



SAPIENZA  
UNIVERSITÀ DI ROMA

## Approccio basato su Deep Learning per la Segmentazione Semantica di Immagini Aeree

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea in Informatica

Candidato

William De Vena

Matricola 1862820

Relatore

Prof. Danilo Avola

Correlatori

Prof. Luigi Cinque

Dott. Alessio Fagioli

Anno Accademico 2021-2022

---

**Approccio basato su Deep Learning per la Segmentazione Semantica di Immagini Aeree**

Tesi di Laurea. Sapienza – Università di Roma

© 2022 William De Vena. Tutti i diritti riservati

Questa tesi è stata composta con L<sup>A</sup>T<sub>E</sub>X e la classe Sapthesis.

Email dell'autore: william98wdv@gmail.com

# Indice

<b>1 Introduzione</b>	<b>4</b>
1.1 Contesto di applicazione . . . . .	4
1.2 Stato dell'arte . . . . .	6
1.3 Contributo ed Outline . . . . .	11
<b>2 Segmentazione di immagini</b>	<b>12</b>
2.1 La segmentazione di immagini negli umani . . . . .	15
2.2 Approcci classici . . . . .	16
2.2.1 Metodi a soglia . . . . .	16
2.2.2 Metodi dividi e fondi . . . . .	17
2.2.3 Metodi basati su grafi . . . . .	17
2.2.4 Metodo dei contorni attivi . . . . .	19
2.2.5 Metodi di clustering . . . . .	22
2.2.6 Algoritmi Supervised . . . . .	27
2.3 Approcci di Deep Learning . . . . .	27
2.3.1 Architetture basate sul contesto . . . . .	27
2.3.2 Architetture basate sul feature-enhacement . . . . .	28
2.3.3 Architetture basate sulla deconvoluzione . . . . .	28
2.3.4 Architetture basate sulle GAN . . . . .	29
<b>3 Deep Learning</b>	<b>31</b>
3.1 Percettrone . . . . .	32
3.2 Percettrone Multistrato . . . . .	33
3.3 Funzioni di attivazione . . . . .	34
3.3.1 Funzione sigmoidea . . . . .	35
3.3.2 Tangente iperbolica . . . . .	35
3.3.3 Rectified Linear Activation Function . . . . .	35
3.3.4 Softmax . . . . .	36
3.4 Apprendimento . . . . .	37
3.4.1 Paradigmi di apprendimento . . . . .	38
3.4.2 Funzioni di perdita . . . . .	38
3.4.3 Retropropagazione dell'errore . . . . .	39
3.5 Regolarizzazione . . . . .	40
3.5.1 Data augmentation . . . . .	42
3.6 Reti Neurali convoluzionali . . . . .	43
3.6.1 Reti neurali convoluzionali avanzate . . . . .	46

<b>4 Architettura e metodi del lavoro</b>	<b>52</b>
4.1 Difficoltà affrontate nel dataset . . . . .	52
4.2 Pulizia del dataset e Data Augmentation . . . . .	54
4.3 DeepLabV3 . . . . .	57
4.3.1 Architettura totale . . . . .	59
4.3.2 Convoluzione Dilatata . . . . .	60
4.3.3 Reti Neurali Residue . . . . .	61
4.3.4 Atrous Spatial Pyramid Pooling . . . . .	66
<b>5 Esperimenti e Risultati</b>	<b>68</b>
5.1 Risorse computazionali . . . . .	68
5.2 Tecnologie usate . . . . .	69
5.3 Dataset FloodNet . . . . .	69
5.4 Esperimenti . . . . .	70
5.5 Paragone con altri lavori . . . . .	74
<b>6 Conclusioni</b>	<b>77</b>
6.1 Sviluppi futuri . . . . .	78

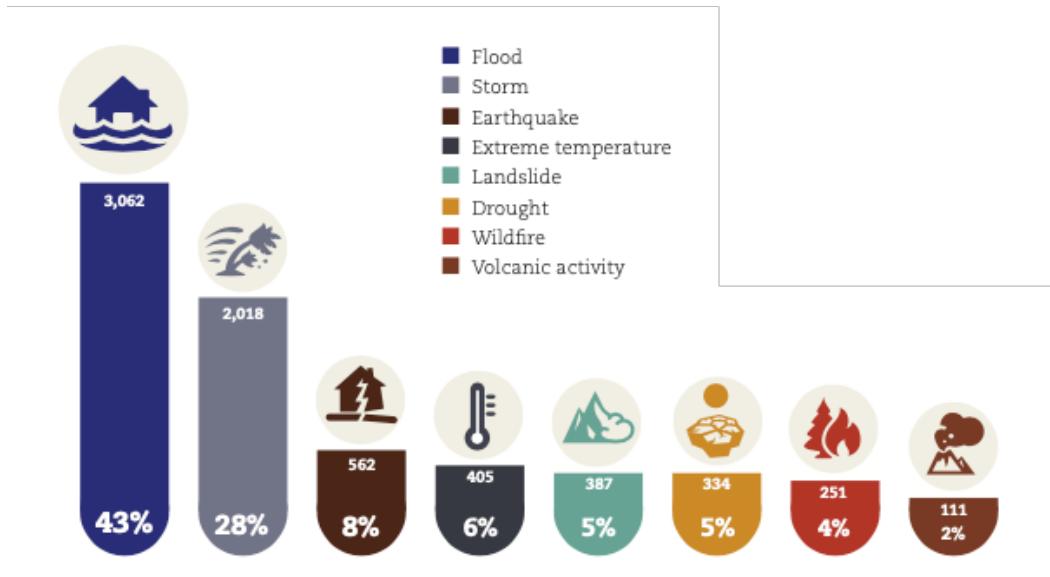
# Capitolo 1

## Introduzione

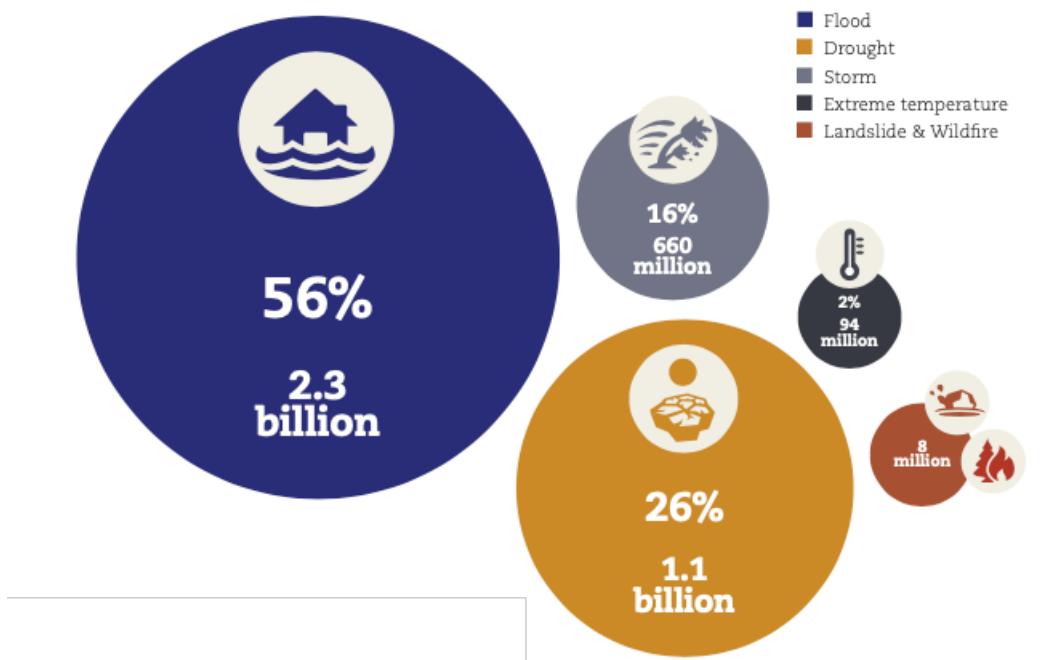
### 1.1 Contesto di applicazione

Nel ventennio dal 1995 al 2015, circa il 90% dei disastri che hanno colpito la popolazione mondiale sono stati causati da fenomeni meteorologici quali inondazioni, tempeste, ondate di caldo ed altri. In questo periodo, l'EM-DAT (Emergency Events Database), uno dei più importanti database di eventi disastrosi mondiali, ha registrato 6457 disastri causati da eventi meteorologici, che hanno causato 606,000 vittime (una media di 30,000 all'anno) e all'incirca 4.1 miliardi di persone totali colpite (tra il 1995 e il 2015) tra feriti, persone lasciate senza una dimora e persone in bisogno di assistenza. Tra questi eventi, le alluvioni sono tra le più frequenti e gravi, infatti costituiscono il 43% del totale degli eventi (Figura 1.1) e hanno causato all'incirca 157,000 morti più 2.3 miliardi di persone colpite (tra il 1995 e il 2015) (Figura 1.2). Le alluvioni sono tra le tipologie più gravi anche dal punto di vista dei danni economici, infatti, solo negli Stati Uniti si sono registrati 662 miliardi di dollari di danni causati da alluvioni (1995-2015) e nel continente europeo all'incirca 262 miliardi (1994-2015). Inoltre, le occorrenze di eventi disastrosi causati da fenomeni meteorologici risultano in crescita, in particolare, nel periodo tra il 2005 e il 2014 si è registrata una media di 335 eventi all'anno, ovvero il 14% in più rispetto al periodo tra il 1995-2004 e quasi il doppio di quelli registrati tra il 1985 e il 1994. Lo stesso trend si può vedere nelle alluvioni, che nello stesso periodo (2005-2014) hanno raggiunto un picco di 171 eventi all'anno, rispetto ai 127 dei decenni precedenti [1].

In questo contesto, risulta essenziale la gestione di questi eventi disastrosi. In particolare, sono fondamentali le tre fasi di gestione delle alluvioni: la fase precedente all'evento, ovvero la fase di prevenzione; la fase durante l'evento stesso, in cui è fondamentale avere conoscenza sulle aree colpite e sul propagarsi dell'alluvione, informazioni che spesso non sono facili da reperire; e infine la fase post-alluvione, in cui si valutano i danni. Negli ultimi anni, nel campo della gestione dei disastri naturali, è risultato molto utile l'impiego di UAV (Unmanned Aerial Vehicle), anche chiamati droni. Uno dei primi utilizzi è stato nel 2005 negli Stati Uniti dove, per la prima volta, i droni sono stati utilizzati dal CRASAR (Center for Robot-Assisted Search and Rescue) dell'Università A&M del Texas per cercare e soccorrere sopravvissuti agli eventi causati dall'Uragano Katrina. Successivamente, nel 2015, il CRASAR ha collaborato con la Measure UAV consulting firm e l'American Red



**Figura 1.1.** Percentuale di eventi di calamità naturali per tipo di disastro (1995-2015) [1].

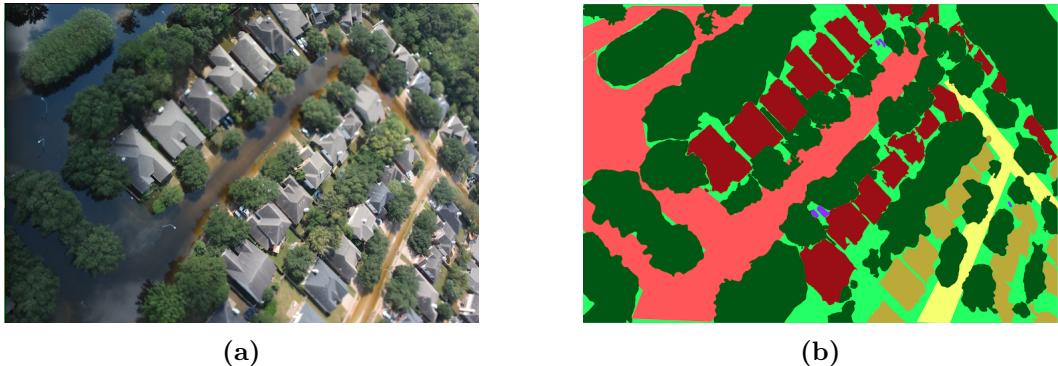


**Figura 1.2.** Número di persone colpite da calamità meteorologiche (1995-2015) (NB: i decessi sono esclusi dal totale delle vittime) [1].

Cross (ARC) per testare le capacità degli UAV in situazioni di emergenza ed eventi disastrosi, il report che ne seguì descrisse i droni come una tecnologia che portò “immediati benefici” sia ai civili che ai soccorritori. Fino ad oggi, i droni sono stati utilizzati in molti eventi di questo tipo, tra i tanti vi sono il terremoto di Wenchuan del 2008, la crisi di Fukushima Daiichi del 2011 e la gestione del disastro causato dal Tifone Haiyan nelle Filippine nel 2013. Infatti ad oggi, molte organizzazioni, principalmente umanitarie, hanno lanciato programmi che hanno come obiettivo quello di integrare gli UAV nelle operazioni di gestione di questi eventi, tra queste: l’WFP (ONU World Food Programme), la Banca Mondiale, l’UNHCR (UN High Commissioner for Refugees). In particolare, i droni hanno trovato utilità in tutte e tre le fasi menzionate precedentemente: nella fase di prevenzione vengono spesso utilizzati per monitorare il letto dei fiumi o altri bacini; durante le alluvioni vengono invece utilizzati per mappare le aree colpite, per operazioni di ricerca e soccorso, per il trasporto di carichi in zone altrimenti non raggiungibili, per fornire informazioni in tempo reale dall’alto sullo stato delle strade per guidare le squadre di soccorso, ma anche per fornire informazioni sugli edifici più colpiti per priorizzare le operazioni; infine, nel post-alluvione per la valutazione dei danni [2]. Uno dei principali vantaggi dei droni risiede nella rapidità del loro impiego e nell’alta risoluzione delle immagini che possono catturare. In particolare, con essi si riesce ad ottenere immagini di più alta risoluzione delle aree colpite ed in tempi più brevi rispetto ad altre risorse, come le immagini satellitari [3]. Inoltre, con l’elaborazione di queste immagini e grazie alla loro alta risoluzione, si riescono ad estrapolare informazioni molto importanti in questo ambito. Ad esempio, tra i vari utilizzi, si possono distinguere gli edifici e le strade allagate rispetto a quelle non allagate, si possono individuare alcuni oggetti di interesse quali persone o veicoli, oppure si può classificare il livello di danni di un edificio. Infatti, questo lavoro riguarda la segmentazione semantica, una delle tipologie di elaborazioni che spesso viene applicata alle immagini, per fornire informazioni molto utili soprattutto nella fase durante l’alluvione [4]. La segmentazione semantica non è altro che la classificazione di ogni singolo pixel dell’immagine in una delle predeterminate classi (Figura 1.3) e si distingue dal task della classificazione, in quanto quest’ultima riguarda l’assegnazione di una singola classe all’intera immagine. In particolare, il modello d’interesse di questo lavoro si occupa dell’assegnazione ad ogni pixel dell’immagine di una delle seguenti 9 classi: edificio allagato, edificio non allagato, strada allagata, strada non allagata, acqua, albero, veicolo e prato. Di queste, risultano di forte interesse nell’ambito della gestione di eventi di alluvione soprattutto le prime quattro classi, in quanto distinguere quali edifici siano allagati da quali no, può risultare fondamentale per guidare e priorizzare le operazioni di soccorso, mentre individuare le strade allagate può invece essere utile per distinguere quali strade siano percorribili via terra.

## 1.2 Stato dell’arte

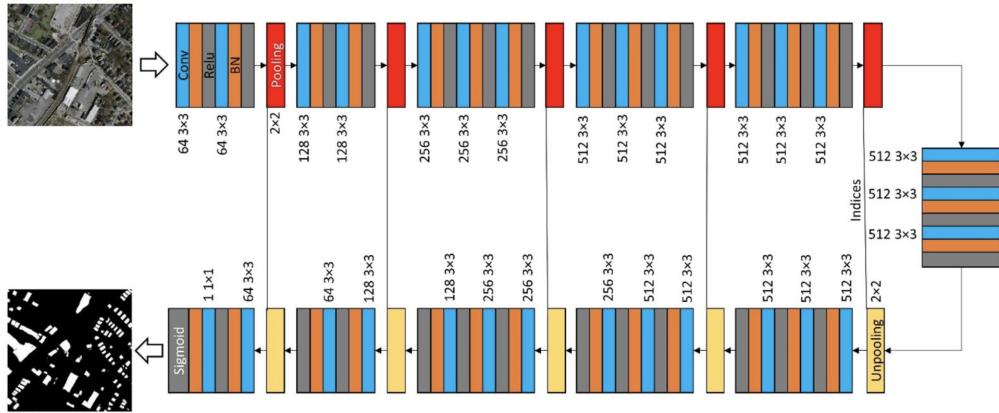
Come vedremo più avanti, le reti neurali rispetto ai metodi più tradizionali riescono a cogliere informazioni di più alto livello. In particolare, la loro complessa architettura gli permette di approssimare pattern molto complessi, che i metodi tradizionali non riescono a cogliere. Infatti, come vedremo in questo paragrafo,



**Figura 1.3.** Un esempio di segmentazione semantica. La (a) mostra l'immagine in input e la (b) l'output del modello, dove ogni colore corrisponde ad una classe: strada allagata (rosso chiaro), edificio allagato (rosso scuro), strada non allagata (giallo chiaro), edificio non allagato (giallo scuro), albero (verde scuro), prato (verde chiaro), veicolo (viola).

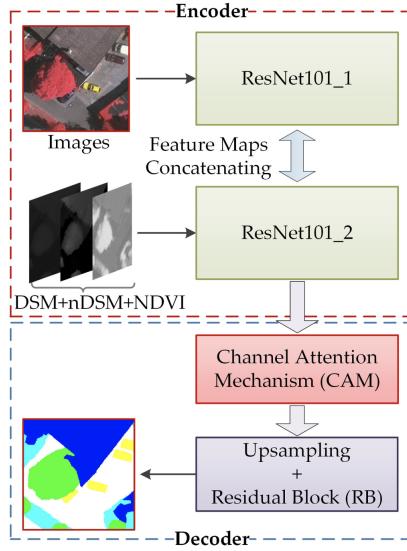
nello stato dell'arte della segmentazione semantica di immagini aeree, la maggior parte delle metodologie utilizzate sono di Deep Learning. Una delle architetture più presenti in letteratura e utilizzate in questo campo è la UNet. In diversi lavori [5, 6, 7] viene utilizzata la versione classica proposta in [8], in altri invece vengono utilizzate delle varianti. Ad esempio, in [9] viene utilizzata la versione con l'aggiunta del meccanismo di attenzione, chiamata AttentionUnet [10]; in [11] viene utilizzata in combinazione con la SegNet [12] per costruire un'architettura di tipo encoder-decoder (Figura 1.4); infine viene ripresa in [13], in cui viene utilizzata la rete MobileNet [14] come backbone, ovvero come la prima parte della UNet (l'encoder) responsabile dell'estrazione delle feature. Ritornando invece su [9], il dataset RescueNet viene utilizzato come benchmark per diversi modelli tra cui, oltre all'AttentionUnet, la PSPNet [15], la DeepLabV3+ [16] e la ENet [17]. In altri lavori invece, vengono utilizzate architetture ensemble composte da più modelli. In particolare, in [18] un ensemble di CNN viene testato sul S-PRS Vaihingen Dataset. In questo caso l'ensemble è costituito da più versioni dello stesso modello, ovvero la stessa architettura addestrata più volte sullo stesso dataset ma con inizializzazioni diverse.

L’idea degli autori è che, essendo lo spazio della loss function in cui si muove il modello estremamente non convesso, a molte dimensioni e con molti minimi locali, inizializzare un modello con parametri diversi significa quasi avere la garanzia che convergerà in punti diversi. Di conseguenza, il loro approccio consiste nell’addestrare queste diverse versioni del modello per poi, in fase di inferenza, prendere la media delle loro predizioni. Un simile approccio viene utilizzato anche in [19], in cui l’ensemble è composto dallo stesso modello ma preso in diverse fasi dell’addestramento. In particolare, gli autori evidenziano come, con questo approccio, sia possibile addestrare un ensemble senza aggiungere ulteriore tempo rispetto ad addestrare un singolo modello. Così come in [9], anche in [20, 21] vengono utilizzati modelli che si basano sul meccanismo di attenzione [22]. In particolare, gli autori di [20] evidenziano come un meccanismo di attenzione ibrido, ovvero l’unione di diverse tipologie di attenzioni (sui canali, sullo spazio e sulle classi) riesca a catturare dipendenze a lungo raggio, aiutando così il modello a produrre delle feature map



**Figura 1.4.** Architettura del modello Seg-Unet proposto in [11] con una combinazione dei componenti della Segnet (pooling indices) e quelli della Unet (skip connection).

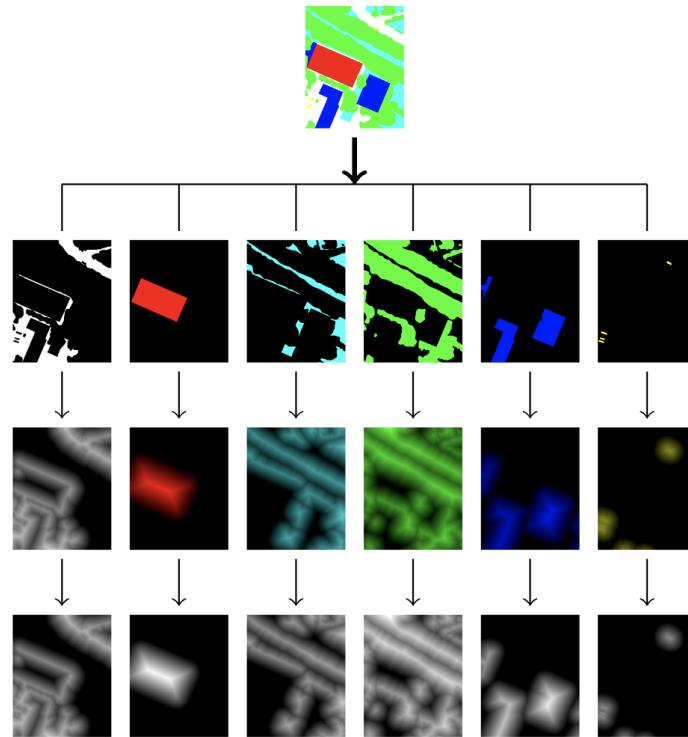
di qualità. Dall'altra parte, in [21] gli autori, oltre a proporre un modello basato sulla struttura di una FCN con l'aggiunta del meccanismo di attenzione sui canali (*channel attention mechanism*), propongono una approccio basato sull'utilizzo di due tipologie di dati corrisposte da due tipologie di backbone. In particolare, la prima parte del modello è composta da due ResNet101, una che prende in input l'immagine e una che invece prende in input dati ausiliari come l'NDVI (Normalized Difference Vegetation Index) ed altri. Nella seconda parte invece, le due feature map vengono concatenate e passate al modulo di *channel attention*, per poi passare infine nella parte di upsample per produrre le maschere finali (Figura 1.5).



**Figura 1.5.** Archittetura del modello basato sull'uso del *channel attention mechanism* proposto in [21].

In [23] le maschere all'interno del dataset, prima di essere utilizzate per ad-

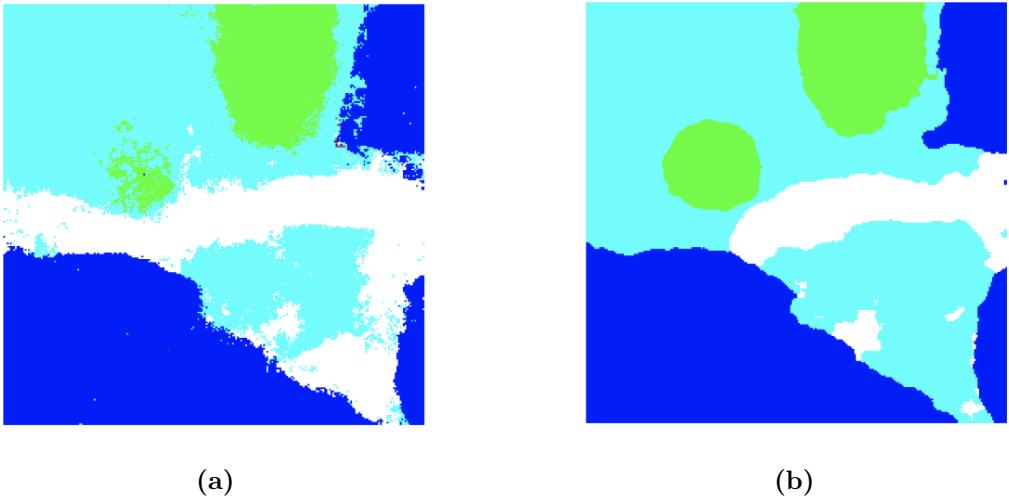
destrare il loro modello, subiscono una prima fase di processing in cui vengono trasformate in delle *distance maps*. In particolare, per ogni maschera e per ogni classe viene prodotta una maschera binaria, i cui valori dei pixel rappresentano la distanza di quel pixel di quella determinata classe dal bordo dell'oggetto di cui fa parte (Figura 1.6).



**Figura 1.6.** Illustrazione della produzione di una *distance map* a partire da una maschera [23].

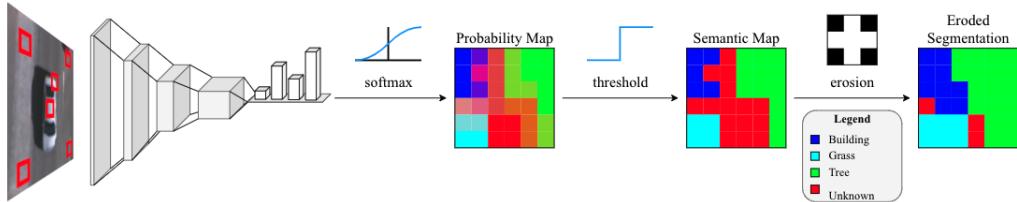
La motivazione per cui utilizzare le distance maps è che, grazie al fatto che codificano informazioni spaziali in più rispetto alle semplici maschere, gli output del modello ottenuto dal loro utilizzo nell'addestramento risultano con meno rumore, con più coerenza spaziale e i bordi degli oggetti appaiono più definiti (Figura 1.7).

Una variante della segmentazione semantica è quella affrontata in [24], ovvero la segmentazione semantica open-set, che consiste nel task della segmentazione semantica, con l'aggiunta del fatto che il numero totale di classi è sconosciuto. Di conseguenza, la segmentazione semantica open-set può essere descritta come il task in cui ogni pixel può essere etichettato come appartenente a una delle classi apprese durante l'addestramento, oppure come una classe sconosciuta. Le difficoltà di questa tipologia di task evidenziate dagli autori sono principalmente: la diversità di pattern nella classe sconosciuta e la similarità tra i pattern delle classi conosciute e quelli delle classi sconosciute. L'approccio proposto dagli autori consiste nel segmentare l'immagine con una CNN, che classifica pixel per pixel, e determinare una soglia per la quale se l'output della softmax della CNN non la supera, quel pixel viene classificato come classe sconosciuta. In aggiunta, propongono anche un ulteriore



**Figura 1.7.** La figura mostra il risultato del modello senza l'utilizzo delle *distance maps* (a) e il risultato del loro utilizzo (b) [23].

step per migliorare la qualità delle maschere prodotte e diminuire il numero di pixel della classe sconosciuta. In particolare, il metodo proposto, chiamato *Morphological Filtering*, consiste nel riassegnare ogni pixel alla classe più presente nel suo vicinato e, nella pratica, l'effetto può essere visto come l'erosione delle regioni classificate come classe sconosciuta (Figura 1.8).



**Figura 1.8.** Illustrazione del metodo proposto in [24].

Altri lavori, come ad esempio [25], utilizzano la segmentazione come step intermedio per altri task. In particolare, in quest'ultimo lavoro citato, le maschere prodotte vengono utilizzate per la stima della profondità delle acque di alluvioni e il modello utilizzato è una versione della FCN [26] con stride 8 (FCN-8s) e VGG-16 come backbone. In [27] vengono testate diverse versioni della UNet [8] e della LinkNet [28], combinando tra loro diversi modelli per la backbone e per la parte di decoder. Inoltre, gli autori hanno mostrato come in media su quasi tutti gli aspetti, i modelli che avevano la backbone pre addestrata su ImageNet abbiano performato meglio. In alcuni lavori invece, come [29], oltre ad architetture di Deep Learning, vengono anche testati algoritmi di Machine Learning più tradizionali. Il loro lavoro riguarda il task della segmentazione delle diverse tipologie di vegetazione e, in particolare, il loro approccio è stato quello di testare diversi classificatori di ML, tra cui Random Forest, alberi decisionali e K-nearest neighbor. Per quanto riguarda, invece, l'archi-

tettura di Deep Learning da loro utilizzata, si tratta della SegNet [12]. Infine, in [30] viene affrontato lo stesso specifico task di questo lavoro, ovvero il dataset FloodNet. In particolare, l'approccio degli autori consiste nel testare varie architetture, tra cui ENet [17], PSPNet [15] e DeepLabV3+ [16]. Nello specifico, gli autori evidenziano come le architetture che performano meglio siano quelle context-based e tra queste, la migliore è risultata la PSPNet.

### 1.3 Contributo ed Outline

In questo lavoro viene esplorato il task della segmentazione semantica di immagini aeree del dataset FloodNet, attraverso approcci di Deep Learning. Nello specifico, viene proposto un nuovo approccio basato sulla risoluzione delle principali difficoltà individuate nel dataset. In particolare, il principale contributo di questo lavoro si può riassumere nei seguenti punti:

- una fase di *data cleaning*, per ovviare alla corposa presenza di errori nelle maschere del dataset.
- una fase di data augmentation offline, utile a far fronte ad un forte sbilanciamento del dataset verso alcune classi e alla presenza di altre in misura notevolmente minore.
- utilizzo di un'architettura context-based, mai utilizzata su questo dataset, volta ad ovviare alle difficoltà intrinseche di particolari classi, la cui semantica è fortemente basata sul loro contesto.

Per quanto riguarda la struttura dell'elaborato, esso è strettamente organizzato come segue. Nel Capitolo 2, verrà esplorato il problema generale della segmentazione trattando le varie metodologie esistenti, sia tradizionali sia di Deep Learning. In seguito, nel Capitolo 3, si tratteranno i principali concetti di Deep Learning, approfondendo anche alcune delle architetture più note. Nel Capitolo 4 invece, si entrerà nel dettaglio del task specifico di questo lavoro e si mostreranno, oltre all'architettura, anche le principali metodologie utilizzate. Successivamente, nel Capitolo 5 si illustreranno i vari esperimenti fatti durante tutto il lavoro e i loro risultati. Infine, nel Capitolo 6 si trarranno le conclusioni del lavoro.

