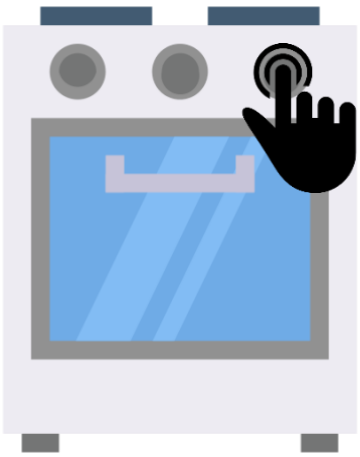

PROGETTO DI MACHINE LEARNING

Giovanni Martucci

Sommario

1. Introduzione al problema trattato

Il task di questo progetto è “classificare i differenti tocchi delle varie manopole in un forno domestico”.

Nello specifico, dato un forno, il nostro algoritmo, attraverso una rete neurale addestrata ad hoc, deve essere in grado di riconoscere se non si sta toccando alcuna manopola oppure, se si, quale nello specifico.

In Fig. 1 vengono mostrate alcune immagini per comprendere meglio il problema:

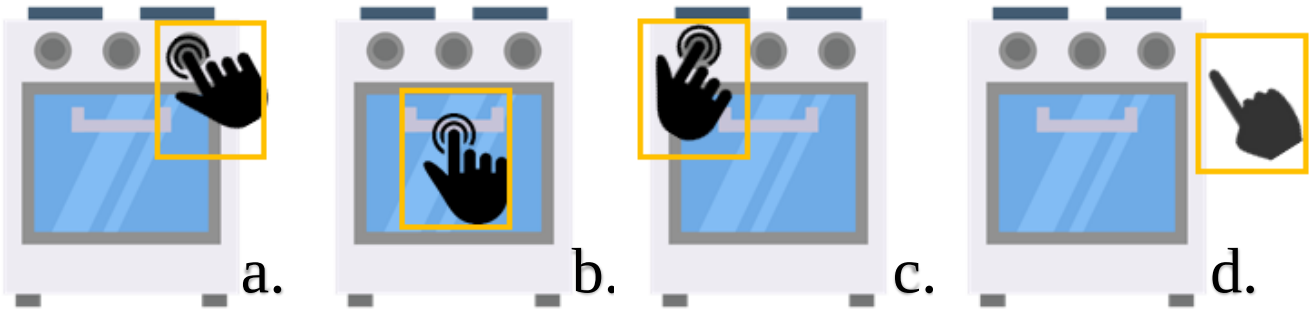

Figure 1. Tocco manopola destra (a), tocco manopola centrale (b), tocco manopola sinistra (c) , nessun tocco (d)

In questo progetto si è dovuto lavorare su delle immagini, utilizzate come input per la nostra rete. Motivo per cui è stata utilizzata una Rete Neurale Convoluzionale (CNN) che permette di applicare le reti neurali al processamento di immagini, riuscendo a scalare immagini di grandi dimensioni e grossi dataset di immagini.

Questo problema della classificazione del tocco/non tocco in un forno domestico, in realtà, può essere esteso a macchinari industriali pensando a delle azioni che conseguono da determinate scelte. Ad esempio, è possibile pensare che dopo aver toccato un determinato pulsante, il sistema dia avvio a una specifica azione o mostri delle istruzioni inerenti a quel pulsante.

2. Dataset

Il dataset utilizzato è stato costruito manualmente, attraverso l'acquisizione video di 11 diversi forni domestici, attraverso dispositivi smartphone con risoluzione 1080 x 1920.

Dopo la prima fase di acquisizione si è proceduto a tagliare i video singolarmente suddividendoli in più parti, così da segmentarli per le relative etichette. La prima parte rappresenta il tocco della manopola destra (Fig. 2a.), la seconda il tocco della manopola sinistra (Fig. 2d.), la terza il tocco della manopola centrale (Fig. 2c.) e la quarta e ultima parte nessun tocco (Fig. 2b.).

Successivamente, è stato utilizzato uno script in Python (videotoframe.py) che permette di suddividere un video in frame, salvandoli direttamente nelle relative cartelle di destinazione (Right, Left, Center e ZNull), rappresentanti le etichette dei relativi frame.

Il numero totale di immagini ottenuto è 5166, le quali poi sono state suddivise dall'algoritmo di Machine Learning in 3050 per il training set, 508 immagini per il validation set e 1526 per il test set.

a.   b.

c.   d.

Figure 2. Tocco manopola destra (a), tocco manopola sinistra (b), tocco manopola centrale (c) , nessun tocco (d)

3. Metodo

Nel presente capitolo saranno descritte le diverse tecniche utilizzate al fine di raggiungere l'obiettivo preposto.

3.1 Rete Neurale utilizzata

La rete utilizzata è stata Resnet18, una rete neurale convoluzionale profonda 18 layer.

ResNet ha ottenuto eccellenti prestazioni di generalizzazione rispetto a tutte le altre reti convoluzionali utilizzate in questo task.

Esistono molte varianti dell'architettura ResNet, ovvero lo stesso concetto ma con un numero diverso di livelli. Abbiamo ResNet-18, ResNet-34, ResNet-50, ResNet-101 ecc.

3.2 Codice sorgente Progetto.py

In questo paragrafo verrà spiegato il codice sorgente "Progetto.py", illustrando nel dettaglio le varie tecniche utilizzate.

La prima classe implementata è stata "CSVImageDataset", la quale ci permette di caricare le immagini dal disco a partire da file CSV (Fig. 3).

```python
class CSVImageDataset(data.Dataset):
    def __init__(self, data_root, csv, transform = None):
        self.data_root = data_root
        self.data = pd.read_csv(csv)
        self.transform = transform
    def __len__(self):
        return len(self.data)
    def __getitem__(self, i):
        im_path, im_label = self.data.iloc[i]['path'], self.data.iloc[i].label
        im = Image.open(join(self.data_root,im_path)).convert('RGB')
        if self.transform is not None:
            im = self.transform(im)
        return im, im_label
```

Figure 3. Progetto.py - CSVImageDataset

Come si evince dal codice, viene effettuata una trasformazione delle immagini del dataset, tramite la funzione ".convert('RGB') in maniera tale da convertire tutte le immagini in RGB, così che nel caso in cui nel dataset ci fossero immagini in scala di grigio, esse verrebbero convertite.

Successivamente, si è proceduto con l'implementazione della funzione "split_tran_val_test" che prende il dataset in input e lo splitta in training set, validation set e test set (Fig. 4).

```python
def split_train_val_test(dataset, perc=[0.6, 0.1, 0.3]):
    train, testval = train_test_split(dataset, test_size = perc[1]+perc[2])
    val, test = train_test_split(testval, test_size = perc[2]/(perc[1]+perc[2]))
    return train, val, test
```

Figure 4. Progetto.py - split_tran_val_test()

3.2.1 Fine tuning

Nella successiva funzione viene utilizzata la tecnica del "Fine Tuning", una tra le tecniche più note del Transfer learning.

Quest'ultima tende a inizializzare i parametri di un modello con quelli di un modello già addestrato per un task diverso e quindi riuscire a svolgere un task simile. Ciò avviene caricando il modello pre-allenato con i rispettivi pesi, adattando la tipologia della rete e "continuando" ad allenare il modello sul nuovo dataset.

Dunque, uno dei vantaggi delle reti neurali è che, una volta allenate per risolvere un dato problema, esse possono essere riutilizzate per risolvere nuovi task mediante questa tecnica.

