ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA

Sede di Forlì

Corso di Laurea in

INGEGNERIA AEROSPAZIALE

Classe L9

ELABORATO FINALE DI LAUREA

In Satelliti e Missioni Spaziali

Implementazione di reti neurali per missioni di telerilevamento satellitare da immagini iperspettrali

CANDIDATO:

Achille Ballabeni

RELATORE:

Prof. Alfredo Locarini

CORRELATORE:

Dott. Alessandro Lotti

Anno Accademico 2021/2022

Abstract

L'utilizzo di reti neurali, applicate a immagini iperspettrali, direttamente a bordo di un satellite, permetterebbe una stima tempestiva ed aggiornata di alcuni parametri del suolo, necessari per ottimizzare il processo di fertilizzazione in agricoltura. Questo elaborato confronta due modelli derivati dalle reti EfficientNet-Lite0 ed EdgeNeXt per la stima del valore di Ph del terreno e delle concentrazioni di Potassio (K), Pentossido di Fosforo (P_2O_5) e Magnesio (Mg) da immagini iperspettrali raffiguranti campi agricoli. Sono stati inoltre testati due metodi di riduzione delle bande: l'Analisi delle Componenti Principali (PCA) e un algoritmo di selezione basato sull'Orthogonal Subspace Projection (OSP). Lo scopo è di ridurre le dimensioni delle immagini al fine di limitare le risorse necessarie all'inferenza delle reti, pur preservandone l'accuratezza. L'esecuzione in tempo reale (23.6 fps) della migliore soluzione ottenuta sul sistema *embedded* Dev Board Mini ne dimostra l'applicabilità a bordo di nanosatelliti.

Indice

Elenco delle figure

Elenco delle tabelle

Acronimi

AI Intelligenza Artificiale.

CNN Reti Neurali Convoluzionali.

DL Deep Learning.

EO Earth Observation.

ESA Agenzia Spaziale Europea.

FIPPI Fast Iterative Pixel Purity Index.

ML Machine Learning.

OSP Orthogonal Subspace Projection.

PCA Analisi delle Componenti Principali.

TPU Tensor Processing Unit.

Introduzione

Questo elaborato finale di tesi è stato realizzato in continuità con l'attività di tirocinio curricolare, svolta presso il Laboratorio di Microsatelliti e Microsistemi Spaziali, facente parte del Centro Interdipartimentale di Ricerca Industriale CIRI Aerospaziale - Aerospace. Le attività di tirocinio sono state finalizzate allo sviluppo di algoritmi volti a partecipare alla competizione "Seeing Beyond the Visible"[1] organizzata da KP Labs, Agenzia Spaziale Europea (ESA) e QZ Solutions, nell'ambito della "IEEE International Conference on Image Processing (ICIP) 2022". L'iniziativa promuove attività di ricerca volte ad equipaggiare la prossima generazione di satelliti di osservazione della Terra con tecniche di intelligenza artificiale al fine di ottimizzare i processi di fertilizzazione in agricoltura. Ciò consentirebbe di ridurre i costi di produzione e di minimizzare l'impatto ambientale limitando il ricorso ai fertilizzanti.

L'attività di tirocinio ha portato all'elaborazione di una soluzione basata sulla rete neurale EfficientNet-Lite0 [2]. Gli sviluppi successivi apportati per l'elaborato finale hanno riguardato il confronto con la rete EdgeNeXt [3], l'implementazione di due algoritmi per la riduzione delle bande delle immagini iperspettrali e infine il deployment sul sistema embedded Coral Dev Board Mini.

Il primo confronto tra le reti è servito per provare una nuova architettura, di recente pubblicazione, non utilizzata nel corso della competizione. Il tentativo è stato fatto per valutare se i blocchi di attenzione di cui è dotata la EdgeNeXt, tipici dei Vision Transformer, potessero contribuire positivamente alla soluzione del problema.

Tra le due reti, la migliore è stata testata in abbinamento ai due algoritmi di riduzione delle bande, che in questo modo sono stati confrontati. La migliore soluzione complessiva è stata implementata sul computer a scheda singola Dev Board Mini.

L'implementazione e il confronto tra l'Analisi delle Componenti Principali (PCA) e

l'algoritmo di selezione delle bande basato sull'Orthogonal Subspace Projection (OSP) proposto in [4], disponibile come funzione built-in di MATLAB [5], sono motivati dalla necessità di limitare il costo computazionale in fase di inferenza in modo da dimostrarne l'implementabilità su un computer a scheda singola di possibile impiego in missioni nanosatellitari.

I codici sviluppati nell'ambito del lavoro di tesi sono stati scritti prevalentemente in linguaggio Python all'interno di vari Jupyter Notebook, utilizzando l'ambiente di sviluppo web Google Colaboratory. Le reti sono state implementate tramite l'API Python di TensorFlow Keras. Gli script principali sono stati allegati alla fine dell'elaborato mentre i Notebook completi sono disponibili al seguente link: `https://github.com/achille-ballabeni/Tesi_Achille_Ballabeni`.

L'elaborato è strutturato come segue:

- I Capitoli 1 e 2, contengono nozioni teoriche sulle reti neurali, sulle immagini iperspettrali e sui metodi di riduzione delle bande utilizzati.

- Il Capitolo 3 riassume brevemente l'attività di tirocinio e illustra il confronto tra la rete presentata alla competizione e quella derivata dalla EdgeNeXt.

- Il Capitolo 4 illustra l'implementazione e il confronto tra i due metodi di riduzione delle bande.

- Il Capitolo 5, descrive l'implementazione della soluzione migliore sul dispositivo embedded e i risultati ottenuti in fase di inferenza.

- Il Capitolo 6, contiene le conclusioni del lavoro svolto e illustra possibili direzioni per sviluppi futuri.

1 | Fondamenti di Reti Neurali

1.1 Intelligenza Artificiale, Machine Learning e Deep Learning

Prima di introdurre il concetto di rete neurale e descrivere la classe di rete usata per affrontare il problema presentato in questo elaborato, è opportuno fare chiarezza sui termini che riguardano questo ambito dell'informatica.

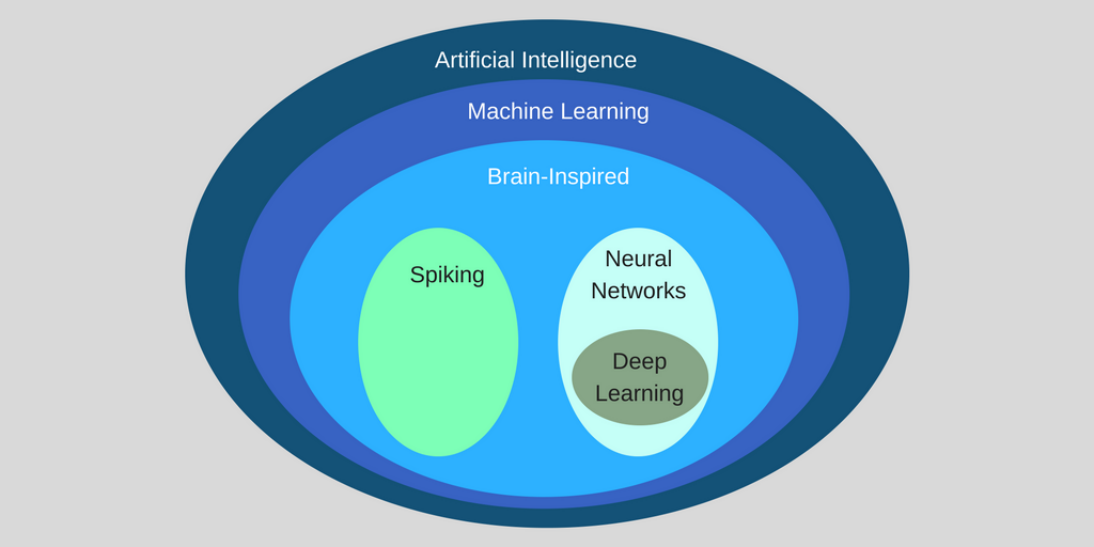

Figura 1.1: Sottogruppi dell'AI [6].

Con Intelligenza Artificiale (AI) si intende la vasta branca dell'informatica che studia lo sviluppo di macchine e programmi in grado di apprendere e svolgere compiti, indipendentemente, per i quali viene solitamente considerata necessaria la presenza di "intelligenza" [7].

Il Machine Learning (ML) è un sottoinsieme dell'AI in cui le macchine "apprendono" a svolgere un compito specifico autonomamente, da un set di dati, *labelled* o *unlabelled*, o interagendo con l'ambiente esterno. In questo caso, per l'apprendimento, non è necessaria una programmazione esplicita [6].

L'ultimo sottogruppo dell'AI è il Deep Learning (DL). Il DL è in realtà a sua volta un sottoinsieme di una classe di algoritmi di ML chiamati reti neurali. Le reti neurali hanno una struttura ispirata a quella del cervello umano, costituita da tante unità computazionali semplici (neuroni) che interagiscono tra di loro e realizzano una funzione complessa. Se la struttura è articolata su molti livelli, rientrano nella categoria del DL. In tal caso si parla di Deep Neural Networks [6].

1.2 Reti Neurali

Negli ultimi anni, con l'avvento di dispositivi più potenti da un punto di vista computazionale, l'accesso a dataset più estesi e il miglioramento degli algoritmi, l'AI è diventata sempre più centrale nel processo di digitalizzazione della società [8]. Le ricerche nel campo della AI hanno coinvolto anche il campo spaziale, in particolare per le missioni di Earth Observation (EO), in cui i satelliti registrano enormi quantità di dati. L'introduzione di algoritmi di DL basati su reti neurali, impiegati come strumenti di analisi, permetterebbe una maggiore velocità di elaborazione dei dati satellitari. Inoltre, grazie all'*onboard processing*, le informazioni rilevanti potrebbero essere estratte dai dati direttamente a bordo, limitando sensibilmente la mole di informazioni che dovrebbe altrimenti essere trasferita al segmento di terra [1]. Ciò si tradurrebbe in una riduzione dei tempi di impiego delle stazioni di terra e di conseguenza del costo delle missioni.

Tra le tecniche utilizzate nell'ambito del ML ci sono le reti neurali. Le reti neurali artificiali consistono in algoritmi in grado di associare ai dati in input un determinato output, a valle di un processo di addestramento. Si compongono di nodi, o neuroni, organizzati in livelli. Ogni nodo esegue una semplice operazione non lineare sui dati ricevuti dal livello precedente, eseguita da una funzione di attivazione. La combinazione di un numero sufficientemente elevato di nodi e livelli consente di approssimare funzioni molto complesse. Qualora ogni nodo riceva in ingresso l'output di tutti i nodi del livello

precedente, si definisce la rete "completamente connessa", contrariamente si definisce il modello "parzialmente connesso" (Figura 1.2).

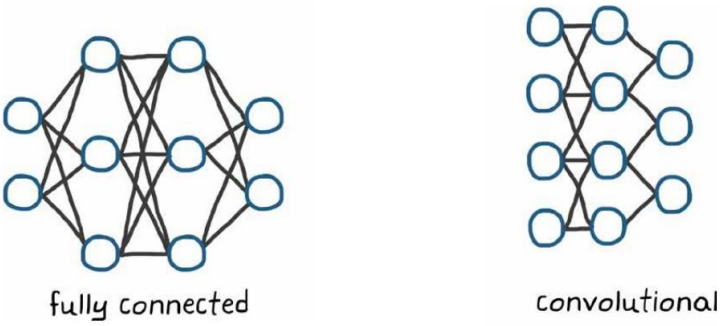

Figura 1.2: Schematizzazione di una rete completamente connessa (sinistra) e di un modello convoluzionale (destra) [9].

