Università degli Studi di Verona

Corso di Laurea in Informatica

Tesi di Laurea in

Studio e utilizzo di tecniche di deep learning per la rilevazione automatica della copertura arborea da immagini satellitari

Anno Accademico 2024/2025

Relatore

Ch.mo Prof. Davide Quaglia

Candidato

Enrico Antonini

matr. VR500151

A Roberto,

in qualche modo, mio compagno di viaggio

Indice

1

Introduzione

Questa ricerca ha l'obiettivo di analizzare e costruire un metodo per identificare la presenza di alberi e aree verdi dalle immagini satellitari delle aree urbane, utilizzando modelli di machine learning.

Negli ultimi anni, infatti, l'avvento dell'intelligenza artificiale ha cominciato a cambiare il modo in cui lavoriamo e studiamo, permettendoci di effettuare ricerche e generare anche media audio/video a partire da un semplice testo digitato in un prompt. Questi tipi di intelligenza artificiale tuttavia, basati su modelli LLM per quanto molto rivoluzionari, sono general purpose e lavorano sul linguaggio naturale.

L'idea è quella di utilizzare dei modelli di machine learning che, sfruttando un algoritmo di visione artificiale, siano in grado di identificare velocemente la presenza di oggetti, come le chiome o la forma degli alberi, per scopi di pianificazione urbanistica o per alimentare indicatori di accesso al verde, come quello determinato dalla regola del 3-30-300.

2

Dettagli tecnici

2.1 Cos'è una rete neurale

Una rete neurale è un modello che simula il funzionamento della mente umana, in modo da poter imparare a riconoscere immagini e testi, grazie ad un processo di apprendimento (training) per poi successivamente dare dei feedback e delle previsioni su elementi simili o della stessa classe, con un processo chiamato inferenza.
È bene specificare che non tutti i modelli sono basati su reti neurali, esistono infatti algoritmi non neurali che possono simulare ragionamenti cognitivi umani, come gli alberi decisionali: essi sono l'esempio perfetto di modelli categorizzati come machine learning, in quanto apprendono dai dati di addestramento forniti in input, che non sono composti da layers e nodi come nelle tradizionali reti neurali [1].
Nello specifico le reti neurali multi livello, o multi layer, trattate in questo capitolo sono un particolare sottoinsieme del machine learning chiamato deep learning.

2.1.1 Layers

Un modello è composto da diversi livelli organizzati in sequenza, chiamati layers, formati a loro volta da nodi che possono essere considerati come neuroni artificiali.
Semplificando il più possibile, i layers possono essere di input, di output oppure intermedi:

- input layer: prende i dati in input. Se usiamo ad esempio un modello YOLO, sarà un'immagine.
- hidden layers: ricevono l'input, elaborano le informazioni attraverso calcoli e funzioni matematiche e passano i risultati al successivo strato. Sono gli strati interni al modello.
- output layer: restituisce la prediction finale del modello. Nel nostro caso, ci dice se ha trovato oggetti all'interno dell'immagine, specificando eventualmente anche informazioni aggiuntive come classi e livello di confidenza.

Questa struttura a layers permette al modello di trasformare e combinare i dati, imparando a riconoscere pattern e oggetti. Ogni layer aggiunge una trasformazione nuova, consentendo al modello di apprendere funzioni sempre più sofisticate [2].

2.1.2 Weights

Ad ogni collegamento tra nodi di layer consecutivi è associato un peso (weight), ovvero un valore numerico che indica la forza della connessione tra due nodi. Durante l'addestramento, questi pesi vengono continuamente aggiornati in modo che, grazie al passaggio dei dati attraverso i layers, il modello possa combinare gli input in modo sempre più efficace per riconoscere pattern e fare previsioni accurate.
I pesi lavorano su informazioni e dati, che vengono trasmessi da un nodo al successivo. Nello specifico un peso alto amplifica le informazioni, mentre un peso basso le attenua: è proprio l'ottimizzazione di questi pesi che permettono al modello di migliorare la sua precisione e la capacità predittiva [3].

2.2 Dataset

Un dataset è una raccolta strutturata di dati, nel caso di questa ricerca è formata prevalentemente da immagini con annotazioni ed è organizzata per essere analizzata e processata da algoritmi di machine learning. I dataset possono essere composti da:

- Training set: immagini usate per il training del modello.
- Validation set: immagini utilizzate durante l'addestramento, per valutare le prestazioni del modello su dati non visti e per effettuare la messa a punto dei parametri.

- Test set: un insieme di immagini utilizzate alla fine dell'addestramento, per valutare le prestazioni finali del modello, su dati completamente nuovi che non sono stati utilizzati per il training.

Il validation set è opzionale, ma molto utile per migliorare la qualità del training del modello.

2.2.1 Importanza del dataset

La qualità e la coerenza dei dati nel dataset sono importanti quanto l'algoritmo stesso: da un dataset con immagini pulite e annotazioni ben fatte è possibile fare training (o transfer learning) ed addestrare un modello accurato.
Il dataset, inoltre, deve contenere immagini nel formato previsto dalle specifiche del modello (e.g. YOLO in 640x640)
La suddivisione in training, validation e test set permette di sviluppare un modello robusto e testarne le performance su dati "mai visti prima"

2.2.2 Annotazione manuale

Per costruire un dataset adatto al tema di riconoscimento degli oggetti, che è scopo di questa ricerca, è necessario avere delle immagini con risoluzione, numero di canali (e.g. RGB) e formato, che siano compatibili con il modello da addestrare.
A seconda della rete neurale da utilizzare poi, le immagini devono essere corredate da un file che descrive il contenuto o l'oggetto da identificare: ad esempio per i modelli YOLO ogni immagine è associata ad un file, che contiene coordinate e classe di ogni oggetto presente all'interno dell'immagine.
A questo punto la rete neurale che sta alla base del modello, durante la procedura di training, imparerà a distinguere gli oggetti che abbiamo annotato nel dataset e proprio grazie a questa procedura, riuscirà poi a valutare in maniera più o meno precisa, a seconda del tipo di rete e della qualità del dataset di addestramento, se una qualsiasi immagine fornitagli in input contiene o meno oggetti simili a quelli annotati, fornendo anche dei valori che indicano la stima di quanto è preciso quel rilevamento.
Piattaforme come Roboflow offrono un editor dove è possibile annotare le immagini e contestualmente generare i files con le annotazioni.

2.2.3 Ricerca di un dataset con immagini già annotate

Una seconda soluzione è quella di utilizzare dataset pre-annotati, che possono essere scaricati da Roboflow o da repository GitHub.
In questo caso è fondamentale sincerarsi che il dataset sia stato creato appositamente per la rete che si vuole addestrare e che le annotazioni siano corrette. In generale i dataset resi disponibili online, se provenienti da progetti di computer vision e documentate in articoli di ricerca, sono piuttosto affidabili.

2.3 Training

Per permettere ad un modello di machine learning di imparare a riconoscere pattern o feature significative in un immagine, è necessario effettuare un'operazione chiamata training, che è di fatto un vero e proprio addestramento del modello.
Senza un adeguato addestramento la rete neurale non avrebbe una strategia per interpretare i dati e svolgere il compito desiderato.

Il training viene effettuato partendo da un dataset, ovvero una collezione di immagini con le relative "annotazioni": in ogni immagine di un dataset viene evidenziata la posizione e la classe dell'oggetto, che è indispensabile per addestrare la rete a riconoscere successivamente questi oggetti nelle immagini che dovremo classificare.
Ci sono due tipologie di addestramento che possono essere eseguite su una rete, il training completo (from scratch) e il transfer learning [4].

2.3.1 Training from scratch

In questo tipo di training, il modello viene addestrato completamente sui dati annotati senza partire da pesi pre-trainati. I pesi (weights) sono valori numerici che vengono assegnati alle connessioni fra i nodi della rete neurale: ogni volta che un input arriva ad un nodo, viene moltiplicato per il peso associato a quella specifica connessione, la somma di tutti questi input ne determinerà poi i valori in output.
Durante la fase di training della rete neurale, i pesi vengono costantemente aggiornati mediante algoritmi di apprendimento, in modo da minimizzare la differenza fra le previsioni del modello

e i valori reali.
Partendo da zero, ovvero from scratch, questi pesi non sono definiti ma vengono assegnati dei numeri casuali che, soltanto durante il processo di training, verranno modificati per ottimizzare le prestazioni.
Questo tipo di training è molto valido quando abbiamo a disposizione un numero considerevole di dati annotati e l'utilizzo del modello è molto specifico.

2.3.2 Transfer learning

Quando il dataset è limitato, o nel caso in cui l'obiettivo sia quello di riuscire a risparmiare tempo e risorse, è possibile partire da un modello pre-trainato.
L'idea è quella di "trasferire" ad un modello provvisto di pesi e con dei parametri di precisione già significativi, la parte di conoscenza necessaria per riconoscere i dati annotati nel nostro dataset, si inizia quindi dal modello pre-trainato e lo si adatta (fine-tuning) al target, spesso "freezando" i primi layer e ottimizzando solo quelli finali.

2.4 Parametri di training

Qui di seguito sono elencati alcuni parametri, che normalmente vengono specificati in fase di addestramento del modello: sono utili per affinare il processo di apprendimento e vengono anche chiamati "Hyperparameters".
Questi parametri sono piuttosto generici, perchè comuni a tante procedure di addestramento anche fra algoritmi diversi [5].

2.4.1 Epoch

Un epoch rappresenta un passaggio completo del dataset attraverso l'algoritmo in fase di learning, è un concetto fondamentale nel training delle reti neurali, che permette al modello di imparare continuando costantemente a rivedere le stesse immagini, o i medesimi elementi del dataset [6].
Il numero di epoch si specifica solitamente in ogni procedura di training e determina quante volte il modello imparerà dal set di train del dataset, influenzando le performance e la qualità del modello.

Scegliere il numero di epoch corretto è molto importante perché se viene scelto in maniera errata può creare due problemi:

- Underfitting: il modello non è stato trainato per un numero sufficiente di epochs. Le performances non sono adeguate né sul set di training né su quello di test
- Overfitting: il modello è stato trainato per troppi epochs. In questo caso memorizza i dati di training in maniera troppo approfondita e perde le sue abilità di analizzare i nuovi dati. Ha una precisione eccellente sul training set ma scarsa sul dataset di validazione.

Una tecnica comune per evitare l'overfitting è quella dell'early stopping, nella quale il training viene fermato quando le performances sul set di validazione smettono di migliorare. A volte è possibile specificare un parametro chiamato patience, che specifica dopo quanti epochs "non migliorativi" il training deve fermarsi.

2.4.2 Batch size

La dimensione del batch è un altro parametro importante che definisce il numero di campioni processati prima che i parametri interni del modello vengano aggiornati. I campioni sono in questo caso le immagini, che vengono usate per il training del modello.
Dividendo il set di training in batch, ovvero dei subset ridotti, evitando di processare tutti i dati simultaneamente, che potrebbe essere computazionalmente problematico in termini di velocità di elaborazione e memoria occupata.
Scegliere un batch size troppo basso però, implica una ridotta velocità di training, in quanto i pesi del modello vengono aggiornati molto più frequentemente.

2.4.3 Loss function

È una funzione matematica fondamentale nel machine learning, che misura quanto le prediction generate dal modello si discostano dai valori reali.

$$\text{Loss} = f(prediction, realValue)$$

La funzione è chiamata sia durante il training per ogni batch di immagini, che in fase di validazione alla fine di ogni epoch. Il valore poi restituito dalla loss function, durante il training,

può venire utilizzato da degli ottimizzatori che permettono di aggiornare i parametri, come ad esempio i pesi, ad ogni epoch come una sorta di "pilota automatico".

2.5 Hardware necessario per il training

I modelli di machine learning hanno bisogno di particolari risorse hardware per poter essere addestrati in quanto, l'utilizzo di un normale computer desktop che sfrutta la sola CPU per "istruire" il modello, è troppo limitante non tanto per la frequenza ma per l'insufficiente numero di core che ha a bordo, è invece necessario avvalersi di hardware che permetta il calcolo parallelo: per questo compito ci vengono in aiuto le GPU.
Queste ultime infatti, soprattutto quelle prodotte da Nvidia, sono compatibili con la piattaforma di calcolo parallelo CUDA (Compute Unified Device Architecture) che sfrutta le centinaia o le migliaia di core presenti a bordo dei chip, riuscendo quindi a svolgere massivamente calcoli matematici e gestire enormi quantità di dati [7].
È importante sottolineare che i modelli di machine learning più pesanti necessitano di molta RAM, ed è quindi doveroso dotarsi di GPU con sufficiente memoria video (VRAM) a bordo, per fare in modo che il modello possa rimanere completamente residente in memoria durante le procedure di training o inferenza.

2.5.1 Bare metal

Per utilizzare CUDA su una macchina locale è necessario acquistare l'hardware fisico, che generalmente in ambito consumer è una scheda di espansione su bus PCIe che permette il collegamento di uno o più monitor ed ha il compito di accelerare calcoli che sarebbero troppo pesanti per la CPU, come ad esempio calcolo di poligoni e shader nei giochi oppure addestramento/inferenza su modelli di machine learning.
In ambito industriale invece, trovano spazio delle soluzioni non progettate per l'ambito ludico, che sono invece commercializzate come acceleratori di calcolo parallelo.
Esistono anche delle versioni ridotte su sistemi embedded/single board computer (e.g. Jetson Orin) che permettono l'utilizzo di CUDA on the edge, per valutare ad esempio le immagini provenienti da una fotocamera.
Tutte queste soluzioni implicano l'acquisto di hardware dedicato, che deve essere configurato correttamente per poter funzionare e, soprattutto i modelli di GPU più prestanti con VRAM

elevata hanno un costo importante.
Per il training e l'inferenza è solitamente utilizzato Python, in quanto provvisto di librerie adatte a lavorare con i modelli (come Pytorch o TensorFlow) ed è necessario che venga sviluppato il codice dedicato.

2.5.2 Risorse cloud

Una vera alternativa all'elaborazione on premise è quella di utilizzare i servizi in cloud che comportano un risparmio in termini di costo hardware, in quanto è tutto demandato alla piattaforma gestita, ma implicano comunque la necessità di lavorare a partire da sorgenti Python per il training o inferenza con i modelli.
Solitamente è previsto un costo mensile preventivabile con un calcolatore, che varia a seconda del servizio e dell'hardware messo a disposizione dalla piattaforma cloud.
I provider più famosi sono Google Cloud Platform, AWS EC2 e Azure VMs, che offrono macchine virtuali Linux o Windows con GPU.
In alternativa, ad un costo inferiore, sono disponibili servizi come Google Colab che supportano solo linguaggi di scripting (e.g. Python, R) offrendo ad un canone mensile l'accesso ad un numero di ore di elaborazione CPU+GPU. Sono un buon compromesso per lavorare con i modelli di machine learning.

2.5.3 Soluzioni all-in-one

Una soluzione che sta prendendo piede negli ultimi anni è quella di affidarsi ai servizi all-in-one che, a fronte di un canone mensile, prevedono la gestione del dataset e la creazione del modello tramite training. Questo tipo di servizi offre anche la possibilità di accedere ai modelli online, tramite servizi REST, con degli endpoint che permettono di fare inferenza senza doversi preoccupare di sviluppare script da eseguire sulla propria macchina o sul cloud.
Tra i servizi più diffusi ci sono sicuramente Roboflow e Ultralytics.

2.6 Modelli a confronto

Al momento della stesura di questo documento (ca. fine 2025) sono disponibili diverse soluzioni, che promettono di essere efficaci nell'individuare oggetti all'interno di un immagine, in maniera

più o meno elaborata.

Al contrario delle reti neurali che nella prima metà degli anni venti sono diventate popolari (e.g. ChatGPT, Gemini, Llama o Grok) per essere molto efficaci con il linguaggio naturale (NLP) data la loro natura di LLM, acronimo di Large Language Model, in questo caso specifico la ricerca tratterà di reti neurali per la computer vision, come ad esempio le reti convoluzionali ricorsive e non.

2.6.1 AdaBoost

È un algoritmo di machine learning che combina più modelli chiamati weak learners per creare un modello evoluto più accurato [8]. I weak learners sono modelli o algoritmi leggermente più performanti di una predizione casuale, ad esempio in una classificazione binaria raggiungono poco più del 50% di precisione. Facendo training sequenziale su questi weak learners e combinandoli, il modello finale non sarà altro che la somma pesata di tutti i modelli più piccoli, risultando più performante e robusto.

Nel caso di AdaBoost il processo è adattivo perchè corregge gli errori fatti nei primi round di training, rendendolo così più efficace.

È importante, come anticipato in precedenza, sottolinare che AdaBoost non è una rete neurale ma, per sua natura combinando modelli più piccoli, è "semplicemente" un albero decisionale.

DetecTree utilizza AdaBoost per classificare, pixel by pixel, cosa fa parte di un albero e cosa no.

2.6.2 CNN

Alcuni algoritmi di object detection sono basati su una rete neurale convoluzionale (CNN) [9], che viene utlizzata per permettere la detection di oggetti nell'immagine. Essa è uno specifico tipo di algoritmo di deep learning, studiato principalmente per analizzare dati visuali, come immagini e video. Questi algoritmi sono catalogati come deep learning, appunto per la loro caratteristica di avere più livelli.

In pratica mima la funzionalità della corteccia cerebrale umana, estraendo automaticamente determinate caratteristiche o oggetti, come ad esempio gli alberi, a partire dall'immagine: questo è possibile tramite filtri chiamati convoluzionali, che vanno alla ricerca di specifici pattern.

Il nome dei filtri deriva dal tipo di operazione che viene effettuata sulle immagini in input, ovvero la convoluzione: si tratta di un'operazione matematica che in termini semplici consiste nel far

scorrere un filtro (detto anche kernel) sopra l'immagine in input. Il filtro esamina l'immagine e calcola una combinazione pesata dei valori dei pixel, producendo una valore in output. Ripetendo il processo più volte si crea una nuova immagine che è in grado di mettere in evidenza caratteristiche specifiche come bordi, angoli o texture: in sostanza permette quindi alla rete di riconoscere pattern visivi in modo efficiente mantenendo anche la posizione spaziale degli oggetti rilevati. Un esempio di CNN è la rete YOLO.