Supponendo che il task originale sia simile al nuovo task per cui si deve adattare il modello:

Si sostituisce l'ultimo livello (quello di output) , originariamente addestrato a riconoscere un certo numero di classi, con un livello che riconosce un numero di classi diverse, quelle richieste dal nuovo task, per cui si adatterà il modello;

Il nuovo livello di output collegato al modello viene quindi addestrato per acquisire le funzionalità di livello inferiore e mapparle alle classi di output desiderate, utilizzando l'algoritmo "Discesa stocastica del gradiente";

Nel dettaglio, la funzione in Figura 5 prende in input il numero di classi di output, e tramite la riga 69 applica il fine-tuning al modello già allenato, inserendo il numero di classi preso in input.

```
66    def get_model(num_class=4):
67        model = resnet18(pretrained=True)
68        num_class = 4
69        model.fc = nn.Linear(512, num_class)
70        model.num_classes = num_class
71        return model
```

Figure 5. Progetto.py - get_model()

3.2.2 Regolarizzazione

Per evitare il verificarsi dell overfitting, ovvero un fenomeno che crea un modello sin troppo complesso, sono state applicate le seguenti tecniche di regolarizzazione:

- Dropout;
- Data augmentation;
- Batch normalization.

Dropout

Il dropout permette di ridurre l'overfitting rimuovendo in maniera casuale dei nodi della rete a training time. La RESNET18, implementa di default tale tecnica.

Data augmentation

Questa tecnica consiste nell'aumentare i dati in maniera sintetica. Ciò permette di forzare il modello a generalizzare rispetto ad alcune condizioni. La tecnica della data augmentation viene applicata trasformando a runtime i dati in input in maniera casuale e facendo in modo che, dopo la trasformazione, l'etichetta sia ancora valida. Alcune trasformazioni comuni sono le seguenti:

- Flip orizzontale;
- Color jittering: si va a perturbare il valore di ogni pixel in maniera casuale;
- Random crop: un crop di dimensione inferiore a quella di input viene estratto casualmente dall'immagine applicando la data augmentation ad ogni epoca.

Nello specifico sono stati creati due oggetti per l'applicazione di tale tecnica:

```
143     train_transform = transforms.Compose([
144         transforms.Resize(256),
145         transforms.RandomCrop(224),
146         transforms.RandomHorizontalFlip(),
147         transforms.ToTensor(),
148         transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
149     ])
150
151     test_transform = transforms.Compose([
152         transforms.Resize(256),
153         transforms.CenterCrop(224),
154         transforms.ToTensor(),
155         transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
156     ])
```

Figure 6. Data augmentation (Progetto.py)

Come si vede in Figura 6 , vi sono due oggetti, uno per i dati di training e uno per quelli di testing. Vediamo che per entrambi è stato applicato un resize delle immagini, una conversione in tensore e infine applicata la normalizzazione.
Per la fase di training sono stati utilizzati il RandomCrop e il RandomHorizontalFlip, che servono a ritagliare e a capovolgere le immagini in maniera casuale.
Non abbiamo queste due trasformazioni in fase di testing perchè esso deve essere sempre replicabile.

Batch normalization

La batch normalization permette di ridurre la varianza delle attivazioni dei layer intermedi della rete e può essere applicata inserendo nell'architettura il layer nn.BatchNorm2d nel caso di layer di convoluzioni e nn.BatchNorm1d nel caso di layer fully connected.

I livelli di batch normalization vanno inseriti prima di ogni layer, eccetto il primo.

3.2.3. Training model

La funzione per l'allenamento del modello è la seguente:

```
class AverageValueMeter():
    def __init__(self):
        self.reset()
    def reset(self):
        self.sum = 0
        self.num = 0
    def add(self, value, num):
        self.sum += value*num
        self.num += num
    def value(self):
        try:
            return self.sum/self.num
        except:
            return None

def trainval_classifier(model, train_loader, test_loader, exp_name='experiment', lr=0.01, epochs=10, momentum=0.99, logdir='logs'):
    criterion = nn.CrossEntropyLoss()
    optimizer = SGD(model.parameters(), lr, momentum=momentum)
    loss_meter = AverageValueMeter()
    acc_meter = AverageValueMeter()
    writer = SummaryWriter(join(logdir, exp_name), flush_secs=1)
    print(join(logdir, exp_name))
    device = "cuda" if torch.cuda.is_available() else "cpu"
    model.to(device)
    loader = {
        'train' : train_loader,
        'test' : test_loader
    }
    global_step = 0
    for e in range(epochs):
        print(e)
        for mode in ['train','test']:
            loss_meter.reset(); acc_meter.reset()
            model.train() if mode == 'train' else model.eval()
            with torch.set_grad_enabled(mode=='train'):
                for i, batch in enumerate(loader[mode]):
                    x=batch[0].to(device)
                    y=batch[1].to(device)
                    output = model(x)
                    n = x.shape[0]
                    global_step += n
                    l = criterion(output,y)
                    if mode=='train':
                        l.backward()
                        optimizer.step()
                        optimizer.zero_grad()
                    acc = accuracy_score(y.to('cpu'),output.to('cpu').max(1)[1])
                    loss_meter.add(l.item(),n)
                    acc_meter.add(acc,n)

                    if mode=='train':
                        writer.add_scalar('loss/train', loss_meter.value(), global_step=global_step)
                        writer.add_scalar('accuracy/train', acc_meter.value(), global_step=global_step)

            writer.add_scalar('loss/' + mode, loss_meter.value(), global_step=global_step)
            writer.add_scalar('accuracy/' + mode, acc_meter.value(), global_step=global_step)
        torch.save(model.state_dict(),'%s-%d.pth'%(exp_name,e+1))
    return model
```

Figure 7. Train_classifier

Come si evince dall'immagine in Figura 7, utilizziamo l'oggetto "AverageValueMeter", per ottenere delle stime di loss e accuracy per ogni epoca.

Tramite l'utilizzo della funzione "train_classifier" inizia la vera fase di training definendo la funzione di loss, il learning rate, il momentum e le epoche.

Tramite la funzione SummaryWriter si dichiara un oggetto write, il quale permette di mostrare i risultati graficamente su TensorBoard (riga 112,113 in Figura X).

Nelle righe 80 e 81 viene verificata la presenza di cuda. In caso contrario viene utilizzata la CPU. Alla riga 82 si definisce un dizionario contenente i loader di training e test; infine vi sono due cicli for: il primo che ciclerà tutte le epoche definite e il secondo itererà le due modalità di training e di test.

All riga 91 vengono abilitati i gradienti solo in fase di training; successivamente verrà aggiornato il global step definito alla riga 86.

Alla riga 96 viene effettuata la classificazione per i campioni; il risultato verrà passato alla funzione di loss (in questo caso la cross-entropy) che fornirà l'errore, calcolando il gradiente della loss rispetto a tutti i parametri.

Successivamente vi è la fase di aggiornamento dei parametri e il calcolo dei valori di accuracy e di loss per ogni epoca.

Tramite la funzione "write.add_scalar" si andrà a generare il grafico su TensorBoard che indicherà se il modello sta convergendo o meno.

Infine, alla riga 114 vengono salvati i pesi del modello allenato.