Le tecniche di apprendimento delle reti neurali si possono dividere in tre categorie [8]:

- Apprendimento supervisionato quando i dataset utilizzati sono *labelled dataset*, ovvero se per ogni campione dato in input alla rete, viene fornito anche il rispettivo output che questa deve imparare a predire.

- Apprendimento non supervisionato se gli input vengono forniti non annotati, ovvero senza i valori di output corrispondenti. La rete deve quindi imparare in modo completamente autonomo a inferire l'output a partire da caratteristiche comuni.

- Apprendimento semi-supervisionato nel caso in cui i dataset siano costituiti da un piccolo numero di campioni annotati e un numero maggiore di esempi non annotati.

Un aspetto chiave nella valutazione delle performance di una rete neurale è la sua capacità di generalizzazione. Due fenomeni che influenzano negativamente la generalizzazione di una rete sono l'*underfitting* e l'*overfitting* (Figura 1.3). Il primo si verifica quando il modello è troppo semplice per il compito richiesto oppure non è stato allenato a sufficienza. In questo caso le predizioni della rete si discostano dai valori reali in quanto, durante il suo addestramento, non è stata in grado di stabilire una valida relazione tra input e output. L'overfitting avviene quando la rete si adatta perfettamente

ai campioni di training, "imparandoli a memoria", senza poi essere in grado compiere predizioni corrette su dati mai visiti. Questa seconda problematica si può verificare quando la rete è fin troppo complessa oppure se il dataset è molto limitato.

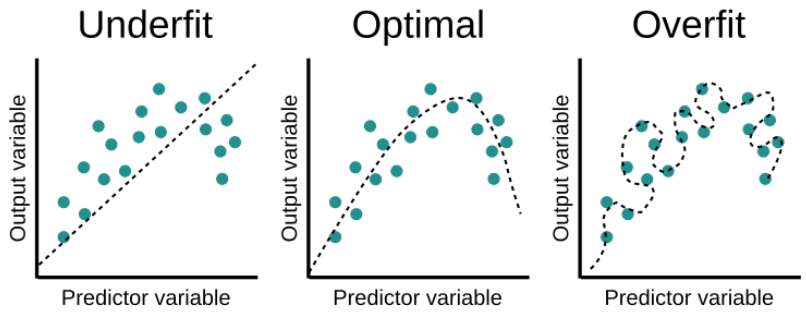

Figura 1.3: Modello in stato di underfitting (sinistra), overfitting (destra) e ottimale (centro) [10].

1.3 Reti Neurali Convoluzionali

Le Reti Neurali Convoluzionali (CNN) sono modelli parzialmente connessi e risultano perciò particolarmente adatte per l'elaborazione delle immagini. Queste infatti contengono un numero molto elevato di dati (pixel) che si tradurrebbe in un costo computazionale notevole qualora venissero processate da modelli completamente connessi.

I livelli, o *layer*, fondanti delle CNN sono tre:

- Layer convoluzionali
- Layer di pooling
- Layer completamente connessi

Altri due livelli frequentemente utilizzati, che verranno nominati nel terzo capitolo, sono i layer di *Batch Normalization* e di *Dropout*.

1.3.1 Layer convoluzionali

Alla base dei layer convoluzionali c'è l'operazione di convoluzione, di seguito descritta nel caso 2D. Ogni livello convoluzionale è costituito da k *kernel*: matrici di dimensione

$n \times n \times c$, dove n è la dimensione spaziale del kernel mentre c rappresenta il numero di canali dell'immagine (o più in generale del tensore di input). L'inseme dei k kernel di un livello costituisce il filtro [11]. I kernel contengo i valori dei pesi che devono essere aggiornati durante il processo di addestramento della rete per renderla il più precisa possibile nello svolgimento del compito assegnato. Ciascun kernel scorre sul tensore con un avanzamento detto *stride*. Durante lo scorrimento, viene calcolato il prodotto elemento per elemento tra l'input del livello e il kernel, per quella determinata posizione di avanzamento. I valori della matrice risultante vengono sommanti e costituiscono un nuovo elemento dell'output. Siccome la convoluzione di ciascun kernel riduce il numero di canali da c a 1, il numero di canali dell'output dipende unicamente dal numero di kernel utilizzati k (Figura 1.4).

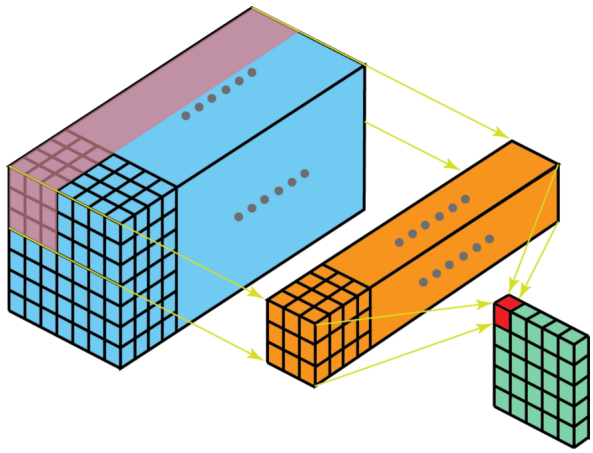

Figura 1.4: Convoluzione bidimensionale di un kernel sul tensore di input e feature map corrispondente in output [11].

La convoluzione di un filtro porta alla formazione di un nuovo tensore detto *feature map*. Ogni kernel assume infatti il ruolo di estrattore di feature e, durante l'addestramento, si "specializza" nell'evidenziare un certo tipo di informazione [12], codificata nel tensore di output.

1.3.2 Layer di pooling

I livelli di pooling compiono un'operazione di *downsampling* delle feature map sul piano spaziale, volta a ridurre il numero dei successivi parametri addestrabili [12]. Sono costituiti da filtri, privi di parametri da allenare, che scorrono sulle feature maps in

modo analogo ai kernel dei layer convoluzionali, applicando l'operazione di pooling alla regione corrispondente.

Esistono filtri *max pooling* oppure *average pooling*. Come suggeriscono i nomi, il primo riporta nella nuova feature map solo i valori massimi delle regioni sottoposte al pooling, il secondo i valori medi. L'applicazione di un filtro di pooling può avvenire anche globalmente, grazie ai livelli *global pooling*, esistenti in entrambe le versioni. Per il pooling globale, l'operazione viene applicata all'intera feature map: una feature map di dimensione $h \times w$ viene ridotta ad una matrice 1×1, lasciando però inalterata la profondità del tensore [12].

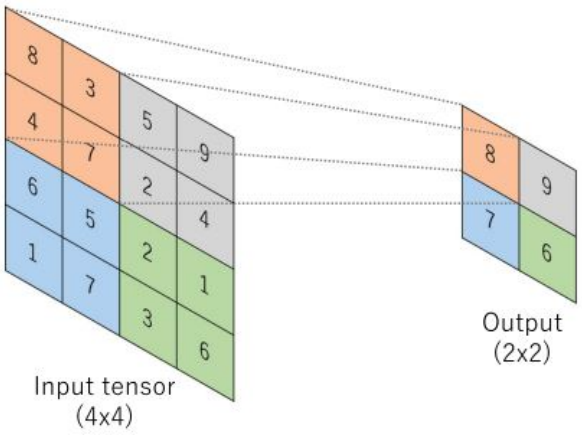

Figura 1.5: Max pooling applicato a una singola feature map con dimensione del filtro 2×2 [12].

1.3.3 Layer completamente connessi

Nonostante le CNN siano dei modelli parzialmente connessi, spesso fanno uso di almeno un livello completamente connesso, anche chiamato *dense layer*, posizionato nella parte finale della rete. Le feature map ottenute vengono trasformate in un vettore monodimensionale (operazione di *flattening*) che è collegato a uno o più livelli completamente connessi. Il compito di questi livelli è creare una mappa finale che porti dalle informazioni estratte nei livelli precedenti, al tipo di output desiderato [12].

L'output risultante dall'ultimo livello completamente connesso coincide con l'output della rete. Questo può essere un vettore contenete le probabilità di appartenenza a delle

classi, nel caso di problemi di classificazione, oppure il vettore dei parametri stimati, per problemi di regressione.

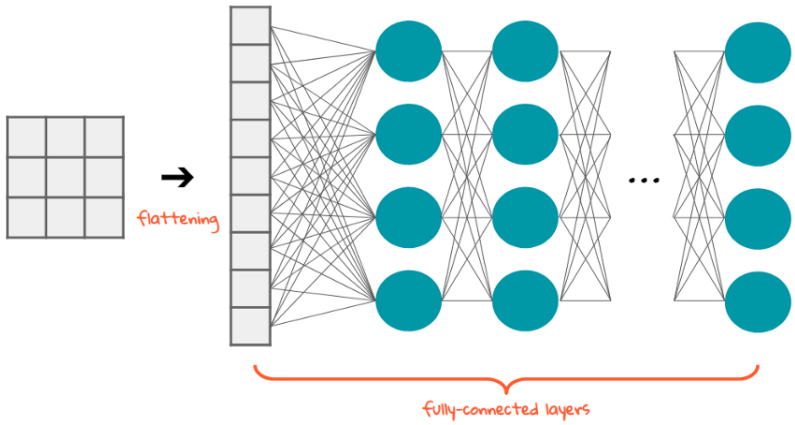

Figura 1.6: Operazione di flattening e livelli completamente connessi [13].

1.3.4 Layer di dropout e batch normalization

I livelli di dropout sono utilizzati per migliorare la generalizzazione della rete e cercare di prevenire l'overfitting. Un layer di dropout elimina, randomicamente, alcune delle connessioni tra i nodi dei due livelli tra cui è interposto, impostando i valori degli input corrispondenti a 0. Un layer di dropout è caratterizzato dal *dropout rate*, ovvero la frazione delle connessioni da eliminare.

Il compito di un layer batch normalization è la normalizzazione dei dati di input dei livelli per rendere più stabile il processo di addestramento. Gli input vengono standardizzati affinché la loro media sia vicina a zero e la deviazione standard sia all'incirca unitaria. Questa normalizzazione è da riferirsi a un intero *batch* del processo di training, ovvero l'insieme dei campioni che la rete deve processare prima di aggiornare i propri pesi.

2 | Immagini iperspettrali e algoritmi di riduzione delle bande

2.1 Immagini iperspettrali

Le immagini iperspettrali consentono di misurare le caratteristiche spaziali e spettrali di un oggetto grazie alle informazioni raccolte da un sensore iperspettrale, un dispositivo in grado di misurare l'intensità del flusso riflesso che lo investe [14]. I dati raccolti dal sensore sono organizzati in una matrice tridimensionale di dimensioni $M \times N \times C$, che prende il nome di *data cube* o *hypercube*. M e N sono le dimensioni spaziali dell'immagine, mentre C è la dimensione spettrale. Quest'ultima indica il numero di bande per cui il sensore è in grado di acquisire informazioni. Un'immagine iperspettrale può allora essere pensata come un insieme di immagini bidimensionali monocromatiche, catturate alle diverse lunghezze d'onda rilevate dal sensore (Figura 2.1).

La visualizzazione del soggetto catturato dall'immagine iperspettrale può essere facilitata da una rappresentazione bidimensionale. È possibile applicare una visualizzazione a schemi di colore che faccia uso di più bande, oppure utilizzare una visualizzazione monocromatica, selezionando un'unica banda [14] (Figura 2.1).

Ciascun pixel del data cube è un vettore contenente i valori di riflettanza di ciascuna banda nella posizione (x, y) (Figura 2.1). Il vettore viene chiamato *pixel spectrum* e rappresenta la firma spettrale della porzione del soggetto corrispondente al pixel in (x, y) [14]. I valori contenuti nel pixel spectrum sono estremamente utili nell'analisi delle immagini iperspettrali in quanto permettono di studiare lo spettro di riflessione del soggetto. Siccome i materiali presentano delle proprie firme spettrali, a partire da ciascun pixel spectrum, o dall'immagine spettrale nella sua completezza, si può risalire

ad alcune informazioni sulle caratteristiche fisiche del soggetto dell'immagine.