2.6.3 R-CNN

In alcuni casi le reti possono essere anche R-CNN, ovvero Region-based CNN che, come le normali reti convoluzionali, sono anch'esse algoritmi di deep learning.
Anche in questo caso sono modelli progettati per la object detection: prima vengono generati molteplici proposte di regioni (aree potenzialmente contenenti oggetti) tramite specifici algoritmi, quindi viene applicata una CNN a ciascuna regione per estrarne caratteristiche, e infine vengono classificati e localizzati gli oggetti in quelle regioni [10]. In sostanza non vengono applicati filtri a tutta l'immagine, ma le operazioni di convoluzione sono effettuate a livello locale in regioni distinte.
Fra le R-CNN possiamo trovare DetecTree2.

2.6.4 ViT

I modelli LLM commerciali citati all'inizio di questa sezione, che negli ultimi anni hanno subito un'evoluzione incredibile, hanno introdotto un'architettura chiamata transformer.
Al contrario delle (R)CNN viste in precedenza, i transformer si basano su meccanismi di self-attention dove parte del dato di input, come una parola o un immagine, viene confrontata e pesata rispetto a tutte le altre parti per capire quali sono più rilevanti nel contesto corrente. In questo modo il modello può valutare l'importanza delle diverse parti di una sequenza, indipendentemente dalla loro posizione [11]. Un Transformer è costituito da:

- Encoder: converte il testo in input in una rappresentazione intermedia. L'encoder è una rete neurale.
- Decoder converte la rappresentazione intermedia in un output testuale. Anche il decoder è una rete neurale.

Recentemente l'architettura transformer è stata applicata anche alle reti dedicate alla visione, creando così i Vision Transformer (ViT): invece di trattare l'immagine come una griglia di pixel, come succede per le (R)CNN, il ViT suddivide l'immagine in piccoli pezzi di dimensioni fisse, che vengono poi trasformati in vettori (embeddings), vengono arricchiti con informazioni sulla posizione spaziale e quindi passati a un encoder transformer, che utilizza meccanismi di self-attention per catturare relazioni sia locali sia globali tra le diverse patch.
Questo procedimento permette di migliorare le prestazioni in compiti come classificazione, segmentazione e riconoscimento di oggetti, specialmente con grandi dataset di addestramento. Segformer è una rete neurale basata su questo algoritmo di deep learning.

2.7 Misurazione delle performances

I modelli, una volta trainati, possono essere valutati sulla base di alcune specifiche metriche, che sono comuni anche fra reti neurali diverse.
Qui sotto sono elencate diverse metriche generali, le metriche specifiche e dedicate a modelli precisi, sono invece riportate nei capitoli successivi.

2.7.1 Precision

Indica la percentuale di oggetti individuati dal modello che sono effettivamente corretti. In altra parole indica, su tutte le volte in cui il modello ha trovato un oggetto in un immagine, quante volte ha effettivamente "avuto ragione". Un'alta precisione significa pochi falsi positivi.

2.7.2 Recall

Misura la percentuale di oggetti realmente presenti nell'immagine che sono stati trovati dal modello, in sostanza ci restituisce un valore relativo al numero di oggetti riconosciuti rispetto a quanti sono stati correntemente annotati come buoni nel dataset. Un recall alto equivale a pochi falsi negativi.

2.7.3 Intersection over Union (IoU)

Metrica fondamentale della computer vision nel riconoscimento degli oggetti, che misura quanto due rettangoli (boxes) si sovrappongono tra loro rispetto alla loro estensione. Si utilizza in

pratica per confrontare quanto la bounding box predetta, coincide con la bounding box reale (chiamata ground truth).

$$IoU = \frac{areaIntersezione}{areaUnione}, \quad IoU \in [0, 1]$$

ad esempio un IoU di 0.6 indica che due box hanno il 60% di area in comune [12].
La soglia di IoU è un valore di input del modello, che stabilisce quanto deve essere "centrato" un oggetto per essere considerata corretta la previsione: più alto è il valore, più rigorosa è la valutazione.

2.7.4 Mean Average Precision (mAP)

Parametro utilizzato per misurare le performance dei modelli di computer vision, come ad esempio YOLO o DetecTree2. La mAP rappresenta la precisione media su diverse soglie di IoU, di tutte le classi presenti nel dataset: viene calcolata come la media aritmetica dei valori di precision di tutte le classi, fornendo una misura complessiva della capacità del modello di classificare correttamente gli oggetti. E' fondamentale per comparare anche modelli diversi che si occupano di svolgere lo stesso task, oppure anche versioni diverse dello stesso modello.

$$\text{mAP} = \frac{1}{N} \sum_{i=1}^{N} AP_i, \quad mAP \in [0, 1]$$

con N numero di classi e APi che corrisponde alla precisione media della classe i [12].

2.7.5 Mean Average Precision 0.50 (mAP50)

Metrica che calcola la media dell'Average Precision su tutte le classi, utilizzando una soglia di IoU fissa di 0.50. Una predizione viene considerata corretta (vero positivo) solo se la sovrapposizione con la ground truth è almeno del 50%.
Questa metrica fornisce una valutazione delle prestazioni del modello con una soglia di accettazione tollerante, ed è utile per applicazioni dove è sufficiente una localizzazione degli oggetti approssimativa.

2.7.6 Mean Average Precision 0.50-0.95 (mAP50-95)

Metrica che calcola la media dell'Average Precision considerando multiple soglie di IoU, da 0.50 a 0.95 con incrementi di 0.05 (totale 10 soglie). Per ogni soglia viene calcolata la mAP, poi si effettua la media di tutti questi valori.
Questa metrica è molto più rigorosa della mAP50, perché valuta la precisione del modello su diversi livelli di difficoltà, richiedendo sia localizzazioni approssimative che molto precise. Un valore elevato di mAP50-95 indica che il modello è accurato in tutti i contesti, dalla detection approssimativa a quella di precisione millimetrica.

2.7.7 F1 score

F1 è la media armonica di precision e recall

$$F1 = 2 * \frac{precision + recall}{precision * recall}, \quad F1 \in [0, 1]$$

questo valore bilancia precision e recall, penalizzando fortemente il caso in cui uno dei due è molto basso, infatti più ci si avvicina ad 1 meno si hanno falsi positivi e falsi negativi. Questo valore è molto importante perché ci indica quanto un modello è preciso e completo nelle sue predizioni.

3

YOLO

YOLO è una popolare rete neurale convoluzionale (CNN) che permette la detection in real-time di oggetti all'interno di immagini.
A differenza di altri algoritmi di computer vision, YOLO effettua la propria analisi in una sola iterazione, da qui l'acronimo di YOLO, You Only Look Once.

3.1 Principio di funzionamento

Il principio di funzionamento dell'algoritmo YOLO è riassumibile nei seguenti steps: [13]

- L'immagine in input è divisa in una griglia n x n.
- Grazie ad una mappa di probabilità definita a priori, vengono predette le posizioni di bounding boxes multipli, che evidenziano l'oggetto rilevato.
- Viene restituita la classe dell'oggetto rilevato insieme alla bounding box corrispondente.

la "classe" dell'oggetto corrisponde al tipo di annotazione presente nel dataset utilizzato poi per effettuare training del modello: nel nostro caso, dato che il modello deve riconoscere la presenza di alberi in una data immagine satellitare, avremo una sola classe relativa agli alberi.

3.2 Tipologia di modello utilizzato

I modelli YOLO hanno subito delle evoluzioni nel corso del tempo e sono tuttora sviluppati per incrementarne la precisione e le performance, ogni nuova versione cerca di aumentare le prestazioni per avere precisione più alta e tempi di inferenza più bassi rispetto ai modelli precedenti. L'algoritmo di deep learning di YOLO è sviluppato a partire dal framework PyTorch.

3.2.1 PyTorch

E' un framework open source di deep learning, in linguaggio Python, creato per facilitare la creazione, il training e l'implementazione di modelli di reti neurali [14]. Fra le sue feature principali è possibile trovare:

- Supporto ai tensor, ovvero agli array multidimensionali, che possono essere elaborati su CPU o GPU con calcolo parallelo.
- Include moduli per la creazione di reti neurali complesse con calcolo dei gradienti per l'ottimizzazione.
- Offre librerie come torchvision per supportare la computer vision e torchtext per l'elaborazione del linguaggio naturale.
- Compatibilità multipiattaforma e ampio supporto grazie alla sua nutrita community.

3.2.2 Versione del modello

La versione di YOLO utilizzata in questo progetto è la 11, introdotta nel 2024, che garantisce miglior accuratezza nelle predizioni e maggiore velocità. Inoltre è disponibile una versione ottimizzata per devices edge, come ad esempio smartphones o sistemi embedded.
A differenza di alcune vecchie versioni di YOLO, qui abbiamo la possibilità di utilizzare nelle procedure di addestramento dei modelli pre-trained, rendendo così possibile il training anche con datasets ridotti.
I modelli YOLO sono disponibili in vari "tagli": esistono infatti oltre alle versioni extralarge, che sono più precise e complete, anche delle versioni medie e small che sono indicate per l'utilizzo su dispositivi con risorse più limitate.
Osservando alcune statistiche di confronto fra i vari sistemi YOLO, a parità di dataset, c'è un

significativo improvement sui valori di precisione medi (mAP50) rispetto alle versioni precedenti anche per modelli di medie dimensioni (e.g. YOLOv11m vs YOLOv10m) e una riduzione del tempo necessario per eseguire l'inferenza dal modello. [15]

3.2.3 YOLO Ultralytics

Ultralytics è una piattaforma commerciale che ha realizzato delle librerie, utilizzabili sotto licenza AGPL-3.0, studiate appositamente per facilitare il training e l'inferenza con modelli YOLO [16].
L'utilizzo di queste librerie offre anche il vantaggio non trascurabile di poter convertire modelli YOLO PyTorch in altri formati, come Tensorflow JS, rendendoli utilizzabili su applicativi frontend web.

3.3 Preparazione del dataset

Come già introdotto nel capitolo precedente, l'importanza di avere un dataset ben formato è cruciale per procedere con il training del modello YOLO: senza un adeguato numero di immagini e di annotazioni infatti, il training risulterebbe poco efficace.
In questo specifico caso, che richiede il rilevamento di alberi nel tessuto urbano, la scelta della fonte dei dati non è banale: la maggior parte dei modelli che riconoscono questo tipo di piante, sono addestrati a partire da dataset che includono immagini satellitari di campagne o aree boschive.
Per il training di questo modello, invece, è stato utilizzato un dataset distribuito online dall'università di Lleida in Spagna [17], composto da un set di immagini di training e un set di valutazione.
Esso contiene annotazioni precise, di alberi in contesto urbano, con immagini già in formato compatibile con YOLO (640x640).

3.4 Script di training

Di seguito sono riportate alcune funzioni fondamentali per il training di un modello YOLO, sfruttando le librerie Ultralytics: non sono da considerarsi esaustive, ma illustrano le operazioni

di base per l'addestramento e la raccolta dei dati statistici, relativi alla qualità e performance del modello.

3.4.1 Init

```
import torch
from ultralytics import YOLO
from pathlib import Path
import os

baseFolderPath = "../myDataset"
datasetYamlPath = f"{baseFolderPath}/dataset.yaml"
```

3.4.2 Parametri

Qui di seguito sono definiti i parametri necessari, specifici per l'addestramento del modello Lleida YOLO11x, da fornire alla funzione di training della libreria Ultralytics:

Proprietà	Tipo	Range	Descrizione
imgsz	int		Dimensione dell'immagine in input, devono essere quadrate, quindi è specificato un solo valore per tutti i lati
device	string	`cpu`, `cuda`, `mps`	Dichiara il tipo di device su cui viene lanciato il training
workers	int		Numero di processi o thread che possono essere eseguiti simultaneamente
save_period	int		Numero di epoch necessari prima di poter salvare i pesi del modello
project	string		Path principale ove sono salvati i dati addestrati
name	string		Nome del progetto, che verrà utilizzato per creare una sottocartella contenente gli output
pretrained	boolean	`true`, `false`	Se true parte da un modello pretrained (transfer-learning)
lr0	float	(1e-5, 1e-1)	Velocità di learning iniziale. Valori bassi rendono il processo di training più stabile, ma allo stesso tempo più lento
lrf	float	(0.01, 1.0)	Velocità di learning finale, specificato come frazione di lr0. Gestisce quanto deve rallentare la velocità di learning durante il training
cos_lr	boolean	`true`, `false`	Se a true lo scheduler della velocità iniziale non cresce linearmente ma segue una curva a coseno. Migliora la convergenza e porta ad una precisione maggiore
cache	string	`none`, `ram`, `disk`	Specifica dove vengono salvati temporaneamente i dati per garantire una maggior velocità di learning
cls	float	(0.2, 4.0)	Moltiplicatore della loss function, utile per controllare l'andamento di training. Se il modello ha difficoltà a riconoscere gli oggetti può essere aumentato.
single_cls	boolean	`true`, `false`	Specificando true il modello considera tutte le annotazioni come un'unica classe

Tabella 3.1: Parametri di training per YOLO11

L'oggetto Python che contiene i parametri è così inizializzato:

```
trainArgs = {
    'data': datasetYamlPath,
    'epochs': 300,
    'imgsz': 640,
    'batch': 16,
```

```
    'device': device,
    'workers': 8,
    'patience': 100,
    'save_period': 5,
    'project': f"{baseFolderPath}/runs/detect",
    'name': 'treeDetectionYolo11Scratch',
    'pretrained': True,
    'lr0': 0.01,
    'lrf': 0.1,
    'cos_lr': True,
    'cache': "disk",
    'single_cls': True
}
```

3.4.3 Funzione di training

```
results = model.train(**trainArgs)
metrics = model.val()
```

Con queste due righe viene lanciata la procedura di training vera e propria.

A riga 1 viene chiamata la funzione *model.train(**trainArgs)* che, grazie ai *train_args* settati in precedenza, inizia il processo di training: i risultati dell'addestramento vengono salvati poi nella variabile *results*.

Nella seconda riga lo script esegue la validazione del modello grazie al validation set, fondamentale per misurare le performance acquisite durante l'addestramento. L'output della funzione restituisce le metriche di valutazione come mAP (mean Average Precision), precision, recall, ecc.

3.4.4 Output della procedura di training

```
Epoch   GPU_mem  box_loss  cls_loss  dfl_loss Instances      Size
1/300     15.3G     1.989     1.919     1.688        63       640: 100%|**********| 54/54
    [00:41<00:00, 1.31it/s]
Class     Images Instances     Box(P         R     mAP50 mAP50-95): 100%|**********| 7/7
    [00:04<00:00, 1.40it/s]
all        218       3262   0.000154   0.00276  7.76e-05  2.15e-05
```

Durante la procedura di training, in console, vengono visualizzate alcune statistiche utili per monitorare l'andamento dell'addestramento.
Sopra riportate vi sono un paio di righe d'esempio corredate da valori forniti in output, sia dalla funzione di training che di validazione.

Statistiche di addestramento:

- *Epoch*: conteggio progressivo degli epoch.
- *GPU_mem*: totale VRAM utilizzata (in questo caso il training è stato fatto con CUDA).
- *box_loss*: misura la distanza tra la posizione e la dimensione delle bounding box previste dal modello rispetto alle bounding box reali (ground truth).
- *cls_loss*: misura la perdita relativa alla classificazione ovvero quanto bene sta identificando gli oggetti il modello.
- *dfl_loss*: miglioramento della regressione della bounding box.
- *Instances*: numero di oggetti processati in questo batch/epoch.
- *Size*: dimensione delle immagini di input.
- *Barra di progresso* %|...|: n/n [00:00:00 z it/s] il numero (n) di batch elaborati in un certo periodo di tempo in secondi a una certa velocità definita come z misurata in batch al secondo.

Statistiche di validazione:

- *Class*: indica per quali classi viene effettuata la validazione.
- *Images*: numero di immagini presenti nel set di validazione.
- *Instances*: numero di alberi presenti nelle immagini del set di validazione.
- *Box(P)*: precisione nella detection della bounding box.
- *R*: recall, vedi sopra

- *mAP50*: vedi sopra
- *mAP50-95*: vedi sopra

3.5 Risultati di training

3.5.1 Grafici

La piattaforma di Ultralytics, dopo aver concluso la fase di training, produce alcuni grafici, che aiutano a visualizzare meglio le performances del modello addestrato. Il grafico forse più significativo è quello relativo alle curva F1:

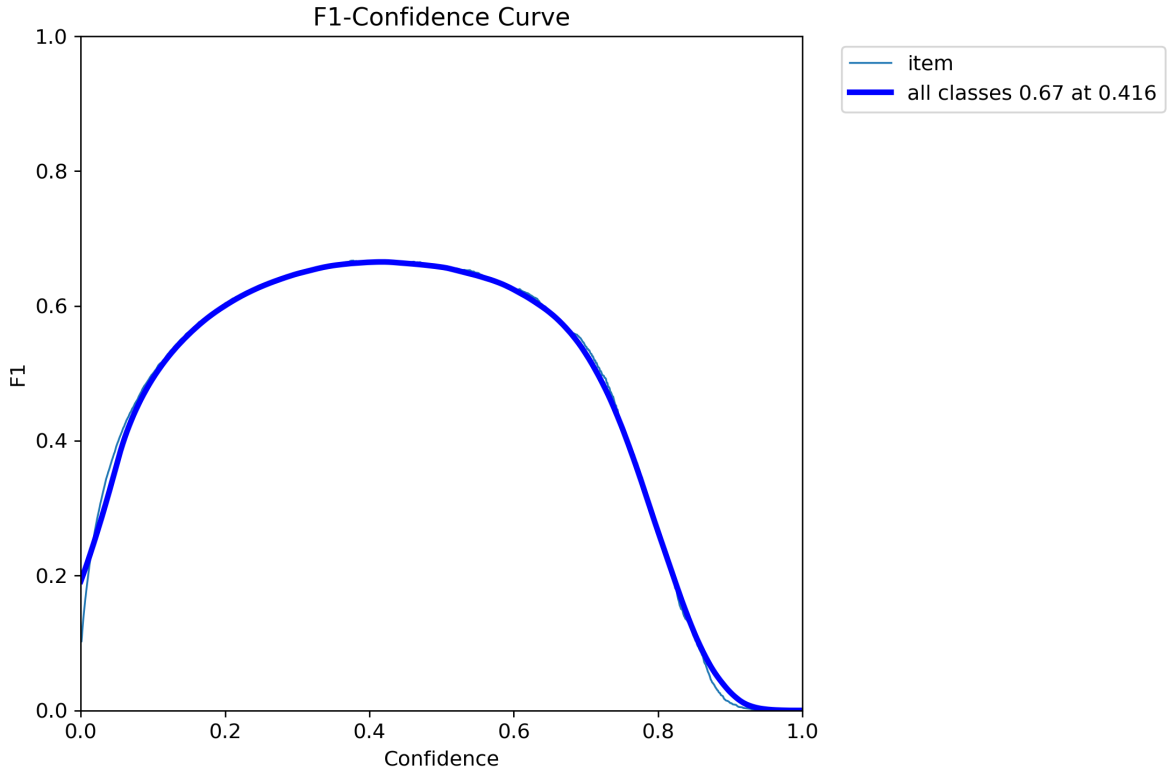

Figura 3.1: F1-Confidence Curve. Sulle ascisse il valore di confidence, sulle ordinate il valore F1.