3.2.4. Test classifier

L'ultima funzione definita in questo codice sorgente è la "test_classifier" (Fig. 8) che valuta le performance del modello, ottenendo le predizioni per ciascuno degli elementi di test:

```
127  def test_classifier(model, loader):
128          device = "cuda" if torch.cuda.is_available() else "cpu"
129          model.to(device)
130          predictions, labels = [], []
131          for batch in loader:
132              x = batch[0].to(device)
133              y = batch[1].to(device)
134              output = model(x)
135              preds = output.to('cpu').max(1)[1].numpy()
136              labs = y.to('cpu').numpy()
137              predictions.extend(list(preds))
138              labels.extend(list(labs))
139          return np.array(predictions), np.array(labels)
140
```

Figure 8. test_classifier

3.3 Codice sorgente Algoritm.py

Il codice sorgente “Algoritm.py” viene utilizzato dalla demo implementata, la quale sarà approfondita nei capitoli successivi, per l’utilizzo del modello allenato.

```
1  import cv2
2  import os
3  import Progetto
4  import torch
5  from PIL import Image
6  from torchvision import models,transforms
7  import numpy as np
8  import argparse
9  import io
10 import requests
11 from torch.autograd import Variable
12 from torch.nn import functional as F
13 import pdb
14 from torch import topk
15 import skimage.transform
16 from matplotlib.pyplot import imshow
17
18 ap = argparse.ArgumentParser()
19 ap.add_argument("-v", "--video", required=True,
20     help="video selected")
21 args = vars(ap.parse_args())
22
23
24 classes = {
25   "0": "Center",
26   "1": "Left",
27   "2": "Right",
28   "3": "Null"
29 }
30
```

Figure 9. Algorithm.py

Nelle prime righe del codice sono state importate le librerie di interesse.

Successivamente, è stato implementato l’oggetto “argparse”, utile per il passaggio di parametri da riga di comando.

Alla riga 24 si ha la dichiarazione di un dizionario contenente le etichette descritte per il task in questione (Figura 9).

```
50 vidcap = cv2.VideoCapture("./uploads/"+args["video"])
51 count = 0
52 success,image = vidcap.read()
53 success = True
54
55
56 #image = Image.open("./uploads/prova4.png")
57 image2 = image[...,::-1]
58 image2 = Image.fromarray(image2)
59 image2 = test_transform(image2)
60 img_w= int(image.shape[1])
61 img_h= int(image.shape[0])
62 fps = 10.0
63 #delay =int(1000*1/fps)
64 out = cv2.VideoWriter("./output/output-"+args["video"][:-4]+".avi",cv2.VideoWriter_fourcc(*'MJPG'), fps, (img_w,img_h),True)
65 model = Progetto.get_model()
66 model.load_state_dict(torch.load('./weights/resnet_dataset_finetuning-99.pth',map_location=torch.device('cpu')))
67 model.eval()
```

Figure 10. Algorithm.py

Tramite l’utilizzo della funzione “VideoCapture” (Figura 10), viene caricato il video da classificare. Successivamente, viene estratto il primo frame, così da poter ottenere le dimensioni width ed height dell’intero video e tramite la riga 64 viene dichiarato l’oggetto out, che rappresenterà il video finale di output.

Alla riga 65 e 66 viene caricato il modello con i relativi pesi allenati precedentemente.

```
71  while success:
72      output = model(image2.unsqueeze(0))
73      output2 = output.tolist()
74      output2 = np.reshape(output2,-1)
75      m = max(output2)
76      index = [i for i, j in enumerate(output2) if j == m]
77      print(index[0])
78      cv2.putText(image, str(classes[str(index[0])]), (50,100), cv2.FONT_HERSHEY_PLAIN, 5, (0, 255, 0), 2)
79      out.write(image)
80
81
82      success,image = vidcap.read()
83      if success:
84          image2 = image[...,::-1]
85          image2 = Image.fromarray(image2)
86          image2 = test_transform(image2)
87      count += 1
88
89  out.release()
90  convert_video('./output/output-'+args["video"][:-4]+'.avi','output/final_output-'+args["video"][:-4])
91  #uso un converitore da avi a mp4 perche se codifico direttamente in mp4 viene troppo di bassa qualità
92  os.remove('./uploads/'+args["video"])
```

Figure 11. Algorithm.py

Nella riga 71 in Figura 11, viene dichiarato un ciclo while, all'interno del quale viene effettuata la classificazione vera e propria per ogni frame estrapolato dal video preso in input (riga 72). Tramite la funzione max() alla riga 75 viene estratta la probabilità massima di appartenenza ad una classe e tramite le righe 78 e 79 viene aggiunto il frame, con la relativa classe di appartenenza stampata su di esso in alto a sinistra (come in figura 12), nel video di output che verrà generato a fine esecuzione.

Figure 12. Frame classificato

Infine, tramite la funzione dichiarata nell'immagine sottostante (Figura 13) viene convertito il video di output in formato .mp4 per essere visualizzato nella demo.

```
38  def convert_video(avi_file_path, output_name):
39      os.popen("ffmpeg -i '{input}' -ac 2 -b:v 2000k -c:a aac -c:v libx264 -b:a 160k -vprofile high -bf 0 -strict experimental -f mp4 '{output}.mp4'".format(input = avi_file_path, output = output_name
40      return True
```

Figure 13. Video convert

4. Valutazioni

Durante la fase di sperimentazione, sono state utilizzate diverse metriche per valutare le performance dei modelli generati: accuracy, recall, e precision.

Un framework utilizzato per l'analisi grafica è Tensorboard, il quale ha permesso di monitorare la fase di training e di testing di ogni esperimento, analizzando i grafici con cui si può osservare se il modello arriva a convergenza o meno.

In Figura 14 è possibile osservare il grafico di ResNet18, ottenuto in fase di allenamento, dove è possibile valutare l'efficienza dell'allenamento.

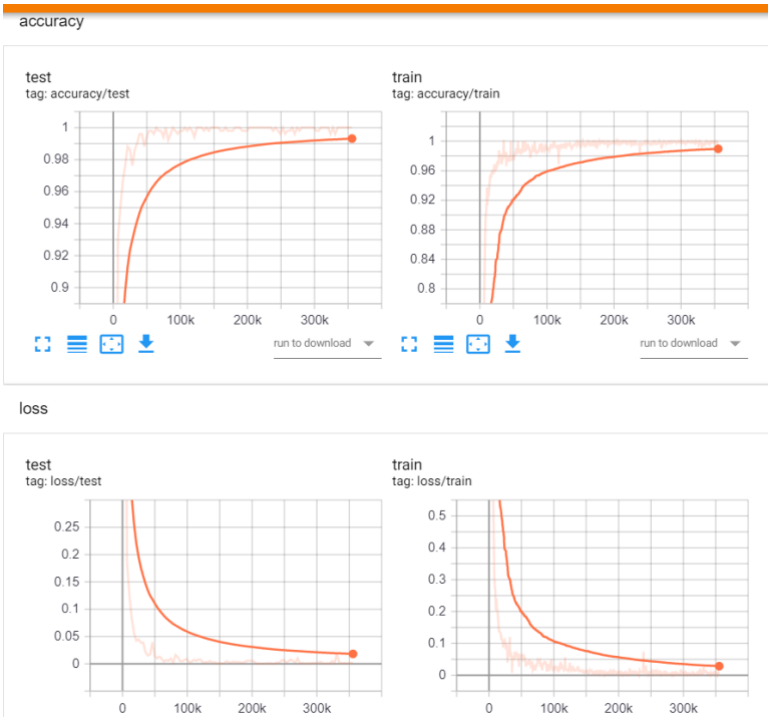

Figure 14. TensorBoard ResNet-18

Tra tutte le reti utilizzate è stata scelta Resnet18, perchè è quella che è riuscita a effettuare le migliori inferenze sul forno.

