



Università
di Catania

Dipartimento di Matematica e Informatica

Corso di Laurea Magistrale in Informatica

Michele Ferrigno
Elvio Santangelo
Manuel Zuccaro

Corso di Machine Learning
A.A. 2023/2024

Sign Language Recognizer

Prof. Giovanni Maria Farinella

Sommario

INTRODUZIONE	3
DATASET	4
Creazione dataset	7
Struttura del Dataset	13
METODI	14
Esplorazione del task: dataset MNIST.....	14
Primo modello migliorato: <i>Simple_CNN</i>	15
Secondo modello migliorato: <i>SignLanguage_DeepCNN</i>	16
Modelli	17
ESPERIMENTI E VALUTAZIONI.....	22
Esperimento n°1	22
Esperimento n°2	23
Esperimento n°3	24
DEMO.....	25
CODICE.....	27
Struttura del Codice	27
Esecuzione del codice.....	28
CONCLUSIONI	29
Possibili miglioramenti	29

INTRODUZIONE

L'argomento di studio del progetto è stato il riconoscimento di caratteri del linguaggio dei segni (Sign Language) attraverso l'utilizzo di un modello CNN.

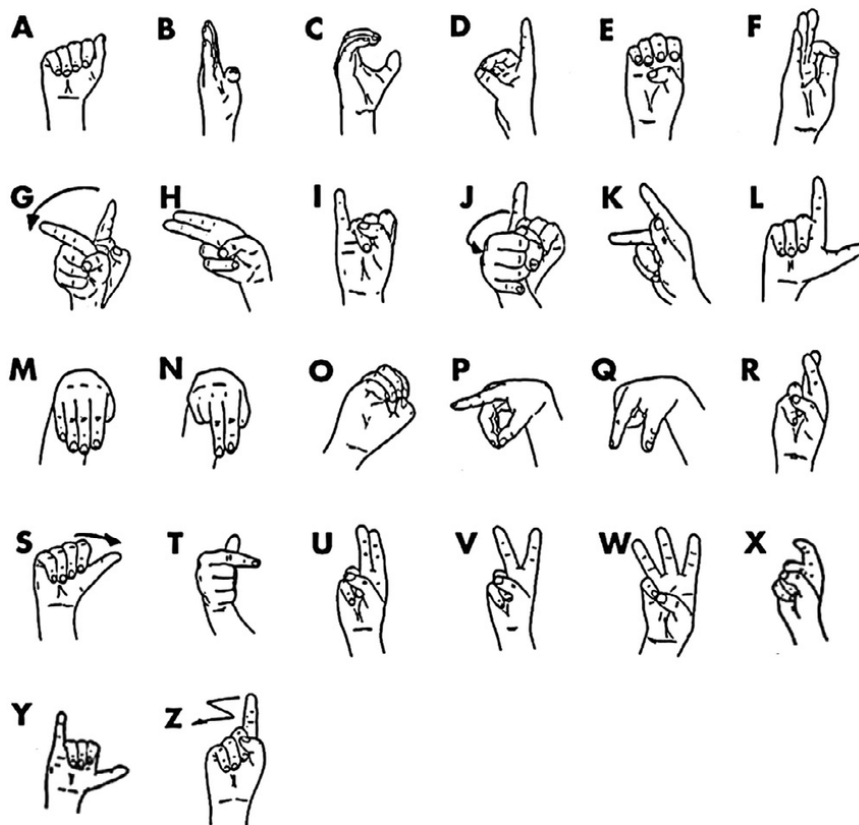
Le Lingue dei Segni (più in generale Sign Languages) costituiscono una famiglia di linguaggi, che vengono utilizzate principalmente da persone sorde o con deficit più o meno grave dell'udito.

Si tratta di veri e propri linguaggi, aventi regole grammaticali e sintattiche proprie, che sfruttano gesti delle mani, del corpo ma anche espressioni del volto.

Esistono diverse Lingue dei Segni in tutto il mondo, ad esempio ASL (American Sign Language) e ISL (Indian Sign Language). In Italia esiste il LIS, Lingua Italiana dei Segni.

Si tratta di un ambito in cui grazie ai notevoli progressi della Computer Vision è logico pensare a una qualche soluzione che permetta un'interpretazione più o meno automatica del linguaggio, che troverebbe applicazioni particolarmente utili ad abbattere le barriere nella comunicazione.

Nel nostro progetto ci siamo concentrati sul riconoscimento delle lettere dell'alfabeto, in particolare quello LIS.



DATASET

Nel progetto sono stati utilizzati due dataset.

1) Dataset ASL- MNIST:

Si tratta di un dataset di esempio per il task di classificazione per linguaggio dei segni americano. Consta di circa 34 mila immagini, divise in train_set (80%) e test_set(20%). Ogni immagine ha dimensione 28x28 pixels e raffigura una delle 24 lettere dell'alfabeto vista frontalmente (la J e la S sono mancanti in quanto richiedono un movimento della mano).

Fonte ed eventuali informazioni aggiuntive al link

<https://www.kaggle.com/datasets/datamunge/sign-language-mnist/>

2) Dataset Custom:

È il dataset che è stato creato appositamente per il progetto. In questo caso è stato scelto lo standard LIS dei segni che prevede 22 lettere (le lettere G, J, S, Z richiedono un movimento). Con l'ausilio di uno script in python per l'acquisizione manuale delle immagini sono state acquisite immagini 64x64 dei vari segni, cercando di variare background/posizione del segno e cercando di includere leggere variazioni nell'angolazione del segno, proprio in virtù delle considerazioni fatte per il dataset MNIST.

Il dataset finale consta di oltre 70 mila immagini che è stato suddiviso in train_set (70%) e test_set(30%).

Nota aggiuntiva:

Il dataset custom consiste in immagini .jpg e quindi è richiesto un pre-processing prima che possano essere utilizzate. La pipeline mostrata implementa proprio questa operazione. Le immagini vengono raccolte per lettera, così che la label è direttamente nota; con Image.open() si carica l'immagine, che subisce due trasformazioni: transform_base e transform_aug. La prima consiste in una conversione in scala di grigio e conversione a tensore, la seconda aggiunge alcune operazioni di augmentation che include RandomPerspective con parametri 0.3 e 0.2, un flip orizzontale con probabilità del 0.5 e una rotazione casuale di al più 20°.

