Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica - 8009

Applicazione di tecniche di Deep Learning per rilevamento non invasivo di crepe su edifici in muratura: un caso di studio reale

Relatore:
Prof. Federico Montori

Correlatori:
Dott. Alfonso Esposito
Dott. Mattia Forlesi

Presentata da:
Andrea Fiorellino

Sessione II
Anno Accademico 2024/25

Sommario

La crack detection è un campo di ricerca molto rilevante nell'ambito dell'ingegneria civile; consiste nell'identificare la presenza, la forma e la distribuzione di crepe su materiali come cemento o muratura. All'inizio, si praticava mediante ispezioni visive o sensori IoT, che tuttavia comportano costi elevati e difficoltà logistiche in contesti complessi (torri, ponti). La ricerca proposta affronta questo tema applicando tecniche di Deep Learning per la rilevazione su superfici in muratura, con lo scopo di trovare un approccio efficace e più economico. Nella fase sperimentale, articolati in due fasi, sono stati impiegati due dataset: il primo, costruito ex-novo raccogliendo e annotando manualmente pixel-wise immagini reali di strutture in muratura lesionate in alcune zone dell'Emilia-Romagna; il secondo invece è una parte del dataset di Dais et al. ([1]). Tutte le immagini sono state preprocessate e arricchite tramite data augmentation.

Nella prima fase sono stati addestrati i modelli sul primo dataset, confrontando più funzioni di perdita fra le più usate per fare segmentazione binaria. I risultati, espressi sotto forma di metriche (come Mean Intersection over Union (MIoU) e F1-score (F1)), mostrano che le funzioni basate su Tversky performano meglio poiché capaci di gestire lo sbilanciamento fra classi. Infatti questa ha raggiunto 0.854 e 0.874 come MIoU e F1.

Nella seconda fase è stato eseguito un addestramento cross-dataset con fine-tuning (FT). I risultati, non solo hanno mostrato un incremento di circa +0.07 in MIoU e +0.08 in F1 dopo il FT, ma hanno anche superato le performance dei modelli addestrati solo sul nostro dataset. Anche le meriche confermano la validità di queste techiche, con valori del modello proposto in questo studio (MurCrackNet) di MDice e F1 pari a 0.874.

Infine, si propongono sviluppi futuri orientati all'ampliamento del dataset — includendo immagini web e di materiali diversi, come il cemento — per costruire modelli più robusti.

Introduzione

Il monitoraggio dello stato di salute degli edifici è un aspetto fondamentale per garantire sicurezza e durabilità delle strutture. In letteratura, uno dei compiti fondamentali del monitoraggio è proprio quello di valutare diversi indicatori di salute strutturale per rilevare fenomeni degenerativi prima che questi si palesino e quindi attuare interventi preventivi [2]. Tra questi indicatori troviamo deformazioni o spostamenti rispetto alla configurazione iniziale, vibrazioni naturali o indotte, la variazione dei modi modali della struttura, la presenza di corrosione, alterazioni del materiale costruttivo, microfessurazioni difficili da percepire visivamente [3].

Quest'ultimo fenomeno riveste un ruolo particolarmente significativo, infatti nei materiali fragili o quasi-fragili, come muratura e calcestruzzo, la formazione delle crepe visibili è spesso preceduta da una fase di microfessurazione diffusa all'interno della cosiddetta Fracture Process Zone (FPZ), una regione caratterizzata da un comportamento non lineare e da fenomeni di progressivo indebolimento del materiale [4].

Proprio questo motivo, il patrimonio strutturale ed edilizio esistente è spesso caratterizzato dalla presenza di crepe, le quali giocano un ruolo fondamentale nella determinazione dello stato di salute delle strutture. La loro presenza ed evoluzione nel tempo forniscono informazioni essenziali per valutare la stabilità e il grado di degrado. Queste crepe possono manifestarsi su diverse tipologie di superfici: dalle pavimentazioni stradali [5], a strutture in calcestruzzo e ponti [6], o su strutture in muratura come abitazioni private (Figura 1a) o edifici storici (Figura 1b).

(a) Crepe su edificio a Bologna

(b) Crepe fra Porta San Donato e Porta Mascarella (Bologna)

Crack Detection

La letteratura, dunque, ha proposto svariate metodologie di crack detection per monitorare lo stato di salute degli edifici, differenziandole principalmente in base a livello di invasività, precisione, scalabilità e costi operativi.

I metodi tradizionali si basano su strumenti fisici di misura, tra cui estensimetri, comparatori meccanici, sensori di spostamento lineare (LVDT, figura 2, immmagine b) e sensori a fibra ottica, in grado di misurare aperture e deformazioni con un'elevata precisione puntuale [7]. Queste soluzioni risultano affidabili e particolarmente utili per monitoraggi localizzati, ma presentano alcune criticità:

- i sensori devono essere installati fisicamente; perciò, non è sempre possibile adoperarli in contesti dove si deve preservare l'integrità dei materiali della struttura oggetto di analisi; basti pensare a zone colpite da catastrofi naturali (terremoti [8]) o su edifici storici dove è sconsigliato per motivi di tutela e conservazione [1];

- ispezioni manuali o il posizionamento dei sensori richiedono personale specializzato e l'utilizzo di impalcature, piattaforme o sistemi di sollevamento per accedere alle zone da analizzare, comportando tempi e costi elevati [9];
- la standardizzazione delle misurazioni è inoltre difficoltosa, essendo fortemente influenzata dalla soggettività dell'operatore [9]

(**a**) (**b**)

Figura 2: sensori di deformazione (a) e LVDT (b) [7]

Per ridurre l'invasività e ampliare la capacità di osservazione, si sono progressivamente diffuse tecniche ottiche e fotogrammetriche, basate sull'acquisizione di immagini mediante fotocamere fisse, sistemi multi-camera o droni. Metodi come la Digital Image Correlation (DIC) permettono di misurare campi di deformazione e variazioni di ampiezza delle crepe in modo non invasivo, con elevata risoluzione spaziale e temporale. Questi approcci risultano particolarmente efficaci nel monitoraggio di superfici ampie (ponti, torri o facciate storiche) dove le ispezioni manuali sarebbero pericolose o logisticamente onerose. Tuttavia, la loro efficacia può essere condizionata da fattori ambientali come l'illuminazione, l'angolazione della ripresa o la presenza di occlusioni [10, 11].

Deep learning per crack detection

Negli ultimi anni, l'evoluzione della computer vision e del Deep Learning (DL) ha portato alla nascita di metodologie data-driven, in grado di automatizzare completamente l'identificazione e la segmentazione delle crepe a partire da immagini ottenute da fotocamere, droni o altri sistemi di monitoraggio remoto. Successivamente, per l'analisi delle foto vengono impiegate tecnologie come Convolutional Neural Network (CNN) e modelli di segmentazione semantica per distinguere con grande accuratezza le crepe dal materiale di fondo anche in condizioni non ideali o su superfici complesse come quelle in muratura [1].

Dunque, l'analisi delle immagini al momento si configura come una strategia economicamente vantaggiosa e meno invasiva; il presente elaborato si inerisce proprio in questo contesto: affronta il problema della crack detection, in particolare su edifici in muratura, attraverso metodologie di segmentazione che sfruttano CNN.

Datasets

Per sfruttare il DL è necessario di un set di dati (in questo caso immagini di crepe) sufficientemente ampi e annotati per l'addestramento delle reti. Generalmente esistono due tipologie di dataset: dataset **ortogonali** realizzati con immagini acquisite con fotocamere posizionate perpendicolarmente alla superficie lesionata oppure generate combinando più scatti di una stessa crepa, successivamente elaborati tramite software dedicati per ottenere una orto-immagine uniforme e corretta geometricamente; oppure dataset **NON ortogonali** costituiti da immagini catturate da angolazioni oblique o aeree.

Relativamente alla raccolta dei dati, le immagini vengono generalmente acquisite tramite fotocamere digitali, smartphone, droni o reperite online (come in Figura 3). Ad esempio:

- Dais et al. (2021) [1] hanno realizzato un dataset attraverso fotocamere digitali e smartphone raccogliendo immagini nella regione di Groningen (Paesi Bassi), con diversi livelli di illuminazione e texture eterogenee;
- nel dataset MCrack1300 (2024) [12] vengono raccolte immagini direttamente in edifici storici con altre provenienti da dataset preesistenti e da fonti web, al fine di garantire una maggiore diversità visiva e strutturale;
- Zou et al. (2018) [13] hanno realizzato quattro dataset dedicati all'addestramento e alla valutazione del modello DeepCrack, composti da immagini di pavimentazioni in cemento, superfici stradali acquisite mediante fotocamere digitali.

Figura 3: Strumenti di acquisizione usati per i dataset della ricerca [14]

Tuttavia, per quanto riguarda la crack detection su strutture in muratura, diversi studi lamentano la mancanza di dataset sufficientemente ampi: Mazleenda et al. (2024) [15] sottolineano come i dataset per murature siano generalmente di dimensioni ridotte, spesso inferiori a 1000 immagini; lo studio di Dais et al. [1] osserva che la maggior parte delle ricerche si concentra su cemento e asfalto, mettendo da parte la muratura. Anche recenti iniziative, come il dataset MCrack1300 (2024) [12], nascono col proposito di colmare questa lacuna.

Per colmare, anche se in parte, tali lacune questa ricerca introduce il dataset NonOrtho, una raccolta di immagini di murature reali annotate pixel-wise, ottenute esclusivamente tramite uno smartphone di fascia media senza l'uso di strumentazioni costose o professionali. L'obiettivo è fornire una base dati accessibile e rappresentativa di scenari reali - come quelli di molti comuni dell'Emilia-Romagna, caratterizzati da una grande varietà di texture murarie - utile per l'addestramento e la validazione di modelli di segmentazione basati su DL. Nonostante non conti ancora 1000 immagini, il dataset si colloca come primo passo verso la costruzione di uno più ampio e diversificato, seguendo l'esempio di iniziative quali MCrack1300.

Struttura generale

- Capitolo 1: introduzione del contesto generale e dello stato dell'arte sul monitoraggio e sulla crack detection. Vengono presentati alcuni fra i metodi principali, spaziando dai sistemi basati su sensori IoT e rilievi ottici, fino ai più moderni approcci basati su DL, evidenziando vantaggi, limiti e applicazioni. Il capitolo si chiude con i contributi che si vogliono apportare allo stato dell'arte attraverso la realizzazione del dataset NonOrtho e la sperimentazione proposta;

- Capitolo 2: descrizione della fase progettuale, includendo obiettivi, architetture adottate, raccolta e annotazione dei dati, operazioni di preprocessing e di data augmentation impiegate per aumentare la varietà e la robustezza dei dati;

- Capitolo 3: illustrazione dei dettagli implementativi, introducendo gli strumenti hardware e software utilizzati. Viene inoltre descritta la pipeline di addestramento dei modelli, con diversi esempi di codice;

- Capitolo 4: panoramica dei risultati sperimentali ottenuti presentandoli mediante tabelle e grafici, accompagnata da un'analisi critica delle metriche ottenute;

- Capitolo 5: si traggono le conclusioni mostrando i risultati più significativi ottenuti e si delineano le possibili prospettive future. Tra queste si annovera l'ampliamento del dataset, al fine di migliorare la capacità di generalizzazione dei modelli.

Indice

Capitolo 1

Stato dell'arte

1.1 Structural Health Monitoring (SHM)

Lo Structural Health Monitoring (SHM) è un approccio sistematico sviluppato per rilevare e valutare nel tempo lo stato di infrastrutture ed edifici, prevenendo cedimenti o degradazioni; ciò è possibile grazie a strumenti di misura, sensori e algoritmi di analisi che monitorano fattori quali deformazioni, vibrazioni o spostamenti [2, 3]. Storicamente, le metodologie SHM si sono sviluppate attraverso tre grandi fasi:

I era: le prime pratiche, risalenti anche a secoli fa, si basavano su osservazioni tattili e sonore per rilevare anomalie, come nel caso del tosaruota ferroviario[1]. Successivamente, con l'avvento delle tecniche di Non-Destructive Evaluation (NDE), furono introdotti strumenti fisici per misurazioni locali (ad esempio emissioni acustiche, strain gauge, accelerometri), ma tali metodi richiedevano una forte interazione umana [2];

II era: dagli anni 2000, progressivamente, sono stati adottati approcci basati su pattern statistici e machine learning (**data-driven**), capaci di estrarre informazioni significative da segnali sperimentali (senza la necessità di un modello fisico predeterminato) per poi interpretarli alla ricerca di pattern anomali indicativi di danni o deterioramenti. Tale metodologia ha permesso di analizzare dati provenienti da sensori eterogenei (accelerometri, strain gauge, sensori ottici, ecc.) in contesti reali complessi — come ponti, infrastrutture o velivoli — dove le condizioni operative e ambientali variano nel tempo [16, 17, 2];

[1] Strumento meccanico e acustico che permette all'operatore di percepire variazioni nelle vibrazioni o nei suoni prodotti dalla rotazione della ruota al fine di rilevare difetti o crepe nelle ruote dei treni

III era: recentemente è emerso un nuovo paradigma denominato Population-Based Structural Health Monitoring (PBSHM), che rappresenta l'estensione naturale dei metodi data-driven, attraverso tecniche di transfer learning e rappresentazioni astratte basate su grafi. L'idea di base prevede lo sfruttamento di una struttura ricca di dati (fonte) per supportare il monitoraggio di una struttura povera di dati (bersaglio) a condizione che queste abbiano una certa similarità. Per verificare questa corrispondenza, le strutture vengono rappresentate in maniera astratta attraverso grafi che ne descrivono la topologia, la funzionalità e le proprietà fisiche degli elementi costitutivi. In questo modo si supera la necessità di avere dataset di addestramento particolarmente ricchi, rendendo il monitoraggio strutturale più generalizzabile e scalabile [2].

Di seguito, nella Tabella 1.1, è presente un sommario degli articoli usati per la realizzazione del presente paragrafo:

Tabella 1.1: Evoluzione storica dello Structural Health Monitoring (SHM)

Era	**Paradigma**
I – Model-based	Monitoraggio basato su modelli fisici, NDE, osservazioni tattili e acustiche; metodi locali e qualitativi [2, 3]
II – Data-driven	Pattern recognition e machine learning su dati da sensori eterogenei; non richiede modello fisico, adatto a contesti complessi [16, 17, 2]
III – Population-Based SHM (PBSHM)	Transfer learning tra strutture simili tramite rappresentazioni astratte e grafi; riduce la dipendenza da dati reali [2]

L'evoluzione delle tecniche di SHM ha aperto la strada a metodologie sempre più mirate all'analisi localizzata dei danni strutturali, tra cui la crack detection.

1.2 Tecniche di deep learning per crack segmentation

Negli ultimi anni la rilevazione automatica delle crepe (*crack detection*), grazie all'evoluzione delle tecniche di DL, ha rappresentato un tema di crescente interesse nella comunità scientifica. Alcuni modelli basati su CNN si sono rivelati essere un valido strumento nella segmentazione di crepe su superfici eterogenee, come cemento, asfalto e muratura, dimostrandosi quindi adatti al monitoraggio dello stato di salute delle infrastrutture.

Le prime architetture di riferimento sono state quelle basate su questa tecnologia, come **DeepCrack**, che adotta una struttura encoder-decoder ispirata a SegNet e combina feature multi-scala per migliorare la localizzazione delle crepe a basso contrasto. Questo approccio ha ottenuto risultati solidi su dataset stradali, raggiungendo F1 superiore a 0.87, ponendo le basi per successive varianti multi-scala [13]. Una di queste che ha apportato modifiche significativa è rappresentata da **CrackFormer**, che integra moduli di *self-attention* e *scaling-attention* per aumentare la sensibilità verso crepe sottili e discontinue, migliorando le metriche Optimal Dataset Scale (ODS) rispetto ai modelli CNN tradizionali [18].

Parallelamente, architetture **U-Net-like** sono state ampiamente adottate per la segmentazione di crepe per via della loro capacità di preservare dettagli spaziali attraverso skip connection. In questo contesto, nella ricerca di **Zhang et al. (2022)**, sono state studiate varianti di U-Net applicate a superfici in cemento, usando tecniche di *data augmentation* e strategie di bilanciamento delle classi per gestire la sproporzione tra pixel "crack" e "background" [19]. Invece, per quanto riguarda strutture in muratura, **Dais et al. (2021)** [1] usano una struttura di questo genere per analizzare un loro dataset, annotato pixel-wise;
Un'altra tecnica affrontata in letteratura è quella di integrare meccanismi di attenzione e Transformer. **Li et al. (2023)** combinano CNN con moduli di attenzione spaziale e canale per affinare la localizzazione delle crepe in scenari complessi, migliorando la precisione e il recall [20]. **Chen et al. (2024)** propongono invece architetture ibride CNN-Transformer che bilanciano prestazioni e complessità computazionale, sfruttando moduli di attenzione globale per catturare correlazioni tra parti dell'immagine che sono lontane tra loro [21]. Alternativamente, **Tao et al. (2023)**, integra *Dilated Residual Blocks* ai moduli Transformer per migliorare la continuità dei bordi e la definizione delle crepe sottili [22].

Altri contributi recenti hanno esplorato approcci multi-scala avanzati come le Feature Pyramid Networks (FPN), che aggregano caratteristiche a diverse risoluzioni per gestire la variabilità dimensionale e la complessità del contesto visivo. Queste soluzioni hanno mostrato miglioramenti in robustezza e capacità di generalizzazione, soprattutto su dataset stradali e cementizi [23].

Quanto detto in questo paragrafo è riassunto nella Tabella 1.2.

