Università degli Studi di Firenze

Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

---

Tesi di Laurea Triennale in Ingegneria Informatica

Localizzazione di scale in planimetrie

Candidato
Manuel Salamino

Relatore
Prof. Simone Marinai

Correlatore
Samuele Capobianco

Anno Accademico 2017/18

Indice

Introduzione

La scopo di questo elaborato è la realizzazione di un programma che permetta, data una planimetria in ingresso, di estrarre le scale presenti al suo interno.
Per realizzare questo programma è stato utilizzato il linguaggio di programmazione Python, nella sua versione 3.6.

Per ottenere il risultato vengono eseguiti una serie di passaggi:

- lettura della planimetria
- estrazione delle componenti connesse
- filtraggio delle componenti non conformi ad essere parte di una scala
- grafo di adiacenza delle componenti rimaste
- scarto parti del grafo che non sono scale

Al termine di queste operazioni, come vedremo, riusciamo ad estrarre la scala dalla planimetria.

Le planimetrie utilizzate come base per la realizzazione ed il test del programma sono divise in due Dataset: Dataset_1 e Dataset_2, i quali differiscono tra loro per dimensione delle immagini, edifici rappresentati e modalità di disegno.

Al fine di effettuare test sul programma, sono state eseguite delle etichettature manuali delle scale all'interno di tutte le planimetrie dei due Dataset, in modo da poter confrontare la regione restituita dal programma con l'effettiva regione etichettata manualmente.
Per effettuare l'etichettatura è stato utilizzato un programma denominato *Sloth*[1], che permette di aprire e visualizzare una planimetria e fornisce la possibilità di porre sopra una certa area un'etichetta.
Tutte le aggiunte inserite vengono poi salvate all'interno di un file *JSON* seguendo un certo ordine logico per cui la successiva lettura dei dati è abbastanza semplice.

Quindi per l'esecuzione del test è stato necessario, dopo aver effettuato manualmente tutte le etichettature, leggere dai file *JSON* l'area corrispondente alle scale e confrontarla con la sezione trovata dal programma.

Dopodichè viene svolto un test del programma per valutare attraverso parametri statistici la performance su una singola planimetria. Ed infine, accumulando i risultati dei test su tutte le planimetrie, viene fatta una valutazione generale dell'algoritmo.

Capitolo 1

Elaborazione dell'immagine

Le planimetrie da analizzare ed utilizzare come campioni del nostro programma sono in formato *PNG* ed hanno una dimensione di circa 2400x3500 per quanto riguarda il Dataset_1, mentre il Dataset_2 ha dimensioni più variabili che vanno da 1500x2000 fino anche a 5000x7000.
Nel primo dataset utilizzato le immagini sono in bianco e nero, mentre nel secondo alcune linee hanno colori differenti.

Per passare dall'immagine in formato *PNG* ad una struttura dati che possa essere utilizzata in linguaggio Python ci siamo serviti di **OpenCV**[2], una libreria presente in tutte le piattaforme e che offre strumenti nell'ambito della visione artificiale in tempo reale, quindi offre anche strumenti per la lettura e modifica di immagini in formato digitale. E' stata utilizzata anche **PIL.Image**, una sezione della libreria PIL, utilizzata anch'essa per la lettura, manipolazione e salvataggio di immagini digitali in diversi formati.

Prima di utilizzare le funzioni di OpenCV l'immagine deve essere formattata in un certo modo, quindi la prima fase dell'elaborazione dell'immagine avrà questo scopo.
Per iniziare queste operazioni utilizziamo il comando:

```
image = PIL.Image.open(path)
```

che trasforma l'immagine presente al percorso indicato in *path* (*Figura 1.1*) in un oggetto di tipo PIL.Image.

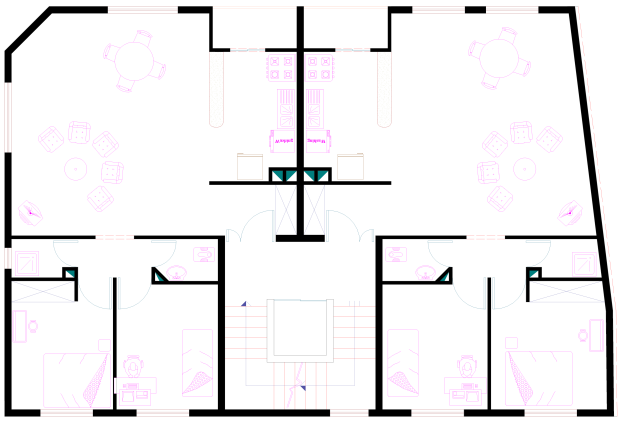


Figura 1.1: Esempio di una planimetria a colori del Dataset_2.

Dopodichè effettuiamo una serie di operazioni per rendere l'immagine utilizzabile dalle funzione di OpenCV di cui necessitiamo:

- **trasformazione in scale di grigio**, attraverso la funzione

  ```
  gray_image = cv2.cvtColor(numpy.asarray(image), cv2.COLOR_BGR2GRAY)
  ```

  che prima trasforma l'immagine appena letta in un *array numpy* (anche **numpy** è una libreria, ed offre strumenti per la gestione di array n-dimensionali) e poi la trasforma in scale di grigio (*Figura 1.3*).

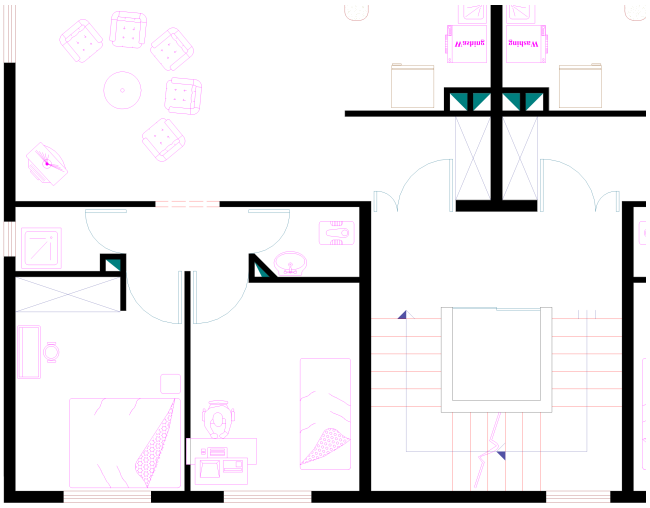


Figura 1.2: Zoom della planimetria in *Figura 1.1*.

