 4**. Immagini provenienti dal dataset 2*

Delle 4500 immagini totali, 3150 sono state riservate per il training, 900 per il test e 450 per la validazione.
Tuttavia, l’addestramento su questo dataset non ha portato ad un buon livello di generalizzazione da parte della rete, questo a causa del limitato numero di sfondi utilizzati durante la creazione del dataset.
Infatti, queste non performavano bene con sfondi diversi da quelli su cui avevano appreso.
Per risolvere tale problema, è stato costruito un dataset con uno sfondo diverso per ciascuna immagine, questo per far apprende alla rete i gesti delle mani indipendentemente dal contesto in cui si trovassero.
In particolare, è stato creato un nuovo dataset più ampio rispetto a quello precedente, con un totale di 30684 immagini, di cui 24444 di training, 4992 per il test e 1248 di validazione.
Inizialmente, attraverso uno script python [8] sono state collezionate per ciascuna classe 10228 immagini.
Successivamente sono stati predisposte 20 immagini di sfondo, ciascuna di dimensione 3840x2160, da cui sono state campionate casualmente delle porzioni di dimensioni 300x300 da sostituire allo sfondo nero.
Quest’ultima operazione è stata eseguita mediante un secondo script python, il cui codice è il seguente:

```
while True:
    if count == num_samples:
        break
    source_path = os.path.join(IMG_SOURCE_PATH, '{}.jpg'.format(count + 1))
    bg_path = os.path.join(BG_PATH, '{}.jpg'.format(random.randint(1, 10))
    save_path = os.path.join(IMG_SAVE_PATH, '{}.jpg'.format(count + 1))
    bg = Image.open(bg_path)
    frame = cv2.imread(source_path)
    x_coord = random.randint(0, 3840-300)
    y_coord = random.randint(0, 2160-300)
    rect = (x_coord, y_coord, x_coord + 300, y_coord + 300)
    bg_array = np.asarray(bg.crop(rect))
    hand_pixels = np.where(
            (frame[:, :, 0] > 7) &
            (frame[:, :, 1] > 7) &
            (frame[:, :, 2] > 7)
        )
    bg_array[hand_pixels] = frame[hand_pixels]
    cv2.imwrite(save_path, bg_array)
    count++
```

Innanzitutto, l'algoritmo utilizza un ciclo while con cui iterare su tutte le immagini di una certa classe.
Per fare ciò, si predispone un contatore count per determinare l'immagine corrente su cui si sta operando.
Successivamente, mediante l'uso delle funzioni predisposte dalla libreria "os", vengono individuati i percorsi all'interno del proprio sistema operativo dell'immagine della mano, dello sfondo e della cartella in cui si vuole salvare l'immagine finale, memorizzati rispettivamente in source_path, bg_path e save_path.
Si osserva che lo sfondo è scelto casualmente ad ogni iterazione da un pool di venti background in bg_path.
Una volta aperte entrambe le immagini, viene individuato un rettangolo di 300x300, cioè delle stesse dimensioni di ciascuna immagine del dataset, con coordinate x_coord e y_coord casuali e il quale definisce la porzione d'immagine a cui sovrapporre il gesto della mano del frame corrente.
Segue poi l'individuazione dei pixel appartenenti alla mano all'interno dell'immagine attuale, controllando che il valore RGB del singolo pixel sia maggiore del colore definito dalla terna (7,7,7), cioè del colore nero.
Infine, tutti i pixel memorizzati in hand_pixel vengono utilizzati in un'operazione di assegnamento per sovrapporre la mano del frame corrente sulla porzione di sfondo estratta precedentemente.

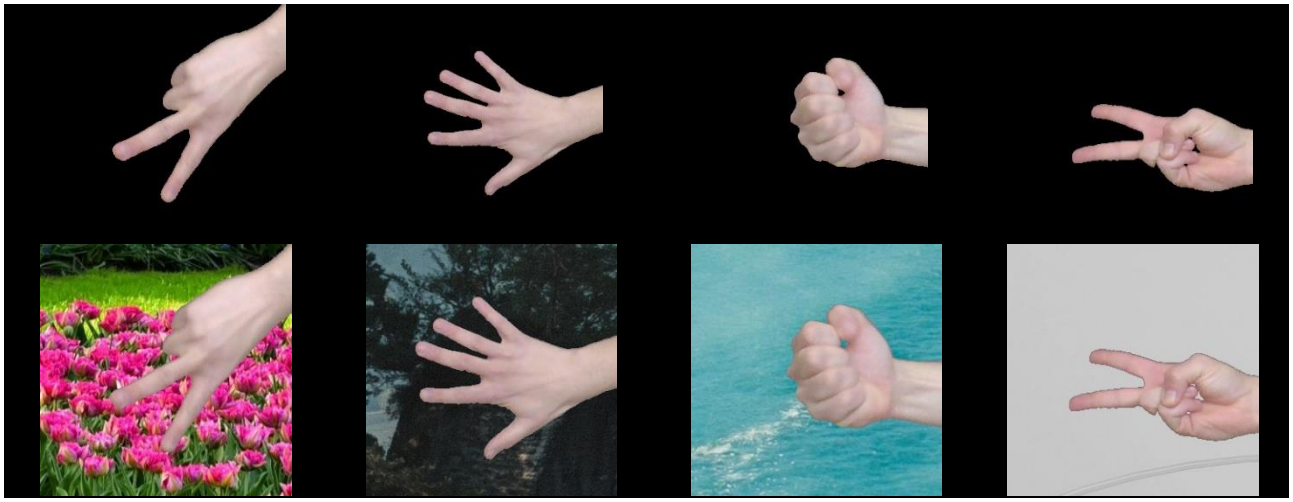

***Figura 5**. Immagini provenienti dal dataset 3*

4.2 Modello creato da zero

L'approccio più veloce per implementare un classificatore che riconosca i gesti delle mani del gioco della morra cinese è la creazione di una rete neurale convoluzionale addestrata sul dataset sul quale si vuole che apprenda, nel nostro caso quello introdotto nel capitolo 4.1.
Per semplificare l'apprendimento della rete, è stato scelto di utilizzare una dimensione dell'input minore rispetto a quelle delle immagini contenute nel dataset, precisamente 48x48x1 invece che 300x300x3.

