



SAPIENZA
UNIVERSITÀ DI ROMA

Cloud Detection su immagini satellitari Sentinel-2A mediante tecniche di Deep Learning

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Laurea in Ingegneria Informatica e Automatica

Isidoro Tamassia
Matricola 1941484

Relatore
Prof. Thomas Alessandro Ciarfuglia

Anno Accademico 2022/2023

Cloud Detection su immagini satellitari Sentinel-2A mediante tecniche di Deep Learning

Tesi di Laurea Triennale. Sapienza Università di Roma

© 2023 Isidoro Tamassia. Tutti i diritti riservati

Questa tesi è stata composta con L^AT_EX e la classe Sapthesis.

Email dell'autore: isidorotamassia@gmail.com

Sommario

Il rilevamento automatico delle nuvole su immagini satellitari è di particolare interesse per via delle problematiche che esse creano durante l'analisi a scopi geologici e metereologici. Negli ultimi anni il grande sviluppo delle reti neurali ha permesso di ottenere molteplici modelli particolarmente efficaci e capaci di sfruttare le varie bande di frequenza disponibili, andando difatti a sostituire i precedenti modelli tradizionali basati su regole euristiche e sullo studio della caratterizzazione fisica e geometrica delle nuvole.

L'obiettivo di questa relazione è descrivere le principali caratteristiche di questi modelli, andando ad esplorare l'attuale stato dell'arte e proponendo, in contrapposizione alla complessità di molti di essi, una rete U-Net molto leggera che ottiene performance particolarmente interessanti, seppure allenata su una quantità relativamente ristretta di dati.

La tesi è suddivisa in capitoli; nell'introduzione viene descritto brevemente il problema, quali approcci sono stati sviluppati nel corso del tempo, e quali strumenti ho scelto di utilizzare per il mio lavoro. Segue un capitolo dedicato alla descrizione dettagliata del dataset di riferimento e di come è necessario gestirlo per fare in modo che sia utilizzabile come input di una rete neurale convoluzionale, compresi i dettagli implementativi. Successivamente viene preso in esame un modello attualmente considerato stato dell'arte, ovvero la rete neurale CD-FM3SF frutto del lavoro di vari ricercatori della Wuhan University, che verrà utilizzata come baseline di riferimento essendo un modello estremamente performante sotto molteplici aspetti. Varie sezioni sono poi interamente dedicate alla struttura e alla fase di addestramento, test ed ottimizzazione della mia rete U-Net, in cui viene inoltre raccontato l'intero iter che ha portato dai primi esperimenti al raggiungimento della versione finale del modello. Infine, nell'ultimo capitolo vengono riportati e commentati i risultati ottenuti, con alcune considerazioni su possibili ampliamenti futuri del lavoro.

Indice

1	Introduzione	1
1.1	Descrizione del task	1
1.2	Strumenti utilizzati	5
2	Il dataset WHUS2-CD+	6
2.1	Struttura del dataset	6
2.2	Formato iniziale, trasformazione in TIFF ed organizzazione del file system	7
2.3	Implementazione della classe WHUSDataset	9
2.4	Visualizzazione delle immagini	12
3	Un modello di confronto	14
3.1	Struttura della rete neurale CD-FM3SF	14
3.2	Traguardi raggiunti	16
4	Struttura della rete U-Net	18
4.1	Panoramica generale	18
4.2	Blocco di contrazione	21
4.3	Blocco di espansione	22
4.4	Skip connections	25
5	Addestramento della rete	26
5.1	Organizzazione dell'addestramento	26
5.2	Salvataggio dei progressi	28
5.3	Funzione di loss e bilanciamento delle classi	29
5.4	Funzione di ottimizzazione	30
5.5	Metriche di valutazione	32
6	Risultati	33
6.1	Valutazione metrica dell'addestramento	33
6.2	Valutazione visiva dei test: punti di forza	35
6.3	Valutazione visiva dei test: punti di debolezza	37
7	Conclusioni	40
	Bibliografia	41

Capitolo 1

Introduzione

1.1 Descrizione del task

La Copernicus Sentinel-2 è una costellazione di due satelliti identici, posizionati sulla stessa orbita sfasati di 180 gradi, con lo scopo di scattare fotografie della superficie terrestre ad alta risoluzione spaziale nel dominio ottico [15]. Il primo, chiamato Sentinel-2A, è stato lanciato nello spazio il 23 Giugno 2015, mentre il Sentinel-2B circa 2 anni più tardi, il 7 Marzo 2017. Il progetto, finanziato dall'Unione Europea, è motivato dalle molteplici applicazioni che giovano dei dati raccolti, tra cui l'agricoltura, il monitoraggio degli ecosistemi terrestri, il monitoraggio della qualità delle acque interne e costiere, la mappatura dei disastri ambientali e la sicurezza civile in aree a rischio.

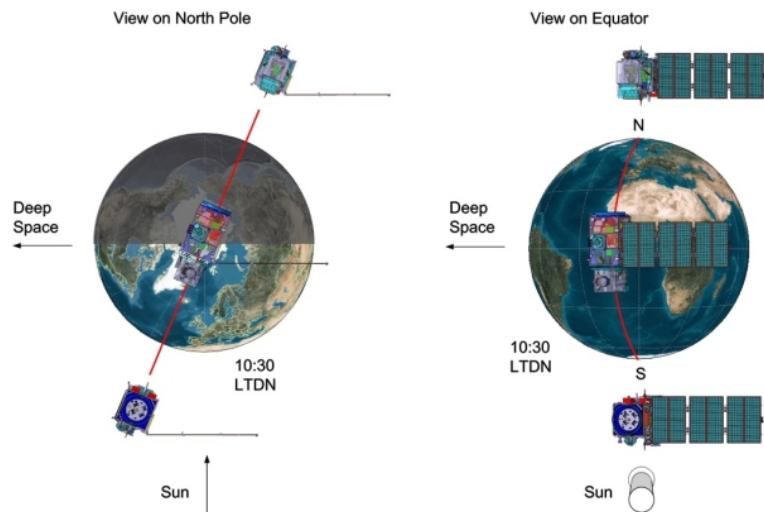


Figura 1.1. Configurazione orbitale dei satelliti gemelli Sentinel-2

In media, le nuvole coprono circa il 66% della superficie terrestre [20], il che rende la loro rilevazione e successiva rimozione dalle immagini una necessità. Ci sono tre metodologie tradizionali principali per effettuare cloud detection

su immagini satellitari, ovvero rule-based, object-based e multitemporal-based. Una descrizione dettagliata di tali metodologie esula dagli scopi della relazione, ma in generale è importante notare come tutte e tre utilizzino principalmente informazioni spettrali ignorando quasi completamente quelle spaziali, che sono a volte molto utili per distinguere le nuvole, ad esempio, dalla neve e dal ghiaccio, e più in generale da ogni tipo di zona particolarmente chiara dell'immagine.

Oltre ai sovraccitati metodi classici, modelli di machine learning (ML) quali Random Forest e SVM hanno contribuito a migliorare i risultati della classificazione; inoltre, negli ultimi anni i progressi maggiori sono stati ottenuti tramite strategie di Deep Learning (DL), in particolare mediante l'utilizzo di reti neurali convoluzionali (CNN) [2]. Le CNN rappresentano evidentemente un approccio ideale al problema, dato che permettono l'estrazione automatica delle feature spaziali dalle immagini mediante alcune operazioni sui pixel, di cui la principale è l'operatore di convoluzione. Data un'immagine H ed un filtro F , il valore del pixel $G[i, j]$ dell'immagine G ottenuta a seguito della convoluzione è:

$$G[i, j] = \sum_m \sum_n H[m, n] \cdot F[i - m, j - n]$$

Questa operazione è inoltre spesso applicata "saltando" alcuni pixel (convoluzione con stride) in modo da ottenere in output una feature map di dimensione minore dell'immagine originale in input [18].

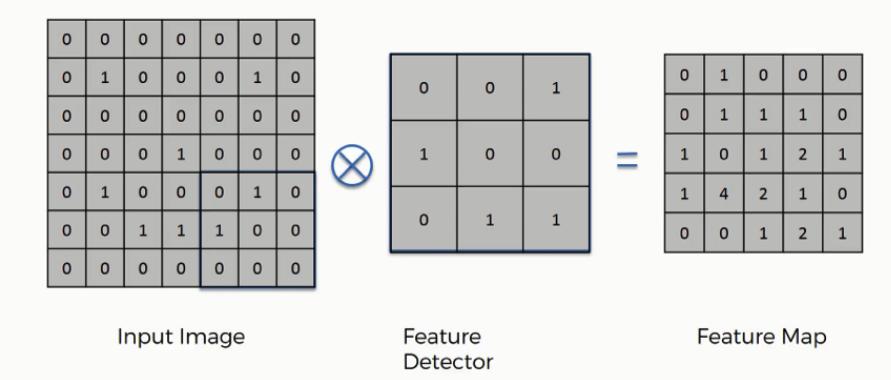


Figura 1.2. Operazione di convoluzione con stride

Molteplici varianti di CNN sono state proposte; tuttavia, la maggior parte di questi modelli ha una dimensione molto grande, essendo costituiti da milioni di parametri, il che risulta essere un problema sia a livello di applicabilità che di explainability (una delle maggiori sfide nel campo del Deep Learning [3]). Pertanto, un numero sempre maggiore di ricercatori e sviluppatori nel campo del ML è interessato alla creazione ed ottimizzazione di reti neurali con un numero ristretto di parametri, adatte ad essere addestrate su device meno potenti di quelli a disposizione delle grandi aziende informatiche e in un quantitativo di tempo minore.

L'obiettivo di questa relazione non è dunque quello di andare ad eguagliare o superare i risultati delle architetture attualmente considerate stato dell'arte, ma mostrare come una CNN di dimensioni moderate creata a partire da una struttura standard nota, ovvero una U-Net, possa ottenere comunque performance di buon livello se addestrata nella maniera corretta. La "maniera corretta" è in questo caso basata sul giusto tuning degli iperparametri, sulla scelta di funzioni di ottimizzazione e di loss che ben si adattano al tipo di problema e, soprattutto, ad un'attenta selezione dei dati del training set. Infatti, addestrare una CNN in locale su un normale computer portatile o su un ambiente di sviluppo principalmente a scopi didattici come Google Colab è decisamente più complesso che farlo su un server particolarmente prestante, e data l'impossibilità di sfruttare tutti i dati a disposizione su un hardware del genere, scegliere in maniera accurata quali immagini utilizzare e quali no diventa cruciale. Questo aspetto verrà particolarmente approfondito nel capitolo dedicato all'addestramento della rete.

Il sistema MSI di Sentinel-2 copre 13 bande spettrali, divise a tre livelli di risoluzione: 10 metri, 20 metri e 60 metri, dove le bande alle risoluzioni più alte (10m e 20m) sono utilizzate per la mappatura del suolo ed il monitoraggio ambientale, mentre le 3 bande a 60 metri sono utili al rilevamento del vapore atmosferico. Esse presentano varie differenze fra cui il valore della larghezza di banda e la lunghezza d'onda a centro banda, riportate nella seguente tabella:

Band no.	Band name	Central Wavelength (μm)	Bandwidth (nm)	Spatial resolution (m)
Band 1	Coastal aerosol	0.443	27	60
Band 2	Blue	0.490	98	10
Band 3	Green	0.560	45	10
Band 4	Red	0.665	38	10
Band 5	Vegetation Red Edge	0.705	19	20
Band 6	Vegetation Red Edge	0.740	18	20
Band 7	Vegetation Red Edge	0.783	28	20
Band 8	NIR	0.842	145	10
Band 8A	Vegetation Red Edge	0.865	33	20
Band 9	Water Vapour	0.945	26	60
Band 10	SWIR-Cirrus	1.375	75	60
Band 11	SWIR	1.610	143	20
Band 12	SWIR	2.190	242	20

Figura 1.3. Tabella delle bande spettrali

Consideriamo ora 5 tipi di copertura terrestre, ovvero ghiaccio/neve, acqua, zone urbane, vegetazione e terreno sterile. Un parametro d'interesse che permette di distinguere queste zone, utile sia in problemi di classificazione binaria che di segmentation (classificazione multiclasse pixel per pixel) è la

TOA (top of atmosphere) reflectance, che misura la quantità di luce riflessa da una superficie terrestre, e che viene rilevata da un satellite in volo sopra l'atmosfera terrestre. La luce solare che raggiunge la superficie interagisce con l'atmosfera prima di essere intercettata dai sensori a bordo dei satelliti utilizzati per acquisire immagini. Durante questo percorso attraverso l'atmosfera, la luce subisce assorbimento, diffusione e rifrazione da parte di particelle atmosferiche, come gas, aerosol e nubi, che quindi andranno ad influire sul valore misurato dal satellite. Di conseguenza, misurando la TOA reflectance di una determinata porzione di immagine satellitare, otterremo un valore sicuramente differente a seconda della presenza di un' intensa copertura nuvolare o meno.

Il seguente grafico mostra la TOA reflectance delle 5 zone terrestri indicate confrontate a quella delle nuvole (ovvero di zone ad alta copertura nuvolare) al variare della lunghezza d'onda. I valori sono stati ottenuti, per ogni categoria, misurando la riflettanza di campioni casuali di pixel e facendo la loro media banda per banda [4].

