Politecnico di Torino

Politecnico di Torino

Corso di Laurea Magistrale in INGEGNERIA INFORMATICA

A.a. 2024/2025

Sessione di laurea 10/25

Estrazione Efficiente di Caratteristiche Visive su Dispositivi Edge a Basso Consumo Energetico per l'Agricoltura

Relatori:

Marcello CHIABERGE

Alessandro NAVONE

Candidati:

Paolo VACCHETTO

Sommario

L'agricoltura di precisione rappresenta un ambito in rapida evoluzione, in cui le tecniche di intelligenza artificiale e i dispositivi a basso consumo energetico possono contribuire in maniera significativa alla diagnosi precoce delle patologie delle colture. L'obiettivo di questa tesi è sviluppare un sistema per il rilevamento di foglie malate attraverso l'impiego di algoritmi di machine learning, con particolare attenzione alla loro implementazione su dispositivi on the edge, secondo i paradigmi del TinyML.

La metodologia proposta prevede una pipeline articolata in più fasi: la raccolta e preparazione di un dataset composto da oltre 130 immagini di foglie, che complessivamente corrispondono a più di 1000 foglie; l'impiego di YOLOv8 per il rilevamento e l'estrazione delle singole foglie; l'utilizzo di un autoencoder per la ricostruzione delle immagini e l'individuazione delle anomalie; e infine l'applicazione di una Random Forest per la classificazione tra foglie sane e malate.

L'addestramento supervisionato del modello ha richiesto un notevole impegno, con la sperimentazione di differenti parametri di training per YOLOv8-seg e l'adozione di tecniche di data augmentation al fine di aumentare la robustezza del sistema. Analogamente, la struttura dell'autoencoder è stata definita dopo vari confronti tra diverse architetture, fino all'individuazione della configurazione più efficace per il compito di ricostruzione delle foglie sane. Per la fase di classificazione finale è stata scelta una Random Forest, che ha mostrato prestazioni superiori rispetto ad altri algoritmi testati, come la Support Vector Machine (SVM).

Infine, i modelli sono stati ottimizzati e convertiti per l'esecuzione on-device su NVIDIA Jetson Nano, dimostrando la fattibilità di un approccio edge AI per la diagnosi precoce delle patologie delle colture. L'inferenza locale riduce la latenza e la dipendenza dalla connettività, consentendo un'elaborazione più affidabile direttamente in campo. I risultati sperimentali confermano l'efficacia del sistema sia nel rilevamento sia nella classificazione, abilitando scenari operativi quali droni per il monitoraggio aereo, sensori fissi distribuiti e dispositivi portatili a supporto dell'agricoltore, per decisioni tempestive e una gestione più sostenibile delle risorse.

Ringraziamenti

Desidero esprimere la mia profonda gratitudine a coloro che hanno reso possibile il completamento di questo percorso.

Un ringraziamento speciale va al Prof. Chiaberge per avermi concesso l'opportunità di lavorare a questo progetto di tesi stimolante e rilevante. La sua fiducia e la visione del lavoro sono state fondamentali. Estendo la mia gratitudine al co-relatore Alessandro Navone per essere stato una guida costante, competente e sempre disponibile in ogni fase della stesura della tesi.

Il raggiungimento di questo traguardo è anche merito del sostegno e della presenza delle persone a me care. Ringrazio di cuore tutti coloro che mi hanno sostenuto lungo il percorso di studi, e in particolare:

A Samantha, la mia fidanzata, per la pazienza, la comprensione e il supporto incondizionato, che sono stati un porto sicuro nei momenti più intensi e decisivi di questi anni. La tua presenza è stata la forza in più che mi ha sostenuto nei momenti più difficili.

A mia madre, per aver costantemente valorizzato l'importanza dello studio nella vita e per avermi trasmesso la convinzione che l'impegno accademico sia la base per ogni successo futuro.

Infine, un ringraziamento lo devo anche a me stesso. Nonostante le significative sfide personali e gli ostacoli portati avanti parallelamente all'attività sportiva, sono riuscito a mantenere la costanza, a non demordere e a completare con successo questa Laurea Magistrale, onorando il mio impegno verso lo studio.

Indice

Elenco delle figure VII

1 Introduzione 1
1.1 Organizzazione della tesi 1
1.2 Obiettivo della tesi 1

2 Stato dell'arte 3
2.1 Metodi tradizionali di rilevamento delle malattie 3
2.2 Deep Learning per il rilevamento di malattie nelle piante 3
2.3 Object Detection e Segmentazione 4
2.4 Approcci di ricostruzione e anomaly detection 4
2.5 Edge AI e deployment su dispositivi embedded 4
2.6 Sintesi critica 5

3 Machine Learning 6
3.1 Cos'è il Machine Learning 6
3.2 Concetti di Machine Learning 7
3.3 Modelli di Apprendimento Automatico 7
3.3.1 Supervised Learning 7
3.3.2 Unsupervised Learning 8
3.3.3 Reinforcement Learning 8
3.3.4 Metodi di Addestramento dei Modelli 8
3.4 Support Vector Machines 11
3.4.1 Classificazione lineare con SVM 11
3.4.2 SVM non lineare 12
3.4.3 Alberi decisionali e Random Forest 13
3.5 Tecniche di Apprendimento non supervisionato 16
3.5.1 Clustering 16
3.5.2 Anomaly detection 17

4 Reti Neurali e Deep Learning 18
4.1 Introduzione alle Reti Neurali . . . 18
4.2 Dai neuroni al Perceptron . . . 19
4.2.1 Il neurone Biologico . . . 19
4.2.2 Logica computazionale con Neuroni . . . 20
4.2.3 Il Perceptron . . . 20
4.2.4 Il Perceptron Multilayer e la Backpropagation . . . 22
4.2.5 Ottimizzazione degli Iperparametri . . . 25
4.3 Reti Neurali Convoluzionali . . . 27
4.3.1 Architettura della Corteccia Visiva . . . 27
4.3.2 Strati Convoluzionali . . . 28
4.3.3 Filtri . . . 30
4.3.4 Strato di Pooling . . . 33
4.3.5 Architetture delle CNN . . . 34
4.3.6 Rilevamento degli Oggetti . . . 36
4.4 Autoencoder . . . 40
4.4.1 Rappresentazione Efficiente dei Dati . . . 41
4.4.2 Gli strati di un Autoencoder . . . 42
4.4.3 Autoencoder Convoluzionali . . . 43

5 AI on the Edge e modelli per dispositivi embedded 44
5.1 Sfide e requisiti nell'Edge AI . . . 44
5.2 Tecniche utili all'ottimizzazione per l'edge . . . 45
5.3 Formati e framework per il deployment *edge* . . . 45

6 Metodologia 47
6.1 Architettura proposta . . . 48
6.1.1 La scelta dell'autoencoder . . . 48
6.2 Dataset Utilizzato . . . 50
6.2.1 Origine delle immagini . . . 50
6.2.2 Segmentazione ed etichettatura . . . 50
6.2.3 Scelta del classificatore: Random Forest . . . 52
6.2.4 Esportazione dei modelli On The Edge . . . 53

7 Apparato sperimentale 55
7.1 Ambiente di sviluppo . . . 55
7.1.1 Librerie e dipendenze software . . . 55
7.2 Divisione dei dataset e coerenza tra le fasi . . . 56
7.2.1 Dataset per la segmentazione (YOLO) . . . 56
7.2.2 Dataset per l'autoencoder (ricostruzione) . . . 56
7.2.3 Dataset per i classificatori (RF e SVM). . . . 57

7.3 Addestramento del modello YOLOv8-seg . . . 58
7.4 Implementazione dell'autoencoder . . . 60
7.4.1 Varianti architetturali valutate . . . 60
7.4.2 Criteri di valutazione . . . 61
7.5 Classificatore supervisionato . . . 64
7.5.1 Valutazione esplorativa e selezione delle feature . . . 64
7.5.2 Classificatore finale: Random Forest . . . 65
7.6 Jetson Nano Developer Kit . . . 66
7.6.1 Ambiente di sistema . . . 66
7.6.2 Caratteristiche hardware rilevanti . . . 67

8 Esperimenti e Risultati 69
8.1 Obiettivi e protocollo di valutazione . . . 69
8.2 Segmentazione YOLOv8-seg (valutazione qualitativa) . . . 70
8.3 Selezione dell'AE e qualità di ricostruzione . . . 72
8.3.1 Valutazione quantitativa e qualitativa . . . 73
8.3.2 Sintesi e scelta del modello . . . 74
8.4 Random Forest . . . 74
8.4.1 Metriche di ricostruzione . . . 76
8.4.2 Metriche di classificazione . . . 77
8.5 Esportazione dei modelli da `.pth` a `.onnx` . . . 83
8.5.1 Procedura di esportazione . . . 83
8.5.2 Difficoltà riscontrate e soluzioni adottate . . . 83
8.5.3 Verifica di compatibilità e test su Jetson Nano . . . 84
8.5.4 Performance su Jetson . . . 84
8.5.5 Sintesi . . . 85

9 Conclusione 86

Bibliografia 89

Elenco delle figure

3.1 Gradient Descent [13, p. 118] 10
3.2 Gradient Descent (sinistra) Stocastich Gradient Descent(destra) [13, pp. 120, 124] 10
3.3 classificazione lineare a margine largo 11
3.4 classificatore SVM con kernel polinomiale 13

4.1 Neurone Biologico [13, p. 282] 19
4.2 computazioni logiche semplici con ANN [13, p. 283] 20
4.3 TLU; un neurone artificiale che calcola la somma ponderata tra input e poi applica una step function [13, p. 284] 21
4.4 esempio di Perceptron con due neuroni di input, un neurone bias e tre neuroni output [13, p. 286] 22
4.5 Funzioni di attivazione e le loro derivate [13, p. 292] 24
4.6 Connessione tra strati convoluzionali [13, p. 449] 29
4.7 Riduzione della dimensionalità usando un passo di 2 [13, p. 450] . . 30
4.8 L'applicazione di due filtri diversi [13, p. 451] 31
4.9 strati convoluzionali con multiple feature map, e immagini RGB [13, p. 452] 32
4.10 Pooling, campo ricettivo 2×2, passo di 2, senza padding [13, p. 457] 33
4.11 Architettura CNN tipica [13, p. 461] 34
4.12 nuove istanze tramite Data Augmentation [13, p. 465] 35
4.13 Una FCN è in grado di processare immagini di diverse dimensioni [13, p. 489] 37
4.14 Segmentazione Semantica [13, p. 493] 40
4.15 Un semplice Autoencoder [13, p. 570] 42
4.16 Autoencoder a strati nascosti 43

5.1 Esempio di grafo ONNX: i cerchi denotano operatori, le frecce l'uso/produzione di tensori. La rappresentazione a grafo abilita ottimizzazioni e portabilità tra backend. 46

6.1 Pipeline sviluppata 47
6.2 Etichettatura con Salt 51
6.3 Esempio di Data Augmentation 52

7.1 (1)AE_BaseSkip, (2)AE_Wide, (3)AE_Base, (4)AE_DeepLowPassSkip 63

8.1 Esempio inferenza YOLOv8-seg foglie sane 70
8.2 Esempio inferenza YOLOv8-seg foglie malate 70
8.3 Esempio segmentazione foglie sane 71
8.4 Esempio segmentazione foglie malate 72
8.5 Ricostruzione tramite Autoencoder 73
8.6 Errore di Colore 75
8.7 SSIM (Structural Similarity index) 75
8.8 Errore di Ricostruzione 75
8.9 Curva ROC dei modelli Random Forest e SVM. 77
8.10 Curva Precision–Recall per i due classificatori. 78
8.11 Matrice di confusione per RF (a sinistra) e SVM (a destra) 79
8.12 Andamento dell'indice F_2 al variare della soglia di decisione per la Random Forest. 79
8.13 Importanza delle feature secondo la Random Forest. 80
8.14 Matrice di correlazione tra le feature derivate dall'autoencoder. . . . 81
8.15 Distribuzione delle principali feature per le due classi. 81
8.16 Proiezione bidimensionale delle feature tramite PCA. 82
8.17 Isolinee di probabilità prodotte dalla Random Forest nello spazio PCA. 82

Capitolo 1

Introduzione

1.1 Organizzazione della tesi

Nella sezione 1.2 si introduce al lettore l'obiettivo della tesi. Vengono inoltre riassunti gli aspetti principali della metodologia adottata nel lavoro.

Il capitolo 2 introduce il lettore allo stato dell'arte, dando una panoramica sulle tecniche e tecnologie presenti in letteratura e i concetti su cui si basa lo sviluppo di questa tesi.

Il capitolo 3 introduce i fondamenti e i concetti principali del machine learning. Questi aspetti serviranno al lettore per comprendere bene i concetti contenuti all'interno della tesi e per comprendere al meglio la soluzione proposta.

Il capitolo 4 approfondisce aspetti legati al modo delle reti neurali, impiegate nello sviluppo di questa tesi.

Nel capitolo 5 vengono trattati elementi di AI on the edge, particolarmente utili nello sviluppo di questa tesi, in quanto il risultato finale è una pipeline che "corre" su Jetson Nano.

Nei capitoli 6, 7 e 8 vengono descritte le metodologie, gli apparati sperimentali e le soluzioni sperimentali attuate per raggiungere l'obiettivo della tesi;

Nel capitolo 9 vengono sintetizzati i risultati chiave esplorando i limiti e la validità delle soluzioni proposte. Si conclude ipotizzando sviluppi futuri e possibili applicazioni di ciò che è stato presentato nel corso della tesi.

1.2 Obiettivo della tesi

L'agricoltura di precisione ha compiuto notevoli progressi negli ultimi anni, utilizzando la tecnologia per ottimizzare tutti gli aspetti relativi al ciclo di produzione fino ad arrivare all'automatizzazione di vari processi. Uno degli aspetti più significativi in questo ambito è sicuramente la lotta alle patologie. In particolare,

nel contesto vitivinicolo, il controllo della proliferazione delle patologie assume un ruolo importante. Le motivazioni spaziano dalle più ovvie, come l'aumento della resa, fino a contesti relativi alla salvaguardia dell'ambiente e al risparmio economico. Naturalmente, una pianta sana avrà una resa maggiore nel corso degli anni, mentre il minor impiego di fitofarmaci, per esempio, porterà a un risparmio in termini economici e alla riduzione di elementi potenzialmente dannosi per l'ambiente. Gioca un ruolo fondamentale, quindi, l'individuazione preventiva al sorgere della malattia. Nelle attuali pratiche di prevenzione agricola, le metodologie manuali sono la norma. Queste tipicamente comportano due approcci principali: o l'attento monitoraggio dei fattori ambientali (come temperatura, umidità o pioggia), che può essere supportato dall'uso di sensoristica specifica, per cogliere l'insorgenza di condizioni favorevoli alla malattia; oppure l'osservazione diretta e in loco delle colture da parte di un operatore umano. Entrambe le soluzioni hanno dei naturali difetti: per quanto riguarda il monitoraggio dei fattori ambientali, si parla di una situazione ipotetica che richiede o l'intervento di un operatore per la verifica (con naturale perdita in termini di tempo e costi) o comunque un trattamento preventivo anche in assenza di patologia. È naturale quindi pensare a una soluzione che preveda il monitoraggio automatizzato. La soluzione proposta è quella di impiegare un motore inferenziale installato su dispositivi di robotica mobile. Il compito del motore inferenziale generato utilizzando le reti neurali convoluzionali (CNN - Convolutional Neural Network) è quello di individuare le foglie di vite malate e di notificare all'operatore l'insorgere della malattia. Per quanto riguarda la tesi proposta, altre soluzioni che vanno nella stessa direzione sono state prese da un paper intitolato Grape Leaf Diseases Identification System Using Convolutional Neural Networks and LoRa Technology [1] che però utilizza reti neurali di classificazione tipo high-level features. La soluzione proposta da noi, invece, propone di lavorare sull'estrazione di feature low level. In particolare, questa soluzione prevede l'impiego di diversi modelli di machine learning. La rete neurale convoluzionale viene addestrata utilizzando un dataset di immagini di foglie sane in modo che sia in grado di ricostruire un'immagine di una foglia sana, il che introduce una significativa sfida legata alla capacità computazionale dei dispositivi *on the edge*. Questi dispositivi, spesso limitati in termini di potenza di calcolo e memoria, richiedono che si presti particolare attenzione all'ottimizzazione delle risorse computazionali effettivamente impiegate. È fondamentale, infatti, che i modelli siano progettati ed eseguiti in modo da essere leggeri ed efficienti, garantendo prestazioni adeguate direttamente sul campo, senza la necessità di un'elaborazione su server remoti. Le varie tecnologie impiegate e metodologie impiegate verranno poi discusse nel corpo della tesi.

Capitolo 2

Stato dell'arte

L'agricoltura di precisione rappresenta un settore in rapida evoluzione, in cui l'uso di tecniche di visione artificiale e intelligenza artificiale consente di automatizzare processi che tradizionalmente dipendevano dall'osservazione diretta dell'agricoltore o del tecnico agronomo. L'individuazione precoce delle patologie fogliari è fondamentale perché abilita interventi tempestivi che limitano le perdite di resa e l'impatto ambientale dovuto a trattamenti tardivi o generalizzati, come evidenziato dalla letteratura e dalle linee guida internazionali [2, 3, 4].

2.1 Metodi tradizionali di rilevamento delle malattie

I primi approcci al rilevamento delle malattie fogliari si basavano su ispezioni visive, una procedura intrinsecamente soggettiva e dunque esposta a bias ed errori di valutazione [5]. In seguito, l'introduzione di sensori spettrali (multi-spettrali/iperspettrali) e termici ha consentito di intercettare alterazioni fisiologiche anche in fase pre–sintomatica, migliorando la tempestività della diagnosi [6]. Tali soluzioni, tuttavia, richiedono strumentazione dedicata e investimenti non trascurabili, fattori che possono limitarne la diffusione su larga scala [7].

2.2 Deep Learning per il rilevamento di malattie nelle piante

Con l'avvento delle tecniche di machine learning, l'attenzione si è spostata verso l'analisi automatica di immagini RGB, acquisite tramite fotocamere convenzionali o droni. Un esempio significativo è lo studio [8], che mostra come algoritmi di classificazione supervisionata (ad esempio SVM o Random Forest) possano essere

addestrati a riconoscere malattie della vite a partire da feature estratte manualmente. Sebbene questi approcci abbiano fornito buoni risultati, dipendono fortemente dalla qualità delle feature ingegnerizzate, risultando meno robusti rispetto ai modelli basati su deep learning.

Il deep learning ha rivoluzionato l'ambito della diagnosi automatica delle colture. In particolare, le reti neurali convoluzionali hanno dimostrato una notevole capacità nel riconoscere pattern complessi nelle immagini fogliari. Un contributo rilevante in questo senso è il lavoro [1], che integra un modello CNN con un sistema di comunicazione a basso consumo basato su LoRa, dimostrando la fattibilità di soluzioni distribuite per il monitoraggio in campo. Un ulteriore avanzamento è rappresentato dallo studio [9], che analizza diverse architetture CNN e dimostra come la segmentazione preliminare delle foglie migliori significativamente le prestazioni di classificazione. Questi risultati evidenziano l'importanza di integrare modelli di visione artificiale con tecniche di pre-processing mirate.

2.3 Object Detection e Segmentazione

Tra gli approcci più recenti si collocano i modelli di object detection e segmentazione, che non si limitano alla classificazione globale dell'immagine, ma consentono di localizzare e isolare le singole foglie. Architetture come YOLO e Mask R-CNN si sono affermate come strumenti fondamentali in questo contesto, grazie al loro bilanciamento tra accuratezza e velocità di inferenza [10]. Questi metodi aprono la strada a pipeline più complesse, in cui la segmentazione costituisce la base per ulteriori fasi di ricostruzione e classificazione.

2.4 Approcci di ricostruzione e anomaly detection

Parallelamente, sono stati sviluppati approcci basati su autoencoder e modelli di anomaly detection, utilizzati per rilevare deviazioni dalle caratteristiche tipiche di foglie sane. Questi modelli consentono di evidenziare regioni sospette senza richiedere un set di dati esaustivo di tutte le possibili patologie, risultando particolarmente utili in scenari reali caratterizzati da forte variabilità [11].

2.5 Edge AI e deployment su dispositivi embedded

L'adozione di modelli di intelligenza artificiale *on-device* su sistemi embedded e SBC (Single Board Computer) sta guadagnando crescente interesse, poiché consente

di eseguire l'inferenza direttamente in campo con **latenza ridotta**, **maggiore affidabilità** in condizioni di connettività variabile e **tutela dei dati** grazie all'elaborazione locale. In questo scenario, la scelta dell'architettura e delle ottimizzazioni di deployment (formato del modello, runtime, precisione numerica) incide in modo sostanziale sia sulle prestazioni sia sull'efficienza energetica. A supporto di questa evidenza, lo studio [12]—sebbene non specifico per lambito agricolo—mostra come modelli leggeri con diversa impostazione (ad es. YOLO vs FOMO) presentino *trade-off* marcati tra accuratezza e consumi quando portati su dispositivi embedded. Tali considerazioni risultano trasferibili anche alle applicazioni agrarie, guidando la selezione di modelli e toolchain in funzione dei vincoli computazionali e dei requisiti di latenza del contesto operativo.

2.6 Sintesi critica

Dalla letteratura emerge una tendenza chiara: l'evoluzione dalle tecniche manuali di machine learning verso modelli end-to-end di deep learning, con un crescente interesse per l'implementazione su dispositivi edge. Tuttavia, permangono alcune lacune. In particolare, sono pochi i lavori che integrano in maniera organica *segmentazione, ricostruzione e classificazione* in un'unica pipeline, e ancora più rari quelli che validano tali approcci direttamente su hardware a risorse limitate in scenari agricoli reali. La presente tesi si inserisce in questo contesto, proponendo una pipeline completa che affronta questi aspetti in maniera unitaria.

Capitolo 3

Machine Learning

Questo capitolo introduce al lettore gli aspetti fondamentali del machine learning. Questo è un primo passo fondamentale per comprendere le tecnologie e tecniche di sviluppo utilizzate nel corso di questa tesi. La prima parte riguarda gli aspetti teorici ad alto livello sui quali si basa il mondo del machine learning, con alcuni riferimenti storici, per ottenere una prima visione d'insieme sull'argomento. Nella seconda parte il capitolo si focalizza maggiormente su aspetti legati alle Artificial Neural Network (ANN) che sono state ampiamente utilizzate all'interno del progetto.

3.1 Cos'è il Machine Learning

Il machine learning è la scienza della programmazione di computer in modo che quest'ultimi possano imparare dai dati [13]. Il termine "machine learning" è stato per la prima volta usato da Arthur Samuel nel 1959 [14] e definiva appunto il campo di studi che permetteva ai computer l'abilità di imparare senza essere esplicitamente programmati. Anche se molto generica, questa definizione racchiude al meglio l'essenza del machine learning, che oggigiorno spesso viene interscambiato erroneamente con altri termini. Spesso concetti come Intelligenza Artificiale (IA), machine learning e Deep Learning (DL) vengono erroneamente usati come sinonimi. Al contrario, la loro relazione è ben precisa.

Per IA si intende la capacità di un sistema informatico di interagire come un essere umano. Il Deep Learning, invece, è una branca particolare del machine learning ed è caratterizzato dall'uso di reti neurali artificiali profonde, modelli di apprendimento formati da vari strati e neuroni.

3.2 Concetti di Machine Learning

Una delle prime domande che potrebbe sorgere spontanea è in quale ambito sia utile impiegare i concetti di machine learning.

Un esempio molto intuitivo potrebbe essere la creazione di un filtro anti-spam per la posta elettronica. La soluzione standard, "manuale", richiederebbe di scrivere una serie molto lunga di regole complesse per individuare le mail da scartare: una soluzione difficile da implementare e mantenere nel tempo. In alternativa possiamo usare un algoritmo di machine learning che automaticamente impara quali siano le parole, frasi, pattern che permettono di individuare testi spam e fare in modo di contrassegnare direttamente la mail come indesiderata. Altri esempi possono evidenziare le potenzialità del machine learning ma per riassumere il machine learning è perfetto per:

- Problemi per i quali sono necessarie lunghe liste o sono richiesti molti ritocchi.
- Problemi complessi, in cui un approccio tradizionale non porta a buone soluzioni.
- Ambienti particolarmente variabili in cui un sistema di machine learning si adatterebbe meglio a nuovi dati.
- Estrapolare informazioni essenziali da problemi con un quantitativo spropositato di dati.

3.3 Modelli di Apprendimento Automatico

In contrasto con i paradigmi di programmazione tradizionali, dove le regole decisionali sono esplicitamente definite, il machine learning si fonda sulla capacità di un algoritmo di apprendere autonomamente da un insieme di dati [13]. Il modello non segue istruzioni rigide, ma elabora un vasto dataset per estrarre e generalizzare pattern, relazioni e dipendenze statistiche. È questo processo induttivo che costituisce il fondamento per la risoluzione di problemi complessi, dall'analisi predittiva alla comprensione del linguaggio naturale. La classificazione dei metodi di machine learning si basa principalmente sulla natura dei dati di addestramento e sulla modalità di apprendimento, delineando tre categorie principali.

3.3.1 Supervised Learning

Nell'apprendimento supervisionato, il modello è addestrato su esempi etichettati (X, y), ovvero a ciascun input è associata la soluzione desiderata (etichetta/target), che guida l'aggiornamento dei parametri del modello [15, cap. 1]. Un tipico compito

di apprendimento supervisionato è la classificazione. Il filtro anti-spam per un servizio di posta elettronica è un ottimo esempio. Un altro compito tipico affidato all'apprendimento supervisionato è la predizione di un particolare valore (come il prezzo di una macchina), dati una serie di features (chilometraggio, marca, anno di immatricolazione, etc.) chiamate predittori. Per addestrare il modello si forniscono molteplici esempi etichettati (predittori più valore dell'auto). Questo tipo di compito viene definito regressione.

3.3.2 Unsupervised Learning

Nell'apprendimento non supervisionato, i dati di addestramento non sono etichettati; il modello cerca autonomamente strutture latenti, raggruppamenti o rappresentazioni nei dati, in assenza di una "soluzione" fornita a priori [15, cap. 9]. Un esempio di utilizzo dell'apprendimento non supervisionato è il Clustering (raggruppamento). Un Social Network vorrebbe, per esempio, raggruppare gli utenti in diverse categorie per fornire un servizio più consono alle loro esigenze. In nessun momento si comunica all'algoritmo quale utente appartiene a quale gruppo. È direttamente l'algoritmo a individuare il giusto sottoinsieme.

3.3.3 Reinforcement Learning

Nell'apprendimento per rinforzo, un *Agente* interagisce con un *Ambiente* scegliendo azioni sulla base dello stato osservato e ricevendo *ricompense/penalità* che guidano l'apprendimento di una politica ottimale nel tempo [16, cap. 1]. Il sistema deve imparare autonomamente qual è la miglior strategia (detta policy) per ottenere la miglior ricompensa nel corso del tempo. La policy definisce la scelta che l'agente deve prendere in una determinata situazione.

3.3.4 Metodi di Addestramento dei Modelli

Il processo di addestramento di un modello di machine learning non consiste semplicemente nel fornirgli dei dati, ma in un'operazione mirata a trovare i valori ottimali dei suoi parametri. A differenza della programmazione tradizionale, in cui le regole decisionali sono esplicitamente codificate, un modello di machine learning impara a definire autonomamente la relazione tra i dati di input e quelli di output.

Per comprendere questo concetto, si può immaginare un semplice modello che deve prevedere il prezzo di un'abitazione in base alla sua superficie. I parametri del modello sono i pesi che l'algoritmo deve imparare a ottimizzare. Ad esempio, il modello può calcolare il prezzo come:

$$Prezzo\ stimato = peso_1 \cdot superficie + peso_2 \tag{3.1}$$

Durante l'addestramento, il modello analizza centinaia di case già vendute e aggiusta i valori di $peso_1$ e $peso_2$ fino a quando il prezzo stimato si avvicina il più possibile al prezzo reale di ogni casa.
Per guidare l'addestramento, è necessaria una metrica che quantifichi l'accuratezza del modello. A questo scopo si utilizza una funzione di costo (nota anche come *loss function* o *cost function*), che misura la "distanza" tra la previsione del modello e il valore reale fornito dal dataset. In termini semplici, più il modello è impreciso, più il valore della funzione di costo è elevato.

L'obiettivo dell'addestramento diventa quindi quello di minimizzare questa funzione. L'algoritmo di apprendimento cerca di trovare un insieme di parametri che riduca al minimo l'errore, avvicinando le previsioni del modello ai valori effettivi. La funzione di costo agisce quindi come una mappa, un "paesaggio" che guida il modello verso la soluzione ottimale, ovvero il punto più basso, dove l'errore è minimo.

Gradient Descent

Il Gradient Descent (GD) è un algoritmo di ottimizzazione iterativo ampiamente utilizzato nel machine learning per trovare la soluzione ottimale a un vasto insieme di problemi. Il suo obiettivo primario è quello di minimizzare la funzione di costo del modello, la quale misura l'errore tra le previsioni e i valori reali [13].