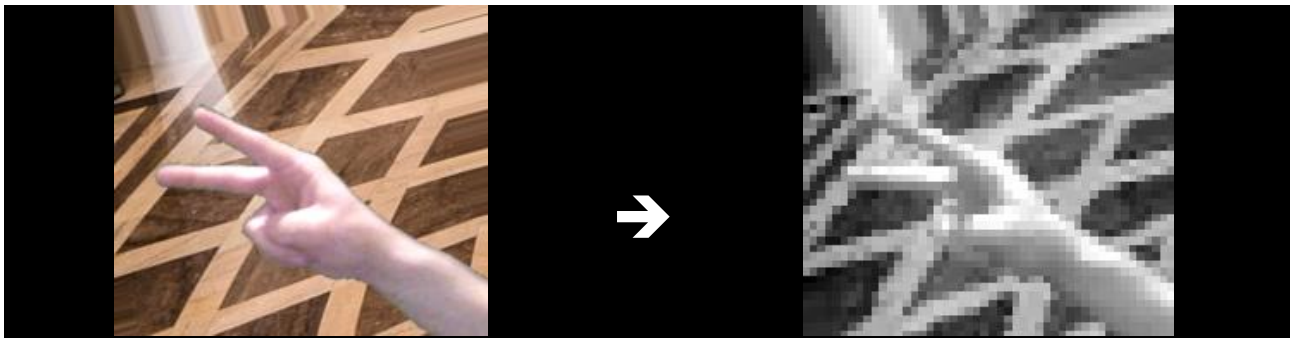

***Figura 6**. Confronto immagini 300x300x3 e 48x48x1*

Intuitivamente, la semplificazione del training risiede nel fatto che l'estrazione delle feature da immagini più piccole è più semplice rispetto a quella da immagini più grandi, avendo meno pixel su cui operare. Inoltre, poiché ciascuna immagine in scala di grigi possiede un solo canale, cioè ogni pixel è individuati da un solo valore, avremo immagini meno complesse da gestire rispetto ad immagini RGB con tre canali.
La struttura completa della rete neurale è la seguente:

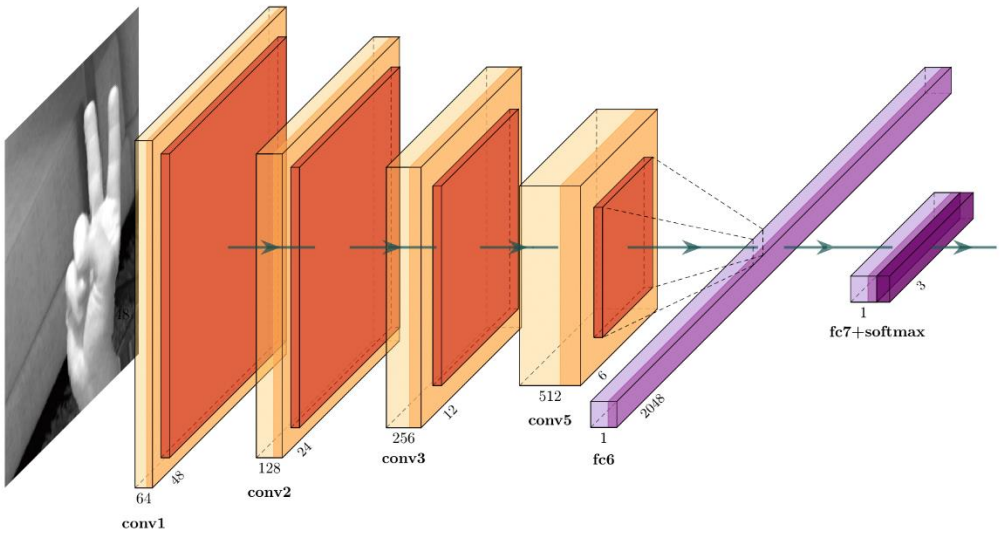

***Figura 7**. Struttura rete creata da zero*

In riferimento alla figura precedente, si può osservare che nel primo strato convoluzionale sono stati predisposti 64 kernel, ciascuno di dimensioni 3x3 e stride 2,
Segue poi un'operazione di max pooling di dimensione 2x2 e stride 2 con il quale si dimezza la dimensione dell'immagine proveniente dallo strato di input, passando da una dimensione di 48x48x64 a 24x24x64.
Successivamente, si procede con un secondo strato convoluzionale, questa volta con 128 filtri e aventi le medesime dimensioni dello strato convoluzionale di input.
A questo punto si esegue un'altra operazione di pooling analoga a quella di prima, dimezzando la dimensione proveniente dal secondo strato ed ottenendo una nuova immagine di dimensione 12x12x128.

Tale procedimento viene iterato altre due volte, predisponendo due strati convoluzionali da 256 e 512 kernel rispettivamente, seguiti entrambi da due operazioni di pooling equivalenti a quelle viste prima.
Per evitare che le operazioni di convoluzioni portino ad una perdita di informazioni, nello specifico la rimozione dei pixel appartenenti ai bordi dell'immagine, è stato impostata la proprietà padding a "same".
Una volta aggiunta la dimensione di 3x3x512, l'immagine convoluta subisce un'operazione di "flattening", cioè viene trasformata in un tensore monodimensionale costituito da 4608 neuroni.
Quest'ultima viene posta in ingresso ad uno strato fully connected da 2048 neuroni, il quale si occuperà di combinare le feature provenienti dagli strati precedenti.
Infine, viene aggiunto lo strato di output, anch'esso fully connected, di dimensione pari al numero delle classi.
Si sottolinea che per mitigare il rischio che la rete entrasse in overfitting, ogni strato della rete è seguito da un'operazione di dropout, precisamente 0.2 per gli strati convolutivi e 0.4 per lo strato denso.

