Sistema di detection dei difetti delle piastrelle magnetiche

Pagnini Lorenzo - 0000942265
{lorenzo.pagnini2@studio.unibo.it}

Giugno 2021

Indice

1 Introduzione **2**

2 Requisiti **4**

3 Dataset utilizzato **5**

4 Approccio 1: Tecniche di Visione Artificiale **7**
4.1 Fase di pre-processing 7
4.2 Identificazione dei difetti 9
4.2.1 Detection dei cracks 10
4.2.2 Detection dei blobs 10
4.3 Esempi di utilizzo 11
4.4 Analisi dei risultati ottenuti 13
4.4.1 Risultati 13
4.4.2 Analisi degli errori 15

5 Approccio 2: Rete Neurale **17**
5.1 U-Net 17
5.2 Addestramento e risultati ottenuti 18

6 Codice **22**

7 Conclusioni **23**

Capitolo 1

Introduzione

L'obbiettivo di questo elaborato è la realizzazione di un sistema software per il rilevamento automatico dei difetti delle piastrelle. Tali difetti incidono negativamente sulla qualità e quantità di produzione da parte di un'azienda. Per ridurre le perdite, molte industrie impiegano controlli di tipo visivo effettuati da operatori umani. Appare evidente che questi compiti si prestano bene per essere svolti da sistemi di detection, in grado di catturare immagini, grazie all'ausilio di telecamere, per automatizzare il processo di rilevamento. In questo modo vengono ridotti tempi, costi e tasso di errore dovuto ad esempio alla stanchezza e perdita di concentrazione degli operatori umani. Per garantire tutto ciò è necessario che questi sistemi presentano una elevata accuratezza in grado di minimizzare l'errore, per ottenere un vantaggio significativo nel controllo della qualità rispetto all'intervento umano.

In questo progetto si è deciso di adottare due approcci differenti per raggiungere lo stesso obbiettivo: rilevare i difetti. La scelta di utilizzare due approcci differenti è dettata dal fatto che sempre più, negli ultimi decenni, si fa uso dell'intelligenza artificiale per risolvere problemi legati al mondo della visione artificiale. Tale metodologia, non era invece possibile applicarla negli anni 90' poiché non si disponeva di grandi quantità di dati per poter allenare le reti, così si ricorreva solamente all'uso di particolari operazioni eseguite sulle immagini in grado di ottenere features di interesse. Per completezza, ho deciso quindi di applicare entrambe le metodologie.

Il primo approccio (capitolo 4) si basa sull'utilizzo di tecniche di visione artificiale che analizzano le immagini attraverso processi di pre-elaborazione con l'impiego di diverse tipologie di filtri e, successivamente identificano i difetti estraendo le features di alto livello. I risultati sono stati successivamente messi a confronto per individuare il miglior sistema di detection tra quelli realizzati.

Il secondo approccio (capitolo 5) si basa sull'implementazione di una rete neurale. E' da precisare che anche in questo approccio si fa uno di tecniche di visione artificiale, sopratutto in fase di pre-processing, ma l'obiettivo del problema viene risolto mediante l'utilizzo di una rete neurale.

A questo *link* è possibile consultare e scaricare il codice sorgente.

Capitolo 2

Requisiti

L'obiettivo del progetto è la realizzazione di un sistema software di visione artificiale da impiegare nel controllo qualità delle industrie che producono piastrelle. In questa sezione vengono descritti i principali requisiti che dovranno essere soddisfatti divisi per approccio. Mediante l'utilizzo delle sole tecniche di visione artificiale, si hanno i seguenti requisiti:

- Identificare i difetti utilizzando tecniche e approcci rapidi;
- Identificare le diverse tipologie dei difetti delle piastrelle;
- Identificare il contorno o la forma del difetto interessato sulla superficie della piastrella.

Per l'approccio basato su una rete neurale i requisiti sono:

- Identificare i difetti restituendo in output la maschera evidenziando il difetto, se presente.

Capitolo 3

Dataset utilizzato

Il dataset utilizzato per lo sviluppo di questo progetto è consultabile al seguente *link* [1]. L'idea iniziale era quello di eseguire la detection su piastrelle di ceramica ma questo non è stato possibile a causa dell'assenza, in rete, di dataset su cui testare il sistema. Per questo motivo si è deciso di utilizzare questo dataset contenenti immagini di piastrelle magnetiche.

Il dataset menzionato è organizzato in diverse directory, ognuna delle quali contiene immagini in scala di grigio e relative maschere binarie di una particolare tipologia di difetto delle piastrelle. Le tipologie di difetti presenti sono:

- **Blowhole**: difetto identificato da un punto isolato (come una goccia) o una macchia irregolare sulla superficie della piastrella simile ad una circonferenza o ellisse;
- **Crack**: difetto identificato da una crepa netta sulla superficie della piastrella;
- **Uneven**: superficie della piastrella irregolare;
- **Break**
- **Fray**
- **Free**: piastrelle non difettose.

Per maggiori dettagli si veda la figura 3.1 e le immagini del dataset. In questo elaborato si è deciso di analizzare due principali difetti superficiali delle piastrelle magnetiche: crack e blob.

Disponendo solo di immagini in scala di grigio (e non RGB), la detection è risultata ancora più difficile in quanto non è stato possibile eseguire l'analisi del colore per migliorare la rilevazione dei difetti.