Per ogni immagine del dataset originale, il dataset contiene una trasformazione base e una trasformazione augmented, raddoppiando il numero di immagini.

```

from torchvision import transforms
import os
from PIL import Image
import torch

#Path dei datasets
DATASET1_PATH = "DataAcquisition/pythonProject/Dataset_Manuel"
DATASET2_PATH = "DataAcquisition/pythonProject/Dataset_michele_esteso/"
DATASET3_PATH = "DataAcquisition/pythonProject/Dataset_Elvio_esteso_black_background"

datasets = [DATASET1_PATH,DATASET2_PATH,DATASET3_PATH]

#Creazione dataset con Data Augmentation
data = []

for letter in dict_alph:
    for dataset in datasets:
        sub_folder = os.path.join(dataset, letter)
        for img_name in os.listdir(sub_folder):
            img_path = os.path.join(sub_folder, img_name)
            im0 = transform_base(Image.open(img_path))
            im1 = transform_aug(Image.open(img_path))
            label = dict_alph[img_name[0]]
            data.append((im0, label))
            data.append((im1, label))

print(data[0][0].shape) # Should be (1, 64, 64) for grayscale images
print("DATASET SIZE:",len(data))

```

```

transform_aug = transforms.Compose([
    transforms.Grayscale(num_output_channels=1),
    transforms.RandomPerspective(0.3,0.2),
    transforms.RandomHorizontalFlip(0.5),
    transforms.RandomRotation(20),
    transforms.ToTensor(),
])

transform_base = transforms.Compose([
    transforms.Grayscale(num_output_channels=1),
    transforms.ToTensor(),
])

```

Concludiamo il pre-processing splittando train_set e test_set e istanziando i DataLoader che verranno passati in input alla funzione di training.

```
from sklearn.model_selection import train_test_split

def split_train_val_test(dataset, perc=None):

    if perc is None:
        perc = [0.7, 0.3]
    train, test = train_test_split(dataset, test_size=perc[1], train_size=perc[0])

    return train, test

train, test = split_train_val_test(data, [0.7, 0.3])
```

```
from torch.utils.data import DataLoader

train_data = DataLoader(train, batch_size=64, num_workers=2, shuffle=True)
test_data = DataLoader(test, batch_size=64, num_workers=2)
```

Creazione dataset

Per la creazione del dataset è stata sviluppata una procedura di acquisizione dei dati. Questo dataset è stato successivamente utilizzato per le fasi di addestramento e test della CNN, al fine di ottimizzare le prestazioni del modello nel riconoscimento del linguaggio dei segni.

La procedura di acquisizione è stata implementata tramite uno script Python, che sfrutta la libreria OpenCV per la cattura delle immagini e Tkinter per la gestione dell'interfaccia grafica.

Processo di Acquisizione delle Immagini

Il processo di acquisizione delle immagini è stato progettato con estrema attenzione per garantire una raccolta dati sistematica e controllata, essenziale per la qualità e l'affidabilità del dataset finale. Questo processo è stato suddiviso in una serie di passaggi chiave, ognuno dei quali riveste un ruolo cruciale nell'assicurare che le immagini acquisite soddisfino gli standard richiesti. Di seguito sono descritti in dettaglio i passaggi chiave del processo:

1. Configurazione dell'Ambiente

Per iniziare, è stata utilizzata la libreria OpenCV per attivare la webcam tramite la funzione `cv2.VideoCapture()`. Questa funzione permette di accedere alla webcam e acquisire il flusso video in tempo reale. Per gestire l'interfaccia grafica, è stata utilizzata la libreria Tkinter, che consente di creare una finestra in cui visualizzare il feed video e fornire strumenti di interazione con l'utente.

```
111 cap = cv2.VideoCapture(0)
112 if not cap.isOpened():
113     print("Errore: Impossibile aprire la webcam")
114     return
115
```

2. Interfaccia Utente (UI)

L'interfaccia utente consente agli utenti di inserire il proprio nome e la lettera della LIS da registrare. Sono stati implementati controlli di validità per assicurare che i dati inseriti siano corretti e completi. Ad esempio, il campo destinato al nome non può essere lasciato vuoto o contenere solo numeri. Analogamente, il campo per la lettera richiede che venga inserita una singola lettera dell'alfabeto italiano, escludendo specifici caratteri ('g', 's', 'j', 'z'). La finestra principale della UI mostra il feed video della webcam, un riquadro verde che indica l'area di cattura dell'immagine, e due pulsanti: uno per acquisire l'immagine e uno per chiudere la finestra.

3. Cattura delle Immagini

Quando l'utente preme il pulsante di acquisizione (o il tasto 'c'), lo script cattura un'immagine dal flusso video. L'immagine ritagliata viene poi salvata in una cartella specifica, in modo tale da organizzare e archiviare tutte le immagini acquisite in modo ordinato. Il nome del file per

ciascuna immagine segue una struttura ben definita, utilizzando il formato <lettera>_<nome>_<numero>.jpg.

4. Salvataggio delle immagini

Le immagini catturate vengono ridimensionate a 64x64 pixel utilizzando la funzione `cv2.resize()`. La dimensione 64x64 è stata scelta per mantenere un equilibrio tra la riduzione della risoluzione e la preservazione dei dettagli necessari per il riconoscimento dei segni della LIS. Le immagini ridimensionate, in fine, vengono poi salvate nella cartella corrispondente alla lettera della LIS inserita dall'utente, utilizzando la funzione `save_image()`.

Funzionalità Implementate

Nello script sono state implementate diverse funzionalità chiave, tra cui:

- **`get_next_capture_number(directory, name, letter)`**

Questa funzione determina il numero progressivo da assegnare ad una nuova immagine catturata, analizzando i file esistenti in una directory specificata. Utilizzando una *regex* per identificare i file (<lettera>_<nome>_<numero>.jpg), la funzione calcola il numero successivo disponibile, senza sovrascrivere i file esistenti. Questo processo facilita la gestione e l'organizzazione del dataset di immagini.

```
21     def get_next_capture_number(directory, name, letter):
22         existing_files = os.listdir(directory)
23         max_number = 0
24         pattern = re.compile(rf"{letter}_{name}_{(\d+)\.jpg}")
25         for file in existing_files:
26             match = pattern.match(file)
27             if match:
28                 number = int(match.group(1))
29                 if number > max_number:
30                     max_number = number
31         return max_number + 1
```

- **`save_image(image, filename)`**

Ridimensiona e salva l'immagine catturata con il nome file specificato.


```
69 def save_image(image, filename):
70     resized_image = cv2.resize(image, dsize=(64, 64), interpolation=cv2.INTER_AREA)
71     cv2.imwrite(filename, resized_image)
72     log_message(f'Immagine salvata come {filename}')
73
```

- **capture_images(name, letter)**

Gestisce l'intero processo di acquisizione, dalla configurazione della webcam alla visualizzazione dell'interfaccia utente, fino alla cattura e al salvataggio delle immagini.

```

95  def capture_images(name, letter):
96      global capture_requested, quit_requested, capture_count, log_text
97      capture_requested = False
98      quit_requested = False
99
100     # Crea la cartella per il dataset se non esiste
101     dataset_dir = os.path.join("Dataset", letter)
102     os.makedirs(dataset_dir, exist_ok=True)
103
104     # Numero di cattura
105     capture_count = get_next_capture_number(dataset_dir, name, letter)
106
107     # Dimensione dell'immagine ritagliata
108     crop_size = 192
109
110     # Apri la webcam
111     cap = cv2.VideoCapture(0)
112     > if not cap.isOpened():...
113
114     print("Usa i pulsanti per catturare l'immagine o chiudere la finestra.")
115
116     # Configura la finestra Tkinter
117     root = tk.Tk()
118     root.title("Acquisizione")
119     root.protocol(name="WM_DELETE_WINDOW", close_window)
120
121     # Dimensioni della finestra
122     window_width = 800
123     window_height = 600
124
125     # Centra la finestra di acquisizione sullo schermo
126     screen_width = root.winfo_screenwidth()
127     screen_height = root.winfo_screenheight()
128     position_top = int(screen_height / 2 - window_height / 2)
129     position_right = int(screen_width / 2 - window_width / 2)
130     root.geometry(f"{window_width}x{window_height}+{position_right}+{position_top}")
131
132     # Canvas per visualizzare il feed della webcam
133     canvas = tk.Canvas(root, width=640, height=480)
134     canvas.pack()
135
136     # Frame per posizionare i pulsanti in basso al centro
137     button_frame = tk.Frame(root)
138     button_frame.pack(side=tk.BOTTOM, pady=10)
139
140
141