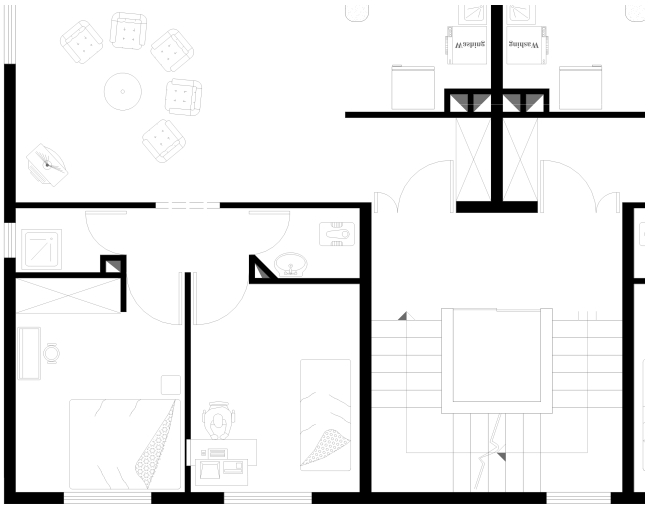


Figura 1.3: Ttrasformazione in scale di grigio.

Prima abbiamo detto che alcune planimetrie sono già in bianco e nero, ma questa operazione è necessaria lo stesso perchè quando l'immagine viene letta il salvataggio viene fatto in formato RGB, quindi ogni pixel

presenta una tupla con tre valori, relativi rispettivamente a rosso(R), verde(G) e blu(B), ed attraverso la trasformazione in scale di grigio ogni pixel presenta un solo valore, che varia da 0 (nero) a 255 (bianco), ed indica appunto il valore di grigio di quel pixel.

- **binarizzazione**, utilizzando la funzione

```
cv2.threshold(gray_image, threshold, maxValue, cv2.THRESH_BINARY)
```

viene impostata manualmente una soglia (threshold) e tutti i valori inferiori ad essa vengono messi a 0 (nero), mentre quelli maggiori vengono messi a 255 (bianco). In questo modo ogni pixel è o bianco o nero (*Figura 1.4*).
La soglia utilizzata è **250**, in modo che anche le sfumature dei colori, che saranno abbastanza vicine al bainco, vengano trasformate in nero, quindi in parte delle linee.

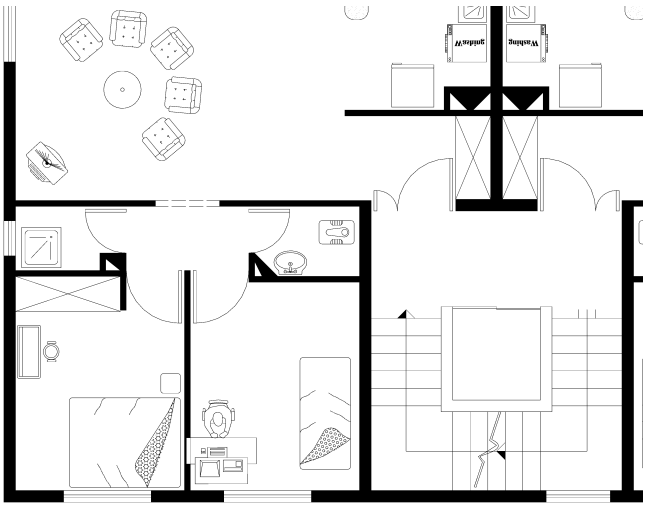


Figura 1.4: Binarizzazione.

- **erosione** e/o **dilatazione**, a causa dei passaggi appena descritti, o della eventuale bassa risoluzione dell'immagine, può succedere che delle linee (specialmente quelle oblique) perdano qualche pixel nelle precedenti trasformazioni, e quindi l'immagine risultante abbia delle linee con pixel mancanti in qualche punto, e questo può costituire un problema nei passaggi che vedremo successivamente.

  Quindi viene applicata l'erosione (del bianco) o la dilatazione (del nero) per colmare queste perdite e ottenere di nuovo delle linee continue (*Figura 1.5*).

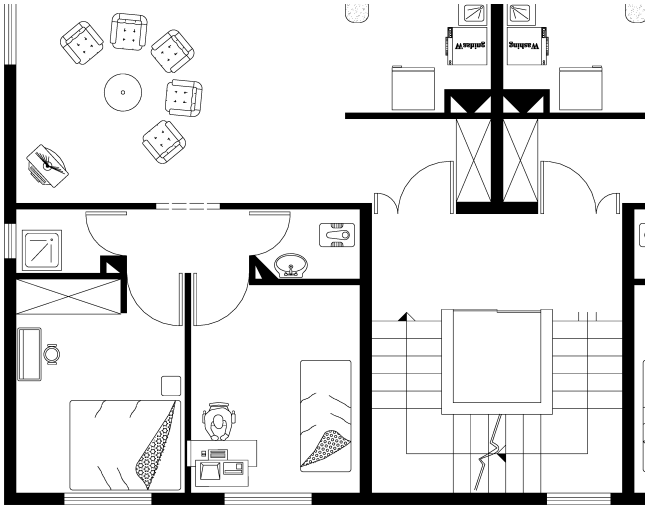


Figura 1.5: Planimetria erosa.

1.1 Componenti Connesse

A questo punto abbiamo trasformato l'immagine in un array numpy che presenta le caratteristiche desiderate.
L'operazione da svolgere ora è:

```
cv2.connectedComponents(image)
```

che prende in ingresso un array numpy binarizzato e ne restituisce uno in cui sono definite le sue componenti connesse.
L'array in output avrà le stesse dimensioni dell'array ricevuto in input, ma ogni elemento, che corrisponde ad un pixel, avrà un valore che va da 0 a n-1, con n = *numero di componenti connesse trovate*. Tutti i pixel che hanno un certo i valore appartengono alla stessa componente connessa.

Al fine di eseguire correttamente la ricerca delle componenti connesse è necessario che le linee non siano interrotte, altrimenti due componenti separate da una linea, però interrotta, risultano un'unica componente connessa. Per questo motivo è stata effettuata l'operazione di erosione/dilatazione.

A partire dall'array restituito dalla funzione sopra descritta, possiamo ricavare le singole componenti connesse ponendo a 255 (bianco) tutti i pixel con un certo valore i e a 0 (nero) gli altri. Un esempio di risultato che si ottiene è mostrato in *Figura 1.6*.

Iteriamo sul numero di componenti connesse facendo variare i e ad ogni iterazione estraiamo la corrispondente componente connessa, dopodiché calcoliamo il suo contorno, lo approssimiamo e se la sua area è all'interno di un range prescelto (in modo da scartare le componenti troppo grandi e quelle troppo piccole) viene creato un oggetto di tipo PossibleStep.

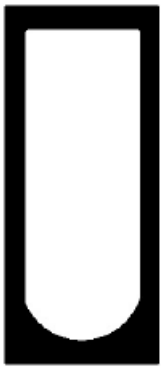


Figura 1.6: Esempio di componente connessa. Fonte:[3]

PossibleStep è una classe che è stata definita per descrivere le componenti candidate ad essere scale. Gli attributi che la descrivono sono:

- **descriptor**, un descrittore che indica se l'elemento è o non è un rettangolo
- **image**, l'immagine che rappresenta la componente connessa, ha la stessa dimensione dell'immagine iniziale
- **centroid**, il centro della componente connessa, trovato utilizzando funzioni di OpenCV
- **contour**, il contorno della componente, trovato ed approssimato grazie a funzioni di OpenCV.