Inoltre alcune directory contengono immagini dello stesso difetto differenziandosi solo per la quantità di luce catturata dal sensore.

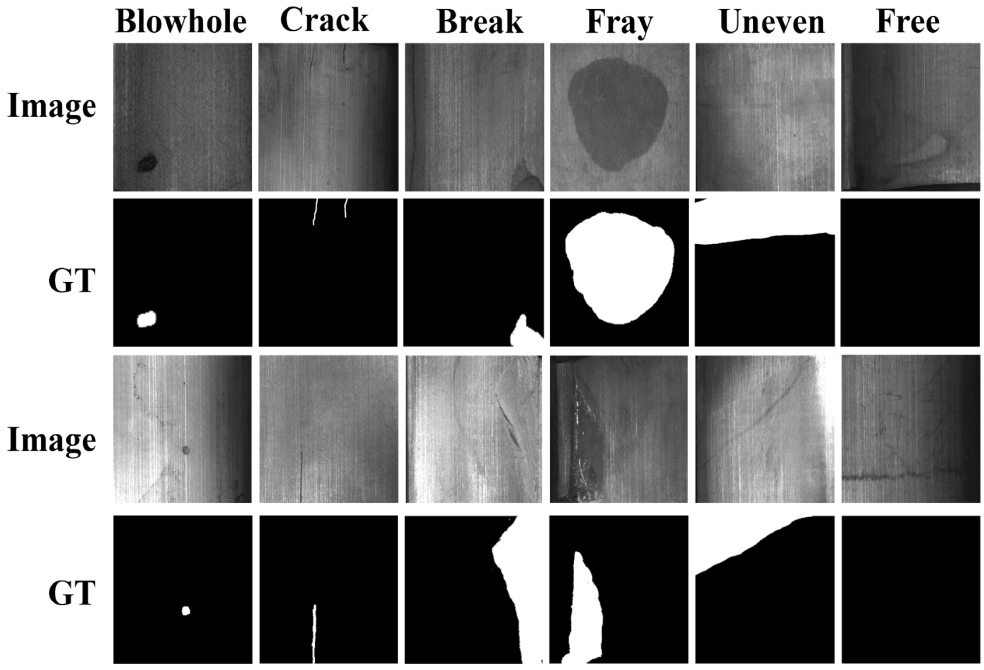

Figura 3.1: Esempi di tipologie di difetti del dataset

Capitolo 4

Approccio 1: Tecniche di Visione Artificiale

Questo capitolo descrive le fasi impiegate per il primo approccio ovvero utilizzando tecniche di visione artificiale. Verranno quindi illustrati nel dettaglio, tutti i processi coinvolti per l'individuazione ed estrazione delle features e successiva fase di identificazione del difetto/i.

4.1 Fase di pre-processing

La prima fase che il sistema mette in atto è chiamata pre-processing. Questa fase ha come obbiettivo principale quello di normalizzare l'immagine, ridurre il rumore presente ed evidenziare gli edge. Le operazioni sono eseguite nel seguente ordine:

1. **Conversione in scala di grigi.**

   Inizialmente l'immagine viene convertita in scala di grigi (poiché l'immagine presenta 3 canali anche se in bianco e nero) in modo tale che le varie operazioni possono essere eseguite in maniera più semplice;

2. **Normalizzazione dell'immagine**.

   I valori di grigio delle immagini con elevata luminanza vengono troncati nel range compreso [0, 170]. Per tutte le altre immagini viene eseguita una normalizzazione compresa tra [0, 255]. Questa operazione viene utilizzata per migliorare il contrasto dell'immagine poiché permette di allungare l'intervallo dei valori di intensità dei pixel. L'idea alla base di questa operazione è quella di aumentare la gamma dinamica del livello di intensità nell'immagine elaborata, così da far emergere i difetti presenti;

3. **Filtraggio.**

   L'immagine viene filtrata applicando un filtro tra quello gaussiano, mediano e quello bilaterale (nell'ultima sezione di questo report sono stati confrontati i risultati ottenuti con l'applicazione dei diversi filtri).

   Il filtro gaussiano è un filtro lineare che sfoca un'immagine con una forma a campana rappresentata dalla sua distribuzione normale.

   Il filtro mediano è un filtro non lineare, molto popolare dato dalla capacità di ridurre il rumore e di non alterare i contorni. Si basa sullo scorrimento di una finestra su una porzione di un'immagine. Successivamente viene calcolata la mediana, tra tutti i valori dei pixel compresi nella finestra, e assegnata al valore del pixel centrale.

   Il filtro bilaterale è anch'esso un filtro non lineare molto popolare ma più lento rispetto agli altri. Utilizza due filtri gaussiani: uno assicura che solo i pixel vicini siano considerati per la sfocatura, l'altro fa in modo che solo quei pixel con intensità simili al pixel centrale siano considerati per la sfocatura;

4. **Denoising dell'immagine**

   Il denoising dell'immagine è stato effettuato utilizzando l'algoritmo *non-local means denoising* [2]. In sintesi, il metodo funziona sostituendo il valore di un pixel con una media dei valori dei pixel simili in varie porzioni dell'immagine.