Tabella 1.2: Confronto tra lavori di crack detection

Articolo	**Metodo principale**	**Tipo di dataset**	**Anno**
Zou et al. (DeepCrack) [13]	CNN multi-scala (SegNet-like)	Stradale (cemento/asfalto)	2018
Dais et al. [1]	CNN (U-Net-like)	Muratura	2021
Liu et al. (CrackFormer) [18]	CNN + Self-Attention	Cemento + Stradale	2021
Zhang et al. [19]	U-Net	Cemento	2022
Ong et al. [23]	FPN multi-scala + Self-Guided Attention	Cemento + Stradale	2023
Tao et al. (CTCrackSeg) [22]	CNN + Transformer (Boundary Awareness)	Cemento + Stradale	2023
Li et al. [20]	CNN + Attention (spaziale e canale)	Muratura + Cemento	2023
Chen et al. [21]	CNN + Transformer ibrido (attenzione globale)	Stradale + Muratura	2024

Contributo dello studio

Il presente studio, consapevole delle lacune esistenti nella letteratura, non si limita alla sola raccolta del dataset, ma mira anche a valutare l'efficacia di modelli di DL per la segmentazione di crepe su murature, un contesto poco esplorato rispetto ad altri materiali come cemento o asfalto. A tal fine, viene creato un nuovo dataset annotato pixel-wise, denominato NonOrtho, e si esplora l'impiego di modelli (realizzati con le indicazioni fornite dallo stato dell'arte) pre-addestrati su dataset pubblici, come quello di Dais et al. [1], per fare transfer learning, valutandone il guadagno. In questo modo, la ricerca non solo fornisce un nuovo dataset di riferimento, ma propone anche una metodologia sperimentale replicabile per fare crack detection su superfici in murature.

Capitolo 2

Progettazione

Lo scopo di questo lavoro è esplorare e valutare l'efficacia degli approcci di DL per il rilevamento automatico di crepe su strutture in muratura. Per fare ciò si realizza un'analisi comparativa su diversi modelli, alcuni dei quali presi da altre ricerche, valutandone l'efficacia su due dataset: uno preso dallo studio di Dais et al.[1] e un secondo, creato ad hoc per questo studio.

Verranno altresì considerati diversi fattori essenziali, quali l'identificazione della funzione di perdita più adatta al contesto e strategie come trasfer learning e fine-tuning. Questo capitolo si propone di descrivere nel dettaglio tutte le fasi della progettazione a partire dalla struttura e preparazione dei dataset, fino all'implementazione dei modelli e delle relative configurazioni.

2.1 Dataset NonOrtho

Come già detto, in letteratura si lamenta la mancanza di dataset per muratura poiché la maggior parte della ricerca si svolge su cemento e asfalto [1]; dunque, per contribuire a colmare anche se parzialmente questa lacuna, in questo lavoro è stato sviluppato un dataset non ortogonale.

Figura 2.1: Esempi di immagini del nostro dataset 224x224

Tabella 2.1: Suddivisione del nostro dataset

Sottoinsieme	**Numero di immagini**
Train	417
Validation	139
Test	140
Totale	**696**

2.1.1 Data collection

Per la raccolta dei dati è stata necessaria una ricerca esaustiva di crepe su edifici di diversi tipi di muratura (in modo che il dataset sia abbastanza eterogeneo) in diverse zone dell'Emilia-Romagna, in particolare nelle aree di Bologna, Reggio Emilia, Bagno, Carpi e Rubiera. Strumento indispensabile per ottimizzare i tempi di raccolta è stato GoogleMaps, che ha reso possibile l'ispezione di ampie aree per poter selezionare solo quelle di interesse al fine della ricerca.

Alle ricerche online seguivano poi sopralluoghi fisici, e nel caso in cui si notavano crepe si proseguiva nel seguente modo:

- venivano posizionati attorno a ogni crepa tre marker numerati (distinti), in modo da formare un triangolo rettangolo come mostrato in figura 2.3;
- si misurava il perimetro del triangolo formato dai markers;

- si scattavano almeno 8–12 fotografie per ogni crepa, inquadrando porzioni di muro diverse, assicurandosi che in almeno in una foto compaiano tutti e 3 i marker e che in tutte le restanti ne compaia almeno 1;
- Acquisizione di una foto della crepa senza marker (Figura 2.2).

Per ogni scatto lo smartphone è stato tenuto in posizione quanto più perpendicolare possibile alla superficie lesionata.

Figura 2.2: Foto crepa senza markers

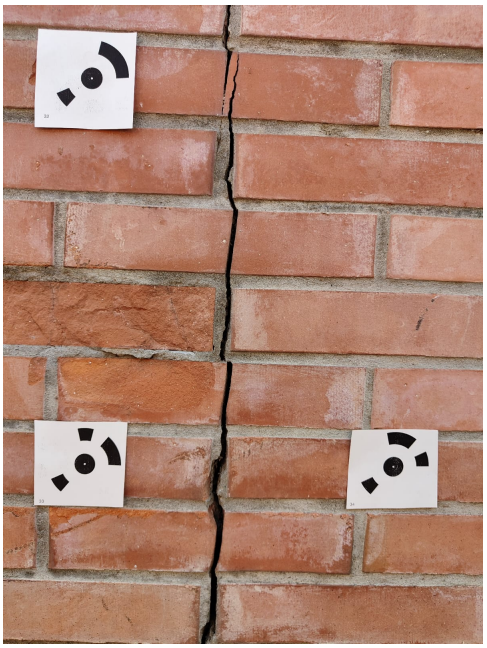

Figura 2.3: Foto con 3 markers attorno alla crepa

Il nostro dataset è un dataset **NON ortogonale**, poiché le immagini usate per la sua costruzione sono quelle scattate senza markers attorno alla crepa. Tuttavia è stato scelto di raccogliere anche le immagini con i marker poiché in futuro sarebbe utile creare un dataset **ortogonale**: attraverso software dedicati è possibile, grazie alle foto che mappano la porzione di muro dove è presente la crepa e alle misurazioni del perimetro del triangolo, creare delle orto-immagini[1].

[1]Immagine corretta geometricamente (orto-rettificata) che permette misurazioni accurate senza distorsioni prospettiche.

2.1.2 Annotazione

Per la costruzione del dataset è stato inoltre necessario produrre le maschere delle immagini raccolte; questa è stata effettuata manualmente a livello *pixel-wise* con il software grafico GIMP: per ogni immagine (Figura 2.4) è stata prodotta una maschera binaria, come in Figura 2.5 (1 = crepa, 0 = background).

Figura 2.4: Esempio immagine crepa (224x224)

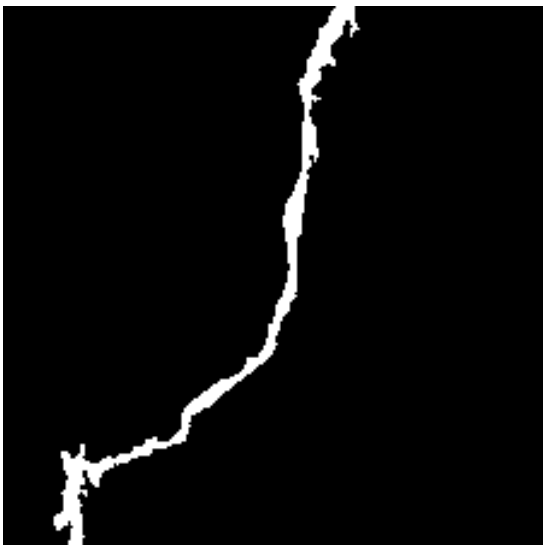

Figura 2.5: Esempio immagine maschera (224x224)

2.2 Dataset di Dais et al

Per i nostri esperimenti è stato usato parte del dataset illustrato nella ricerca [1]. È stato reperito nella repository GitHub del progetto in questione nella cartella `https://github.com/dimitrisdais/crack_detection_CNN_masonry/tree/main/dataset`.

Figura 2.6: Esempi di immagini del dataset Dais et al 224x224

Tabella 2.2: Suddivisione del dataset di Dais et al

Sottoinsieme	**Numero di immagini**
Train	168
Validation	36
Test	37
Totale	**241**

Come si evince dall'articolo [1], il dataset è stato costruito impiegando diverse tipologie di dispositivi, tra cui fotocamere DSLR e smartphone, con l'obiettivo di simulare un contesto il più fedele possibile alla realtà. Inoltre, una parte delle immagini è stata reperita da internet per ampliare ulteriormente la varietà di condizioni ambientali, illuminazioni e texture, consentendo ai modelli di generalizzare meglio. Le immagini raccolte ritraggono superfici in muratura sia con che senza crepe, includendo anche elementi di disturbo come finestre, cavi, vegetazione o insegne (esempi in Figura 2.6).

2.3 Preprocessing e Data Augmentation

I due dataset appena descritti presentano un forte sbilanciamento e un numero limitato di campioni; dunque, per NonOrtho, tutte le immagini sono state suddivise in patch 224×224; inoltre, alcuni studi mostrano che in situazioni simili, l'arricchimento della diversità dei dati aiuta i modelli a diventare più robusti e a ridurre l'overfitting[2] [24]. Per questo motivo, è stata adottata la data augmentation per aumentare la quantità di esempi, tra cui:

- Flipping orizzontale e verticale;
- Rotazioni di 0°, 90°, 180° o 270°;
- Traslazioni e piccoli jitter di scala;
- Modifiche casuali di luminosità, contrasto, saturazione e tonalità;
- Applicazione opzionale di blur o rumore.

Queste trasformazioni sono state applicate in maniera uniforme su entrambi i dataset descritti nelle sezioni 2.1 e 2.2.

La figura 2.7 illustra le fasi di preprocessing e di trattamento delle immagini. A partire da una singola immagine (ottenuta durante la ricerca sul campo per il dataset NonOrtho), la sola suddivisione in patch consente già di ottenere anche sei immagini distinte. Successivamente, a ciascuna patch (per entrambi i dataset) sono state applicate diverse trasformazioni di data augmentation. Nella stessa figura sono riportati alcuni esempi di rotazione e flipping delle patch, da cui si evince che — partendo da un'unica immagine originale — è possibile generare fino a 18 campioni differenti, incrementando in modo significativo la varietà del dataset.

[2] condizione in cui un modello apprende troppo bene i dati di addestramento, perdendo capacità di generalizzazione su dati non visti.

Figura 2.7: Esempio di Preprocessing e Data Augmentation su campione del dataset NonOrtho

2.4 Modelli

In questo capitolo vengono analizzate diverse architetture di deep learning per la segmentazione di crepe, selezionate in base alla loro rilevanza in letteratura. Tra queste figurano modelli basati su architetture encoder–decoder come la U-Net [25], scelta per la sua comprovata efficacia nel preservare dettagli spaziali durante la ricostruzione dell'immagine grazie agli skip connection. Vengono inoltre considerati modelli Fully Convolutional Network (FCN) con backbone ResNet-50 [26], utilizzati come baseline per la loro capacità di produrre mappe dense a partire da immagini di dimensione variabile e per via della loro stabilità durante l'addestramento grazie ai blocchi residuali. Sono inclusi anche approcci multi-scala più recenti come DeepCrack [13] e una sua evoluzione, DeepCrackAT [14], che introducono meccanismi di pooling indicizzato, fusioni multi-livello e attenzione per catturare sia dettagli locali sia pattern più estesi delle crepe. Infine, viene presentato un modello personalizzato denominato *MurCrackNet*, sviluppato per integrare e sperimentare tecniche avanzate di attenzione e fusione multi-scala, combinando feature profonde, convoluzioni dilatate e moduli Tok-MLP, al fine di migliorare la rappresentazione dei pattern lineari tipici delle crepe e la capacità di generalizzazione in scenari con dati variabili o complessi.

2.4.1 U-Net

Il modello U-Net [25] (Figura 2.8) rappresenta una delle architetture più diffuse per la segmentazione semantica e si caratterizza per la sua struttura simmetrica encoder–decoder. L'encoder estrae progressivamente rappresentazioni a livello più astratto tramite convoluzioni e downsampling, mentre il decoder ricostruisce la risoluzione spaziale attraverso operazioni di upsampling. Elemento distintivo di questa rete sono le *skip connection*, che mettono in collegamento diretto i livelli corrispondenti di encoder e decoder, consentendo di reintegrare i dettagli persi durante la fase di compressione.

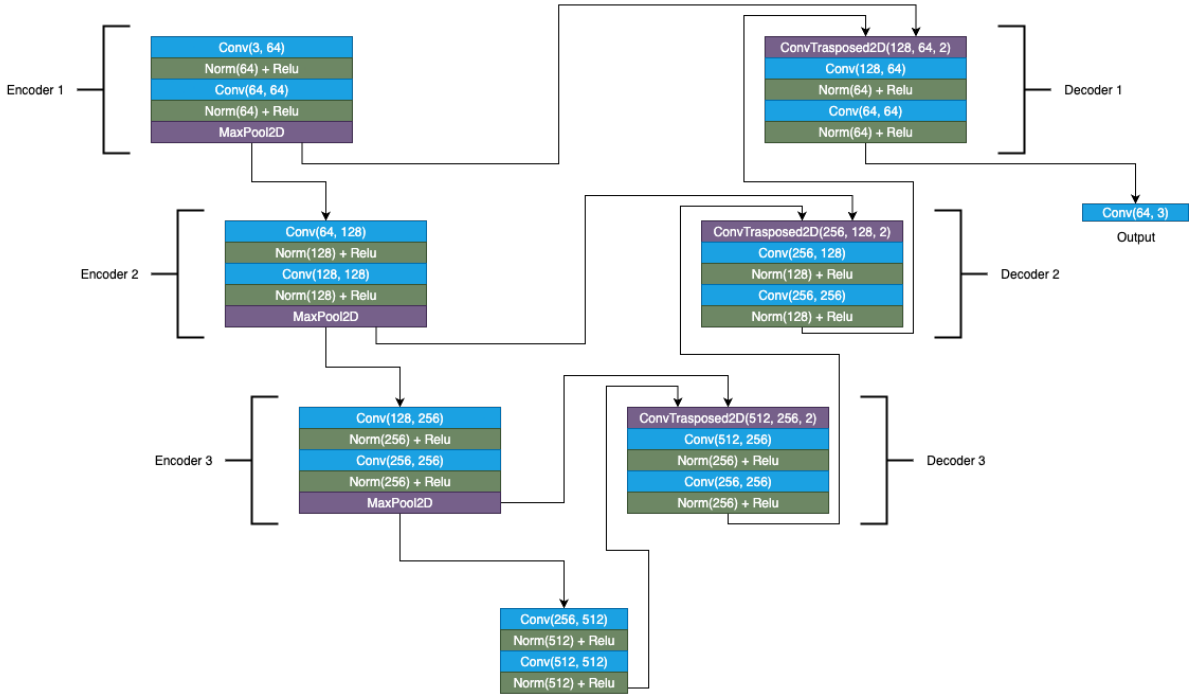

Figura 2.8: Struttura del modello Unet

2.4.2 ResNet

Per avere un termine di confronto, è stato considerato anche un modello di tipo Fully Convolutional Network con backbone ResNet-50 [26], già ampiamente utilizzato come baseline in compiti di segmentazione semantica. La presenza dei blocchi residuali permette di addestrare reti più profonde riducendo i problemi di vanishing gradient, mentre la struttura interamente convoluzionale consente di produrre mappe di segmentazione dense a partire da immagini di input di dimensione variabile.

2.4.3 DeepCrack

Il modello DeepCrack [13] si basa su un'architettura encoder–decoder ispirata a SegNet [27]; utilizza il pooling indicizzato per preservare le informazioni spaziali e un meccanismo di fusione multi-scala: l'encoder esegue sequenze di convoluzioni seguite da operazioni di pooling salvando gli indici; tali indici sono poi usati dal decoder per ricostruire le mappe originali attraverso operazioni di unpooling. A ogni livello, vi sono delle fusioni fra le feature prodotte dall'encoder e dal decoder, e mappe di probabilità a diverse scale vengono combinate.