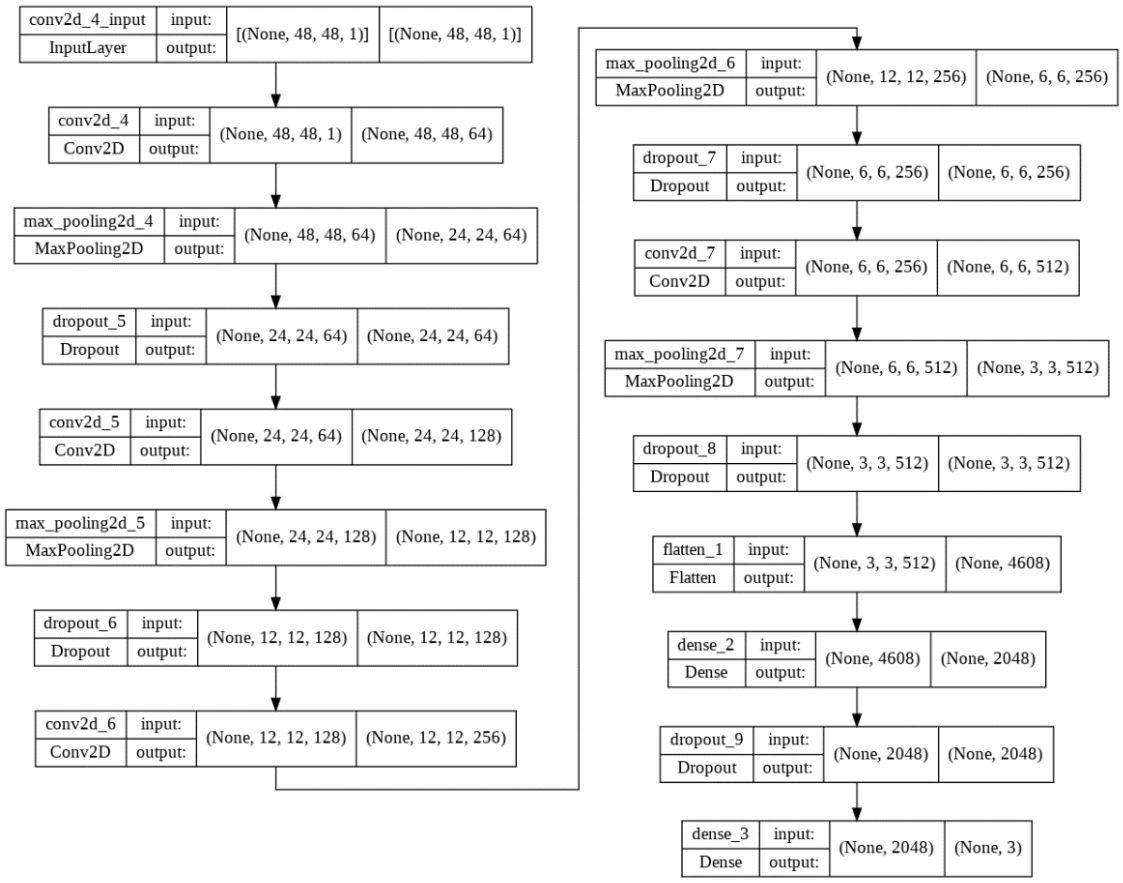

***Figura 8**. Struttura a blocchi della rete creata da zero*

4.3 Transfer Learning con VGG16

Una delle metodologie più popolari nell'ambito del deep learning è il transfer learning [2], una tecnica che prevede l'uso di una rete già addestrata per svolgere un certo compito su un dataset su cui non ha appresso.
In particolare, il transfer learning prevede la sostituzione degli strati di output, cioè quelli che si occupano di combinare le feature provenienti dagli strati interni ed effettuare la classificazione, con nuovi strati densi.
Mantenendo inalterati i pesi degli strati precedenti ed addestrando solamente i nuovi strati, è possibile ottenere degli ottimi risultati senza la necessità di addestrare un'intera rete da zero.
Un'estensione della tecnica precedente è il fine tuning, in cui non mantengo inalterato il valore di tutti i pesi della rete pre-addestrata, ma solamente una parte, questo per migliorare l'estrazione delle feature dai dati.

Per questo progetto è stato scelto di utilizzare come base il modello pre-addestratto VGG16 [3], una rete neurale costituita da 16 strati, 13 convoluzionali e 3 fully-connected, intervallati da alcune operazioni di pooling per ridurre progressivamente la dimensione dell'immagine, ma senza perdere informazioni.
La rete possiede 138.4 milioni di pesi, un'accuracy del 90.1% e un tempo d'inferenza su CPU di 69.5 ms.
Per utilizzare VGG16, keras mette a disposizione l'omonima classe e il cui costruttore è il seguente:

```
keras.applications.VGG16(weights, include_top, input_shape)
```

Il primo argomento è weights e il quale definisce il set di pesi utilizzati quando la rete verrà caricata.
Di solito si scelgono i pesi di "imagenet", siccome sono quelli associati al dataset su cui ha appreso VGG16.
Successivamente è presente include_top, un booleano con il quale indicare se al momento del caricamento è necessario includere nella struttura della rete anche gli strati densi di output.
Qualora si scegliesse di porre include_top = False, sarà necessario aggiungere gli strati densi di output.
Prima di fare questo, tuttavia, è necessario inibire l'aggiornamento dei pesi degli strati intermedi come segue:

```
for layer in preTrainedNet.layers:
    layer.trainable = False
```

Intuitivamente, iterando tramite un ciclo for su ciascun layer dell'istanza di VGG16 preTrainedNet, pongo il booleano che determina se esso possa essere addestrato a False.
L'ultimo argomento del costruttore è input_shape ed è quello che stabilisce la dimensione dell'input.
Si sottolinea che sebbene VGG16 sia stata addestrata su immagini RGB 224x224, quindi aventi 3 canali, è possibile definire le dimensioni e il numero di canali di input diversi in base al dataset con cui si sta lavorando.
Per questo motivo e per le considerazioni fatte nel capitolo precedente, l'input è stato fissato a 300x300x3.
La struttura completa della rete neurale utilizzata per questo progetto è la seguente:

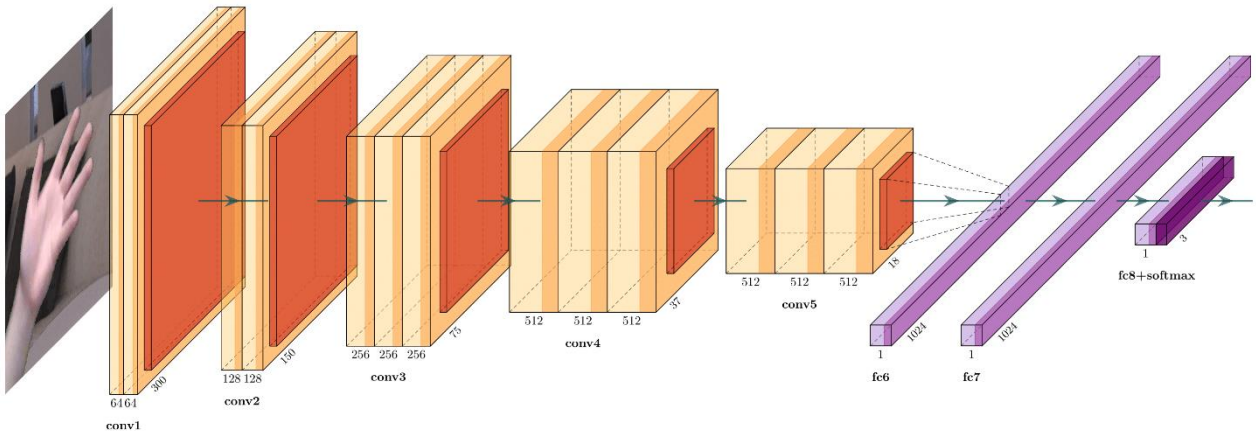

***Figura 9**. Struttura rete VGG16 con transfer learning*

La rete prevede come input due strati convoluzionali con 64 kernel, ciascuno con dimensione 3x3 e stride 2.
Segue poi un'operazione di max pooling di dimensione 2x2 e stride 2, quindi delle stesse dimensioni e stride dei kernel, e con il quale viene dimezzata la dimensione dell'immagine provenienti dagli strati convolutivi, passando da una dimensione di input di 300x300x64 a 150x150x64.
A questo punto vengono predisposti altri due strati convoluzionali, questa volta con 128 filtri ciascuno.
Dopodiché si ha un'altra operazione di pooling analoga a quella di prima e con la quale si va a dimezzare la dimensione dell'immagine proveniente degli strati convolutivi, ottenendo un'immagine 75x75x128.
Seguono poi tre stati convoluzionali adiacenti con 256 filtri 3x3 con stride 2 e un'operazione di pooling.
Si osserva che la struttura costituta da tre strati convoluzionali con l'operazione di pooling finale viene riproposta anche per i due blocchi successivi, questa volta con 512 kernel 3x3 e stride 2 per ciascuno strato.

A questo punto l'immagine convoluta di dimensione 9x9x512 subisce un'operazione di "flattening", cioè viene trasformata in un tensore monodimensionale costituito da 41472 neuroni.
Successivamente troviamo una coppia di strati fully-connected di dimensione 1x1024 che vanno a sostituire la coppia di strati originali di VGG16 di dimensione 1x4096.
Infine, viene aggiunto un terzo strato denso di dimensione pari al numero di classi, in questo caso tre.
Poiché siamo in presenza di classificazione multiclasse, i valori forniti dai neuroni di questo strato mapperanno una distribuzione di probabilità, cioè la loro somma sarà pari a 1.
Per determinare la dimensione ottimale degli strati fully-connected sono state create più strutture della rete e in maniera empirica, mediante diversi esperimenti, è stata scelta quella che ha fornito una migliore risposta.
Come nel caso della rete creata da zero, al fine di ridurre la probabilità che la rete entrasse in overfitting, è stata utilizzata la tecnica del dropout con valore 0.4 sugli strati densi di output.
Gli esperimenti condotti su questa rete verranno illustrati successivamente nel capitolo 5.

4.4 Classificazione Real-Time

Una volta verificato che la rete classifichi correttamente le immagini di input contenenti i gesti delle mani del gioco della "morra cinese", è possibile estendere il suo funzionamento alla classificazione "real time".
Lo sviluppo di tale funzionalità è basato sui moduli OpenCV e CVZone, entrambi discussi nel capitolo 4.
Attraverso le funzioni predisposte da quest'ultimi è stato possibile effettuare inizialmente un'operazione detection, ottenendo la porzione d'immagine dallo stream video contenente la mano dell'utente, e successivamente, dopo una fase di pre-elaborazione dell'immagine, la classificazione da parte della rete.
Il codice che implementa ciò che è stato appena descritto è il seguente:

```
import cv2
from cvzone.HandTrackingModule import HandDetector

cap = cv2.VideoCapture(0)
cap.set(3, 640)
cap.set(4, 480)
detector = HandDetector(detectionCon = 0.8, maxHands = 1)
flag = True
confVal = 40

while True:
    success, img = cap.read()
    img = detector.findHands(img, draw=False)
    lmList, bbox = detector.findPosition(img, draw=False)
    if bbox:
        x,y,w,h = bbox["bbox"]
        x = x - (confVal * 2); y = y - confVal
        h = h + (confVal * 3); w = w + (confVal * 3)
        if(x >= 0 and y >= 0 and (x+w) < 640 and (y+h) < 480):
            input_image = img[y:y+h, x:x+w]
            bbox["bbox"] = x,y,w,h
            if flag:
                input_image = cv2.resize(input_image, dsize=target_size)
                input_image = np.expand_dims(input_image, axis=0)
                cvzone.cornerRect(img, bbox["bbox"])
                output = newModel.predict(input_image)
                output = np.argmax(output, axis=1)
            flag = not flag
                 cv2.putText(img, classNames[output[0]], (15, 30),
                 cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 0), thickness=4)
    cv2.imshow("Image", img)
```