Di seguito le immagini rappresentanti l'accuracy sia del training set che del test set (Figura 15) che come si può osservare sono stati ottenuti ottimi risultati, 99,87% per il training e 99,75% per il testing.

```
99
Accuarcy di training: 99.87%
Accuracy di test - ultimo modello: 99.75%
Precision scores
(array([100.        , 100.        ,  99.51980792,  99.93108201]),)
Recall scores
(array([100.        ,  99.75550122,  99.76905312,  99.57627119]),)
```

Figure 15. Valori metriche

5. Esperimenti

Durante lo sviluppo del task sono stati effettuati molteplici esperimenti, provando diverse reti neurali, confrontando le performance, tramite i valori di accuracy, precision e recall.
In una seconda fase è stata applicata la tecnica del "Detectron2" al dataset per tentare di migliorare le performance del modello sviluppato.
Nello specifico sono state prese in esame: Squeezenet, GoogleNet, VGG16, ResNet18 e ResNet50.
Di seguito, riportiamo l'elenco delle reti utilizzate con i relativi valori di Accuracy, precision e recall ottenuti:

- **Squeezenet**: nonostante siano stati ottenuti ottimi valori di accuracy, il modello non è riuscito a effettuare una buona inferenza sui video forniti nella demo (Figura 16-17);

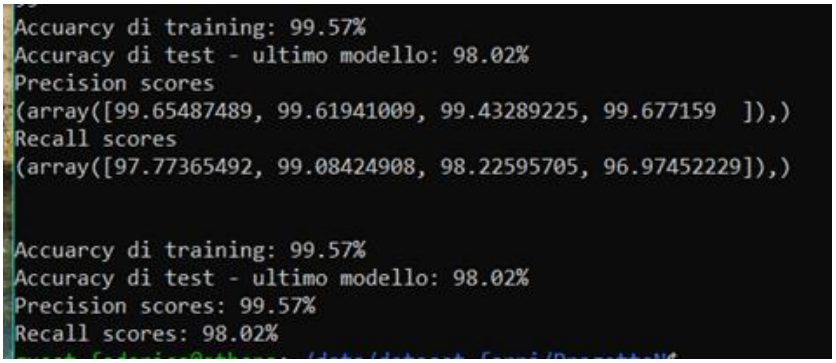

Figure 16. valori Squeezenet

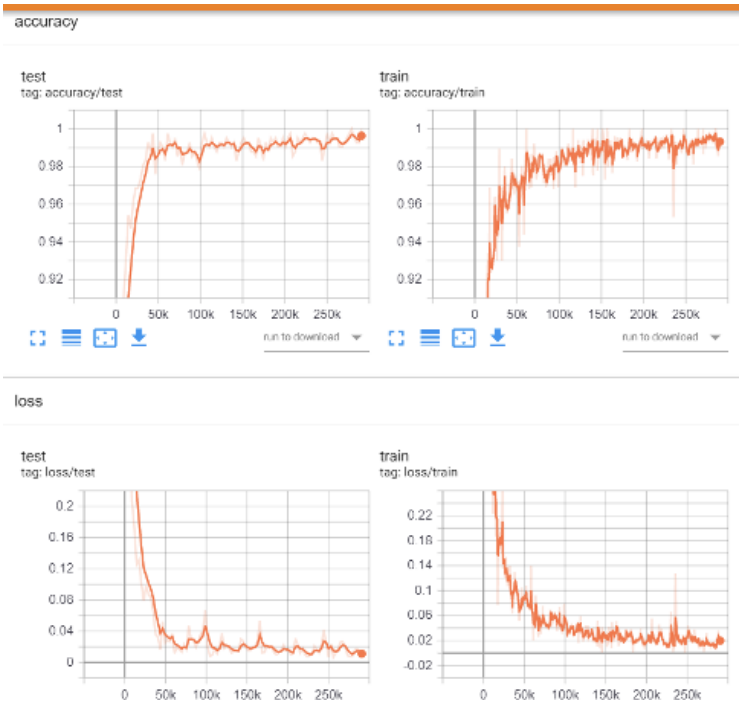

Figure 17. Grafico TensorBoard Squeezenet

- **GoogleNet**: ha dato fin dall'inizio dei buoni risultati, sia per la convergenza che per le metriche utilizzate, ma non è riuscita a classificare correttamente; questo modello è riuscito a classificare 2 etichette su 4 (Figura 18-19).

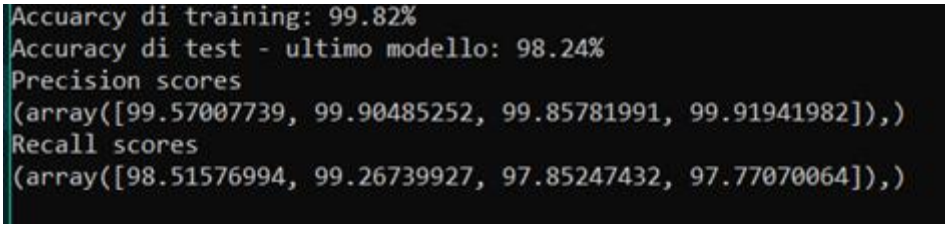

Figure 18. valori GoogleNet

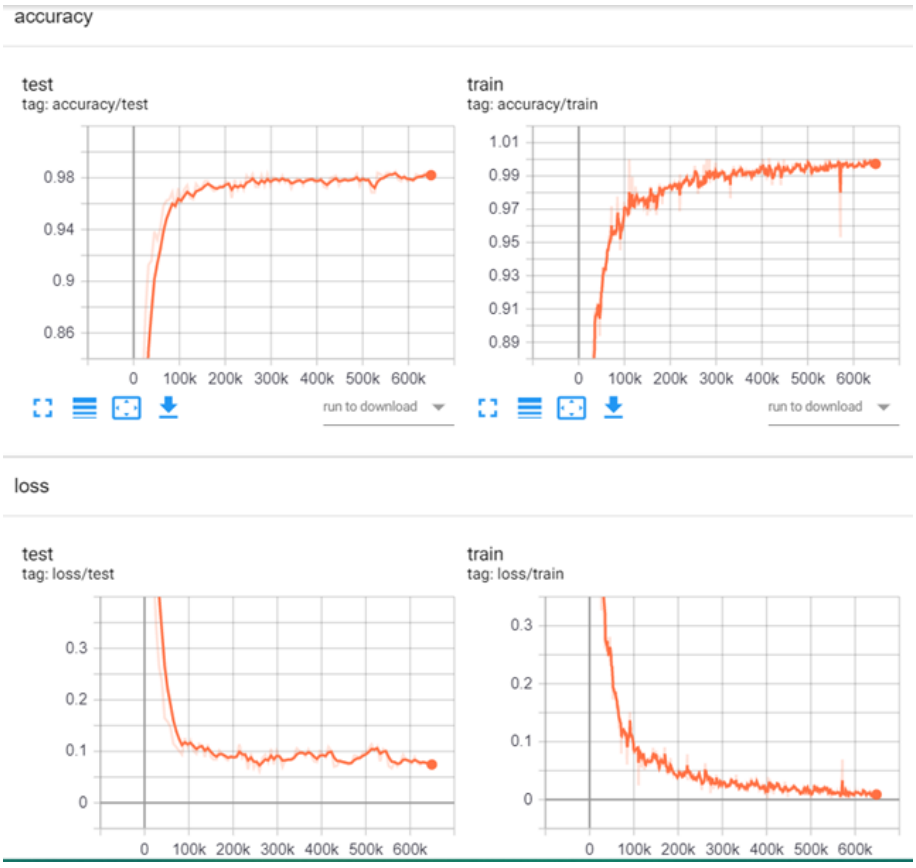

Figure 19. Grafico TensorBoard GoogleNet

- **VGG16**: una rete abbastanza pesante, che come si evince dalle immagini sottostanti si è rivelata non adatta al nostro task, avendo dei valori di accuracy bassi rispetto a tutte le altre reti e dunque sbagliando totalmente a classificare (Fig. 20-21);

```
Accuracy di training: 88.34%
Accuracy di test - ultimo modello: 85.32%
Precision scores
(array([88.5655278, 87.4903031, 89.5638318, 88.3986694]),)
Recall scores
(array([89.       , 88.5784398, 88.9954822, 89.1127583]),)
```