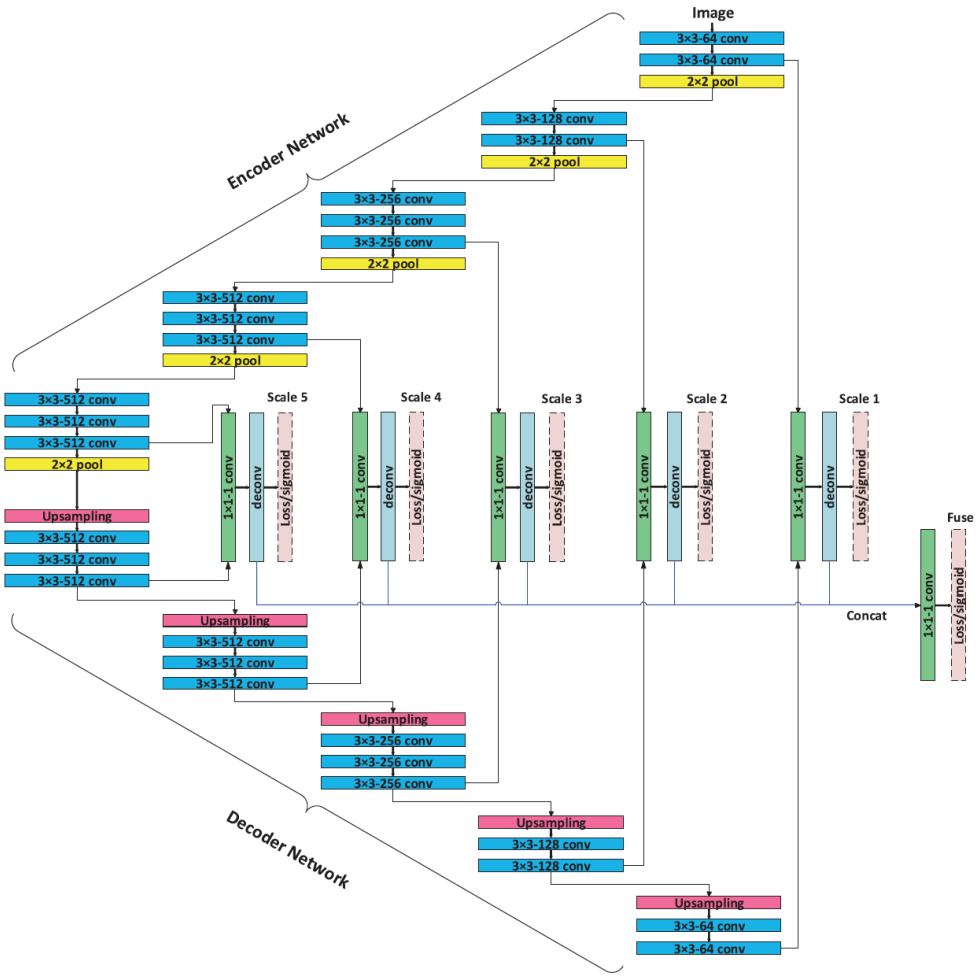

Figura 2.9: Struttura del modello DeepCrack [13]

2.4.4 MurCrackNet

L'architettura proposta in questo studio, denominata *MurCrackNet* è ispirato a recenti approcci basati su FPN come quello di Ong et al. [23]. Il modello è strutturato secondo uno schema encoder–decoder, con connessioni laterali che permettono la fusione di feature multi-scala provenienti dai vari livelli dell'encoder. Come estrattore di feature profonde è stato usato un backbone ResNet-50, come comunemente impiegato in letteratura per compiti di segmentazione semantica, grazie alla sua capacità di addestrare reti profonde limitando il fenomeno del vanishing gradient[3] [26]. Al backbone vengono affiancati blocchi Hybrid Dilated Convolution (HDC), che permette di ampliare il campo recettivo mediante convoluzioni dilatate, catturando così sia dettagli locali che quelli globali [28]. Per migliorare la capacità di individuare regioni rilevanti anche in scenari rumorosi sono stati inseriti nei livelli più profondi blocchi Convolutional Block Attention Module (CBAM), introducendo un meccanismo di attenzione sia spaziale che per canale [29]. È stato anche usato un blocco Tok-MLP per proiettare le feature estratte in uno spazio più compatto e meno ridondante, incrementando la robustezza nella rappresentazione dei pattern lineari tipici delle crepe [30]. Infine, il decoder fa upsampling progressivi e fonde le feature provenienti dai livelli intermedi dell'encoder, per recuperare i dettagli spaziali persi durante la compressione.

2.4.5 DeepCrackAT

DeepCrackAT (Figura 2.10) è un'architettura encoder-decoder progettata per la segmentazione pixel-wise delle crepe [14]. La rete integra HDC[4] nei primi strati per aumentare il campo recettivo senza perdere dettagli locali [28], e utilizza un blocco Tok-MLP[5] nel bottleneck per ridurre i parametri e migliorare la rappresentazione delle informazioni spaziali e globali [30]. I side outputs generati a diverse scale vengono combinati tramite un Attentional Skip–layer Fusion Block (ASFB)[6] con CBAM[7], permettendo di preservare e rafforzare le feature significative provenienti dai livelli più bassi [29].

[3]Fenomeno che si verifica quando i gradienti calcolati durante il backpropagation diventano molto piccoli, rendendo difficile l'addestramento dei primi strati di reti profonde

[4]HDC: serie di convoluzioni dilatate (che "saltano" dei pixel tra un elemento del filtro e l'altro) con tassi diversi per aumentare il campo recettivo e catturare dettagli locali e globali.

[5]Tok-MLP: modulo che trasforma le feature estratte dalla CNN in "token" compatti, li elabora con un multi-layer perceptron e li riporta nello spazio originale, riducendo la ridondanza e migliorando la robustezza nella rappresentazione dei pattern lineari come le crepe.

[6]ASFB: modulo che fonde le feature tra encoder e decoder con attenzione per migliorare la segmentazione pixel-wise.

[7]CBAM: modulo che applica attenzione lungo due dimensioni delle feature: spaziale (localizzazione dei pattern importanti nell'immagine) e per canale (peso differenziato per ciascun canale della feature map, enfatizzando quelli più rilevanti per il task).

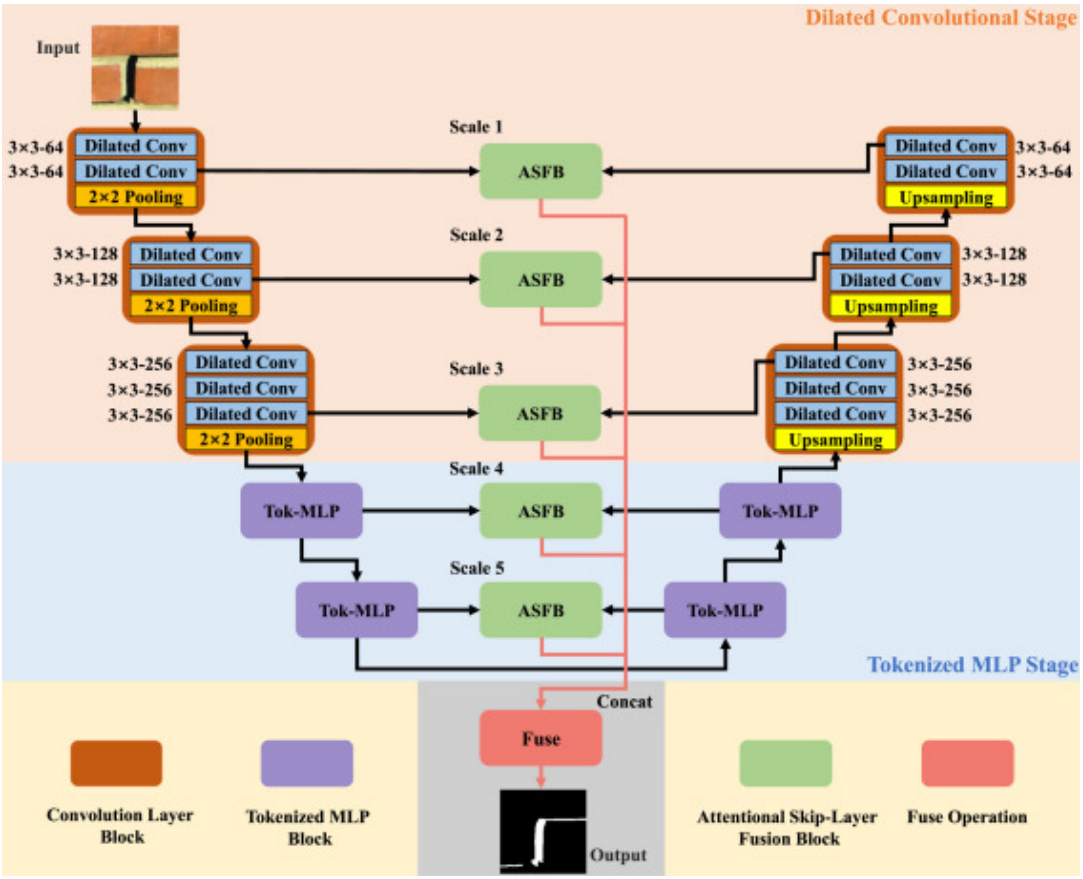

Figura 2.10: Struttura del modello DeepCrackAT [14]

2.5 Addestramento modelli sul dataset Nonortho

In questa prima fase della ricerca, l'obiettivo principale è stato addestrare i modelli sul dataset **NonOrtho** per valutare oltre che la qualità del dataset, anche la fattibilità dell'approccio di crack detection proposto. Altro punto cruciale dell'esperimento è quello di trovare la funzione di perdita (loss function) più appropriata all'addestramento dei modelli; questo poiché, come evidenziato da diversi studi, la scelta di una loss function influenza non poco le capacità del modello di apprendere, soprattutto in contesti in cui è presente un forte sbilanciamento fra classi [31].

2.5.1 Funzioni di loss

Di seguito le loss utilizzate e le loro formulazioni (notazione: p_i = predizione (probabilità) per il pixel i; $g_i \in \{0, 1\}$ ground truth):

Binary Cross Entropy La Binary Cross Entropy (BCE), anche conosciuta come *log loss*, è la loss standard per compiti di classificazione binaria. Ha le sue origini nella teoria dell'informazione di **Shannon (1948)** e nella *logarithmic scoring rule* [32]; misura la distanza tra distribuzione predetta e distribuzione reale, penalizzando le predizioni errate. È stata introdotta in ambito statistico e poi adottata nel machine learning con l'introduzione della *backpropagation*[8] [33].

$$\mathcal{L}_{BCE} = -\frac{1}{N}\sum_{i=1}^{N}\big[g_i \log(p_i) + (1-g_i)\log(1-p_i)\big].$$

Dice loss Il *Dice coefficient* nasce in ambito statistico come misura di similarità [34]. La sua riformulazione come funzione di perdita per segmentazione è stata proposta in V-Net[9] [35]. Questa funzione di perdita enfatizza la sovrapposizione tra predizione e ground truth, risultando adatta in scenari con classi particolarmente sbilanciate.

$$\text{Dice} = \frac{2\sum_i p_i g_i + \varepsilon}{\sum_i p_i + \sum_i g_i + \varepsilon},$$

la relativa Dice loss (da minimizzare) è:

$$\mathcal{L}_{Dice} = 1 - \text{Dice}.$$

Combinazione BCE + Dice Spesso si usa una combinazione lineare delle due funzioni per avvalerci sia della stabilità della BCE che della robustezza rispetto allo sbilanciamento delle classi della Dice [35]. La formula è:

$$\mathcal{L}_{BCE+Dice} = \mathcal{L}_{BCE} + \lambda_{Dice}\,\mathcal{L}_{Dice},$$

con λ_{Dice} impostato con valori fra 0.0 a 1.0 in base alla sperimentazione.

Focal Loss La Focal Loss è stata introdotta per affrontare il problema degli esempi facili che dominano il gradiente. Essa introduce un fattore di modulazione $(1-p)^\gamma$ che aumenta il peso dei campioni difficili, riducendo quello dei campioni classificati correttamente [36]. La formula per la classificazione binaria:

$$\mathcal{L}_{Focal} = -\frac{1}{N}\sum_{i=1}^{N}\alpha(1-p_i)^\gamma g_i \log(p_i) + (1-\alpha)p_i^\gamma(1-g_i)\log(1-p_i),$$

dove γ è il *focusing parameter* (es. 2) e α bilancia le classi.

[8] algoritmo che consente l'addestramento efficiente delle reti neurali calcolando il gradiente della funzione di perdita rispetto ai pesi mediante la regola della catena.

[9] V-Net è un'architettura che si ispira all'U-Net; quest'ultima è principalmente progettata per immagini 2D, mentre la V-net è stata sviluppata per gestire dati volumetrici tridimensionali: hanno struttura simile ma utilizza convoluzioni 3D.

Tversky Loss Il Tversky index è stato proposto da Tversky (1977) come generalizzazione dell'indice di similarità, introducendo parametri α e β per pesare diversamente falsi positivi e falsi negativi [37]. Essa è particolarmente adatta quando l'interesse è ridurre errori su regioni piccole e difficili. La formula per il calcolo del Tversky Index (T) è

$$T = \frac{\sum_i p_i g_i + \varepsilon}{\sum_i p_i g_i + \alpha \sum_i p_i(1 - g_i) + \beta \sum_i (1 - p_i) g_i + \varepsilon},$$

con solitamente $\alpha + \beta = 1$. La formula della Tversky loss é:

$$\mathcal{L}_{Tversky} = 1 - T.$$

Focal Tversky loss Combina la Tversky con il principio della Focal Loss, enfatizzando ulteriormente i casi difficili. Si è dimostrata efficace in segmentazioni con estremo sbilanciamento fra classi [31].

$$\mathcal{L}_{Focal\_Tversky} = (1 - T)^{\gamma}.$$

Dunque, usando il dataset NonOrtho (descritto nella sezione 2.1), confronteremo le performance di ciascun modello in combinazione con le loss function appena elencate, ponendosi come obiettivo quello di scegliere quella che massimizza le metriche illustrate nella sezione 4.1

2.6 Transfer learning e fine-tuning

Come già ribadito nel corso del presente studio, in letteratura è nota la carenza di dataset sufficientemente ampi e variegati dedicati alle superfici in muratura [1]. Per questo motivo, il secondo esperimento è stato concepito per affrontare indirettamente tale limite, analizzando la possibilità di migliorare le prestazioni dei modelli attraverso strategie di transfer learning e fine-tuning. Come evidenziato da [24], queste tecniche risultano particolarmente efficaci quando i dati disponibili sono pochi o difficilmente reperibili: partendo da reti pre-addestrate su dataset più ampi e generici, è possibile trasferire le conoscenze acquisite e adattarle a un nuovo dominio.

Nel contesto di questo studio, il dataset parziale di Dais et al. è stato impiegato per la fase di pre-addestramento, mentre il dataset NonOrtho, raccolto ad hoc, è stato utilizzato per la valutazione e il fine-tuning del modello. L'obiettivo è dunque verificare se un modello inizialmente allenato su un dataset pubblico (ridotto) possa, tramite tale tecnica, incrementare le proprie prestazioni.

Dopo aver trovato, grazie al primo esperimento, la funzione di perdita ottimale per il nostro caso specifico, l'abbiamo sfruttata per condurre questo secondo esperimento; il procedimento seguito è il seguente:

1. Suddividere il dataset di Dais et al. (sezione 2.2) in 70% train / 15% val / 15% test.
2. Per ciascun modello:
   (a) Addestrare (train) e validare (val) su Dais (70/15), salvando i pesi di ciascun modello.
   (b) Testare sul nostro dataset (sezione 2.1) e salvare le i risultati;
   (c) Applicare *fine-tuning* dei modelli salvati in precedenza utilizzando il dataset NonOrtho e valutare di nuovo sul medesimo test set;
   (d) Confrontare i risultati **prima** e **dopo** il fine-tuning per misurare l'eventuale incremento;
3. Riportare, per ogni modello (sezione 2.4), le metriche calcolate come descritto nella sezione 4.1; inoltre presentare anche una valutazione qualitativa con esempi di predizioni.

Capitolo 3

Implementazione

Dopo aver descritto il problema, progettato una strategia per affrontarlo e delineato un piano sperimentale è necessario capire quali sono gli strumenti più adatti (sia software che hardware) che ci possono aiutare a verificare attraverso risultati tangibili la fattibilità dell'approccio proposto. Perciò questo capitolo servirà proprio per illustrare nel dettaglio le tecnologie e i framework utilizzati per l'implementazione dei modelli, delle funzioni di perdita e degli esperimenti descritti nei capitoli precedenti, nonché gli strumenti impiegati per la gestione e l'esecuzione del codice sorgente.

3.1 Strumenti utilizzati

Nel corso dello sviluppo sono stati utilizzati diversi strumenti, sia hardware che software, indispensabili per la raccolta, l'elaborazione e l'analisi dei dati.

3.1.1 Hardware

- **Server High Performance Computing (HPC)** fornito dall'universitaria per l'addestramento dei modelli, con l'utilizzo di una GPU **NVIDIA L40**.

3.1.2 Software

- **Google Colab**: ambiente di calcolo in cloud offerto da Google, che permette di eseguire notebook Jupyter su GPU, semplificando la sperimentazione e l'analisi dei dati;
- **Github** è una piattaforma online per la gestione del codice sorgente, collaborazione e controllo versione basata su Git;

- **Python**: linguaggio di programmazione ad alto livello, interpretato e multiparadigma; è risultato particolarmente adatto al contesto per via delle diverse librerie dedicate al DL;
- **GIMP**: software open-source di fotoritocco e manipolazione di immagini, utilizzato in questo lavoro per l'annotazione manuale delle maschere pixel-wise.

3.1.3 Librerie e Framework

- **PyTorch**: framework di DL open-source che fornisce strumenti per la costruzione e l'addestramento di reti neurali, con supporto ottimizzato per GPU.
- **Torchvision**: libreria complementare a PyTorch, che offre modelli pre-addestrati e funzionalità di trasformazione e gestione di dataset di immagini.
- **NumPy**: libreria per il calcolo numerico in Python, basata su array multidimensionali ad alte prestazioni e operazioni matematiche ottimizzate.
- **Pandas**: libreria per la manipolazione e l'analisi dei dati, che introduce strutture come `DataFrame` per organizzare e visualizzare i dati in forma tabellare.
- **Matplotlib**: libreria per la creazione di grafici e visualizzazioni, utilizzata per rappresentare metriche e risultati degli esperimenti.
- **scikit-learn**: libreria di machine learning che fornisce funzioni per la valutazione dei modelli, il calcolo di metriche e strumenti per l'analisi dei dati.
- **tqdm**: libreria che permette di aggiungere barre di progresso ai cicli di addestramento e validazione per poter costantemente monitorare lo stato degli esperimenti.
- **segmentation_models_pytorch (SMP)**: libreria che fornisce alcune delle funzioni di loss usate all'interno degli esperimenti *segmentation_models_pytorch.losses*; queste supportano diverse modalità (binaria, multiclass e multilabel), ma saranno usate in modalità binaria.

3.2 Modelli

In questa sezione vengono illustrate le implementazioni delle architetture descritte nella Sezione 2.4, presentando il relativo codice scritto in `PyTorch`.