   L'algoritmo si basa su una proprietà: il rumore è considerato come una variabile casuale con media zero. Considerando un pixel affetto da rumore

   $$p = p_0 + n$$

   dove p_0 è il vero valore del pixel e n il rumore in quel pixel. Se si considera un numero elevato di pixel uguali (ad esempio N) da immagini diverse e si calcoli la loro media, si ottiene che $p = p_0$ poiché la media del rumore è zero. Considerando un pixel di un'immagine, si prenda una piccola finestra intorno e si cerchi finestre con patch simili nell'immagine. A questo punto si esegue la media di tutte le finestre e si sostituisce il valore del pixel con il risultato ottenuto.

5. **Edge detection.**

   Questa operazione permette di identificare i bordi all'interno dell'immagine. Per questo progetto si è deciso di utilizzare il metodo Canny in cui viene calcolato il gradiente di intensità dell'immagine priva di rumore (filtrata) e viene applicata una soppressione non massima al gradiente di

intensità per rimuovere i pixel indesiderati che potrebbero non costituire un bordo;

6. **Applicazioni tecniche morfologiche**

   Al risultato ottenuto dai precedenti step viene applicato una serie di operazioni morfologiche come la dilatazione, l'erosione che rappresentano un'operazione di chiusura. Il risultato è un riempimento di buchi e di piccole concavità, rafforzando la connessione di regioni unite debolmente.

Il risultato della fase di pre-processing è un'immagine binaria che verrà usata per identificare le diverse tipologie di difetti.

La fase di pre-processing è il risultato di un insieme di prove, utilizzando altre metodologie: per la sogliatura si era utilizzato inizialmente quella di Otsu, ma poichè si ottengono buoni risultati solo per immagini bimodali (immagini che presentano due picchi nell'istogramma) si è deciso di non prenderlo in considerazione. Altri tentativi sono stati effettuati utilizzando Sobel per la edge detection e la modifica della gamma di un immagine per evidenziare i difetti.

4.2 Identificazione dei difetti

La fase di identificazione dei difetti prevede le seguenti analisi:

- Ricerca delle componenti connesse;
- Analisi delle caratteristiche geometriche di ogni componente.

La ricerca delle componenti connesse risulta essere la medesima per entrambi i difetti mentre l'analisi delle caratteristiche geometriche si differenzia a seconda della tipologia del difetto da identificare.

Ricerca delle componenti connesse

Inizialmente si cercano tutte le componenti connesse presenti nell'immagine. L'algoritmo utilizzato, per la ricerca delle componenti, esegue una ricerca in profondità (DFS - *depth-first-search*). In particolare controlla ogni valore di ogni pixel nell'immagine: se è uguale a 1 allora considera i valori dei suoi 8 pixel vicini e trova quali sono uguali a 1 in maniera ricorsiva, fino alla ricerca di tutta la componente. Per fare ciò, l'immagine viene precedentemente normalizzata nel range di valori [0,1]. Durante la ricerca, l'algoritmo calcola la dimensione della componente così da poter eseguire una prima selezione dei difetti da considerare per l'analisi successiva, scartando componenti di piccole dimensioni.

Con tale ricerca, si ottengono anche le coordinate di tutti i pixel coinvolti nella componente. Questo permette di eseguire un'altra selezione considerando l'intensità media della componente (ottenuta grazie ai pixel precedentemente individuati) rispetto all'intensità media di tutta l'immagine originale. Se

la componente ha un'intensità media minore rispetto all'immagine allora può essere una candidata per classificarla come difetto.

4.2.1 Detection dei cracks

Come accennato in precedenza un crack è una crepa che può interessare tutta o solo una parte della superficie della piastrella.

Analisi delle caratteristiche geometriche

Dopo la ricerca delle componenti segue la fase di analisi che esamina le caratteristiche geometriche. Tramite la funzione *findContours()* di OpenCV si ottengono i contorni di ogni componente. Per ognuno di questi viene calcolata l'area, così da scartare piccoli contorni, perimetro e la circolarità (per una linea è pari a 0).

Se vengono individuati dei cracks, questi vengono disegnati nell'immagine originale e il risultato di questa detection viene sotratto (tramite una sottrazione bit a bit) all'immagine ottenuta dalla fase di pre-processing così che la successiva fase di identificazione non consideri difetti già individuati.

4.2.2 Detection dei blobs

Questa fase si pone come obbiettivo quello di individuare i blob sulla superficie della piastrella. Un blob può essere definito come un punto, cerchio o una macchia irregolare più scura rispetto al colore della piastrella.

Analisi delle caratteristiche geometriche

Prima di identificare i contorni delle componenti rimanenti, vengono eseguite alcune operazioni morfologiche: dilatazione e chiusura. Successivamente si ottengono i contorni per eseguire un analisi geometrica della componente. I primi parametri calcolati sono area, perimetro e la circolarità (per un cerchio regolare è pari a 1). Se non vengono identificati dei cerchi si cerca di individuare degli ellissi. Per quest'ultime viene calcolata l'eccentricità che per un ellisse è maggiore di 0.8.

Se vengono individuati dei blobs, questi vengono disegnati nell'immagine originale.

4.3 Esempi di utilizzo

Per interfacciarsi con il sistema di detection ed apprezzare i risultati tra i vari filtri messi a disposizione, è stata creata una piccola GUI tramite la libreria *tkinter* di python. Tramite la GUI è possibile caricare l'immagine da dare in input al sistema e scegliere il filtro da utilizzare.

