Image to Text

Progetto di Machine Learning

Anno Accademico 2024-2025

Matteo Galletta, Marco Gionfriddo, Kevin Speranza

Indice

Capitolo 1

Problema

Da anni ormai si tenta di risolvere il problema del riconoscimento di testi contenuti in immagine. Il problema è noto come Optical Character Recognition (OCR) e consiste nel riconoscere i caratteri di un testo contenuto in un'immagine. Il problema è complesso e presenta una serie di insidie che non sono di immediata risoluzione. Nonostante ciò, allo stato dell'arte esistono diversi algoritmi che consentono di ottenere risultati soddisfacenti. Quello che viene presentato in questo documento è un modello che mira a semplificare il problema a una sottoclasse di immagini, avendo il vantaggio di ottenere un algoritmo più leggero ed efficiente, a discapito della sua versatilità.

Capita spesso che le immagini da cui è utile estrarre il testo siano screenshot. L'algoritmo presentato si occupa di estrarre il testo contenuto in uno screenshot, indipendentemente dal font e dai colori utilizzati. In realtà, viene inizialmente affrontato il problema assumendo che lo screenshot comprenda una sola parola. Il problema viene ulteriormente semplificato ai font in stampatello e agli alfabeti italiano e latino esteso (punteggiatura compresa), escludendo il corsivo e altri alfabeti. Tramite l'uso di euristiche, si può estendere facilmente l'implementazione comprendendo frasi (purché non siano divise su più righe).

Figura 1.1: Screenshot di esempio

Capitolo 2

Soluzione proposta

La soluzione proposta è suddivisa in due fasi principali:

- **Fase 1**: Suddivisione in caratteri
- **Fase 2**: Classificazione del carattere

Per la prima fase vengono utilizzati algoritmi di image processing per suddividere la parola in caratteri. Per la seconda fase viene utilizzato un modello di deep learning per classificare i singoli caratteri.

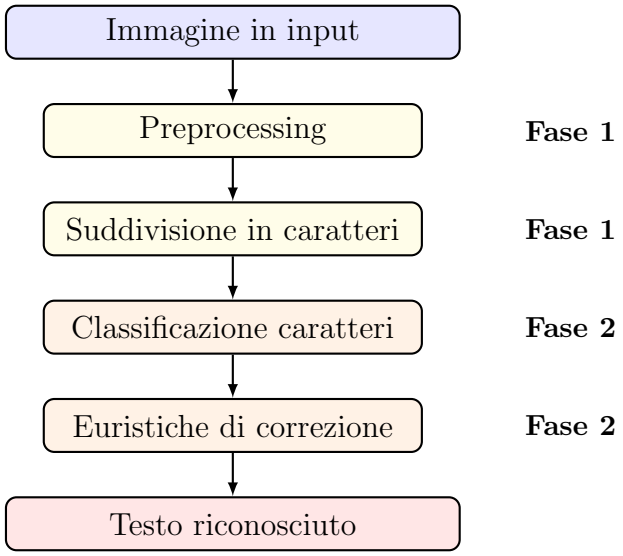


Figura 2.1: Schema della pipeline di riconoscimento

2.1 Fase 1: Suddivisione in caratteri

Prima di procedere alla suddivisione dell'immagine in singoli caratteri, vengono eseguite alcune operazioni di pre-processing.

Innanzitutto, l'immagine viene convertita in scala di grigi, così da operare su un unico canale. Successivamente, l'immagine viene normalizzata nell'intervallo 0-255, aumentando così il contrasto tra le aree chiare e scure, migliorando la visibilità dei dettagli. Nel passaggio successivo viene calcolata l'intensità media dei pixel, utile per stimare la luminosità complessiva dell'immagine. Se tale valore supera una soglia prefissata, si assume che l'immagine abbia uno sfondo chiaro; in tal caso, viene applicata un'inversione dei colori, trasformando i pixel chiari in scuri e viceversa. Questa procedura è particolarmente utile perché la rete neurale utilizzata è stata addestrata su immagini con testo bianco su sfondo nero; l'euristica basata sull'intensità media permette quindi di invertire automaticamente i colori, se necessario, per uniformare l'input al formato atteso dalla rete.

Infine, al solo obiettivo di suddividere i caratteri, l'immagine viene convertita in formato binario: attraverso un'operazione di thresholding, i pixel vengono trasformati in nero (0) o bianco (255). Al modello di classificazione verrà poi fornita l'immagine in scala di grigi.

Figura 2.2: Immagine dopo il Preprocessing.

Il primo approccio utilizzato per la suddivisione in caratteri è stato quello di considerare le proiezioni verticali dell'immagine. Come prima cosa si individuano le colonne in cui è presente almeno un pixel bianco. L'euristica quindi considera due colonne consecutive come appartenenti allo stesso carattere se presentano entrambe almeno un pixel bianco. Nonostante questo approccio possa sembrare ragionevole, gli esperimenti effettuati mostrano non essere efficace per immagini a bassa risoluzione. Infatti, in questo caso, i caratteri tendono a sovrapporsi e le colonne consecutive presentano pixel bianchi in comune. Per questo motivo, si è deciso di utilizzare un approccio alternativo.

L'approccio adottato è quello di considerare le componenti connesse bianche dell'immagine. Questo metodo è più efficace, consentendo di individuare più facilmente caratteri diversi, anche se parzialmente sovrapposti orizzontal-

mente. L'algoritmo non consente di individuare correttamente i caratteri non contigui, come nel caso di 'i' e 'j', che presentano un punto sopra il corpo del carattere. Un'ulteriore euristica risolve il problema in maniera efficace, andando a unire due componenti connesse se parzialmente sovrapposte orizzontalmente. Nello specifico, si prende in considerazione la componente connessa più piccola in larghezza e la si confronta con la parte in sovrapposizione con l'altra componente connessa. Se più del 30% (valore verificato sperimentalmente) della larghezza è sovrapposto, allora le componenti vengono unite. In questo modo, si riesce a ottenere un'immagine in cui ogni carattere è rappresentato da una singola componente connessa.

Una volta individuati i bounding box dei caratteri, si procede anche a calcolare un bounding box globale che racchiude tutti i caratteri. L'utilità di questo bounding box viene mostrata nella fase di classificazione.

Figura 2.3: Individuazione Bounding Box

2.2 Fase 2: Classificazione del carattere

Per la classificazione del carattere viene utilizzato un modello di deep learning. Il modello è stato addestrato su un dataset di immagini di caratteri e simboli in stampatello, con una risoluzione di 28x28 pixel. È quindi necessario ridimensionare i caratteri estratti dalla fase 1 prima di applicare l'inferenza. Il dataset viene approfondito nella sezione dedicata.

Fornendo al modello esclusivamente il carattere ridimensionato, di quest'ultimo verrebbero ignorate la dimensione e la posizione all'interno della parola. Questa semplificazione causerebbe problemi nella classificazione della punteggiatura e di caratteri *confondibili*.

Senza informazioni sulla posizione, il modello non sarebbe in grado di distinguere tra ',' e ''. Inoltre, non sarebbe in grado di distinguere tra maiuscole e minuscole *confondibili*.

Per carattere *confondibile* si intende una lettera in cui la rappresentazione in stampatello minuscolo coincide con quella in stampatello maiuscolo, se ridimensionata. Ad esempio, 'C' e 'c' sono caratteri confondibili, così come 'S' e 's', mentre 'A' e 'a' non lo sono. L'insieme dei caratteri confondibili

maiuscoli (CI) è definito come segue:

$$\text{CI} = \{C, J, K, O, P, S, U, V, W, X, Z\}$$

Ovviamente la controparte minuscola contiene gli stessi caratteri.

