POLITECNICO DI TORINO

Corso di Laurea Magistrale in
INGEGNERIA INFORMATICA

Tesi di Laurea Magistrale

RICONOSCIMENTO AUTOMATICO DI TRACCE DI PARTICELLE NELLO SCATTERING NUCLEARE

Supervisori	Candidato
Prof. FABRIZIO LAMBERTI	CIRO MAIELLO
Prof.ssa LIA MORRA	

Anno Accademico 2021/2022

Abstract

Nell'ambito della fisica nucleare, il **PAINUC experiment** è un esperimento effettuato all'interno della *Self-shunted streamer chamber* (SSSC) del *Joint Institute for Nuclear Research* (JINR) a Dubna. L'obiettivo di tale esperimento è lo studio della fenomenologia dell'interazione nucleare alle energie di attivazione della risonanza Δ. All'interno della SSSC, per fotografare l'evento nucleare, vengono utilizzate due camere che catturano da due punti di vista diversi (vista *Left* e vista *Right*) i possibili scontri nucleari. A partire da tali fotografie, il team *DubTo-PAINUC* ha il compito di tracciare le traiettorie generate dalle particelle passanti nella SSSC, farne una ricostruzione 3D e studiarne il comportamento. Il tracciamento delle particelle presenta diverse problematiche e criticità a causa di un forte rumore di fondo dovuto alla ionizzazione dell'Elio della SSSC e al rumore gaussiano presente lungo le traiettorie. Questa tesi, quindi, ha l'obiettivo di proporre un sistema alternativo per il tracciamento delle traiettorie delle particelle nucleari catturate nella SSSC. Tale sistema è composto da tre parti fondamentali: una rete neurale convoluzionale che si occupa di segmentazione semantica, un algoritmo di divisione delle tracce e uno studio dell'intensità delle tracce trovate. In particolare, la rete si occupa di generare una maschera di segmentazione dell'intero evento; successivamente l'algoritmo, a partire da tale maschera e studiandone la geometria, attraverso trasformazioni morfologiche nell'ambito della Computer Vision, rileva le singole tracce; infine, si applica lo studio dell'intensità ai fini di poter avere un tracciamento che descriva fedelmente i bulbi di luce di cui è composta la traiettoria della particella.
Il sistema è stato validato e testato sperimentalmente: gli esperimenti dimostrano delle performance incoraggianti. Il modello arriva a rilevare correttamente il 44% degli eventi analizzati.

Indice

Capitolo 1

Introduzione

Lo studio e la ricostruzione della cinematica di una collisione tra particelle negli acceleratori è indispensabile per la ricerca dei processi che avvengono durante l'interazione. L'approfondimento degli scontri tra Pioni e nuclei di Elio permette di indagare una serie di questioni fondamentali aperte nella fisica nucleare e delle particelle: può fornire informazioni sull'evoluzione dell'Universo durante la fase di transizione dall'era adronica all'era della nucleosintesi primordiale. Durante questa transizione la forza forte (cromodinamica quantistica) è stata responsabile di un insieme non banale di fenomeni fisici che hanno comportato l'emergere di sistemi complessi collettivi auto-interagenti esistenti su scala quantistica.
L'obiettivo, quindi, del **PAINUC experiment** è quello di esaminare il comportamento e gli effetti delle collisioni appena descritte.

1.1 Il PAINUC experiment

Presso il Joint Institute for Nuclear Research (JINR) a Dubna (Russia) sono stati osservati più di 30 mila eventi di scattering pione-nucleo di elio nella **Self-Shunted Streamer Chamber** (SSSC)[1] del fasotrone del *Laboratory of nuclear problems* (LNP). Tale camera, riempita di elio (He+$CH_4 \approx 0.05\%$+$air \approx 0.05\%$+$H_2O \approx 0.1\%$), ha la capacità di rivelare la cinematica completa dell'interazione particellare e, quindi, di fotografarla (Figura 1.1).
Lo scopo del team *DubTo-PAINUC* (collaborazione tra l'LNP del JINR e il *Dipartimento di Fisica Generale "A. Avogadro"* dell'Università di Torino), in particolare, è quello di studiare la fenomenologia dell'interazione nucleare alle energie di attivazione della risonanza Δ e la determinazione delle distribuzioni di vari parametri cinematici dell'evento (come momenti delle particelle, energie, masse efficaci, ecc.) per la possibile rivelazione di processi sottili tra nucleoni nel nucleo di Elio. A tal fine, è essenziale il riconoscimento automatico e la ricostruzione 3D delle traiettorie (tracce) particellari iniziali e secondarie relative agli eventi di interazione nucleare. Un altro obiettivo è quello di fornire un potente strumento didattico per introdurre la fisica nucleare e delle particelle agli studenti e al pubblico in generale.

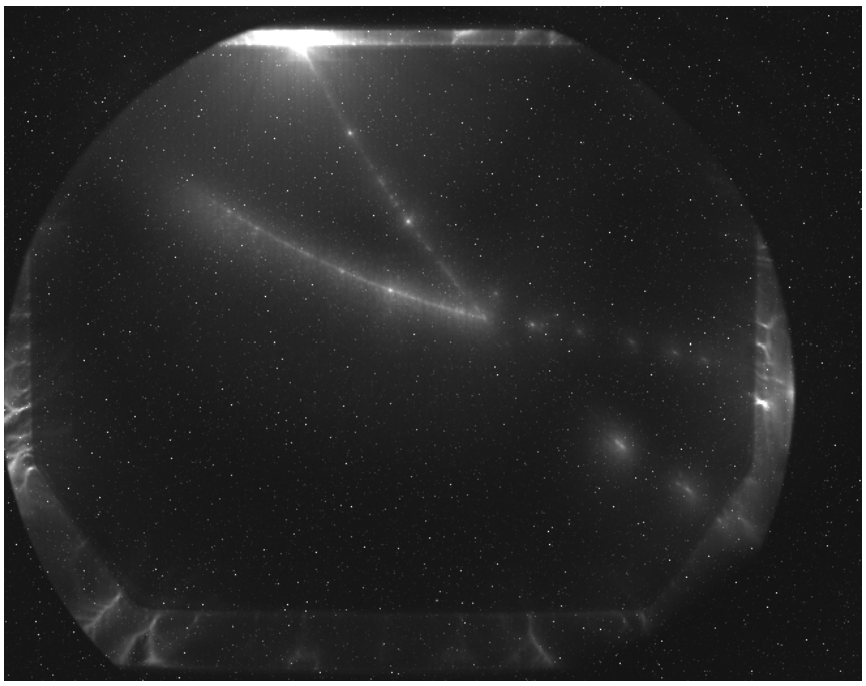


Figura 1.1. Esempio di collisione catturata nella SSSC.

1.1.1 Self-Shunted Streamer Chamber (SSSC)

La SSSC, come precedentemente accennato, è in grado di catturare la cinematica completa dell'interazione fino ai momenti dei prodotti secondari delle interazioni a bassissima energia, come, ad esempio, i protoni con un'energia di appena $1MeV$.

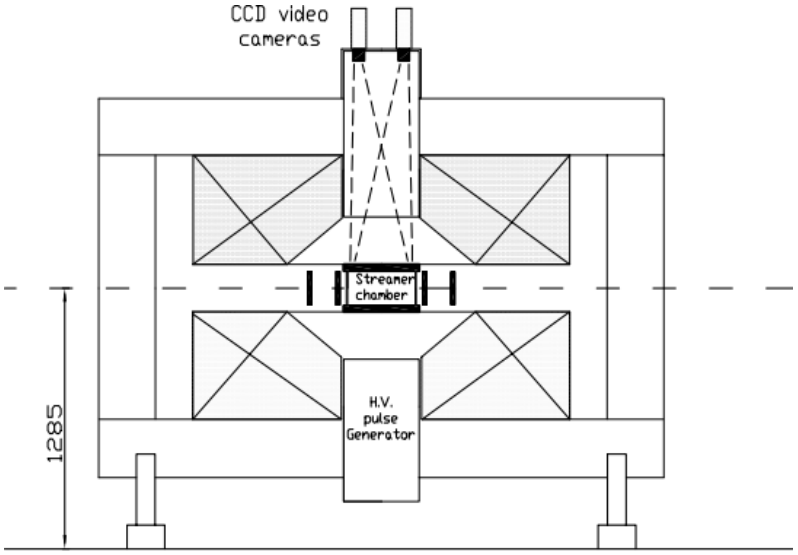
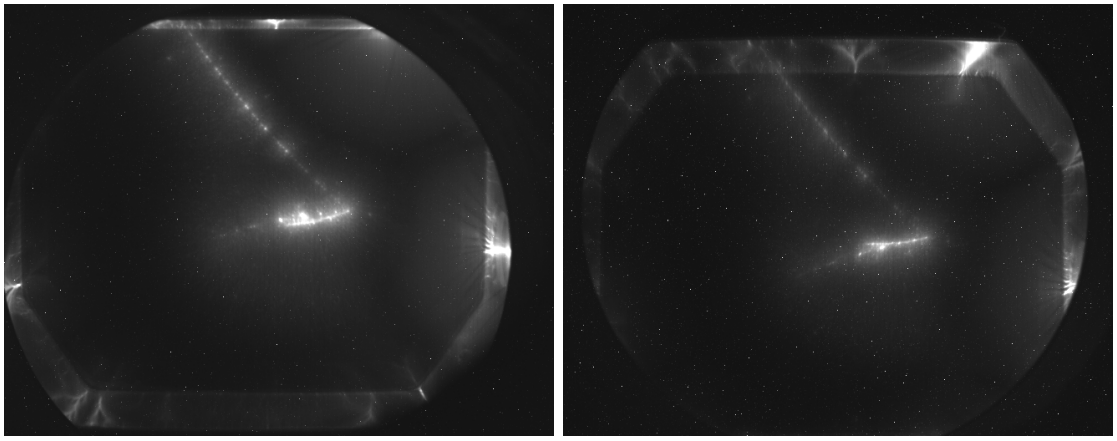


Figura 1.2. Schema della SSSC.[1]

Il fascio di pioni (π) che arriva nella camera (pressione $P = 1atm$), proveniente dal fasotrone del JINR, ha un'energia $E_\pi = 68 - 106MeV$. La camera presenta due videocamere CCD (Charge Coupled Device)[2] che permettono di fotografare la luce visibile emessa, lungo le tracce delle particelle, dagli atomi di elio eccitati. Le videocamere, come si può notare dalla Figura 1.2, sono state posizionate in modo da catturare due angolazioni diverse della camera (Figura 1.3) così da consentire la ricostruzione 3D dell'interazione dopo aver rilevato le tracce e quindi poterne valutare la profondità e i movimenti nella terza dimensione dopo lo scontro.



(a) Visione *Left* dell'interazione. **(b)** Visione *Right* dell'interazione.

Figura 1.3. Esempio di evento fotografato dalle due video camere CCD.

Fotografare le traiettorie delle particelle con dei sensori fotometrici CCD è possibile poiché se i contatori a scintillazione (C1-C7) rivelano che si è verificato un evento nucleare all'interno del volume della camera, l'**High Voltage Pulse Generator Marx-Arkadiev (HVPG)**, il cui voltaggio è $\Delta V = 250KV$, si attiva e genera un impulso ad alta tensione applicato agli elettrodi dell'SSSC; di conseguenza, la particella carica lascia lungo la sua traiettoria coppie elettrone-ione che emettono luce visibile.
Intorno alla camera vi è un elettromagnete **MC-4A**. Di conseguenza, le particelle cariche in movimento descrivono un movimento ad elica a causa della forza di Lorentz dovuta al campo magnetico con direzione verticale al piano. E' per questo motivo che dalla prospettiva delle due videocamere CCD le particelle descrivono degli archi di circonferenza sul piano. Infatti, sul piano verticale, le orbite sono lineari.
Le immagini catturate dai sensori CCD hanno una risoluzione 1317×1035 (ogni pixel è uguale a $1mm$ della camera) e una profondità di 12 bit per pixel, quindi il valore di ogni pixel può variare da 0 a 4095 (4096 livelli di grigio).

1.1.2 Reconstruction Code

Per rilevare le tracce dalle immagini ottenute nella SSSC e ricostruire in tre dimensioni il comportamento delle traiettorie particellari, il team DubTo-PAINUC è già in

possesso di un tool: il **Reconstruction Code**.
Il tool permette di poter inserire tre punti per traiettoria in modo da calcolarne l'arco di circonferenza passante. Successivamente, compie un *fitting* degli archi di circonferenza sul piano e passa per i pixel aventi un valore di intensità più alto della media dell'intensità dei pixel dell'intera immagine. Ciò avviene all'interno di un range, rispetto ai tre punti scelti, di ± 5 pixel.
Nello step seguente, avviene l'estrazione delle coordinate sull'asse z. Grazie alle due immagini dello stesso evento, si cercano dei punti corrispondenti tra loro così da generare delle coordinate che diano informazioni sulla profondità. Quindi, avviene il *fitting* delle traiettorie, ora in tre dimensioni, rispetto a delle eliche tridimensionali.
Per il rilevamento delle tracce, è stata presa in considerazione l'idea di utilizzare gli **automa cellulari**[3].
Per il rilevamento delle tracce all'interno degli eventi, le performance di tale algoritmo di ricostruzione è del 10%, quindi riesce a rilevare tutte le tracce all'interno di un evento senza alcun errore il 10% delle volte.

1.2 Definizione del problema

L'obiettivo di questa tesi è di proporre un sistema alternativo al Rceonstruction Code per il tracciamento delle traiettorie delle particelle nucleari per permettere, successivamente in un lavoro futuro, una ricostruzione 3D automatica dell'evento.
Il sistema creato per risolvere questo problema verrà descritto in modo più approfondito nel capitolo "Metodi": ho applicato un approccio ibrido costituito sia da tecniche di Machine Learning che da trasformazioni nell'ambito della Computer Vision. Tale sistema è composto da tre parti fondamentali: una rete neurale convoluzionale che si occupa di segmentazione semantica, un algoritmo di divisione delle tracce e uno studio dell'intensità delle tracce trovate. In particolare, la rete si occupa di generare una maschera di segmentazione dell'intero evento; successivamente l'algoritmo, a partire da tale maschera e studiandone la geometria, attraverso trasformazioni morfologiche nell'ambito della Computer Vision, rileva le singole tracce; infine, si applica lo studio dell'intensità ai fini di poter avere un tracciamento che descriva fedelmente i bulbi di luce di cui è composta la traiettoria della particella.

1.3 Schema della tesi

Questa sezione intende essere uno schema per i lettori.
Dopo questo capitolo introduttivo, il capitolo 2 espone alcuni dei concetti teorici, e non solo, del Machine Learning e in particolare della segmentazione e dell'architettura (U-Net) che utilizzo per questo lavoro di tesi.
Il sistema implementato verrà descritto dettagliatamente nel capitolo 3 specificando tutte le operazioni effettuate: dalla generazione delle maschere di segmentazione allo studio dell'intensità passando per l'algoritmo di divisione delle tracce e le metriche

utilizzate per la valutazione delle performance.
La creazione del dataset, il suo *splitting* e l'*augmentation* applicata verranno esposte nel capitolo 4. Qui parlerò anche degli iperparametri utilizzati per eseguire gli allenamenti della rete neurale e dei parametri dell'algoritmo.
Nel capitolo 5 vengono mostrati i risultati raggiunti da tutte e tre le parti del sistema. Vi sono sia i valori delle metriche di valutazione ottenuti durante gli esperimenti che degli output visivi. Vengono anche confrontati i risultati di esperimenti diversi e tra le tre dimensionalità del sistema.
Infine, nel capitolo 6, ci saranno le conclusioni e dei possibili sviluppi futuri per il miglioramento del sistema o per il proseguio del progetto.

Capitolo 2

Machine Learning Background

In questo capitolo esporrò la teoria, gli studi e lo stato dell'arte su cui si basa il sistema sviluppato per questo lavoro di tesi. In particolare, spiegherò che cosa si intende per *segmentazione d'immagini* nell'ambito del Machine Learning e, più in generale, della Computer Vision. Parlerò dei tipi di segmentazione e task che si possono risolvere con specifiche architetture e concluderò con lo stato dell'arte riguardante la segmentazione di traiettorie e linee e alcuni lavori correlati riguardanti la rilevazione di traiettorie particellari.

2.1 Segmentazione d'immagini

Figura 2.1. Esempio di Semantic e Instance Segmentation.[4]

In Computer Vision, la *segmentazione* consiste nella divisione di un'immagine in più *regioni* o *oggetti* rappresentati da set di pixel. Più precisamente, l'obiettivo della segmentazione consiste nell'assegnazione di un'etichetta a tutti i pixel di un'immagine in modo che i pixel con stessa etichetta abbiano delle caratteristiche in comune (intensità, colore o forma).
Esistono due modi di approcciare alla segmentazione:

- Computer Vision tradizionale.
- Machine Learning.

In questa sezione mi focalizzerò sulle tecniche basate sul Machine Learning poiché sono quelle che ho utilizzato maggiormente per questa tesi. Nonostante ciò, ho fatto uso anche di alcune tecniche appartenenti alla prima categoria come la *thresholding*[28] oppure la trasformazione *watershed*[32].
Un'altra classificazione distingue tra tre diverse tecniche di segmentazione:

- **Semantic Segmentation:** associa ad ogni pixel nell'immagine un'etichetta della classe a cui appartiene.
- **Instance Segmentation:** associa ad ogni pixel nell'immagine l'istanza dell'oggetto a cui appartiene.
- **Panoptic Segmentation:** è un approccio intermedio tra i due precedenti. Associa ad ogni pixel l'etichetta della classe come nella Semantic Segmentation, ma allo stesso tempo fa una distinzione tra le istanze di una stessa classe.

2.1.1 Semantic Segmentation

Figura 2.2. Esempio di Semantic Segmentation.

La Semantic Segmentation fa in modo che ogni pixel all'interno dell'immagine abbia un'etichetta, ma non distingue tra oggetti diversi di una stessa classe. Un'esempio

di questo comportamento è facilmente riconoscibile in Figura 2.2 dove, appunto, le mucche non vengono riconosciute come istanze, quindi oggetti, differenti della stessa classe, ma come un'unica regione di pixel che condividono delle caratteristiche.
Nell'ambito del Machine Learning, tale risultato si può raggiungere allenando una rete neurale convoluzionale (CNN) utilizzando un dataset che proponga per ogni pixel un'etichetta. In particolare, tale problema potrebbe essere affrontato con un meccanismo di tipo *sliding window*, quindi l'immagine viene divisa in *finestre* e viene classificato il pixel centrale per ognuna di esse. Una volta finita la classificazione della prima finestra si passa alla prossima e così via fino alla fine dell'immagine. Un approccio di questo tipo teoricamente potrebbe sembrare valido ma in realtà è poco efficiente: le informazioni dei pixel classificati nella finestra precedente non vengono utilizzate per la classificazione dei pixel della finestra successiva. Di conseguenza, un metodo più efficace nello stato dell'arte richiede una singola CNN che si occupi della classificazione di tutti i pixel dell'immagine[7].
Un'architettura che può occuparsi di segmentazione semantica è formata da due parti fondamentali: un *encoder* e un *decoder*. L'encoder è la prima parte dell'architettura e si occupa, data in ingresso un'immagine ($H \times W \times 3$) da segmentare, di estrarne le *feature* attraverso una serie di convoluzioni e *downsampling*. Il decoder, invece, a partire dalla *feature map* generata dall'encoder, genera una mappa di segmentazione ($H \times W$) attraverso convoluzioni e tecniche di *upsampling* (Figura 2.3).

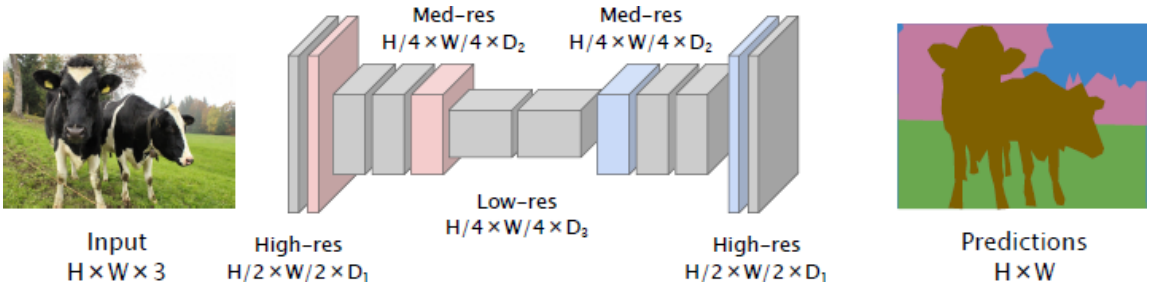


Figura 2.3. Esempio di architettura con encoder-decoder.

Allo stato dell'arte esistono varie architetture con caratteristiche diverse che hanno lo scopo di risolvere task di Semantic Segmentation ma per obiettivi e contesti differenti: U-Net[8], LinkNet[22], PSPNet[23], FPN[24] e altre.
Nel mio lavoro di tesi ho utilizzato un'architettura di tipo U-Net, quindi ritengo necessario descrivere i suoi elementi distintivi.