1.1.1 Descrittore della Componente Connessa

Vediamo in dettaglio in cosa consiste l'attributo *descriptor* citato tra i parametri presenti nell'oggetto PossibleStep.

Come descritto nell'articolo[3] possiamo utilizzare diversi descrittori per, appunto, descrivere l'immagine ed ottenere dei parametri utilizzabili per ef-

fettuare confronti con altre immagini.
Dato lo scopo del nostro programma, il descrittore che risultava più appropriato è stato **contour descriptor** (naturalmente potevano essere utilizzati più descrittori, ma non avrebbero fornito nessun tipo di vantaggio).

Questo descrittore è utile perché permette di ottenere parametri che possiamo utilizzare per testare se la componente è considerabile un rettangolo. Per calcolarlo effettuiamo una serie di passaggi:

- trovare il *boundingBox* dell'immagine, cioè il box esterno che la contiene. Per fare questo ci siamo serviti di una funzione di OpenCV:

  ```
  cv2.boundingRect(image)
  ```

  che nello specifico ritorna il rettangolo che contiene l'immagine

- consideriamo l'angolo giro posto al centro dell'immagine, dividiamolo in S settori (*Figura 1.8*), nel nostro programma è stato preso S=45, come consigliato anche nell'articolo.

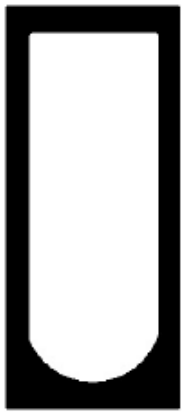


Figura 1.7: Componente connessa e suo BoundingBox. Fonte:[3]

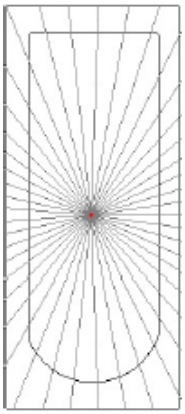


Figura 1.8: Componente connessa divisa in settori. Fonte:[3]

- consideriamo tutti i segmenti con cui è stata divisa l'immagine.
  Calcoliamo la distanza tra il centro ed il punto del contorno dell'immagine intersecato dal segmento, e la distanza tra il centro ed il punto del boundingBox intersecato.
  Salviamo questi due valori rispettivamente in due liste

A questo punto normalizziamo i valori, dividendoli tutti per il più grande, e tracciamo il grafico dei due descrittori (*Figura 1.9*): quello delle distanze dal contorno (in blu) e quello delle distanze del boundingBox (in rosso), ponendo nell'asse delle ordinate i valori normalizzati, e nelle ascisse il conteggio dei segmenti che generano i determinati valori.

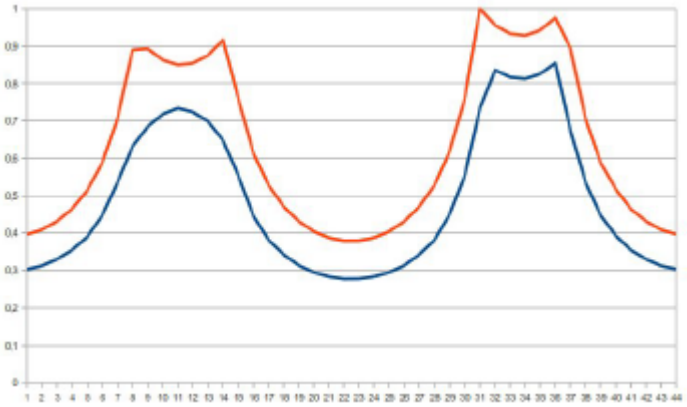


Figura 1.9: Descrittore della componente connessa in *Figura 1.7*. Fonte:[3]

Nell'esecuzione del nostro programma quello che ci serve è un criterio per stabilire se una componente è o meno un rettangolo. Ci serviamo quindi di questi profili per verificarlo.
Nell'articolo è anche descritto come calcolare la differenza tra due descrittori, quindi noi utilizziamo questa formula per calcolare la differenza tra il descrittore di una certa immagine e quello del suo boundingBox.

Se la loro differenza è minore di una certa soglia (0.2 nel nostro caso, valore che va bene per tutte le componenti perchè sono state normalizzate) possiamo considerare la componente come un rettangolo.
In realtà non tutte le componenti che restituiscono risultato positivo sono rettangoli, infatti potremmo vedere questo test come "*la componente è abbastanza vicina ad essere un rettangolo?*".

In questo modo, utilizzando un descrittore, possiamo stabilire se una componente è considerabile un rettangolo, e questo è utile perché gli elementi di una scala sono spesso dei rettangoli.

Capitolo 2

Grafo di Adiacenza

Dopo aver trovato tutte le componenti connesse dell'immagine e scartato quelle con dimensione troppo grande o troppo piccola per essere parte di una scala, possiamo costruire un grafo di adiacenza, cioè un grafo in cui ogni vertice è una componente connessa, nel nostro caso quindi ogni vertice è un oggetto di tipo PossibleStep.
Per stabilire invece se tra un vertice e l'altro c'è un arco andranno definiti dei criteri:

- distanza tra due componenti
- intersezione tra due componenti

Per la gestione del grafo ci serviamo della libreria **Networkx**[4], che offre operazioni per memorizzare nodi e archi, e permette anche di mostrarlo a video attraverso un'immagine.

2.1 Distanza tra due componenti

Per definire la distanza tra due componenti si deve innanzitutto trovare quali sono i lati adiacenti in questa coppia.

Per fare questo vengono considerate tutte le possibile coppie di componenti e tracciando una linea tra i due centri, vedere quali lati del contorno intersecano con essa. A questo punto si ottengono i lati adiacenti delle due componenti e quindi è possibile la loro distanza.

L'idea per fare il calcolo è quella di trovare la distanza di ognuno dei 4 vertici (2 di un lato e 2 dell'altro) rispetto all'altro lato della coppia e prendere poi il valore minore tra questi.

Si possono però trovare casi in cui questo passaggio va un po' modificato: se le componenti della planimetria non sono affiancate, quindi i lati non sono posti uno accanto all'altro, ma due vertici sono molto vicini tra loro (*Figura 2.1*).

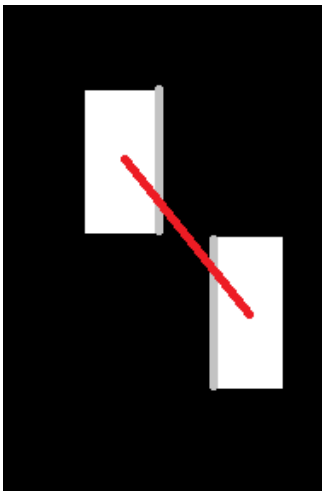


Figura 2.1: Componenti vicine, in rosso il segmento che congiunge i due centri ed in grigio i lati adiacenti.

In questo caso la distanza considerata sarà quella relativa ai due vertici ravvicinati, in realtà dovremmo stabilire quali sono le coppie di vertici, in modo da considerare, per calcolare la distanza, solo gli elementi della coppia. Ed in questo caso le coppie sarebbero (vertice in alto del lato a sx, vertice in alto del lato a dx) e (vertice in basso del lato a sx, vertice in basso del lato a dx).