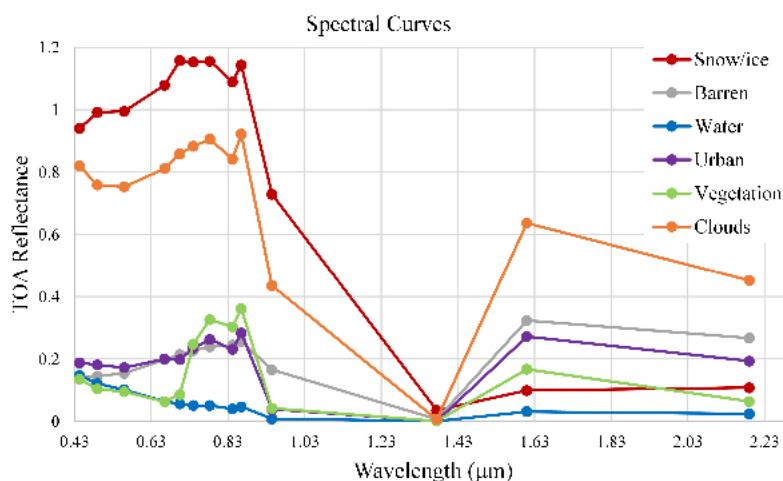


Figura 1.4. TOA reflectance di 5 coperture terrestri e delle nuvole

Come possiamo notare dalle curve graficate, la riflettanza delle nuvole è in generale distante da quelle delle superfici considerate, ad eccezione della neve. Tuttavia, anche in questo caso i valori diventano più distanti se si considera una lunghezza d'onda sufficientemente elevata, ovvero quella delle bande 10/11. Questo giustifica la scelta recente di alcuni progettisti del campo di utilizzare tutte le bande e non solamente quelle a 10 metri come fatto in passato su molti modelli. In questa relazione, sebbene la mia rete neurale utilizzi solamente RGB e NIR, andrò comunque ad esplorare i possibili miglioramenti che è possibile ottenere sfruttando anche le bande a 20 e 60 metri.

1.2 Strumenti utilizzati

L’addestramento di reti neurali convoluzionali può richiedere una potenza di calcolo molto elevata. Più in generale, il fatto che i modelli di Deep Learning tendano a migliorare all’aumentare della profondità della rete (se si ha una sufficiente quantità di dati a disposizione) ha portato negli ultimi anni allo sviluppo di architetture dal numero di parametri enorme, con conseguente necessità di hardware all’altezza, con incluse GPU (Graphic Processing Unit) o TPU (Tensor Processing Unit) e una quantità di memoria RAM decisamente elevata. Non avendo a disposizione una macchina sufficientemente potente per il tipo di task, mi sono avvalso dell’ambiente di sviluppo Google Colab [11], che dà la possibilità di addestrare i propri modelli tramite server remoti che dispongono, a seconda del piano di abbonamento che si intende sottoscrivere, di una quantità più o meno elevata di memoria RAM e di GPU con una buona potenza di calcolo.

Il linguaggio di programmazione che ho utilizzato, integrato nell’ambiente Colab, è Python, da anni la scelta nettamente prevalente per quasi ogni task riguardante l’intelligenza artificiale, l’analisi dei dati ed il machine learning. Esso permette, essendo ad alto livello, di concentrarsi sulla logica del programma piuttosto che sui dettagli implementativi sottostanti alle architetture, e presenta una serie di librerie e pacchetti specifici per task di machine learning che sono stati ottimizzati nel corso degli anni da molteplici esperti del settore, e rappresentano oggi il punto di partenza di qualsiasi progetto nell’ambito. In particolare, per la creazione della classe di gestione del dataset e per l’implementazione della rete neurale U-Net, ho utilizzato il noto framework di sviluppo PyTorch [12], che dispone di potenti tool per la creazione di reti neurali basate sull’estensione delle classi di base già implementate nel codice della libreria. Il modello di confronto CD-FM3SF è stato invece implementato dai creatori utilizzando un’alternativa a PyTorch molto nota, ovvero TensorFlow, che mantiene principi di funzionamenti simili [17].

Per quanto riguarda l’apertura, la creazione e la visualizzazione delle immagini ho utilizzato il framework Python di OSGeo (Open Source Geospatial Foundation), una collezione di strumenti e librerie open source per l’elaborazione e la manipolazione di dati geospaziali [10]. In particolare, la libreria GDAL (Geospatial Data Abstraction Library) offre un set completo di funzionalità per leggere, scrivere e manipolare una vasta gamma di formati di immagine, tra cui il formato TIFF che è proprio quello che andremo ad utilizzare. Ho inoltre sfruttato la libreria TensorBoard [16] per la rappresentazione grafica delle varie componenti del modello U-Net. TensorBoard è un ambiente grafico integrato nel framework di TensorFlow che permette di creare diagrammi, grafici e molto altro a partire da modelli costruiti utilizzando lo stesso TensorFlow o PyTorch, con cui è compatibile. Tutto il codice a cui farò riferimento nel resto di questa relazione è disponibile sulla mia repository github [13].

Capitolo 2

Il dataset WHUS2-CD+

2.1 Struttura del dataset

Il dataset base di riferimento, chiamato WHUS2-CD, è un insieme di 32 immagini Sentinel-2A (rispettivamente 24 di training e 8 di test set) scattate sul suolo della Cina in diverse regioni, per fare in modo di ricoprire diversi tipi di ambiente e conformazione del terreno [4]. Nello specifico, gli autori del dataset hanno suddiviso le diverse tipologie di ambiente in dieci classi: terreni agricoli, foreste, prati, arbusteti, zone umide, acqua, tundra, aree urbane, aree sterili e neve o ghiaccio.

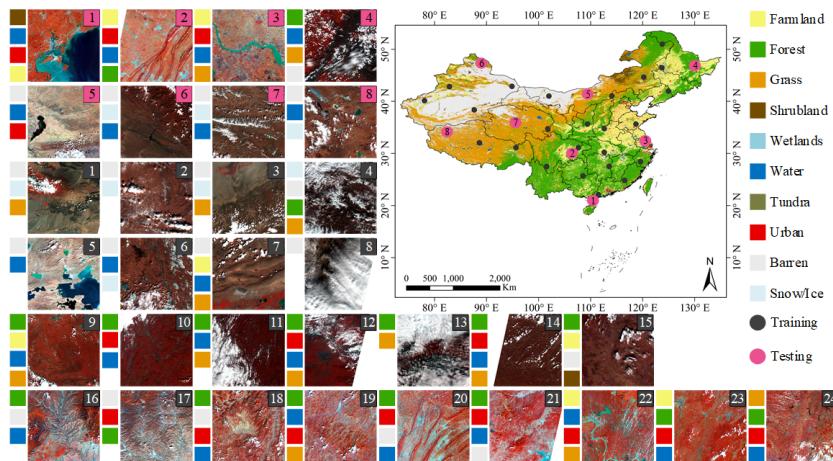


Figura 2.1. Panoramica del dataset WHUS2-CD

Inoltre sono state considerate tutte le stagioni, essendo le immagini suddivise su un intervallo temporale che va da Aprile 2018 a Maggio 2020 [4]. Complessivamente sono stati coperti circa 380.000 chilometri quadrati di superficie terrestre.

Dato che una delle più grandi sfide nel campo della cloud detection è fare in modo che il modello non confonda la neve o il ghiaccio con le nuvole

2.2 Formato iniziale, trasformazione in TIFF ed organizzazione del file system

[5], sono state aggiunte 4 immagini extra di questa tipologia, per arrivare ad una versione definitiva del dataset chiamata WHUS-CD+ contenente 36 immagini. Le maschere (label) di riferimento sono state create manualmente alla risoluzione di 10 metri, la più alta disponibile. I pixel nuvola delle maschere sono settati al valore 255, mentre tutti gli altri a 128.

2.2 Formato iniziale, trasformazione in TIFF ed organizzazione del file system

I file sono stati resi disponibili dagli autori del dataset al seguente link: <https://zenodo.org/record/5511793>. Le maschere di riferimento sono già nel formato standard TIFF (Tagged Image File Format), mentre le immagini sono organizzate in zip che contengono al loro interno una serie di file di configurazione. E' necessario seguire una serie di passi per trasformarle anch'esse in TIFF, in quanto ciò sarà poi necessario per poter addestrare il modello.

Il formato TIFF è particolarmente utilizzato in questo campo per via di alcuni vantaggi che esso offre, tra cui:

- Supporto di diversi algoritmi di compressione senza perdita di dati
- Capacità di gestire diverse profondità di colore
- Possibilità di archiviare diversi metadati all'interno del file, quali dati di geolocalizzazione, profili di colore e informazioni sulla fotocamera
- Ampia compatibilità con software di grafica ed elaborazione di immagini

Tuttavia, l'utilizzo di tale formato può portare ad ottenere file di grandi dimensioni. Nel nostro caso, dopo una prima fase di fusione dei file di configurazione delle cartelle, si ottengono delle immagini TIFF di dimensione 10980x10980 e peso che va tra i 700 e i 900 megabyte circa. Ciò rende necessaria una seconda fase di elaborazione in cui è necessario "tagliare" (fase di cut) queste immagini in patch di dimensione drasticamente inferiore, sia a livello geometrico che di peso.

La struttura del file system necessaria per addestrare il modello che prenderemo come baseline di riferimento (CD-FM3SF) è la stessa che andrò ad adottare per il mio modello, e consiste in una serie di cartelle organizzata come nelle seguenti immagini:

2.2 Formato iniziale, trasformazione in TIFF ed organizzazione del file system

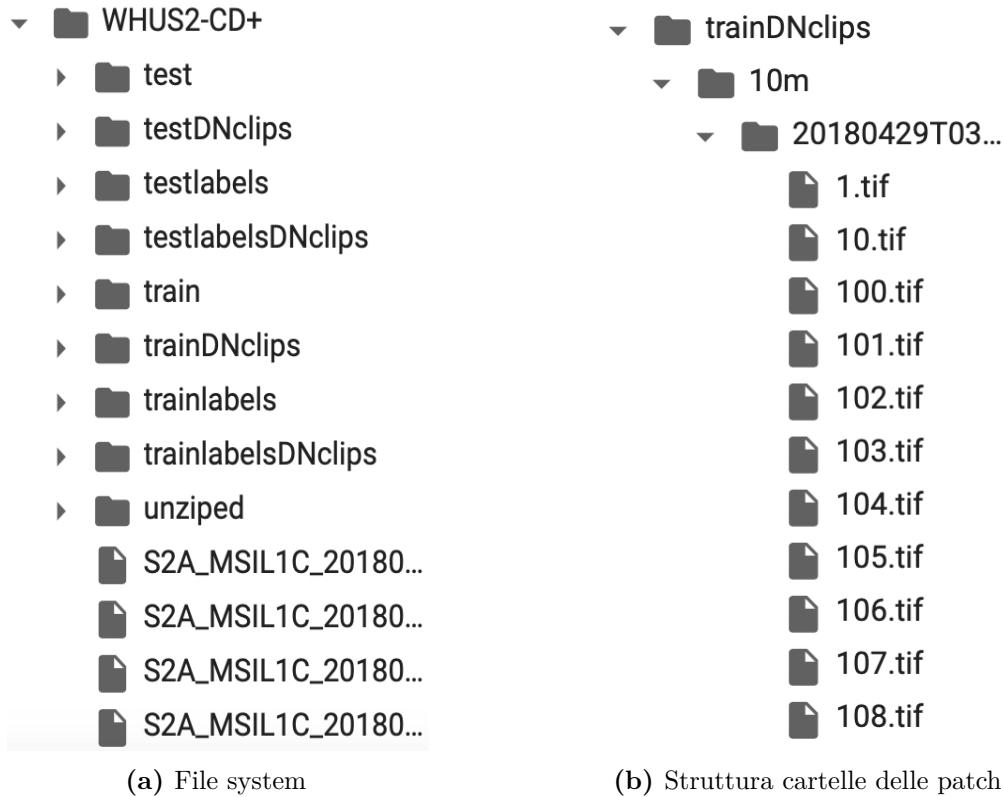


Figura 2.2. Panoramica del file system

Il risultato finale che si vuole ottenere è una serie di cartelle con dati e label di training e test come nell'immagine a destra, dove per ogni immagine abbiamo una cartella il cui nome è il codice dell'immagine stessa (meno il prefisso S2A_MSIL1C_ che è uguale per tutte) e al cui interno sono presenti 784 patch di uguale dimensione. Tale dimensione varia a seconda che siano a 10, 20 o 60 metri. Per il mio modello, allenato solamente sulle bande a 10m ovvero RGB + NIR, è pertanto necessaria solo la cartella 10m con 784 patch da 384x384 pixel per ogni immagine iniziale. Ad esse corrispondono una ad una le label, che si trovano nelle cartelle trainlabelsDNclips e testlabelsDNclips, organizzate anch'esse esattamente come descritto sopra. Tutte le cartelle nominate senza la sillaba DN nell'immagine a sinistra sono generate nel processo per arrivare al risultato finale e non sono più necessarie una volta ottenute le patch.