2.1.1.1 U-Net

Nata per la segmentazione di immagini biomediche, U-Net presenta caratteristiche che la distinguono dalle altre reti che si occupano di segmentazione semantica. Come si può vedere dalla Figura 2.4, la prima caratteristica che balza all'occhio è sicuramente la presenza di concatenazioni che uniscono i blocchi dell'encoder con i blocchi del

decoder. L'encoder si aspetta in input un'immagine ad un canale. Esso è formato da convoluzioni 3×3 seguite da una ReLU. Ogni blocco convolutivo è seguito da un *downsampling* effettuato tramite *Max Pooling*[9]. Dopo le due convoluzioni del blocco intermedio, inizia il decoder che ha la caratteristica di effettuare l'*upsampling* tramite delle *convoluzioni trasposte* (o *up-convolution*) 2×2 che dimezzano il numero dei canali. Qui entrano in gioco le concatenazioni con i blocchi dell'encoder che permettono di non perdere informazioni durante gli upsampling. In uscita, la rete genera una mappa di segmentazione a due canali.

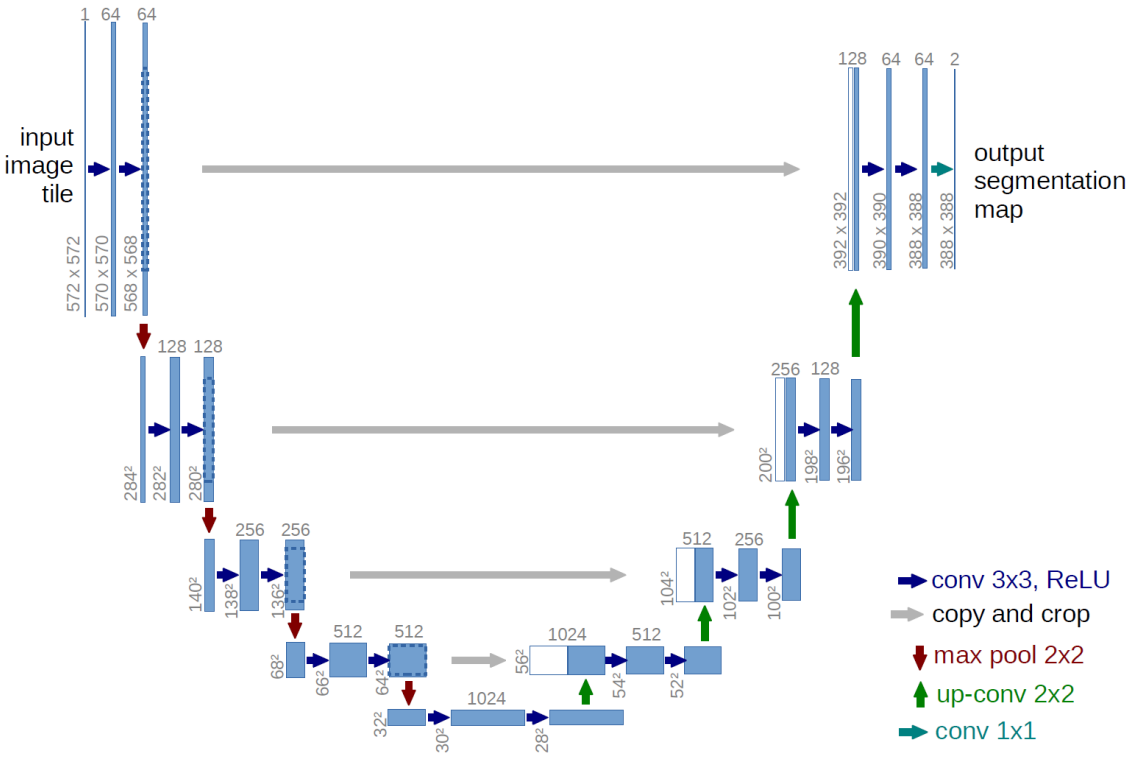


Figura 2.4. Architettura U-Net.

La presenza di così tanti canali nel decoder non solo permette di ottenere maggiori informazioni di contesto rispetto alle altre architetture, ma fa anche in modo che la segmentazione vada a concentrarsi maggiormente sulla parte centrale dei blocchi convolutivi. Questa caratteristica peculiare della rete garantisce la segmentazione di immagini ad alta risoluzione senza avere problemi di performance.
Tale architettura, e sue varianti, sono risultate utili in campo biomedico, ad esempio per la segmentazione di immagini cerebrali (BraTS[10]), oppure per la predizione dei siti di legame delle proteine[11].

2.1.2 Instance Segmentation

L'Instance Segmentation permette di riconoscere i pixel associati ad una classe e si occupa di rilevare e delineare ogni istanza distinta di un oggetto che appare in un'immagine. Di conseguenza, la mappa di segmentazione non riguarda solo le classi da rilevare, ma anche le istanze distinte di ogni classe (Figura 2.5).

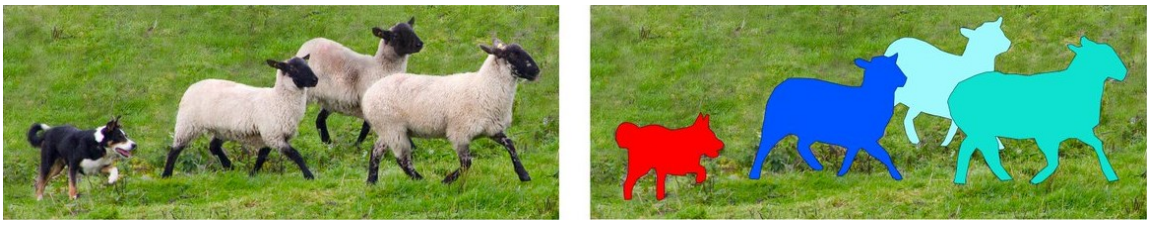


Figura 2.5. Esempio di Instance Segmentation.

L'Instance Segmentation è formata da due parti fondamentali:

- **Object Detection:** è la rilevazione degli oggetti nell'immagine così da riconoscere le dimensioni e la posizione degli stessi (rilevamento *bounding box*).
- **Semantic Segmentation:** in questa fase, per ogni bounding box della fase precedente, viene effettuata una segmentazione e, quindi, vengono etichettati i pixel in base alla possibile classe di appartenenza.

Una delle architetture più utilizzate per task di Instance Segmentation è sicuramente Mask R-CNN[12], la quale estende l'architettura R-CNN[13], in particolare la sua ultima versione (Faster R-CNN[14]), aggiungendo un ramo per la previsione di maschere di segmentazione per ogni *Region of Interest* (RoI).

2.2 Lavori correlati

In questa sezione esporrò e descriverò alcuni dei lavori che si sono occupati di tracciamento di linee curve o traiettorie. Tali lavori si basano su tecniche di Machine Learning.
Durante la mia ricerca di soluzioni che potessero risolvere un task di segmentazione di traiettorie (nel caso sotto esame rappresentabili da linee curve), mi sono imbattuto in diversi studi per il riconoscimento di linee stradali (*Lane detection*). La **CondLaneNet** di Lizhe Liu et al.[15] è uno di questi: l'architettura è formata da una CNN condizionale e una rete ricorsiva. La prima ha l'obiettivo di trovare le istanze delle tracce e successivamente predirre la forma delle traiettorie mentre la seconda aiuta a rilevare le traiettorie con topologie complesse (discontinuità). Un altro lavoro che può essere di ispirazione è quello di Amartansh Dubey e K. M. Bhurchandi[16], in cui risolvono un task di line detection estendendo la trasformata di Hough: un algoritmo

che traccia le linee curve dissezionando ognuna di esse in infinite linee di Hough, inoltre utilizza un *Continuous Frame Feedback Algorithm* per correggere situazioni con discontinuità o problemi di illuminazione.
I due studi appena citati presentano dei task simili all'obiettivo di questa tesi, ma con delle grosse differenze legate al contesto di studio diverso:

- La lane detection tiene conto, di solito, della prospettiva stradale. Invece, per la rilevazione di una traiettoria particellare vi è assenza di prospettiva.
- Le immagini in nostro possesso hanno un tipo di rumore (di fondo lineare e gaussiano intorno alle traiettorie) caratteristico.
- Le linee stradali sono, di solito, continue o tratteggiate. Le traiettorie particellari, invece, sono formate da bulbi di luce, a volte tanto vicini da sembrare continui, di dimensioni variabili.

Per queste motivazioni, sono andato alla ricerca di lavori che fossero più in linea con il contesto della fisica quantistica e dello studio delle particelle.
Georg Stimpfl-Abele e Lluis Garrido[17] sono stati tra i primi ad utilizzare una *Recurrent Neural Network* (RNN)[18] per tracciare le traiettorie particellari. Ci sono anche i lavori di Steven Farrell et al.[19] e Dmitriy Baranov et al.[20] che trovano soluzioni differenti allo stesso tipo di problema.
Anche questi in casi, però, siamo lontani dal task di instance segmentation per la rilevazione di tracce. Infatti, in questi lavori, si utilizzano degli *hit points* per predirre la posizione della traiettoria e non ci sono scontri particellari da analizzare attraverso delle immagini.
Di conseguenza, la soluzione è stata quella di lavorare su un sistema creato appositamente per questo task.

Capitolo 3

Metodi

In questo capitolo mi occuperò di descrivere le varie parti di cui è composta l'architettura creata e come si è giunti a progettarla. Partendo dalle fotografie scattate all'interno della SSSC, l'obiettivo di tale modello è rilevare le singole tracce di un evento generato all'interno del fasotrone del JINR al fine di estrarre da tali tracce le informazioni riguardanti le stesse: Angolo iniziale, angolo finale e raggio medio. In seguito, studiare l'intensità delle traiettorie al fine di migliorarne la rilevazione.
Il primo scoglio da affrontare è stato comprendere il tipo di task da affrontare e, successivamente, generare un dataset che potesse aiutare a raggiungere gli obiettivi preposti. Nei seguenti paragrafi vedremo come si è arrivati al suddetto sistema.

3.1 Task di Instance Segmentation

Non è stato semplice comprendere fin da subito il tipo di task da affrontare. Ogni immagine in nostro possesso rappresenta la fotografia dell'evento particellare e presenta una o più tracce che si incrociano o scontrano in modi e posizioni diverse. Il focus principale di questo studio però, non è l'evento nella sua interezza, ma il comportamento delle singole tracce (dunque delle particelle) valutate come oggetti distinti nell'immagine. Per questo motivo si è giunti alla conclusione che si trattasse di un problema di Instance Segmentation: si ha una sola classe ("Traccia") di cui possiamo avere molteplici istanze con caratteristiche in comune all'interno della stessa immagine.
Lo step successivo è stato quello di selezionare l'architettura giusta per i nostri scopi, arrivando alla valutazione di due opzioni.
La prima opzione, la più scontata, è un'architettura di tipo Mask R-CNN[12] che prende come input le immagini degli eventi e dà in uscita le maschere di segmentazione delle tracce presenti dalle quali vengono estratte le informazioni di ogni traccia.
La seconda opzione è un sistema ibrido formato da due parti:

- **Generazione delle maschere:** una rete neurale convoluzionale che si occupa di Semantic Segmentation e che, quindi, segmenta una sola classe ("Traccia"), ma senza tener conto delle istanze della stessa all'interno dell'immagine. Di

conseguenza, ogni maschera presenta una o più tracce senza distinguerle e si valuta e si segmenta l'evento fotografato nella sua interezza.

- **Divisione delle tracce:** in questa fase le maschere precedentemente generate dalla rete vengono processate da un algoritmo di Computer Vision che si occupa di suddividerle nelle singole tracce di cui sono composte. Le operazioni morfologiche, di cui è formato tale algoritmo, sfruttano le proprietà geometriche delle tracce.

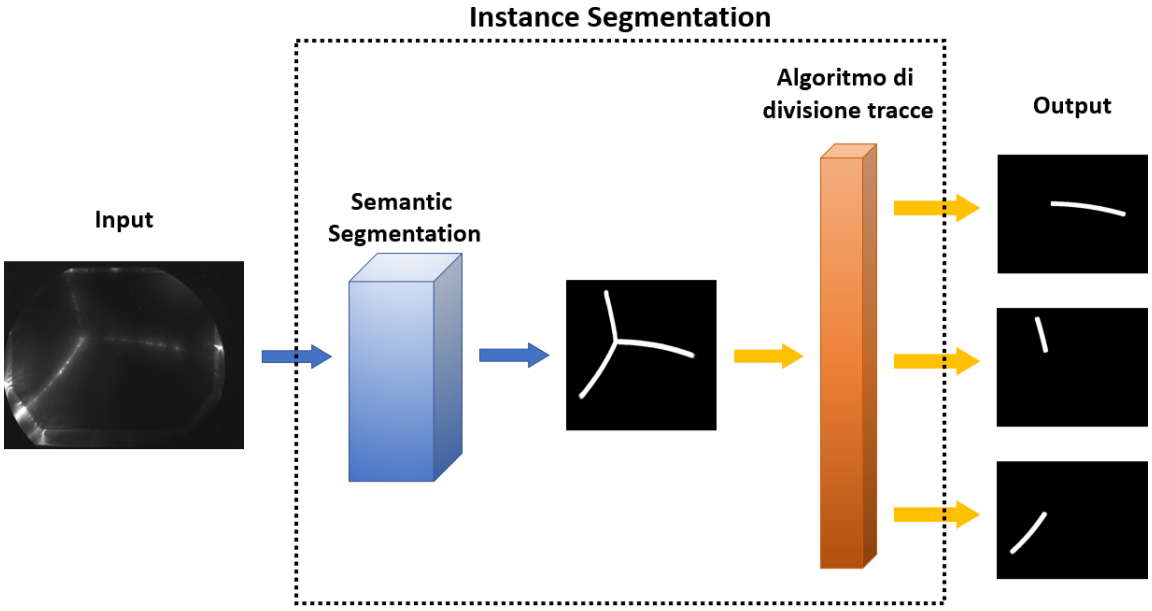


Figura 3.1. Schema riassuntivo delle prime due parti del sistema creato per questo lavoro di tesi.

Infine, si studia l'intensità delle tracce trovate.
La scelta definitiva è ricaduta sulla seconda architettura lasciando la prima a possibili sviluppi futuri. Ho preferito lavorare con il secondo modello poiché ho ritenuto necessario sfruttare a pieno le caratteristiche delle particelle cariche in movimento. Con il sistema che andrò ad esporre in questo capitolo, le traiettorie verranno rilevate in base alle loro proprietà geometriche (rappresentabili da archi di circonferenza). Al contrario di un approccio come quello della prima architettura: la rilevazione delle istanze avviene solo in base alle caratteristiche dei pixel di una certa istanza della classe "Traccia".

3.2 Generazione delle maschere

In questa sezione descriverò la prima parte del sistema e come si è arrivati ad alcune delle scelte implementative.

In primo luogo, ho pensato ad una rete neurale che si occupi di segmentazione di immagini ma che presenti un modello preallenato. In questo modo avremmo potuto utilizzare il *fine-tuning* sia per velocizzare l'allenamento, sia per superare il problema di avere un dataset non troppo esteso.
Per queste motivazioni, abbiamo trovato la libreria *Segmentation Models*[21]: una repository di modelli di segmentazione (ha quattro tipi di architetture: U-Net, LinkNet[22], PSPNet[23] e FPN[24]) con *Backbone* preallenate su Imagenet[25].
Sia per ragioni di semplicità che di conoscenza pregressa, ho deciso di utilizzare un'architettura di tipo U-Net.

3.2.1 Implementazione della U-Net

La libreria *Segmentation Models* dà la possibilità di scegliere la Backbone da utilizzare come encoder del nostro modello: ho scelto una VGG-16[26].
Prima di discutere dell'architettura utilizzata, parlerò dell'input e dell'output di tale modello. Come precedentemente esposto, l'input deve essere l'immagine dell'evento di scontro particellare, mentre l'output una maschera che segmenta la totalità delle tracce presenti. Per questo motivo, l'input avrà tre canali (R, G e B), mentre, trovandoci di fronte ad un task monoclasse (Traccia o Background), l'output ne ha solo uno. Per ciò che concerne le dimensioni di tali immagini, invece, si è pensato di utilizzare il modello con tre diverse risoluzioni (la dimensione delle immagini del Dataset è 658×517): 128×128, 256×256 e 512×512. Ci sarà un'analisi più approfondita delle implicazioni legate a tale scelta nel capitolo "Risultati".

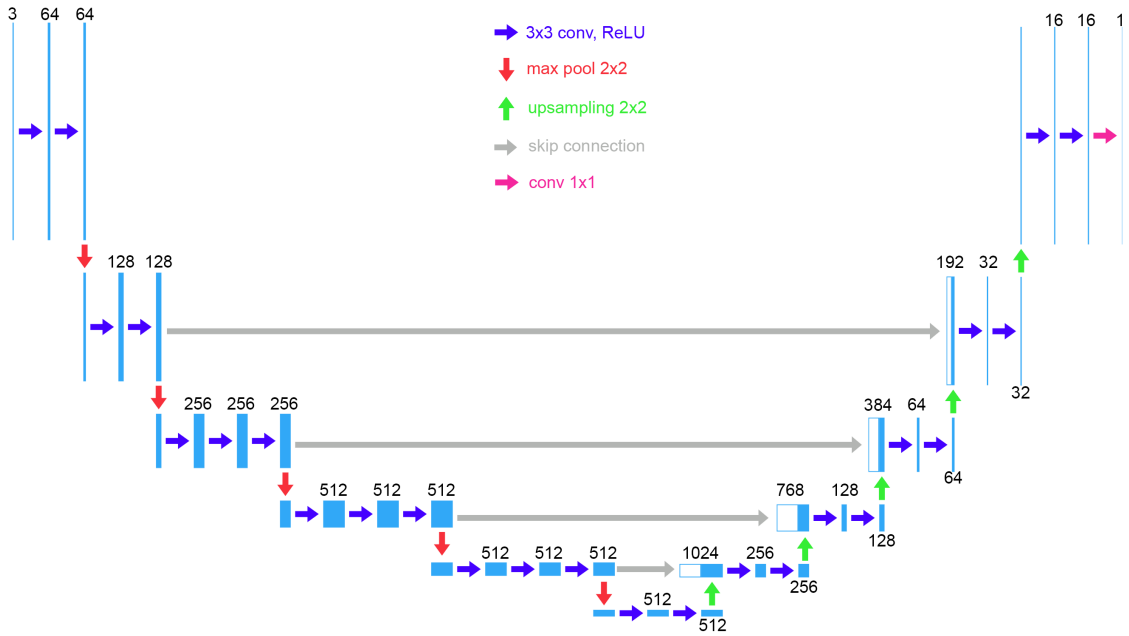


Figura 3.2. Architettura utilizzata per la segmentazione delle tracce.

L'architettura (Figura 3.2) presenta 5 blocchi convolutivi per l'encoder (VGG-16), un blocco centrale e altri cinque blocchi per il decoder. I primi due blocchi dell'encoder presentano due livelli convoluzionali, mentre gli altri tre ne contengono tre; ogni blocco è delimitato dal successivo da un *Max Pooling* 2×2. Il blocco centrale presenta due convoluzioni seguite da *Batch Normalization*. Ogni blocco del decoder mostra due convoluzioni e *Batch Normalization* ed è introdotto da un *Upsampling* 2×2; l'ultimo blocco ha infine una convoluzione 1×1 con una *sigmoide* (classificazione binaria) come funzione di attivazione in uscita. I primi quattro blocchi del decoder sono concatenati con gli ultimi quattro dell'encoder tramite *skip connections*. Tutte le convoluzioni sono effettuate con un filtro 3×3 e con funzione di attivazione *ReLU*.

3.2.2 Loss, Ottimizzatore e Metrica della Semantic Segmentation

Per quanto riguarda la loss, la scelta è sembrata obbligatoria, infatti, per questo tipo di task con classificazione binaria, allo stato dell'arte vi è l'utilizzo della *Binary Crossentropy*; L'ottimizzatore che ho deciso di utilizzare è Adam[27] (le cui learning rate utilizzate saranno discusse nel capitolo "Esperimenti").
Inizialmente ho pensato di utilizzare una metrica semplice per valutare le performance del modello: la *Pixel Accuracy* calcolata con la formula mostrata in 3.1 ma per ogni pixel dell'immagine.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1}$$

In cui TP sono i *True Positive*, ovvero i pixel classificati correttamente, TN sono i *True Negative* quindi i pixel non classificati correttamente, FP sta per *False Positive* e rappresenta i pixel classificati con errore e FN sono i *False Negative*: i pixel non classificati con errore.
Tale metrica mi dà una valutazione sulla predizione della maschera generata rispetto al Ground Truth nel suo complesso (Traccia + Background) e, siccome la maschera rappresenta al massimo il 10% dei pixel totali dell'immagine, la valutazione avrebbe potuto ridare delle performance quasi sempre superiori al 90% e quindi poco rilevanti per il feedback che vorremmo.
Di conseguenza, una metrica più adatta al nostro scopo è stata sicuramente la *Binary Intersection over Union* (Binary IoU) che calcola l'IoU per le classificazioni binarie.

$$IoU = \frac{TP}{TP + FP + FN} \tag{3.2}$$

Quindi, specifico come *target* la classe Traccia e si indica come pixel TP quello che supera una soglia dell'IoU dello 0.5: si stimano le prestazioni del modello valutando la maschera predetta rispetto alla maschera del Ground Truth ignorando i pixel riguardanti il background.