## Capitolo 2

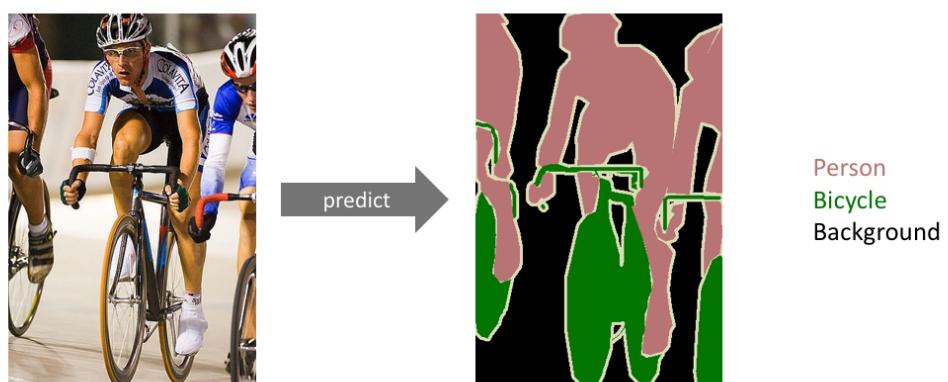
# Segmentazione di immagini

La segmentazione è uno dei task più popolari nel campo del Deep Learning applicato alle immagini e, in generale, nel campo della Computer Vision. In particolare, consiste nel produrre una maschera delle stesse dimensioni dell'immagine in input, dove il valore di ogni pixel rappresenta la classe a cui è stato assegnato, o detto in altre parole, partizionare un'immagine in regioni con un valore semantico ben preciso (Figura 2.1).

Grazie alle informazioni spaziali che si possono ottenere con la segmentazione, molti campi ne beneficiano, tra cui il campo dei veicoli autonomi, la detection di pedoni, diagnosi medica assistita da computer e molti altri [4].

In letteratura la segmentazione è stata applicata a una grande varietà di dati. Alcune tra le tipologie più utilizzate sono:

- **scala di grigi**: un tipo di immagine che presenta un solo canale e il valore dei pixel rappresentano, sostanzialmente, la quantità di luce del pixel.
- **RGB (red, green and blue)**: probabilmente la tipologia più nota, presenta tre canali che rappresentano rispettivamente la quantità di rosso, di verde e di blu del pixel.



**Figura 2.1.** Un esempio del risultato della segmentazione di un'immagine.

- **RGB-D**: un tipo di immagine che, oltre ad avere i tre canali del RGB, ha anche un quarto canale che rappresenta la profondità di quel pixel, ovvero la distanza dalla camera, informazione spesso molto utile nel campo della segmentazione semantica [31].
- **3D**: un tipo di dato spesso presente nella letteratura medica, come ad esempio nel campo della TC (tomografia computerizzata), dove le immagini vengono catturate con strumenti a raggi X [32].

Come detto in precedenza, la segmentazione è uno dei task più popolari e negli anni nel campo della ricerca, sono stati fatti molti passi avanti, sviluppando modelli e algoritmi sempre più performanti. In particolare, le performance di questi modelli sono state misurate testandoli su dei dataset, che sono stati usati come benchmark per paragonare i vari metodi. Tra i dataset di benchmark più noti ed utilizzati ci sono:

- **PASCAL-VOC** (PASCAL Visual Object Classes) [33]
- **Cityscapes dataset** [34]
- **ADE20K** [35]

Nel campo della computer vision, la segmentazione viene affrontata con diversi metodi. In particolare, esistono metodi tradizionali, che non fanno uso di tecniche di Machine Learning; metodi di Machine Learning; e infine metodi di Deep Learning, che sfruttano invece architetture che, come vedremo più avanti, riescono ad apprendere pattern molto complessi che i metodi tradizionali e di Machine Learning spesso non riescono a cogliere. Per quanto riguarda i metodi tradizionali, le tecniche sono svariate: alcune sono basate sull'estrazione di feature e altre sul preprocessing dell'immagine in input; invece, per quanto riguarda i metodi di Machine Learning, essi sono basati sul trasformare i pixel dell'immagine in una rappresentazione che poi può essere utilizzata dagli svariati algoritmi di Machine Learning per classificarli, come il K-Means o l'SVM (Support Vector Machines). Invece, le architetture di Deep Learning sono diverse e tra le più utilizzate ci sono la UNet[8], la DeepLabV3[36], la PSPNet[15], la FCN[26] e altre. Vedremo più avanti i concetti di Deep Learning sui quali si basano queste architetture. Al di là delle metodologie utilizzate, un aspetto molto importante dello sviluppare un algoritmo, un'architettura, oppure utilizzarne una già esistente, consiste nella scelta della metrica di valutazione. Di metriche ne esistono diverse:

- **Accuracy**: questa è probabilmente la più semplice e più intuitiva, ovvero misura il numero di pixel correttamente classificati, chiaramente in proporzione al numero totale di pixel. Questa metrica è generalmente una delle più utilizzate nel campo del Machine Learning. Viene definita come:

$$Accuracy = \frac{\sum_{i=1}^k n_{ii}}{\sum_{i=1}^k t_i}. \quad (2.1)$$

Dove  $k$  è il numero di classi,  $n_{ij}$  è il numero di pixel di classe  $i$  e classificati come di classe  $j$  e  $t_i$  il numero totale di pixel della classe  $i$  ovvero:

$$t_i = \sum_{j=1}^k n_{ij}. \quad (2.2)$$

Inoltre, a parte l'accuracy generale, che in pratica misura l'accuratezza del modello, ma non facendo caso alle singole classi, possiamo anche utilizzare l'accuracy di una classe. In particolare, l'accuracy di una singola classe  $i$  è definita come il rapporto tra il numero di pixel di quella classe correttamente classificati dal modello e il numero totale di pixel di quella classe:

$$Accuracy_i = \frac{n_{ii}}{t_i}. \quad (2.3)$$

Il principale svantaggio nell'utilizzo dell'accuracy è che in molti task alcune classi prevalgono rispetto alle altre, di conseguenza utilizzare l'accuracy può portare ad avere un valore alto, ma che in realtà è causato dalla predominanza di quella classe. Ad esempio, se in un'immagine l'80% dei pixel è di una classe, un modello che classifica tutti i pixel dell'immagine come appartenente a quella classe, otterrebbe una accuracy dell'80%.

- **Accuracy media:** questa ed altre metriche fanno fronte al problema sopra menzionato dell'accuracy. Questa metrica, in particolare, rappresenta la media delle accuracy delle singole classi e risolve il problema normalizzando l'accuracy rispetto al numero totale dei pixel:

$$MeanAccuracy = \frac{1}{k} \sum_{i=1}^k Accuracy_i = \frac{1}{k} \sum_{i=1}^k \frac{n_{ii}}{t_i}. \quad (2.4)$$

- **Mean intersection over union (mIoU):** anche chiamato *Jaccard index*, è la media dell'*intersection over union (IoU)* dell  $k$  classi, dove l'IoU di una classe è essenzialmente il rapporto tra l'intersezione e l'unione di due insiemi: l'insieme dei pixel di quella classe nella maschera e l'insieme dei pixel di quella classe nella predizione, ovvero:

$$IoU = \frac{\text{target} \cap \text{prediction}}{\text{target} \cup \text{prediction}}. \quad (2.5)$$

definibile anche come

$$IoU = \frac{n_{ii}}{t_i - n_{ii} + \sum_{j=1}^k n_{ij}}. \quad (2.6)$$

e di conseguenza la mIoU è definita come:

$$mIoU = \frac{1}{k} \sum_{i=1}^k IoU_i = \frac{1}{k} \sum_{i=1}^k \frac{n_{ii}}{t_i - n_{ii} + \sum_{j=1}^k n_{ij}}. \quad (2.7)$$

- **Frequency weighted intersection over union (FWIoU)**: una variante della mIoU, che invece di calcolare semplicemente la media delle IoU, calcola una media pesata rispetto al numero di pixel delle classi (frequenza), ovvero:

$$FWIoU = \left( \sum_{i=1}^k t_i \right)^{-1} \sum_{i=1}^k t_i IoU_i = \left( \sum_{i=1}^k t_i \right)^{-1} \sum_{i=1}^k t_i \frac{n_{ii}}{t_i - n_{ii} + \sum_{j=1}^k n_{ij}}. \quad (2.8)$$

- **Matrice di confusione**: non è una vera e propria metrica, ma è un metodo molto utilizzato anche e soprattutto nel campo della classificazione. La matrice di confusione è molto utile in particolare quando si vuole approfondire la natura degli errori del proprio metodo. Nello specifico, ci mostra per ogni classe  $i$  il numero di pixel correttamente predetti  $n_{ii}$  ma, ancora più importante, per ogni classe  $i$  ci mostra la suddivisione degli errori nelle altre classi, ovvero  $n_{ij}$  per  $j \in [1, 2, \dots, i-1, i+1, \dots, k]$ . Di conseguenza, possiamo rappresentare formalmente la matrice di confusione come:

$$\text{ConfusionMatrix} = [n_{ij}]. \quad (2.9)$$

dove  $i, j \in [1, 2, \dots, k]$ .

Inoltre, nel campo della Computer Vision, così come nel mondo algoritmico, i metodi vengono valutati secondo due ulteriori criteri: la complessità temporale, molto importante soprattutto in alcuni campi di applicazione dove il tempo necessario per processare un'immagine e produrre la sua segmentazione non deve essere maggiore di una certa soglia, come il campo dei veicoli autonomi; e la complessità spaziale, ovvero la quantità di memoria di cui ha bisogno l'algoritmo, metrica molto importante soprattutto in campi dove il processamento avviene su dispositivi che non abbondano di memoria, come smartphones e fotocamere.

## 2.1 La segmentazione di immagini negli umani

Partendo da come gli umani percepiscono un'immagine e riescono naturalmente e in modo intuitivo a segmentarla, il funzionamento di tali intuizioni si basa su quella che è chiamata "psicologia della Gestalt" [37] (dal tedesco *Gestaltpsychologie*, 'psicologia della forma' o 'rappresentazione'). La psicologia della Gestalt spiega come un umano riesca a percepire un'immagine e organizzarla in un sistema complesso. In particolare, spiega come, guardando il mondo intorno a sé, riesca a percepire scene complesse composte da molti gruppi di oggetti su uno sfondo, che a loro volta possono essere costituiti da altre parti e via dicendo. Il modo in cui gli umani riescano a raggiungere un risultato percettivo così notevole, visto il fatto che l'input visivo è, in un certo senso, solo una distribuzione spaziale di punti individuali variamente colorati, è basato su dei principi che definiscono le intuizioni dietro il raggruppamento di parti dell'immagine. Non esiste una lista precisa dei principi della Gestalt, ma ne esistono alcuni che sono più discussi e più comunemente utilizzati [38]:

- **principio di buona forma:** la struttura di oggetti che tendiamo a percepire è sempre la più semplice.
- **principio di prossimità:** formalizza l'intuizione dietro il fatto che tendiamo a raggruppare elementi vicini tra loro.
- **principio di destino comune:** tendenza a raggruppare elementi che hanno un movimento coerente tra loro.
- **principio di somiglianza:** tendenza a raggruppare gli elementi simili tra loro (in colore, forma, grandezza, ...)
- **principio di buona continuità:** tendenza a raggruppare elementi per formare oggetti e forme continue e coerenti nello spazio.
- **principio dell'esperienza passata:** l'osservatore tende a raggruppare gli elementi visivi in modo coerente al modo in cui li ha visti nel suo passato.

In generale, questi principi formalizzano delle intuizioni comuni alla psicologia di tutti gli umani. Il problema è che queste intuizioni sono spesso molto difficili da trasformare in un linguaggio matematico o in un algoritmo, e da qui nasce la difficoltà della segmentazione e di altri task nel campo della Computer Vision, che come quest'ultima risultano spesso intuitivi, ma molto difficili da trasformare in un algoritmo. Inoltre, anche per gli umani, a volte, il task della segmentazione risulta complesso e i risultati, quando cambia il soggetto, sono spesso diversi tra loro. Questo ci fornisce un'intuizione di come, a differenza di altri task come la classificazione, il problema della segmentazione sia particolarmente difficile anche solo da definire [39].

## 2.2 Approcci classici

Come detto in precedenza, il task della segmentazione d'immagini può essere affrontato con diversi approcci classici. Alcuni di questi trasformano il task in un problema di ottimizzazione, cercando poi di risolverlo con algoritmi iterativi, altri invece lo trasformano in un problema di *clustering*, trasportando l'immagine in uno spazio a più dimensioni per poi utilizzare algoritmi di Machine Learning.

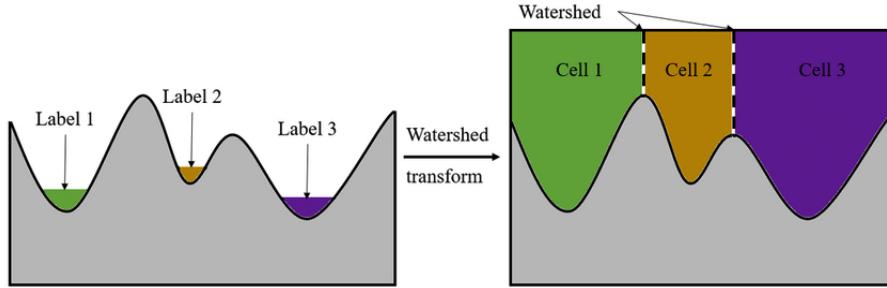
### 2.2.1 Metodi a soglia

Uno dei metodi più semplici e intuitivi è il metodo a soglia (*thresholding*). In pratica, esso consiste nel definire una soglia numerica di una feature dell'immagine, per poi classificare i pixel in base a questa soglia. Nella sua forma più semplice, definendo un'unica soglia l'immagine viene segmentata in due classi; nella sua variante multiclasse, invece, si definiscono più soglie, creando così degli intervalli che rappresentano le diverse classi. La feature utilizzata più spesso è quella dell'intensità dei pixel (in immagini in scala di grigi), ma se ne possono utilizzare tante altre, questo dipende molto dal campo di applicazione e dalla natura dell'immagine. Ad esempio, una feature spesso molto utile per segmentare un'immagine è la profondità. Il problema principale consiste nel fatto che questo tipo di feature, a parte il

caso in cui sia fornito direttamente dall'immagine (RGB-D), è spesso molto difficile da estrarre. Chiaramente, questo metodo restituisce spesso risultati approssimativi e inoltre la ricerca del valore ottimale della soglia non è affatto semplice e richiede spesso un'approfondita conoscenza del dominio.

### 2.2.2 Metodi dividi e fondi

Come abbiamo visto nel paragrafo precedente, una delle tecniche più semplici è definire una soglia con cui classificare i pixel. Sfortunatamente, nella maggior parte dei casi trovare questa soglia è molto difficile e molto spesso non è nemmeno unica, ovvero le immagini a volte presentano forti differenze da una regione all'altra, di conseguenza una buona soglia per una parte può risultare non buona per un'altra. Data questa difficoltà, una tecnica per risolverla consiste nel suddividere l'immagine in parti diverse oppure, al contrario, cercare di unire parti di immagine simili tra loro. Uno dei primi algoritmi che rientra in questa categoria è l'algoritmo degli spartiacque (in inglese *watershed*) [40], che considera le immagini in scala di grigi come una mappa topografica, trasformando l'intensità dei pixel nell'altezza. In particolare, utilizzando la metafora di un'alluvione, l'algoritmo suddivide l'immagine in diversi bacini idrografici, ovvero le regioni segmentate delle immagini sono composte da tutti quei punti da cui l'acqua finirebbe nello stesso punto (il minimo) (Figura 2.2). Sfortunatamente, anche questo algoritmo spesso restituisce risultati approssimativi, in particolare, uno dei maggiori problemi è che, associando una regione ad ogni minimo locale, spesso over-segmenta l'immagine. Infatti, questo algoritmo è soprattutto usato in sistemi interattivi dove la segmentazione è assistita da un utente che definisce il centro degli oggetti d'interesse.



**Figura 2.2.** Illustrazione di come l'algoritmo degli spartiacque trova le soglie (watershed nell'immagine) con cui segmentare le immagini.

Infine, ci sono altri metodi che, invece che cercare di suddividere l'immagine in regioni, cercano al contrario di unire tra loro regioni che abbiano una certa similarità con un approccio *bottom-up*. Una categoria di questi sono i metodi basati su grafi.

### 2.2.3 Metodi basati su grafi

Questa categoria di metodi si basa sulla rappresentazione dell'immagine come un grafo  $G = (V, E)$  dove i vertici  $V$  sono i pixel e gli archi  $E$  sono tra i pixel adiacenti.

Uno degli algoritmi che fa parte di questa categoria è quello proposto in [41], in cui i pesi rappresentano una misura di dissimilarità tra i pixel che connettono. Questa misura di dissimilarità  $w(e)$  può prendere diverse forme e quella più semplice è probabilmente la differenza di intensità tra i due pixel. Una qualsiasi regione  $R$  ha una misura di differenza interna chiamata  $Int(R)$ , che è definita come il massimo peso di un arco all'interno dell'albero ricoprente minimo di  $R$ , ovvero  $MST(R)$ :

$$Int(R) = \max_{e \in MST(R)} w(e). \quad (2.10)$$

Inoltre, per due regioni adiacenti, ovvero che hanno minimo un arco che le connette, si definisce la differenza tra loro come il peso minimo di un arco che le connette:

$$Dif(R_1, R_2) = \min_{e=(v_1, v_2) | v_1 \in R_1, v_2 \in R_2} w(e). \quad (2.11)$$

L'algoritmo unisce iterativamente tra loro due qualsiasi regioni  $R_1$  e  $R_2$  se la loro differenza  $Dif(R_1, R_2)$  è minore della minima differenza interna delle due regioni  $MInt(R_1, R_2)$ , dove

$$MInt(R_1, R_2) = \min(Int(R_1) + \tau(R_1), Int(R_2) + \tau(R_2)). \quad (2.12)$$

Dove  $\tau(R)$  rappresenta quanto la differenza tra le due regioni debba essere più grande delle differenze interne per non essere fuse, ma segmentate in due regioni diverse, ed è definito come:

$$\tau(R) = \frac{k}{|R|}. \quad (2.13)$$

Dove  $k$  è un parametro costante e  $|R|$  è la dimensione della regione. In realtà, questo componente  $\tau$  può assumere diverse forme a seconda dello specifico task e in base a come si voglia definire la bontà di una regione. Infine, la regola con cui l'algoritmo decide iterativamente di unire o meno due regioni è la seguente: unire  $R_1$  e  $R_2$  se  $Dif(R_1, R_2) < MInt(R_1, R_2)$ .



**Figura 2.3.** Esempio del risultato dell'algoritmo basato su grafi proposto in [41] applicato su un'immagine in scala di grigi.

### Tagli normalizzati

Un altro algoritmo molto popolare che appartiene a questa categoria è quello proposto in [42]. In questo algoritmo, il peso di un arco  $w_{i,j}$  tra i due pixel  $i$  e  $j$  rappresenta la loro affinità. L'idea dietro l'algoritmo è partizionare il grafo in regioni che abbiano affinità deboli, ovvero regioni connesse da archi con pesi bassi. Per definire il costo di un taglio tra due regioni  $A$  e  $B$ , viene definita la seguente misura:

$$cut(A, B) = \sum_{i \in A, j \in B} w_{i,j}. \quad (2.14)$$

Definito questo costo, non basta che trovare i tagli che lo minimizzino. Il problema è che utilizzando solamente questo costo, i tagli risultanti sono normalmente quelli che isolano singoli pixel, di conseguenza va aggiunto al costo un altro componente che penalizza i sottografi piccoli. Una migliore misura del costo di un taglio è la seguente misura, ovvero il costo del taglio normalizzato rispetto all'associazione dei due sottografi:

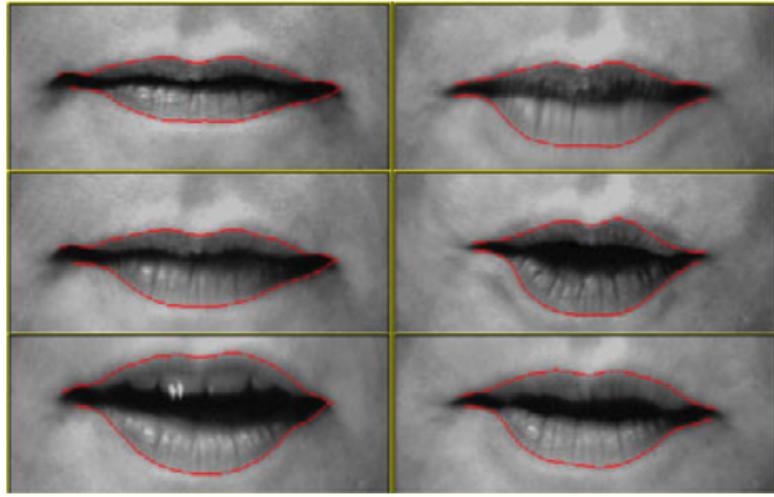
$$NCut(A, B) = \frac{cut(A, B)}{assoc(A, V)} + \frac{cut(A, B)}{assoc(B, V)}. \quad (2.15)$$

Dove  $assoc(A, V) = \sum_{u \in A, t \in V} w_{ut}$  ovvero la somma dei pesi degli archi tra i nodi  $A$  e tutti i nodi del grafo totale  $V$  (compresi anche quelli  $A$  stesso). In questo modo, normalizzando il costo rispetto all'associazione, penalizziamo i sottografi molto piccoli. Purtroppo però, la difficoltà di questo algoritmo sta nel fatto che minimizzare  $NCut$  è un problema NP-completo.

#### 2.2.4 Metodo dei contorni attivi

A volte, il task della segmentazione viene trasformato nel task della detection delle frontiere degli oggetti, ovvero dei loro bordi. In questo contesto vengono utilizzati i contorni attivi (*active contours*), anche detti *snakes* (per la loro forma), che non sono altro che delle curve che, iterativamente, si vanno a stringere intorno ad un oggetto, prendendo mano a mano la sua forma fino a che, auspicabilmente, non la egualino alla perfezione. Questo tipo di metodo è spesso molto utile in contesti dove, ad esempio, sia necessario tracciare la forma di un oggetto in un video o di un oggetto che cambi prospettiva; il motivo è che le performance di questo metodo dipendono molto dall'inizializzazione della curva. In particolare, quando la curva viene inizializzata con una certa forma e abbastanza vicina all'oggetto, si hanno più probabilità di ottenere una segmentazione più precisa. Ad esempio, se in un video l'algoritmo è riuscito a segmentare l'oggetto del frame precedente, oppure quest'ultimo è stato segmentato a mano dall'utente, quel contorno, salvo grandi cambiamenti, è un'ottima inizializzazione per il frame successivo (Figura 2.4).

Questo tipo di algoritmo si basa su informazioni locali del gradiente dell'intensità dei pixel e le utilizza per attrarre la curva verso i bordi dell'oggetto. In particolare i bordi, essendo caratterizzati da un cambiamento improvviso dell'intensità, presentano un gradiente elevato, di conseguenza l'algoritmo cerca iterativamente di portare i punti che formano la curva in pixel con gradienti alti. Questo meccanismo



**Figura 2.4.** Esempio di applicazione dei contorni attivi: lip tracking.

viene formalizzato con un problema di ottimizzazione, ovvero si vuole massimizzare la somma dei gradienti dell'intensità dei pixel dove sono i punti della curva. Per quanto riguarda la definizione di un contorno, esso è definito come un insieme di punti nello spazio bidimensionale connessi da linee rette:

$$V = \{v_i = (x_i, y_i) | i = 0, 1, 2, \dots, n - 1\}. \quad (2.16)$$

Dove  $n$  è il numero dei punti che formano la curva. Come detto precedentemente, si vuole massimizzare la somma dell'intensità dei gradienti e per fare ciò viene utilizzata la magnitudine del gradiente dell'intensità dei pixel al quadrato, ovvero  $\|\nabla I\|^2$ . Inoltre, dato che la curva deve essere attratta dai gradienti più alti, bisogna espandere il "campo di forza" dei picchi del gradiente, altrimenti un punto non vicinissimo a un bordo non verrebbe attratto. Per fare questo utilizziamo la sfocatura gaussiana applicata alla magnitudine del gradiente e di conseguenza il valore da massimizzare diventa  $\|\nabla n_\sigma * I\|^2$ . Infine, il problema viene trasformato nel trovare il minimo di  $E_{image}$ , dove:

$$E_{image} = - \sum_{i=0}^{n-1} \|\nabla n_\sigma * I(v_i)\|^2. \quad (2.17)$$

A questo punto, l'algoritmo non fa altro che cercare iterativamente di ottimizzare  $E_{image}$  portando ogni punto in una nuova posizione, fino a che questo non raggiunge una certa soglia stabilità in anticipo. Il problema di questa versione è che è molto sensibile al rumore e molto spesso il contorno tende ad assumere forme strane, proprio a causa della presenza del rumore del gradiente intorno all'oggetto. Per superare questa difficoltà, e perché in generale si vuole una curva che rispetti alcune caratteristiche naturali, come se simulasse un oggetto fisico, come l'elasticità e la morbidezza, al problema di ottimizzazione viene aggiunto un secondo componente  $E_{contour}$  che rappresenta questi vincoli di forma. In particolare,  $E_{contour}$  è composta a sua volta da altri due componenti  $E_{elastic}$  e  $E_{smooth}$ , che rappresentano le due

caratteristiche appena menzionate. In realtà, a  $E_{contour}$  possono essere aggiunti altri componenti a seconda del task specifico e di come si voglia vincolare la forma della curva. Ad esempio, molto spesso quando si conosce a priori la forma dell'oggetto d'interesse, si può aggiungere al problema un componente che penalizzi forme diverse da questa. Di conseguenza, la formula di  $E_{contour}$  è la seguente:

$$E_{contour} = \alpha E_{elastic} + \beta E_{smooth}. \quad (2.18)$$

Dove  $\alpha$  e  $\beta$  sono due coefficienti che rappresentano il peso che vogliamo dare ai due vincoli di elasticità e morbidezza. Per quanto riguarda il vincolo di elasticità, esso rappresenta il desiderio che la curva simuli il comportamento di un elastico, ovvero quando due punti della curva sono più distanti c'è una sorta di forza di attrazione più forte e di conseguenza la curva tende a non distanziare troppo i punti tra loro. Di seguito la formula del componente di elasticità di un punto del contorno  $v(s)$ .

$$E_{elastic}(v(s)) = \left| \frac{\delta}{\delta s} v(s) \right|^2. \quad (2.19)$$

Ovvero il quadrato della derivata prima del punto del contorno, che rappresenta l'intuizione del fatto che non si vogliono grandi distanze da un punto all'altro. Per quanto riguarda il componente di morbidezza, invece, esso è definito come:

$$E_{smooth}(v(s)) = \left| \frac{\delta^2}{\delta^2 s} v(s) \right|^2. \quad (2.20)$$

Ovvero il quadrato della derivata seconda del punto del contorno, che invece rappresenta l'intuizione del fatto che non si vogliono cambiamenti improvvisi. Dato che in realtà la curva del contorno è composta da punti discreti, le formule dei due componenti diventano:

$$E_{elastic}(v(i)) = (x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2. \quad (2.21)$$

$$E_{smooth}(v(i)) = (x_{i+1} - 2x_i + x_{i-1})^2 + (y_{i+1} - 2y_i + y_{i-1})^2. \quad (2.22)$$

e quindi

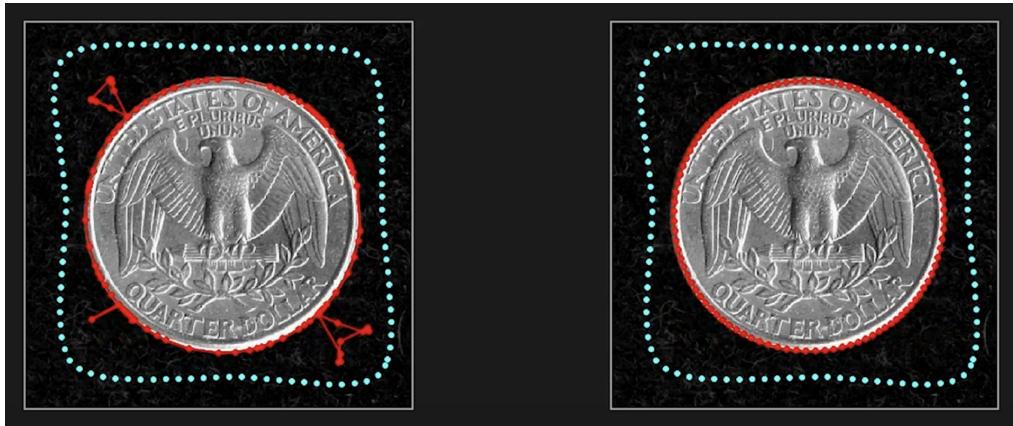
$$E_{elastic} = \sum_{i=0}^{n-1} (x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2. \quad (2.23)$$

$$E_{smooth} = \sum_{i=0}^{n-1} (x_{i+1} - 2x_i + x_{i-1})^2 + (y_{i+1} - 2y_i + y_{i-1})^2. \quad (2.24)$$

Infine, il problema totale diventa minimizzare il seguente valore:

$$E_{total} = E_{image} + E_{contour}. \quad (2.25)$$

La Figura 2.5 illustra un esempio di come il risultato migliori aggiungendo il componente  $E_{contour}$ .



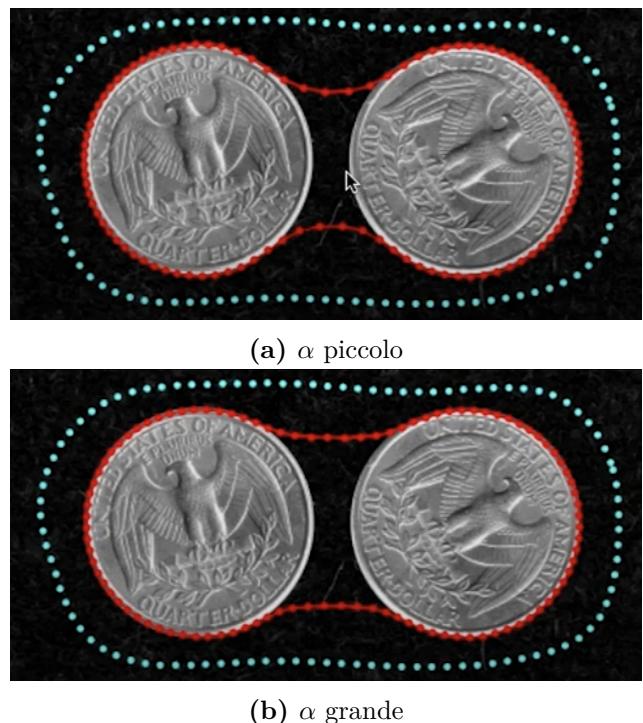
**Figura 2.5.** Esempio di applicazione del metodo del contorno attivo. In entrambe le immagini possiamo vedere un contorno blu (contorno di inizializzazione) e uno rosso (contorno finale). In particolare, a sinistra è stata utilizzata la versione dell'algoritmo che minimizza  $E_{image}$ , mentre a destra viene minizzato  $E_{total}$  e possiamo notare come quello a destra ottenga un risultato migliore.