L'idea fondamentale del Gradient Descent è quella di navigare il "paesaggio" della funzione di costo, procedendo per passi successivi nella direzione di massima pendenza negativa. Matematicamente, questa direzione è data dal gradiente della funzione di costo. Si parte da un'inizializzazione casuale dei parametri del modello e si aggiorna iterativamente ogni parametro in base al gradiente, fino a quando l'algoritmo non converge a un minimo locale (o globale).

La dimensione di ogni passo è regolata da un cruciale iperparametro chiamato tasso di apprendimento (*learning rate*).

Un tasso di apprendimento troppo piccolo rende l'addestramento molto lento, poiché richiede un numero elevato di iterazioni per raggiungere la convergenza. Al contrario, un tasso di apprendimento troppo grande può causare un *salto* oltre il punto di minimo, impedendo al modello di convergere o addirittura facendolo divergere. La scelta di un *learning rate* appropriato è quindi fondamentale per l'efficacia e la stabilità del processo di addestramento.

Oltre al Gradient Descent "classico", un'altra variante molto diffusa è lo Stochastic Gradient Descent (SGD). La principale differenza risiede nella quantità di dati utilizzati per calcolare il gradiente in ogni iterazione. Mentre il Batch Gradient

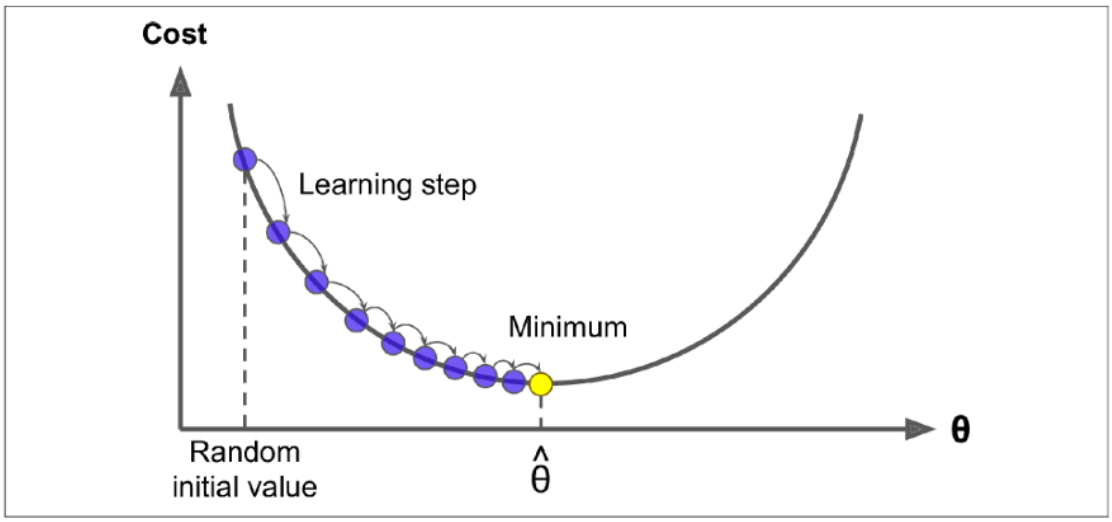

Figura 3.1: Gradient Descent [13, p. 118]

Descent calcola la pendenza della funzione di costo sull'intero dataset di addestramento, l'SGD calcola il gradiente utilizzando un solo campione di addestramento, scelto in modo casuale, in ogni step [13].

Questo approccio, pur essendo più "rumoroso" e meno stabile, offre vantaggi significativi. In primo luogo, è notevolmente più veloce ed efficiente dal punto di vista computazionale su dataset di grandi dimensioni. In secondo luogo, il rumore intrinseco dell'SGD può aiutare il modello a "sfuggire" dai minimi locali e a convergere verso una soluzione più ottimale.

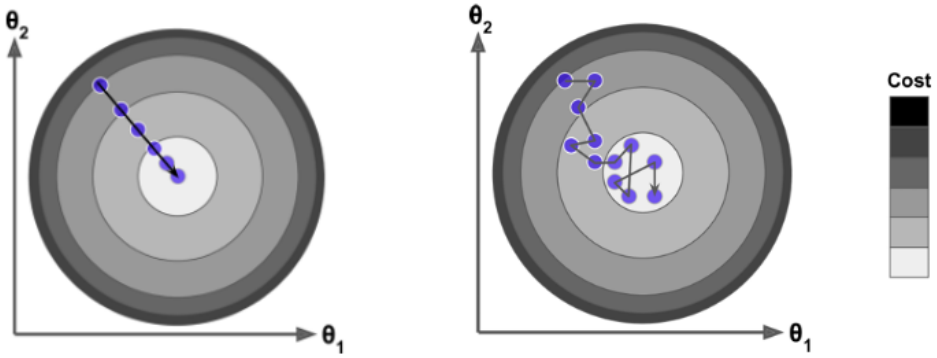

Figura 3.2: Gradient Descent (sinistra) Stocastich Gradient Descent(destra) [13, pp. 120, 124]

Esiste anche una via di mezzo, il Mini-batch Gradient Descent, che calcola il gradiente su un piccolo sottoinsieme casuale (un "mini-batch") dei dati. Questo metodo combina l'efficienza computazionale dell'SGD con la stabilità del Batch Gradient Descent, ed è l'approccio più utilizzato nelle architetture di deep learning moderne.

3.4 Support Vector Machines

Una Support Vector Machine (SVM) è un modello di machine learning capace di eseguire classificazione lineare o classificazione non lineare. È uno dei modelli più popolari ed è particolarmente indicato per la classificazione di dataset di dimensioni limitate o mediamente limitate.

3.4.1 Classificazione lineare con SVM

La classificazione lineare rappresenta il fondamento per la separazione di due o più classi di dati attraverso un iperpiano (una linea in 2D, un piano in 3D). L'immagine allegata illustra questo concetto applicato al celebre dataset dei fiori di iris, dove l'obiettivo è classificare un fiore come appartenente alla specie *Iris versicolor* o *Iris setosa* basandosi sulla dimensione del suo petalo.

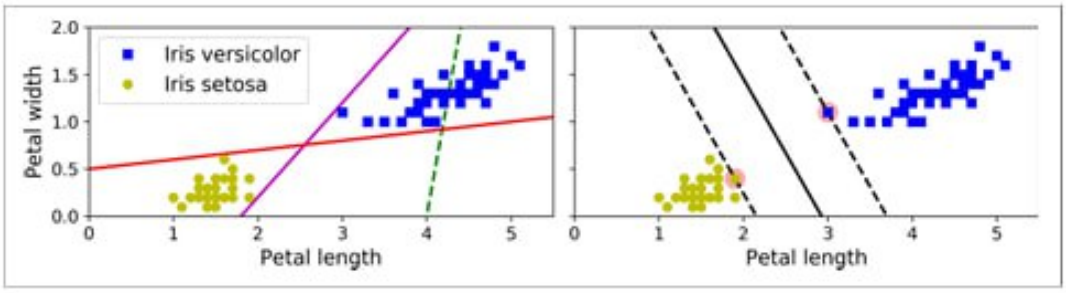

Figura 3.3: classificazione lineare a margine largo

Il grafico a sinistra mostra i limiti decisionali di tre ipotetici classificatori lineari. La linea tratteggiata rappresenta un modello chiaramente inefficace, incapace di separare correttamente le due classi. Al contrario, le altre due linee continue sembrano funzionare perfettamente sul dataset di addestramento. Tuttavia, i loro confini decisionali sono estremamente vicini ad alcune istanze delle classi, rendendo i modelli vulnerabili a piccole variazioni nei dati. Questo implica un'elevata probabilità che non siano in grado di generalizzare in modo efficace su nuove istanze non ancora viste.
Il grafico a destra, invece, illustra il principio della Support Vector Machine (SVM),

un classificatore che non si limita a separare le classi, ma cerca l'iperpiano che massimizza la distanza tra le istanze più vicine di ciascuna classe. Questo particolare confine decisionale, denominato classificazione a margine largo, non solo separa correttamente le istanze di addestramento, ma si posiziona il più lontano possibile dai dati più vicini al confine.
Il principio del margine largo è cruciale per la robustezza e la generalizzazione del modello. L'obiettivo non è solo trovare un confine di separazione, ma un confine che sia il più stabile possibile. I dati più vicini a questo confine, detti vettori di supporto, sono quelli che definiscono la posizione dell'iperpiano ottimale. Matematicamente, l'iperpiano è determinato unicamente da questi vettori, e l'aggiunta o la rimozione di altre istanze di addestramento, non vettori di supporto, non altererebbe il risultato finale. Questa caratteristica rende le SVM particolarmente efficienti in termini computazionali, poiché il loro processo di ottimizzazione dipende solo da un sottoinsieme dei dati di addestramento.

3.4.2 SVM non lineare

I classificatori SVM lineari sono efficienti e performano molto bene in molteplici casi. La maggior parte dei dataset su cui è comune lavorare però non sono neanche lontanamente linearmente separabili. L'approccio per risolvere questo tipo di situazioni consiste nell'aggiungere features, utilizzando la tecnica del kernel trick. Il "trucco del kernel" consente di mappare i dati da uno spazio di input a bassa dimensionalità a uno spazio a dimensionalità superiore, dove le classi diventano separabili da un iperpiano lineare. L'elemento rivoluzionario di questa tecnica risiede nel fatto che non è necessario calcolare esplicitamente le coordinate dei dati in questo nuovo spazio. Al contrario, una funzione kernel calcola direttamente il prodotto scalare tra i vettori in questo spazio trasformato, partendo dai vettori originali. In questo modo, il costo computazionale associato a un'esplicita trasformazione e al calcolo in uno spazio di alta dimensionalità viene drasticamente ridotto, rendendo il metodo scalabile e computazionalmente efficiente. Esistono diverse funzioni kernel, ciascuna adatta a modellare confini di decisione di diversa complessità. Tra le più diffuse vi sono il kernel polinomiale, il kernel RBF (Radial Basis Function) e il kernel sigmoide.

Il kernel polinomiale è particolarmente rilevante per la sua capacità di generare confini di decisione curvilinei nello spazio di input originale. La sua formula è definita come:

$$K(x, z) = (x \cdot z + c)^d \tag{3.2}$$

dove x e y sono i vettori di input nel dataset, d rappresenta il grado del polinomio e c è una costante opzionale che contribuisce a controllare l'influenza del termine di grado 0. Il valore di d influenza la complessità del modello e deve essere scelto attentamente per bilanciare bias e varianza. Un grado (d) elevato può portare a un

modello molto flessibile in grado di adattarsi perfettamente ai dati di addestramento, ma aumenta il rischio di overfitting. Al contrario, un grado basso potrebbe non essere sufficiente per catturare le relazioni non lineari, portando a un underfitting. La scelta ottimale di d è, pertanto, cruciale e viene tipicamente determinata tramite tecniche di cross-validation.

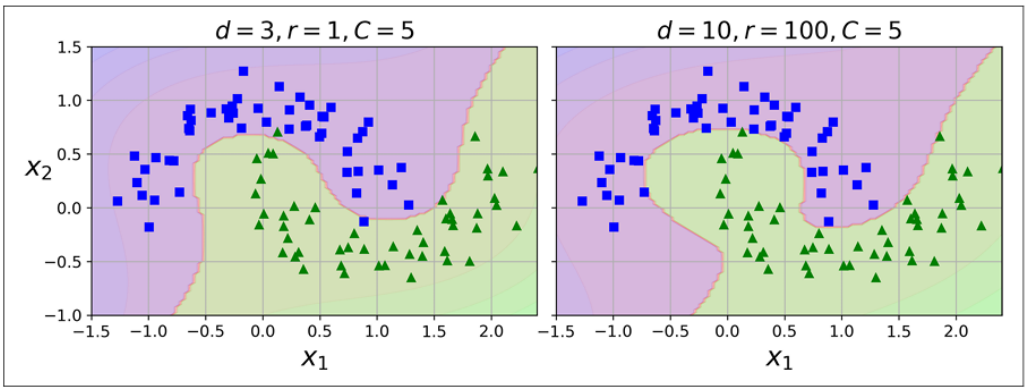

Figura 3.4: classificatore SVM con kernel polinomiale

L'uso di un kernel polinomiale trasforma la ricerca di un iperpiano lineare nello spazio proiettato in quella di una superficie di decisione complessa e non lineare nello spazio originale, permettendo alle SVM di risolvere problemi di classificazione complessi e non lineari con grande precisione.

3.4.3 Alberi decisionali e Random Forest

Come gli SVM, gli alberi decisionali sono potenti algoritmi di machine learning, il cui scopo è quello di classificare, fare regressione e multi-output (modello che stima simultaneamente più variabili dipendenti, invece di una sola). Gli alberi decisionali sono componenti fondamentali per l'algoritmo Random Forest, che è uno degli algoritmi di machine learning disponibili più potenti.

Alberi Decisionali

Gli alberi decisionali sono modelli di apprendimento supervisionato utilizzati sia per problemi di classificazione che di regressione. Essi hanno una struttura ad albero, in cui il nodo radice rappresenta l'inizio della decisione, i nodi interni corrispondono a test su determinate caratteristiche (feature), e le foglie rappresentano l'output finale del modello (classe predetta o valore stimato). In pratica, il modello suddivide ricorsivamente lo spazio dei dati in regioni sempre più omogenee [17].

La scelta della partizione in ciascun nodo avviene sulla base di un criterio di impurità, che misura quanto omogeneo è l'insieme dei dati all'interno del nodo. Nei problemi di classificazione, i criteri più comuni sono:

- Indice di Gini:
$$Gini(S) = 1 - \sum_{i=1}^{C} p_i^2 \tag{3.3}$$
dove p_i è la proporzione degli elementi appartenenti alla classe i nel nodo.
- Entropia (Information Gain):
$$Entropia(S) = -\sum_{i=1}^{C} p_i log_2(p_i) \tag{3.4}$$
che misura il grado di disordine del nodo [18].
-

Nei problemi di regressione, invece, la suddivisione dei dati è scelta in modo da minimizzare l'errore quadratico medio (MSE Mean Squared Error) o la varianza all'interno dei nodi:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y})^2 \tag{3.5}$$

dove y_i sono i valori osservati e $\hat{y}$ il valore medio previsto nel nodo.

La crescita dell'albero prosegue fino a quando vengono soddisfatte determinate condizioni di arresto (ad esempio profondità massima raggiunta, numero minimo di campioni per nodo o guadagno informativo troppo basso). In alternativa, per evitare fenomeni di overfitting, si applicano tecniche di potatura (pruning), che eliminano i rami poco significativi, migliorando la capacità di generalizzazione del modello [18].

Un vantaggio degli alberi decisionali è la loro interpretabilità: il modello fornisce regole semplici del tipo if...then..., facilmente leggibili e comprensibili anche da non esperti. Tuttavia, un limite importante è la loro instabilità: piccole variazioni nei dati possono produrre strutture molto diverse, motivo per cui spesso vengono utilizzati all'interno di metodi ensemble come le Random Forest [19].

Ensemble Learning e Random Forest

L'ensemble learning è un approccio del machine learning che mira a migliorare le prestazioni predittive combinando più modelli detti weak learners (apprenditori deboli), i quali, presi singolarmente, possono avere una capacità limitata. L'idea alla

base è che un insieme di modelli, se sufficientemente diversificati, possa produrre una predizione più accurata e robusta rispetto a ciascun modello individuale. Questo principio si rifà al concetto statistico secondo cui "la saggezza della folla" tende a fornire stime più affidabili rispetto ai singoli individui.

Formalmente, dato un insieme di modelli $h_1, h_2, \ldots, h_M$ addestrati sullo stesso problema, la predizione di un ensemble può essere ottenuta combinando le singole uscite:

- per la regressione con la media aritmetica:

$$\hat{y}(x) = \frac{1}{M} \sum_{m=1}^{M} h_m(x) \tag{3.6}$$

- per la classificazione con la regola della maggioranza (o media delle probabilità, nel caso di classificatori probabilistici):

$$\hat{y}(x) = arg \overset{\max}{c} \sum_{m=1}^{M} 1(h_m(x) = c) \tag{3.7}$$

  dove $1(\cdot)$ è la funzione indicatrice

I vantaggi principali dell'ensemble learning sono la riduzione della varianza (grazie alla media delle predizioni), la riduzione del bias (in alcuni metodi come boosting), e una maggiore robustezza agli outlier e al rumore nei dati. Tuttavia, il costo computazionale può aumentare sensibilmente, e l'interpretabilità tende a ridursi rispetto a modelli più semplici.

- Bagging (Bootstrap Aggregating), che riduce la varianza addestrando modelli su sottoinsiemi bootstrap del dataset;
- Pasting, simile al bagging ma senza reinserimento dei campioni;
- Boosting, che combina modelli in sequenza, dando più peso agli esempi mal classificati;
- Stacking, che sfrutta un modello di livello superiore (meta-learner) per combinare le predizioni di diversi modelli base.

Un'applicazione particolarmente efficace del bagging è rappresentata dalle Random Forest [19], un metodo ensemble in cui il modello di base è costituito da alberi decisionali. L'idea consiste nell'addestrare numerosi alberi su differenti campioni bootstrap del dataset, combinando poi le loro predizioni. Inoltre, le Random Forest introducono un'ulteriore fonte di casualità: ad ogni nodo, invece di considerare tutte

le feature disponibili per la scelta dello split, viene selezionato casualmente solo un sottoinsieme. Questo meccanismo aumenta la diversità tra gli alberi, riducendo la correlazione tra i modelli e migliorando così la robustezza complessiva.

I principali vantaggi delle Random Forest sono la buona capacità di generalizzazione, la resistenza all'overfitting, e la possibilità di stimare l'importanza delle variabili (feature importance). Tuttavia, l'aumento del numero di alberi incrementa il costo computazionale e riduce l'interpretabilità rispetto a un singolo albero decisionale.

3.5 Tecniche di Apprendimento non supervisionato

Nel capitolo precedente, abbiamo fornito una panoramica dei principali paradigmi dell'apprendimento automatico, distinguendo tra i modelli supervisionati, non supervisionati e a rinforzo in base alla natura dei dati di addestramento e alla modalità di apprendimento. In particolare, è emerso che l'apprendimento non supervisionato si basa sulla capacità di un algoritmo di scoprire la struttura intrinseca in un insieme di dati non etichettato, senza la guida di un "insegnante" esterno.

Per comprendere appieno l'importanza e la versatilità di questo approccio, il presente capitolo si propone di approfondire le principali tecniche e i metodi che ne costituiscono il fondamento. Esamineremo come questi algoritmi siano in grado di rivelare pattern e relazioni nascoste, rendendoli strumenti essenziali per l'analisi esplorativa dei dati, la segmentazione e la riduzione della dimensionalità. L'obiettivo è analizzare le metodologie più rilevanti, illustrandone il funzionamento, le applicazioni pratiche e i rispettivi punti di forza e debolezza.

3.5.1 Clustering

Tra le tecniche più diffuse in questo ambito, il *clustering* ricopre un ruolo fondamentale: mira a raggruppare istanze simili in insiemi (cluster) senza necessità di etichette, risultando ideale per l'analisi esplorativa e per mettere in luce pattern altrimenti non evidenti [20].

Il clustering trova applicazione in una varietà di domini, tra cui:

- Customer Segmentation:Raggruppare i clienti in base a caratteristiche o comportamenti comuni. Questa tecnica è cruciale per le strategie di marketing mirato e per lo sviluppo di sistemi di raccomandazione, come quelli che suggeriscono contenuti che sono piaciuti a utenti nello stesso gruppo.

- Data analysis: Il clustering consente di suddividere un grande dataset in sottoinsiemi più piccoli e gestibili. Analizzando ogni cluster individualmente, è possibile identificare le caratteristiche e i pattern specifici di quel gruppo, migliorando la comprensione generale del dataset.
- Rappresentazione compatta dei dati: Utilizzando i centroidi dei cluster come rappresentanti del gruppo, è possibile ottenere una rappresentazione sintetica e compatta dei dati. Questa tecnica è particolarmente utile per la visualizzazione di dataset ad alta dimensionalità.
- Segmentazione delle immagini: Il clustering può essere usato per raggruppare i pixel di un'immagine in base al colore, alla texture o alla posizione. Questo processo semplifica l'immagine, rendendola più facile da analizzare per altri algoritmi di visione artificiale, come quelli per il riconoscimento di oggetti.

3.5.2 Anomaly detection

Mentre il clustering si concentra sull'identificazione di gruppi di dati simili, il rilevamento delle anomalie si occupa di un compito correlato ma distinto: identificare i punti dati che deviano in modo significativo dalla norma. Queste istanze, note come anomalie o outlier, sono rare e sospette perché non si conformano al comportamento atteso della maggior parte del dataset. A differenza del clustering, che può essere visto come un modo per trovare la "normalità" dei dati, il rilevamento delle anomalie si concentra esplicitamente sulla ricerca dell'eccezione.

- Malfunzionamenti o guasti in sistemi industriali
- Frodi in transazioni bancarie
- Attività di rete dannose in ambito di cybersecurity

Poiché le anomalie sono eventi rari e imprevedibili, il rilevamento di esse è intrinsecamente un problema di apprendimento non supervisionato, in quanto i dati di addestramento non contengono etichette che indichino quali istanze sono anomale. Sebbene alcune tecniche di clustering possano essere impiegate per questo scopo (identificando i punti che non appartengono a nessun cluster), esistono metodi specifici progettati per massimizzare la precisione e l'efficacia nel rilevamento di outlier.

Capitolo 4

Reti Neurali e Deep Learning

4.1 Introduzione alle Reti Neurali

L'uomo ha preso ispirazione molteplici volte da eventi naturali per grandi invenzioni che hanno cambiato il corso della storia. Gli uccelli hanno ispirato gli aerei, la bardana il velcro. Per costruire quindi una macchina intelligente si è guardato al cervello umano e così facendo si è giunti alle reti neurali artificiali (Artificial Neural Network - ANN) [13, p. 279].

Le Reti Neurali Artificiali (ANN) sono state introdotte per la prima volta nel 1943 dal neurofisiologo Warren McCulloch e dal matematico Walter Pitts. Nel loro storico paper [21] "A Logical Calculus of Ideas Immanent in Nervous Activity", presentarono un modello computazionale semplificato di come i neuroni biologici possano lavorare insieme nel cervello per svolgere computazioni complesse usando la logica proposizionale. Questo modello è considerato la prima ANN.

Il successo iniziale delle ANN portò alla convinzione che si sarebbe presto comunicato con una vera macchina intelligente. Tuttavia, questa convinzione si rivelò prematura e le ANN vennero in parte dimenticate. Nel frattempo, vennero inventati altri algoritmi di machine learning, come le SVM (Support Vector Machine) negli anni '90, che sembravano offrire risultati migliori in termini di precisione e performance.

Negli ultimi decenni, le ANN hanno riacquistato un ruolo centrale e stanno assumendo un'importanza crescente. Questo "secondo rinascimento" è stato reso possibile da tre fattori cruciali:

- L'aumento della potenza di calcolo: La disponibilità di GPU (Graphics Processing Unit) ha permesso di addestrare reti neurali con miliardi di parametri in tempi ragionevoli.
- La disponibilità di grandi dataset: L'esplosione dei "big data" ha fornito il carburante necessario per addestrare modelli complessi senza il rischio di overfitting.
- Lo sviluppo di nuovi algoritmi: L'introduzione di tecniche come il dropout e le funzioni di attivazione ReLU ha risolto problemi chiave come il vanishing gradient, rendendo l'addestramento di reti molto profonde (Deep Learning) una pratica efficace.

Questi sviluppi hanno permesso alle reti neurali di superare i limiti passati e di diventare l'architettura dominante in settori come la visione artificiale e l'elaborazione del linguaggio naturale, rivoluzionando di fatto l'intero campo dell'intelligenza artificiale.

4.2 Dai neuroni al Perceptron

4.2.1 Il neurone Biologico

Prima di discutere delle ANN e dei neuroni artificiali è consigliato dare un'occhiata alla struttura di un neurone biologico.

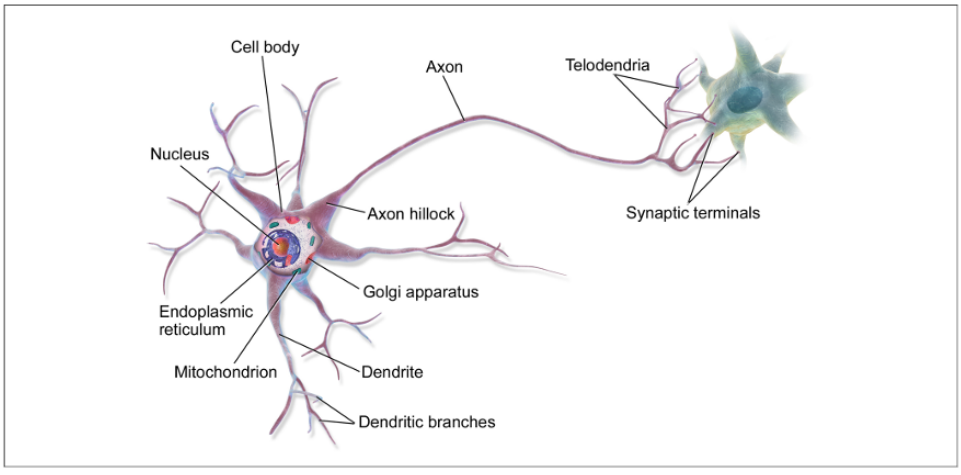

Figura 4.1: Neurone Biologico [13, p. 282]

Il neurone è la cellula fondamentale del sistema nervoso, un'unità specializzata nella ricezione, elaborazione e trasmissione di segnali elettrici e chimici. Esso è

composto da un corpo cellulare (o soma) che contiene il nucleo e la maggior parte degli organuli cellulari. Dal soma si estendono numerose e ramificate diramazioni, i dendriti, che agiscono come "antenne" per ricevere i segnali in ingresso da altri neuroni. Un'unica, e spesso molto lunga, diramazione chiamata assone si diparte dal corpo cellulare per trasmettere i segnali in uscita. All'estremità dell'assone, una serie di ramificazioni chiamate telodendria terminano in piccole strutture a bulbo, le sinapsi. Le sinapsi rappresentano i punti di contatto tra l'assone di un neurone e i dendriti o il corpo cellulare di un altro neurone. È qui che avviene la trasmissione del segnale, tipicamente attraverso la liberazione di neurotrasmettitori (segnali chimici) in uno spazio microscopico. Quando la somma dei segnali in entrata raggiunge una soglia di attivazione, il neurone "spara" un segnale elettrico (potenziale d'azione) lungo l'assone, che a sua volta stimola altri neuroni.

4.2.2 Logica computazionale con Neuroni

McCullock e Pitts [21] proposero un semplice modello di neurone biologico, il quale prese in seguito il nome di neurone artificiale. Ha uno o più input binari (on/off) e un output binario. Il neurone artificiale attiva il suo canale di output quando i giusti canali di input vengono attivati. Questo semplice modello permette di costruire semplici ANN che replicano diverse computazioni logiche.

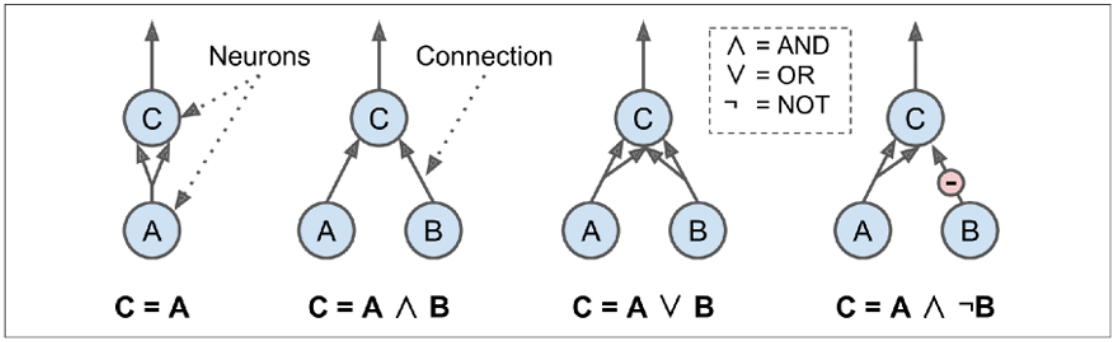

Figura 4.2: computazioni logiche semplici con ANN [13, p. 283]

Le varie combinazioni di ANN possono essere combinate per eseguire espressioni logiche molto più complesse.

4.2.3 Il Perceptron

Il Perceptron è una delle più semplici architetture ANN, inventata nel 1957 da Frank Rosenblatt [22]. Questa ANN si basa sulla Threshold Logic Unit (TLU), un neurone artificiale leggermente diverso rispetto a quello visto in precedenza. L'input e l'output di questa architettura sono numeri e ogni input è associato ad

un peso. Il TLU calcola una somma ponderata tra i vari input, dopodiché applica una funzione a gradino (step function) alla somma elaborando così il risultato.