Il massimo della curva rappresenta il punto ottimale in cui la soglia di confidence fornisce il miglior compromesso tra precision e recall. Il punteggio F1 massimo di 0.67 è ottenuto con confidence 0.416.

Questo grafico è molto importante per scegliere il valore di confidence da fornire in fase di

inferenza, dove appunto si minimizza il più possibile i fenomeni di falso positivo (precision) e falso negativo (recall).

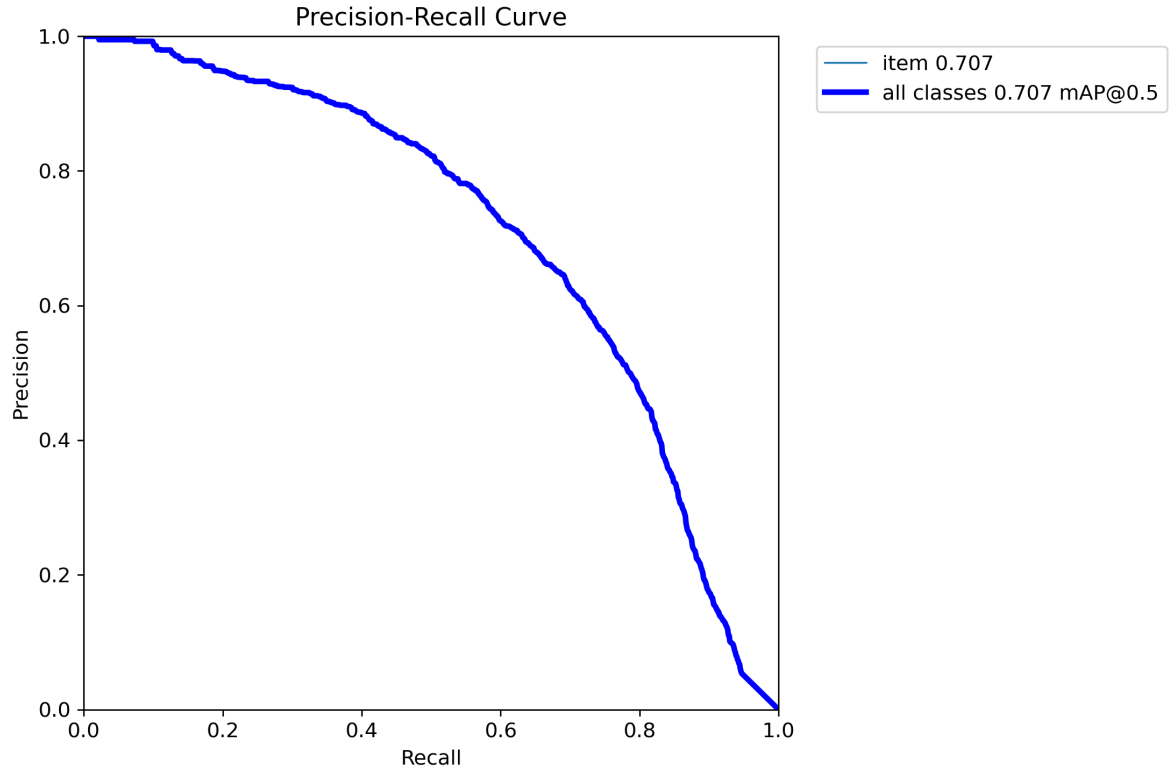

Figura 3.2: Precision-Recall Curve. Sulle ascisse il valore di recall, sulle ordinate il valore di precision.

Questo grafico mostra la relazione fra precision e recall, con il valore di mAP50 è 0.707, e indica la media della precision agli intervalli di recall per tutte le classi, in questo caso solo per gli alberi, alla soglia di IoU di 0.5.

In sostanza evidenzia il bilanciamento tra la capacità del modello di identificare in maniera corretta oggetti (precision) e la capacità di trovarli tutti (recall) per i vari livelli di soglia di confidenza.

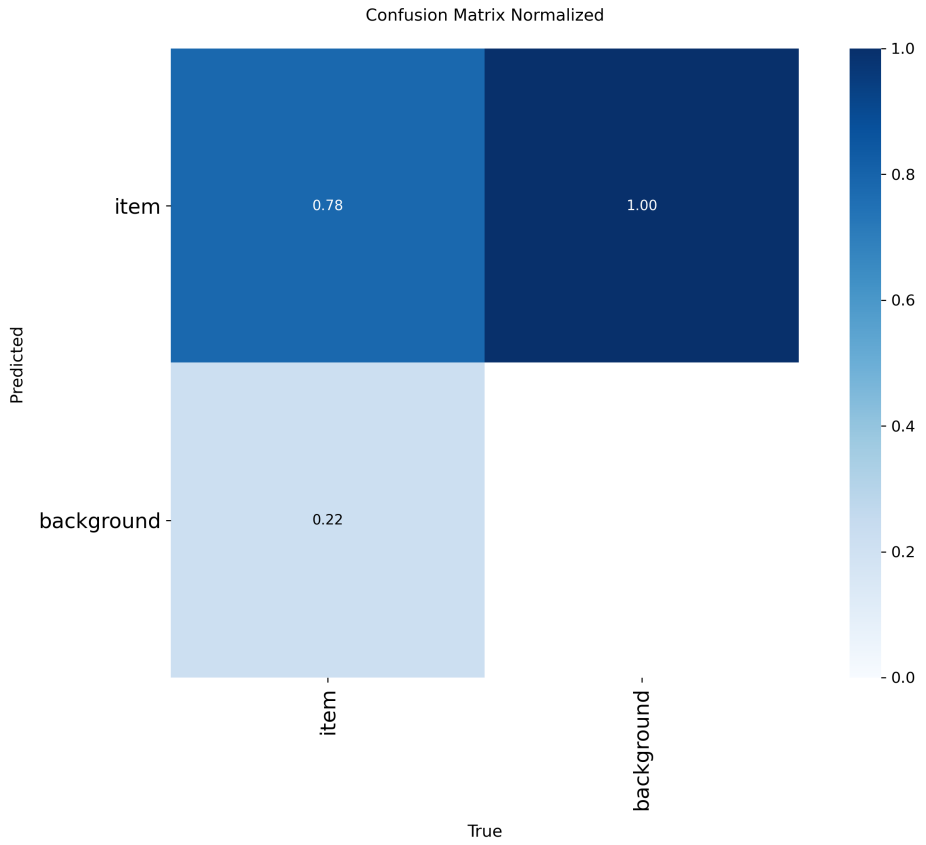

Figura 3.3: Confusion matrix. Sono visualizzate le classi che il modello è in grado di rilevare.

La confusion matrix mostra come il modello classifica le classi su cui è stato addestrato. In questo caso vi è solo una classe "item", ovvero l'albero, mentre "background" è tutto quello che non è considerato come albero. Il grafico è così riassumibile:

- Predicted "item", True "item" (0.78): il modello ha identificato correttamente il 78% degli alberi.
- Predicted "background", True "item" (0.22): Il 22% degli alberi è stato classificato come "background" (falso negativo).
- Predicted "item", True "background" (1.00): Tutto il "background" è stato classificato come tale (vero negativo).
- Predicted "background", True "background" (0.00): Nessun "background" è stato erroneamente classificato come albero (falso positivo assente).

3.6 Inferenza

Dopo aver addestrato il modello con il dataset di Lleida, è possibile procedere con l'inferenza su altre immagini per verificare la presenza di alberi.
Qui di seguito è dettagliata la funzione Python per l'inferenza con YOLO di Ultralytics, tramite la libreria PyTorch, insieme ad OpenCV utilizzato in questo caso per lavorare con le immagini in input e output.
È importante assicurarsi che le immagini in input siano dello stesso formato di quelle preparate per il training, in questo caso 640x640, devono quindi rispettare la risoluzione e la composizione dei channels RGB.

3.6.1 Funzione di analisi predittiva

```
1  def analyze_tree_coverage(imagePath, modelPath, conf=0.05):
2      model = YOLO(modelPath)
3      img = cv2.imread(imagePath)
4      results = model(img, conf=conf)
5
6      for result in results:
7          if result.boxes is not None:
8              boxes = result.boxes.xyxy.cpu().numpy()
9              confidences = result.boxes.conf.cpu().numpy()
10
11             for box, confidence in zip(boxes, confidences):
12                 if confidence >= conf:
13                     x1, y1, x2, y2 = box.astype(int)
14                     cv2.rectangle(img, (x1, y1), (x2, y2), (0, 255, 0), 2)
15
16     outputPath = Path(imagePath).stem + "_result.jpg"
17     cv2.imwrite(outputPath, img)
```

La funzione delegata alla ricerca degli alberi all'interno dell'immagine necessita di tre parametri in input:

- *image_path*: percorso dell'immagine da analizzare
- *model_path*: percorso del modello addestrato da utilizzare per l'inferenza

- *conf*: livello minimo di confidenza entro cui il predittore considera l'oggetto come valido

Le prime quattro righe riguardano l'inizializzazione del modello YOLO, la lettura dell'immagine grazie a OpenCV e la chiamata diretta alla funzione di prediction a riga 4, dove vengono salvati i risultati della detection in *results*.
OpenCV è una libreria molto versatile che offre funzioni e utilities per la computer vision: qui viene utilizzata per leggere l'immagine da file e per rappresentare in overlay i bounding boxes che delimitano gli alberi rilevati [18].
Successivamente vengono analizzati i risultati della predizione uno ad uno: per ogni risultato si verifica se è associato un bounding box, ovvero il rettangolo che delinea l'oggetto trovato all'interno dell'immagine. Se è presente ne estrae le coordinate e la confidence di rilevamento, salvandole rispettivamente in *boxes* e *confidences*.
Ogni box estratto poi viene disegnato, tramite OpenCV, sull'immagine di output, solo se la confidence rilevata supera la soglia specificata come parametro in input.

3.7 Importanza del dataset urbano

Il dataset utilizzato per il training di questo specifico modello YOLO, consta di una serie di immagini satellitari che contengono alberi, localizzati in ambito urbano. Questi dataset sono poco diffusi perché molto specifici. La maggior parte di quelli disponibili online o sulle principali piattaforme riguarda infatti immagini raccolte in contesti rurali. Inoltre, grazie all'ampia quantità di immagini presenti nei dataset rurali, i valori di mAP50, mAP50-95 e F1 risultano generalmente molto più elevati.
Considerando quindi la difficoltà di accedere a questo tipo di dati, perché utilizzare e addirittura addestrare un modello su un dataset urbano quando sono disponibili molti modelli pre-trained? La risposta è meno banale di quello che si pensa, infatti non è rilevante solamente l'oggetto annotato, è importante anche il suo "contesto", come in questo caso l'ambiente urbano circostante.
Il test effettuato è stato molto semplice: eseguire l'inferenza di una specifica immagine sia col modello addestrato con il dataset di Lleida, sia con un modello addestrato da un dataset disponibile su Roboflow, che mostra immagini e annotazioni in contesto extraurbano [19].
I risultati della comparazione, effettuata su un'immagine satellitare del lungadige San Giorgio a Verona, mostrano che il modello addestrato con il dataset di Lleida rileva correttamente la maggior parte degli alberi presenti lungo il fiume, mentre il modello addestrato su foreste identifica

erroneamente solo poche chiome, confondendo la vegetazione urbana con lo sfondo.
Il modello addestrato con dataset contenente immagini urbane performa quindi notevolmente meglio di quello addestrato con immagini di foreste e piantagioni.
Una simile differenza è stata riscontrata anche da ricercatori dell'università di Pechino, dove in un articolo discutono l'uso di YOLOv4-Lite per il rilevamento degli alberi nelle piantagioni urbane e non (come ad esempio frutteti), segnalando difficoltà dovute all'elevato tasso di rilevamenti errati nelle foreste, causati dalla somiglianza tra i colori dello sfondo e quelli della chioma [20].

4

DetecTree

DetecTree è un algoritmo di machine learning che non si limita all'individuazione dei singoli alberi, ma consente anche di identificare intere aree boschive o zone verdi all'interno di immagini satellitari. Come introdotto nel capitolo tecnico in precedenza, non è catalogabile come rete neurale, in quanto l'algoritmo basato sul detector AdaBoost non è composto da nodi e layers ma "semplicemente" da un albero decisionale.

4.1 Principio di funzionamento

Il principio di funzionamento dell'algoritmo DetecTree è riassumibile nei seguenti punti [21]:

- Suddivisione in tile: l'immagine aerea viene suddivisa in tile più piccole, per generare un mosaico di "tessere" di dimensione specifica.
- Scelta delle tile di training: vengono selezionati i tile da usare per l'addestramento tramite dei descrittori GIST, ovvero array numerici che sintetizzano caratteristiche visive e semantiche di una porzione d'immagine.
- Ground truth masks: per ciascun tile scelto per il training, si devono fornire delle maschere binarie "albero/non-albero", con strumenti di editing delle immagini.

- Training: per ogni pixel, viene estratto un array di 27 feature e si addestra un classificatore AdaBoost, che è a tutti gli effetti un classificatore binario, che associa il vettore di feature alle classi "albero/non-albero".
- Testing: procedendo con l'inferenza sui tile di test, la classificazione viene poi raffinata con un algoritmo, migliorando la detection su pixel adiacenti classificati come albero.

L'insieme delle 27 feature è una rappresentazione che viene data ad ogni singolo pixel, dal classificatore AdaBoost in fase di training, per permettere al modello di distinguere i pixel appartenenti o non appartenenti ad un albero. Questo tipo di classificazione viene eseguita in base sia alle informazioni cromatiche che al contesto dell'immagine.
Le feature si distinguono in:

- 6 di colore: contengono dati sul colore del pixel, come ad esempio ai canali RGB e possibili combinazioni derivate.
- 18 di texture: sono misure statistiche (es. media, varianza, contrasto, ecc.) calcolate su regioni locali attorno al pixel, spesso derivate da modelli.
- 3 di entropia: misura l'imprevedibilità dei valori di intensità in una specifica regione attorno al pixel.

4.2 Tipologia di modello utilizzato

In questo caso la repository del progetto github mette a disposizione il modello pre-trained, in modo da poter essere utilizzata direttamente senza dover ricorrere alla procedura di addestramento ma, soprattutto, senza dover ricorrere alla annotazione manuale di immagini per il dataset di training.

4.3 Parametri

Nel caso di DetecTree, oltre all'immagine che evidenzia la copertura arborea, è disponibile anche il valore percentuale della copertura presente all'interno dell'immagine analizzata.

4.4 Inferenza

```
import skops.io as sio

untrusted_types = sio.get_untrusted_types(file="detectree_model.skops")
model = sio.load("detectree_model.skops", trusted=untrusted_types)

clf = dtr.Classifier(clf=model)
pred = clf.predict_img(image_path)
```

Queste cinque righe di codice sono responsabili del caricamento del modello e dell'inferenza con DetecTree.

Per poter classificare l'immagine, è necessario prima di tutto importare la libreria Skops [22], che è una libreria Python utilizzata per importare i modelli basati su Scikit-learn, celebre piattaforma open source per il machine learning in linguaggio Python.

Il modello sarebbe potenzialmente importabile direttamente da Scikit, ma non è consigliabile procedere in tal senso, in quanto la libreria sfrutta un modulo standard di Python (Pickle) per serializzare e deserializzare i modelli, che può eseguire codice arbitrario durante la deserializzazione, esponendo a potenziali attacchi l'host sul quale è in esecuzione.

Nelle righe successive vengono caricati i types del modello che definiscono quale struttura o classe ha l'oggetto (come ad esempio un modello, un vettore o un dato correlato). In questo caso viene scelto di caricare tutti i types, anche quelli che non sono automaticamente considerati sicuri per essere caricati: anche caricando tipi considerati "non verificati", rimane comunque la protezione che Skops offre contro gli attacchi di deserializzazione di Pickle.

Caricato il modello si passa alla classificazione, eseguendo la predizione con *predict_img*, che restituisce la mappa di etichette (ad es. array 2D di classi) per l'immagine di input.

4.5 Falsi positivi

L'utilizzo di AdaBoost introduce comunque una criticità non trascurabile in quanto, lavorando per similarità, non solo è poco preciso a distinguere tra alberi o prati, ma è fortemente penalizzato nel caso in cui le aree verdi sono vicine ad altre zone con caratteristiche simili, come ad esempio i corsi d'acqua.

In questo caso, infatti, DetecTree fatica a delimitare i confini delle aree verdi, andando a classificare erroneamente aree che non dovrebbe considerare come boschive.

Un esempio significativo di questo problema è riscontrabile nell'area di ponte Aleardi a Verona, nei pressi dei giardini del cimitero monumentale: l'algoritmo classifica erroneamente come area arborea una porzione delle acque del fiume Adige, a causa della vicinanza cromatica con le chiome degli alberi di Lungadige Porta Vittoria.

5

Detectree2

Questo algoritmo di deep learning non è una diretta evoluzione di DetecTree, in quanto l'implementazione non è derivata da AdaBoost ma è invece una vera e propria rete neurale convoluzionale region based (R-CNN) [23].
Alla base di Detectree2 vi è Detectron2, una piattaforma avanzata ideata dal FAIR (Facebook AI Research), costruita sul framework PyTorch [24].