2.2.1 Utilizzo del Bounding Box globale

È possibile utilizzare il bounding box globale per fornire al modello informazioni sulla posizione e la dimensione del carattere all'interno della parola. Partendo dal bounding box del carattere e da quello globale, è possibile estrarre il margine superiore e inferiore del carattere rispetto al bounding box globale. Una volta normalizzati rispetto all'altezza del bounding box globale, il margine superiore e inferiore del carattere possono essere utilizzati come due parametri aggiuntivi per il modello.

Con questo accorgimento, il modello è adesso in grado distinguere tra ',' e ''. Inoltre, nel caso della parola 'Bob', è in grado di classificare correttamente la 'o'. Questo è possibile in quanto il margine superiore della 'o' è solo presente nel caso in cui il carattere sia maiuscolo.

2.2.2 La necessità di euristiche per il *case*

Nonostante quest'ultimo approccio possa sembrare efficace, non è sempre in grado di distinguere tra maiuscole e minuscole. Mostriamo il motivo attraverso un esempio e lo formalizziamo successivamente. Consideriamo due parole d'esempio:

- 'Cocco': la prima lettera non ha margine superiore e inferiore, e deve essere classificata come maiuscola.
- 'cocco': la prima lettera non ha margine superiore e inferiore, e deve essere classificata come minuscola.

Il modello non è quindi in grado di classificare correttamente i caratteri confondibili quando hanno la stessa altezza del bounding box globale. È necessario utilizzare un'euristica che, confrontando l'altezza dei vari caratteri, sia in grado di 'correggere' la forma maiuscola o minuscola del carattere.

Guardando la distribuzione dei caratteri rispetto al loro margine superiore, è possibile notare quando è possibile classificare con certezza i caratteri confondibili.

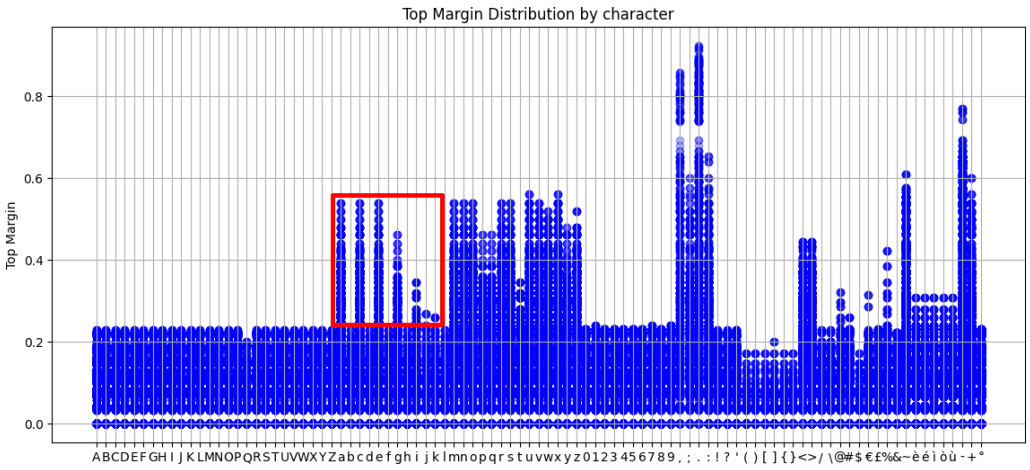


Figura 2.4: Distribuzione dei caratteri rispetto al margine superiore. Il rettangolo rosso evidenzia i caratteri confondibili identificabili con affidabilità.

Rimane comunque il fatto che parole come 'COCCO' e 'cocco' rimangono indistinguibili anche all'occhio umano, se non affiancate da altre parole che possano disambiguarne il *case*.

2.2.3 L'applicazione dell'euristica per il *case*

Data una determinata immagine, per carattere *affidabile* intendiamo un carattere non confondibile oppure un carattere confondibile di forma minuscola con margine superiore *significativo* (osservando la Figura 2.4 si può notare come si aggiri tra il 20% e il 25%). Un carattere è quindi affidabile quando la sua interpretazione agli occhi del modello è priva di ambiguità.

L'euristica per la correzione di maiuscole e minuscole può essere applicata solo quando è presente un carattere confondibile la cui altezza coincide con quella del bounding box globale. In questo caso, l'euristica confronta l'altezza del carattere con quella degli altri caratteri affidabili della parola. L'unico caso in cui l'altezza di un carattere confondibile coincide con quella del bounding box globale è quando non sono presenti caratteri che aumentano l'altezza del bounding box globale. Tra questi sono inclusi tutti i caratteri maiuscoli e qualche carattere minuscolo, come 'b', 'd' e 'h'.

Ci si potrebbe limitare ad applicare l'euristica nei casi sopra applicati. Purtroppo però, i concetti sopra esposti nella pratica non funzionano. Questo avviene in quanto in ogni font i caratteri minuscoli hanno una proporzione di altezza diversa rispetto ai caratteri maiuscoli. Per questo motivo, l'euristica viene applicata sempre, se possibile, ovvero se è presente almeno un carattere affidabile.

2.2.4 Spazi e classi aggiuntive

La fase 1, utilizzando le componenti connesse, non è in grado di estrarre direttamente gli spazi. Per poterli individuare, è necessario utilizzare delle euristiche aggiuntive. Si considera il margine tra i vari caratteri e sulla base di questo si decide se aggiungere uno spazio o meno. In particolare, viene inserito uno spazio se la distanza tra due caratteri è superiore al 60% della differenza tra la distanza massima e la minima tra i caratteri. Sperimentalmente, questo valore si è rilevato efficace per frasi di senso compiuto in lingua italiana. Tuttavia, per sequenze di caratteri generate randomicamente, il risultato ottenuto è piuttosto deludente, come approfondito nella sezione dedicata alla valutazione.

Inoltre, le virgolette, essendo composte da due componenti connesse, vengono classificate artificialmente andando a unire una sequenza di due apostrofi consecutivi.

Capitolo 3

Dataset

Essendo il problema dell'OCR uno dei più studiati in ambito di Computer Vision, esistono diversi dataset pubblici che possono essere utilizzati per addestrare e testare i modelli. Tuttavia, la maggior parte di questi dataset sono stati creati per risolvere problemi generali e non sono specifici per il riconoscimento di testi contenuti in screenshot. Per questo motivo, è stato necessario creare una coppia di dataset ad hoc per il problema in questione.

In particolare, essendo l'algoritmo diviso in due fasi, avere due dataset distinti consente di poter valutare in modo individuale ognuna delle due fasi, consentendo di valutare l'accuratezza del modello in modo più preciso.

I due dataset creati sono i seguenti:

- **Dataset dei simboli singoli**: contiene immagini di singoli simboli, che consente di testare esclusivamente la fase di classificazione del singolo carattere.
- **Dataset degli screenshot**: contiene immagini di screenshot contenenti una sequenza di simboli casuali. Questo consente di testare la pipeline nella sua interezza, comprendendo sia la suddivisione in caratteri che la classificazione del singolo carattere.

3.1 Dataset dei simboli singoli

Questo dataset è utilizzato per addestrare il modello di classificazione del singolo carattere. Viene inoltre utilizzato per estrarre delle metriche di valutazione della fase di classificazione, in modo da poter valutare l'accuratezza del modello.

I caratteri considerati nel dataset comprendono quelli dell'alfabeto latino in stampatello maiuscolo e minuscolo, le cifre da 0 a 9 e i seguenti simboli di punteggiatura:

, ; . : ! ? ’ ( ) [ ] { } < > / \ @ # $ € £ % & ˜ à è é ì ò ù - + °

In totale, il dataset contiene 96 classi.

Sono stati presi in considerazione 58 font differenti, ognuno con caratteristiche diverse (ad esempio, grassetto, sottile, italic, ecc.).