```

```

142     # Aggiungi i pulsanti
143     btn_capture = tk.Button(button_frame, text="Acquisisci (c)", command=capture_image)
144     btn_capture.pack(side=tk.LEFT, padx=20)
145     btn_close = tk.Button(button_frame, text="Chiudi (q)", command=close_window)
146     btn_close.pack(side=tk.RIGHT, padx=20)
147
148     # Text widget per loggare messaggi
149     log_text = tk.Text(root, height=10, state=tk.DISABLED)
150     log_text.pack(side=tk.BOTTOM, fill=tk.X)
151
152     > def update_frame():...
153
154     update_frame()
155
156     # Mappa i tasti "c" e "q" per i pulsanti corrispondenti
157     root.bind('<c>', lambda event: capture_image())
158     root.bind('<q>', lambda event: close_window())
159
160     log_message("Usa i pulsanti per catturare l'immagine o chiudere la finestra.")
161
162     root.mainloop()
163
164

```

- `get_user_input()`

Mostra una finestra di dialogo per ottenere il nome e la lettera della LIS dall'utente.

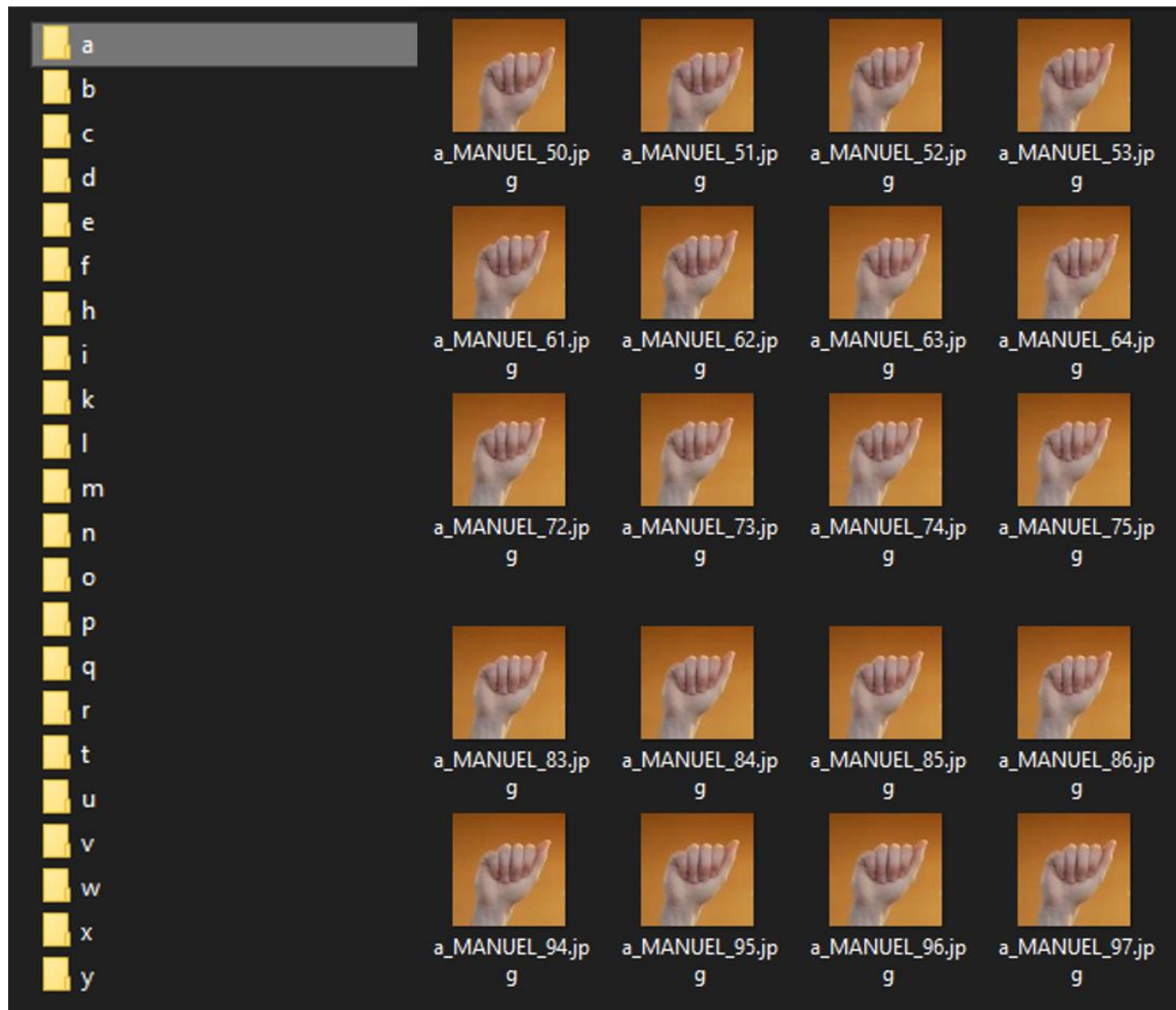
```

208 def get_user_input():
209     def on_submit(event=None):
210         global user_name, user_letter
211         user_name = name_entry.get().upper()
212         user_letter = letter_entry.get().lower()
213
214         # Controlli di validità
215         > if not user_name:...
218         > if user_name.isdigit():...
221         > if not user_letter or len(user_letter) != 1:...
224         > if user_letter in ['g', 's', 'j', 'z']:...
228         > if user_letter.isdigit():...
231
232         dialog.destroy()
233
234     def on_close():
235         global window_closed
236         window_closed = True
237         dialog.destroy()
238
239     dialog = tk.Tk()
240     dialog.title("Inserisci le informazioni")
241
242     # Centra la finestra di dialogo sullo schermo
243     window_width = 400
244     window_height = 300
245     screen_width = dialog.winfo_screenwidth()
246     screen_height = dialog.winfo_screenheight()
247     position_top = int(screen_height / 2 - window_height / 2)
248     position_right = int(screen_width / 2 - window_width / 2)
249     dialog.geometry(f"{window_width}x{window_height}+{position_right}+{position_top}")
250
251     tk.Label(dialog, text="Nome:", font=("Arial", 14)).pack(pady=10)
252     name_entry = tk.Entry(dialog, font=("Arial", 14))
253     name_entry.pack(pady=10)
254
255     tk.Label(dialog, text="Carattere della LIS:", font=("Arial", 14)).pack(pady=10)
256     letter_entry = tk.Entry(dialog, font=("Arial", 14))
257     letter_entry.pack(pady=10)
258
259     submit_button = tk.Button(dialog, text="Submit", font=("Arial", 14), command=on_submit)
260     submit_button.pack(pady=20)
261
262     dialog.bind('<Return>', on_submit)
263     dialog.protocol(name="WM_DELETE_WINDOW", on_close)
264
265     dialog.mainloop()

```

Struttura del Dataset

Il dataset creato è stato organizzato in modo sistematico, con una struttura a cartelle che facilita la gestione e l'accesso alle immagini. Ogni cartella è nominata con una singola lettera dell'alfabeto e contiene tutte le immagini corrispondenti a quella lettera. All'interno delle cartelle, ogni immagine è denominata nel formato <lettera>_<nome>_<numero>.jpg ed ha dimensione 64x64 pixel, garantendo una chiara identificazione e una facile gestione delle immagini.