Il codice necessario per eseguire le varie fasi di conversione e organizzazione dei file è stato fornito dai creatori del dataset e l'ho riportato, modificato all'occorrenza e utilizzato nel Colab linkato sulla mia repository [13]. E' bene fare attenzione, nel caso in cui si volesse riutilizzarlo per riprodurre l'ambiente di addestramento, a cambiare i nomi dei path a seconda di dove si intende posizionare i file nel proprio file system.

2.3 Implementazione della classe WHUSDataset

Come accennato nell'introduzione, il framework PyTorch permette di definire classi per gestire dataset creando delle sottoclassi della predefinita "Dataset", andando a chiamare per prima cosa il suo costruttore. Definire la classe WHUSDataset a partire da Dataset non è obbligatorio ma consigliato, in quanto questo permette poi di utilizzare la classe DataLoader di PyTorch per andare a definire gli split in training, validation e test set, senza dover creare appositamente anche un dataloader specifico per il problema.

```

1  class WHUSDataset(Dataset):
2      def __init__(self, root_dir, root_dir_masks,
3                   indexes, pytorch=True):
4          super().__init__()
5          self.root_dir = root_dir
6          self.root_dir_masks = root_dir_masks

```

Listing 2.1. Parametri della classe WHUSDataset

I parametri di input sono il path della directory dove sono contenute le cartelle di patch delle immagini, il path della directory con le cartelle di patch delle maschere, una coppia di indici che servirà a scegliere quali immagini effettivamente selezionare ed un booleano (utile in seguito). Un oggetto WHUSDataset deve contenere una lista di percorsi del file system (semplici stringhe) per le immagini ed un'altra per le rispettive maschere, avendo cura che alla j-esima immagine nella prima lista corrisponda la j-esima label nella seconda. Dato che ogni cartella di patch ha un codice alfanumerico univoco (il suo nome), è sufficiente ordinare le liste una volta inseriti tutti i path per ottenerle perfettamente simmetriche.

```

1  self.filenames = [dir for dir in root_dir.iterdir()]
2  self.masksnames = [dir for dir in
3                     root_dir_masks.iterdir()]
4  self.files = []
5  self.masks = []
6
6  files = sorted([[str(f) for f in dir.iterdir()] for
7                  dir in self.filenames])
7  masks = sorted([[str(f) for f in dir.iterdir()] for
8                  dir in self.masksnames])

```

Listing 2.2. Ordinamento maschere-label

Quello che si ottiene arrivati a questo punto sono due liste con i nomi delle directory delle immagini. Un'altra operazione fondamentale è la rimozione delle patch prive di una patch corrispettiva della maschera e viceversa. Infatti, durante il processo di cut alcune patch, sia tra le immagini che tra le maschere,

vengono perdute in quanto prive di dati, e questo può generare errori durante la fase di training. Un modo semplice di risolvere il problema è direttamente ignorare tutte le patch provenienti da cartelle (immagine o maschera) che non contengono esattamente 784 patch, senza dunque aggiungere alle due liste i loro path. Una volta fatto ciò vengono creati gli array finali, delle matrici bidimensionali alla cui coppia di indici (i, j) corrisponde la j -esima patch dell' i -esima immagine.

```

1 self.filenames = [dir for dir in root_dir.iterdir() ]
2 self.masksnames = [dir for dir in
3     root_dir_masks.iterdir() ]
4 self.files = []
5 self.masks = []
6
7 files = sorted([[str(f) for f in dir.iterdir()] for
8     dir in self.filenames])
masks = sorted([[str(f) for f in dir.iterdir()] for
    dir in self.masksnames])

```

Listing 2.3. Creazione array di file definitivi

A questo punto abbiamo bisogno di funzioni che ci permettano, a partire dai path corretti, di leggere le immagini corrispondenti e trasformarle in array multidimensionali. Ho definito due diverse funzioni a seconda che si voglia aprire l' i -esima immagine o l' i -esima maschera; ciò è necessario in quanto esse hanno dimensioni diverse, essendo le maschere ovviamente a canale unico (i pixel assumono solo due valori) mentre le immagini a 4 canali (o 3 se si esclude la banda NIR).

```

1 def open_as_array(self, i, invert=True,
2     include_nir=True):
3     img = imread(self.files_array[i])
4     if invert:
5         img = img.transpose((2, 0, 1))
6     if not include_nir:
7         img = img[:3, :, :]
return (img / np.iinfo(img.dtype).max)

```

Listing 2.4. Funzione di apertura delle immagini

```

1 def open_mask(self, i, add_dims=False):
2     mask = imread(self.masks_array[i])
3     if add_dims:
4         mask = np.expand_dims(mask, axis=0)
5     return (mask / np.iinfo(mask.dtype).max)

```

Listing 2.5. Funzione di apertura delle maschere

Entrambe le funzioni eseguono tre operazioni principali:

- Chiamano una funzione ausiliaria *imgread()* che apre l'immagine correttamente e restituisce un array
- Effettuano alcune operazioni aggiuntive a seconda di determinati valori dei parametri in input
- Normalizzano i valori dell'array di output

La funzione *imgread()* in questione deve essere in grado di aprire un file TIFF, ed è quindi un'operazione più complessa della lettura di una semplice immagine PNG o JPEG. Fortunatamente, il pacchetto GDAL della libreria OSGeo fornisce alcuni strumenti aggiuntivi che permettono di definire la funzione nel modo seguente:

```

1 def imgread(path):
2     img = gdal.Open(path)
3     c = img.RasterCount
4     img_arr = img.ReadAsArray()
5     if c>1:
6         img_arr = img_arr.swapaxes(1,0)
7         img_arr = img_arr.swapaxes(2,1)
8     del img
9     return img_arr

```

Listing 2.6. Funzione di apertura per immagini TIFF

Dopodichè, nel caso delle immagini a 4 bande abbiamo un parametro opzionale *invert* che se settato a true inverte degli assi dell'array multidimensionale; questo è utile per fare in modo che le dimensioni dell'input si adattino a quelle richieste dalla rete neurale. Inoltre, il parametro *include_nir* permette di decidere se includere o meno la quarta banda. Nel nostro caso la rimuoveremo solo quando dovremo andare a visualizzare a schermo le immagini. Per le maschere è invece presente un parametro *add_dims* che permette eventualmente di aggiungere una dimensione artificialmente all'array, anche questo per motivi di dimensionalità dell'input della rete. Infine, viene sfruttato un metodo apposito di numpy per normalizzare i valori dei pixel in un intervallo limitato.

Ora che è possibile accedere ad ogni immagine e trasformarla in un array, l'ultimo passo rimasto è fare in modo che la classe DataLoader di PyTorch sia in grado di gestire il nostro dataset; per farlo dobbiamo andare a definire il metodo *__getitem__*, cioè come effettivamente si accede agli elementi del Dataset mediante indicizzazione. Nel nostro caso gli elementi sono le coppie (immagine, maschera), e definiamo il metodo semplicemente servendoci delle funzioni già create:

```

1 def __getitem__(self, i):
2     x = torch.tensor(self.open_as_array(i,
3         invert=self.pytorch), dtype=torch.float32)
4     y = torch.tensor(self.open_mask(i,
5         add_dims=False), dtype=torch.int64)
6     return x, y

```

Listing 2.7. Metodo `__getitem__`

2.4 Visualizzazione delle immagini

Essendo a 4 bande, le patch a 10 metri non sono direttamente visualizzabili come normali immagini RGB. E' dunque necessario rimuovere la banda NIR ed effettuare alcune operazioni algebriche sui pixel per riuscire a stamparle a schermo correttamente.

```

1 def acv_inv_proc(img):
2     img_rgb = img*255.0
3     return img_rgb.astype(np.float32)
4
5 def get_normal_picture(image):
6     image = image.numpy()
7     image= image[[0,1,2],:,:]
8     low ,high=np.percentile(image ,(2 ,98 ))
9     image[low> image]=low
10    image[image>high]=high
11    rescaled_img=(image-low)/(high-low)
12    x_image = acv_inv_proc(rescaled_img)
13    return torch.tensor(np.uint8(x_image))

```

Listing 2.8. Operazioni per visualizzare un'immagine a 4 bande RGB + NIR

La funzione `get_normal_picture` prende in input un tensore che rappresenta un'immagine a 4 bande e lo trasforma in un array numpy. Dopo aver eliminato la quarta banda si procede ad una normalizzazione; in particolare vengono estratti il secondo (low) ed il novantottesimo (high) percentile e si settano tutti i pixel minori di low o maggiori di high a questi stessi valori, per fare in modo che eventuali outlier non compromettano la visibilità dell'immagine. Dopodichè si procede ad una riscalatura e vengono riportati tutti i valori nell'intervallo 0-255. Infine viene rifatto il casting inverso da array numpy a tensore.

Andando a stampare a schermo le coppie (immagine, maschera) tramite la libreria matplotlib, il risultato ottenuto è il seguente:

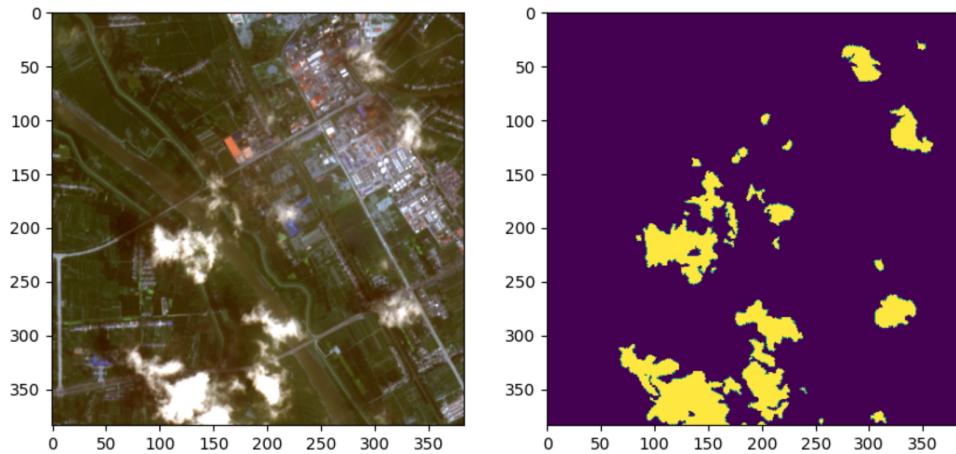


Figura 2.3. Immagine-label 1

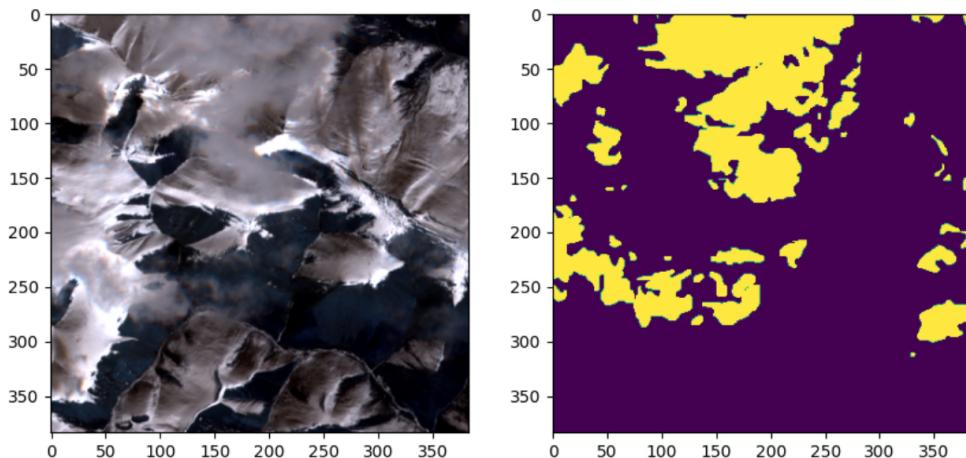


Figura 2.4. Immagine-label 2

Avere la possibilità di visualizzare correttamente le immagini è molto importante, in quanto ci permetterà di individuare quali particolari pattern dell'ambiente confondono il nostro modello inducendolo a sbagliare, e quali sono invece correttamente individuati. Infatti, per quanto sia importante valutare la bontà di una rete neurale di classificazione basandosi su metriche matematiche (che verranno discusse in seguito), è difficile capire a partire da esse quali feature spaziali o di colore possano aver confuso il predittore. Poter valutare visivamente il dataset è quindi di vitale importanza.