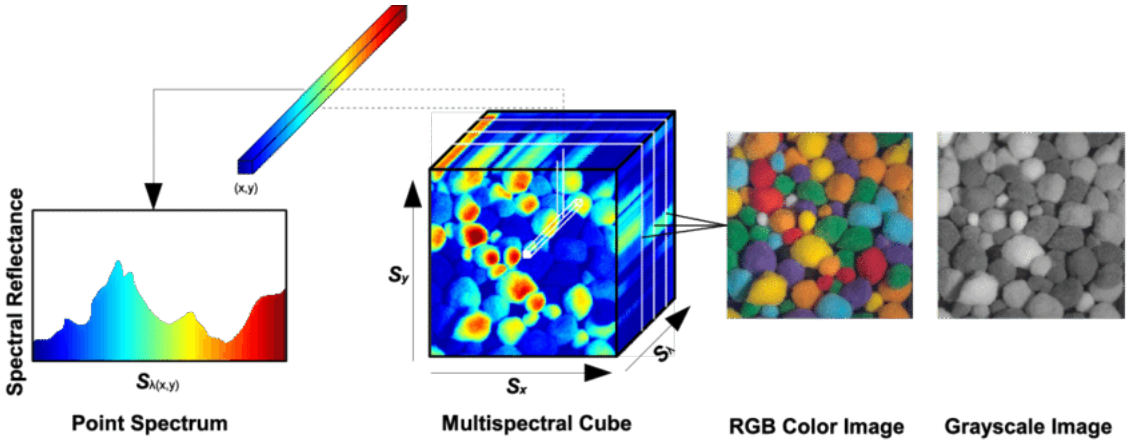

Figura 2.1: Data cube in pseudocolore al centro, pixel spectrum a sinistra e due immagini estratte dall'hypercube a destra [15].

L'esempio più comune di utilizzo di immagini iperspettrali è quello relativo alla classificazione dei pixel. A partire dai *pixel spectra* è possibile identificare il materiale, o l'oggetto, che quel determinato pixel rappresenta nel soggetto catturato (Figura 2.2). L'associazione dello spettro di riflessione contenuto in un pixel con lo spettro del materiale di riferimento può essere fatta con algoritmi di *spectral matching* oppure algoritmi di intelligenza artificiale.

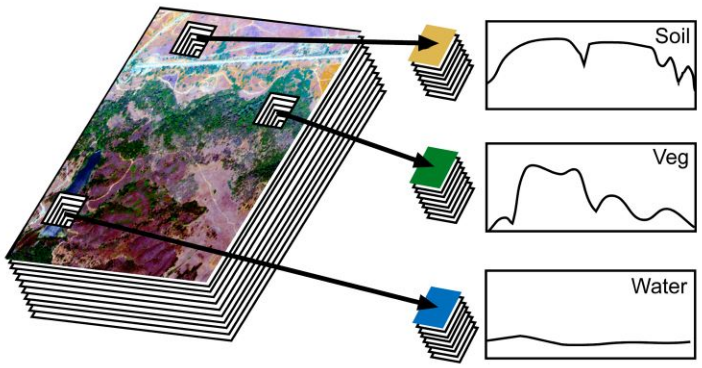

Figura 2.2: Spettro di riflessione di ciascun pixel e associazione con materiali [16].

Oltre che ai più studiati problemi di classificazione, le immagini iperspettrali possono essere utili persino per stimare le concentrazioni di svariati parametri del suolo, come nel caso del problema affrontato in questo elaborato.

2.2 Algoritmi di riduzione delle bande

Se da una parte le immagini iperspettrali rappresentano un valido strumento di analisi per missioni di EO, la loro capacità di registrare una grande quantità di informazioni può essere un ostacolo dal punto di vista computazionale. Questo è particolarmente vero nel caso in cui l'elaborazione dei dati avvenga direttamente a bordo di un nanosatellite: i requisiti sul consumo di potenza pongono limiti sulla scelta dei processori utilizzati e di conseguenza sulle risorse computazionali disponibili. Inoltre, anche quando le risorse sono sufficienti per il processamento dei dati a bordo, è comunque opportuno cercare di limitare il più possibile il costo computazionale dei processi, in modo tale da ridurre la potenza assorbita dal processore e rendere più rapido il tempo di esecuzione dei programmi.

Il processo di minimizzazione delle risorse richieste passa sia per la scelta di algoritmi sufficientemente leggeri, sia per un eventuale preselezione delle informazioni che conviene effettivamente elaborare. Nel caso delle immagini iperspettrali, gli approcci per una compressione delle informazioni contenute in ciascun data cube prevedono principalmente metodi che agiscono nella direzione spettrale. Ridurre il numero delle bande delle immagini significa fornire agli algoritmi di elaborazione un numero minore di dati, con conseguente risparmio sul costo computazionale.

In questo elaborato sono stati testati due metodi di riduzione delle bande che agiscono seguendo due principi diversi: la PCA e il metodo di selezione delle bande proposto in [4], disponibile come funzione *built-in* di MATLAB. La PCA agisce creando un nuovo spazio in cui proiettare i dati e selezionando le bande rilevanti in questa nuova proiezione. L'algoritmo proposto in [4] permette di selezionare le bande più informative direttamente tra quelle dell'immagine originale.

2.2.1 Analisi delle Componenti Principali

La PCA consiste in una trasformazione lineare che proietta i dati in un nuovo sistema di riferimento, in cui gli assi sono orientati secondo le direzioni di massima varianza. Ogni asse rappresenta una variabile del nuovo spazio, ottenuta come combinazione lineare di quelle originariamente utilizzate per descrivere i dati. In questo modo è possibile eliminare la correlazione spesso presente tra le variabili iniziali e creare un

nuovo spazio di variabili ordinate per rilevanza, ovvero per la loro capacità di esprimere un certo grado di varianza nei dati. Siccome le nuove variabili sono ottenute come combinazioni lineari di quelle iniziali, è possibile mantenere, tra di esse, solamente quelle che esprimono la maggiore quantità di varianza e, al tempo stesso, preservare buona parte delle informazioni contenute nei dati.

Nel caso di immagini iperspettrali, lo spazio delle variabili è costituito dalle bande registrate dall'immagine. Ogni pixel è un'osservazione che assume valori numerici per ogni variabile (o banda). L'insieme dei pixel costituiscono il set di dati, cioè l'immagine, sui cui applicare la PCA. La PCA proietta le immagini, secondo il principio descritto nel paragrafo precedente, in un nuovo spazio di bande, ordinate per importanza nella descrizione delle informazioni dell'immagine (Figura 2.3). È importante notare che le nuove bande che descrivono la proiezione dell'immagine iperspettrale sono prive di significato fisico.

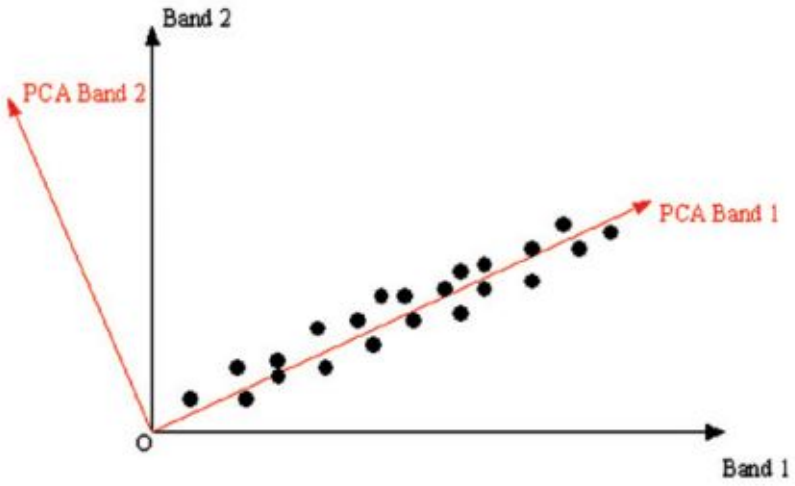

Figura 2.3: Interpretazione geometrica della PCA applicata a un'immagine con due bande: ciascun punto rappresenta un pixel [17]. Si percepisce visivamente come il set di dati possa essere descritto quasi unicamente utilizzando la proiezione sull'asse *PCA Band 1*.

L'efficacia dell'applicazione della PCA alle immagini iperspettrali si basa sul presupposto che le bande vicine siano strettamente correlate e forniscano praticamente lo stesso tipo di informazione [17]. Ci si aspetta, quindi, che a valle dell'applicazione della

PCA, solamente un piccolo quantitativo delle nuove bande contribuisca a descrivere la maggior parte delle informazioni contenute dall'immagine.

L'applicazione della PCA a un set di dati passa per la decomposizione agli autovalori della matrice di covarianza dei dati. La trattazione matematica che viene utilizzata nei paragrafi di seguito, per illustrare il processo di applicazione della PCA, è stata presa da [17].

Supponendo di voler applicare la PCA a un immagine iperspettrale di dimensioni $m \times n \times c$, dove m e n sono le dimensioni spaziali e c il numero di bande, si procede come segue. Per prima cosa, ciascun pixel dell'immagine deve essere pensato come una singola osservazione, le cui variabili descrittive sono le bande rappresentate dall'immagine. I pixel dell'immagine sono dei vettori:

$$\boldsymbol{x}_i = [x_1, x_2, \ldots, x_c]_i^T \tag{2.2.1}$$

dove $x_1, x_2, \ldots, x_c$ sono i valori assunti dal pixel per la banda corrispondente. I pixel devono essere riportati in una matrice X, di dimensioni $c \times k$, dove $k = m \cdot n$. Ogni colonna della matrice è una singola osservazione (pixel).

$$X_{c\times k} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{c1} & x_{c2} & \cdots & x_{ck} \end{bmatrix} \tag{2.2.2}$$

La matrice di covarianza viene calcolata con la seguente equazione:

$$Cov_{c\times c} = \frac{1}{k} \sum_{i=1}^{k} (\boldsymbol{x}_i - \boldsymbol{m}) (\boldsymbol{x}_i - \boldsymbol{m})^T \tag{2.2.3}$$

dove $\boldsymbol{m}$ è il vettore che ha per elementi la media dei valori assunti dai pixel per ciascuna banda:

$$\boldsymbol{m} = \frac{1}{k} \sum_{i=1}^{k} [x_1, x_2, \ldots, x_c]_i^T = \frac{1}{k} \sum_{i=1}^{k} \boldsymbol{x}_i \tag{2.2.4}$$

L'espressione della matrice Cov può essere riscritta sviluppando il prodotto:

$$Cov = \frac{1}{k} \sum_{i=1}^{k} \left(\boldsymbol{x}_i \boldsymbol{x}_i^T - \boldsymbol{x}_i \boldsymbol{m}^T - \boldsymbol{m} \boldsymbol{x}_i^T + \boldsymbol{m} \boldsymbol{m}^T \right)$$

$$Cov = \frac{1}{k}\sum_{i=1}^{k} \boldsymbol{x}_i \boldsymbol{x}_i^T - \boldsymbol{m}\boldsymbol{m}^T - \boldsymbol{m}\boldsymbol{m}^T + \boldsymbol{m}\boldsymbol{m}^T$$

$$Cov = \frac{1}{k}\sum_{i=1}^{k} \boldsymbol{x}_i \boldsymbol{x}_i^T - \boldsymbol{m}\boldsymbol{m}^T \tag{2.2.5}$$

In definitiva, il primo termine dell'equazione sopra può essere riscritto tramite la matrice X, portando alla formulazione più compatta:

$$Cov = \frac{1}{k} X X^T - \boldsymbol{m}\boldsymbol{m}^T \tag{2.2.6}$$

La matrice Cov, simmetrica e semidefinita positiva, può essere scomposta agli autovalori:

$$Cov = ADA^T \tag{2.2.7}$$

dove A è la matrice che ha per colonne gli autovettori di Cov, mentre D è la matrice diagonale con gli autovalori corrispondenti.