Figure 20. valori VGG16

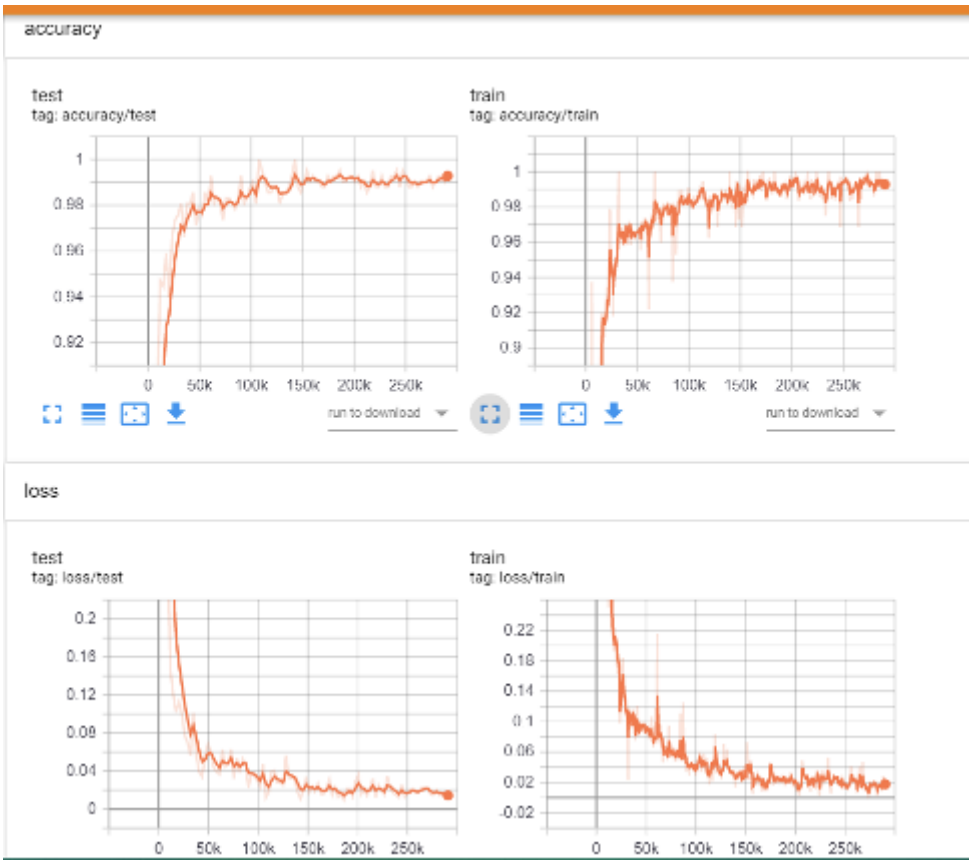

Figure 21. Grafico TensorBoard VGG16

- **ResNet50**: ottimi valori di accuracy, buona inferenza (Fig. 22-23);

```
99
Accuarcy di training: 99.80%
Accuracy di test - ultimo modello: 99.67%
```

Figure 22. valori Resnet50

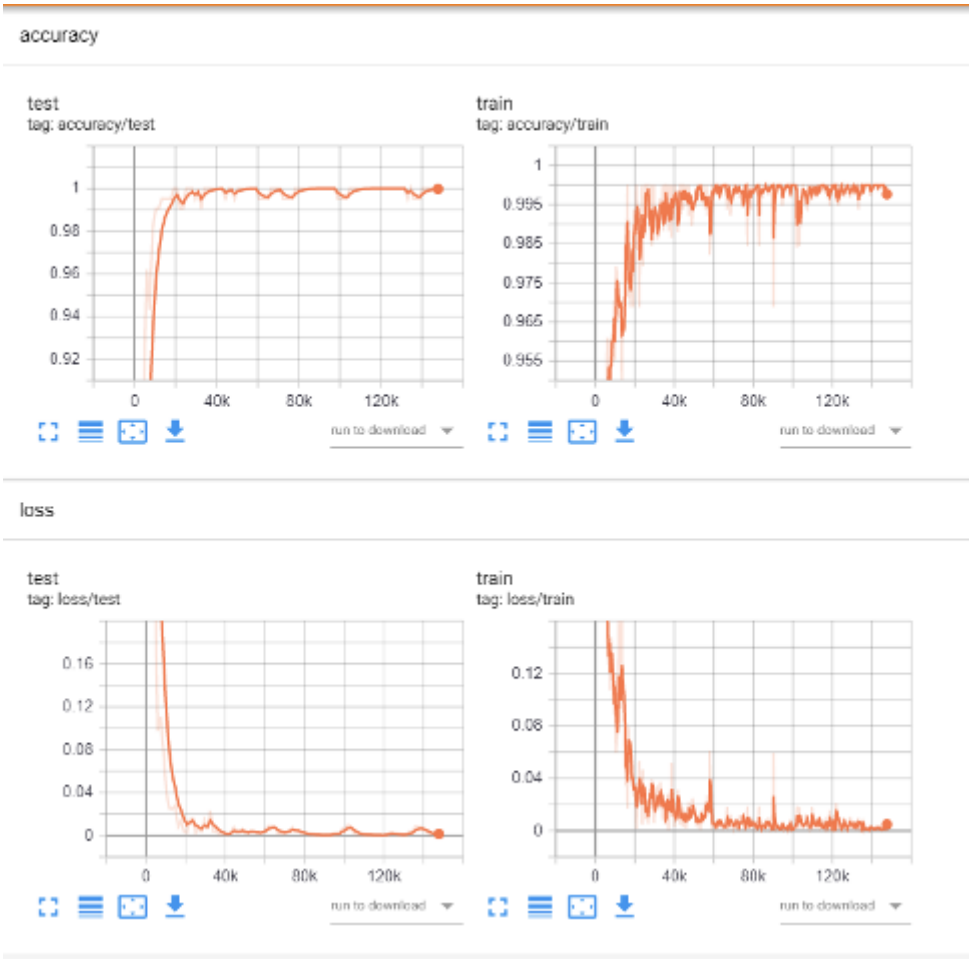

Figure 23. Grafico TensorBoard ResNet-50

- **ResNet18**: ottimi valori di accuracy, recall e precision. Qui si ottiene il migliore modello che riesce ad effettuare le migliori inferenze su i nuovi dati (Fig. 24-25).

```
gg
Accuarcy di training: 99.87%
Accuracy di test - ultimo modello: 99.75%
Precision scores
(array([100.        , 100.        ,  99.51980792,  99.93108201]),)
Recall scores
(array([100.        ,  99.75550122,  99.76905312,  99.57627119]),)
```