3.2.1 U-Net

Il modello **U-Net**, descritto nella Sezione 2.4.1, è composto da blocchi `DoubleConv`, ciascuno costituito da due convoluzioni 3×3 seguite da Batch Normalization e ReLU:

Listing 3.1: Blocco DoubleConv in U-Net

```
class DoubleConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )
    def forward(self, x):
        return self.conv(x)
```

L'architettura principale segue una struttura encoder–decoder simmetrica: le feature vengono progressivamente compresse tramite max pooling e poi ricostruite mediante upsampling e concatenazione con le feature corrispondenti dell'encoder (skip connections). La convoluzione finale 1×1 produce la mappa binaria di segmentazione:

Listing 3.2: Struttura della rete U-Net

```
class UNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=1):
        super().__init__()
        self.enc1 = DoubleConv(in_channels, 64)
        self.enc2 = DoubleConv(64, 128)
        self.enc3 = DoubleConv(128, 256)
        self.pool = nn.MaxPool2d(2)
        self.bottleneck = DoubleConv(256, 512)
        self.up3 = nn.ConvTranspose2d(512, 256, 2, stride=2)
        self.dec3 = DoubleConv(512, 256)
        self.up2 = nn.ConvTranspose2d(256, 128, 2, stride=2)
        self.dec2 = DoubleConv(256, 128)
        self.up1 = nn.ConvTranspose2d(128, 64, 2, stride=2)
        self.dec1 = DoubleConv(128, 64)
        self.final = nn.Conv2d(64, out_channels, kernel_size=1)

    def forward(self, x):
        e1 = self.enc1(x)
        e2 = self.enc2(self.pool(e1))
        e3 = self.enc3(self.pool(e2))
        b = self.bottleneck(self.pool(e3))
        d3 = self.dec3(torch.cat([self.up3(b), e3], dim=1))
```

```
        d2 = self.dec2(torch.cat([self.up2(d3), e2], dim=1))
        d1 = self.dec1(torch.cat([self.up1(d2), e1], dim=1))
        return self.final(d1)
```

3.2.2 ResNet–FCN

Come baseline è stato impiegato un modello FCN con backbone **ResNet–50**, descritto nella Sezione 2.4.2. L'implementazione utilizza il modulo nativo di `torchvision.models.segmentation`

Listing 3.3: Implementazione del modello FCN–ResNet50

```
from torchvision.models.segmentation import fcn_resnet50

class ResNet(nn.Module):
    def __init__(self, num_classes=1):
        super().__init__()
        self.model = seg_models.fcn_resnet50(weights=None, num_classes=num_classes)

    def forward(self, x):
        return self.model(x)['out']
```

3.2.3 DeepCrack

L'implementazione di **DeepCrack**, descritta nella Sezione 2.4.3; si segue un'architettura encoder–decoder basata su blocchi `Down` e `Up` con max pooling indicizzato e unpooling.

Listing 3.4: Struttura del modello DeepCrack

```
class DeepCrack(nn.Module):
    def __init__(self):
        super().__init__()

        # Encoder: 5 livelli di convoluzione e pooling
        self.down1 = Down(nn.Sequential(ConvRelu(3,64), ConvRelu(64,64)))
        self.down2 = Down(nn.Sequential(ConvRelu(64,128), ConvRelu(128,128)))
        self.down3 = Down(nn.Sequential(ConvRelu(128,256), ConvRelu(256,256),
    ConvRelu(256,256)))
        self.down4 = Down(nn.Sequential(ConvRelu(256,512), ConvRelu(512,512),
    ConvRelu(512,512)))
        self.down5 = Down(nn.Sequential(ConvRelu(512,512), ConvRelu(512,512),
    ConvRelu(512,512)))

        # Decoder: 5 livelli simmetrici con unpooling
        self.up1 = Up(nn.Sequential(ConvRelu(64,64), ConvRelu(64,64)))
        self.up2 = Up(nn.Sequential(ConvRelu(128,128), ConvRelu(128,64)))
        self.up3 = Up(nn.Sequential(ConvRelu(256,256), ConvRelu(256,256), ConvRelu
    (256,128)))
```

```
        self.up4 = Up(nn.Sequential(ConvRelu(512,512), ConvRelu(512,512), ConvRelu(512,256)))
        self.up5 = Up(nn.Sequential(ConvRelu(512,512), ConvRelu(512,512), ConvRelu(512,512)))

        # Moduli di fusione multi-scala
        self.fuse5 = Fuse(ConvRelu(512+512,64), scale=16)
        self.fuse4 = Fuse(ConvRelu(512+256,64), scale=8)
        self.fuse3 = Fuse(ConvRelu(256+128,64), scale=4)
        self.fuse2 = Fuse(ConvRelu(128+64,64), scale=2)
        self.fuse1 = Fuse(ConvRelu(64+64,64), scale=1)

        self.final = Conv3X3(5,1)
```

Ogni livello dell'encoder salva gli indici del pooling per consentire al decoder di ricostruire la dimensione spaziale originale.

Listing 3.5: Blocco Down con MaxPool indicizzato

```
class Down(nn.Module):
    def __init__(self, nn):
        super(Down,self).__init__()
        self.nn = nn
        self.maxpool_with_argmax = torch.nn.MaxPool2d(2,2,return_indices=True)

    def forward(self, inputs):
        down = self.nn(inputs)
        outputs, indices = self.maxpool_with_argmax(down)
        return outputs, down, indices, down.size()
```

Il blocco `Up` ricostruisce la risoluzione spaziale originaria tramite `MaxUnpool2d`, utilizzando gli indici salvati dall'encoder per garantire una corrispondenza perfetta tra feature map:

Listing 3.6: Blocco UP con unpooling

```
class Up(nn.Module):
    def __init__(self, nn):
        super().__init__()
        self.nn = nn
        self.unpool=torch.nn.MaxUnpool2d(2,2)

    def forward(self,inputs,indices,output_shape):
        outputs = self.unpool(inputs, indices=indices, output_size=output_shape)
        outputs = self.nn(outputs)
        return outputs
```

Per le fusione multi-scala viene usato il modulo `Fuse`, che concatena feature provenienti da diverse profondità e ne interpola la dimensione per la somma finale:

Listing 3.7: Fusione multi-scala in DeepCrack

```
class Fuse(nn.Module):
    def __init__(self, nn, scale):
        super().__init__()
        self.nn = nn
        self.scale = scale
        self.conv = nn.Conv2d(64,1,3,padding=1)

    def forward(self, down_inp, up_inp):
        x = torch.cat([down_inp, up_inp], dim=1)
        x = F.interpolate(x, scale_factor=self.scale, mode='bilinear')
        return self.conv(self.nn(x))
```

Il forward produce cinque fusioni (`fuse1-5`) e una mappa finale di predizione, concatenando tutte le scale e applicando un'ulteriore convoluzione 1×1. L'output principale è una maschera binaria della stessa dimensione dell'immagine di input.

Listing 3.8: Forward del modello DeepCrack

```
def forward(self, x):
    out, d1, i1, s1 = self.down1(x)
    out, d2, i2, s2 = self.down2(out)
    out, d3, i3, s3 = self.down3(out)
    out, d4, i4, s4 = self.down4(out)
    out, d5, i5, s5 = self.down5(out)

    up5 = self.up5(out, i5, s5)
    up4 = self.up4(up5, i4, s4)
    up3 = self.up3(up4, i3, s3)
    up2 = self.up2(up3, i2, s2)
    up1 = self.up1(up2, i1, s1)

    fuse5 = self.fuse5(d5, up5)
    fuse4 = self.fuse4(d4, up4)
    fuse3 = self.fuse3(d3, up3)
    fuse2 = self.fuse2(d2, up2)
    fuse1 = self.fuse1(d1, up1)

    out = self.final(torch.cat([fuse5,fuse4,fuse3,fuse2,fuse1], dim=1))
    return out, fuse5, fuse4, fuse3, fuse2, fuse1
```

3.2.4 DeepCrackAT

L'architettura **DeepCrackAT**, illustrata nella Sezione 2.4.5, estende il modello originale `DeepCrack` con l'integrazione di moduli HDC, l'uso di blocchi di attenzione CBAM tramite ASFB e l'aggiunta di un *Token–MLP* nel bottleneck.

Listing 3.9: Struttura del modello DeepCrackAT

```
class DeepCrackAT(nn.Module):
    def __init__(self, in_channels:int=3, base_ch:int=32):
        super().__init__()

        # Encoder
        self.enc1 = EncoderStage(in_channels, base_ch, use_hdc=False)
        self.enc2 = EncoderStage(base_ch, base_ch*2, use_hdc=True)
        self.enc3 = EncoderStage(base_ch*2, base_ch*4, use_hdc=True)
        self.enc4 = EncoderStage(base_ch*4, base_ch*8, use_hdc=False)

        # Bottleneck
        self.bottleneck = nn.Sequential(
            ConvBNReLU(base_ch*8, base_ch*16),
            TokMLPBlock(base_ch*16)
        )

        # Decoder
        self.dec4 = DecoderStage(base_ch*16, base_ch*8)
        self.dec3 = DecoderStage(base_ch*8, base_ch*4)
        self.dec2 = DecoderStage(base_ch*4, base_ch*2)
        self.dec1 = DecoderStage(base_ch*2, base_ch)

        # Blocchi di fusione attentive (ASFB)
        self.asfb4 = ASFB(base_ch*8)
        self.asfb3 = ASFB(base_ch*4)
        self.asfb2 = ASFB(base_ch*2)
        self.asfb1 = ASFB(base_ch)

        # Uscite multi-scala
        self.out5 = nn.Conv2d(base_ch*8, 1, 1)
        self.out4 = nn.Conv2d(base_ch*4, 1, 1)
        self.out3 = nn.Conv2d(base_ch*2, 1, 1)
        self.out2 = nn.Conv2d(base_ch, 1, 1)
        self.out1 = nn.Conv2d(base_ch, 1, 1)

        # Fusione finale
        fuse_in = base_ch*(8 + 4 + 2 + 1)
        self.fuse_conv = nn.Conv2d(fuse_in, base_ch, 1, bias=False)
        self.fuse_out = nn.Conv2d(base_ch, 1, 1)
```

Ogni `EncoderStage` applica convoluzioni standard o dilatate a seconda del livello, come mostrato in 3.10. Si usa inoltre `HDCBlock` per ampliare il campo recettivo senza ridurre la risoluzione spaziale:

Listing 3.10: Hybrid Dilated Convolution

```
class HDCBlock(nn.Module):
    def __init__(self, in_ch:int, out_ch:int, dilations=(1,2,3)):
        super().__init__()
        self.convs = nn.ModuleList([
            nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=d, dilation=d)
            for d in dilations
        ])
        self.bn = nn.BatchNorm2d(out_ch)
        self.act = nn.ReLU(inplace=True)
    def forward(self, x):
        out = sum(conv(x) for conv in self.convs) / len(self.convs)
        return self.act(self.bn(out))
```

Il blocco `ASFB` consente di fondere feature di skip connection passando prima dal CBAM:

Listing 3.11: Attentional Skip-layer Fusion Block

```
class ASFB(nn.Module):
    def __init__(self, in_ch:int):
        super().__init__()
        self.conv1x1 = nn.Conv2d(in_ch, in_ch, 1, bias=False)
        self.cbam = CBAM(in_ch)
    def forward(self, x):
        return self.cbam(self.conv1x1(x))
```

Dopo il bottleneck, con il blocco Tok-MLP, le mappe dei decoder vengono interpolate per riportarle alla risoluzione originale e infine combinate in cinque output parziali (`out1-out5`); questi, successivamente vengono fusi per genera una mappa binaria finale, risultante dall'integrazione multi-scala delle informazioni globali e locali.

Listing 3.12: Forward di DeepCrackAT

```
def forward(self, x):
        B,C,H,W = x.shape

        # Encoder
        e1, p1 = self.enc1(x)
        e2, p2 = self.enc2(p1)
        e3, p3 = self.enc3(p2)
        e4, p4 = self.enc4(p3)

        # Bottleneck
        b = self.bottleneck(p4)

```

```
        # Decoder + ASFB
        d4 = self.dec4(b, self.asfb4(e4))
        d3 = self.dec3(d4, self.asfb3(e3))
        d2 = self.dec2(d3, self.asfb2(e2))
        d1 = self.dec1(d2, self.asfb1(e1))

        # Outputs (upsampled to input size)
        out5 = F.interpolate(self.out5(d4), size=(H,W), mode='bilinear', align_corners=False)
        out4 = F.interpolate(self.out4(d3), size=(H,W), mode='bilinear', align_corners=False)
        out3 = F.interpolate(self.out3(d2), size=(H,W), mode='bilinear', align_corners=False)
        out2 = F.interpolate(self.out2(d1), size=(H,W), mode='bilinear', align_corners=False)
        out1 = F.interpolate(self.out1(d1), size=(H,W), mode='bilinear', align_corners=False)

        # Fuse
        fused = torch.cat([
            F.interpolate(d4, size=(H,W), mode='bilinear', align_corners=False),
            F.interpolate(d3, size=(H,W), mode='bilinear', align_corners=False),
            F.interpolate(d2, size=(H,W), mode='bilinear', align_corners=False),
            F.interpolate(d1, size=(H,W), mode='bilinear', align_corners=False)
        ], dim=1)
        fused = self.fuse_out(self.fuse_conv(fused))

        return { fused, out5, out4, out3, out2, out1 }
```

3.2.5 MurCrackNet

L'implementazione di **MurCrackNet**, introdotta nella Sezione 2.4.4, adotta un backbone **ResNet–50** pre–addestrato come estrattore di feature, arricchito con moduli di attenzione e convoluzioni dilatate.

Listing 3.13: Struttura del modello MurCrackNet

```
class MurCrackNet(nn.Module):
    def __init__(self, num_classes=1):
        super().__init__()
        backbone = resnet50(pretrained=True)
        self.enc1 = nn.Sequential(backbone.conv1, backbone.bn1, backbone.relu)
        self.enc2 = backbone.layer1
        self.enc3 = backbone.layer2
        self.enc4 = backbone.layer3
        self.enc5 = backbone.layer4

        # HDC sulle feature iniziali
```

```
        self.hdc1 = HDCBlock(64,64)
        self.hdc2 = HDCBlock(256,256)

        # Tok-MLP sulle feature profonde
        self.tokmlp = TokMLP(2048)

        # Attenzione CBAM sui livelli medi
        self.cbam3 = CBAM(512)
        self.cbam4 = CBAM(1024)

        # Decoder
        self.up4 = nn.ConvTranspose2d(2048, 1024, 2, 2)
        self.up3 = nn.ConvTranspose2d(1024, 512, 2, 2)
        self.up2 = nn.ConvTranspose2d(512, 256, 2, 2)
        self.up1 = nn.ConvTranspose2d(256, 64, 2, 2)
        self.final = nn.Conv2d(64, num_classes, 1)
```

Il blocco `HDCBlock` sfrutta più convoluzioni dilatate con diversi tassi di dilatazione per estendere il campo recettivo senza aumentare i parametri:

Listing 3.14: Blocco HDC per MurCrackNet

```
class HDCBlock(nn.Module):
    def __init__(self, in_ch, out_ch, dilations=[1,2,3]):
        super().__init__()
        self.convs = nn.ModuleList([
            nn.Conv2d(in_ch, out_ch, 3, padding=d, dilation=d)
            for d in dilations
        ])
        self.bn = nn.BatchNorm2d(out_ch)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        out = sum(conv(x) for conv in self.convs) / len(self.convs)
        return self.relu(self.bn(out))
```

Per enfatizzare le feature più significative, viene adottato il modulo CBAM, composto da due sotto–moduli: `ChannelAttention` e `SpatialAttention`.

Listing 3.15: Modulo CBAM usato in MurCrackNet

```
class CBAM(nn.Module):
    def __init__(self, in_planes):
        super().__init__()
        self.ca = ChannelAttention(in_planes)
        self.sa = SpatialAttention()
    def forward(self,x):
        x = x * self.ca(x)
        x = x * self.sa(x)
        return x
```

Il Token-MLP (Token-based Multi-Layer Perceptron) è un blocco che rielabora le feature estratte dalla CNN trattandole come token, cioè piccoli vettori rappresentativi delle regioni dell'immagine; dopo aver estratto questi token, applica trasformazioni non lineari tramite livelli Multi-Layer Perceptron (MLP) per poi riproiettare il tutto in uno spazio 2D

Listing 3.16: Blocco Tok–MLP

```
class TokMLP(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dwconv = nn.Conv2d(dim, dim, kernel_size=3, padding=1, groups=dim)
        self.norm = nn.LayerNorm(dim)
        self.fc = nn.Linear(dim, dim)
    def forward(self, x):
        B,C,H,W = x.shape
        out = self.dwconv(x)
        out = out.flatten(2).transpose(1,2)
        out = self.fc(self.norm(out))
        return out.transpose(1,2).view(B,C,H,W)
```

Nel decoder del modello, le feature estratte nei livelli profondi vengono progressivamente ricostruite e riallineate alle dimensioni originali mediante upsampling bilineare per poi sommare tali mappe in modo residuo con le features dell'encoder (skip connection).

Listing 3.17: Forward del modello MurCrackNet

```
def forward(self,x):
    e1 = self.hdc1(self.enc1(x))
    e2 = self.hdc2(self.enc2(e1))
    e3 = self.cbam3(self.enc3(e2))
    e4 = self.cbam4(self.enc4(e3))
    e5 = self.tokmlp(self.enc5(e4))

    d4 = self.up4(e5)
    d4 = F.interpolate(d4, size=e4.shape[2:], mode='bilinear') + e4
    d3 = self.up3(d4)
    d3 = F.interpolate(d3, size=e3.shape[2:], mode='bilinear') + e3
    d2 = self.up2(d3)
    d2 = F.interpolate(d2, size=e2.shape[2:], mode='bilinear') + e2
    d1 = self.up1(d2)
    d1 = F.interpolate(d1, size=e1.shape[2:], mode='bilinear') + e1
    out = self.final(d1)
    return F.interpolate(out, size=x.shape[2:], mode='bilinear')
```

3.3 Gestione dei dataset

Per la presente ricerca sono stati utilizzati i due dataset descritti nelle sezioni 2.1 e 2.2. Per gestire l'accesso ai dati sono state implementate due classi Python, rispettivamente una per dataset, estendendo `torch.utils.data.Dataset`.