Come già anticipato, le performance di questo algoritmo dipendono molto dalla bontà dell'inizializzazione, inoltre performa bene soprattutto con immagini che contengono un solo oggetto, anche se si può utilizzare su immagini che ne contengono più di uno. In particolare, a seconda della natura degli oggetti si possono variare i parametri  $\alpha$  e  $\beta$ , per dar modo al contorno di adattarsi meglio agli oggetti (Figura 2.6). In altri casi, come già menzionato, può risultare utile aggiungere ulteriori componenti a  $E_{contour}$  oltre a  $E_{elastic}$  e  $E_{smooth}$ . Infine, esiste una variante dell'algoritmo che, invece di far partire il contorno da una forma più larga per poi farlo contrarre sull'oggetto, fa partire il contorno dal suo interno e lo fa gonfiare fino a modellare i suoi bordi.

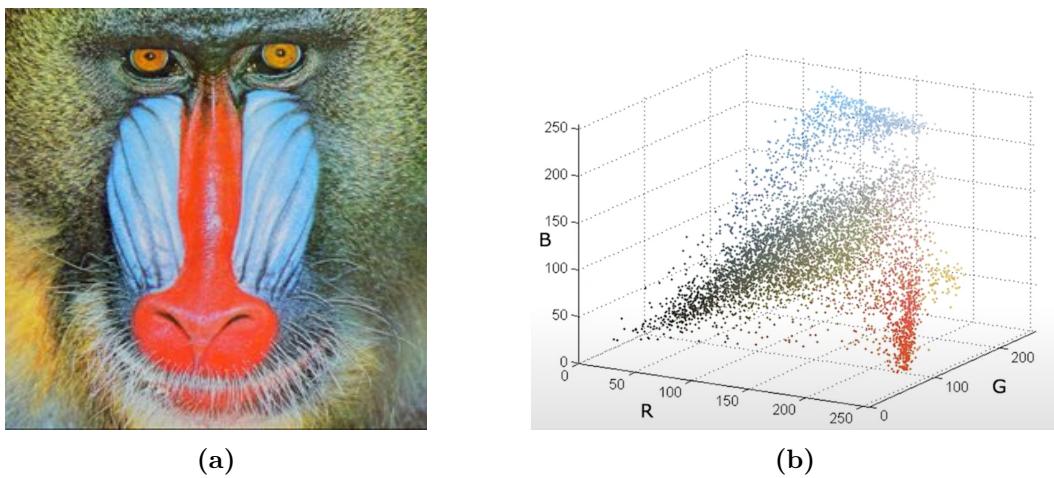
### 2.2.5 Metodi di clustering

Così come i metodi basati su grafi hanno trasformato l'immagine in un grafo facendo corrispondere a ogni pixel un nodo, allo stesso modo i metodi di clustering trasformano l'immagine in una distribuzione in uno spazio euclideo a più dimensioni. In particolare, la prima fase di un metodo di questa categoria è capire su quali feature basarsi per mappare ogni pixel in un punto dello spazio. Una delle scelte più semplici, nel caso di immagini RGB, è quella di utilizzare i tre canali dell'immagine, ovvero la quantità di rosso, di verde e di blu (Figura 2.7). Altre feature molto utilizzate sono la luminosità, la posizione del pixel, la profondità (RGB-D), ma anche feature più complesse come la texture. Ancora una volta, la scelta di queste feature è molto importante, ma soprattutto dipende fortemente dal campo di applicazione e dalla natura delle immagini, di conseguenza è necessaria un'approfondita conoscenza di entrambi.

Una volta completata la mappatura dell'immagine nello spazio scelto, ogni pixel  $i$  viene rappresentato con un vettore di feature  $f_i = [f_i^1, f_i^2, f_i^3, \dots, f_i^k]$  dove  $k$  è il numero di feature scelte. Così come in altri metodi, per guidare l'algoritmo nel raggruppare i pixel, viene utilizzata una misura della loro similarità. In particolare,



**Figura 2.6.** Esempio di segmentazione di due oggetti con un contorno attivo, variando però il coefficiente  $\alpha$ . Come si può notare, abbassando  $\alpha$  si dà modo al contorno di adattarsi meglio ai due oggetti. Chiaramente, come modificare i due coefficienti  $\alpha$  e  $\beta$  dipende dalla natura dell'immagine.

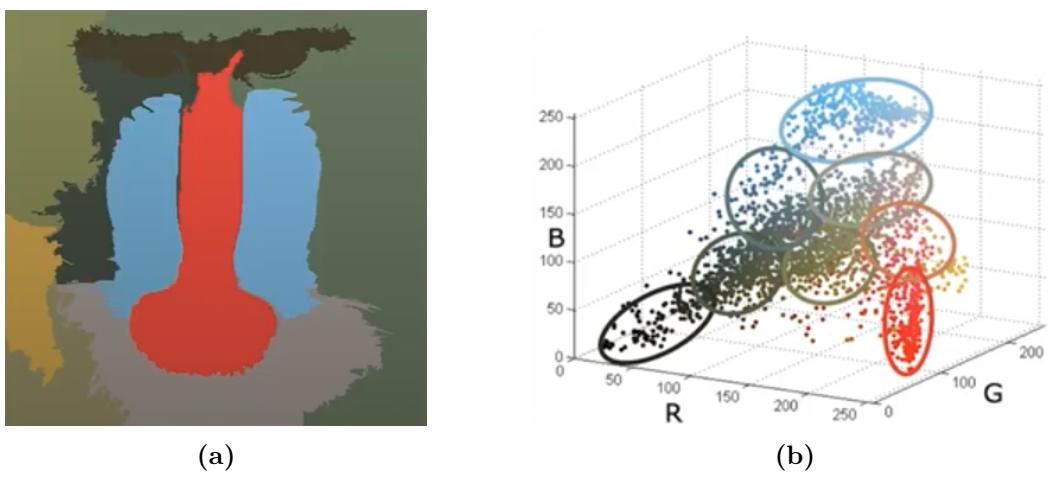


**Figura 2.7.** Le due figure sono un esempio di come un’immagine (a) si possa mappare in un spazio tridimensionale (b) utilizzando come feature i tre canali dell’immagine (RGB).

la similarità viene rappresentata dalla distanza nello spazio dei vettori di feature dei pixel e per calcolarla viene utilizzata quella che è chiamata distanza  $L^2$  o distanza euclidea, definita in uno spazio a  $k$  dimensioni come:

$$S(f_i, f_j) = \sqrt{\sum_k (f_i^k - f_j^k)^2}. \quad (2.26)$$

A questo punto, il task della segmentazione è stato totalmente trasformato in un problema di clustering (Figura 2.8) e può essere utilizzato un qualsiasi algoritmo che risolva questo tipo di problema. Di algoritmi che risolvono questo tipo di problema ne esistono diversi, tra i più noti ci sono K-Means e Mean Shift.



**Figura 2.8.** Le due immagini mostrano il risultato di un algoritmo di clustering applicato alla mappatura dell’immagine della Figura 2.7a. Nella (a) il risultato finale, ovvero l’immagine segmentata; la (b) invece mostra i clusters nello spazio tridimensionale.

### K-means

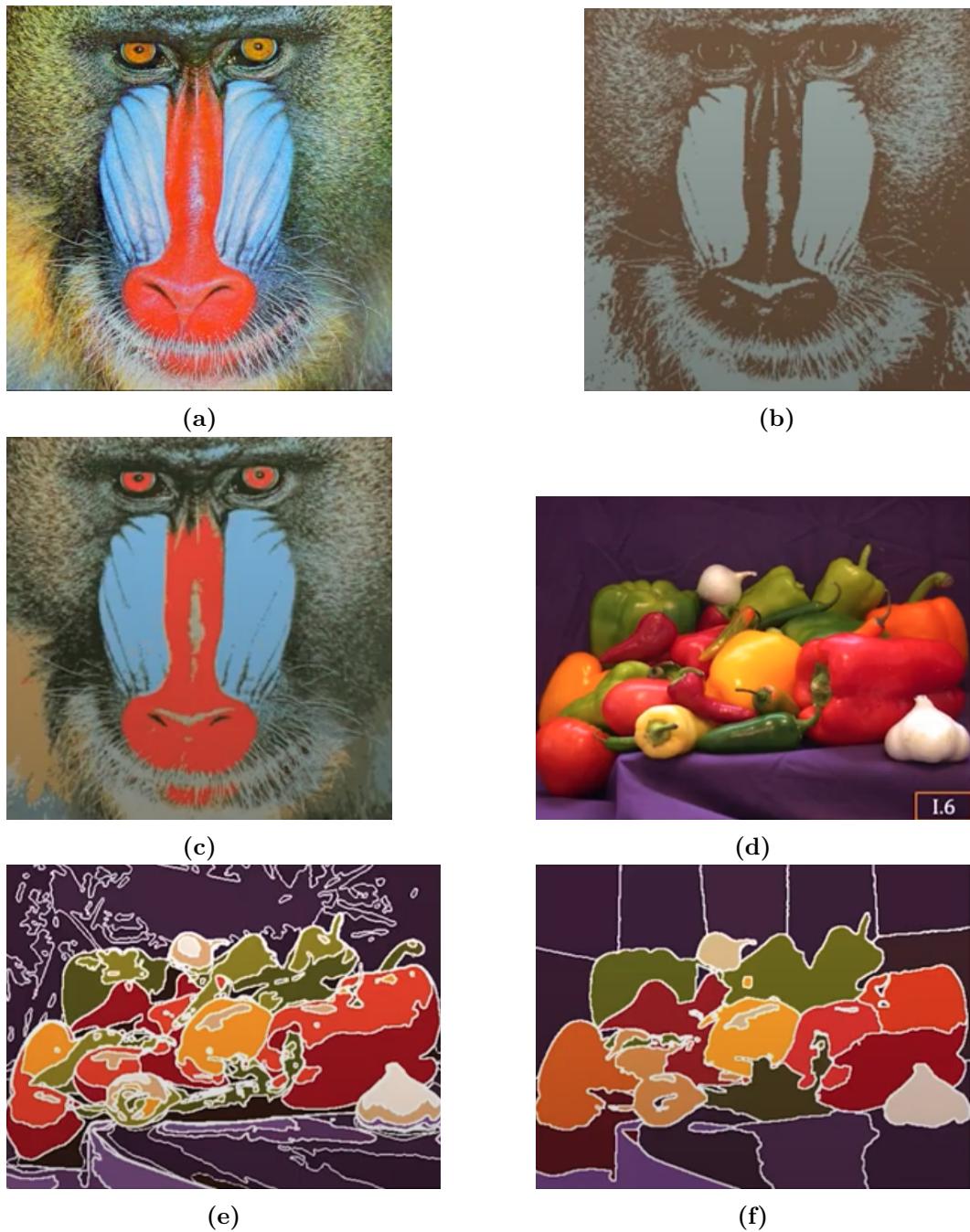
Probabilmente, l'algoritmo di clustering più noto è il *k-means* [43, 44]. Il suo funzionamento è abbastanza semplice e si basa sull'obiettivo di trovare il centroide o punto medio, di ogni cluster, ovvero trovare il punto che minimizzi la varianza totale intra-gruppo  $Var$ , che non è altro che una misura della variabilità all'interno del cluster. Formalmente, il problema è definito come trovare:

$$\arg \min_S \sum_{i=1}^k |S_i| Var(S_i). \quad (2.27)$$

Dove  $S = S_1, S_2, \dots, S_k$  è l'insieme dei cluster. Una volta trovati i centroidi di ogni cluster, il cui numero  $k$  deve essere definito in anticipo, l'algoritmo classifica ogni punto nello spazio determinando quale dei  $k$  centroidi sia il più vicino. Per quanto riguarda la prima fase della ricerca dei centroidi, l'algoritmo parte inizializzando  $k$  centroidi e calcolando i cluster secondo la regola del centroide più vicino. A questo punto, l'algoritmo ripete iterativamente questo meccanismo, ricalcolando ogni volta i centroidi dei cluster fino a non convergere, ovvero fino a che i centroidi non cambiano più posizione o comunque il loro cambiamento è sotto una certa soglia. Ancora una volta purtroppo, le performance di questo algoritmo dipendono fortemente dalla bontà dell'inizializzazione. In questo caso, l'inizializzazione può essere fatta in diverse maniere: la cosa più semplice è quella di inizializzarli randomicamente, ma chiaramente è anche la scelta meno performante; un altro metodo molto semplice consiste nell'inizializzarli randomicamente, controllando però che nessuno dei centroidi sia molto vicino e in quel caso reinizializzarli; un terzo metodo è quello di scegliere dei centroidi che siano uniformemente distribuiti nello spazio; infine, un ultimo metodo può essere utilizzare il k-means prima su una sottoporzione della distribuzione, e poi utilizzare i centroidi risultanti come inizializzazione per applicare l'algoritmo a tutta la distribuzione.

### Mean Shift

Come detto nel paragrafo precedente, l'algoritmo k-mean ha lo svantaggio di essere molto sensibile rispetto all'inizializzazione dei centroidi. Inoltre, un altro svantaggio del k-mean è che l'utente deve decidere a priori il numero di cluster. L'algoritmo *mean shift* [45, 46] cerca di mitigare questi due problemi. Anche mean shift è un algoritmo iterativo e in particolare, il meccanismo generale si basa sul trovare le mode della distribuzione dei punti nello spazio, che ad alto livello possono essere pensate come i punti dove la distribuzione è più densa. Il funzionamento dell'algoritmo è il seguente: per ogni punto seleziona intorno a questo una finestra di dimensione  $W$ , che è l'unico parametro dell'algoritmo, e successivamente calcola la moda della distribuzione dei punti all'interno di quella finestra, che sarà il centro della finestra all'iterazione successiva. Ripetendo questo meccanismo finché il vettore di spostamento della moda, ovvero la distanza della moda di un' iterazione a quella dell'iterazione successiva, non si azzeri oppure non scenda sotto una certa soglia, l'algoritmo trova la moda verso la quale il punto di partenza è stato "attratto". Infine, il mean shift crea i cluster trovando i "bacini di attrazione", ovvero gli



**Figura 2.9.** Le quattro immagini mostrano il risultato dell'algoritmo k-means applicato a immagini RGB: (a) e (d) sono le due immagini originali; (b) mostra il risultato con  $k = 2$ ; (c) con  $k = 8$ ; (e) è il risultato con  $k = 16$  e infine (f) è il risultato con  $k = 16$  ma in uno spazio a cinque dimensioni ovvero  $(R, G, B, x, y)$  dove  $x$  e  $y$  rappresentano le coordinate dei pixel nell'immagine, mentre per tutte le altre è stato utilizzato uno spazio a tre dimensioni ovvero  $(R, G, B)$ .

insiemi di punti che vengono attratti verso la stessa moda. La Figura 2.9 mostra alcuni esempi dell'applicazione del mean shift.

### 2.2.6 Algoritmi Supervised

In generale, come già menzionato, una volta mappata l’immagine in uno spazio multidimensionale di feature, il task della segmentazione si può ricondurre, nel caso in cui il numero di classi fosse noto a priori, in un classico task di classificazione Machine Learning. Di conseguenza, tutti gli algoritmi di Machine Learning per la classificazione diventano validi per risolvere il problema della segmentazione. Di algoritmi di classificazione ne esistono svariati e tra i più popolari vi sono l’SVM (Support Vector Machines), il Regressore logistico, alberi decisionali, K-nearest neighbor e molti altri. La principale differenza con gli algoritmi menzionati nei paragrafi precedenti (2.2.1, 2.2.2, 2.2.3, 2.2.4, 2.2.5) è che questi ultimi rientrano nella categoria di algoritmi unsupervised, mentre tutti i classificatori di Machine Learning appena menzionati rientrano in quella supervised, ovvero hanno bisogno di un dataset per poter apprendere i propri parametri e riuscire a classificare.

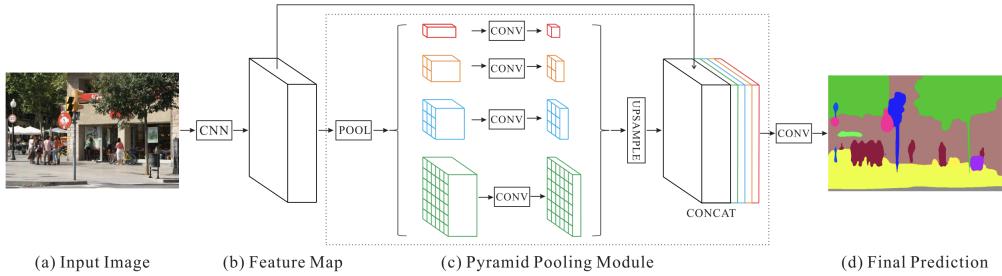
## 2.3 Approcci di Deep Learning

Oltre agli approcci classici, vi sono tecniche di Deep Learning che utilizzano architetture complesse, per riuscire ad apprendere il pattern d’interesse e in particolare, per apprendere le feature che determinano la semantica dei pixel, cosa che invece deve essere fatta manualmente quando si utilizzano gli algoritmi di Machine Learning (fase della *feature extraction*).

### 2.3.1 Architetture basate sul contesto

Questa categoria di architetture utilizza una tipologia di informazione molto importante e che i metodi di cui abbiamo parlato nei capitoli precedenti non utilizzano, ovvero il contesto di un pixel. In particolare, il contesto risulta molto importante per la classificazione semantica di un pixel, in quanto spesso le feature locali di un pixel come il colore, la luminosità, ma anche altre più complesse, non sono sufficienti per distinguere due pixel di classi diverse. Come già visto, in generale nel campo della Computer Vision la tipologia di architettura più utilizzata è la CNN, che si basa sulle presenza di strati convoluzioni, ed è proprio grazie a questi che l’architettura riesce ad estrapolare il contesto di un pixel. In particolare, la categoria delle architetture context-based fa un forte uso delle convoluzioni in diverse forme. Una delle varianti più utilizzate per catturare un contesto più ampio del pixel, senza però andare ad aumentare di troppo il costo computazionale, è la convoluzione dilatata. In particolare alcune, per fare in modo di costruire campi ricettivi con diverse ampiezze, fanno uso delle convoluzioni dilatate a diverse scale. Ad esempio, la DilatedNet [47] fa uso di cinque diversi parametri di dilatazione, ovvero 1,2,4,8 e 16, ottendendo una mIoU del 67.6% sul dataset PASCAL VOC 2012. In [48] le feature locali di un pixel vengono fuse con le feature globali dell’immagine, così facendo aggiungono al contesto locale del pixel quello globale, ottenendo una mIoU di 69.8% sul PASCAL VOC 2012. In [15] invece, viene utilizzato un modulo chiamato *Pyramid Pooling Module* (PPM), che utilizza uno strato di pooling a diverse scale per poi fondere i risultati di questi pooling insieme, in modo da avere informazioni di contesto a di-

verse scale (Figura 2.10). In particolare, l’idea è che con il solo utilizzo del contesto globale, si ha una perdita di informazioni nelle sotto regioni, mentre utilizzando il PPM vengono catturate le informazioni a diverse scale, compresa quella globale. Il PPM ha poi ispirato molti altri lavori che, apportando modifiche, hanno migliorato ancora di più le performance, come ad esempio la DeepLabV2 [49], che a partire dall’idea del PPM ha costruito il modulo Atrous Spatial Pyramid Pooling (ASPP), sostituendo in sostanza la normale convoluzione e il pooling con la convoluzione dilatata a diverse scale, ovvero con diverse dilatazioni, ottenendo su PASCAL VOC una mIoU di 79.7%.



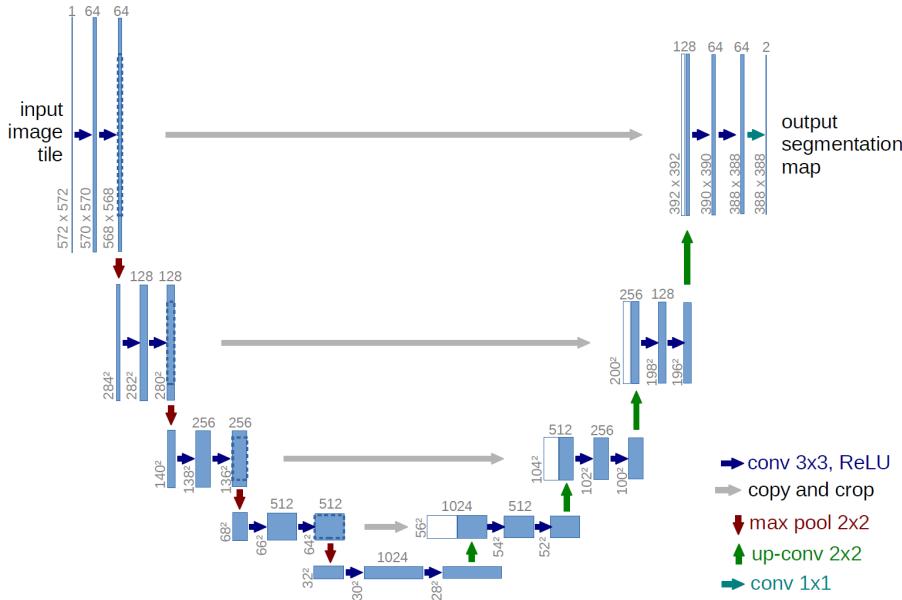
**Figura 2.10.** Illustrazione del Pyramid Pooling Module (PPM) [15].

### 2.3.2 Architetture basate sul feature-enhacement

Il concetto generale alla base di questa categoria di architettura è che, nelle CNN, l’estrazione strato per strato di feature sempre più di alto livello causa una perdita di informazioni spaziali. In particolare, a causa soprattutto del pooling e delle convoluzioni con stride, le feature avanzando progressivamente nella rete, diminuiscono di risoluzione e di conseguenza perdono le informazioni spaziali, che in task come la classificazione non sono importanti, ma che invece lo sono nel task della segmentazione. La soluzione proposta da questa categoria di architetture per risolvere questo problema è unire i due aspetti, ovvero unire le feature dei primi livelli, che hanno soprattutto informazioni spaziali, con quelle dei livelli più profondi, che invece non hanno molta informazione spaziale ma contengono informazioni sulle feature di alto livello. Una delle prime architetture di questo tipo è stata la FCN [26], che mette in atto questa idea aggiungendo delle *skip connections*, collegando così le feature degli strati intermedi con quelle degli ultimi strati. La UNet [8] è un’altra architettura che si rifà a questo concetto ed è una delle architetture più note nel campo della segmentazione semantica. In particolare, la UNet, chiamata così per la sua peculiare forma a U, a differenza della FCN utilizza delle skip connections tra tutti gli strati, ovvero connette tutte le feature della prima parte della rete, responsabile della feature extraction e chiamata encoder, a quelle della seconda parte della rete, chiamata decoder (Figura 2.11).

### 2.3.3 Architetture basate sulla deconvoluzione

La prima architettura di questo tipo è stata la DeconvNet [50], che ha una struttura encoder-decoder. In particolare, questa rete utilizza, a parte i classici moduli di poo-

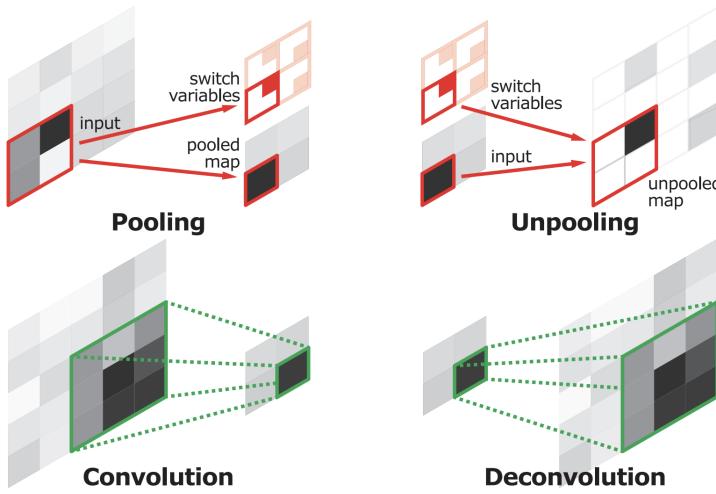


**Figura 2.11.** Architettura della UNet [8].

ling, convoluzione, e via dicendo, altre due tipologie di modulo, ovvero l'unpooling e la deconvoluzione. L'idea alla base del loro utilizzo è far fronte al problema della perdita di precisione nelle informazioni spaziali, nella parte di decoder in cui viene fatto l'upsample delle feature map per riportarle alla risoluzione originale. In particolare, negli strati di pooling dell'encoder, viene memorizzato l'indice della cella più grande, in modo che, nella fase di upsample, il modulo di unpooling possa ricreare la stessa finestra che era passata dentro lo strato di pooling. L'output dell'unpooling, tuttavia, ha delle celle azzerate, questo poiché nel corrispettivo pooling nell'encoder viene memorizzata soltanto la cella maggiore, ma non le altre. Per risolvere questo problema l'output del modulo di unpooling viene passato alla deconvoluzione che, grazie ai parametri appresi, riesce a ricostruire la feature map (Figura 2.12).

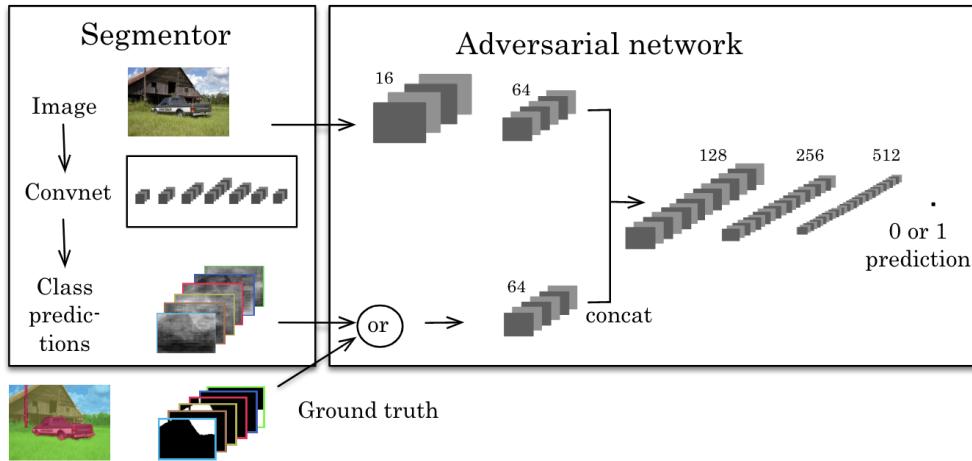
### 2.3.4 Architetture basate sulle GAN

Questa tipologia di architettura è composta da due reti chiamate "segmentatore" e "rete avversaria" (*adversarial network*). La prima ha la responsabilità, partendo dall'immagine in input, di creare una partizione in sotto-regioni non intersecanti, ovvero la maschera. La seconda, invece, ha la responsabilità di distinguere tra l'output della prima rete e la maschera corretta. In particolare, le due reti sono messe una contro l'altra: il segmentatore ha l'obiettivo di non far capire alla seconda rete la differenza tra il suo output e la vera maschera, mentre la seconda rete (Figura 2.13) ha l'obiettivo di capire la differenza. L'ANet [51] è una delle prime reti di questo tipo utilizzate per la segmentazione semantica e in particolare, il concetto principale alla sua base è che la rete avversaria, a differenza di una classica loss function utilizzata per l'addestramento di una rete, riesce a cogliere differenze di più alto livello tra l'output del segmentatore e la maschera, come ad esempio differenze nella forma degli oggetti o nelle loro proporzioni. La prima rete, quella



**Figura 2.12.** Illustrazione dei due moduli di unpooling e di deconvoluzione [50] e della loro differenza con, rispettivamente, il pooling e la convoluzione.

responsabile della segmentazione, utilizza per l’addestramento una loss function composta, ovvero la loss function finale è il risultato della somma pesata di due ulteriori loss function, una classica cross entropy e un’adversarial loss function. In particolare, la seconda loss function rappresenta quanto la rete avversaria riesca a distinguere tra il suo output e la maschera corretta.



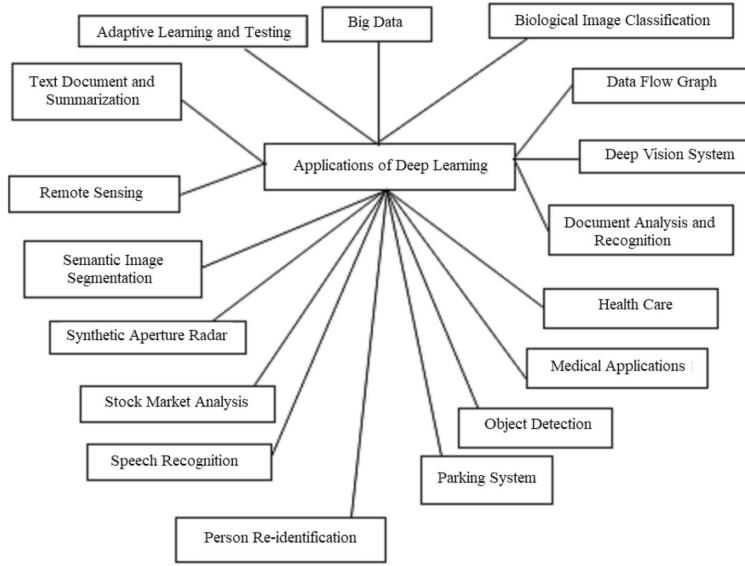
**Figura 2.13.** Panoramica dell’architettura basata sulle GAN proposta in [51]. A sinistra il segmentatore prende l’immagine RGB come input, e produce la maschera. A destra la rete avversaria prende la maschera prodotta dal segmentatore e produce una label (1 = maschera corretta, o 0 = maschera sintetica del segmentatore). La rete avversaria opzionalmente può prendere anche l’immagine RGB come input.

# Capitolo 3

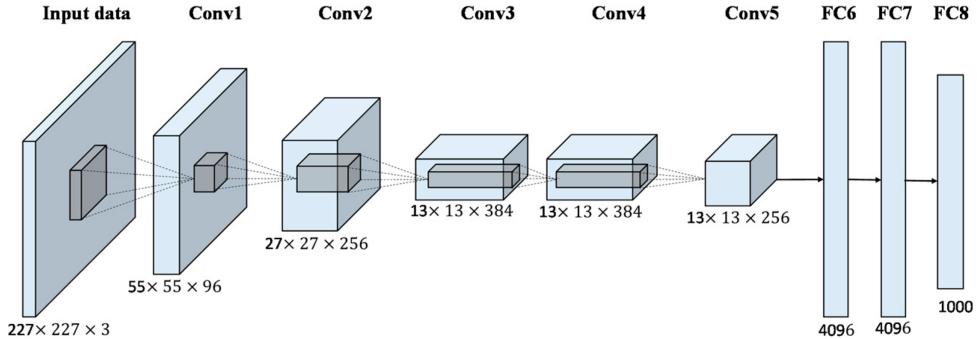
## Deep Learning

Il Machine Learning (ML) è una branca dell’Intelligenza Artificiale (IA) che si occupa di costruire modelli che apprendono dai dati che gli vengono forniti. Il Deep Learning, a sua volta, è una branca del ML e anch’esso si occupa di costruire modelli che apprendono pattern dai dati. La peculiarità del DL sta nella tipologia di modelli utilizzati, ovvero le reti neurali, anche dette Artificial Neural Networks (ANN) o Deep Neural Network (DNN). Una ANN è un modello di calcolo ispirato alla struttura delle reti neurali del cervello, ed è costituita da un gran numero di componenti di calcolo di base (neuroni), che sono collegati tra loro in una rete di comunicazione complessa attraverso la quale è in grado di approssimare funzioni molto complesse. L’utilizzo delle reti neurali infatti, diventa fondamentale quando il pattern che si cerca di apprendere è molto complesso e soprattutto non lineare. Proprio questa capacità di apprendere pattern complessi ha reso il DL un tool molto utilizzato e che ha portato netti miglioramenti in molti campi, tra cui il Natural Language Processing, speech recognition, campo medico, sistemi di trasporto intelligenti [52] e molti altri [53]. La Figura 3.1 illustra diversi campi di applicazione del DL. Un altro campo in cui il DL ha avuto molto successo è stato quello dell’analisi d’immagini. In particolare, per alcuni task come la classificazione e la segmentazione, le reti neurali hanno apportato netti miglioramenti. Infatti, a partire dal 2012, quando l’AlexNet [54] (Figura 3.2) vinse la ImageNet challenge battendo gli altri concorrenti con una rateo di errore di circa il 16%, contro il 26% del vincitore dell’anno precedente, le reti neurali hanno sempre dominato le challenge riguardanti task di Computer Vision. Un’altra peculiarità del DL, che lo distingue dal ML e che lo rende così efficiente in campi come la Computer Vision, è che la fase di *feature extraction* del ML, in cui vengono manualmente estratte le feature dal dato, è implicita nell’apprendimento della rete neurale (Figura 3.3). Questa caratteristica risulta fondamentale in quanto spesso questa fase risulta molto complessa, sia per la difficoltà dell’estrazione stessa, soprattutto per dati di tipo immagine, sia perché spesso per conoscere quali siano le feature da estrarre, è necessaria una conoscenza specifica del campo di applicazione. Nelle reti neurali invece, è proprio attraverso il meccanismo di apprendimento che la rete capisce quali sono le feature importanti e che determinano la natura del dato.