METODI

Nello sviluppo del progetto è stato seguito un approccio costruttivo, partendo dall'esplorazione del task per poi passare alle effettive implementazioni. Tale relazione seguirà proprio tale struttura, descrivendo le varie fasi, i diversi problemi affrontati e discutendo infine i modelli e le loro valutazioni.

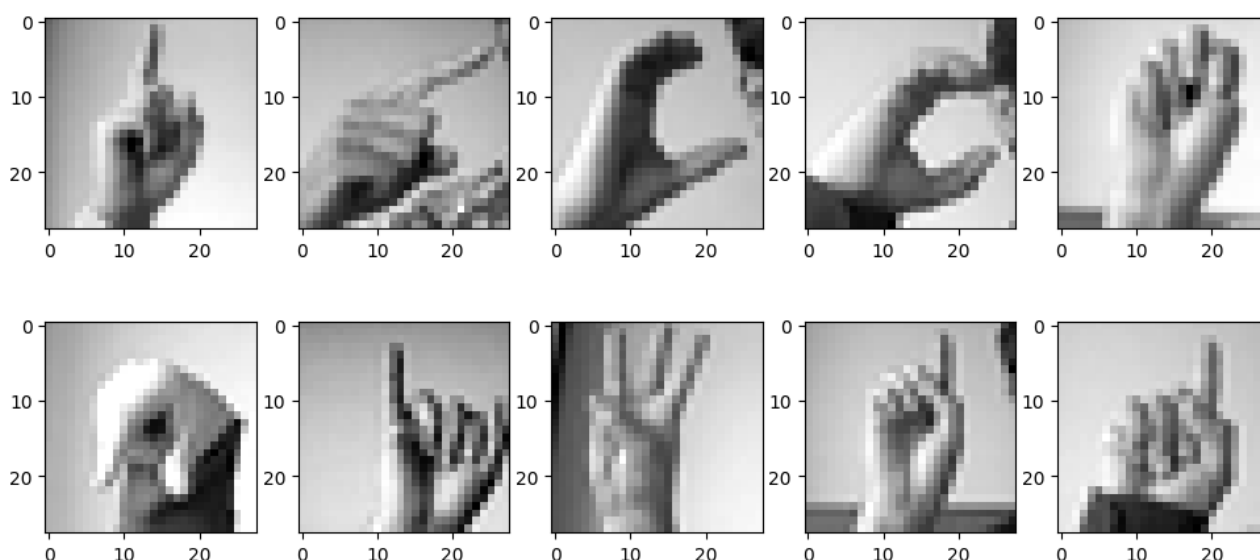
Esplorazione del task: dataset MNIST

Il primo passo è stato quello di esplorare il task, verificando se esistesse qualche dataset di prova, così da poter vedere concretamente alcuni esempi.

Siamo quindi partiti dal dataset MNIST per il ASL, linguaggio dei segni americano. Molto simile come struttura al dataset Hand Written Digits, si propone come dataset di esempio per esplorare proprio il task di nostro interesse.

Si tratta di un dataset di circa 34.000 immagini totali, già divise in train_set (80%) e test_set (20%).

Sono immagini di dimensione 28x28 pixels, ciascuna raffigurante frontalmente una lettera dell'alfabeto, per un totale di 24 lettere (sono state escluse le lettere J e S, in quanto i loro segni richiedono movimento e quindi impossibile raffigurare con una singola immagine).



Prendendo spunto proprio dalle soluzioni viste a laboratorio per la classificazione delle cifre scritte a mano, abbiamo implementato una CNN per il suddetto task.

L'esperimento migliore ha raggiunto un' Accuracy del 0.96 % sul test_set fornito da MNIST stesso.

Usando immagini diverse per testare ulteriormente il modello (usando la webcam ad esempio), il modello fatica un po', mostrando la sua incapacità di generalizzare al di fuori del dataset.

Da un lato prevedibile data la regolarità delle immagini (sfondo regolare e costante, vista frontale del segno, luminosità costante), inoltre, è un modello abbastanza semplice che lavora su immagini di appena 28x28 pixels e non permette l'effettivo apprendimento di pattern complessi.

Proprio partendo da queste considerazioni, abbiamo quindi iniziato a lavorare su come creare un dataset nostro, che risolvesse parte dei limiti visti in quello MNIST e di conseguenza come modificare adeguatamente la CNN.

Primo modello migliorato: *Simple_CNN*

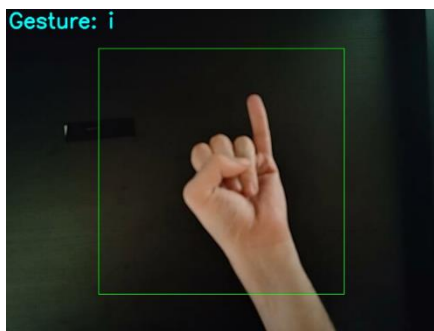
A questo punto, ci siamo occupati di rivedere il modello CNN e capire come poterlo migliorare per far uso dei nuovi dati.

Avendo adesso un dataset più ampio e vario, è necessario aumentare la capacità della rete così che possa catturare un maggior numero di pattern.

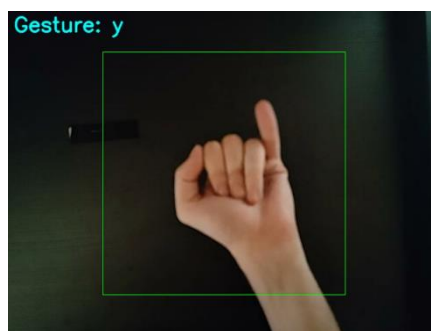
La prima prova è stata aumentare i nodi del layer Fully Connected, con l'idea che facendo così le feature maps ottenute dai layer convolutivi riescono ad essere mappate meglio alle classi, in virtù di un maggiore numero di pattern rilevati.

Effettivamente il modello risulta migliorare rispetto al Modello_MNIST in termini di riconoscimento dei caratteri, con una Accuracy sul test_set di circa 0.75 a discapito di una loss di 2.39, valore alto per poter definire il modello soddisfacente.

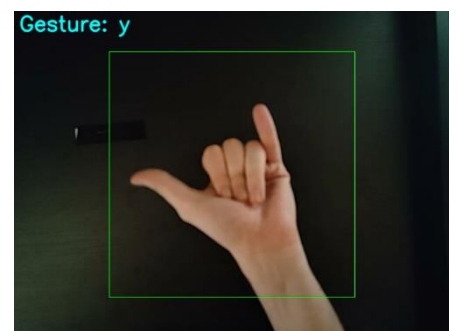
Alcune riflessioni importanti su questo modello che hanno poi suggerito le modifiche da apportare riguardano la difficoltà del modello nel distinguere segni simili tra loro. Ad esempio, le lettere O e C differiscono per la chiusura del gesto, ma per il resto la forma della mano è praticamente uguale. Oppure ancora, le lettere I e Y cambiano solo per la posizione del pollice, il resto della mano è invariato. Altre coppie di lettere che riscontrano un meccanismo analogo sono V e W, N e M.



Lettera I predetta come I



Lettera I predetta come Y



Lettera Y predetta come Y

Alla luce di queste considerazioni, è stato messo a punto il terzo modello del progetto.