3.3 Divisione delle tracce

La seconda parte del sistema è composta da un algoritmo che si occupa di prendere in ingresso le maschere generate dalla U-Net e dividerle nelle maschere delle singole tracce di cui sono formate. Tale lavoro è stato sicuramente il più difficile da attuare, poiché ha richiesto l'utilizzo di diverse tecniche di Computer Vision e la realizzazione di diverse trasformazioni sulle immagini. Alla base di tali operazioni vi è la conoscenza delle proprietà geometriche delle tracce (archi di circonferenza) descriventi il movimento dalle particelle nella SSSC. Inoltre, si è dato per scontato che non ci siano sovrapposizioni di tracce. In caso contrario, ovvero nel caso in cui due tracce dovessero sovrapporsi, l'algoritmo rileverà il punto di sovrapposizione come un punto di scontro tra le due traiettorie particellari, quindi, le maschere delle due tracce verranno suddivise all'altezza dell'intersezione in quattro. Una casistica del genere verrà, ovviamente, valutata come sbagliata durante il calcolo delle metriche.
Prima di entrare nel merito dell'algoritmo utilizzato, bisogna fare una premessa: come sopraindicato, le immagini date alla U-Net, per poter essere elaborate dal modello, vengono prima rese quadrate e, di conseguenza, lo sono anche le maschere in uscita. Quindi, prima di far partire l'algoritmo, la maschera in uscita dal modello va elaborata in modo tale da riottenere la *ratio* dell'immagine originale. Tale operazione è importante per recuperare le caratteristiche geometriche delle tracce le quali vengono perse quando l'immagine viene resa quadrata. Pertanto, essendo le immagini originali 658×517, la *ratio* è $1.\overline{27}$, quindi le maschere 128×128, 256×256 e 512×512 le ho ridimensionate rispettivamente in 128×101, 256×201 e 512×402. Ho fatto in modo che una delle due dimensioni rimanesse uguale (la larghezza), mentre l'altra diminuisse rispettando la ratio di cui sopra (con arrotondamento per eccesso o per difetto a seconda del caso).

3.3.1 Implementazione algoritmo

L'algoritmo che andrò a discutere presenta diversi parametri che verranno analizzati nel dettaglio nella sezione "Parametrizzazione algoritmo". Per semplicità esporrò le operazioni effettuate dividendole in punti:

1. Binarizzo l'immagine data in ingresso utilizzando l'operazione di *Thresholding*[28], impostando una soglia, così da rendere 0 il background e 255 la maschera. Tale operazione è fondamentale per poter lavorare con più facilità sull'immagine segmentata.

2. Creo un *kernel* circolare e lo utilizzo per la trasformata *Top-hat*[29] per sottrarre dalla maschera il punto di congiunzione delle traiettorie che, la maggior parte dei casi in cui mi sono imbattuto, risulta essere "più grande" rispetto alla maschera delle tracce. In questo modo, eliminando il centro, gli elementi rimanenti nell'immagine saranno le tracce. Successivamente effettuo una *Opening*[30] (una *Erosion* seguita da una *Dilation*) per eliminare eventuale rumore causato dalla Top-hat.

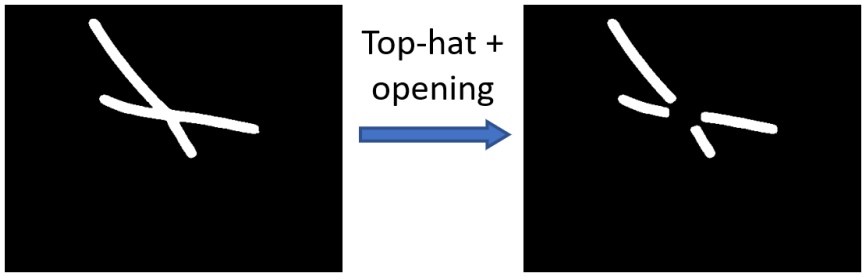


Figura 3.3. Esempio di top-hat con successiva opening.

3. Grazie all'operazione *Connected Component Labeling (CCL)*[31], si rilevano le regioni di pixel collegate (in alcuni casi le tracce divise) e si etichettano. Il risultato ci permetterà di distinguere le maschere delle tracce. Quindi, successivamente, con la trasformazione *Watershed*[32] recupero i pixel del centro "persi" durante la Top-hat e salvo le maschere generate.
   L'applicazione di tale operazione è mostrata di seguito:

```
ker_wat = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (30, 30))
# sure background area
sure_bg = cv2.dilate(opening,ker_wat,iterations=3)
# Finding unknown region
unknown = cv2.subtract(sure_bg,opening)
# Marker labelling
ret, markers = cv2.connectedComponents(opening)
# Add one to all labels so that sure background is not 0, but 1
markers = markers+1
# Now, mark the region of unknown with zero
markers[unknown==255] = 0
markers = cv2.watershed(mask,markers)
```

   Come si può notare, l'operazione CCL viene effettuata direttamente sul risultato della Opening del punto precedente. L'output risultante è una maschera di segmentazione che associa ad ogni pixel l'etichetta identificante la regione di pixel collegata (`markers`). Successivamente, per utilizzare questi markers e, allo stesso tempo, recuperare i pixel sottratti dalla maschera durante la Top-hat, si utilizza la trasformata Watershed. Si può vedere che prima si applica una dilation alla maschera e poi si fa una sottrazione con la maschera non dilatata per avere i pixel "`unknown`". Tale maschera di pixel è fondamentale poiché, associata ai markers e etichettandola con il valore 0, permette alla trasformata Watershed di poter recuperare dalla maschera iniziale (`mask`) i pixel. Ora che ad ogni traccia è associato un valore distintivo, posso salvare, quindi, ognuna di esse in un'immagine isolata.

4. Per ognuna delle nuove maschere isolate, calcolo gli autovalori della matrice Hessiana e ne estraggo i *ridge*[33]. I ridge di un'immagine sono i pixel che si trovano vicino ad altri nettamente più luminosi o meno luminosi. Calcolando gli autovalori della matrice Hessiana si ottengono λ_1 e λ_2 di ogni pixel nell'immagine. Ciò che cerco sono i pixel per cui λ_1 è negativo e piccolo. Tale caratteristica corrisponde ai pixel che rappresentano i picchi dell'immagine, ovvero un cambio netto di intensità rispetto ai pixel adiacenti. Quindi, nel contesto delle nostre maschere di segmentazione, i ridge corrispondono agli estremi della maschera e ai cambi di direzione netti della maschera.
   Per estrarre i ridge, dopo aver calcolato tutti i λ_1 dei pixel, li inserisco in una lista e ordino, quest'ultima, dal valore più piccolo al più grande. Dalla lista, quindi, seleziono gli elementi la cui posizione è al di sotto di una certa soglia: rilevo come ridge effettivi tutti i pixel i cui λ_1 sono stati selezionati. Di seguito, conto i ridge rilevati calcolando i componenti connessi (CCL) e contando il numero di regioni presenti, nel caso esse siano più di due (più di due ridge), dopo averle allargate, le sottraggo alla maschera (se i ridge sono più di due si suppone che due rappresentino gli estremi della maschera, mentre il terzo è il cambio di direzione netto della maschera). Tale operazione è fondamentale poiché, come accennato, potrebbe accadere che la trasformata top-hat non divida le tracce, quindi, per i casi in cui la trasformata fallisca e nella maschera ci siano più di una traccia che generano uno scontro (più di due ridge per maschera), si prova a fare un'ulteriore divisione sottraendo i ridge e, quindi, provando ad eliminare il punto di incontro delle traiettorie. Poi, si riutilizza la CCL e la trasformazione Watershed per identificare le nuove tracce rilevate e recuperare i possibili pixel cancellati. Infine salvo ogni traccia in un'immagine.

Figura 3.4. Esempio di ridge detection con sottrazione dei ridge rilevati.

   Dalla Figura 3.4 si può ben vedere come i tre ridge rilevati sono i due estremi della maschera e la curva causata dallo scontro tra le traiettorie.

5. Per ogni traccia isolata applico la trasformazione *thinning*[34], la quale scheletrizza la maschera generando uno scheletro i cui punti siano equidistanti dai bordi.

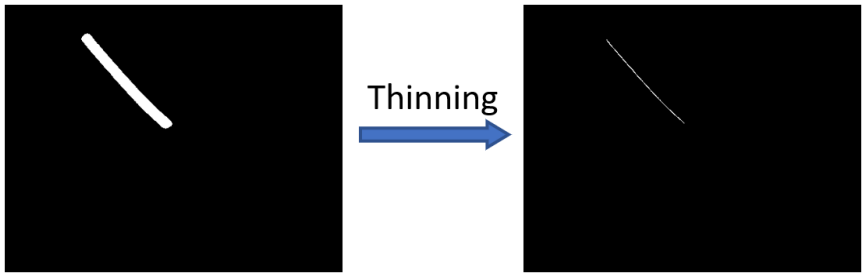


Figura 3.5. Esempio di scheletrizzazione con trasformazione thinning.

Successivamente, ne estraggo gli *end points* con la seguente funzione:

```
def get_end_pnts(pnts, img):
    extremes = []
    for p in pnts:
        x = p[0]
        y = p[1]
        n = 0
        if x==(img.shape[1]-1) or y==(img.shape[0]-1):
            n = 1
        else:
            n += img[y - 1,x]
            n += img[y - 1,x - 1]
            n += img[y - 1,x + 1]
            n += img[y,x - 1]
            n += img[y,x + 1]
            n += img[y + 1,x]
            n += img[y + 1,x - 1]
            n += img[y + 1,x + 1]
            n /= 255
        if n == 1:
            extremes.append(p)
    return extremes
```

Infine, estraggo anche un punto intermedio dello scheletro affinché i tre punti possano rappresentare un arco di circonferenza.

6. Da questo step in poi, l'algoritmo è molto simile a quello mostrato nella sottosezione "Generazione del Ground Truth". Infatti, si controlla se i punti siano allineati o meno e di conseguenza se trattare la traccia come un segmento o un arco di circonferenza.
   Vi è anche una terza possibilità in questo caso. Potrebbe accadere che la

maschera della traccia rilevata sia così piccola che la sua scheletrizzazione sia un singolo punto. In questa circostanza, la traccia è quindi un punto.

7. Infine, salvo le informazioni riguardanti le tracce rilevate (centro, raggio, angolo iniziale e finale per gli archi; coordinate degli estremi per i segmenti; coordinate per i punti) in un file *CSV* apposito strutturato in modo tale che ogni riga presenti le informazioni di una traccia e, in particolare:

   - La prima colonna contiene il nome dell'evento a cui appartiene la traccia rilevata.
   - La seconda colonna descrive se la traccia rilevata è un arco di circonferenza, un segmento o un punto. Da questo dato la composizione delle colonne seguenti è differente. Infatti:
     (a) Nel caso in cui la seconda colonna sia "Arco", le colonne successive, tranne l'ultima (la settima), contengono rispettivamente i valori delle coordinate del centro dell'arco, la lunghezza del raggio, l'angolo e l'angolo iniziale.
     (b) Nel caso in cui la seconda colonna sia "Segmento", le due colonne successive, contengono le coordinate dei due estremi del segmento.
     (c) Nel caso in cui la seconda colonna sia "Punto", la colonna successiva contiene le coordinate del punto.
   - L'ultima colonna presenta l'identificativo della traccia all'interno dello stesso evento: "Track1, Track2, ...".

   Segue un esempio delle righe appena descritte:

```
...
L1Ap5_12-1-79;Arco;(129.29661016949152, 1269.7966101694915);
1111.52;5.239;277.584;Track1

L1Ap5_12-1-79;Arco;(778.3969957081545, -381.9914163090129);
762.924;10.666;136.827;Track2

L1Ap5_14-1-101;Arco;(196.75510204081633, 965.0204081632653);
799.712;14.8;272.383;Track1
...
```

   Si può notare come il punto e virgola abbia lo scopo di dividere le colonne.

Quindi, all' uscita dell'algoritmo, abbiamo le informazioni delle tracce, salvate in un file apposito. Tali dati saranno fondamentali per il prossimo, e ultimo, step: Lo studio dell'intensità.

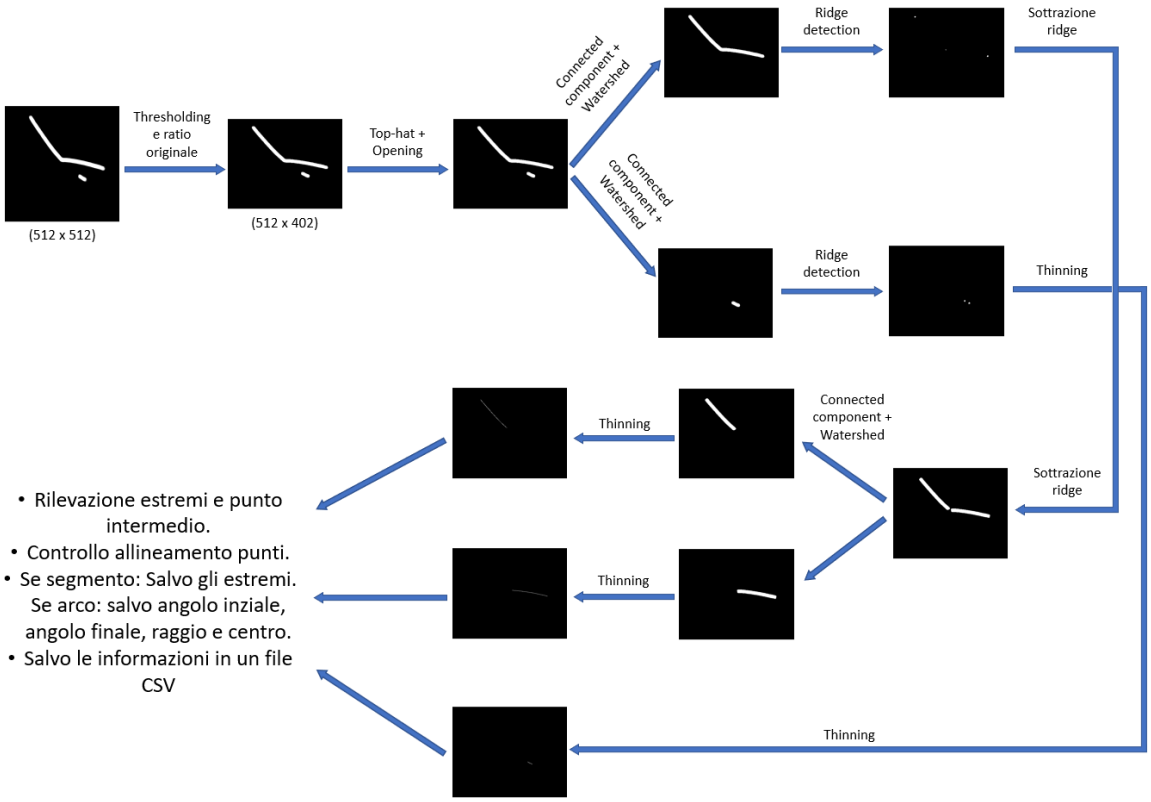


Figura 3.6. Esempio di applicazione dell'algoritmo con una maschera 512×512 in ingresso.

3.3.2 Valutazione delle prestazioni e metriche

Prima di discutere lo studio dell'intensità, bisogna trattare le metriche utilizzate per valutare le performance dell'algoritmo e, in generale, di tutto il sistema.
Allo stato dell'arte, le metriche più utilizzate per task di Instance Segmentation sono sicuramente *Precision* e *Recall* che danno, rispettivamente, un feedback su quanto, in proporzione, le predizioni positive sono effettivamente corrette, e quanto, in proporzione, i positivi effettivi sono stati predetti correttamente. Calcolando poi, di conseguenza, la curva Precision-Recall si ottiene anche la *Mean Average Precision* (mAP)[35] misurata su tutte le classi e utilizzando diverse soglie della IoU.
Nel nostro caso, però, c'è una differenza sostanziale: avendo una sola classe con più istanze, non può essere calcolata la mAP. Inoltre, ci sono altre metriche che sono di nostro interesse e che possono darci un feedback più rilevante sulle prestazioni del nostro sistema. Il primo passo è sicuramente quello di calcolare una Binary IoU indicando come target la maschera delle tracce, come già realizzato per la metrica della U-Net, tra le tracce rilevate dal sistema e il Ground Truth, utilizzando una soglia. Da qui si possono quindi stabilire dei *True Positive* (previsioni positive con IoU al di sopra della soglia), *False Positive* (previsioni positive con IoU al di sotto della soglia) e *False Negative* (previsioni negative errate). Bisogna notare che i concetti

di TP, FP e FN sono cambiati rispetto alla sezione "Loss, Ottimizzatore e Metrica della Semantic Segmentation". Infatti, questi riguardano l'intera traccia segmentata e rilevata rispetto al Ground Truth, mentre, per la valutazione delle prestazioni della U-Net, TP, FP e FN riguardavano la classificazione del singolo pixel.
Date queste informazioni, ho potuto calcolare due metriche che sono risultate fondamentali per lo studio delle prestazioni. La prima descrive il numero di eventi di cui sono state predette correttamente tutte e sole le tracce (eventi per cui risultano $FP = 0$ e $FN = 0$), riportando anche il numero di TP rilevati; la seconda esprime il numero di eventi di cui non è stata predetta correttamente alcuna traccia (eventi per cui risulta $TP = 0$). A queste, poi, sono state aggiunte la Precision e la Recall per avere, come descritto precedentemente, un riscontro sulla precisione e la sensibilità del sistema.
Quindi, a partire dal calcolo della Binary IoU e stabilendo una soglia oltre la quale valutare come TP una predizione positiva, le metriche sono le seguenti:

- Numero eventi con $FP = 0$ e $FN = 0$.
- Numero eventi con $TP = 0$.
- Numero di TP.
- Numero di FP.
- $$Precision = \frac{TP}{TP + FP} \tag{3.3}$$
- $$Recall = \frac{TP}{TP + FN} \tag{3.4}$$

3.4 Studio dell'intensità

In questa sezione, infine, parlerò dell'ultimo step del nostro sistema. L'obiettivo è lo studio dell'andamento dell'intensità dei pixel delle traiettorie particellari per poter, successivamente, selezionare un area più precisa in cui è presente la traiettoria. Per farlo abbiamo bisogno di isolare le tracce all'interno dell'evento e poi successivamente prendere dei campioni.
Quindi, come si può vedere in Figura 3.7, la prima cosa da fare per isolare la traccia dalle altre, è utilizzare le maschere delle tracce divise che riceviamo dallo step precedente (sezione "Divisione delle tracce"). Sottolineo, però, che prima di compiere tale isolamento, la maschera viene allargata, utilizzando un kernel ellittico, per assicurarmi di coprire tutta la traiettoria della particella.

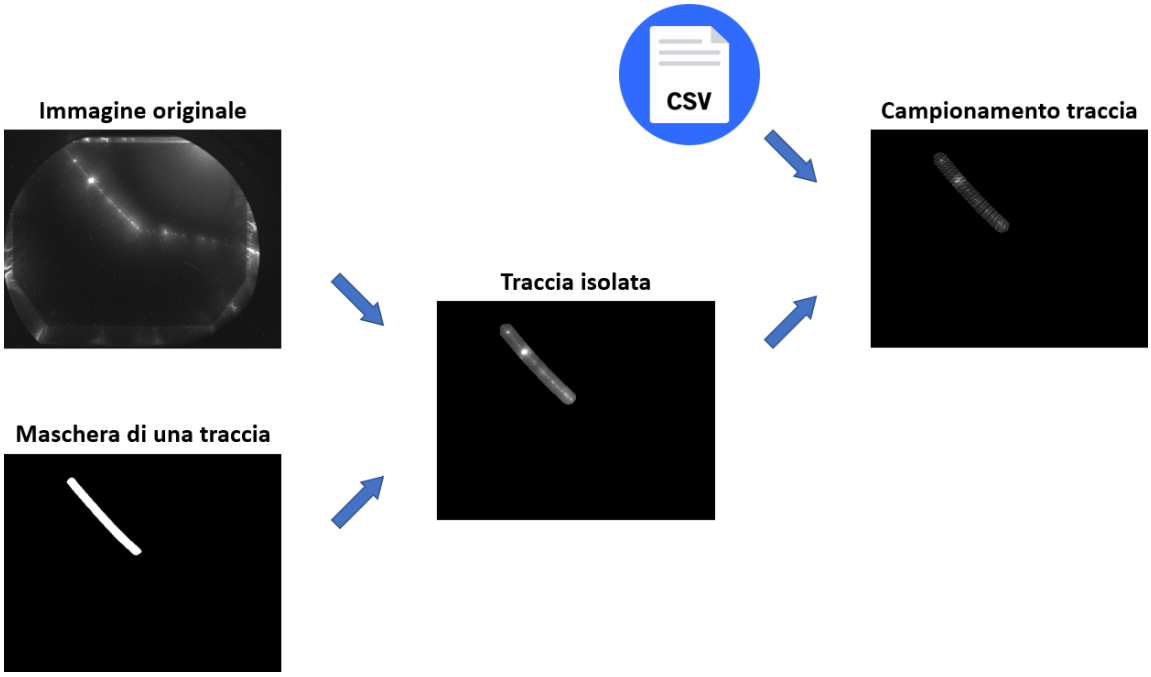


Figura 3.7. Esempio di campionamento.

