SAPIENZA
UNIVERSITÀ DI ROMA

Riconoscimento e Risoluzione di Espressioni Matematiche con Tecniche di Machine Learning

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea in Ingegneria Informatica e Automatica

Emanuele Frasca
Matricola 1836098

Relatore
Prof. Thomas Alessandro Ciarfuglia

Anno Accademico 2021/2022

Riconoscimento e Risoluzione di Espressioni Matematiche con Tecniche di Machine Learning
Sapienza Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: frasca.1836098@studenti.uniroma1.it

E resto, come se capissi
Il pensarti e lo starci distanti
A Daniele

Sommario

La tesi che segue tratta il problema del riconoscimento di espressioni matematiche scritte a mano da parte di un calcolatore elettronico per mezzo di tecniche di *machine learning* e *computer vision*.

In particolare si mostrano le metodologie impiegate per sviluppare e testare un tale sistema.

Per risolvere il problema, si utilizza un database contenente i simboli potenzialmente presenti in un'espressione matematica, che viene elaborato per normalizzarne la forma e associare a ogni simbolo la classe corrispondente.

Si definisce una *rete neurale convoluzionale* addestrata con i dati precedentemente elaborati, i cui *iperparametri* vengono trovati grazie a una *k-fold cross-validation* e una *grid search*.

Per testare la validità della rete, si effettuano delle predizioni su diverse espressioni, suddividendole negli elementi atomici che le compongono utilizzando tecniche di *computer vision*. Effettuate le predizioni sui singoli elementi, l'espressione originale viene ricomposta per poter essere poi facilmente elaborata.

Successivamente, si effettua il *parsing* dell'espressione riconosciuta al fine di calcolare il risultato atteso.

Si descrivono infine i punti di forza e le debolezze del sistema, e se ne effettua un'analisi qualitativa delle potenzialità.

Indice

Capitolo 1

Introduzione

Il riconoscimento e la risoluzione automatica di espressioni matematiche scritte a mano rappresenta una sfida complessa nel campo dell'informatica. A oggi tale problema viene affrontato utilizzando approcci basati sull'*intelligenza artificiale* (IA) e, in particolare, di *machine learning* (ML).
Il *machine learning*, oggi molto popolare, viene impiegato in diversi settori. Il suo obiettivo è quello di risolvere problemi complessi di riconoscimento e classificazione, automatizzando il processo di apprendimento del calcolatore.

La capacità di poter riconoscere e risolvere automaticamente tali formule ha importanti applicazioni nella ricerca e nell'istruzione.

Nell'istruzione essa può essere infatti utilizzata dagli studenti per poter facilmente riconoscere errori nello svolgimento di esercizi, mentre potrebbe essere utilizzata dai professori per sviluppare sistemi automatizzati per la correzione e valutazione dei compiti d'esame.

Nella ricerca può essere utilizzata per analizzare grandi quantità di documenti contenenti formule matematiche permettendone un'elaborazione automatizzata. Inoltre, l'inserimento di espressioni matematiche complesse in ricerche scientifiche richiede spesso una quantità di tempo non indifferente che potrebbe essere risparmiato utilizzando tali sistemi.

A oggi il problema del riconoscimento è discusso nella *International Conference on Document Analysis and Recognition* (ICDAR), una conferenza accademica internazionale che viene tenuta ogni due anni. Lo scopo della conferenza è proprio quello di discutere e proporre sistemi in grado migliorare lo stato dell'arte riguardo l'analisi e il riconoscimento automatizzato dei documenti. In particolare il problema del riconoscimento delle espressioni matematiche è affrontato nella *Competition on Recognition of Online Handwritten Mathematical Expressions* (CROHME).

Tuttavia la risoluzione di questo problema rappresenta tutt'oggi una sfida significativa per la comunità scientifica. Le difficoltà risiedono nella variabilità individuale della scrittura e nella complessità grafica delle espressioni.

Il lavoro di tesi è incentrato sullo sviluppo, la descrizione e la valutazione di un modello in grado di risolvere questo problema.

Capitolo 2

Metodologia

Nel capitolo che segue si descrive brevemente la metodologia e gli strumenti utilizzati per progettare un sistema in grado di affrontare il problema in questione.

2.1 Panoramica generale

Per riuscire a riconoscere la composizione e la sintassi di un'espressione matematica scritta a mano, è fondamentale effettuare un'analisi dei simboli che la compongono. L'analisi effettuata deve essere in grado di classificare questi simboli tenendo in considerazione la loro disposizione nell'espressione per poter comprendere la loro relazione reciproca.

La classificazione dei simboli è effettuata mediante l'utilizzo del *supervised learning*, una sottocategoria del *machine learning*. Nel *supervised learning*, sono impiegati dati di apprendimento contenuti in un *dataset* che consentono al calcolatore di acquisire conoscenza identificando pattern e relazioni al loro interno, al fine di produrre classificazioni.

Nel sistema descritto è impiegato un *dataset* (suddiviso in *training-set*, utilizzato per allenare il modello, e in *test-set*, utilizzato per valutarne le prestazioni) di immagini contenente simboli matematici, che viene utilizzato come input per un *modello neurale convoluzionale* (CNN) accuratamente progettato, il cui scopo è quello di effettuare classificazioni sui simboli che riceve in input. I modelli CNN sono ampiamente utilizzati per il riconoscimento delle immagini grazie alla loro capacità di saper estrarne automaticamente le caratteristiche geometriche più rilevanti. In questo modo, si è in grado di individuare gli elementi salienti delle immagini senza dover ricorrere a un'analisi manuale.

Le variabili (*iperparametri*) utilizzate per il *tuning* del modello sono scelte effettuando una *k-fold cross-validation* assieme a una *grid search*. La *k-fold cross-validation* è una tecnica di valutazione della performance di un modello. Il *training-set* viene diviso in k sottoinsiemi di dimensioni identiche. Il modello viene quindi addestrato su k-1 di questi sottoinsiemi e valutato sulla base della performance ottenuta sul restante sottoinsieme. Questo permette di evitare il problema dell'*overfitting* nel quale il modello si adatta troppo specificatamente ai dati di *train*. Questa procedura è ripetuta per ogni combinazione di parametri contenuti in una griglia di ricerca

accuratamente definita. Terminata la valutazione per ogni combinazione, si definisce il modello utilizzando i parametri che hanno prodotto il punteggio migliore.

L'estrazione dei simboli e l'analisi strutturale dell'espressione è effettuata con l'impiego della *computer vision*. In particolare, si estraggono i simboli che costituiscono l'espressione ricercando le componenti connesse nell'immagine, tenendo conto della loro posizione, per consentirne un'analisi accurata e completa.

Per effettuare la risoluzione dell'espressione si utilizza un sistema di *computer algebra system* (CAS). I CAS permettono di risolvere espressioni matematiche simboliche e sono in grado di eseguire operazioni come la semplificazione di espressioni, la risoluzione di equazioni, la derivazione e l'integrazione di funzioni.

Infine vengono effettuati dei test con 3 espressioni matematiche rappresentanti rispettivamente un'equazione di secondo grado, un'equazione di primo grado e un'espressione algebrica per valutare le prestazioni complessive del modello.

2.2 Librerie impiegate

L'implementazione [1] del sistema è stata effettuata con l'ausilio del linguaggio di programmazione *Python*, particolarmente adatto allo sviluppo di applicazioni di *intelligenza artificiale* grazie alla grande quantità di librerie disponibili.

2.2.1 Keras

Keras [2] è una libreria *open-source* per l'apprendimento automatico. Essa fornisce un'interfaccia per la libreria *Tensorflow* ed è stata sviluppata per semplificare la creazione e la formazione di modelli di *reti neurali*. *Keras* consente agli utenti di definire, addestrare e valutare modelli di apprendimento automatico in modo rapido e semplice.

Keras viene impiegato per definire, allenare e testare il modello di *machine learning* progettato nell'elaborato.

2.2.2 OpenCV

La libreria *OpenCV* [3] (*Open Source Computer Vision Library*) implementa più di 2500 algoritmi di *computer vision* e *machine learning*. La sua dinamicità le permette di essere utilizzata in vari settori, tra cui l'*image processing*, la *video analysis*, la *camera calibration* e l'*object detection*.

OpenCV viene utilizzato in varie fasi, sia per il *data processing* che per l'estrazione e l'analisi posizionale delle espressioni.

2.2.3 SymPy

SymPy [4] è una libreria utilizzata per la manipolazione e risoluzione di matematica simbolica. Essa può essere utilizzata per vari compiti tra cui la risoluzione di espressioni ed equazioni oltre che per il calcolo di derivate e integrali.

SymPy viene utilizzato per effettuare la risoluzione delle espressioni riconosciute.

Capitolo 3

Data processing

Nel seguente capitolo viene presentato il *dataset* utilizzato e viene fornita una descrizione dettagliata della metodologia impiegata per la sua elaborazione.

3.1 I dati nel machine learning

In un modello di *machine learning* i dati costituiscono un elemento fondamentale. Essi infatti rappresentano l'input che il sistema riceve e utilizza per poter creare dei modelli che possano riconoscere gli elementi salienti che caratterizzano i dati e diventare così in grado di riconoscere elementi mai visti in precedenza.

La qualità dei dati deve essere quindi elevata ma è importante tener conto del fatto che essa può essere influenzata da vari fattori tra cui possibili disturbi ed errori nella fase di acquisizione. Dunque per garantirne la qualità, è necessario svolgere un'attenta selezione seguita da una fase di elaborazione dei dati. In quest'ultima fase si cerca di normalizzare la forma dei dati e di eliminare possibili elementi di rumorosità per poterne successivamente estrarre solo i caratteri più importanti.