Secondo modello migliorato: *SignLanguage_DeepCNN*

L'intuizione suggerita dal problema appena riscontrato è che magari non vi siano abbastanza feature maps per differenziare effettivamente segni simili.

Da ciò è stata rielaborata anche la parte di feature extraction del modello, configurando inizialmente 3 layer convolutivi (uno in più rispetto al modello precedente). Da una prima prova la soluzione non sembrava ancora essere sufficiente a risolvere il problema. Ciò ci ha portato quindi alla soluzione finale con 5 layer convolutivi. Di conseguenza sono stati aumentati anche i layer Fully Connected.

Il modello così ottenuto non solo ha superato in Accuracy il secondo modello (0.97 contro 0.75) ma ha persino raggiunto valori di loss molto buoni 0.12 sul train_set e 0.08 sul test_set, contro 2.39 del secondo modello.

Eseguendo l'applicazione con questo modello, quasi tutte le ambiguità di segni erano state risolte: O e C adesso sono sempre distinguibili, come V e W. Qualche volta I e Y vengono ancora confuse mentre N e M rimangono indistinguibili.

L'intuizione su come risolvere il problema sembrerebbe essere corretta e anche le modifiche apportate sono sulla buona strada.

Modelli

Le reti CNN che sono state sviluppate e valutate sono tre.

1) Modello_MNIST:

E' il modello sviluppato nella fase esplorativa del progetto, utilizzando il dataset MNIST. Vengono utilizzati 3 layer convoluzionali, ciascuno seguito da un layer di MaxPooling. Segue un layer di Dropout con parametro 0.2 e infine due layer Fully Connected, di cui il secondo avente 24 nodi pari alle classi del task. Nei layer conv si applica una classica ReLU, mentre per l'ultimo viene applicata una LogSoftmax con 24 classi.

```
class ModelloMNIST(nn.Module):
    def __init__(self):
        super(ModelloMNIST, self).__init__()
        self.conv1 = nn.Conv2d(1,10,3)
        self.conv2 = nn.Conv2d(10,20,3)
        self.conv3 = nn.Conv2d(20,30,3)

        self.pool = nn.MaxPool2d(2)
        self.dropout = nn.Dropout2d(0.2)

        self.fc1 = nn.Linear(30*3*3, 270)
        self.fc2 = nn.Linear(270,26)

        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self,x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.pool(x)

        x = self.conv2(x)
        x = F.relu(x)
        x = self.pool(x)

        x = self.conv3(x)
        x = F.relu(x)
        x = self.dropout(x)

        x = x.view(-1, 30 * 3 * 3)
        x = F.relu(self.fc1(x))
        x = self.softmax(F.relu(self.fc2(x)))

        return(x)
```

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 10, 26, 26]	100
MaxPool2d-2	[-1, 10, 13, 13]	0
Conv2d-3	[-1, 20, 11, 11]	1,820
MaxPool2d-4	[-1, 20, 5, 5]	0
Conv2d-5	[-1, 30, 3, 3]	5,430
Dropout2d-6	[-1, 30, 3, 3]	0
Linear-7	[-1, 270]	73,170
Linear-8	[-1, 26]	7,046
LogSoftmax-9	[-1, 26]	0
Total params: 87,566		
Trainable params: 87,566		
Non-trainable params: 0		
Input size (MB): 0.00		
Forward/backward pass size (MB): 0.09		
Params size (MB): 0.33		
Estimated Total Size (MB): 0.43		

2) SimpleCNN:

È la prima rete allenata sul dataset custom, che quindi richiede una maggiore capacità. Essendo la prima prova di “miglioramento” abbiamo provato a potenziare la parte full connected della rete ed effettivamente abbiamo ottenuto risultati migliori rispetto al primo modello. La rete consiste di 2 layer convolutivi, ciascuno seguito da MaxPooling e funzione di attivazione Relu. In questo caso usiamo anche BatchNormalization, dal momento che la quantità di dati è maggiore e le immagini sono più varie. Infine, due layer Fully Connected preceduti da Dropout a 0.20, restituiscono il risultato dopo aver applicato una Softmax con 22 classi (nel LIS i caratteri considerati sono 22)

```

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()

        self.feature_extraction = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1), # input (3, 64, 64) output (16, 64, 64)
            nn.ReLU(), # activation function
            nn.MaxPool2d(kernel_size=2, stride=2), # output (16, 32, 32)

            nn.BatchNorm2d(16),
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1), # output (32, 32, 32)
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # output (32, 16, 16)
        )

        # 22 letters in the alphabet, 22 classes in output in one fc layer

        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Dropout(0.20), # dropout layer
            nn.Linear(32 * 16 * 16, out_features=128), # input (16 * 32 * 32) output (22)
            nn.ReLU(),
            nn.Linear(in_features=128, out_features=22)
        )

    def forward(self, x):
        x = self.feature_extraction(x)
        x = self.classifier(x)
        x = F.softmax(x, dim=1)
        return x

```

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 16, 64, 64]	160
ReLU-2	[-1, 16, 64, 64]	0
MaxPool2d-3	[-1, 16, 32, 32]	0
BatchNorm2d-4	[-1, 16, 32, 32]	32
Conv2d-5	[-1, 32, 32, 32]	4,640
ReLU-6	[-1, 32, 32, 32]	0
MaxPool2d-7	[-1, 32, 16, 16]	0
Flatten-8	[-1, 8192]	0
Dropout-9	[-1, 8192]	0
Linear-10	[-1, 128]	1,048,704
ReLU-11	[-1, 128]	0
Linear-12	[-1, 22]	2,838
Total params: 1,056,374		
Trainable params: 1,056,374		
Non-trainable params: 0		
Input size (MB): 0.02		
Forward/backward pass size (MB): 1.94		
Params size (MB): 4.03		
Estimated Total Size (MB): 5.98		

3) SignLanguage_DeepCNN:

E' il terzo e ultimo modello sviluppato per il progetto.

Partendo dalla seconda rete, il nuovo modello ha 5 layer convolutivi e 4 operazioni di BatchNormalization, proprio con l'obiettivo di migliorare la capacità della fase di feature extraction alla luce del problema riscontrato con il secondo modello.

Infine, abbiamo 4 layer Fully Connected con due operazioni di Dropout a parametro 0.2 (all'aumentare della capacità del modello c'è un maggiore rischio di overfitting e il dropout cerca di mitigare tale effetto).

```
4 usages  ▲ Darakuu *
class SignLanguage_DeepCNN(nn.Module):
    ▲ Darakuu
    def __init__(self):
        super(SignLanguage_DeepCNN, self).__init__()
        self.feature_extraction = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1), #input: 1x64x64, output: 16x64x64
            nn.ReLU(),

            nn.BatchNorm2d(16),
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1), # input: 16x64x64, output: 32x64x64
            nn.MaxPool2d(kernel_size=2, stride=2), #input: 32x64x64, output: 32x32x32
            nn.ReLU(),

            nn.BatchNorm2d(32),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1), #input: 32x32x32, output: 64x32x32
            nn.MaxPool2d(kernel_size=2, stride=2), # input: 64x32x32, output: 64x16x16
            nn.ReLU(),

            nn.BatchNorm2d(64),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1), #input: 64x16x16, output: 64x16x16
            nn.ReLU(),
            # nn.MaxPool2d(2, 2),

            nn.BatchNorm2d(64),
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1), #input: 64x16x16, output: 128x16x16
            nn.MaxPool2d(kernel_size=2, stride=2), #input: 128x16x16, output: 128x8x8
            nn.ReLU(),
        )

        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Dropout(0.20),
            nn.Linear(128 * 8 * 8, out_features=256, bias=True),
            nn.ReLU(),
            nn.Dropout(0.20),
            nn.Linear(in_features=256, out_features=128, bias=True),
            nn.ReLU(),
            nn.Linear(in_features=128, out_features=64, bias=True),
            nn.ReLU(),
            nn.Linear(in_features=64, out_features=22, bias=True),
        )

    ▲ Darakuu *
    def forward(self, x):
        x = self.feature_extraction(x)
        x = self.classifier(x.view(x.shape[0],-1))
        x = F.softmax(x, dim=1)
        return x
```