La trasformazione lineare che proietta i dati nel nuovo spazio è:

$$Y = A^T X \tag{2.2.8}$$

Ordinando gli autovettori della matrice A secondo l'ordine decrescente degli autovalori corrispondenti e utilizzando solo i primi h autovettori per la proiezione (prime h righe di A^T), si ottiene la rappresentazione Z di X nel nuovo spazio di sole h bande:

$$Z_{h\times k} = \begin{bmatrix} z_{11} & z_{12} & \cdots & z_{1k} \\ z_{21} & z_{22} & \cdots & z_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ z_{h1} & x_{h2} & \cdots & z_{hk} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{h1} & \cdots & a_{c1} \\ a_{12} & a_{22} & \cdots & a_{h2} & \cdots & a_{c2} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{1h} & a_{2h} & \cdots & a_{hh} & \cdots & a_{ch} \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{h1} & x_{h2} & \cdots & x_{hk} \\ \vdots & \vdots & \ddots & \vdots \\ x_{c1} & x_{c2} & \cdots & x_{ck} \end{bmatrix} \tag{2.2.9}$$

L'autovettore $\mathbf{a}_i = [a_1, a_2, \ldots, a_c]_i$ è il vettore contenente i coefficienti della combinazione lineare che proietta i dati sull'asse i del nuovo spazio e prende il nome di componente principale. La rilevanza dell'asse i, in termini di varianza espressa, è legata al valore numerico dell'autovalore corrispondente alla componente principale usata per la proiezione. Da qui l'utilità di ordinare la matrice A secondo l'ordine decrescente

degli autovalori: se si vogliono tenere solamente le prime h bande più importanti basta troncare la matrice A in modo tale che contenga i primi h autovettori.

Infine, da Z si può ritornare al formato tipico di un'immagine iperspettrale, ritrasformandolo in un data cube di dimensioni ridotte $m \times n \times h$.

2.2.2 Algoritmo di selezione delle bande con OSP

L'algoritmo di selezione non supervisionata delle bande con OSP, proposto in [4], segue un approccio differente rispetto alla PCA nella riduzione del numero di bande delle immagini iperspettrali. L'algoritmo permette l'identificazione delle bande più informative tra quelle dell'immagine originale. Questo consente di applicare una riduzione della dimensione spettrale dell'immagine, mantenendo solamente le bande più informative trovate, oppure scegliendone da questo inseme un numero ancora più limitato. Uno dei vantaggi di un approccio di questo tipo è la conservazione del significato fisico delle bande originali, evitando così di alterare la risposta spettrale degli oggetti rappresentati.

Il termine non supervisionato fa riferimento al caso in cui le informazioni spettrali di un oggetto, per esempio la sua firma spettrale, non siano note a priori. Quando le informazioni sono note, l'algoritmo si limita a trovare le bande per cui tali informazioni sono più presenti. In caso contrario, è necessario selezionare le bande più distintive e informative [4].

Questo metodo di selezione delle bande è stato utilizzato tramite la corrispondente funzione built-in di MATLAB "selectBands", come spiegato in sezione 4.2, disponibile con la libreria "Hyperspectral Imaging Library" del "Image Processing Toolbox". L'implementazione dell'algoritmo non è rientrata nelle attività svolte per l'elaborato, dunque, la formulazione matematica non verrà affrontata nel dettaglio come per la PCA. Per una spiegazione approfondita si consiglia di consultare [4].

3 | Implementazione di reti neurali

3.1 Competizione "Seeing Beyond the Visible"

Dal 9 febbraio al 1 luglio 2022, la piattaforma AI4EO[1], sostenuta da ESA, ha ospitato una competizione internazionale con lo scopo di sollecitare il mondo della ricerca a sviluppare algoritmi in grado di sfruttare i dati satellitari per ottimizzare, e dunque limitare, l'uso dei fertilizzanti in agricoltura.

Fondamentale per la selezione del corretto mix di fertilizzanti è la conoscenza tempestiva, aggiornata e puntuale di alcuni parametri del suolo, quali le concentrazioni di potassio (K), pentossido di fosforo (P_2O_5), magnesio (Mg) e il valore del pH. Attualmente, il processo di ottenimento di queste informazioni è manuale: è necessario raccogliere campioni del terreno che devono poi essere analizzati da laboratori specializzati. Questo comporta costi elevati e notevoli ritardi.

L'approccio proposto dalla competizione si basa sull'utilizzo di algoritmi di intelligenza artificiale che a partire da immagini iperspettrali, catturate da un satellite, siano in grado di stimare, direttamente a bordo, il valore dei parametri richiesti. Una soluzione di questo tipo permetterebbe di velocizzare l'intero processo e consentirebbe di ottenere dati in maniera capillare. L'immagine, infatti, può essere facilmente frazionata in sezioni minori in modo da ottenere una stima puntuale dei parametri del suolo, superando i limiti posti dalla necessità di prelevare e processare un elevato numero di campioni sul campo.

Le informazioni fornite dagli organizzatori consentono lo sviluppo di reti neurali allenate in modalità supervisionata. In particolare, sono stati rilasciati due set di immagini iperspettrali: uno per il training, comprensivo dei reali valori dei 4 parametri

[1] `https://platform.ai4eo.eu/seeing-beyond-the-visible`

per ciascuna immagine, e uno per il test. La classifica della competizione si basa sulla seguente metrica:

$$Score = \frac{\sum_{i=1}^{4} \left(MSE_i / MSE_i^{base} \right)}{4} \tag{3.1.1}$$

dove

$$MSE_i = \frac{\sum_{j=1}^{|\psi|} \left(p_j - \hat{p_j} \right)^2}{|\psi|} \tag{3.1.2}$$

Nell'Equazione 3.1.2, il termine $|\psi|$ denota la cardinalità del set di test, mentre $\hat{p_j}$ e p_j rappresentano rispettivamente il valore dell'i-esimo parametro stimato e reale, sulla j-esima immagine. Si procede poi al calcolo di un errore quadratico medio (MSE_i) per ogni parametro sulle immagini di test. Questi vengono successivamente divisi per i corrispondenti MSE_i^{base}, ovvero gli errori conseguiti da un algoritmo che approssima i $\hat{p_j}$ con i valori medi riscontrati nei dati di training. Trattandosi di una misura dell'errore di approssimazione, uno *Score* più basso indica una soluzione migliore. Il calcolo del punteggio viene eseguito automaticamente dal server della competizione, dopo aver caricato sul sito un file csv contenente la stima dei parametri sul dataset di test. Il miglior algoritmo classificato verrà testato a bordo del satellite "Intuition-1", progettato da KP Labs, il cui lancio è previsto nel primo quadrimestre del 2023.

3.2 Riassunto delle attività svolte nel tirocinio

Le attività svolte durante il tirocinio sono state finalizzate alla partecipazione alla competizione di cui sopra. Poiché il ML e le reti neurali non sono state affrontate nel corso di laurea triennale, è stato dapprima necessario seguire alcuni brevi corsi introduttivi a questi argomenti.

A seguire, è stata svolta una ricerca in letteratura riguardante metodi di processing di immagini iperspettrali con reti neurali. La ricerca ha evidenziato la scarsità di lavori che trattino argomenti simili a quelli della competizione. In particolare, la maggior parte degli articoli verte su problemi di classificazione anziché di regressione.

La parte centrale del tirocinio ha riguardato lo sviluppo di codici per l'analisi del dataset e l'implementazione di due reti CNN. La rete alla base della migliore soluzione presentata è una modifica dell'EfficientNet-Lite0, la versione più leggera della famiglia di reti EfficientNet-Lite [2]. Tale soluzione ha ottenuto l'ottavo posto, su 47 partecipanti,

nella classifica provvisoria stilata alla chiusura della competizione, che si basa su un sottoinsieme delle immagini di test. La classifica definitiva verrà resa disponibile in ottobre durante la "IEEE International Conference on Image Processing (ICIP) 2022".

Il punto di partenza per questo elaborato è stata proprio la soluzione che ha ottenuto il miglior risultato alla sfida. Le attività svolte hanno portato ad alcune modifiche nell'organizzazione e nell'analisi del dataset, oltre che al confronto con la rete EdgeNeXt.

3.3 Partizionamento del dataset e k-fold Cross Validation

Il lavoro presentato in questo elaborato, è stato sviluppato a valle della competizione. Tuttavia, dal momento che i valori *ground truth* del test set non sono stati rilasciati, non è stato possibile utilizzare tali immagini. Senza i valori reali p_j di ogni parametro i, non è possibile calcolare i due errori quadratici medi utilizzati in Equazione 3.1.1 per ottenere il punteggio di valutazione, precedentemente calcolato automaticamente dal server. Alla luce di questa considerazione, si è dovuto ricorrere all'utilizzo del solo dataset di training, già piuttosto limitato, suddividendolo ulteriormente per ottenere nuove partizioni di training e di test.

Come meglio specificato nella prossima sezione di analisi (sezione 3.4), il dataset di training fornito dalla competizione, come del resto anche quello originale di test, risulta essere molto variabile: le immagini hanno dimensioni spaziali differenti, persino di un ordine di grandezza. Una semplice suddivisione di tale dataset in due sole partizioni, da usare per il training e per il test, rischierebbe di produrre dei risultati poco significativi. Infatti, la partizione usata per il test, e quindi la valutazione della rete, potrebbe contenere un elevato numero di campioni molto diversi da quelli che la rete ha visto in fase di addestramento, o, viceversa, delle immagini molto simili. Questo può portare a una sovrastima o sottostima delle performance della rete.

Per far fronte a questo problema, è stato deciso di compiere i training delle reti con il metodo di *k-Fold Cross Validation*. Ciò consiste nel dividere il dataset in k partizioni. Ognuna di queste partizioni svolge, a turno, il ruolo di set di test, mentre le restanti sono utilizzate per il training (Figura 3.1). In totale, la rete esegue k processi di addestramento, ognuno indipendente dagli altri, in cui però i dataset di training e

di test variano. Alla fine dei training, si calcola la media dei punteggi ottenuti per ciascuna iterazione. In questo modo, si compensa, almeno parzialmente, la possibilità di avere dei dataset di training e di test relativamente "facili" o "difficili", rendendo i risultati più rappresentativi della bontà dell'algoritmo. Inoltre, la rete viene testata sul dataset completo, senza che questo sia visto simultaneamente.

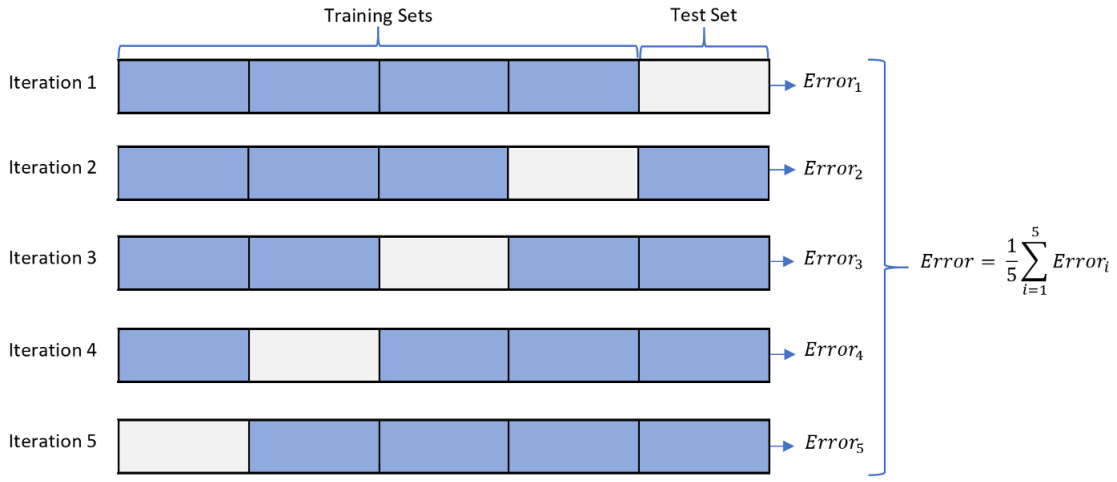

Figura 3.1: k-Fold Cross Validation con suddivisione del dataset in 5 partizioni [18].