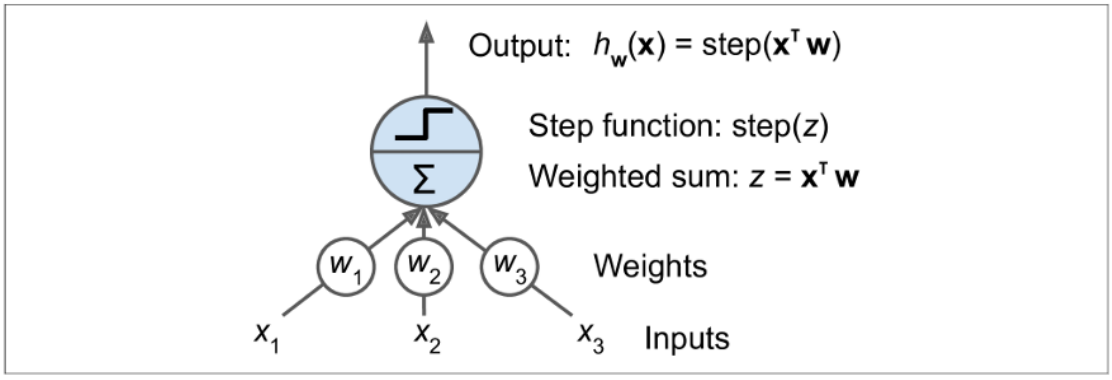

Figura 4.3: TLU; un neurone artificiale che calcola la somma ponderata tra input e poi applica una step function [13, p. 284]

Nel momento in cui calcoliamo la media ponderata dobbiamo tenere conto dei pesi associati a ogni input:

$$Somma\ Ponderata : z = w_1 x_1 + w_2 x_2 + ... + w_n x_n = x^T w \tag{4.1}$$

A questa somma applichiamo poi la step function. la più comune step function è la Heaviside Step Function:

$$step(z) = \begin{cases} 0 & \text{se } z < 0 \\ 1 & \text{se } z \geq 0 \end{cases} \tag{4.2}$$

Un Perceptron è composto da uno strato di TLU con ogni TLU collegato a ogni input. Quando tutti i neuroni in uno strato sono collegati a ogni neurone nello strato precedente si dice che è uno strato completamente connesso (o strato denso). Gli input del perceptron sono dati in pasto a degli speciali neuroni chiamati *neuroni input*, che non fanno nient'altro che passare i dati al primo strato. Questi formano lo *strato input*. In genere si aggiunge un elemento di bias chiamato *neurone bias* o *nodo bias*. Risulta possibile calcolare l'output di uno strato pienamente connesso tramite la formula:

$$h_{W,b}(X) = \phi(XW + b) \tag{4.3}$$

Dove X rappresenta la matrice di input, W la matrice contenente i *pesi* delle connessioni, b è il vettore di bias (neuroni bias) e la funzione ϕ è la *funzione di attivazione*.
Si può vedere un esempio di Perceptron nell'immagine.

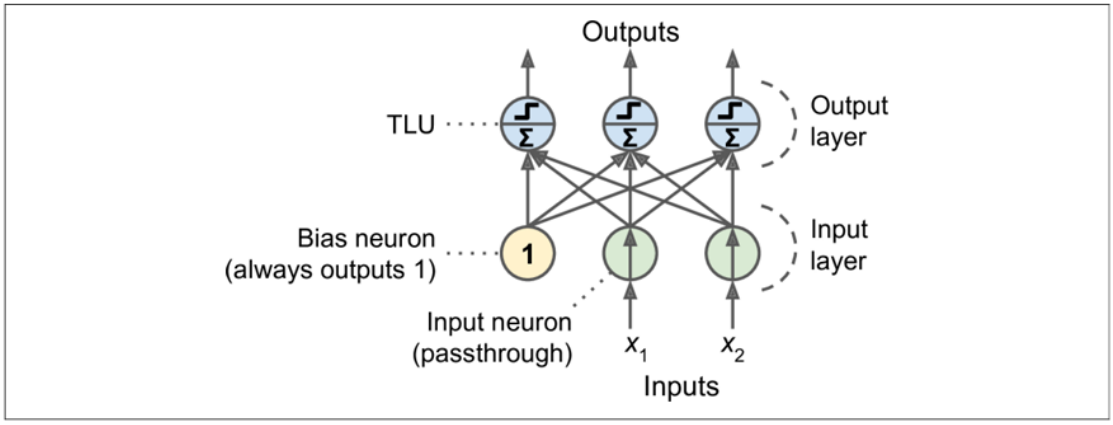

Figura 4.4: esempio di Perceptron con due neuroni di input, un neurone bias e tre neuroni output [13, p. 286]

Il ruolo storico del Perceptron è di fondamentale importanza: è stato il primo modello di neurone artificiale in grado di apprendere in modo autonomo da un set di dati. Questo ha segnato l'inizio della storia del deep learning, dimostrando che un algoritmo poteva adattare i propri pesi per fare previsioni, senza essere esplicitamente programmato.

Nonostante i suoi limiti, il Perceptron ha gettato le basi per lo sviluppo di reti neurali più complesse, diventando il "mattoncino" elementare da cui sono poi evolute le architetture moderne.

4.2.4 Il Perceptron Multilayer e la Backpropagation

Il Perceptron, pur essendo un'innovazione storica, è limitato ai soli problemi linearmente separabili. Classici controesempi (es. XOR) mostrano che non può apprendere confini non lineari [23]. Questa debolezza ha motivato lo sviluppo di architetture più espressive.

Il Multilayer Perceptron (MLP) rappresenta l'evoluzione naturale del Perceptron ed è alla base del deep learning: una rete feed-forward con strato di input, uno o più strati nascosti e uno di output, con connessioni dense tra strati successivi [24, cap. 6]. In virtù degli strati nascosti non lineari, gli MLP possono approssimare funzioni non lineari generali (teorema di approssimazione universale) [25].

Quando una ANN contiene un insieme di strati nascosti viene chiamata Rete Neurale Profonda (Deep Neural Network - DNN).

Per molti anni, uno dei maggiori ostacoli all'uso delle reti neurali multi-strato (MLP) fu la difficoltà nel loro addestramento. Questa sfida fu superata solo con l'introduzione dell'algoritmo di backpropagation (retropropagazione dell'errore) da parte di David Rumelhart, Geoffrey Hinton e Ronald Williams nel loro influente

paper [26] del 1986. Questo algoritmo ha fornito il metodo efficiente necessario per aggiornare i pesi in reti neurali profonde, sbloccando di fatto il potenziale del deep learning.

La retropropagazione dell'errore (backpropagation) è l'algoritmo che rende l'addestramento delle reti neurali efficiente, permettendo al Gradient Descent di calcolare il gradiente in modo automatico. In sole due passate attraverso la rete, l'algoritmo è in grado di calcolare il gradiente della funzione di costo in relazione a ogni singolo peso (parametro) del modello.

Il processo si svolge in due passaggi principali per ogni iterazione di addestramento:

- Forward Pass (Passaggio in Avanti): In questa fase, un'istanza di addestramento viene alimentata alla rete, attraversando ogni strato, da quello di input a quello di output. La rete calcola una previsione, che viene poi confrontata con il valore reale per misurare l'errore del modello utilizzando la funzione di costo.

- Backward Pass (Passaggio all'Indietro): È qui che entra in gioco la retropropagazione. L'algoritmo attraversa la rete a ritroso, partendo dall'errore calcolato nel passaggio in avanti. A ogni strato, calcola il contributo di ogni singola connessione (peso) all'errore totale.

In sintesi, la retropropagazione non è un algoritmo di ottimizzazione, ma un metodo efficiente per calcolare il gradiente di ogni parametro rispetto alla funzione di costo. Una volta che il gradiente è stato calcolato per tutti i parametri, l'algoritmo di Gradient Descent modifica i pesi nella direzione che riduce l'errore, preparandosi per la successiva iterazione. Questo ciclo si ripete per migliaia o milioni di iterazioni, fino a quando il modello non converge.

Per far sì che l'addestramento tramite Gradient Descent potesse funzionare correttamente, è stato necessario apportare una modifica fondamentale alla struttura dei MLP. La funzione a gradino (o step function), utilizzata nel Perceptron, è discontinuamente non derivabile in un punto e ha una derivata nulla ovunque, rendendo impossibile il calcolo del gradiente. Questo impedisce all'algoritmo di retropropagazione (backpropagation) di misurare la "pendenza" dell'errore per ogni peso e, di conseguenza, di aggiornare i pesi del modello. La backpropagation, infatti, si basa sulla regola della catena del calcolo differenziale, che richiede che le funzioni di attivazione siano continue e derivabili. Per superare i limiti della funzione a gradino, l'apprendimento automatico moderno ha adottato funzioni di attivazione non lineari, continue e derivabili. Le più utilizzate in assoluto sono la funzione sigmoide, la funzione tangente iperbolica e la funzione ReLU.

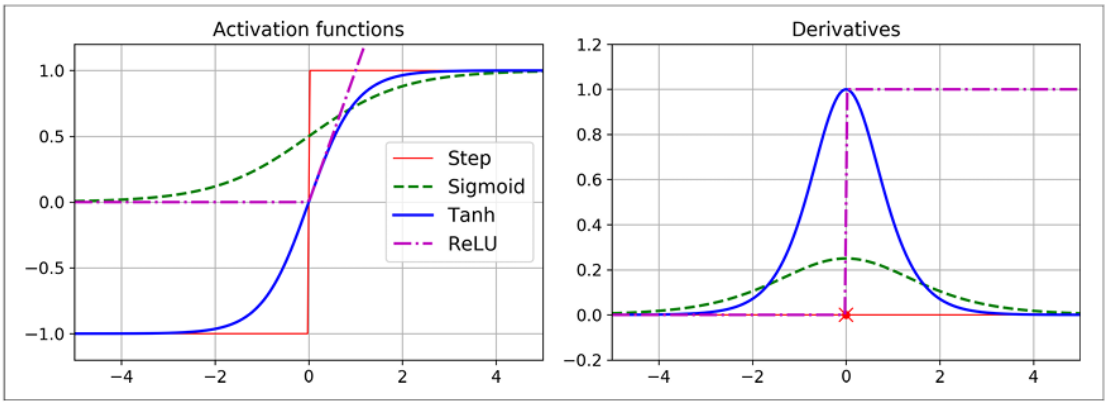

Figura 4.5: Funzioni di attivazione e le loro derivate [13, p. 292]

Funzione Sigmoide

È una funzione a forma di S che "schiaccia" qualsiasi valore in ingresso in un intervallo tra 0 e 1. Era molto popolare nelle reti neurali classiche, ma tende a soffrire del problema del vanishing gradient per valori di input molto grandi o molto piccoli.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{4.4}$$

Funzione Tangente Iperbolica

Simile alla Sigmoide, anche la tangente iperbolica ha una forma a S, ma mappa gli input in un intervallo che va da -1 a 1. Essendo centrata su 0, tende a risolvere in parte il problema del vanishing gradient della sigmoide.

$$tanh(z) = 2\sigma(2z) - 1 \tag{4.5}$$

Rectified Linear Unit (ReLU)

La ReLU è la funzione di attivazione più utilizzata nel deep learning. La sua formula è molto semplice: se l'input è positivo, restituisce l'input stesso; se l'input è negativo o nullo, restituisce 0. La sua popolarità deriva dal fatto che è computazionalmente efficiente e risolve in modo efficace il problema del vanishing gradient, anche se presenta il problema del "dying ReLU" (quando i neuroni rimangono inattivi).

$$ReLU(z) = max(0, z) \tag{4.6}$$

4.2.5 Ottimizzazione degli Iperparametri

Nel machine learning, non tutti i parametri di un modello vengono appresi in modo autonomo dall'algoritmo. Esiste una categoria fondamentale di valori che deve essere impostata manualmente dallo sviluppatore prima che l'addestramento abbia inizio. Questi sono gli iperparametri, che agiscono come le "manopole" di controllo del modello. A differenza dei pesi e dei bias, che vengono ottimizzati dal processo di Gradient Descent, gli iperparametri definiscono la struttura, le regole e il modo in cui il modello apprende, influenzando in modo determinante le sue prestazioni e la sua capacità di generalizzare su dati non visti [27].

Numero di strati nascosto e neuroni

Il numero di strati nascosti e di neuroni in ciascun strato è una delle decisioni architettoniche più critiche e rappresenta un iperparametro fondamentale che determina la capacità del modello di apprendere. In linea generale, più strati e neuroni ha una rete, più complessi sono i pattern che può riconoscere. Tuttavia, è necessario trovare un delicato equilibrio:

- Underfitting: Se l'architettura della rete è troppo semplice o i livelli di astrazione sono insufficienti, il modello non avrà la capacità necessaria per catturare le relazioni e le regolarità dei dati. In questo scenario, il modello non riesce a imparare né i dati di addestramento né quelli di test, un fenomeno noto come underfitting. È come cercare di spiegare un argomento complesso con un vocabolario limitato: si riescono a cogliere solo le informazioni più superficiali.
- Overfitting: Al contrario, se l'architettura è troppo complessa, la rete potrebbe non solo imparare i pattern rilevanti, ma anche i rumori e le specificità del solo dataset di addestramento. Questo porta a una memorizzazione dei dati piuttosto che a un vero e proprio apprendimento, un fenomeno noto come overfitting. Un modello che soffre di overfitting si comporta in modo eccellente sui dati di addestramento ma fallisce miseramente quando incontra dati nuovi e non visti. È come uno studente che impara a memoria le risposte di un solo test, ma non capisce la materia e fallisce all'esame finale.

L'obiettivo è quindi trovare una rete con una complessità "just right", che sia abbastanza potente da apprendere le relazioni sottostanti nei dati, ma non così complessa da memorizzare ogni singola variazione casuale, garantendo così una buona generalizzazione.

Tasso di Apprendimento

Il tasso di apprendimento (learning rate) è uno degli iperparametri più cruciali e, forse, il più influente sulla performance di un modello. Il suo scopo è quello di

determinare la dimensione del passo che l'algoritmo di Gradient Descent compie in ogni iterazione per scendere lungo il "paesaggio" della funzione di costo e raggiungere il punto di minimo. Si può paragonare il tasso di apprendimento alla grandezza dei passi che una persona compie per scendere da una montagna:

- Un tasso di apprendimento troppo elevato è come fare passi enormi. Si rischia di "saltare" completamente il punto di minimo, non riuscendo mai a convergere o, peggio, finendo per divergere e rendere l'addestramento instabile. Il modello non trova il punto di ottimo, ma rimbalza intorno ad esso senza mai stabilizzarsi. Ad esempio, con un tasso di apprendimento di 10, la funzione di costo non solo non diminuisce, ma può aumentare a ogni iterazione, segno che il modello sta costantemente superando il punto di minimo.

- Al contrario, un tasso di apprendimento troppo basso è come muoversi con passi minuscoli. Sebbene si sia certi di non superare il minimo, l'addestramento diventa estremamente lento e inefficiente, richiedendo un numero di iterazioni molto elevato per raggiungere un risultato soddisfacente. Il modello impiega una quantità di tempo eccessiva per convergere. Ad esempio, un tasso di apprendimento di 10^{-5} farà convergere il modello in modo sicuro, ma richiederà un tempo di addestramento sproporzionato per raggiungere una buona performance.

La scelta del tasso di apprendimento ideale è quindi un compromesso tra la velocità di addestramento e la garanzia di convergenza. Un valore ben calibrato assicura che il modello impari in modo efficiente, raggiungendo una soluzione ottimale in un tempo ragionevole.

Numero di Epoche

Il numero di epoche (epochs) è un iperparametro che definisce quante volte l'intero dataset di addestramento viene utilizzato per aggiornare i pesi del modello. Ogni epoca rappresenta un ciclo completo in cui l'algoritmo elabora tutti i campioni del dataset, aggiornando i pesi a ogni iterazione (o per ogni mini-batch).

La scelta del numero di epoche è un equilibrio tra l'addestramento completo del modello e l'evitamento dell'overfitting.

- Poche Epoche: Se il numero di epoche è troppo basso, il modello potrebbe non avere il tempo sufficiente per imparare i pattern sottostanti nei dati. Questo porta a un caso di underfitting, dove la performance del modello rimane insoddisfacente sia sui dati di addestramento che su quelli di test.

- Troppe Epoche: Al contrario, addestrare il modello per un numero eccessivo di epoche può portare all'overfitting. In questo scenario, il modello inizia a

memorizzare i dati di addestramento e a imparare il rumore e le peculiarità del dataset specifico, compromettendo la sua capacità di generalizzare su dati nuovi.

Una tecnica comune per determinare il numero ideale di epoche è l'early stopping. Questa strategia interrompe l'addestramento non appena la performance del modello sui dati di validazione smette di migliorare, trovando il punto di equilibrio ottimale tra apprendimento e generalizzazione.

4.3 Reti Neurali Convoluzionali

Nonostante i notevoli successi del machine learning in vari ambiti, per molto tempo le macchine hanno faticato a eseguire compiti che per gli esseri umani risultano banali, come riconoscere oggetti in un'immagine o distinguere parole in una traccia audio. Questi compiti, apparentemente semplici, richiedono la capacità di analizzare, filtrare e interpretare un'enorme quantità di dati percettivi.

Dallo studio di come funziona il sistema visivo dei mammiferi [28], in particolare la corteccia cerebrale visiva, è nata l'ispirazione per le reti neurali convoluzionali. Questi studi hanno dimostrato che il cervello umano elabora le immagini attraverso una gerarchia di neuroni specializzati: alcuni rispondono a forme semplici e linee, mentre altri, più in profondità, combinano queste informazioni per riconoscere pattern sempre più complessi, come oggetti interi. Le CNN sono state progettate per emulare questo processo.
In questo capitolo approfondiremo le CNN, la loro origine e come sono costruite. Parleremo poi delle migliori architetture CNN e di compiti di visione artificiale come il riconoscimento di oggetti e la segmentazione semantica.

4.3.1 Architettura della Corteccia Visiva

Il lavoro pionieristico di David H. Hubel e Torsten Wiesel negli anni '50 e '60 (che valse loro il Premio Nobel nel 1981) [28] ha fornito la comprensione fondamentale della struttura della corteccia visiva. I loro esperimenti hanno rivelato che i neuroni della corteccia visiva hanno un piccolo campo recettivo locale, reagendo in modo selettivo solo a stimoli visivi che compaiono in una porzione limitata del campo visivo. L'intera immagine viene elaborata dal cervello attraverso la combinazione delle risposte di questi neuroni specializzati.
Proseguendo nella loro ricerca, scoprirono che i neuroni non solo reagivano a porzioni specifiche del campo visivo, ma mostravano anche una selettività per l'orientamento. Ad esempio, alcuni neuroni rispondevano esclusivamente a linee orizzontali, altri a linee verticali, e così via. Notarono inoltre che alcuni neuroni avevano campi recettivi più ampi e reagivano a schemi più complessi, che erano

combinazioni di schemi più semplici. Queste osservazioni portarono a un'intuizione rivoluzionaria: i neuroni di livello superiore si basano sull'output di neuroni di livello inferiore per costruire una rappresentazione gerarchica. È proprio questa architettura a strati che permette al sistema visivo di riconoscere tutti i possibili pattern visivi in qualsiasi area del campo visivo.

Queste scoperte sulla corteccia visiva ispirarono l'introduzione del neocognitron nel 1980, un modello sviluppato da Kunihiko Fukushima [29] che si basava proprio su neuroni organizzati in strati gerarchici con campi recettivi locali. Questa architettura, seppur embrionale, fu il precursore diretto delle reti neurali convoluzionali.

Una delle pietre miliari nella storia delle CNN fu la celebre architettura LeNet-5, introdotta da Yann LeCun nel 1998 [30]. Progettata per il riconoscimento dei numeri scritti a mano sugli assegni, la LeNet-5 si basava su concetti già consolidati, come gli strati pienamente connessi e la funzione di attivazione sigmoide. Tuttavia, introdusse due concetti importantissimi che avrebbero definito le architetture successive: gli strati convoluzionali e gli strati di pooling.

4.3.2 Strati Convoluzionali

Lo strato convoluzionale è il componente fondamentale di una rete neurale convoluzionale. A differenza di uno strato pienamente connesso, i neuroni in uno strato convoluzionale non sono collegati a tutti i pixel dell'immagine di input. Al contrario, ciascun neurone si connette solo a un piccolo gruppo di neuroni nello strato precedente, un'area definita come campo ricettivo.

Questa architettura gerarchica permette alla rete di estrarre e assemblare progressivamente le caratteristiche visive. I neuroni nel primo strato convoluzionale si concentrano su dettagli di basso livello, come bordi, angoli e texture. Nei successivi strati, i neuroni combinano queste caratteristiche di base per riconoscere pattern sempre più complessi, come parti di oggetti, fino a riconoscere l'oggetto intero negli strati più profondi.

Questa struttura gerarchica e localizzata riflette la natura delle immagini nel mondo reale, dove i pattern più complessi sono sempre composti da elementi più semplici. È proprio per questo che le CNN si sono dimostrate estremamente efficaci nei compiti di visione artificiale.

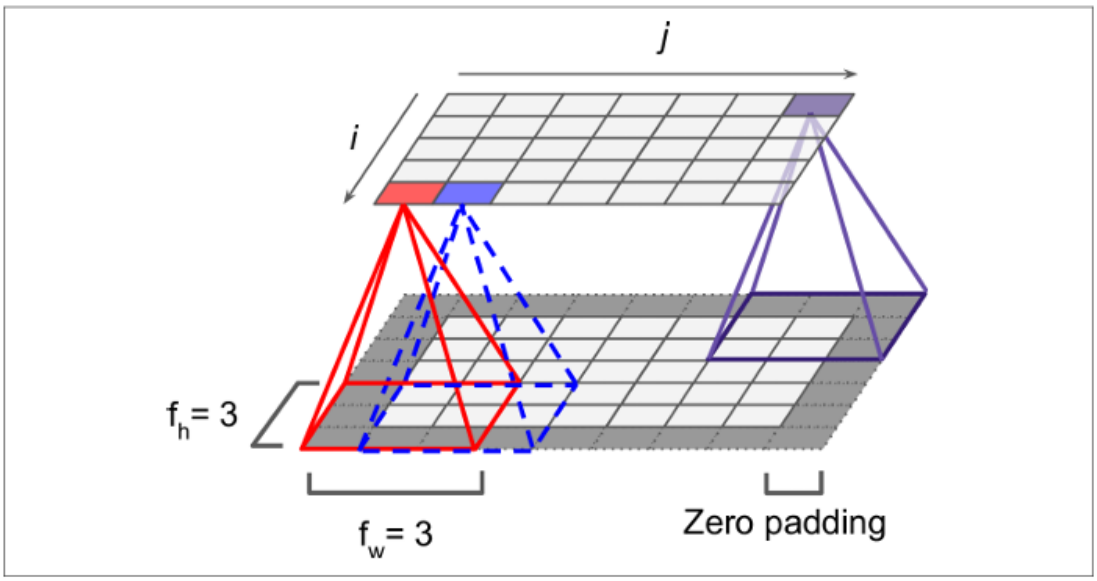

Figura 4.6: Connessione tra strati convoluzionali [13, p. 449]

Un neurone della riga i, colonna j di un particolare strato è connesso all'output di un neurone del precedente strato in posizione riga dalla i alla $i + f_h - 1$, colonna dalla j alla $j + f_w - 1$, dove f_h e f_w sono l'altezza e la larghezza del campo ricettivo. In genere per far si che lo strato abbia la stessa altezza e larghezza dello strato precedente è pratica comune aggiungere degli 0, come mostrato in figura, che prende il nome di *zero padding*.
E' possibile connettere uno strato di input *grande* a uno molto più piccolo, distanziando maggiormente i campi ricettivi. Il distanziamento permette di ridurre drasticamente la complessità computazionale del modello. Lo spostamento viene chiamato *passo* (stride).

Utilizzando il concetto di passo, un neurone in riga i, colonna j, nello strato superiore è connesso all'output di un neurone dello strato precedente in riga $i \times s_h + f_h - 1$, colonna $j \times s_w + f_w - 1$, dove s_h e s_w sono il passo verticale e orizzontale.

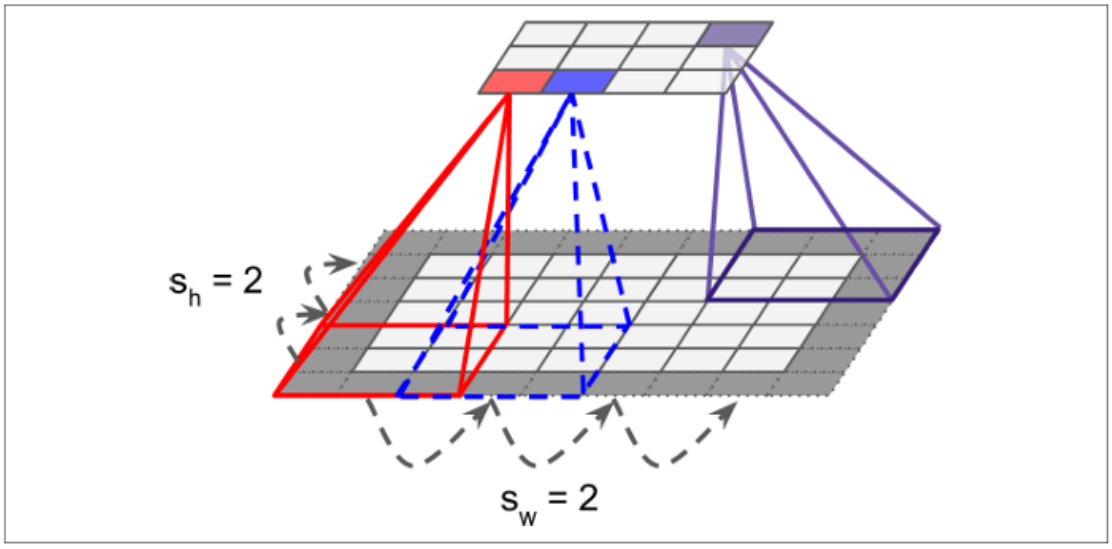

Figura 4.7: Riduzione della dimensionalità usando un passo di 2 [13, p. 450]

4.3.3 Filtri

Il *peso* di un neurone può essere rappresentato come una piccola immagine della dimensione del campo ricettivo.

Nell'immagine x, si possono vedere due insiemi di pesi, chiamati filtri (oppure kernel convoluzionali). Il primo rappresenta un quadrato nero con una linea orizzontale bianca al centro. I neuroni che usano questa immagine come peso ignoreranno qualsiasi input nel loro campo ricettivo eccetto per le informazioni che si trovano in corrispondenza della linea bianca (in nero viene reappresentato con bit = 0, mentre il bianco con bit = 1). La stessa cosa succede con il secondo filtro che usa come peso l'immagine con la striscia bianca orizzontale.

Se applicassimo il filtro verticale a tutti i neuroni e dessimo in pasto al modello l'immagine di input otterremmo come risultato l'immagine in alto a sinistra, in cui le linee verticali vengono messe in risalto mentre il resto invece è sfuocato. Viceversa, applicando il secondo filtro si metteranno in risalto le linee orizzontali. Uno strato di neuroni che usa lo stesso filtro estrapola una *feature map*, che evidenzia le aree di un'immagine in cui il filtro viene maggiormente attivato.

I filtri non devono essere impostati manualmente: durante l'addestramento della CNN gli strati imparano ad utilizzare il filtro migliore per il loro compito, combinandosi con gli strati superiori per ottenere pattern molto più complessi.
L'esempio proposto è una semplificazione della realtà: nella maggior parte dei casi uno strato convoluzionale ha più filtri e estrae una feature map per ogni filtro. In pratica uno strato convoluzionale applica simultaneamente molteplici filtri

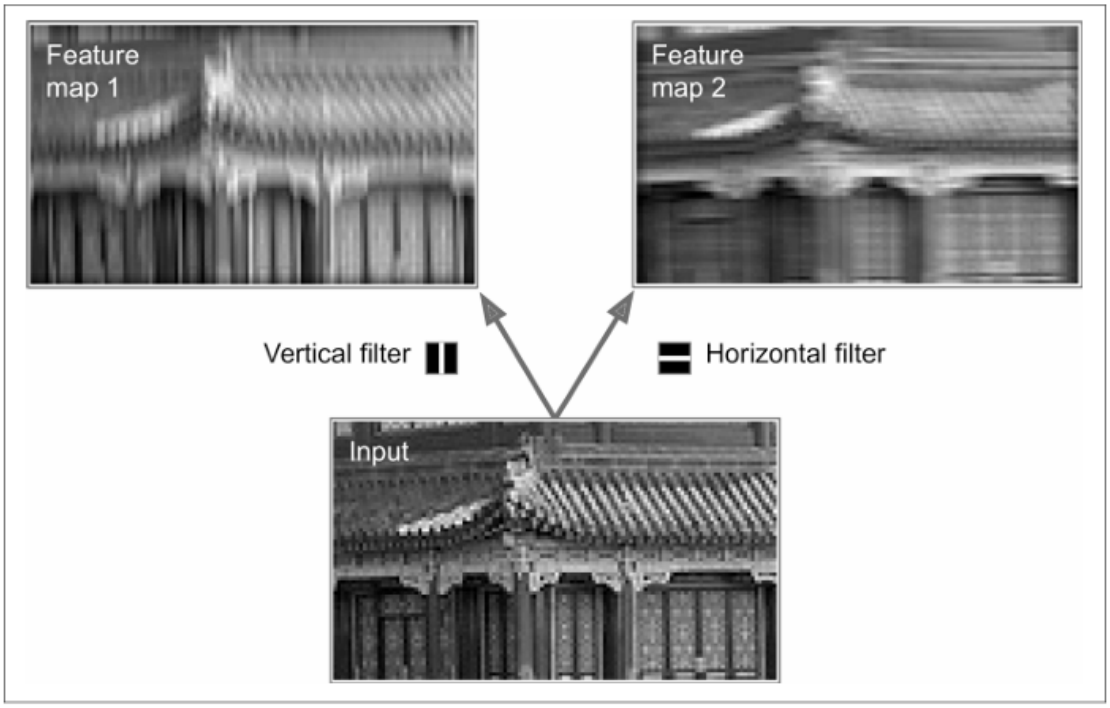

Figura 4.8: L'applicazione di due filtri diversi [13, p. 451]

addestrabili ai suoi input, permettendo di riconoscere molteplici informazioni in tutti i suoi input.