Oltre ai vantaggi, le reti neurali presentano anche diversi svantaggi. Uno di questi è che per alcuni aspetti le reti neurali sono dei modelli black-box. In particolare,



**Figura 3.1.** Alcuni dei campi di applicazione del Deep Learning [53].

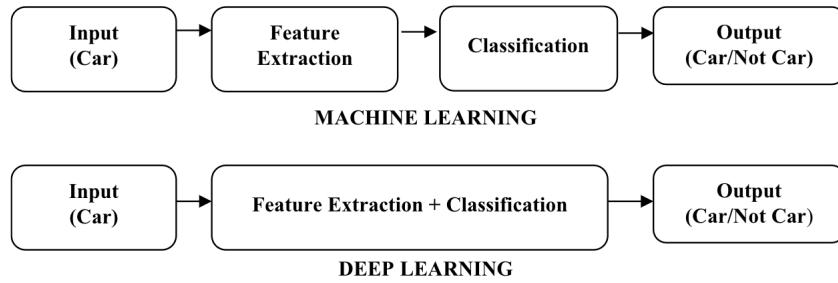


**Figura 3.2.** Architettura della rete AlexNet.

il loro funzionamento è molto complesso e spesso, a differenza degli algoritmi di ML, il loro comportamento e i loro output diventano difficili da comprendere e di conseguenza da spiegare. Infatti negli ultimi anni, un campo dell'IA in forte crescita è l'Explainable AI [55], che si occupa di algoritmi che forniscono trasparenza e interpretabilità a metodologie come le reti neurali.

### 3.1 Percettrone

Il percettrone è un modello matematico ispirato al modello del neurone biologico [56] e costituisce il componente di calcolo di base delle reti neurali. Mentre il neurone biologico è costituito da quattro principali componenti (soma, dendriti, assone e sinapsi), il percettrone è composto da due parti (Figura 3.4): nella prima

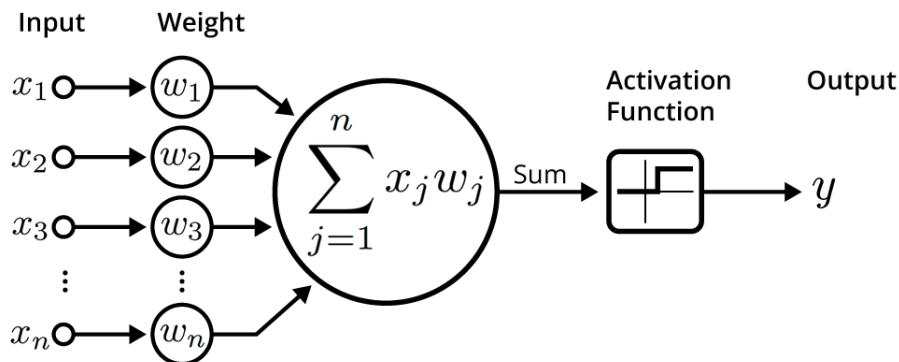


**Figura 3.3.** Differenza tra Machine Learning e Deep Learning [53].

parte, viene calcolato il prodotto scalare tra il vettore d'input  $X = (x_1, x_2, \dots, x_n)$  e i parametri del percettrone  $W = (w_1, w_2, \dots, w_n)$ , mentre nella seconda parte il risultato del prodotto scalare viene dato in input alla funzione di attivazione  $g$ , che ci restituisce uno scalare. Di seguito la formula dell'output del percettrone:

$$y = g\left(\sum_{j=1}^n x_j w_j\right). \quad (3.1)$$

La funzione di attivazione è una parte critica del percettrone, in quanto fornisce alla rete la non linearità, proprietà fondamentale per approssimare funzioni non lineari complesse. In particolare, senza funzioni di attivazione, per quanto complesse e profonde possano essere le reti neurali, non rappresenterebbero altro che una funzione lineare dell'input e diventerebbero un semplice regressore lineare. Più avanti vedremo nei dettagli quali sono le funzioni di attivazione più utilizzate nelle moderne reti neurali.

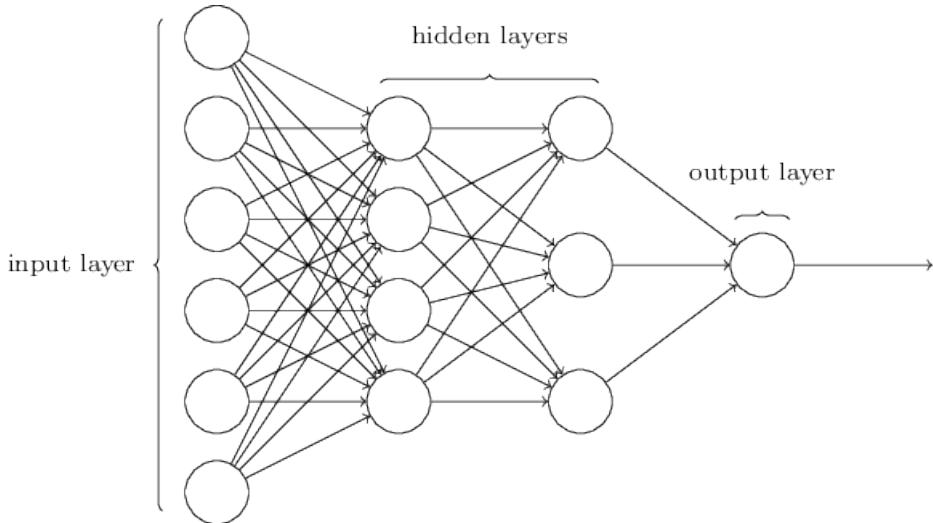


**Figura 3.4.** Illustrazione della struttura di un percettrone.

## 3.2 Percettrone Multistrato

Una delle tipologie più semplici di reti neurali è il Percettrone Multistrato, in inglese Multi-Layer Perceptron (MLP). L'MLP non è altro che un'insieme di percetroni

collegati tra di loro e organizzati in strati (Figura 3.5), dove gli output dei percettroni di uno strato sono l'input dei percettroni dello strato successivo. Gli strati, minimo tre, sono suddivisi in strato d'input, strato d'output e strati nascosti. Lo strato d'input è quello che riceve per primo il dato da elaborare, lo strato di output è quello che invece restituisce l'output finale della rete e gli strati nascosti sono tutti gli altri che si trovano nel mezzo.



**Figura 3.5.** Generica architettura di un Percettrone Multistrato.

Come detto precedentemente, grazie all'utilizzo delle funzioni di attivazione, l'output finale della rete risulta una funzione complessa e fortemente non lineare dell'input, complessità e non linearità che aumentano con l'aumentare delle dimensioni della rete. Questa proprietà risulta fondamentale e peculiare delle reti neurali, che si distinguono dagli altri modelli di ML proprio per la capacità di rappresentare funzioni molto complesse. Infatti, le reti neurali hanno preso il sopravvento soprattutto in quei task dove le funzioni o i pattern da apprendere sono molto complessi, come nel campo dell'analisi di immagini.

### 3.3 Funzioni di attivazione

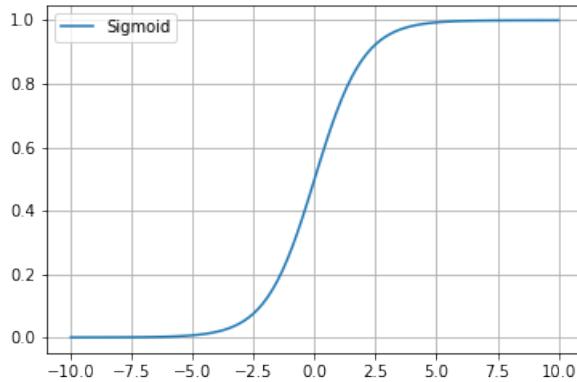
Come detto precedentemente, le funzioni di attivazione rappresentano un concetto cardine delle reti neurali. In particolare, la scelta delle funzioni di attivazione è una scelta fondamentale e può avere un grande impatto sulle performance del modello. All'interno di una rete, spesso vi sono diverse tipologie di funzioni di attivazione e nella maggior parte dei casi, se ne usano diverse tipologie per le diverse parti della rete. Tipicamente, gli strati nascosti utilizzano la stessa tipologia, mentre la scelta di quella dello strato di output dipende soprattutto dal task e dal tipo di output che bisogna fornire. Di seguito, le funzioni di attivazione più utilizzate e presenti in letteratura.

### 3.3.1 Funzione sigmoidea

Una delle più note è la funzione sigmoidea (Figura 3.6), anche detta sigmoid, una funzione matematica con la tipica curva ad S o curva sigmoid. In particolare, la sigmoid non fa altro che mappare l'input (un qualsiasi reale) su un range che va da 0 a 1: più l'input è grande e più sarà vicino a 1, mentre più sarà piccolo e più sarà vicino a 0. Di seguito la formula:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \quad (3.2)$$

La sigmoid viene spesso utilizzata nell'ultimo strato della rete neurale, soprattutto nei task di classificazione binaria, dove l'output finale della rete deve essere mappato nell'intervallo  $[0,1]$ , in quanto rappresenta la probabilità che l'input appartenga a una delle due classi predeterminate.



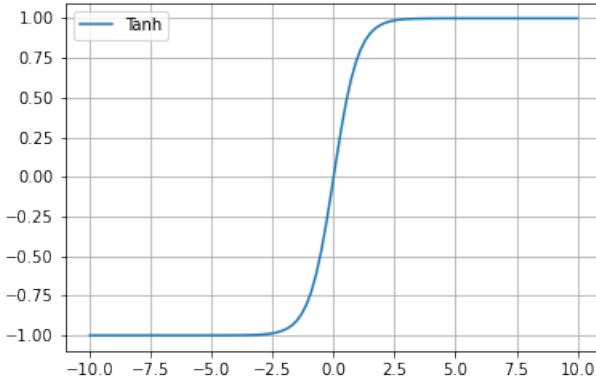
**Figura 3.6.** Grafico della funzione sigmoidea.

### 3.3.2 Tangente iperbolica

La tangente iperbolica (Figura 3.7), detta spesso “tanh”, è una funzione di attivazione molto simile alla sigmoid. La principale differenza è che mappa l'input nell'intervallo  $[-1, +1]$  invece che in  $[0, +1]$ .

### 3.3.3 Rectified Linear Activation Function

La ReLU (Rectified Linear Unit) (Figura 3.8) è forse la funzione di attivazione più utilizzata. Il motivo sta nel fatto che risolve uno dei principali problemi delle funzioni di attivazione, ovvero la scomparsa del gradiente. Quest'ultima riguarda il fatto che nel meccanismo di retropropagazione (che affronteremo più avanti), soprattutto



**Figura 3.7.** Grafico della tangente iperbolica.

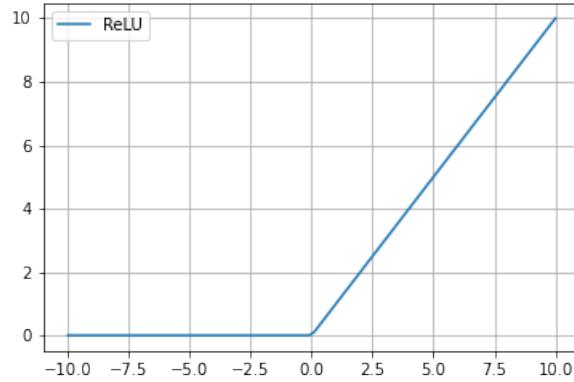
quando si hanno molti strati, i gradienti che vengono moltiplicati, se hanno valori piccoli, tendono ad andare verso lo 0 molto velocemente e questo porta ad avere cambiamenti, soprattutto nei primi strati della rete (gli ultimi nella retropropagazione), molto piccoli se non nulli. Di seguito la formula:

$$f(x) = x^+ = \max(0, x). \quad (3.3)$$

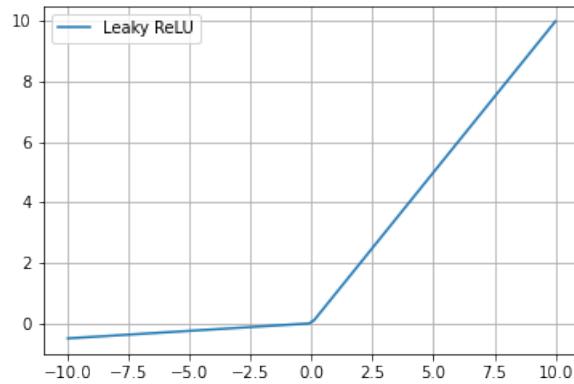
Inoltre la ReLU, vista la sua formula, risulta anche molto più semplice da calcolare rispetto alle altre e questa proprietà diventa critica soprattutto nelle reti neurali, dove spesso il costo computazionale diventa un ostacolo. Dall'altra parte, uno svantaggio della ReLU risiede nel fatto che per i valori negativi, la funzione ha una derivata nulla. Mentre questo può essere anche un punto di forza (per la sparsità della rete), rappresenta un problema quando molti dei valori in input alle funzioni di attivazione sono negativi perché molti dei neuroni vengono azzerati. Quando questo succede la rete tende a "morire", ovvero tende a non aggiornare più i suoi parametri (da qui il nome del problema Dying ReLU). Per risolvere questo problema, è stata introdotta una variante chiamata Leaky ReLU o ReLU Parametrica (Figura 3.9). In particolare, la variante, a differenza della versione originale, presenta nella parte negativa una piccola pendenza, che non è un parametro appreso durante l'addestramento, ma viene scelto in precedenza.

### 3.3.4 Softmax

Nei problemi di classificazione multiclasse, dove il numero di classi è maggiore di due, nello strato di output della rete non può essere usata la funzione sigmoid, che invece viene usata in ambito di classificazione binaria. Similmente alla sigmoid, l'output della softmax può essere interpretato come le probabilità associate ad ogni classe, infatti tutti i valori sommati restituiscono 1. Quando il numero di classi è uguale a due, la softmax è equivalente alla sigmoid e volendo si può considerare la



**Figura 3.8.** Grafico della ReLU.



**Figura 3.9.** Grafico della Leaky ReLU.

sigmoid un particolare caso della softmax o, in modo equivalente, la softmax una generalizzazione della sigmoid. Di seguito la formula della softmax.

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}. \quad (3.4)$$

## 3.4 Apprendimento

La fase di apprendimento è una fase in comune a tutti gli algoritmi di ML. In generale, in questa fase il modello che viene addestrato aggiorna i propri parametri cercando di migliorare le proprie performance, ovvero cercando di approssimare nel miglior modo possibile il pattern d'interesse.

### 3.4.1 Paradigmi di apprendimento

Nel campo del ML, esistono quattro principali paradigmi di apprendimento:

- *Supervisionato*: come detto in precedenza, l'approccio del ML si differenzia dall'approccio tradizionale nel fatto che chi sviluppa il modello non definisce le regole di funzionamento del mondo d'interesse, ma piuttosto si occupa di costruire un'architettura in grado di apprenderle dai dati. Detto ciò, risulta critico il ruolo dei dati. Infatti, soprattutto nel campo del DL dove i pattern del mondo d'interesse sono molto complessi, è fondamentale fornire al modello dati di quantità sufficiente, oltre che rappresentativi. In questo paradigma viene fornito al modello un insieme di dati, chiamato *dataset*, composto da sole coppie di tipo X e Y, dove la X rappresenta l'input e la Y rappresenta il corretto output. Le coppie che compongono il dataset vengono utilizzate dal modello come esempio del funzionamento del pattern che sta cercando di approssimare.
- *Non supervisionato*: in questo paradigma invece, nel dataset fornito c'è solo l'input senza il corretto output. Chiaramente, questo paradigma viene utilizzato solamente quando ottenere le corrispondenti label non è possibile oppure molto difficile. Un esempio di ciò è il campo delle immagini mediche, in cui solo degli specialisti possono produrre le label. Un altro esempio è quando la loro produzione risulta un processo molto lungo e tedioso, cosa che avviene spesso nell'ambito della segmentazione. Quando questo succede e il modello non ha esempi di input-output su cui basarsi, l'approccio più tipico è quello di cercare negli elementi del dataset delle similarità, in modo da poterli raggruppare in quelli che vengono chiamati *cluster*.
- *Semi Supervisionato*: questo paradigma è un punto d'incontro tra il paradigma supervisionato e quello non supervisionato. In particolare, in questo caso solo ad una parte del dataset è associata la corrispondente label.
- *Per rinforzo*: qui il modello, detto anche *agente*, apprende interagendo con l'ambiente esterno. In particolare, l'agente compie delle azioni e successivamente ne valuta i risultati, utilizzando un valore numerico di "ricompensa", che ha l'obiettivo di incoraggiare azioni corrette e scoraggiare quelle scorrette.

### 3.4.2 Funzioni di perdita

Nella fase di apprendimento, un aspetto critico è come il modello valuta le proprie performance. In particolare, ad ogni iterazione dell'apprendimento il modello deve avere la capacità di valutare quanto il suo output sia distante da quello corretto. Di conseguenza, il modello per apprendere ha bisogno di una misura di questa distanza. Questo ruolo è ricoperto dalla funzione di perdita, che viene decisa dagli sviluppatori prima della fase di addestramento. La scelta della funzione di perdita è fondamentale e può avere un grande impatto sulle performance del modello. Una caratteristica fondamentale della funzione di perdita è la derivabilità. In particolare, è fondamentale in quanto la sua derivata viene utilizzata proprio per addestrare la

rete (entremo nel dettaglio sul meccanismo di apprendimento più avanti). Di funzioni di perdita ne esistono molte tipologie che vengono scelte in base, soprattutto, al task:

- nei problemi di regressione, ovvero dove l'output è un valore all'interno di un intervallo continuo, una delle funzioni di perdita più comuni è la *MSE* (*Mean Squared Error*), che calcola la media delle differenze al quadrato tra gli elementi del vettore delle predizioni  $pred$  e gli elementi del vettore di output corretto  $y$ .

$$MSE = \frac{1}{n} \sum_{i=1}^n (pred_i - y_i)^2. \quad (3.5)$$

- nei problemi di classificazione binaria, una delle più utilizzate è la *Binary Cross Entropy*.

$$BCE = -\frac{1}{n} \sum_{i=1}^n (y_i \log(pred_i) + (1 - y_i) \log(1 - pred_i)). \quad (3.6)$$

- nei problemi di classificazione multiclasse, invece, viene spesso utilizzata la Cross Entropy

$$CE = -\frac{1}{n} \sum_{i=1}^n y_i \log(pred_i). \quad (3.7)$$

### 3.4.3 Retropropagazione dell'errore

Nelle reti neurali, così come in quasi tutti gli algoritmi di ML, la fase di apprendimento è una fase iterativa. Ad ogni iterazione, la rete, con i suoi parametri attuali, elabora uno o più elementi del dataset, confrontando poi l'output con la label corretta attraverso la funzione di perdita. A questo punto, la rete aggiorna i propri i parametri cercando di migliorare. Per capire come dovrebbe essere lo strato di output, abbiamo dei valori di riferimento, mentre per gli strati nascosti non abbiamo una diretta indicazione. L'aggiornamento dei parametri, generalizzata sia per lo strato di output che per quelli nascosti, è descritta dalla *regola delta*:

$$\Delta w_{jk} = -\eta \delta_j o_k. \quad (3.8)$$

Dove  $\Delta w_{jk}$  è la variazione dei parametri tra il neurone  $k$  e  $j$ ,  $\eta$  è il learning rate,  $o_k$  è l'output dl neurone  $k$ , ovvero l'input del neurone  $j$  e  $\delta_j$  è l'errore dell'output del neurone  $j$ . La differenza tra strato d'output e strato nascosto sta nel valore di  $\delta_j$ , ovvero per lo strato d'output è il seguente

$$\delta_j = y_j(1 - y_j)(o_j - y_j). \quad (3.9)$$

Dove  $o_j$  è l'output del neurone  $j$ , mentre  $y_j$  è l'output corretto. Dall'altra parte, per gli strati nascosti è equivalente a

$$\delta_j = y_j(1 - y_j) \sum_k \delta_k w_{kj}. \quad (3.10)$$

Dove invece  $\sum_k \delta_k w_{jk}$  è la somma pesata degli errori di tutti i neuroni collegati a  $j$ . Infine, tutti i neuroni aggiornano i propri parametri utilizzando la seguente formula:

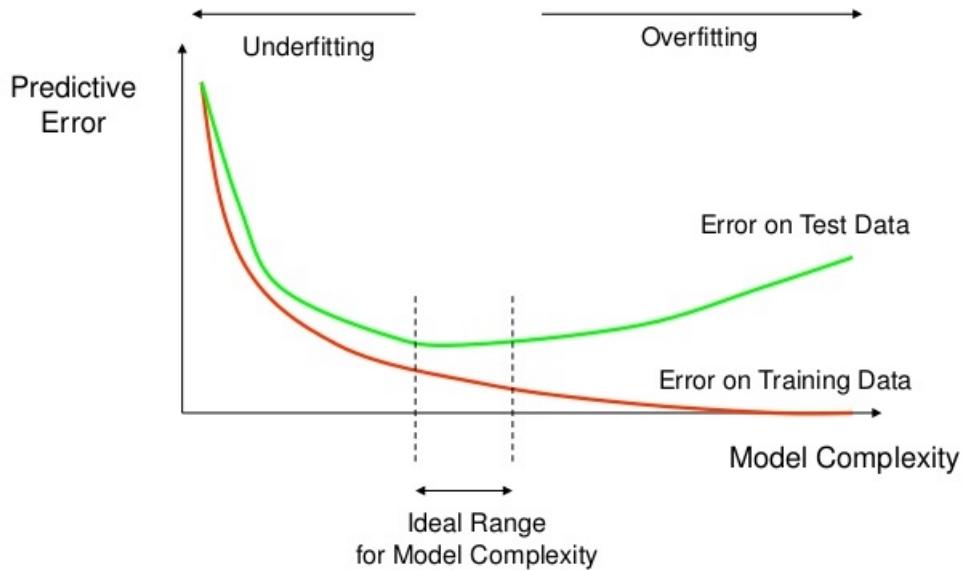
$$w_{jk} = w_{jk} + \Delta w_{kj}. \quad (3.11)$$

### 3.5 Regolarizzazione

In generale, nel campo del Machine Learning, così come nel campo dell'ottimizzazione di funzioni, quello che si cerca di fare è minimizzare una funzione, o in altri termini, minimizzare un errore. La principale differenza tra i due campi è che, nel ML, oltre all'errore sul training set (*training error*), viene anche preso in considerazione il *test error*, anche detto "errore di generalizzazione". In particolare, alla fine della fase di addestramento, è importante che il modello costruito abbia delle buone performance non solo sui dati di addestramento (training set), ma anche e soprattutto su dati mai visti (test set) e questa abilità viene chiamata generalizzazione. La capacità di un modello di saper generalizzare è molto importante e può dipendere da molti fattori. In particolare, oltre che dal giusto metodo e quantità di addestramento, dai giusti iperparametri e da molti altri fattori, la generalizzazione può soprattutto dipendere dalla scelta della complessità del modello. In generale, per quanto riguarda la complessità, il modello si può trovare in tre situazioni: nella prima il modello non ha approssimato bene il pattern interessato e la sua complessità è troppo bassa (*underfitting*), nella seconda invece il modello è riuscito a cogliere il vero pattern e infine, nella terza situazione, il modello costruito ha alzato troppo la sua complessità, approssimando un pattern più complesso ma che include quello interessato (*overfitting*) (Figura 3.10).

Nel caso della prima situazione ci sono diverse soluzioni, ma in generale vanno riviste le scelte di design del modello e in ogni caso va aumentata la sua complessità. Nel caso dell'*overfitting*, invece, un metodo molto utilizzato è quello della regolarizzazione, il cui obiettivo è proprio quello di portare il modello dalla terza alla seconda situazione. In generale, possiamo definire la regolarizzazione come qualsiasi modifica che apportiamo al nostro modello o all'algoritmo di apprendimento, con l'obiettivo di ridurre il suo errore di generalizzazione ma non quello di addestramento. Spesso nella pratica, soprattutto nei campi di applicazione del Deep Learning, anche con un modello molto complesso, non è detto che si riesca ad includere il pattern interessato. In particolare nel mondo del DL, i modelli sono applicati a domini estremamente complessi e, di conseguenza, quasi sicuramente il vero pattern è al di fuori di quello del modello. Questo comporta che nella maggior parte dei casi, la scelta migliore è quella di grandi modelli con un'alta complessità, affiancati dalle giuste tecniche di regolarizzazione [57].

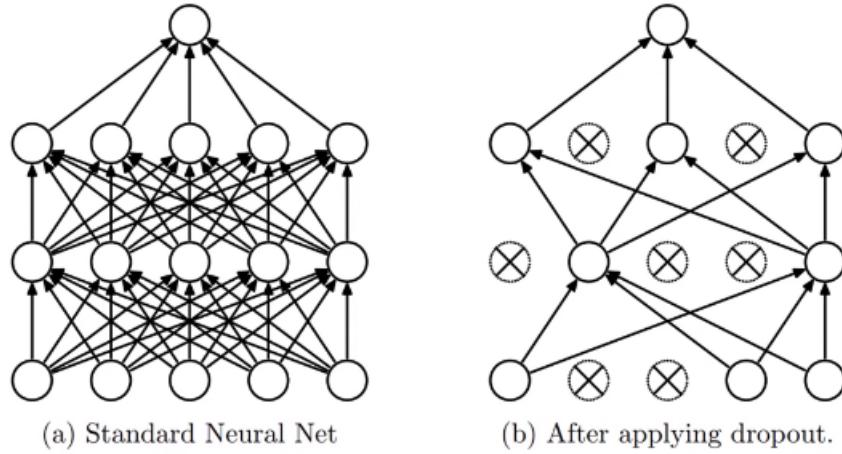
Di tecniche di regolarizzazione ne esistono svariate, ma uno dei metodi generali più utilizzati è quello di limitare la complessità del modello, anche detta capacità, aggiungendo alla funzione di perdita una componente di penalità  $\Omega(W)$ , trasformando la funzione di perdita, o anche detta funzione obiettivo,  $J(W; X, y)$  in  $\tilde{J}(W; X, y)$  dove:



**Figura 3.10.** Grafico degli errori sul training set e sul test set in funzione della complessità del modello e illustrazione dei due stati di underfitting e overfitting.

$$\tilde{J}(W; X, y) = J(W; X, y) + \alpha\Omega(W). \quad (3.12)$$

Dove  $\alpha \in [0, \infty]$  è un iperparametro che rappresenta il peso del contributo della penalità,  $W$  sono i parametri del modello,  $X$  rappresenta i dati di training e  $y$  l'output corretto. In particolare, questa componente rappresenta l'intenzione di voler penalizzare la complessità del modello, di conseguenza, con la sua aggiunta durante la fase di apprendimento, l'algoritmo non cerca solo di minimizzare il training error, ma anche la complessità del modello. Nel dettaglio, esistono diversi metodi di questo genere e in ognuno  $\Omega(W)$  prende una diversa forma. Due dei metodi più noti ed utilizzati di questa categoria sono la regolarizzazione L1 e la regolarizzazione L2. Un altro metodo di regolarizzazione molto usato nelle reti neurali è il *dropout*. Il funzionamento di questa tecnica è molto semplice e intuitivo: durante l'addestramento, con una certa probabilità  $p$ , alcuni nodi della rete vengono disattivati, trasformando così a tutti gli effetti la rete in una meno complessa e riducendo la sua complessità (Figura 3.11). Nel dettaglio, così come anche le altre tecniche di regolarizzazione menzionate prima, il dropout tende a tenere i parametri del modello più bassi possibili e lo fa evitando di far prevalere un nodo sugli altri. In particolare, disattivando a turno randomicamente i vari nodi, costringe la rete a non sbilanciare i propri parametri verso un nodo, ottenendo dei parametri più bilanciati e di conseguenza più piccoli, mentre senza il dropout, il rischio è che durante l'addestramento la rete possa aumentare di molto i parametri di un nodo e azzerare quelli di altri.



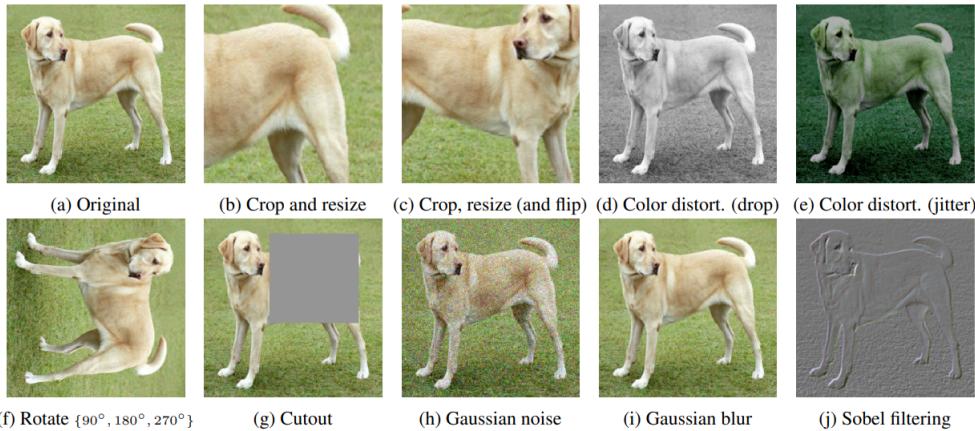
**Figura 3.11.** Illustrazione della tecnica di dropout.