Layer (type)	Output Shape	Param #
=====		
Conv2d-1	[-1, 16, 64, 64]	160
ReLU-2	[-1, 16, 64, 64]	0
BatchNorm2d-3	[-1, 16, 64, 64]	32
Conv2d-4	[-1, 32, 64, 64]	4,640
MaxPool2d-5	[-1, 32, 32, 32]	0
ReLU-6	[-1, 32, 32, 32]	0
BatchNorm2d-7	[-1, 32, 32, 32]	64
Conv2d-8	[-1, 64, 32, 32]	18,496
MaxPool2d-9	[-1, 64, 16, 16]	0
ReLU-10	[-1, 64, 16, 16]	0
BatchNorm2d-11	[-1, 64, 16, 16]	128
Conv2d-12	[-1, 64, 16, 16]	36,928
ReLU-13	[-1, 64, 16, 16]	0
BatchNorm2d-14	[-1, 64, 16, 16]	128
Conv2d-15	[-1, 128, 16, 16]	73,856
MaxPool2d-16	[-1, 128, 8, 8]	0
ReLU-17	[-1, 128, 8, 8]	0
Flatten-18	[-1, 8192]	0
Dropout-19	[-1, 8192]	0
Linear-20	[-1, 256]	2,097,408
ReLU-21	[-1, 256]	0
Dropout-22	[-1, 256]	0
Linear-23	[-1, 128]	32,896
ReLU-24	[-1, 128]	0
Linear-25	[-1, 64]	8,256
ReLU-26	[-1, 64]	0
Linear-27	[-1, 22]	1,430
=====		
Total params: 2,274,422		
Trainable params: 2,274,422		
Non-trainable params: 0		
=====		
Input size (MB): 0.02		
Forward/backward pass size (MB): 5.01		
Params size (MB): 8.68		
Estimated Total Size (MB): 13.70		
=====		

ESPERIMENTI E VALUTAZIONI

I modelli sviluppati sono stati testati e valutati nel contesto dei dataset di riferimento. Per ogni modello sono stati testati diversi preset di configurazione dei parametri per cercare quelli ottimali.

Durante il progetto sono stati eseguiti >30 esperimenti di training/testing alla ricerca dei parametri migliori, tuttavia, la maggior parte di questi erano delle prove per verificare diverse ipotesi. Di seguito verranno illustrati solo gli esperimenti più significativi nell'ottica del task e in virtù delle considerazioni fatte sui modelli.

Esperimento n°1

Modello: Modello_MNIST

Dataset: Sign Language MNIST

Funzione di Loss: CrossEntropyLoss

Ottimizzatore: SGD

Learning Rate: 0.001

Epoche: 80

Momentum: 0.90

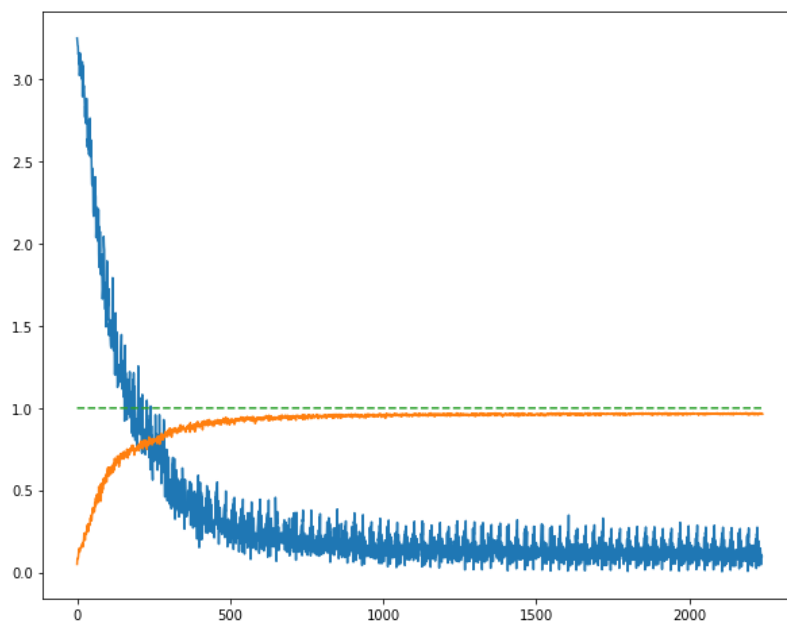
Batch Size: 32

Data augmentation: No

Risultati:

Accuracy: 0.96

Loss: 0.14



Esperimento n°2

Modello: SimpleCNN

Dataset: Dataset custom

Funzione di Loss: CrossEntropyLoss

Ottimizzatore: SGD

Learning Rate: 0.001

Epoche: 100

Momentum: 0.90

Batch Size: 64

Data augmentation: No

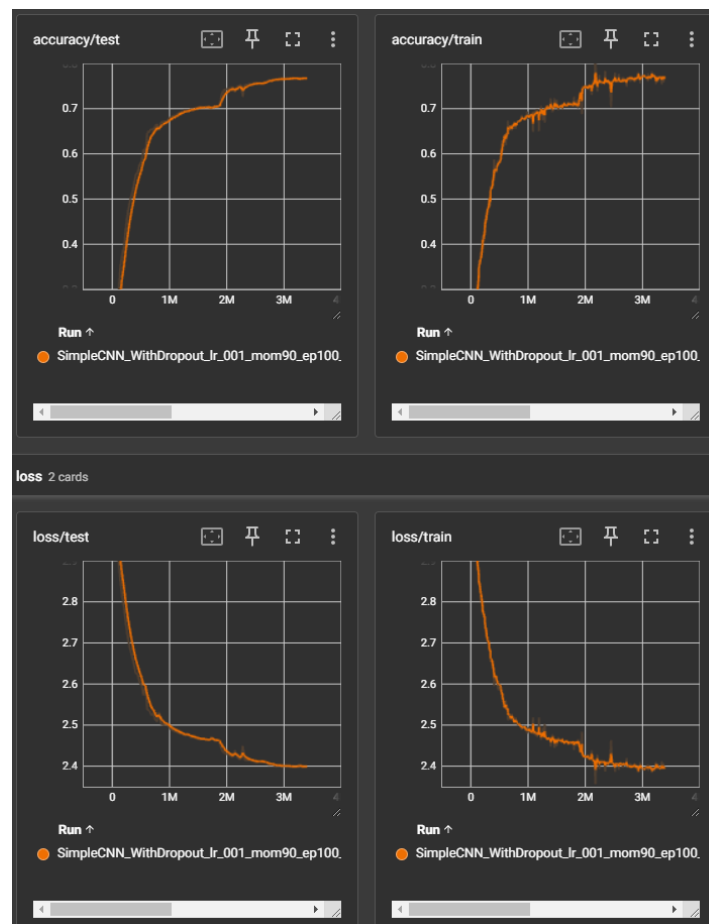
Risultati:

Accuracy/test: 0.76

Accuracy/train: 0.77

Loss/test: 2.39

Loss/train: 2.39



Esperimento n°3

Modello: SignLanguage_DeepCNN

Dataset: Dataset custom

Funzione di Loss: CrossEntropyLoss

Ottimizzatore: SGD

Learning Rate: 0.001

Epoche: 100

Momentum: 0.90

Batch Size: 64

Data augmentation: Si

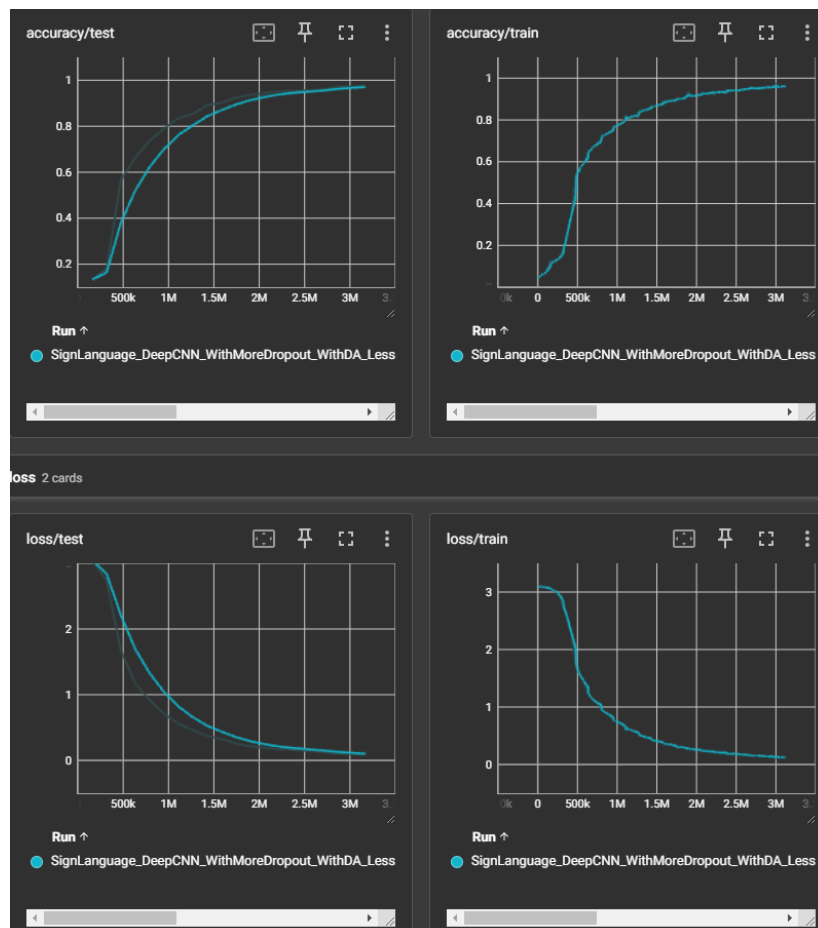
Risultati:

Accuracy/test: 0.97

Accuracy/train: 0.96

Loss/test: 0.08

Loss/train: 0.125



DEMO

Il progetto implementa tre applicativi: due sono messi a disposizione dell'utente, mentre invece il terzo è una utility per la creazione di dataset.

I due software (SignRecognizer, SignRecognizer_Offline) a disposizione dell'utente permettono di individuare in tempo reale, o meno, le 22 lettere dell'alfabeto su cui è stato addestrato il modello. In particolare, l'applicazione riconosce in real-time il gesto raffigurato dall'utente, con una frequenza di 60 immagini al secondo, direttamente dal feed della videocamera in input.

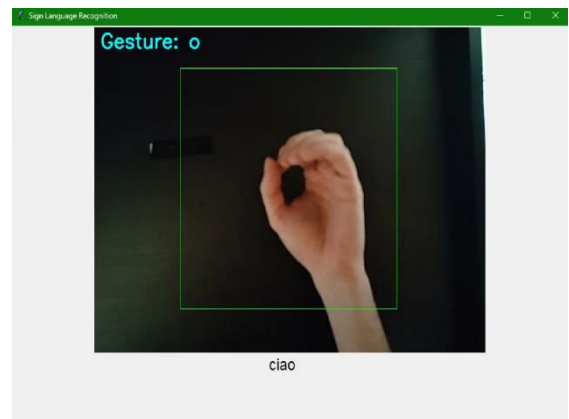
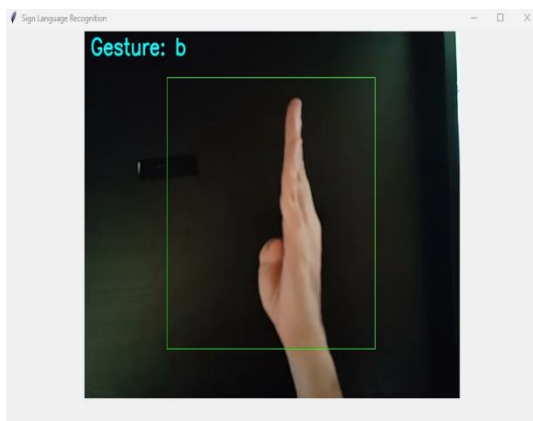
Ogni immagine acquisita viene successivamente processata tramite un ridimensionamento a 64x64 e convertita a scala di grigi. L'immagine viene poi data in input al modello, che visualizzerà a schermo la lettera riconosciuta.

L'utente ha altresì la capacità di salvare le lettere individuate, componendo parole e frasi dalle singole lettere, premendo la barra spaziatrice e salvandole nella porzione inferiore della finestra.

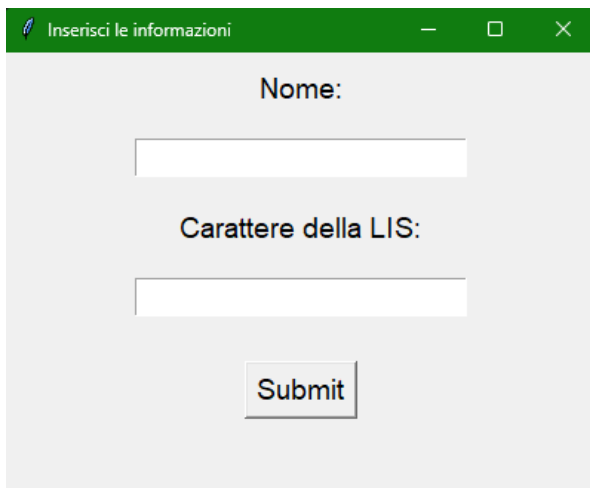
Le lettere possono anche essere cancellate con backspace, in caso di errore.

Infine, per uscire dall'applicazione basta premere il tasto ESC.