In generale per trovare le coppie si considera, tra le possibili combinazioni, quella in cui le distanze tra gli elementi della coppia è minore.

Dopo aver ottenuto le coppie di vertici esatte, possiamo calcolare la distanza corretta tra i due lati, prendendo la coppia delle due che ha distanza minore ed impostare la loro distanza anche come distanza tra le due componenti.

Se la distanza tra le due componenti è maggiore di una certa soglia, scelta in questo caso come 6 px, si scarta questa coppia, altrimenti si procede nell'esecuzione del programma e si calcola l'intersezione.

2.2 Intersezione tra due componenti

Per trovare l'intersezione invece l'algoritmo è un po' più complesso.
L'idea che si segue è quella di calcolare quale è l'intersezione del *lato a* sul *lato b*, e viceversa, dopodiché prendere la minore dei due. Naturalmente i lati in questione sono i lati adiacenti delle due componenti.
Questo perchè se una componente è più piccola dell'altra e i due lati adiacenti sono paralleli e molto vicini (*Figura 2.2*), in un senso verrà che l'intersezione

è 100%, nell'altro invece risulterà quella esatta.

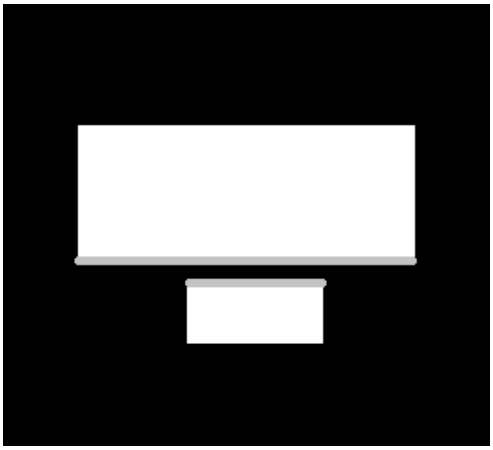


Figura 2.2: Componenti vicine con lati adiacenti di dimensione molto diversa

Per calcolare questa intersezione si prende la proiezione del *lato a* sul *lato b* e si calcola il rapporto tra la lunghezza del *lato a* e la lunghezza di questa proiezione.

Una volta calcolata anche l'intersezione si verifica che essa (che sarà come detto prima la minore delle due intersezioni trovate) sia maggiore di una certa soglia, in questo caso 40%, e se lo è allora si può creare l'arco tra queste due componenti, altrimenti no.

Il controllo dell'intersezione è necessario perchè si possono verificare casi in cui due componenti siano vicine, ma la loro differenza di dimensione renda impossibile che siano entrambi scalini, o comunque elementi di una stessa scala, quindi in base alla loro intersezione si stabilisce se poter generare un arco tra i due.

Al termine di questi passaggi il risultato sarà un grafo (*Figura 2.4*) che rappresenta tutti i possibleStep presenti nella planimetria data in input (*Figura 2.3*).

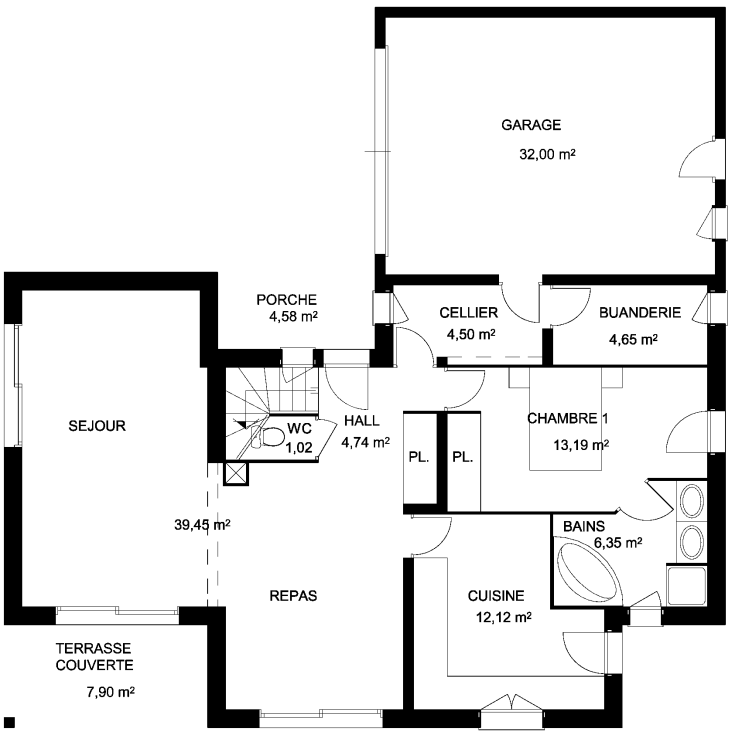


Figura 2.3: Planimetria del Dataset_1.

Osservando la *Figura 2.3* si può notare che i nodi hanno colori differenti:

- **giallo**, nodi che sono rettangoli (per come li abbiamo definiti precedentemente) e che hanno archi collegati
- **arancione**, nodi che non sono rettangoli ma sono collegati ad altri nodi tramite archi
- **rossi**, nodi che non hanno archi che li collegano ad altri nodi, sia rettangoli che non rettangoli.

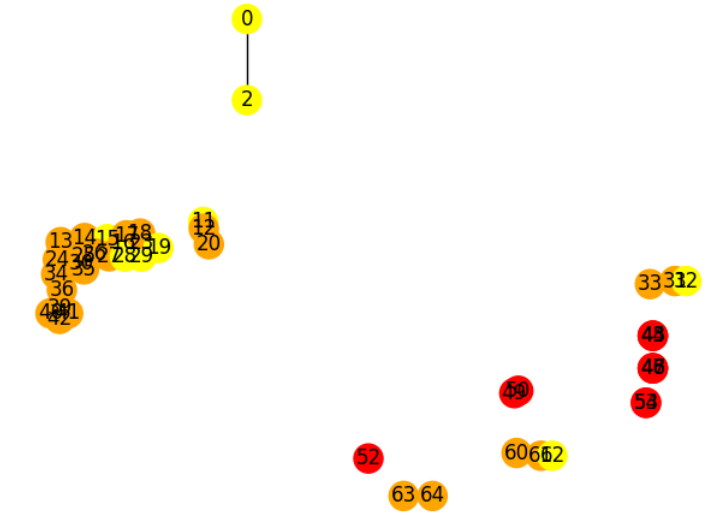


Figura 2.4: Grafo corrispondente alla planimetria.

La colorazione differente quindi dovrebbe esprimere diversi livelli di probabilità di essere parte di una scala.

2.3 Riduzione Grafo di Adiacenza

La modellazione, secondo quanto appena descritto, di un'immagine in un grafo è utile, perchè permette di lavorare l'immagine attraverso una struttura agevole, ma ai fini del risultato che vogliamo ottenere, avere un grafo complesso come quello mostrato in *Figura 2.4* non porta ancora a nessun risultato.
Dunque possiamo ridurre i nodi che compongono questo grafo per avvicinarci al nostro obiettivo.