Capitolo 3

Un modello di confronto

3.1 Struttura della rete neurale CD-FM3SF

Come anticipato nell'introduzione, da anni le reti neurali convoluzionali rappresentano lo stato dell'arte nel campo della cloud detection, e varie tecniche sono state introdotte per far fronte alle sfide più complesse. In particolare, il mio studio si è concentrato su un modello noto come CD-FM3SF [4], una CNN basata sul modello U-Net (descritto nel dettaglio nel capitolo 4) che aggiunge alcune nuove operazioni per aumentare l'efficacia nell'attuare il task. La struttura della rete è la seguente:

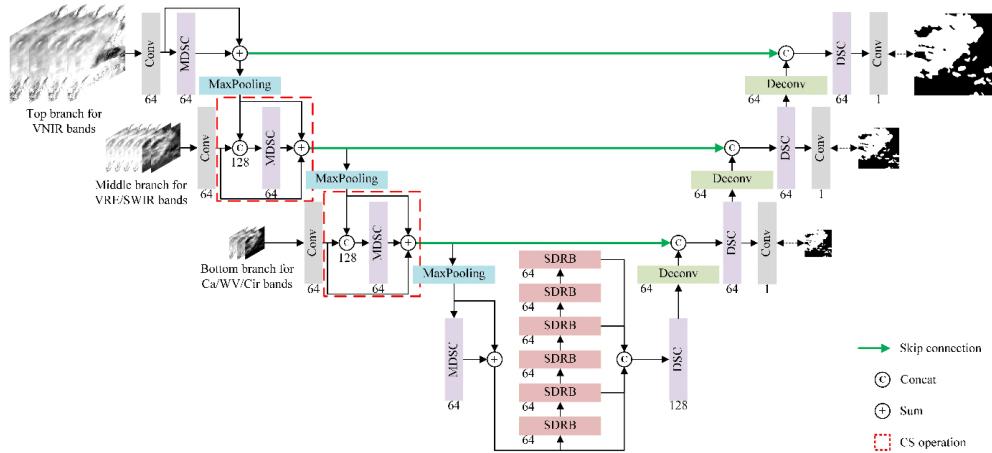


Figura 3.1. Struttura del modello CD-FM3SF

Sebbene lo scopo di questa relazione non sia quello di approfondire i dettagli della seguente rete, bensì quello di proporre un modello alternativo più semplice, è comunque interessante avere una panoramica di come le particolarità introdotte dagli autori influiscano effettivamente sui risultati ottenuti.

La prima scelta originale è stata quella di fornire in input al modello tutti e tre i gruppi di bande spettrali (10, 20 e 60 metri) e non solo quelle a 10

metri come fatto da molti modelli precedenti. Questo permette di cogliere le differenze tra alcuni tipi di copertura superficiale in maniera più efficace, sfruttando ad esempio la TOA reflectance. Il modello è stato addestrato dopo aver attuato le stesse operazioni sul dataset descritte nel capitolo precedente, più una fase di data augmentation per avere ancora più patch. In totale, quindi, gli autori hanno allenato il modello su 367145 patch, decisamente di più di quelle che sono riuscito ad utilizzare io per il mio modello U-Net. La grande varietà di immagini di input è stata sicuramente un fattore determinante per la riuscita dell'addestramento.

Come si vede dalla struttura riportata nell'immagine, le tre tipologie di bande vengono prese in input dal modello in tre rami diversi e poi fuse insieme dopo aver subito alcune operazioni nei rispettivi rami, ovvero una convoluzione, un pooling (operazione volta a ridurre le dimensioni di un'immagine in input) e, nel mezzo, il passaggio per un layer MDSC, proprio della rete, che essenzialmente ha la proprietà di estrarre e fondere feature contestuali sia a corto che a lungo raggio dalle immagini, ed è una combinazione di più layer DSC [1] (depth-wise separable convolution).

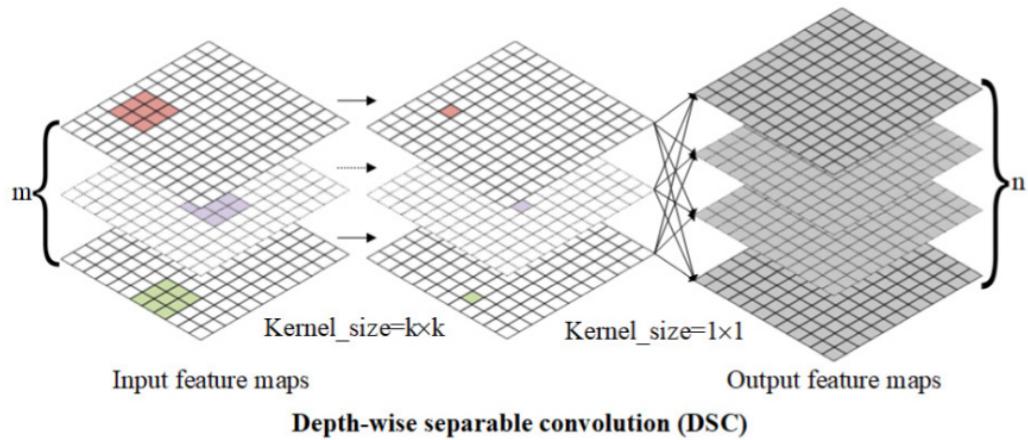


Figura 3.2. Struttura di un layer DSC

La metodologia di fusione delle bande è una nuova operazione proposta dagli autori detta CS (Concatenation and Sum); l'idea è sfruttare un'architettura residuale, in cui feature estratte in un certo punto della rete vengono sommate pixel per pixel a quelle ottenute nel frattempo passando per altri layer. Ad esempio, nel primo ramo il risultato della convoluzione viene sommato a quello ottenuto dall'output del layer MDSC, prima di effettuare il pooling della loro somma. Per motivi dimensionali, tuttavia, non sarebbe possibile sommare direttamente pixel per pixel anche i risultati uscenti da rami diversi, perciò viene effettuata una concatenazione tra i risultati della convoluzione del secondo e del terzo ramo con, rispettivamente, gli output del pooling del primo e del secondo ramo. Il risultato della concatenazione va direttamente in

un layer MDSC in entrambi i rami, e questo permette la fusione effettiva delle feature multispettrali.

La seconda fase di attraversamento della rete da parte dei dati comprende una serie di deconvoluzioni (convoluzioni trasposte che aumentano la dimensione, i dettagli relativi sono riportati nel capitolo 4) e layer MDSC; tuttavia, prima di ciò i dati vengono passati in un'altra nuova tipologia di layer, ovvero SDRB, che sfrutta un'architettura residuale con blocchi di convoluzione con dilation e convoluzioni condivise. La convoluzione con dilation è in grado di ottenere feature a lungo raggio dalle immagini mediante un metodo di interpolazione degli zeri tra gli elementi di un normale kernel convolutivo. La convoluzione condivisa sfrutta lo stesso kernel per estrarre informazioni spaziali da diverse immagini. La combinazione di questi due tipi di convoluzione nel blocco SDRB permette di ottenere i vantaggi della dilation evitando il cosiddetto grid-effect, ovvero la perdita di informazione a corto raggio.

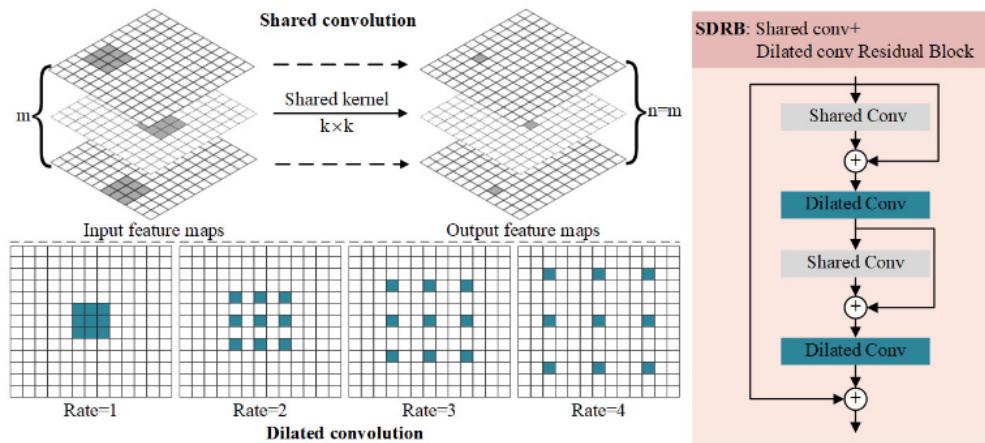


Figura 3.3. Rappresentazione delle due operazioni di convoluzione con dilation e shared convolution, e del blocco SDRB

Oltre ad encoder e decoder sono presenti delle skip connections per permettere il passaggio di informazioni posizionali delle feature. Così come le deconvoluzioni, anche il funzionamento di queste connessioni viene descritto dettagliatamente nel capitolo 4 dedicato alla struttura della mia rete U-Net.

3.2 Traguardi raggiunti

La rete descritta ha ottenuto performance migliori, sotto vari aspetti, dei modelli più popolari noti fino al momento della pubblicazione del paper di riferimento [4]. La seguente tabella mostra le performance, valutate secondo varie metriche, di alcuni modelli noti tradizionali e neurali, a confronto con quelle di CD-FM3SF:

Methods	OA	Precision	Recall	F1-score	IoU
Fmask	92.62%	59.90%	95.14%	0.7098	0.5713
Sen2Cor	90.89%	64.20%	69.36%	0.6252	0.4698
MSCFF	98.40%	96.42%	78.77%	0.8637	0.7645
MSCFF-13	98.78%	96.22%	86.48%	0.9102	0.8363
RS-Net	98.03%	88.77%	79.90%	0.8372	0.7259
RS-Net-13	98.59%	93.63%	85.06%	0.8900	0.8038
CD-FM3SF	98.86%	96.50%	87.75%	0.9186	0.8503

Figura 3.4. Confronto dei risultati di vari modelli sulle immagini di test

Il modello esaminato risulta essere il migliore sia in termini di falsi positivi che in termini di falsi negativi, come evidenziato dagli alti valori di Precision e Recall. In generale, i metodi tradizionali basati sulle informazioni spettrali come Fmask e Sen2Cor mostrano buoni risultati sulle aree verdi o acquose (bassi valori di TOA reflectance), ma falliscono per valori di TOA reflectance simile alle nuvole (ad esempio, zone urbane o neve); i metodi di Deep Learning quali MSCFF e RS-Net sono molto efficaci su svariate tipologie di copertura comprese urbana e terreno sterile, ma tendono comunque a fallire di fronte alla neve utilizzando soltanto le bande RGB + NIR (è, inoltre, il caso del mio modello U-Net). CD-FM3SF, prendendo in input tutte le bande disponibili e sfruttando l'architettura descritta sopra, risulta performante a prescindere dalla copertura superficiale presa in esame, ed è quindi considerabile, sotto praticamente ogni aspetto, l'attuale stato dell'arte.

Capitolo 4

Struttura della rete U-Net

4.1 Panoramica generale

Una U-Net è una rete neurale convoluzionale formata solitamente da tre componenti: encoder, bridge e decoder. L'encoder è una serie di layer convoluzionali, volti ad estrarre le feature dalle immagini, con l'aggiunta di operazioni di pooling per ridurre la loro dimensione. Il bridge è anch'esso una serie di convoluzioni ma senza utilizzare il pooling, e quindi senza ridurre ulteriormente la dimensione. Infine, il decoder applica una serie di convoluzioni classiche ed almeno una trasposta, in cui viene aumentata la dimensione dell'immagine fino a tornare a quella originale, per poi produrre un output. La struttura descritta porta ad una forma "ad U", come evidenziato dal seguente schema concettuale [19]:

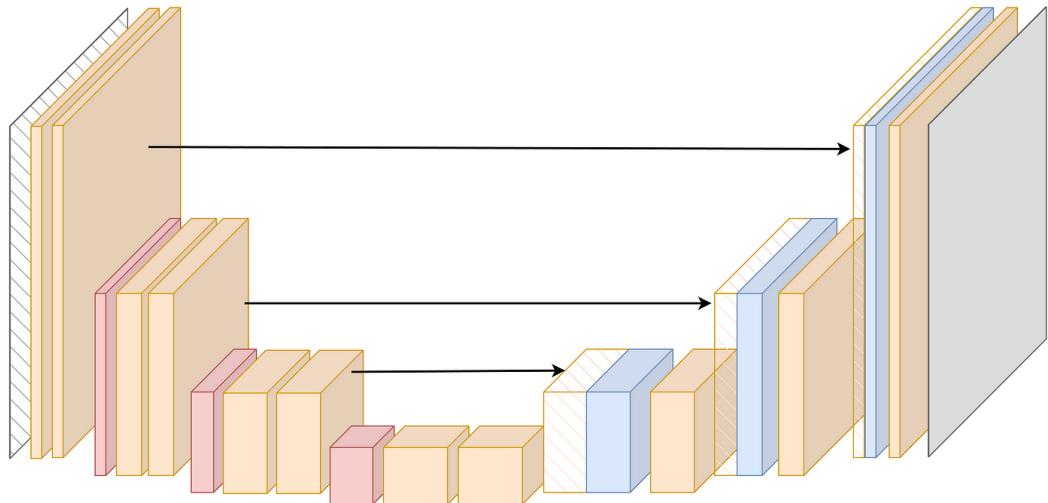


Figura 4.1. Diagramma concettuale U-Net

Inoltre, all'output di ogni convoluzione in ogni parte della rete viene applicata una funzione di attivazione, necessaria per la non linearità del modello. Le frecce presenti nel diagramma rappresentano una caratteristica cardine

dell’architettura U-Net, ovvero le skip connections, elementi che collegano direttamente alcuni layer convoluzionali dell’encoder al decoder e che differenziano questa particolare struttura da un classico autoencoder.