Il dataset di training della competizione, d'ora in poi dataset completo, è stato convertito in formato TFRecord durante l'attività di tirocinio. Questo consiste in una codifica delle informazioni ottimizzata per la lettura sequenziale, dunque in grado di minimizzare i tempi di caricamento. Un ulteriore vantaggio consiste in un'associazione univoca di ogni immagine con il relativo label, ossia il vettore contenente i 4 parametri oggetto della competizione, in un unico record. Le funzioni, d'ora in poi frequentemente richiamate, per il caricamento dei file TFRecords contenenti i dataset sono riportate in Allegato 1.

Per applicare la tecnica di Cross Validation, il dataset è stato mescolato randomicamente e successivamente diviso in 5 partizioni. Ognuna di queste è stata salvata a sua volta in formato TFRecord. Il risultato è la creazione di 5 file TFRecord, uno per partizione. Durante la fase di addestramento della rete, a turno, 4 delle 5 partizioni costituiscono il dataset di training, mentre la partizione rimanente svolge il ruolo di set di test. Così facendo si ottengono dei dataset di training contenenti circa l'80% dei campioni totali.

3.4 Analisi del dataset

Prima di procedere con l'implementazione delle due reti neurali, si è provveduto ad analizzare il dataset completo ai fini di una corretta predisposizione delle operazioni di pre- e post- processing.

Questo consta di 1732 immagini. Di conseguenza, tre delle partizioni contengono 346 immagini, mentre le restanti due ne hanno 347.

Le immagini iperspettrali in oggetto sono composte da 150 bande (canali), corrispondenti alle lunghezze d'onda comprese tra $462nm$ e $942nm$, con una risoluzione spettrale di $3.2nm$. Per ogni immagine viene fornito un filtro che permette di delimitare la zona di interesse, assegnando un valore logico a ciascun pixel: "True" se il pixel non appartiene all'area di studio (e quindi il filtro deve escluderlo) oppure "False" se deve essere incluso (Figura 3.2). Tale notazione è controintuitiva e potrebbe sembrare errata, ma i valori logici devono essere pensati come riferiti all'azione di esclusione della maschera.

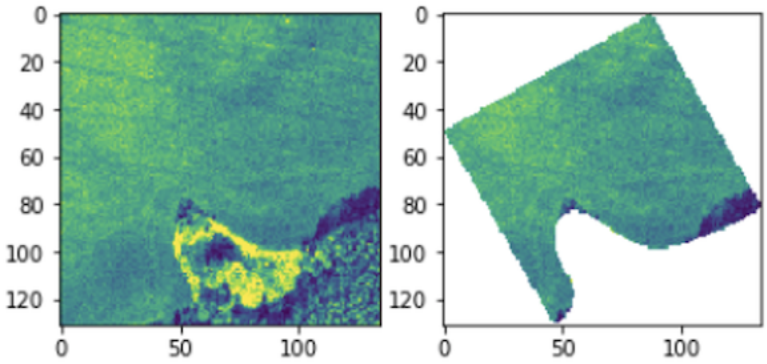

Figura 3.2: Rappresentazione monocromatica di una delle 150 bande su un'immagine casuale (sinistra) e applicazione del filtro (destra) [1].

Siccome le reti CNN implementate richiedono come input un'immagine quantomeno rettangolare sul piano spaziale e di dimensioni costanti, non è possibile pensare di passare un'immagine costituita dai soli pixel inclusi dalla maschera. Per risolvere questo problema, è stato deciso di "disattivare" tutti i pixel al di fuori della zona di interesse impostandone l'intensità a zero.

Quando si ricorre a reti neurali è bene normalizzare i dati di input ed output, al fine di facilitare il processo di training. In vista di una futura normalizzazione dei valori

assunti dai pixel nelle immagini si è provveduto al calcolo della deviazione standard. Questa è stata calcolata distintamente per ciascuna banda, considerando i pixel di tutte le immagini contemporaneamente, con l'accorgimento di escludere quelli nulli, in quanto tali a seguito dell'applicazione del filtro e quindi non significativi:

$$\sigma_i = \sqrt{\frac{\sum_{j=1}^{N} (x_{ji} - \mu_i)^2}{N}} \quad , \quad i = 1, \ldots, 150 \tag{3.4.1}$$

dove

$$\mu_i = \frac{1}{N} \sum_{j=1}^{N} x_{ji} \quad , \quad i = 1, \ldots, 150 \tag{3.4.2}$$

In Equazione 3.4.1, σ_i rappresenta la deviazione standard dei valori dei pixel nell'i-esimo canale, N è il numero di pixel non nulli di tutte le immagini e x_{ji} è l'intensità del j-esimo pixel nella i-esima banda. Infine μ_i denota la media delle intensità dei pixel sulla banda i calcolata come da Equazione 3.4.2.

Anche per i parametri del suolo sono stati calcolati i valori massimi, da usare come valori normalizzanti. Per una migliore comprensione delle caratteristiche di tali dati di output, si è provveduto inoltre al calcolo di media, mediana, deviazione standard e valore minimo, ottenendo i risultati riportati in Tabella 3.1. I valori di media sono stati fondamentali anche per il calcolo degli errori MSE_i^{base}, come descritto in sezione 3.1, usati per ottenere lo *Score* delle reti testate (Equazione 3.1.1).

Come già accennato, le immagini fornite si riferiscono a lotti di terreno di estensione diversa e presentano pertanto dimensioni variabili. In un primo momento dunque si è provveduto a verificare la distribuzione delle dimensioni delle immagini. A questo fine sono stati generati i due istogrammi riportati in Figura 3.3 e Figura 3.4. Come si può notare, le immagini vanno da una dimensione minima di 11 pixel, per entrambe le direzioni, a un'altezza massima di 268 pixel e una larghezza massima di 284 pixel.

Parametro del suolo	Media	Deviazione standard	Mediana	Massimo	Minimo
P_2O_5 [ppm]	70.30	29.50	65.1	325.0	20.3
K [ppm]	227.99	61.87	216.0	625.0	21.1
Mg [ppm]	195.28	39.86	155.0	400.0	26.8
Ph	6.78	0.26	6.8	7.8	5.6

Tabella 3.1: Risultati ottenuti dall'analisi dei valori di output.

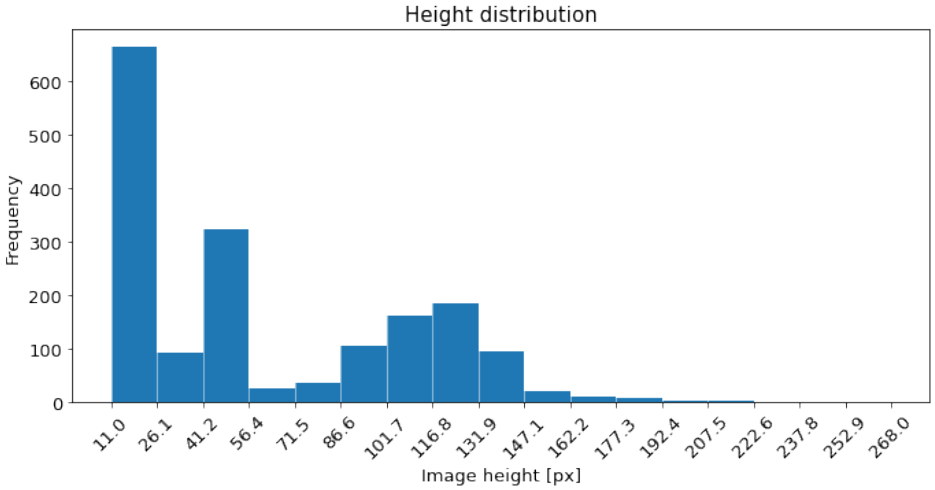

Figura 3.3: Istogramma rappresentante la distribuzione dell'altezza delle immagini.

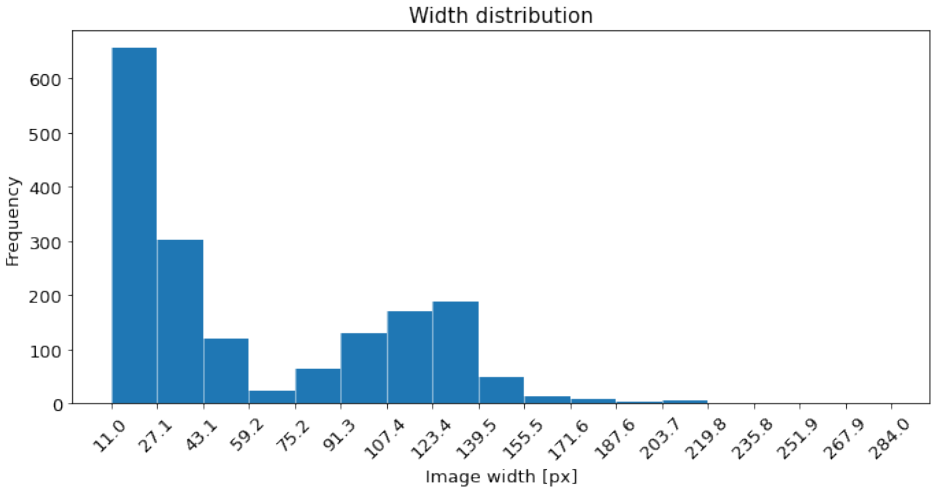

Figura 3.4: Istogramma rappresentante la distribuzione della larghezza delle immagini.

3.5 Preprocessing delle immagini

A partire dalle analisi svolte precedentemente, sono state definite alcune operazioni di pre-processing sui dataset di training e di test che possono essere riassunte nelle seguenti funzioni:

1. Normalizzazione
2. Ridimensionamento
3. Aggiunta del rumore
4. Mescolamento

Queste sono state implementate attraverso l'API tensorflow.data, come funzioni da assegnare al metodo "map" oppure grazie all'impiego di metodi già esistenti. Le prime due vengono applicate sia in fase di training che di test, mentre le ultime solo durante il training delle reti neurali. Queste funzioni di pre-processing sono riportate in Allegati 2 e 3.

3.5.1 Normalizzazione

Le intensità dei pixel vengono normalizzate per la deviazione standard corrispondente alle rispettive bande, come calcolato in Equazione 3.4.1. Tuttavia, per garantire dei valori sufficientemente bassi, la deviazione standard è prima stata moltiplicata per 10. La scelta di questa particolare normalizzazione anziché altre (massimo assoluto, massimo relativo alla banda) si basa su un osservazione pratica fatta durante l'implementazione della PCA che verrà spiegata in sottosezione 4.1.2.

I parametri esatti forniti alla rete con il set di training vengono anch'essi normalizzati per i rispettivi valori massimi riscontrati in fase di analisi. Fa eccezione il pH che è stato normalizzato rispetto a 14: massimo valore fisicamente possibile. In fase di training, la normalizzazione è stata applicata una sola volta ricorrendo al metodo "map": il dataset così trasformato è stato successivamente memorizzato nella cache per ottimizzare il codice.