È stata generata un'immagine per ogni simbolo, per ogni font. La dimensione considerata è di 28x28 pixel, in modo da essere compatibile con il modello di classificazione utilizzato nella fase 2.

Per la generazione di ogni immagine è stato utilizzato Pillow, una libreria Python per la manipolazione delle immagini. Ogni immagine è stata generata con uno sfondo nero e il simbolo in bianco (con scala di grigi per i bordi). Per come vengono estratti i caratteri dall'immagine da classificare, è necessario che il simbolo nel dataset non abbia margini e che il suo bounding box coincida con il quadro dell'immagine. Per garantire questo vincolo, l'immagine viene inizialmente generata in un'immagine di dimensioni più grandi, successivamente il simbolo viene centrato all'interno dell'immagine e stampato con una dimensione del font arbitraria (grande abbastanza da far entrare il simbolo nel quadro). Dall'immagine viene dunque estratto un bounding box che viene poi ridimenzionato, in modo da ottenere un'immagine di dimensioni 28x28 pixel.

Figura 3.1: Esempio di simboli singoli del dataset.

3.1.1 Permutazioni di margini

Come menzionato precedentemente, il modello di classificazione utilizza, oltre al bounding box del simbolo, anche i margini superiore e inferiore del simbolo rispetto al bounding box globale dell'intera parola. Per questo motivo è necessario che il dataset contenga questa informazione per far sì che il modello possa apprendere le relazioni tra il simbolo e il bounding box globale.

L'approccio utilizzato è il seguente, per ogni font:

- Si scelgono dimensione di font e quadro dell'immagine arbitrari, font sufficientemente piccolo e quadro abbastanza grande da consentire a qualsiasi simbolo stampato di entrare all'interno dell'immagine.
- Si genera un'immagine quadrata per ogni simbolo, con il simbolo centrato all'interno dell'immagine.
- Si calcolano, per ogni simbolo, i margini superiore e inferiore del simbolo rispetto al quadro dell'immagine.
- Si normalizzano i margini rispetto all'altezza del quadro dell'immagine, in modo da ottenere valori compresi tra 0 e 1.
- Vengono create delle permutazioni dei margini, in modo che nel dataset sia presente ogni possibile combinazione di margini per ogni simbolo.

Il dataset finale conterrà quindi un simbolo per ogni font, con le varie combinazioni di margini. Nel caso dei font considerati, il dataset contiene un totale di 179292 tuple.

È importante notare che a causa di questa ultima aggiunta il dataset dei simboli singoli non rimane perfettamente bilanciato: non tutte le classi (simboli) sono rappresentate dallo stesso numero di campioni. Ad esempio, le parentesi tonde (`(` e `)`) sono presenti con circa 600 campioni ciascuna, mentre il trattino (−) conta circa 5000 campioni.

Il dataset è stato suddiviso in due sottoinsiemi: train e test. La suddivisione è stata effettuata secondo una proporzione 75% per il training e 25% per il test, necessario per la valutazione delle prestazioni del modello.

3.1.2 Struttura del dataset

Il dataset si compone di una serie di cartelle, una per ogni font, contenenti le immagini dei simboli. Ogni cartella è denominata con il nome del font e contiene le immagini dei simboli in formato PNG. I nomi delle immagini contengono il nome del simbolo rappresentato. Sullo stesso livello della cartella del font, sono presenti due file CSV: uno per il training e uno per il test. Questi file contengono l'elenco dei campioni, con la seguente struttura:

(font, immagine, simbolo, margine_superiore (%), margine_inferiore (%))

3.2 Dataset degli screenshot

Il dataset degli screenshot è stato creato per testare l'intera pipeline, dalla suddivisione in caratteri alla classificazione del singolo carattere. Il suo unico

scopo è quello di testare l'accuratezza della pipeline per poterne valutare le prestazioni.

Il dataset contiene immagini di screenshot contenenti una sequenza di 10 simboli casuali. In aggiunta ai simboli considerati nel dataset dei simboli singoli, sono stati aggiunti i caratteri "spazio" e "virgolette" (‘"‘), in modo da poter testare le euristiche relative al riconoscimento di questi caratteri.

Per la generazione di ogni immagine viene utilizzato un approccio simile a quello utilizzato per il dataset dei simboli singoli. Viene utilizzato Pillow per generare gli screenshot sintetici, scegliando un font casuale e stampando al suo interno la sequenza di simboli da testare. Per aumentare la precisione delle metriche, vengono inoltre selezionati dei colori casuali per il testo e lo sfondo, in modo da simulare screenshot reali e verificare che le operazioni di pre-processing siano efficaci.

Sono state previste due varianti del dataset, in modo da poter testare la pipeline in modo più completo e verificare che le euristiche implementate funzionino correttamente.

- **Sequenze di 10 simboli casuali**: prevede immagini di sequenze di 10 caratteri estratti randomicamente e concatenati per formare una parola (o una frase, considerando che lo spazio è incluso tra i simboli stampati).

- **Frasi di senso compiuto**: contiene immagini di screenshot di frasi di senso compiuto, estratte da un dataset pubblico di citazioni in lingua inglese. In particolare, le frasi sono state selezionate da un dataset pubblico disponibile su Hugging Face [1]

[1] Abir ELTAIEF, *english_quotes (Revision 7b544c4)*, 2023. Disponibile su: `https://huggingface.co/datasets/Abirate/english_quotes`, DOI: 10.57967/hf/1053.

Capitolo 4

Modello di Classificazione

Per la classificazione dei caratteri estratti dalla fase 1 è stato sviluppato un modello di deep learning. La rete è una Convolutional Neural Network, simile all'architettura LeNet-5: è composta da 2 layer convoluzionali, ognuno dei quali è seguito da un layer di pooling e da una funzione di attivazione ReLU. Dopo i layer convoluzionali, la rete è composta da tre layer fully connected, che producono l'output finale. Per evitare l'overfitting riscontrato sperimentalmente durante l'addestramento, viene utilizzato il dropout tra i layer fully connected.

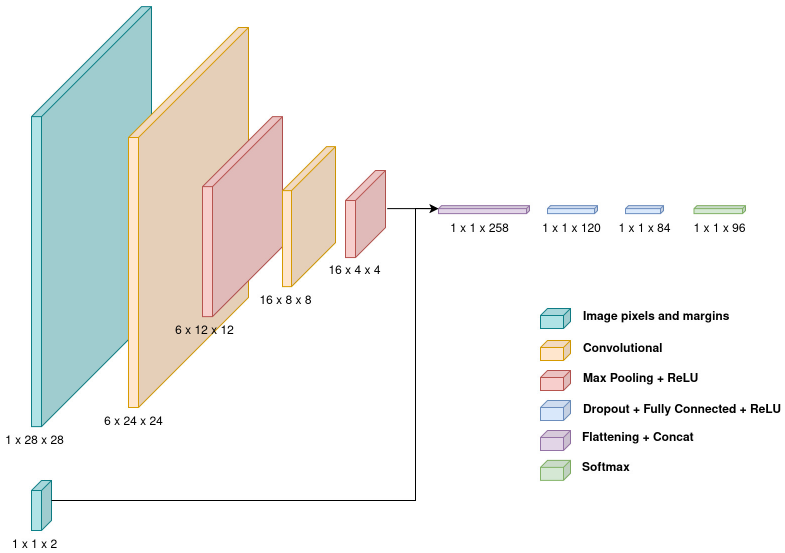
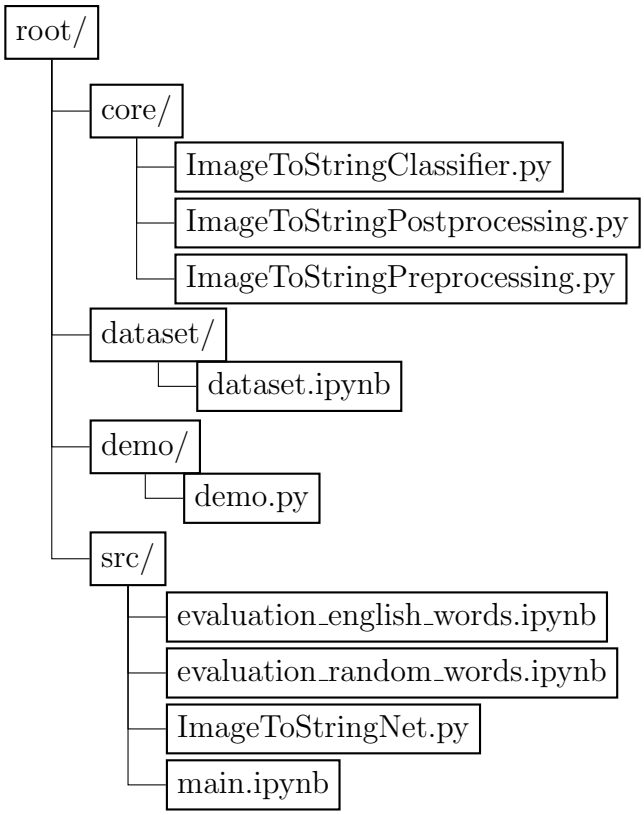