5.1 Principio di funzionamento

Detectree2 ha l'obiettivo di rilevare, all'interno di un immagine satellitare, gli alberi contenuti in essa cercando l'insieme dei rami e delle foglie che si trovano nella parte superiore del tronco, ovvero la corona.
Per poterlo fare è stato adattato l'algoritmo R-CNN di Detectron2, che ha modelli pretrainati su cui è possibile effettuare transfer learning ed "aggiungere" quindi la conoscenza necessaria per poterli impiegare nel rilevamento arboreo.

5.1.1 Detectron2

Alla base di Detectron2 vi è una R-CNN-FPN, ovvero una rete convoluzionale region based con Feature Pyramid Network, che utilizza una piramide di feature map a più risoluzioni. Questo permette di migliorare il rilevamento degli oggetti, estraendo rappresentazioni dell'immagine a

diversi livelli di dettaglio, facilitando così il riconoscimento di oggetti di diverse dimensioni.
Una feature map è una rappresentazione prodotta da uno strato convoluzionale di una CNN: durante l'elaborazione, la rete applica dei filtri ai dati in input per rilevare specifiche caratteristiche, come ad esempio bordi, texture o pattern particolari.
Schematicamente l'architettura della rete è riassumibile in tre blocchi [25]:

- Backbone Network: estrae le feature map dall'immagine di input con differenti risoluzioni tramite FPN.
- Region Proposal Network (RPN): analizza la posizione delle feature map per rilevare le regioni contenenti gli oggetti, restituendo bounding box e valori di confidence.
- Box Head: taglia e ridimensiona le feature map corrispondenti alle regioni rilevate, elaborandole per classificare l'oggetto ed affinare ulteriormente la posizione delle bounding box.

5.1.2 Modifiche introdotte da Detectree2

Detectree2 aggiunge alla libreria di Detectron2 la funzionalità di gestire input e output georeferenziati e di delineare le corone degli alberi in maniera individuale, tramite la generazione di maschere che circoscrivono esattamente l'oggetto nell'immagine.
Si incrementa così la precisione e le prestazioni nella segmentazione delle chiome, ottenendo quindi un miglioramento rispetto alla detection offerta da Detectron2.

5.2 Tipologia di modello utilizzato

Anche in questo caso, come per DetecTree, la repository del progetto GitHub mette a disposizione più modelli pre-trained [26].
I modelli sono stati addestrati con diversi dataset, come specificato nel readme della repo, è necessario selezionare quindi il modello più affine al tipo di alberi da rilevare. Nel caso specifico di questa ricerca, il modello utilizzato è il *urban_trees_Cambridge20230630* che è stato appunto addestrato per rilevare corone arboree in ambito urbano.

5.3 Parametri

I parametri per Detectree2 sono quelli comuni alle reti neurali convoluzionali, come quelli già visti per YOLO. Infatti il modello Cambridge, riporta fra le sue statistiche alcuni parametri che sono stati introdotti nel capitolo dei dettagli tecnici:

- Learning rate: 0.01709
- Workers: 6
- Batch size: 623
- AP50: 62.0

In virtù delle considerazioni viste nei capitoli precedenti, un AP50 di 62.0 può essere considerato un buon risultato: indica che il modello è in grado di localizzare gli oggetti per il quale è stato addestrato, con un buon equilibrio tra precisione e recall.

5.4 Inferenza

Dopo aver scaricato il modello appropriato, in questo caso il Cambridge, è possibile procedere con l'inferenza.
Secondo i requisiti elencati sulla repository GitHub, perché la detection sia precisa, è necessario che le immagini abbiano lo stesso formato di quelle utilizzate in fase di training, nelle quali sono state usate tiles quadrate che coprono c.a. 200m. Dopo una rapida stima, considerato il rapporto coordinate/pixels delle immagini fornite dal provider e tenuto conto anche del livello di zoom impostato, la risoluzione corrisponde circa a 364x364. A questo parametro si aggiunge anche, come requisito fondamentale, l'ordine dei canali che deve essere BGR.
Qui di seguito sono elencate alcuni parti di codice dedicate all'inferenza con questo modello specifico.

5.4.1 Configurazione iniziale

```
1 from detectron2.engine import DefaultPredictor
2 from detectree2.models.train import setup_cfg
3
```

```
4  def init(modelPath: str, outputFolder: str):
5      cfg = setup_cfg(update_model=modelPath)
6      cfg.OUTPUT_DIR = outputFolder
7      cfg.MODEL.DEVICE = "cpu"
8
9      predictor = DefaultPredictor(cfg)
```

Nella prima parte di codice è fondamentale inizializzare correttamente Detectron2 e Detectree2. Dato che il primo sfrutta la struttura e la rete del secondo, salvo alcune feature già elencate in precedenza, è necessario importare il *DefaultPredictor* di Detectron2 e la configurazione dei modelli dei Detectree2, come è visibile dalle prime due righe del listato.
Nelle righe successive viene inizializzata la configurazione del modello, specificando come parametri il percorso del modello definito in *update_model*, insieme al percorso della directory di output e al tipo di dispositivo da utilizzare per l'inferenza.
L'ultima riga invece riguarda l'inizializzazione del *DefaultPredictor* a partire dalla configurazione dichiarata per Detectree2.

5.4.2 Prediction

```
1  import cv2
2  from shapely.geometry import Polygon
3
4  def predict(
5          predictor: DefaultPredictor,
6          imagePath: str,
7          conf: float = 0.5) -> Dict[str, Any]:
8      image = cv2.imread(imagePath)
9      outputs = predictor(image)
10
11     instances = outputs["instances"].to("cpu")
12     scores = instances.scores.numpy()
13     validIndices = scores >= conf
14     filteredScore = scores[validIndices].tolist()
15
16     polygons = []
17     if hasattr(instances, 'pred_masks'):
```

```
        masks = instances.pred_masks.numpy()[validIndices]

        for mask in masks:
            contours, _ = cv2.findContours(mask.astype(np.uint8), cv2.RETR_EXTERNAL,
                cv2.CHAIN_APPROX_SIMPLE)
            if contours:
                largestContour = max(contours, key=cv2.contourArea)
                polygonCoords = largestContour.reshape(-1, 2).tolist()

                if len(polygonCoords) >= 3:
                    try:
                        crownGeometry = Polygon(polygonCoords).simplify(0.3,
                            preserve_topology=True)
                        if crownGeometry.is_valid and hasattr(crownGeometry, 'exterior'):
                            exterior_coords = crownGeometry.exterior.coords
                            coords_without_last = exterior_coords[:-1]
                            readable_coords = []
                            for x, y in coords_without_last:
                                readable_coords.append([float(x), float(y)])
                        else:
                            polygons.append([])
                    except:
                        polygons.append([])

    results = {
        "polygons": polygons,
        "scores": filteredScore,
        "num_detections": len(filteredScore)
    }

    return results
```

Questa funzione è utilizzata principalmente per lanciare l'inferenza sull'immagine e trovare i poligoni che compongono le corone degli alberi. Viene restituito al chiamante della funzione un dictionary, contenente poligoni, punteggio di prediction e numero di corone rilevate.
Per prima cosa viene caricata l'immagine da OpenCV, che la trasforma in un array multidimensionale in formato *ndarray* di NumPy, ovvero la libreria Python per eccellenza per l'elaborazione

di array multidimensionali e calcolo numerico [27]. Questo permette all'immagine di essere trasformata in un dato che ha una shape, quindi una forma derivata dall'immagine, corredata da uno specifico tipo di dato chiamato dtype. Più in generale, nell'ambito del machine learning, questa matrice a più dimensioni è chiamata anche tensor.
Successivamente è lanciata la prediction grazie al *predictor* inizializzato nell'init, proprio sull'immagine in formato matriciale, che restituisce un oggetto *instances* contenente i seguenti parametri rilevati dall'immagine:

- *pred_masks*: matrice multidimensionale (tensor) di forma (N, H, W) dove N corrisponde al numero di elementi trovati, H altezza e W larghezza, che rappresenta una maschera per ogni oggetto trovato, quindi per ogni corona. Ogni maschera evidenzia i pixel che appartengono a quel preciso albero rilevato.
- *pred_boxes*: Bounding boxes per ogni oggetto trovato, come coordinate del rettangolo che circonda la maschera, equivalente a YOLO.
- *scores*: Punteggio di confidenza per ogni albero rilevato.
- *pred_classes*: Classe di ogni oggetto rilevato, in questo caso alberi dato che Detectree2 si limita a quelli.

il *.to("cpu")* applicato sull'oggetto contenente le istanze rilevate, converte i tensors in un formato elaborabile da una cpu.
Nelle righe 15, 16 e 16 viene applicato il filtro sulla confidence, tutto quello che non è uguale o superiore a quella soglia viene automaticamente scartato, in quanto l'array *filteredScore* contiene solamente dati relativi agli alberi rilevati con una confidenza tollerata.

Lo step successivo è poi fondamentale per estrarre le maschere degli elementi considerati validi, in quanto superiori alla soglia.
Ogni maschera rilevata da Detectree2 viene trasformata in un poligono, grazie alla funzione *findContours* di OpenCV, valutato poi quale è il più ampio (l'albero potrebbe essere stato identificato con chiome di dimensione diversa) e controllato che sia veramente un poligono con almeno tre punti per evitare di considerare erroneamente corone anche i punti o le rette, ne vengono estratte le coordinate grazie a Shapely, che lo rappresenta con la sua struttura di tipo *Polygon*.

Shapely è una libreria Python che permette la manipolazione e l'analisi di oggetti geometrici, come punti, linee e poligoni [28].
Sul poligono viene anche applicato un ulteriore metodo, *simplify(0.3, preserve_topology=True)* che è fondamentale per ridurre il numero dei vertici del poligono, mantenendo però la forma dello stesso. Questo va a ridurre il peso dell'output, che potrebbe essere veramente significativo nel caso in cui non ci fosse nessun filtro sui vertici.
Una volta isolata la corona, si procede con l'ultima fase che consiste nel parsing delle coordinate esterne del poligono *crownGeometry*. In questo passaggio viene rimosso l'ultimo punto, che coincide con il primo per garantire la chiusura del poligono (*coords_without_last*), e viene generata una lista di coordinate rappresentate come coppie di valori decimali. La funzione restituisce così le corone identificate con i corrispondenti score.

5.5 Confronto con DetecTree

DetecTree e Detectree2 sono due soluzioni ideate per isolare, nelle immagini satellitari urbane e non, aree boschive o anche singoli alberi piantumati in parchi cittadini. Pur condividendo il nome e il target di oggetti (alberi) ai quali questi modelli sono dedicati, non utilizzano la stessa tipologia di rete ne lo stesso stack tecnologico.
L'obiettivo di questa ricerca è quello di identificare gli alberi, in entrambi i casi è possibile farlo, ma soltanto nel caso di Detectree2 possono essere isolati singolarmente, senza confonderli con altre aree verdi, come riesce ad esempio a fare YOLO.
In maniera molto schematica e tabellare è possibile confrontare i due modelli, tenendo conto delle definizioni e dei dettagli tecnici discussi finora:

Feature	**DetecTree**	**Detectree2**
segmentazione	semantica, distingue tra i pixel albero e non albero	oggettiva, individua le corone degli alberi
modello	classificatore tradizionale (AdaBoost)	Deep Learning (R-CNN)
inputs	RGB	RGB + immagini multispettro
outputs	mappa di copertura arborea, percentuale di copertura rispetto all'immagine	poligoni delle corone degli alberi, classi multiple
utilizzo	analisi copertura arborea, pianificazione urbana	studio individuale degli alberi

Tabella 5.1: Confronto tra DetecTree e Detectree2

6

SegFormer

L'avvento delle moderne LLM ha introdotto numerose innovazioni nel campo del machine learning: sebbene progettate principalmente per il linguaggio naturale, hanno portato caratteristiche applicabili anche alla computer vision, come nel caso dei transformer.
Questi costituiscono la base di un algoritmo di deep learning chiamato SegFormer, portmanteau di "segmentation" e "transformer", che utilizza i transformer per classificare i pixel degli oggetti all'interno delle immagini tramite quella che, come illustrato nel secondo capitolo, viene definita architettura vision transformer (ViT) [29].

6.1 Principio di funzionamento

SegFormer è un algoritmo che utilizza i transformer per isolare gli oggetti, ma non si limita soltanto a quello: effettua una vera e propria segmentazione semantica, ovvero assegna una classe a ciascun pixel dell' immagine, in questo caso "tree" e "background".
Il transformer implementato in questo specifico algoritmo è una variante del ViT classico, che prende il nome di MixTransformer (MiT). In aggiunta al MiT è anche presente un decoder, chiamato Multi-layer Perceptron (MLP) che è fondamentale per la segmentazione semantica delle immagini.
Il funzionamento può essere così riassunto:

- MixTransformer è il backbone di SegFormer, ovvero la parte di rete neurale che estrae

tutte le feature dai dati in input, ovvero le immagini. Il MiT, che in pratica è l'encoder della rete, implementa una struttura gerarchica che divide l'immagine in parti (patch) e poi le elabora con blocchi transformer.

- Ogni stage del MiT produce delle feature map che hanno risoluzione e profondità diverse, come visto in precedenza per DetecTron2, mantenendo una struttura spaziale che ha complessità ridotta rispetto al ViT che invece lavora direttamente sull'intera immagine.
- Il decoder di SegFormer è un MLP, che aggrega le feature multi-scale ottenute dal MiT durante la fase di encoding, fornendo una rappresentazione completa pixel per pixel. Vengono così combinate sia il contesto globale (disposizione degli oggetti nell'immagine, chi sta "davanti", chi sta "dietro" in secondo piano) che le informazioni locali (bordi degli oggetti).
- L'aggregazione avviene senza convoluzioni, dato che il decoder impara a pesare e combinare le informazioni ricevute dal backbone, rendendo le predizioni più robuste e precise.

Appreso il funzionamento di SegFormer sarebbe facile paragonare questo tipo di algoritmo alle CNN o R-CNN, già incontrate nei capitoli precedenti, in fin dei conti processano immagini e le trasformano in numeri, come ad esempio tensors o matrici. La vera differenza è come vengono elaborate le informazioni spaziali e contestuali dell'immagine: le CNN usano filtri convoluzionali ed estraggono caratteristiche e oggetti da immagini, analizzando piccole porzioni della stessa e seguendo una gerarchia spaziale, mentre i ViT suddividono l'immagine in patch e le trattano come se fossero token indipendenti, simili al modo in cui le LLM per il linguaggio naturale trattano le parole. In sostanza le CNN sono più efficaci ad estrarre oggetti locali evidenziandoli, come può essere ad esempio la singola corona di un albero, mentre i trasformer sono più efficaci a cogliere il contesto complessivo, utilizzando meccanismi per modellare le relazioni globali tra tutte le patch.
Proprio per questo motivo SegFormer non è in grado di distinguere un singolo albero da un altro, isolandone la corona, ma sa riconoscere dove sono gli alberi in maniera estremamente efficace [30].

6.2 Tipologia di modello utilizzato

In rete è possibile scaricare diversi modelli di SegFormer, con varianti più o meno sviluppate di encoder e decoder, adatti ad isolare oggetti di vario genere all'interno delle immagini. Alcuni

sono anche linkati e disponibili dalla repo ufficiale su NVlabs, il progetto di ricerca di Nvidia sull'intelligenza artificiale [31].
I modelli reperibili e addestrabili sono in formato pt, quindi possono essere caricati con PyTorch come già visto nei capitoli precedenti.

6.2.1 Versione del modello

All'inizio del capitolo abbiamo visto che SegFormer è formato da encoder e decoder. Non esiste soltanto però una tipologia di encoder MiT, ma sono disponibili diverse versioni e revisioni che nel tempo sono state rilasciate per aumentare la precisione e la scalabilità del modello, in questo caso il modello di partenza è di Nvidia e prende il nome di *mit-b3*.
Il MiT viene rilasciato infatti in diversi tagli, da b0 a b5, dove b3 indica una capacità intermedia: all'aumentare della versione di b, aumentano anche i parametri, la profondità e la capacità rappresentativa.
È doveroso anche precisare che il termine "modello di partenza" è importante, poichè esso non viene usato as-is, che potrebbe anche funzionare se ci trovassimo nel caso in cui la necessità fosse quella di individuare varie tipologie di oggetti, ma viene utilizzato per creare un nuovo modello dopo averlo addestrato con un dataset dedicato in quanto, come in precedenza visto per YOLO, è necessario effettuare un training specifico su un dataset urbano per far si che le uniche due classi da riconoscere siano background e tree.

6.3 Preparazione del dataset

Data la relativamente recente implementazione di SegFormer, non è semplice acquisire dataset ben formati per il training del modello, soprattutto se sono specifici come possono essere richiesti da questa ricerca. In particolare, visti i requisiti legati agli alberi fotografati da satellite in tessuto urbano, l'idea è stata quella di utilizzare il dataset YOLO, che risponde a tutte le caratteristiche necessarie, convertendolo per addestrare un modello SegFormer.
Il concetto è quello di utilizzare le tiles originali, con l'unica differenza di sostituire i files contenenti le coordinate dei bounding boxes con delle maschere, in modo da far risultare il dataset compatibile con SegFormer. Per ogni immagine del dataset viene quindi generata una corrispondente maschera binaria, dove l'area coperta dalle annotazioni YOLO viene evidenziata.

L'area della maschera è appunto bianca, ma lo è soltanto per mettere in evidenza a scopo didattico qual è l'area che viene considerata come "coperta" da alberi. In realtà, in questo caso specifico, il colore dei riquadri sarebbe quasi impercettibile: SegFormer può essere addestrato per riconoscere più oggetti contemporaneamente, ad ogni classe di oggetto deve essere assegnato un colore, il nero è comunemente considerato come colore di sfondo, mentre nel caso degli alberi è assegnato un colore molto vicino al nero, quasi impercettibile ad occhio nudo, che rappresenta la prima classe. Questi colori non sono vere e proprie tinte RGB, ma valori numerici (interi) in una immagine in scala di grigi o in formato indicizzato, che SegFormer interpreta come etichette delle classi.