Nel mio caso, la scelta è ricaduta su una versione semplificata del modello generale ed orfana del bridge. Difatti, essendoci già un sufficiente numero di layer convoluzionali nella fase di encoding, non sempre è necessario aggiungerne altri se si ottiene un buon risultato senza. Nel seguito andrò ad analizzare dettagliatamente ogni componente della mia rete, considerando separatamente l'encoder, il decoder e le skip connections . Prima di fare ciò, ecco un diagramma, generato grazie a TensorBoard, che mostra la struttura generale che ho creato nella sua totalità:

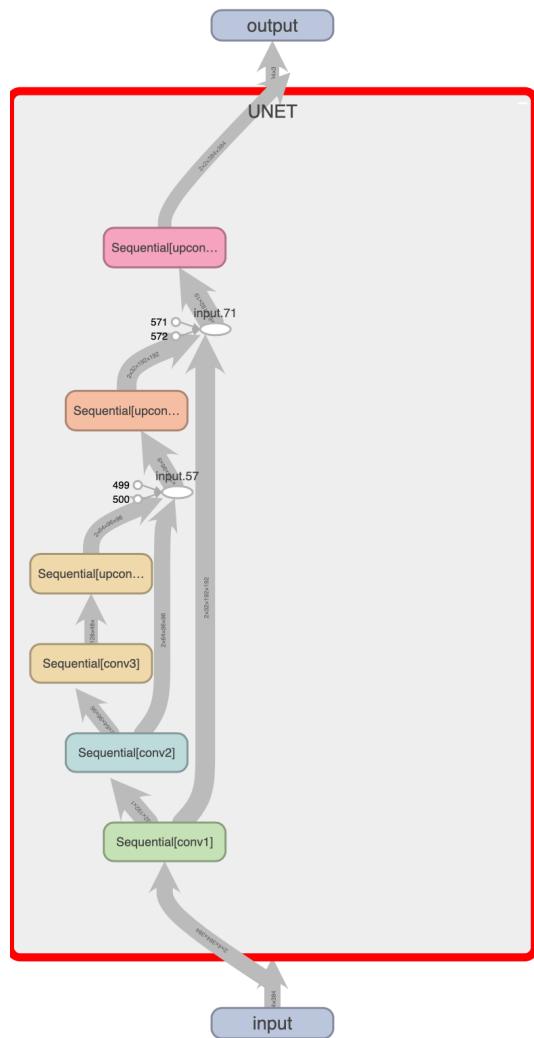


Figura 4.2. Struttura della U-Net

Essa presenta 3 blocchi di contrazione e 3 blocchi di espansione a cui vengono aggiunte le skip connections, e viene creata a partire dalla classe predefinita nn.Module della libreria PyTorch. Vengono specificati come parametri il numero di canali di input della rete (nel nostro caso 4, ovvero RGB + NIR) e il numero di canali di output (nel nostro caso 2, rappresentano rispettivamente la probabilità che un pixel sia nuvola e quella che sia non nuvola).

```

1  class UNET(torch.nn.Module):
2      def __init__(self, in_channels, out_channels):
3          super().__init__()
4
5          self.conv1 =
6              self.contract_block(in_channels, 32, 7, 3)
7          self.conv2 = self.contract_block(32, 64, 3,
8              1)
9          self.conv3 = self.contract_block(64, 128, 3,
10             1)
11
12         self.upconv3 = self.expand_block(128, 64, 3,
13             1)
14         self.upconv2 = self.expand_block(64*2, 32,
15             3, 1)
16         self.upconv1 = self.expand_block(32*2,
17             out_channels, 3, 1)
18
19
20     def __call__(self, x):
21
22         # downsampling
23         conv1 = self.conv1(x)
24         conv2 = self.conv2(conv1)
25         conv3 = self.conv3(conv2)
26
27         #upsampling
28         upconv3 = self.upconv3(conv3)
29         upconv2 = self.upconv2(torch.cat([upconv3,
30             conv2], 1))
31         upconv1 = self.upconv1(torch.cat([upconv2,
32             conv1], 1))
33
34
35     return upconv1

```

Listing 4.1. Struttura della U-Net

4.2 Blocco di contrazione

Il blocco di contrazione contiene al suo interno una serie di layer convoluzionali, nel nostro caso 2, intervallati da due operazioni fondamentali, ovvero la batch normalization e l'applicazione di una funzione di attivazione. Infine viene effettuato un pooling volto a ridurre il numero dei pixel.

```

1 def contract_block(self, in_channels, out_channels,
2                     kernel_size, padding):
3
4     contract = torch.nn.Sequential(
5         torch.nn.Conv2d(in_channels, out_channels,
6                         kernel_size=kernel_size, stride=1,
7                         padding=padding),
8         torch.nn.BatchNorm2d(out_channels),
9         torch.nn.ReLU(),
10        torch.nn.Conv2d(out_channels, out_channels,
11                         kernel_size=kernel_size, stride=1,
12                         padding=padding),
13        torch.nn.BatchNorm2d(out_channels),
14        torch.nn.ReLU(),
15        torch.nn.MaxPool2d(kernel_size=3, stride=2,
16                           padding=1)
17    )
18
19    return contract

```

Listing 4.2. Blocco di contrazione

L'operazione di convoluzione è effettuata dai layer nn.Conv2d, che applicano un kernel della dimensione data in input con stride uguale ad 1, cioè senza saltare dei pixel e ridurre l'immagine di per sè. L'operazione di batch normalization è invece utile a ridurre la covarianza tra i canali di output della convoluzione che la precede [8], e questo è ottenuto prima calcolando le medie e deviazioni standard dei minibatch, normalizzando ogni elemento di essi secondo i valori ottenuti e poi effettuando un'operazione detta di "scale and shift", per evitare che la normalizzazione cambi sostanzialmente ciò che il layer convoluzionale può rappresentare. Queste fasi sono riassunte e formalizzate matematicamente nel seguente algoritmo [8]:

Input:	Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;
	Parameters to be learned: γ, β
Output:	$\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$
	$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$ // mini-batch mean
	$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$ // mini-batch variance
	$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$ // normalize
	$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$ // scale and shift

Figura 4.3. Algoritmo per la batch normalization

Inoltre, diversi esperimenti provano l'aumento della capacità di generalizzazione delle CNN addestrate mediante batch normalization, andando a ridurre o addirittura eliminare l'overfitting a seconda dei casi [8]

Per quanto riguarda la funzione di attivazione, ho scelto la tradizionale ReLu in quanto vari studi dimostrano la sua efficacia per l'applicazione della non linearità nelle reti neurali [6]. Essa non è altro che una funzione a tratti che vale 0 per input negativi e corrisponde all'identità per valori positivi. Ciò implica che, in caso di input della rete sempre positivi (come ad esempio i pixel di un'immagine), pesi diventati negativi durante l'addestramento portino il neurone corrispondente a "morire" per sempre senza possibilità di influire da quel momento in poi. La sua espressione matematica è la seguente:

$$f(x) = \max(0, x)$$

Il fatto che certi neuroni "muoiano" in seguito al suo effetto può a volte rivelarsi un vantaggio, diventando difatti un dropout più o meno randomico dei pesi.

Infine, ho utilizzato un MaxPooling con stride 2 che applica un kernel 3x3 all'immagine lasciando solo i valori maggiori, andando così a ridurne la dimensione, mantenendo le feature più rilevanti. Nello specifico, essendo l'altezza e la larghezza delle nostre immagini in input uguali, ed indicandole con il valore W_{in} , otterremo una dimensione di output W_{out} data dalla seguente formula:

$$W_{out} = \frac{W_{in} + 2 \cdot padding - 1}{stride} + 1$$

4.3 Blocco di espansione

Il blocco di espansione è difatti molto simile a quello di contrazione, ma differisce nell'ultima operazione applicata; anzichè effetturare un pooling come

fatto precedentemente, alla fine del blocco troviamo un layer che effettua una convoluzione trasposta dell'input, talvolta detta impropriamente deconvoluzione (l'operatore applicato non è, a livello matematico, l'effettiva funzione inversa della convoluzione). Ci sono più modi per ottenere una convoluzione trasposta con effetto di upsampling, uno di questi è combinare nella maniera corretta i parametri di stride e padding in modo da applicare ad un'immagine di una certa dimensione un filtro di dimensione maggiore, il contrario di ciò che succede normalmente con una convoluzione.

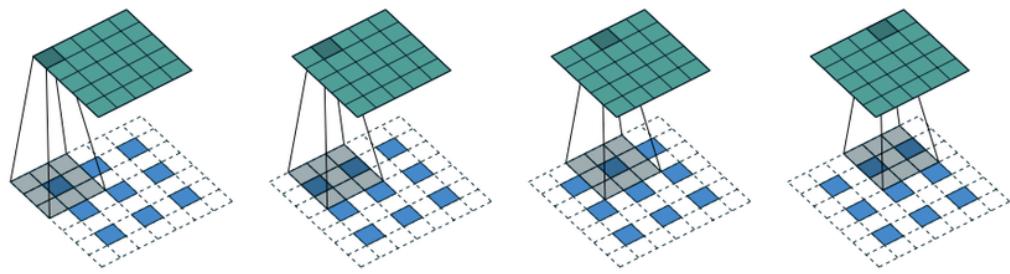


Figura 4.4. Operazione di convoluzione trasposta mediante stride e padding

L'esempio mostrato nella figura sfrutta sia una stride diversa da 1 che del padding (di zeri), ma esistono molte varianti che non utilizzano rispettivamente il padding o la stride. Quale scegliere dipende dalle dimensioni dell'immagine in input, da quelle del kernel e dalle dimensioni di output che si desidera ottenere. Una trattazione dettagliata dell'aritmetica delle convoluzioni trasposte è disponibile qui [7].

Prendiamo ora in esame il caso specifico della convoluzione trasposta applicata dal layer ConvTranspose2d che ho utilizzato. Data W_{in} , la dimensione di uscita W_{out} a seguito della convoluzione trasposta si trova mediante la seguente formula:

$$W_{out} = (W_{in} - 1) \cdot stride - 2 \cdot padding + outputpadding + 1$$

Questa formula vale se un altro parametro opzionale, ovvero la dilation, è lasciato come predefinito a zero. Il blocco di espansione è quindi definito attraverso il seguente codice:

```

1 def expand_block(self, in_channels, out_channels,
2   kernel_size, padding):
3
4     expand = torch.nn.Sequential(
5       torch.nn.Conv2d(in_channels, out_channels,
6         kernel_size, stride=1, padding=padding),
7       torch.nn.BatchNorm2d(out_channels),
8       torch.nn.ReLU(),
9       torch.nn.Conv2d(out_channels, out_channels,
10      kernel_size, stride=1, padding=padding),
11      torch.nn.BatchNorm2d(out_channels),
12      torch.nn.ReLU(),
13      torch.nn.ConvTranspose2d(out_channels,
14        out_channels, kernel_size=3, stride=2,
15        padding=1, output_padding=1)
16    )
17
18    return expand

```

Listing 4.3. Blocco di espansione

Confrontate, le due tipologie di blocchi appaiono così:

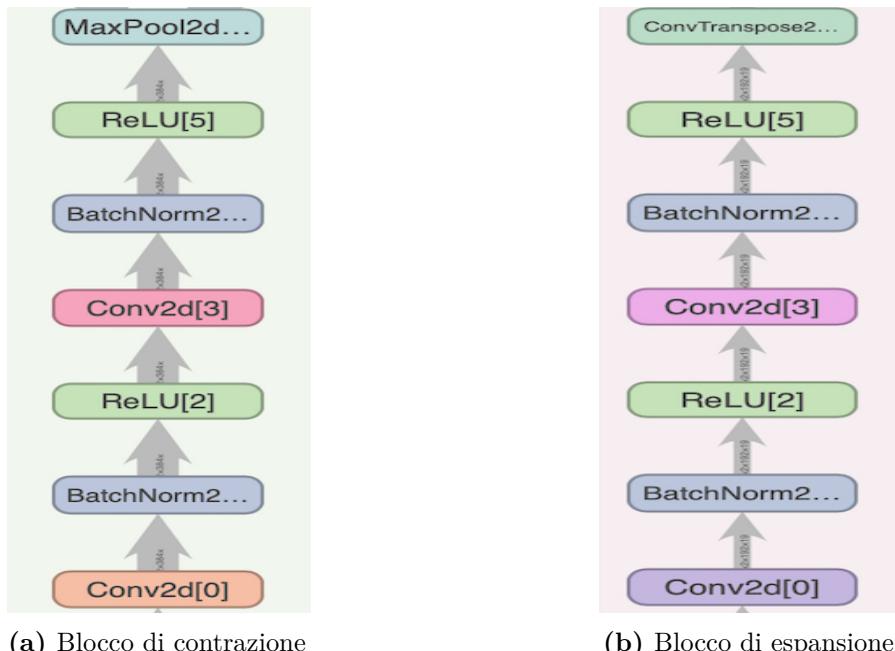


Figura 4.5. Confronto tra le due tipologie di blocchi

4.4 Skip connections

Come anticipato, le skip connections (o connessioni di salto) differenziano la U-Net da un semplice autoencoder. In un autoencoder (volto alla compressione e decompressione di un'immagine), infatti, encoder e decoder devono essere completamente separati, mentre per quanto riguarda la U-Net a scopi di classificazione/segmentazione non abbiamo questa restrizione. Sfruttiamo dunque queste connessioni aggiuntive per trasferire informazione dai layer convoluzionali dell'encoder direttamente a quelli di deconvoluzione del decoder; nello specifico, ciò che viene passato ai layer del decoder è la posizione originale della feature estratta dall'encoder. Questo viene ottenuto concatenando l'ultimo layer convoluzionale in un blocco di contrazione al primo layer del blocco di espansione corrispondente (la U-Net è una struttura simmetrica e quindi la dimensione di due layer opposti sarà la stessa). Ciò permette alla rete di performare due diverse operazioni fondamentali [19]:

- Feature extraction, tramite il passaggio delle feature dal layer precedente al successivo
- Feature localization, tramite il passaggio dell'informazione riguardando la posizione delle feature nell'immagine tra due layer convoluzionali opposti

Essendo dunque sufficiente, in concreto, una concatenazione tra tensori per creare delle skip connections, ho utilizzato il metodo `torch.cat` andando a collegare i layer come ho descritto.

```

1 #upsampling
2 upconv3 = self.upconv3(conv3)
3 upconv2 = self.upconv2(torch.cat([upconv3, conv2],
4                               1))
5 upconv1 = self.upconv1(torch.cat([upconv2, conv1],
6                               1))
```

Listing 4.4. Concatenazione tra tensori per le skip connections

Capitolo 5

Addestramento della rete

5.1 Organizzazione dell’addestramento

Così come per qualsiasi problema di ML, prima di cominciare ad addestrare la rete è necessaria un’accurata selezione dei dati di input. Di partenza, come anticipato nel capitolo dedicato al dataset, abbiamo disponibili 24 immagini di training set, divise ognuna, dopo il processo già descritto, in 784 patch per immagine. Dunque ci dovremmo ritrovare con 18816 patch a 10 metri. Tuttavia, alcune di esse risultano prive di dati dopo il processo di cut (e questo è normale secondo quanto descritto dagli autori del dataset), dunque vanno scartate e va scartata la corrispettiva label. Facendolo mi sono ritrovato con circa 11760 patch, ovvero quelle corrispondenti a 15 immagini. La mia idea iniziale è stata ovviamente quella di utilizzarle tutte, ma purtroppo i limiti tecnologici dell’ambiente di sviluppo Colab me l’hanno impedito. Infatti, durante l’allenamento i 25 GB di RAM disponibile (sotto piano di abbonamento Pro) si esauriscono dopo poche epoche se esse sono troppo lunghe, ed il runtime va in crash. Per evitare di dover ricominciare da zero l’addestramento ogni volta ho dunque implementato un meccanismo di salvataggio dei progressi; tuttavia è comunque necessario che il modello faccia in tempo a finire almeno un’epoca prima che l’utilizzo della RAM raggiunga il limite disponibile. Di conseguenza, dopo una serie di prove, ho scelto di utilizzare per il training 1568 patch, selezionate in modo che fossero presenti sia immagini con poche nuvole che immagini con molte, ed un buon equilibrio tra zone rurali e urbane. Nonostante possano sembrare poche rispetto al totale disponibile, questo mi ha permesso di addestrare la rete per un numero sufficiente di epoche in tempi accettabili e non ha influito in maniera troppo pesante sui risultati ottenuti.

Una volta fatto ciò, mi sono servito della classe `DataLoader` di PyTorch per dividere il training set iniziale in training e validation. Ho dunque usato circa un quinto dei dati per validation e il restante come training, e scelto una dimensione di batch pari a 12. Così facendo, la rete effettua 100 step di training e 31 step di validation ogni epoca. E’ molto importante fare in modo che la divisione in training e validation set sia fatta in maniera randomica ma secondo

un seed impostato manualmente e non variabile a seconda dell’esecuzione. Infatti, essendo necessario rieseguire tutte le celle di codice da zero ogni volta che si riprende l’addestramento, se cambiasse il seed dello split cambierebbe anche il modo in cui sono divisi i dati, ed il modello si ritroverebbe ad addestramento in corso dati di validation che prima erano di training e viceversa. Questo sarebbe un errore molto grave, in quanto porterebbe il modello a fare validation su dati identici a quelli su cui si è addestrato precedentemente, danneggiando pesantemente la capacità di generalizzare del modello e dunque aumentando il rischio di overfitting.

```

1 train_ds, valid_ds =
2     torch.utils.data.random_split(data, (1200, 368),
3                                     generator=torch.Generator().manual_seed(10))
4 train_dl = DataLoader(train_ds, batch_size=12,
5     shuffle=True)
6 valid_dl = DataLoader(valid_ds, batch_size=12,
7     shuffle=True)
8 test_dl = DataLoader(test_data, batch_size = 12,
9     shuffle = True)

```

Listing 5.1. Divisione del dataset in training, validation e test set

Ora che i dati sono suddivisi in maniera corretta, è importante soffermarsi su quanto le due classi, banalmente nuvola e non nuvola, siano effettivamente sbilanciate. Ciò influisce infatti sia sull’andamento dell’addestramento che sulla complessiva valutazione dei risultati. Per ovviare a questa problematica è necessario seguire tre passi:

- Contare i pixel nuvola (valore 1) e non nuvola (valore 0) delle maschere
- Andare a bilanciare il calcolo della loss in base a quanto maggiore è la quantità di pixel di una classe rispetto all’altra
- Scegliere una metrica di valutazione che tenga conto dello sbilanciamento

I punti 2 e 3 sono trattati rispettivamente nelle sezioni dedicate alla loss e alle metriche di questo capitolo. Per quanto riguarda il calcolo del numero di pixel nuvola e non, ho sfruttato i metodi della libreria NumPy per evitare di dover effettuare troppi cicli for che, dovendo esaminare pixel per pixel le maschere aprendole una per una, prenderebbe molto tempo. Dunque apro ogni patch maschera e restituisco il tensore corrispondente mediante il metodo `__get_item__` (che ho definito nella classe WHUSDataset per poter restituire le patch mediante indicizzazione), che a sua volta chiama la funzione `open_mask`, dopodichè rendo l’array piatto (unidimensionale) e conto i pixel nuvola con `np.count_nonzero`.

```

1 def count_pixels(data):
2     num_clouds = 0
3     num_not = 0
4     for i in range(len(data)):
5         array = data[i][1].flatten()
6         nc = np.count_nonzero(array)
7         nn = len(array) - nc
8         num_clouds += nc
9         num_not += nn
10    return (num_not, num_clouds)

```

Listing 5.2. Funzione per il calcolo dei pixel nuvola

I pixel nuvola, calcolati mediante la funzione appena mostrata, sono circa lo 0.026% dei pixel non nuvola. Anche se ipoteticamente si sarebbe potuto ridurre questo sbilanciamento alla radice, evitando di selezionare immagini senza alcuna nuvola (che ovviamente influiscono pesantemente sulla proporzione), ciò rischierebbe di diminuire la capacità del modello di distinguere alcuni pattern urbanistici o naturali dalle nuvole e potrebbe rivelarsi controproducente.

5.2 Salvataggio dei progressi

Come anticipato, poter salvare i progressi di addestramento della rete è fondamentale quando non si ha la possibilità di effettuare il processo di training in una sola sessione. Per farlo, la libreria PyTorch fornisce *torch.save* per memorizzare le informazioni rilevanti e *torch.load* per caricarle una volta ripreso l'addestramento. Nello specifico, le informazioni cruciali da memorizzare sono quattro:

- Il valore dei pesi del modello
- Lo stato dell'ottimizzatore
- L'ultima epoca completata (sia fase di training che di validation)
- Lo storico dei valori di loss e di accuracy fino a quell'epoca.

Per farlo, la funzione *torch.save* prende in input un dizionario; alla chiave *model_state_dict* associamo lo stato (valore dei parametri e altre informazioni rilevanti) del modello tramite la chiamata al metodo built-in *model.state_dict()* della classe *nn.Module*, che è classe padre di quella del mio modello. Specularmente, carichiamo questo valore con il metodo *model.load_state_dict()*. Inoltre, analoghi metodi built-in sono anche disponibili per salvare lo stato dell'ottimizzatore. E' un errore comune salvare i parametri del modello senza però curarsi di fare lo stesso con quelli della funzione di ottimizzazione, questo porta a riprendere l'addestramento con gli ultimi pesi ottenuti ma senza tenere traccia delle variabili interne dell'ottimizzatore quali (prendendo come esempio

Adam) i momenti dei gradienti e gli accumuli delle derivate seconde; ciò rischia di influire sensibilmente sulla convergenza del modello. Inoltre, se è presente una componente inerziale come in Adam, che è volta a indirizzare l'upgrade dei pesi verso una certa direzione in base agli update passati, si potrebbe rallentare molto il processo di ottimizzazione o rischiare divergenza. Infine, se utilizziamo iperparametri variabili come il learning rate adattivo o coefficienti di regolarizzazione dinamici, essi verranno resettati ogni ripresa, perdendo l'effetto desiderato.

```

1 torch.save({
2     'epoch': epoch + 1,
3     'model_state_dict': model.state_dict(),
4     'optimizer_state_dict': optimizer.state_dict(),
5     'valid_loss': valid_loss,
6     'train_loss': train_loss,
7     'valid_acc': valid_acc,
8     'train_acc': train_acc
9 }, CHECKPOINT_PATH + '/' + model_name)

```

Listing 5.3. Salvataggio dei parametri fondamentali

```

1 checkpoint = torch.load(CHECKPOINT_PATH + '/' +
2     model_name)
3 model.load_state_dict(checkpoint['model_state_dict'])
4 optimizer.load_state_dict(
5     checkpoint['optimizer_state_dict'])
6 last_epoch = checkpoint['epoch']
7 train_loss = checkpoint['train_loss']
8 train_acc = checkpoint['train_acc']
9 valid_loss = checkpoint['valid_loss']
10 valid_acc = checkpoint['valid_acc']

```

Listing 5.4. Caricamento dei parametri salvati

5.3 Funzione di loss e bilanciamento delle classi

La funzione di loss che ho scelto di utilizzare è la binary cross entropy loss (BCE). Chiamando y_{p_i} l'i-esima predizione e y_{t_i} la ground truth corrispondente, la BCE su N sample è data dalla seguente formula:

$$BCE = -\frac{1}{N} \cdot \sum_{i=0}^N y_{t_i} \log(y_{p_i}) + (1 - y_{t_i}) \log(1 - y_{p_i})$$

Solitamente, le y_i sono valori tra 0 ed 1 e rappresentano la probabilità che l'i-esimo sample sia di una delle due classi; ciò si ottiene, ad esempio, tramite una sigmoide o una softmax posta nell'ultimo layer di una rete neurale, o comunque mediante un'altra funzione che normalizza l'input in un range tra

0 ed 1 applicata sui logits. Con logits si intendono i valori assegnati ad ogni classe da un modello predittivo, in un problema di classificazione, prima che venga applicata una softmax che li trasformi in probabilità. Nel mio caso però questo passaggio non è stato necessario, avendo sfruttato la flessibilità della classe CrossEntropyLoss disponibile nella libreria PyTorch. Essa è di base progettata per la classificazione multiclasse, ma può ovviamente essere utilizzata anche nel caso binario; inoltre, dalla documentazione si evince come possa calcolare correttamente la loss anche prendendo in input i logits originali, senza dover applicare manualmente una softmax o similari.

Un altro vantaggio fondamentale della classe CrossEntropyLoss è la possibilità, tramite il parametro weight, di aggiungere una regolarizzazione basata sullo sbilanciamento delle classi. Come detto precedentemente, nel mio caso il dataset era particolarmente sbilanciato, con un quantitativo di pixel nuvola che è circa lo 0.026% dei non nuvola. Un mio primo tentativo di addestrare la rete senza bilanciamento dei pesi si è rivelato fallimentare; infatti, il modello raggiungeva velocemente valori di loss estremamente bassi perché molte immagini erano prive di nuvole, e andava difatti ad overfittare verso queste ultime, ignorando praticamente sempre i pixel nuvola. Come questo si rifletteva sulle metriche di valutazione è trattato nella prossima sezione.

Dunque ho settato il parametro weight in modo tale da penalizzare molto meno un falso positivo (non nuvola classificato come nuvola) rispetto ad un falso negativo (nuvola classificata come non nuvola).

```
1 loss_fn = torch.nn.CrossEntropyLoss(weight =
    torch.tensor([0.026, 1]))
```

Listing 5.5. Cross entropy loss con bilanciamento

Questo ha portato a sensibili miglioramenti nei risultati finali.

5.4 Funzione di ottimizzazione

Le funzioni di ottimizzazione più utilizzate, sia per problemi di regressione che di classificazione, sono principalmente tre:

- Stochastic Gradient Descent (SGD)
- SGD con momentum
- Adaptive Momentum Estimation (Adam)

L'algoritmo principale su cui sono tutte basate e che quindi le accomuna è la discesa del gradiente (GD) [14]. In breve, si va a calcolare il gradiente (vettore delle derivate parziali prime) della funzione di loss e si modificano i pesi del modello secondo l'indicazione direzionale che esso fornisce, tentando

di diminuire sempre di più il valore di tale loss. Questo non garantisce di arrivare alla soluzione ottima quando si utilizza, come nel nostro caso, la binary cross entropy loss, in quanto la funzione che si cerca di minimizzare non è convessa, ovvero non c'è un solo minimo (globale), ma possono essere presenti vari minimi locali. In questo senso, la scelta dell'ottimizzatore diventa fondamentale dato che ognuno applica diverse variazioni all'algoritmo originale che facilitano l'apprendimento, andando ad evitare che il processo si blocchi in presenza di un minimo locale o, al contrario, che vada a divergere.