Terminata la detection il sistema mostrerà una griglia contenenti le seguenti immagini:

- Immagine originale data in input;
- Immagine con cracks rilevati (se presenti) colorati di verde, e relativa maschera binaria utilizzata dal sistema per identificare il difetto;
- Immagine con blobs rilevati (se presenti) colorati di rosso, e relativa maschera binaria utilizzata dal sistema per identificare il difetto;
- Istogramma dei livelli di grigio dell'immagine di input.

Per poter identificare le varie immagini nella griglia, è stato aggiunto un piccolo rettangolo bianco, in basso all'immagine, in cui inserire la descrizione o tipo di difetto rilevato.

Nella GUI, sono presenti i pulsanti *upload*, *start* e *cancel* rispettivamente per caricare un'immagine nel sistema, iniziare la detection e chiudere il programma. I filtri sono selezionabili tramite i check-boxs. Inoltre viene mostrato il tempo impiegato dal sistema per eseguire la detection e vengono mostrati dei messaggi di informazione (blu) e di errore (rosso) a seconda dell'azione intrapresa con la finestra di interfacciamento. Si veda la figura 4.1.

A seguire vengono mostrati alcuni esempi di detection di crack e blob (figura 4.2). Per motivi di spazio, nelle immagini della relazione sono state omesse gli istogrammi.

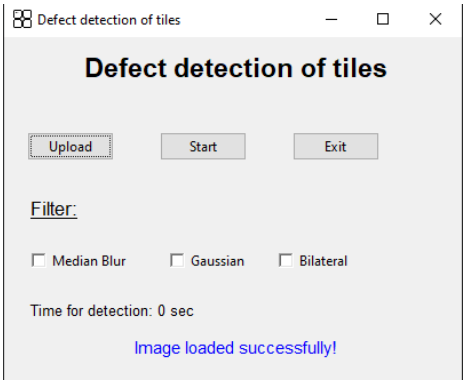

Figura 4.1: GUI del sistema

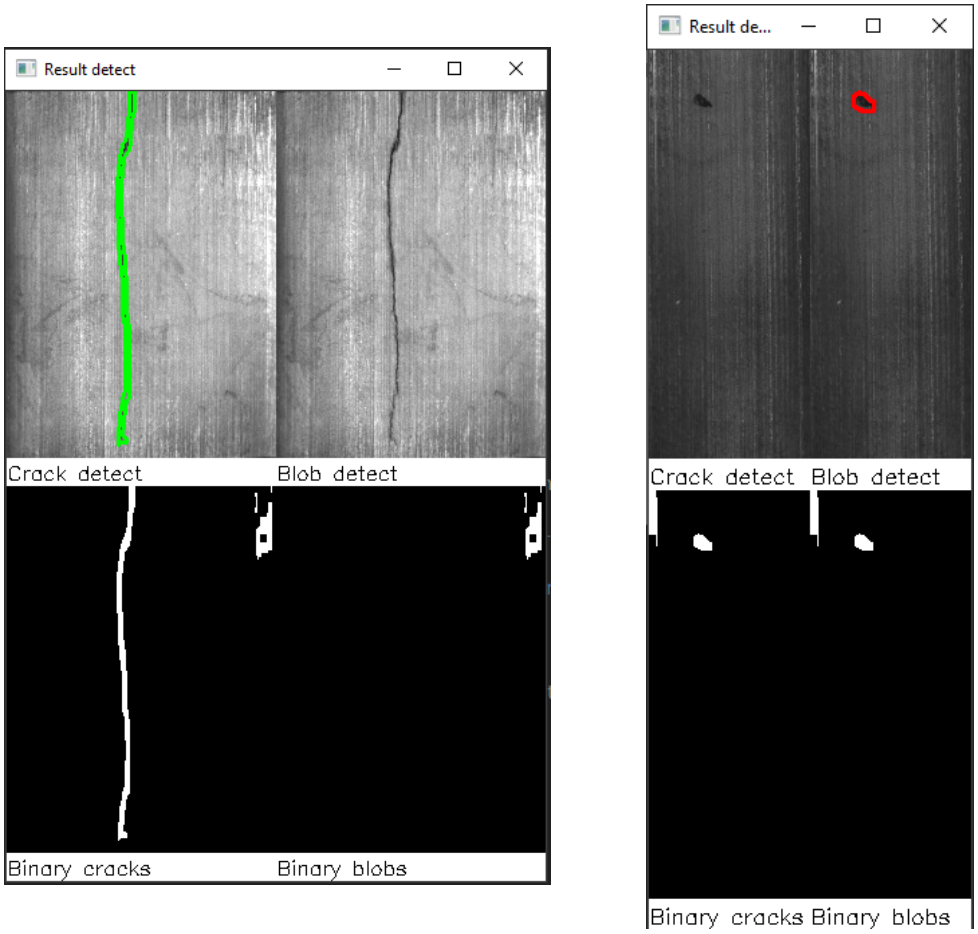

Figura 4.2: Crack (sinistra) e blob (destra) individuati dal sistema.

4.4 Analisi dei risultati ottenuti

4.4.1 Risultati

In questo capitolo vengono mostrati i risultati ottenuti, utilizzando i diversi filtri a disposizione e l'accuratezza del sistema rispetto alle tipologie di difetti.