Figura 4.1: Schema dell'architettura della rete neurale utilizzata.

Capitolo 5

Codice

Il codice del progetto è suddiviso in quattro cartelle ciascuna delle quali si occupa di un aspetto specifico del flusso. Di seguito è riportata la struttura delle cartelle e dei file principali, per avere una panoramica dell'organizzazione del progetto.



5.1 Core

La cartella *core* contiene il codice delle tre classi dedicate a preprocessing, classificazione e postprocessing. Ciascun file definisce una classe omonima.

5.1.1 PreProcessing

La classe `ImageToStringPreprocessing` prepara l'immagine contenente testo per la fase successiva di classificazione, segmentando le lettere e normalizzandole in un formato uniforme. A partire da un'immagine in input, esegue operazioni come conversione in scala di grigi, binarizzazione e inversione del contrasto se necessario. Successivamente, rileva e raggruppa le componenti connesse per identificare le singole lettere, calcolando anche informazioni spaziali come distanze relative e disegnando le relative bounding box sull'immagine originale. Ogni lettera viene poi ritagliata, ridimensionata proporzionalmente e centrata su un'immagine nera 28x28, rendendola pronta per le fasi successive. La classe inoltre fornisce metodi per accedere all'immagine segmentata, alle lettere preprocessate e alla loro visualizzazione.

5.1.2 Classificazione

A partire da un'immagine contenente una sequenza di caratteri, la classe `ImageToStringClassifier` gestisce l'intero processo di riconoscimento integrando preprocessing, postprocessing e `ImageToStringNet` per la classificazione.

5.1.3 PostProcessing

La classe `ImageToStringPostprocessing` a partire dalla lista delle lettere classificate con relative informazioni spaziali, applica le euristiche discusse nei capitoli precedenti per decidere dove inserire spazi tra parole, basandosi sulle distanze orizzontali tra i caratteri. Inoltre, sfrutta la posizione verticale delle lettere rispetto ai bounding box generale per correggere l'uso errato delle maiuscole e minuscole in caratteri ambigui, confrontando ciascun carattere incerto con il primo considerato affidabile. Il risultato è una sequenza di caratteri più coerente, utile per migliorare l'output finale del sistema di OCR.

5.2 Dataset

La cartella *dataset* contiene il notebook `dataset.ipynb`, in cui sono descritte e implementate tutte le procedure necessarie per la creazione dei dataset dei simboli singoli e degli screenshot.

5.2.1 Dataset dei simboli singoli

Nella prima parte del notebook, sono definite le funzioni per la generazione automatica delle immagini dei caratteri che permettono di creare immagini sintetiche di lettere, al variare di font e margini. Segue una fase di normalizzazione delle immagini, in cui ciascuna immagine viene convertita in scala di grigi, ridimensionata e centrata su uno sfondo uniforme, 28x28 pixel. Il notebook include le procedure per la suddivisione del dataset in insiemi di training e test, distribuiti rispettivamente in 75% e 25%. Viene definito il salvataggio dei dati in formato compatibile con PyTorch (`torch.utils.data.Dataset`) e come importarli rapidamente nei notebook di addestramento e valutazione.

5.2.2 Dataset degli screenshot

Nella parte finale del notebook è presente la funzione incaricata di generare il dataset degli screenshot, nelle sue due varianti. Una prima cella la richiama per generare immagini contenenti sequenze di 10 caratteri casuali, mentre una seconda cella la utilizza per generare frasi di senso compiuto, selezionate da un dataset di citazioni. In entrambi i casi, per ogni font presente nella lista fornita, vengono create 100 immagini con uno sfondo di un colore scelto casualmente. Per garantire una buona leggibilità del testo, viene calcolata la luminosità dello sfondo e, in base a essa, viene determinato se usare testo bianco o nero. Dopo aver centrato il testo all'interno dell'immagine, quest'ultima viene salvata all'interno della directory corrispondente al font, utilizzando un nome univoco generato automaticamente.

5.3 Demo

La cartella *demo* contiene il file `demo.py`, che fornisce l'interfaccia semplice per testare il modello descritta nel Capitolo 7. Lo script permette di caricare un'immagine, eseguire il preprocessing, la classificazione e visualizzare il risultato finale.

5.4 Src

La cartella *src* contiene i file principali per l'addestramento, la valutazione e l'esecuzione del modello. In particolare:

- `main.ipynb`
- `ImageToStringNet.py`
- `evaluation_english_words.ipynb`
- `evaluation_random_words.ipynb`

5.4.1 main.ipynb

Il notebook `main.ipynb` implementa l'intero workflow di addestramento e valutazione del modello. Viene definita la classe `DigitDataset` che consente di gestire il dataset facilmente, dopo opportune trasformazioni, all'interno della pipeline di addestramento e valutazione in PyTorch.

Una volta impostata la dimensione del batch per l'addestramento, vengono definite le trasformazioni da applicare alle immagini ovvero conversione in tensori e normalizzazione. Viene suddiviso il dataset in training, validation e test separando i font, garantendo che quelli utilizzati per il test non siano mai presenti nell'addestramento. I dati vengono poi caricati in memoria tramite i `DataLoader` di PyTorch, che gestiscono automaticamente il batching e lo shuffle. Viene inizializzata la rete neurale implementata in `ImageToStringNet.py` e definita la funzione di loss Cross Entropy e l'ottimizzatore SGD. Viene inoltre utilizzato un modulo di logger per TensorBoard, per monitorare visivamente un batch di immagini, salvare la struttura del modello, le embedding delle immagini, la loss e l'accuracy.

Per l'addestramento, il flusso è diviso in varie fasi:

- **Gestione delle configurazioni:** per ogni configurazione viene avviata un'esecuzione indipendente dell'intero ciclo di addestramento e valutazione.
- **Costruzione e configurazione del modello:** per ogni configurazione viene istanziata una nuova rete con il tasso di dropout specificato. Viene poi configurato un ottimizzatore SGD con i parametri previsti.
- **Logging dei risultati con Tensorboard:** viene creato un writer che registra l'andamento della loss e dell'accuracy durante la fase di addestramento e valutazione e una visualizzazione grafica delle predizioni rispetto le etichette reali.

- **Caricamento dei pesi preesistenti:** se esistono dei pesi per una determinata configurazione, vengono caricati prima dell'addestramento.
- **Ciclo di addestramento ed epoche:** il modello viene addestrato per un numero definito di epoche.
- **Salvataggio dei pesi:** al termine di ogni configurazione, i pesi aggiornati vengono salvati.