### 3.5.1 Data augmentation

A parte le tecniche di regolarizzazione, il metodo sicuramente migliore per aumentare la capacità di generalizzazione di un modello è quello di fornigli più dati in input. Purtroppo però, come già detto, nella pratica i dati sono spesso molto difficili da reperire, soprattutto in alcuni campi di applicazione del DL. Spesso, per sopperire a questa difficoltà, si fa uso di una famiglia di tecniche di regolarizzazione chiamata *data augmentation*. In generale, per data augmentation si intende una qualsiasi tecnica con cui si producono dati finti, partendo da quelli veri. In particolare, partendo da uno o più elementi del dataset e applicando una trasformazione di qualche tipo, si può produrre un dato artificiale, ma comunque realistico. Questa tecnica viene spesso utilizzata nel campo della Computer Vision, dove le trasformazioni applicate ai dati sono ad esempio rotazioni, traslazioni, modifica del contrasto o della luminosità e molti altri (Figura 3.12). In questo campo, per alcuni task come la classificazione, la tecnica del data augmentation risulta spesso molto semplice, in quanto, dato che i classificatori devono risultare invarianti rispetto a molte trasformazioni dei dati [57], le trasformazioni fatte durante la data augmentation vanno applicate solamente alle immagini e non alle label, che invece devono rimanere uguali, cosa non vera, invece, nel campo della segmentazione semantica, dove andrebbero applicate in modo equo anche alle maschere.

Un'altra tipologia di data augmentation molto utilizzata è la *noise injection*, ovvero la tecnica nella quale viene randomicamente applicato un certo rumore ai dati del dataset. Questa tecnica risulta spesso molto importante, poiché l'invarianza dei modelli rispetto al rumore nel dato in input è una caratteristica molto importante e desiderata, e la sua mancanza può portare a dei risultati negativi [58], anche nell'ottica dell'Explainable AI. In particolare, spesso le immagini con l'aggiunta di un certo tipo di rumore risultano agli occhi umani identiche a quelle originali, di conseguenza il variare dell'output del modello da un'immagine all'altra può risultare un problema per l'interpretabilità del modello.

Come precedentemente menzionato, la data augmentation risulta più semplice per alcuni task, mentre per altri meno. Inoltre, recenti ricerche [59] hanno dimostrato



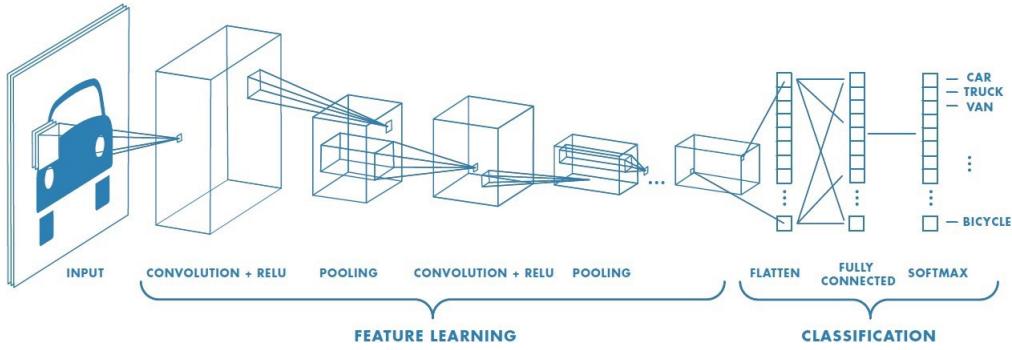
**Figura 3.12.** Alcuni esempi delle possibili trasformazioni per applicare data augmentation a un dataset d'immagini.

che questa tecnica può, da un lato portare a grandi miglioramenti nella capacità del modello di generalizzare, dall'altro penalizzare alcune classi. In particolare, i suoi effetti sono *class-dependent*, ovvero dipendono dalla natura delle classi del dataset, di conseguenza l'utilizzo di queste tecniche può portare a miglioramenti in alcune classi, ma anche ad un calo delle performance in altre, soprattutto quando si applicano le stesse trasformazioni all'intero dataset.

## 3.6 Reti Neurali convoluzionali

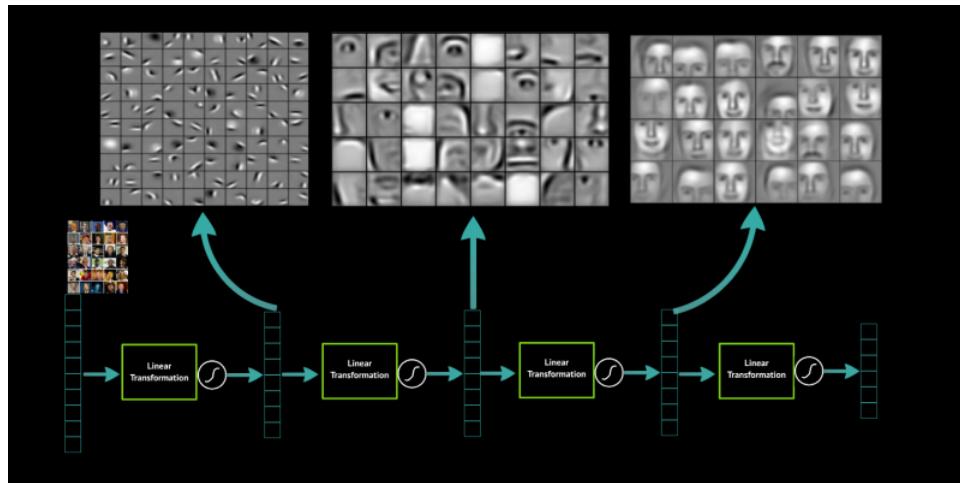
Le reti convoluzionali [60], come il percettore multistrato, presentano un layer d'input, uno di output e diversi layer nascosti. La differenza tra le reti classiche e quelle convoluzionali sta proprio nella presenza, tra quelli nascosti, degli strati convoluzionali (Figura 3.13). In particolare, le convoluzioni sono delle matrici la cui dimensione viene decisa in anticipo, che rappresentano dei filtri che vengono applicati sull'immagine scorrendo su di essa per estrarre determinati pattern. Uno dei principali vantaggi è che scorrendo sull'immagine, la convoluzione è in grado di riconoscere il pattern che ha appreso durante l'addestramento indipendentemente dalla posizione nell'immagine, a differenza delle normali reti neurali dove i neuroni, in particolare i primi, sono associati a precise porzioni dell'immagine.

Le convoluzioni applicate alle immagini, anche chiamate *kernel*, arrivano dal mondo della Computer Vision, dove vengono utilizzate per il processamento delle immagini. La differenza tra le convoluzioni tradizionali e quelle usate nelle reti neurali, sta nel fatto che mentre i valori di quelle tradizionali sono scelti in anticipo e pensati per applicare un determinato filtro, quelle nelle reti partono con dei valori inizializzati randomicamente, ed è proprio con l'addestramento che la rete impara quali sono i pattern da evidenziare. All'interno di una rete convoluzionale, solitamente, ci sono più strati convoluzionali e andando avanti nella rete, l'immagine in input diminuisce nelle prime due dimensioni (altezza e larghezza) e aumenta nella terza dimensione, ovvero quella del numero di feature, spesso chiamate canali. In particolare, ogni convoluzione applicata ad un'immagine produce una feature e più



**Figura 3.13.** Architettura di una generica rete convoluzionale.

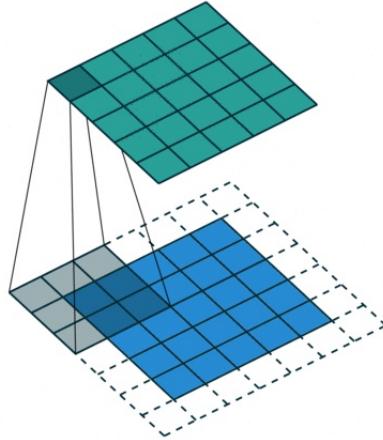
si va verso lo strato d'output, più le feature rappresentano pattern di più alto livello. Infatti, nei primi strati le feature rappresentano pattern più elementari, ad esempio forme geometriche come punti o linee. Negli strati più profondi invece, rappresentano strutture geometriche più complesse, oppure strutture con un significato semantico. La Figura 3.14 illustra in modo intuitivo il concetto di appena spiegato.



**Figura 3.14.** Illustrazione del concetto di features di basso e alto livello.

Nella fase di costruzione della rete o successivamente nella fase di ottimizzazione degli iperparametri, riguardo le convoluzioni vanno stabiliti i valori di tre iperparametri:

- *dimensione del kernel*: la dimensione della matrice della convoluzione.
- *stride*: indica il numero di pixel con cui la finestra si muove ad ogni operazione.
- *padding*: denota il processo di aggiunta di zeri a ciascun lato dell'input e questo iperparametro indica quanti aggiungerne. In particolare, il padding ha lo scopo di poter passare il kernel anche sui pixel più vicini ai bordi, e gli zero aggiunti servono proprio a riempire la parte del kernel che esce dall'input (Figura 3.15).



**Figura 3.15.** Illustrazione del padding.

Oltre agli strati convoluzionali, solitamente ci sono altre quattro tipologie di strati in una rete convoluzionale:

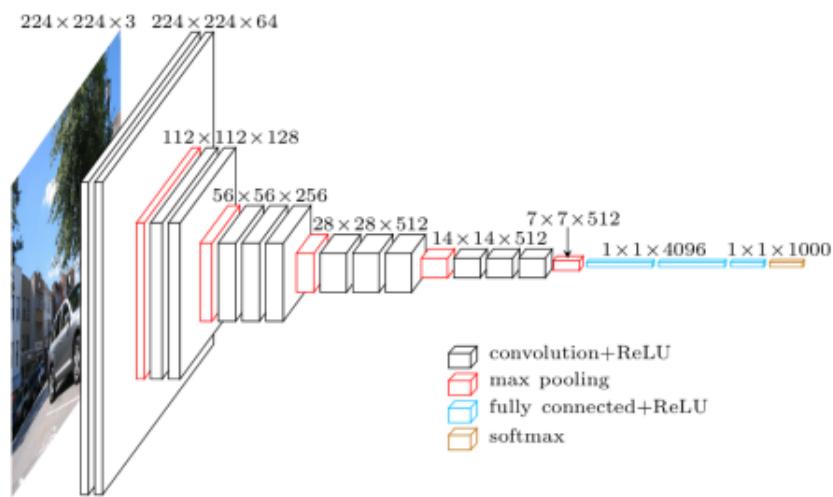
- *strato di pooling*: questo strato ha il ruolo di ridurre le prime due dimensioni del dato che attraversa la rete e anch'esso è un sorta di finestra che scorre sull'immagine, dando in output uno scalare. Ne esistono diverse tipologie, ma in generale, per una finestra dell'input restituisce un valore statistico di quest'ultima. Una delle tipologie più utilizzate è il *max pool*, che restituisce il valore massimo all'interno della finestra; un altro molto usato è l'*average pool*, che invece restituisce la media di tutti i valori nella finestra. A differenza della convoluzione, lo strato di pool non ha parametri apprendibili e il suo unico parametro è la dimensione della finestra, che più aumenta più riduce l'altezza e la larghezza del dato. L'importanza dello strato di pooling sta nel fatto che fornisce allo strato successivo l'invarianza rispetto a piccole traslazioni dell'input, ovvero modificando leggermente l'input dello strato di pooling, la maggior parte del suo output rimane invariato. Questa invarianza è fondamentale, soprattutto in alcuni task come la classificazione, in cui non è tanto importante dove sia la feature, ma piuttosto la sua presenza.
- *strato di attivazione*: questa è una tipologia di strato in comune con tutte i tipi di rete neurale. Infatti, come già detto, questo strato ha il ruolo di fornire la non linearità alla rete, fondamentale per approssimare pattern complessi. Nel caso delle reti convoluzionali, la funzione di attivazione più comune è la ReLU e segue spesso gli strati di convoluzione.
- *strato completamente connesso*: questa tipologia di strato, anche chiamato strato denso, è a tutti gli effetti strutturato come un percettore multistrato e si trova sempre alla fine della rete. Questa parte della rete è la parte responsabile della classificazione, ovvero le feature ad alto livello prodotte dai primi strati vengono passate in input ai neuroni dello strato completamente connesso, così da produrre l'output finale.

- *strato di attivazione finale*: questo strato rappresenta l'ultimo strato della rete ed ha il ruolo di traslare l'output nel range desiderato. Le due tipologie più utilizzate sono la *sigmoid*, nel caso di classificazione binaria, e la softmax, nel caso di classificazione multclasse.

### 3.6.1 Reti neurali convoluzionali avanzate

VGG

Il lavoro [61], che propose nel 2014 l’architettura chiamata VGG (Visual Geometry Group, ovvero il nome del gruppo di ricerca del lavoro) investigò il ruolo della profondità delle architetture nelle performance delle reti convoluzionali. Da questo studio ne risultò una delle architetture più note nel campo delle reti convoluzionali e della Computer Vision. Per quanto riguarda la struttura generale dell’architettura (Figura 3.16), che però presenta diverse versioni a seconda della profondità, è composta da un primo strato, ovvero quello che prende in input l’immagine, che ha una dimensione fissa di 224x224, di conseguenza con questa versione della rete, tutte le immagini date in input devono avere quella dimensione e questo aspetto è dovuto al fatto che alla fine della rete sono presenti degli strati densi, anche detti *fully connected* (FC), che non permettono alla rete di avere input di dimensioni diverse. Dopodichè, questo primo strato è seguito da uno stack di strati di convoluzioni 3x3, insieme, chiaramente, agli strati della funzione di attivazione (ReLU), con stride 1 e padding 1 (in modo da preservare la risoluzione spaziale dopo la convoluzione). In alcuni strati della rete sono presenti gli strati di pooling (max pooling), in particolare, in totale sono 5 e presentano una finestra 2x2 con stride 2. Infine, l’ultima parte della rete è composta da tre strati densi, di cui il primo e il secondo hanno una dimensione di 4096 canali, mentre l’ultimo ne ha 1000, poichè è stata costruita per essere testata su ImageNet che ha 1000 classi e un ultimo strato di Softmax.



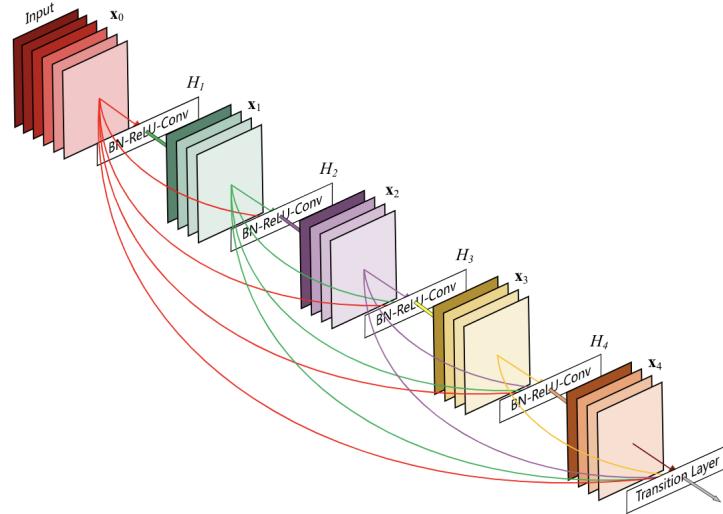
**Figura 3.16.** Architettura generica della VGG.

Come detto precedentemente, l'architettura appena descritta assume diverse forme a seconda della profondità dello stack di convoluzioni. In particolare, le

diverse versioni vanno da un minimo di 11 strati (8 strati di convoluzioni e 3 strati densi) fino ad un massimo di 19 (16 strati di convoluzioni e 3 strati densi), mentre l'ampiezza della rete (numero di canali) rimane coerente in tutte le versioni: parte da 64 fino ad arrivare negli ultimi strati di convoluzioni a 512. La differenza principale tra la VGG e le architetture precedenti è soprattutto l'uso di convoluzioni piccole. In particolare, precedentemente la tendenza era quella di aumentare il campo ricettivo con convoluzioni sempre più grandi, come ad esempio nell'AlexNet [54], che usa convoluzioni  $11 \times 11$  con stride 4, oppure nell'architettura proposta in [62] e nella OverFeat di [63], che usano convoluzioni  $7 \times 7$  con stride 2. I vantaggi che gli autori di [61] evidenziano riguardo l'uso di convoluzioni  $3 \times 3$ , sono principalmente dovuti al fatto che lo stesso campo ricettivo prodotto da una convoluzione  $7 \times 7$  è riproducibile con uno stack di tre convoluzioni  $3 \times 3$ , con il vantaggio di incorporare tre strati di attivazione, che rende la funzione totale più discriminativa, e al fatto che in questo modo si riducono il numero di parametri della rete. In particolare, uno stack di tre convoluzioni ha  $3(3^2 C^2) = 27C^2$  parametri, dove  $C$  è il numero di canali dell'input, mentre una convoluzione  $7 \times 7$  ha  $7^2 C^2$  parametri.

### DenseNet

Un'altra delle architetture più note è la DenseNet [64], un modello di rete convoluzionale ispirato al modello *feed-forward*. In particolare, la peculiarità della DenseNet è che, a differenza delle classiche reti convoluzionali all'interno delle quali con  $L$  strati si hanno  $L$  collegamenti, ovvero uno tra ogni strato e il suo successivo, questa architettura presenta  $\frac{L(L+1)}{2}$  collegamenti, ovvero uno tra ogni strato e tutti gli strati successivi (Figura 3.17).

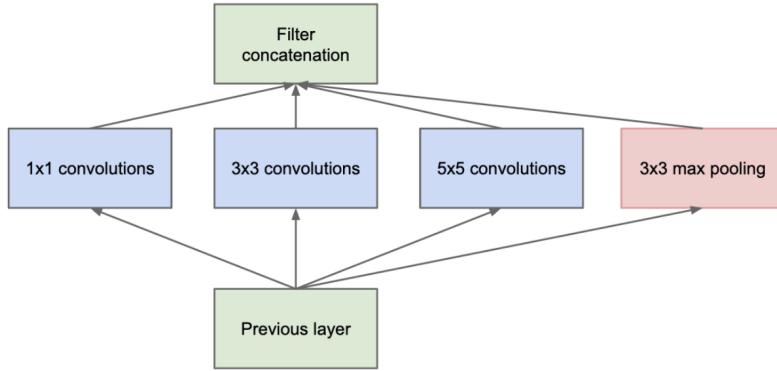


**Figura 3.17.** Struttura di un blocco della DenseNet: ogni strato prende in input gli output di tutti gli strati precedenti [64].

I vantaggi evidenziati dagli autori di [64] sono soprattutto riguardo uno dei maggiori problemi presenti nel campo delle reti neurali, ovvero la scomparsa del gradiente (*vanishing gradients* in inglese). Come già menzionato, questo problema riguarda il fatto che nel meccanismo di retropropagazione dell'errore, quando si ha una rete molto profonda, si rischia che l'informazione sul gradiente, che viaggia dagli ultimi strati della rete fino ai primi, svanisca passando attraverso gli strati e non arrivi ai primi. In realtà, lo stesso problema può essere visto da un'altra prospettiva, ovvero nelle reti molto profonde non sono solo le informazioni sul gradiente a rischiare di scomparire, ma anche le informazioni sull'input che viaggiano dai primi agli ultimi strati. Nel corso degli anni, fino alla pubblicazione della DenseNet, in altri lavori [65, 66, 67, 68] questo problema è stato affrontato con diverse tecniche, che però hanno tutte avuto una cosa in comune, ovvero l'utilizzo di brevi connessioni per connettere strati vicini. Il lavoro fatto dagli autori, infatti, ha cercato di distillare e generalizzare questi approcci proposti precedentemente, creando un semplice schema di connessioni con l'obiettivo di massimizzare il flusso di informazioni tra gli strati della rete. Inoltre, a differenza di [65], la DenseNet, per minimizzare la perdita delle informazioni che giungono da strati precedenti, non utilizza la somma come operazione di fusione di diversi input, bensì la concatenazione.

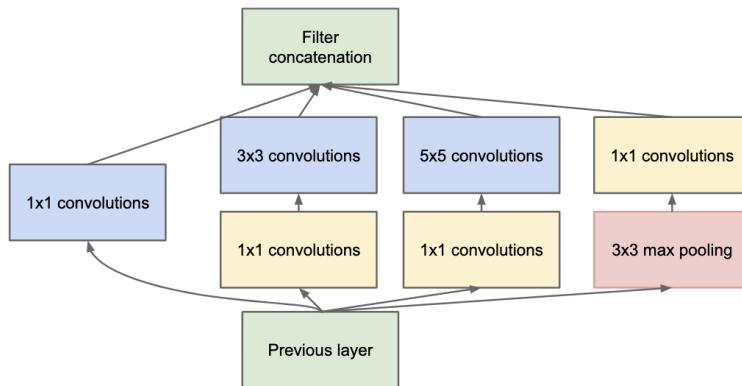
### Modulo Inception e GoogLeNet

Il lavoro [69], seguendo la tendenza di quegli anni di aumentare sempre di più la complessità delle reti convoluzionali per aumentare le performance, ha avuto l'obiettivo di trovare un metodo per aumentare la complessità delle reti, senza però aumentare il suo costo computazionale. In particolare, l'idea dietro il loro lavoro è stata quella di creare un modulo, chiamato modulo "Inception", all'interno del quale venissero utilizzate contemporaneamente tutte le tipologie di componenti (convoluzioni con diverse dimensioni, pooling, ...), che solitamente venivano alternate strato per strato. L'intuizione di questo meccanismo è che la scelta dell'operazione da utilizzare per ogni strato non è più di chi costruisce il modello, ma del modello stesso, che ad ogni strato non è più limitato all'informazione della singola operazione fatta in quello strato, ma ha a disposizione le informazioni risultato di diverse operazioni fatte sullo stesso input. In particolare, la scelta degli autori su quali operazioni fare in parallelo nel modulo Inception è ricaduta su: convoluzione 1x1, convoluzione 3x3, convoluzione 5x5 e infine max pooling. Questa scelta è frutto di esperimenti fatti su diverse combinazioni, che come risultato hanno giudicato questa come la migliore. In realtà però, nei successivi sono state proposte diverse varianti e nuove versioni del modulo Inception, che hanno utilizzato diverse combinazioni [70, 71, 72, 73]. Entrando nel dettaglio del suo funzionamento, il modulo Inception si basa sul dare lo stesso input parallelamente alle quattro diverse operazioni, i cui risultati vengono poi concatenati e passati al prossimo strato (Figura 3.18). Per concatenare i quattro diversi output, chiaramente le loro prime due dimensioni (altezza e larghezza) devono coincidere, per far questo le tre operazioni di convoluzioni hanno dei parametri di stride e padding mirati. Per quanto riguarda il maxpooling, invece, che a differenza della convoluzione riduce per forza altezza e larghezza dell'input, è necessaria una certa quantità di padding per rendere il suo output coerente con gli altri.



**Figura 3.18.** Illustrazione del modulo Inception [69], la versione senza riduzione delle dimensioni con costo computazionale molto alto.

Il problema di questa versione del modulo è che, aumentando di molto l'ampiezza degli strati della rete senza nessun tipo di accorgimento particolare, anche il suo costo computazionale è aumentato. Per far fronte a questo aumento, gli autori hanno proposto un approccio che si basa sull'utilizzo di convoluzioni 1x1 per ridurre le dimensioni dell'input. In particolare, per le operazioni computazionalmente più costose, ovvero la convoluzione 3x3 e la convoluzione 5x5, aggiungono prima una convoluzione 1x1, chiamata in questo caso *bottleneck*, che ha lo scopo di portare l'input ad una dimensione per cui applicare le convoluzioni ha un costo computazionale meno elevato. Inoltre, per quanto riguarda il maxpooling, la convoluzione 1x1 viene aggiunta dopo, dato che in questo caso non è tanto il costo computazionale dell'operazione a creare problemi, ma piuttosto la dimensione del suo output (che ha lo stesso numero di canali dell'input), il max pooling è seguito da una convoluzione 1x1, per far in modo di ridurre il numero di canali del suo output (Figura 3.19).



**Figura 3.19.** Illustrazione del modulo Inception [69], la versione con la riduzione delle dimensioni attraverso strati di convoluzioni 1x1.

Oltre alla concezione del modulo Inception, gli autori di [69] hanno anche proposto un'intera architettura basata proprio su questo tipo di modulo, la "GoogLeNet", il cui nome rende omaggio alla LeNet [74], una delle prime reti neurali convoluzionali sviluppate. La GoogleNet (Figura 3.20), in particolare, è essenzialmente composta da uno stack di moduli Inception. Una sua peculiarità, al di là dell'uso del modulo Inception, è la presenza di due ulteriori strati intermedi di output. In particolare, la rete presenta in due dei moduli Inception dei rami addizionali di output, che terminano con uno strato di Softmax, trasformando l'output dei due strati intermedi in output effettivi della rete. A renderli due output effettivi della rete, è il fatto che la loss function totale, che la rete utilizza per addestrarsi, è calcolata con una somma pesata di tutti e tre gli output (questi due più quello finale), dando più peso a quello finale. Così facendo, la loss function constringe la rete ad avere le feature map di quei due strati intermedi, già di una buona qualità per fare una predizione e l'idea dietro questo meccanismo è che si ottiene un effetto di regolarizzazione, spingendo la rete a non cadere nell'overfitting.

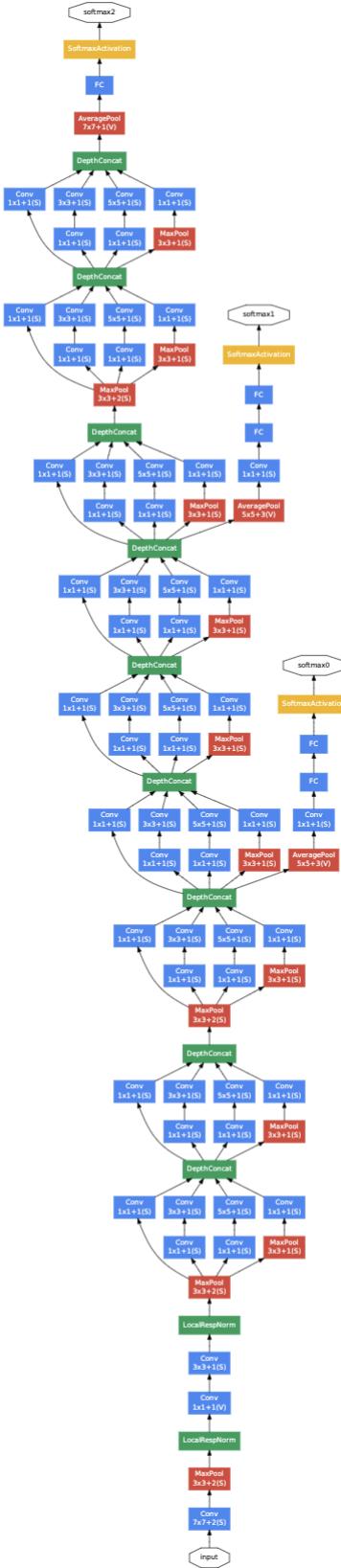


Figura 3.20. Architettura della GoogLeNet [69].

## Capitolo 4

# Architettura e metodi del lavoro

In questo capitolo vengono illustrati tutti i metodi utilizzati durante questo lavoro per affrontare la segmentazione semantica delle immagini del dataset FloodNet. In particolare, l'intero lavoro si è basato sul cercare di risolvere le principali difficoltà trovate nell'affrontare questo dataset, che verranno approfondite nel Paragrafo 4.1. Per quanto riguarda l'architettura utilizzata, l'idea base è stata quella di utilizzare un'architettura di Deep Learning, ovvero una rete neurale. La motivazione principale di questa scelta la si può trovare nel fatto che, come già evidenziato soprattutto nel paragrafo 3, le reti neurali riescono ad approssimare pattern molto più complessi rispetto ai metodi più tradizionali. Infatti, i lavori più attuali, che affrontano task simili a quello di questo lavoro 1.2, utilizzano per la maggior parte metodi di Deep Learning. Nello specifico, la scelta dell'architettura DeepLabV3 [36], si è basata invece sull'ovviare a due delle principali problematiche affrontate nel dataset (Paragrafo 4.1).

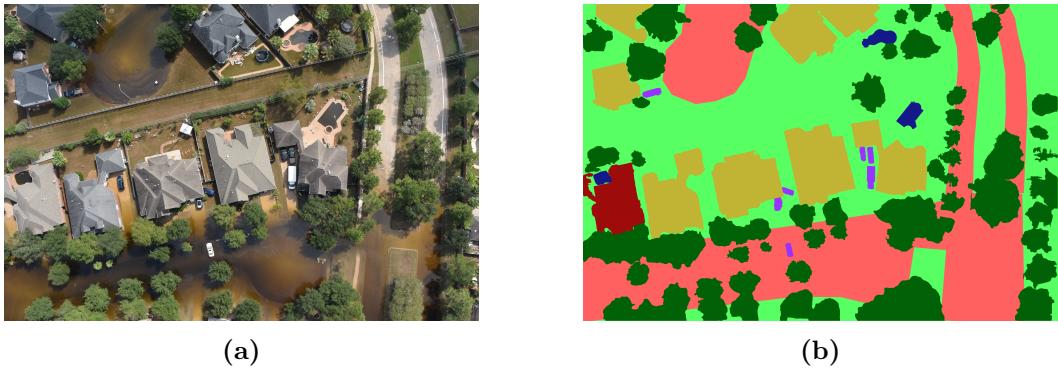
### 4.1 Difficoltà affrontate nel dataset

Come spesso accade, le principali difficoltà affrontate durante questo lavoro sono state inerenti al dataset utilizzato. In particolare, questo aspetto è piuttosto comune, ovvero una grande porzione del tempo speso in questa tipologia di lavori spesso consiste nell'analizzare le caratteristiche del dataset per capirne le peculiarità, ma soprattutto le difficoltà. Riguardo le specifiche difficoltà affrontate nel dataset FloodNet, di seguito se ne riportano le quattro principali:

- le nove classi presenti all'interno del dataset sono fortemente sbilanciate. In particolare, alcune classi, come "prato" e "albero", sono molto più presenti nel dataset rispetto ad altre classi, ovvero il numero di immagini che le contengono è molto più alto rispetto a quello delle altre.
- oltre allo sbilanciamento dal punto di vista delle occorrenze, le classi sono anche fortemente sbilanciate dal punto di vista della scala. In particolare, alcune classi, che sono le stesse del problema menzionato sopra, sono rappresentate da regioni molto più grandi rispetto ad altre. Ad esempio, la classe "veicolo", così come la classe "piscina", è rappresentata da oggetti molto più piccoli rispetto a quelli della classe "prato" o "albero" e di conseguenza, aggiungendo il

problema menzionato al punto precedente, il numero di pixel di queste classi all'interno del dataset risulta notevolmente inferiore rispetto a quello di altre.