L'accuratezza è stata calcolata come segue:

$$Accuratezza : \frac{TP + TN}{TP + TN + FP + FN} * 100$$

in cui i parametri della formula sono da intendersi come mostrato dalla matrice di confusione:

TRUE CLASS

PREDICTED CLASS		Positive	Negative
	Positive	True Positive (TP)	False Positive (FP)
	Negative	False Negative (FN)	True Negative (TN)

Figura 4.3: Matrice di confusione

La tabella sottostante mostra il numero totale di immagini classificate correttamente dal sistema differenziandole per tipologie di difetto e il numero delle classi predette.

Per numero di immagini correttamente classificate si intende il risultato, in cui il sistema identifica correttamente il difetto senza falsi positivi e negativi e che quindi coinciderà perfettamente con la relativa maschera binaria presente nel dataset, per ogni immagine.

Come mostrato dalla tabella, in generale l'accuratezza del sistema non è molto alta. Il filtro con cui si sono ottenuti i migliori risultati è il mediano. Il bilaterale, filtro molto utilizzato nell'ambito della visione artificiale, è stato quello con cui si sono ottenuti pessimi risultati con un accuratezza appena del 52,12%. Confrontando i risultati con gli altri filtri, l'accuratezza del bilaterale è fortemente influenzata dall'elevato numero di False Positive e False Negative identificati (più alti rispetto agli altri).

Tipologia di filtro:	**Mediano**	**Gaussiano**	**Bilaterale**
Numero totale di immagini:	186	186	186
Numero totale di immagini classificate correttamente:	95	86	74
Numero totale di immagini con difetti da cracks:	57	57	57
Numero totale di cracks classificate correttamente:	26	27	17
Numero totale di immagini con difetti da blobs:	89	89	89
Numero totale di blobs classificate correttamente	41	34	31
Numero totale di immagini prive di difetti:	40	40	40
Numero totale di immagini prive di difetti classificate correttamente	28	25	26
True Positive (TP)	113	102	97
True Negative (TN)	28	25	26
False Positive (FP)	58	55	69
False Negative (FN)	26	39	44
Accuratezza del sistema:	**62,67 %**	**57,47 %**	**52,12 %**

Tabella 4.1: Accuratezza del sistema rispetto ai diversi filtri utilizzati.

Il tempo medio impiegato dal sistema per la detection dei difetti è di circa 0,6 secondi eseguita su una macchina con le seguenti caratteristiche: CPU AMD Ryzen 7 e RAM 16 GB. E' importante far notare che il tempo aumenterà proporzionalmente con l'aumentare della dimensione (e quindi di pixels) dell'immagine selezionata e del tipo di filtro utilizzato.

4.4.2 Analisi degli errori

Una parte degli errori è data dal grado di irregolarità dei vari difetti presenti. Il sistema implementato, analizza i difetti attraverso delle proprietà geometriche. Ad esempio se consideriamo un ellisse (o un cerchio) irregolare molto probabilmente non soddisferà la proprietà di eccentricità (o di circolarità) e il difetto non verrà identificato anche se nella maschera binaria questo può essere più o meno presente.

Gran parte degli errori sono però attribuibili alla quantità di luce catturata dal sensore. Con una piastrella in cui è presente poca luce, i difetti non saranno ben visibili e la fase di pre-processing non sarà in grado di metterli in risalto. Viceversa in una piastrella con molta luce, il sistema identificherà il difetto minimizzando gli errori. L'immagine successiva mostra quanto spiegato: a sinistra è presente una piastrella con un blob e poca luce, a destra è presente la stessa piastrella ma con una quantità di luce maggiore.

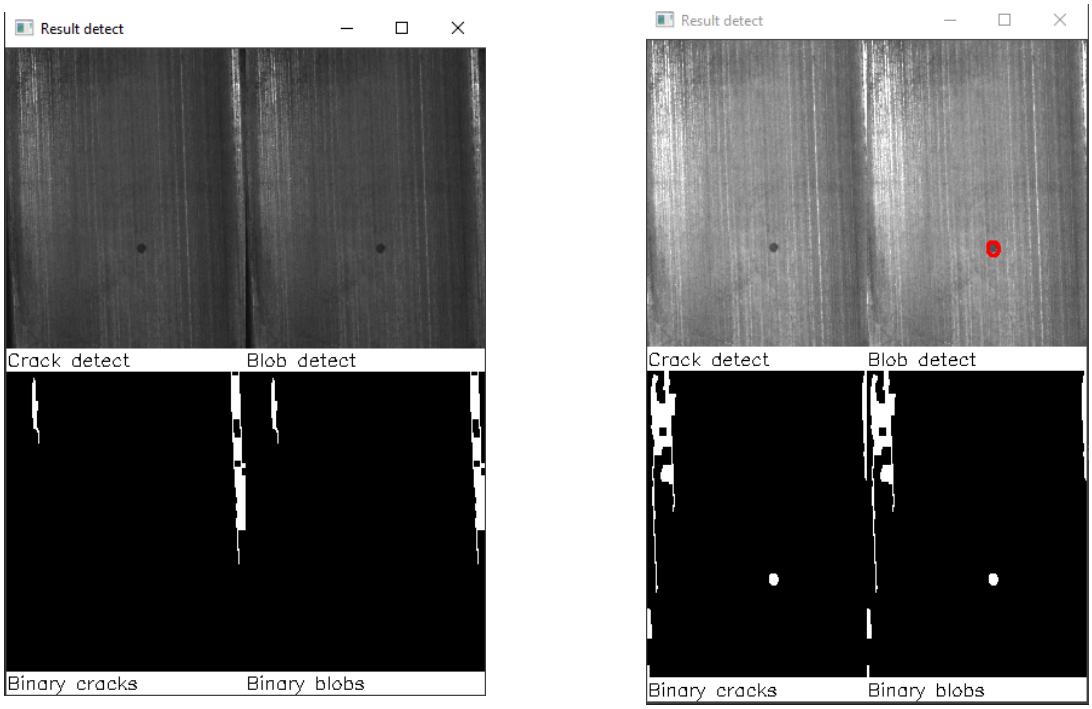

Figura 4.4: Confronto dei risultati ottenuti sulla stessa piastrella ma con differente quantità di luce presente.