Per la riduzione del grafo si effettuano pochi passaggi:

- utilizzando una funzione di Networkx si ricavano tutti i sottografi presenti nel grafo:

```
networkx.connected_component_subgraphs(graph, copy=False)
```

- iteriamo su tutti i sottografi trovati
- se il numero di nodi del sottografo è > 5, lo mantengo e vado al sottografo successivo
- se il numero di nodi è $<= 5$ si controllano le intersezioni tra le componenti: si contano il numero di archi che hanno intersezione $>= 0{,}9$, e se sono maggiori o uguali a 1 (se il numero di archi è < 4) o a 2 (se il numero di archi è 4 o 5) manteniamo questo sottografo, altrimenti lo scartiamo e le componenti che ne fanno parte vengono rimosse dal grafo principale.

Applicando questi passaggi al grafo mostrato in *Figura 2.4* otteniamo il grafo in *Figura 2.5*.

Quindi tutte le componenti che non sono parte della scala sono state scartate e sono riuscito a trovare correttamente la scala.
Nella *Figura 2.6* viene mostrato il risultato ottenuto con la riduzione del grafo, stampando le immagine rappresentate dai nodi rimanenti.

Non per tutte le planimetrie è sufficiente applicare la riduzione del grafo per ottenere un risultato corretto.
Quindi è necessario implementare strategie e meccanismi che riescano a restituire un risultato corretto anche quando la riduzione del grafo non è sufficiente.

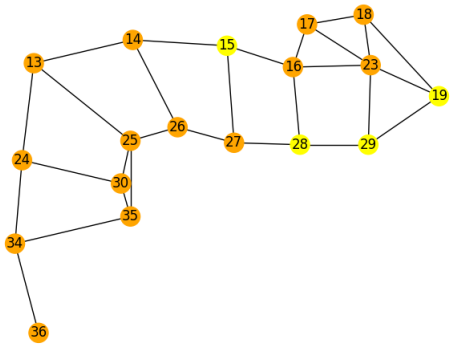


Figura 2.5: Grafo ridotto.

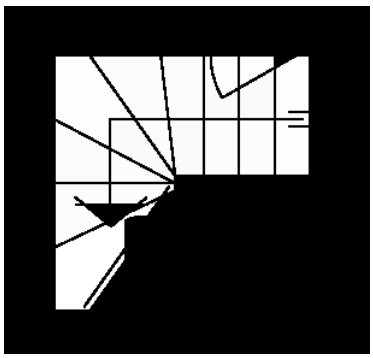


Figura 2.6: Immagine corrispondente al grafo ridotto

Capitolo 3

Rete Neurale

Al fine di ottimizzare i risultati del programma è stata inserita al suo interno una rete neurale.
Innanzitutto una Rete Neurale è un modello computazionale utilizzato nell'ambito dell'Intelligenza Artificiale. La sua struttura è composta da strati, ad ogni stato appartengono più nodi, che sono i **neuroni**, cioè le unità elaborative del sistema.

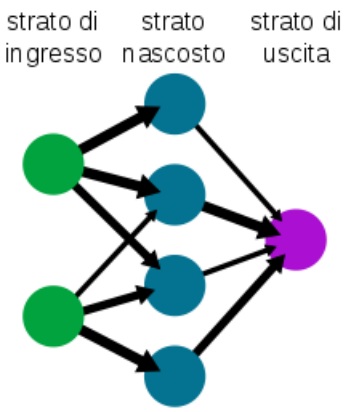


Figura 3.1: Rappresentazione semplice di una Rete Neurale. Fonte: Wikipedia[5]

Come mostrato in *Figura 3.1*, il numero di strati può essere approssimato a tre (ognuno dei quali però può coinvolgere migliaia di neuroni ed ancora più connessioni):

- **strato degli ingressi (Input)**: riceve ed elabora i segnali in ingresso adattandoli alle richieste dei neuroni della rete;
- **strato nascosto (Hidden)**: effettua l'elaborazione vera e propria;
- **strato di uscita (Output)**: raccoglie i risultati dell'elaborazione dello strato hidden e li adattata alle richieste del blocco successivo.

I dati che vengono posti in ingresso ad una rete neurale quindi subiranno una serie di passaggi:

- *formattazione* secondo quanto si aspetta lo strato di ingresso
- passaggio attraverso lo *strato di ingresso*, che come descritto precedentemente li rende compatibili con i livelli successivi
- passaggio attraverso gli *strati intermedi*, i quali effettuano l'elaborazione vera e propria, fornendo ai livelli successivi dati più evoluti, dettagliati
- infine i dati elaborati dagli strati precedenti passeranno dallo *strato di uscita*, che li rende significativi per il programma che contiene la rete e quindi utilizzabili per operazioni successive.

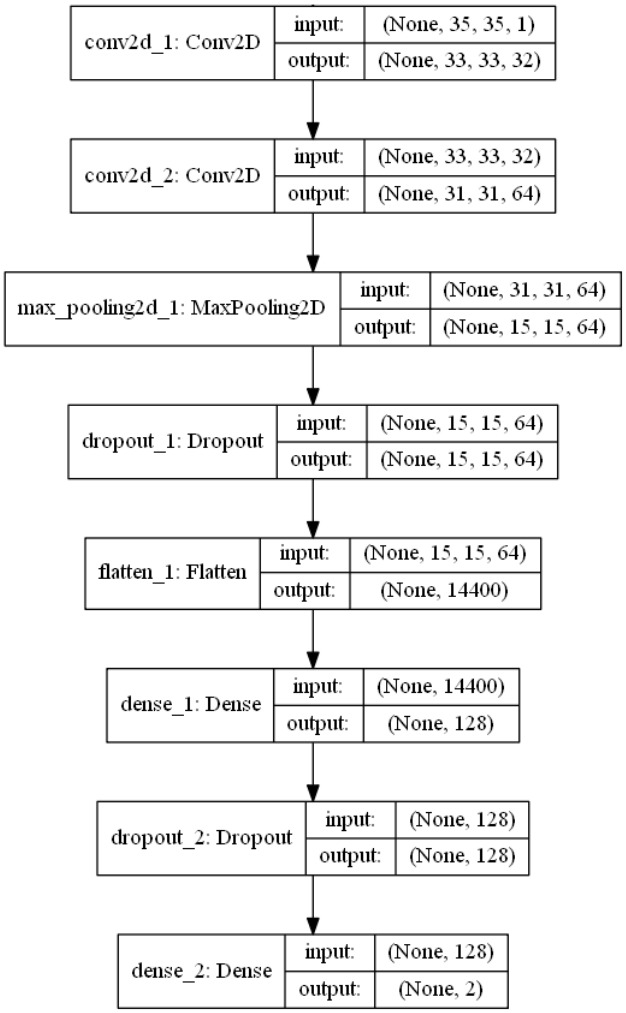
3.1 Rete Neurale nel nostro programma

Spiegato cosa è una rete neurale dal punto di vista più generico possibile, andiamo a vedere come è stata integrata nel nostro programma.