Oltre alla qualità dei dati è importante tenere conto anche della loro quantità. Un *dataset* con pochi elementi potrebbe non contenere abbastanza informazioni per rendersi in grado di generalizzare tutte le situazioni che si possono incontrare in fase di utilizzo. I dati devono dunque rappresentare in maniera accurata tutto il dominio su cui il modello verrà successivamente utilizzato in modo da poter costruire un sistema robusto e affidabile.

3.2 Composizione del dataset

I dati utilizzati sono stati presi dal *CROHME dataset* il quale è composto da più di 12.000 espressioni matematiche scritte e opportunamente classificate da centinaia di ricercatori di tutto il mondo. Da queste espressioni sono stati estrapolati i possibili simboli che possono apparire in un'espressione matematica. Un sottoinsieme dei simboli estrapolati e utilizzati nella tesi è visibile in Figura 3.1.

Il *dataset* è stato opportunamente suddiviso in due *set* distinti: il *training-set*, utilizzato per addestrare il modello, e il *test-set*, utilizzato per valutarne le prestazioni. Questa è una tecnica utilizzata nello sviluppo di sistemi di *machine learning* per

evitare che il modello si ritrovi in una situazione nella quale, essendosi eccessivamente adattato ai dati di addestramento, esso diventi incapace di compiere previsioni accurate su elementi non contenuti sul set di addestramento stesso. In questo modo, è possibile valutare la capacità di generalizzazione del modello e, se necessario, apportare modifiche per migliorarne le prestazioni.

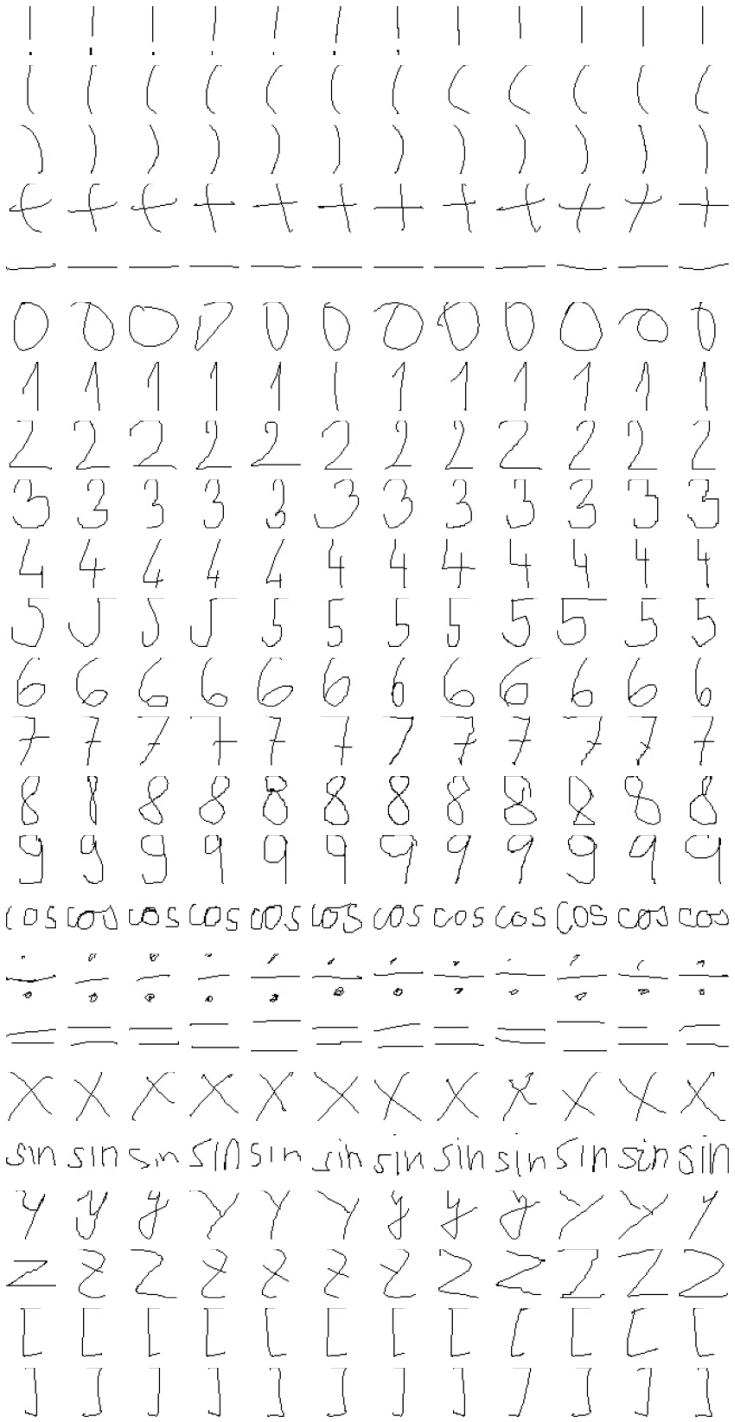

Figura 3.1. Esempio di dati contenuti nel *dataset*.

Dal grafico visibile in Figura 3.2 è possibile visionare la variabilità del *training-set*. In particolare è facile notare come siano presenti 24 classi e come alcuni simboli siano quantitativamente meno presenti di altri.

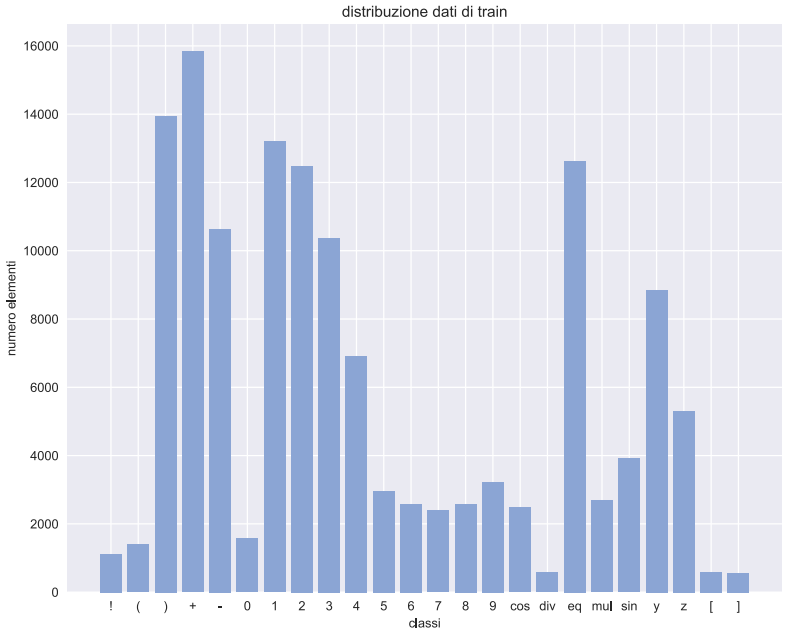

Figura 3.2. Distribuzione delle classi nel *training-set*.

Mentre nel grafico visibile in Figura 3.3 è possibile valutare la variabilità del *test-set*.

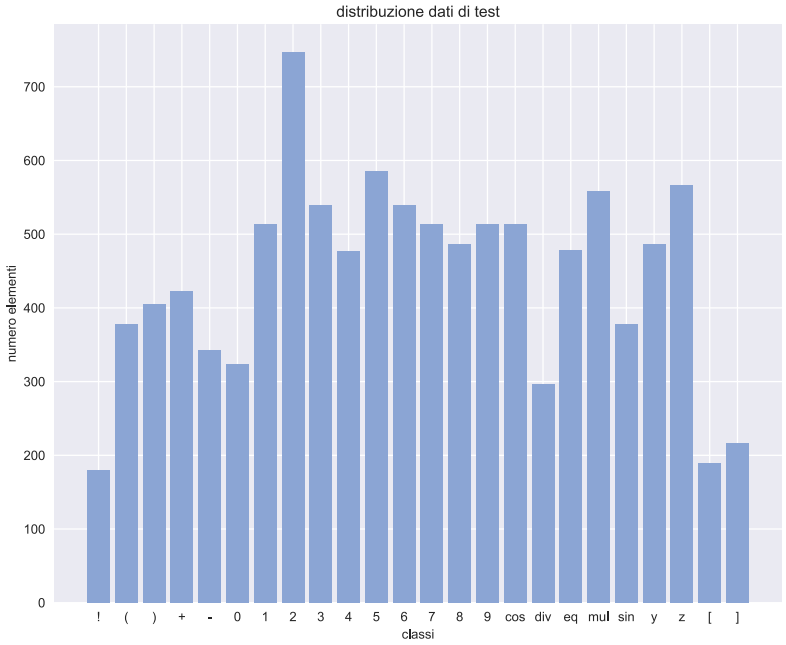

Figura 3.3. Distribuzione delle classi nel *test-set*.

3.3 Elaborazione delle immagini

Nella fase di elaborazione delle immagini vengono effettuate alcune operazioni che permettono ai dati di assumere una forma uniforme cercando di eliminare possibili elementi di rumore non utili per il *training* del modello.

Per eliminare potenziali elementi che potrebbero complicare il lavoro del riconoscitore, ogni immagine viene trasformata in scala di grigi poiché il colore stesso con cui si scrive un'espressione non ha rilevanza informativa al fine del riconoscimento.