Figure 24. valori ResNet-18

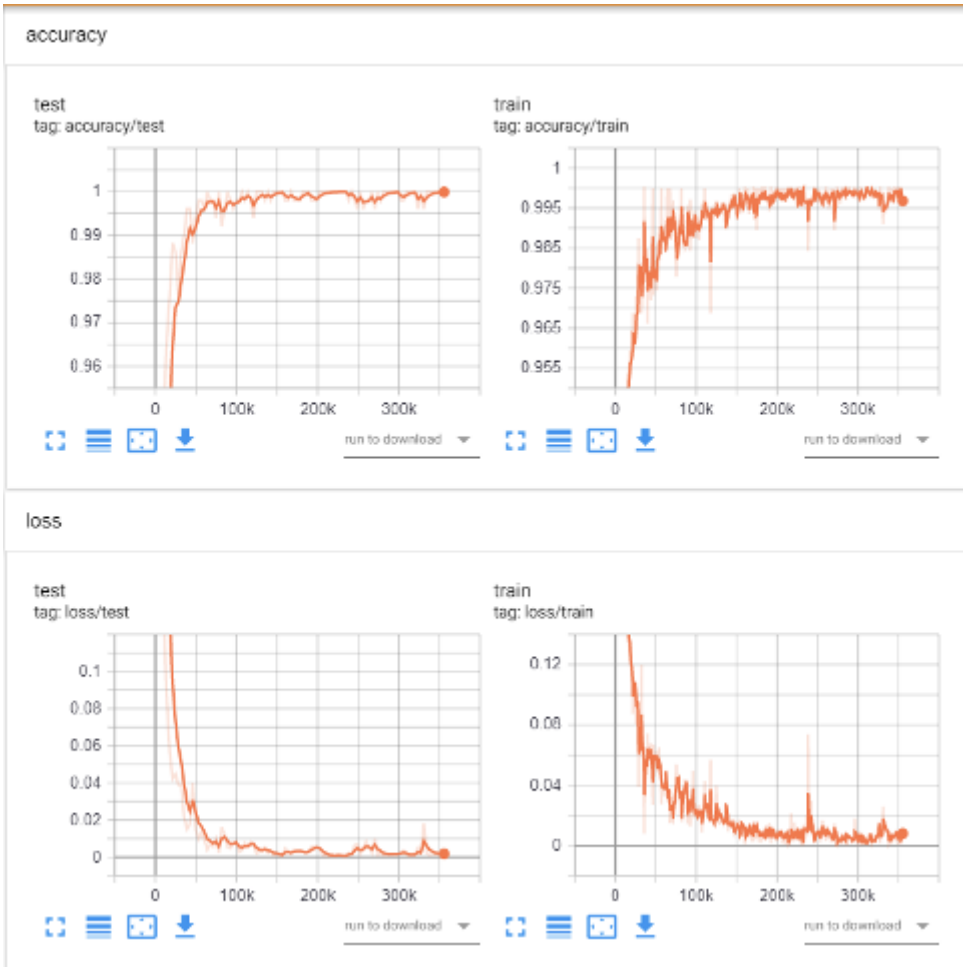

Figure 25. Grafico TensorBoard ResNet-18

Tra tutti i test effettuati e tutte le reti neurali utilizzate è stata scelta la ResNet-18, rivelandosi la più performante. Infatti, è riuscita a effettuare delle ottime inferenze, riuscendo a distinguere correttamente quale manopola del forno si stia toccando. (Figura 26)

Figure 26. Classificazione Resnet18

Ai fini di ottenere un confronto di prestazioni, apportando delle modifiche al dataset utilizzato, è stato utilizzato il software Detectron2, il quale implementa algoritmi all'avanguardia per il rilevamento degli oggetti. Un modello pre-allenato è stato applicato all'intero dataset, creando nuove immagini in cui è applicata una fase di detection iniziale degli oggetti in questione, disegnando delle bounding box su ognuno di essi, con il relativo nome della classe di appartenenza, insieme alla percentuale di esattezza e infine ogni oggetto viene segmentato per colore (Fig. 27).

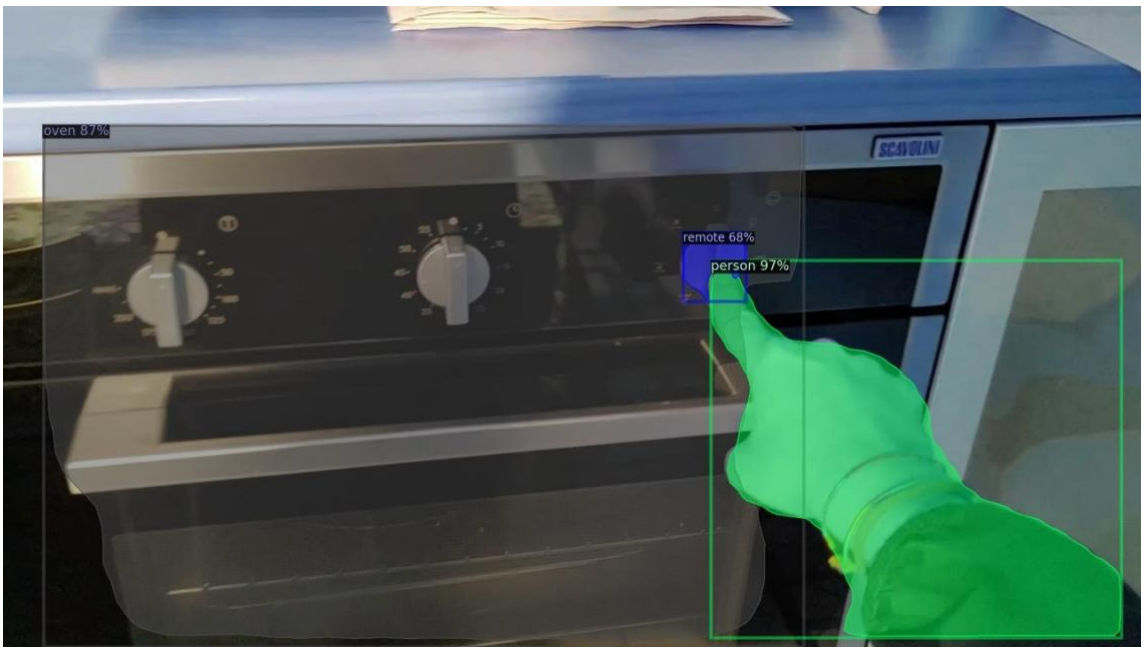

Figure 27. Detectron2 example

Il codice dell'algoritmo è abbastanza semplice e viene espletato in poche righe:

```
1  import torch, torchvision
2  import detectron2
3  from detectron2.utils.logger import setup_logger
4  import numpy as np
5  import os, json, cv2, random
6  from google.colab.patches import cv2_imshow
7  from detectron2 import model_zoo
8  from detectron2.engine import DefaultPredictor
9  from detectron2.config import get_cfg
10 from detectron2.utils.visualizer import Visualizer
11 from detectron2.data import MetadataCatalog, DatasetCatalog
12
13 for folder in os.listdir(os.getcwd()+"/dataset/images"):
14     if folder == "Right" or folder == "Left" or folder == "Center" or folder == "ZNull":
15         pass
16     else:
17         for once in os.listdir("./dataset/images/"+folder):
18             im = cv2.imread("./dataset/images/"+folder+"/"+once)
19             cfg = get_cfg()
20             cfg.merge_from_file(model_zoo.get_config_file("COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml"))
21             cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.5
22             cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml")
23             predictor = DefaultPredictor(cfg)
24             outputs = predictor(im)
25             v = Visualizer(im[:, :, ::-1], MetadataCatalog.get(cfg.DATASETS.TRAIN[0]), scale=1.2)
26             out = v.draw_instance_predictions(outputs["instances"].to("cpu"))
27             folderoutput = folder[:-1]
28             print("./dataset/images/"+folderoutput+"/"+once)
29             cv2.imwrite("./dataset/images/"+folderoutput+"/"+once,out.get_image())
```

Figure 28. Detectron.py

Come si può vedere dall'immagine in Fig. 28, oltre al caricamento delle librerie d'interesse, sono stati implementati due cicli for, il primo che cicla per ogni etichetta del nostro dataset, non considerando le nuove cartelle di destinazione (Right, Left, Center, ZNull). Tramite il secondo ciclo si estrapola ogni singola immagine all'interno di ogni singola etichetta.