Le celle successive del notebook sono dedicate alla valutazione del modello, che viene effettuata solo dopo aver trovato i migliori parametri (`exps5`). Per ogni batch nel set di test, vengono calcolate e stampate la loss media, l'accuracy media, precision, recall e F1-score.

Dopo aver uniformato le etichette predette per renderle case-insensitive, si calcolano le curve precision-recall per ciascuna classe salvandole su TensorBoard. Infine viene calcolata la matrice di confusione per valutare la distribuzione degli errori del modello.

5.4.2 ImageToStringNet.py

Il file `ImageToStringNet.py` contiene la classe che implementa la rete neurale convoluzionale (CNN) utilizzata per la classificazione dei caratteri estratti, discussa nel Capitolo 4.

Il costruttore della classe `ImageToStringNet` definisce l'architettura della rete, ed è suddiviso in due moduli principali:

- **Feature Extractor**: implementa i due blocchi convoluzionali, dove ciascuno consiste di convoluzione, max pooling e ReLU.
- **Classifier**: prende in input le feature provenienti dal `feature_extractor` e le elabora attraverso la rete di tre layer fully connected dove i primi due utilizzano dropout per prevenire l'overfitting seguite da ReLU e l'ultimo layer di output mappa gli 84 neuroni finali al numero di classi possibili.

Nel metodo `forward`, viene definito il flusso dell'input attraverso la rete, l'immagine viene processata dal modulo `feature_extractor` restituendo un tensore che viene appiattito e vengono concatenati i due margini superiori e inferiori, infine viene passato al modulo `classifier` che produce l'output finale.

5.4.3 Notebook per Evaluation

I notebook `evaluation_english_words.ipynb` ed `evaluation_random_words.ipynb` sono dedicati alla valutazione delle prestazioni del modello sulle due varianti del dataset degli screenshot.

In ogni notebook, per ciascuna immagine del dataset associato si esegue il riconoscimento del testo. Il risultato viene confrontato con il testo atteso tramite la distanza di Levenshtein. I risultati ottenuti per ogni immagine vengono raccolti e usati per calcolare media, varianza, valore minimo e valore massimo. Per la valutazione dell'accuracy, viene definita: la precisione case-sensitive e case-insensitive per le lettere confondibili, versioni che ignorano gli spazi interni e le lettere come la "i" e la "l" maiuscole.

Capitolo 6

Esperimenti

Setup testing

Come anticipato precedentemente, il modello scelto ha un'architettura simile a quella di LeNet-5. Trattandosi di un task di classificazione vicino a quello del riconoscimento di cifre scritte a mano, la scelta di un'architettura ispirata a LeNet-5 è stata naturale, essendo consolidata per questo tipo di problemi. Sono state comunque effettuate delle modifiche rispetto alla classica architettura sopra citata, sia per adattarla alle features del nostro task di classificazione (che prevedono un paio di input in più), sia per poter ottenere risultati migliori in fase di training (aggiungendo un paio di layer di dropout).
Per poter ottenere la miglior combinazioni di iperparametri, sono stati effettuati diversi esperimenti, variando *learning rate*, *batch size*, *dropout rate*, *momentum* e *numero di epoche*. Per ognuna delle varianti negli iperparametri nella fase di training, viene generato un log TensorBoard che contiene le coppe di *Loss* e *Accuracy* in entrambi i dataset di training e validation. Inoltre, vengono salvati i pesi al termine delle epoche, utile per poterli ricaricare successivamente. Di seguito vengono evidenziati gli esperimenti effettuati.

6.1 Exps 1

Il primo esperimento prevede l'iterazione di una griglia di parametri presenti all'interno di un file di configurazione. In particolare, la griglia prevede tutte le triple dei seguenti parametri nei corrispettivi range:

- **Learning rate**: {0.01, 0.001, 0.005, 0.0001, 0.0005}
- **Dropout rate**: {0.2, 0.3, 0.4, 0.5}

- **Momentum**: {0, 0.5, 0.9}

Essendo un modello piuttosto piccolo, è stato ritenuto opportuno impostare una batch size di **2560**, garantendo un calcolo del gradiente più preciso comparato a una batch size piccola. Per semplicità, il numero di epoche è fissato a 50.

Questo approccio di *grid search*, per quanto semplice, potrebbe già restituire dei risultati approssimativi sui range più opportuni per risolvere il problema, rendendo possibile ulteriori esperimenti nei range vicini alla tripla migliore.

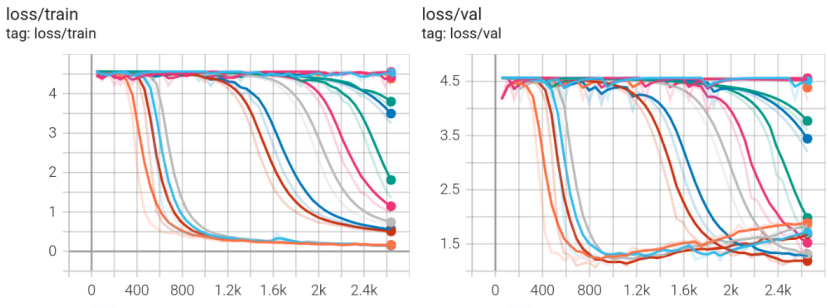


Figura 6.1: Loss Exps 1

Durante la fase di training si è notato come i tempi di training siano dilatati, rendendo questo approccio inefficiente. Inoltre, a colpo d'occhio, durante l'iterazione dei vari iperparametri, si è notato come le prestazioni del modello fossero parecchio scadenti. Qualche training terminava in overfitting evidente, mentre altri sembravano non convergere entro le 50 epoche.

Per queste ragioni, senza ultimare il training con tutte le permutazioni, si è preferito procedere per via iterativa, come approfondito nella sezione successiva.

6.2 Exps 2

La soluzione più efficiente per ottimizzare il flusso precedentemente configurato si è rilevato essere un processo iterativo con l'intervento umano che regola gli iperparametri più opportuni man mano che gli esperimenti avvengono.

Il secondo esperimento evidenzia come, nonostante venga variato il dropout, il fenomeno di overfitting rimanga persistente.

Questo è probabilmente dovuto alla batch size parecchio grande, non consentendo di avere un grado di regolarizzazione sufficiente alto.

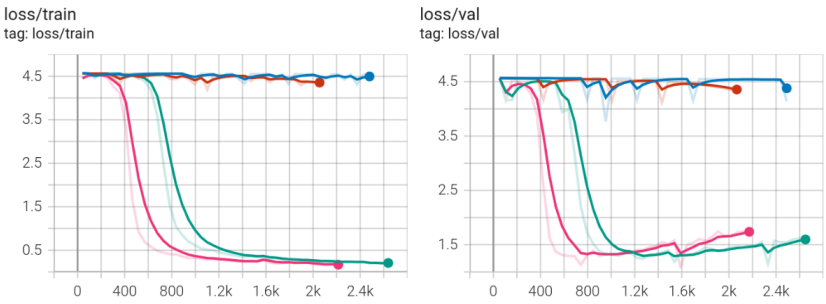


Figura 6.2: Loss Exps 2

6.3 Exps 3

Per ovviare il problema della precedente sperimentazione, si è deciso di ridurre la batch size a **256**, consentendo di aumentare la regolarizzazione del modello.

Inoltre, un ulteriore tentativo di migliorare il modello si è configurato nella scelta di andare a riutilizzare i migliori pesi man mano che i parametri vengono cambiati. Questo consente di utilizzare un learning rate più basso quando in prossimità del minimo locale.