Successivamente, abbiamo bisogno del file CSV in cui abbiamo salvato le informazioni riguardanti le tracce. In particolare, utilizzando gli angoli, il centro e il raggio (aumentato per l'occasione per poter coprire l'intera traccia allargata) posso campionare la nostra traccia dall'angolo iniziale a quello finale impiegando un certo angolo di campionamento che dipenda dal raggio. Il campionamento che andremo a fare sarà sempre trasversale rispetto all'andamento della traccia e, quindi, perpendicolare rispetto alla tangente all'arco.
Prima di scegliere l'angolo di campionamento bisogna specificare che il raggio della traccia è definito in pixel e quindi lo stesso deve essere per l_s (lunghezza dell'arco di circonferenza tra un campione e l'altro). Prendendo in considerazione che vorremmo evitare il più possibile perdite di informazioni e, quindi, parti della traccia non campionate, idealmente $l_s = 0.5pixel$.
Di conseguenza, l'angolo di campionamento θ_s è:

$$\theta_s = \frac{0.5pixel}{r} \tag{3.5}$$

Quindi, tale angolo è tanto più piccolo quanto l'arco di circonferenza rilevato è più distante dal centro. In questo modo, posso assicurarmi un numero di campioni sempre elevato e una distanza tra i campioni che non comporti possibili perdite di informazioni importanti.

3.4.1 Estrazione e studio curve gaussiane

Ottenuti i campioni, bisogna estrarne il contenuto. Ogni campione non è altro che un segmento i cui punti rappresentano specifici pixel dell'immagine. Pertanto, inserisco i valori di tali pixel in una lista, ordinandoli dal pixel con distanza minore dal centro dell'arco di circonferenza a quello con distanza maggiore.
Prima di esporre la procedura con cui ho trovato i parametri per generare una curva che potesse descrivere la distribuzione dei pixel nei campioni, ho bisogno di parlare del controllo effettuato sulla lunghezza della lista e, quindi, sul numero di pixel del campione analizzato. In particolare, dopo diverse prove ed esperimenti, ho deciso di analizzare per lo studio dell'intensità, solo i campioni che presentano un numero di pixel superiore a 13. Questa scelta l'ho fatta poiché ho notato una notevole difficoltà da parte delle funzioni di fitting di costruire una funzione adatta per i campioni con un numero di pixel inferiore a 13.
Partendo, quindi, dalla lista creata, genero un grafico in cui sull'asse x vi è il numero di pixel del campione mentre lungo l'asse y vi sono i valori d'intensità dei pixel ($h(x)$). Come si può vedere, nell'esempio in Figura 3.8, il campione preso come esempio esprime i valori di 20 pixel. Si può notare come il comportamento del grafico potrebbe descrivere una funzione gaussiana, o più di una. Per studiare al meglio il comportamento della curva $h(x)$, abbiamo bisogno di eliminare il rumore di fondo lineare che nell'evento fotografato rappresenta il background. Per questo motivo, ho modellato una funzione ($f(x)$) formata da due curve gaussiane (una associata al segnale della traiettoria e una al rumore generato) e una retta rappresentante il background lineare. Tale funzione ha il compito di descrivere al meglio i valori estratti dai pixel dei campioni. Quindi:

$$f(x) = g_s(x) + g_l(x) + r(x) \tag{3.6}$$

Dove $g_s(x)$ è la funzione gaussiana con la devazione standeard più piccola che ha il compito di descrivere il comportamento dei pixel della traiettoria, $g_l(x)$ è la funzione gaussiana con la deviazione standard più grande rappresentante il rumore gaussiano intorno alla traiettoria, mentre $r(x)$ è la retta descrivente il background lineare.

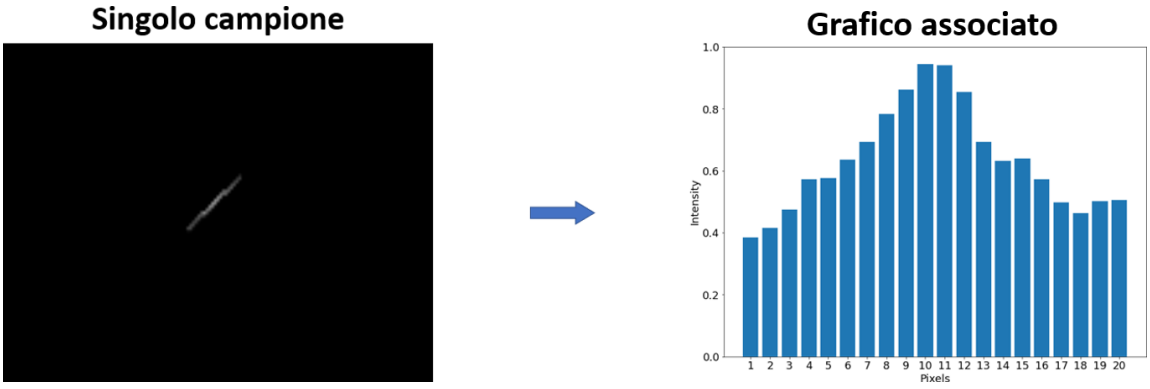


Figura 3.8. Esempio di grafico del campionamento.

Prima di parlare del *fitting* di $h(x)$, occorre specificare che l'operazione effettuata consiste nel trovare i valori ottimi dei parametri della funzione.
Infatti, la funzione $f(x)$ può essere scritta come:

$$f(x) = A_s e^{-\frac{(x-x0_s)^2}{2\sigma_s^2}} + A_l e^{-\frac{(x-x0_l)^2}{2\sigma_l^2}} + (mx + q) \tag{3.7}$$

Dove x è la variabile indipendente, A_s e A_l sono le ampiezze delle due funzioni gaussiane, $x0_s$ e $x0_l$ sono i punti medi, σ_s e σ_l sono le deviazioni standard e, infine, m e q sono, rispettivamente, coefficiente angolare e intercetta della retta. I parametri appena descritti devono essere trovati per poter *fittare* al meglio i dati a disposizione. Per farlo, ho utilizzato il modulo `lmfit`[36], il quale dà la possibilità di poter modellare la funzione da dover fittare indicando la variabile indipendente (nel nostro caso il numero di pixel) con la funzione `Model()`. Successivamente, con la funzione `Parameters()`, si aggiungono i parametri da ottenere e i limiti da imporre per ogni parametro.
Per il nostro task, i limiti imposti sono:

- $A_l < A_s < +\inf$
- $0 < A_l < A_s$
- $-\inf < x0_s < +\inf$
- $-\inf < x0_l < +\inf$
- $0.5 < \sigma_s < 5$
- $\sigma_s < \sigma_l < +\inf$
- $-1 < m < 1$
- $0 < q < \min(h(x))$

In seguito, con la funzione `fit()`, si possono ottenere i valori ottimali dei parametri. Di seguito la porzione di codice che si occupa di imporre i limiti ai parametri e far partire il *fitting*:

```
fit_model = Model(fitting, independent_vars=['x'])
params = Parameters()
params.add('x02', value=12, vary=True)
params.add('x01', value=12, vary=True)
params.add('m', value=0, vary=True, min=-1, max=1)
params.add('q', value=0.1, vary=True, min=0, max=min(y)-0.01)
params.add('a2', value=0.1, vary=True, min=0.01)
params.add('a1_minus_a2', value=0.1, vary=True, min=0.05)
params.add('a1', expr='a2 + a1_minus_a2')
params.add('sig1', value=2, vary=True, min=0.5, max=5)
```

```
params.add('sig2_minus_sig1', value=1, vary=True, min=0.1)
params.add('sig2', expr='sig1 + sig2_minus_sig1')
result = fit_model.fit(y, params, x=x)
```

si può notare come, per imporre dei limiti dipendenti da altri parametri (ad esempio $\sigma_s < \sigma_l < +\text{inf}$, con `sig1` $= \sigma_s$ e `sig2` $= \sigma_l$), bisogna creare dei nuovi parametri (`sig2_minus_sig1`) che permettano di relazionare i parametri tra loro ($\sigma_l - \sigma_s > 0$). Quindi, ho potuto calcolare $g_s(x)$, $g_l(x)$ e $r(x)$ e le ho potute mostrare tramite un grafico dopo aver sommato a $g_s(x)$ e $g_l(x)$ i valori di $r(x)$.

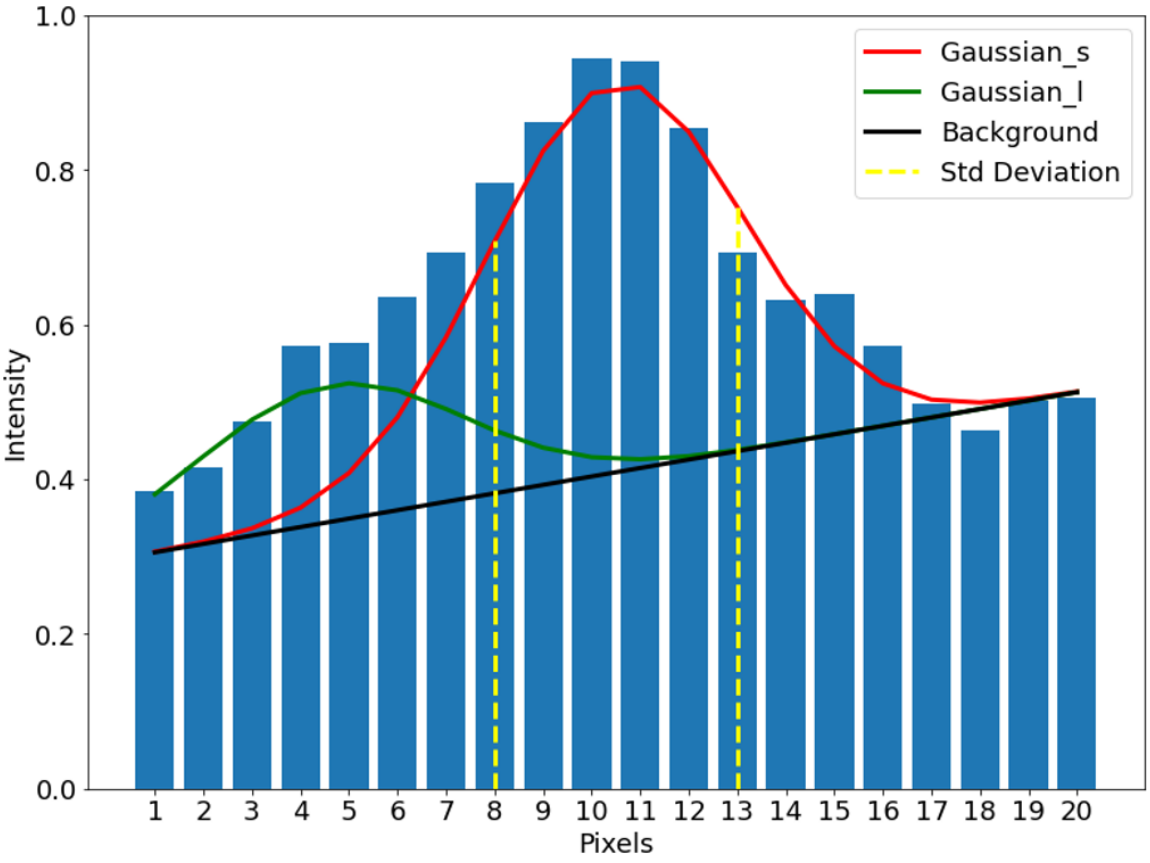


Figura 3.9. Esempio di *fit* del background e delle die curve gaussiane.

Si può notare, in figura 3.9, che nel grafico viene mostrata anche la deviazione standard di $g_s(x)$. Questo poiché l'area compresa tra $\lfloor x0_s - \sigma_s \rceil$ e $\lfloor x0_s + \sigma_s \rceil$ è di nostro interesse ai fini di selezionare i valori dei pixel rappresentativi della traiettoria particellare.
L'ultima operazione da effettuare è quella di acquisire tali valori sottratti del background e della gaussiana associata al rumore, quindi avremo:

$$clear(x) = h(x) - r(x) - g_l(x) \tag{3.8}$$

Con $x \in [\lfloor x0_s - \sigma_s \rfloor, \lceil x0_s + \sigma_s \rceil]$.
Quindi, tali valori vengono reinseriti nel campione, dopo averli moltiplicati per 255, per poterli visualizzare e fare in modo che possano rappresentare la traiettoria studiata (Figura 3.10).
Unisco, quindi, tutti i campioni ridotti di ciascuna traccia di un evento.
Successivamente, al fine di avere un maggior distacco tra la traiettoria particellare e il background (nullo), seleziono il valore maggiore di intensità dei pixel all'interno dell'evento e lo saturo a 255, così poi scalo di conseguenza tutti gli altri valori lasciando il background a 0. Il risultato finale è quello mostrato in Figura 3.11.

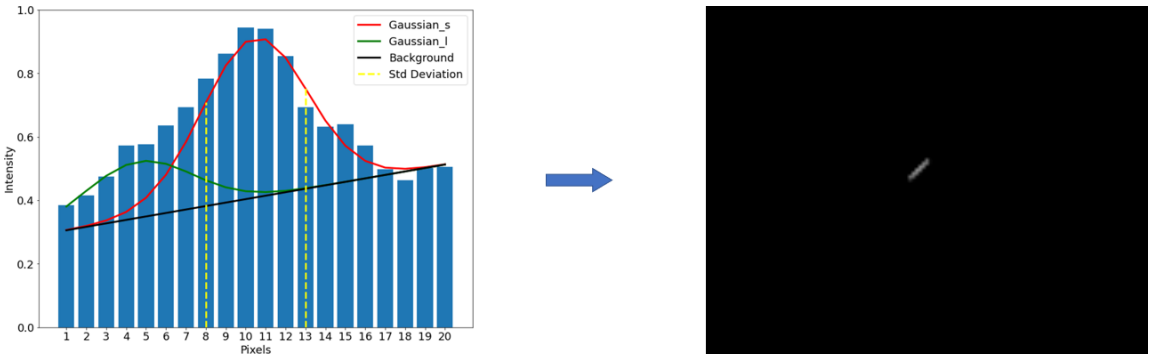


Figura 3.10. Esempio di riduzione del campione.

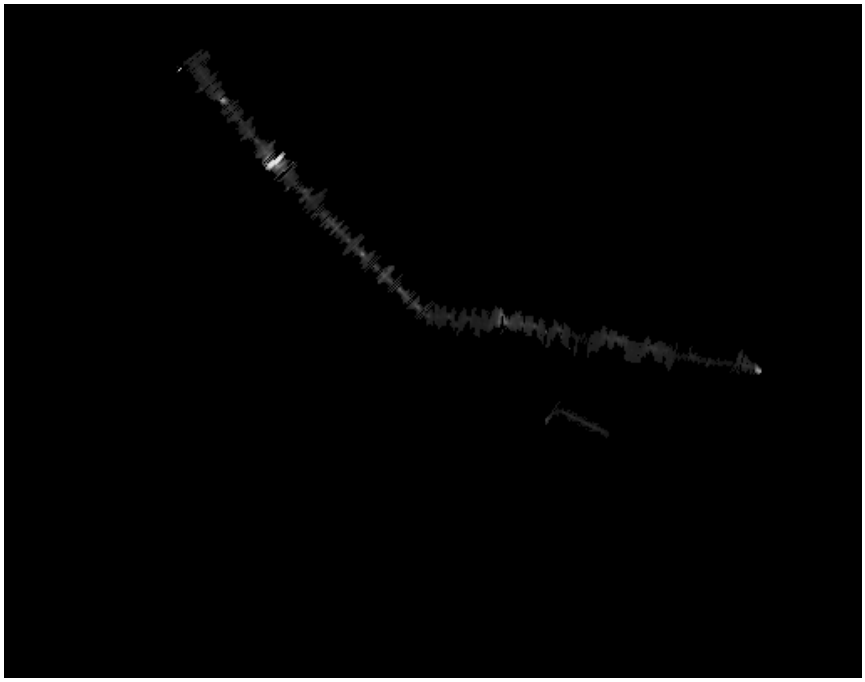


Figura 3.11. Output studio intensità.

Capitolo 4

Esperimenti

In questo capitolo mi occuperò di discutere gli esperimenti effettuati per testare e valutare le varie parti di cui è costituito il sistema. In particolare, prima esporrò come ho creato il dataset, successivamente parlerò dei vari iperparametri utilizzati per la U-Net e le motivazioni che mi hanno portato ad utilizzarli e a modificarli e infine lo stesso avverrà per i diversi parametri e kernel dell'algoritmo di divisione delle tracce.

4.1 Creazione Dataset

Il primo step per la creazione di un Dataset è stato prendere un campione di immagini partendo dagli eventi forniti dal team DubTo-PAINUC. Sono state scelte randomicamente 1000 immagini di cui 665 *Left* e 335 *Right*, costituenti l'input principale del sistema. Ogni immagine *Right* è sempre associata ad una *Left*.
Le immagini ricevute erano in bitmap con una profondità di 12 bit per pixel e una risoluzione di 658×517. Ho scalato ogni pixel da 12 bit a 8 bit e quindi avere una profondità di 256 livelli di grigio.
Successivamente, si è avuto bisogno di un *Ground Truth* per poter confrontare e valutare i risultati ottenuti sia dalla rete convoluzionale che dall'algoritmo di divisione tracce, quindi sono state necessarie delle annotazioni che localizzassero le tracce nelle immagini. A questo proposito si è utilizzato **Computer Vision Annotation Tool (CVAT)**[37].

4.1.1 Annotazione con CVAT

CVAT è un tool open-source di annotazione di immagini e video per la Computer Vision. Il suo funzionamento è abbastanza semplice. Si crea un progetto e vi si caricano gli elementi da annotare. In questo caso, ho creato il progetto "Annotazione tracce CERN-JINR" e ho caricato le 1000 immagini da annotare. Ognuna di esse rappresenta un *job* da compiere all'interno dello stesso progetto. Quando un'immagine viene annotata completamente, il job risulta compiuto.

Dopo diverse valutazioni, si è ritenuto opportuno annotare ogni traccia con tre punti della stessa classe, una scelta dovuta a due motivazioni principali:

- Essendo le tracce rappresentabili da archi di circonferenza, con tre punti distinti (non allineati) diventa abbastanza semplice poterne estrarre le coordinate e generare tali archi.
- Per ogni immagine vi sono, in media, da una a cinque tracce. Moltiplicando questa quantità per il numero di immagini, il numero di tracce annotate sono circa tremila. Quindi, un tipo di annotazione più approfondita (ad esempio, un'annotazione dell'area delle tracce) avrebbe comportato un ingente impiego di tempo.

Considerato che per annotare tali immagini si necessitava di una conoscenza approfondita degli eventi fisici aventi luogo all'interno della SSSC, l'annotazione è stata svolta da elementi del team DubTo-PAINUC.

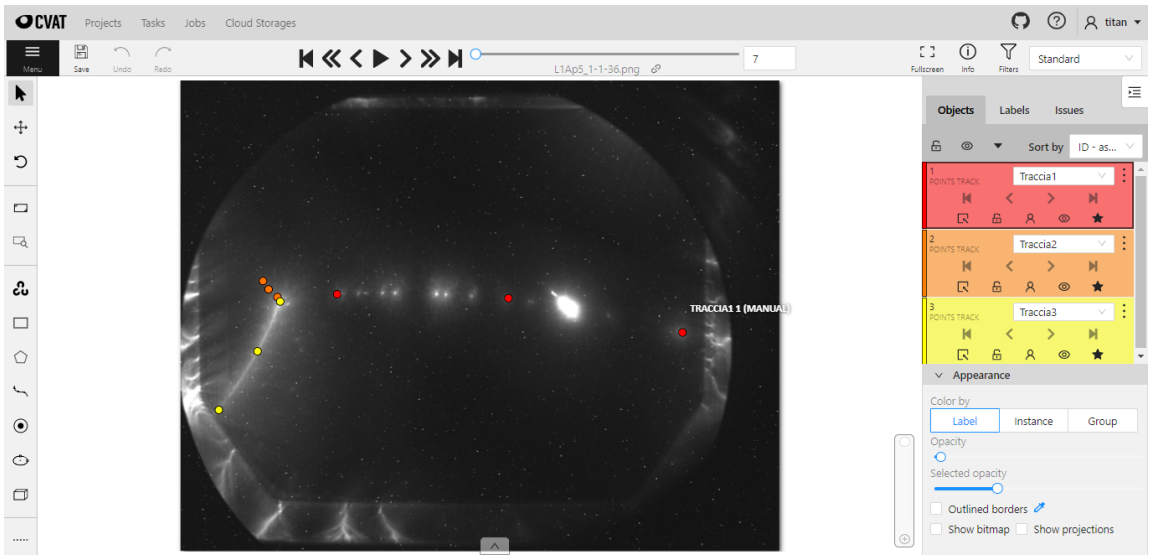


Figura 4.1. Esempio di annotazione di un evento con CVAT.