Solitamente le immagini sono composte da molteplici sotto strati: uno per *canale di colore* (tipicamente 3: rosso, verde e blu - RGB). Ciò significa che un neurone della riga j della feature map k, in un dato strato convoluzionale l è connesso all'output dei neuroni nello strato $l-1$, che si trova nelle righe dalla $i \times s_h$ alla $i \times s_h + f_h - 1$ e nelle colonne dalla $j \times s_w$ alla $j \times s_w + f_w - 1$, attraverso tutte le feature map (dello strato $l-1$).

Si crea una situazione alquanto complessa che è riassumibile nella seguente equazione:

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f'_n-1} x_{i',j',k'} \cdot w_{u,v,k',k} \text{ con } \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases} \tag{4.7}$$

In questa equazione:

- $z_{i,j,k}$ è l'output del neurone nella riga i, colonna j nella feature map k dello strato convoluzionale l.
- s_h e s_w sono i passi verticali e orizzontali, f_h e f_w sono l'altezza e la larghezza del campo ricettivo e $f_{n'}$ è il numero di feature map dello strato precedente.

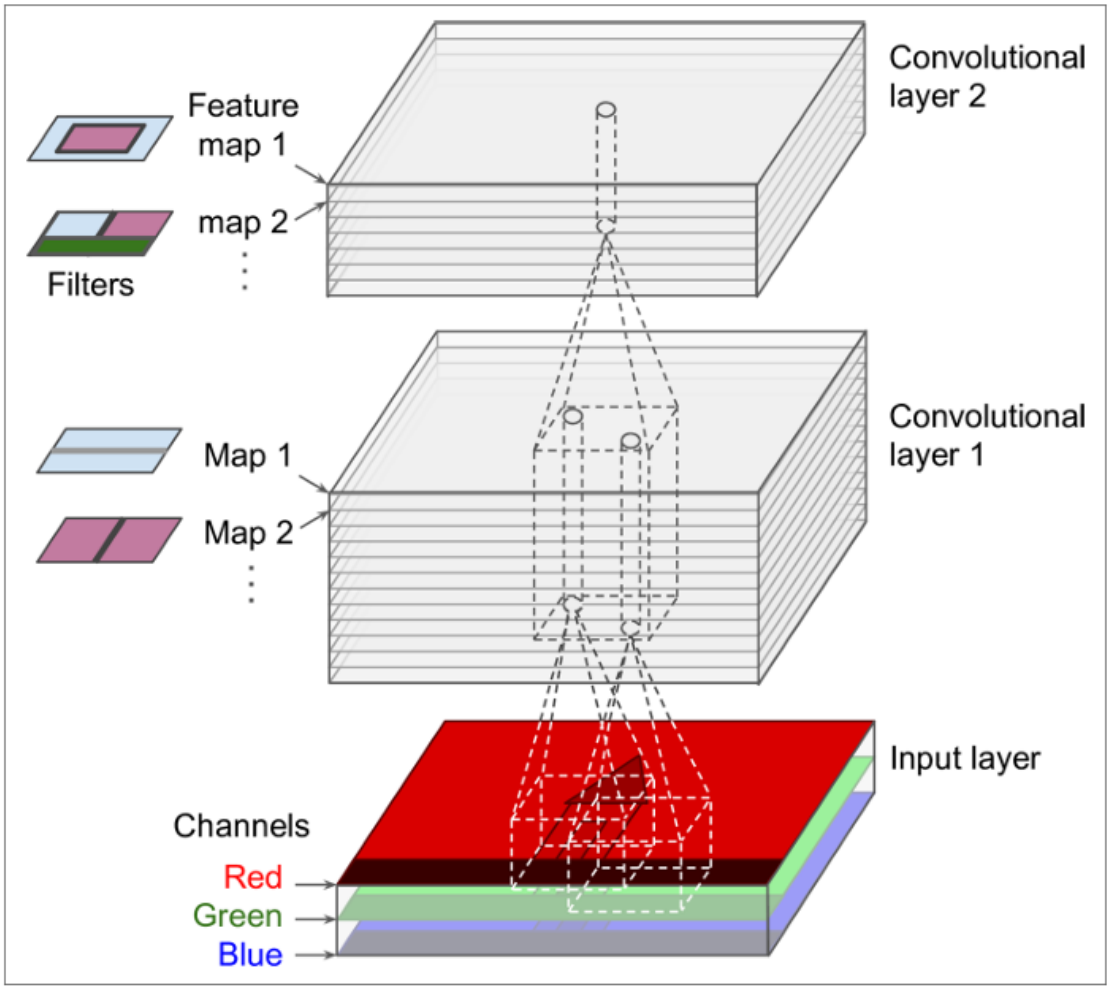

Figura 4.9: strati convoluzionali con multiple feature map, e immagini RGB [13, p. 452]

- $x_{i',j',k'}$ è l'output del neurone localizzato nello strato $l-1$, riga i', colonna j' feature map k'.

- b_k è il termine di bias per la feature map k (nello strato l).

- $w_{u,v,k',k}$ è il peso di connessione tra i neuroni della feature map k dello strato l e i suoi input localizzati nella riga u, colonna v (relativi al campo ricettivo) e feature map k.

4.3.4 Strato di Pooling

Una volta affrontato il concetto di strato convoluzionale, il concetto di pooling è semplice da afferrare. L'obiettivo è sotto-campionare (restringere) l'immagine di input per ridurre il carico computazionale, l'utilizzo di memoria e il numero di parametri (riducendo di fatto l'overfitting). Come per gli strati convoluzionali, ogni neurone è connesso all'output di un numero limitato di neuroni dello strato precedente, ognuno di questi localizzato in campo recettivo rettangolare. Come visto in precedenza bisogna impostare secondo le proprie esigenze la dimensione del campo ricettivo, il *passo*, e il tipo di padding. La differenza sostanziale con gli strati convoluzionali è che uno strato di pooling non ha *peso*. Semplicemente utilizza una funzione di aggregazione, come il massimo o la media aritmetica.

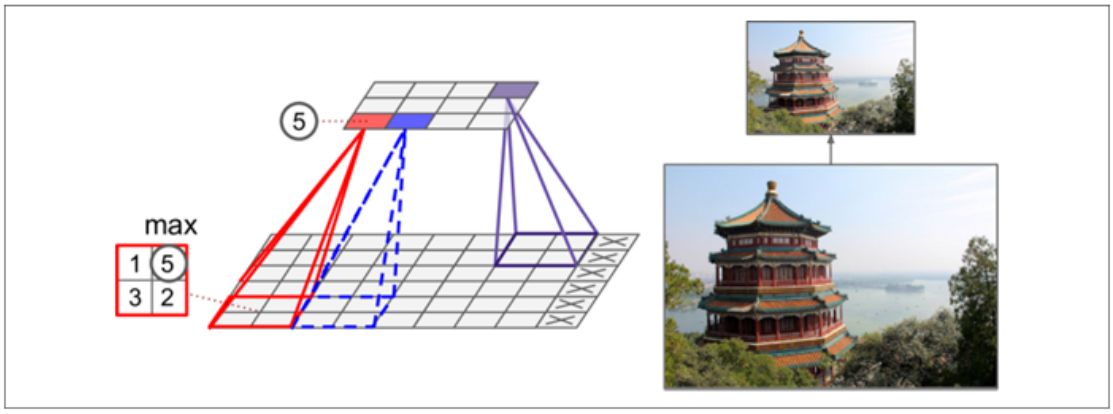

Figura 4.10: Pooling, campo ricettivo 2×2, passo di 2, senza padding [13, p. 457]

Oltre a ridurre il carico computazionale, l'utilizzo di memoria e il numero di parametri, uno strato di max pooling (in cui la funzione di aggregazione è l'estrazione del valore massimo) introduce un livello di invarianza a piccole traslazioni. Il concetto di invarianza è la capacità di un modello di riconoscere un'entità (come un oggetto, una faccia, o un pattern) indipendentemente dalle sue piccole trasformazioni, come lo spostamento o il ridimensionamento ed è una proprietà fondamentale per i compiti di visione artificiale. Abbiamo visto come il pooling estragga il feature più significativo all'interno di un campo ricettivo. Il feature più significativo può trovarsi in uno qualsiasi dei neuroni s cui viene applicata la funzione di aggregazione. Il che significa che anche se il feature più significativo si sposta leggermente all'interno del campo ricettivo, l'output dello strato di pooling rimane lo stesso o subisce una variazione insignificante.

Non sempre l'invarianza è un risultato positivo. Nella segmentazione semantica (che vedremo più avanti) è preferibile l'equi-varianza. In questo caso l'output si trasforma nello stesso modo in cui si trasforma l'input.

4.3.5 Architetture delle CNN

E' possibile considerare la famosa architettura della CNN LeNet-5 come la progenitrice delle CNN. Da quel momento in poi sono nate altre famose architetture le quali naturalmente si differenziano per scopi e funzionalità. Tutte le architetture CNN hanno però caratteristiche comuni.

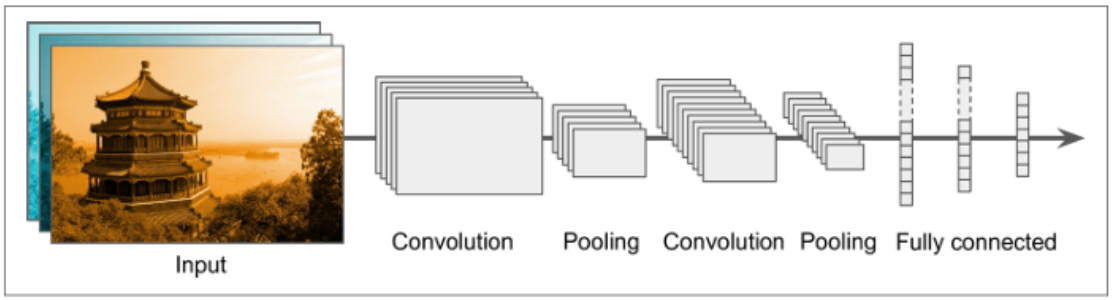

Figura 4.11: Architettura CNN tipica [13, p. 461]

Le CNN tipicamente impilano strati convoluzionali, ognuno di essi in genere seguito da uno strato ReLU (applicazione della funzione ReLU all'output di uno strato precedente). Poi si applica uno strato di pooling, poi altri strati convoluzionali, altri strati di pooling e così di seguito. L'immagine diventa sempre più piccola ma con un numero crescente di feature map, grazie agli strati convoluzionali. In genere alla cima della pila di strati si posiziona una rete neurale feedforward regolare (RNN).

LeNet-5

L'architettura di LeNet-5 è probabilmente la più conosciuta tra tutte le CNN. Creata nel 1998 da Yann LeCun [30] e utilizzata per il riconoscimento delle cifre scritte a mano.

Strato	Tipo	Mappe	Dimensione	dim. Kernel	Passo	funz. Attivazione
Out	Pienamente Connesso	-	10	-	-	RBF
F6	Pienamente Connesso	-	84	-	-	tanh
C5	Convoluzione	120	1×1	5×5	1	tanh
S4	pooling Medio	16	5×5	2×2	2	tanh
C3	Convoluzione	16	10×10	5×5	1	tanh
S2	pooling Medio	6	14×14	2×2	2	tanh
C1	Convoluzione	6	28×28	5×5	1	tanh
In	Input	1	32×32	-	-	-

Tabella 4.1: Architettura Le-Net5

Nel corso degli anni architetture più sofisticate e complesse sono state crate a partire da questa. Una buona misura del progresso nell'ottenere architetture migliori è il sottoporre il modello a sfide come la "ILSVRC ImageNet". In questa competizione il tasso di errore nel riconoscimento dei top 5 è calato dal 26% al 2.3% in soli 6 anni. I top 5 sono le architetture che performano con il tasso di errore più basso. Le più famose architetture a vincere questa competizione sono state AlexNet(2012) che verrà brevemente descritta, GoogLeNet (2014) e ResNet(2015).

AlexNet

L'architettura AlexNet vinse la ILSVRC ImageNet challenge con ampio margine sui rivali(1° posto con tasso di errore del 17%, 2° posto con tasso di errore del 26%) [31]. L'innovazione significativa di questa architettura fu l'utilizzo di due tecniche di regolarizzazione, in particolare quella del "data augmentation".

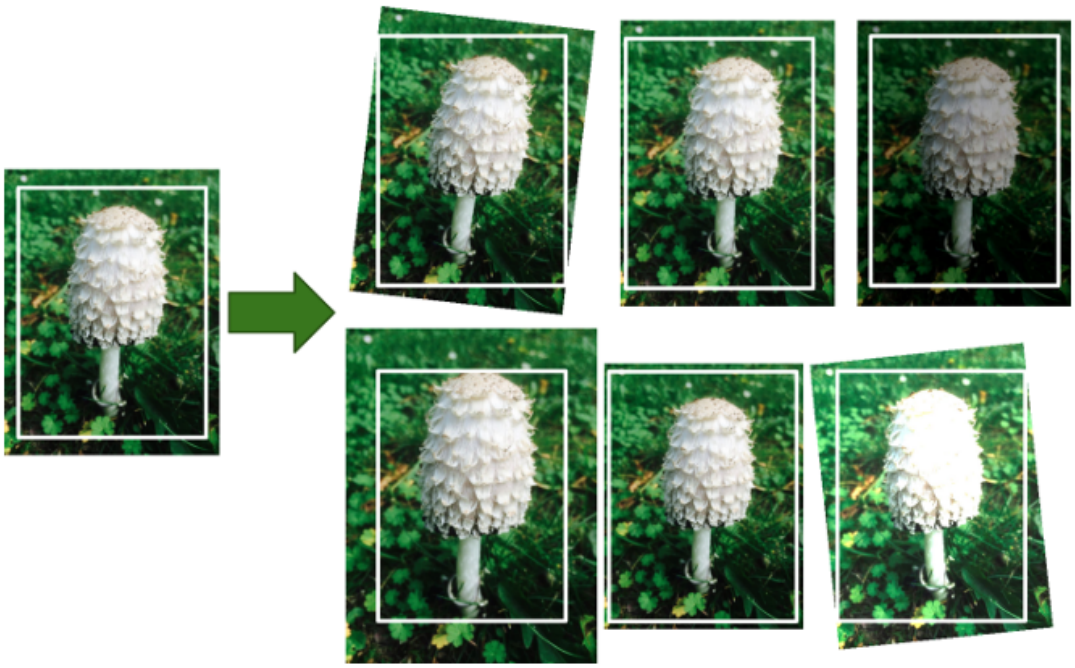

Figura 4.12: nuove istanze tramite Data Augmentation [13, p. 465]

Il "data augmentation" (lett. aumento dei dati) incrementa artificialmente la dimensione dell'insieme di addestramento, sorpassando una delle difficoltà dell'addestramento di algoritmi di machine learning. Produce varianti realistiche dell'insieme di partenza, riducendo l'overfitting. Per esempio, un'immagine viene leggermente spostata, ruotata e ridimensionata e aggiunta all'insieme di addestramento originale.

Questo processo forza il sistema a essere molto più tollerante. Stesso discorso con la luminosità e il contrasto delle immagini.

4.3.6 Rilevamento degli Oggetti

Il compito di classificare e localizzare diversi oggetti in un'immagine viene chiamato "Rilevamento degli Oggetti" (Object Detection). Fino a qualche anno fa il problema veniva risolto utilizzando una CNN che era stata precedentemente addestrata a classificare e localizzare un singolo oggetto, e farle controllare una porzione di immagine per volta. Questa tecnica è molto diretta nell'affrontare il problema ma ha alcune pecche; la più importante è che un oggetto verrà riconosciuto più e più volte. E' necessario un lavoro di post-process che pero' è dispendioso in termini pratici e di fatto molto. Le azioni da compiere sono:

1. Aggiungere all'output della CNN un "grado di somiglianza ad oggetto", ovverola probbailità che un'area specifica contenga effettivamente un oggetto da individuare. Dopodiché, impostata una soglia , si eliminano tutte le bounding box in cui il grado di somiglianza è inferiore.

2. Trovare tutte le bounding box con il grado di somiglianza più alto ed eliminare quelle bounding box che si sovrappongono maggiormente. Per stabilire il grado di sovrapposizione si una metrica chiamata IoU (Intersection over Union), $IoU = \frac{\text{Area di Intersezione}}{\text{Area di Unione}}$.

3. Ripetere i punti 1 e 2 fino a che non ci siano più bounding box da eliminare.

Fortunatamente c'è una soluzione molto più conveniente rispetto a quella proposta, l'utilizzo di una Rete Completamente Convoluzionale (Fully Convolutional Network - FCN). L'idea di una FCN venne introdotta per la prima volta nel 2015 da Jonathan Long et al. e venne proposta di fatto per la segmentazione semantica [32]. Gli autori proposero una sostituzione dei primi strati densi alla cima di una CNN con strati convoluzionali. Porto un esempio per capire meglio la situazione: supponiamo che uno strato denso con 200 neuroni si trovi sopra uno strato convoluzionale che produce 100 feature map, ciascuna di dimensione 7×7 provenienti dallo strato convoluzionale. Se sostituiamo lo strato denso con uno strato convoluzionale utilizzando 200 filtri, ciascuno di dimensione 7×7 e con un padding "valido", lo strato produrrà 200 feature map, ciascuna di 1×1. In altre parole produrrà 200 numeri come ha fatto lo strato denso. La differenza sta nel fatto che l'output dello strato denso era un tensore di dimensione (*dim. batch*,200) mentre l'output dello strato convoluzionale sarà un tensore di forma (*dim. batch*,1,1,200). Sembra una differenza poco importante, ma non lo è. Uno strato denso si aspetta una dimensione di input specifica, mentre uno strato convoluzionale elaborerà immagini di qualsiasi dimensione. Dal momento che le FCN hanno solo strati convoluzionali, la

conclusione più importante è che può essere addestrata e poi utilizzata su immagini di qualsiasi dimensione.

Per esempio, immaginiamo di avere una CNN addestrata a classificare e localizzare fiori in immagini di 224x224 pixel. Questa rete produce un'output di 10 numeri per ogni immagine, dove 5 sono per le probabilità di classe (es. "rosa", "girasole"), 1 per lo score di oggettività e 4 per le coordinate del bounding box. Se convertiamo gli strati densi di questa CNN in strati convoluzionali, possiamo adattare la rete per processare immagini di qualsiasi dimensione, senza doverle ridimensionare.

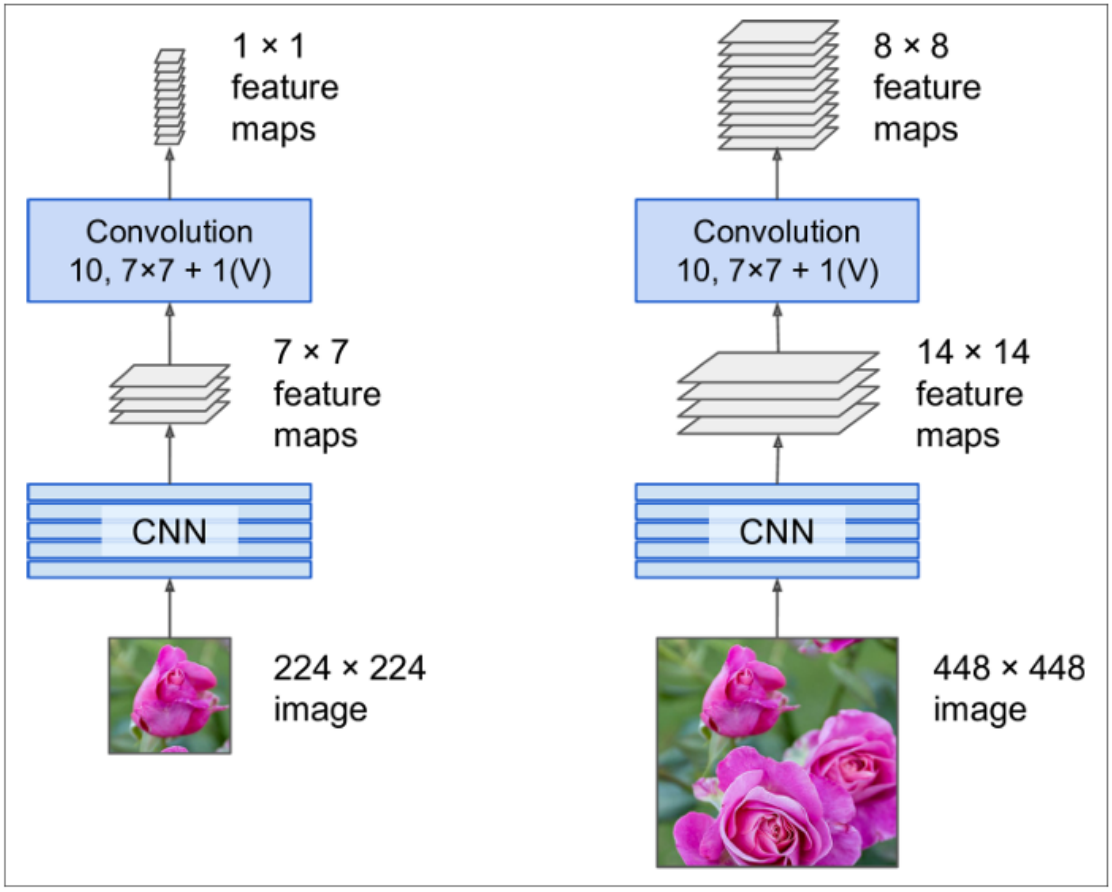

Figura 4.13: Una FCN è in grado di processare immagini di diverse dimensioni [13, p. 489]

Consideriamo il seguente scenario: lo strato convoluzionale finale (chiamato anche bottleneck layer) di questa FCN produce una mappa di caratteristiche di 7x7 quando l'input è un'immagine di 224x224 pixel. Se l'input raddoppia le sue dimensioni a 448x448 pixel, la mappa di caratteristiche non sarà più 7x7, ma

diventerà 14x14. Sostituendo lo strato di output denso con uno strato convoluzionale (utilizzando 10 filtri di 7x7 con stride 1), l'output finale sarà una griglia di 8x8, dove ogni cella conterrà i 10 numeri che avevamo nell'output iniziale.

Questo risultato è equivalente a far scorrere la CNN originale su 64 diverse porzioni dell'immagine grande, ma l'FCN esegue tutto in un unico passaggio, garantendo un'efficienza computazionale molto superiore.

Yolo (You Only Look Once)

YOLO (You Only Look Once) è un'architettura per il rilevamento degli oggetti estremamente veloce ed accurata. Proposta per la prima volta nel 2015 da Joseph Redmon et al. è in continuo miglioramento, anche se l'ultima versione proposta dagli autori risale al 2018 (YOLOv3) [33, 34, 35].

L'architettura YOLOv3 ha segnato un'evoluzione cruciale nel campo del rilevamento di oggetti, raffinando il concetto di approccio a stadio singolo (one-stage detector) e superando le limitazioni dei modelli precedenti. Sebbene le reti interamente convoluzionali avessero già dimostrato l'efficienza del processamento in un singolo passaggio, YOLOv3 ha introdotto una serie di miglioramenti mirati che hanno perfezionato la precisione, soprattutto nel rilevamento di oggetti di piccole dimensioni e in contesti complessi, pur mantenendo una notevole velocità.

Una delle principali distinzioni tra il rilevamento in YOLO e la semplice classificazione è la separazione della previsione della confidenza. Per ogni potenziale bounding box, YOLOv3 calcola due tipi di probabilità distinte. La prima probabilità si basa sul grado di somiglianza detto Objectness Score: questo valore, compreso tra 0 e 1, indica la probabilità che il bounding box contenga effettivamente un oggetto, a prescindere dalla sua classe specifica. La seconda si basa sulla probabilità di Classe: queste sono le probabilità che l'oggetto rilevato appartenga a una delle classi predefinite del dataset (es. "gatto", "automobile", "bicicletta").

La predizione finale di confidenza per una specifica classe viene ottenuta moltiplicando questi due valori. Questo approccio a due stadi all'interno di un'unica rete ha un vantaggio significativo: la rete impara prima a localizzare gli oggetti e solo in un secondo momento a classificarli.
Mentre le architetture precedenti faticavano a rilevare più oggetti in una singola cella della griglia o a gestire oggetti di forme e proporzioni non standard, YOLOv3 ha risolto questo problema introducendo gli anchor boxes (o box priors) e un meccanismo di previsione basato su offset. Gli anchor box sono un set di riquadri di delimitazione predefiniti con forme e dimensioni specifiche (es. alti e stretti per una persona, larghi per un'automobile). Questi "priors" servono come punti di partenza per le previsioni del modello. Per ogni cella della griglia, la rete genera una previsione per ciascuno degli anchor box associati a quella cella. Il numero di anchor box e le loro dimensioni sono tipicamente determinati attraverso un'analisi

di clustering (ad esempio, k-means) sul dataset di addestramento. Il modello non prevede le coordinate assolute di un bounding box. Invece, per ogni anchor box, predice quattro offset (spostamenti) che servono a trasformare le coordinate e le dimensioni dell'anchor box per adattarlo all'oggetto reale. Questa regressione di piccoli valori è molto più stabile e più facile da imparare per la rete rispetto alla previsione delle coordinate assolute.

Un'altra limitazione dei modelli precedenti a stadio singolo era la loro scarsa performance nel rilevamento di oggetti piccoli. YOLOv3 ha risolto questo problema implementando il rilevamento su tre scale differenti. Il modello non genera una singola mappa di previsione, ma tre, ciascuna con una risoluzione diversa:

- Scala per Oggetti Grandi: La mappa con la risoluzione più bassa (es. 13x13 per un'immagine di 416x416) è responsabile del rilevamento di oggetti grandi. . Ogni cella della griglia copre un'ampia area dell'immagine originale.
- Scala per Oggetti Medi: La mappa a risoluzione intermedia (es. 26x26) viene utilizzata per oggetti di dimensioni medie.
- Scala per Oggetti Piccoli: La mappa con la risoluzione più alta (es. 52x52) è cruciale per il rilevamento di oggetti piccoli, poiché ogni cella copre un'area ridotta e può quindi individuare dettagli più fini.

Per ottenere queste tre scale, l'architettura Darknet-53 di YOLOv3 utilizza delle "connessioni di salto" (skip connections), portando le informazioni di basso livello (con alta risoluzione spaziale) dagli strati iniziali della rete agli strati finali di previsione. Questa fusione di informazioni a diverse risoluzioni permette al modello di avere sia un'ampia visione contestuale per gli oggetti grandi, sia una visione dettagliata per gli oggetti piccoli. Il risultato è un netto miglioramento dell'accuratezza, senza compromettere la velocità.

Segmentazione Semantica

Nella Segmentazione Semantica, ogni pixel viene classificato in base alla classe dell'oggetto a cui appartiene. Il problema principale di questo compito è che quando le immagini attraversano una CNN classica, perdono in risoluzione spaziale (dimensione minima di un dettaglio distinguibile). In pratica, una CNN potrebbe essere in grado di rilevare una persona in un'immagine, ma non sarà più precisa di così [36].