3.3.1 Preprocessing e normalizzazione

Viene usata la libreria `torchvision` per rendere i dati compatibili con le CNN; le immagini vengono convertite in tensori e normalizzate con media e deviazione standard pari a 0.5 per ciascun canale RGB:

Listing 3.18: Normalizzazione delle immagini

```
self.base_transform = T.Compose([
    T.ToTensor(),
    T.Normalize(mean=[0.5,0.5,0.5], std=[0.5,0.5,0.5])
])
```

Invece, le maschere binarie vengono convertite in tensori e sottoposte a una sogliatura, così da ottenere valori 0 o 1 per ogni pixel:

Listing 3.19: Preprocessing delle maschere binarie

```
mask = self.mask_transform(mask)
mask = (mask > 0.5).float()
```

3.3.2 Data augmentation

Sempre grazie alla libreria `torchvision`, riusciamo ad avere augmentation dei dati per aumentare la varietà delle immagini e migliorare la generalizzazione del modello. Le trasformazioni principali includono flip orizzontali e verticali, rotazioni casuali di 0, 90, 180 e 270 gradi, e variazioni di luminosità, contrasto, saturazione e tonalità:

Listing 3.20: Esempio di data augmentation

```
if self.augment:
    if torch.rand(1) < 0.5:
        image = TF.hflip(image); mask = TF.hflip(mask)
    if torch.rand(1) < 0.5:
        angle = int(np.random.choice([0, 90, 180, 270]))
        image = TF.rotate(image, angle); mask = TF.rotate(mask, angle)
    if torch.rand(1) < 0.5:
        image = TF.adjust_brightness(image, 0.8 + 0.4*torch.rand(1).item())
```

Queste trasformazioni vengono applicate in maniera coerente a immagini e maschere.

3.3.3 Creazione dei DataLoader

Per addestramento, validazione e test sono stati creati *DataLoader* tramite PyTorch, che gestiscono batching, shuffle e campionamento casuale.
La funzione `get_dataloader` consente di costruire rapidamente un `DataLoader` a partire da directory di immagini e maschere, richiamando la classe del dataset 2.1

Listing 3.21: Funzione get_dataloader per il dataset creato

```
def get_dataloader(img_dir, mask_dir, transform=None, batch_size=8,
        shuffle=True, num_samples=None, augment=False, normalize=True):
    dataset = NonOrthoDataset(img_dir, mask_dir,
            transform=transform, augment=augment, normalize=normalize)

    if num_samples is not None:
        sampler = RandomSampler(dataset, replacement=True, num_samples=num_samples)
        loader = DataLoader(dataset, batch_size=batch_size, sampler=sampler)
    else:
        loader = DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)

    return dataset, loader
```

Per il dataset 2.2 è stata implementata la funzione `get_Dais_loaders`, che effettua automaticamente lo split dei dati in training, validation e test (70%, 15%, 15%), abilitando l'augmentation solo sul training set e creando tre `DataLoader` distinti:

Listing 3.22: Funzione get_Dais_loaders per Dais et al.

```
def get_Dais_loaders(img_dir, mask_dir, batch_size=8, train_ratio=0.7,
        val_ratio=0.15, num_samples=None, normalize=True):

    full_dataset = DaisDataset(img_dir, mask_dir, augment=False, normalize=normalize)

    n_train = int(train_ratio * len(full_dataset))
    n_val = int(val_ratio * len(full_dataset))
    n_test = len(full_dataset) - n_train - n_val

    train_dataset, val_dataset, test_dataset = random_split(
        full_dataset, [n_train, n_val, n_test])

    train_dataset.dataset.augment = True

    if num_samples is not None:
        train_sampler = RandomSampler(train_dataset,
            replacement=True, num_samples=num_samples)
        train_loader = DataLoader(train_dataset,
            batch_size=batch_size, sampler=train_sampler)
    else:
        train_loader = DataLoader(train_dataset,
            batch_size=batch_size, shuffle=True)
```

```
23
24      val_loader = DataLoader(val_dataset,
25          batch_size=batch_size, shuffle=False)
26      test_loader = DataLoader(test_dataset,
27          batch_size=batch_size, shuffle=False)
28
29      return train_loader, val_loader, test_loader
```

3.4 Funzioni di loss

Le principali funzioni di *loss* impiegate per l'addestramento dei modelli sono descritte nella sezione 2.5.1. Per gli esperimenti, le implementazioni standard delle loss BCE, Dice, Focal e Tversky sono state utilizzate direttamente dalla libreria `Segmentation Models PyTorch (SMP)`. Per alcune funzioni, invece, è stata fornita una nostra implementazione:

- La **BCEDice** è stata implementata combinando la BCE e la Dice Loss;
- La **Focal Tversky Loss (FTL)** è stata implementata sotto la classe `FocalTverskyLoss`.

Listing 3.23: Implementazione della funzione di loss BCEDice

```
1  class BCEDiceLoss(nn.Module):
2      def __init__(self, lambda_dice=0.65, pos_weight=None, log_loss=False):
3          super().__init__()
4          self.lambda_dice = lambda_dice
5          self.pos_weight = pos_weight
6          self.dice = DiceLoss(mode="binary", log_loss=log_loss)
7
8      def forward(self, pred, target):
9          if self.pos_weight is not None:
10             pos_weight = torch.tensor([self.pos_weight], dtype=torch.float32, device=
       pred.device)
11             bce = nn.BCEWithLogitsLoss(pos_weight=pos_weight)(pred, target)
12         else:
13             bce = nn.BCEWithLogitsLoss()(pred, target)
14
15         d_loss = self.dice(pred, target)
16         return bce + self.lambda_dice * d_loss
```

Listing 3.24: Implementazione della funzione di loss FTL

```
1  class FocalTverskyLoss(nn.Module):
2      def __init__(self, alpha=0.7, beta=0.3, gamma=4/3, smooth=1e-6):
3          super().__init__()
4          self.alpha = alpha
5          self.beta = beta
```

```
        self.gamma = gamma
        self.smooth = smooth

    def forward(self, preds, targets):
        preds = torch.sigmoid(preds).view(-1)
        targets = targets.view(-1)

        TP = (preds * targets).sum()
        FP = ((1 - targets) * preds).sum()
        FN = (targets * (1 - preds)).sum()

        TI = (TP + self.smooth) / (TP + self.alpha * FP + self.beta * FN + self.
    smooth)
        return (1 - TI) ** self.gamma
```

3.5 Metriche

Le metriche descritte nella sezione 4.1 sono state calcolate usando l'apposita libreria *scikit-learn* per entrambe le classi (foreground, cls=1) e background (cls=0), includendo anche la media tra le due:

Listing 3.25: Codice per il calcolo delle metriche

```
def compute_metrics(outputs: torch.Tensor, labels: torch.Tensor,
        threshold: float = 0.5):
    preds = (torch.sigmoid(outputs) > threshold).long().cpu().numpy().ravel()
    labels = (labels > 0.5).long().cpu().numpy().ravel()

    per_class_metrics = {}
    all_metrics = {"iou": [], "dice": [], "precision": [], "recall": [], "f1": []}

    # ciclo su background=0 e foreground=1
    for cls, name in zip([0, 1], ["background", "foreground"]):
        precision = precision_score(labels, preds, pos_label=cls)
        recall = recall_score(labels, preds, pos_label=cls)
        f1 = f1_score(labels, preds, pos_label=cls)
        iou = jaccard_score(labels, preds, pos_label=cls)
        dice = (2 * precision * recall) / (precision + recall + 1e-6)

        per_class_metrics[name] = {
            "iou": float(iou), "dice": float(dice), "f1": float(f1)
            "precision": float(precision), "recall": float(recall),
        }

        all_metrics["iou"].append(iou)
        all_metrics["dice"].append(dice)
        all_metrics["precision"].append(precision)
```

```
        all_metrics["recall"].append(recall)
        all_metrics["f1"].append(f1)

    # media tra le due classi
    mean_metrics = {k: float(np.mean(v)) for k, v in all_metrics.items()}

    return {    "mean": mean_metrics,
                "foreground": per_class_metrics["foreground"],
                "background": per_class_metrics["background"]  }
```

3.6 Esperimenti

Per gli esperimenti effettuati è stato innanzitutto sviluppato un file `config.py` con al suo interno i valori dei parametri di addestramento dei modelli:

Listing 3.26: Codice config.py

```
NUM_EPOCHS = 200
PATIENCE_ES = 10
PATIENCE_LR = 3
FACTOR_LR = 0.5

MODEL_CONFIGS = {
    "DeepCrack": {
        "lr": 1e-5,
        "weight_decay": 1e-5,
        "pos_weight": 10.0,
        "gradient_clip": 1.0,
        "criterion": "TverskyLoss"
    },
    "DeepCrack": {
        "lr": 1e-5,
        "weight_decay": 1e-5,
        "pos_weight": 10.0,
        "gradient_clip": 1.0,
        "criterion": "TverskyLoss"
    },
    "UNet": {
        "lr": 1e-5,
        "weight_decay": 1e-5,
        "pos_weight": 10.0,
        "gradient_clip": 1.5,
        "criterion": "TverskyLoss"
    },
    "ResNet": {
        "lr": 2e-5,
        "weight_decay": 1e-5,
```

```
        "pos_weight": 10.0,
        "gradient_clip": 3.0,
        "criterion": "TverskyLoss"
    },
    "MurCrackNet": {
        "lr": 5e-4,
        "weight_decay": 1e-5,
        "pos_weight": 10.0,
        "gradient_clip": 1.0,
        "criterion": "TverskyLoss"
    }
}
```

3.6.1 Training e Test

Per l'addestramento, validazione e test dei modelli (in entrambi gli esperimenti) è stato realizzato il modulo `train_utils.py`, con le funzioni `train_model` e `test_model`.

La funzione `train_model` gestisce l'addestramento per un numero prefissato di epoche (preso come parametro) per un singolo modello. All'inizio questo viene spostato sul dispositivo scelto (CPU o GPU) e vengono inizializzati i parametri per l'*early stopping*, in modo da interrompere l'allenamento nel caso in cui la loss di validazione non migliori per un certo numero di epoche consecutive. Viene inoltre creato uno storico in cui registrare, epoca dopo epoca, l'andamento delle loss e delle metriche calcolate (4.1).

Per ogni epoca, attraverso un ciclo sui batch forniti dal `DataLoader`, si esegue la backpropagation, il calcolo della loss e l'aggiornamento dei pesi gestendo anche gli output multipli di `DeepCrack`, adottando la tecnica della *deep supervision*: la loss totale è ottenuta sommando le loss calcolate su ciascun output. Terminata il training, si richiama `test_model` per la validazione. In questa fase vengono calcolate sia la loss media (mean) sia le restanti metriche, con l'ausilio della funzione `compute_metrics`. Infine, viene salvato il modello con la migliore loss di validazione.

La funzione `test_model`, invece, scorre sull'intero `DataLoader` di validazione. Anche in questo caso viene gestita la specificità di `DeepCrack` con i suoi output multipli. Anche qui viene usata `compute_metrics` e si ritornano le metriche calcolate.

Listing 3.27: Codice per addestramento validazione e test dei modelli

```
def train_model(model, train_loader, val_loader, criterion, optimizer,
        device, num_epochs=25, save_path=None, patience=5,
        scheduler=None, verbose=False, batch_size=None):
    model.to(device)
```

```
    best_loss = float("inf")
    epochs_no_improve = 0

    history = { "train_loss": [], "val_loss": [], "mean_iou": [],
        "mean_dice": [], "mean_precision": [],
        "mean_recall": [], "mean_f1": [] }

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0

        loop = tqdm(train_loader, desc=f"Epoch [{epoch+1}/{num_epochs}] Training",
    leave=False)
        for inputs, targets in loop:
            inputs, targets = inputs.to(device), targets.to(device)
            optimizer.zero_grad()

            if isinstance(model, DeepCrack) or isinstance(model, DeepCrackAT):
                outputs, fuse1, fuse2, fuse3, fuse4, fuse5 = model(inputs)
                loss = (
                    criterion(outputs.view(-1, 1), targets.view(-1, 1)) / batch_size
    +
                    criterion(fuse1.view(-1, 1), targets.view(-1, 1)) / batch_size +
                    criterion(fuse2.view(-1, 1), targets.view(-1, 1)) / batch_size +
                    criterion(fuse3.view(-1, 1), targets.view(-1, 1)) / batch_size +
                    criterion(fuse4.view(-1, 1), targets.view(-1, 1)) / batch_size +
                    criterion(fuse5.view(-1, 1), targets.view(-1, 1)) / batch_size
                )
            else:
                outputs = model(inputs)
                loss = criterion(outputs, targets)

            loss.backward()
            optimizer.step()
            running_loss += loss.item()

            loop.set_postfix(loss=loss.item())

        avg_train_loss = running_loss / len(train_loader)

        val_results = test_model(model, val_loader, criterion, device,
                                 epoch=epoch+1, num_epochs=num_epochs)

        if verbose:
            print(f"\nEpoch [{epoch+1}/{num_epochs}] "
                  f"Train Loss: {avg_train_loss:.4f} | Val Loss: {val_results['loss
    ']:.4f} | "
                  f"IoU: {val_results['mean']['iou']:.4f} | Dice: {val_results['mean
    ']['dice']:.4f}")
```

```
50
51          if scheduler:
52              if isinstance(scheduler, torch.optim.lr_scheduler.ReduceLROnPlateau):
53                  scheduler.step(val_results["loss"])
54              else:
55                  scheduler.step()
56
57          if val_results["loss"] < best_loss:
58              best_loss = val_results["loss"]
59              epochs_no_improve = 0
60              if save_path:
61                  torch.save(model.state_dict(), save_path)
62          else:
63              epochs_no_improve += 1
64              if epochs_no_improve >= patience:
65                  print(f"Early stopping at epoch {epoch+1}")
66                  break
67
68          history["train_loss"].append(avg_train_loss)
69          history["val_loss"].append(val_results["loss"])
70          history["mean_iou"].append(val_results["mean"]["iou"])
71          history["mean_dice"].append(val_results["mean"]["dice"])
72          history["mean_precision"].append(val_results["mean"]["precision"])
73          history["mean_recall"].append(val_results["mean"]["recall"])
74          history["mean_f1"].append(val_results["mean"]["f1"])
75      return history
76
77  def test_model(model, data_loader, criterion, device, epoch=None, num_epochs=None):
78      model.eval()
79      running_loss = 0.0
80      agg = { "mean": {"iou":0, "dice":0, "precision":0, "recall":0, "f1":0},
81          "foreground": {"iou":0, "dice":0, "precision":0, "recall":0, "f1":0},
82          "background": {"iou":0, "dice":0, "precision":0, "recall":0, "f1":0} }
83
84      desc = f"Validation" if epoch is None else f"Epoch [{epoch}/{num_epochs}]
    Validation"
85      loop = tqdm(data_loader, desc=desc, leave=False)
86
87      with torch.no_grad():
88          for inputs, targets in loop:
89              inputs, targets = inputs.to(device), targets.to(device)
90
91              if isinstance(model, DeepCrack) or isinstance(model, DeepCrackAT):
92                  outputs, fuse1, fuse2, fuse3, fuse4, fuse5 = model(inputs)
93                  loss = (
94                      criterion(outputs.view(-1, 1), targets.view(-1, 1)) / batch_size
    +
95                      criterion(fuse1.view(-1, 1), targets.view(-1, 1)) / batch_size +
96                      criterion(fuse2.view(-1, 1), targets.view(-1, 1)) / batch_size +
```

```
                    criterion(fuse3.view(-1, 1), targets.view(-1, 1)) / batch_size +
                    criterion(fuse4.view(-1, 1), targets.view(-1, 1)) / batch_size +
                    criterion(fuse5.view(-1, 1), targets.view(-1, 1)) / batch_size
                )
            else:
                outputs = model(inputs)
                loss = criterion(outputs, targets)

            running_loss += loss.item() * inputs.size(0)
            metrics = compute_metrics(outputs, targets)
            for section in ["mean", "foreground", "background"]:
                for k in agg[section].keys():
                    agg[section][k] +=
                        metrics[section][k] * inputs.size(0)

    n = len(data_loader.dataset)
    return { "loss": running_loss / n,
        "mean": {k: v/n for k,v in agg["mean"].items()},
        "foreground": {k: v/n for k,v in agg["foreground"].items()},
        "background": {k: v/n for k,v in agg["background"].items()} }
```

3.6.2 Esperimento 1: Scelta loss migliore

Il primo esperimento è stato implementato all'interno del file `experiment_1.py`. L'obiettivo è valutare le architetture proposte(2.4) in combinazione con le diverse funzioni di loss presentate (2.5.1) sul nostro dataset.

Listing 3.28: Definizione dei modelli e delle losses da testare

```
models = {  "UNet": UNet(),
            "MurCrackNet": MurCrackNet(),
            "DeepCrack": DeepCrack(),
            "ResNet": ResNet(),
            "DeepCrackAT": DeepCrackAT() }

losses = {  "BCE": nn.BCEWithLogitsLoss(),
            "Dice": DiceLoss(mode="binary", log_loss=False),
            "BCEDice": BCEDiceLoss(lambda_dice=0.65, pos_weight=10.0),
            "Focal": FocalLoss(mode="binary"),
            "Tversky": TverskyLoss(mode="binary", alpha=0.4, beta=0.6,
                log_loss=False),
            "Focal+Tversky": FocalTverskyLoss(alpha=0.4, beta=0.6) }
```

Il nucleo dell'esperimento consiste in un doppio ciclo annidato su modelli e funzioni di loss. Per ogni combinazione viene creato l'ottimizzatore (`AdamW`) e lo scheduler (`ReduceLROnPlateau`) per il controllo dinamico del learning rate; si richiama quindi la

funzione `train_model` che esegue l'addestramento e salva il modello con le migliori prestazioni su validation set. Terminata questa fase, il modello viene ricaricato e valutato tramite la funzione `test_model` sull'insieme di test salvando i risultati.