Come si può vedere in Figura 4.1, nell'immagine dell'evento ci sono tre tracce, le quali vengono annotate con classi differenti (Traccia1, Traccia2, Traccia3). Per ogni classe, vengono utilizzati tre punti che indicano l'inizio e la fine della traccia e un punto intermedio per poter descrivere al meglio la traiettoria.
Ultimata la fase di annotazione, si è potuto accedere all'intero progetto. Il tool CVAT salva le annotazioni effettuate in un file XML che presenta il campo `<annotations>`, senza alcun attributo, al cui interno vi sono 1000 campi `<image>`: uno per ogni immagine del Dataset. Ogni campo `<image>` presenta gli attributi `id`, `name`, `width` e `height` i quali indicano rispettivamente l'id dell'immagine da 0 a 999 (nel nostro caso), il nome dell'immagine, la larghezza e l'altezza. Questi ultimi due attributi sono

identici per tutte le immagini ,la cui risoluzione è 658×517. A sua volta, all'interno di ogni campo `<image>` vi sono i campi `<points>`, uno per ogni classe (traccia) annotata nell'immagine, quindi, una ogni tre punti per traccia. Gli attributi presenti sono `label`, `occluded`, `source`, `points` e `z_order`, i quali determinano, rispettivamente l'etichetta a cui appartengono i tre punti (Traccia1, Traccia2, ecc.), se i punti sono occlusi o meno (0 o 1), se la fonte dei punti è automatica o manuale, le coordinate x e y dei tre punti descriventi la traiettoria della traccia e la presenza dell'asse Z.
Ecco un esempio dei campi appena descritti:

```
<annotations>
  <image id="0" name="L1Ap5_1-1-109.png" width="658" height="517">
    <points label="Traccia2" occluded="0" source="manual"
      points="411.65,248.47;312.63,149.45;223.32,36.45" z_order="0">
    </points>
    <points label="Traccia3" occluded="0" source="manual"
      points="384.92,256.98;339.97,263.66;406.79,249.08" z_order="0">
    </points>
  </image>
  ...
<\annotations>
```

Si può notare come le coordinate dei punti siano divise tra loro da un punto e virgola, mentre la divisione tra coordinata x e y di uno stesso punto, da una virgola.

4.1.2 Generazione del Ground Truth

Il passo successivo è stato quello di creare lo script per l'estrazione dei punti dal file XML dai quali generare le tracce. Tale algoritmo lavora per ogni campo `<image>` del file e, all'interno di ognuno di questi campi, estrae le informazioni di ogni campo `<points>`.
Quindi, la prima cosa da fare è stata quella di estrarre i punti di ogni traiettoria annotata. Tale task è stato abbastanza semplice. In alcuni casi, però, ho riscontrato degli errori come l'annotazione di più di tre punti per traccia o meno di tre punti; nel primo caso è stato cancellato uno dei due punti centrali, nel secondo caso i punti annotati non sono stati presi in considerazione e quindi la traccia è stata eliminata.
Il secondo task è stato sicuramente più complesso del primo: ho creato un piccolo script per la creazione delle maschere di segmentazione sia per il task di Semantic Segmentation che per l'intero sistema (task di Instance Segmentation). Per semplicità dividerò l'algoritmo creato in più punti:

1. Ho controllato che i tre punti estratti, che da adesso chiameremo p_1, p_2, p_3, non fossero allineati tra loro calcolando l'area A del triangolo generato dai tre punti. Se $A = 0$ i punti non sono allineati e quindi descrivono un arco di circonferenza, in caso contrario descrivono un segmento.

2. Ci sono, quindi, due possibilità:

(a) Se $A \neq 0$, si cerca quali, dei tre punti, siano i due estremi e infine si disegna la maschera della traccia, ovvero il segmento, con il modulo `matplotlib.lines.Line2D()`[38].

(b) Se $A \neq 0$, vanno calcolate le coordinate del centro x_C e y_C e il raggio r della circonferenza descritta dai tre punti con la seguente funzione:

```
def define_circle(p1, p2, p3):
    temp = p2[0]*p2[0]+p2[1]*p2[1]
    bc = (p1[0]*p1[0]+p1[1]*p1[1]-temp)/2
    cd = (temp-p3[0]*p3[0]-p3[1]*p3[1])/2
    det = (p1[0]-p2[0])*(p2[1]-p3[1])-(p2[0]-p3[0])*
          (p1[1]-p2[1])

    cx = (bc*(p2[1]-p3[1])-cd*(p1[1]-p2[1]))/det
    cy = ((p1[0]-p2[0])*cd-(p2[0]-p3[0])*bc)/det
    radius = np.sqrt((cx-p1[0])**2+(cy-p1[1])**2)
return ((cx, cy), radius)
```

`p1[0]` e `p1[1]` rappresentano, rispettivamente, le coordinate x e y del punto $p1$ (lo stesso vale per $p2$ e $p3$), mentre `np.sqrt()` è la radice quadrata. La funzione mostrata traduce in codice la formula seguente:

$$bc = \frac{x_1^2 + y_1^2 - x_2^2 - y_2^2}{2} \tag{4.1}$$

$$cd = \frac{x_2^2 + y_2^2 - x_3^2 - y_3^2}{2} \tag{4.2}$$

$$det = (x_1 - x_2)(y_2 - y_3) - (x_2 - x_3)(y_1 - y_2) \tag{4.3}$$

$$x_C = \frac{bc(y_2 - y_3) - cd(y_1 - y_2)}{det} \tag{4.4}$$

$$y_C = \frac{cd(x_1 - x_2) - bc(x_2 - x_3)}{det} \tag{4.5}$$

$$r = \sqrt{(x_C - x_1)^2 + (y_C - y1)^2} \tag{4.6}$$

Successivamente, cerco i due estremi dell'arco di circonferenza da costruire. Per raggiungere tale obiettivo ho percorso la circonferenza generata con un angolo di campionamento $\theta_c = 0.0001$. Quindi, memorizzo il primo e il terzo punto trovato. Pertanto, dati i due estremi, mi calcolo l'angolo iniziale a partire dall'asse x in senso orario (dal quarto al primo quadrante) (ϕ_0) e l'angolo (ϕ) dell'arco (entrambi in gradi sessagesimali). Infine, disegno la maschera della traccia (arco di circonferenza) con il modulo `matplotlib.patches.Arc()`[39].

3. L'algoritmo si conclude con la maschera creata che viene ingrandita utilizzando il modulo `cv2.dilate()` con un kernel ellittico 18×18 per avere delle tracce abbastanza larghe (in media una larghezza di 15 pixel per traccia) per poter lavorare al meglio sia per l'allenamento della rete neurale convoluzionale, che per dividere le tracce con l'algoritmo.

Dopo aver generato le maschere delle tracce, bisogna infine creare il Ground Truth.

4.1.3 Due Ground Truth

Come già esposto nel capitolo Metodi, il nostro sistema è diviso in due parti fondamentali più una terza per lo studio dell'intensità. Le due parti principali (Generazione delle maschere e Divisione delle tracce) hanno bisogno di due differenti tipi di Ground Truth per poterne valutare le prestazioni e quindi calcolare delle metriche adatte.
Per la prima parte sono state salvate, per ogni evento, le maschere di tutte le tracce nella stessa immagine con background nullo (task di Semantic Segmentation); di conseguenza, per ogni immagine di un evento vi è la propria maschera di segmentazione di tutte le traiettorie presenti. Per la seconda parte, invece, ho salvato, per ogni evento, tante immagini quante sono le tracce da rilevare (task di Instance Segmentation).
Con la Figura 4.2 mostro un esempio dei due Ground Truth creati per un solo evento. Infatti, possiamo notare in basso a sinistra il Ground Truth per la Semantic Segmentation (maschera di tutte le traiettorie), mentre a destra vi sono due immagini del Ground Truth per l'intero sistema.

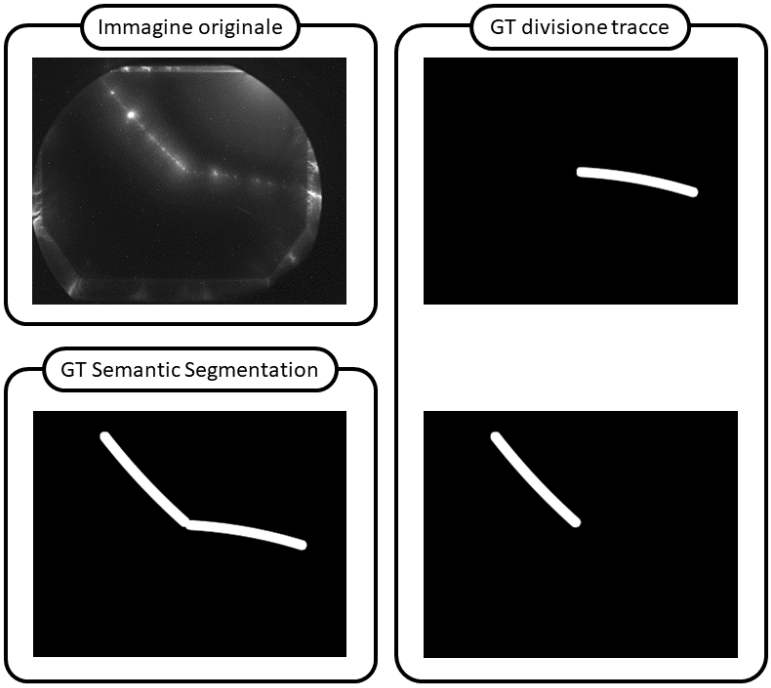


Figura 4.2. Esempio di evento fotografato e dei due Ground Truth generati.

4.1.4 Split e Preprocessing

Il dataset creato, quindi, presenta 1000 eventi e complessivamente 3049 traiettorie: la distribuzione del numero di tracce nel dataset ci indica una media di 3 traiettorie per evento e una deviazione standard di $1,229$.
Dopo aver creato il dataset (Input + Ground Truth), l'ultimo passo è stato quello di dividerlo in *Train*, *Validation* e *Test*. Ho deciso di fare uno split $8:1:1$, quindi l'80% del dataset viene destinato al training della U-Net, un 10% per il validation e il restante 10% per il test set.
Di conseguenza, siccome il dataset è formato da 1000 eventi, 800 eventi sono riservati al training set, 100 al validation set e altri 100 al test set. Tale divisione non è avvenuta casualmente: come accennato in precedenza, gli eventi a mia disposizione sono 665 *Left* e 335 *Right*, quindi risulta un rapporto del $1,98$ tra *Left* e *Right*; il mio obiettivo è stato quello di poter mantenere questo rapporto all'interno di ogni set per avere una coerenza tra gli esperimenti e i risultati ottenuti. Pertanto, gli 800 eventi del training set sono formati da 533 eventi *Left* e 267 *Right*, mentre i 100 eventi, sia del validation set che del test set, sono composti da 66 eventi *Left* e 34 *Right*.
Gli eventi all'interno di ogni set sono stati, quindi, scelti randomicamente ma facendo in modo che ogni evento *Right* avesse il proprio evento *Left* all'interno dello stesso set.
Ai fini di poter studiare il comportamento del sistema per diverse dimensioni, ho ridimensionato il dataset a seconda del tipo di esperimento di interesse. Ad esempio, volendo fare un esperimento con dimensionalità 128×128, ridimensiono l'input e il Ground Truth per l'allenamento della rete a 128×128 mentre il Ground Truth per l'algoritmo di divisione tracce (quindi per l'intero sistema) a 128×101. Per praticità, da ora in poi, quando parlerò delle dimensioni del dataset, mi riferirò ad esse come *Dataset 128*, *Dataset 256* e *Dataset 512*.

4.2 Allenamento U-Net

L'allenamento della rete U-Net ha riguardato tutti i blocchi che la componevano, a partire dalla VGG-16, preallenata su ImageNet, usata come backbone e quindi encoder del modello.
Nei seguenti sottoparagrafi parlerò del sistema su cui ho potuto allenare tale modello, della preparazione dei dati per l'allenamento e degli esperimenti effettuati.

4.2.1 HPC e Slurm

Prima di poter allenare la rete interamente e sull'intero dataset, ho fatto alcune prove utilizzando *Google Colab*[40]. Dopo aver risolto tutti i possibili errori di compilazione, sono passato all'utilizzo del sistema *HPC*[41] del Politecnico di Torino.
L'iniziativa di supercalcolo HPC@POLITO, gestita dal DAUIN (Dipartimento di Automatica e Informatica del Politecnico di Torino), fornisce risorse di calcolo ad alte prestazioni per attività di ricerca accademica e didattica. Il centro di calcolo è dotato di tre cluster: *CASPER*, *HACTAR* e *LEGION*, questi tre sistemi sono integrati tra

loro e condividono il sistema di autenticazione e i sistemi di storage. Per gestire i job nei vari cluster, gli utenti possono sottometterli per mezzo dell'*SLURM* (Simple Linux Utility for Resource Management)[42], lo scheduler open-source utilizzato dai sistemi.
Io ho deciso di utilizzare il cluster LEGION per motivi di calcolo, infatti dei tre è quello che garantisce più TERAFLOPS. Per accedervi si utilizza SSH e bisogna connettersi al nodo di login in cui verrà inserito il proprio username e la password decisa in fase di iscrizione al sistema. Una volta all'interno di LEGION, il sistema dà a disposizione uno spazio di archiviazione dove poter salvare i modelli allenati, il dataset ed eventuali checkpoints e una bash da cui poter far partire i job e gestirli grazie a SLURM.
Dopo aver caricato il dataset, il codice di allenamento *train.py* e essersi assicurati che sul sistema siano installate tutte le librerie utilizzate, per far partire un job su SLURM bisogna prima creare un file *script.sbatch* dove inserire la configurazione del job e i comandi da far partire.
Un esempio di questo file:

```
#!/bin/bash
#SBATCH --job-name=Sem256
#SBATCH --mail-type=ALL
#SBATCH --mail-user=ciro.maiello@studenti.polito.it
#SBATCH --partition=cuda
#SBATCH --time=120:00:00
#SBATCH --nodes=1
#SBATCH --gres=gpu:1

python train.py
```

Si noti che i comandi seguiti da `#SBATCH` siano informazioni per SLURM del job in questione, mentre dopo viene mandato il comando `python train.py` per far partire il compilatore di Python e quindi l'allenamento della rete.
Per mettere nella coda dello scheduler il job, si utilizza il comando `sbatch` seguito dal nome del file di configurazione del job. Così, al job viene stabilito un identificativo e messo in coda. Esso potrà partire quando ci sarà disponibilità sul nodo messo a disposizione. Per controllare la coda si utilizza il comando `squeue`, con la possibilità di poter utilizzare anche dei filtri per visionare solo parte della coda. Quando il job utilizza le risorse di calcolo, SLURM crea un file `slurm-298293.out` (il numero seguente `slurm-` rappresenta l'identificativo del job) al cui interno è presente l'output del job.

4.2.2 Data Augmentation

Per avere una maggiore generalizzazione dei risultati, al dataset è stata applicata una *augmentation*[43], quindi, parte del dataset è stato modificato utilizzando tecniche di Computer Vision.

Per effettuare tale operazione è stato utilizzato il modulo `albumentations`[44]. Nello specifico, ho deciso di applicare una rotazione casuale dell'immagine da 0° a 90° e un flip orizzontale.
Di seguito il pezzo di codice che si occupa dell'augmentation:

```
def get_training_augmentation():
    train_transform = [
        A.HorizontalFlip(p=P_AUGMENTATION),
        A.Rotate(limit=[0, 90], p=P_AUGMENTATION)
    ]
    return A.Compose(train_transform)
```

`P_AUGMENTATION` rappresenta la probabilità con cui può avvenire la trasformazione effettuata sul dataset. Tale valore è un iperparametro che ho modificato durante i diversi esperimenti realizzati.

4.2.3 Esperimenti

Come già esposto precedentemente, affinché si potessero valutare al meglio le performance della rete per questo task, tutti gli allenamenti sono stati eseguiti per le tre dimensioni in cui è stato creato il dataset e modificando opportunamente gli iperparametri a seconda dei risultati delle metriche. La batch size utilizzata per le tre risoluzioni è di 16 per il *Dataset 512*, 32 per il *Dataset 256* e 64 per il *Dataset 128*.
Il primo allenamento, per tutte e tre le dimensioni, è stato effettuato con gli iperparametri descritti nella seguente Tabella:

Hyperparameter	**Value**
Learning Rate	0.001
Optimizer	Adam
Epoche	500
Dropout	False
P. Augmentation	0.25

Tabella 4.1. Setup iniziale degli iperparametri della rete

Succesivamente, ho modificato la percentuale di augmentation passando a 0.33 oppure 0.5, ho cambiato il learning rate per puntare ad una migliore generalizzazione e ho provato ad utilizzare un Dropout per l'encoder con percentuale del 50%.
Parlerò più approfonditamente dei risultati raggiunti e della miglior combinazione di iperparametri secondo i miei esperimenti nel capitolo "Risultati", ma, complessivamente, i risultati migliori si sono raggiunti con il *Dataset 512*.
I modelli delle migliori configurazioni di iperparametri per tutte e tre le dimensioni sono stati poi utilizzati per generare l'input per il secondo step del sistema e quindi per la parametrizzazione dell'algoritmo di divisione tracce.

4.3 Parametrizzazione algoritmo

L'algoritmo di divisione delle tracce presenta dei parametri che hanno richiesto modifiche e cambiamenti durante gli esperimenti per arrivare alla migliore configurazione e, quindi, ottenere i risultati migliori per le metriche calcolate. Anche in questo caso gli esperimenti sono stati condotti per tutte e tre le risoluzioni del dataset, ma, rispetto agli iperparametri dello step precedente, i valori dei parametri sono estremamente differenti tra una dimensione e l'altra. Ciò è dovuto al fatto che le trasformazioni effettuate durante l'algoritmo sono strettamente legate a spessore e grandezza delle tracce le quali sono, come si può immaginare, differenti tra le diverse dimensioni.
Gli esperimenti sono stati tutti effettuati sul validation set. Di conseguenza, l'algoritmo analizza i 100 eventi e per ogni evento le tracce presenti.

4.3.1 Elenco dei parametri

Segue un elenco dei parametri utilizzati con una breve descrizione della loro funzione all'interno dell'algoritmo:

- **Soglia della thresholding**: parametro che serve per indicare i valori dei pixel da alterare. I valori al di sotto della soglia diventano 0, mentre i valori al di sopra della soglia 255. Tale soglia è la stessa per il *Dataset 256* e il *Dataset 512* poiché il valore scelto è risultato quello ottimale per le performance del sistema.

- **Kernel della top-hat**: parametro il cui utilizzo è fondamentale per la resa della trasformata top-hat. Il kernel l'ho scelto circolare affinché potesse adattarsi al meglio con la forma dello scontro delle tracce.

- **Kernel per le trasformazioni morfologiche sulle tracce**: parametro utilizzato come kernel per le operazioni di opening, dilation o erosion effettuate direttamente sulle maschere delle tracce.

- **Connettività CCL**: valore che indica quanti pixel vanno considerati intorno al pixel di riferimento per compiere la connessione delle regioni.

- **Kernel della preparazione alla Watershed**: parametro indispensabile per la buona riuscita della trasformazione Watershed. Grazie a questo kernel ci assicuriamo di coprire un'area abbastanza grande del background per poter indicare la regione dove la Watershed può lavorare per recuperare i valori dei pixel azzerati dalle operazioni precedenti.

- **Deviazione Standard matrice Hessiana**: valore da assegnare alla deviazione standard del kernel gaussiano per il calcolo della matrice Hessiana della maschera. Tale parametro è uguale per tutte e tre le dimensioni, poiché con dei valori diversi si è avuta difficoltà a calcolare in modo ottimale, per i nostri obiettivi, la matrice Hessiana da cui estrarre gli autovalori λ_1 e λ_2.

- **Soglia di estrazione dei ridge**: parametro scelto per indicare il valore da cui estrarre i ridge dagli autovalori della matrice Hessiana. In particolare, i ridge si trovano al di sotto di tale soglia.
- **Kernel della dilation dei ridge**: parametro che indica il kernel da utilizzare per poter ingrandire i ridge prima di essere sottratti alla maschera.
- **Kernel dell'ultima dilation**: kernel utilizzato per l'ultima dilation prima di calcolare le metriche di valutazione delle prestazioni della rilevazione delle tracce.