6.3.1 Convertitore del dataset YOLO

La conversione del dataset può essere effettuata con un semplice script Python, che si occupa di creare le maschere partendo dal path del dataset YOLO, a patto che sia correttamente organizzato nelle cartelle val, train e test con immagini e file che definiscono le bounding boxes, quindi che contengono le annotazioni.

```
import os
from PIL import Image
import numpy as np

def create_segmentation_mask(boxes, img_width, img_height, num_classes=1):
    mask = np.zeros((img_height, img_width), dtype=np.uint8)

    for i, (class_id, x1, y1, x2, y2) in enumerate(boxes):

        orig_coords = (x1, y1, x2, y2)
        x1 = max(0, x1)
        y1 = max(0, y1)
        x2 = min(img_width, x2)
        y2 = min(img_height, y2)

        if x2 <= x1 or y2 <= y1:
            continue

        mask_value = int(class_id) + 1
```

```
        if mask_value > num_classes:
            mask_value = 1

        mask[y1:y2, x1:x2] = mask_value
        actual_value = mask[y1, x1] if y1 < img_height and x1 < img_width else -1

    return mask
```

Alla funzione vengono passati diversi parametri:

- *boxes*: è la lista dei bounding boxes definiti per l'immagine, nel formato [*class_id*, *angolo_alto_sx_x1*, *angolo_alto_sx_y1*, *angolo_basso_dx_x2*, *angolo_basso_dx_y2*]
- *img_width*: larghezza dell'immagine in input.
- *img_height*: altezza dell'immagine in input.
- *num_classes*: numero di classi, valore intero che deve essere settato a 1 nel caso in cui la classe sia unica, come in questo.

Utilizzando Pillow [32], libreria per la manipolazioni delle immagini, viene inizializzata un'immagine vuota della risoluzione pari a quella dell'immagine di origine. Successivamente per ogni bounding box si crea, a partire dalle coordinate di origine in alto a sinistra e in basso a destra, la relativa maschera controllando però se è formata da un quadrato valido o le coordinate sono errate .
Successivamente viene riempita l'area della bounding box, tenendo conto della relative classe dichiarata nella annotazione di YOLO. In questo caso specifico il *class_id* potrà essere soltanto 1, in quanto esiste solo un tipo di oggetto annotato che corrisponde all'albero, mentre lo zero è utilizzato per il background.
Infine la funzione ritorna la maschera di segmentazione sotto forma di array NumPy, che rappresenta l'immagine della maschera. Dalla maschera in formato NumPy è poi possibile salvarla in un immagine rgb grazie a OpenCV *cv2.imwrite(str(outputPath), mask)*.

6.4 Script di training

Come per YOLO, per ridurre i considerevoli tempi di addestramento dei modelli SegFormer, l'idea è stata quella di ricorrere alla tecnica del transfer learning, soprattutto per mit-b4 e mit-

b5.

In questo caso è obbligatorio dotarsi di una GPU con supporto a CUDA, qualsiasi altra soluzione che non ne preveda l'uso potrebbe rivelarsi veramente poco efficiente.

6.4.1 Init

```
import os
import numpy as np
import argparse
from PIL import Image
from transformers import (
    SegformerForSemanticSegmentation,
    SegformerImageProcessor,
    pipeline
)

image_processor = SegformerImageProcessor.from_pretrained("nvidia/mit-b3")
```

Oltre ai già noti OpenCV, NumPy e Torch, queste funzioni si basano sull'utilizzo delle librerie di Hugging Face. *SegformerForSemanticSegmentation* e *SegformerImageProcessor* sono parte della libreria Hugging Face transformer, permettono di addestrare e di utilizzare il modello SegFormer. Hugging Face è una piattaforma molto nota nella community open source sull'intelligenza artificiale, soprattutto per la sua libreria transformer e il suo Model Hub, dove gli sviluppatori possono condividere modelli preaddestrati [33].

Infine è dichiarato come *image_processor* il pre-processore di Hugging Face ottimizzato per il modello SegFormer da usare, ovvero il già citato *nvidia/mit-b3*. Il processore effettua sulle immagini in input una serie di operazioni, come ad esempio:

- trasformazione dell'immagine dal formato originale, in questo caso NumPy, a quello Tensor compatibile con PyTorch.
- normalizzazione effettuata sui canali RGB dell'immagine in input per essere uniforme al formato utilizzato nel modello preaddestrato.

6.4.2 Parametri

```
model = SegformerForSemanticSegmentation.from_pretrained(
            "nvidia/mit-b3",
            num_labels=self.numClasses,
            ignore_mismatched_sizes=True
        )

training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=40,
    per_device_train_batch_size=2,
    eval_strategy="epoch",
    remove_unused_columns=False,
)
```

L'array *treeClasses* contiene le classi per la segmentazione arborea, in questo caso solamente background e albero, con i relativi id numerici e i colori da associare a ciascuna classe per produrre la visualizzazione delle maschere in overlay (se implementato).
Sono poi presenti, un po come già visto per il training del modello YOLO, i parametri di training:

- *model*: È il modello, in questo caso pre-trained, per la segmentazione semantica caricato con la funzione dedicata della libreria di Hugging Face.
- *num_train_epochs*: Numero di epochs di training.
- *per_device_train_batch_size*: Numero di immagini (o campioni) processati in contemporanea.
- *eval_strategy*: Parametro che indica quando effettuare la fase di validation. Se specificato "epoch", viene effettuata al termine di ogni epoch.
- *output_dir*: Percorso di output dove vengono salvati i modelli PyTorch addestrati, al termine di ogni epoch.

6.4.3 Funzione di training

```
def trainModel():
    trainer = Trainer(
```

```
        model=model,
        args=training_args,
        train_dataset=trainingSet
    )

    train_result = trainer.train()
    trainer.save_model()
```

Questa funzione è responsabile del training vero e proprio del modello, che avviene grazie alle librerie di Hugging Face.
Per prima cosa inizializza il *Trainer*, una classe di alto livello che gestisce il training del modello, inclusi:

- Forward e backward pass del modello.
- Calcolo del loss e ottimizzazione della loss function.
- Valutazione periodica delle metriche.
- Salvataggio dei checkpoint al termine di ogni epoch e alla fine del training.
- Logging e monitoraggio in console delle info sull'addestramento.

Il forward pass è il processo di propagazione dell'input attraverso la rete, per arrivare all'output di uscita, in questo caso specifico è la predizione di copertura arborea.
Il backward pass è successivo alla prediction, si calcola la funzione di loss che misura l'errore fra predizione e valore reale mappato nella maschera. Così facendo vengono aggiornati i pesi a ritroso, tramite un ottimizzatore.

Con *trainer.train()* viene lanciato il training, i risultati vengono poi copiati in *train_result* e successivamente *trainer.save_model()* salva il modello nel path specificato per l'output. Questo modello può essere poi utilizzato successivamente per l'inferenza.

6.4.4 Output della procedura di training

```
epoch Training Loss Validation Loss Mean Iou Mean Accuracy Overall Accuracy Per Category Iou
    Per Category Accuracy
```

```
1 0.988000 0.408625 0.712276 0.807701 0.914090 [0.9053025529456493, 0.5192490975926009]
    [0.955735232841766, 0.6596663623426008]
```

La procedura produce, come per l'addestramento di YOLO una serie di valori per ogni epoch, che indicano l'andamento del training del modello.

Training Loss e *Validation Loss* misurano la qualità di ottimizzazione del modello. Bassi valori di loss indicano che l'apprendimento in fase di training e la validazione sono buoni.

Mean Iou è la media delle IoU (Intersections over Union) calcolate per ogni classe. Come visto per YOLO, indica il rapporto dell'area di sovrapposizione fra quella inferita dalla predizione del modello e il ground truth. Il risultato è migliore con un valore alto.

Mean Accuracy media dell'accuracy per ciascuna classe, mentre *Overall Accuracy* accuratezza globale su tutti i pixel dell'immagine.

Per Category Iou e *Per Category Accuracy* mostrano un array con i valori per ogni classe di IoU e Accuracy. In questo caso vengono mostrate due classi, "background" e "tree".

6.5 Inferenza

Una volta addestrato il modello è possibile procedere all'inferenza sulle immagini satellitari. La procedura di inferenza è anch'essa realizzata grazie alle librerie di Hugging Face, che in questo caso facilitano l'operazione di predizione, ma non sono strettamente necessarie: il modello prodotto in ogni caso ha un formato che può essere caricato da PyTorch, quindi per utilizzarlo è anche possibile procedere con l'implementazione direttamente pubblicata da Nvidia, oppure con altre librerie di terze parti.

In questo caso, per coerenze e praticità, è mostrato il codice che sfrutta le librerie Hugging Face.

6.5.1 Funzione di analisi predittiva

```python
def imagePrediction(image_path: str, model_path: str, confidence: float):
    image = Image.open(image_path).convert("RGB")
    processor = SegformerImageProcessor.from_pretrained("nvidia/mit-b3")
    model = SegformerForSemanticSegmentation.from_pretrained(model_path)

    inputs = processor(image, return_tensors="pt")
    with torch.no_grad():
```

```
8           outputs = model(**inputs)
9
10      logits = outputs.logits
11
12      probabilities = torch.nn.functional.softmax(logits, dim=1)
13
14      tree_prob = probabilities[0, 1].cpu().numpy()
15      mask = (tree_prob > confidence).astype(np.uint8)
```

La funzione di predizione accetta in input due parametri, *image_path* e *model_path*, rispettivamente percorso dell'immagine e del modello addestraato. Inoltre è anche richiesto di specificare la soglia di confidenza che, come per gli altri modelli visti finora, è il limite entro il quale un oggetto rilevato viene considerato buono o meno.
Successivamente è necessario, come in precedenza per il dataset, preprocessare le immagini grazie al *SegformerImageProcessor*, dove anche in questo caso va specificato il "modello di origine" che stabilisce il formato dei dati. Grazie a questo preprocessing si ha un grande vantaggio perchè è possibile fornire in input immagini anche di risoluzione diverse, dato che sarà poi il processor a standardizzarle.
Il vero e proprio caricamento del modello addestrato avviene a riga quattro, e alla riga successiva viene poi effettvamente processata l'immagine.
L'inferenza è lanciata sull'immagine grazie alla chiamata che la libreria espone, direttamente nella classe che gestisce il modello *model(**inputs)* e fornisce in output alcune informazioni importanti:

- *.logits*: scores grezzi della prediction.
- *.loss*: valore della loss function.
- *.attentions*: attention weights, ovvero i valori numerici che vengono assegnati ai soggetti "osservati" dal modello, ad esempio per un albero potrebb essere 0.8 per le foglie, 0.6 per il tronco. Più è vicino a 1, più è importante il valore.

In questo specifico caso per definire la maschera che isola gli alberi sono sufficienti gli scores, che vengono passati ad una funzione di PyTorch detta di attivazione. Questa particolare funzione, la Softmax, prende i dati grezzi degli scores che assegano un punteggio al pixel, come gli attention

weights e viene convertito in una probabilità, ovvero più punteggio viene totalizzato da un pixel, più ha probabilità di essere un albero.
Le ultime due istruzioni sono quelle che estraggono la maschera binaria contenente il risultato di predizione "albero" o "non albero", estraendo le probabilità della classe "albero" con NumPy e scartando quelle che sono al di sotto di una determinata soglia di confidenza.

6.5.2 Qualità della prediction

Per poter visualizzare il risultato della predizione è possibile poi, con OpenCV andare a sovrapporre la maschera all'immagine originale. L'implementazione non è qui riportata ma è reperibile, con qualche variazione, sulla repo originale di SegFormer nei tools e nelle demo.
È significativo vedere il risultato della predizione che viene effettuata da SegFormer, infatti la rappresentazione grafica mette in evidenza tutti i dati del modello quali la copertura arborea, con la relativa maschera binaria che va a delineare i contorni delle zone urbane piantumate e anche la probabilità, sotto forma di heatmap che rappresenta la distribuzione di probabilità del modello.
È interessante notare che, a differenza di DetecTree, vengono evidenziate le aree coperte da alberi ma non per la semplice vicinanza di colore dei pixel come visto appunto con l'algoritmo AdaBoost: in questo caso nemmeno le ombre degli alberi stessi, che si stagliano sul fiume Adige, sono incluse nella maschera.
Il risultato della predizione include la maschera binaria che delinea i contorni delle zone piantumate e una heatmap che rappresenta la distribuzione di probabilità del modello, permettendo di visualizzare con precisione le aree identificate come copertura arborea.

6.6 Applicazione di watershed

SegFormer è molto efficace nell'isolare gruppi di alberi all'interno di un'immagine, senza però essere in grado di distinguerli in maniera puntuale. A tal proposito è possibile utilizzare uno tecnica di post-processing chiamata watershed, a valle quindi della detection di SegFormer [34]. Questa è una tecnica basata sulla morfologia matematica, che analizza l'immagine separando oggetti adiacenti in maniera del tutto agnostica riguardo il loro contenuto: non è in grado di distinguere cosa è albero e cosa non lo è ma, partendo dalle zone isolate grazie a SegFormer, l'unico tipo di oggetti isolabili saranno implicitamente alberi, consentendo così di individuare le

chiome ed il numero di alberi presenti.

Riassumendo il funzionamento in breve:

- L'algoritmo prende un'immagine in scala di grigi in input.
- L'immagine viene interpretata come una superficie, dove i pixel rappresentano altitudini.
- Si selezionano pixel o regioni di partenza chiamati marker, che possono identificare le etichette dei punti più bassi, i minimi locali, oppure quelli più alti come i massimi locali. Nel nostro caso saranno i massimi, ovvero le punte degli alberi, ad essere individuate.
- L'immagine viene segmentata tramite un procedimento di flooding, da qui il termine watershed (riempimento di bacini), dove le regioni vengono "riempite" a partire dai marker.
- In ogni step i pixel confinanti con le regioni flooded vengono ordinati in base al valore del gradiente, inserendoli in una coda prioritaria: il pixel di minore (o maggiore nel caso dei massimi) valore viene assegnato al bacino del marker più vicino.
- I confini tra i bacini costituiscono la linea di watershed, ovvero il bordo fra i diversi elementi dell'immagine.

6.6.1 Funzione di segmentazione

Qui viene riportata una funzione che si occupa di trovare le regioni e gli elementi grazie al processo di watershed, partendo da una maschera fornita in input come parametro.

Per rendere l'operazione più efficiente, è possibile sfruttare due librerie molto note in ambito Python, come Scikit e SciPy: la prima viene usata per l'elaborazione delle immagini ed è già stata introdotta nel capitolo su Detectree, mentre la seconda è una libreria open-source che offre numerosi strumenti di elaborazione dati, tra cui algoritmi di ottimizzazione, interpolazione ed elaborazione di segnali [35].

```
1 from scipy.ndimage import distance_transform_edt, label
2 from skimage.feature import peak_local_max
3 from skimage.morphology import watershed
4 from skimage.measure import regionprops
5 import numpy as np
6
```

```
def watershedSegmentation(mask, minDistance, offsetX=0, offsetY=0):
    distanceMap = distance_transform_edt(mask)

    peakCoords = peak_local_max(
        distanceMap,
        min_distance=minDistance,
        labels=mask,
        footprint=np.ones((3, 3))
    )

    markers = np.zeros_like(mask, dtype=np.int32)
    for i, (y, x) in enumerate(peakCoords, start=1):
        markers[y, x] = i

    labelsWatershed = watershed(-distanceMap, markersLabeled, mask=mask)

    treeCrowns = []
    for treeId in range(1, labels.max() + 1):
        crownMask = (labels == treeId)
        treeCrowns.append(crownMask)

    return treeCrowns
```

La funzione *distance_transform_edt* di SciPy viene utilizzata per creare la mappa delle distanze, che viene successivamente fornita in input alla funzione *peak_local_max*, appartenente alla libreria scikit-image, per individuare i marker.
Successivamente vengono trasformate le coordinate dei picchi in una mappa di marker numerici, dove ogni marker ha un'etichetta univoca assegnata alla crescita dei bacini watershed.
Poi viene invocata *watershed*, funzione di scikit-image, che "allaga" progressivamente l'immagine partendo dai marker, per poi fermarsi solamente quando le linee di delimitazione si incontrano, rilevando così i confini degli oggetti.
Infine, vengono isolate le maschere degli oggetti rilevati e memorizzate nell'array *treeCrowns*, che possono essere utilizzate per il calcolo di area, perimetro o per la visualizzazione in overlay sugli oggetti delimitati.

6.6.2 Risultato della segmentazione

Il processo di segmentazione watershed produce tre output progressivi: partendo dall'immagine satellitare sorgente, viene prima generata la maschera di SegFormer che evidenzia in verde le aree arboree rilevate, e successivamente viene applicata la segmentazione watershed che distingue le singole corone degli alberi, identificandole con colori differenti. Questo permette di contare e localizzare ogni singolo albero all'interno delle aree precedentemente identificate come piantumate.

7

Applicazione della regola del 3-30-300 e mappa degli alberi

I capitoli precedenti trattano l'analisi dei modelli e delle reti che, opportunamente addestrate e parametrizzate, riescono ad isolare aree verdi e alberi.
La ricerca è focalizzata in particolare alla detection degli alberi presenti nel tessuto urbano, non solo per mettere a confronto e valutare le prestazioni dei modelli in se, ma anche per cercare di ricavare in maniera del tutto automatica le coordinate di ogni albero, in latitudine e longitudine, in modo da poterlo mappare su sistemi informativi quali ad esempio GIS.
Il Geographic Information System (GIS) è una tecnologia che consente di analizzare dati georeferenziati per visualizzare elementi cartografici organizzati in layer sovrapposti, permettendo di ottenere rappresentazioni visuali strutturate a partire da dati grezzi [36].

7.1 Regola del 3-30-300

L'obiettivo finale di questa ricerca è quindi quello di produrre un flusso dati, opportunamente formattato a seconda del software GIS o della soluzione implementata, che possa contenere una posizione geografica ben precisa degli alberi nella città di Verona.
Questo verrà poi utilizzato per calcolare un particolare indicatore, in riferimento alla regola del 3-30-300.

7.1.1 Definizione

La regola del 3-30-300 è un principio ideato dal professor Cecil Konijnendijk, che definisce quali sono i criteri per garantire l'accesso al verde nelle città, in modo da renderle più vivibili e più sostenibili [37]. I tre criteri sono:

- 3 alberi visibili: ogni residente dovrebbe poter vedere almeno 3 alberi dalla propria finestra.
- 30% di copertura arborea nel quartiere: almeno il 30% della superficie del quartiere dovrebbe essere coperto da alberi, considerando le chiome.
- 300 metri di distanza dallo spazio verde più vicino: ogni residente dovrebbe avere a disposizione un parco al massimo a 300 metri dalla propria abitazione.