Dopo aver importato le librerie cv2, cioè quella corrispondente a openCV, e il modulo HandDetector dalla libreria di CVZone come riportato sopra, si procede con la creazione dell'istanza cap della classe VideoCapture con la quale acquisire il video dalla webcam del proprio computer.
Segue poi la definizione della risoluzione del video tramite la funzione *cap.set*, in questo caso 640x480.
Dopodiché creo detector come istanza di HandDetector e con la quale effettuare il tracking della mano, ponendo 1 come il numero massimo di mani a cui applicare tale operazione e 0.8 come valore di confidenza.
Successivamente viene predisposta una variabile booleana flag per poter ottimizzare le predizioni della rete.
Per comprendere meglio come questa ottimizzazione funzioni, è necessario porre l'attenzione sul fatto che senza alcun controllo nel codice, la rete effettuerà una predizione ad ogni frame.
Chiaramente, questo comporta una grande mole di calcoli, portando ad un rallentamento dell'applicazione.
Poiché non è necessario che la rete operi su ogni singolo frame, viene utilizzata la variabile flag come se fosse un interruttore, abilitando la predizione della rete a frame alterni.
Segue che se la variabile flag è posta a True, allora la rete effettuerà la predizione sull'immagine di input.
Viceversa, se flag contiene il valore False, la predizione della rete verrà inibita fino al frame successivo.
Infine, predispongo un valore di confidenza *confVal* posto a 40 e il cui obbiettivo verrà illustrato in seguito.
Una volta istanziate tutte le variabili, si procede con l'esecuzione di un ciclo while nel quale avviene l'acquisizione delle immagini dalla webcam richiamando *read()* sull'oggetto cap.
Quest'ultimo restituirà il booleano *success*, il quale indica se l'acquisizione è avvenuta con successo, e l'immagine ottenuta come array della classe numpy.
Segue poi l'invocazione del metodo *findposition()* sull'istanza detector con cui si ottengono, se la mano è presente nell'immagine, le coordinate della bounding box e i punti di riferimento illustrati nel capitolo 3.4.
Dopo aver controllato se bbox contiene almeno un elemento, vengono memorizzate nelle variabili x, y, w e h le coordinate dell'angolo in alto a sinistra, la larghezza e l'altezza del riquadro individuato da bbox.
Dopodiché si procede con la modifica dei loro valori utilizzando il parametro confVal introdotto inizialmente.
In particolare, ciò che viene fatto è sottrarre il valore di confVal alle variabili x e y come riportato nel codice, in modo tale che l'angolo della bounding box venga spostato in alto e a sinistra.
Parallelamente, i valori di w e h, che si ricordano essere rispettivamente la larghezza e l'altezza della bounding box, vengono incrementati addizionandogli la variabile confVal come riportato in alto.
Le operazioni svolte permettono di incrementare l'area del rettangolo individuata da bbox e, di conseguenza, ottenere una porzione d'immagine più grande rispetto a quella fornita originariamente da findposition().
Intuitivamente, ciò viene fatto per includere tutti i pixel della mano e avere una predizione più accurata.

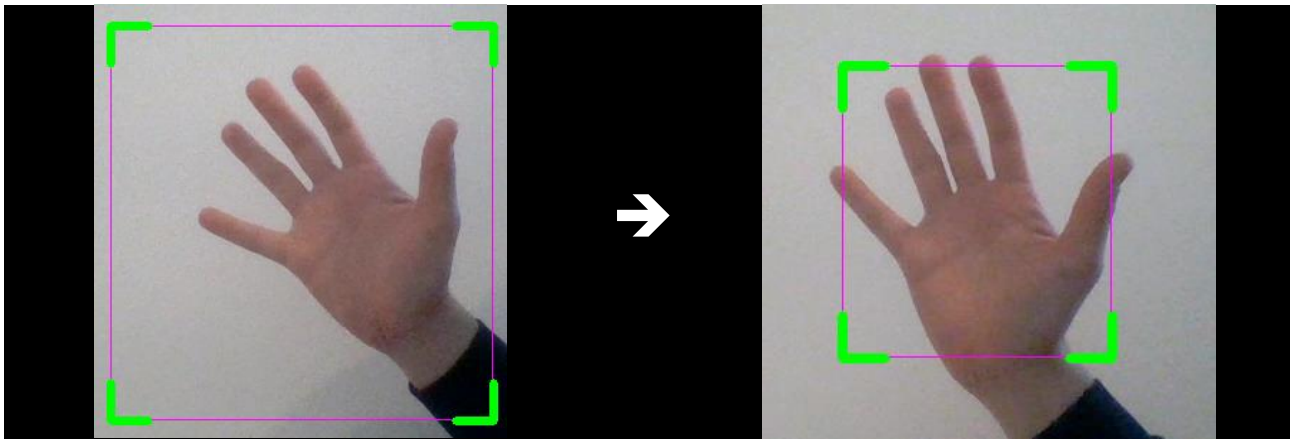

***Figura 10**. Confronto dimensioni della bounding box con e senza parametro di confidenza*

Prima di estrarre ed operare sulla sola porzione d'immagine individuata da bbox, è necessario controllare che l'espansione dell'area del rettangolo non abbia incluso porzioni esterne all'immagine, questo assicurandosi che i valori delle variabili x e y non siano negativi e contemporaneamente che w ed h non siano maggiori delle dimensioni con cui è stata inizializzata la risoluzione della webcam.
Se tutte le condizioni sono verificate si prosegue estraendo l'immagine individuata da bbox mediante un'operazione slicing, specificando le varie coordinate come riportato in alto.

Una volta fatto ciò posso aggiornare i valori della bbox con quelli calcolati precedentemente.
Dopo aver verificato che il valore di flag sia True, cioè che la predizione da parte della rete sul frame corrente sia abilitata, inizia una fase pre-processing in cui si ridimensiona l'immagine di input a 300x300, ovvero come quelle del dataset su cui ha appreso la rete, ed infine la si converte in un array di numpy.
Avendo a disposizione i nuovi valori della bbox, attraverso la funzione *cornerRect()* dalla libreria CVZone posso stampare a video il bounding box contenente la mano.
Questa operazione è stata fatta esternamente a *findPosition()*, perché l'esecuzione di quest'ultima funzione è posta prima che i nuovi valori di bbox vengano calcolati e aggiornati.
Terminata la fase di pre-elaborazione, si procede con la predizione del modello con il metodo *predict*(), il cui l'unico argomento è l'array image_input costruito poco fa.
La variabile output conterrà un array di lunghezza pari al numero di classi, in questo caso tre, e i cui valori definiscono una distribuzione di probabilità.
Poiché siamo interessati alla classe predetta dal modello, cioè l'indice dell'array a cui corrisponde il valore più alto, per ottenerla si sfrutta il metodo *argmax()* della classe numpy sull'array di output.
Ad esempio, supponendo che l'array di output fornito dalla classificazione sia [0.2, 0.1, 0.7], la funzione "np.argmax(output)" fornirà il valore 2, perché nella posizione 2 dell'array è contenuto il valore più alto.
In altri termini, 2 è la classe predetta dal modello a cui appartiene con più probabilità l'immagine di input.
In seguito all'ottenimento della predizione, lo stato della variabile flag viene negato tramite l'operatore not, questo per abilitare o inibire la predizione della rete al frame successivo.
Poiché si ha a disposizione l'index della classe predetta, è possibile prelevare dall'array di stringhe classNames il nome della classe e stamparla successivamente a video mediante *putText()*.
Tale funzione è predisposta da OpenCV e, come suggerisce il nome, aggiunge un testo ad un'immagine.
Tra gli argomenti previsti da *putText()* vi sono l'immagine sorgente da modificare, in questo caso il frame corrente img, la stringa da aggiungere, il font del testo, il colore del testo e le coordinate in cui posizionarlo.
Una volta apportate le modifiche necessarie, il frame è visualizzato in output attraverso *imshow()*, una funzione anch'essa resa disponibile da OpenCV e la quale prevede come argomenti l'immagine da stampare a video e il nome finestra che verrà visualizzata.