Tramite le righe 19-20-21-22-23 viene caricato il modello e successivamente con la riga 24 si va a effettuare le detection sull'immagine in input, successivamente vengono aggiunte all'immagine processata le 3 componenti principali (etichettatura degli oggetti, percentuale, e delle bounding box attorno a essi).

Infine, le immagini vengono salvate nelle cartelle relative alle loro etichette.

6. Demo

La demo è stata realizzata tramite l'utilizzo di due Framework: Nodejs ed Express.js; come comunicazione client-server è stata implementata una comunicazione tramite socket, utilizzando la libreria javascript "Socket.io".

Avviato il server da terminale (tramite comando 'node index.js'), vi si reca all'indirizzo http://*localhost:8080* in cui troveremo la pagina iniziale. Qui la demo inizia la sua esecuzione prendendo in input uno o più video da dover mandare in pasto al modello, il quale effettuerà

un’inferenza sulla classe di appartenenza per ogni frame contenuto nel video: come vediamo dalla gif in basso (Figura 29).

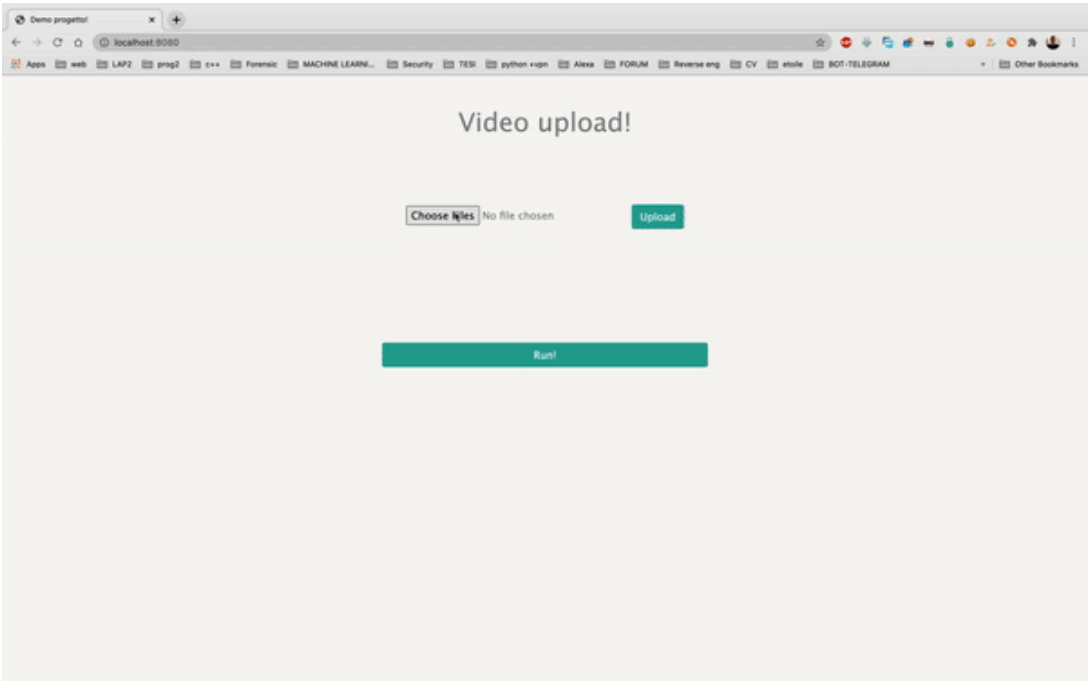

Figure 29. Demo upload

Vi è la possibilità di rimozione per ogni video caricato ed infine vi è il tasto “Run”, con il quale si da inizio all’esecuzione dell’algoritmo.

Dopo aver cliccato il tasto Run ci troveremo davanti una schermata di attesa (Figura 30).

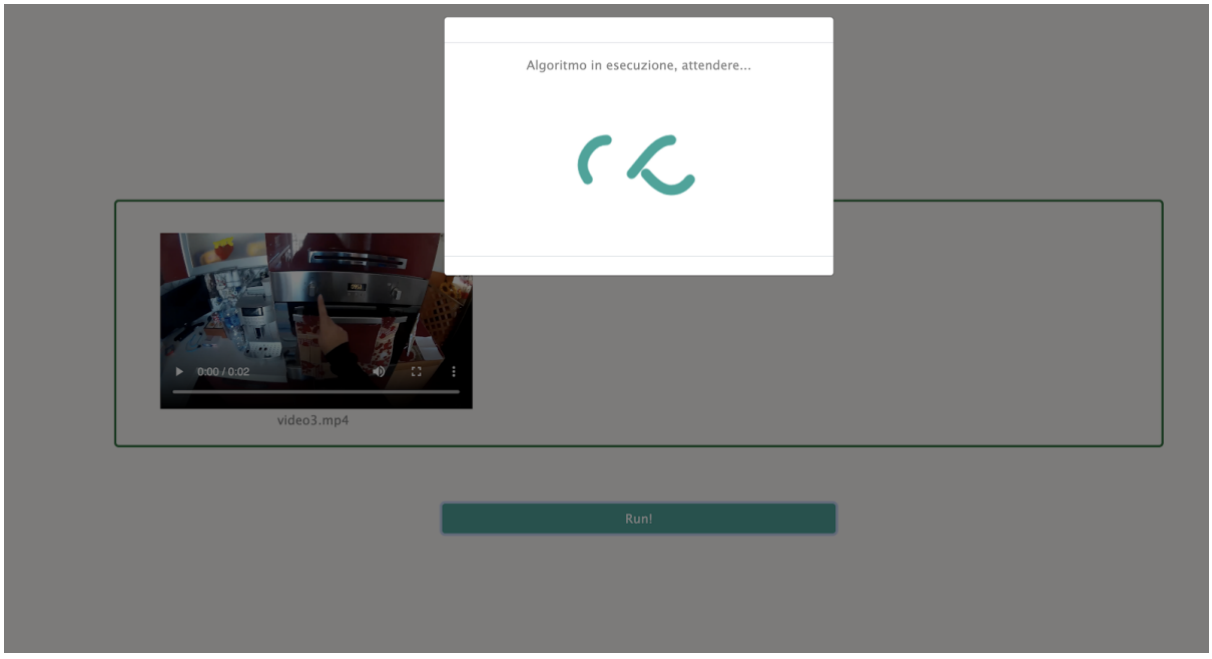

Figure 30. Demo (fase di attesa)

Infine, una volta terminata l’esecuzione su i video in input, ci ritroveremo nella schermata di output che mostrerà i risultati della classificazione , così come viene mostrato in Figura 31.