3.5.2 Ridimensionamento

Poiché il dataset comprende immagini di dimensioni variabili da 11 a 286 px, sin da subito si è posto il problema di come uniformare una così pronunciata dispersione. Inizialmente si è provveduto ad eseguire un "padding" di tutte le immagini alle dimensioni massime individuate nella parte di analisi. Questo processo consiste nell'aggiunta di pixel di intensità nulla ai bordi fino al raggiungimento delle dimensioni desiderate, in questo caso 268×284 px. Questa scelta, di primo tentativo, si è rivelata piuttosto inefficiente: nella maggioranza dei casi, la maggior parte dell'immagine conteneva più pixel nulli che pixel informativi, limitando fortemente la capacità del modello di apprendere dai dati. Nondimeno, il numero di parametri della rete richiesta per processare un input di tali dimensioni, nonché la memoria occupata, sarebbero stati difficilmente compatibili con un'implementazione in tempo reale a bordo di un piccolo satellite.

A valle di queste considerazioni, si è deciso di ridurre le dimensioni delle immagini a 32×32 px, anche dopo aver osservato dagli istogrammi in Figura 3.3 e Figura 3.4, che la maggior parte di esse ha dimensioni inferiori. Inoltre, 32×32 px è anche la dimensione minima di input richiesta dalla Efficientnet-Lite0, poiché nel progredire, la rete dimezza 5 volte le dimensioni. Allo stesso tempo è stato introdotto un algoritmo di pre-processing più elaborato, sfruttando quando possibile l'API tensorflow.image:

- Le immagini che presentano altezza o larghezza maggiore di 32 px vengono ridotte alla dimensione target con la funzione "resize", usando come metodo l'interpolazione bilineare.

- Le immagini che hanno almeno una dimensione inferiore o uguale a 16 px (e quindi che sono ripetibili almeno una volta in una direzione) vengono ripetute e concatenate, il massimo numero intero di volte possibile, fino a raggiungere i 32 px. Qualora tale dimensione non sia raggiungibile in modo esatto con la sola ripetizione, viene eseguito un ulteriore padding tramite la funzione "pad_to_bounding_box".

- Le immagini con dimensioni comprese tra 16 e 32 px, quindi non ripetibili interamente, vengono ingrandite alla dimensione target tramite la funzione "resize" con metodo di interpolazione bilineare.

La scelta di un processo di affiancamento delle immagini, spiegato al secondo punto, consente di minimizzare il numero di pixel non informativi ed è giustificata dall'obiettivo della rete. Poiché scopo dell'indagine è regredire delle concentrazioni (e il pH), la ripetizione della stessa immagine per intero non ne influenza il valore numerico. Inoltre, ad ogni ripetizione, l'immagine viene randomicamente riflessa rispetto agli assi orizzontale e verticale. Questa trasformazione agisce a tutti gli effetti come un'aumentazione volta ad ampliare la variabilità del dataset così da prevenire fenomeni di overfitting e possibilmente migliorare la generalizzazione delle reti.

Per assicurare la ripetibilità nella stima dei parametri da parte delle reti, e quindi evitare l'influenza di possibili fattori esterni, le immagini del dataset di test non sono sottoposte a riflessioni randomiche. Questo ha facilitato molto l'implementazione dell'algoritmo di ripetizione poiché è stato sufficiente applicare la funzione "tf.tile", che ripete e concatena l'immagine per il numero specificato di volte nelle direzioni assegnate. Tale funzione non permette tuttavia di aggiungere ripetizioni randomiche, come invece voluto per il dataset di training. Per questo motivo si è dovuto ricorrere a un'implementazione *custom* dell'algoritmo di ripetizione per il training set (le differenze descritte sono visibili confrontando le funzioni "pad_with_patches_train" e "pad_with_patches_test", rispettivamente in Allegato 2 e Allegato 3).

In Figura 3.5 vengono riportati degli esempi, per dare un'idea più chiara del processo di ridimensionamento.

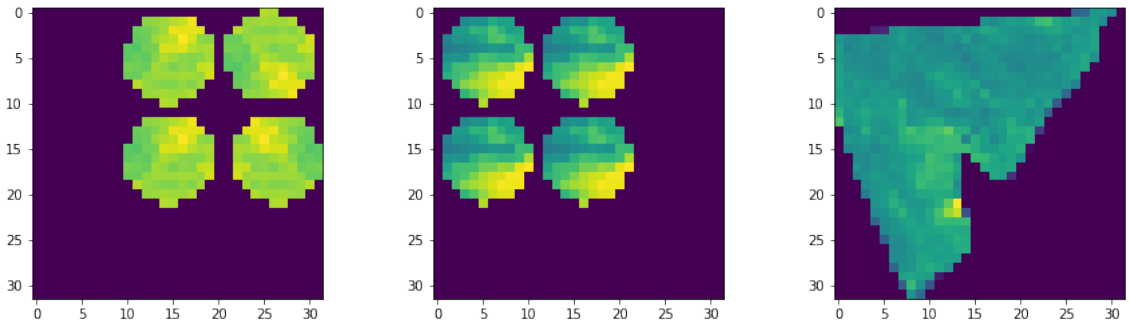

Figura 3.5: Immagine ripetuta con riflessioni randomiche per il training (sinistra), immagine ripetuta senza riflessioni randomiche per il test set (centro), immagine ridotta (destra).

3.5.3 Aggiunta del rumore

Durante la partecipazione alla competizione, è stata notata la tendenza delle reti impiegate all'overfitting. Per cercare di risolvere questo problema, una misura efficace si è rivelata essere l'aggiunta di rumore alle immagini. Il tipo di rumore scelto è un rumore gaussiano. A questo scopo, dopo il ridimensionamento delle immagini, viene generato un tensore delle stesse dimensioni, i cui valori sono ottenuti da una distribuzione normale caratterizzata da media nulla e deviazione standard pari a 0.05. Tale tensore viene poi sommato all'immagine per ottenere l'effetto del rumore. Il risultato visivo è riportato in Figura 3.6.

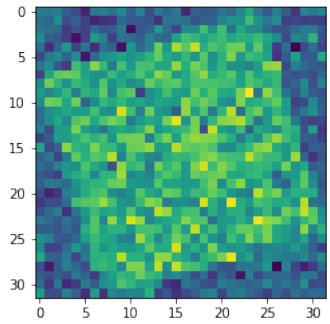

Figura 3.6: Effetto del rumore su una delle bande dell'immagine iperspettrale.

L'aggiunta del rumore avviene solo per il dataset destinato al training della rete (funzione "add_gauss_noise" in Allegato 2). Utilizzare il rumore in fase di test porterebbe a delle stime sempre differenti dei parametri, rendendo i punteggi ottenuti dalle reti non confrontabili. È infatti necessario assicurarsi che le immagini di test utilizzate per ottenere le predizioni siano sempre identiche, in modo tale da poter confrontare le prestazioni di diverse soluzioni.

3.5.4 Mescolamento

Nonostante le partizioni siano state ottenute da un dataset già mescolato randomicamente, è comunque opportuno che durante il training la rete non riceva i campioni sempre nello stesso ordine. In questo modo si evitano eventuali *pattern* o *bias*, a favore di una distribuzione più uniforme e quindi di una migliore capacità di generalizzazione

della rete. Il mescolamento del dataset usato per il training viene ripetuto all'inizio di ogni epoca ed è eseguito con il metodo "shuffle" dell'API tensorflow.data.

3.6 La rete EfficientNet-LiteB0mod

La prima rete utilizzata in questo confronto è la soluzione che, in abbinamento alle tecniche di pre-processing delle immagini sopra descritte, ha ottenuto il miglior punteggio alla competizione durante l'attività di tirocinio. Questa soluzione si basa sulla rete EffcientNet-Lite0, versione più leggera della famiglia di reti EfficientNet-Lite [2]. Tali reti derivano dai modelli EfficientNet [19] ottimizzati per il processing di immagini in ambito *mobile* e su dispositivi embedded. Le versioni "lite" di questi modelli sono compatibili con l'API TensorFlow Lite, requisito necessario per una successiva implementazione sul dispositivo embedded utilizzato.

Le reti EfficientNet-Lite sono destinate alla classificazione di immagini RGB. Per rendere compatibile la versione "0" con i dataset di immagini iperspettrali, e predisporla al compito di regressione dei parametri del suolo, sono state apportate alcune modifiche che hanno dato vita alla EfficientNet-Lite0mod. Il numero di canali in ingresso alla prima convoluzione è stato aumentato da 3 a 150 mentre le dimensioni spaziali dell'input sono state impostate a 32 px (dimensioni delle immagini dopo il pre-processing). Questa modifica ha tuttavia determinato l'impossibilità di ricorrere al transfer learning inizializzando la rete con i pesi preaddestrati sul dataset ImageNet, perché incompatibili con la nuova struttura. Infatti, il modello originale è realizzato per lavorare con immagini di dimensioni $224 \times 224 \times 3$. La testa di regressione della rete è stata realizzata con un livello completamente connesso, costituito da 4 neuroni e privo di attivazione. Al fine di ridurre ulteriormente il costo computazionale e il numero di parametri della rete, i due coefficienti di larghezza (width) e profondità (depth) sono stati impostati a 0.5. Questi controllano il numero di blocchi e di livelli che costituiscono il modello rispetto alla struttura originaria della EfficientNet-Lite0. Infine, il dropout rate è stato scelto del 10%. I dettagli sull'architettura del modello sono riassunti in Tabella 3.2, mentre una rappresentazione visiva ottenuta tramite la libreria VisualKeras [20] è riportata in Figura 3.7.

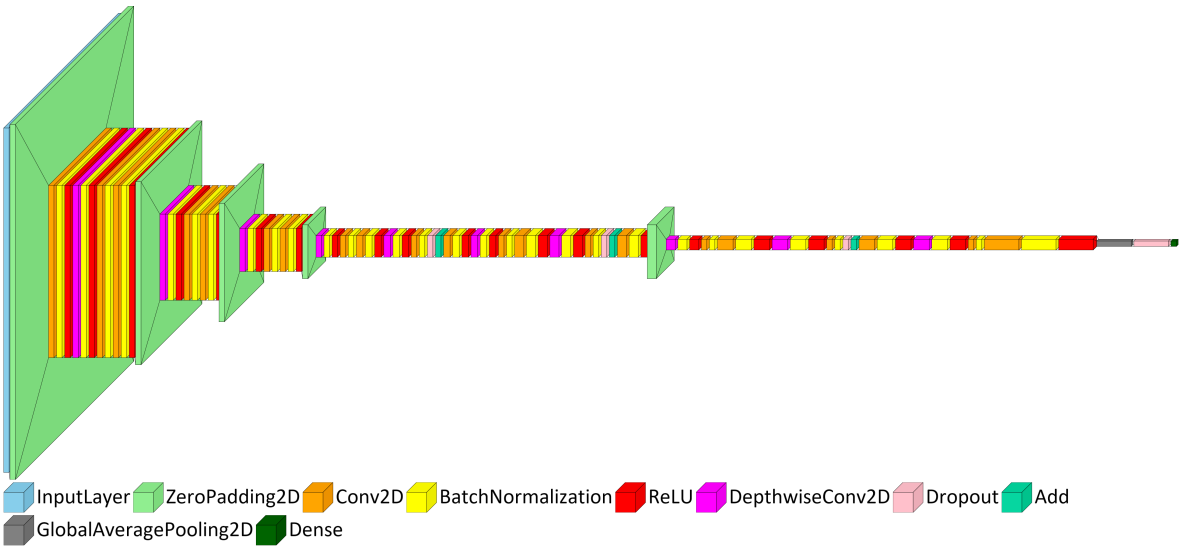

Figura 3.7: Schema dell'EfficientNet-Lite0mod.