Capitolo 5

Risultati

I risultati ottenuti a seguito degli esperimenti brevemente esposti nel capitolo "Esperimenti" sono presentati in questo capitolo seguendo le diverse configurazioni sia di iperparametri (Generazione maschere) che di parametri (Algoritmo di divisione tracce). Mostrerò anche degli output visivi per presentare al meglio le performance ottenute dal sistema.
Ai fini di una più semplice fruizione dei risultati, dividerò gli stessi rispetto alle tre parti del sistema facendo anche confronti tra le dimensioni e i diversi setup sia degli iperparametri (allenamento della rete) che dei parametri (algoritmo di divisione delle tracce).

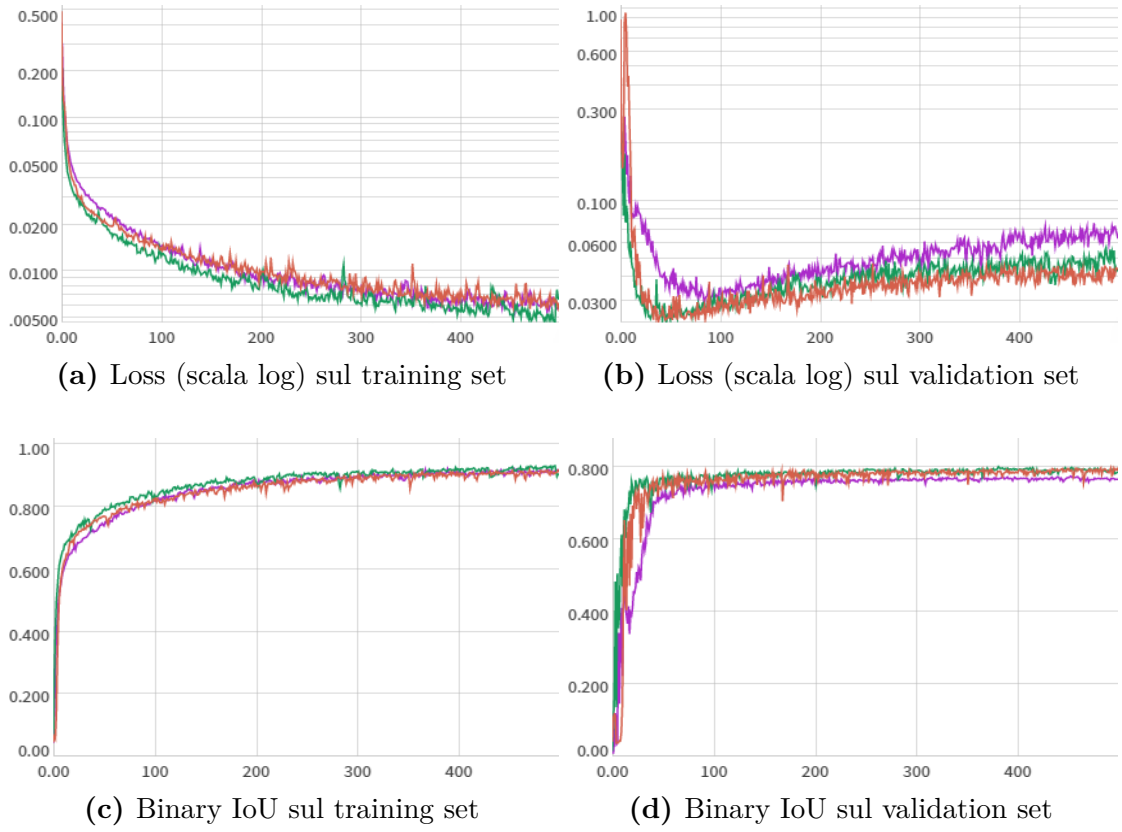
5.1 Allenamento U-Net

I risultati dell'allenamento della rete neurale convoluzionale, verranno divisi in due sottosezioni: "Risultati preliminari" e "Risultati finali". Tale divisione è necessaria per distinguere i risultati raggiunti durante gli allenamenti per arrivare al setup migliore, dai risultati con la configurazione ottimale degli iperparametri. Inoltre, i modelli che hanno mostrato i migliori risultati (i tre modelli della sottosezione "Risultati finali"), sono quelli che poi ho utilizzato per lavorare sull'algoritmo di divisione delle tracce.

5.1.1 Risultati preliminari

Il primo allenamento effettuato è stato fatto con il setup iniziale di iperparametri descritto in Tabella 4.1. I risultati ottenuti per i tre dataset (*Dataset 128*, *Dataset 256* e *Dataset 512*) sono mostrati in Figura 5.1. Per una maggiore comprensione dei dati e per rendere il confronto più semplice, ho deciso di mostrare i valori della loss in scala logaritmica. In Tabella 5.1, invece, mostro i risultati finali della Binary IoU per questi allenamenti.
Come si può vedere, i modelli, con questa configurazione, tendono ad essere affetti da *overfitting* sul validation set. Ciò sta ad indicarci che i modelli generalizzino poco e sono inclini ad imparare il training set. Questa intuizione ci viene suggerita anche

dal fatto che la Binary Iou risulta in tutti e tre i casi superiore sul training set (circa 0.9) rispetto al validation set (circa 0.8 o inferiore).



(a) Loss (scala log) sul training set

(b) Loss (scala log) sul validation set

(c) Binary IoU sul training set

(d) Binary IoU sul validation set

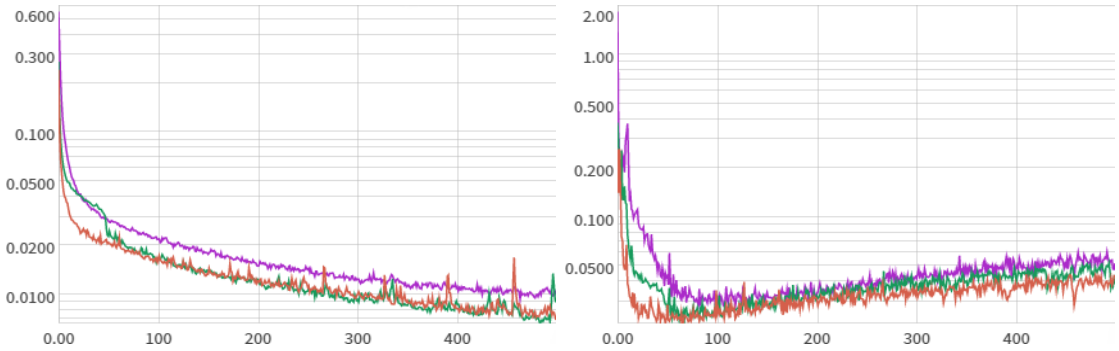
Figura 5.1. Curve delle loss e metriche di valutazione durante il primo allenamento per i tre dataset (viola: *Dataset 128*, verde: *Dataset 256*, arancione: *Dataset 512*)

Dim. Dataset	Train set	Val set
Dataset 128	0.92	0.76
Dataset 256	0.9	0.787
Dataset 512	0.91	0.79

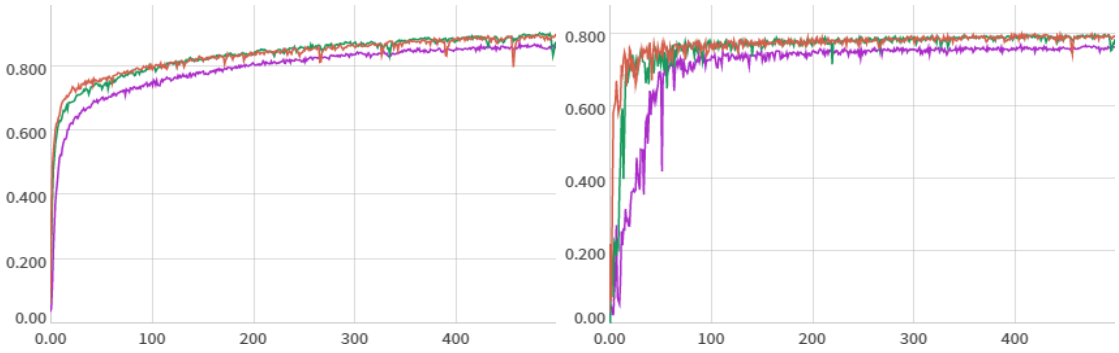
Tabella 5.1. Risultati ultima epoca Binary IoU per il training e per il validation set con la configurazione in Tabella 4.1

Vi è comunque un aspetto da sottolineare: i valori mostrati dalla loss non sono direttamente collegati con la metrica da noi scelta per calcolare le performance della rete neurale. Infatti, la loss utilizzata, la Binary Cross-Entropy, viene calcolata sull'intero output rispetto al Ground Truth. Lo stesso non si può dire della Binary IoU che, come sottolineato nel sezione "Loss, Ottimizzatore e Metrica della Semantic Segmentation", calcola le performance del modello solo sulla maschera della traccia

generata dalla rete, rispetto alla maschera del Ground Truth, e non sull'intera immagine. Quindi, è per questo motivo che, nonostante i modelli generino overfitting sul validation set, tale complicazione non causi un crollo netto delle performance.
Si può notare che il modello allenato sul *Dataset 128* è, dei tre, il più affetto da overfitting e anche quello con le performance più basse in termini di Binary IoU. Invece, gli allenamenti con le altre due dimensionalità sono molto simili in termini di performance sia sul training che sul validation set. La Binary Cross-Entropy pare avere un minore overfitting sul validation set con le immagini 512×512.
Successivamente, per cercare di abbassare la curva della Binary Cross-Entropy sul validation set e migliorare anche le performance, ho modificato la percentuale di augmentation del dataset per cercare di avere una maggiore generalizzazione dei modelli. Di conseguenza, tali esperimenti li ho effettuati sui tre dataset e cambiando la percentuale di augmentation per due volte per ogni dimensione. Quindi, ho modificato la percentuale da 0.25 a 0.33 e, successivamente, anche a 0.5.



(a) Loss (scala log) sul training set **(b)** Loss (scala log) sul validation set



(c) Binary IoU sul training set **(d)** Binary IoU sul validation set

Figura 5.2. Curve delle loss e metriche di valutazione durante l'allenamento con P. Augmentation = 0.5 per i tre dataset (viola: *Dataset 128*, verde: *Dataset 256*, arancione: *Dataset 512*)

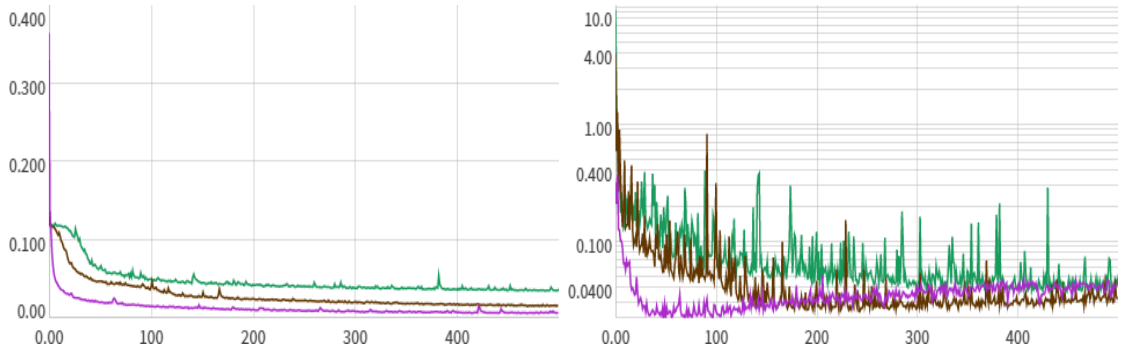
I risultati in Figura 5.2 mostrano che aumentando la percentuale di augmentation a 0.5 l'overfitting diminuisce leggermente rispetto ai casi precedenti, ma le performance

sul validation sono non troppo differenti. Ciò si può notare dalla Tabella 5.2 (risultati sul validation set inferiori allo 0.8).

Dim. Dataset	Train set	Val set
Dataset 128	0.86	0.76
Dataset 256	0.876	0.784
Dataset 512	0.9	0.79

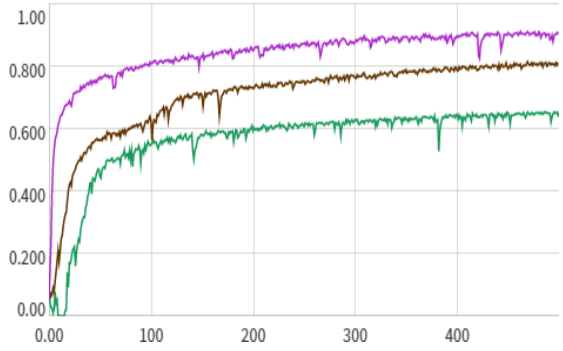
Tabella 5.2. Risultati ultima epoca Binary IoU per il training e per il validation set con la P. Augmentation = 0.5

L'idea successiva, quindi, è stata quella di cambiare la learning rate dell'ottimizzatore. Infatti, l'overfitting che cerchiamo di eliminare potrebbe essere causato da una difficoltà da parte del modello di raggiungere il minimo locale. Quindi, ho effettuato degli esperimenti aumentando di dieci volte la learning rate. Di conseguenza, essa è passata da 0.001 a 0.01 fino a 0.1 come si può vedere nei risultati in Figura 5.3 per il *Dataset 512* con percentuale di augmentation fissa allo 0.33.

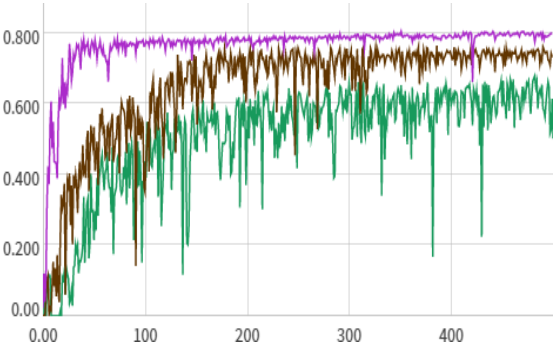


(a) Loss (scala log) sul training set

(b) Loss (scala log) sul validation set



(c) Binary IoU sul training set



(d) Binary IoU sul validation set

Figura 5.3. Curve delle loss e metriche di valutazione durante gli allenamenti con il *Dataset 512* e diverse learning rate (viola: 0.001, marrone: 0.01, verde: 0.1)

Come ci si poteva aspettare, l'overfitting sul validation diminuisce aumentando la learning rate, però tali cambiamenti comportano anche un peggioramento delle performance del modello sia sul training set (Figura 5.3c) che sul validation set (Figura 5.3d). Di conseguenza, il modello con learning rate a 0.001 probabilmente aveva già raggiunto un minimo locale ottimale. Il crollo di prestazioni si può notare facilmente anche dalla Tabella 5.3: con una learning rate $LR = 0.1$ abbiamo una Binary IoU sul validation set che sfiora lo 0.6 e che, quindi, è di 0.2 più bassa del valore della metrica con $LR = 0.001$.

Learning rate	Train set	Val set
0.001	0.91	0.8
0.01	0.81	0.73
0.1	0.677	0.596

Tabella 5.3. Risultati ultima epoca Binary IoU per il training e per il validation set per il *Dataset 512*

L'ultimo cambiamento che ho deciso di effettuare per provare a diminuire l'overfitting è stato quello di aggiungere del Dropout alla rete neurale. In particolare, l'ho inserito per ogni blocco dell'encoder della U-Net.
Con quest'ultima modifica la curva della Binary Cross-Entropy sul validation non è cambiata di molto, ma in alcuni casi le performance sono migliorate considerevolmente.

5.1.2 Risultati finali

Hyperparameter	Dataset 128	Dataset 256	Dataset 512
Learning Rate	0.001	0.001	0.001
Optimizer	Adam	Adam	Adam
Epoche	500	500	500
Dropout	True	True	False
P. Augmentation	0.25	0.33	0.33

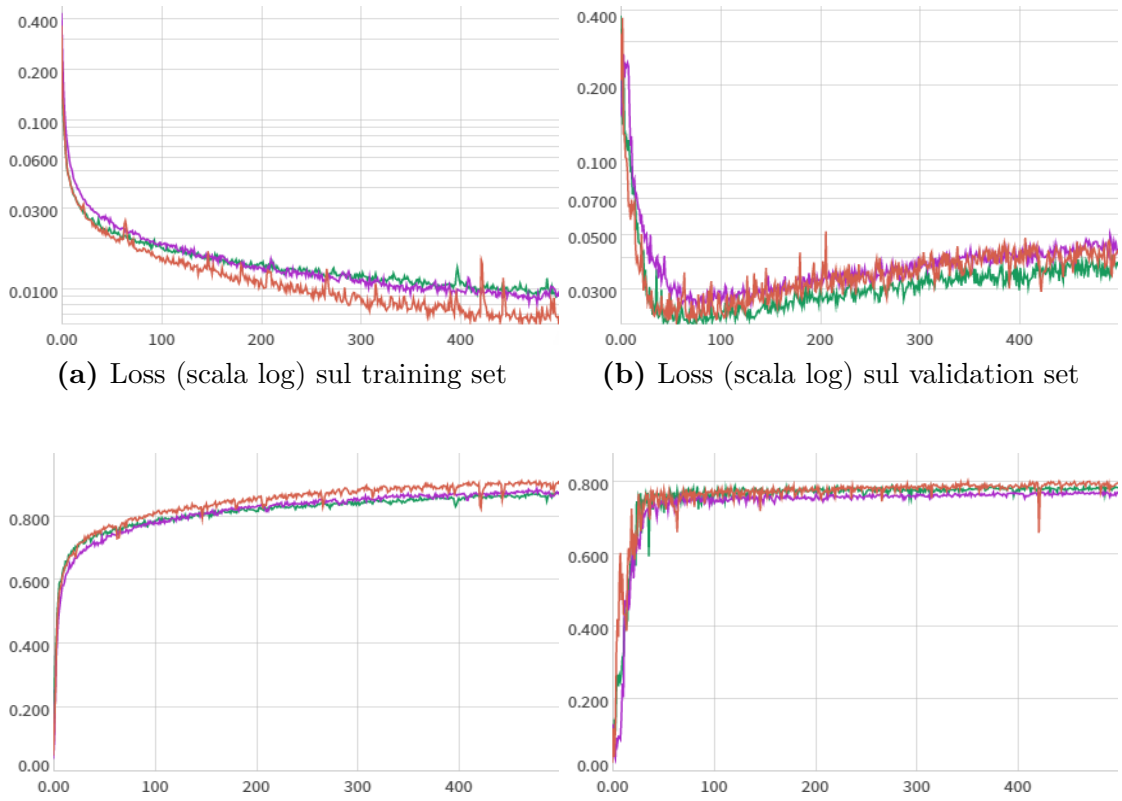
Tabella 5.4. Setup finale degli iperparametri per i tre dataset

Per ogni dimensione, infine, ho scelto il modello che mi desse i risultati migliori sia in termini di performance che per l'output mostrato. Le tre configurazioni dei tre migliori modelli sono riassunte nella Tabella 5.4. Si può notare come la learning rate con le performance migliori per questo task sia 0.001 e come il Dropout sia risultato utile per il *Dataset 128* e il *Dataset 256*.
In Figura 5.4 si possono apprezzare le differenze tra i tre modelli e come le prestazioni migliori vengano raggiunte dal modello allenato sul *Dataset 512*. Vi è ancora

dell'overfitting sul validation, ma, come sottolineato precedentemente, tale problema non va ad infinciare sulle performance dei modelli. Infatti, sia sul validation set che sul test set abbiamo i risultati delle metriche mostrati in Tabella 5.5.

Dim. Dataset	Train set	Val set	Test set
Dataset 128	0.877	0.77	0.753
Dataset 256	0.872	0.787	0.78
Dataset 512	0.91	0.8	0.77

Tabella 5.5. Risultati ultima epoca Binary IoU per il validation e per il test set



(a) Loss (scala log) sul training set

(b) Loss (scala log) sul validation set

(c) Binary IoU sul training set

(d) Binary IoU sul validation set

Figura 5.4. Curve delle loss e metriche di valutazione durante gli allenamenti con performance migliori per i tre dataset (viola: *Dataset 128*, verde: *Dataset 256*, arancione: *Dataset 512*)

Le predizioni sul test set che mostro in Figura 5.5 mostrano quanto i risultati siano visivamente soddisfacenti anche nei casi più complessi come nella quarta riga. In generale si può vedere una maggiore precisione nella segmentazione per il *Dataset 256* e il *Dataset 512*. Si rilevano, inoltre, dei casi in cui la rete vede tracce che ad occhio nudo è stato difficile notare in fase di annotazione oppure che non sono prensenti nel

Ground Truth a causa di un errore di annotazione: vi è un esempio nella terza riga dove i tre modelli segmentano una traccia che nel Ground Truth non è presente. Si possono anche osservare alcuni errori fatti dai modelli come la quarta traccia per il quinto esempio che non viene mascherata dal modello allenato sul *Dataset 256*.