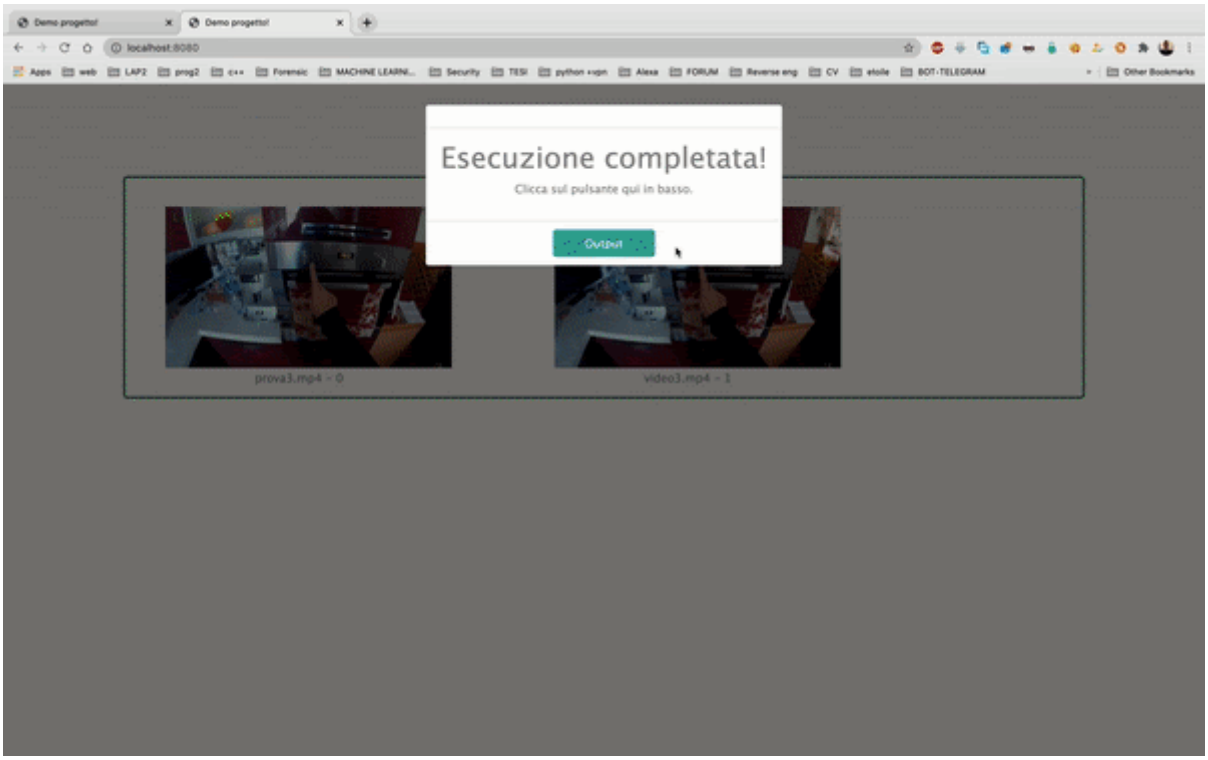

Figure 31. Demo (fase di output)

7. Codice

La cartella “Progetto Team 04” è strutturata nel seguente modo:

Progetto Team 04

Name	Date Modified	Size	Kind
__pycache__	19 Jul 2020 at 15:29	--	Folder
dataset	22 Jul 2020 at 09:38	--	Folder
ffmpeg	2 Oct 2019 at 15:20	--	Folder
logs	19 Jul 2020 at 15:21	--	Folder
node_modules	24 Jun 2020 at 17:00	--	Folder
output	28 Jul 2020 at 17:46	--	Folder
public	11 Jul 2020 at 14:26	--	Folder
uploads	28 Jul 2020 at 17:46	--	Folder
weights	28 Jul 2020 at 16:43	--	Folder
index.js	22 Jul 2020 at 12:18	3 KB	JavaScript
package-lock.json	24 Jun 2020 at 17:00	8 KB	JSON Document
package.json	24 Jun 2020 at 17:00	251 bytes	JSON Document
algorithm.py	19 Jul 2020 at 16:56	7 KB	Python Source
detectron.py	29 Jul 2020 at 12:12	2 KB	Python Source
Progetto.py	19 Jul 2020 at 15:05	9 KB	Python Source

Figure 32. Struttura cartella

- **Dataset**: cartella contenente il dataset diviso per etichette (Right, Left, Center, ZNull) e 4 file csv, di cui 3 contenenti i dataframe suddivisi per training, validation e test set e 1 file csv denominato ‘classes.csv’ contenente le corrispondenze tra classi e relativi id;
- **ffmpeg:** cartella contenente file per la conversione e manipolazione di stream audio e video. Nel caso specifico viene utilizzato per generare l’output della demo, effettuando una conversione video da .avi a .mp4 ;
- **logs:** cartella contenente il file di log generato dalla funzione “SummaryWriter”, al fine di mostrare il risultato di training su Tensorboard;
- **nodes_modules:** cartella contenente i moduli di nodejs ed express.js;
- **output:** cartella contenente gli output video generati ad ogni esecuzione (non viene effettuato nessuna segmentazione utente, perchè la demo è solo a scopo illustrativo).
- **public:** cartella contenente lo spazio utente che fornisce il server ad ogni collegamento con un nuovo client. Nello specifico ad ogni collegamento viene restituita la pagina “index.html” che contiene l’interfaccia grafica con cui l’utente si interfaccia per utilizzare l’algoritmo.
- **uploads:** cartella contenente i video caricati dall’utente che saranno dunque verranno processati dall’algoritmo.
- **weights:** cartella contenente i pesi generati in fase di training;
- **index.js:** è il sorgente server, che gestisce tutta la fase di upload, gestisce la fase di esecuzione dell’algoritmo di machine learning e infine la fase di output;
- **package-lock.json, package.json:** questi due file contengono vari metadati rilevanti per il progetto. Entrambi vengono utilizzati per fornire informazioni npm; permettono quindi di identificare il progetto e gestire le dipendenze del progetto.
- **algorithm.py:** file sorgente che utilizza il modello generato in fase di training, analizzando i video presi in input dall’utente, e generando dei video di output contenenti l’etichettatura/classe di appartenenza per ogni frame.
- **Progetto.py:** file sorgente in cui vi è tutta la fase di generazione del modello, dunque vi è la fase di training, validation e test.
- **Detectron.py:** file sorgente in cui si applica la tecnica di Detectron2 che implementa algoritmi di rilevamento oggetti, così come spiegato nei paragrafi precedenti.

Conclusione

In conclusione, dopo una prima fase di acquisizione dei dati per la formazione del dataset, si è proceduto alla fase di training del modello per risolvere il task proposto. Durante tutto il periodo di sviluppo sono state riscontrate diverse difficoltà che hanno portato all'esecuzione di diversi esperimenti, cambiando dataset e reti neurali. Diverse tecniche note sono state implementate per il raggiungimento dell'obiettivo: è stato utilizzato Detectron2 pre-allenato sul dataset, per favorire l'apprendimento della rete; la tecnica di Class Activation Map (CAM) è stata utilizzata per visualizzare la concentrazione della rete sui soggetti coinvolti (mano e forno). Nonostante ciò tra i vari esperimenti e confronti si è deciso (tramite il confronto delle prestazioni) di utilizzare il dataset originale perché ha dato i migliori risultati sul nostro modello.

Dunque, da questo progetto è stato appreso il concetto di rete neurale e modello di classificazione, apprendendo le varie tecniche, le best practices e i vari tricks presenti per migliorare in termini di prestazioni e velocità i modelli generati.

I risultati ottenuti sono abbastanza ottimali, ma per migliorare il metodo proposto si potrebbe allenare Detectron2 appositamente per il nostro task specifico, ignorando tutti gli oggetti al di fuori del forno, delle manopole e delle mani. Passando, successivamente, il dataset ottenuto a una rete neurale, la quale dovrà apprendere soltanto gli spostamenti della mano in base alla manopola, risolvendo molto probabilmente questo punto debole, riscontrato durante lo sviluppo di questo task preposto.