Layer	Kernel	Output Size	Output Channels	#Layers
Image	-	32×32	150	1
Conv2D + BN + ReLU	3×3	16×16	32	1
MBConv6	3×3	16×16	8	1
MBConv6	3×3	8×8	16	1
MBConv6	5×5	4×4	24	1
MBConv6	3×3	2×2	40	2
MBConv6	5×5	2×2	56	2
MBConv6	5×5	1×1	96	2
MBConv6	3×3	1×1	160	1
Conv2D + BN + ReLU	1×1	1×1	1280	1
Dense	-	1×1	4	1
Model Parameters		734,020		

Tabella 3.2: Architettura della rete EfficientNet-Lite0mod e numero di parametri. Conv2D indica la convoluzione 2D, BN denota Batch Normalization, ReLU è l'omonima funzione di attivazione, MBConv6 è il mobile inverted bottleneck [21].

3.7 La rete EdgeNeXt-XXSmod

La rete EdgeNeXt è stata sviluppata con l'intento di creare un'architettura efficiente, a basso costo computazionale, compatibile con dispositivi *edge*, che sfrutti i punti di forza delle CNN e dei Vision Transformer [3]. È caratterizzata da un'architettura ibrida, costituita da livelli convoluzionali e moduli di *self-attention*, frequentemente utilizzati nei Transformer. I primi permettono di riconoscere e modellare i caratteri locali delle immagini, gli ultimi ampliano il campo ricettivo mettendo in relazione le caratteristiche multiscala [3].

Esistono tre versioni della rete EdgeNeXt. Si differenziano non tanto per l'architettura quanto per il numero di kernel che ciascun livello utilizza. Anche in questo caso è stata scelta la versione con il minore numero di parametri, chiamata EdgeNeXt-XXS (Figura 3.8). Essendo una rete destinata alla classificazione di immagini di dimensioni $256 \times 256 \times 3$, si è ricorso a una modifica del tensore di input, portandolo a $32 \times 32 \times 150$. Le dimensioni dei kernel e il numero di blocchi Conv Encoder sono tarati basandosi sulle dimensioni originali delle immagini, ben più grandi di 32 px. Per questo motivo, sono stati rimossi due Conv Encoder dal terzo stadio e uno dal primo. Inoltre, le convoluzioni di Downsampling sono state fatte con kernel di dimensione 2×2, mentre quelle dei Conv Encoder con kernel 3×3. La testa di regressione è stata costruita con un livello completamente connesso dotato di solo 4 nodi e privo di attivazione. La nuova rete è stata chiamata EdgeNeXt-XXSmod e i dettagli sono riportati in Tabella 3.3.

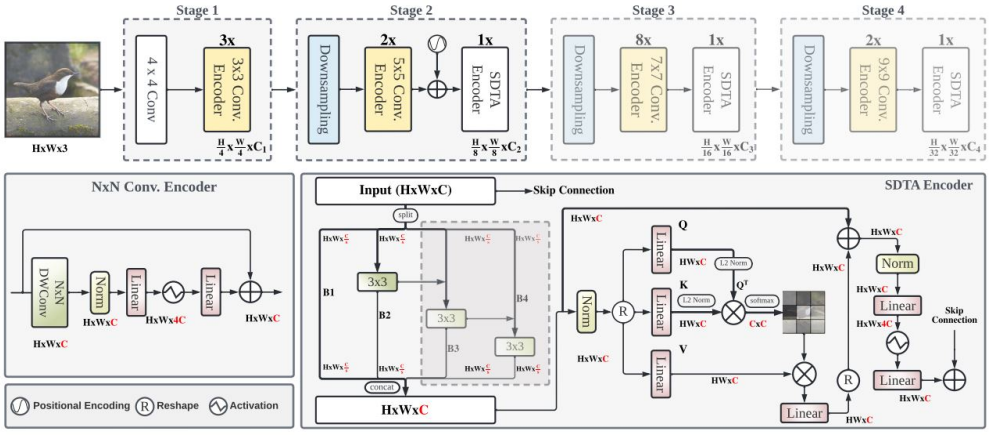

Figura 3.8: Architettura originale della rete EdgeNeXt (in alto) e dettaglio dei blocchi che la costituiscono (in basso) [3].

Layer	Kernel	Output Size	Output Channels	#Layers
Image	-	32×32	150	1
Stem	2×2	16×16	24	1
Conv. Encoder	3×3	16×16	24	2
Downsampling	2×2	8×8	48	1
Conv. Encoder	3×3	8×8	48	2
STDA Encoder	-	8×8	48	1
Downsampling	2×2	4×4	88	1
Conv. Encoder	3×3	4×4	88	6
STDA Encoder	-	4×4	88	1
Downsampling	2×2	2×2	168	1
Conv. Encoder	3×3	2×2	168	2
STDA Encoder	-	2×2	168	1
Global Average Pooling	-	1×1	168	1
Dense	-	1×1	4	1
Model Parameters		1,141,676		

Tabella 3.3: Dettaglio struttura EdgeNeXt-XXSmod.

3.8 Training delle reti e confronto

Le reti sono state implementate tramite l'API di TensorFlow Keras. Come conseguenza del ricorso alla k-Fold Cross Validation (sezione 3.3), che ha portato alla suddivisone del dataset in 5 partizioni, per ciascuna delle reti sono stati eseguiti 5 training. Ognuno di essi si differenzia dagli altri per l'utilizzo di una particolare partizione come dataset di test, quindi delle restanti per il training. Il processo di training e di valutazione delle reti avviene nel seguente modo:

1. Tutte e 5 le partizioni, salvate sotto forma di file TFRecord, vengono importate come classe Dataset di TensorFlow. Dopo aver specificato quale partizione de-

ve costituire il dataset di test, le restanti 4 vengono concatenate con il metodo "concatenate" per costituire un unico dataset di training.

2. Il dataset di training è mappato attraverso gli algoritmi di pre-processing descritti in sezione 3.5 (Allegato 4).

3. La rete da addestrare viene costruita e vengono impostati i parametri di training, dopodiché la rete viene compilata e addestrata. Alla fine del training, il modello viene esportato.

4. Il dataset di test viene sottoposto agli algoritmi di pre-processing, come spiegato in sezione 3.5 (Allegato 5).

5. Alla rete addestrata viene fornito in input il dataset di test. La rete elabora una stima dei parametri per ciascuna immagine. Una serie di funzioni di post-processing salva i valori stimati in un file csv e infine calcola il punteggio come da Equazione 3.1.1.

Per un confronto alla pari e per consentire la ripetibilità dei risultati, si è provveduto a impostare un seed globale ed uno specifico per ogni operazione randomica, inclusa l'inizializzazione dei pesi della rete. Inoltre le condizioni di training sono state mantenute invariate, nel corso di tutti gli addestramenti e per entrambi i modelli. Le reti sono state compilate con errore quadratico medio come funzione di loss ed una metrica personalizzata, rappresentativa di quella utilizzata per il calcolo del punteggio durante la competizione. Il training è stato eseguito su server GPU di Google Colab per 400 epoche, con un batch size di 32, learning rate iniziale di 0.005, diminuito progressivamente in base ad una legge cosine decay (Figura 3.9), e ottimizzatore Adam.

Si è provveduto a monitorare l'andamento dell'errore quadratico medio e della metrica personalizzata in funzione delle varie epoche. Tali andamenti sono stati monitorati anche per un piccolo set di validazione, costituito da immagini scartate nel processo di batching, in virtù del fatto che il numero di immagini di training non è divisibile per le dimensioni del batch.

I punteggi parziali ottenuti dalle reti a valle dei 5 training e i loro valori medi sono riportati in Tabella 3.4. Come si può notare dall'elevata variabilità dei punteggi ottenuti

usando partizioni differenti, l'approccio con Cross Validation è stato fondamentale per ottenere una stima più attendibile della vera capacità di predizione delle reti.

Complessivamente, la rete EfficientNet-Lite0mod è risultata più accurata della EdgeNeXt-XXSmod, confermando oltretutto la bontà della soluzione proposta alla competizione durante l'attività di tirocinio. Sulla base di questi risultati, la soluzione costituita dagli algoritmi di pre-processing descritti finora e dal modello EfficientNet-Lite0mod è stata impiegata congiuntamente ai metodi di riduzione delle bande, la cui implementazione viene affrontata nel prossimo capitolo. Inoltre, i punteggi ottenuti da questa rete verranno utilizzati come *benchmark* per confrontare le prestazioni della rete in abbinamento alla riduzione delle bande.

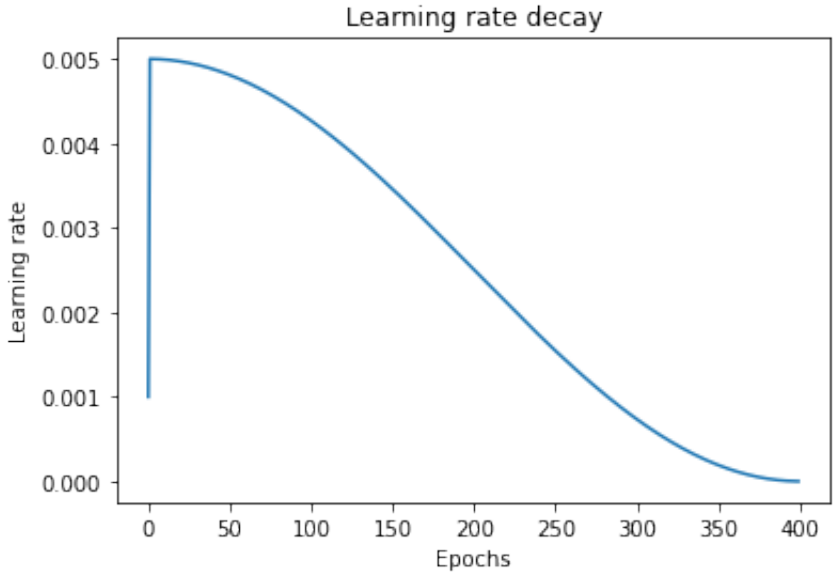

Figura 3.9: Decadimento del learning rate.

Model	Intermediate Score					Final Score
	1	2	3	4	5	
EfficientNet-Liteb0mod	0.881	0.985	0.997	0.894	0.768	0.905
EdgeNeXt-XXSmod	0.928	1.007	1.052	0.954	0.856	0.960

Tabella 3.4: Punteggi ottenuti dalle reti.

4 | Implementazione di algoritmi di riduzione delle bande

4.1 Implementazione della PCA

Nonostante esistano librerie che permettono di applicare la PCA su Python (TensorFlow Tranform, scikit-learn) si è ricorso a un'implementazione custom per ottenere una soluzione più versatile, efficiente, per il caso in esame, e per avere un maggiore controllo. Sorgono infatti due necessità nell'applicazione della PCA al dataset di immagini iperspettrali studiato: la gestione dei pixel nulli e il rientrare nei limiti di RAM dei dispositivi utilizzati.

Per quanto riguarda il primo punto, le funzioni per la PCA delle librerie sopra menzionate applicano la trasformazione all'immagine intera, senza la possibilità di esclusione dei pixel nulli. Ricordando quanto detto in sezione 3.4, i pixel nulli non sono quelli per cui l'intensità è originariamente pari a zero, ma semplicemente quelli esclusi poiché non appartenenti alla zona di interesse. Non essendo ritenuti significativi per lo spettro di riflessione complessivo, sono stati esclusi nel calcolo delle componenti principali. È bene notare che a prescindere dall'esclusione, i pixel nulli rimangono comunque tali anche a seguito della proiezione dell'immagini nel nuovo spazio.

Riguardo la seconda richiesta, le versioni implementate da queste librerie permettono di applicare la PCA considerando una sola immagine alla volta, sotto forma di una matrice bidimensionale come descritto in sottosezione 2.2.1. Visto l'utilizzo di reti neurali, non è possibile pensare di applicare la PCA alle immagini individualmente, in quanto porterebbe a proiezioni basate tutte su matrici degli autovettori differenti. Questo equivarrebbe a ottenere un dataset in cui ciascuna immagine utilizza una legge

di proiezione differente. Questa soluzione si scontra con la necessità da parte delle reti di ricevere un dataset omogeneo, costituito da elementi uniformi tra loro. È quindi necessario trovare un modo per eseguire la PCA considerando tutti i pixel di tutte le immagini del dataset contemporaneamente.