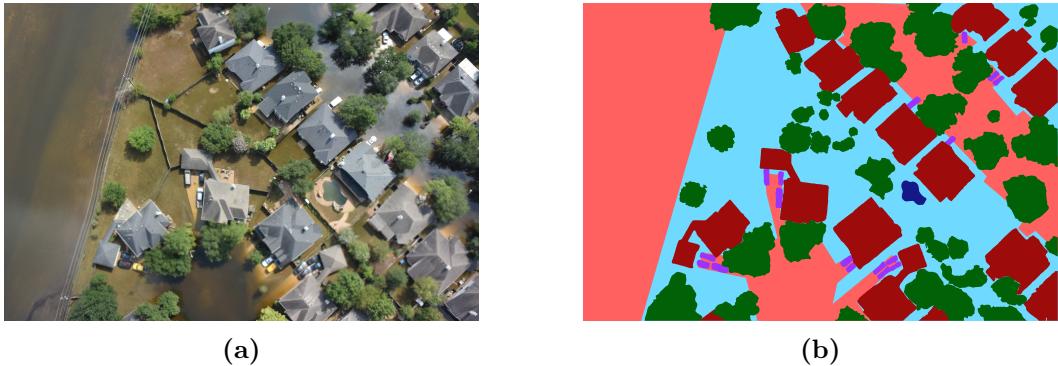
- alcune classi, più di altre, presentano una maggiore difficoltà intrinseca. In particolare, le due classi che sono risultate più difficoltose da apprendere sono state "strada allagata" e "edificio allagato". La motivazione, oltre ad essere presenti in misura minore all'interno del dataset, è soprattutto la loro natura semantica. Per quanto riguarda la classe "edificio allagato", la sua semantica deriva totalmente dal suo contesto e per nulla dalle sue caratteristiche locali. In particolare, se prendiamo solamente i pixel di un edificio, uno allagato è indistinguibile da uno non allagato, in quanto l'essere allagato o meno deriva dalla presenza di acqua di alluvione intorno all'edificio. Per quanto riguarda invece la classe "strada allagata", la sua semantica può derivare sia dal suo contesto, sia dalle sue caratteristiche locali, ma in alcuni casi grandi porzioni di una strada allagata possono essere indistinguibili da una non allagata. Nello specifico, una strada per essere considerata allagata non deve necessariamente presentare acqua in tutte le sue parti, ma è sufficiente una sola porzione allagata (Figura 4.1). Inoltre, un'ulteriore difficoltà presente in entrambe le due classi sopra citate deriva dal fatto che la loro definizione risulta più vaga rispetto ad altre classi, prestandosi maggiormente a interpretazioni, caratteristiche che, nelle maschere del dataset, si presentano sotto forma di ambiguità.



**Figura 4.1.** Le due figure mostrano un esempio di come una strada allagata può presentare grandi porzioni senza acqua (nella parte in alto a destra), risultando indistinguibile da una strada allagata se non consideriamo il suo contesto.

- il dataset al suo interno presenta una grande quantità di rumore, ovvero in alcune maschere sono presenti degli errori. Inoltre, la maggior parte di questi errori sono proprio in immagini che presentano abbondanza di quelle classi menzionate nel punto precedente ("edificio allagato" e "strada allagata"). Di conseguenza, viste già le problematiche menzionate nei punti precedenti, la difficoltà nell'apprenderle aumenta ancora di più. Nello specifico, la maggior parte degli errori trovati nelle maschere del dataset può essere diviso in tre categorie:

- la prima tipologia di errore è rappresentata da maschere all'interno delle quali alcuni pixel sono classificati erroneamente. In particolare, i due casi più frequenti sono quelli in cui la classe "strada allagata" oppure la classe "prato" vengono invece classificati come "acqua" (Figura 4.3).

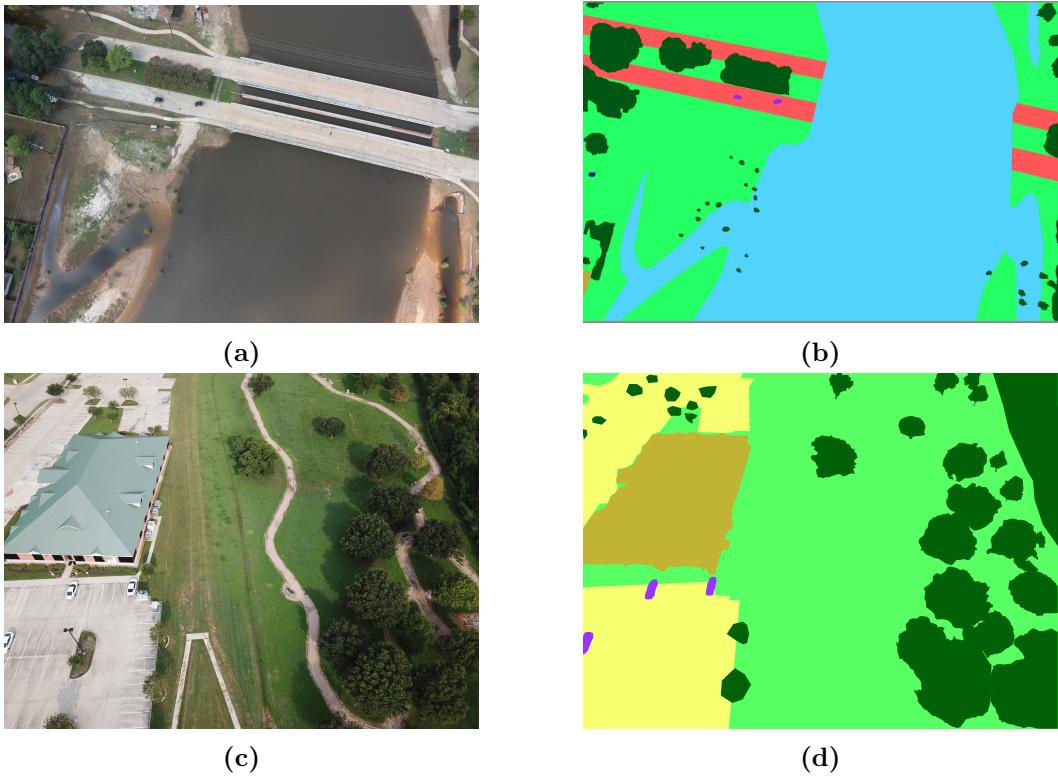


**Figura 4.2.** La figura mostra un esempio di errore presente nel dataset. In particolare, come si può notare i pixel che dovrebbero essere classificati come "prato" vengono invece classificati come "acqua".

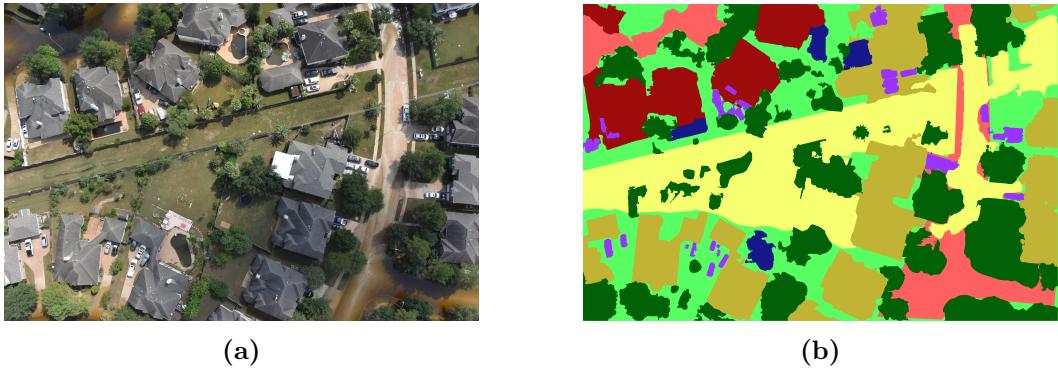
- la seconda categoria invece, riguarda quei casi in cui alcune occorrenze di classi o parti di esse non vengono rappresentate nella maschera, ma i corrispettivi pixel vengono invece classificati come appartenenti ad altre classi (Figura 4.3). I casi più frequenti riguardano le classi "strada" e "strada allagata".
- il terzo caso riguarda invece, la presenza di porzioni d'immagine fortemente incoerenti tra loro, oltre che con un certo grado di "confusione" tra le classi. La Figura 4.4 ne illustra un esempio.

## 4.2 Pulizia del dataset e Data Augmentation

Per far fronte a due delle quattro principali difficoltà menzionate nel paragrafo precedente, si sono messe in atto due fasi principali di pulizia del dataset (*data cleaning*) e di data augmentation offline. In particolare, queste due fasi sono risultate fondamentali per fornire un dataset composto da immagini elevate sia da un punto di vista qualitativo che quantitativo, fattore in generale indispensabile per addestrare una rete neurale. Partendo dalla pulizia del dataset, che è servita soprattutto ad ovviare al problema della presenza degli errori, questa fase è stata lunga e articolata, in quanto è stata effettuata una scansione manuale, confrontando ogni immagine del dataset con la sua maschera corrispondente, al fine di individuare eventuali errori. Da questa scansione, si è riscontrato che in totale 182 immagini presentavano errori importanti, come quelli descritti nel paragrafo precedente. Di queste 182, ben 160 erano immagini contenenti entrambe le classi "allagate" (strada ed edificio). Confrontando quest'ultimo dato con il numero totale di immagini contenente queste due classi (all'incirca 200), si può notare come, oltre alle difficoltà intrinseche delle due classi, questa notevole presenza di errori le abbia fortemente svantaggiate. In

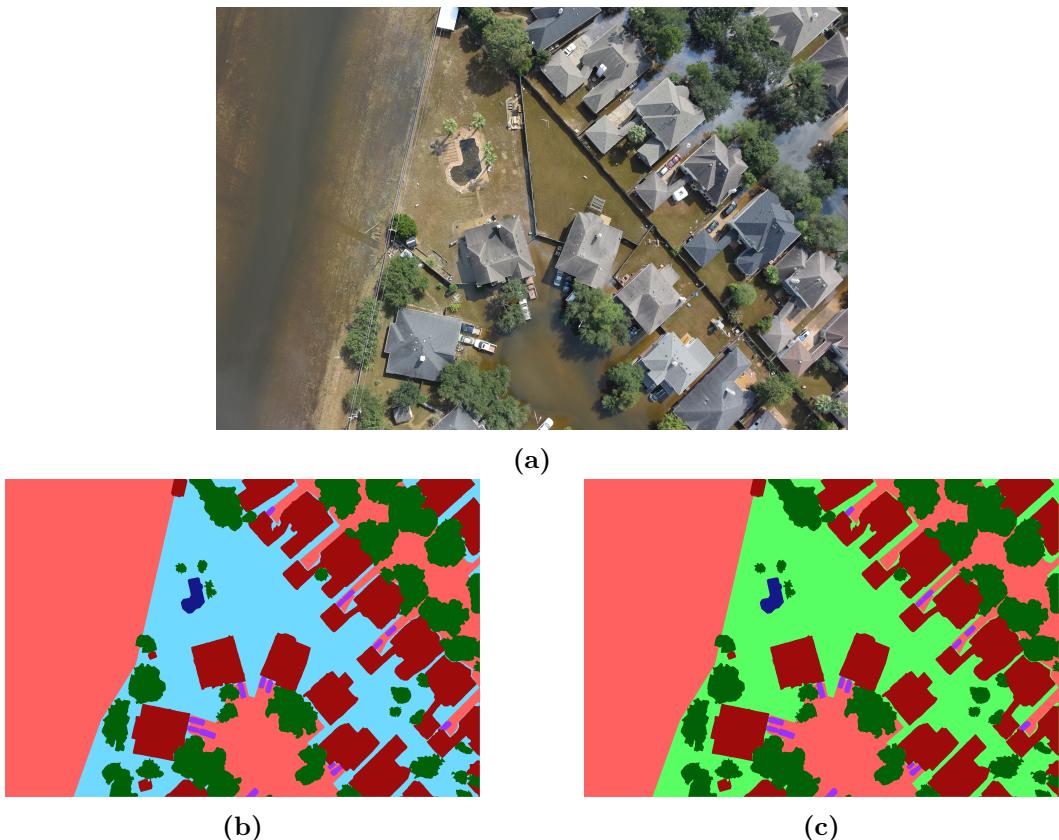


**Figura 4.3.** La figura mostra un esempio della seconda tipologia di errore presente nel dataset. In particolare, come si può notare, nella figura (b) una grossa porzione di un’occorrenza della classe “strada allagata” è mancante e i corrispettivi pixel vengono classificati come “acqua”. Nella figura (d) invece un’occorrenza della classe “strada” non viene segnalata e i corrispettivi pixel vengono classificati come “prato”.



**Figura 4.4.** La figura mostra un esempio della terza tipologia di errore presente nel dataset. In particolare, come si può notare, nella parte destra della maschera c’è un incoerenza rappresentata dal fatto che porzioni diverse della stessa strada vengono classificate come “strada non allagata” e “strada allagata”, anche se vi è presenza di acqua. Inoltre, alcuni edifici circondati da strade allagate vengono classificati come “edificio non allagato”. Infine, nella parte centrale dell’immagine viene segnalata un’occorrenza della classe “strada non allagata”, quando in realtà nell’immagine non risulta.

seguito a questa fase di accertamento della consistenza del problema, si è proseguito con la fase di correzione. Durante questa fase, una parte degli errori è risultata corregibile attraverso diversi metodi, a seconda della natura dell'errore, altri invece, a causa della loro natura non hanno permesso la correzione in tempi non troppo lunghi e di conseguenza, per non rallentare troppo il lavoro, sono state scartate. In particolare, delle 182 totali, 45 non state corrette ma scartate. La Figura 4.5 mostra un esempio di come una maschera contenente un errore sia stata corretta.



**Figura 4.5.** Le tre figure sono un esempio di correzione fatto sulle maschere del dataset FloodNet. La figura (a) è l'immagine a cui si riferiscono le due maschere (b) e (c). Come si può notare, nella (b) una porzione di prato è invece classificata come acqua (celeste), mentre la (c) è la versione corretta.

Mentre la fase di pulizia ha cercato di ovviare soprattutto alla quarta problematica, la fase successiva di data augmentation offline ha invece cercato di risolvere soprattutto la problematica dello sbilanciamento verso alcune classi.

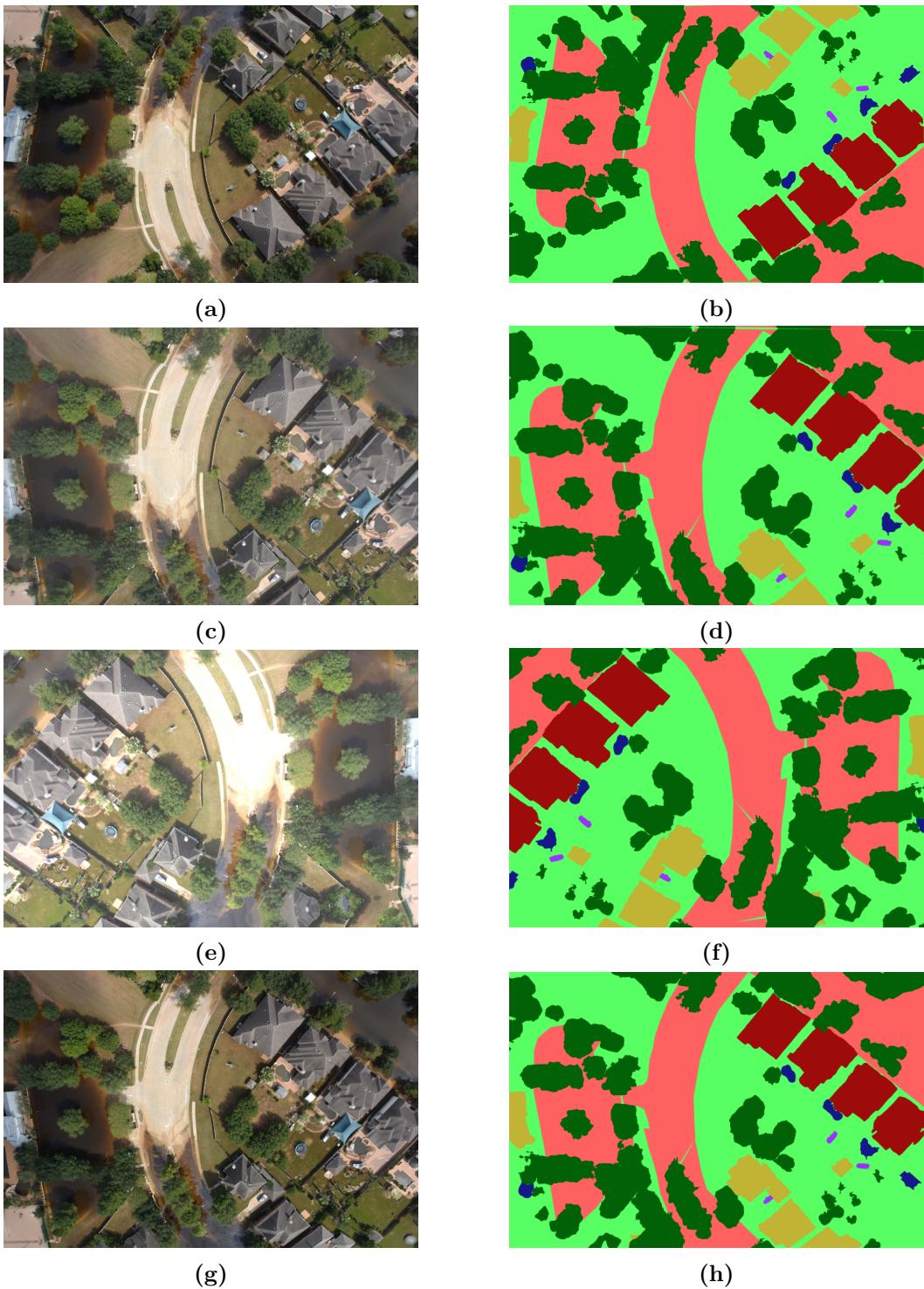
Come evidenziato dagli autori di [59], la data augmentation può creare degli effetti indesiderati, svantaggiando una classe rispetto ad altre. Per questo motivo, questa fase di data augmentation offline non è stata fatta su tutto il dataset, ma su un gruppo di immagini selezionate. In particolare, come già detto, lo scopo della selezione è stato di aumentare la presenza di quelle classi che riguardano la problematica dello sbilanciamento, specialmente le classi "strada allagata" e "edificio allagato", senza però sbilanciare ancora di più il dataset verso classi molto pre-

senti, come la classe "prato". Per l'implementazione di questa data augmentation è stata utilizzata la nota libreria Albumentations [75] e il meccanismo generale è stato il seguente: ad ognuna delle 140 immagini selezionate è stata applicata per tre volte una combinazione randomica di quattro operazioni (non sempre tutte e quattro), producendo per ogni immagine ulteriori tre (Figura 4.6). In particolare, le trasformazioni utilizzate sono: *Rotate*, che consiste nella rotazione di una quantità randomica di gradi dell'immagine; *VerticalFlip*, ovvero l'immagine viene ruotata su sé stessa sull'asse che la attraversa orizzontalmente; *HorizontalFlip*, ovvero la stessa trasformazione ma sull'asse verticale; e infine *RandomBrightnessContrast*, che consiste nell'alterare randomicamente la luminosità e il contrasto dell'immagine.

Oltre ad utilizzare la data augmentation offline, per aumentare il numero di immagini del dataset ed ovviare al problema dello sbilanciamento, è stata utilizzata una seconda fase di data augmentation online, ovvero durante l'addestramento. Lo scopo, in questo caso, non era aumentare il numero di immagini ma ottenere un effetto regolarizzante apportando ad ogni epoca dell'addestramento una diversa modifica ad ogni immagine. L'intuizione dietro questo metodo è che in questo modo, ad ogni epoca il modello vede una versione del dataset leggermente diversa e questo evita che andando avanti nell'addestramento il modello cada nel meccanismo di overfitting. In particolare, anche qui per l'implementazione è stata utilizzata la libreria Albumentations e il meccanismo generale è il seguente: durante un'epoca, ad ogni caricamento di un'immagine per la costruzione di una batch viene applicata un'operazione di *VerticalFlip*, una di *HorizontalFlip* e infine una di *RandomBrightnessContrast*. La chiave di questo metodo è che, come nella data augmentation fatta offline, ad ogni trasformazione viene associata una probabilità, di conseguenza ad ogni epoca non vengono applicate tutte e tre le trasformazioni, ma solo un sottoinsieme randomico delle tre, creando nella pratica una versione diversa del dataset ad ogni epoca.

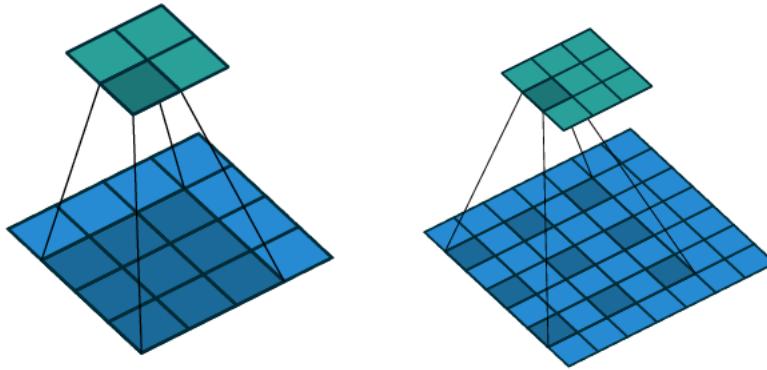
### 4.3 DeepLabV3

Le principali difficoltà che gli autori di [76, 49, 36] hanno evidenziato riguardo l'applicazione delle DCNN al task generale della segmentazione semantica sono due: la prima si riferisce alla bassa risoluzione delle feature map prodotte dalle DCNN, mentre la seconda riguarda la presenza all'interno dell'immagine di oggetti a diversa scala. In particolare, soprattutto la seconda difficoltà presenta un forte parallelo con una delle principali problematiche affrontate in questo lavoro. Ed è proprio questa la motivazione per cui si è scelto di utilizzare questo modello. Riprendendo il discorso delle difficoltà delle DCNN nella segmentazione semantica, la prima difficoltà è soprattutto causata dalla combinazione e dall'alternarsi di strati di pooling e convoluzioni con stride. In particolare, questa è una caratteristica che rappresenta uno dei punti di forza delle DCNN in campi come la classificazione, dove ridurre la risoluzione delle feature map serve proprio ad apprendere rappresentazioni molto astratte del dato e ad acquisire l'invarianza rispetto a sue trasformazioni locali. Chiaramente, questo punto di forza si trasforma in una debolezza nel task della segmentazione, in cui la localizzazione è fondamentale. La seconda difficoltà invece, è causata dalla natura dei dati e riguarda sostanzialmente la presenza nell'immagine di oggetti di



**Figura 4.6.** Le otto figure mostrano un esempio della data augmentation utilizzata offline prima dell’addestramento. Le figure (a) e (b) sono l’immagine e la maschera originali, mentre tutte le altre sono il risultato dell’applicazione di un sottoinsieme delle operazioni Rotate, HorizontalFlip, VerticalFlip e RandomBrightnessContrast.

diverse dimensioni. Per risolvere la prima problematica, gli autori propongono un approccio basato sull'utilizzo di una particolare tipologia di convoluzione, ovvero la convoluzione dilatata (*dilated convolution* o anche *atrous convolution*). In particolare, gli autori propongono di rimuovere la fase di downsampling e utilizzare una convoluzione il cui kernel ha subito un upsample, inserendo dei vuoti al suo interno (Figura 4.7).



**Figura 4.7.** Illustrazione della differenza tra una convoluzione dilatata (a destra) e una normale convoluzione (a sinistra).

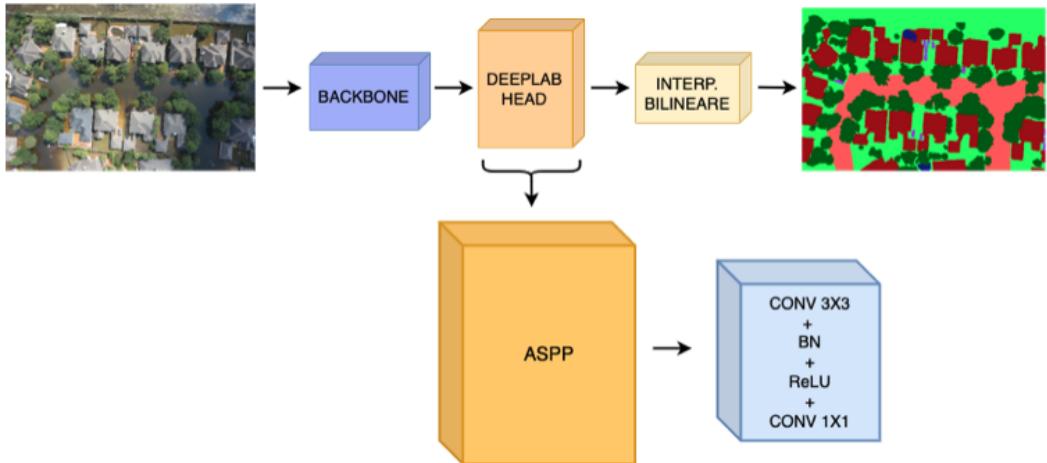
Come detto in precedenza, la motivazione principale della scelta di questo modello specifico, consiste nel fatto che il lavoro degli autori mira proprio a risolvere il problema degli oggetti a diversa scala. In particolare, ispirandosi all'idea del SPP di [15], propongono una simile struttura chiamata ASPP (Atrous Spatial Pyramid Pooling), che si basa sul concetto di utilizzare convoluzioni dilatate con diversa dilatazione in parallelo. Oltre a risolvere tale problematica, la DeepLabV3, essendo uno dei più noti modelli della categoria context-based 2.3.1, si presta a risolvere anche la problematica della difficoltà intrinseca delle due classi "edificio allagato" e "strada allagata". Per quanto riguarda la specifica versione dell'architettura utilizzata in questo lavoro, per la prima parte della rete, chiamata *backbone*, che ha il ruolo di produrre le feature map che verranno poi passate all'ASPP per produrre la maschera finale, viene utilizzata la ResNet101, ovvero una versione specifica dell'architettura basata su strati residui proposta in [65].

### 4.3.1 Architettura totale

Partendo dalla struttura generale dell'architettura, essa è composta da:

- **backbone:** prima parte della rete responsabile della feature extraction. Produce a partire dall'immagine in input una feature map di 2048 canali. In particolare, viene utilizzata una versione modificata con convoluzioni dilatate della ResNet101.
- **DeepLabHead:** seconda parte della rete, responsabile della produzione della maschera finale. Composta dall'ASPP e da un blocco convoluzionale che produce un volume  $9 \times H \times W$ , dove 9 è il numero delle classi.

- **Interpolazione bilineare:** responsabile dell'ultima fase di upsampling, grazie alla quale l'output della DeepLabHead viene portata alle dimensioni originali dell'immagine.

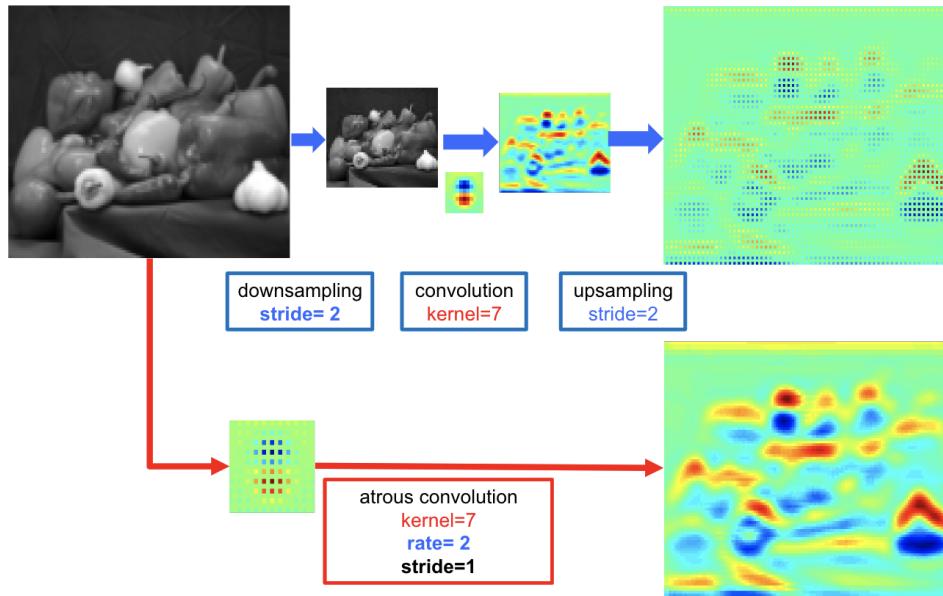


**Figura 4.8.** Illustrazione dell'architettura totale utilizzata.

### 4.3.2 Convoluzione Dilatata

Come già detto, all'interno della DeepLabV3, la convoluzione dilatata viene utilizzata per far fronte al problema della ridotta risoluzione delle feature map. In particolare, il problema nasce quando, nella seconda parte della rete, le feature subiscono un upsample per tornare alla risoluzione originale e produrre l'output finale. Nello specifico, la qualità della maschera risultante è scarsa e non contiene informazioni spaziali molto dettagliate. Con l'utilizzo della convoluzione dilatata invece, accoppiata con l'interpolazione bilineare per la fase di upsample, la maschera prodotta contiene informazioni spaziali più dettagliate (Figura 4.9).