7.1.2 Raccolta dei dati

Per poter arrivare a calcolare l'indicatore 3-30-300, che è oggetto di un'altra ricerca specifica sull'argomento, non è più quindi sufficiente accontentarsi di rilevare gli alberi, ma è necessario invece creare una vera e propria mappa a partire dalle immagini satellitari, in modo da poter localizzare ogni albero in maniera distinta e con una posizione geografica.

7.2 Mappa degli alberi

I modelli fin qui analizzati, per individuare gli alberi dalle immagini satellitari, restituiscono immagini con annotazioni e bounding boxes per evidenziare confidenza e posizione degli alberi nell'immagine stessa: sarebbe possibile utilizzare questo tipo di informazioni, per individuare anche la posizione degli alberi in maniera non solo visiva ma anche geografica? Per rispondere a questo quesito è necessario prender confidenza un particolare formato di immagine, che coniuga appunto coordinate dell'immagine con dati geografici: il GeoTIFF.

7.2.1 TIFF e GeoTIFF

TIFF è un popolare formato di immagini raster, che è nato come metodo per lo scambio di immagini tra stampanti e scanner, data la sua caratteristica di poter supportare proprietà e dati correlati all'immagine stessa. Da un lato porta questo formato ad essere molto flessibile, dall'altro il fatto di poter supportare molti schemi di compressione e feature, porta TIFF ad essere

complicato da usare perché l'applicazione che lo deve leggere deve poter supportare pienamente il formato [38].

GeoTIFF sfrutta la flessibilità di TIFF per includere metadati relativi alla posizione geografica, diventando così uno standard de facto per le organizzazioni e la community di persone che lavorano con dati geospaziali [39].
Questo tipo di formato è anche supportato dai sistemi GIS.

7.2.2 Da JPG a GeoTIFF

Purtroppo in molti casi le mappe satellitari reperibili sulle piattaforme di location, commerciali o open source, sono in formato vettoriale oppure raster JPG o PNG ed è quindi impossibile utilizzare quel tipo di immagini in un contesto georeferenziato. Per questo obiettivo, le semplici immagini satellitari utilizzate per la detection degli alberi, non sono sufficienti per estrarre dati utilizzabili per fare analisi su mappe e indicatori tramite GIS.
Per ovviare a questo problema ci viene in aiuto Rasterio, una libreria Python che non solo riesce a leggere i formati GeoTIFF, ma è anche in grado di manipolarli associando una specifica posizione in pixel ad una posizione spaziale in termini di coordinate latitudine/longitudine [40]. Rasterio si appoggia sul noto GDAL [41], acronimo di Geospatial Data Abstraction Library, infatti offre l'interfaccia Python per andare a sfruttare la potente libreria in C++ che elabora i file raster.

7.2.3 Conversione pixel-coordinate

I provider di mappe satellitari solitamente offrono degli endpoint che permettono di scaricare piccole "mattonelle" di una mappa, una alla volta, chiamate tiles: sarebbe impensabile infatti scaricare l'intera mappa di una città intera. Per poter accedere alle tiles è necessario specificare tre parametri, ovvero lo zoom e le coordinate in lat/long dell'angolo nord-ovest della tile, così facendo si avrà una tile di piccole dimensioni (e.g. 512x512) ma che ha una precisa posizione geografica.
Per arrivare al risultato è possibile procedere in due modi:

- Trasformare le tiles in GeoTIFF: se il modello da utilizzare per la detection degli alberi accetta in input immagini che hanno la stessa risoluzione delle tiles scaricate, è possibile

trasformare in GeoTIFF direttamente l'immagine con Rasterio.

- Unire le tiles in un immagine più grande e convertirla in GeoTIFF: se il modello ha in input specifiche di risoluzione diverse da quelle delle tiles scaricate, allora è necessario unire tutte le immagini e poi tagliarla nelle tiles della dimensione corretta, convertendo prima in GeoTIFF l'immagine "collage" derivata dalle composizione delle altre tiles.

7.2.4 Detection e generazione di GeoJSON

Una volta che le tiles sono state convertite in GeoTIFF è possibile procedere con l'inferenza su ogni singola tile e rapportare la posizione in pixel alla posizione geografica, operazione possibile grazie a Rasterio.
Terminato il processo di detection, per poter condividere le coordinate precise degli alberi presenti nelle immagini analizzate in modo che siano fruibili da GIS, la soluzione indicata è quella di esportare un GeoJSON. Quest'ultimo è proprio un formato particolare di JSON che viene utilizzato per lo scambio di dati geospaziali: un GeoJSON può descrivere un oggetto rilevato sulla mappa come punto geografico, con in aggiunta anche delle proprietà esplicative, come ad esempio la classe dell'oggetto rappresentato o il nome.
Inoltre supporta anche dei tipi di geometria, come punti, linee e poligoni, rendendolo quindi indicato anche per rappresentare ad esempio i bounding boxes restituiti da YOLO [42].

7.3 Conversione in GeoTIFF con Rasterio

Per fare il merge e le conversioni delle tiles utilizzando Rasterio, è necessario usare Python. A differenza delle soluzioni viste in precedenza per il training dei modelli, nel caso delle operazioni con le immagini e la semplice inferenza per la detection degli alberi, non è strettamente necessario l'utilizzo di una GPU: è sufficiente lanciare gli script sulla propria macchina avendo cura di valorizzare i parametri corretti.

Di seguito sono riportati alcuni spezzoni di codice, che costituiscono le parti importanti degli script di elaborazione delle tiles. Viene inoltre presentata la fase di inferenza con YOLO11 che, pur rappresentando una delle possibili alternative per la rilevazione degli alberi, consente di illustrare l'intero processo fino alla generazione dei file GeoJSON. Questo codice è responsabile del salvataggio di un immagine raster JPG o PNG, in un immagine GeoTIFF. Prima di arrivare

alla funzione di salvataggio, nello script completo sono presenti una lista di comandi che servono ad estrarre dal filename il livello di zoom, latitudine e longitudine, tali righe sono tralasciate in quanto di banale implementazione.
Più interessante invece è osservare come vengono convertiti questi valori in coordinate della tile, come matrice di quadrati, operazione necessaria per creare dei bounds da salvare successivamente come GeoTIFF.

7.3.1 Conversione lat/long e zoom in coordinate slippy

```
1 def deg2num(lat_deg: float, lon_deg: float, zoom: int) -> Tuple[int, int]:
2     lat_rad = math.radians(lat_deg)
3     n = 2.0 ** zoom
4     x = int((lon_deg + 180.0) / 360.0 * n)
5     y = int((1.0 - math.asinh(math.tan(lat_rad)) / math.pi) / 2.0 * n)
6     return (x, y)
```

deg2num è una funzione di conversione standard per questo tipo di operazione, in rete se ne possono trovare altre che utilizzano librerie di terze parti, ma molto importante per comprendere il meccanismo di conversione in coordinate tile.
Questo tipo di coordinate tile sono anche chiamate "slippy" [43], che si riferisce al tipo di mappa che noi tutti consultiamo sul web, che consente di muoversi ("to slip" significa "scivolare") e ingrandire/rimpicciolire la visualizzazione, con tanto di zoom (e.g. Google Maps).
La conversione di queste coordinate avviene attraverso la proiezione chiamata Web Mercator, che è una variante "digitale" della proiezione cartografica di Mercatore, cartografo e astronomo che nel XVI secolo ha convertito la superficie terrestre sferica in superficie piana, creando così le mappe che consultiamo anche ai giorni nostri.

7.3.2 Calcolo dei confini geografici

```
1 def calculateBounds(x: int, y: int, zoom: int) -> Tuple[float, float, float, float]:
2     n = 2.0 ** zoom
3     west = x / n * 360.0 - 180.0
```

```
    east = (x + 1) / n * 360.0 - 180.0

    north_rad = math.atan(math.sinh(math.pi * (1 - 2 * y / n)))
    north = math.degrees(north_rad)

    south_rad = math.atan(math.sinh(math.pi * (1 - 2 * (y + 1) / n)))
    south = math.degrees(south_rad)

    bounds = (west, south, east, north)
```

Questa parte di codice è fondamentale per trovare quelli che vengono chiamati "bounds" dell'immagine. In questo specifico caso, le tiles scaricate via api dal provider di mappe, indicano come coordinata lat/long del centro nord-ovest della tile, è necessario a questo punto isolare quali sono i bounds nei quattro punti cardinali, in modo da trovare quali sono i limiti geografici della tile:

- *west*: longitudine del bordo sinistro (ovest) della tile
- *south*: latitudine del bordo inferiore (sud) della tile
- *east*: longitudine del bordo destro (est) della tile
- *north*: latitudine del bordo superiore (nord) della tile

questi quattro bounds saranno poi necessari a Rasterio per poter creare il GeoTIFF.

7.3.3 Salvataggio del GeoTIFF

```
def saveGeotiff(bounds: Tuple[float, float, float, float], outputPath: str) -> bool:
    west, south, east, north = bounds
    outputImage = Image.new('RGB', (TILE_SIZE, TILE_SIZE))

    transform = Affine.translation(west, north) * Affine.scale((east - west) / width, (south -
        north) / height)

    imageArray = np.array(outputImage)
    imageArray = np.transpose(imageArray, (2, 0, 1))

```

```
10      with rasterio.open(
11          outputPath,
12          'w',
13          driver='GTiff',
14          height=height,
15          width=width,
16          count=imageArray.shape[2],
17          dtype=imageArray.dtype,
18          crs=CRS.from_epsg(4326),
19          transform=transform,
20          compress='lzw'
21      ) as dst:
22          dst.write(imageArray)
```

La funzione per il salvataggio dell'immagine come GeoTIFF prende in input i bounds, dalla *calculate_bounds* e l'*output_path* dove salvare l'immagine. Prima di tutto deve essere creata un'immagine RGB che possa contenere la tile, che viene istanziata con Pillow.
L'immagine poi viene passata alla libreria Rasterio come array tramite una conversione che avviene grazie a NumPy.
Rasterio, per scrivere correttamente i dati nel file GeoTIFF, si aspetta infatti un array così strutturato:

- shape: bande, altezza, larghezza
- tipo numerico: uint8, float32...

Per procedere con la scrittura dell'immagine, a riga 4, deve prima essere effettuata una creata una matrice di trasformazione particolare:

- *Affine.translation(west, north)*: è una trasformazione di traslazione: l'angolo superiore sinistro dell'immagine corrisponde a nord-est in coordinate geografiche.
- *Affine.scale(...)*: esegue uno scale dei pixel, dove ogni pixel, sull'asse orizzontale, vale *(east - west) / width* gradi di longitudine, mentre ogni pixel, sull'asse y (verticale), vale *(south - north) / height* gradi di latitudine.
- Moltiplicazione: Ottiene una matrice affine che permette a Rasterio di collegare ogni pixel dell'immagine all'esatta posizione georeferenziata.

L'ultima operazione prima della scrittura del GeoTIFF consiste nella trasposizione degli assi dell'immagine. Questa modifica è necessaria poiché l'immagine RGB di Pillow utilizza il formato (altezza, larghezza, bande), mentre Rasterio richiede il formato (bande, altezza, larghezza), con il numero di bande (tre nel caso del formato RGB) specificato come primo parametro. Dopodichè è sufficiente invocare Rasterio, fornendo tutti i parametri necessari per far si che il GeoTIFF in output sia perfettamente compatibile con i sistemi GIS:

- *outputPath*: path del file di output
- *'w'*: modalità di apertura del file, in questo caso write
- *driver*: tipo di formato raster da utilizzare, "GTiff" è il nostro GeoTIFF
- *height*: numero di righe in pixel dell'immagine
- *width*: numero di colonne in pixel dell'immagine
- *count*: numero di bande dell'immagine
- *dtype*: tipo di dato dei pixel (es. uint8, float32)
- *crs*: Coordinate Reference System, in questo caso imposta il sistema WGS84 (lat/lon, standard GPS).
- *transform*: matrice di trasformazione affine che collega i pixel alle coordinate geografiche (bounding box + risoluzione).
- *compress*: algoritmo di compressione per il file

7.3.4 Detection degli alberi

```
1 def analyzeGeotiffTreeCoverage(imagePath, modelPath, conf=0.25):
2     with rasterio.open(imagePath) as src:
3         transform = src.transform
4 
5         imageArray = src.read([1, 2, 3])
6         imageArray = np.transpose(imageArray, (1, 2, 0))
7 
8         crs = src.crs
9         width = src.width
```

```
10        height = src.height
```

Come già anticipato in precedenza ci sono molti modi di effettuare la detection degli alberi, è qui riportato il codice per l'inferenza con YOLO a titolo di esempio.
Lo script svolge, almeno in parte, il processo inverso di trasformazione da immagine Rasterio a NumPy. Questo perché YOLO chiede in input un'immagine di tipo RGB, quindi con la matrice dei canali organizzata in maniera standard. Per la conversione viene utilizzato nuovamente Pillow.

```
1    model = YOLO(model_path)
2    results = model(img_array, conf=conf)
```

In seguito viene lanciata l'inferenza sull'immagine.
Il codice per questo tipo di inferenza differisce da quello utilizzato per la semplice inferenza con YOLO. Mentre quest'ultima identifica la posizione dell'albero nell'immagine restituendo un'immagine con i bounding box evidenziati tramite OpenCV, in questo caso la posizione rilevata viene convertita in coordinate geografiche (latitudine/longitudine) corrispondenti alla localizzazione effettiva dell'albero.

```
1    for result in results:
2        if result.boxes is not None:
3            boxes = result.boxes.xyxy.cpu().numpy()
4            confidences = result.boxes.conf.cpu().numpy()
5
6            for box, confidence in zip(boxes, confidences):
7                if confidence >= conf:
8                    x1, y1, x2, y2 = box.astype(int)
9
10                   centerX = (x1 + x2) / 2
11                   centerY = (y1 + y2) / 2
12
13                   lon, lat = transform * (centerX, centerY)
```

Ecco come vengono analizzati i risultati, in maniera simile a quanto visto nel capitolo YOLO ma in questo caso, grazie alla matrice di trasformazione fornita da Rasterio, la funzione riesce

appunto a ricavare le coordinate geografiche dal punto preciso in pixel dell'albero rilevato nell'immagine.

Arrivati a questo punto possiamo costruire, grazie a tutti i parametri raccolti dalla detection nell'immagine e dalla conversione dei pixel in lat/long, il GeoJSON che andrà ad alimentare GIS.

Nel caso di DetecTree2, l'output della predizione da scrivere nel GeoJSON non è rappresentato da un punto, bensì da un poligono che delimita l'intera corona dell'albero.

7.3.5 Generazione GeoJSON

Ricavate tutte le coordinate di latitudine e longitudine degli alberi rilevati nelle immagini, l'ultimo passaggio prevede di creare il GeoJSON, seguendo questo formato per mantenere la compatibilità con i software GIS [42]:

```
{
"type": "FeatureCollection",
"crs": {
    "type": "name",
        "properties": {
            "name": "EPSG:4326"
        }
    },
"features": [
    ...
    ]
}
```

Un file GeoJSON deve specificare esplicitamente la tipologia di annotazioni contenute. In questo caso si tratta di *FeatureCollection*, che indica la presenza di un array di elementi geografici quali poligoni o punti.

La proprietà crs, invece, indica il tipo di coordinate che sono specificate nelle features come ad esempio EPSG:4326, conosciuto anche come WGS84, che equivale alla classica notazione di latitudine e longitudine con gradi decimali.

Infine l'array di features conterrà una collezione di oggetti, ovvero i punti corrispondenti alla

posizione degli alberi, che hanno questo formato:

```
{
    "type": "Feature",
    "geometry": {
        "type": "Point",
        "coordinates": [
          10.986155775846253,
          45.445288495847386
        ]
    },
    "properties": {
        "tree_id": 0,
        "confidence": 0.810298502445221,
        "image_name": "image.tif",
        "image_path": "/data1/data2/image.tif",
        "source_crs": "EPSG:4326",
        "bbox_x1": 600,
        "bbox_y1": 84,
        "bbox_x2": 634,
        "bbox_y2": 123,
        "pixel_x": 617.0,
        "pixel_y": 103.5
    }
}
```

Fondamentale è la dichiarazione del type "Feature", dove viene specifiato il tipo di feature geografica, in questo caso Point, con le relative coordinate nel formato dichiarato dal parametro del json dedicato.

Inoltre è possibile in un GeoJSON dichiarare delle properties custom, che contengono dati affini alla feature dichiarata. Nel caso di YOLO ad esempio ha senso riportare un id numerico dell'albero rilevato, il livello di confidence, il path dell'immagine e le coordinate delle bounding boxes. E' importante sottolineare che le properties non sono comunque obbligatorie, soltanto il tipo e la geometria della feature lo sono.

7.4 Risultato su QGIS

Importando il GeoJSON così generato su QGIS, un software GIS open source, gli alberi rilevati appaiono come punti georeferenziati sulla mappa, permettendo di visualizzare la distribuzione della copertura arborea.

8

Analisi dei risultati

Nei precedenti capitoli sono stati trattati diversi algoritmi che permettono, a partire da immagini satellitari, di ricavare la posizione degli alberi sia come coordinate in pixel che geografiche, esportando infine un GeoJSON che ne definisce in maniera più o meno precisa la forma della corona o la bounding box che delimita l'albero stesso. Altri algoritmi, come SegFormer e DetecTree, sono invece in grado di isolare gruppi di alberi.
Queste due modalità sono interessanti in quanto applicabili per il calcolo dell'indicatore trattato nel capitolo 7, dove è considerata sia la distanza da singoni alberi che quella da parchi o aree verdi piantumate.
Per verificare però quale dei modelli ha ottenuto i risultati migliori, è necessario prima di tutto individuare la reale posizione geografica degli alberi presenti sul tessuto urbano.

8.1 Dataset di ground truth

Per procedere con la verifica è fondamentale poter accedere ad un dataset contenente la posizione in latitudine e longitudine degli alberi piantumati, in altre parole il nostro ground truth, ovvero i dati di riferimento verificati. In caso contrario è impossibile poter confrontare l'accuratezza dei vari modelli visti in precedenza, che si basano solo sulla detection visiva.