Si effettua poi un ridimensionamento delle immagini utilizzando l'*interpolazione bilineare*. In questa tipologia di interpolazione il valore di un pixel nella nuova immagine viene calcolato come una combinazione lineare dei valori dei pixel vicini nella vecchia immagine. In particolare, il valore di ogni pixel dell'immagine finale viene calcolato come la media pesata dei valori dei pixel vicini, dove i pesi sono determinati dalla distanza del pixel dalla posizione del pixel di destinazione nella nuova immagine. Questa operazione è necessaria per evitare di dare in input al nostro sistema immagini troppo grandi che potrebbero rallentare la fase di addestramento oltre che rendere più difficoltoso per l'algoritmo riconoscere gli elementi caratterizzanti di un'immagine.

Viene successivamente effettuata un'operazione di *thresholding binario* (Figura 3.4): tale operazione consente di convertire l'immagine originale in una binaria dove i pixel con valore inferiore alla soglia impostata sono neri mentre quelli con valore superiore o uguale sono bianchi. Questo consente di fornire al sistema delle immagini nelle quali ogni gradazione di grigio è stata eliminata portando il "peso" del colore al minimo nella fase di addestramento.

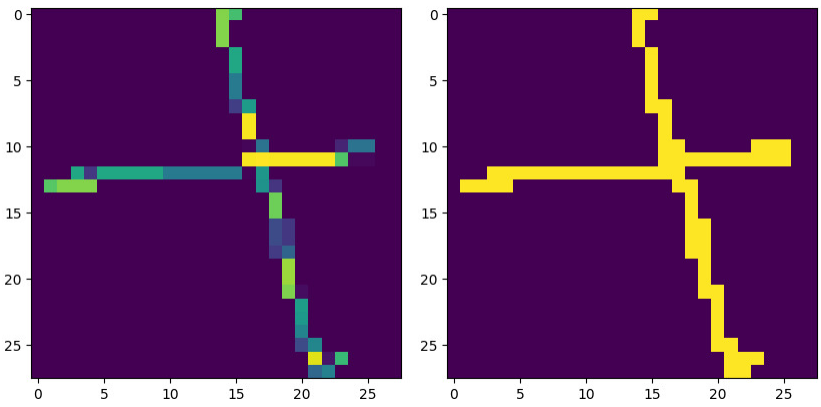

Figura 3.4. Esempio di *thresholding* su di un simbolo.

3.4 Aggiunta e codifica delle etichette

La fase di *labeling* è una fase cruciale nel processo di creazione e successivo addestramento di una rete neurale. In questa fase i dati presenti nel *dataset* vengono etichettati manualmente per poter indicare al modello a quale classe essi appartengano. In questa fase si effettua una mappatura nella quale ogni etichetta *testuale* viene tradotta in una corrispettiva etichetta *numerica*.

Nel caso in esame il *dataset* è stato suddiviso in varie cartelle che definiscono la classe a cui appartengono come è visibile in Figura 3.5.

Figura 3.5. Cartelle presenti nel *dataset*.

Dunque per effettuare il *labeling*, presa ogni singola immagine, ne viene associata un'etichetta denominata come la cartella di cui essa fa parte come visibile in Figura 3.6.

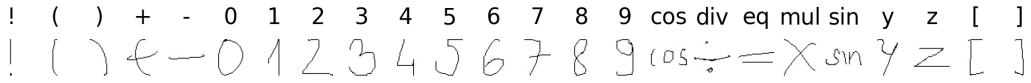

Figura 3.6. Esempio di *labeling*.

Successivamente, come espresso in precedenza, si mappano le etichette testuali con le corrispettive etichette numeriche come visibile in Listing 3.1.

```
"0": "0"
"1": "1"
"2": "2"
"3": "3"
...
...
"!": "19",
"sin": "20"
"cos": "21"
"[": "22"
"]": "23"
```

Listing 3.1. mappatura etichette.

Infine si aggiunge in ultima posizione dell'immagine vettorializzata la corrispettiva etichetta, come visibile nell'esempio 3.1

$$[0, 0, 255, 0, 255, ..., 255, 0, 19] \tag{3.1}$$

Le classi sono ora rappresentate con un decimale ma per essere utilizzate dal modello devono subire un ulteriore modifica ed essere codificate in *one-hot encoding* (Figura 3.7).

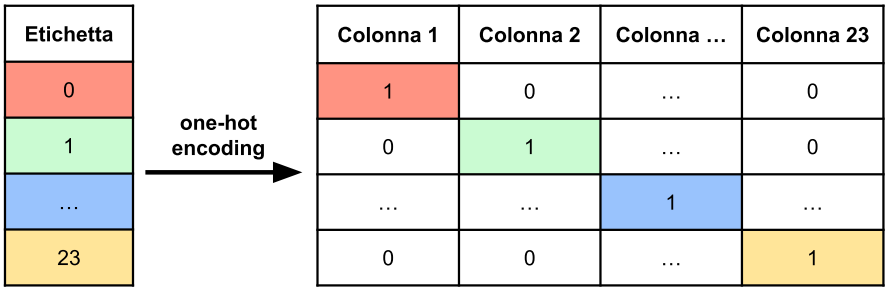

Figura 3.7. Esempio di *one-hot encoding*.

L'*one-hot encoding* viene utilizzato per rappresentare le classi in un modello di *machine learning* come un insieme di vettori binari dove ogni posizione corrisponde a una delle categorie possibili, e solo una di esse può avere come valore 1. Questo vettore binario viene quindi utilizzato come input per il modello per definire a quale categoria appartengano i dati corrispondenti.

Nel sistema implementato sono presenti 24 classi, dunque ogni vettore categorico avrà una lunghezza pari a 24. Per esempio nel modello in questione se abbiamo come input l'immagine in Figura 3.8 il corrispettivo vettore sarà pari a 3.2.

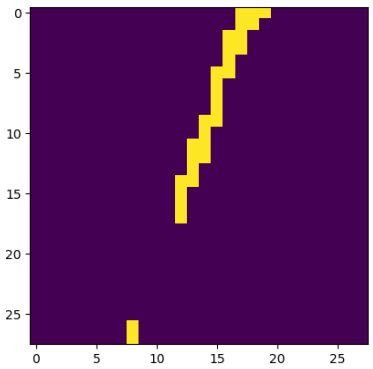

Figura 3.8. Esempio di elemento estrapolato dal *dataset*.

$$[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0] \tag{3.2}$$

A questo punto si ha un *dataset* preprocessato ed etichettato. Sarà infine necessario riportare le immagini in forma matriciale e impostare correttamente la rete neurale in modo tale da accettare in input elementi così definiti.

Capitolo 4

Definizione e tuning del modello

In questo capitolo si delinea il funzionamento di una rete neurale per poter definire con criterio il modello utilizzato. Viene inoltre mostrata la metodologia impiegata per la ricerca degli *iperparametri* più opportuni.

4.1 Funzionamento di una rete neurale

L'obiettivo di un modello di *machine learning* è quello di elaborare dati in input per poter successivamente effettuare delle previsioni o classificazioni. Questo è possibile grazie all'utilizzo di *neuroni artificiali*, che compongono le unità fondamentali di processazione delle informazioni in una rete neurale.

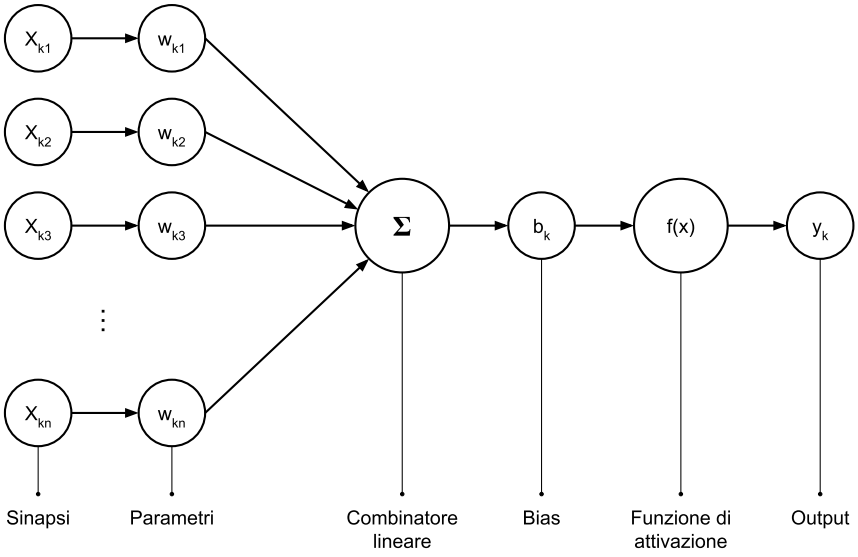

Figura 4.1. Struttura di un neurone.

Un neurone, visibile in Figura 4.1, ha un insieme di dati di input (*sinapsi*) che vengono moltiplicati per dei valori numerici (*parametri* o *pesi*). Si effettua dunque una combinazione lineare dei risultati a cui viene sommato un *bias*. Infine il tutto viene inserito in una *funzione di attivazione* per riportare in output un risultato normalizzato. Il *bias* consente di regolare il punto di partenza della *funzione di*

attivazione, il che può risultare importante per problemi in cui gli input non sono centrati intorno allo zero. La scelta della funzione dipende dal tipo di problema che si sta affrontando e dalla struttura della rete. Durante il processo di apprendimento i *pesi* del neurone sono aggiornati per adattasi ai nuovi dati che riceve.

In questo lavoro di tesi si utilizza la libreria *Keras* per definire un modello sequenziale (Figura 4.2).