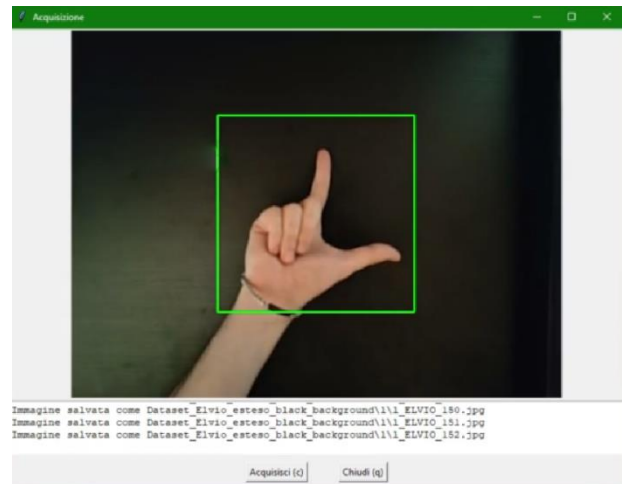
Esiste anche una versione alternativa non-real-time, dove è l'utente a decidere quando mandare l'immagine al modello, e il riconoscimento avviene successivamente. Questa modalità offre un'esperienza talvolta più stabile, permettendo all'utente di inviare immagini una per volta.



Infine, il terzo e ultimo software è stato progettato per la creazione veloce di dataset. Essendo la fase di acquisizione dei dati particolarmente importante per ottenere prestazioni ottimali dal modello, e al contempo potenzialmente difficoltosa, abbiamo sviluppato un applicativo per ottimizzare questo processo.



The screenshot shows a window titled "Inserisci le informazioni" with a green header bar. It contains two text input fields: the first is labeled "Nome:" and the second is labeled "Carattere della LIS:". Below the second field is a "Submit" button.



L'interfaccia utente, creata con Tkinter, consente agli utenti di inserire il proprio nome e il carattere della LIS da acquisire tramite una finestra di dialogo.

La finestra principale mostra il feed della webcam, permettendo di catturare immagini con un semplice clic. Le immagini vengono ritagliate a una dimensione di 256x256 pixel e poi ridimensionate a 64x64 pixel prima di essere salvate.

Il programma assegna automaticamente un numero incrementale alle immagini per evitare sovrascritture e assicurare l'unicità dei file, con dei controlli di validità sull'input utente.

La cartella per il dataset viene creata automaticamente, organizzando le immagini per lettera della LIS. Il feed della webcam è mostrato in tempo reale, con un riquadro verde che indica l'area di cattura. Gli utenti possono usare pulsanti o tasti di scelta rapida per catturare immagini o chiudere il software.

CODICE

Il progetto e il codice sono strutturati secondo la seguente **folder hierarchy**:

Struttura del Codice

LIS-Project-Final

- **DataAcquisition**
 - PythonProject
 - Dataset_Elvio_esteso_black_background
 - Dataset_Manuel
 - Dataset_michele_esteso
 - *DataAcquisitionProgram.py*
- **model**
 - *model_def_new.py*
 - *model_def_v1.py*
 - *model_mnist.py*
- **TrainedModels**
 - *SignLanguage_DeepCNN_WithDropout_WithDA_LessParams_lr_001_mom99_ep20_elvio*
 - *SimpleCNN_WithDropout_WithDA_lr_001_mom99_ep20_elvio*
- DemoVideoML.mp4
- requirements.txt
- *signRecognizer_main.py*
- *signRecognizer_offline.py*
- *training_pipeline.ipynb*

Descrivendo la struttura seguendo un approccio top-down:

Le cartelle di **DataAcquisition** contengono i 3 dataset utilizzati (per un totale di circa 70000 immagini), il programma Python per la loro creazione, e il Dataset_Completo.zip che li racchiude tutti.

La cartella **model** contiene le definizioni dei modelli descritti in precedenza. In particolare:

- *model_mnist.py* contiene il modello MNIST 28x28, il primo creato nel progetto.
- *model_def_new.py* contiene **SimpleCNN**
- *model_def_v1.py* contiene **SignLanguage_DeepCNN**, la rete finale del progetto.

La cartella **TrainedModels** contiene i file .pt, ossia i modelli addestrati e pronti all'uso dagli applicativi user-facing.

DemoVideoML.mp4 è un video-tutorial e informativo sull'uso degli applicativi.

requirements.txt serve appunto ad installare i pacchetti richiesti dal progetto per il corretto funzionamento.

signRecognizer_main.py è il cuore del progetto, è il file main da eseguire per utilizzare in real-time i modelli addestrati.

signRecognizer_offline.py è la versione non in real-time del main descritto pocanzi, l'acquisizione dell'immagine e del gesto raffigurato viene eseguita solo dopo uno user-input esplicito.

Infine, **training_pipeline.ipynb** è un Jupyter Notebook che esegue tutte le fasi di caricamento del dataset, training, testing, logging con SummaryWriter, e salvataggio del modello addestrato. Con minimi cambiamenti permette anche di addestrare più modelli cambiando automaticamente i parametri (con criteri user-defined), e anche con refresh automatico del modello sul device.

Esecuzione del codice

Applicazione

1. Creare un virtual environment .venv;
2. Installare i requirements con pip;
3. Eseguire signRecognizer_main.py
 - a. NB: Assicurarsi di avere un dispositivo webcam o simile collegato.

Training

1. Installare tutti i requirements come descritto prima
2. Eseguire il file Jupyter Notebook **training_pipeline.ipynb**.
 - a. Cambiare modello, nomi e parametri all'occorrenza. Il modello viene salvato automaticamente col nome dell'esperimento e dei parametri assegnato dall'utente (che a sua volta corrisponde con i log su TensorBoard)

CONCLUSIONI

Il task affrontato nel progetto è stato la realizzazione di una CNN per il riconoscimento dei caratteri alfabetici della lingua dei segni.

L'esplorazione e la realizzazione di una soluzione per il task richiesto, ha sicuramente consentito di approfondire ulteriormente aspetti sia pratici che teorici.

In primis, è stata messa in pratica la formalizzazione di una pipeline di Machine Learning, ed è stato fondamentale definire ogni step adeguatamente prima di poter passare a quello successivo.

Proprio queste fasi hanno portato inizialmente all'esplorazione del task tramite un dataset dello stato dell'arte (Sign Language MNIST), passando poi alla definizione di un dataset custom che risolvesse i limiti del primo.

Il lavoro centrale ha riguardato la definizione dei modelli CNN che meglio si adattassero al task ed è la fase che ha rivelato anche le criticità maggiori.

In particolare, cercare di capire come migliorare il modello CNN ha rappresentato la sfida maggiore. Ad esempio, nel capire come risolvere il problema di accuracy alta e loss alta del secondo modello. Oppure ancora il problema di indistinguibilità tra coppie di lettere simili (I e Y, N e M, O e C).

Possibili miglioramenti

Nonostante i buoni risultati ottenuti, il task non è risolto alla perfezione e il modello commette degli errori di classificazione, specialmente nel caso di sfondi irregolari che non permettono una netta distinzione tra la mano e l'ambiente.

Alla luce di tale considerazione, si potrebbe pensare ad una soluzione che implementi segmentazione della mano, integrando in un'unica CNN o definendo due modelli separati (quest'ultimo approccio probabilmente non sarebbe ottimale per un'applicazione real-time).

Altra considerazione riguarda il problema della scala. Ottenere un dataset che includa ogni possibile livello di scala è poco pratico.

Ciò suggerisce che una soluzione con Object Detection possa fare al caso per questo genere di task, automatizzando anche la fase di individuazione delle mani (invece di utilizzare una finestra fixed) e classificare il gesto come sotto task.