Innanzitutto per effettuare queste operazioni è stato utilizzato **Keras**[7], libreria per l'implementazione di reti neurali.

3.1.1 Architettura

L'architettura utilizzata nel nostro programma è la stessa usata per risolvere il problema della classificazione di caratteri scritti a mano MNIST[6].



3.1.2 Train

La prima operazione svolta dalla nostra rete neurale è quella di imparare a riconoscere, tra tutte le componenti connesse nelle planimetrie, quelle che sono parte di scale e quelle che invece non lo sono.
Quindi inizialmente viene effettuato il *Train*, cioè la fase in cui imparata queste distinzione.
Avendo due Dataset che presentano tipologie di planimetrie differenti, sono stati fatti due apprendimenti differenti, uno per dataset.

Le planimetrie utilizzate per l'apprendimento sono state scelte in base alle tipologie di scale che presentavano. Naturalmente le planimetrie presentavano diversi tipi di scale: tutte lineari, con una curva, con due curve, con pianerottolo, ecc... quindi è stata fatta una selezione per fare apprendere alla rete tutte le tipologie di scale, altrimenti si correva il rischio di non riconoscere determinate categorie.

Nell'esecuzione dell'operazione di Train, innanzitutto, è stato creato il *Training Set*, cioè due array in cui, nel primo sono presenti tutte le immagini delle componenti connesse, nel secondo la classe corrispondente ad ogni immagine. Le due classi possibili sono "Scalino" e "Non scalino", codificate rispettivamente con i codici "1" e "0".

Nella creazione del Training Set sono state estratte dalle immagini selezionate tutte le componenti connesse (come descritto precedentemente) e sono state ritagliate in modo da ottenere un'immagine quadrata con la componente centrata in essa.
Prima di ultimare l'operazione è stato fatto in modo che non ci fosse troppo

differenza tra il numero di componenti "Scalino" e "Non scalino", eliminando quindi quelle in eccesso se una categoria aveva dimensione molto maggiore dell'altra.
A questo punto tutte le immagini selezionate vengono portate ad una dimensione standard attraverso la funzione di OpenCV:

```
resize_img = cv2.resize(img, (IMG_ROWS, IMG_COLS))
```

che nel nostro caso è stata scelta essere 35x35 (px), e poi aggiunte al training set.

Quindi il training set è formato da due array:

- il primo contenente le immagini delle componenti connesse ritagliate e scalate ad una dimensione fissa
- il secondo contenente le corrispondenti classi di appartenenza di ogni componente.

Dopo aver creato il training set, questo viene suddiviso in *train* e *validation* secondo una proporzione 80/20, parte fondamentale per l'esecuzione del train. Dopodichè viene creata la rete neurale che dovrà apprendere in maniera autonoma come riconoscere le componenti o meno di una scala.

Al termine di questa operazione viene restituito un **modello**[8], salvato in un file con estensione *H5*. Un modello è una struttura che permette il successivo riconoscimento delle componenti.
Quindi questo modello viene salvato e vengono mostrati i risultati relativi a *loss* e *accuracy*, cioè i parametri che mostrano la perdita che si è avuta durante l'operazione e la sua accuratezza, questi parametri sono calcolati

tramite il validation set su cui viene effettuato il test, cioè viene effettuata la predizione utilizzando i dati del train e confrontata con la sua reale classe.

3.1.3 Prediction

Dopo la fase di apprendimento è necessario accedere al modello in cui sono stati salvati i dati ricavati, per effettuare delle predizioni su nuove componenti, non etichettate.

Anche in questo caso sarà necessaria una funzione che, date le componenti da testare, restituisca un *Test Set*. Il test set si differenzia dal training set perchè è composto soltanto dall'array con le immagini ritagliate e ridimensionate, in quanto dovendo effettuare una predizione non è possibile conoscere la classe reale di ogni componente.

Dunque, una volta ricavato il test set si carica il modello e si effettuano le predizioni, tutto utilizzando funzioni di *Keras*:

```
model = load_model('my_model.h5')
proba = model.predict_proba(test_set, batch_size=1)
```

la funzione che offre le predizione che è stata considerata più appropriata è *predict_proba* perchè restituisce, per ogni componente, la probabilità che ha di appartenere ad una classe o all'altra. Altrimenti si potrebbe effettuare una predizione che restituisce la classe a cui appartiene la componente, che semplicemente la sceglie in base a quale tra le due ha probabilità più alta.

Dopo l'esecuzione del test quindi abbiamo per ogni componente la probabilità che essa ha di appartenere ad una classe o all'altra. Per determinare la classe di ogni componente non ci affidiamo puramente alla percentuale dettata dal nostro classificatore, ma teniamo conto che esiste la possibilità

che alcune componenti siano parte di una scala anche se la probabilità che gli viene attribuita è inferiore al 0,5. Quindi quello che facciamo è scartare tutte le componenti la cui probabilità è molto bassa, al di sotto di una certa soglia (ad esempio 0,3) e, con le componenti restanti, costruire il grafo come descritto nel capitolo precedente.

In conclusione la rete neurale è un passaggio intermedio che si inserisce tra l'estrazione di tutte le componenti connesse dell'immagine ed il disegno del grafo delle componenti connesse, effettuando un'attività di filtraggio.

Capitolo 4

Valutazione dei risultati

4.1 Esecuzione del programma

Esistono due versioni del programma e le confronteremo per vedere i risultati migliori: una in cui non si utilizza la rete neurale ed una in cui le componenti, prima di essere usate per creare il grafo, vengono filtrate dalla rete neurale.
Vediamo adesso quali sono i passaggi che vengono eseguiti durante il flusso del programma, ricapitolando quello che è stato descritto fino ad ora:

- **lettura della planimetria** partendo dal suo percorso
- **elaborazione**, trasformazione in scale di grigi e successivamente binarizzazione
- **divisione in componenti connesse**, scartando quelle troppo piccole e quelle troppo grandi
- **(PRESENTE SOLO NELLA VERSIONE CON RETE NEURALE) filtraggio delle componenti** attraverso la rete neurali man-

tenendo solo quelle con probabilità di essere scalino al di sopra di una soglia prestabilita

- **grafo di adiacenza**, utilizzando tutte le componenti connesse trovate se si esegue la versione senza Rete Neurale, solo con le componenti rimaste se prima si filtrano con la Rete Neurale
- **scarto delle componenti connesse del grafo** che non rispettano determinati parametri
- **mostrare il risultato trovato sotto forma di immagine**, aggregando tutte le componenti connesse che sono rimaste nel grafo.

Quindi i flussi di esecuzione delle due versioni variano solo per la presenza o meno del filtraggio delle componenti connesse da parte della rete neurale.

4.2 Metriche

Per testare i risultati ottenuti dopo l'esecuzione del programma si deve confrontarli con le etichettature effettuate precedentemente, e per fare queste operazione utilizziamo il **Coefficiente di Jaccard**.