La mia scelta è ricaduta su Adam, per via dei seguenti vantaggi che essa offre [9]:

- Tasso di apprendimento adattivo
- Utilizzo dei momenti di primo e secondo ordine
- Correzione del bias
- Robustezza rispetto all'inizializzazione degli iperparametri

L'adattamento dinamico del tasso di apprendimento (learning rate) permette di calibrare efficacemente la velocità di apprendimento per diversi parametri o feature con scale diverse. Inoltre, tale adattamento rende Adam più tollerante a tassi di apprendimento iniziali errati, a differenza di altri algoritmi più semplici. Proprio a questo scopo, mantiene stime dei momenti di primo ordine (a partire dalla media dei gradienti) e dei momenti di secondo ordine (a partire dalla media dei gradienti al quadrato) per adattare il tasso di apprendimento considerando la variazione del gradiente nel tempo. Queste stime nelle prime fasi possono essere influenzate da una condizione iniziale in cui i momenti sono bassi; Adam adotta una correzione del bias per ovviare anche a questo problema. Per tutti questi motivi, e considerando anche quanti esperti del settore hanno provato l'efficacia di Adam per problemi di classificazione e regressione [9], la scelta è stata naturale.

Nonostante tutti questi meccanismi, rimane comunque importante settare un learning rate iniziale adeguato. I valori più utilizzati in letteratura sono 0.1, 0.01 e 0.001. Valori troppo piccoli possono portare ad un apprendimento più lento, valori troppo grandi ad un apprendimento instabile con rischio di divergenza. Dopo alcuni test ho scelto di impostare il valore iniziale a 0.01. Anche in questo caso non è necessario implementare da zero la funzione, in quanto PyTorch la fornisce built-in in uno dei suoi pacchetti.

```
1 opt = torch.optim.Adam(model.parameters(), lr=0.01)
```

Listing 5.6. Ottimizzatore Adam, label = adamcode

E' anche possibile modificare altri parametri opzionali, oltre al learning rate, che influiscono sui meccanismi di calcolo dei momenti descritti sopra. Tuttavia,

in questo caso, ho scelto di lasciare i valori di default, che si sono rivelati efficaci.

5.5 Metriche di valutazione

Le metriche di valutazione che ho esplorato sono due tra le più note, ovvero l'accuracy e l'F1 score:

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN} \quad F1 \text{ score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Dove $\text{Precision} = \frac{TP}{TP+FP}$ e $\text{Recall} = \frac{TP}{TP+FN}$. L'accuracy misura il rapporto tra il totale delle prediction corrette e il totale degli elementi esaminati, ed è una metrica molto generale che può essere più o meno indicativa a seconda di quanto è bilanciato il dataset. E' dunque facile intuire anche a priori come nel nostro caso possa non essere una scelta ottimale, in quanto il dataset è decisamente sbilanciato verso la classe non nuvola. L'F1 score invece esegue un calcolo tenendo conto di precision e recall, altre due metriche molto note. La recall misura il rapporto tra i positive individuati ed il totale effettivo del dataset. Consideriamo come classe "positive" i pixel nuvola, dunque la recall è tanto più alta quanto il nostro modello è in grado di individuare pixel nuvola, ed il valore aumenta anche se esso commette errori classificando pixel non nuvola come nuvola (false positive). La precision invece calcola il rapporto tra i true positive e i valori individuati come positivi dal modello. Questo valore scende se il modello considera nuvole anche dei pixel che non lo sono, ma non scende se sbaglia a classificare come non nuvola un pixel che lo è (false negative). L'F1 score rappresenta, in generale, un buon compromesso.

E' importante notare come nè nella formula della precision nè in quella della recall compaiano i true negative; infatti, l'F1 score non è influenzato dal numero di predizioni corrette della classe considerata "negative", nel nostro caso non nuvola. Questo ci permette di evitare che i tanti valori true negative indotti dalla grande maggioranza numerica di pixel non nuvola vadano ad influire sulla valutazione rendendola troppo ottimistica, come succedeva per l'accuracy.

Capitolo 6

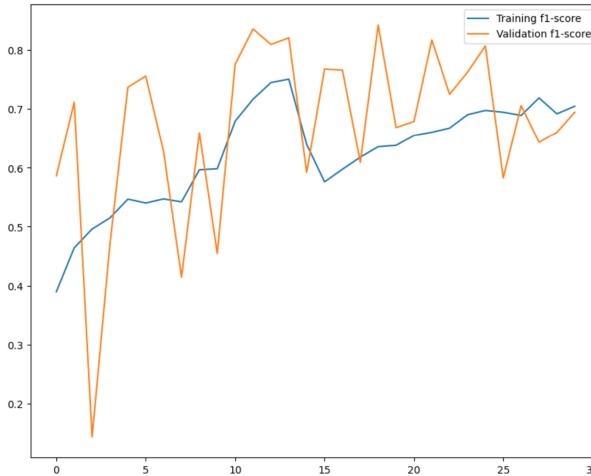
Risultati

6.1 Valutazione metrica dell'addestramento

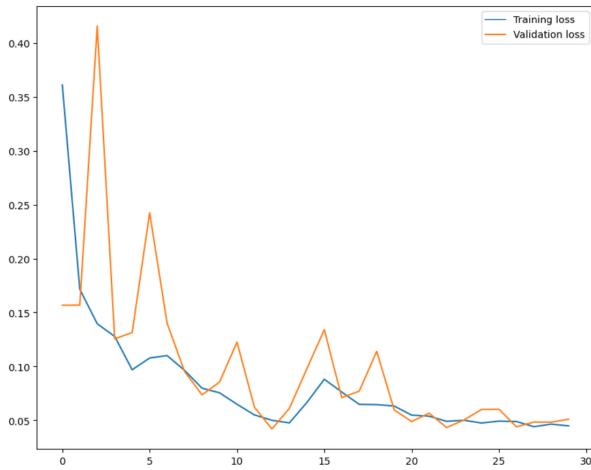
Ricapitolando, il modello è stato addestrato seguendo le seguenti specifiche:

- Training set: 1200 patch
- Validation set: 368 patch
- Epoche di addestramento: 30
- Dimensione del batch: 12
- Funzione di loss: Binary cross entropy loss
- Ottimizzatore: Adam con 0.01 di learning rate
- Bilanciamento dei pesi: tramite parametri della funzione di loss, assegnando un valore 37 volte più grande ai pixel nuvola

I primi esperimenti che ho fatto utilizzando come metrica l'accuracy si sono rivelati poco indicativi, in quanto essa raggiungeva a volte anche il 98% dopo poche epoche seppure, difatti, la rete prediceva tendenzialmente sempre non nuvola. Dunque, una volta scelto definitivamente di valutare il modello tramite F1-score, ho ottenuto un andamento evidenziato dal seguente grafico:

**Figura 6.1.** F1-score di training e validation a confronto

E' possibile notare come l'F1-score di training, sebbene con un grande drop intorno alla dodicesima epoca, tende a salire all'aumentare delle epoche e lascia intravedere un possibile miglioramento ipotizzando un training più lungo del modello. Il massimo F1-score raggiunto in fase di training sulle 30 epoche è del 75% circa. In fase di validation il modello oscilla maggiormente, ma raggiunge un massimo di circa 84%, molto migliore del corrispettivo valore di training. Questo è segno di un ottimo comportamento della rete, in quanto essa non overfittà sul dataset, ma anzi avrebbe teoricamente il potenziale per migliorare ancora di più. Tuttavia, vedendo come questo massimo venga poi raggiunto di nuovo ma mai superato dalla dodicesima epoca in poi, ho deciso di praticare early stopping. Andiamo ora a vedere il comportamento della loss al variare delle epoche:

**Figura 6.2.** Loss di training e validation a confronto

Sia in fase di training che di validation (sempre con più oscillazione nel secondo caso) la loss scende drasticamente fino alla dodicesima epoca circa,

per poi iniziare ad avere un andamento più piatto. In generale, il modello sembra avere il potenziale per fare risultati ancora migliori, ma evidentemente la poca quantità di dati utilizzati rende l'addestramento oltre un certo numero di epoche non particolarmente utile.

Verifichiamo come effettivamente questi numeri si riflettono sulla bontà della classificazione; di seguito analizzerò alcuni risultati di test, mostrando le immagini con le rispettive maschere e prediction, per cercare di capire quali pattern sono correttamente individuati dalla rete, e quali a volte confusi.

6.2 Valutazione visiva dei test: punti di forza

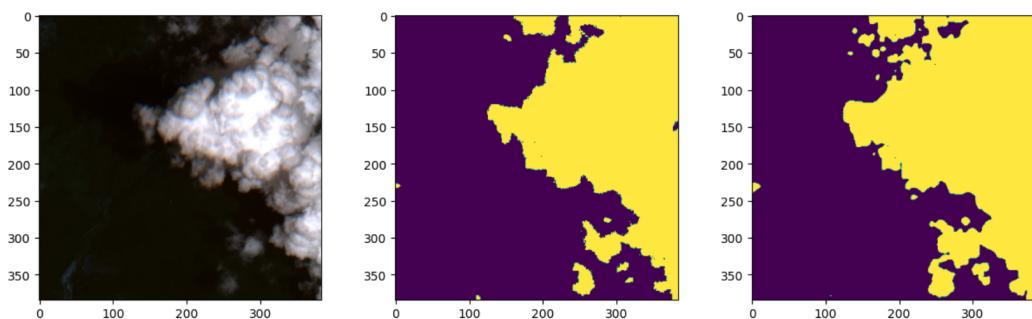


Figura 6.3. Risultato 1

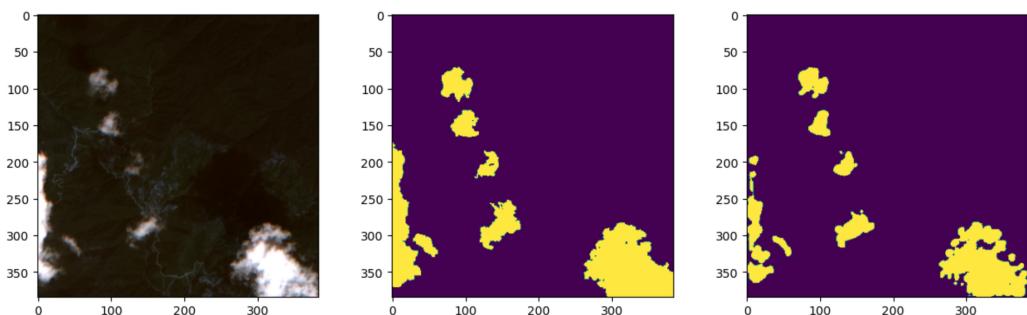


Figura 6.4. Risultato 2

La prima immagine è ad alto contenuto di pixel nuvola, con una superficie di tipo rurale e molto oscurata. La classificazione del modello è, almeno a livello visivo, molto buona, in quanto individua correttamente la forma della nuvola e fa un certo numero di falsi positivi che però non compromettono la bontà della maschera predittiva creata. Discorso simile per la seconda, sempre con sfondo rurale molto oscurato; in questo caso le nuvole sono varie e di minori dimensioni, ma il modello riesce comunque a individuarle efficacemente.