Capitolo 5

Approccio 2: Rete Neurale

Questo capitolo descrive le fasi impiegate per il secondo approccio per implementare una rete neurale. A differenza del precedente approccio, in cui si eseguono operazioni sulle immagini al fine di ottenere features interessanti, in questo caso ho addestrato un modello utilizzando l'intelligenza artificiale.

5.1 U-Net

La rete neurale utilizzata per questo contesto è chiamata U-Net. Il suo nome deriva dal fatto che, guardando la sua architettura sembra avere una forma ad U. E' una *fully convolutional network* pensata per essere utilizzata in campo medico attraverso la segmentazione di immagini: infatti per essere applicata e allenata, è necessario avere come dati di *training* oltre alle immagini, anche le relative *maschere* binarie.

La sua architettura è costituita da:

- Un **encoder** (down-sample) che riduce l'immagine in ingresso in una feature map, attraverso pooling layers, estraendone gli elementi chiave;
- Un **decoder** (up-sample) che amplifica la feature map in una immagine, usando i livelli di deconvoluzione, impiegando cioè i pooling layers appresi per permettere la localizzazione degli elementi.

A questi due elementi devono essere aggiunte le *skip connection* (frecce grigie nell'immagine) che creano un ponte tra l'encoder, prima dell'iniziale filtro di pooling e il decoder dopo l'ultima operazione di deconvoluzione, permettondole di creare una combinazione di informazioni locali e contestuali.

E' stato deciso di utilizzare questa rete poiché non necessità di grandi quantità di dati per essere addestrata (visto le dimensioni ridotte del dataset preso in considerazione) ed anche perché risulta essere particolarmente efficiente.

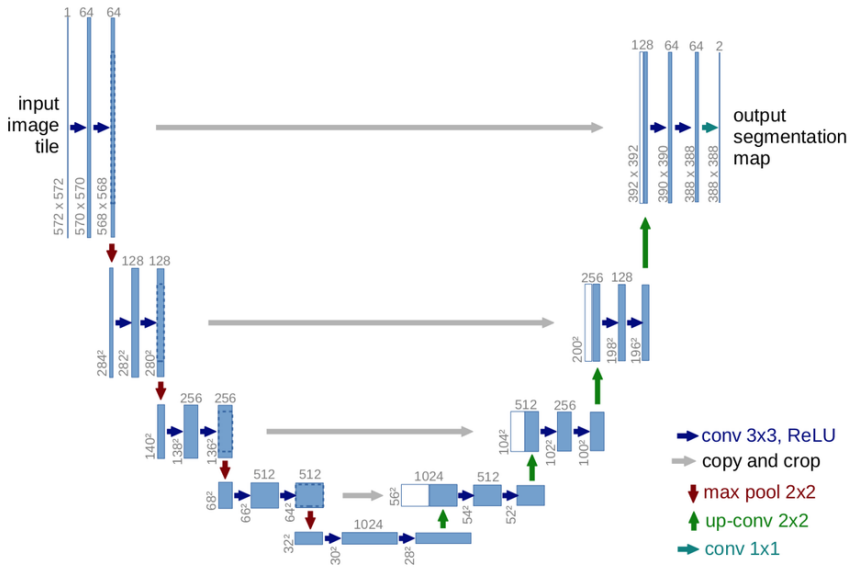

Figura 5.1: Architettura U-Net.

La rete da me implementata, come si vede in figura, presenta 4 blocchi di down-sampling, un livello bottleneck (o collo di bottiglia) e 4 livelli di u-sampling. Ogni blocco è composto dai seguenti livelli:

```
-Down: 2-1
    └─Conv2d: 3-1
    └─ReLU: 3-2
    └─BatchNorm2d: 3-3
    └─Conv2d: 3-4
    └─ReLU: 3-5
    └─BatchNorm2d: 3-6
    └─MaxPool2d: 3-7
```

Figura 5.2:
Blocco di down-sample.

```
-Bottleneck: 2-5
    └─Conv2d: 3-29
    └─ReLU: 3-30
    └─BatchNorm2d: 3-31
    └─Conv2d: 3-32
    └─ReLU: 3-33
    └─BatchNorm2d: 3-34
```

Figura 5.3:
Blocco di bottleneck.

```
-Up: 2-9
    └─Upsample: 3-59
    └─Concat: 3-60
    └─Conv2d: 3-61
    └─ReLU: 3-62
    └─BatchNorm2d: 3-63
    └─Conv2d: 3-64
    └─ReLU: 3-65
    └─BatchNorm2d: 3-66
```

Figura 5.4:
Blocco di up-sample.