4.2.1 Coefficiente di Jaccard

L'*Indice di Jaccard* è un indice statistico, utilizzato per confrontare due insiemi e testarne la similarità e diversità.
Il **Coefficiente di Jaccard** misura la loro similarità ed è definita come la dimensione dell'intersezione dei due insiemi divisa per la dimensione dell'u-

nione:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Nel nostro caso gli insiemi A e B rappresentano rispettivamente i pixel bianchi nell'immagine risultante dal nostro programma e nell'immagine risultante dall'etichettatura effettuata con *Sloth*. Quindi attraverso le funzioni rese disponibili da OpenCV si crea un'immagine che rappresenta l'intersezione delle due ed una che rappresenta l'unione delle due. A questo punto si conta il numero di pixel bianchi in ognuna della due immagini e si fa il rapporto.

E' molto importante l'utilizzo di questo parametro per la valutazione dell'operato perchè permette di valutare il risultato considerando soltanto le parti che sono state trovate con esattezza.
Se si considerassero per esempio soltanto le aree delle due immagini e l'indice di similarità fosse dato dal rapporto di queste aree, potrei trovare un risultato soddisfacente anche se il risultato del programma restituisse delle aree che non corrispondo alle scale ma con area di dimensione simile.
Utilizzando come indice il coefficiente di Jaccard evitiamo errori di questo tipo poichè l'intersezione risulterebbe nulla e quindi sarebbe nullo anche il rapporto, che indica similarità nulla.

Il valore restituito dal calcolo avrà un valore contenuto nell'intervallo $[0, 1]$, con 0 che indica similarità nulla ed 1 che indica similarità totale.

4.2.2 Composizione di componenti connesse

Nell'esecuzione del test si può riscontrare un problema: il programma trova correttamente una scala, ma il grafo che la costituisce è composto da più componenti connesse separate.
Quando facciamo il test su una certa planimetria dobbiamo testare ogni singola componente connessa del grafo risultante con ogni scala etichettata per quel corrispondente file, perchè ogni planimetria può presentare più di una scala. Non posso quindi confrontare il risultato di tutte le componenti del grafo con l'isieme di tutte le etichette, devo fare una valutazione scala per scala. In più non possiamo sapere quale componente del grafo corrisponde ad una certa scala, quindi ogni componente trovata si testa con ogni scala etichettata e si prende quella in cui il risultato è maggiore.
Come detto, però, possiamo effettuare test in cui il grafo relativo ad una certa scala sia diviso in più parti, in questi casi si dovrà operare per *composizione*.
Andiamo quindi a vedere come si svolge il test per evitare di riscontrare i problemi descritti:

- itero su tutte le componenti connesse del grafo
- eseguo il test su ognuna e calcolo il migliore coefficiente di Jaccard
- salvo questo valore come risultato temporaneo del test
- si effettuano i test sulle composizioni:
  - si sommano due componenti
  - se il test su questa nuova componente dalla somma restituisce risultato migliore rispetto al valore di entrambi i risultati temporanei, unisco le due componenti e considero questo ultimo valore calcolato come suo coefficiente di Jaccard

- effettuo le somme tra tutte le componenti in modo da eseguire il test su tutte le combinazioni
- al termine di questa iterazione ne eseguo un'altra: nel caso in cui una nuova componente ottenuta vada sommata ad altre
- termino le iterazioni quando non ho ottenuto unioni di componenti nell'ultima iterazione

4.2.3 Valutazione dell'Algoritmo

Dopo aver ottenuto tutti i risultati dei test su ogni planimetria dovranno essere messi insieme i risultati per valutare l'algoritmo.
Per la valutazione delle performance ci basiamo su due parametri: **Precision** e **Recall**.

$$Precision = \frac{\#trovati\_giusti}{\#trovati} \qquad Recall = \frac{\#trovati\_giusti}{\#giusti}$$

$\#trovati$ = numero di elementi restituiti dal programma come possibile scala
$\#giusti$ = numero di scale presenti (etichettate)

Per quanto riguarda i $\#trovati\_giusti$ si deve stabilire una soglia, dopodiché tutti gli elementi trovati che hanno un coefficiente di Jaccard maggiore della soglia sono considerati $\#trovati\_giusti$.
$\#trovati\_giusti$ = numero di scale trovate con Coefficiente di Jaccard $>$ threshold

Capitolo 5

Risultati

Il programma, come detto precedentemente, è stato utilizzato su due dataset, che differivano in termini di dimensioni delle planimetrie, tipologia di edificio rappresentato e modalità di disegno.
Di conseguenza sono stati effettuati test su entrambi i Dataset per vedere come si comporta l'algoritmo se eseguito su planimetrie con caratteristiche differenti.
Di conseguenza anche i risultati riportati in questo capitolo saranno divisi a seconda del Dataset testato, ed ognuno dei due è testato in entrambe le versioni del programma.

Prima però descriviamo i valori ritenuti più rilevanti per il test e il perchè siano stati mostrati proprio questi.
La tabella che contiene i risultati ha come obiettivo quello di mostrare i valori relativi al *numero di scale trovate giuste*, *Precision* e *Recall* al variare della soglia *Jaccard_ Threshold*, che definisce quando un elemento trovato può essere riconosciuto come scala. Minore è questa soglia e maggiore sarà il numero di elementi che vengono riconosciuti come giusti e maggiori saranno

i valori di Precision e Recall.

I valori di *Jaccard_ Threshold* considerati sono nell'intervallo [0.6, 0.8] per quanto riguarda i testi sul Dataset_1 e nell'intervallo [0.6, 0.9] per il Dataset_2. Le scelte sono ricadute su questi intervalli perchè sono le soglie che rispecchiano maggiormente la validità dell'esecuzione dell'algoritmo.
Infatti quando si etichetta una scala, l'area che viene considerata è tutta quella compresa nei bordi esterni (*Figura 5.2*), mentre le componenti connesse dell'immagine della scala presentano una divisione tra l'una e l'altra (*Figura 5.1*), quindi si forma una piccola differenza tra il numero di pixel dell'immagine etichettata ed il numero di quella trovata, anche se questa venisse trovata perfettamente.

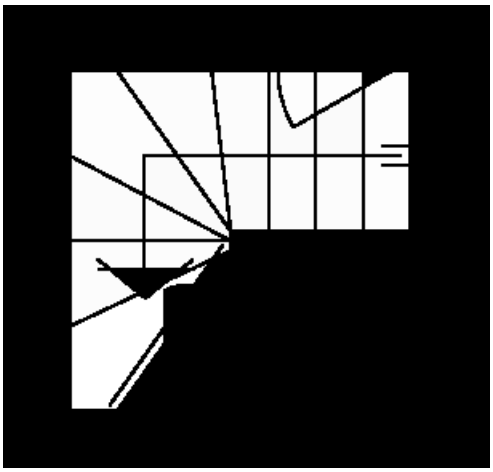


Figura 5.1: Scala trovata dal programma

In più come detto prima viene effettuata anche l'erosione del bianco per evitare di unire alcune componenti connesse, quindi questa differenza aumenta maggiormente.