Listing 3.29: Loop su modelli e funzioni di loss

```
results = []
for model_name, model in models.items():
    config = MODEL_CONFIGS[model_name]
    for loss_name, loss_fn in losses.items():
        optimizer = optim.AdamW(model.parameters(), lr=config["lr"],
            weight_decay=config["weight_decay"])
        scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
             mode='min', factor=FACTOR_LR, patience=PATIENCE_LR, min_lr=1e-6)

        best_model_path = os.path.join(MODEL_DIR, f"{model_name}_{loss_name}_best.pth
    ")

        history = train_model(
            model, train_loader_nonortho, val_loader_nonortho, criterion=loss_fn,
            optimizer=optimizer, device=DEVICE, num_epochs=NUM_EPOCHS,
            save_path=best_model_path, scheduler=scheduler, batch_size=BATCH_SIZE)

        model_best = model.__class__()
        model_best.to(DEVICE)
        model_best.load_state_dict(torch.load(best_model_path, map_location=DEVICE))

        test_metrics = test_model(model_best, test_loader_nonortho, loss_fn,
            device=DEVICE, batch_size=BATCH_SIZE)

        results.append({
            "Model": model_name, "Loss": loss_name,
            "Val_Loss": history["val_loss"][-1],
            "Mean_IoU": history["mean_iou"][-1],
            "Mean_Dice": history["mean_dice"][-1],
            "Mean_Precision": history["mean_precision"][-1],
            "Mean_Recall": history["mean_recall"][-1],
            "Mean_F1": history["mean_f1"][-1],
            "Test_Loss": test_metrics["loss"],
            "Test_IoU": test_metrics["mean"]["iou"],
            "Test_Dice": test_metrics["mean"]["dice"],
            "Test_Precision": test_metrics["mean"]["precision"],
            "Test_Recall": test_metrics["mean"]["recall"],
            "Test_F1": test_metrics["mean"]["f1"],
        })
```

3.6.3 Esperimento 2: Transfer learning e fine-tuning

Il secondo esperimento, implementato nel file `experiment_2.py`, ha come obiettivo lo studio di un approccio di *transfer learning* e *fine-tuning* su due dataset distinti: da un lato una versione ridotta del dataset di Dais et al. (descritto in 2.2), e dall'altro il dataset creato da noi (descritto in 2.1).

Inizialmente ciascun modello viene addestrato sul dataset di Dais et Al. con la funzione di loss specificata in `MODEL_CONFIGS`. I modelli migliori vengono salvati e i risultati del training archiviati:

Listing 3.30: Training dei modelli su Dais

```
results_Dais = {}
for model_name, model in models.items():
    config = MODEL_CONFIGS[model_name]
    if config["criterion"] == "BCEDiceLoss":
        criterion = BCEDiceLoss(lambda_dice=0.65, pos_weight=config["pos_weight"]).to
    (DEVICE)
    elif config["criterion"] == "TverskyLoss":
        criterion = TverskyLoss(mode="binary", alpha=0.4, beta=0.6, log_loss=False).
    to(DEVICE)
    else:
        pos_weight = torch.tensor([config["pos_weight"]]).to(DEVICE)
        criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
    optimizer = optim.AdamW(model.parameters(), lr=config["lr"], weight_decay=config[
    "weight_decay"])
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, mode='min', factor=FACTOR_LR,
        patience=PATIENCE_LR, min_lr=1e-6
    )
    history = train_model(
        model.to(DEVICE), train_loader=train_loader_v,
        val_loader=val_loader_v, criterion=criterion,
        optimizer=optimizer, device=DEVICE,
        num_epochs=NUM_EPOCHS, patience=PATIENCE_ES,
        save_path=os.path.join(MODEL_DIR, f"{model_name}_Dais_best.pth"),
        scheduler=scheduler, verbose=True, batch_size=BATCH_SIZE
    )
    results_Dais[model_name] = history
```

Dopo, i modelli addestrati nella fase precedente vengono caricati e valutati sull'insieme di test del dataset auto-prodotto, senza alcun riaddestramento, per osservare la capacità di generalizzazione cross-dataset:

Listing 3.31: Valutazione su NonOrtho prima del fine-tuning

```
metrics_nonortho_pre = {}
for model_name, model in models.items():
    model.load_state_dict(torch.load(
        os.path.join(MODEL_DIR, f"{model_name}_Dais_best.pth")))

    metrics_nonortho_pre[model_name] = test_model(
        model.to(DEVICE), test_loader_nonortho,
        criterion, DEVICE, batch_size=BATCH_SIZE )
```

Infine, facciamo fine-tuning: ogni modello viene inizializzato con i pesi ottenuti dalla prima fase e riaddestrato per un numero ridotto di epoche sul dataset NonOrtho. In questo caso viene utilizzata la funzione `get_optimizer` per applicare un *layer-wise fine-tuning* e affinare selettivamente parti della rete:

Listing 3.32: Fine-tuning dei modelli su NonOrtho

```
results_finetune = {}
metrics_nonortho_post = {}
for model_name, model in models.items():
    model.load_state_dict(torch.load(os.path.join(MODEL_DIR, f"{model_name}_Dais_best
    .pth")))
    config = MODEL_CONFIGS[model_name]
    # definizione del criterion come sopra
    optimizer = get_optimizer(model, model_name, config)
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, mode='min', factor=FACTOR_LR, patience=PATIENCE_LR, min_lr=1e-6
    )
    history_ft = train_model(
        model, train_loader_nonortho, val_loader_nonortho,
        criterion, optimizer, DEVICE, NUM_EPOCHS // 2,
        os.path.join(MODEL_DIR, f"{model_name}_NonOrthoFT_best.pth"),
        PATIENCE_ES, scheduler, batch_size=BATCH_SIZE )

    results_finetune[model_name] = history_ft
    metrics_nonortho_post[model_name] = test_model(
        model, test_loader_nonortho, batch_size=BATCH_SIZE
        criterion=criterion,ndevice=DEVICE)
```

Capitolo 4

Risultati

In questo capitolo vengono presentate prima le metriche di valutazione usate come riferimento per gli esperimenti, per poi discutere i risultati ottenuti dagli esperimenti descritti nei capitoli precedenti.

4.1 Metriche di valutazione

La valutazione dei modelli sviluppati per la segmentazione semantica richiede metriche in grado di quantificare la qualità delle predizioni rispetto alle maschere ground truth. Nel contesto della crack detection, dove la classe di interesse (*foreground*) rappresenta le crepe e il *background* è costituito dal resto dell'immagine, vengono comunemente utilizzate le metriche seguenti.

4.1.1 Intersection over Union

L'Intersection over Union (IoU), o *Jaccard Index*, misura il rapporto tra l'intersezione e l'unione delle aree predette e delle aree di riferimento:

$$\text{IoU} = \frac{TP}{TP + FP + FN},$$

dove TP indica i pixel correttamente classificati come crepa, FP i falsi positivi e FN i falsi negativi. Valori più alti indicano una migliore sovrapposizione tra predizione e ground truth.

4.1.2 Coefficiente di Dice

Il coefficiente di Dice, noto anche come *Sørensen–Dice index* [35], è una metrica di similarità particolarmente usata nella segmentazione medica e per problemi con classi sbilanciate. È definito come:

$$\text{Dice} = \frac{2TP}{2TP + FP + FN}.$$

Rispetto all'IoU, la formula enfatizza maggiormente la corretta individuazione della classe di interesse (foreground).

4.1.3 Precisione e Recall

La **precisione** misura la proporzione di predizioni corrette sulla classe positiva:

$$\text{Precision} = \frac{TP}{TP + FP}.$$

Un valore alto indica che il modello produce pochi falsi positivi.

Il **Recall** (o *sensitivity*) misura invece la capacità del modello di identificare tutti i pixel appartenenti alla classe positiva:

$$\text{Recall} = \frac{TP}{TP + FN}.$$

Un valore alto indica che il modello riesce a catturare la maggior parte delle crepe, anche se al costo di introdurre qualche falso positivo.

4.1.4 F1-Score

F1 è la media armonica tra precisione e recall:

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.$$

Questa metrica bilancia entrambe le dimensioni; risulta cruciale nel momento in cui le classi sono molto sbilanciate.

Poiché nel problema affrontato si considerano due classi (foreground e background), le metriche vengono calcolate separatamente per ciascuna di esse. Successivamente, si riporta la *media aritmetica* tra le classi (mean), ottenendo una valutazione complessiva più bilanciata.

4.2 Esperimento 1: Addestramento su NonOrtho

L'obiettivo principale di questo primo esperimento è stato l'addestramento dei modelli sul dataset *NonOrtho*, al fine di analizzarne le prestazioni complessive. Parallelamente, è stata condotta un'analisi comparativa tra diverse funzioni di perdita, con lo scopo di individuare quella in grado di garantire le migliori metriche di valutazione: durante il training sono state testate differenti loss function confrontandone le prestazioni su ciascun modello descritto in 2.4. I risultati ottenuti vengono riportati in forma tabellare e grafica, così da agevolare il confronto tra le varie configurazioni. Le tabelle riportano, per ogni modello, la **loss di validazione (Loss)**, la **Mean Intersection over Union (MIoU)**, la **Mean Dice (MDice)** e la **Mean F1-Score (MF1)** calcolate in validazione, insieme ai valori di MIoU, MDice, MF1 ottenuti in fase di test (**TMIoU**, **TMDice**, **TMF1**).

4.2.1 DeepCrack

Tabella 4.1: Risultati per DeepCrack sul dataset NonOrtho

Loss	**MIoU**	**MDice**	**MF1**	**TMIoU**	**TMDice**	**TMF1**
BCE	0.731	0.808	0.808	0.729	0.805	0.805
Dice	0.837	0.837	0.837	0.767	0.845	0.845
BCE+Dice	0.815	0.815	0.815	0.742	0.824	0.824
Focal	0.835	0.835	0.835	0.769	0.843	0.843
Tversky	**0.849**	**0.849**	**0.849**	**0.785**	**0.859**	**0.859**
FTL	0.840	0.840	0.840	0.786	0.859	0.859

I risultati ottenuti con DeepCrack evidenziano un andamento coerente con la letteratura: le funzioni di perdita basate su Tversky e sulle sue varianti risultano le più efficaci, mostrando una migliore capacità di distinguere le crepe dal background nonostante l'elevato squilibrio del dataset. La **Tversky Loss** si conferma la più performante in termini di *MIoU* e *MDice* sia in validazione che in fase di test, mentre la *Focal+Tversky* mantiene valori simili, segno di una maggiore robustezza nei casi di classi fortemente sbilanciate.

4.2.2 MurCrackNet

Tabella 4.2: Risultati per MurCrackNet sul dataset NonOrtho

Loss	MIoU	MDice	MF1	TMIoU	TMDice	TMF1
BCE	0.702	0.771	0.771	0.713	0.772	0.772
Dice	0.772	0.843	0.843	0.781	0.845	0.845
BCE+Dice	0.745	0.823	0.823	0.756	0.830	0.830
Focal	0.753	0.824	0.824	0.789	0.853	0.853
Tversky	**0.775**	**0.847**	**0.847**	**0.787**	**0.853**	**0.853**
FTL	0.762	0.832	0.832	0.780	0.847	0.847

Anche in questo caso la Tversky Loss si conferma tra le migliori, ma è interessante notare che la Focal Loss ha portato a un incremento consistente di Dice e F1, segno di una maggiore sensibilità nel rilevare crepe sottili.

4.2.3 U-Net

Tabella 4.3: Risultati per U-Net sul dataset NonOrtho

Loss	MIoU	MDice	MF1	TMIoU	TMDice	TMF1
BCE	0.759	0.822	0.822	0.780	0.842	0.842
Dice	0.759	0.827	0.827	0.788	0.854	0.854
BCE+Dice	0.739	0.817	0.817	0.760	0.835	0.835
Focal	0.750	0.816	0.816	0.791	**0.856**	**0.856**
Tversky	**0.773**	**0.840**	**0.840**	**0.782**	0.849	0.849
FTL	0.767	0.835	0.835	0.786	0.851	0.851

Per U-Net, la Tversky Loss emerge come la funzione più bilanciata, mentre la Focal Loss porta al miglior Dice sul test. Entrambe si dimostrano particolarmente efficaci nella segmentazione di crepe sottili.

4.2.4 ResNet

Tabella 4.4: Risultati per ResNet (FCN-ResNet50) sul dataset NonOrtho

Loss	**MIoU**	**MDice**	**MF1**	**TMIoU**	**TMDice**	**TMF1**
BCE	0.714	0.790	0.790	0.735	0.803	0.803
Dice	0.735	0.809	0.809	0.752	0.825	0.825
BCE+Dice	0.742	0.822	0.822	0.757	0.832	0.832
Focal	0.757	0.833	0.833	0.749	0.820	0.820
Tversky	**0.766**	**0.844**	**0.844**	0.769	0.842	0.842
FTL	0.762	0.838	0.838	**0.780**	**0.851**	**0.851**

Per ResNet, la Tversky Loss ottiene le metriche medie più alte, mentre la combinazione FTL fornisce il miglior risultato sul test in termini di F1 e Dice. Questo indica che, pur essendo un modello meno complesso rispetto a U-Net o MurCrackNet, riesce comunque a generalizzare bene con funzioni di perdita focalizzate.

4.2.5 DeepCrackAT

Tabella 4.5: Risultati per DeepCrackAT sul dataset NonOrtho

Loss	**MIoU**	**MDice**	**MF1**	**TMIoU**	**TMDice**	**TMF1**
BCE	0.844	0.844	0.844	0.777	0.851	0.851
Dice	0.853	0.853	0.853	0.792	0.863	0.863
BCE+Dice	0.844	0.844	0.844	0.784	0.858	0.858
Focal	0.843	0.843	0.843	0.791	0.865	0.865
Tversky	**0.854**	**0.854**	**0.854**	**0.805**	**0.874**	**0.874**
FTL	0.851	0.851	0.851	0.803	0.872	0.872

I risultati di DeepCrackAT mostrano le migliori prestazioni complessive tra i modelli analizzati. La Tversky Loss risulta la più efficace, raggiungendo valori massimi di TMIoU e TMDice pari a, rispettivamente, 0.805 e 0.874, segno di un'elevata precisione nella segmentazione. Anche le funzioni Dice e Focal Loss offrono risultati competitivi, evidenziando una buona capacità nel rilevare crepe sottili. Nel complesso, il modello mostra una notevole stabilità tra validazione e test, confermando la validità dell'approccio multi-scala e dei meccanismi di attenzione integrati.

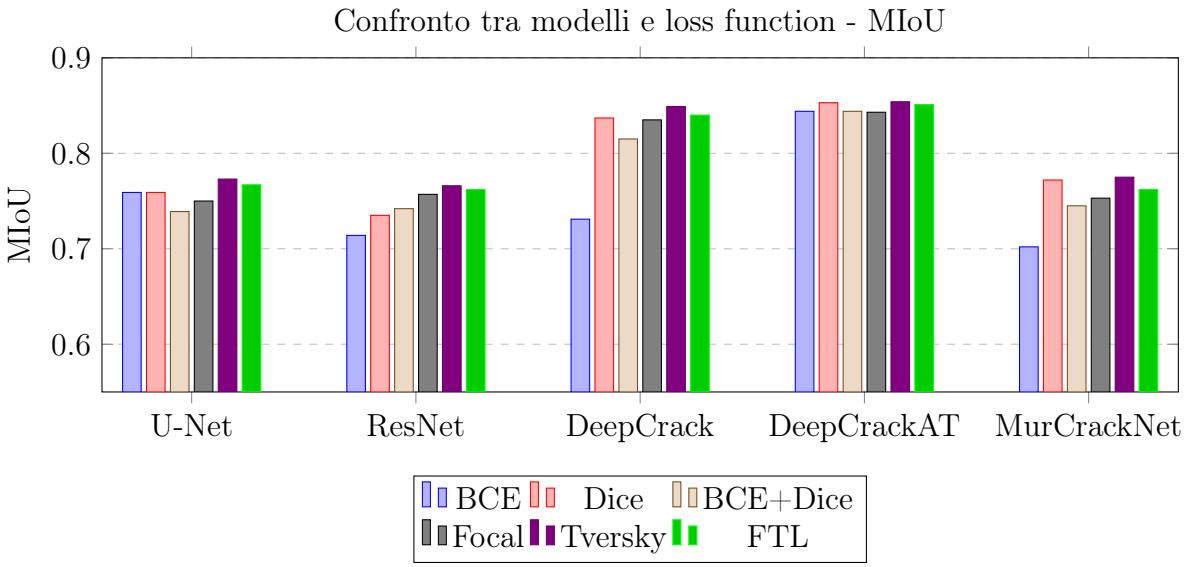

Figura 4.1: Confronto tra modelli e loss function sulla metrica MIoU.