Come era successo per il rilevamento degli oggetti, ci sono più approcci per risolvere questo problema. Una soluzione abbastanza semplice fu proposta sempre da Jonathan Long et al.. Gli autori utilizzarono una CNN pre-addestrata e la convertirono in una FCN. La CNN applica un passo di 32 all'immagine di input, il che significa che l'ultimo strato genera feature map che sono 32 volte più piccole

Figura 4.14: Segmentazione Semantica [13, p. 493]

rispetto all'immagine di input. Essendo l'immagine troppo "grossolana" aggiunsero uno strato di sovra-campionamento per moltiplicare la risoluzione per 32. In generale, per il sovra-campionamento, si usano tecniche come l'interpolazione bilineare, che però funziona bene moltiplicando la risoluzione $\times 4$ o $\times 8$. Usarono quindi uno *strato convoluzionale trasposto*. Questo strato prima "stiracchia" l'immagine inserendo colonne e righe vuote (piene di zeri), poi esegue una convoluzione regolare. Per migliorare ulteriormente la soluzione proposta, funzionante ma ancora imprecisa, gli autori aggiunsero le cosiddette *skip connection* (architettura in cui un livello di una rete neurale riceve come input l'output di un livello precedente). Ciò permise di recuperare qualità in termini di risoluzione spaziale (tecniche come la super-risoluzione permettono di ottenere immagini più grandi rispetto all'originale)

4.4 Autoencoder

Gli Autoencoder sono reti neurali in grado di imparare *rappresentazione densa* dai dati in ingresso, chiamata rappresentazione latente (o codifiche), senza nessuna supervisione (apprendimento non supervisionato). Queste codifiche in genere hanno dimensionalità molto inferiori ai dati in input, rendendo gli autoencoder utili per scopi come la riduzione della dimensionalità, specialmente per compiti di visualizzazione. Possono inoltre comportarsi come rilevatori di feature ed essere utilizzati per l'addestramento non supervisionato di reti neurali profonde. Infine possono essere usati come *modelli generativi*: possono generare dati nuovi che assomigliano in tutto e per tutto agli originali. In generale gli autoencoder sono associati ai GAN (Generative Adversarial Network - non trattati in questa tesi) in quanto entrambi apprendono *rappresentazione densa*, possono essere usati entrambi

come modelli generativi e hanno applicazioni simili. Il loro funzionamento però è diverso:

- Gli autoencoder imparano a *copiare* i dati dall'input all'output. Limiti che poi vederemo in seguito rendono il compito complicato e in pratoca il modello è frozato ad imparare modi effecienti per rappresentare i dati.

- I Gan sono composti da due reti neurali. Un *generatore*, che prova a generare dati che assomiglino ai dati usati per l'addestramento e un *discriminatore* che prova a determinare quali dati siano originali e quali falsi. Le due reti neurali "competono" tra di loro questo tipo di addestramento, chiamato "addestramento avversario", è ampiamente considerato una delle tecniche più innovatove degli ultimi anni.

4.4.1 Rappresentazione Efficiente dei Dati

La relazione tra memoria, percezione e corrispondenza dei modelli fu ampiamente studiata negli anni 70' da William Chase e Herbert Simon [37, 38]. Essi osservarono che i giocatori di sacchi erano in grado di memorizzare la posizione dei pezzi sulla scacchiera in soli 5 secondi, un compito che la maggior parte delle persone troverebbe impossibile. Tuttavia notarono che ciò era possibile solo quando la posizione dei pezzi sulla scacchiera era realistica (come se effettivamente si stesse giocando una partita), ma non quando i pezzi erano sparsi casualmente. In pratica, riscontrarono che i giocatori esperti di scacchi non avessero doti mnemoniche migliori, bensì che fossero in grado di riconoscere meglio i *pattern* grazie alla loro esperienza di gioco. Possiamo riassumere il funzionamento di un autoencoder rapportandolo al comportamento di un giocatore di scacchi. L'autoencoder osserva i dati, li converte in una rappresentazione latente efficiente e genera dati che assomigliano fortemente ai dati in ingresso. Un autoencoder è sempre costituito da due parti:

- un Encoder: converte i dati in input in rappresentazione latente.

- un Decoder: converte la rappresentazione latente interna in output.

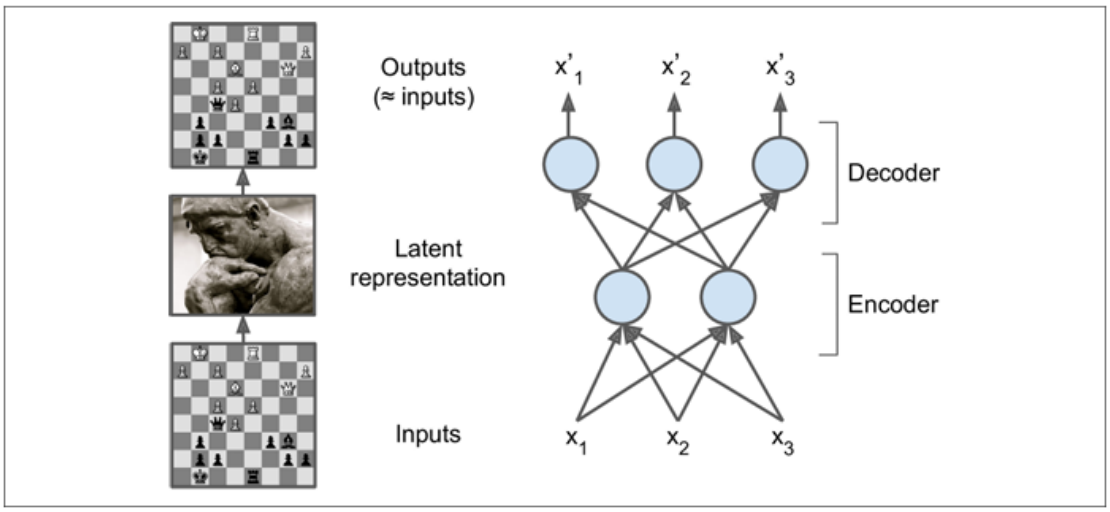

Figura 4.15: Un semplice Autoencoder [13, p. 570]

La struttura di un autoencoder ricorda lontanamente la struttura di un MLP, eccetto per il numero di neuroni nello strato di output. Ciò che l'autoencoder genera viene solitamente chiamato *ricostruzione*, dal momento che esso tenta appunto di ricostruire gli input. La funzione di costo del modello considera una "perdita di ricostruzione" che penalizza il modello quando le ricostruzioni non sono fedeli agli input.

4.4.2 Gli strati di un Autoencoder

Come altre reti neurali, anche gli autoencoders possono avere strati nascosti. In questo caso vengono chiamati *stacked autoencoder* (o *deep autoencoder*). Aggiungere strati a un autoencoder aiuta il sistema a imparare *coding* più complessi. Detto ciò è controproducente rendere un autoencoder troppo potente: il risultato potrebbe essere un autoencoder che mappa ogni input a un numero arbitrario e, di conseguenza, il decoder non farà altro che ricostruire perfettamente il dato in input senza però aver appreso nessuna rappresentazione latente utile nel processo. Generalmente la struttura di un autoencoder è simmetrica rispetto allo strato nascosto centrale. Nell'immagine possiamo vedere uno schema semplice di un autoencoder con 784 input seguito da uno strato nascosto di 100 neuroni, uno strato centrale nascosto di 30 neuroni e specularmente uno strato di altri 100 neuroni che culminano in 784 output.

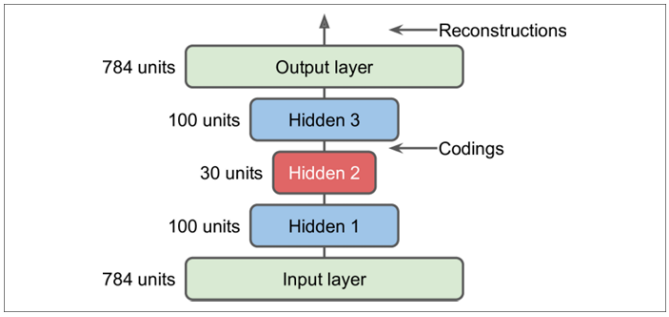

Figura 4.16: Autoencoder a strati nascosti

4.4.3 Autoencoder Convoluzionali

Come visto in precedenza, le reti neurali convoluzionali sono molto più adatte rispetto alle reti *dense* quando si trattano immagini. Nel momento in cui si vuole costruire un autoencoder che come dato in input abbia delle immagini, la soluzione migliore è quella di creare un autoencoder convoluzionale [39]. L'encoder è una CNN regolare composta di strati di convoluzione e di pooling. Riduce la dimensionalità spaziale e nel frattempo incrementa la profondità (numero di feature map). Il decoder deve fare il compito opposto. Si usano quindi strati convoluzionali trasposti abbinati con strati di sovra-campionamento.

Capitolo 5

AI on the Edge e modelli per dispositivi embedded

Con **AI on the Edge** si intende l'esecuzione locale di modelli di apprendimento automatico su dispositivi embedded o sistemi a scheda singola (SBC Single Board Computer) collocati vicino alla sorgente dei dati, senza ricorrere a un backend cloud per l'inferenza. In questo paradigma, l'attenzione è posta sulla *fase di inferenza*—cioè sull'esecuzione del modello in tempo reale—e non sulla fase di addestramento, tipicamente svolta su macchine ad alte prestazioni [40]. Rispetto al cloud, l'elaborazione *on-device* offre tre vantaggi chiave:

- **latenza ridotta**, poiché non è necessario trasmettere i dati.
- **affidabilità e autonomia** in scenari con connettività limitata o intermittente.
- **tutela della privacy** e contenimento dei costi di comunicazione, dato che i dati sensibili restano in loco.

5.1 Sfide e requisiti nell'Edge AI

L'AI on the Edge introduce vincoli progettuali specifici:

- **Risorse computazionali e memoria limitate** rispetto a server/desktop, seppur superiori a quelle dei microcontrollori: la progettazione del modello deve considerare il profilo hardware del dispositivo target (CPU/GPU, RAM, memoria non volatile).
- **Consumi e termica**: il budget energetico è inferiore a quello di server o workstation; la continuità operativa richiede attenzione a frequenze, carico GPU e dissipazione.

- **Latenza end-to-end**: obiettivo primario nei casi d'uso real-time (dall'acquisizione passando per pre-processing, inferenza e post-processing), con requisiti di throughput e jitter.

- **Affidabilità e robustezza**: l'esecuzione in campo deve tollerare variabilità ambientale (illuminazione, vibrazioni, rete non stabile) e recuperare da errori.

5.2 Tecniche utili all'ottimizzazione per l'edge

Sebbene non si tratti di Tiny machine learning (TinyML - machine learning su microcontrollori) in senso stretto, molte tecniche tipiche di modelli "leggeri" restano utili anche su SBC come Jetson Nano, perché migliorano latenza e footprint senza penalizzare eccessivamente l'accuratezza:

- **Quantizzazione** di pesi/attivazioni (es. da FP32 a FP16/INT8) per ridurre memoria e accelerare l'inferenza, con supporto nei runtime edge (*post-training* e/o *quantization-aware*).

- **Pruning** (strutturato) di canali/filtri per diminuire parametri e MACs; utile se poi il backend (es. TensorRT) può sfruttare il modello compresso.

- **Knowledge distillation**, per trasferire conoscenza da un modello "teacher" ad uno "student" più compatto.

5.3 Formati e framework per il deployment *edge*

La portabilità dei modelli è un requisito centrale nell'AI on the Edge. In questo lavoro si è adottato **ONNX** (Open Neural Network Exchange) [41] come formato intermedio standard per la rappresentazione dei modelli. ONNX descrive il modello come un grafo computazionale di *operatori* (nodi) e *tensori* (archi), consentendo l'esportazione da framework eterogenei (PyTorch, TensorFlow, scikit-learn) e l'esecuzione tramite *ONNX Runtime* o motori ottimizzati per l'hardware di destinazione [42].

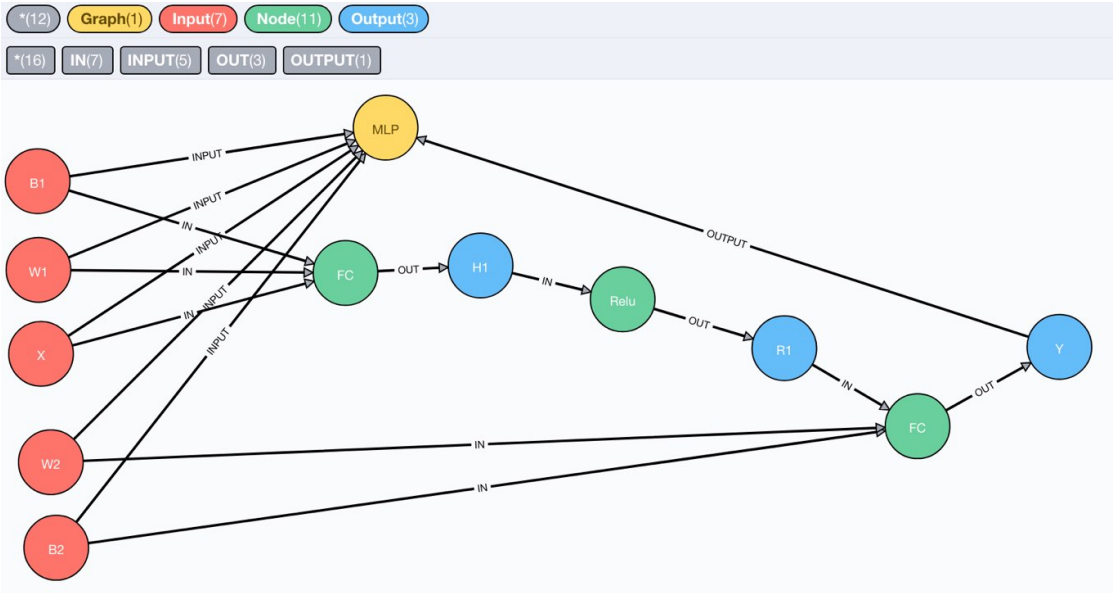

Figura 5.1: Esempio di grafo ONNX: i cerchi denotano operatori, le frecce l'uso/produzione di tensori. La rappresentazione a grafo abilita ottimizzazioni e portabilità tra backend.

Nel contesto Jetson, l'uso di ONNX è funzionale a pipeline che includono **TensorRT** [43] per l'ottimizzazione a runtime (fusioni di operatori, precisioni ridotte, scheduling efficiente). In generale:

- **Modelli PyTorch/Sklearn**: esportazione in `.onnx`;
- **Inferenza** con ONNX Runtime e/o conversione/ottimizzazione per TensorRT quando supportato dall'operatore;
- **Gestione dei casi particolari** (es. varianti di opset o *dtype* richiesti su specifici operatori) per massimizzare la compatibilità.

Altri formati restano rilevanti in ottica multi-piattaforma: **TensorFlow Lite** per dispositivi mobili e, nella sua variante Micro, per MCU; **CoreML** nell'ecosistema Apple; **PyTorch Mobile** su smartphone. In questa tesi, tuttavia, il *deployment* principale è stato realizzato via ONNX su Jetson Nano per ragioni di compatibilità e prestazioni.

Capitolo 6

Metodologia

Dopo aver delineato i fondamenti teorici e lo stato dell'arte, questo capitolo descrive in dettaglio la metodologia adottata per affrontare il problema della rilevazione precoce delle malattie fogliari. L'obiettivo è progettare un sistema capace di distinguere in modo affidabile foglie sane da foglie affette da patologie, riducendo al minimo i falsi negativi, che in ambito agronomico possono comportare gravi conseguenze in termini di diffusione del danno.

La strategia proposta si articola in più fasi, come schematizzato in figura 6.1: in primo luogo, si estrapolano da un dataset specifico, immagini di foglie di vite, utilizzando YOLOv8-seg come tecnologia di segmentazione.Viene poi introdotta un'architettura di autoencoder convoluzionale, utilizzata per ricostruire foglie sane e, successivamente, stimare l'errore di ricostruzione in presenza di anomalie. Tali errori vengono quindi trasformati in feature statistiche, sulle quali si addestra un classificatore supervisionato.Il classificatore scelto in questo caso è una Random Forest.

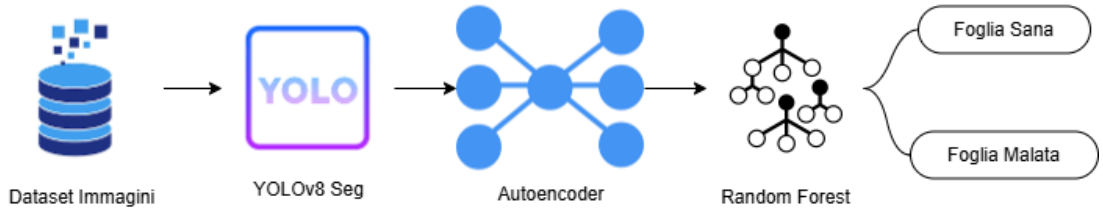

Figura 6.1: Pipeline sviluppata

Il capitolo descrive nel dettaglio il dataset impiegato, che comprende immagini di foglie sane e malate, e le operazioni di preprocessing e segmentazione adottate per uniformare i dati. Viene infine discusso il set di metriche di valutazione (accuracy, precision, recall, F1 e F2), con particolare attenzione al recall della classe "malata",

la cui ottimizzazione risulta prioritaria per ridurre il rischio di errori di mancata rilevazione.

6.1 Architettura proposta

6.1.1 La scelta dell'autoencoder

Per l'estrazione di feature rappresentative è stata adottata un'architettura di autoencoder convoluzionale. L'autoencoder è un modello non supervisionato che apprende a ricostruire il dato in input comprimendolo in uno spazio latente di dimensioni ridotte. Nel caso in esame, esso è stato addestrato esclusivamente su immagini di foglie sane, in modo da apprendere la distribuzione delle caratteristiche tipiche della classe "sana". Durante l'inferenza, la ricostruzione di un'immagine malata presenta inevitabilmente un errore maggiore, che viene utilizzato come segnale discriminante per la rilevazione di anomalie.

La scelta dell'autoencoder si giustifica per tre motivi principali:

- capacità di modellare strutture complesse dei dati senza vincoli di linearità
- possibilità di estrarre feature latenti significative senza etichette;
- utilizzo diretto dell'errore di ricostruzione come indice di anomalia.

Grazie all'uso di strati convoluzionali e funzioni di attivazione non lineari, l'autoencoder è in grado di apprendere relazioni complesse e pattern nascosti nei dati. L'addestramento non supervisionato consente di sfruttare anche dataset privi di annotazioni, aspetto vantaggioso in ambito agronomico dove l'etichettatura manuale può essere costosa e soggetta a errore umano. Infine, la differenza tra l'immagine originale e la ricostruzione fornisce un segnale immediato e interpretabile per distinguere foglie sane da foglie malate, riducendo la necessità di definire feature manuali.

In letteratura sono stati proposti diversi approcci alternativi per l'estrazione di feature da immagini. Un metodo classico è la Principal Component Analysis (PCA), che consente di ridurre la dimensionalità proiettando i dati lungo le direzioni di massima varianza. La PCA ha il vantaggio di essere computazionalmente leggera e di fornire feature interpretabili, ma risulta limitata nella capacità di catturare pattern complessi, poiché si basa esclusivamente su relazioni lineari.

Un secondo approccio consiste nell'impiego di reti convoluzionali pre-addestrate (*transfer learning*), come ResNet o EfficientNet, addestrate su dataset di larga scala quali ImageNet. In questo caso, le attivazioni dei layer intermedi vengono utilizzate come feature descrittive. Tale strategia consente di ottenere embedding robusti anche con dataset ridotti, trasferendo conoscenza da domini generici al

problema specifico. Tuttavia, essa richiede immagini relativamente simili al dominio di pretraining e risulta meno flessibile in contesti caratterizzati da forte variabilità morfologica.

Rispetto a tali soluzioni, l'autoencoder convoluzionale rappresenta un compromesso efficace: permette di modellare relazioni non lineari, non necessita di etichette in fase di addestramento e utilizza direttamente l'errore di ricostruzione come indice di anomalia, risultando quindi particolarmente adatto al problema in esame.

Caratteristica	**PCA**	**CNN pre-addestrate**	**Autoencoder conv.**
Supervisionato	✗	✓	✗
Relazioni non lineari	✗	✓	✓
Interpretabilità feature	✓	✗	✗
Richiede etichette	✗	✓	✗
Peso computazionale	Basso	Medio/Alto	Medio/Basso
Robustezza a variabilità	Bassa	Alta	Media
Uso diretto per anomaly detection	✗	✗	✓

Tabella 6.1: Confronto sintetico tra metodi di estrazione delle feature

Un'ulteriore alternativa è rappresentata dai *Variational Autoencoder* (VAE), che estendono il paradigma degli autoencoder introducendo un approccio probabilistico alla rappresentazione latente. Invece di apprendere una singola codifica deterministica, il VAE stima una distribuzione (tipicamente gaussiana) nello spazio latente, consentendo di modellare l'incertezza e la variabilità intrinseca dei dati. Questa caratteristica rende i VAE particolarmente adatti per scenari di generazione e per la rilevazione di anomalie, poiché campioni anomali tendono a discostarsi dalle distribuzioni apprese. Tuttavia, il loro addestramento risulta più complesso rispetto a un autoencoder standard e richiede un tuning accurato dei parametri, al fine di evitare ricostruzioni poco nitide o fenomeni di *posterior collapse*.

Prima della diffusione dei metodi di deep learning l'estrazione di feature in ambito di visione artificiale era basata prevalentemente su descrittori manuali. Tra i più comuni si annoverano gli istogrammi di colore in spazi HSV o Lab, gli operatori di texture quali i Local Binary Patterns (LBP) e i filtri di Gabor, nonché descrittori di forma basati su contorni e momenti statistici. Tali approcci presentano il vantaggio di essere computazionalmente leggeri e facilmente interpretabili, consentendo un'analisi diretta delle caratteristiche visive rilevanti. Tuttavia, la loro efficacia è fortemente condizionata dalle condizioni di acquisizione delle immagini (illuminazione, rumore, variazioni di scala e rotazione), risultando meno robusti e meno discriminanti rispetto alle feature apprese da modelli neurali.

6.2 Dataset Utilizzato

6.2.1 Origine delle immagini

Le immagini utilizzate per la costruzione del dataset provengono da tre fonti principali. In primo luogo, è stato impiegato un sottoinsieme del dataset *Pic4Ser*, sviluppato presso il Politecnico di Torino, che include immagini di foglie raccolte in condizioni controllate e già annotate in base allo stato fitosanitario.

In secondo luogo, sono state integrate le immagini del dataset open-access *wGrapeUNIPD-DL*, realizzato dal *Department of Land, Environment, Agriculture and Forestry (TESAF)* dell'Università di Padova. Tale corpus comprende 373 immagini di grappoli di uva bianca acquisite nel 2020 in sei località italiane e in diverse fasi fenologiche della pianta (BBCH 69–83). Le immagini, raccolte principalmente con fotocamera reflex Nikon D300 e obiettivo Sigma 18–200, sono corredate da annotazioni in formato YOLO per il rilevamento dei grappoli e, in un sottoinsieme, dai conteggi reali dei grappoli presenti e visibili.

Infine, un ulteriore contributo è stato fornito da una raccolta manuale effettuata direttamente in campo, in diverse fasi fenologiche della crescita delle piante. Questa acquisizione ha permesso di arricchire il dataset con immagini eterogenee, caratterizzate da variabilità in termini di illuminazione, morfologia fogliare e condizioni ambientali, rendendo così l'intero corpus più rappresentativo di scenari reali.

6.2.2 Segmentazione ed etichettatura

Per individuare automaticamente le regioni di interesse (ROI - Region Of Interest) corrispondenti alle foglie, è stato adottato un approccio basato su *object detection*. A tale scopo, è stato addestrato un modello YOLO, che richiede la disponibilità di annotazioni sotto forma di bounding box.

Le etichette sono state generate utilizzando **SALT (Semi-Automatic Labeling Tool)**, che ha consentito di accelerare significativamente il processo di annotazione: lo strumento fornisce proposte di segmentazione automatica, successivamente corrette manualmente dall'operatore. Questa modalità ha permesso di ridurre il tempo necessario rispetto a una completa annotazione manuale, garantendo al contempo la coerenza e l'accuratezza delle bounding box.

Le annotazioni generate tramite SALT sono state esportate nel formato COCO (Common Objects in Context), uno standard ampiamente utilizzato per dataset di visione artificiale. Il formato COCO utilizza file in estensione `.json`, che includono la descrizione delle immagini, delle categorie e delle annotazioni associate. In particolare, ogni bounding box è rappresentata mediante una quadrupla di valori `[x, y, width, height]`, che identifica la posizione e le dimensioni dell'oggetto rilevato.

Figura 6.2: Etichettatura con Salt

L'adozione del formato COCO ha reso le annotazioni compatibili con le principali librerie di deep learning (PyTorch, TensorFlow) e ha permesso l'impiego diretto delle API `pycocotools` per la gestione dei dati. Successivamente, le annotazioni in formato COCO sono state convertite nel formato richiesto da YOLO per l'addestramento del modello di rilevamento.

Il dataset così etichettato è stato utilizzato per l'addestramento del modello YOLO, che in seguito ha permesso l'estrazione automatica delle singole foglie dalle immagini originali.

Data augmentation per l'addestramento di YOLO

Per aumentare la variabilità del dataset e migliorare la capacità di generalizzazione del modello di *object/instance segmentation*, è stata impiegata una strategia di *data augmentation* combinando trasformazioni fotometriche, geometriche e compositive.

Le trasformazioni sul colore sono state applicate nello spazio HSV con ampiezze controllate (variazione di tonalità $h = 0.02$, saturazione $s = 0.6$, valore $v = 0.4$). Sul piano geometrico sono state introdotte rotazioni limitate ($\pm 5°$), traslazioni fino all'8% dell'immagine, *scaling* con fattore $\pm 50\%$, riflessione orizzontale con probabilità 0.5 e senza riflessione verticale (coerente con la natura degli oggetti). Sono state inoltre adottate tecniche compositive: *Mosaic* con probabilità 0.2 (ridotta per preservare la coerenza delle maschere), *Copy-Paste* con probabilità 0.3 per

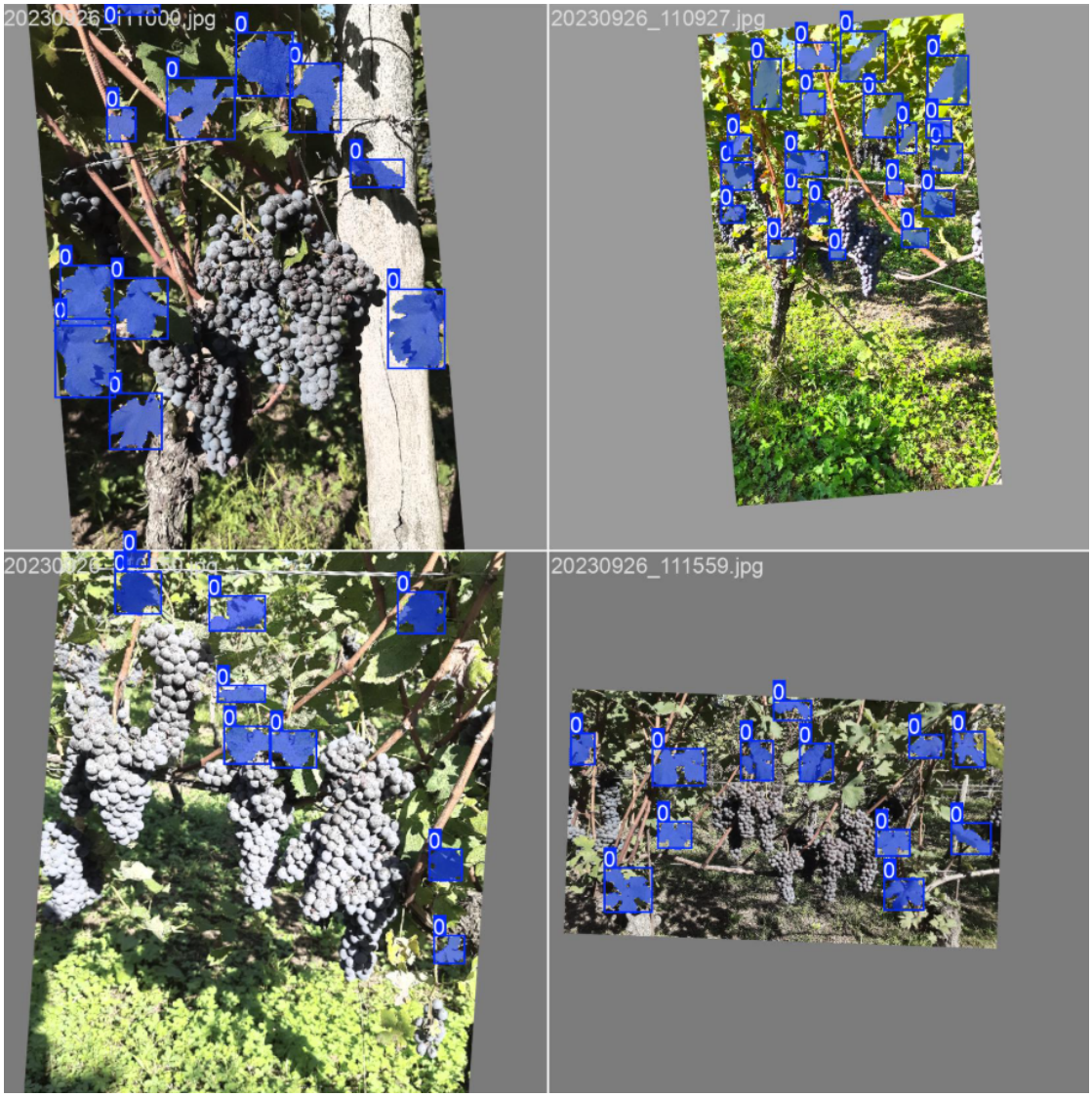

Figura 6.3: Esempio di Data Augmentation

arricchire la densità degli oggetti, mentre *MixUp* è stato disattivato poiché spesso sub-ottimale in segmentazione. Per stabilizzare l'addestramento nelle fasi finali, il *Mosaic* è stato disattivato nelle ultime 10 epoche.