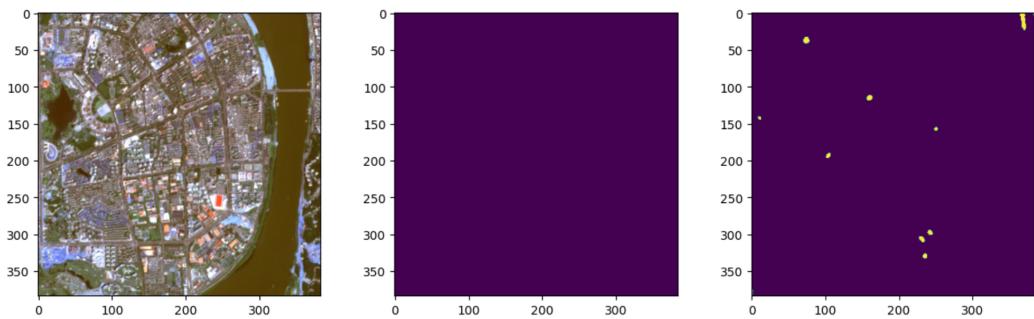


Figura 6.5. Risultato 3

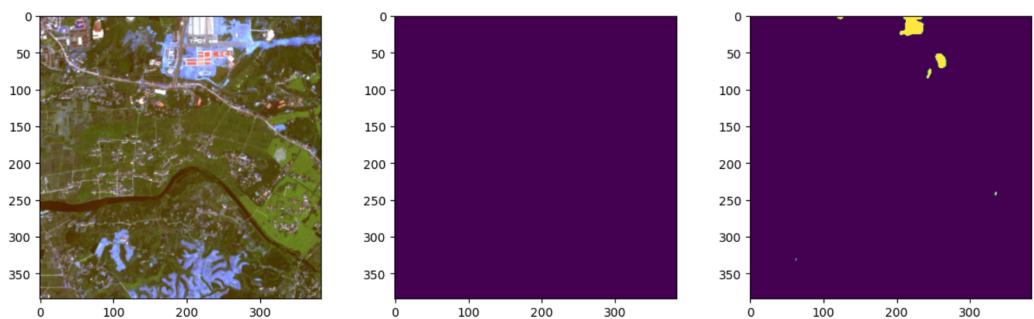
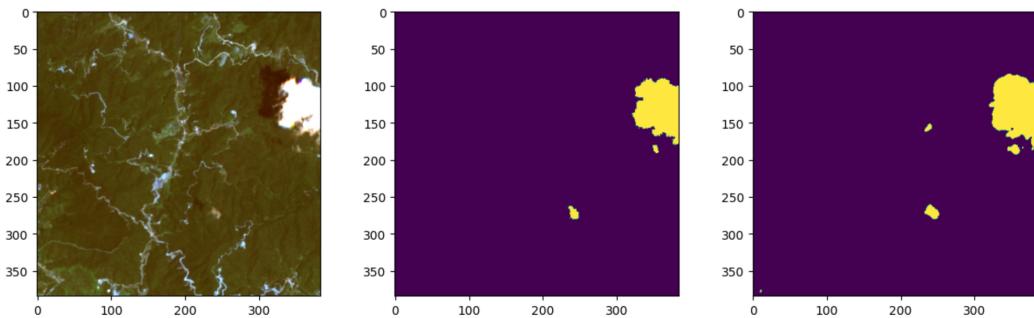
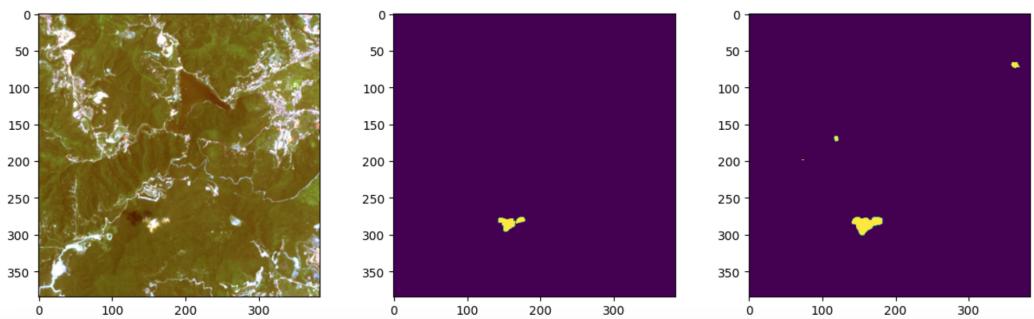


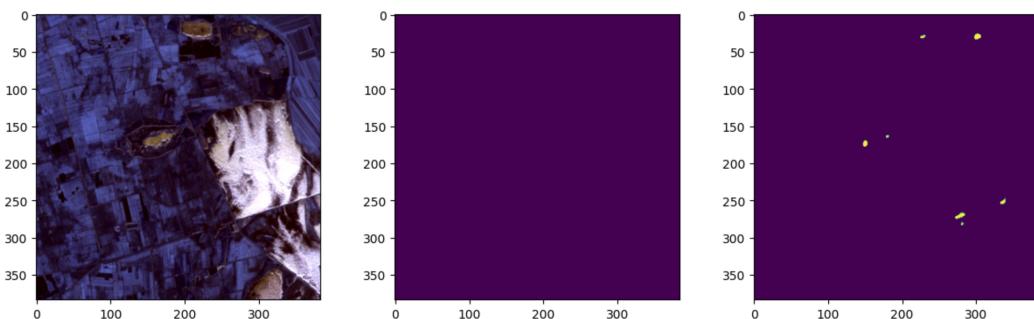
Figura 6.6. Risultato 4

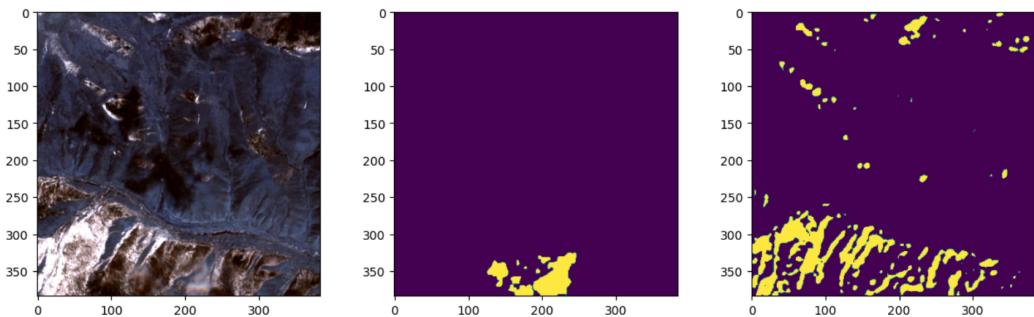
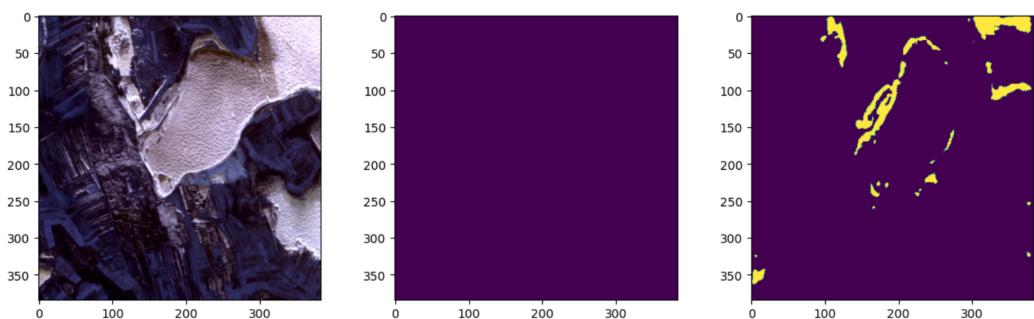
La terza immagine è priva di nuvole, e il tipo di copertura della superficie è quasi interamente urbanistico. Come possiamo notare, il modello classifica correttamente la stragrande maggior parte dei pixel. Nella quarta immagine, similmente, il modello classifica correttamente la quasi totalità dei pixel, ma possiamo evincere come la falsa nuvola individuata in alto a destra sia stata causa della confusione con un elemento urbanistico particolarmente chiaro. Questo tipo di errore è occorso sporadicamente durante i vari test, ma è comunque un difetto del modello da tener presente e che sicuramente potrebbe essere migliorato, ad esempio, utilizzando più bande oltre a quelle a 10 metri. Nel complesso, però, possiamo dire che la rete è capace di distinguere correttamente edifici ed altri elementi urbanistici dalle nuvole.

**Figura 6.7.** Risultato 5**Figura 6.8.** Risultato 6

La quinta e la sesta immagine sono accumulate dalla presenza di fiumi e dal quantitativo basso (ma non nullo) di nuvole. Le fonti d'acqua sono note come possibili cause di problematiche per i modelli di cloud detection, eppure da questi due esempi si evince come la rete non confonda affatto le varie insenature, e riesca inoltre ad individuare con buona precisione le nuvole nonostante in questo caso si presentino in "blocchi" molto più piccoli, specialmente nella sesta immagine.

6.3 Valutazione visiva dei test: punti di debolezza

**Figura 6.9.** Risultato 7

**Figura 6.10.** Risultato 8**Figura 6.11.** Risultato 9

Ho condotto una serie di test su immagini contenenti un alto quantitativo di zone innevate. Come evidenziato durante la descrizione del problema, distinguere neve e ghiaccio dalle nuvole è uno dei problemi più complessi della cloud detection, e in letteratura vari studi dimostrano come sia d'aiuto l'utilizzo di altre bande oltre a quelle a 10 metri per effettuare correttamente questa differenziazione. Nel nostro caso, la rete fa uso solo delle bande RGB + NIR e quindi commette più errori di questo genere. Nella prima immagine di questo tipo riportata il tetto innevato è correttamente classificato come non nuvola nella sua interezza, ma nelle due successive notiamo come il modello confonda molti pixel classificandoli erroneamente come nuvole. Questo è probabilmente dovuto non solo al mancato utilizzo delle bande a 20 e 60 metri, ma anche alla poca quantità di esempi di questo genere presenti nella porzione di dataset che ho utilizzato. Un addestramento condotto andando a selezionare più immagini di questa tipologia potrebbe apportare dei miglioramenti sostanziali alla rete.

Tuttavia, sebbene i risultati ottenuti non siano considerabili allo stesso livello di quelli raggiunti attraverso il modello CD-FM3SF, è bene anche tener presente la differenza di dimensione della mia U-Net.

```
1 def count_parameters(model):
2     return sum(p.numel() for p in model.parameters()
3                if p.requires_grad)
4 print(count_parameters(model))
5 #risultato: 538934 parametri
```

Listing 6.1. Calcolo del numero di parametri della mia U-Net

Essa infatti presenta poco più di mezzo milione di parametri, come mostrato dal calcolo eseguito in figura, che si contrappongono al milione di parametri di CD-FM3SF (esso stesso considerato un modello leggero [4]) e a quelli estremamente maggiori di architetture antecedenti come MSCFF (circa 2 milioni) e RS-Net (quasi 8 milioni). Aver ottenuto un modello con circa 84% di f1-score sul validation set ed una buona capacità di distinguere le categorie di copertura superficiale descritte dalle nuvole, fatta eccezione per la neve, è quindi considerabile un buon risultato complessivo.

Capitolo 7

Conclusioni

Il campo della cloud detection è un ambito di ricerca più che mai attivo, e nonostante l'attuale stato dell'arte abbia raggiunto ottimi risultati si è ancora alla ricerca di possibili miglioramenti. Per quanto riguarda la mia rete, lo studio condotto ha dimostrato la possibilità di ottenere buoni risultati utilizzando una U-Net dai caratteri standard ed una quantità ristretta di dati, imposta dai limiti dell'hardware a disposizione. Coerentemente, i difetti evidenziati dalla fase di test dimostrano come l'aggiunta di operazioni più complesse sui dati (come quelle effettuate in CD-FM3SF) e il training prolungato su una mole di dati maggiore di quella utilizzata possano fare una concreta differenza.

Possibili ulteriori sviluppi del lavoro sarebbero senza dubbio un addestramento più corposo della rete su un'architettura adeguata, l'esplorazione di altri parametri già disponibili nei layer utilizzati (come ad esempio la dilation per i layer convoluzionali) e l'implementazione di un metodo che permetta anche nel mio caso di utilizzare tutte le bande spettrali disponibili per ottenere migliori performance.

Bibliografia

- [1] A. G. Howard et al. “MobileNets: Efficient convolutional neural networks for mobile vision applications”. In: *arXiv:1704.04861* (2017).
- [2] D. Chai et al. “Cloud detection for landsat imagery by combining the random forest and superpixels extracted via energy-driven sampling segmentation approaches”. In: *Remote Sens. Environ.* 248 (2020).
- [3] Derek Doran et al. “Explainable Deep Learning: A Field Guide for the Uninitiated”. In: *arXiv:2004.14545* (2021).
- [4] Jun Li et al. “A Lightweight Deep Learning-Based Cloud Detection Method for Sentinel-2A Imagery Fusing Multiscale Spectral and Spatial Features”. In: *IEEE transactions on geoscience and remote sensing* 66 (2022).
- [5] Y. Zhan et al. “Distinguishing cloud and snow in satellite images via deep convolutional network”. In: *IEEE Geosci. Remote Sens. Lett.* 14 (2017).
- [6] Yuhan Bai. “RELU-Function and Derived Function Review”. In: *SHS Web of Conferences* 144 (2022).
- [7] Vincent Dumoulin e Francesco Visin. “A guide to convolution arithmetic for deep learning”. In: *arXiv:1603.07285* (2018).
- [8] S. Ioffe e C. Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *Proc. Int. Conf. Mach. Learn.* (2015).
- [9] Diederik P. Kingma e Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *arXiv:1412.6980* (2014).
- [10] *Open Source Geospatial Foundation repository*. URL: <https://github.com/OSGeo>.
- [11] *Overview of Colaboratory Features*. URL: https://colab.research.google.com/notebooks/basic_features_overview.ipynb.
- [12] *PyTorch documentation*. URL: <https://pytorch.org/docs/stable/index.html>.
- [13] *Repository del progetto*. URL: <https://github.com/TheEmotionalProgrammer/CloudDetectionThesis>.
- [14] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *arXiv:1609.04747* (2016).
- [15] *Sentinel-2 mission guide*. URL: <https://sentinel.esa.int/web/sentinel/missions/sentinel-2>.

- [16] *TensorBoard Overview*. URL: <https://www.tensorflow.org/tensorboard>.
- [17] *TensorFlow guide*. URL: <https://www.tensorflow.org/guide?hl=en>.
- [18] *The Ultimate Guide to Convolutional Neural Networks (CNN)*. URL: <https://www.superdatascience.com/blogs/the-ultimate-guide-to-convolutional-neural-networks-cnn>.
- [19] *U-Net Explained: Understanding its Image Segmentation Architecture*. URL: <https://towardsdatascience.com/u-net-explained-understanding-its-image-segmentation-architecture-56e4842e313a>.
- [20] Y. Zhang. “Calculation of radiative fluxes from the surface to top of atmosphere based on ISCCP and other global data sets: Refinements of the radiative transfer model and the input data”. In: *J. Geophys. Res.* vol. 109 (2004).