Oltre al discorso inerente la risoluzione delle feature map, un altro vantaggio delle convoluzioni dilatate, molto utile nel contesto di questo lavoro, è la capacità di aumentare la dimensione di una convoluzione senza aumentare il numero di parametri e di conseguenza senza aumentare il costo computazionale dell'architettura, che diventa spesso un problema. In particolare, una delle funzioni principali delle convoluzioni è catturare il contesto di un pixel, ovvero la regione intorno al pixel, da cui dipende fortemente la sua semantica. Spesso però, il contesto di un pixel è più ampio di quello che la finestra della convoluzione, chiamata *campo ricettivo*, può catturare e qui entra in gioco il vantaggio delle convoluzioni dilatate. In particolare, entrando nel dettaglio del loro funzionamento, esse presentano un parametro in più rispetto alla convoluzione tradizionale, chiamato dilatazione, che rappresenta la quantità di vuoti che vengono inseriti nel kernel. Più questo parametro aumenta, più la finestra si ingrandisce e più aumenta l'ampiezza del contesto catturato, non cambiando però il numero effettivo di parametri. Inoltre, possiamo considerare le convoluzioni



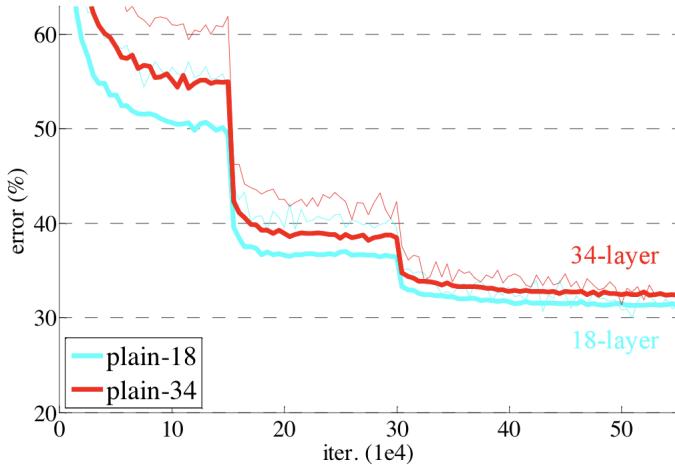
**Figura 4.9.** Illustrazione della convoluzione dilatata in 2-D. Come si può notare, la maschera prodotta nella riga superiore (convoluzione classica) presenta informazioni più sparse e meno dettagliate rispetto alla maschera prodotta nella riga inferiore (convoluzione dilatata e interpolazione bilineare).

dilatate come una generalizzazione delle normali convoluzioni, in quanto le seconde sono un caso particolare delle prime, ovvero con la dilatazione impostata a 1 una convoluzione dilatata diventa una normale convoluzione. Generalizzando al caso di dati 1-D, l'output  $y[i]$  di una convoluzione dilatata con parametri  $w$  di lunghezza  $k$ , che prende in input  $x[i]$  è definita come:

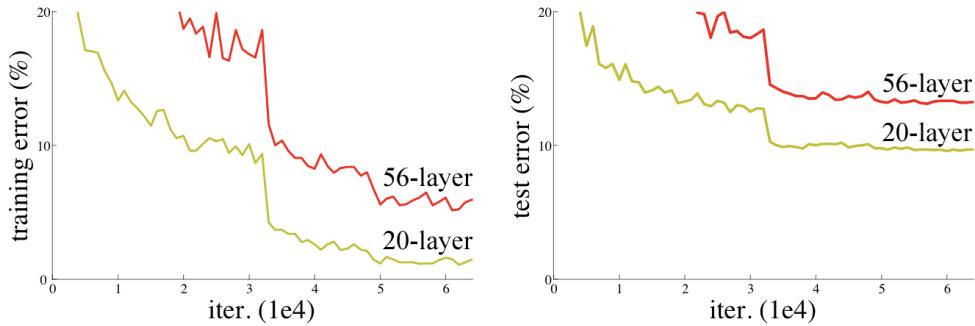
$$y[i] = \sum_{k=1}^K x[i + rk]w[k]. \quad (4.1)$$

### 4.3.3 Reti Neurali Residue

A partire dal 2012, per aumentare le performance delle reti neurali si sono costruiti modelli sempre più profondi, ed è nata la convinzione che aumentando sempre di più gli strati delle reti si possa ottenere una maggiore accuratezza. Questa convinzione, tuttavia, è stata sfidata. In particolare, si è dimostrato che, mentre in teoria reti più profonde possono approssimare pattern più complessi e di conseguenza ottenere performance migliori, nella pratica esiste una certa soglia al di sopra della quale l'accuratezza dei modelli si satura. Ad esempio, è stato dimostrato che la "profondità ottima" dei modelli testati sul noto dataset ImageNet è tra i 16 e i 30 strati, e che un modello con 18 strati performa meglio di uno con 34 (Figura 4.10). Un simile risultato lo possiamo notare anche con il dataset CIFAR-10, con cui testando un modello con 20 strati e uno con 56, il fenomeno è lo stesso (Figura 4.11).



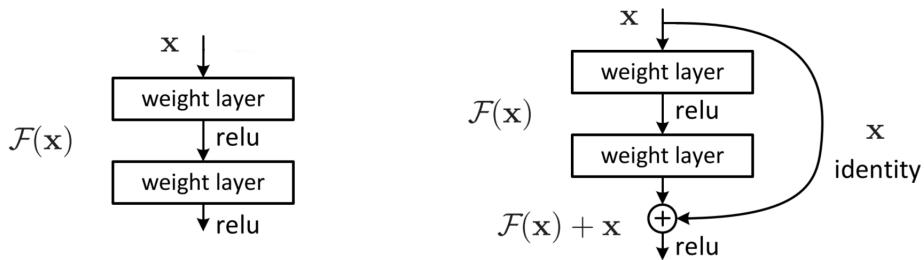
**Figura 4.10.** Addestramento su ImageNet di architetture senza strati residui (*plain*). Le curve sottili denotano l’errore di training, mentre quelle in grassetto denotano l’errore di validation. Come si può notare, la rete con 34 strati ha il training error più alto di quella con 18 strati durante tutto l’addestramento [65].



**Figura 4.11.** Addestramento su CIFAR-10 di architetture senza strati residui. Anche qui, durante tutto l’addestramento, la rete con più strati (56) ha sia il training error sia il test error più alto di quella con meno strati (20) [65].

La causa principale della saturazione dell’accuratezza risiede nella difficoltà di ottimizzare la rete a causa di problemi come la *scomparsa del gradiente* e l’*esplosione del gradiente*. Per alleviare questa difficoltà, è stato introdotto un tipo di strato chiamato strato residuo [65], ovvero uno strato all’interno del quale l’output non è  $F(x)$  (dove  $x$  è l’input dello strato) come nei normali strati di una rete, bensì  $F(x) + x$  (Figura 4.12). In particolare, l’input dello strato viene sommato all’output e questo meccanismo viene implementato con quelle che sono chiamate *skip connections*. L’aggiunta della  $x$  all’output dello strato risolve il problema della scomparsa del gradiente, poiché nel caso dell’azzeramento dei parametri dello strato, il suo output non sarebbe comunque azzerato, ma sarebbe uguale all’input. Questo aspetto, oltre a risolvere la scomparsa del gradiente, rappresenta un secondo vantaggio degli strati

residui, ovvero l'aggiunta di uno strato residuo non può causare particolari peggioramenti della rete, sia in termini di ottimizzazione che di performance. In particolare, l'aggiunta di uno strato residuo garantisce, oltre ai possibili miglioramenti discussi precedentemente, di non aggiungere particolari difficoltà all'ottimizzazione e di non peggiorare le performance della rete. Questa garanzia è data dal fatto che, nel caso in cui l'aggiunta non migliorasse le performance, comunque non ne complicherebbe più di tanto l'ottimizzazione, poiché con l'aggiunta della  $x$ , alla rete basta azzerare i parametri dello strato per approssimare la funzione identità. In particolare, la funzione identità prende in input  $x$  e restituisce  $x$ , non cambiando di conseguenza l'output dello strato precedente.

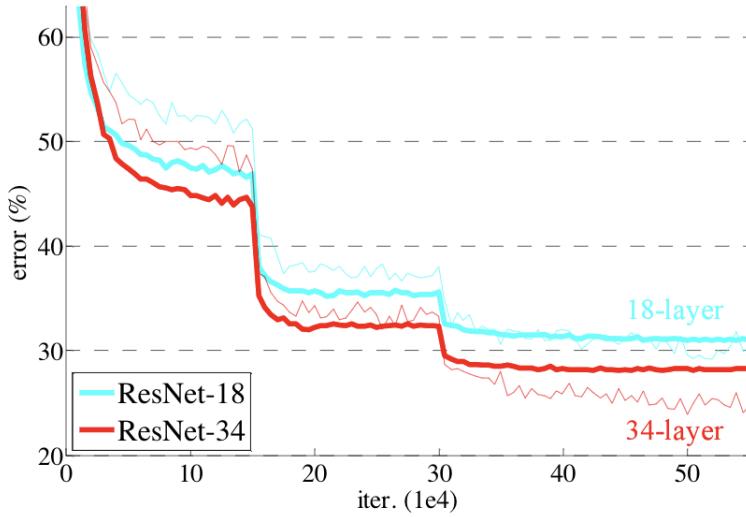


**Figura 4.12.** Illustrazione della differenza tra uno strato classico e uno strato residuo.

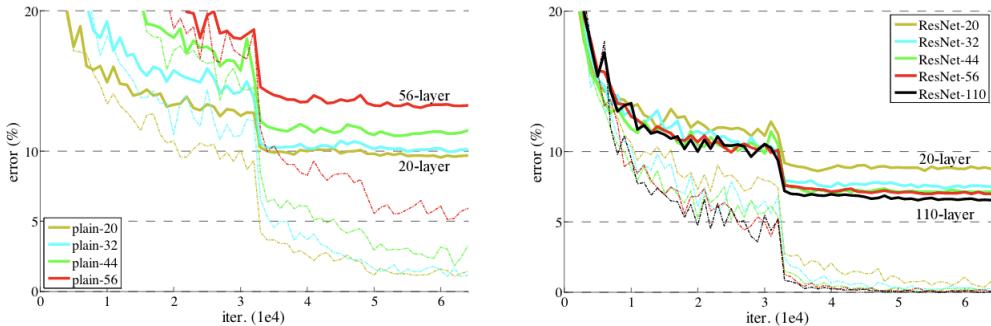
Le Figure 4.13 e 4.14 mostrano l'effetto degli strati residui. In particolare, si può notare come nella Figura 4.13, rispetto alla Figura 4.10, la situazione sia invertita, ovvero il modello con strati residui e con in totale 34 strati (ResNets-34) ha, durante tutto l'addestramento, sia un training error che un validation error più basso rispetto al corrispettivo con meno strati (ResNet-18). Allo stesso modo, nel grafico delle architetture con strati residui addestrati con CIFAR-10 (Figura 4.14 a destra) la situazione è invertita rispetto al grafico delle architetture *plain* (Figura 4.14 a sinistra). Oltre a questo, che secondo gli autori di [65] è la dimostrazione che gli strati residui risolvono i problemi di ottimizzazione di architetture più profonde, questi grafici dimostrano che l'uso di strati residui può migliorare le performance dei singoli modelli, al di là della presenza di problemi di ottimizzazione. Infatti, nel confrontare le Figure 4.10 e 4.13 si può notare come anche le performance del modello più piccolo siano migliorate.

### ResNet101 come backbone

Come menzionato precedentemente, la versione utilizzata come feature extractor (backbone) nell'architettura di questo lavoro è la ResNet101, una particolare versione dell'architettura proposta in [65] composta da 101 strati totali. In particolare, riprendendo quanto menzionato nel paragrafo 4.3, la versione originale della rete con strati residui viene modificata per trasformarla da un classificatore ad un estrattore di feature, mantenendo comunque il numero di parametri invariato grazie alle convoluzioni dilatate. Entrando nel dettaglio dell'architettura utilizzata, essa è composta da un primo blocco all'interno del quale troviamo una convoluzione 7x7, che prende in input i 3 canali dell'RGB e ne restituisce 64 (numero dei kernel); uno strato di



**Figura 4.13.** Addestramento su ImageNet di architetture con strati residui. Le curve sottili denotano l'errore di training, mentre quelle in grassetto denotano l'errore di validation [65].

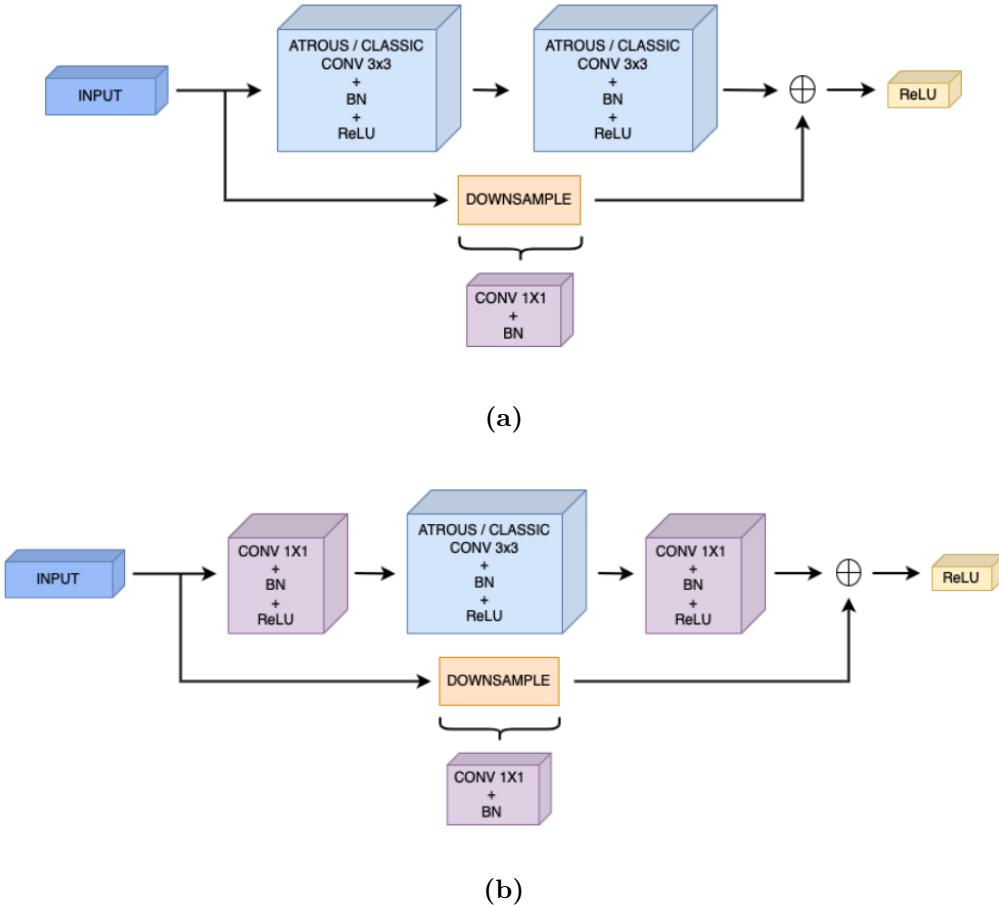


**Figura 4.14.** Addestramento su CIFAR-10. A sinistra il grafico dell'addestraemnto di architetture senza strati residui (*plain*) che mostra come le architetture più profonde siano peggiori. A destra invece viene mostrato lo stesso grafico ma riguardante architetture con strati residui [65].

batch normalization; uno di attivazione (ReLU) e infine uno di max pool. Dopodichè, l'architettura segue uno schema in comune tra tutte le altre versioni (ResNet18, ResNet34, ResNet50, ...), composto da quattro blocchi. Questi ultimi, composti da un tipo di blocco chiamato *bottleneck* (Figura 4.15b), differiscono tra loro solo nel numero di bottleneck e nell'utilizzo o meno della convoluzione dilatata.

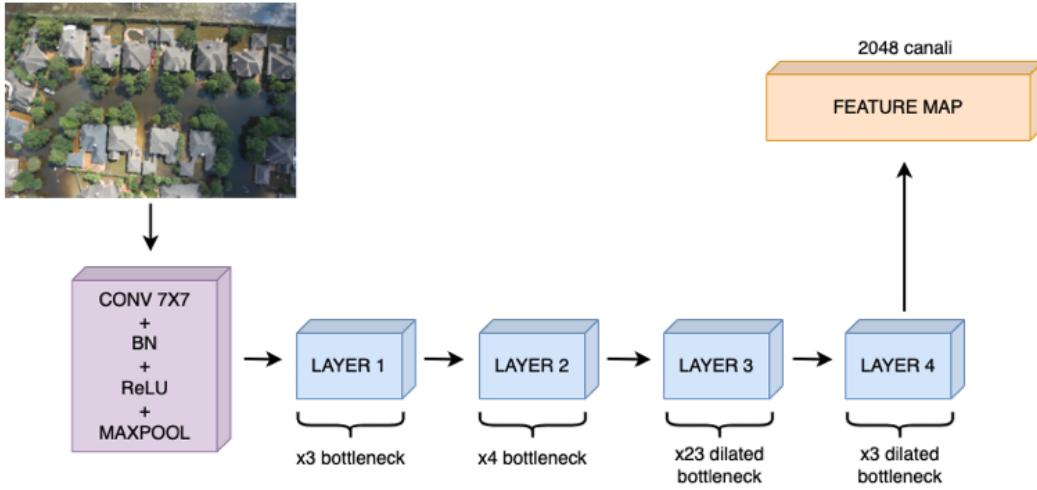
Nella ResNet101 in particolare, i quattro blocchi sono composti rispettivamente da 3,4,23 e 3 blocchi bottleneck e in alcuni blocchi la convoluzione dilatata è sostituita a quella classica (Figura 4.16).

Entrando nel dettaglio dei bottleneck, la struttura generale è composta da una sequenza di tre convoluzioni: una convoluzione 1x1, una 3x3 (che può essere classica



**Figura 4.15.** La riga sopra mostra la versione originale dello strato residuo, mentre quella sotto la versione utilizzata nelle architetture più profonde per diminuire il costo computazionale del singolo blocco (bottleneck).

o dilatata a seconda dello strato) e infine un'altra 1x1. Inoltre, ogni convoluzione è seguita da uno strato di batch normalization e chiaramente da uno di attivazione (tranne l'ultima convoluzione che è seguita solo dalla batch normalization). Infine, come descritto nel paragrafo precedente, attraverso una skip connection l'input del bottleneck viene sommato con l'output dell'ultima convoluzione e il risultato viene poi passato in uno strato di attivazione. Dato che, passando attraverso le convoluzioni, l'input riduce le sue dimensioni, per sommare input e output il primo viene precedentemente fatto passare in uno strato di downsample (Figura 4.15). La motivazione dell'uso di queste convoluzioni e in particolare di quelle 1x1 consiste nel ridurre il numero di canali prima di passare alla convoluzione 3x3, per motivi di costo computazionale. In particolare, la prima convoluzione 1x1 ha il ruolo di ridurre i canali, mentre la seconda ha invece il ruolo di riportarli al numero originale. La versione originale dell'architettura, pensata per la classificazione, dopo questi quattro blocchi aveva uno strato di average pool e infine un blocco di strati densi. Per i nostri scopi invece, l'output utilizzato è stato quello del quarto e ultimo blocco



**Figura 4.16.** Architettura totale della ResNet101 con convoluzioni dilatate, usata come backbone nella DeepLabV3.

convoluzionale, che restituisce una feature map di 2048 canali. La motivazione per cui è stata scelta questa versione della ResNet è che si è stabilito di utilizzare una delle versioni più profonde, in quanto gli autori evidenziano come, aumentando il numero di strati residui, le performance migliorano. Allo stesso tempo però, visti i grossi limiti avuti dal punto di vista di risorse computazionali, non sono state scelte versioni più profonde, come la ResNet152, in quanto anche se in alcuni casi le performance sono migliori, la differenza spesso è minima, mentre non lo è la differenza dal punto di vista di complessità ( $7.6 \times 10^9$  FLOPs per la Resnet101 e  $11.3 \times 10^9$  FLOPs per la ResNet152) [65].

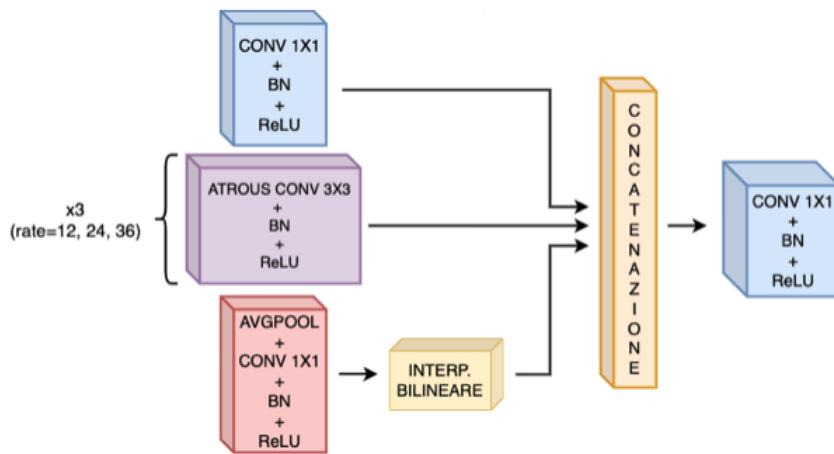
#### 4.3.4 Atrous Spatial Pyramid Pooling

Una volta prodotta la feature map, questa viene poi passata alla seconda parte della rete, di frequente chiamata *DeepLabHead*. In particolare, questa parte è composta da due blocchi: l'ASPP e un blocco che al suo interno ha in sequenza una convoluzione 3x3, batch normalization, ReLU e infine un'ultima convoluzione 1x1 che produce un volume di 9 canali, equivalenti al numero della classi del task. Entrando nel dettaglio dell'ASPP, l'idea generale è quella di utilizzare più convoluzioni dilatate in parallelo, variando la dilatazione, per poi concatenare i volumi risultanti. L'intuizione è quella di catturare contesti e informazioni spaziali a diversa scala. In particolare, nell'architettura utilizzata l'ASPP è composta da:

- un blocco con una convoluzione classica 1x1, batch normalization e ReLU.
- un blocco per ogni parametro di dilatazione utilizzato (12, 24 e 36) composto da una convoluzione dilatata 3x3, batch normalization e ReLU.

- un blocco all'interno del quale viene eseguito un global average pooling, ovvero un pooling che riduce il volume in input a  $C \times 1 \times 1$  dove  $C$  è il numero di canali, una convoluzione  $1 \times 1$ , batch normalization, ReLU e infine un'operazione di interpolazione bilineare, che ha lo scopo di riportare il risultato del blocco alle giuste dimensioni per essere concatenato con gli altri.

Di conseguenza, in totale l'ASPP ha 5 blocchi che vengono eseguiti in parallelo. Infine, gli output di tutti i blocchi vengono concatenati e il risultato viene passato in un ultimo blocco composto da una convoluzione  $1 \times 1$ , batch normalization e ReLU (Figura 4.17).



**Figura 4.17.** Illustrazione dell'architettura del modulo ASPP.

## Capitolo 5

# Esperimenti e Risultati

In questo capitolo verranno descritti gli esperimenti e i risultati del lavoro. In particolare, verranno descritti alcuni dettagli implementativi, come le librerie e i framework utilizzati, ma anche le risorse computazionali sfruttate. Inoltre, verranno descritte nel dettaglio le strategie adottate durante l'addestramento, gli iperparametri e i vari metodi testati negli esperimenti.

### 5.1 Risorse computazionali

Dato che l'addestramento di una rete neurale comporta un costo computazionale molto elevato, non avendo disponibilità di una macchina locale con abbastanza potenza computazionale per riuscire ad addestrare il modello in tempi accettabili, è stata utilizzata una nota piattaforma, chiamata Google Colab. In particolare, Google Colab permette di eseguire codice da remoto su delle macchine virtuali equipaggiate con potenza di calcolo elevata, dando soprattutto accesso a macchine virtuali con GPU, che sono fondamentali per l'addestramento di una rete neurale. Purtroppo però, Google Colab presenta delle forti limitazioni, non tanto dal punto di vista di potenza computazionale delle risorse, bensì dal punto di vista della loro disponibilità. Nello specifico, per poter offrire gratuitamente risorse computazionali e accogliere l'elevato numero di richieste, la disponibilità di risorse hardware subisce forti fluttuazioni. In particolare, teoricamente la vita massima di una macchina virtuale utilizzata è di 12 ore. Nella pratica però questo numero è sempre inferiore, sia per le fluttuazioni di richieste menzionate precedentemente, sia a causa di altri meccanismi:

- I notebook utilizzati per eseguire il codice hanno un timeout di inattività. Di conseguenza, soprattutto per l'addestramento di una rete neurale, che comporta dei lunghi tempi passivi, questo aspetto è risultato fortemente limitante.
- L'utilizzo delle GPU vengono priorizzate per gli utenti che utilizzano Colab in modo più interattivo. Anche qui, troviamo un forte contrasto con la tipologia di utilizzo che riguarda l'addestramento di una rete neurale.

Di conseguenza, visti tutti questi aspetti, le risorse di Google Colab non sono state né garantite né sempre disponibili. Questo vuol dire che, durante il lavoro

la disponibilità delle risorse necessarie a portare avanti gli esperimenti, ha avuto notevoli limitazioni, causando di conseguenza grandi rallentamenti. Nella pratica, le risorse hardware di Google Colab sono state disponibili per una media di circa 5/6 ore giornaliere. Le specifiche tecniche dell'hardware fornito sono le seguenti:

- **CPU:** Intel(R) Xeon(R)
- **RAM:** 12GB
- **GPU:** NVIDIA K80 12 GB

## 5.2 Tecnologie usate

Per quanto riguarda l'implementazione di tutte le metodologie, sono state utilizzate svariate librerie e framework. In particolare, le principali utilizzate sono:

- **PyTorch:** un framework open source di Machine Learning basato sul linguaggio di programmazione Python e sulla libreria Torch. È uno dei framework più utilizzati nel campo del Deep Learning, insieme a Tensorflow.
- **CUDA (Compute Unified Device Architecture):** un'architettura hardware per il calcolo parallelo sviluppata da NVIDIA. In particolare, CUDA permette di sfruttare al meglio la potenza di calcolo di una GPU, parallelizzando le computazioni in modo efficiente e il risultato è che le prestazioni, soprattutto per quanto riguarda la fase di addestramento e di inferenza, sono superiori a quelle di una CPU.
- **Albumentations** [75]: un tool di Computer Vision basato sul linguaggio di programmazione Python, che ha lo scopo di rendere la data augmentation veloce e flessibile. Esso implementa in modo efficiente una ricca varietà di trasformazioni dell'immagine, ottimizzate per le prestazioni, fornendo un'interfaccia di data augmentation concisa ma potente per diversi task di Computer Vision, tra cui la classificazione, segmentazione, object detection e altri.
- **OpenCV (Open Source Computer Vision Library):** una libreria open source di Computer Vision e Machine Learning costruita per fornire un'infrastruttura comune per le applicazioni di Computer Vision e per accelerare l'uso della *machine perception* nei prodotti commerciali.

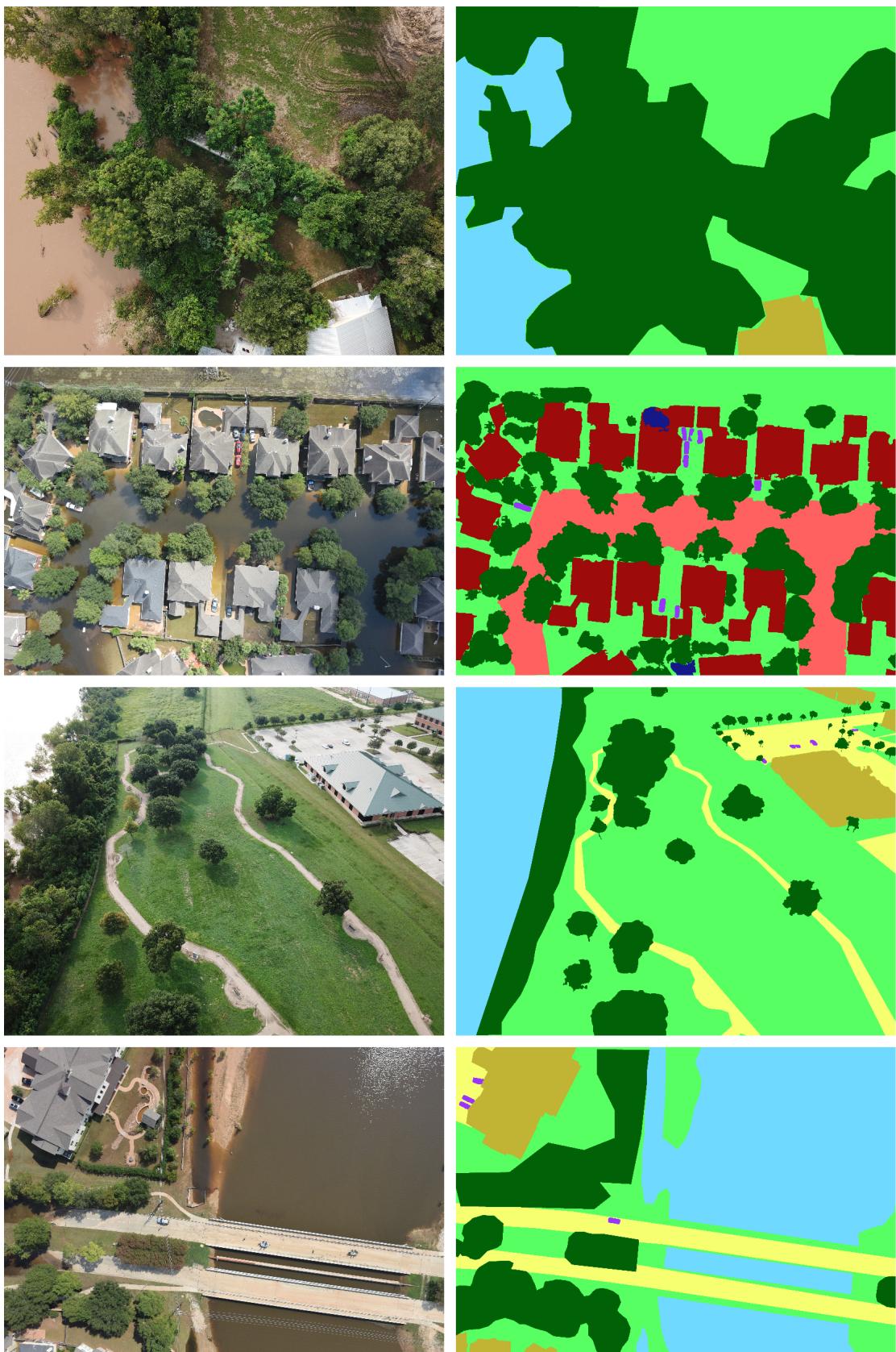
## 5.3 Dataset FloodNet

Per l'addestramento e la valutazione dei metodi e delle architetture usate in questo lavoro, è stato utilizzato il dataset *FloodNet* [30]. In particolare, questo dataset è stato pubblicato nel 2021 a seguito della FloodNet Challenge del workshop EARTH-VISION 2021 tenuto dal CVPR 2021 (Computer Vision and Pattern Recognition Conference), uno dei principali eventi annuali di Computer Vision, che comprende numerose conferenze e workshops. Floodnet è stato uno dei primi dataset pubblici del suo genere: in particolare, è stato il primo dataset pubblico con immagini e video

di UAV ad alta risoluzione e a bassa altitudine, riguardanti la fase immediatamente successiva ad un disastro naturale. Inoltre, FloodNet è uno dei pochi dataset in questo ambito a poter essere utilizzato per ben tre task differenti, ovvero classificazione, segmentazione semantica e VQA (Visual Question Answering). Le immagini del dataset sono state catturate tra il 30 Agosto e il 4 Settembre del 2017 in Texas (USA), immediatamente dopo il disastro causato dall'Uragano Harvey, con un quadricottero DJI Mavic Pro a 200 piedi di altitudine. In totale sono state raccolte 2343 immagini con una risoluzione di 1.5cm per pixel. Per quanto riguarda il task della segmentazione, ogni pixel di ogni immagine è stata annotata con una delle 9 classi ("edificio allagato", "edificio non allagato", "strada allagata", "strada non allagata", "acqua", "albero", "veicolo", "piscina" e "prato"). Come detto in precedenza, FloodNet, al momento della sua pubblicazione, è risultato un dataset unico nel suo genere. In particolare, prima della sua pubblicazione la maggior parte dei dataset pubblici con immagini riguardanti disastri naturali erano di origine satellitare. Il problema di questo tipo di immagini è che spesso, per ottenere questo tipo di dato, le attese sono di diversi giorni e dunque non si hanno dati fedeli alla fase immediatamente successiva al disastro. Inoltre, la loro risoluzione è chiaramente molto più bassa rispetto alle immagini a bassa altitudine, di conseguenza spesso scarseggiano di informazioni dettagliate sui danni causati dall'evento. Il dataset viene fornito già diviso nelle tre parti di training, validation e testing, con le seguenti proporzioni:  $\sim 60\%$  per il training set,  $\sim 20\%$  per il validation set e  $\sim 20\%$  per il test set. Come già menzionato nel paragrafo 4.2, durante il lavoro il numero delle immagini totali che compongono il dataset ha subito delle variazioni. In particolare, il dataset ha avuto tre versioni: una versione all'interno della quale sono state scartate tutte le 182 immagini trovate con errori importanti, per un totale di 2161 immagini; una versione dove di queste 182, 137 sono state corrette e reinserite, per un totale 2298; e infine un'ultima versione dove oltre a quelle corrette, sono state aggiunte 420 immagini, frutto della data augmentation offline, per un totale di 2718 immagini. Nella Figura 5.1 vengono mostrati alcuni esempi delle immagini e maschere presenti nel dataset FloodNet.