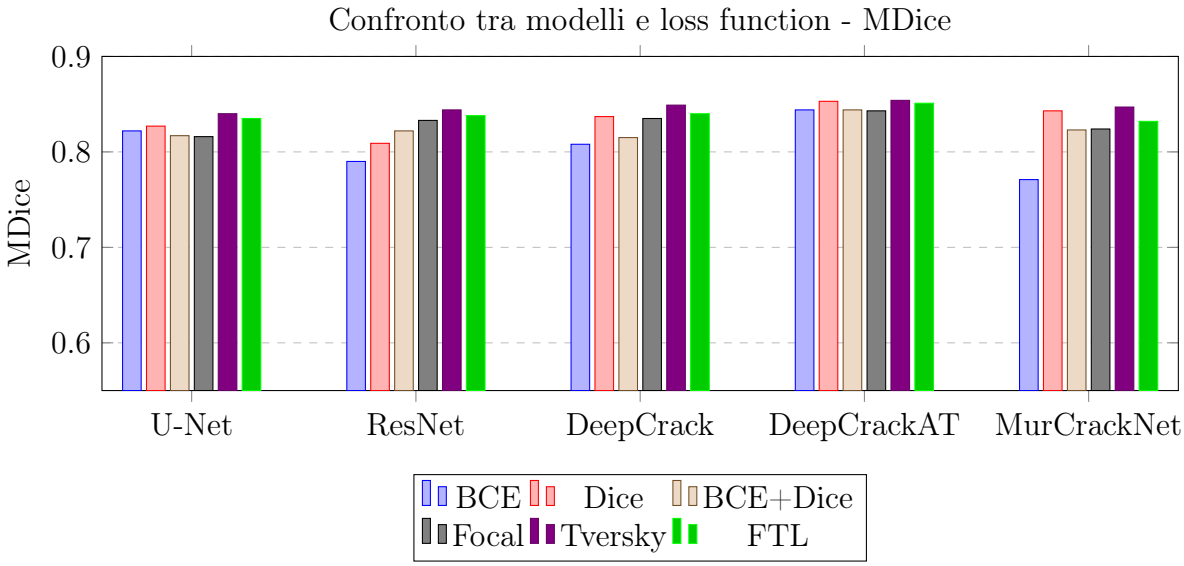

Figura 4.2: Confronto tra modelli e loss function sulla metrica MDice.

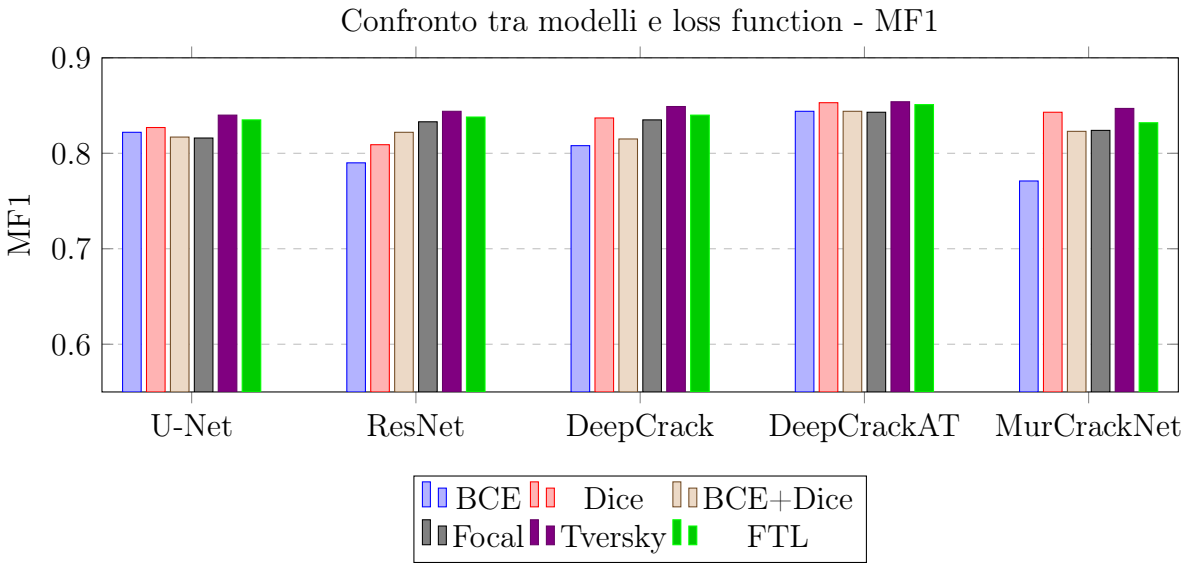

Figura 4.3: Confronto tra modelli e loss function sulla metrica MF1.

L'analisi comparativa dei risultati riportati nei grafici 4.1, 4.2 e 4.3 mostra chiaramente che la **Tversky Loss** emerge come la funzione di perdita più performante nella maggior parte dei casi, superando le alternative su tutte le metriche chiave (*MIoU*, *MDice* e *MF1*). Questo comportamento conferma quanto già osservato in letteratura, ovvero la capacità della Tversky Loss di gestire efficacemente dataset con forti squilibri di classe, come nel caso della crack detection, dove le aree di interesse (il foreground, ossia la crepa) occupano una porzione molto ridotta rispetto al background.

Alla luce di tali risultati, per il **secondo esperimento** è stata selezionata proprio la **Tversky Loss** come loss function principale per tutti i modelli.

4.3 Esperimento 2: Transfer learning e fine-tuning

Dopo aver osservato i risultati dell'esperimento 1 (4.2), notiamo che l'impiego della loss function basata sull'indice di Tversky garantisce risultati nettamente migliori rispetto ad altre funzioni; dunque, verrà usata per condurre l'esperimento 2.

In questo secondo esperimento verranno usate tecniche di transfer learning e fine-tuning; inizialmente verranno addestrati tutti i modelli (salvandone i pesi) sul dataset parziale di Dais et al. per poi testarli sul dataset NonOrtho salvando i risultati (visualizzabili per ciascuna Tabella 4.6 nella riga "Pre" per ciascun modello).
Successivamente, vengono ricaricati i pesi dei modelli addestrati nella fase precedente e si procede facendo fine-tuning sul dataset NonOrtho per poi ripetere il test su questo stesso dataset archiviandone i risultati (anche questi mostrati nella Tabella 4.6, ma nella riga "Post" per ciascun modello).

È importante precisare che la loss function utilizzata in fase di pre fine-tuning è stata selezionata in base ai risultati ottenuti sul dataset NonOrtho, che rappresenta il contesto reale di applicazione del modello. Addestrare i modelli su un dataset pubblico, come quello di Dais, nella prima fase avrebbe reso le metriche poco confrontabili con il fine-tuning successivo. In questo modo, le performance iniziali riflettono direttamente l'efficacia dei modelli sul dataset raccolto, mentre l'Esperimento 2 evidenzia come il transfer learning da dataset pre-addestrati possa migliorare le prestazioni, mantenendo il nostro dataset come riferimento per la valutazione finale.

I valori che seguono mostreranno la Mean Intersection Over UNion (MIoU), Mean Dice (MDice), Mean F1-Score (MF1), Mean Precision (MPrec) e Mean Recall (MRecall).

Tabella 4.6: Metriche pre e post fine-tuning su NonOrtho

Modello	**Fase**	**MIoU**	**MDice**	**MPrec**	**MRecall**	**MF1**
DeepCrack	Pre	0.586	0.670	0.643	0.781	0.670
	Post	**0.739**	**0.820**	**0.819**	**0.851**	**0.820**
DeepCrackAT	Pre	0.744	0.823	0.840	0.830	0.823
	Post	**0.800**	**0.870**	**0.875**	**0.882**	**0.870**
MurCrackNet	Pre	0.736	0.806	0.827	0.829	0.806
	Post	**0.812**	**0.874**	**0.882**	**0.886**	**0.874**
U-Net	Pre	0.723	0.792	0.831	0.802	0.792
	Post	**0.802**	**0.868**	**0.883**	**0.876**	**0.868**
ResNet	Pre	0.703	0.771	0.811	0.772	0.771
	Post	**0.777**	**0.847**	**0.839**	**0.871**	**0.847**

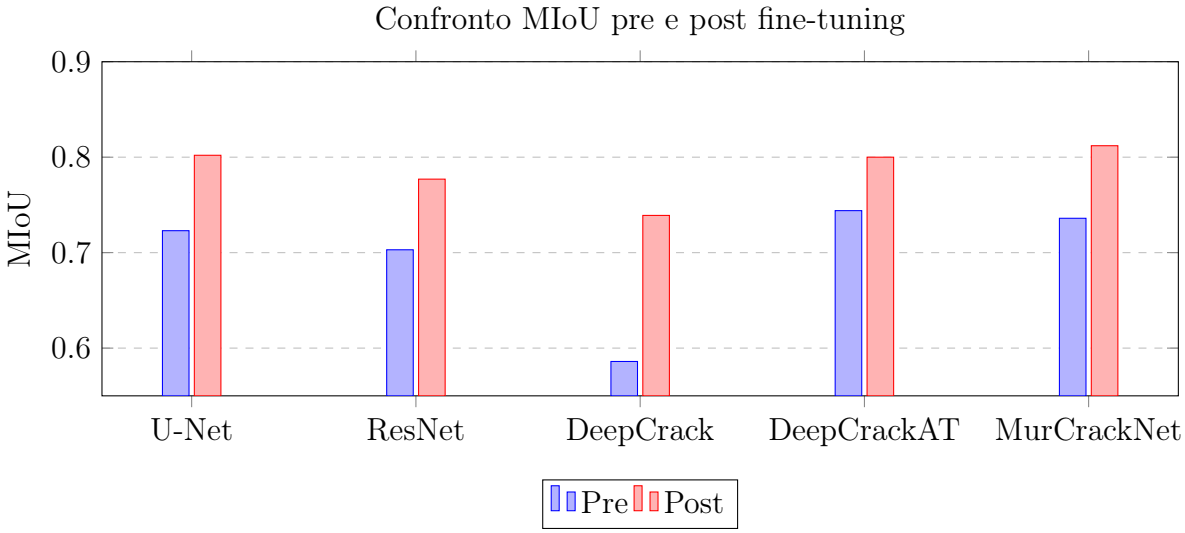

Figura 4.4: Confronto MIoU tra fase pre e post fine-tuning per ciascun modello.

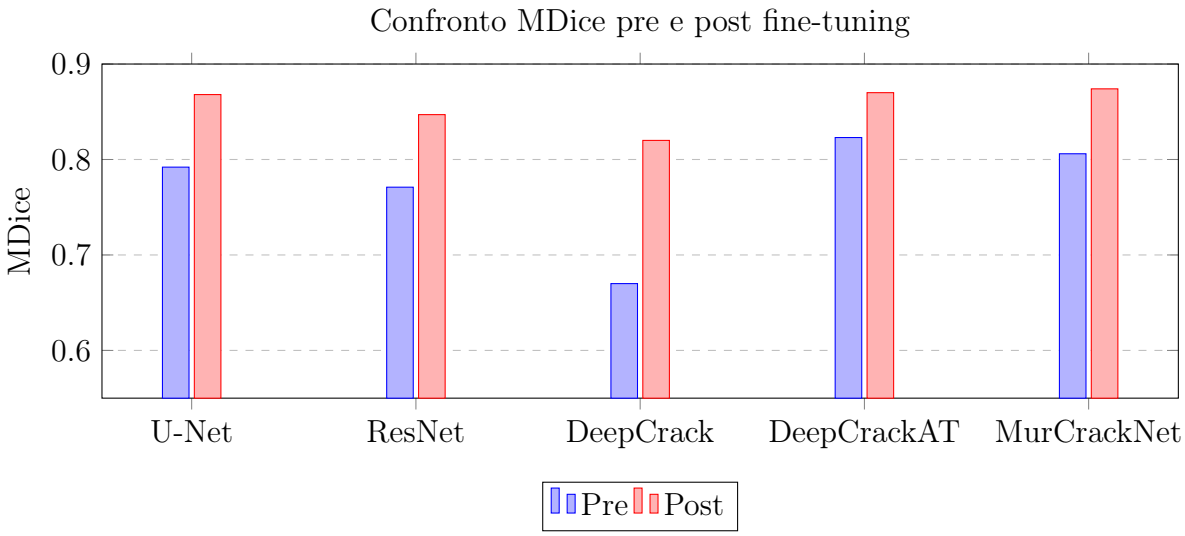

Figura 4.5: Confronto MDice tra fase pre e post fine-tuning per ciascun modello.

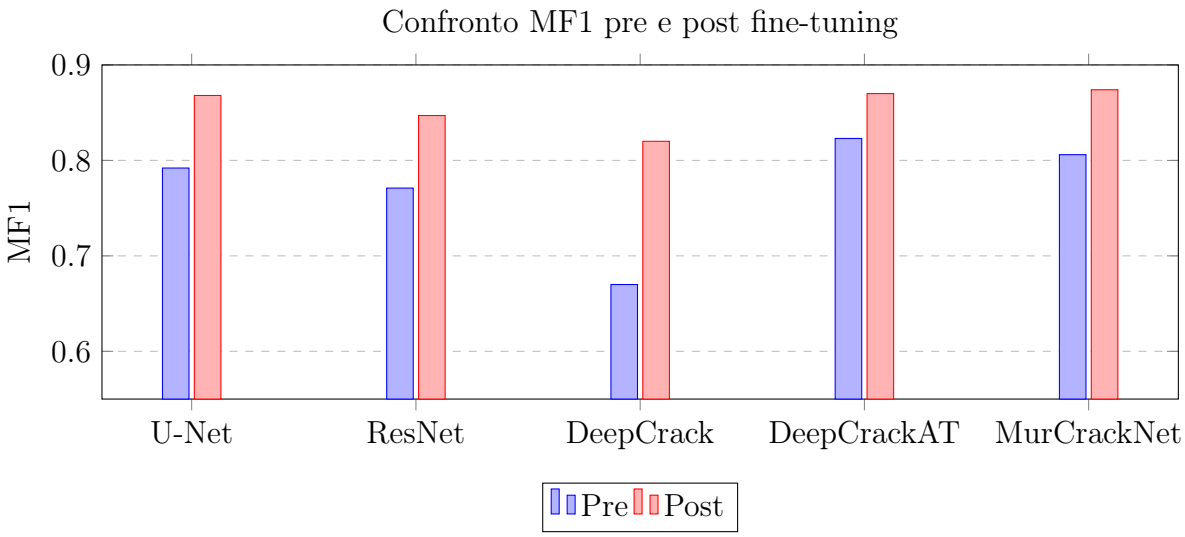

Figura 4.6: Confronto MF1 tra fase pre e post fine-tuning per ciascun modello.

Nella Tabella 4.7 verranno calcolati i miglioramenti (Δ) ottenuti da ogni modello in ciascuna metrica di valutazione facendo la differenza dei valori delle righe della fase di post fine-tuning (Post) con quell di pre fine-tuning (Pre)

Tabella 4.7: Miglioramenti (Δ) post fine-tuning rispetto al pre

Modello	**Δ MIoU**	**Δ MDice**	**Δ MPrec**	**Δ MRecall**	**Δ MF1**
DeepCrack	+0.153	+0.150	+0.176	+0.070	+0.150
DeepCrackAT	+0.056	+0.047	+0.035	+0.052	+0.047
MurCrackNet	+0.076	+0.068	+0.055	+0.057	+0.068
U-Net	+0.079	+0.076	+0.052	+0.074	+0.076
ResNet	+0.074	+0.076	+0.028	+0.099	+0.076

Dall'analisi dei risultati riportati in Tabella 4.6 e dai grafici 4.4, 4.5 e 4.6, emerge chiaramente che tutti i modelli hanno tratto beneficio dal fine-tuning sul dataset di Dais et al. In particolare, **DeepCrack** mostra un incremento marcato in MIoU e MDice, passando rispettivamente da 0.586 a 0.739 e da 0.670 a 0.820, evidenziando un miglioramento di oltre 15 punti percentuali. **DeepCrackAT** ottiene un aumento più contenuto ma costante in tutte le metriche principali, con MIoU, MDice e MF1 che migliorano di circa 5 punti percentuali ciascuna. Il modello **ResNet** mostra una capacità di adattamento significativa, con un incremento particolarmente rilevante in Recall, che passa da 0.772 a 0.871 (+10%), evidenziando una maggiore sensibilità nel rilevare le crepe. Infine, **MurCrackNet** e **U-Net** migliorano le proprie prestazioni in maniera equilibrata, con incrementi medi di circa 7–8 punti percentuali nelle metriche principali.
Nella tabella che segue sono presentate alcune predizioni dei modelli, per consentire anche un'analisi visiva dei risultati ottenuti. Rispettivamente, per colonna, abbiamo l'immagine 224×224 (Img), il Ground Trouth creato pixel-wise con GIMP (GT) e le predizioni dei modelli ResNet, U-Net, MurCrackNet (MCNet), DeepCrack (DC) e DeepCrackAT (DCAT)

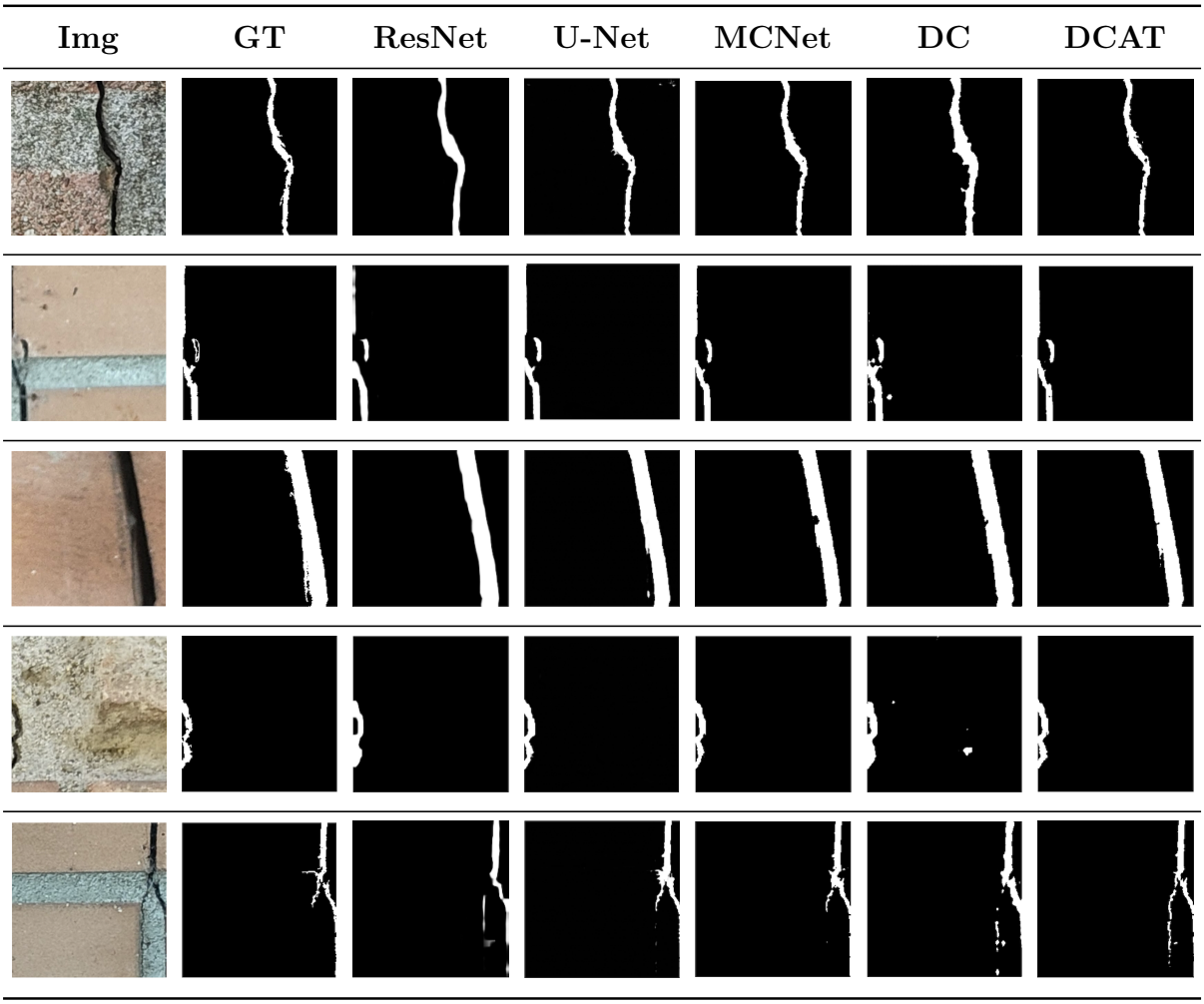

Figura 4.7: Confronto visivo delle predizioni dei modelli sul dataset *NonOrtho*. Ogni riga mostra la stessa immagine di input, la ground truth e le segmentazioni prodotte dai diversi modelli.