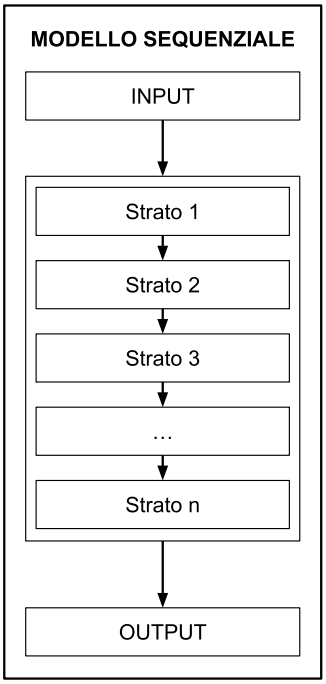

Figura 4.2. Struttura di modello sequenziale.

Un modello sequenziale prende in input dei dati che passano per una serie di strati (*layers*) definiti in sequenza, portando l'output di ogni strato a essere utilizzato come input per il successivo fino ad arrivare allo strato finale. Ogni strato ha uno scopo ben preciso e bisogna utilizzare i *layer* più congeniali in base alle specifiche del sistema che si vuole sviluppare.

Dopo aver definito un modello, è necessario procedere alla sua compilazione, ovvero determinare come verrà addestrato e come verranno valutate le sue prestazioni. In particolare vanno definiti tre parametri:

1. *loss function*: la funzione di perdita viene utilizzata per misurare l'errore tra le previsioni del modello e le corrette etichette date in input dal *dataset*.
2. *optimizer*: la funzione di ottimizzazione cerca di minimizzare la *loss function*. L'ottimizzatore fa ciò regolando i pesi del modello in modo da ridurre l'errore tra le previsioni e i valori di verità.
3. *metrics*: vanno definite le metriche che si desiderano utilizzare per valutare le prestazioni del modello durante l'addestramento.

Infine un modello va allenato, cioè esso viene esposto ai dati di addestramento per adattare i suoi parametri in modo da minimizzare l'errore sulle previsioni che dovrà effettuare in futuro.

4.2 Definizione del modello

Il modello sequenziale definito si basa sulla struttura neurale *LeNet-5*. Il codice del modello è stato definito con *Keras* ed è visibile nel Listing 4.1.

```
model = Sequential()
model.add(Conv2D(32, (3,3), input_shape=(28, 28, 1),
    activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(15, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(n_neurons, activation='relu'))
model.add(Dense(25, activation='softmax'))
```

Listing 4.1. Modello definito.

Come primo strato del modello troviamo un *Conv2D* il cui funzionamento è visibile in Figura 4.3. Una convoluzione è un'operazione matriciale, essa si serve di un kernel, una piccola matrice di pesi che scorre sui i dati di input effettuando una moltiplicazione elemento per elemento con la parte dell'input su cui si trova, per poi sommare in output.

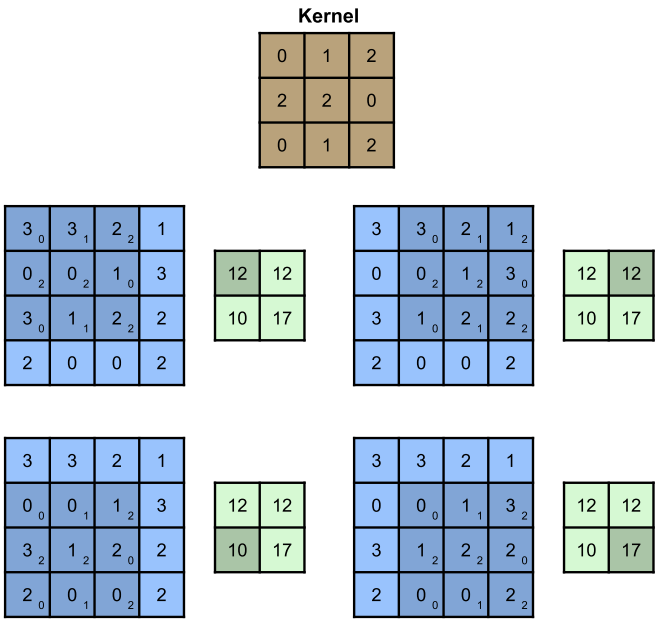

Figura 4.3. Esempio di *Conv2D*.

Dopo lo strato convoluzionale si trova un *MaxPooling2D* (Figura 4.4). Esso scansiona l'immagine ottenuta dalla convoluzione attraverso una finestra di dimensione data e restituisce il valore massimo all'interno di quella finestra come valore di output. Quindi, l'immagine di input viene divisa in piccoli blocchi e, per ciascuno di essi, viene mantenuto solo il valore massimo, riducendo così la dimensione dell'immagine. Questo processo aiuta a ridurre il numero di parametri della rete, rendendo il modello più efficiente e facilitando il rilevamento delle caratteristiche salienti dell'immagini.

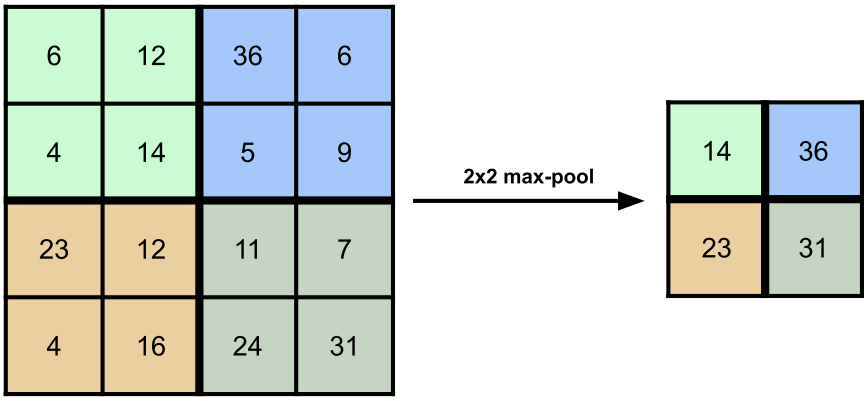

Figura 4.4. Esempio di *MaxPooling2D*.

Successivamente, per migliorare le capacità di apprendimento del modello i due *layer* precedenti vengono ripetuti.

Nello strato successivo si trova un *Flatten layer* (Figura 4.5). Esso prende in input un tensore di 2 o più dimensioni e ne effettua uno "srotolamento" per permettere ai dati di essere utilizzati con dei *layer* che hanno bisogno come input di un tensore monodimensionale.

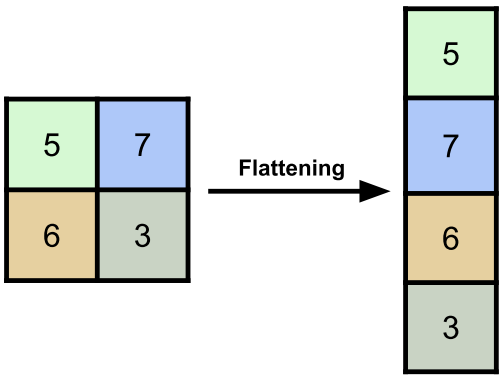

Figura 4.5. Esempio di *Flatten*.

Come ultimi strati sono presenti due *Dense layer* (Figura 4.6), i quali hanno la peculiarità per cui tutti i loro neuroni sono collegati a tutti i neuroni dei *layer* precedenti. Grazie a questa sua caratteristica essi sono in grado di modellare dati e caratteristiche non lineari particolarmente complesse.

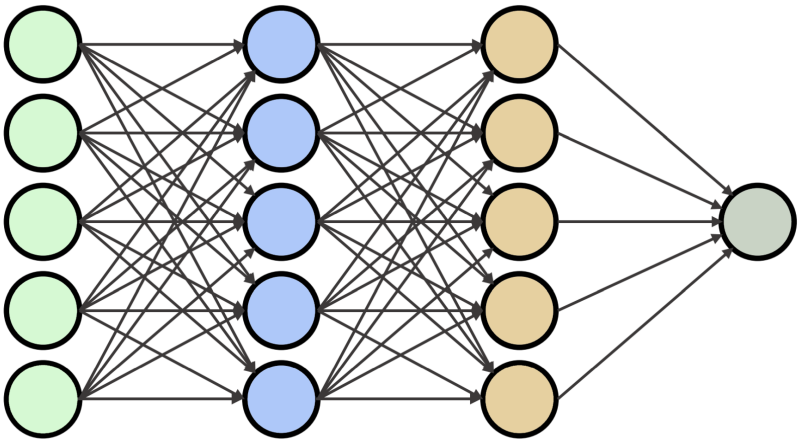

Figura 4.6. Esempio di *Dense layer*.

Il modello così definito crea una *rete neurale convoluzionale* (CNN). Una rete CNN è spesso utilizzata quando bisogna trattare problemi di ML che riguardano immagini.

Come detto in precedenza, dopo aver definito il modello, esso deve essere compilato e devono essere effettuate delle scelte in base alle finalità del modello stesso.

Come ottimizzatore è stato scelto Adam, un algoritmo basato su dei gradienti che utilizza il momento del primo e del secondo ordine per adattare la velocità di apprendimento del modello. Il momento del primo ordine è una media mobile esponenziale del gradiente, mentre il momento del secondo ordine è una media mobile esponenziale del gradiente al quadrato. Utilizzando i momenti l'algoritmo è in grado di adattare la velocità di apprendimento in maniera differente per ogni parametro del modello. Ogni peso viene individuato con l'equazione 4.1.

$$w = w - (learning\_rate/(\sqrt{v\_hat} + epsilon)) * m\_hat \tag{4.1}$$

dove:

- w è un peso del modello
- $learning\_rate$ è la velocità di apprendimento
- m_hat è il momento del primo ordine
- v_hat è il momento del secondo ordine
- $epsilon$ è un valore molto piccolo, utilizzato per prevenire l'instabilità numerica.