## 5.4 Esperimenti

I principali esperimenti effettuati sono quattro. Nel primo è stato testato il modello DeepLabV3 sulla prima versione del dataset, ovvero quella da 2161 immagini totali. Nello specifico, al di là della regolarizzazione fatta con il dropout, non è stata utilizzata nessuna particolare strategia o tecnica. L'obiettivo infatti, era quello di impostare una baseline da cui poi cercare di migliorare. Le immagini sono state ridimensionate a  $600 \times 800$ , è stata usata una *batch size* di 2, un learning rate di 0.01, l'ottimizzatore Adam e la funzione di perdita Focal Loss. La tabella 5.1 mostra i risultati e come si può notare, sulla maggior parte delle classi il modello ottiene performance scarse. La motivazione, probabilmente, risiede nel fatto che utilizzando questa versione del dataset, molte classi risultano troppo poco presenti nelle immagini e inoltre, senza nessun tipo particolare di regolarizzazione (tranne il dropout), il modello fatica a generalizzare.



**Figura 5.1.** Alcuni esempi delle immagini (a sinistra) del dataset FloodNet con abbinate le corrispondenti maschere (a destra).

Classe 1	Classe 2	Classe 3	Classe 4	Classe 5	Classe 6	Classe 7	Classe 8	Classe 9	mIoU
0.0018	0.087	0.0002	0.28	0.14	0.339	0.006	0.0	0.309	0.129

**Tabella 5.1.** Risultati del primo esperimento. La tabella 5.2 mostra la legenda delle classi.

Classe 1	Edificio allagato
Classe 2	Edificio non allagato
Classe 3	Strada allagata
Classe 4	Strada non allagata
Classe 5	Acqua
Classe 6	Albero
Classe 7	Veicolo
Classe 8	Piscina
Classe 9	Prato

**Tabella 5.2.** Legenda delle classi.

Nel secondo esperimento è stata introdotta la fase di data augmentation online di cui si è parlato nel paragrafo 4.2. Inoltre, è stato applicato anche un metodo di normalizzazione dell'input, ovvero tutti e tre i canali dell'RGB sono stati divisi per 255. Lo scopo di questa normalizzazione è di portare l'input dal range [0, 255] a [0, 1], e così facendo si velocizza l'addestramento, ottenendo di conseguenza un modello migliore con lo stesso numero di epoche. Il funzionamento di questo meccanismo si basa sul fatto che normalizzando l'input, la funzione di costo che cerchiamo di minimizzare durante l'addestramento prende una forma che si rivela più semplice da ottimizzare. Oltre a questo, gli iperparametri del primo esperimento rimangono invariati. La tabella 5.3 mostra i risultati di questa configurazione. Come si può notare, la normalizzazione e soprattutto l'aggiunta della data augmentation online hanno portato un leggero miglioramento, ma le performance risultano ancora scarse, soprattutto sulle classi più difficilose, ovvero "edificio allagato", "strada allagata", "veicolo" e "piscina".

Classe 1	Classe 2	Classe 3	Classe 4	Classe 5	Classe 6	Classe 7	Classe 8	Classe 9	mIoU
0.0003	0.12	0.001	0.27	0.23	0.38	0.0	0.03	0.52	0.175

**Tabella 5.3.** Risultati del secondo esperimento.

Nel terzo esperimento invece, viene utilizzata la seconda versione del dataset, ovvero quella da 2298 immagini, all'interno della quale sono state reinserite le maschere corrette. Oltre a questo, si è cercato di alzare il più possibile la risoluzione delle immagini, in quanto, alcune classi come "veicolo", essendo rappresentate da oggetti molto piccoli, risultano fortemente svantaggiate dal ridimensionamento delle immagini. In particolare, essendo limitati dalle risorse computazionali e cercando

di mantenere una batch size di minimo 2, la risoluzione è stata alzata a 750\*1000. La tabella 5.4 mostra i risultati di questo terzo esperimento. Come si può notare, l'applicazione di queste due strategie ha portato un netto miglioramento, sia per quanto riguarda le classi più semplici, ma soprattutto per quanto riguarda quelle più difficoltose. La motivazione, oltre al fatto che alzando la risoluzione il modello riceve più informazioni, è soprattutto che, grazie alla pulizia e correzione del dataset si è reintrodotto un numero importante di immagini. Nello specifico, non è tanto il numero totale di immagini aggiunte ad aver fatto la differenza, ma piuttosto il fatto che la maggior parte di queste immagini contenevano soprattutto le classi su cui si avevano le performance più scarse. Infatti, come viene mostrato nella Tabella 5.5, le classi che hanno avuto in proporzione un miglioramento più grande, sono "edificio allagato", "strada allagata", "veicolo" e "piscina". Anche qui, gli iperparametri come learning rate, batch size, ottimizzatore e funzione di perdita sono rimasti invariati.

Classe 1	Classe 2	Classe 3	Classe 4	Classe 5	Classe 6	Classe 7	Classe 8	Classe 9	mIoU
1	2	3	4	5	6	7	8	9	
0.14	0.47	0.06	0.48	0.46	0.55	0.34	0.26	0.81	0.402

**Tabella 5.4.** Risultati del terzo esperimento esperimento.

Classe 1	Classe 2	Classe 3	Classe 4	Classe 5	Classe 6	Classe 7	Classe 8	Classe 9	mIoU
+46500%	+291%	+5900%	+77%	+100%	+44%	(inf)	+766%	+55%	+129%

**Tabella 5.5.** Miglioramenti avuti nel terzo esperimento rispetto al secondo.

Infine, nel quarto ed ultimo esperimento, viene utilizzata l'ultima versione del dataset, ovvero quello da 2718 immagini totali, frutto della data augmentation offline. La tabella 5.6 mostra i risultati. Anche qui troviamo un netto miglioramento rispetto alla configurazione precedente su tutte le classi, ma soprattutto, come viene mostrato nella Tabella 5.7, sulle due classi "edificio allagato" e "strada allagata". Anche qui la motivazione principale è che, grazie all'aggiunta di immagini contenenti soprattutto queste due classi, il modello ha più materiale per apprenderle.

Classe 1	Classe 2	Classe 3	Classe 4	Classe 5	Classe 6	Classe 7	Classe 8	Classe 9	mIoU
1	2	3	4	5	6	7	8	9	
0.32	0.51	0.24	0.55	0.55	0.6	0.4	0.44	0.84	0.5

**Tabella 5.6.** Risultati del quarto esperimento esperimento.

Classe 1	Classe 2	Classe 3	Classe 4	Classe 5	Classe 6	Classe 7	Classe 8	Classe 9	mIoU
+128%	+8%	+300%	+14%	+19%	+9%	+17	+69%	+3%	+24%

**Tabella 5.7.** Miglioramenti avuti nel quarto esperimento rispetto al terzo.

Visti i miglioramenti apportati con questa configurazione, si è deciso di spingere il più possibile la fase di addestramento. In particolare, ispirandosi al lavoro di [76, 49, 36] si è adottata la seguente strategia: a partire dalla settecentesima epoca, si è fatto diminuire il learning rate ogni 100 epoch, moltiplicandolo per 0.1. Spingendo l’addestramento fino a 1000 epoch si sono ottenuti i risultati mostrati nella Tabella 5.8. Nello specifico, quest’ultimo addestramento ha richiesto un tempo totale di circa 350 ore, che a causa delle limitazioni dal punto di vista delle risorse computazionali (Paragrafo 5.1), sono state suddivise in circa 5/6 ore giornaliere, portando a un tempo totale di addestramento di circa 60 giorni.

Classe 1	Classe 2	Classe 3	Classe 4	Classe 5	Classe 6	Classe 7	Classe 8	Classe 9	mIoU
0.41	0.60	0.32	0.6	0.57	0.65	0.49	0.52	0.86	0.564

**Tabella 5.8.** Risultati del quinto esperimento.

Infine, la Figura 5.2 mostra un esempio di paragone tra le maschere corrette e l’output del modello.

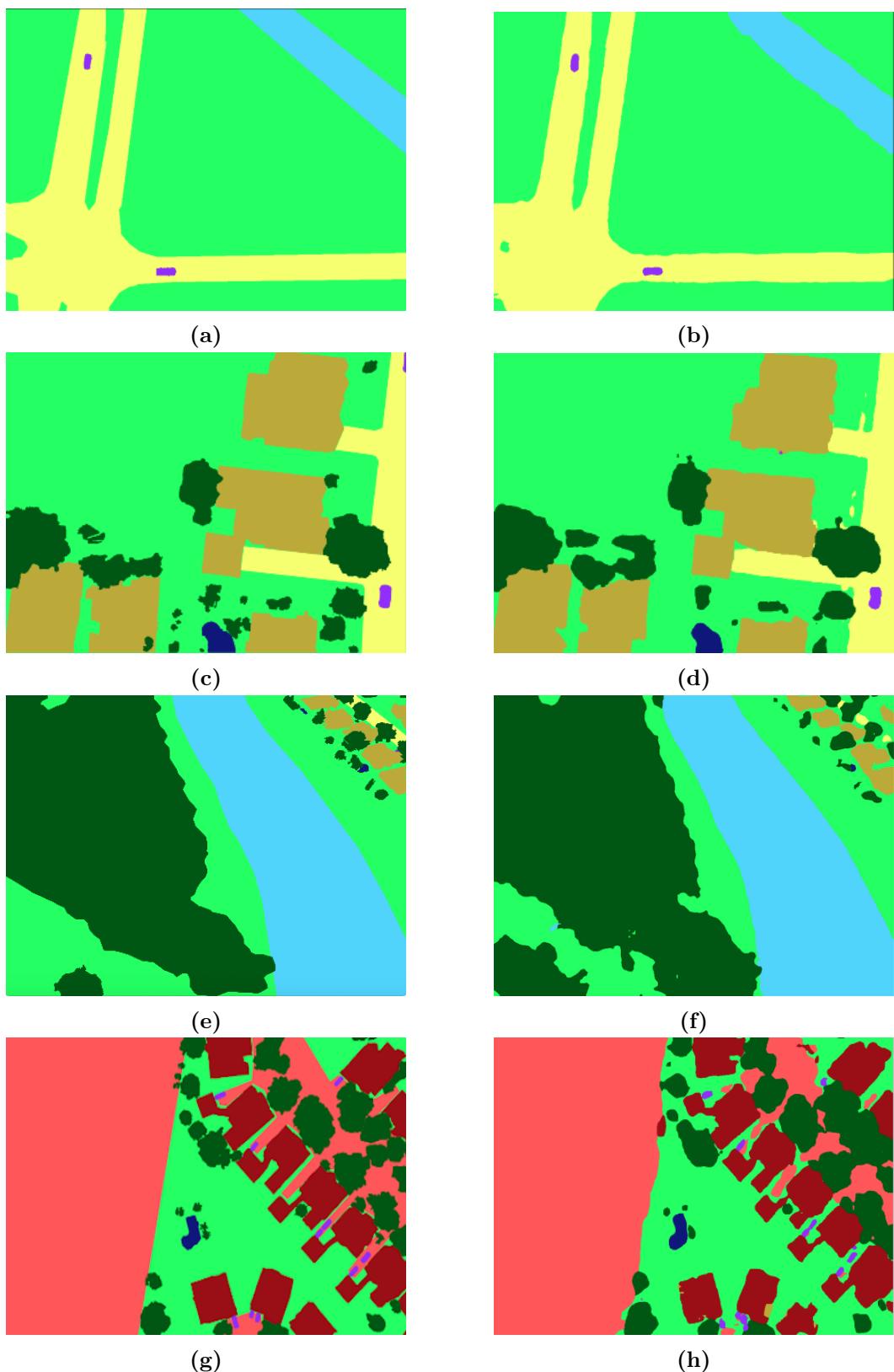
## 5.5 Paragone con altri lavori

Nella Tabella 5.9 viene riportato il paragone tra il metodo proposto e un altro lavoro dello stato dell’arte [30] che riguarda lo stesso dataset. Un fattore molto importante da considerare nel paragonare i risultati di questo lavoro con quelli di [30], è che la loro versione del dataset contiene 3200 immagini totali, a fronte delle 2343 della versione di questo lavoro. Dunque, la loro versione del dataset presenta un +36% di immagini totali, il che comporta inevitabilmente un importante vantaggio nella fase di addestramento. Questo è anche comprovato dal fatto che, nel metodo proposto, sia con la fase di pulizia del dataset sia con quella di data augmentation offline, aggiungendo corrispettivamente +6% e +18% di immagini, si sono ottenuti importanti miglioramenti, dimostrando dunque il vantaggio di avere a disposizione il 36% di immagini aggiuntive. Un loro ulteriore vantaggio è stato dal punto di vista delle risorse computazionali. In particolare, a differenza di questo lavoro che, come già detto, è stato fortemente rallentato per le ragioni sopra citate (Paragrafo 5.1), il loro lavoro si è basato sulla disponibilità continua di una macchina locale con la GPU Nvidia GeForce RTX 2080 Ti. La disponibilità continua di potenza computazionale che una macchina locale fornisce, rappresenta un importante vantaggio, soprattutto se paragonato ad una disponibilità di 5/6 ore giornaliere.

Inoltre, sempre dalla Tabella 5.9 si può notare che, nonostante lo svantaggio derivante da una minor disponibilità di immagini e da una ridotta accessibilità alle risorse computazionali, il metodo proposto migliora le IoU delle quattro classi "edificio allagato", "veicolo", "piscina" e "prato". Inoltre, un importante fattore da considerare è che le classi "edificio allagato", "veicolo" e "piscina" sono tre delle quattro, insieme a "strada allagata", che durante il lavoro si sono rivelate le più difficili da apprendere. Infine, tutti gli aspetti menzionati precedentemente dimostrano che il metodo proposto è promettente e che con risorse più adeguate e con un dataset più esteso, potrebbe raggiungere performance più alte.

Modello	Classe 1	Classe 2	Classe 3	Classe 4	Classe 5	Classe 6	Classe 7	Classe 8	Classe 9	mIoU
ENet	0.069	0.473	0.124	0.484	0.489	0.683	0.322	0.424	0.762	0.426
DeepLabV3+	0.327	0.728	0.52	0.7	0.75	0.77	0.42	0.47	0.84	0.61
<b>DeepLabV3</b>	<b>0.41</b>	0.60	0.32	0.6	0.57	0.65	<b>0.49</b>	<b>0.52</b>	<b>0.86</b>	0.564

**Tabella 5.9.** Paragone tra l'approccio sviluppato e quello di [30].



**Figura 5.2.** Le otto immagini mostrano un esempio di paragone tra le maschere corrette (a), (c), (e), (g) e le maschere prodotte dal modello (b), (d), (f), (h).

# Capitolo 6

## Conclusioni

In questo elaborato si è presentato un approccio basato su strategie di Deep Learning per la segmentazione semantica delle immagini del dataset FloodNet. In particolare, l'approccio sviluppato si è basato soprattutto su: un'architettura di Deep Learning context-based mai utilizzata su questo dataset, ovvero la DeepLabV3; una fase di data cleaning con cui è stata affrontata la presenza di errori nelle maschere del dataset; e infine su una fase di data augmentation offline, che ha avuto l'obiettivo di aumentare in modo mirato il numero di immagini, ma anche di far fronte ad altre difficoltà incontrate nell'affrontare questo task (Paragrafo 4.1). I risultati ottenuti dimostrano l'efficacia del metodo proposto, con risultati che sono prossimi a quelli dello stato dell'arte a causa delle seguenti ragioni:

- Come menzionato nel Paragrafo 5.1, vi sono stati consistenti limitazioni dal punto di vista delle risorse computazionali. In particolare, queste limitazioni hanno portato a lunghi tempi di overhead, che hanno prorogato di molto il tempo totale degli esperimenti, condizionando inevitabilmente i risultati ottenuti. Ad esempio, l'ultimo esperimento effettuato, che è stato anche il più corposo, con una disponibilità continua delle stesse risorse hardware utilizzate, si sarebbe concluso in circa 14/15 giorni, a differenza dei circa 60 giorni che invece ha impiegato.
- Per quanto riguarda il dataset, la versione qui utilizzata presenta 2343 immagini, a fronte delle 3200 (+36%) del lavoro dello stato dell'arte con cui si sono confrontati i risultati. Dunque, anche questo fattore è da considerarsi limitante ai fini della valutazione dei risultati.
- Il paragone con l'architettura DeepLabV3+ (Paragrafo 5.5), molto simile alla DeepLabV3 utilizzata in questo lavoro, considerando anche tutti i limiti menzionati nei punti precedenti, evidenzia ulteriormente la bontà dell'approccio sviluppato. In particolare, anche se la mIoU risulta leggermente più bassa (0.564 contro 0.61), le IoU di ben tre classi, ovvero "edificio allagato", "veicolo" e "piscina" risultano più alte (rispettivamente 0.41 contro 0.32; 0.49 contro 0.42; 0.52 contro 0.47). Inoltre, va considerato che le tre classi precedentemente menzionate, nelle quali si sono ottenuti tali miglioramenti, rappresentano tre delle quattro classi più difficoltose.

In conclusione, tenendo conto di tutti questi fattori, l'approccio sviluppato e i suoi risultati, sono da considerarsi più che validi. In particolare, le metodologie utilizzate, soprattutto la data augmentation offline e la fase di pulizia del dataset, hanno apportato un consistente contributo e applicandole ad altri modelli, con risorse più adeguate e tempistiche più distese, potrebbero migliorare i risultati dello stato dell'arte.

## 6.1 Sviluppi futuri

Gli sviluppi futuri che potrebbero essere apportati al nostro approccio sono:

- Reinserire le 45 immagini scartate durante la fase di data cleaning, riuscendo a correggerle attraverso tool come ad esempio la piattaforma V7 Darwin.
- Aumento della quantità di data augmentation offline applicata al dataset. In particolare, visti i miglioramenti apportati dall'aggiunta di immagini, attraverso sia la fase di data cleaning sia la fase di data augmentation, e visto che dopo queste due fasi il dataset presentava ancora sbilanciamenti, anche se più lievi, una fase di data augmentation offline ancora più corposa potrebbe portare ulteriori miglioramenti.
- Oltre all'aumento dal punto di vista della quantità, si potrebbero aggiungere altre tecniche di data augmentation, come ad esempio le tecniche descritte in [77], utili soprattutto a migliorare le performance sulle classi rappresentate da oggetti piccoli, come "veicolo" e "piscina".

# Bibliografia

- [1] Margareta Wahlstrom, Debarati Guha-Sapir et al. “The human cost of weather-related disasters 1995–2015”. In: *Geneva, Switzerland: UNISDR* (2015).
- [2] Sharifah Mastura Syed Mohd Daud et al. “Applications of drone in disaster management: A scoping review”. In: *Science & Justice* 62.1 (2022), pp. 30–42.
- [3] Faine Greenwood, Erica L Nelson e P Gregg Greenough. “Flying into the hurricane: A case study of UAV use in damage assessment during the 2017 hurricanes in Texas and Florida”. In: *PLoS one* 15.2 (2020), e0227808.
- [4] Shijie Hao, Yuan Zhou e Yanrong Guo. “A brief survey on semantic segmentation with deep learning”. In: *Neurocomputing* 406 (2020), pp. 302–321.
- [5] Joe Kinahan e Alan F Smeaton. “Image Segmentation to Identify Safe Landing Zones for Unmanned Aerial Vehicles”. In: *arXiv preprint arXiv:2111.14557* (2021).
- [6] Ye Lyu et al. “UAVid: A semantic segmentation dataset for UAV imagery”. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 165 (2020), pp. 108–119.
- [7] Teja Kattenborn, Jana Eichel e Fabian Ewald Fassnacht. “Convolutional Neural Networks enable efficient, accurate and fine-grained segmentation of plant species and communities from high-resolution UAV imagery”. In: *Scientific reports* 9.1 (2019), pp. 1–9.
- [8] Olaf Ronneberger, Philipp Fischer e Thomas Brox. “U-net: Convolutional networks for biomedical image segmentation”. In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.
- [9] Tashnim Chowdhury, Robin Murphy e Maryam Rahnemoonfar. “RescueNet: A High Resolution UAV Semantic Segmentation Benchmark Dataset for Natural Disaster Damage Assessment”. In: *arXiv preprint arXiv:2202.12361* (2022).
- [10] Ozan Oktay et al. “Attention u-net: Learning where to look for the pancreas”. In: *arXiv preprint arXiv:1804.03999* (2018).
- [11] Abolfazl Abdollahi, Biswajeet Pradhan e Abdullah M Alamri. “An ensemble architecture of deep convolutional Segnet and Unet networks for building semantic segmentation from high-resolution aerial images”. In: *Geocarto International* (2020), pp. 1–16.

- [12] Vijay Badrinarayanan, Alex Kendall e Roberto Cipolla. “Segnet: A deep convolutional encoder-decoder architecture for image segmentation”. In: *IEEE transactions on pattern analysis and machine intelligence* 39.12 (2017), pp. 2481–2495.
- [13] Imran Ahmed, Misbah Ahmad e Gwanggil Jeon. “A real-time efficient object segmentation system based on U-Net using aerial drone images”. In: *Journal of Real-Time Image Processing* 18.5 (2021), pp. 1745–1758.
- [14] Andrew G Howard et al. “Mobilenets: Efficient convolutional neural networks for mobile vision applications”. In: *arXiv preprint arXiv:1704.04861* (2017).
- [15] Hengshuang Zhao et al. “Pyramid scene parsing network”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 2881–2890.
- [16] Liang-Chieh Chen et al. “Encoder-decoder with atrous separable convolution for semantic image segmentation”. In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 801–818.
- [17] Adam Paszke et al. “Enet: A deep neural network architecture for real-time semantic segmentation”. In: *arXiv preprint arXiv:1606.02147* (2016).
- [18] Dimitrios Marmanis et al. “Semantic segmentation of aerial images with an ensemble of CNSS”. In: *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, 2016* 3 (2016), pp. 473–480.
- [19] Kaiqiang Chen et al. “Semantic segmentation of aerial images with shuffling convolutional neural networks”. In: *IEEE Geoscience and Remote Sensing Letters* 15.2 (2018), pp. 173–177.
- [20] Ruigang Niu et al. “Hybrid multiple attention network for semantic segmentation in aerial images”. In: *IEEE Transactions on Geoscience and Remote Sensing* 60 (2021), pp. 1–18.
- [21] Haifeng Luo et al. “High-resolution aerial images semantic segmentation using deep fully convolutional network with channel attention mechanism”. In: *IEEE journal of selected topics in applied earth observations and remote sensing* 12.9 (2019), pp. 3492–3507.
- [22] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).
- [23] Dengfeng Chai, Shawn Newsam e Jingfeng Huang. “Aerial image semantic segmentation using DCNN predicted distance maps”. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 161 (2020), pp. 309–322.
- [24] Caio CV da Silva et al. “Towards open-set semantic segmentation of aerial images”. In: *2020 IEEE Latin American GRSS & ISPRS Remote Sensing Conference (LAGIRS)*. IEEE. 2020, pp. 16–21.
- [25] Asmamaw A Gebrehiwot e Leila Hashemi-Beni. “Three-Dimensional Inundation Mapping Using UAV Image Segmentation and Digital Surface Model”. In: *ISPRS International Journal of Geo-Information* 10.3 (2021), p. 144.

- [26] 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*. 2015, pp. 3431–3440.
- [27] Ananya Gupta, Simon Watson e Hujun Yin. “Deep learning-based aerial image segmentation with open data for disaster impact assessment”. In: *Neurocomputing* 439 (2021), pp. 22–33.
- [28] Abhishek Chaurasia e Eugenio Culurciello. “Linknet: Exploiting encoder representations for efficient semantic segmentation”. In: *2017 IEEE Visual Communications and Image Processing (VCIP)*. IEEE. 2017, pp. 1–4.
- [29] Saheba Bhatnagar, Laurence Gill e Bidisha Ghosh. “Drone image segmentation using machine and deep learning for mapping raised bog vegetation communities”. In: *Remote Sensing* 12.16 (2020), p. 2602.
- [30] Maryam Rahnemoonfar et al. “Floodnet: A high resolution aerial imagery dataset for post flood scene understanding”. In: *IEEE Access* 9 (2021), pp. 89644–89654.
- [31] Xiaojuan Qi et al. “3d graph neural networks for rgbd semantic segmentation”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 5199–5208.
- [32] Shiying Hu, Eric A Hoffman e Joseph M Reinhardt. “Automatic lung segmentation for accurate quantitation of volumetric X-ray CT images”. In: *IEEE transactions on medical imaging* 20.6 (2001), pp. 490–498.
- [33] Mark Everingham et al. “The PASCAL visual object classes challenge 2007 (VOC2007) results”. In: (2008).
- [34] Marius Cordts et al. “The cityscapes dataset for semantic urban scene understanding”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 3213–3223.
- [35] Bolei Zhou et al. “Semantic understanding of scenes through the ade20k dataset”. In: *International Journal of Computer Vision* 127.3 (2019), pp. 302–321.
- [36] Liang-Chieh Chen et al. “Rethinking atrous convolution for semantic image segmentation”. In: *arXiv preprint arXiv:1706.05587* (2017).
- [37] Wolfgang Köhler. “Gestalt psychology”. In: *Psychologische Forschung* 31.1 (1967), pp. XVIII–XXX.
- [38] Dejan Todorovic. “Gestalt principles”. In: *Scholarpedia* 3.12 (2008), p. 5345.
- [39] David Martin et al. “A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics”. In: *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*. Vol. 2. IEEE. 2001, pp. 416–423.
- [40] Luc Vincent e Pierre Soille. “Watersheds in digital spaces: an efficient algorithm based on immersion simulations”. In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 13.06 (1991), pp. 583–598.

- [41] Pedro F Felzenszwalb e Daniel P Huttenlocher. “Efficient graph-based image segmentation”. In: *International journal of computer vision* 59.2 (2004), pp. 167–181.
- [42] Jianbo Shi e Jitendra Malik. “Normalized cuts and image segmentation”. In: *IEEE Transactions on pattern analysis and machine intelligence* 22.8 (2000), pp. 888–905.
- [43] Stuart Lloyd. “Least squares quantization in PCM”. In: *IEEE transactions on information theory* 28.2 (1982), pp. 129–137.
- [44] James MacQueen et al. “Some methods for classification and analysis of multivariate observations”. In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. Vol. 1. 14. Oakland, CA, USA. 1967, pp. 281–297.
- [45] Keinosuke Fukunaga e Larry Hostetler. “The estimation of the gradient of a density function, with applications in pattern recognition”. In: *IEEE Transactions on information theory* 21.1 (1975), pp. 32–40.
- [46] D. Comaniciu e P. Meer. “Mean shift: a robust approach toward feature space analysis”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24.5 (2002), pp. 603–619. DOI: 10.1109/34.1000236.
- [47] Fisher Yu e Vladlen Koltun. “Multi-scale context aggregation by dilated convolutions”. In: *arXiv preprint arXiv:1511.07122* (2015).
- [48] Wei Liu, Andrew Rabinovich e Alexander C. Berg. “ParseNet: Looking Wider to See Better”. In: *CoRR* abs/1506.04579 (2015). arXiv: 1506 . 04579. URL: <http://arxiv.org/abs/1506.04579>.
- [49] Liang-Chieh Chen et al. “Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs”. In: *IEEE transactions on pattern analysis and machine intelligence* 40.4 (2017), pp. 834–848.
- [50] Hyeonwoo Noh, Seunghoon Hong e Bohyung Han. “Learning deconvolution network for semantic segmentation”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1520–1528.
- [51] Pauline Luc et al. “Semantic segmentation using adversarial networks”. In: *arXiv preprint arXiv:1611.08408* (2016).
- [52] Shi Dong, Ping Wang e Khushnood Abbas. “A survey on deep learning and its applications”. In: *Computer Science Review* 40 (2021), p. 100379.
- [53] Shaveta Dargan et al. “A survey of deep learning and its applications: a new paradigm to machine learning”. In: *Archives of Computational Methods in Engineering* 27.4 (2020), pp. 1071–1092.
- [54] Alex Krizhevsky, Ilya Sutskever e Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).

- [55] Filip Karlo Došilović, Mario Brčić e Nikica Hlupić. “Explainable artificial intelligence: A survey”. In: *2018 41st International convention on information and communication technology, electronics and microelectronics (MIPRO)*. IEEE. 2018, pp. 0210–0215.
- [56] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.
- [57] Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep learning*. MIT press, 2016.
- [58] Douglas Heaven. “Deep trouble for deep learning”. In: *Nature* 574.7777 (2019), pp. 163–166.
- [59] Randall Balestriero, Leon Bottou e Yann LeCun. “The Effects of Regularization and Data Augmentation are Class Dependent”. In: *arXiv preprint arXiv:2204.03632* (2022).
- [60] Jiuxiang Gu et al. “Recent advances in convolutional neural networks”. In: *Pattern Recognition* 77 (2018), pp. 354–377.
- [61] Karen Simonyan e Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).
- [62] Matthew D Zeiler e Rob Fergus. “Visualizing and understanding convolutional networks”. In: *European conference on computer vision*. Springer. 2014, pp. 818–833.
- [63] Pierre Sermanet et al. “Overfeat: Integrated recognition, localization and detection using convolutional networks”. In: *arXiv preprint arXiv:1312.6229* (2013).
- [64] Gao Huang et al. “Densely connected convolutional networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.
- [65] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.
- [66] Gao Huang et al. “Deep networks with stochastic depth”. In: *European conference on computer vision*. Springer. 2016, pp. 646–661.
- [67] Rupesh K Srivastava, Klaus Greff e Jürgen Schmidhuber. “Training very deep networks”. In: *Advances in neural information processing systems* 28 (2015).
- [68] Gustav Larsson, Michael Maire e Gregory Shakhnarovich. “Fractalnet: Ultra-deep neural networks without residuals”. In: *arXiv preprint arXiv:1605.07648* (2016).
- [69] Christian Szegedy et al. “Going deeper with convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.
- [70] Sergey Ioffe e Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

- [71] Christian Szegedy et al. “Rethinking the inception architecture for computer vision”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2818–2826.
- [72] Christian Szegedy et al. “Inception-v4, inception-resnet and the impact of residual connections on learning”. In: *Thirty-first AAAI conference on artificial intelligence*. 2017.
- [73] François Chollet. “Xception: Deep learning with depthwise separable convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1251–1258.
- [74] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.
- [75] Alexander Buslaev et al. “Albumentations: Fast and Flexible Image Augmentations”. In: *Information* 11.2 (2020). ISSN: 2078-2489. DOI: 10.3390/info11020125. URL: <https://www.mdpi.com/2078-2489/11/2/125>.
- [76] Liang-Chieh Chen et al. “Semantic image segmentation with deep convolutional nets and fully connected crfs”. In: *arXiv preprint arXiv:1412.7062* (2014).
- [77] Mate Kisantal et al. “Augmentation for small object detection”. In: *arXiv preprint arXiv:1902.07296* (2019).