8.1.1 Accesso a database pubblici

In alcuni casi la regione, il comune o enti terzi forniscono API o dataset che mappano gli alberi e/o le piantumazioni. Qualora questo tipo di informazioni siano facilmente accessibili, il processo di costruzione del database di ground truth si semplifica notevolmente, poiché già disponibili come dati validati e puliti, ovvero filtrati da possibili falsi positivi o falsi negativi.
È necessario però considerare che gli alberi mappati in questi dataset, potrebbero contenere solo le aree e gli alberi che costituiscono la vegetazione su suolo pubblico, non considerando quindi proprietà private piantumate che vengono rilevate invece dalle sole immagini satellitari.
Il formato dei dati varia da ente pubblico, regione o comune: in alcuni casi vengono fornite le semplici coordinate in JSON/CSV, in altri in GeoJSON pronti ad essere importati in applicativi GIS, in altri ancora sono sotto forma di pacchetto CHM. Quest'ultimo è un particolare formato che viene generato da sensori LIDAR (Light Detection and Ranging) posti su velivoli che generano, grazie alla tecnologia di impulsi laser, una rappresentazione in 3D del territorio e delle corone degli alberi.

8.1.2 Annotazione manuale

Un'altra alternativa è quella di creare un dataset manualmente, con applicativi come QGIS, importando l'immagine satellitare di riferimento e annotando le coordinate in maniera punta e clicca su un layer, successivamente esportabile in GeoJSON.
Questa soluzione rende possibile l'annotazione di tutti gli alberi, su suolo pubblico e non, ma è estremamente oneroso in termini di tempo. È inoltre fondamentale verificare che l'immagine satellitare utilizzata nel software (ad esempio un file GeoTIFF) sia correttamente georeferenziata, poiché una georeferenziazione errata comporterebbe un'annotazione imprecisa delle coordinate geografiche di ciascun albero rilevato.

8.2 Benchmark dei modelli

Una volta definito il ground truth, è possibile procedere al confronto con le predizioni generate dai modelli: in base alla fonte dei dati è necessario stabilire i parametri di confronto sui quali calcolare le metriche di qualità del rilevamento.
Per questa analisi è stato utilizzato un GeoJSON messo a disposizione dal comune di Milano

[44], che mappa tutti gli alberi piantumati sul suolo pubblico all'interno dei confini comunali. Per ragioni di costi e di complessità computazionale, la detection è stata effettuata sulla mappa satellitare che misura 3km di raggio a partire dal centro di milano. I dati sono liberamente disponibili con licenza Creative Commons

8.2.1 Requisiti

Come anticipato precedentemente, questo GeoJSON proveniente da fonte istituzionale include esclusivamente gli alberi su aree pubbliche, escludendo quelli su suolo privato. Non è quindi possibile determinare con precisione i falsi positivi effettivi, poiché numerosi alberi assenti dal dataset comunale potrebbero essere stati rilevati correttamente dai modelli precedentemente analizzati, creando così una sovrastima degli errori di rilevamento.
Inoltre, i dati forniti dal comune mappano gli alberi come singoli punti georeferenziati anziché come poligoni quindi, per rendere possibile questa comparazione, un albero viene considerato correttamente rilevato quando il punto georeferenziato ricade all'interno di una bounding box YOLO o di un poligono di segmentazione (che rappresenta la chioma per DetecTree2 o l'area piantumata per SegFormer). Gli alberi i cui punti non ricadono in nessun poligono vengono classificati come non rilevati.
Per confrontare correttamente i modelli, è tuttavia necessario distinguere tra quelli che identificano singoli alberi (come YOLO e DetecTree2) e quelli che identificano aree più ampie, come le aree piantumatie o le zone verdi (SegFormer). Questa distinzione è fondamentale poiché i diversi approcci di rilevamento richiedono metodologie di valutazione differenti.

8.2.2 Parametri dei modelli

I risultati sono basati su detection effettuate con i seguenti parametri di confidence:

- YOLO11: 0.416, che come definito dalla curva F1 dopo il training è il valore che bilancia perfettamente precision e recall. Per compensare la precisione della detection, viene applicato un buffer di prossimità di 11m attorno alle bounding box.
- Detectree2: 0.25, definito per una detection che favorisce valori di recall più alti ma riduce mancate detection.

- SegFormer: 0.30, offre il miglior bilanciamento fra precision e recall, individuato dopo un certo numero di test (SegFormer non ha un valore di F1 globale, lavora a livello di pixel con probabilità per classe).
- DetecTree: non ha parametrizzazione della confidenza, effettua anch'esso una classificazione dei pixel, ma in maniera diversa da SegFormer.

8.2.3 Script di rilevamento

La funzione di confronto tra il file GeoJSON del comune di Milano, utilizzato come ground truth, e i poligoni predetti dai modelli è relativamente semplice e lineare:

```
1  import geopandas as gpd
2
3  def calculateTreeCoverage(groundTruthPath: str, modelTreesDetectedPath: str):
4      trees = gpd.read_file(groundTruthPath)
5      detections = gpd.read_file(modelTreesDetectedPath)
6      detectedTrees = set()
7
8      for _, polygon in detections.iterrows():
9          treesInside = trees[polygon.geometry.contains(trees.geometry)]
10         detectedTrees.update(treesInside.index)
11
12     coveragePercentage = (len(detectedTrees) / len(trees)) * 100
```

Nelle prime due righe vengono caricati i files in due dataframe GeoPandas, una libreria Python che permette di elaborare dati in tabelle e csv, con il supporto per la gestione e l'analisi di dati geospaziali [45].
Dopo aver inizializzato un set per contenere gli effettivi alberi rilevati, si passa all'incrocio dei dati: per ciascun poligono rilevato dal modello di machine learning, viene usata l'operazione geometrica *contains()*, che isola tutti gli alberi di ground truth contenuti all'interno del poligono di rilevazione. Gli indici di questi alberi vengono aggiunti poi al set *detectTrees*.
A questo punto la percentuale di rilevamento non è altro che il rapporto tra alberi unici rilevati e il totale di alberi nel ground truth.

8.3 Rilevamento aree arboree

In questo caso sono due gli algoritmi che restituiscono aree popolate da alberi, anzichè singole corone: per completezza è stato incluso DetecTree.

8.3.1 SegFormer

Metrica	**Valore**
Alberi rilevati	25.029
Alberi non rilevati	19.365
Poligoni con alberi	4.076
Poligoni senza alberi	9.202
Percentuale alberi rilevati	**56,38%**

Tabella 8.1: Risultati rilevamento SegFormer

SegFormer si conferma molto efficace a rilevare alberi in contesti urbani e, considerando che il dataset di partenza è lo stesso usato per il training del modello YOLO11, è significativamente più preciso se consideriamo l'area di copertura anzichè i singoli alberi, dimostrando una buona selettività nel rilevare aree effettivamente piantumate.

8.3.2 DetecTree

Metrica	**Valore**
Alberi rilevati	17.563
Alberi non rilevati	27.294
Poligoni con alberi	8.006
Poligoni senza alberi	436.865
Percentuale alberi rilevati	**39,2%**

Tabella 8.2: Risultati rilevamento DetecTree

DetecTree si conferma in grado di poter rilevare la maggioranza degli alberi ma, come introdotto nel capitolo dedicato, per la sola copertura arborea è estremamente poco preciso perché mescola alle corone anche le aree verdi. A dimostrazione di questo vi è proprio il fatto che sono stati identificati dall'algoritmo aree verdi suddivise in più di 400 mila poligoni, di queste solo l'1,8% contengono alberi: sicuramente alcuni poligoni conterranno piantumazioni in aree private, ma la maggioranza conterrà generico "verde urbano".

8.4 Rilevamento singoli alberi

Vediamo ora i risultati sui dati inferiti dai due modelli di object detection, che sfruttano reti convoluzionali: come anticipato in precedenza essi lavorano sulle feature/caratteristiche dell'immagine, per poterne isolare gli oggetti in esse contenuti.

8.4.1 YOLO11

Metrica	**Valore**
Alberi rilevati	22.956
Alberi non rilevati	21.438
Poligoni con alberi	10.817
Poligoni senza alberi	11.861
Media alberi per poligono	2,62
Percentuale alberi rilevati	**51,71%**

Tabella 8.3: Risultati rilevamento YOLO11

In questo caso i poligoni sono ricavati dalle bounding boxes che, essendo progettate per mettere in evidenza l'oggetto rilevato, non costituiscono la vera e propria corona dell'albero.
Considerando solamente gli alberi presenti nel GeoJSON di Milano, abbiamo una media di 2,62 alberi per detection: ovvero il 24,2% delle bounding box contiene un singolo albero, mentre il 71,0% contiene piccoli gruppi da 2 a 5 alberi, dimostrando una buona precisione nel rilevamento di alberi individuali e piccoli gruppi.

8.4.2 DetecTree2

Metrica	**Valore**
Alberi rilevati	10.355
Alberi non rilevati	34.039
Poligoni con alberi	5.216
Poligoni senza alberi	9.659
Media alberi per poligono	2.01
Percentuale alberi rilevati	**23,33%**

Tabella 8.4: Risultati rilevamento DetecTree2

DetecTree2 è un modello complicato da bilanciare, in quanto a valori bassi di confidence è si in grado di rilevare più alberi ma introduce un numero importante di falsi positivi. A questo

valore di confidence, identificato come buono dopo vari test di inferenza su immagini satellitari, sembra essere nettamente meno efficace degli altri due: probabilmente il fatto di trovarsi in un contesto urbano non lo aiuta, infatti sembra essere molto più adatto per foreste o ecosistemi forestali [46].
Considerando gli alberi che è riuscito a rilevare con successo, possiamo comunque notare una media di 2,01 alberi per poligono, quindi DetecTree2 mostra una buona capacità di segmentare singole corone: il 49,3% dei poligoni contiene un singolo albero e il 47,1% contiene piccoli gruppi da 2 a 5 alberi.

8.4.3 SegFormer + Watershed

Watershed è una tecnica di segmentazione delle immagini che permette di separare oggetti connessi. È stata applicata ai risultati di SegFormer per tentare di suddividere le aree arboree in alberi individuali.

Metrica	**Valore**
Alberi rilevati	24.280
Alberi non rilevati	20.114
Poligoni con alberi	1.917
Poligoni senza alberi	1.845
Media alberi per poligono	13.87
Percentuale alberi rilevati	**54,69%**

Tabella 8.5: Risultati rilevamento SegFormer + Watershed

L'applicazione di watershed ai risultati di SegFormer peggiora leggermente le performance: la detection rate scende dal 56,38% al 54,69%. Questo suggerisce che tentare di suddividere ulteriormente le aree rilevate può introdurre alcuni errori.
Con una media di 13,87 alberi per poligono, l'algoritmo SegFormer + Watershed rileva principalmente aree con gruppi di alberi: solo il 5,6% dei poligoni contiene un singolo albero, mentre il 39,9% contiene gruppi superiori a 10 alberi.

8.5 Confronto comparativo

I modelli analizzati mostrano caratteristiche distintive a seconda della loro architettura e approccio alla detection.

8.5.1 Rilevamento alberi singoli

Per il rilevamento di alberi individuali, i modelli con le migliori performance in termini di precisione sono:

Algoritmo	**Copertura**	**Alberi per poligono**
YOLO11	51,71%	2,62
DetecTree2	23,33%	2,01
SegFormer+Watershed	54,69%	13,87

Tabella 8.6: Confronto algoritmi per rilevamento alberi singoli

YOLO11 e DetecTree2 si distinguono per la capacità di identificare alberi individuali (media 2-3 alberi per detection), con DetecTree2 particolarmente preciso nella segmentazione di singole corone (49,3% detection con un solo albero). YOLO11 con buffer di 11m raggiunge una detection rate di 51,71% mantenendo una media di 2,62 alberi per detection, risultando molto efficace per rilevamento di alberi individuali. SegFormer+Watershed, pur avendo una detection rate leggermente superiore (54,69%), identifica principalmente gruppi di alberi (media 13,87 alberi/detection), risultando meno adatto per applicazioni che richiedono conteggio preciso di alberi individuali.

8.5.2 Rilevamento aree arboree

Per identificare aree piantumate e gruppi di alberi, SegFormer base risulta il più efficace:

Algoritmo	**Detection rate**
SegFormer	56,38%
DetecTree	39,2%

Tabella 8.7: Confronto algoritmi per rilevamento aree arboree

SegFormer offre la miglior detection rate (56,38%) con la sua capacità di identificare in maniera precisa i gruppi di alberi, mentre DetecTree ha un tasso di falsi positivi eccessivo (98,19%) che lo rende inadatto per applicazioni pratiche di solo rilevamento arboreo.

8.6 Analisi e miglioramenti

I risultati raggiunti dai vari modelli denotano una discreta capacità di rilevamento degli alberi: SegFormer, il modello più performante tra i tre, rileva poco meno del 57% degli alberi piantu-

mati sul territorio comunale. Si tratta di una percentuale significativa per questo specifico task, notoriamente complesso data la necessità di disporre di immagini satellitari di elevata qualità per ottenere buoni livelli di rilevamento. I fattori più comuni che possono compromettere i risultati sono legati alla qualità del dataset di training e al tipo di immagini in input utilizzate per l'inferenza.

8.6.1 Specie di alberi nel dataset di Lleida

Il dataset utilizzato per il training dei due modelli che hanno restituito i migliori riscontri in termini di copertura proviene da studi condotti nell'area urbana di Lleida, in Spagna, precedentemente introdotto nel capitolo dedicato a YOLO. La documentazione sulle specie arboree del territorio di Lleida [47] evidenzia la presenza di alcune specie tipiche dell'ambiente mediterraneo e fluviale, alcune delle quali potrebbero non essere completamente rappresentative dell'ecosistema urbano milanese.
La differenza climatica e biogeografica tra Lleida e Milano potrebbe influenzare l'efficacia del modello addestrato su questo dataset quando applicato al contesto lombardo, caratterizzato da un clima continentale temperato e da una composizione arborea urbana differente.

8.6.2 Proximity buffer

Come abbiamo visto, un modello come SegFormer è in grado di dare un risultato accettabile in termini di detection. Per poter migliorare ulteriormente le prestazioni, vi è una metodologia applicabile in fase di post-processing, che consiste nell'implementare un buffer di prossimità attorno alle rilevazioni esistenti [48].
Per valutare se applicare un buffer sulla detection effettuata da SegFormer, ovvero la più performante, è necessario identificare il problema: il 74.0% degli alberi mancati si trova entro i 50m dalle detection già mappate, quindi sono visualmente presenti ma non rilevati a causa di:

- Ombre che mascherano gli alberi, causate dall'orario in cui le immagini satellitare sono state scattate.
- Bordi delle chiome dove la segmentazione si interrompe prematuramente.

- Sovrapposizioni tra alberi adiacenti che confondono la rete neurale.

Rilevando un albero in una data posizione, è probabile statisticamente che ci siano altri alberi nelle sue vicinanze che il modello non ha visto: il buffer aiuta a recuperare i falsi negativi.
I risultati con questo procedimento di post-processing spaziale riescono a rilevare 38.304 alberi contro i 25.029, portando a 86,28% la detection rate degli alberi del territorio comunale milanese.

8.6.3 Valutazione delle performances

Questo approccio di rilevamento grazie al deep learning + post-processing spaziale, che restituisce l'86,28% degli alberi supera ampiamente gli standard accettabili per tree detection urbano (tipicamente 60-80%) ed è da considerarsi un risultato eccellente per diversi motivi:

- Alta densità urbana con sovrapposizioni complesse tra edifici e vegetazione.
- Diversità delle specie arboree con morfologie variabili.
- Ombre urbane e occlusioni dovute all'architettura cittadina.
- Potature intensive che modificano le forme naturali delle chiome.

Il lavoro manuale di verifica si riduce al 13,72% del dataset totale, rappresentando un significativo risparmio di tempo e risorse per applicazioni di gestione forestale urbana.

9

Servizi di mappe satellitari

Per poter effettuare l'inferenza sui modelli di machine learning descritti nei capitoli precedenti, è necessario disporre di immagini satellitari di qualità adeguata. Il mercato offre diverse piattaforme per l'acquisizione di questo tipo di immagini, ciascuna con caratteristiche tecniche, piani tariffari e limitazioni d'uso differenti.
In questo capitolo vengono analizzati i principali servizi disponibili, confrontandoli in base a criteri rilevanti per applicazioni di rilevamento automatico della copertura arborea: risoluzione massima (livello di zoom), possibilità di download, compatibilità con inferenza ML e costi.

9.1 MapTiler

MapTiler è una piattaforma che offre servizi di mappe satellitari con ottima risoluzione, particolarmente in Europa e USA.

9.1.1 Piani tariffari

- **FREE**: $0/mese (uso non commerciale limitato) - 100K crediti
- **FLEX**: da $25/mese - 500K crediti
- **UNLIMITED**: $295/mese - 5M crediti
- **CUSTOM**: su richiesta - fatturazione annuale/trimestrale, sconti volume

9.1.2 Dettagli tecnici

- **Zoom massimo**: fino a z20
- **Limiti download**: piano FREE limitato, piani a pagamento con quote basate su tile requests (1 tile satellite = 4 crediti)
- **Inferenza ML**: permessa via API
- **Download**: non consentito (salvo licenze custom on-premise)

MapTiler offre anche satellite imagery on-demand tramite partnership con Satellogic. Questa piattaforma è stata utilizzata per l'inferenza sulle 24 città italiane oggetto di studio.

9.2 Stadia Maps

Stadia Maps è un provider di mappe con buona copertura globale, ma con termini restrittivi per applicazioni di machine learning.

9.2.1 Piani tariffari

- **FREE**: $0/mese - 200K crediti (50K tiles satellite) - NO uso commerciale, NO satellite imagery
- **STARTER**: $49/mese - 2M crediti (500K tiles satellite) - NO satellite imagery
- **STANDARD**: $249/mese - 15M crediti (3.75M tiles satellite) - satellite imagery inclusa
- **PROFESSIONAL**: $799/mese - 60M crediti (15M tiles satellite) - trial 14gg gratuito
- **ENTERPRISE**: custom - miliardi di crediti/anno - SLA, on-premises, licenze perpetue

9.2.2 Dettagli tecnici

- **Zoom massimo**: z18-19
- **Limiti download**: basato su tile requests (tiles 512x512px, 1 tile satellite = 4 crediti)
- **Inferenza ML**: non consentita (uso commerciale richiede licenza)
- **Download**: non consentito

Nonostante la buona qualità delle immagini, i termini di servizio non permettono l'utilizzo per applicazioni di machine learning.