Poichè si sta risolvendo un problema di classificazione multi-classe, come funzione di perdita, è stata scelta la *"categorical-crossentropy"*. Questa funzione misura la differenza fra le previsioni che il modello effettua durante la fase di *training* e i valori effettivi che dovrebbe prevedere. In particolare la funzione *softmax* presente nell'ultimo *layer* del modello produce sui 24 possibili output una distribuzione di probabilità, utilizzata nel calcolo della funzione stessa. Essa viene matematicamente definita nell'equazione 4.2.

$$L_{CE} = -\sum_{i=1}^{n} t_i log(p_i) \tag{4.2}$$

con n numero delle classi, t_i i-esimo valore di verita e p_i la probabilità per la $i-esima$ classe.

Infine, per valutare le prestazioni del modello, si è scelto di utilizzare come metrica l'*accuracy*, in quanto l'obiettivo del sistema in esame è quello di massimizzare la percentuale di previsioni corrette. L'*accuracy* misura infatti la percentuale di previsioni esatte rispetto al totale delle previsioni effettuate. In questo modo, durante la fase di addestramento, il sistema si impegnerà a massimizzare tale valore.

4.3 K-fold cross-validation e iperparametri

In un sistema di *machine learning* gli *iperparametri* sono tutte quelle variabili che devono essere impostate a monte del processo d'apprendimento. Essi influenzano le prestazioni del modello e, se scelti correttamente, possono portare a un modello più accurato e generalizzante.

Per trovare gli *iperparametri* più funzionali per il modello proposto è stato utilizzato il workflow visibile in Figura 4.7 e, in particolare, è stata impiegata la tecnica della *k-fold cross-validation* assieme a una *grid search*. [5]

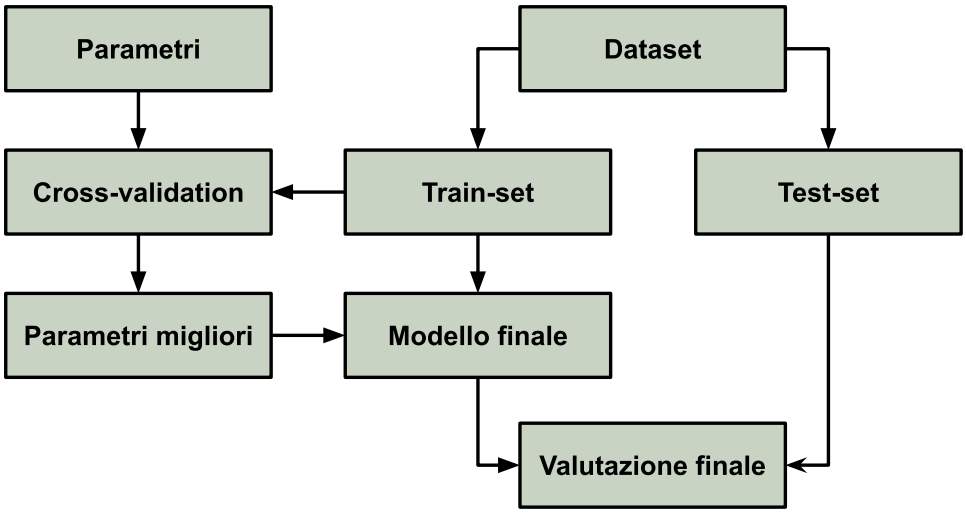

Figura 4.7. Tipico *workflow* per il *training* di un modello.

La *k-fold cross-validation* (stilizzata in Figura 4.8) viene utilizzata per risolvere il problema dell'*overfitting* nel quale, il modello, essendosi eccessivamente adattato al *training-set*, perde la capacità di predire l'output corretto su dati che mai aveva analizzato prima. Operativamente il *dataset* di addestramento viene suddiviso in k parti uguali. Il modello viene quindi addestrato su k-1 porzioni e viene testato sulla parte rimanente. Questo processo viene ripetuto k volte, in modo che ogni porzione venga utilizzato una volta come set di test. Infine viene valutata la prestazione del modello effettuando una media tra i risultati dati delle predizioni di ogni *fold*.

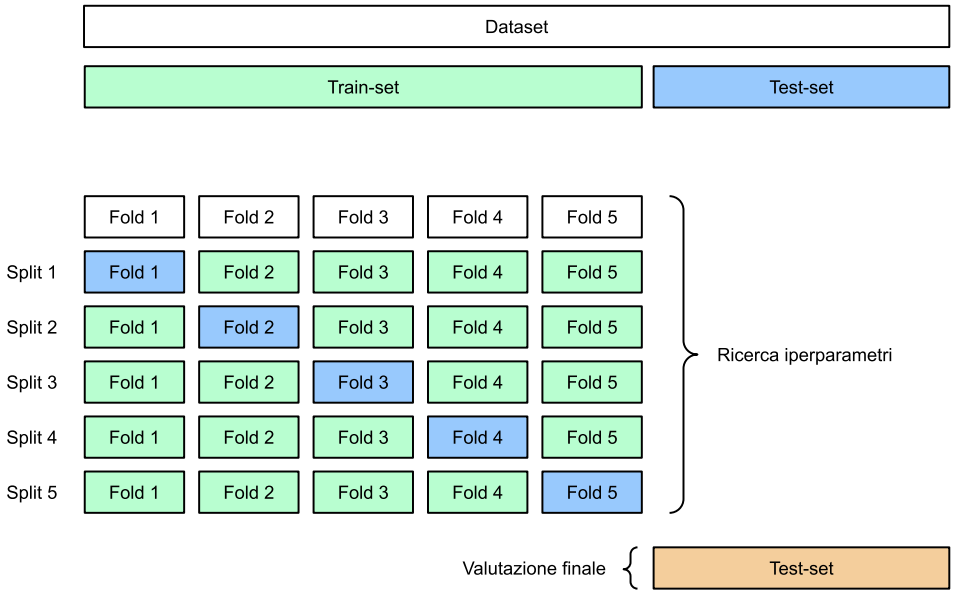

Figura 4.8. Funzionamento della *k-fold cross-validation*.

La *grid search* viene invece utilizzata per testare un insieme di parametri dati sul modello stesso e ricevere in output un punteggio che aiuta a definire qual è la combinazione parametrica migliore.

I parametri più utili da ricercare per il modello sono il *numero di epochs*, il *batch size*, e il *numero di neuroni* dello strato denso finale.
Un *epoch* rappresenta una singola iterazione completa del modello sull'intero *set* di dati di addestramento. Durante l'addestramento della rete neurale, i dati di input vengono divisi in *batch* di una grandezza definita e per ogni iterazione dell'addestramento, la rete neurale elabora un *batch* di dati di input alla volta, aggiorna i pesi sinaptici in base all'errore calcolato e passa poi al *batch* successivo.

Cercando la combinazione migliore di *epochs*, *batch size* e *numero di neuroni* si può trarre il meglio dal *training-set*.

Capitolo 5

Estrazione dei simboli, riconoscimento e risoluzione

In questo capitolo viene illustrato come sia possibile utilizzare la *computer vision* per estrarre tutti i simboli presenti in un'immagine, effettuare previsioni su ciascun elemento e calcolare dunque il risultato finale dell'espressione.

5.1 Estrazione dei simboli

Per il riconoscimento delle componenti atomiche che formano l'espressione si fa uso della libreria *OpenCV*.
Applicato un *thresholding binario* (Figura 3.4) all'espressione data, si utilizza la funzione *findContours* per trovarne i contorni (Figura 5.1). Essa, basandosi sulle differenze di colore tra pixel vicini riesce a distinguere gli elementi di interesse dallo sfondo dell'immagine.

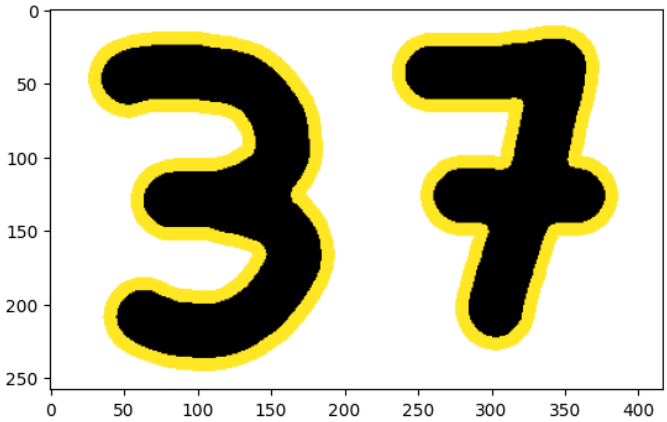

Figura 5.1. Esempio d'uso di *findContours*.

È necessario trovare il rettangolo che comprende tutto il simbolo, dunque utilizzando i valori x, y, w, h datoci dalla funzione *boundingRect* per ogni contorno è possibile isolare ogni simbolo come visibile in Figura 5.2.

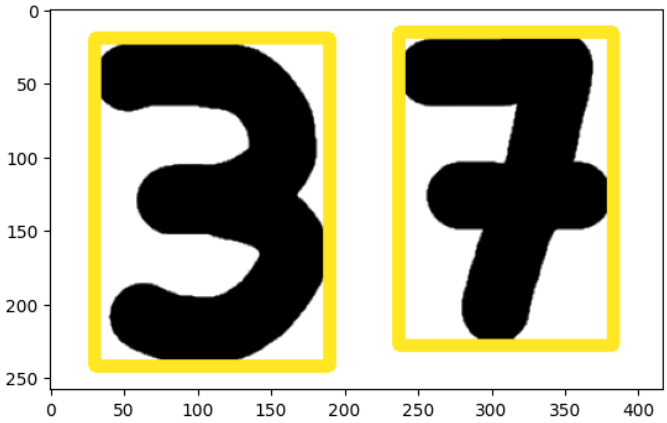

Figura 5.2. Esempio d'uso di *boundingRect*.