Dall'analisi qualitativa delle predizioni (Figura 4.7) e dai risultati riportati nelle metriche di valutazione (Tabella Tabella 4.6), emergono differenze significative tra i vari modelli testati. I modelli **MurCrackNet**, **U-Net** e **DeepCrackAT** si dimostrano i più fedeli nella segmentazione, producendo mappe coerenti con la ground truth e riuscendo a mantenere un buon equilibrio tra rilevamento delle crepe sottili e assenza di falsi positivi.

In particolare, **MurCrackNet** risulta il più stabile e accurato, come confermato dalle metriche più elevate in MF1 e MDice, mentre **DeepCrackAT** mostra prestazioni molto vicine ma leggermente inferiori, compensando con una maggiore regolarità nei bordi segmentati. U-Net, pur non raggiungendo le stesse metriche dei precedenti, evidenzia una notevole capacità di eliminare il rumore di fondo e di isolare le crepe reali, risultando particolarmente efficace in presenza di texture complesse.

Al contrario, **DeepCrack** tende a sovrastimare le aree di frattura, rilevando crepe anche dove non presenti o ampliando eccessivamente quelle reali; ciò si riflette nelle metriche inferiori di precisione e IoU. Invece **ResNet** mostra risultati meno consistenti: riesce talvolta a identificare correttamente le crepe, ma spesso produce segmentazioni frammentate o irregolari, perdendo la continuità della forma reale. Nel complesso, i risultati quantitativi (Tabella 4.6) e qualitativi (Figura 4.7) concordano nel confermare che i modelli **MurCrackNet**, **U-Net** e **DeepCrackAT** rappresentano le soluzioni più affidabili per la segmentazione di crepe in murature del dataset *NonOrtho*.

Capitolo 5

Conclusioni

Lo scopo di questo lavoro era verificare l'efficacia di approcci basati su DL per la segmentazione automatica delle crepe su superfici in murature, comparando varie architetture e strategie di training. Inoltre, il lavoro si è posto anche l'obiettivo di fornire un dataset originale ed eterogeneo per colmare anche se in parte la scarsa disponibilità di dataset in muratura lamentata in letteratura [1, 12, 15].

Il primo obiettivo è stato raggiunto, come dimostrato dai risultati ottenuti dal piano sperimentale:

1. Durante l'addestramento dei modelli sul dataset Nonortho, sono state rilevate metriche abbastanza buone, raggiungendo valori di TMF1 e TMDice pari a 0.874 col modello DeepCrackAT; inoltre queste stesse metriche sono ottenute usando la Tversky loss, confermando l'importanza della scelta della funzione di perdita in contesti con forte sbilanciamento fra classi: come mostrato dai grafici comparativi (Figura 4.1, Figura 4.2) notiamo che con le altre funzioni di perdita, addestrando lo stesso modello, sono stati osservati risultati diversi; ad esempio, con BCE e BCE+Dice abbiamo ottenuto sulle stesse metriche 0.851 0.851 per la prima loss function e 0.858 0.858 per la seconda.

2. L'uso di transfer learning e fine-tuning (utilizzando la funzione Tversky, scelta come più promettente) ha apportato notevoli miglioramenti in quasi tutti i modelli (vedasi i valori "Pre" a "Post" della Tabella 4.6), consentendo ad alcuni modelli di guadagnare anche 10/15 punti percentuali su metriche come MF1, MDice e MIoU; si nota anche che nei valori di post fine tuning abbiamo avuto un miglioramento sostanziale anche rispetto alle metriche ottenute nel primo esperimento addestrando solo sul dataset NonOrtho. Infatti, abbiamo raggiungo con MurCrackNet per MF1 0.874, nettamente migliore rispetto al risultato sia dello stesso modello che di DeepCrackAT nello scorso esperimento.

Anche per quanto riguarda la creazione del dataset ci si può ritenere soddisfatti: la data collection effettuata sul territorio dell'Emilia-Romagna ha fornito il dataset denominato **NonOrtho**, il quale è composto da immagini reali e originali di strutture di diverso tipo di muratura.

Nel complesso, l'impiego di funzioni di perdita mirate a compensare lo sbilanciamento tra classi, tipico dei problemi di crack detection, unito all'uso di strategie di transfer learning e fine-tuning sul dataset, si è dimostrato efficace per ottenere modelli in grado di generalizzare in modo più robusto e accurato.

Il codice degli esperimenti, dei modelli e di tutto quello che è stato descritto nei capitoli precedenti, è reperibile nella repository del mio account GitHub: `https://github.com/aNdReA9111/deep-crack-detection`.

Prospettive future

In ottica di prospettive future, un ampliamento delle dimensioni e dell'eterogeneità del dataset rappresenta un passo fondamentale per sviluppare modelli più robusti e generalizzabili:

- Altri studi, per costruire un loro dataset, hanno anche usato immagini provenienti da internet e campioni che includevano elementi disturbatori quali finestre, tubi, porte al fine di avere un dataset più robusto ed eterogeneo che consenta ai modelli di migliorare le loro capacità di generalizzazione [1];
- come anticipato nella sezione 2.1.1, si vorrebbe sviluppare un dataset Ortogonale, aumentando così la qualità dello stesso;
- arricchire il nostro set di dati con superfici eterogenee, come cemento o manto stradale;
- è stato dimostrato che l'uso di Generative Adversarial Network (GAN)[1] permettano di creare immagini sintetiche realistiche di crepe partendo da pattern strutturali appresi dal dataset reale, arricchendo il training e migliorando sensibilità e robustezza del modello [39].

[1] classe di modelli di apprendimento automatico composti da due reti neurali — un generatore e un discriminatore — che competono tra loro in un processo minmax per migliorare progressivamente la qualità dei dati generati [38].

Bibliografia

[1] Dimitris Dais et al. "Automatic crack classification and segmentation on masonry surfaces using convolutional neural networks and transfer learning". In: *Automation in Construction* 125 (2021), p. 103606. DOI: 10.1016/j.autcon.2021.103606. URL: https://linkinghub.elsevier.com/retrieve/pii/S0926580521000571.

[2] C. R. Farrar, N. Dervilis e K. Worden. "The Past, Present and Future of Structural Health Monitoring: An Overview of Three Ages". In: *Strain* 61.1 (2025). e12495 5547601, e12495. DOI: https://doi.org/10.1111/str.12495. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/str.12495. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/str.12495.

[3] Charles Farrar e Keith Worden. *Structural Health Monitoring A Machine Learning Perspective*. Gen. 2013. ISBN: 978-1-119-99433-6. DOI: 10.1002/9781118443118.

[4] Xu Li et al. "Micromechanisms of a macrocrack propagation behavior affected by short to long fatigue microcracks". In: *Mechanics of Advanced Materials and Structures* 29.19 (2022), pp. 2726–2739. DOI: 10.1080/15376494.2021.1876282. eprint: https://doi.org/10.1080/15376494.2021.1876282. URL: https://doi.org/10.1080/15376494.2021.1876282.

[5] Guo Hu et al. "Pavement Crack Detection Method Based on Deep Learning Models". In: *Wireless Communications and Mobile Computing* 2021 (mag. 2021), pp. 1–13. DOI: 10.1155/2021/5573590.

[6] Ruoxian Li et al. "Automatic bridge crack detection using Unmanned aerial vehicle and Faster R-CNN". In: *Construction and Building Materials* 362 (2023), p. 129659. ISSN: 0950-0618. DOI: https://doi.org/10.1016/j.conbuildmat.2022.129659. URL: https://www.sciencedirect.com/science/article/pii/S0950061822033153.

[7] G. Świt, A. Krampikowska e P. Tworzewski. "Non-Destructive Testing Methods for In Situ Crack Measurements and Morphology Analysis with a Focus on a Novel Approach to the Use of the Acoustic Emission Method". In: *Materials* 16.23 (2023), p. 7440. DOI: 10.3390/ma16237440. URL: https://doi.org/10.3390/ma16237440.

[8] Gokhan Kilic. "Assessment of historic buildings after an earthquake using various advanced techniques". In: *Structures* 50 (2023), pp. 538–560. ISSN: 2352-0124. DOI: https://doi.org/10.1016/j.istruc.2023.02.033. URL: https://www.sciencedirect.com/science/article/pii/S2352012423001911.

[9] Debra F. Laefer, Jane Gannon e Elaine Deely. "Reliability of Crack Detection Methods for Baseline Condition Assessments". In: *Journal of Infrastructure Systems* 16.2 (2010), pp. 129–137. DOI: 10.1061/(ASCE)1076-0342(2010)16(129).

[10] Z. Xu et al. "Crack Detection of Bridge Concrete Components Based on Large-Scene Images Using an Unmanned Aerial Vehicle". In: *Sensors* 23.14 (2023), p. 6271. DOI: 10.3390/s23146271. URL: https://doi.org/10.3390/s23146271.

[11] Nicola Gehri, Jaime Mata-Falcón e Walter Kaufmann. "Automated crack detection and measurement based on digital image correlation". In: *Construction and Building Materials* 256 (2020), p. 119383. ISSN: 0950-0618. DOI: https://doi.org/10.1016/j.conbuildmat.2020.119383. URL: https://www.sciencedirect.com/science/article/pii/S095006182031388X.

[12] Zehao Ye et al. "Sam-based instance segmentation models for the automation of structural damage detection". In: *Advanced Engineering Informatics* 62 Part C (set. 2024). DOI: 10.1016/j.aei.2024.102826.

[13] Qin Zou et al. "Deepcrack: Learning Hierarchical Convolutional Features for Crack Detection". In: *IEEE Transactions on Image Processing* 28.3 (2019), pp. 1498–1512.

[14] Qinghua Lin et al. "DeepCrackAT: An effective crack segmentation framework based on learning multi-scale crack features". In: *Engineering Applications of Artificial Intelligence* 126 (2023), p. 106876. ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2023.106876. URL: https://www.sciencedirect.com/science/article/pii/S0952197623010606.

[15] Mazleenda Mazni et al. "Crack Recognition in Masonry Structures: CNN Models with Limited Data Sets". In: *ELEKTRIKA- Journal of Electrical Engineering* 23.1 (2024), 133–140. DOI: 10.11113/elektrika.v23n1.528. URL: https://elektrika.utm.my/index.php/ELEKTRIKA_Journal/article/view/528.

[16] Arman Malekloo et al. "Machine learning and structural health monitoring overview with emerging technology and high-dimensional data source highlights". In: *Structural Health Monitoring* 21.4 (2022), pp. 1906–1955. DOI: 10.1177/14759217211036880. eprint: https://doi.org/10.1177/14759217211036880. URL: https://doi.org/10.1177/14759217211036880.

[17] Osama Abdeljaber et al. "Real-Time Vibration-Based Structural Damage Detection Using One-Dimensional Convolutional Neural Networks". In: *Journal of Sound and Vibration* 388 (feb. 2017), pp. 154–170. DOI: 10.1016/j.jsv.2016.10.043.

[18] Huajun Liu et al. "CrackFormer: Transformer Network for Fine-Grained Crack Detection". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2021, pp. 3783–3792.

[19] Wei Zhang et al. "Concrete crack segmentation using improved U-Net architecture". In: *Automation in Construction* (2022).

[20] Jian Li et al. "Attention-based convolutional neural networks for crack detection". In: *Construction and Building Materials* (2023).

[21] Ming Chen et al. "Hybrid CNN-Transformer architecture for crack segmentation". In: *Computer-Aided Civil and Infrastructure Engineering* (2024).

[22] Huaqi Tao et al. "A Convolutional-Transformer Network for Crack Segmentation with Boundary Awareness". In: *2023 IEEE International Conference on Image Processing (ICIP)*. IEEE. 2023, pp. 86–90.

[23] Jeremy CH Ong et al. "Feature pyramid network with self-guided attention refinement module for crack segmentation". In: *Structural Health Monitoring* 22.1 (2023), pp. 672–688. DOI: 10.1177/14759217221089571. eprint: https://doi.org/10.1177/14759217221089571. URL: https://doi.org/10.1177/14759217221089571.

[24] Stamos Katsigiannis et al. "Deep learning for crack detection on masonry façades using limited data and transfer learning". In: *Journal of Building Engineering* 76 (2023), p. 107105. ISSN: 2352-7102. DOI: https://doi.org/10.1016/j.jobe.2023.107105. URL: https://www.sciencedirect.com/science/article/pii/S2352710223012846.

[25] Olaf Ronneberger, Philipp Fischer e Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV]. URL: https://arxiv.org/abs/1505.04597.

[26] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[27] Vijay Badrinarayanan, Alex Kendall e Roberto Cipolla. *SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation*. 2016. arXiv: 1511.00561 [cs.CV]. URL: https://arxiv.org/abs/1511.00561.

[28] Fisher Yu e Vladlen Koltun. *Multi-Scale Context Aggregation by Dilated Convolutions*. 2016. arXiv: 1511.07122 [cs.CV]. URL: https://arxiv.org/abs/1511.07122.

[29] Sanghyun Woo et al. *CBAM: Convolutional Block Attention Module*. 2018. arXiv: 1807.06521 [cs.CV]. URL: https://arxiv.org/abs/1807.06521.

[30] Ilya Tolstikhin et al. *MLP-Mixer: An all-MLP Architecture for Vision*. 2021. arXiv: 2105.01601 [cs.CV]. URL: https://arxiv.org/abs/2105.01601.

[31] Nabila Abraham e Naimul Mefraz Khan. *A Novel Focal Tversky loss function with improved Attention U-Net for lesion segmentation*. 2018. arXiv: 1810.07842 [cs.CV]. URL: https://arxiv.org/abs/1810.07842.

[32] Isidore Jacob Good. *Probability and the Weighing of Evidence*. Originally published by Charles Griffin and Company Limited, London, 1950. London: C. Griffin, 1950, pp. viii, 119. ISBN: 978-0852640587.

[33] David E Rumelhart, Geoffrey E Hinton e Ronald J Williams. "Learning representations by back-propagating errors." In: *Nature* 323.6088 (1986), pp. 533–536.

[34] Lee R. Dice. "Measures of the Amount of Ecologic Association Between Species". In: *Ecology* 26.3 (1945), pp. 297–302. ISSN: 00129658, 19399170. URL: http://www.jstor.org/stable/1932409.

[35] Fausto Milletari, Nassir Navab e Seyed-Ahmad Ahmadi. *V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation*. 2016. arXiv: 1606.04797 [cs.CV]. URL: https://arxiv.org/abs/1606.04797.

[36] Tsung-Yi Lin et al. "Focal Loss for Dense Object Detection". In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 2017.

[37] Seyed Sadegh Mohseni Salehi, Deniz Erdogmus e Ali Gholipour. *Tversky loss function for image segmentation using 3D fully convolutional deep networks*. 2017. arXiv: 1706.05721 [cs.CV]. URL: https://arxiv.org/abs/1706.05721.

[38] Ian J. Goodfellow et al. "Generative adversarial nets". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Montreal, Canada: MIT Press, 2014, 2672–2680.

[39] Alberto Botana López et al. *GAN-based data augmentation for crack detection*. Ver. final. Set. 2021. DOI: 10.5281/zenodo.7074639. URL: https://doi.org/10.5281/zenodo.7074639.