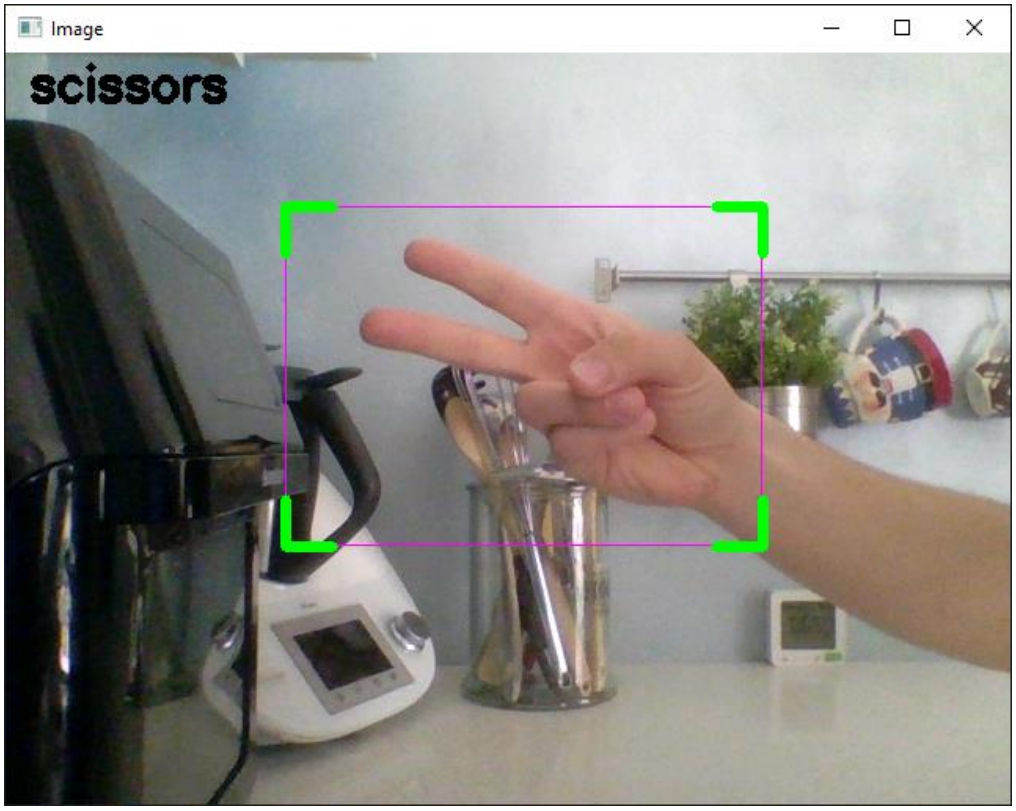

Figura 11. *Predizione real-time*

5. Esperimenti

In questo capitolo, dopo una breve discussione riguardo gli obbiettivi relativi al progetto, verranno illustrati gli esperimenti effettuati durante lo sviluppo delle reti convoluzionali, descrivendo i miglioramenti apportati progressivamente che hanno portato all'individuazione delle architetture finali.
Segue poi la discussione dei risultati ottenuti in seguito all'addestramento delle reti, ponendo particolare attenzione ai grafici relativi alla variazione dei valori di loss e accuracy al progredire delle epoche.
Infine, dopo un confronto tra le architetture, verrà stabilito l'approccio cha ha fornito risultati migliori.

5.1 Obbiettivi

L'obbiettivo di questo progetto è la realizzazione di una rete neurale convoluzionale che classifichi con un certo grado di accuratezza delle immagini al cui interno sono presenti i gesti del gioco della morra cinese. Inoltre, attraverso l'uso della webcam del proprio dispositivo, la rete deve essere in grado di classificare in tempo reale i gesti della mano.