Dato che oltre all'isolamento dei singoli simboli si è interessati alla loro posizione, nell'immagine viene utilizzata la funzione *sorted* che permette, utilizzando la posizione lungo l'asse x di ogni elemento, di restituire in output la lista ordinata da sinistra a destra dei simboli (in Figura 5.3 un esempio di espressione ordinata).

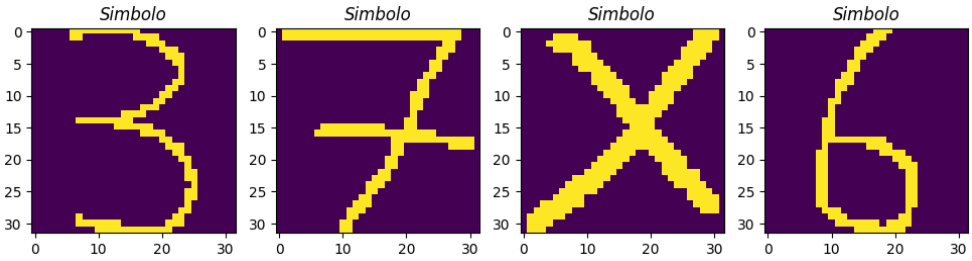

Figura 5.3. Esempio di estrapolazione dei simboli.

Si cerca infine di comprendere se il simbolo in esame possa essere un esponente o meno. Questo lo si fa tenendo conto della posizione e dimensione dell'esponente rispetto al simbolo che lo precede. Essendo l'esponente uguale a un qualsiasi numero dal punto di vista del modello in questa fase è necessario *marcare* il simbolo come possibile esponente per poterlo poi inserire correttamente nell'espressione finale. L'algoritmo utilizzato per la marcatura di un simbolo come esponente è visibile nell'Algorithm 1.

Data: x, y, h e w posizione contorni possibile esponente e xp, yp, hp, wp posizione contorni elemento precedente
Result: Una lista di contorni marcati come esponenti
for *contorno[i] in contorni con i != 1* **do**
  if $y < yp$ *e* $h < hp$ *e* $w < wp$ **then**
    marca il simbolo come possibile esponente
  end
end

Algorithm 1: Algoritmo per la marcatura dell'esponente.

5.2 Riconoscimento e risoluzione

Avendo ora una lista di simboli matematici ordinati opportunamente marcati e un modello allenato è possibile effettuare delle previsioni su ogni singolo simbolo trovato in precedenza.

Il risultato di ogni singola previsione sui simboli produrrà in output un vettore della stessa lunghezza del numero delle classi presenti nel modello. Tale vettore conterrà, nella corrispondente posizione i, la probabilità che quel simbolo appartenga alla classe i. Si è quindi interessati a cercare il valore di probabilità più alto presente nel vettore per trovare la corrispettiva classe che, secondo il modello, è la più probabile.

Ad esempio, se si prende in considerazione un simbolo estratto da un'espressione matematica come quello mostrato in Figura 5.4 e si chiede al modello a quale classe appartenga, si riceverà in output il vettore delle probabilità 5.1.

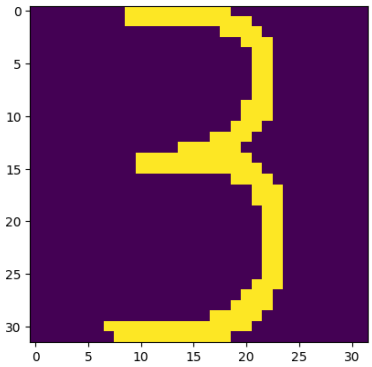

Figura 5.4. Elemento da predire.

$$[70, 87, 13, 100, 17, 31, 8, 9, 20, 22, 54, 78, 18, 31, 27, 66, 7, 11, 70, 76, 61, 13, 26, 21] \tag{5.1}$$

Dove se si calcola l' $indexof(max(v))$ si ha in output l'etichetta *numerica* 3, che, utilizzando la *label_map* espressa in Listing 3.1 si può tradurre nella corrispettiva etichetta *testuale* che, in questo caso, è proprio pari al numero 3.

Effettuate le previsioni in ordine esse possono essere trasformate nei corrispettivi simboli *ASCII* e concatenate in un'unica stringa.

Per risolvere le espressioni precedentemente riconosciute, si utilizza la libreria *SymPy*, che permette di effettuare il *parsing* e la risoluzione di espressioni matematiche. Nel caso in cui l'espressione non contenga incognite viene effettuato un semplice calcolo algebrico mentre nel caso in cui sia presente si eguaglia l'espressione a 0 per trasformarla in un'equazione e permettere il calcolo della soluzione.

Capitolo 6

Test e valutazione del sistema

In questo capitolo ci si concentra sull'analisi del sistema prodotto, cercando di valutarne i punti di forza e le debolezze. Si cerca di identificare i migliori *iperparametri* per il modello proposto, in modo da poterlo allenare in maniera ottimale. Successivamente si effettua una valutazione preliminare del sistema basandosi sui risultati dati dalle previsioni effettuate sul *test-set* e infine si testa l'affidabilità del sistema nel riconoscimento e la risoluzione di tre espressioni matematiche scritte a mano.

6.1 Ricerca degli iperparametri

Dopo aver effettuato il caricamento e il processamento del *dataset* si effettua la ricerca degli *iperparametri* precedentemente definiti. In particolare, per ogni combinazione parametrica, è effettuata una *5-fold cross validation* utilizzando come possibili valori della *grid search* gli elementi riportati nel Listing 6.1.

```
param_grid = {
    'n_neurons': [128,256],
    'epochs': [2,4,6],
    'batch_size':[32,64]
}
```

Listing 6.1. Griglia parametrica.

Effettuata la ricerca si ottiene per ogni combinazione parametrica un punteggio compreso tra 0 ed 1. Si è interessati alla combinazione migliore, per cui è d'interesse il punteggio più elevato. La ricerca degli *iperparametri* porta ai risultati visibili nel grafico in Figura 6.1 che definisce come valori migliori:

- *epochs*: 4
- *n_neurons*: 128
- *batch_size*: 32

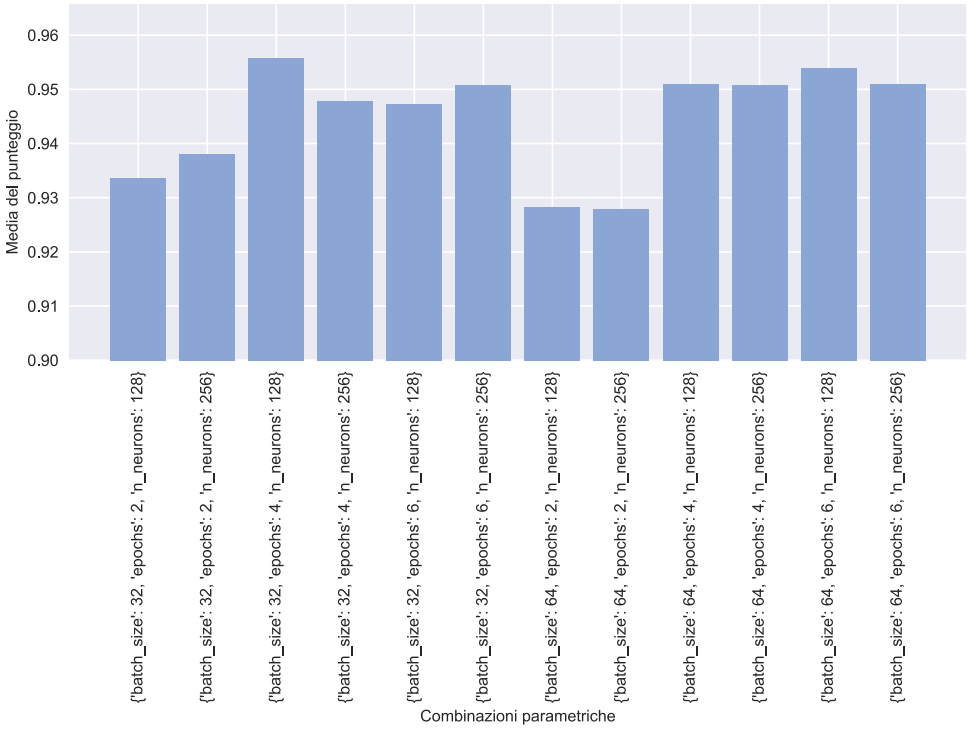

Figura 6.1. Grafico iperparametri.

Effettuata la ricerca degli *iperparametri*, è possibile la definizione finale del modello che è visibile nel Listing 6.2.

```
Model: "Modello"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_66 (Conv2D)           (None, 32, 32, 32)        320
_________________________________________________________________
max_pooling2d_66 (MaxPool (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_67 (Conv2D)           (None, 14, 14, 15)        4335
_________________________________________________________________
max_pooling2d_67 (MaxPool (None, 7, 7, 15)          0
_________________________________________________________________
flatten_33 (Flatten)         (None, 735)               0
_________________________________________________________________
dense_66 (Dense)             (None, 128)               94208
_________________________________________________________________
dense_67 (Dense)             (None, 24)                3096
=================================================================
Total params: 101,959
Trainable params: 101,959
Non-trainable params: 0
_________________________________________________________________
```

Listing 6.2. Modello finale.