5.2 Addestramento e risultati ottenuti

Questa sezione mostra la tabella con tutti gli addestramenti effettuati, gli iper-parametri utilizzati e i risultati ottenuti. In fase di implementazione sono stati effettuati più addestramenti: la tabella riporta solo quelli più interessanti per

iperparametri utilizzati e tipo di ottimizzatore. Per evitare l'*over-fitting* ho utilizzato l'*early stopping* con *patience* pari a 5. Ho anche utilizzato uno scheduler per il learning rate: *ExponentialLr* di Pytorch. Le metriche considerate sono l'accuratezza e l'Intersection Over Union (IoU o *Jaccard index*).

Epoche	**Batch size**	**Ottimizzatore**	**LR**	**Loss function**	**Accuratezza**	**IoU**
100/100	4	SGD	0.001	Binary Cross Entropy	Valid: 99.993, Test: 99.931	Valid: 0.839, Test: 0.827
10/100	4	SGD	0.025	Binary Cross Entropy	Valid: 99.993, Test: 99.931	Valid: 0.839, Test: 0.827
19/100	4	Adam	0.001	Binary Cross Entropy	Valid: 99.993, Test: 99.931	Valid: 0.839, Test: 0.827

Considerazioni Il valore ottenuto dell'accuratezza potrebbe essere fuorviante: poiché nel dataset considerato, i difetti sono piccoli rispetto alla dimensione dell'immagine, le classi sono sbilanciate portando quindi ad una male interpretazione del dato ottenuto. Ad esempio, supponiamo di considerare una piastrella che presenta un piccolo difetto di blob per cui il 95% dell'area rappresenta il ground-truth (pixel neri) e solo il 5% rappresenta il difetto (pixel bianchi). Se la rete restituisce una maschera completamente nera, allora avrà una accuratezza del 95% poichè ha classificato correttamente tutti i pixel neri. In realtà il modello non ha predetto correttamente il risultato. Questo dimostra come un'elevata precisione dei pixel non implica sempre un'elevata capacità di segmentazione. Per questo motivo, si è deciso di considerare anche l'IoU che viene espresso come l'area di sovrapposizione (o intersezione) tra la segmentazione prevista e il ground-truth, divisa per l'area di unione tra la segmentazione prevista e il ground-truth risultando essere una metrica più affidabile rispetto al contesto.

Ho anche notato che durante gli addestramenti effettuati, i valori delle metriche sul validation set non cambiavano mai. Inoltre, dalla tabella, si può notare che i risultati ottenuti nelle metriche, sono gli stessi indipendentemente dal tipo di addestramento considerato. La figura 5.5 mostra i risultati ottenuti utilizzando SGD.

Facendo una ricerca sul web, ho trovato che alcune possibili cause sono:

- Il tipo di ottimizzatore non è adeguato per il dataset scelto ma questo non è possibile poiché ho provato ad eseguire degli addestramenti sia con SGD che con Adam;
- Utilizzare una funzione di attivazione non lineare al livello di output. La mia rete utilizza una sigmoide che è non lineare;

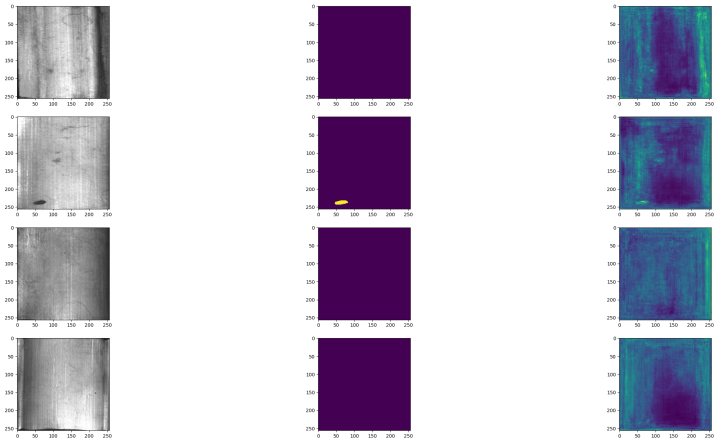

Figura 5.5: Risultati ottenuti con la mia architettura. Da sinistra si può vedere: immagine del test set, relativa maschera e immagine predetta dalla rete.

- La dimensione dei batch;
- Un learning rate iniziale elevato che non rientra nel mio caso;
- Dataset troppo piccolo o dati non adeguati.

Per individuare ancora meglio le cause, ho deciso di effettuare un *fine-tuning* ovvero considerare una U-Net preaddestrata e allernarla per il contesto di studio.

La rete presa in considerazione è consultabile al seguente *link*. L'addestramento è stato eseguito con i seguenti iperparametri:

Epoche	**Batch size**	**Ottimizzatore**	**LR**	**Loss function**	**Accuratezza**	**IoU**
30/30	4	SGD	0.001	Binary Cross Entropy	Valid: 99,949, Test: 99,939	Valid: 0,835, Test: 0,845

Come si può notare dalla tabella, i valori dell'accuratezza e IoU non sono uguali a quelli ottenuti dagli addestramenti sulla mia rete. L'immagine successiva mostra i risultati ottenuti, evidenziando una maggiore accuratezza.

Da questo piccolo test si può dedurre che i cattivi risultati ottenuti sono da attribuire maggiormente all'implementazione del modello e al dataset troppo piccolo ma non alla qualità dei dati di allenamento.