9.3 LandViewer (EOSDA)

LandViewer, sviluppato da EOS Data Analytics, è una piattaforma orientata all'analisi di immagini satellitari con strumenti avanzati integrati.

9.3.1 Piani tariffari

- **FREE**: $0 - 10 download/giorno di immagini medie (Sentinel, Landsat), accesso completo agli strumenti di analisi, storage cloud personale 256GB, 20+ indici vegetazionali pre-impostati, visualizzazione illimitata
- **PREMIUM**: prezzo su richiesta - mensile/annuale, limiti di download aumentati, funzionalità avanzate, supporto prioritario
- **HIGH-RES (On-Demand)**: pay-per-km² - Airbus Archive 50cm: $3.80/km² (no ordine minimo), Airbus Archive 30cm: $18/km² (ordine minimo 25km²)

9.3.2 Dettagli tecnici

- **Zoom massimo**: z19-20+ (fino a 0.30m/px con high-res)
- **Limiti download**: FREE 10 download/giorno, Premium e High-Res in base al piano
- **Inferenza ML**: permessa
- **Download**: consentito (GeoTIFF e vari formati)

La piattaforma include strumenti di analisi come NDVI e change detection, con fonti dati quali Sentinel-2, Landsat, MODIS, KOMPSAT, SuperView e Gaofen. Tuttavia risulta estremamente costosa per immagini ad alta risoluzione: per una città di medie dimensioni il costo può aggirarsi intorno ai 2000 dollari.

9.4 Google Maps Platform

Google Maps Platform offre una copertura eccellente a livello globale con ottima risoluzione.

9.4.1 Piani tariffari

- **FREE**: 10K-100K chiamate gratuite/mese per SKU (dal marzo 2025)
- **Pay-as-you-go**: dopo il limite gratuito - Dynamic Maps $7/1000 requests, Static Maps $2/1000 requests, Map Tiles API vari tier

9.4.2 Dettagli tecnici

- **Zoom massimo**: variabile z18-22 (dipende dalla zona, verificabile tramite MaxZoomService)
- **Limiti download**: basato su API calls
- **Inferenza ML**: non consentita
- **Download**: non consentito direttamente

Nonostante l'ottima risoluzione e la presenza di un watermark su ogni tile, i termini e le condizioni non permettono l'utilizzo per inferenza con modelli di machine learning.

9.5 Mapbox

Mapbox è una piattaforma molto utilizzata per applicazioni web e mobile, con ottima qualità delle immagini.

9.5.1 Piani tariffari

- **FREE tier**: disponibile
- **Piani a pagamento**: da $50/mese - satellite tiles: tier 750K, 2M, 4M, 20M

9.5.2 Dettagli tecnici

- **Zoom massimo**: z0-21+ (può fare overzoom)
  - z0-8: NASA MODIS
  - z9-12: Maxar + Landsat
  - z13-16: Maxar Vivid

  - z16+: Vexcel aerial (sub-metro in Nord America/Europa)

- **Limiti download**: basato su tile requests e MAU
- **Inferenza ML**: da verificare (menzionano supporto ML ma con limiti)
- **Download**: non direttamente consentito

9.6 Copernicus Data Space Ecosystem / Sentinel Hub

Il programma Copernicus dell'Unione Europea offre accesso completamente gratuito ai dati satellitari Sentinel.

9.6.1 Piani tariffari

- **COMPLETAMENTE GRATUITO**: accesso libero e illimitato a tutti i dati

9.6.2 Dettagli tecnici

- **Zoom massimo**: z16 (Sentinel-2: 10m/px, Sentinel-1 SAR disponibile)
- **Limiti download**: nessun limite, accesso libero
- **Inferenza ML**: pienamente supportato (disponibili librerie dedicate come eo-learn)
- **Download**: illimitato e gratuito

Nonostante l'accesso gratuito e il pieno supporto per applicazioni ML, la risoluzione massima di z16 (10m/px) risulta insufficiente per il rilevamento di singoli alberi, rendendo questa piattaforma inadatta per gli scopi di questa ricerca.

9.7 Planet Labs

Planet Labs gestisce una delle più grandi costellazioni di satelliti per l'osservazione terrestre.

9.7.1 Piani tariffari

- **Pricing su richiesta**: disponibili programmi per ricerca/educazione fino a 3000km^2

9.7.2 Dettagli tecnici

- **Zoom massimo**: z20+ (da 3m fino a 30cm a seconda del satellite)
- **Limiti download**: basati su piano sottoscritto
- **Inferenza ML**: permessa con licenza appropriata
- **Download**: consentito

La registrazione richiede approvazione con tempi lunghi e i limiti sono scarsi per utenti privati. Potrebbe rappresentare una buona soluzione per progetti accademici, previa verifica della disponibilità di API.

9.8 Maxar SecureWatch / DigitalGlobe

Maxar Technologies, attraverso DigitalGlobe, offre alcune delle immagini satellitari commerciali a più alta risoluzione disponibili.

9.8.1 Piani tariffari

- **Enterprise**: prezzi su richiesta

9.8.2 Dettagli tecnici

- **Zoom massimo**: z22+ (fino a 30cm/px)
- **Limiti download**: enterprise
- **Inferenza ML**: N/D
- **Download**: N/D

Piattaforma molto costosa e orientata principalmente a clienti governativi e enterprise. L'alta qualità delle immagini non la rende accessibile per progetti accademici singoli.

9.9 Geoportale Regione Veneto

Il Geoportale della Regione Veneto fornisce accesso a ortofoto del territorio regionale.

9.9.1 Piani tariffari

- **FREE**: $0 - accesso completo senza registrazione, uso non commerciale e commerciale consentiti sotto licenza IODL 2.0

9.9.2 Dettagli tecnici

- **Zoom massimo**: ortofoto AGEA 2021/2018 (z19-20)
- **Limiti download**: solo visualizzazione per le ortofoto (no download)
- **Inferenza ML**: N/D
- **Download**: non consentito (solo visualizzazione)

Offre buona risoluzione per il territorio veneto, ma la limitazione alla sola visualizzazione ne impedisce l'utilizzo per applicazioni di machine learning.

9.10 Confronto e considerazioni

La scelta della piattaforma dipende fortemente dal caso d'uso specifico. Per applicazioni di rilevamento automatico della copertura arborea tramite deep learning, i fattori determinanti sono:

- **Risoluzione**: è necessario almeno z18-19 per poter identificare singoli alberi
- **Compatibilità ML**: molte piattaforme vietano esplicitamente l'uso per inferenza
- **Possibilità di download**: essenziale per processare le immagini localmente
- **Costi**: le soluzioni ad alta risoluzione risultano spesso proibitive

Per questa ricerca è stato scelto MapTiler come compromesso tra qualità delle immagini (z20), permesso di utilizzo per inferenza ML e costi contenuti. Le piattaforme gratuite come Copernicus, pur essendo pienamente utilizzabili per ML, non raggiungono la risoluzione necessaria per identificare singoli alberi in contesto urbano.

9.11 Disclaimer

Tutti i marchi menzionati in questo documento appartengono ai rispettivi proprietari. La loro citazione avviene esclusivamente a scopo informativo e non implica alcuna affiliazione, approvazione o sponsorizzazione da parte dei titolari dei marchi.

Bibliografia

[1] Rene Y Choi et al. «Introduction to Machine Learning, Neural Networks, and Deep Learning». In: *Translational Vision Science & Technology* 9.2 (2020), p. 14. DOI: 10.1167/tvst.9.2.14. URL: https://doi.org/10.1167/tvst.9.2.14.

[2] IBM. *Cos'è una rete neurale?* Accessed: 2025-08-19. 2021. URL: https://www.ibm.com/it-it/think/topics/neural-networks.

[3] Google Developers. *Glossario del machine learning: pesi (weights)*. Accessed: 2025-08-19. 2025. URL: https://developers.google.com/machine-learning/glossary/fundamentals?hl=it.

[4] Nathan Isong. *Building Efficient Lightweight CNN Models*. 2025. arXiv: 2501.15547 [cs.CV]. URL: https://arxiv.org/abs/2501.15547.

[5] Google Developers. *Machine Learning Glossary*. Accessed: 2025-08-19. 2025. URL: https://developers.google.com/machine-learning/glossary.

[6] Ultralytics. *Epoch in Machine Learning (ML)*. Accessed: 2025-08-19. 2025. URL: https://www.ultralytics.com/glossary/epoch.

[7] Fred Oh. *What is CUDA?* visited on 2025-08-06. 2012. URL: https://blogs.nvidia.com/blog/what-is-cuda-2/.

[8] Perceval Beja-Battais. *Overview of AdaBoost: Reconciling its views to better understand its dynamics*. 2023. arXiv: 2310.18323 [cs.LG]. URL: https://arxiv.org/abs/2310.18323.

[9] Keiron O'Shea e Ryan Nash. *An Introduction to Convolutional Neural Networks*. 2015. arXiv: 1511.08458 [cs.NE]. URL: https://arxiv.org/abs/1511.08458.

[10] Ross Girshick et al. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2014. arXiv: 1311.2524 [cs.CV]. URL: https://arxiv.org/abs/1311.2524.

[11] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[12] V7 Labs. *Mean Average Precision (mAP) Explained: Everything You Need to Know*. https://www.v7labs.com/blog/mean-average-precision. [Online; accessed 3-August-2025]. 2025.

[13] Mihir Durve et al. «Tracking droplets in soft granular flows with deep learning techniques». In: *The European Physical Journal Plus* 136 (ago. 2021). DOI: 10.1140/epjp/s13360-021-01849-3.

[14] Adam Paszke et al. «PyTorch: An Imperative Style, High-Performance Deep Learning Library». In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[15] Nidhal Jegham et al. *Evaluating the Evolution of YOLO (You Only Look Once) Models: A Comprehensive Benchmark Study of YOLO11 and Its Predecessors*. Ott. 2024. DOI: 10.48550/arXiv.2411.00201.

[16] Ultralytics Inc. *Ultralytics YOLO11*. 2025. URL: https://docs.ultralytics.com/models/yolo11/.

[17] Luisa Velasquez-Camacho et al. *Aerial and satellite images and labels to train deep learning models to detect urban canopies*. Ver. V1. visited on 2025-08-01. 2024. DOI: 10.34810/data1151. URL: https://doi.org/10.34810/data1151.

[18] G. Bradski. «The OpenCV Library». In: *Dr. Dobb's Journal of Software Tools* (2000).

[19] Esam. *Tree Project Dataset*. https://universe.roboflow.com/esam-ycmfl/tree-project-cyyr8. Open Source Dataset. visited on 2025-08-08. 2025. URL: https://universe.roboflow.com/esam-ycmfl/tree-project-cyyr8.

[20] Yueyuan Zheng e Gang Wu. «YOLOv4-Lite–Based Urban Plantation Tree Detection and Positioning With High-Resolution Remote Sensing Imagery». In: *Frontiers in Environmental Science* Volume 9 - 2021 (2022). URL: https://www.frontiersin.org/journals/environmental-science/articles/10.3389/fenvs.2021.756227.

[21] Martí Bosch. «DetecTree: Tree detection from aerial imagery in Python». In: *Journal of Open Source Software* 5.50 (2020). Model available at https://huggingface.co/martibosch/detectree, p. 2172. DOI: 10.21105/joss.02172.

[22] skops Developers. *skops: Python Library for Secure Persistence and Sharing of scikit-learn Models*. https://skops.readthedocs.io. Open source software. 2022–2025. URL: https://skops.readthedocs.io.

[23] J.G.C. Ball et al. «Accurate delineation of individual tree crowns in tropical forests from aerial RGB imagery using Mask R-CNN». In: *Remote Sens Ecol Conserv* 9.5 (2023), pp. 641–655. DOI: 10.1002/rse2.332. URL: https://doi.org/10.1002/rse2.332.

[24] Yinghao Wu et al. *Detectron2*. https://github.com/facebookresearch/detectron2. 2019.

[25] Yinghao Wu et al. *Detectron2*. https://detectron2.readthedocs.io/en/latest/modules/modeling.html. Documentazione ufficiale del framework Detectron2 sviluppato da Facebook AI Research. 2019.

[26] James Ball e Christopher Kotthoff. *detectree2 trained models*. 2025. DOI: 10.5281/zenodo.15863800. URL: https://doi.org/10.5281/zenodo.15863800.

[27] Charles R. Harris et al. «Array programming with NumPy». In: *Nature* 585 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[28] Sean Gillies e contributors. *Shapely: Manipulation and analysis of geometric objects in the Cartesian plane*. Version 2.1.1. Shapely Developers. 2025. URL: https://shapely.readthedocs.io.

[29] Enze Xie et al. *SegFormer: Simple and Efficient Design for Semantic Segmentation with Transformers*. 2021. arXiv: 2105.15203 [cs.CV]. URL: https://arxiv.org/abs/2105.15203.

[30] VisionPlatform. *Il potere dei ViT, Transformer della visone artificale per il riconoscimento delle immagini (computer vision).* https://visionplatform.ai/it/il-potere-dei-vit-transformer-della-visone-artificale-per-il-riconoscimento-delle-immagini-computer-vision/. Accesso il 9 settembre 2025. 2025.

[31] NVlabs. *SegFormer: Simple and Efficient Design for Semantic Segmentation with Transformers*. https://github.com/NVlabs/SegFormer. Official PyTorch implementation. 2021. URL: https://github.com/NVlabs/SegFormer.

[32] Alex Clark e Contributors. *Pillow (PIL Fork) Documentation*. 2015. URL: https://pillow.readthedocs.io/.

[33] Thomas Wolf et al. «Transformers: State-of-the-Art Natural Language Processing». In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, ott. 2020, pp. 38–45. URL: https://aclanthology.org/2020.emnlp-demos.6.

[34] Alexey S. Kornilov e Ivan V. Safonov. «An overview of watershed algorithm implementations in open source libraries». In: *Frontiers in Neuroinformatics* 12 (2018), p. 76. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC9146301/.

[35] Pauli Virtanen et al. «SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python». In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[36] Pamela Ann. Public domain Sexton. *What is a geographic information system (GIS)?* URL: https://www.usgs.gov/faqs/what-a-geographic-information-system-gis.

[37] Cecil C. Konijnendijk. «Evidence-based guidelines for greener, healthier, more resilient neighbourhoods: Introducing the 3-30-300 rule». In: *Journal of Forestry Research (Harbin)* 34.3 (2023), pp. 821–830. DOI: 10.1007/s11676-022-01523-z.

[38] James D. Murray e William vanRyper. *Encyclopedia of Graphics File Formats, 2ª ed.* O'Reilly, 1996.

[39] Emmanuel Devys et al. «OGC GeoTIFF standard». In: *OGC standards* (2019). URL: https://docs.ogc.org/is/19-008r4/19-008r4.html.

[40] Mapbox. *Rasterio library*. 2016. URL: https://rasterio.readthedocs.io/en/stable/intro.html.

[41] GDAL Development Team. *GDAL - Geospatial Data Abstraction Library: Version 3.11.3*. Open Source Geospatial Foundation, 2025. URL: https://gdal.org.

[42] Wikipedia contributors. *GeoJSON – Wikipedia, l'enciclopedia libera*. https://en.wikipedia.org/wiki/GeoJSON. [Online; accesso 16-ago-2025]. 2025.

[43] OpenStreetMap Contributors. *Slippy map tilenames*. https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames. Accessed: 2025-08-16. 2006.

[44] Comune di Milano - Direzione Urbanistica, Verde e Agricoltura. *Localizzazione georeferenziata degli alberi del patrimonio verde pubblico*. Dataset in formato shapefile, sistema di riferimento RDN2008-TM32 (ETRS89-ETRF00). Milano, Italia: Geoportale SIT - Sistema Informativo Territoriale, 2024. URL: https://dati.comune.milano.it/dataset/ds2484_infogeo_alberi_localizzazione (visitato il giorno 25/09/2025).

[45] Kelsey Jordahl et al. *geopandas/geopandas: v0.8.1*. Ver. v0.8.1. Lug. 2020. DOI: 10.5281/zenodo.3946761. URL: https://doi.org/10.5281/zenodo.3946761.

[46] James G. C. Ball et al. *detectree2 1.0.8 documentation*. https://patball1.github.io/detectree2/. Forest Ecology and Conservation Group, University of Cambridge. 2022. (Visitato il giorno 27/09/2025).

[47] Paeria de Lleida. *Espècies d'arbrat de Lleida*. Servei de Parcs i Jardins. File XLS - Inventario municipal degli alberi urbani. URL: https://urbanisme.paeria.cat/sostenibilitat/fitxers/parcs-i-jardins/especies-arbories-lleida.

[48] Shifeng Zhang et al. «Proximity-aware Object Detection with Spatial Context». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.5 (2018), pp. 1119–1133.

10

Ringraziamenti

Dopo tante pagine di scienza, è doveroso inserire anche una pagina umana: i miei ringraziamenti. Questo è il capitolo conclusivo di un percorso iniziato nel 2004 ma che, in qualche modo, è finito soltanto nel 2025: per certi versi è stato un vero e proprio viaggio.

Innanzitutto devo ringraziare la mia famiglia, mia moglie Linda, mia madre Carlina e i miei "secondi genitori" Susanna e Aldo, perché senza di loro sarei stato come un naufrago in mezzo al mare in balia di equazioni differenziali, condensatori sferici e trasformate di Fourier. Loro mi hanno aiutato a tenere la barra dritta in vista della terra.

Ringrazio anche il Prof. Davide Quaglia, che mi ha fatto non solo da relatore ma anche da mentore, sempre disponibile per uno studente atipico come me, che studiava più di notte che di giorno.

E poi non posso non spendere una parola anche per i miei supporters, la mia personale "curva sud", amici, colleghi e compagni di mille avventure: Alberto R., Alberto S., Christian, Francesco, Manuel e Michele, che mi hanno seguito in tutti i successi e gli intoppi che ho avuto in questo percorso, grazie davvero.

Beh ultimo, ma non per importanza, grazie alla mia mascotte Milo. Woof.