6.2.3 Scelta del classificatore: Random Forest

Nella fase di classificazione delle feature sono state valutate diverse strategie. Un primo approccio ha previsto l'impiego di un classificatore One-Class SVM (OCSVM), con l'obiettivo di modellare la distribuzione della classe "sana" e rilevare anomalie

come deviazioni da tale distribuzione. Tuttavia, la natura sparsa e ad alta variabilità delle feature estratte ha reso difficile la definizione di iperparametri stabili, con risultati prossimi a un comportamento casuale (circa 50% di accuratezza).

Sono state inoltre considerate altre tecniche supervisionate, tra cui *Support Vector Machine* (SVM) e reti neurali leggere a più strati (*Multi-Layer Perceptron*). Le SVM hanno mostrato buona capacità discriminativa, ma hanno richiesto un tuning accurato dei parametri (kernel, C, γ) e una gestione delicata dello sbilanciamento tra le classi, risultando meno pratiche in termini di robustezza e scalabilità. Le reti neurali, pur flessibili, hanno sofferto della limitata dimensione del dataset disponibile, con elevato rischio di overfitting e necessità di procedure di regolarizzazione avanzate.

Alla luce di tali considerazioni, l'adozione di un classificatore *Random Forest* si è rivelata la scelta più pratica ed efficace. La Random Forest, grazie all'aggregazione di più alberi decisionali addestrati su sottoinsiemi casuali di dati e feature, ha garantito una maggiore stabilità dei risultati, una buona interpretabilità attraverso l'analisi dell'importanza delle feature e, soprattutto, un incremento significativo del *recall* sulla classe delle foglie malate, metrica considerata prioritaria nel contesto applicativo. Il percorso metodologico è stato quindi naturale: a partire da un tentativo iniziale con OCSVM, si è giunti progressivamente alla Random Forest come soluzione definitiva, più robusta e coerente con le esigenze del problema.

6.2.4 Esportazione dei modelli On The Edge

Nella progettazione della pipeline è stato necessario individuare un formato di esportazione che garantisse sia la portabilità dei modelli tra diversi framework, sia l'efficienza in fase di deployment su dispositivi edge. Per questo motivo è stato scelto il formato **ONNX** (Open Neural Network Exchange), standard aperto e ampiamente supportato dall'industria.

La principale motivazione risiede nella sua elevata **interoperabilità**: ONNX consente di esportare modelli sviluppati in PyTorch, TensorFlow o Keras e di eseguirli in maniera trasparente tramite *ONNX Runtime*, indipendentemente dall'ambiente di addestramento [42]. In questo modo, il processo di deployment risulta svincolato dal framework originario, riducendo il rischio di *lock-in* tecnologico e aumentando la flessibilità della pipeline.

Un ulteriore vantaggio è rappresentato dalle **ottimizzazioni integrate**: ONNX Runtime offre strumenti per la semplificazione del grafo computazionale e per la quantizzazione post-training, permettendo di ridurre la memoria occupata e la latenza di inferenza. Queste caratteristiche risultano particolarmente rilevanti in un contesto TinyML, dove l'obiettivo è mantenere un equilibrio tra l'accuratezza del modello e i vincoli hardware.

Alla luce di tali considerazioni, ONNX si è dimostrato il formato più adatto per supportare la fase di deployment su piattaforme edge come la **Jetson Nano Developer Kit**, utilizzata in questo lavoro come dispositivo di test. Nonostante le caratteristiche della Jetson Nano non rientrino specificatamente nel contesto del TiyML, ONNX risulta essere comunque il formato migliore in termini di ottimizzazione dei modelli su dispositivi a risorse limitate.

Caratteristica	**ONNX**	**TensorFlow Lite**	**CoreML**
Interoperabilità	Alta (multi-framework)	Limitata (ecosistema TF)	Limitata (ecosistema Apple)
Ottimizzazioni	Graph opt., quantizzazione	Quantizzazione, pruning	Ottimizzazioni Apple HW
Supporto hardware	CPU, GPU, FPGA, NPU, Edge	Microcontrollori, Mobile, Edge	iOS/macOS devices
Ecosistema	Open, cross-platform	Forte in IoT/embedded	Integrato in Apple

Tabella 6.2: Confronto sintetico tra principali formati per il deployment di modelli

Capitolo 7

Apparato sperimentale

7.1 Ambiente di sviluppo

Gli esperimenti sono stati condotti su una workstation con sistema operativo **Ubuntu 22.04** (kernel 6.8.0). Il sistema è equipaggiato con un processore **Intel Core i7-9700K** (8 core, frequenza base 3.60 GHz, fino a 4.9 GHz in boost) e **62 GB di memoria RAM**, con 15 GB di memoria di swap configurata.

Per l'elaborazione parallela sono state utilizzate due GPU dedicate, nello specifico **NVIDIA GeForce RTX 2080** con 8 GB di VRAM ciascuna, supportate dai driver NVIDIA versione 560.35.05 e runtime CUDA 12.6. Tale configurazione ha consentito l'addestramento efficiente dei modelli convoluzionali (YOLOv8-seg e autoencoder), sfruttando l'accelerazione hardware per il calcolo tensoriale.

Tabella 7.1: Specifiche hardware e software della workstation utilizzata

Componente	**Caratteristiche**
Sistema operativo	Ubuntu 22.04 (kernel 6.8.0)
CPU	Intel Core i7-9700K (8 core, 3.60 GHz, fino a 4.9 GHz)
RAM	62 GB + 15 GB di swap
GPU	2x NVIDIA GeForce RTX 2080 (8 GB VRAM ciascuna)
Driver GPU	NVIDIA 560.35.05
CUDA Toolkit	Versione 12.6

7.1.1 Librerie e dipendenze software

Gli esperimenti sono stati implementati in linguaggio Python e condotti all'interno di un ambiente virtuale dedicato. Come ambiente di sviluppo integrato è stato utilizzato **Visual Studio Code**, con il supporto ai **Jupyter Notebook**, che

ha consentito di integrare in un'unica interfaccia la scrittura del codice, la sua esecuzione e la visualizzazione dei risultati.

Le principali librerie impiegate sono state:

- **PyTorch 2.7.0** e **Torchvision 0.22.0** per l'implementazione dell'autoencoder convoluzionale;
- **Ultralytics 8.3.123** per l'addestramento del modello YOLOv8-seg;
- **scikit-learn 1.7.1** per i classificatori Random Forest e SVM;
- **OpenCV 4.11.0** e **scikit-image 0.25.2** per il preprocessing delle immagini;
- **Albumentations** (incluso in Ultralytics) per la data augmentation;
- **pandas 2.2.3** e **NumPy 2.2.5** per la gestione e l'analisi dei dati;
- **Matplotlib 3.10.1** e **Seaborn 0.13.2** per la visualizzazione dei risultati;
- **pycocotools 2.0.10** per la gestione delle annotazioni in formato COCO.

7.2 Divisione dei dataset e coerenza tra le fasi

7.2.1 Dataset per la segmentazione (YOLO)

Il dataset utilizzato per addestrare YOLOv8-seg è **fisso** e non distingue tra foglie sane e malate: l'obiettivo del detector è rilevare *tutte* le foglie presenti nelle immagini. Le annotazioni sono state preparate in formato COCO/YOLO. Il corpus comprende **129** immagini, suddivise in **93** per l'addestramento e **36** per il test; il test set è rimasto *disgiunto* e non è stato impiegato né per la selezione degli iperparametri né per la calibrazione delle soglie. Al termine del training, le foglie individuate sono state ritagliate per costituire l'input delle fasi successive. La validazione durante l'addestramento è stata gestita con *early stopping* (pazienza 50) e chiusura del *mosaic* nelle ultime 10 epoche.

7.2.2 Dataset per l'autoencoder (ricostruzione)

L'autoencoder è stato addestrato solo su foglie **sane**, così da apprendere la distribuzione della normalità e usare l'errore di ricostruzione come segnale di anomalia. Il dataset di input è stato **popolato mediante inferenza** del modello YOLOv8-seg addestrato: su ciascuna immagine sono state rilevate le foglie, quindi **ritagliate** (a partire da bounding box/maschere) per ottenere i *crop* da fornire dell'autoencoder. Dall'insieme dei ritagli sono stati selezionati/validati i campioni sani e sottoposti a

un leggero *ritocco*: rimozione del canale alfa (conversione RGBA→RGB) e pulizia dello sfondo residuo (in particolare porzioni di cielo) al fine di evitare che pattern non pertinenti vengano appresi dal modello. Tutti i crop sono stati infine ridimensionati a 128×128 e normalizzati in [0,1]. Questa procedura assicura coerenza tra i ritagli prodotti dal detector e gli input dell'autoencoder, riducendo il *covariate shift* lungo la pipeline.

7.2.3 Dataset per i classificatori (RF e SVM).

Per la valutazione dei classificatori supervisionati sono stati impiegati due insiemi distinti:

- **foglie sane** (compatibili con l'addestramento dell'autoencoder);
- **foglie malate** (derivate da acquisizioni reali, usando le tecniche di segmentazione e pulizia citate prima).

Questa scelta riflette la disponibilità reale dei dati. Le soglie decisionali sono state calibrate in validazione con criterio *F2-oriented*, privilegiando il *recall* della classe malata.

Nota metodologica

La differenza di *contesto* tra immagini sane (spesso senza sfondo) e malate (talvolta con sfondo residuo) può introdurre un *confondente*. Per mitigarlo, abbiamo:

- standardizzato risoluzione e spazio colore.
- applicato un post–processing dei ritagli per ridurre il cielo residuo (filtro *target–color*).
- basato la classificazione su **feature statistiche** derivate dalla ricostruzione/errore dell'autoencoder, meno sensibili alle texture di sfondo rispetto ai pixel grezzi.

Le feature statistiche sono descrittori compatti che aggregano informazione a livello di immagine o mappa d'errore (distribuzioni, soglie, indici di bordi/texture, statistiche di colore della ricostruzione), con l'obiettivo di catturare pattern anomali senza dipendere eccessivamente dal dettaglio locale dello sfondo.

Feature usate (TOP20). Nel seguito utilizziamo un set di 20 feature organizzate in sei gruppi, riportando tra parentesi i nomi esatti impiegati nel codice:

- **Bordi/texture** — sensibilità a discontinuità e ruvidità nelle aree anomale (calcolate sulla mappa d'errore): `lap_var_x` (varianza del Laplaciano, densità di bordi), `lap_pair_absdiff` (differenza assoluta media tra risposte Laplaciane adiacenti, rugosità/irregolarità).
- **Colore della ricostruzione** — statistiche globali dei canali RGB della *ricostruzione* per intercettare drift cromatici: `rec_r_mean`, `rec_g_mean`, `rec_g_std`, `rec_b_mean`.
- **Errori globali** — intensità media delle discrepanze: `mse` (errore quadratico medio), `mae` (errore assoluto medio) sulla mappa d'errore $|x - \hat{x}|$ (media sui canali).
- **Istogramma dell'errore (low–mid bins)** — forma della distribuzione degli errori: `err_hist_bin0`...`bin7` (conteggi normalizzati in intervalli di errore crescenti nelle fasce bassa–media), utili a distinguere anomalie diffuse vs. locali.
- **Quantili dell'errore** — robusti a outlier: `err_p25` (25° percentile), `err_p50` (mediana) della mappa d'errore.
- **Frazioni over–threshold** — estensione relativa delle aree più anomale: `err_over_0.10`, `err_over_0.20` (percentuale di pixel con errore > 0.10 e > 0.20).

Queste feature, essendo riassuntive (istogrammi, quantili, frazioni su soglia) e/o calcolate sulla **mappa d'errore** anziché direttamente sui pixel dell'immagine, risultano meno influenzate da piccole contaminazioni di sfondo e più coerenti con l'obiettivo di enfatizzare le discrepanze *foglia vs. modello di normalità* appreso dell'autoencoder.

7.3 Addestramento del modello YOLOv8-seg

Per la fase di segmentazione delle foglie è stato addestrato un modello YOLOv8-seg a partire da un dataset annotato manualmente e semi-automaticamente con l'ausilio dello strumento SALT [44] (Segment Anything Labelling Tool), che utilizza il modello SAM e consente l'esportazione delle maschere in formato COCO.

Data augmentation

Per aumentare la variabilità del dataset e migliorare la capacità di generalizzazione del modello, è stata applicata una strategia di *data augmentation* comprendente trasformazioni fotometriche (variazioni HSV), geometriche (rotazioni, traslazioni,

scaling, riflessioni) e compositive (mosaic, copy-paste). I parametri utilizzati sono riportati nella Tabella 7.2, come descritto nel Capitolo 6.

Trasformazione	**Valore**	**Nota/Razionale**
HSV Hue (`hsv_h`)	0.02	Piccola variazione di tonalità per robustezza a differenze cromatiche leggere.
HSV Saturation (`hsv_s`)	0.6	Ampia variazione della saturazione per gestire condizioni di luce diverse.
HSV Value (`hsv_v`)	0.4	Variazione della luminosità per mitigare ombre e riflessi.
Rotazione (`degrees`)	5.0	Rotazioni moderate ($\pm 5°$) per non distorcere le maschere.
Traslazione (`translate`)	0.08	Fino all'8% della dimensione dell'immagine.
Scala (`scale`)	0.5	Zoom $\pm 50\%$ per variare la dimensione apparente degli oggetti.
Shear (`shear`)	0.0	Disattivato per evitare deformazioni delle maschere.
Prospettiva (`perspective`)	0.0	Disattivata per stabilità delle annotazioni.
Flip orizzontale (`fliplr`)	0.5	Probabilità 50%: utile e sicuro per il dominio.
Flip verticale (`flipud`)	0.0	Disattivato: gli oggetti non sono semanticamente capovolgibili.
Mosaic (`mosaic`)	0.2	Bassa probabilità per preservare coerenza delle maschere; `close_mosaic`=10.
MixUp (`mixup`)	0.0	Disattivato: tipicamente poco vantaggioso in segmentazione.
Copy-Paste (`copy_paste`)	0.3	Aumenta densità e co-occorrenza di oggetti mantenendo maschere consistenti.

Tabella 7.2: Parametri data augmentation utilizzati per YOLO

Il training del modello è stato eseguito per 200 epoche con *early stopping* (pazienza 50), dimensione delle immagini pari a 832 pixel, batch size 4 e learning rate iniziale fissato a 0.001. L'addestramento non è stato condotto da zero, ma in modalità di *fine-tuning*, utilizzando YOLOv8-seg con pesi *pretrained*, derivati dal pre-addestramento sul dataset COCO. Questa scelta ha permesso di sfruttare le feature generali già acquisite dal modello di base, riducendo i tempi di convergenza e migliorando la capacità di generalizzazione, particolarmente importante in presenza di un dataset di dimensioni limitate.

Parametro	Valore	Descrizione
Epoche	200	Numero massimo di cicli di addestramento
Early stopping	50	*Patience*: interruzione se non si osservano miglioramenti
Dimensione immagine	832	Risoluzione di input (`imgsz`)
Batch size	4	Numero di immagini per iterazione
Learning rate iniziale	0.001	Tasso di apprendimento per l'ottimizzazione
Strategia	Fine-tuning	Addestramento non da zero, ma a partire da pesi *pretrained*
Modello di partenza	YOLOv8-seg (COCO)	Pesi pre-addestrati sul dataset COCO

Tabella 7.3: Iperparametri utilizzati per l'addestramento di YOLOv8-seg

7.4 Implementazione dell'autoencoder

Per la fase di ricostruzione delle immagini è stato implementato un autoencoder convoluzionale, testato in più configurazioni architetturali con l'obiettivo di individuare la struttura più adatta al dataset disponibile.

7.4.1 Varianti architetturali valutate

Per selezionare l'architettura di autoencoder più adatta, sono state implementate e confrontate quattro varianti convoluzionali con skip connections, differenziate per ampiezza del decoder, profondità degli skip e tecniche di regolarizzazione.

AE_BaseSkip. Architettura con encoder a quattro blocchi (canali: 64–128–256–512, stride 2, kernel 4×4) e decoder simmetrico. Nel primo upsampling (`decoder4`) concatena le *deep features* con sé stesse ($[x_4, x_4]$, 1024 canali) prima di proiettare a 256; mantiene uno skip esplicito da x_3 a 16×16. Attivazioni ReLU, uscita Sigmoid, loss L1 combinata con loss percettiva. Questo modello utilizza una **ridondanza** strategica nel collo di bottiglia.

AE_Base. Come `AE_BaseSkip` ma **rimuove** la ridondanza in `decoder4` (niente $[x_4, x_4]$): usa solo 512 → 256 al primo upsampling, mantenendo lo skip moderato da x_3. Riduce parametri e potenziale sovra-adattamento sul collo di bottiglia, favorendo una rappresentazione più **compatta**.

AE_Wide. Decoder **più largo** (384→192→96 canali) e skip "ricco" da x_3 (concat $[d4, x_3]$ con 640 canali in ingresso a `decoder3`). Nessuno skip da x_2/x_1 per

contenere la complessità; mira a migliorare la **qualità**di**ricostruzione** preservando contesto profondo.

AE_DeepLowPassSkip. Come `AE_Wide` (decoder 384 → 192 → 96) ma lo skip profondo è **filtrato** e ridotto: proiezione 1×1 (256 → 64) su x_3, low-pass con AvgPool 3×3 e Dropout2d p=0.3 prima della concatenazione. Obiettivo: trasferire contesto ı**pulito**ȷ e a bassa capacità per evitare il **leakage** di dettagli anomali nelle ricostruzioni.

Criteri di selezione. Le varianti sono state confrontate su tre assi:

- **MSE** di ricostruzione.
- **valutazione visiva** di venature/bordi/lesioni.
- impatto downstream sul **recall** della classe malata con Random Forest (feature dall'errore di ricostruzione).

La scelta finale privilegia il miglior compromesso tra fedeltà di ricostruzione e sensibilità alle anomalie.

7.4.2 Criteri di valutazione

La selezione dell'architettura più adatta di autoencoder è stata effettuata sulla base di tre criteri complementari:

- **Errore numerico di ricostruzione**: per ciascun modello è stato calcolato il Mean Squared Error (MSE) tra immagine originale e ricostruzione, utilizzato come indicatore quantitativo della fedeltà del modello. Oltre al MSE, sono state usati altri parametri che basano il loro principio sull'errore di ricostruzione e sulla distribuzione dell'errore di distribuzione.
- **Valutazione visiva delle ricostruzioni**: le immagini generate dal decoder sono state confrontate con gli input originali, verificando la capacità del modello di preservare dettagli strutturali quali venature, bordi e ed eliminare le aree danneggiate, ricostruendo di fatto una foglia sana.
- **Prestazioni con classificatore Random Forest**: le mappe di errore ottenute da ciascun autoencoder sono state trasformate in feature statistiche e utilizzate come input per la Random Forest. L'efficacia delle ricostruzioni è stata quindi valutata anche indirettamente, osservando l'impatto sul *recall* della classe minoritaria (foglie malate).

Nella scelta delle architetture il criterio guida è stato il livello di dettaglio da preservare. Un eccesso di dettaglio rischia di trascinare nel modello anche rumori e indizi spuri di malattia; al contrario, un dettaglio insufficiente comporta una perdita generica di qualità e delle feature discriminanti utili anche a riconoscere una foglia sana. L'obiettivo è stato quindi trovare un compromesso che conservasse le strutture rilevanti (venature, bordi, texture) riducendo al minimo il contenuto non informativo. L'autoencoder impiega una skip connection tra il terzo livello dell'encoder e il decoder (schema tipo U-Net) per preservare informazioni spaziali che andrebbero perse con i downsampling. Nel nostro caso la skip è filtrata (conv $1\times1 \rightarrow$ avg-pool $\rightarrow$ dropout): la riduzione di canali e il low-pass attenuano il rumore fine e i dettagli non informativi, mentre il dropout limita l'overfitting. Questa scelta è coerente con il criterio progettuale sul livello di dettaglio: trasferire al decoder le strutture utili (venature, bordi, pattern locali della lamina) senza trascinare micro-variazioni spurie che potrebbero somigliare a segnali di malattia. In combinazione con il background removal cromatico (rimozione del cielo), la skip connection filtrata consente una ricostruzione più focalizzata sulla tessitura della foglia, migliorando sia la qualità delle feature da ricostruzione (MSE/quantili/pixel-soglia/Laplaciano) sia la separabilità finale tra foglie sane e malate.

Figura 7.1: (1)AE_BaseSkip, (2)AE_Wide, (3)AE_Base, (4)AE_DeepLowPassSkip

Scelta finale e motivazioni

I risultati mostrano che, dopo calibrazione *F2-oriented*, tutte le varianti raggiungono **recall = 1.0** sulla classe malata; tra queste, **AE_DeepLowPassSkip** garantisce il miglior compromesso complessivo: **accuracy 0.875**, **precision 0.647**, $F_1 =$

0.786, $F_2 = 0.902$ e **ROC-AUC 0.975**. La soglia ottimale risulta bassa ($\tau \approx 0.143$), coerente con l'enfasi sul recall e lo sbilanciamento di classe. **AE_Base** e **AE_BaseSkip** mantengono recall pieno ma con precisione inferiore (0.478 e 0.440) e accuracy più bassa (0.75 e 0.708). **AE_Wide** è la più penalizzata in precisione (0.379) e accuracy (0.625), producendo più falsi positivi. Sulla base di questi elementi, **AE_DeepLowPassSkip** è selezionato come modello di ricostruzione per la pipeline finale.

7.5 Classificatore supervisionato

Sebbene siano stati inizialmente presi in considerazione diversi approcci (SVM e One-Class SVM), i risultati sperimentali hanno mostrato prestazioni non soddisfacenti, in particolare sul *recall* della classe minoritaria (foglie malate). Per questo motivo si è deciso di adottare come unico classificatore la **Random Forest**, che ha garantito maggiore stabilità, robustezza agli sbilanciamenti del dataset e migliori metriche complessive.

La valutazione delle prestazioni è stata effettuata utilizzando le seguenti metriche:

- **Accuracy**: frazione di campioni correttamente classificati sul totale.
- **Precision (classe malata)**: frazione di campioni effettivamente malati tra quelli predetti come tali.
- **Recall (classe malata)**: frazione di campioni malati correttamente individuati sul totale dei malati.
- **F1-score**: media armonica tra precision e recall, con uguale peso.
- **F2-score**: variante del F-score che assegna maggiore peso al recall, scelta come metrica prioritaria in quanto nel contesto applicativo è più importante ridurre i falsi negativi (foglie malate non riconosciute) rispetto ai falsi positivi.
- τ: è la *soglia* applicata al punteggio del classificatore $s(x) \in [0,1]$ (probabilità/punteggio di "malata"). Se $s(x) \geq \tau \Rightarrow$ etichetta *malata*; altrimenti *sana*.

7.5.1 Valutazione esplorativa e selezione delle feature

Prima di adottare la Random Forest come classificatore definitivo, è stata condotta un'analisi esplorativa con SVM per comprendere la rilevanza delle feature e la separabilità del problema. Sia per la SVM che per la Random Forest sono stati presi in considerazione insiemi di feature diversi, considerando le migliori feature per il numero di feature preso in considerazione. In particolare SLIM6, TOP20 e

FULL40 sono stati i sottoinsiemi maggiormente studiati, valutando contemporaneamente aspetti come il peso computazionale complessivo e i risultati effettivi di classificazione.

Per stimare l'importanza relativa dei descrittori sono state impiegate due tecniche complementari:

- **Permutation Importance** scoring F1 sul test
- **RFECV** con LinearSVC su dati standardizzati.

L'analisi è stata completata con la **PCA 2D** (post-scaling) e l'ispezione delle **distribuzioni univariate** delle feature per classe.

7.5.2 Classificatore finale: Random Forest

Nonostante l'analisi esplorativa con SVM, il classificatore adottato in modo definitivo è la **Random Forest**, in quanto ha mostrato prestazioni più stabili e un **miglior recall sulla classe malata**, metrica prioritaria nel contesto applicativo. La valutazione è stata effettuata tramite **accuracy**, **precision** e **recall** (classe malata), **F1** e **F2** (con enfasi su F_2 per privilegiare il recall). Le soglie decisionali sono state calibrate in validazione con criterio *F2-oriented*.

Selezione delle feature per Random Forest.

I risultati indicano che l'aumento del numero di feature migliora le prestazioni fino a una sostanziale saturazione. **SLIM6** si ferma a:

- *Accuracy* $= 0.715 \pm 0.061$.
- *Precision* = (malate) 0.369 ± 0.101.
- $F_2 = 0.558 \pm 0.158$.

TOP20 raggiunge:

- *Accuracy* $= 0.820 \pm 0.030$.
- *Precision* $= 0.572 \pm 0.083$ e $F_2 = 0.811 \pm 0.046$.

FULL40 risulta solo marginalmente superiore:

- *Accuracy* $= 0.839 \pm 0.030$.
- *Precision* $= 0.577 \pm 0.022$
- $F_2 = 0.824 \pm 0.016$

Poiché il costo computazionale della Random Forest non varia sensibilmente con il numero di feature ma l'export/serving in *ONNX* beneficia di input più compatti, si adotta **TOP20** come set di riferimento (soglia ottimale media $\sim 0.236 \pm 0.050$ contro 0.216 ± 0.027 di FULL40), bilanciando semplicità e prestazioni.

Tabella 7.4: Alcune feature del set **TOP20** usate dalla Random Forest (nomi operativi del codice).

Feature	**Definizione**	**Intuizione (foglie malate)**
`lap_var_x`	Varianza della risposta del filtro Laplaciano applicato alla mappa d'errore $\lvert x-\hat{x}\rvert$ (media canali).	Lesioni e bordi irregolari aumentano le discontinuita' $\Rightarrow$ varianza piu' alta.
`lap_pair_absdiff`	Differenza assoluta media tra risposte Laplaciane adiacenti (rugosita'/irregolarita' della mappa d'errore).	Tessiture anomale e margini frastagliati fanno crescere la rugosita' locale.
`rec_r_mean`	Media del canale R della ricostruzione $\hat{x}$.	Drift cromatico della ricostruzione (dominanti atipiche) legato a mismatch strutturali.
`rec_g_mean`	Media del canale G della ricostruzione.	Variazioni del verde ricostruito (stress/necrosi alterano la resa del canale G).
`rec_g_std`	Deviazione standard del canale G della ricostruzione.	Eterogeneita' cromatica ricostruita; maggiore variabilita' puo' riflettere anomalie.
`rec_b_mean`	Media del canale B della ricostruzione.	Riflessi/sfondo residuo o alterazioni cromatiche possono spingere il canale B.
`mse`	Errore quadratico medio sulla mappa d'errore ($\mathrm{MSE} = \mathbb{E}[(x-\hat{x})^2]$).	Intensita' globale dell'anomalia: lesioni estese $\Rightarrow$ MSE piu' alto.
`mae`	Errore assoluto medio sulla mappa d'errore ($\mathrm{MAE} = \mathbb{E}[\lvert x-\hat{x}\rvert]$).	Misura robusta dell'errore medio, meno sensibile ai picchi rispetto a MSE.
`err_hist_bin0`	Quota di pixel nei bin piu' bassi dell'istogramma dell'errore (fascia low).	Ricostruzione molto fedele: prevalgono errori bassi nelle foglie sane.
`err_hist_bin1`	Conteggio normalizzato nel bin successivo (low).	Spostamento verso errori maggiori segnala anomalie piu' diffuse.
`err_hist_bin2`	Conteggio normalizzato nel bin 2 (low–mid).	Aumento nei bin low–mid indica errore piu' pervasivo.
`err_p25`	25-esimo percentile della mappa d'errore.	Se cresce, una parte ampia dei pixel presenta errore non trascurabile.
`err_p50`	Mediana della mappa d'errore.	Misura robusta dell'intensita' tipica; alta nelle foglie con anomalie diffuse.
`err_over_0.10`	Frazione di pixel con errore > 0.10.	Estensione relativa delle aree piu' anomale (soglia media).
`err_over_0.20`	Frazione di pixel con errore > 0.20.	Estensione relativa delle aree molto anomale (soglia alta).

7.6 Jetson Nano Developer Kit

7.6.1 Ambiente di sistema

Il dispositivo impiega Linux for Tegra (L4T) su Ubuntu 18.04 con kernel `4.9.253` (JetPack 4.x). L4T integra lo stack NVIDIA per l'esecuzione di modelli: CUDA,

cuDNN e TensorRT. Sui sistemi Jetson non è disponibile `nvidia-smi`; per la diagnostica si utilizzano `tegrastats` e `nvpmodel`.

7.6.2 Caratteristiche hardware rilevanti

- **CPU**: Quad-core ARM Cortex-A57 (64 bit, architettura `aarch64`).
- **GPU**: NVIDIA Maxwell, 128 CUDA cores, compute capability `5.3`.
- **Memoria**: **4 GB LPDDR4** condivisa CPU/GPU (UMA), banda ∼25.6 GB/s.
- **Storage**: microSD (128 Gb).
- **Power modes**: **5 W/10 W** via `nvpmodel`; `jetson_clocks` per fissare i clock.