5.2 Esperimenti sulle reti convoluzioni

Il primo tentativo per l'implementazione di un classificatore che riconoscesse i gesti del gioco "sacco, carta, forbice" è stato la creazione di una rete neurale con quattro strati convoluzionali, i primi due con 16 kernel, poi due strati da 32, seguiti infine da due strati fully-connected con 64 e 3 neuroni rispettivamente, dove 3 è il numero di classi che vogliamo che la rete predica.
La rete è stata addestrata sul dataset "rps" predisposto da tensorflow, cioè quello avente immagini con lo sfondo bianco, normalizzando ed effettuando data augmentation su ciascuna immagine fornita in input.
La data augmentation, che si ricorda essere utilizzata per rendere la rete maggiormente invariante alla posizione dei pixel nell'immagine, prevede una rotazione dell'immagine di un angolo che varia da -60° a 60°, una modifica casuale della luminosità dell'immagine pari al 20%, uno spostamento dell'immagine orizzontale e verticale pari al 10% della dimensione dell'immagine, in questo caso 300x300x3, e un ribaltamento orizzontale effettuato con una probabilità del 50%.
Questo primo tentativo è stato fallimentare, con una rete risultante che operava in regime di funzionamento underfitting e con una predizione delle immagini caotica e imprecisa.
Per ovviare a questo problema è stata costruita una seconda rete con un numero di filtri e di neuroni pari al doppio di quella precedente, precisamente 32 filtri per i primi due strati, 64 kernel per i due successivi, uno strato denso da 128 neuroni ed infine un altro strato denso da 3 che mappasse le classi da predire.
Questo nuovo tentativo ha portato ha ottimi risultati in presenza di immagini con sfondi bianchi, ma pessime prestazioni con mani con sfondi differenti.
Per far sì che la rete generalizzasse meglio e gestisse immagini di mani con sfondi diversi da quello bianco, è stata addestrata la medesima rete sul dataset illustrato nel capitolo 4.1, nello specifico quello avente cinque sfondi per ciascuna classe.
Sebbene i risultati ottenuti in seguito al training fossero buoni, con un'accuracy nell'ordine di 0.94 e loss di 0.03, la rete non aveva sviluppato una buona capacità di gestione dei background, questo a causa dell'esigua quantità di sfondi presenti nel dataset.
Per sopperire a questo problema è stato scelto di ampliare e complicare ulteriormente il dataset come spiegato nel capitolo 4.1, utilizzando uno sfondo diverso per ciascun esempio del dataset.
Tuttavia, mantenendo la stessa architettura di prima, i risultati ottenuti suggerivano una pessima generalizzazione da parte della rete causata da una limitata capacità della stessa.
La soluzione a quest'ultimo problema prevede l'uso di una nuova struttura costituta anch'essa da quattro strati convoluzioni e con un numero di filtri crescente, nello specifico 64, 128, 256 e 512, mentre i due strati densi d'uscita hanno una dimensione di 2048 e 3 neuroni rispettivamente.
Quest'ultima rete convoluzionale ha fornito un'ottima risposta in uscita sia su predizione statica, predicendo correttamente i gesti delle mani diversi contesti, posizioni e livelli di luminosità, sia in tempo reale.

Per avere un confronto significativo con quest'ultima architettura, è stato scelto di implementare una nuova rete convoluzionale basata sulla rete pre-addestrata VGG16 e a cui è stato applicato la tecnica del transfer learning come riportato nel capitolo 4.3.
Quest'ultima è stata addestra parallelamente ai tentativi descritti precedentemente e su ciascun dataset.
La dimensione ottimale degli strati densi d'uscita è stata individuata in maniera empirica, partendo da 512 neuroni per ciascuno strato e raddoppiando la loro dimensione fino a 2048.
In questo caso, la rete che ha fornito i risultati migliori è quella con la coppia di strati densi da 1024 neuroni.
Ciascun addestramento di VGG16 ha portato ad ottimi risultati, questo perché era una rete già addestrata e aveva già appresso le feature principali con cui effettuare la classificazione delle immagini.

5.3 Risultati

Di seguito sono riportati i risultati ottenuti in seguito all'addestramento della rete neurale convoluzionale creata da hoc e di VGG16, dove a ciascuna architettura è associata una coppia di grafici in cui si mettono in relazione gli andamenti dei valori di loss e di accuracy relativi al set di training e al set di validation:

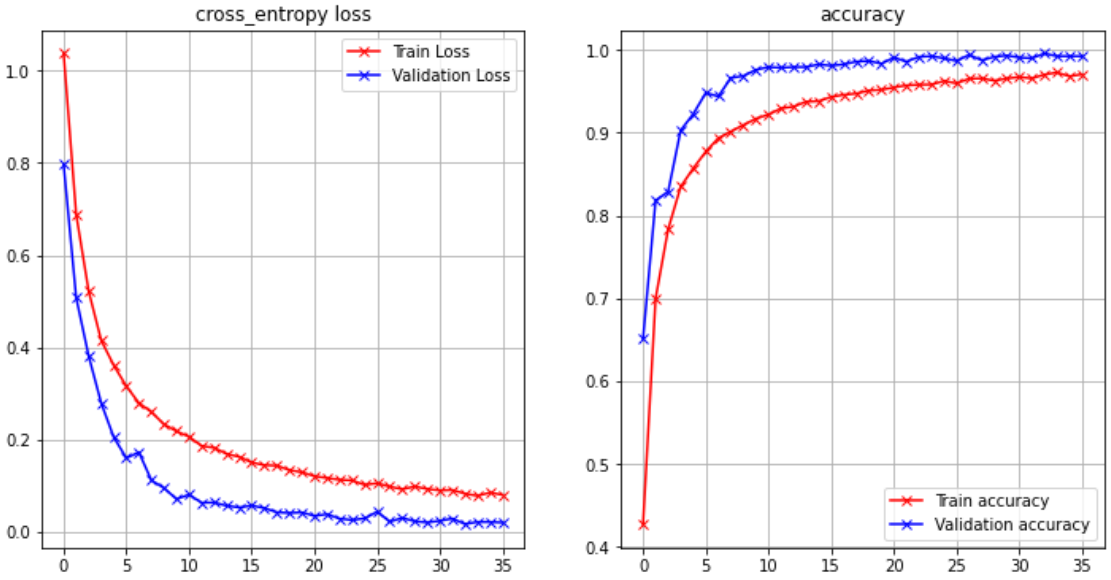

***Figura 12**. Grafici andamento loss e accuracy della rete creata da zero*

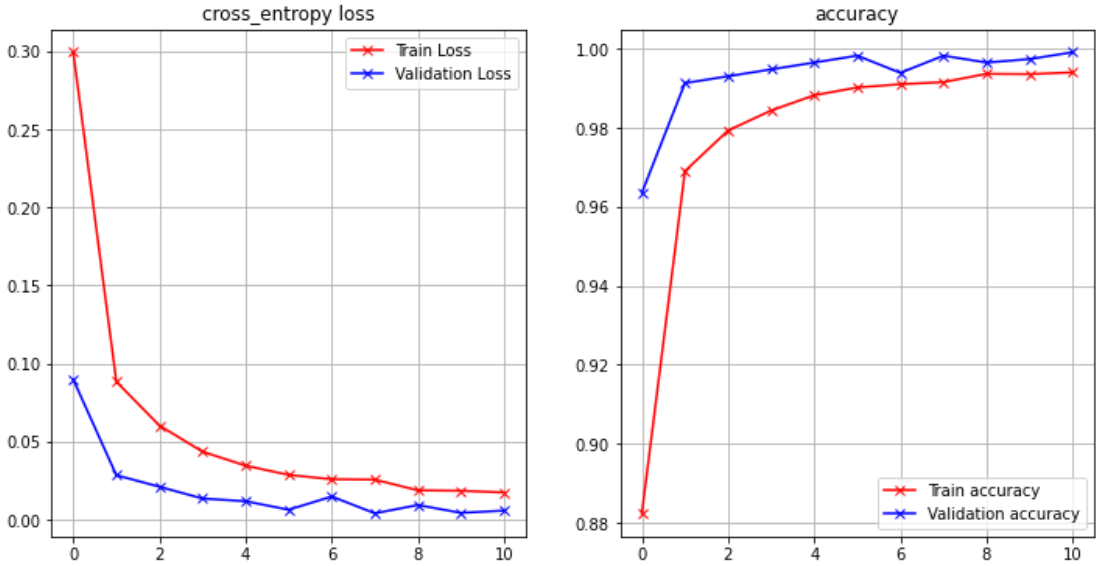

***Figura 13**. Grafici andamento loss e accuracy di VGG16*

Ponendo l'attenzione sulla prima coppia di grafici relativi all'addestramento della rete convoluzionale creata da zero, in particolare quello in cui viene riportato l'andamento della loss al progredire delle epoche, ci si accorge che entrambe le curve hanno un andamento decrescente tendente verso lo zero.
La forma delle curve suggerisce una buona capacità di generalizzazione da parte della rete, con un errore di predizione sul validation set che diminuisce in maniera concorde con l'errore sul training set.
Se la rete fosse in regime di funzionamento overfitting, la curva della loss sul validation avrebbe un trend crescente dopo una certa epoca, mentre in questo caso l'andamento è completamente decrescente.
Il modello migliore individuato dall'early stopping è quello relativo all'epoca 33 e a cui corrispondono i valori di loss 0.0824 sul training set e 0.0164 sul validation set.
Considerando ora il grafico a lato, cioè quello in cui sono riportati i valori di accuracy relativi al training set e al validation set, si nota come entrambe le curve abbiano un andamento crescente.
Questo è dovuto al fatto che se una rete apprende correttamente dal dataset su cui sta operando, ad una diminuzione dei valori di loss sul validation set corrisponde ad un incremento dell'accuracy sul validation set.
Per tale motivo, si può affermare che la rete individuata lavora in un regime di funzionamento corretto.
All'epoca 33, i valori di accuracy relativi al training e al validation sono rispettivamente 0.9704 e 0.9959.
Passando ora alla coppia di grafici in cui sono riportati i dati dell'addestramento di VGG16, si osserva come le considerazioni fatte per la rete creata ad hoc siano valide anche in questo caso.
Infatti, osservando il primo grafico relativo all'andamento del valore di loss al progredire delle epoche, ci si accorge che entrambe le curve tendono a decrescere verso lo zero e, per ciò che è stato detto prima, questo suggerisce che la rete individuata lavora correttamente, cioè non è entrata in overfitting o underfitting.
In questo caso il modello ottimale individuato dall'early stopping è quello dell'epoca 8 e cui corrispondono i valori 0.0257 per la loss sul training set e 0.0041 per la loss sul validation set.
Allo stesso modo, il grafico della variazione dell'accuracy segue un andamento pressoché simile a quello visto per l'altra architettura, con un trend crescente per entrambe le curve.
Da ciò segue che anche questa architettura presa in esame funziona correttamente, con un valore di accuracy sul training set pari a 0.9916, mentre 0.9983 sul validation set.
Di seguito è presente una tabella al cui interno si trovano i valori di loss e accuracy discussi in precedenza:

	Validation Accuracy	***Validation Loss***	***Train Accuracy***	***Train Loss***
Rete creata da zero	*0.9959*	*0.0164*	*0.9704*	*0.0824*
VGG16	*0.9983*	*0.0041*	*0.9916*	*0.0257*

6. Conclusioni

In questa ricerca è stata discussa l'implementazione di un classificatore d'immagini che riconoscesse i gesti delle mani del gioco della morra cinese.
Nella prima parte vengono introdotte le reti neurali convoluzioni, ponendo particolare attenzione sulle metodologie utilizzate per fare in modo che le architetture individuate lavorassero in un regime di funzionamento corretto, passando successivamente alle tecnologie utilizzate per la creazione del progetto.
Nella seconda parte, invece, viene descritta inizialmente l'implementazione dei classificatori, illustrando i vari strati che li compongono, passando alle tecniche utilizzate per interfacciare le reti neurali alla webcam del proprio dispositivo per effettuare classificazione in tempo reale, fino ad arrivare alla discussione dei risultati ottenuti in seguito all'addestramento.
I risultati ottenuti dimostrano come le due reti convoluzioni siano state addestrate correttamente e come queste abbiano sviluppato una buona capacità di generalizzazione, apprendendo le feature principali per effettuare una corretta classificazione delle immagini.
Tuttavia, sia per la maggiore facilità nell'apprendimento del dataset, sia per i risultati ottenuti in seguito al training, si può concludere che l'approccio basato sull'uso di una rete pre-addestrata sia preferibile.
In futuro si potrebbe pensare di sfruttare il lavoro svolto ed ampliarlo per la classificazione simultanea di più mani in tempo reale, rendendo l'applicazione maggiormente interattiva.

Bibliografia

[1]. *Api di ImageDataGenerator* [Online]. Available:
https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator
[2]. *Google Colaboratory* [Online]. Available:
https://research.google.com/colaboratory/
[3]. *Introduzione a Jupyter* [Online]. Available:
https://www.ionos.it/digitalguide/siti-web/programmazione-del-sito-web/notebook-jupyter/
[4]. *Tensorflow* [Online]. Available:
https://www.tensorflow.org/
[5]. *Open Source Computer Vision Library (OpenCV)* [Online]. Available:
https://docs.opencv.org/
[6]. *Computer Vision Zone (CVZone)* [Online]. Available:
https://github.com/cvzone
[7]. *Mediapipe* [Online]. Available:
https://google.github.io/mediapipe/
[8]. *Script python per acquisizione di immagini dalla webcam* [Online]. Available:
https://github.com/SouravJohar/rock-paper-scissors/blob/master/gather_images.py