Figura 5.2: Scala etichettata

Inoltre all'interno della planimetria ci sono anche altre linee, che vanno ad intersecare la scala e formano piccole componenti che spesso hanno un area molto piccola e rientrano tra quelle che vengono scartate.

Per questo motivo gli indici *Jaccard_ Threshold* considerati per il Dataset_2 hanno un intervallo che contiene valori più alti, perchè dato che le planimetrie hanno una dimensione maggiore, le parti che dividono le componenti connesse dell'immagine incidono molto meno ed i parametri di valutazione sono accettabili anche se la soglia è posta più in alto.
Infine il risultato che si ottiene è:

- una presunta scala che presenta un indice di similarità con una scala etichettata > 0.75 è ritenuta una scala completa (al netto delle piccole componenti appena citate)
- una presenta scala che presenta un indice di similarità > 0.6 è una scala alla quale probabilmente manca un elemento, per esempio uno scalino, ma è comunque riconoscibile come scala.

Quindi se prendessi come soglia valori maggiori scarterei scale che sono state trovate correttamente, se prendessi valori minori accetterei scale a cui mancano troppe parti per ritenere che il programma ha operato correttamente ed abbia fornito un risultato soddisfacente.

5.1 Dataset_1

5.1.1 Senza Rete Neurale

Vediamo i risultati per il Dataset_1 utilizzando la versione dell'algoritmo senza rete neurale:

Jaccard_Threshold	#Trovati_giusti	Precision	Recall
0.6	35	0.83	0.97
0.65	33	0.79	0.92
0.7	31	0.74	0.86
0.75	29	0.69	0.81
0.8	22	0.52	0.61

#Trovati = 42

#Giusti = 36

5.1.2 Con Rete Neurale

Passiamo ai risultati, sempre per il Dataset_1, con rete neurale:

Jaccard_Threshold	#Trovati_giusti	Precision	Recall
0.6	34	0.85	0.94
0.65	33	0.83	0.92
0.7	27	0.68	0.75
0.75	23	0.58	0.64
0.8	13	0.33	0.36

#Trovati = 40

#Giusti = 36

Confrontando le due tabelle si evidenzia che la rete neurale tende ad escludere probabilmente troppe componenti, portando *Precision* e *Recall* a livelli inferiori rispetto alla versione senza rete neurale. Unica eccezione è la *Precision* per soglia bassa, quindi significa che i risultati dei test sono molto concentrati sul intervallo [0.6, 0.7), infatti successivamente si nota un crollo evidente dei risultati.

Nella versione senza rete neurale invece i risultati sono molto più omogenei all'aumentare della soglia impostata, questo evidenzia che i risultati dei test sono concentrati su valori più alti.

Quindi i risultati trovati, in termine di scale, per il Dataset_1, sono più precisi e completi se si utilizza la versione del programma senza rete neurale.

5.2 Dataset_2

5.2.1 Senza Rete Neurale

Risultati relativi al test del Dataset_2 senza rete neurale:

Jaccard_Threshold	#Trovati_giusti	Precision	Recall
0.6	30	0.73	0.94
0.65	28	0.68	0.875
0.7	28	0.68	0.875
0.75	27	0.65	0.84
0.8	26	0.63	0.81
0.85	26	0.63	0.81
0.9	25	0.61	0.78

#Trovati = 41 #Giusti = 32

5.2.2 Con Rete Neurale

Risultati del test al Dataset_2 con rete neurale:

Jaccard_Threshold	#Trovati_giusti	Precision	Recall
0.6	26	0.78	0.81
0.65	23	0.70	0.72
0.7	23	0.70	0.72
0.75	23	0.70	0.72
0.8	22	0.67	0.69
0.85	20	0.61	0.625
0.9	20	0.61	0.625

#Trovati = 33 #Giusti = 32

Confrontando i risultati delle due versioni si nota che nella versione senza rete si ottengono risultati migliori per quanto riguarda la *Recall*, mentre nella versione in cui è presente la rete è migliore la *Precision*.

Questo risultato è dovuto al fatto che la rete, con il filtraggio delle componenti, riesce ad escluderne alcune in maniera corretta e quindi ad evitare che il programma restituisca scale che in realtà non lo sono (per questo la *Precision* è migliore), d'altra parte in alcuni casi vengono escluse componenti che in realtà sono parte di scale (per questo la *Recall* è peggiore.

Questo errore che si riscontra è una caratteristica dell'apprendimento automatico: che per definizione presenta errore.

Capitolo 6

Conclusioni

In conclusione possiamo dire che il programma fornisce buoni risultati, perchè in quasi la totalità delle planimetrie viene riconosciuta la scala se presente e quasi tutti gli oggetti che vengono trovati sono scale.
Come detto precedentemente, nell'esecuzione delle planimetrie del Dataset_1 il programma non trae beneficio dalla presenza della rete neurale perchè i risultati evidenziano performance peggiori, invece prendendo in considerazione il Dataset_2 migliora sensibilmente l'utilità della rete neurale, specialmente per quanto riguarda la *Precision*.

Quindi possiamo concludere dicendo che il programma nella versione con rete neurale si presta maggiormente a planimetrie con un numero di oggetti presenti maggiori, come certifica il Dataset_2, perchè il training permette di avere una conoscenza più ampia e questo si riflette quando si effettua la predizione.

6.1 Futuri Sviluppi

Per dei futuri miglioramenti dell'algoritmo, al fine di raggiungere dei risultati migliori, si potrebbe procedere con uno studio più approfondito delle planimetrie da utilizzare per la fase di Train. In questo modo si riuscirebbero ad offrire al programma conoscenza più ampie che porterebbero sicuramente a perfomance migliori.

Altro passo da eseguire potrebbe essere quello di potenziare il funzionamento della rete neurale: non utilizzarla esclusivamente per la classificazione delle singole componenti connesse, ma per classificare più componenti connesse insieme. In modo da ottenere che una componente sia parte della scala anche se la probabilità che le si assegna è bassa, perchè ha archi che la collegano a componenti che sono parte di una scala.
In questo modo si dovrebbe riuscire ad includere delle componenti che con la versione attuale potrebbero venir escluse.

Riferimenti

[1] Sloth source, *https* : *//bitbucket.org/thecape/sloth*$_v$*2/src/master*

[2] OpenCV Guide, *https* : *//docs.opencv.org*/3.4.2/*index.html*

[3] A. Barducci and S. Marinai, *"Object Recognition in Floor Plans by Graphs of White Connected Components"*, 2012

[4] Networkx, *https* : *//networkx.github.io/*

[5] Neural Network, Wikipedia, *https* : *//it.wikipedia.org/wiki/Rete_neurale_artificiale#/media/File* : *Neural_network_example_it.svg*

[6] MNIST on Keras, *https* : *//github.com/keras − team/keras/blob/master/examples/mnist_cnn.py*

[7] Keras, *https* : *//keras.io/*

[8] Keras Model, *https* : *//keras.io/getting − started/faq/#how − can − i − save − a − keras − model*