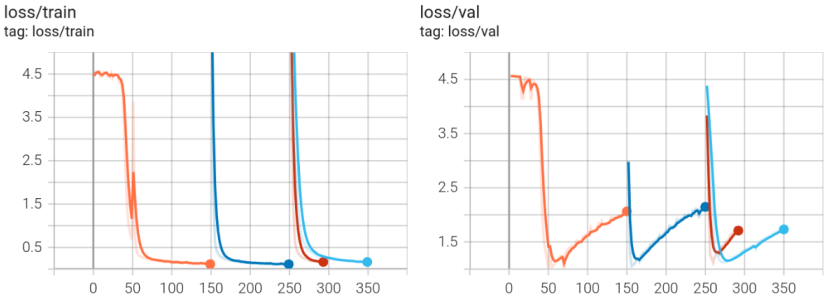


Figura 6.3: Loss Exps 3

I tentativi mostrano uno scarso risultato nel combattere la varianza esibita dalle curve di train e loss.

6.4 Exps 7

Ulteriori tentativi di miglioramento non hanno miglioramenti nella risoluzione del problema di overfitting. In compenso, al settimo esperimento, l'accuracy sul validation set è migliorata, raggiungendo un valore di circa l'82%. La

principale ragione di unoverfitting così evidente è probabilmente dovuta all'ambiguità dei dati e dalle inconsistente nel dataset che rendono impossibile, agli occhi del modello, la corretta distinzione tra i caratteri confondibili, come evidenziato nella sezione apposita. Per questo motivo, per individuare il termine del training ci si è basati sull'accuratezza del validation set, piuttosto che sulla curva di loss.

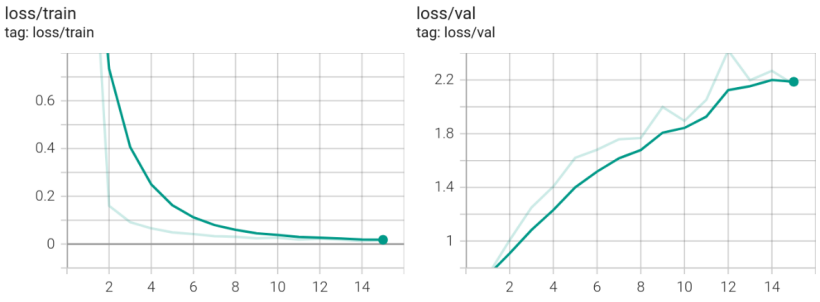


Figura 6.4: Loss Exps 7

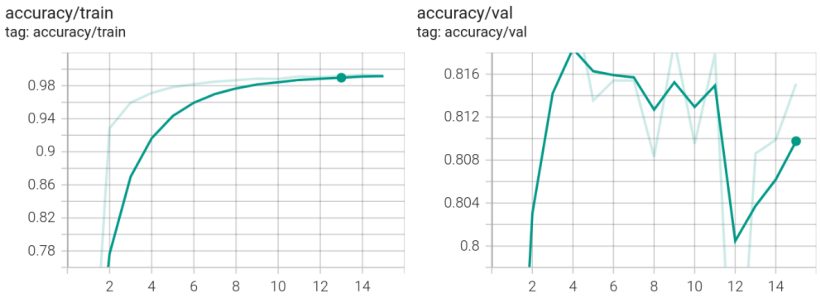


Figura 6.5: Accuracy Exps 7

Il modello finale prevede l'utilizzo di una **batch size di 16**, un **learning rate di 0.02**, un **dropout rate di 0.5** e un **momentum di 0.5**. Il numero di epoche è fissato a 15, ma il modello scelto ha mostrato il picco di accuratezza al termine della quarta epoca, prima che il fenomeno di overfitting si facesse evidente anche nella curva di accuracy.

6.5 Valutazioni

Una volta fissati i pesi ottimali del modello, è stata eseguita la valutazione delle prestazioni sul set di test. Per analizzare in modo più approfondito il comportamento del modello, la valutazione è stata condotta sia a livello di singolo carattere che a livello di stringa.

6.5.1 Valutazione caratteri

La valutazione sui singoli caratteri si articolano in diverse analisi:

- Analisi score
- Curve Precision-Recall
- Matrice di confusione

Analisi score

Per le migliori run di ciascun esperimento, oltre all'**accuracy** sono state calcolate anche le metriche di **precision**, **recall** e **F1-score**.

Esperimento	**Accuracy**	**Precision**	**Recall**	**F1-score**
1	0.85	0.82	0.80	0.81
2	0.86	0.83	0.82	0.82
3	0.88	0.85	0.84	0.84
4	0.89	0.86	0.85	0.85
5	**0.91**	**0.89**	**0.88**	**0.88**

Tabella 6.1: Metriche di valutazione sui cinque esperimenti.

Dall'analisi della tabella si può osservare come l'esperimento 5 abbia ottenuto le migliori prestazioni su tutte le metriche considerate.

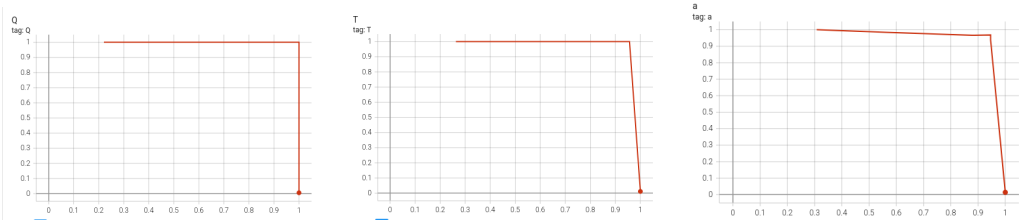
Curve Precision-Recall

Le curve Precision-Recall (PR) consentono di analizzare il bilanciamento tra *precision* e *recall* nelle predizioni del modello, mostrando quanto esso riesca a mantenere alta la precisione man mano che aumenta la quantità di caratteri correttamente riconosciuti. Un indicatore sintetico della qualità complessiva è l'area sotto la curva (AUC-PR), che risulta tanto più elevata quanto migliore è la capacità del modello di conciliare accuratezza e sensibilità nel riconoscimento dei caratteri.

In Figura 6.6 sono riportate le curve per un sottoinsieme rappresentativo di classi non confondibili.

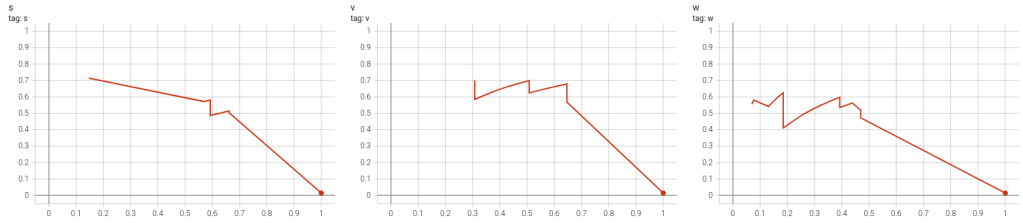
Nel complesso, il modello mostra buone prestazioni, con curve PR ampie e stabili per la maggior parte delle classi.

Tuttavia, alcune classi risultano più problematiche a causa della somiglianza con il case opposto.