L'esecuzione su Jetson Nano è stata containerizzata tramite **Docker** con **nvidia-container-runtime** per l'accesso alla GPU all'interno del container. È stata utilizzata l'immagine ufficiale `nvcr.io/nvidia/l4t-ml:r32.7.1-py3`, che integra CUDA/cuDNN, PyTorch, TensorFlow, OpenCV e Jupyter preconfigurati per L4T R32.7.1.

All'interno del container è stato creato un **ambiente virtuale** per isolare le dipendenze del progetto. Questa scelta garantisce riproducibilità e separazione rispetto ai pacchetti preinstallati nell'immagine l4t-ml [45]. I dataset e gli artefatti di inferenza/addestramento sono stati resi persistenti montando le directory host come volumi.

Librerie utilizzate per la conversione da `.pth` a `.onnx`

Per la conversione dei modelli addestrati in formato PyTorch (`.pth`) verso il formato standard `ONNX`, sono state impiegate diverse librerie Python appartenenti a ecosistemi differenti. Ciascuna ha svolto un ruolo specifico nel processo di esportazione, validazione e semplificazione del grafo computazionale.

- **PyTorch (`torch`, `torch.onnx`)** Principale libreria di deep learning utilizzata per la definizione e l'addestramento dei modelli (Autoencoder e YOLOv8). Fornisce la funzione `torch.onnx.export()`, che consente di serializzare un modello in un grafo compatibile con ONNX, specificando input fittizi e versione dell'opset.
- **ONNX (`onnx`)** Libreria ufficiale per la manipolazione e validazione dei grafi ONNX. È stata utilizzata per caricare i file esportati, ispezionare gli operatori presenti, sostituire i tensori di tipo `INT64` con `INT32` e verificare la consistenza strutturale tramite `onnx.checker.check_model()`.

- **ONNX Simplifier (`onnxsim`)** Strumento di semplificazione automatica dei grafi ONNX, impiegato principalmente per YOLOv8-seg. Ha permesso di rimuovere nodi ridondanti e di sostituire operatori non supportati da TensorRT, generando le versioni finali `_int32_sim.onnx`.
- **NumPy (`numpy`)** Utilizzata per la conversione dei tensori e la manipolazione diretta dei dati numerici (ad esempio durante la sostituzione dei tensori `INT64`).
- **TensorRT Tools (`trtexec`)** Strumento a riga di comando fornito da NVIDIA per il parsing, la validazione e la compilazione dei modelli ONNX in engine ottimizzati (`.engine`). È stato utilizzato per verificare la compatibilità dei modelli su Jetson Nano e generare le versioni FP16 eseguibili sulla GPU integrata.
- **scikit-learn e skl2onnx** Utilizzate per la parte di apprendimento classico (Random Forest). Il pacchetto `skl2onnx` ha permesso di esportare direttamente il classificatore `RandomForestClassifier` in formato ONNX, preservando i parametri e la struttura dei nodi decisionali. Non sono state necessarie semplificazioni aggiuntive né conversioni di tipo.
- **ONNX Runtime (`onnxruntime`)** Framework di inferenza usato per testare e verificare i modelli ONNX su CPU e su Jetson Nano (nel caso della Random Forest). Ha permesso di validare le predizioni rispetto ai risultati ottenuti in ambiente di addestramento.
- **joblib** Utilizzata per salvare e ricaricare in formato binario gli oggetti Python (modelli, parametri, soglie di classificazione) associati alle fasi di addestramento, prima dell'esportazione in ONNX.

In sintesi, la combinazione di queste librerie ha consentito di coprire l'intero ciclo di esportazione — dalla conversione dei modelli PyTorch, alla semplificazione del grafo, fino alla validazione e inferenza in ambiente embedded — garantendo la compatibilità con la piattaforma NVIDIA Jetson Nano.

Capitolo 8

Esperimenti e Risultati

8.1 Obiettivi e protocollo di valutazione

Obiettivo e costo degli errori. Nel contesto applicativo, i **falsi negativi** (foglie malate classificate come sane) sono più critici dei falsi positivi; per questo la valutazione privilegia la capacità di intercettare la classe malata.

Metriche di base. Sia $\mathrm{TP}, \mathrm{FP}, \mathrm{TN}, \mathrm{FN}$ la matrice di confusione.

$$\text{Accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{FP} + \mathrm{TN} + \mathrm{FN}} \tag{8.1}$$

$$\text{Precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{8.2}$$

$$\text{Recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{8.3}$$

In presenza di *class imbalance* l'accuracy può risultare fuorviante; pertanto le metriche guida sono **Precision** e **Recall** sulla classe malata.

Calibrazione della soglia. I punteggi del classificatore sono trasformati in etichette mediante una soglia τ scelta in **validazione** per massimizzare F_2 (F2-oriented). La soglia ottimale $\tau^\star$ è quindi applicata una sola volta al test set per la stima imparziale delle prestazioni.

8.2 Segmentazione YOLOv8-seg (valutazione qualitativa)

L'inferenza è stata eseguita con YOLOv8-seg (input 832 px, `conf`=0.25, `iou`=0.50, GPU), sul test set di immagini contenenti foglie sane. Per ciascuna immagine si producono maschere e bounding box delle foglie; i ritagli risultanti costituiscono l'input delle fasi di ricostruzione (autoencoder) e classificazione.

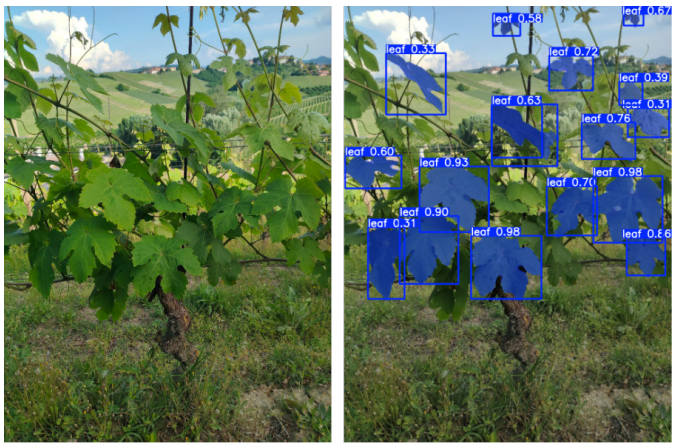

Figura 8.1: Esempio inferenza YOLOv8-seg foglie sane

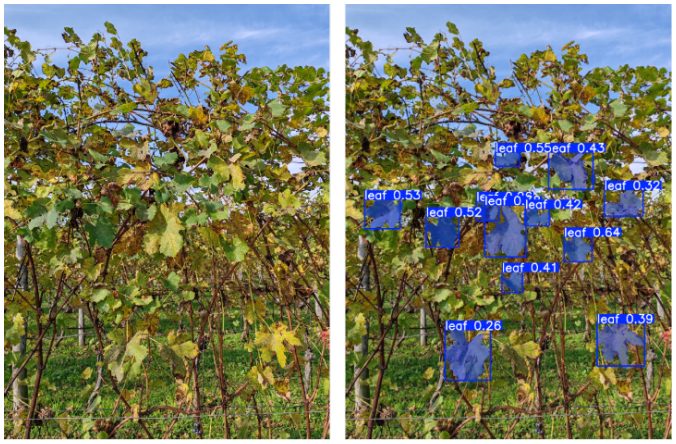

Figura 8.2: Esempio inferenza YOLOv8-seg foglie malate

In inferenza (input 832 px, `conf`=0.25, `iou`=0.50, GPU RTX 2080), YOLOv8-seg ha mostrato una latenza media pari a **1,137592 ms/immagine** (mediana = latenza media), corrispondenti a circa **886.625 FPS** con batch=1. Non sono stati rilevati particolari picchi di memoria (VRAM allocata e riservata vicini allo 0).

Tabella 8.1: Efficienza di inferenza di YOLOv8-seg (imgsz 832, conf 0.25, iou 0.50, RTX 2080)

Metrica	**Valore**
Tempo medio	1,137592 ms/img
Tempo mediano	1,137592 ms/img
FPS	∼886,625

Il modello YOLOv8-seg produce immagini con canale α (RGBA). Tale canale viene rimosso convertendo i frame in RGB, così da lavorare unicamente su immagini effettivamente segmentate. Successivamente, dopo il cropping basato sui bounding box generati da YOLOv8-seg, è stata adottata una segmentazione basata sul colore per il background removal: in particolare, si definisce in spazio HSV un insieme di intervalli cromatici (intorno ai toni del cielo) e si eliminano i pixel che ricadono in tali range e nei loro intorni. Questa scelta consente di ridurre il rumore di fondo e i dettagli non pertinenti, migliorando sia la fase di addestramento dell'autoencoder sia l'inferenza, che risulta più pulita e focalizzata sulla tessitura fogliare.

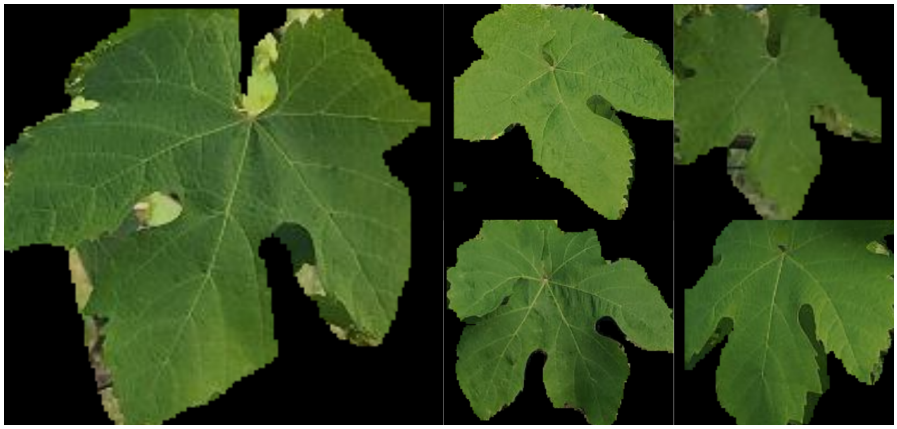

Figura 8.3: Esempio segmentazione foglie sane

Figura 8.4: Esempio segmentazione foglie malate

8.3 Selezione dell'AE e qualità di ricostruzione

Dopo aver generato diversi modelli di Autoencoder, si sono dovute confrontare le varianti architetturali (AE_Base, AE_BaseSkip, AE_Wide, AE_DeepLowPassSkip) e selezionare il modello più adatto alla pipeline, secondo tre criteri:

- errore numerico di ricostruzione (MSE)
- valutazione visiva
- impatto downstream sul *recall* della classe malata con Random Forest.

Tutte le varianti sono state addestrate con loss MSE tra immagine originale e ricostruzione, input 128×128, ottimizzatore Adam (lr iniziale 0.001), batch size 32 ed *early stopping*. Le ricostruzioni sono state generate su un validation set disgiunto; dalle mappe di errore ($|x - \hat{x}|$ medio sui canali) sono state derivate le feature statistiche impiegate nella classificazione.

8.3.1 Valutazione quantitativa e qualitativa

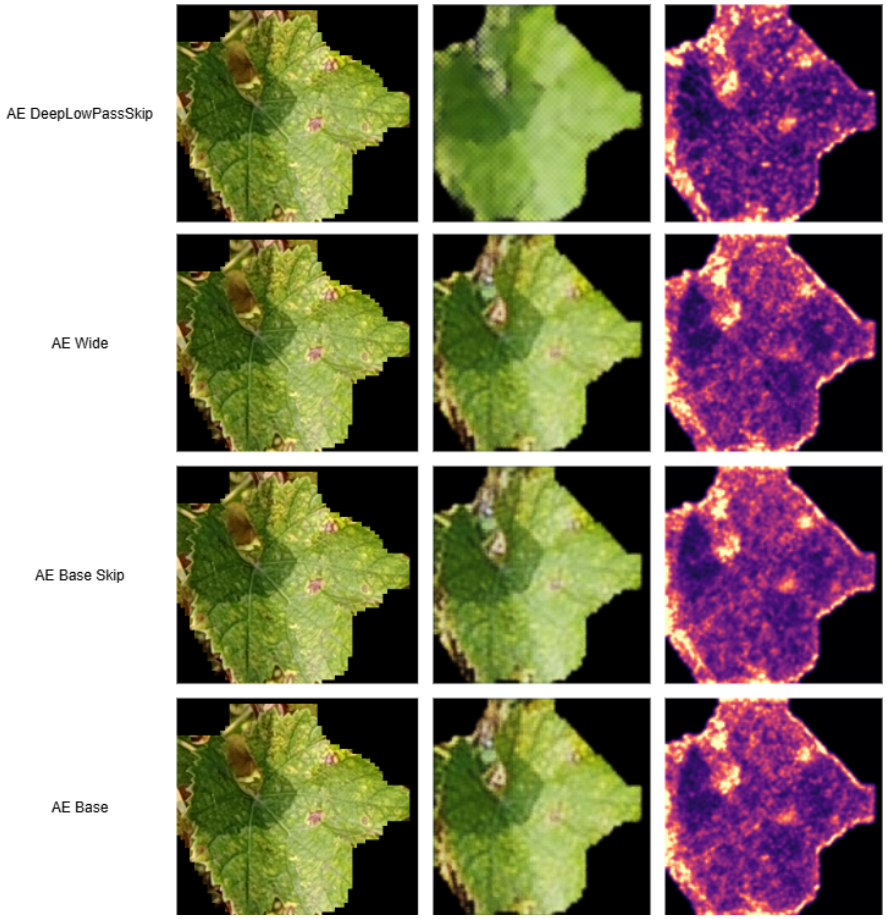

Figura 8.5: Ricostruzione tramite Autoencoder

Per ciascun modello sono stati ispezionati qualitativamente i risultati su esempi rappresentativi (foglie sane e malate). Le ricostruzioni di **AE_DeepLowPassSkip** generano immagini che perdono completamente dettagli di ricostruzione che non rientrino nello spettro delle foglie sane. Ciò permette poi in fase di confronto con l'originale una discrepanza maggiore, il che risulta nell'enfatizzazione delle aree effettivamente malate;

Metrica	AE_DeepLowPassSkip	AE_Base	AE_BaseSkip	AE_Wide
Accuracy	**0.875**	0.750	0.708	0.625
Recall M	**1.000**	**1.000**	**1.000**	**1.000**
Precision M	**0.647**	0.478	0.440	0.379
F1 M	**0.786**	0.647	0.611	0.550
F2 M	**0.902**	0.821	0.797	0.753
ROC_AUC	**0.975**	0.875	0.885	0.889
Threshold	0.143	0.147	0.118	0.105

Nota: *Threshold* indica la soglia decisionale applicata sulla probabilità di classe `malata`; non viene evidenziata perché non rappresenta una metrica di qualità.

8.3.2 Sintesi e scelta del modello

L'architettura **AE_DeepLowPassSkip** si è dimostrata la più performante tra i modelli di autoencoder confrontati. Essa combina una struttura profonda con meccanismi di *skip connection* e un filtraggio a bassa frequenza (*low-pass*) che consente di catturare le informazioni di contesto spaziale e, al tempo stesso, di preservare i dettagli strutturali rilevanti nelle regioni anomale. Dal punto di vista quantitativo, **AE_DeepLowPassSkip** ottiene la migliore accuratezza complessiva (0.875) e i punteggi più elevati su tutte le metriche principali — in particolare **F1_malata = 0.7857**, **F2_malata = 0.9016** e **ROCAUC = 0.975**. Questi valori indicano una capacità superiore nel bilanciare precisione e richiamo, aspetto cruciale in applicazioni agronomiche dove l'obiettivo è massimizzare la *recall* delle foglie malate.

Al contrario, le architetture più semplici come **AE_Base** e **AE_Wide** presentano un'evidente perdita di precisione (rispettivamente 0.478 e 0.379), segno di una ricostruzione meno accurata e di un minor potere discriminante nelle feature derivate. Il modello **AE_BaseSkip**, pur mantenendo un'ottima *recall*, mostra valori intermedi e una tendenza a sovrastimare la classe "malata".

In sintesi, l'architettura AE_DeepLowPassSkip rappresenta un buon compromesso tra complessità computazionale e capacità di generalizzazione: l'inclusione di connessioni di salto consente una migliore propagazione dell'informazione, mentre il filtraggio low-pass riduce il rumore nelle mappe d'errore, rendendola particolarmente adatta come generatore di feature per la classificazione Random Forest.

8.4 Random Forest

In fase sperimentale si è notato come la Random Forest fosse la scelta migliore per la classificazione. Facendo inferenza con l'autoencoder è stato possibile analizzare e mettere a confronto l'immagine generata con l'immagine originale. Il primo approccio intuitivo è stato quello di valutare diverse tipologie di errore di ricostruzione.

Vengono riportati i grafici relativi agli errori di ricostruzione dei pixel, di colore e di SIIM, con la loro mappa d'errore su un caso di esempio di foglia malata;

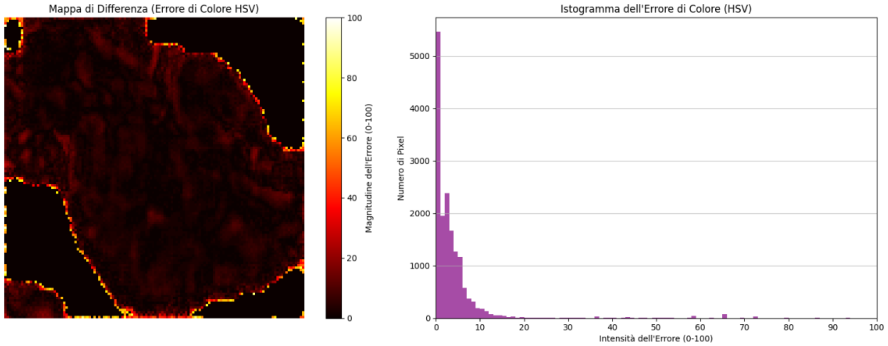

Figura 8.6: Errore di Colore

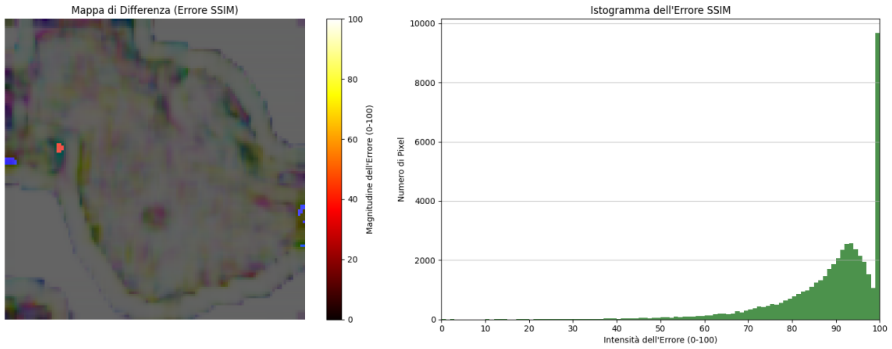

Figura 8.7: SSIM (Structural Similarity index)

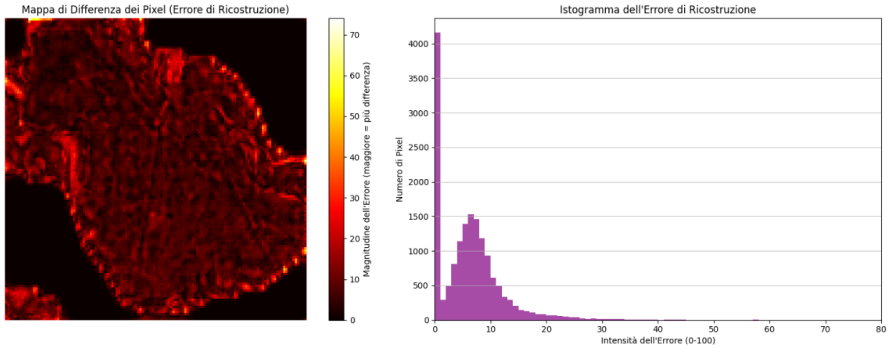

Figura 8.8: Errore di Ricostruzione

L'intuizione si è rivelata interessante ma non sufficiente. I classificatori presi in considerazione non riuscivano a discernere in maniera efficace le due classi, necessitando di altre feature. Di seguito viene inserito un piccolo riassunto numerico a supporto della tesi e i grafici relativi al confronto tra SVM e RF che giustificano, insieme ai dati sperimentali, la scelta finale dell'adozione della Random Forest come classificatore.

Random Forest

- **ROC-AUC = 0.975** : le feature dall'AE separano bene sane vs malate.
- **Soglia scelta** (F2-oriented, validazione): $\tau = 0.1601$
- **Recall (malate)**: 0.892
- **Precision (malate)**: 0.568
- F_2 **(malate)**: 0.801
- **Accuracy**: 0.882

SVM

- **ROC-AUC = 0.500** : equivalente a tirare una moneta (le score non ordinano le classi).
- **Recall = 1.0, ma Accuracy = 0.233 e Precision = 0.233** : praticamente inutilizzabile.

8.4.1 Metriche di ricostruzione

Per ogni immagine, l'autoencoder produce una ricostruzione della foglia di partenza; la differenza tra input e output viene analizzata per estrarre una serie di metriche di errore che descrivono la qualità della ricostruzione. Tra queste figurano l'errore quadratico medio (MSE), l'errore assoluto medio (MAE) e il Peak Signal-to-Noise Ratio (PSNR), che quantificano rispettivamente l'ampiezza e la coerenza globale dell'errore. Sono stati inoltre calcolati diversi quantili dell'errore utili a caratterizzare la distribuzione dell'anomalia e a cogliere variazioni locali significative. Ulteriori indicatori, come il numero e la percentuale di pixel oltre determinate soglie e la varianza del Laplaciano dell'errore, consentono di misurare rispettivamente l'estensione spaziale e la complessità strutturale delle differenze. Queste metriche costituiscono il vettore di feature utilizzato per l'addestramento dei classificatori, fornendo una rappresentazione numerica della "distanza" tra la ricostruzione dell'autoencoder e l'immagine originale.

8.4.2 Metriche di classificazione

Le prestazioni dei modelli di classificazione (Random Forest e SVM) sono state valutate attraverso un insieme di metriche standard, volte a misurare l'accuratezza e l'affidabilità del processo decisionale. In particolare, sono stati considerati **accuratezza**, **precisione**, **richiamo (recall)**, F_1**-score**, F_2**-score** e **AUC** (Area Under the ROC Curve). L'accuratezza indica la proporzione di predizioni corrette, mentre precisione e richiamo quantificano, rispettivamente, la purezza e la completezza della classe positiva (foglie malate). Gli indici F_1 e F_2 combinano i due aspetti in una singola misura bilanciata, attribuendo nel caso dell'F_2 un peso maggiore al richiamo, più adatto in scenari dove è prioritario evitare falsi negativi. Infine, l'AUC valuta la capacità del modello di discriminare correttamente le classi indipendentemente dalla soglia di decisione, offrendo una misura complessiva della qualità del classificatore.

Curva ROC – RF vs SVM

La curva ROC in Figura 8.9 mostra l'andamento del True Positive Rate rispetto al False Positive Rate per entrambi i classificatori. La Random Forest evidenzia un'eccellente capacità discriminativa con un'area sotto la curva (AUC) pari a 0.914, mentre la SVM ottiene un valore di 0.50, equivalente a una classificazione casuale. Ciò conferma che il modello RF riesce a sfruttare efficacemente le feature derivate dell'autoencoder, separando nettamente foglie sane e malate.

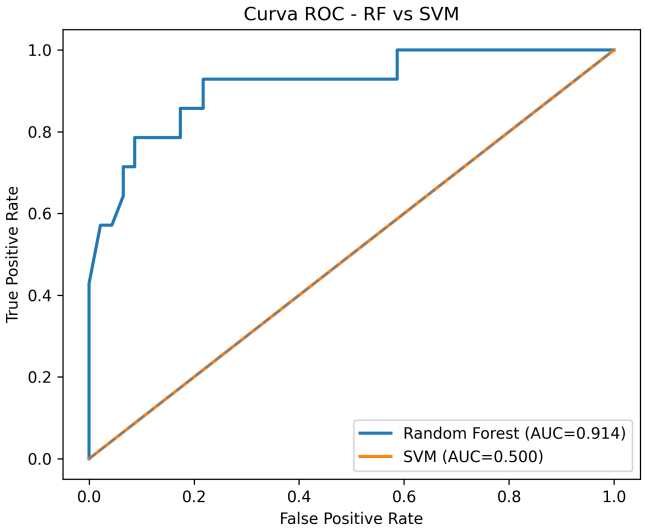

Figura 8.9: Curva ROC dei modelli Random Forest e SVM.

Curva Precision–Recall

La curva Precision–Recall (Figura 8.10) evidenzia ulteriormente la superiorità della Random Forest, che mantiene una precisione elevata anche per alti valori di recall. La SVM, invece, mostra un comportamento degenerato, con una precisione molto bassa nonostante il recall unitario. Questa differenza riflette il fatto che la SVM classifica praticamente tutte le immagini come malate, generando numerosi falsi positivi.

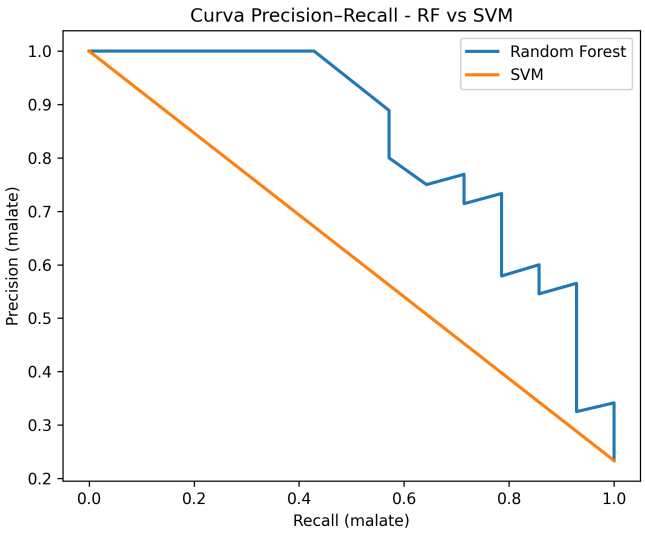

Figura 8.10: Curva Precision–Recall per i due classificatori.

Matrice di confusione RF

Le matrici di confusione (Figura 8.11) mostrano in modo chiaro le prestazioni sui dati di test. La Random Forest ha ottenuto 13 veri positivi e solo un falso negativo, mantenendo un'elevata sensibilità (recall = 0.93). I 10 falsi positivi rappresentano un compromesso accettabile in ottica di diagnosi precoce. La SVM, invece, ha previsto tutte le foglie come malate, con 46 falsi positivi e nessun vero negativo, riducendo l'accuratezza complessiva a 0.23.

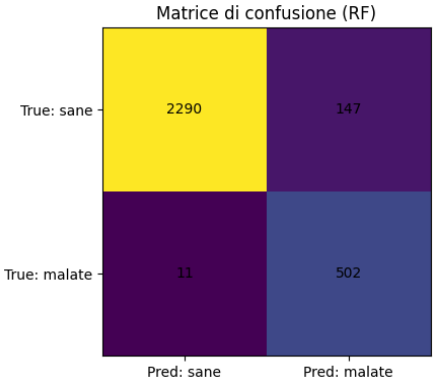

Figura 8.11: Matrice di confusione per RF (a sinistra) e SVM (a destra)

Curva F2 vs soglia RF

La curva F2 in funzione della soglia di decisione (Figura 8.12) consente di visualizzare la stabilità del modello RF rispetto al criterio di classificazione. Il massimo F_2 è pari a **0.801** ed è raggiunto per una soglia $\tau = 0.1601$, che rappresenta il miglior compromesso fra richiamo e precisione per la classe "malata". La forma regolare della curva conferma la robustezza del modello a variazioni moderate della soglia.

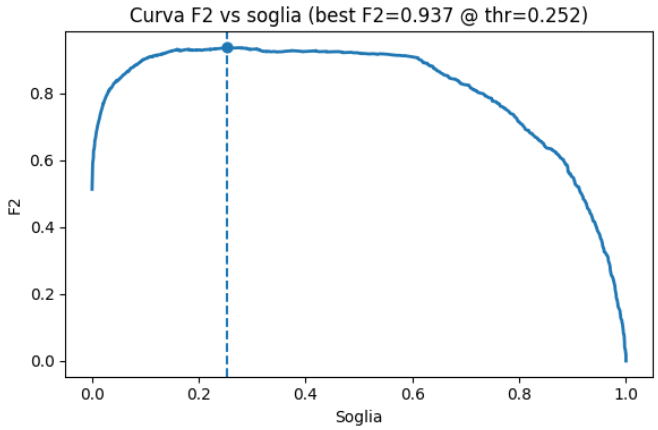

Figura 8.12: Andamento dell'indice F_2 al variare della soglia di decisione per la Random Forest.

Importanza delle feature

La Figura 8.13 riporta le quindici feature più rilevanti secondo la Random Forest. Le metriche basate sui quantili dell'errore di ricostruzione, il numero di pixel sopra soglia e la varianza del Laplaciano risultano tra le più informative, indicando che le

foglie malate presentano regioni localmente meno uniformi e con errori più elevati nella ricostruzione AE. Ciò conferma la coerenza semantica delle feature rispetto al dominio visivo.