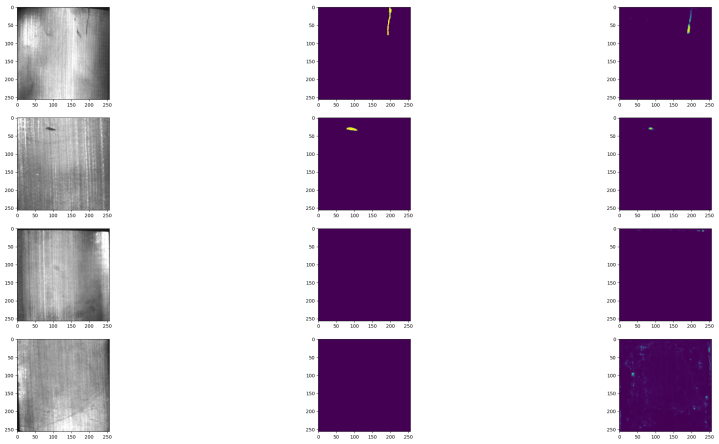

Figura 5.6: Risultati ottenuti tramite il *fine-tuning*. Da sinistra si può vedere: immagine del test set, relativa maschera e immagine predetta dalla rete.

Capitolo 6

Codice

In questo capitolo viene brevemente illustrata la struttura del progetto.

Il codice è stato scritto in Python. Le principali librerie utilizzate sono: *OpenCV* per l'applicazione delle tecniche di visione artificiale e *PyTorch* per costruzione e addestramento della rete neurale.

Il progetto è organizzato nelle seguenti directory:

- **Features**: contiene il codice utilizzato per implementare il primo approccio. Al suo interno sono contenute le directory:
  - *GUI*: contiene il codice per implementare la GUI;
  - *Defect*: contiene il codice per rilevare i difetti di crack e blob;
  - *Preprocessing*: contiene il codice per eseguire le operazioni di preprocessing precedentemente descritte.
- **UNet**: contiene il codice utilizzato per implementare il secondo approccio. Al suo interno sono contenute le directory:
  - *ArchitectureNet*: contiene il codice per *costruire* il modello della UNet con relativi blocchi: down, up e bottleneck;
  - *DatasetTiles*: contiene le immagini del dataset e il file *dataset.py* per implementare la classe relativa al dataset;
  - Ulteriori file per l'addestramento, la visualizzazione dei risultati e la gestione delle metriche.
- **doc**: contiene la documentazione del progetto *.pdf* e un notebook jupyter per eseguire il secondo approccio;
- File **main-gui.py** con cui far partire il programma per il primo approccio.
- File **main-net.py** con cui far partire il programma per il secondo approccio (addestramento e visualizzazione dei risultati).

Capitolo 7

Conclusioni

Sulla base dell'analisi delle immagini effettuate, possiamo concludere che l'etichettatura automatica del set di dati dei difetti sulle piastrelle magnetiche con alto grado di varianza nelle variazioni di luminanza, rumore e difetti può essere eseguita dalle tecniche di elaborazione delle immagini utilizzando algoritmi come non-local means denoising e di edge detection o l'utilizzo di rete neurali.

Per quanto riguarda il primo approccio, i risultati ottenuti non sono quelli attesi, ci si aspettava un accuratezza intorno al 75-80%. Il fattore che ha portato ad un maggior errore è dato dall'elevata variazione di luminanza presente nella piastrelle del dataset, il quale non si è riuscito a gestire nel migliore dei modi per ottenere l'accuratezza prevista. Il tempo medio impiegato dal sistema per eseguire il rilevamento non è utilizzabile per applicazioni in tempo reale che invece richiedono un tempo medio di circa 0.2, 0.3 secondi.

Anche per il secondo approccio i risultati ottenuti non sono quelli previsti. Si è comunque approfondito sulla causa che portava la rete a predirre in modo errato. Attraverso il *fine-tuning* si è potuto osservare un netto miglioramento dei risultati ottenuti. Questo porta a concludere che il dataset è troppo piccolo per essere utilizzato e che la mia rete costruita ha dei limiti e potrebbe essere migliorata in futuro. Nonostante le numerose risorse in rete che disponevano di modelli già pronti per essere addestrati (anche da zero), ho voluto costruire la rete da solo per poter capire al meglio la sua architettura e i *meccanismi* dietro implementati come le skip connection, sapendo di incorrere in molti più problemi e tempo necessario per realizzare l'elaborato. Inoltre la U-Net non è una rete adatta per essere applicata in tempo reale.

Se dovessi fare un confronto tra i due approcci, il primo identifica con più successo i difetti rispetto alla rete neurale. Però sono convinto che avendo avuto a disposizione più immagini, la rete neurale avrebbe avuto un'accuratezza maggiore rispetto al primo approccio.

Bibliografia

[1] Yibin Huang1, Congying Qiu, Yue Guo, Xiaonan Wang, and Kui Yuan. *Saliency of magnetic tile surface defects*, 2018. *https://www.researchgate.net/publication/325882869_Surface_Defect_Saliency_of_Magnetic_Tile*

[2] Janaki Sankirthana Saraswatula and Rajasekhar Punna. *Defect Detection and Analysis using Image Processing*, 2021. *https://www.researchgate.net/publication/349004481_Defect_Detection_and_Analysis_using_Image_Processing*