(a) PR-curve {Q} (b) PR-curve {T} (c) PR-curve {a}

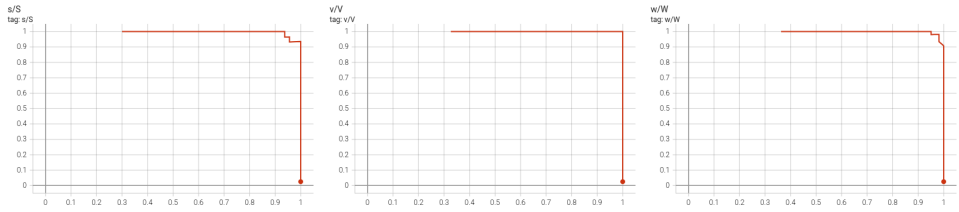
Figura 6.6: PR-curves per caratteri non confondibili



(a) PR-curve {s} (b) PR-curve {v} (c) PR-curve {w}

Figura 6.7: PR-curves per caratteri confondibili

Per valutare l'impatto della distinzione tra maiuscole e minuscole, è stata ripetuta l'analisi ignorando il case. Come mostrato in Figura 6.8, l'area sotto la curva migliora sensibilmente, suggerendo che una parte consistente degli errori è dovuta a una difficoltà del modello nel distinguere il case piuttosto che a un'incapacità di riconoscere la forma del carattere.



(a) PR-curve {s/S} (b) PR-curve {v/V} (c) PR-curve {w/W}

Figura 6.8: PR-curves ignorando il case

Un caso particolarmente significativo è riportato in Figura 6.9, dove, nonostante l'ignoramento del case, le prestazioni del modello risultano ancora insoddisfacenti.

Questo comportamento anomalo può essere attribuito a confusioni residue legate alla somiglianza visiva tra questo tipo di caratteri ed i numeri.

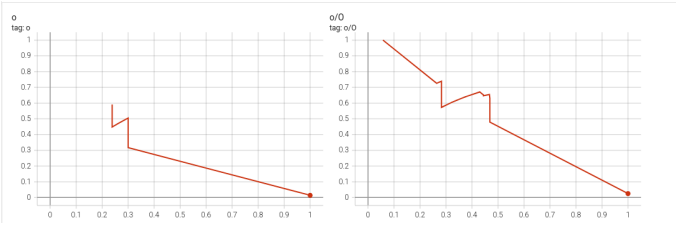


Figura 6.9: PR-curve per il carattere {o}

Matrice di Confusione

La matrice di confusione fornisce una rappresentazione dettagliata degli errori di classificazione commessi dal modello. Ogni cella (i, j) indica quante volte un carattere appartenente alla classe i è stato classificato come j.

(a) Standard (b) Case insensitive

Figura 6.10: Confronto tra le matrici di confusione

Come si può osservare in Figura 6.10(a), la diagonale principale è ben marcata, indicando che la maggior parte delle predizioni corrisponde correttamente alla classe attesa. Le deviazioni più significative dalla diagonale si concentrano principalmente tra le classi confondibili, spesso legate a differenze di maiuscole e minuscole.

Analizzando la matrice case-insensitive mostrata in Figura 6.10(b), si osserva una riduzione significativa degli errori di classificazione, confermando quanto discusso nella Sezione 6.5.1.

Nel repository del progetto sono disponibili le matrici di confusione completa sotto forma di report HTML.

6.5.2 Valutazione parole

Per l'analisi a livello di parola sono state adottate due metriche principali:

1. Distanza di edit: (Levenshtein distance[1]);
2. *String Accuracy.*

Le metriche sono state calcolate sui due dataset descritti nella Sezione 3.2.

Distanza di edit

La distanza di edit (o *Levenshtein distance*) rappresenta il numero minimo di operazioni elementari — inserimenti, cancellazioni o sostituzioni — necessarie per trasformare una stringa nella corrispondente stringa di riferimento (ground truth). Per rendere il confronto equo, il valore ottenuto è stato normalizzato rispetto alla lunghezza della parola di riferimento.

Di seguito si riportano le principali statistiche estratte.

Dataset	Min	Max	Mean	Std
Stringhe casuali	0	41	5.42	22.04
Stringhe inglesi	0	2195	57.22	27521.79

Tabella 6.2: Statistiche distanza di edit

Da questi risultati emergono alcuni comportamenti anomali del modello, in particolare una marcata difficoltà nell'elaborazione di font molto sottili. Come evidenziato in Figura 6.11, il modello fatica a rilevare correttamente i tratti dei caratteri sottili, principalmente a causa dei bounding box. I riquadri generati in questi casi limite risultano spesso mal posizionati o incompleti, compromettendo i dati in ingresso e ostacolando la corretta interpretazione del carattere.

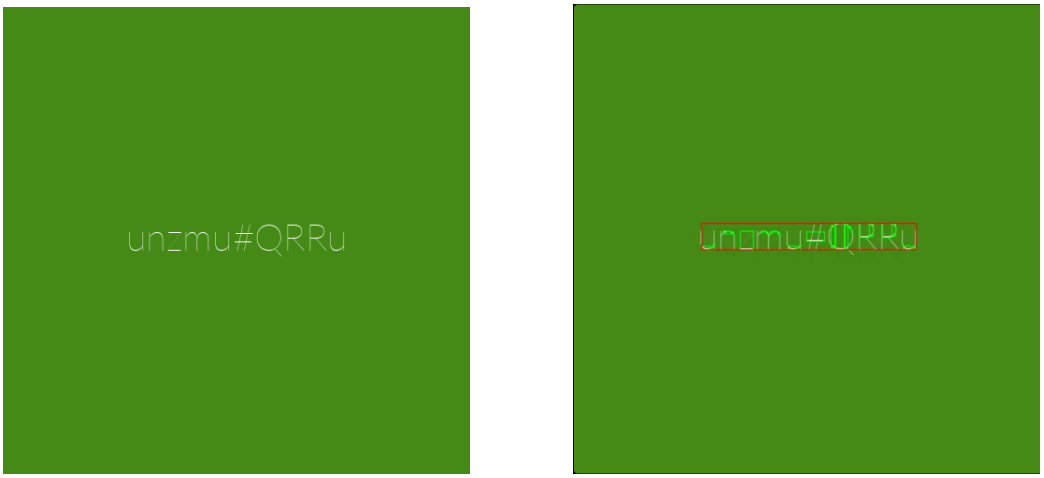
Ground Truth	**Output**
PEj%6DuZi8	[:. E-j :C::: ::: [::: é...Z. !d
unzmu#QRRu	é... eN.Z .-NP-N. é...@ :::\.. [: [: é...

Tabella 6.3: Esempi di riconoscimenti errati

String Accuracy

La string accuracy è definita come la percentuale di stringhe riconosciute esattamente nella loro interezza.

[1]V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, pp. 707-710, 1966.



(a) Immagine originale (b) Bounding box errato

Figura 6.11: Bounding box errati con font sottili

Per una valutazione più dettagliata, sono state considerate le seguenti varianti di string accuracy, che tengono conto di diverse esigenze di confronto:

- **Accuracy case sensitive (CS)**: confronto rigoroso che distingue tra maiuscole e minuscole;
- **Accuracy case insensitive (CI)**: confronto che ignora le differenze tra maiuscole e minuscole, utile per valutare la capacità di distinguere caratteri simili o confondibili;
- **Accuracy case sensitive senza spazi (CSNS)**: confronto che ignora gli spazi ma distingue tra maiuscole e minuscole, utile per valutare la capacità di riconoscimento senza considerare errori di spaziatura.
- **Accuracy case insensitive senza spazi (CINS)**: confronto che ignora sia il case sia gli spazi, utile per gestire eventuali errori di segmentazione o spaziatura

La tabella seguente riporta i valori di string accuracy per ciascuna casistica, calcolati separatamente sui due dataset degli screenshot.

Gran parte degli errori riscontrati deriva dalle euristiche utilizzate nel post-processing, in particolare quelle legate alla generazione degli spazi tra le parole. Inoltre, per capire l'impatto delle lettere visivamente simili, come lo zero rispetto alla lettera "O" o la "I" rispetto alla "L", abbiamo definito una versione estesa della metrica CINS, indicata come **CINS***, che considera corretti anche queste confusioni. Come si osserva nella Tabella 6.4,

Dataset	CS	CI	CSNS	CINS	CINS*
Stringhe casuali	2.87	3.60	35.00	47.80	53.00
Stringhe inglesi	3.93	9.13	13.33	34.27	39.47

Tabella 6.4: Risultati string accuracy

l'inclusione di questi casi comporta un ulteriore miglioramento della string accuracy, particolarmente evidente nel caso delle stringhe casuali.