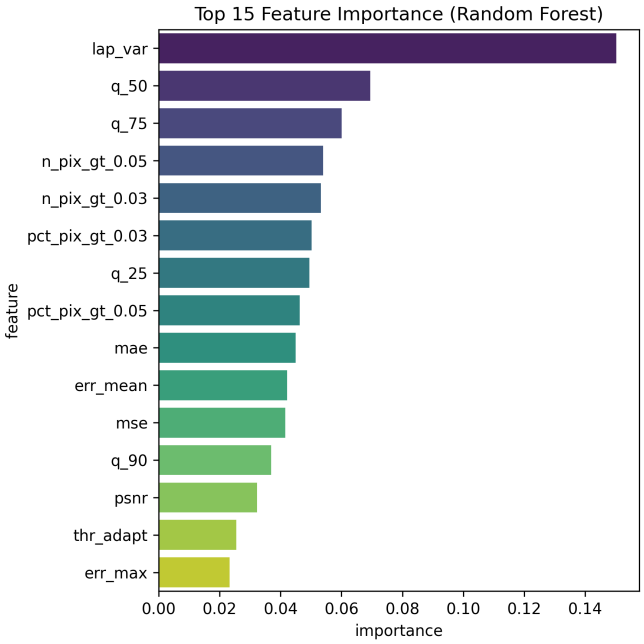

Figura 8.13: Importanza delle feature secondo la Random Forest.

Matrice di correlazione

La heatmap di correlazione (Figura 8.14) evidenzia una forte correlazione tra le metriche di errore globale (MSE, MAE, PSNR) e i quantili inferiori, mentre le feature legate ai pixel sopra soglia e alla varianza spaziale forniscono informazione complementare. Questa struttura ridondante spiega perché la Random Forest, grazie alla selezione implicita di feature, riesca a ottenere risultati migliori rispetto alla SVM.

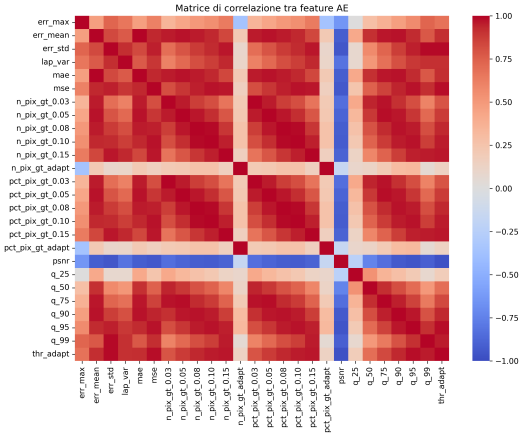

Figura 8.14: Matrice di correlazione tra le feature derivate dall'autoencoder.

Boxplot delle feature principali

I boxplot in Figura 8.15 mostrano la distribuzione delle cinque feature più importanti per ciascuna classe. Le foglie malate presentano sistematicamente valori più elevati di errore medio e quantili superiori dell'errore di ricostruzione, coerenti con la maggiore complessità strutturale dovuta alle lesioni. La separazione visiva fra le due classi testimonia la buona discriminabilità delle feature AE.

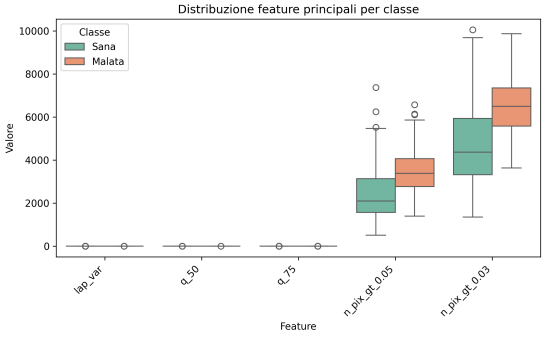

Figura 8.15: Distribuzione delle principali feature per le due classi.

sectionPCA 2D – Distribuzione delle classi

La proiezione PCA in due dimensioni (Figura 8.16) mostra come le feature derivate dell'autoencoder consentano di separare parzialmente le due classi anche in uno spazio ridotto. Le osservazioni malate tendono a concentrarsi in regioni specifiche del

piano, mentre le sane risultano più compatte, a conferma della diversa distribuzione dell'errore di ricostruzione.

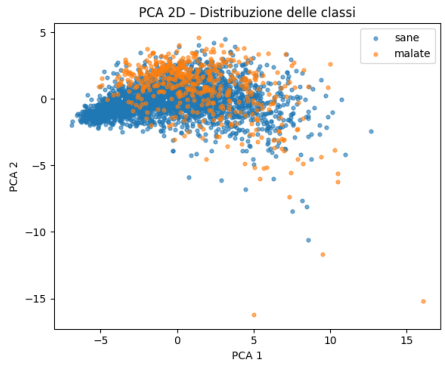

Figura 8.16: Proiezione bidimensionale delle feature tramite PCA.

Isolinee di probabilità RF (spazio PCA)

In Figura 8.17, le isolinee di probabilità prodotte dalla Random Forest nello spazio PCA mostrano frontiere di decisione regolari e coerenti con la distribuzione delle classi. Le regioni ad alta probabilità di "malata" coincidono con le aree densamente popolate da campioni positivi, indicando che il modello ha appreso una separazione stabile anche in bassa dimensionalità.

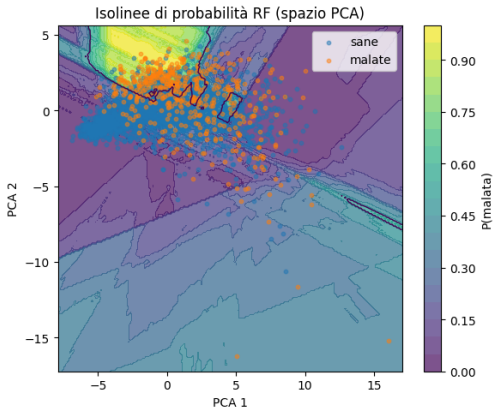

Figura 8.17: Isolinee di probabilità prodotte dalla Random Forest nello spazio PCA.

8.5 Esportazione dei modelli da `.pth` a `.onnx`

La fase di esportazione dei modelli addestrati in formato PyTorch (`.pth`) verso il formato `ONNX` ha rappresentato un passaggio cruciale per la successiva fase di *deployment* su dispositivi embedded, in particolare sulla **Jetson Nano Developer Kit** da 4 GB. Il formato ONNX (*Open Neural Network Exchange*) costituisce infatti uno standard aperto che consente l'interoperabilità tra framework di addestramento e ambienti di inferenza ottimizzati, come TensorRT, ONNX Runtime o OpenVINO.

L'esportazione in ONNX ha permesso di verificare la correttezza delle architetture addestrate e di separare in modo pulito le fasi di training e di inferenza.

8.5.1 Procedura di esportazione

La conversione è stata effettuata a partire dai pesi addestrati in formato `.pth` mediante la funzione `torch.onnx.export`, specificando:

- il modello PyTorch in modalità `eval()`;
- un input fittizio con la dimensione corretta;
- il percorso di destinazione del file `.onnx`;
- l'opset desiderato (nel nostro caso `opset_version = 13`);
- l'opzione `dynamic_axes=False` per generare modelli a dimensione fissa, più semplici da ottimizzare su dispositivi embedded.

8.5.2 Difficoltà riscontrate e soluzioni adottate

Durante la fase di esportazione e successiva ottimizzazione, si sono presentate diverse criticità legate al tipo di modello e al supporto dei diversi operatori da parte di TensorRT.

Modello YOLOv8-seg Il modello di segmentazione YOLOv8-seg ha richiesto una particolare attenzione. L'esportazione diretta tramite il comando:

```
yolo export model=best.pt format=onnx opset=12 simplify=True
```

produce un file formalmente corretto ma non compatibile con TensorRT su Jetson Nano. In particolare, il parser segnalava errori dovuti alla presenza di tensori `INT64` e di operatori non supportati o ridondanti. La soluzione è stata duplice:

1. **Conversione dei tensori INT64 in INT32**, sostituendo in modo sistematico ogni inizializzatore, `Cast` o `Constant` con tipo intero a 32 bit;

2. **Semplificazione del grafo** mediante il pacchetto `onnxsim`, ottenendo un file finale denominato `_int32_sim.onnx`.

Solo questa versione semplificata e con casting corretto è risultata compatibile con TensorRT 8.2 su Jetson Nano, consentendo la generazione dell'engine (`.engine`) in circa 40 s e l'esecuzione dell'inferenza con accelerazione GPU.

Modello Autoencoder Per l'autoencoder, la situazione si è rivelata più semplice ma concettualmente analoga. Anche in questo caso il modello PyTorch esportato presentava alcune costanti interne in formato `INT64`, non accettate da TensorRT. È stata pertanto applicata una conversione automatica `da INT64 a INT32` su pesi, costanti e tipi di I/O, verificando infine il modello con:

```
onnx.checker.check_model()
```

Il risultato, salvato come `ae_int32.onnx`, è stato direttamente compilabile in TensorRT senza ulteriori semplificazioni. L'engine FP16 risultante (`ae_fp16.engine`) è stato impiegato con successo per l'inferenza e la generazione delle mappe d'errore in tempo reale su Jetson Nano.

Classificatore Random Forest Diversamente dai modelli di deep learning, il classificatore Random Forest è stato addestrato con `scikit-learn` e successivamente esportato tramite la libreria `skl2onnx`. In questo caso, non sono state necessarie modifiche aggiuntive: TensorRT non è coinvolto e l'inferenza è stata eseguita tramite `onnxruntime`, che supporta nativamente gli operatori `TreeEnsembleClassifier`. Il file `RF_TOP20_v1.onnx` ha pertanto mantenuto la compatibilità diretta e la correttezza dei risultati, riproducendo in ambiente embedded le stesse predizioni ottenute in fase di validazione su workstation.

8.5.3 Verifica di compatibilità e test su Jetson Nano

Ogni modello ONNX è stato analizzato preliminarmente con lo strumento `trtexec` fornito da NVIDIA, per verificarne la compatibilità con TensorRT e la corretta definizione delle dimensioni d'ingresso.

Solo dopo il superamento della fase di parsing è stato possibile procedere con la generazione dell'engine ottimizzato e con la successiva inferenza in Jupyter Lab sul container Docker ML fornito da NVIDIA (`l4t-ml:r32.7.1-py3`).

8.5.4 Performance su Jetson

Le performance su Jetson si sono rivelate all'altezza delle aspettative: Il modello YOLOv8s, ottimizzato in formato .onnx, rappresenta il collo di bottiglia della

pipeline per quanto riguarda l'efficienza temporale. Ha performato a una media di 8750,3 ms per immagine producendo una media di circa 15 crops per immagine. L'Autoencoder è riuscito a ricostruire i crop a una media di 357,13 ms con FPS per-crop stimati a circa 28.00. La random forest ha classificato i crop a una velocità media di 2,365 ms per crop, rispettando le aspettative.

8.5.5 Sintesi

La Tabella 8.2 riassume le principali differenze tra i modelli e i passaggi necessari per renderli eseguibili su Jetson Nano.

Modello	**Problemi riscontrati**	**Soluzione adottata**	**File finale**
YOLOv8 Segmentation	Tensori INT64, operatori non supportati	Conversione INT32 + Semplificazione (`onnxsim`)	`yolo_int32_sim.onnx`
Autoencoder	Costanti INT64 nel grafo	Conversione INT32	`ae_int32.onnx`
Random Forest	Ricalibrazione soglia decisionale	Export diretto da `skl2onnx`	`RF_TOP20_v1.onnx`

Tabella 8.2: Riepilogo delle conversioni dei modelli in formato ONNX.

In conclusione, l'esportazione in formato ONNX ha permesso di unificare la pipeline di inferenza, rendendo possibile l'esecuzione coordinata dei modelli (da Autoencoder a estrazione feature alla Random Forest) direttamente su Jetson Nano, garantendo al contempo tempi di esecuzione contenuti e compatibilità con le librerie NVIDIA.

Capitolo 9

Conclusione

L'agricoltura di precisione si sta progressivamente orientando verso l'integrazione di tecnologie sempre più raffinate ed efficienti, con l'obiettivo di ridurre l'intervento umano specializzato e automatizzare i processi di monitoraggio e gestione delle colture. In particolare, nel contesto della viticoltura di precisione, una delle sfide più rilevanti riguarda l'individuazione tempestiva di malattie e patologie fogliari che possono compromettere la salute della pianta e la qualità del raccolto. Le tecniche tradizionali di rilevamento si basano spesso su sensoristica multispettrale o iperspettrale, costosa e complessa da implementare su larga scala. L'introduzione di metodologie basate su reti neurali e apprendimento automatico rappresenta quindi una valida alternativa, capace di fornire soluzioni più accessibili, adattabili e autonome. La tesi ha avuto come principale obiettivo l'ideazione e lo sviluppo di una pipeline di analisi automatica per l'individuazione di foglie di vite affette da patologie, utilizzando un approccio di apprendimento prevalentemente non supervisionato. Il nucleo del sistema è costituito da un autoencoder convoluzionale, addestrato a ricostruire immagini di foglie sane: la comparazione tra l'immagine originale e la ricostruzione permette di evidenziare le aree in cui la ricostruzione fallisce, corrispondenti a possibili lesioni o sintomi di malattia. Questa componente viene preceduta da un modulo di segmentazione YOLOv8, che isola automaticamente le foglie all'interno delle immagini, e seguita da un classificatore Random Forest, che valuta le metriche di ricostruzione e stima la probabilità che la foglia sia malata o sana. L'integrazione di questi elementi — noti singolarmente in letteratura, ma mai combinati in questa forma — ha portato alla realizzazione di una pipeline originale, modulare e facilmente riadattabile ad altri contesti agricoli.

Un ulteriore obiettivo del lavoro è stato il porting dell'intera pipeline su dispositivi edge, caratterizzati da risorse computazionali limitate ma ideali per applicazioni di campo in tempo reale. In particolare, è stato scelto di utilizzare la Jetson Nano Developer Kit, una piattaforma NVIDIA basata su GPU, che consente di eseguire modelli di deep learning in modo efficiente grazie al supporto di TensorRT e al

formato standardizzato ONNX. Tutti i modelli addestrati in PyTorch sono stati convertiti in ONNX per garantire interoperabilità e ottimizzazione hardware:

- per YOLOv8 è stato necessario produrre una versione semplificata e compatibile con TensorRT, a causa della presenza di operatori non supportati e tensori in formato INT64;
- per l'autoencoder è stata sufficiente la conversione dei tensori INT64 in INT32;
- per la Random Forest, invece, è stato possibile esportare direttamente il modello tramite la libreria skl2onnx, senza ulteriori modifiche.

I risultati sperimentali ottenuti sono stati pienamente soddisfacenti: la pipeline eseguita su Jetson Nano si è dimostrata stabile e reattiva, in grado di elaborare immagini in modo fluido e con tempi di inferenza compatibili con un utilizzo operativo in campo. L'inferenza del modello YOLOv8 (particolarmente esigente in termini di memoria) e la ricostruzione dell'autoencoder risultano coerenti sia dal punto di vista qualitativo — con ricostruzioni visivamente accurate — sia da quello quantitativo, grazie alla capacità della Random Forest di discriminare correttamente le foglie sane da quelle malate. L'enfasi è stata posta in particolare sulla massimizzazione del recall per la classe "malata": è preferibile classificare come malata una foglia sana, piuttosto che rischiare di non rilevare una patologia effettiva. Questo approccio, orientato alla sicurezza e alla prevenzione, è particolarmente indicato per sistemi di supporto decisionale in agricoltura.

Gli sviluppi futuri del progetto sono molteplici. Poiché la componente di ricostruzione si basa su un modello di apprendimento non supervisionato, la pipeline può essere facilmente adattata ad altre colture vegetali semplicemente sostituendo il dataset di partenza. L'unico elemento da riaddestrare rimane il modello di segmentazione, che deve essere adattato alla nuova morfologia fogliare. Inoltre, la natura modulare del sistema consente di integrare componenti di robotica agricola o dispositivi mobili, come rover e droni, per realizzare applicazioni di monitoraggio distribuito on the edge. In questo modo, l'analisi e la diagnosi avverrebbero direttamente sul dispositivo di raccolta dati, senza la necessità di trasmissione continua verso server remoti o infrastrutture cloud, con un significativo risparmio di banda e una maggiore autonomia operativa.

Un'ulteriore prospettiva di ricerca consiste nella semplificazione e miniaturizzazione dei modelli, per rendere possibile la loro esecuzione su piattaforme ancora più leggere, in ottica TinyML. Questo consentirebbe di estendere il sistema anche a microcontrollori o sensori a basso consumo energetico, favorendo la diffusione capillare della tecnologia in ambito agricolo. Nonostante gli ottimi risultati ottenuti, resta spazio per migliorare ulteriormente la qualità dei dataset di addestramento, aumentandone la varietà e la rappresentatività, nonché per esplorare strategie di ottimizzazione dei modelli che bilancino meglio accuratezza e tempi di inferenza.

In conclusione, il lavoro svolto dimostra la possibilità concreta di realizzare una pipeline di diagnosi fitosanitaria completamente automatizzata, economica e scalabile, in grado di operare in tempo reale su dispositivi edge. Questa soluzione rappresenta un passo avanti verso l'applicazione pratica dell'intelligenza artificiale nell'agricoltura di precisione, con l'obiettivo di migliorare l'efficienza, la sostenibilità e la tempestività degli interventi in campo.

Bibliografia

[1] Zinon Zinonos, Socratis Gkelios, Ala Khalifeh, Diofantos Hadjimitsis, Yiannis Boutalis e Savvas Chatzichristofis. «Grape Leaf Diseases Identification System Using Convolutional Neural Networks and LoRa Technology». In: *IEEE Access* PP (dic. 2021), pp. 1–1. DOI: 10.1109/ACCESS.2021.3138050 (cit. alle pp. 2, 4).

[2] Food and Agriculture Organization of the United Nations. *Early warning systems for plant health*. Accessed: 2025-10-08. 2024. URL: https://www.fao.org/one-health/highlights/early-warning-systems-for-plant-health/en (cit. a p. 3).

[3] Food and Agriculture Organization of the United Nations. *Plant Production and Protection Division — Key facts*. "Up to 40% of global crop production is lost annually to plant pests and diseases." Accessed: 2025-10-08. 2023. URL: https://openknowledge.fao.org/server/api/core/bitstreams/f700bc9e-bdda-4818-9d29-a295d0aad0f6/content (cit. a p. 3).

[4] Jean B. Ristaino et al. «The persistent threat of emerging plant disease pandemics to global food security». In: *Proceedings of the National Academy of Sciences (PNAS)* 118.23 (2021). URL: https://www.pnas.org/doi/10.1073/pnas.2022239118 (cit. a p. 3).

[5] Jayme Garcia Arnal Barbedo. «A review on the main challenges in automatic plant disease identification based on visible light images». In: *Computers and Electronics in Agriculture* 142 (2016), pp. 52–64. DOI: 10.1016/j.compag.2016.06.001 (cit. a p. 3).

[6] Anne-Katrin Mahlein. «Plant Disease Detection by Imaging Sensors—Parallels and Specific Demands». In: *Plant Disease* 100.2 (2016), pp. 241–251. DOI: 10.1094/PDIS-03-15-0340-FE (cit. a p. 3).

[7] U.S. Government Accountability Office. *Precision Agriculture: Technologies and Challenges*. GAO-22-104438. Discusses costs and infrastructure barriers to adoption of precision agriculture technologies. U.S. GAO, 2022. URL: https://www.gao.gov/ (cit. a p. 3).

[8] S.M. Jaisakthi, P. Mirunalini, D. Thenmozhi e Vatsala. «Grape Leaf Disease Identification using Machine Learning Techniques». In: *2019 International Conference on Computational Intelligence in Data Science (ICCIDS)*. 2019, pp. 1–6. DOI: 10.1109/ICCIDS.2019.8862084 (cit. a p. 3).

[9] Parul Sharma, Yash Berwal e Wiqas Ghai. «Performance Analysis of Deep Learning CNN Models for Disease Detection in Plants using Image Segmentation». In: *Information Processing in Agriculture* 7 (nov. 2019). DOI: 10.1016/j.inpa.2019.11.001 (cit. a p. 4).

[10] Li Liu, Wanli Ouyang, Xiaogang Wang, Paul Fieguth, Jie Chen, Xin Liu e Matti Pietikäinen. «Deep Learning for Generic Object Detection: A Survey». In: *Pattern Recognition* 108 (2020). Panoramica su one-stage (es. YOLO) e two-stage (es. R-CNN/Mask R-CNN), con discussione su trade-off accuratezza/velocità., p. 107334. DOI: 10.1016/j.patcog.2019.107334 (cit. a p. 4).

[11] Guansong Pang, Chunhua Shen, Longbing Cao e Anton van den Hengel. «Deep Learning for Anomaly Detection: A Review». In: *ACM Computing Surveys* 54.2 (2021). Ampia rassegna su metodi di anomaly detection deep, inclusi autoencoder e ricostruzione., 38:1–38:38. DOI: 10.1145/3439950 (cit. a p. 4).

[12] Jordao Silva, Thommas Flores, Silvan Júnior e Ivanovitch Silva. «TinyML-Based Pothole Detection: A Comparative Analysis of YOLO and FOMO Model Performance». In: ott. 2023, pp. 1–6. DOI: 10.1109/LA-CCI58595.2023.10409357 (cit. a p. 5).

[13] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 2ª ed. Sebastopol, CA: O'Reilly Media, 2019 (cit. alle pp. 6, 7, 9, 10, 18–22, 24, 29–35, 37, 40, 42).

[14] Arthur L. Samuel. «Some Studies in Machine Learning Using the Game of Checkers». In: *IBM Journal of Research and Development* 3.3 (1959), pp. 210–229. DOI: 10.1147/rd.33.0210 (cit. a p. 6).

[15] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. New York: Springer, 2006. ISBN: 978-0387310732 (cit. alle pp. 7, 8).

[16] Richard S. Sutton e Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2ª ed. Cambridge, MA: MIT Press, 2018. ISBN: 978-0262039246. URL: http://incompleteideas.net/book/the-book-2nd.html (cit. a p. 8).

[17] J. Ross Quinlan. «Induction of Decision Trees». In: *Machine Learning* 1.1 (1986), pp. 81–106 (cit. a p. 13).

[18] Leo Breiman, Jerome H. Friedman, Richard A. Olshen e Charles J. Stone. *Classification and Regression Trees*. Belmont, CA: Wadsworth International Group, 1984. ISBN: 978-0412048418 (cit. a p. 14).

[19] Leo Breiman. «Random Forests». In: *Machine Learning* 45.1 (2001), pp. 5–32 (cit. alle pp. 14, 15).

[20] Anil K. Jain, M. Narasimha Murty e Patrick J. Flynn. «Data Clustering: A Review». In: *ACM Computing Surveys* 31.3 (1999), pp. 264–323. DOI: 10.1145/331499.331504 (cit. a p. 16).

[21] Walter Pitts Warren S. McCulloch. «A Logical Calculus of Ideas Immanent in Nervous Activity». In: *The Bulletin of Mathematical Biology* 5.4 (1943), pp. 115–113 (cit. alle pp. 18, 20).

[22] Frank Rosenblatt. *The Perceptron: A Perceiving and Recognizing Automaton*. Rapp. tecn. 85-460-1. Technical Report. Buffalo, NY: Cornell Aeronautical Laboratory, 1957 (cit. a p. 20).

[23] Marvin Minsky e Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: MIT Press, 1969 (cit. a p. 22).

[24] Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*. Cambridge, MA: MIT Press, 2016. URL: https://www.deeplearningbook.org/ (cit. a p. 22).

[25] Kurt Hornik. «Multilayer Feedforward Networks are Universal Approximators». In: *Neural Networks* 2.5 (1989), pp. 359–366. DOI: 10.1016/0893-6080(89)90020-8 (cit. a p. 22).

[26] David Rumelhart et al. «Laerning Internal Representation by Error Propagation». In: *Defense Technical Information Center technicl report* (1985) (cit. a p. 23).

[27] James Bergstra e Yoshua Bengio. «Random Search for Hyper-Parameter Optimization». In: *Journal of Machine Learning Research* 13 (2012). Mostra l'efficacia della ricerca casuale rispetto alla griglia nel tuning degli iperparametri, pp. 281–305. URL: http://jmlr.org/papers/v13/bergstra12a.html (cit. a p. 25).

[28] David H. Hubel e Torsten N. Wiesel. «Receptive fields, binocular interaction and functional architecture in the cat's visual cortex». In: *The Journal of Physiology* 160.1 (1962), pp. 106–154. DOI: 10.1113/jphysiol.1962.sp006837 (cit. a p. 27).

[29] Kunihiko Fukushima. «Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position». In: *Biological Cybernetics* 36.4 (1980), pp. 193–202. DOI: 10.1007/BF00344251 (cit. a p. 28).

[30] Yann LeCun, Léon Bottou, Yoshua Bengio e Patrick Haffner. «Gradient-Based Learning Applied to Document Recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791 (cit. alle pp. 28, 34).

[31] Alex Krizhevsky, Ilya Sutskever e Geoffrey E. Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 25. 2012, pp. 1097–1105. URL: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html (cit. a p. 35).

[32] Jonathan Long, Evan Shelhamer e Trevor Darrell. «Fully Convolutional Networks for Semantic Segmentation». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 3431–3440. DOI: 10.1109/CVPR.2015.7298965. URL: https://openaccess.thecvf.com/content_cvpr_2015/html/Long_Fully_Convolutional_Networks_2015_CVPR_paper.html (cit. a p. 36).

[33] Joseph Redmon, Santosh Divvala, Ross Girshick e Ali Farhadi. «You Only Look Once: Unified, Real-Time Object Detection». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91 (cit. a p. 38).

[34] Joseph Redmon e Ali Farhadi. «YOLO9000: Better, Faster, Stronger». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 7263–7271. DOI: 10.1109/CVPR.2017.690 (cit. a p. 38).

[35] Joseph Redmon e Ali Farhadi. *YOLOv3: An Incremental Improvement*. 2018. arXiv: 1804.02767 [cs.CV]. URL: https://arxiv.org/abs/1804.02767 (cit. a p. 38).

[36] Olaf Ronneberger, Philipp Fischer e Thomas Brox. «U-Net: Convolutional Networks for Biomedical Image Segmentation». In: *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Gli autori evidenziano la perdita di risoluzione dovuta al downsampling e propongono skip connections per unire contesto e localizzazione fine. Springer, 2015, pp. 234–241. DOI: 10.1007/978-3-319-24574-4_28. URL: https://arxiv.org/abs/1505.04597 (cit. a p. 39).

[37] William G. Chase e Herbert A. Simon. «Perception in Chess». In: *Cognitive Psychology* 4.1 (1973), pp. 55–81. DOI: 10.1016/0010-0285(73)90004-2 (cit. a p. 41).

[38] William G. Chase e Herbert A. Simon. «The Mind's Eye in Chess». In: *Visual Information Processing*. A cura di William G. Chase. New York: Academic Press, 1973, pp. 215–281 (cit. a p. 41).

[39] Jonathan Masci, Ueli Meier, Dan Cireşan e Jürgen Schmidhuber. «Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction». In: *Artificial Neural Networks and Machine Learning – ICANN 2011*. Vol. 6791. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 52–59. DOI: 10.1007/978-3-642-21735-7_7. URL: https://doi.org/10.1007/978-3-642-21735-7_7 (cit. a p. 43).

[40] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li e Lanyu Xu. «Edge Computing: Vision and Challenges». In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198. URL: https://cse.buffalo.edu/faculty/tkosar/cse710_spring20/shi-iot16.pdf (cit. a p. 44).

[41] ONNX Runtime Team. *ONNX Runtime Documentation*. https://onnxruntime.ai/docs/. Cross-platform ML inference engine; execution providers per CPU/GPU/NPUs. 2025. (Visitato il giorno 09/10/2025) (cit. a p. 45).

[42] ONNX Community. *ONNX Runtime: Cross-platform, high performance scoring engine for ML models*. https://onnxruntime.ai/. Accessed: 2025-09-29. 2023 (cit. alle pp. 45, 53).

[43] *NVIDIA TensorRT Developer Guide*. Optimizations for inference (layer fusion, FP16/INT8, profiling); version 8.x. NVIDIA. 2024. URL: https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-861/developer-guide/index.html (visitato il giorno 09/10/2025) (cit. a p. 46).

[44] Anurag Ghosh. *SALT: Segment Anything Labelling Tool*. GitHub repository. Accessed: 2025-10-03. 2023. URL: https://github.com/anuragxel/salt (cit. a p. 58).

[45] NVIDIA. *NVIDIA L4T ML: Machine Learning Container for Jetson*. NVIDIA NGC Catalog. Image tag utilizzata: nvcr.io/nvidia/l4t-ml:r32.7.1-py3. Accesso: 2025-10-03. 2022. URL: https://catalog.ngc.nvidia.com/orgs/nvidia/containers/l4t-ml (cit. a p. 67).