6.2 Addestramento della rete

Dopo aver effettuato la ricerca degli *iperparametri* e avendo quindi un modello definitivo si può addestrare la rete con i dati presenti nel *training-set*. Nella fase di addestramento, come detto in precedenza, si cerca di minimizzare la *loss function* del modello.
Visualizzando il grafico in Figura 6.2 che descrive l'andamento della *loss function* e dell'*accuracy* in relazione a ogni *epoch* si può analizzare il fatto che entrambe le curve seguano l'andamento previsto (minimizzazione della *loss function* e massimizzazione dell'*accuracy*).

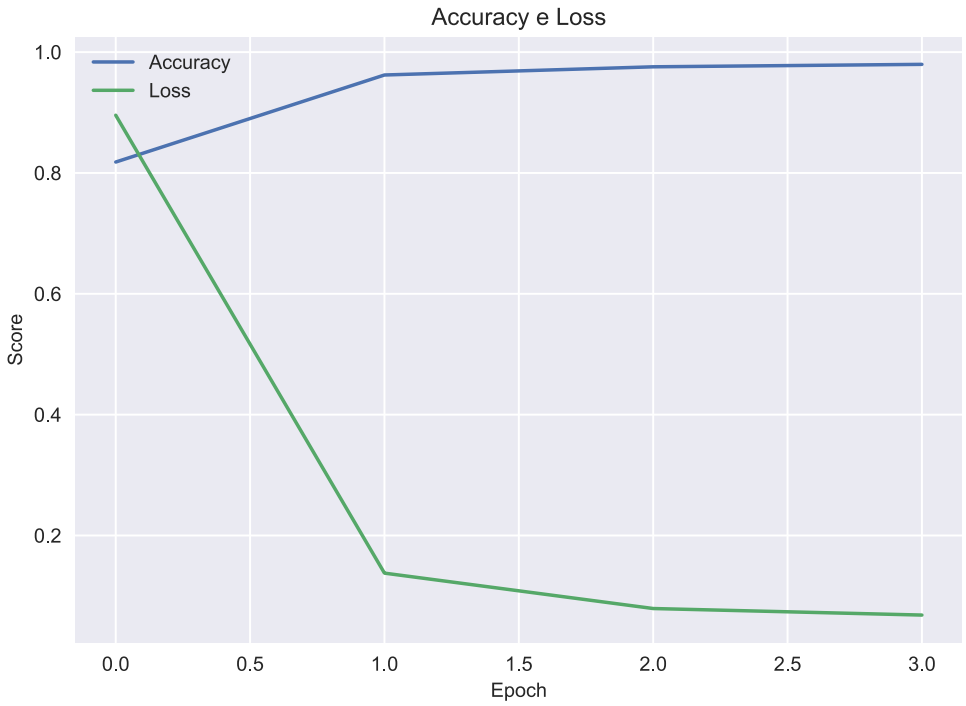

Figura 6.2. Grafico di accuracy e loss.

Infine si effettua una valutazione del modello utilizzando il *test-set*. Effettuando le previsioni vengono trovati i risultati espressi nella Tabella 6.1.

variabile	valore
test_accuracy	95%
test_loss	0,20

Tabella 6.1. Risultati ottenuti.

Si può visualizzare graficamente la validità delle predizioni effettuate per mezzo della *confusion matrix* visibile in Figura 6.3: Essa permette di comprendere a colpo d'occhio quali sono gli elementi che il modello confonde più facilmente e permette di lavorarci sopra per poter migliorare il sistema.

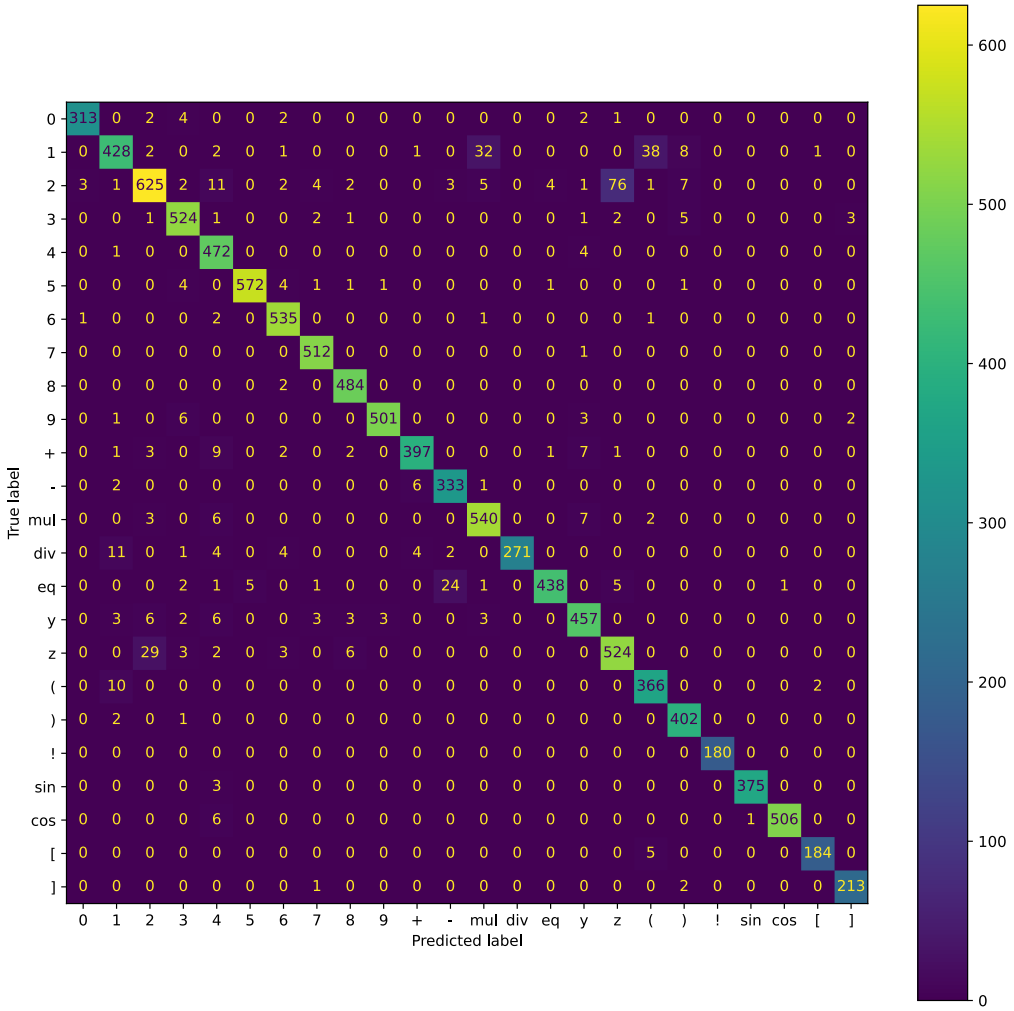

Figura 6.3. Matrice di confusione.

Analizzando la matrice di confusione si possono percepire i punti di debolezza del nostro modello: la somiglianza intrinseca del simbolo "1" con i grafemi associati a ")" e "div" porta il sistema a effettuare vari errori in fase di *testing*. La stessa cosa accade per il simbolo "2" e la lettera "z" per lo stesso motivo definito poco addietro.

6.3 Espressioni di test

Sono state scritte con l'uso di una tavoletta grafica 3 espressioni matematiche da far analizzare al modello per poter valutare le sue prestazioni. Le espressioni sono visibili nelle Figure 6.4, 6.5 e 6.6.

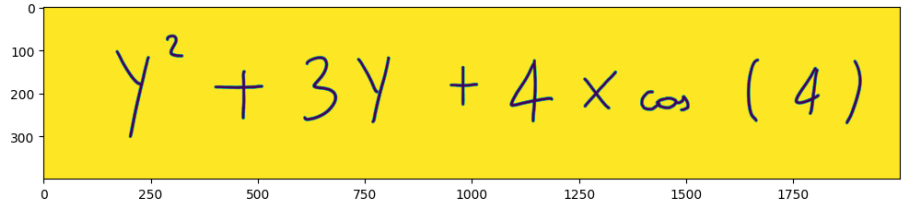

Figura 6.4. Espressione 1.

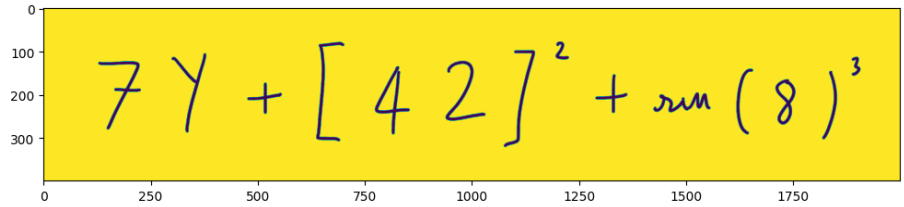

Figura 6.5. Espressione 2.

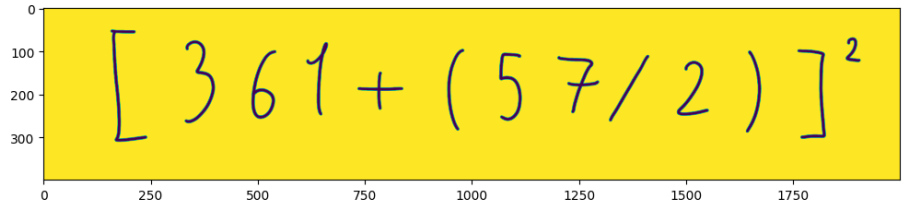

Figura 6.6. Espressione 3.

L'espressione 1 testerà il sistema con un'equazione di secondo grado, essa viene riconosciuta come tale ed eguagliata a 0 per calcolarne il risultato.

L'espressione 2 invece dovrà essere riconosciuta come un equazione di primo grado e risolta di conseguenza.

L'espressione 3 testerà il sistema nel calcolo di espressioni senza incognite.