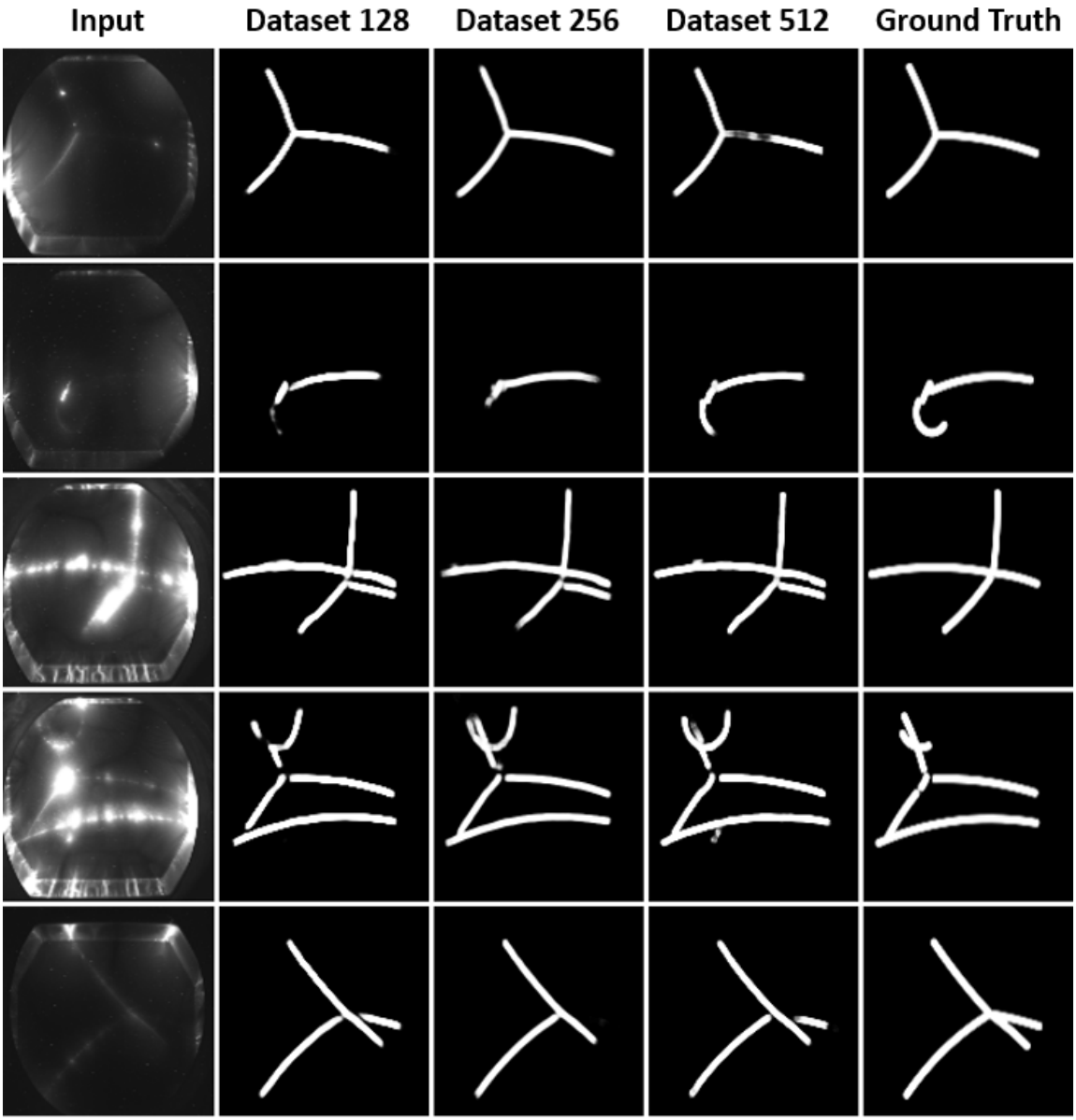


Figura 5.5. Esempi di predizioni per le tre dimensioni con confronto con GT delle maschere di segmentazione

5.2 Prestazioni del sistema complessivo

Come già precedentemente sottolineato, questa parte del sistema è stata migliorata osservando i risultati esperimento dopo esperimento. L'ottimizzazione di tale algoritmo è avvenuta gradualmente cambiando, di caso in caso, i parametri per arrivare ai migliori risultati possibili. Per avere il sistema con le performance più alte sul dataset a nostra disposizione, ho dovuto modificare i parametri circa venti volte per dimensione. Di conseguenza, parlare dei singoli esperimenti effettuati, come proposto in parte nella sezione precedente, può essere complesso. Per questo motivo, indicherò come ho trasformato i vari parametri a seconda della dimensione per arrivare all'ottimizzazione dell'algoritmo (quindi del sistema) senza mostrare tutti i risultati dei singoli esperimenti che mi hanno guidato verso la configurazione ottimale.

Parameter	**Dataset 128**	**Dataset 256**	**Dataset 512**
Threshold	10	50	50
Kernel top-hat	$6 \times 6px$ circolare	$10 \times 10px$ circolare	$19 \times 19px$ circolare
Kernel trasf.	$3 \times 3px$ ellittico	$3 \times 3px$ ellittico	$9 \times 9px$ ellittico
Connettività	8	8	8
Kernel wat.	$30 \times 30px$ ellittico	$30 \times 30px$ ellittico	$50 \times 50px$ ellittico
σ matrice Hessiana	3	3	3
Soglia ridge	25	35	75
Kernel ridge	$5 \times 5px$ ellittico	$10 \times 10px$ ellittico	$16 \times 16px$ ellittico
Kernel finale	1px	$2 \times 2px$ ellittico	$3 \times 3px$ ellittico

Tabella 5.6. Setup iniziale dei parametri per i tre dataset

Nella Tabella 5.6 sono mostrati i valori ottimali per i parametri con il dataset a disposizione e per le operazioni che vengono effettuate nell'algoritmo (sezione "Divisione delle tracce"). Si può notare che, come ci si poteva attendere, quanto più le dimensioni delle immagini del dataset sono grandi, tanto più sono grandi i valori e le dimensioni dei kernel utilizzati per le trasformazioni nell'algoritmo. Di conseguenza, ad esempio, il kernel della top-hat per il *Dataset 128* è più piccolo di quello del *Dataset 256* il quale è a sua volta di dimensioni inferiori a quello del *Dataset 512*.
Tale configurazione è stata raggiunta tenendo conto dei risultati delle metriche (sottosezione "Valutazione delle prestazioni e metriche") calcolate sul validation set. Per il calcolo delle metriche bisogna stabilire una soglia di Binary IoU. Per i risultati mostrati ho utilizzato una soglia di 0.5.
I risultati delle metriche sul validation set sono mostrate nella Tabella 5.7.

Metric	Dataset 128	Dataset 256	Dataset 512
Eventi con $FP = 0$, $FN = 0$	18/100	44/100	42/100
Eventi con $TP = 0$	15/100	2/100	0/100
TP	177/324	255/324	248/324
FP	148	98	123
Precision	177/325=0.545	255/353=0.722	248/371=0.668
Recall	177/324=0.546	255/324=0.787	248/324=0.765

Tabella 5.7. Risultati metriche val-set per i tre dataset con soglia $IoU = 0.5$.

Il validation set è formato da 100 eventi con 324 tracce. La terza e la quarta riga, i TP e gli FP, sottolineano il numero di tracce rilevate che sono state valutate come Veri Positivi e Falsi Positivi. I valori mostrati nella Tabella mostrano che, utilizzando il *Dataset 256*, per 44 eventi il sistema ha rilevato tutte e sole le tracce presenti, e nessun evento ha nemmeno una traccia rilevata. In particolare, si può notare come il sistema con le performance peggiori sia sicuramente quello costruito con il *Dataset 128*. Ciò può essere dovuto al fatto che con una dimensione di 128×101 può diventare difficile lavorare su delle maschere così piccole. La Tabella sottolinea anche come in tutti e tre i casi si ha una Recall più alta della Precision. Ciò sta a dimostrare che c'è una più alta presenza di Falsi Positivi che Falsi Negativi nelle predizioni del sistema. QUesta presenza così alta di Falsi Positivi è dovuta, la maggior parte delle volte, a tracce che vengono spezzettate in più tracce dall'algoritmo.
I risultati delle metriche sul test set (100 eventi e 291 tracce) sono visibili nella Tabella 5.8.

Metric	Dataset 128	Dataset 256	Dataset 512
Eventi con $FP = 0$, $FN = 0$	21/100	39/100	36/100
Eventi con $TP = 0$	4/100	3/100	5/100
TP	205/291	227/291	215/291
FP	136	88	124
Precision	205/341=0.601	227/315=0.721	215/339=0.634
Recall	205/291=0.704	227/291=0.78	215/291=0.739

Tabella 5.8. Risultati metriche test set per i tre dataset con soglia $IoU = 0.5$.

In questo caso il numero di tracce è inferiore, ma i risultati delle metriche sono simili al validation set. Infatti, anche con questo set di immagini il sistema con le migliori performance è quello modellato sul *Dataset 256* con 39 eventi totalmente rilevati e 3 eventi con nessuna traccia rilevata. Invece, il sistema con le performance peggiori si conferma quello con il *Dataset 128*. Anche con il test set abbiamo una Precision inferiore alla Recall.
In Figura 5.6 si può notare, come intuibile dalle metriche, che le maschere sul *Dataset*

128 sono quelle con più errori mentre pare che il sistema modellato con il *Dataset 256* sia più preciso.
Bisogna sottolineare che gli errori nella rilevazione di una traccia sono dovuti sia all'algoritmo che alla segmentazione. Ci sono casi infatti in cui la prima parte del sistema segmenta in modo errato e quindi tale errore viene trasferito all'algoritmo. Ne è un esempio la rilevazione del quinto evento alla terza colonna: vengono rilevate tre tracce e non quattro. Tale errore è dovuto alla segmentazione, infatti, in Figura 5.5 alla stessa posizione si può notare come la segmentazione per la traccia proveniente da destra sia sbagliata.
In altri casi, invece, la maschera generata della U-Net è corretta mentre l'algoritmo sbaglia, un esempio di questa casistica possiamo ritrovarla alla quinta riga per il modello con il *Dataset 512*: la maschera di segmentazione segue bene le traiettorie, mentre l'algoritmo non riesce a dividere due tracce e le considera una singola traccia.

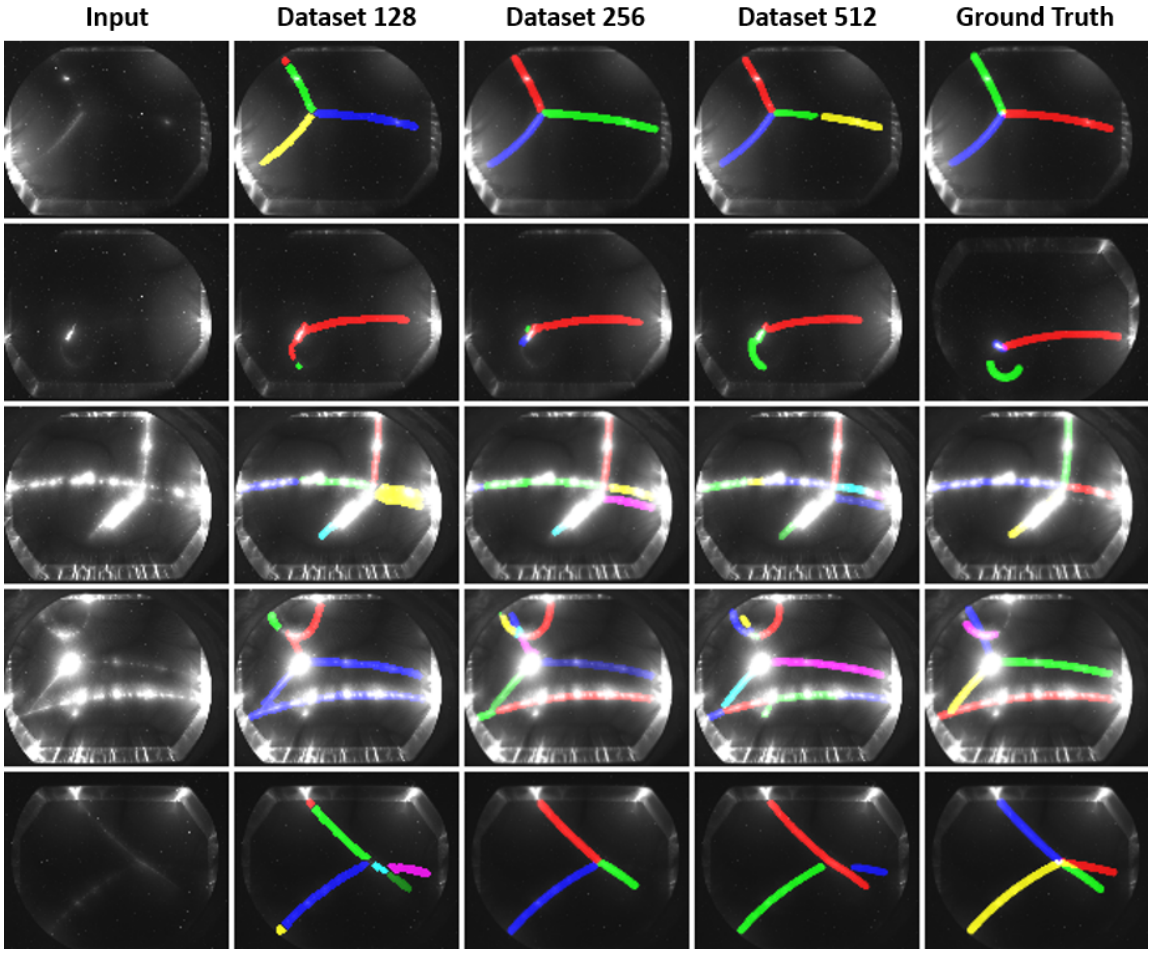


Figura 5.6. Esempi di rilevazioni per le tre dimensioni con confronto con GT delle tracce

Di seguito, nelle Figure 5.7, 5.8 e 5.9, mostro alcuni risultati totalmente corretti (tutte e sole le tracce rilevate, quindi nessun Falso Positivo o Falso Negativo). Si può

vedere come tutti i casi abbiano al più tre tracce. Il motivo principale è dovuto al fatto che tutti e tre i modelli creati hanno più difficoltà a rilevare completamente le tracce di un evento se vi sono più di tre traiettorie visibili (situazioni facilmente notabili in Figura 5.6.

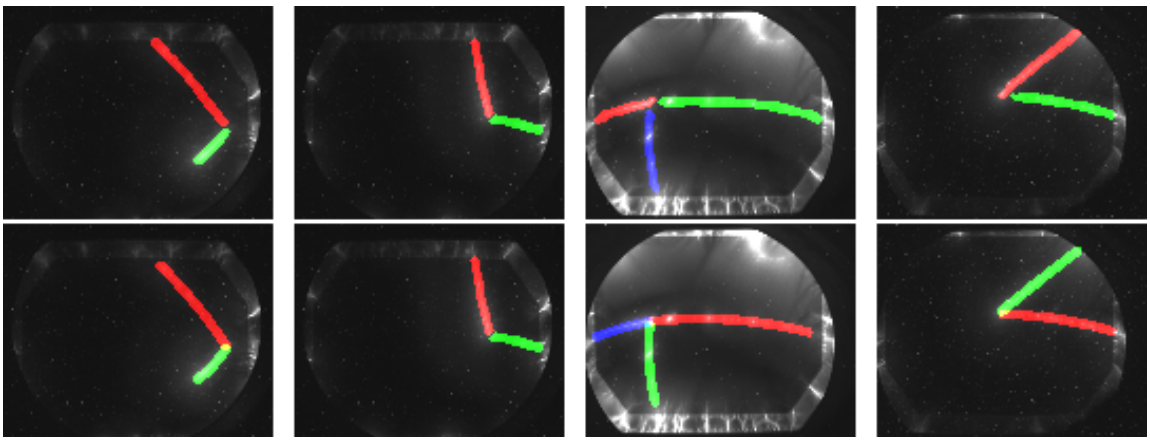


Figura 5.7. Esempi di rilevazioni totalmente corrette per il *Dataset 128* (prima riga: rilevazione; seconda riga: Ground Truth).

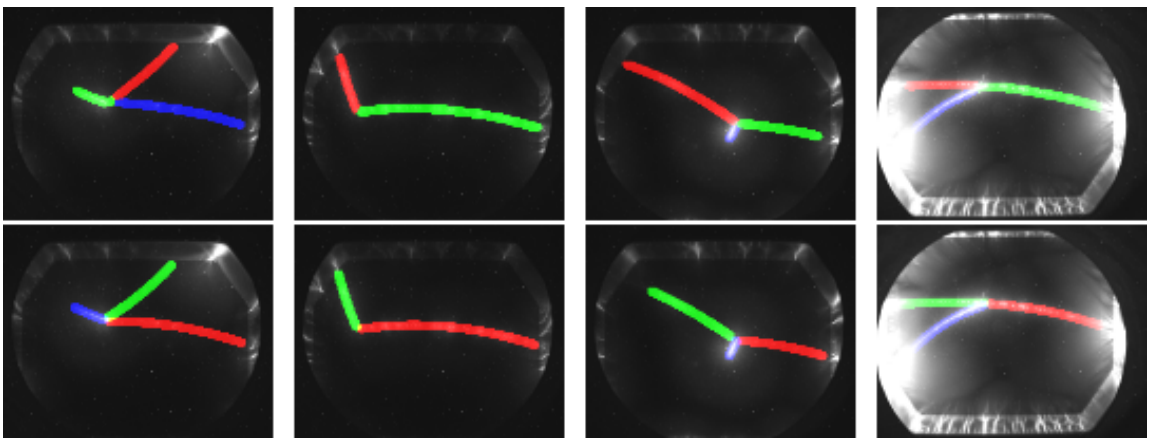


Figura 5.8. Esempi di rilevazioni totalmente corrette per il *Dataset 256* (prima riga: rilevazione; seconda riga: Ground Truth).

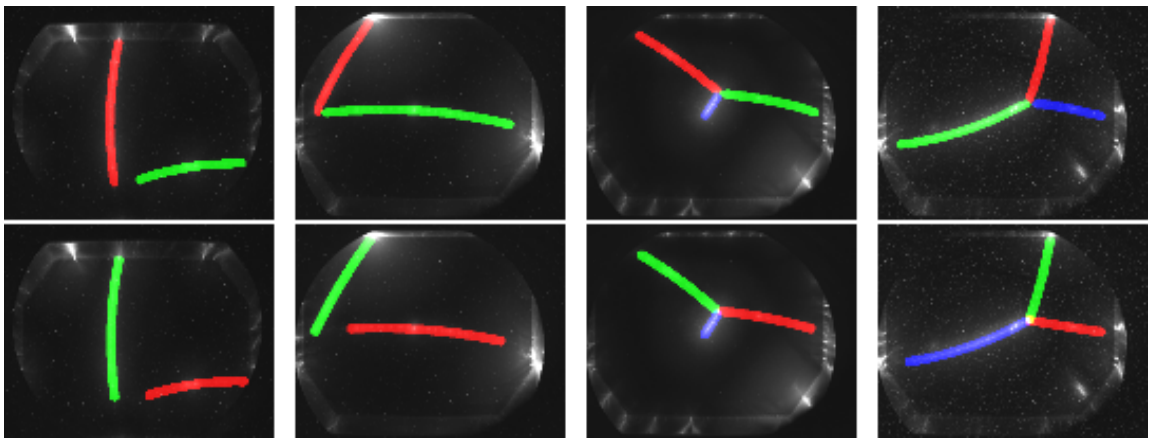


Figura 5.9. Esempi di rilevazioni totalmente corrette per il *Dataset 512* (prima riga: rilevazione; seconda riga: Ground Truth).

5.3 Risultati dello studio dell'intensità

Per lo studio dell'intensità delle traiettorie delle particelle non sono state implementate delle metriche che possano valutare le performance. In questa sezione, quindi, mi occuperò di mostrare alcuni risultati ottenuti e dei casi critici dove il sistema non ha funzionato al meglio.
In Figura 5.10, ci sono degli esempi di fitting. In particolare, si possono notare sia dei fitting avvenuti con successo (ad esempio, il primo fitting) che altri in cui la libreria `lmfit` ha riscontrato delle difficoltà (ultimo esempio). I casi in cui la funzione `fit()` ha funzionato al meglio sono sicuramente quelli dove il picco di intensità è ben più visibile rispetto agli altri valori dei pixel del campione (la differenza tra i pixel del picco e gli altri è netta). Un esempio lampante è il terzo fitting, al contrario di ciò che avviene nell'ultimo esempio: in questo caso tutti i valori sono simili, di conseguenza, il fitting risulta sbagliato. Nel quinto campione, invece, il calcolo delle curve pare sia avvenuto con successo, ma, in realtà, c'è un errore: anche se il picco di intensità è ben visibile, in realtà il valore dell'ottavo pixel è del rumore causato dalla ionizzazione dell'elio nella SSSC; quindi, la gaussiana del segnale costruita intorno a quel pixel è in realtà sbagliata (avrebbe dovuto avere, probabilmente, una media intorno al quindicesimo valore). Un altro esempio di fitting non andato a buon fine è quello nel penultimo grafico: la gaussiana associata alla traiettoria viene tagliata e lascia supporre che la traiettoria sia oltre i 15 pixel del campione; quindi, in questo caso l'errore è scaturito poiché probabilmente la larghezza della traccia non era sufficientemente grande.