Capitolo 7

Demo

Per illustrare il funzionamento dell'algoritmo, è stata realizzata una demo interattiva utilizzando la libreria Python `Gradio`. Gradio consente di avviare un web server locale con un'interfaccia grafica accessibile da browser, collegata direttamente a funzioni Python, permettendo di testare il modello in modo semplice.

Come mostrato in figura 7.1, l'interfaccia guida l'utente attraverso le tre fasi principali:

1. Caricamento dell'immagine.
2. Pre-processing.
3. Riconoscimento (OCR).

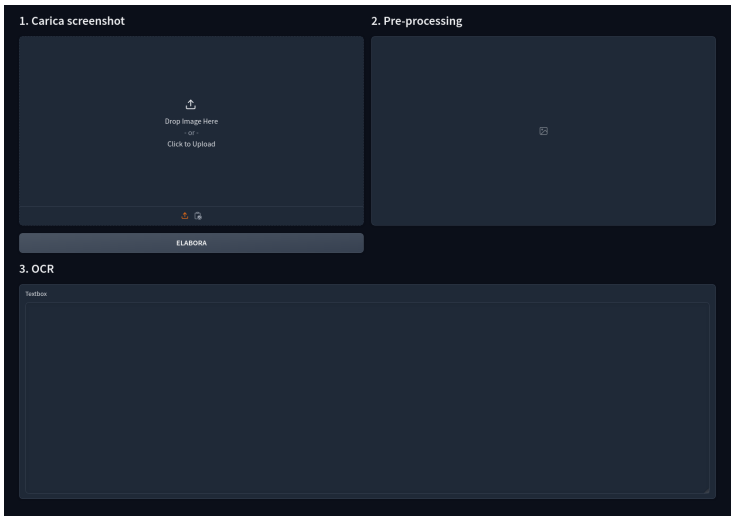


Figura 7.1: Interfaccia della demo

Caricamento dell'immagine

In questa fase l'utente può caricare un'immagine da elaborare, selezionandola direttamente da una cartella oppure incollandola dalla clipboard.

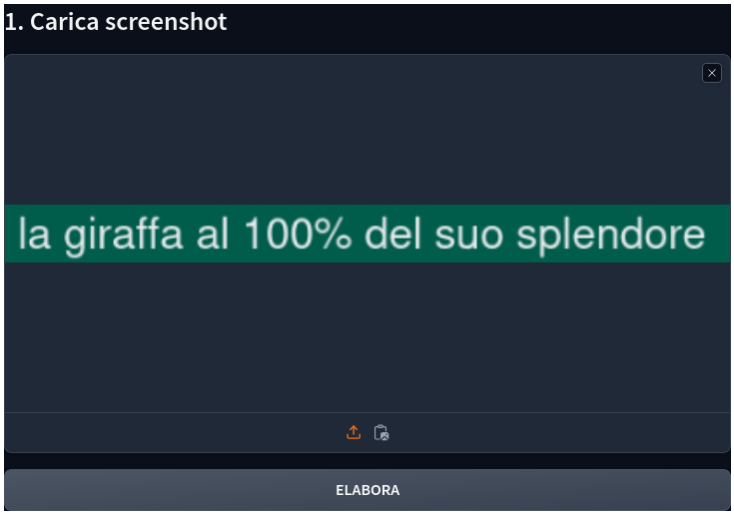


Figura 7.2: Interfaccia di caricamento dell'immagine

Pre-processing

Una volta cliccato il pulsante `ELABORA`, viene avviato il pre-processing dell'immagine. In output vengono mostrati due tipi di bounding box: uno **rosso** che delimita l'intera frase, e uno **verde** per ciascun carattere.

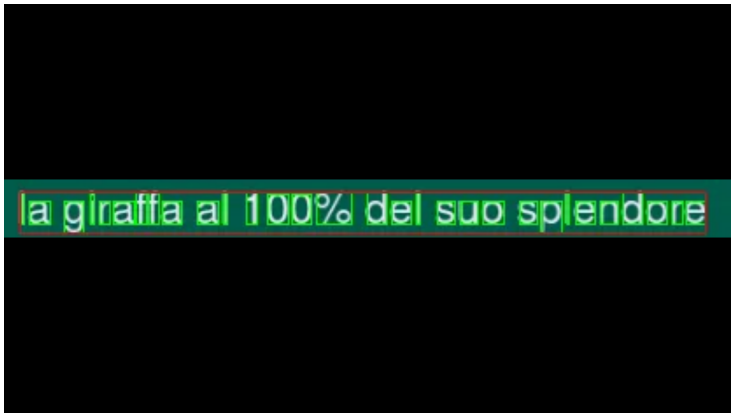


Figura 7.3: Immagine pre-processata

OCR

Infine viene mostrato il testo riconosciuto dal modello

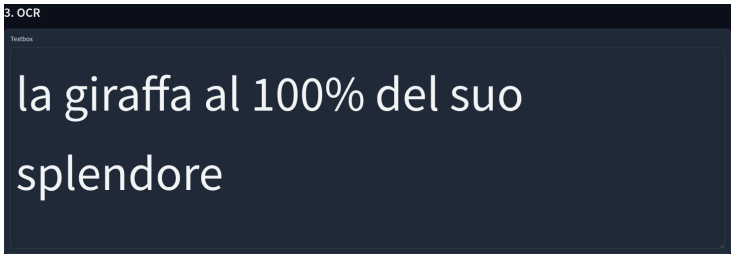


Figura 7.4: Output modello

Per una dimostrazione completa del funzionamento della demo, è disponibile un video nella repository github.

Capitolo 8

Conclusione

Il progetto ha approcciato il problema dell'OCR con un approccio ibrido che combina tecniche di image processing e deep learning. La relazione mostra la pipeline utilizzata e le euristiche, gli algoritmi e i modelli che ne fanno parte. Infine, si mostrano i risultati ottenuti, evidenziando le valutazioni complessive dell'intera pipeline e le prestazioni del modello finale.

I risultati hanno evidenziato come l'approccio individuato non sia particolarmente robusto, ma abbia comunque raggiunto un buon livello di accuratezza. Il modello ha mostrato una buona capacità di generalizzazione, ma è stato limitato dalla qualità del dataset e dalla sua ambiguità intrinseca dovuta all'architettura proposta.

Ulteriori evolutive che mantengono la semplicità dell'approccio utilizzato potrebbero includere l'estensione del riconoscimento di paragrafi, sfruttando la separazione di righe attraverso una proiezione laterale dell'immagine. Inoltre, si potrebbe considerare una modifica nella fase di generazione del dataset, andando a generare immagini non ambigue che il modello potrebbe riconoscere con maggiore facilità.

Durante lo sviluppo del progetto, sono stati appresi una serie di concetti e tecniche che potranno sicuramente essere utili in futuro. Vedere la curva di accuracy crescere insieme alla loss è stato affascinante e inaspettato. Il processo di training nella sua interezza ha richiesto del tempo per essere completato ma ha consentito di ottenere una dimestichezza con gli strumenti utilizzati, PyTorch in particolare. Un altro aspetto interessante è stato quello di dover trovare gli iperparametri ottimali per il modello, che ha richiesto un certo grado di sperimentazione che nessuno di noi si aspettava, mostrandoci come la ricerca della combinazione ottimale sia un processo cruciale ma complesso e non sempre lineare.