6.4 Estrazione dei simboli

In questa fase si testa la capacità del sistema di riuscire a estrarre correttamente i simboli dalle singole espressioni. In particolare ogni elemento è segnato come "Simbolo" o come "Esponente" per poterli differenziare nella fase successiva di *parsing*. Le griglie contenenti ogni simbolo estratto per l'espressioni date sono visibili nelle Figure 6.7, 6.8 e 6.9.

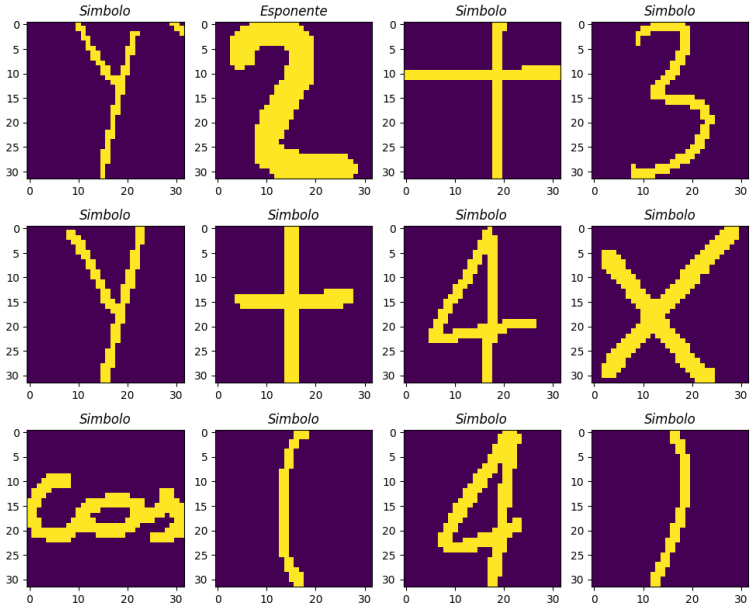

Figura 6.7. Griglia espressione 1.

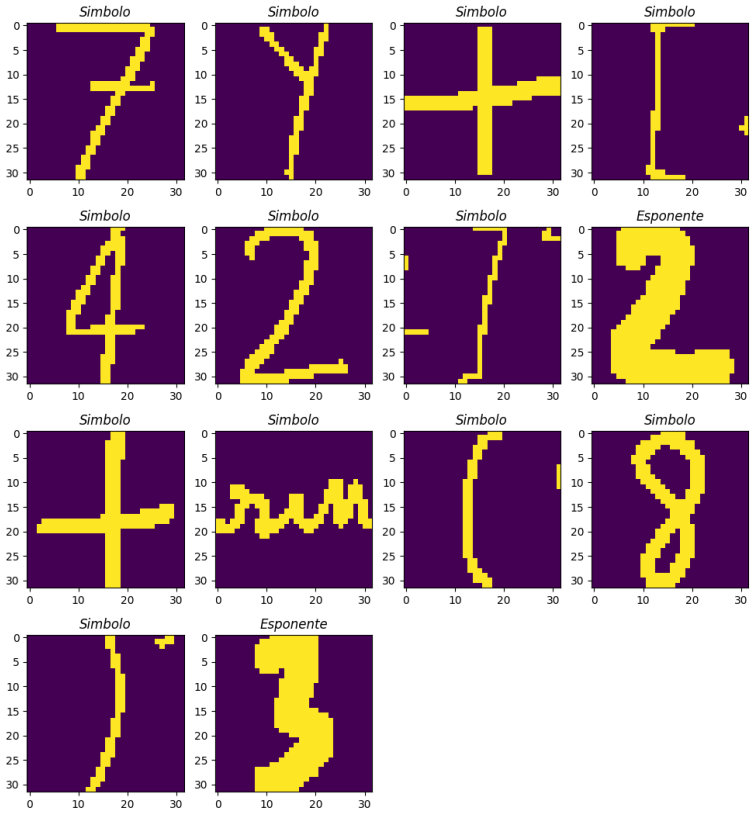

Figura 6.8. Griglia espressione 2.

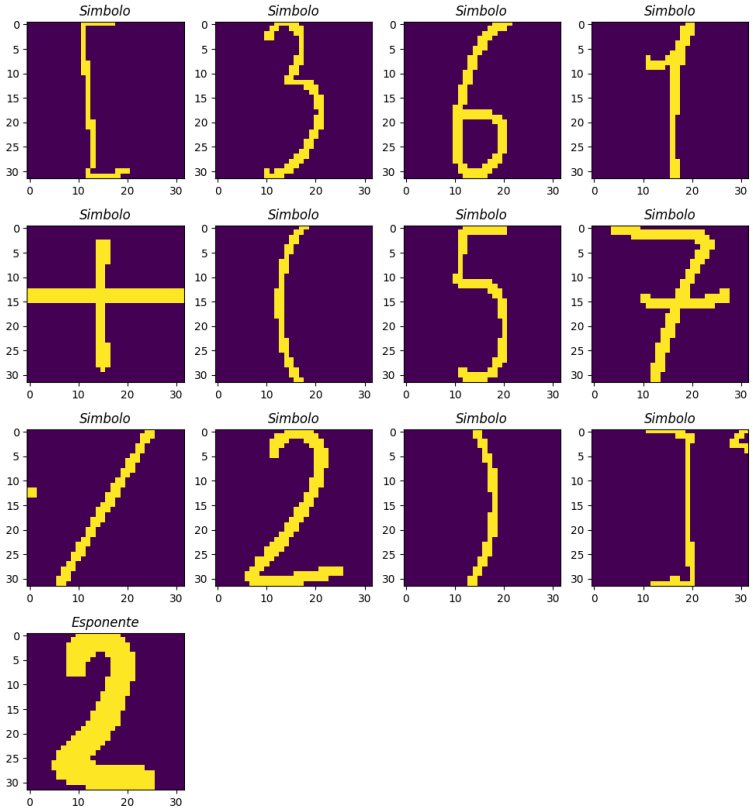

Figura 6.9. Griglia espressione 3.

Nell'espressione 1 il sistema rileva in *posizione 2* un probabile esponente e lo marca come tale.

Nell'espressione 2 il sistema rileva in *posizione 8* e *15* dei probabili esponenti e li marca come tali.

Nell'espressione 3 il sistema rileva in *posizione 13* un probabile esponente e lo marca come tale.

6.5 Predizioni e risoluzione

Nella fase finale si prende dunque la lista di immagini precedentemente prodotta e la si dà in pasto al modello allenato. Si utilizza una mappa per tradurre ogni predizione (che il modello restituisce sotto forma di numero) nel corrispondente simbolo ASCII e si aggiunge, ove necessario, il simbolo "**" per evidenziare la possibile presenza di un esponente nelle posizioni precedentemente marcate.

Effettuando le previsioni e la risoluzione delle 3 espressioni date in esame si ha come risultato:

Espressione 1 riconosciuta: 1: $y**2 + 3y + 4 * cos(4)$ con una probabilità del 98.16%.

Il sistema riconoscendo un'equazione di secondo grado la risolve e stampa a video il vettore contente le due radici:

$$[-3/2 + sqrt(9 - 16 * cos(4))/2, -sqrt(9 - 16 * cos(4))/2 - 3/2]$$

Espressione 2 riconosciuta: $7y + [421 * *2 + sin(8) * *3$ con una probabilità del 90.89%.

Il sistema riconoscendo un *parsing* errato (non chiusura della parentesi quadra) stampa a video un messaggio d'errore.

Espressione 3 riconosciuta: $[361 + (57/2)] * *2$ con una probabilità del 95.45%.

Il sistema ha riconosciuto un espressione senza incognite, dunque calcola e stampa a video il valore 151710.25.

Come si può notare nell'espressione 2 il sistema ha scambiato una parentesi quadra per un uno, portando a un errore nella fase di *parsing* e calcolo dell'espressione.

Capitolo 7

Conclusioni

I test effettuati hanno mostrato che il problema del riconoscimento di espressioni matematiche è potenzialmente risolvibile.

Il sistema sviluppato dimostra infatti di saper estrarre e riconoscere i simboli che compongono un'espressione con un'affidabilità media del 94,83%.

È importate effettuare anche delle considerazioni circa i suoi limiti. Esso infatti è stato progettato per comprendere esclusivamente espressioni scritte su una singola riga, dunque sarà impossibile aspettarsi, ad esempio, il riconoscimento di una frazione.

Le potenzialità del riconoscitore potrebbero inoltre essere espanse implementando la capacità di identificare altre funzioni canoniche ed elementi matematici più complessi su cui effettuare delle analisi relazionali e posizionali specifiche, come ad esempio integrali, limiti e sommatorie.

Sitografia

[1] E. Frasca, *noostale/Riconoscitore-e-risolutore-di-espressioni-matematiche-scritte-a-mano*, 2023. indirizzo: `https://github.com/noostale/Riconoscitore-e-risolutore-di-espressioni-matematiche-scritte-a-mano` (visitato il 07/03/2023).

[2] Keras, *Keras API reference*, 2023. indirizzo: `https://keras.io/api/` (visitato il 07/03/2023).

[3] OpenCV, *OpenCV: OpenCV modules*, 2023. indirizzo: `https://docs.opencv.org/4.x/` (visitato il 07/03/2023).

[4] S. D. Team, *SymPy 1.11 documentation*, 2022. indirizzo: `https://docs.sympy.org/latest/index.html` (visitato il 07/03/2023).

[5] S.-L. Developers, *Cross-validation: evaluating estimator performance*, 2023. indirizzo: `https://scikit-learn.org/stable/modules/cross_validation.html` (visitato il 07/03/2023).