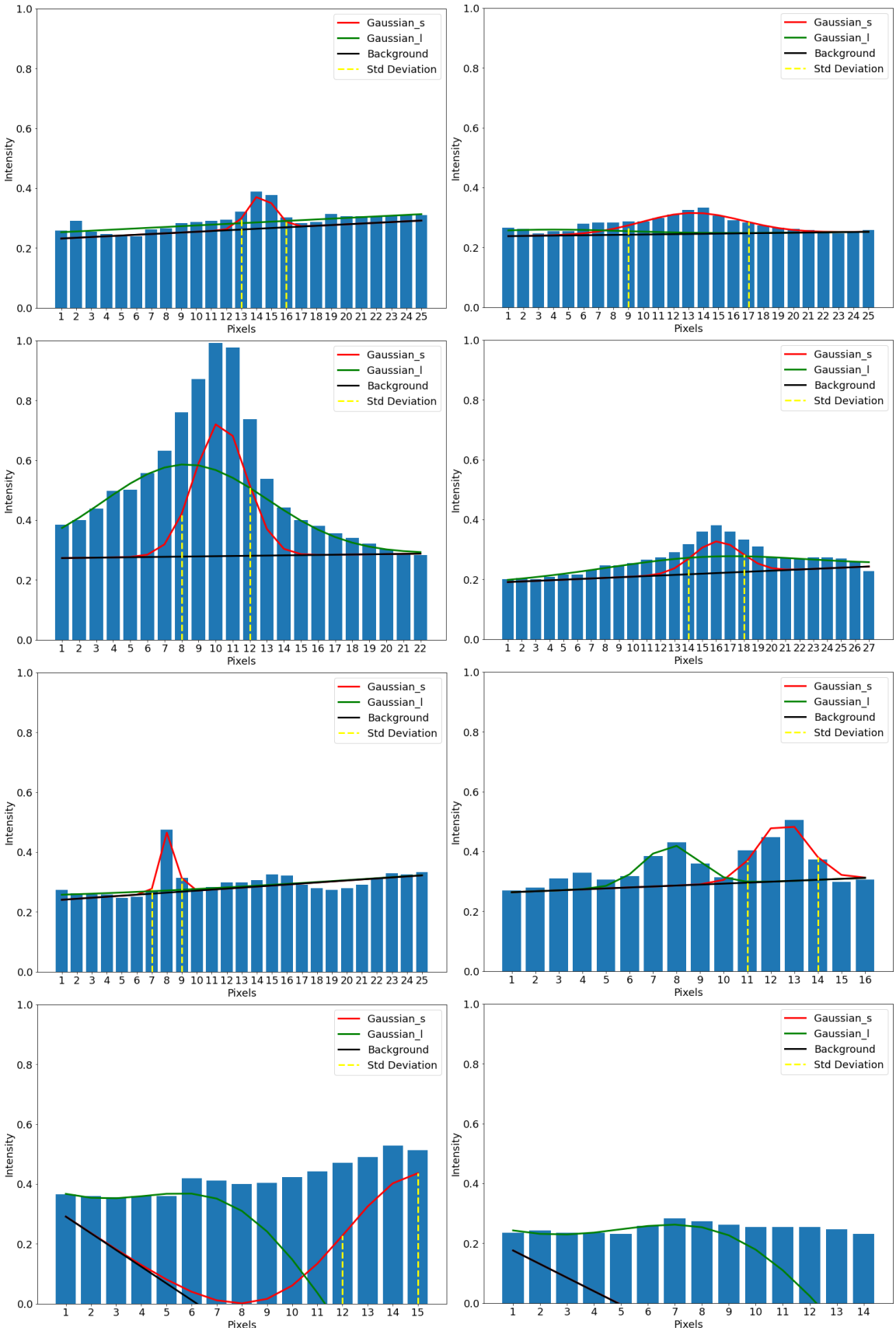


Figura 5.10. Esempi di *fitting*.

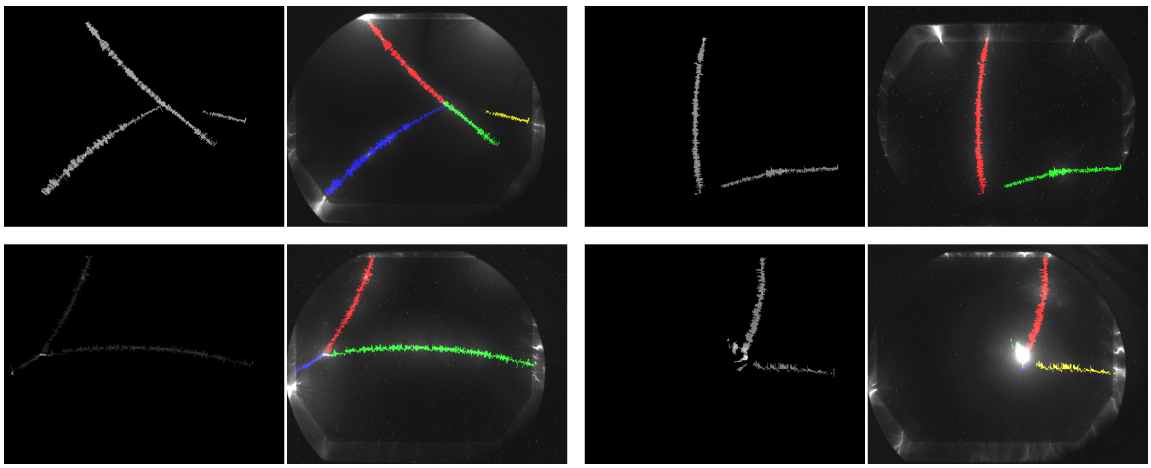


Figura 5.11. Esempi di output (a sinistra) con localizzazione della traccia nell'evento (a destra).

Come già descritto nella sezione "Studio dell'intensità", l'obiettivo finale della riduzione dei campioni è quello di avere un rilevamento più efficiente delle traiettorie particellari. In Figura 5.11 si possono vedere quattro esempi di output finale. Siccome per questa parte del sistema non abbiamo un Ground Truth per valutare le prestazioni, per ogni rilevamento mostro anche la sua localizzazione all'interno dell'evento originale al fine di poter fare una valutazione visiva.
Nei quattro esempi si può vedere come, il fatto che abbia scomposto ogni traccia in più campioni e poi, dopo averli ridotti, li abbia riuniti, è facilmente intuibile dal comportamento "a tratti" delle traiettorie. Lo studio dell'intensità del secondo campione è indipendente dallo studio dell'intensità del primo campione e del terzo campione, di conseguenza, campioni adiacenti possono mostrare valori (i parametri delle curve: A_s, $x0_s$ e σ_s per la gaussiana associata al segnale, A_l, $x0_l$ e σ_l per la gaussiana associata al rumore e m e q per la retta) diversi. Inoltre, con questo procedimento, si avranno sempre dei problemi nei casi in cui la fotografia dell'evento ha una forte sovraesposizione e. quindi, un'intensità che raggiunge la saturazione per alcune tracce (quarto esempio della Figura 5.11).

Capitolo 6

Conclusioni e sviluppi futuri

Il sistema creato ha dimostrato che, con un approccio ibrido costituito sia da tecniche di Machine Learning per la segmentazione semantica delle traiettorie che da trasformazioni nell'ambito della Computer Vision per la divisione delle maschere delle tracce, sia possibile risolvere un task di rilevazione di traiettorie particellari con risultati incoraggianti. La metrica di interesse maggiore per gli obiettivi da raggiungere, e per poter poi lavorare sulla ricostruzione 3D delle tracce, è il numero di interazioni fotografate le cui traiettorie sono state tutte rilevate correttamente senza alcun Falso Positivo o Falso Negativo. Le performance calcolate ci indicano, quindi, un sistema che riesce, con il *Dataset 256*, a segmentare correttamente 44 eventi su 100 sul validation set. Le prestazioni del sistema sono superiori a quelle del *Reconstruction Code* che ha una percentuale di efficienza del 10%.
Si può osservare, oltretutto, come i risultati sul test set risultino in tutti i casi inferiori rispetto a quelli sul validation set. Questi valori delle metriche possono essere causati da due motivazioni principali:

- La creazione, e la successiva parametrizzazione, dell'algoritmo sono state effettuate sul validation set. Da ciò consegue che i valori dei parametri scelti e le operazioni e trasformazioni utilizzate mi hanno permesso di avere buone performance sul validation set portando il sistema a generalizzare poco e, quindi, a non essere altrettanto performante con un altro set di immagini.

- Il test set presenta più casi critici rispetto al validation set. Ritroviamo più interazioni con una forte sovraesposizione (saturazione dei pixel della traiettoria e del rumore gaussiano intorno ad essa) e diversi casi che non rispettano l'ipotesi iniziale su cui si basa l'algoritmo di divisione tracce: non esistono traiettorie sovrapposte.

Attraverso la validazione e i test sperimentali, inoltre, si è potuto comprendere in quali casi il sistema ha avuto più difficoltà a rilevare le traiettorie. Infatti, osservando la Tabella 5.5 si può notare come i valori della Binary IoU, calcolati per la prima parte del sistema, non si discostino di molto tra un dataset e l'altro. Non si può dire lo stesso leggendo le Tabelle 5.7 e 5.8. I valori delle metriche calcolate per la

valutazione delle performance del sistema mostrano delle prestazioni nettamente migliori sul *Dataset 512* e *Dataset 256* rispetto al *Dataset 128*. Ciò può implicare che le trasformazioni morfologiche, adottate nell'algoritmo di divisione tracce, non siano adeguate alle dimensioni del *Dataset 128*: lo studio dei ridge attraverso gli autovalori dell'Hessiano, ad esempio, è un'operazione i cui risultati dipendendono dalle dimensioni delle maschere; quindi, con maschere più sottili (caso del *Dataset 128*) vi è una maggiore predisposizione all'errore (mancata rilevazione di un ridge o rilevazione di un ridge ove non presente).
In tutti i casi che abbiamo visto e analizzato, il valore della Recall è sempre superiore a quello della Precision. Tale dato, come già specificato nella sezione "Prestazioni del sistema complessivo", ci indica una maggior presenza di Falsi Positivi che di Falsi Negativi. Ciò può essere causato dal fatto che il sistema, in alcuni casi, tende a rilevare più tracce di quelle che ci sono in realtà. Un esempio lampante si può vedere nella prima riga della Figura 5.6: per il *Dataset 128* si noti come la traccia in alto venga divisa in due tracce; oppure lo stesso si può vedere anche con il *Dataset 512* dove invece è la traiettoria proveniente da destra ad essere divisa in due tracce.
Un'altra difficoltà del sistema, condivisa per tutti e tre i dataset, è facilmente riconoscibile per alcune immagini con più di quattro o cinque traiettorie particellari. Tale problema non è causato dal numero di traiettorie in sé, ma dal forte rumore gaussiano prodotto dalle tracce presenti a schermo. In questi casi l'immagine dell'evento risulta sovraesposta; di conseguenza, la segmentazione dell'evento potrebbe non essere efficace.
Le criticità appena descritte potrebbero essere superate approcciandosi al task di Instance Segmentation utilizzando un modello *fully convolutional*, come un'architettura Mask R-CNN[12]. In questo caso, si evitano alcune delle complicazioni legate alle trasformazioni morfologiche in ambito Computer Vision, ma, allo stesso tempo, non si sfruttano le proprietà geometriche delle traiettorie particellari. Con un modello di questo tipo si potrebbero avere dei risultati migliori soprattutto per i casi più particolari: tante traiettorie con forte rumore di fondo e gaussiano oppure sovrapposizioni (anche multiple) di traiettorie.
Un modo per migliorare lo studio dell'intensità, dopo aver segmentato e rilevato le tracce, potrebbe essere quello di fare un'analisi dei campioni che dipenda dai campioni precedenti e che non sia quindi indipendente. In questo modo si potrebbe avere una maschera della traiettoria che sia ancora più precisa e che, quindi, segua in modo più efficiente i bulbi di luce generati dalla ionizzazione.
A partire da questo mio lavoro di tesi, il prossimo step sarà quello di ricostruire automaticamente in 3D l'interazione particellare fotografata. Questo task diventa possibile grazie alle due videocamere CCD. Infatti, utilizzando la rilevazione delle tracce (effettuata dal sistema descritto in questa tesi) delle due visioni di uno stesso evento, si può ricostruire la profondità e cercare di capire come le particelle si muovono nella terza dimensione così da studiare i loro parametri cinematici (momenti, energie, ecc.) e poter utilizzare tale rappresentazione 3D per un fine didattico.

Elenco delle figure

Elenco delle tabelle

Bibliografia

[1] E.M. Andreev, N.S. Angelov, S.A. Baginyan, Yu.A. Batusov, I.A. Belolaptikov, T.D. Blokhintseva, A.Yu. Bonyushkina, V.A. Butenko, A.A. Dem'yanov, V.A. Drozdov, I.V. Falomkin, V.N. Frolov, V.M. Grebenyuk, V.V. Ivanov, A.S. Kirilov, V.E. Kovalenko, V.I. Lyashenko, A.S. Moiseenko, V.A. Panyushkin, G.B. Pontecorvo, V.I. Prikhod'ko, V.I. Pryanichnikov, A.M. Rozhdestvensky, N.A. Russakovich, O.V. Savchenko, F. Balestra, L. Busso, M.P. Bussa, M.L. Colantoni, L. Fava, A. Ferrero, L. Ferrero, R. Garfagnini, A. Grasso, A. Maggiora, M. Maggiora, A. Manara, G. Piragino, F. Tosello, G.F. Zosi, L.A. Kondratyuk, M.G. Schepkin. *Self-shunted streamer chamber spectrometer with CCD video cameras for studying pion interactions with light nuclei at energies below the Δ-resonance* in *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment; Volume 489, Issues 1–3; pp. 99-113 (2002)*

[2] W. S. Boyle, G. E. Smith. *Charge coupled semiconductor devices* in *The Bell System Technical Journal; Volume 49, Issue 4, pp. 587 - 593 (1970).* Available at: `https://ieeexplore.ieee.org/document/6768140`

[3] Tommaso Toffoli, Norman Margolus. *Cellular Automata Machines - A new environment for modeling (1987)*

[4] Mrinal Tyagi in Towards Data Science (2021). *Image Segmentation: Part 1.* Available at: `https://towardsdatascience.com/image-segmentation-part-1-9f3db1ac1c50`

[5] Linda G. Shapiro, George C. Stockman. *Image Segmentation* in *Computer Vision; pp. 305–351 (2001)*

[6] Heet Sankesara in Towards Data Science (2019). *UNet.* Available at: `https://towardsdatascience.com/u-net-b229b32b4a71`

[7] Jonathan Long, Evan Shelhamer, Trevor Darrell (2015). *Fully Convolutional Networks for Semantic Segmentation.* Available at: `https://arxiv.org/abs/1411.4038`

[8] Olaf Ronneberger, Philipp Fischer, Thomas Brox (2015). *U-Net: Convolutional Networks for Biomedical Image Segmentation*. Available at: `https://arxiv.org/abs/1505.04597`

[9] Hossein Gholamalinezhad, Hossein Khosravi (2009). *Pooling Methods in Deep Neural Networks, a Review*. Available at: `https://arxiv.org/ftp/arxiv/papers/2009/2009.07485.pdf`

[10] *RSNA-ASNR-MICCAI Brain Tumor Segmentation (BraTS) Challenge*. Available at: `http://braintumorsegmentation.org/`

[11] Fatemeh Nazem, Fahimeh Ghasemi, Afshin Fassihi, Alireza Mehri Dehnavi (2021). *3D U-Net: A voxel-based method in binding site prediction of protein structure*. Available at: `https://pubmed.ncbi.nlm.nih.gov/33866960/`

[12] Kaiming He, Georgia Gkioxari, Piotr Dollár, Ross Girshick (2017). *Mask R-CNN*. Available at: `https://arxiv.org/abs/1703.06870`

[13] Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik (2014). *Rich feature hierarchies for accurate object detection and semantic segmentation*. Available at: `https://arxiv.org/abs/1311.2524`

[14] Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun (2016). *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. Available at: `https://arxiv.org/abs/1506.01497`

[15] Lizhe Liu, Xiaohao Chen, Siyu Zhu, Ping Tan (2021). *CondLaneNet: a Top-to-down Lane Detection Framework Based on Conditional Convolution*. Available at: `https://arxiv.org/abs/2105.05003`

[16] Amartansh Dubey, K. M. Bhurchandi (2015). *Robust and Real Time Detection of Curvy Lanes (Curves) with Desired Slopes for Driving Assistance and Autonomous Vehicles*. Available at: `https://arxiv.org/abs/1501.03124`

[17] Georg Stimpfl-Abele, Lluis Garrido (1991). *Fast track finding with neural networks*. Available at: `https://www.sciencedirect.com/science/article/abs/pii/001046559190048P`

[18] David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams (1986). *Learning representations by back-propagating errors*. Available at: `https://www.nature.com/articles/323533a0`

[19] Steven Farrell, Paolo Calafiura, Mayur Mudigonda, Prabhat, Dustin Anderson, Josh Bendavid, Maria Spiropoulou, Jean-Roch Vlimant, Stephan Zheng, Giuseppe Cerati, Lindsey Gray, Keshav Kapoor, Jim Kowalkowski, Panagiotis Spentzouris, Aristeidis Tsaris, Daniel Zurawski (2017). *Particle Track Reconstruction with Deep Learning*. Available at: `https://dl4physicalsciences.github.io/files/nips_dlps_2017_28.pdf`

[20] Dmitriy Baranov, Sergey Mitsyn, Pavel Goncharov, Gennady Ososkov (2018). *The particle track reconstruction based on deep learning neural networks.* Available at: `https://arxiv.org/abs/1812.03859`

[21] Pavel Iakubovskii (2019). *Segmentation Models* Available at: `https://github.com/qubvel/segmentation_models`

[22] Abhishek Chaurasia, Eugenio Culurciello (2017). *LinkNet: Exploiting Encoder Representations for Efficient Semantic Segmentation* Available at: `https://arxiv.org/abs/1707.03718`

[23] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, Jiaya Jia (2017). *Pyramid Scene Parsing Network* Available at: `https://arxiv.org/abs/1612.01105`

[24] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, Serge Belongie (2016). *Feature Pyramid Networks for Object Detection* Available at: `https://arxiv.org/abs/1612.03144`

[25] Li Fei-Fei, Jia Deng, Olga Russakovsky, Alex Berg, Kai Li (2006). *Imagenet* Available at: `https://www.image-net.org/`

[26] Karen Simonyan, Andrew Zisserman (2016). *Very Deep Convolutional Networks for Large-Scale Image Recognition* Available at: `https://arxiv.org/abs/1409.1556`

[27] Diederik P. Kingma, Jimmy Ba (2014). *Adam: A Method for Stochastic Optimization* Available at: `https://arxiv.org/abs/1412.6980`

[28] Rafael C. Gonzales, Richard E. Woods. *Thresholding* in *Digital Image Processing, 3rd edition; pp. 760-783 (2002)*

[29] Jean Serra. *Image Analysis and Mathematical Morphology (1982)*

[30] Rafael C. Gonzales, Richard E. Woods. *Opening and closing* in *Digital Image Processing, 3rd edition; pp. 657-661 (2002)*

[31] H. Samet, M. Tamminen. *Efficient component labeling of images of arbitrary dimension represented by linear bintrees* in *IEEE Transactions on Pattern Analysis and Machine Intelligence; Volume 10, Issue 4 (1988)*

[32] Rafael C. Gonzales, Richard E. Woods. *Segmentation Using Morphological Watersheds* in *Digital Image Processing, 3rd edition; pp. 791-798 (2002)*

[33] Robert M. Haralick. *Ridges and Valleys on Digital Images* in *Computer Vision, Graphics and Image Processing; Volume 22; pp. 28-38 (1983)*

[34] Rafael C. Gonzales, Richard E. Woods. *Thinning* in *Digital Image Processing, 3rd edition; p. 671 (2002)*

[35] Paul Henderson, Vittorio Ferrari (2016). *End-to-end training of object class detectors for mean average precision* Available at: https://arxiv.org/abs/1607.03476

[36] *lmfit:* https://lmfit.github.io/lmfit-py/

[37] *CVAT:* https://github.com/opencv/cvat

[38] *Line2D:* https://matplotlib.org/stable/api/_as_gen/matplotlib.lines.Line2D.html

[39] *Arc:* https://matplotlib.org/stable/api/_as_gen/matplotlib.patches.Arc.html

[40] *Google Colaboratory:* https://colab.research.google.com/

[41] *HPC:* https://www.hpc.polito.it/index.php

[42] *SLURM:* https://slurm.schedmd.com/documentation.html

[43] Luis Perez, Jason Wang (2017). *The Effectiveness of Data Augmentation in Image Classification using Deep Learning* Available at: https://arxiv.org/abs/1712.04621

[44] *Albumentation:* https://albumentations.ai/