Per usufruire delle librerie disponibili e non incorrere nel problema sopra menzionato, sarebbe stato necessario salvare tutti i pixel delle immagini in un unica matrice, soluzione inefficiente computazionalmente e difficilmente realizzabile, vista l'elevata quantità di memoria RAM che questa andrebbe a occupare. Sfruttando le proprietà della matrice di covarianza è invece possibile costruirla considerando un'immagine alla volta e allo stesso tempo fare in modo che essa sia riferita all'intero set di dati.

4.1.1 Premessa teorica

Si supponga di voler calcolare la matrice di covarianza associata all'intero dataset di immagini iperspettrali. Per prima cosa, come spiegato in sottosezione 2.2.1, bisogna riordinare i pixel di tutte le immagini in una matrice X di dimensioni $c \times k$, dove c è il numero di bande e k è la somma del numero di pixel di tutte le immagini. Si riporta nuovamente, per comodità, l'equazione della matrice di covarianza già espressa in Equazione 2.2.6:

$$Cov = \frac{1}{k} XX^T - \boldsymbol{m}\boldsymbol{m}^T \tag{4.1.1}$$

dove $\boldsymbol{m}$ è il vettore della media (si veda sottosezione 2.2.1 per i dettagli). Se il dataset contiene ψ immagini, allora, la matrice dei pixel X non è altro che il concatenamento delle matrici individuali dei pixel:

$$X = [X_1, X_2, \ldots, X_\psi] \tag{4.1.2}$$

Il prodotto XX^T di Equazione 4.1.1 può così essere riscritto come:

$$XX^T = X_1X_1^T + X_2X_2^T + \ldots + X_\psi X_\psi^T \tag{4.1.3}$$

Da questa nuova formulazione si può osservare come in realtà non sia affatto necessario memorizzare tutti i pixel nella matrice X, bensì è sufficiente calcolare i prodotti $X_iX_i^T$ con $i = 1, \ldots, \psi$ sequenzialmente, sommandoli di volta in volta. Esaurite tutte le immagini, si ottiene il prodotto completo XX^T da sostituire in Equazione 4.1.1 per il calcolo della matrice di covarianza.

Chiaramente, anche il calcolo del vettore della media $\boldsymbol{m}$ può essere fatto senza ricorrere a un'unica matrice X. In questo caso sarà sufficiente sommare le intensità dei pixel di ciascuna immagine per ogni banda, eseguire una somma progressiva di tali somme intermedie e infine dividere per k.

4.1.2 Calcolo della matrice di covarianza

Sfruttando i metodi descritti nella sottosezione precedente, sono state definite una serie di funzioni che permettono di calcolare la matrice di covarianza senza ricorrere alla memorizzazione di una matrice dei pixel comprensiva di tutte le immagini. Il calcolo avviene secondo i seguenti passi (Allegato 6):

1. Il dataset completo, composto dalle 5 partizioni, viene caricato come classe Dataset dell'API tensorflow.data. Una prima funzione trasforma le immagini nelle matrici dei pixel bidimensionali X_i, eliminando però i pixel nulli, in quanto tali solo per azione del filtro e quindi non significativi.

2. Facendo ricorso al metodo "reduce", vengono calcolati il vettore della media $\boldsymbol{m}$ e la deviazione standard relativa alle intensità di ogni banda.

3. Le matrici dei pixel X_i vengono normalizzate come spiegato in sottosezione 3.5.1.

4. Sempre utilizzando il metodo "reduce" si calcola la somma progressiva dei contributi $X_i X_i^T$ e infine la matrice di covarianza.

5. La matrice di covarianza e il vettore della deviazione standard vengono salvati per un utilizzo futuro.

La normalizzazione per la deviazione standard è il risultato di un'osservazione empirica fatta durante il calcolo della matrice di covarianza. Essendo il valore medio dell'intensità dei pixel generalmente crescente con l'aumentare della lunghezza d'onda, la varianza delle ultime bande risulta per forza di cose maggiore di quella delle prime. Il compito della PCA è proprio proiettare i dati lungo gli assi di massima varianza. Senza una normalizzazione, si rischierebbe di falsare i risultati ottenendo delle componenti che apparentemente raccolgono la totalità delle informazioni solamente perché caratterizzati da intensità, e quindi varianza, più elevate. Tra le normalizzazioni testate,

quella che ha permesso di ottenere un maggiore numero di bande rilevanti è proprio la normalizzazione per la deviazione standard.

La correttezza delle funzioni implementate è stata validata salvando almeno una volta la matrice completa X e applicando la funzione "cov" della libreria NumPy.

4.1.3 Applicazione della PCA in pre-processing

Una volta calcolata la matrice di covarianza è necessario decomporla agli autovalori, così da ottenere la matrice degli autovettori, utilizzata per la proiezione dei dati. A questo proposito, sono state definite due funzioni, applicate ai Dataset di TensorFlow con il metodo "map". Una prima funzione ("reshape_and_normalize" in Allegato 7) trasforma le immagini a cui si vuole applicare la PCA nel formato corretto: queste vengono riorganizzate nelle matrici dei pixel e poi normalizzate. Una seconda funzione ("apply_pca" in Allegato 7) calcola la matrice degli autovettori decomponendo la matrice di covarianza (funzione "eigh" dell'API tensorflow.linalg), seleziona il numero di autovettori corrispondenti al numero di bande che si vogliono avere nella nuova rappresentazione, proietta la matrice dei pixel X_i usando gli autovettori scelti e infine ritrasforma X_i nel data cube tridimensionale.

Queste due funzioni diventano a tutti gli effetti delle nuove funzioni di pre-processing delle immagini. In caso di training con PCA, la prima va a sostituire completamente l'originale funzione di normalizzazione, mentre la seconda viene applicata prima del processo di ridimensionamento. I dataset vengono poi memorizzati nella cache e seguono le restanti funzioni di pre-processing precedentemente definite: quindi ridimensionamento (sottosezione 3.5.2), rumore (sottosezione 3.5.3) e mescolamento (sottosezione 3.5.4)

Le funzioni di pre-processing introdotte per l'applicazione della PCA e le modifiche di quelle già esistenti sono riportate in Allegato 7. L'ordine di applicazione di queste funzioni è descritto subito sopra e deve essere messo in relazione a quello originale riportato negli Allegati 4 e 5.

4.1.4 Osservazioni sull'applicazione della PCA

Requisito fondamentale per un corretto utilizzo della PCA come metodo di riduzione delle dimensionalità è la scelta del numero di componenti principali, quindi di bande,

da mantenere nella nuova rappresentazione. Come accennato in sottosezione 2.2.1, lo strumento più immediato a cui si può ricorrere per compiere tali valutazioni è il modulo degli autovalori associati agli autovettori utilizzati per proiettare i dati sui nuovi assi: maggiore è il valore numerico e più l'asse è rilevante in termini di varianza che riesce ad esprimere. La misura statistica che descrive la percentuale di varianza espressa da ciascuna componente principale è l'*explained variance*:

$$ev_i = \frac{\lambda_i}{\lambda_1 + \lambda_2 + \ldots \lambda_c} \tag{4.1.4}$$

dove λ_i è l'i-esimo autovalore della matrice di covarianza (associato all'i-esima componente principale) e c è il numero totale degli assi originali (in questo caso il numero di bande).

Calcolata in questo modo, l'explained variance permette di valutare la percentuale di varianza espressa dalla singola componente principale i. Sommando le explained variance individuali, in ordine decrescente, si ottiene l'andamento cumulativo, di maggiore uso pratico per capire quale sia l'effettivo numero di bande "significative". A valle dell'implementazione delle funzioni che permettono di applicare la PCA, è allora stata calcolata anche la *cumulative explained variance*, in modo tale da poter scegliere un range appropriato da cui estrarre i numeri di bande da confrontare in fase di training (Figura 4.1).

Osservando Figura 4.1 si nota come la maggior parte della varianza sia espressa solamente dalle prime due componenti principali, indice del fatto che le intensità dei pixel assunte nelle diverse bande dell'immagine iperspettrale originale sono strettamente correlate tra di loro. Per avere ulteriore conferma della rilevanza di questo risultato, è stato analizzato anche lo spettro di intensità dei pixel dopo l'applicazione della PCA. In Figura 4.2 è riportata l'intensità media dei pixel di un'immagine, già trasformata, per le prime 10 bande.

Il grafico in Figura 4.2, conferma come effettivamente le prime 3 bande contengano la maggior parte delle informazioni. Dalla quarta banda in poi l'intensità dei pixel si stabilizza, assumendo valori quasi nulli. Da questa figura, sorgono due ulteriori problematiche che devono essere prese in considerazione prima di effettuare un training:

- I moduli dei valori medi dei pixel delle prime due bande, valutati sull'intero dataset, fanno presupporre la presenza di intensità relativamente elevate e una di-

stribuzione dei valori di input in un intervallo abbastanza ampio. Da un'analisi successiva è infatti stato riscontrato un valore massimo assoluto di intensità pari circa a 6. Un range [0,6] delle intensità dei pixel potrebbe essere troppo esteso. Solitamente in letteratura, le reti vengono addestrate su immagini con valori normalizzati tra [-1,1] o [0,1]. Per questo motivo si è provveduto a un'ulteriore normalizzazione delle immagini in fase di pre-processing, questa volta dopo l'applicazione della PCA, dividendo per la massima intensità riscontrata tra tutte le bande e tutte le immagini del dataset (funzione "post_pca_normalization" in Allegato 7).

- I valori medi dei pixel delle prime bande sono molto variabili. Applicare un rumore caratterizzato da una deviazione standard fissa (si veda sottosezione 3.5.3), renderebbe di difficile scelta il valore di deviazione standard stesso. Un rumore adeguato per una banda rischia infatti di non esserlo per un'altra: potrebbe essere troppo grande rispetto al valore dei pixel in essa contenuti, oppure irrisorio, vanificando il proprio effetto. Alla luce di questa considerazione, la funzione di aggiunta del rumore, quando viene eseguito un training con PCA, imposta un valore di deviazione standard della distribuzione normale variabile con le bande, pari al 30% del valore medio dell'intensità dei pixel di ciascuna banda (funzione "add_gauss_noise" in Allegato 7).

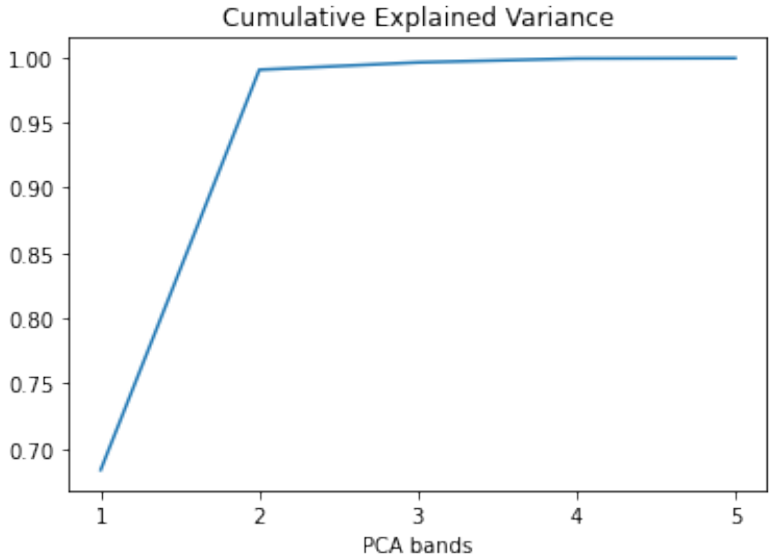

Figura 4.1: Explained Variance cumulativa.

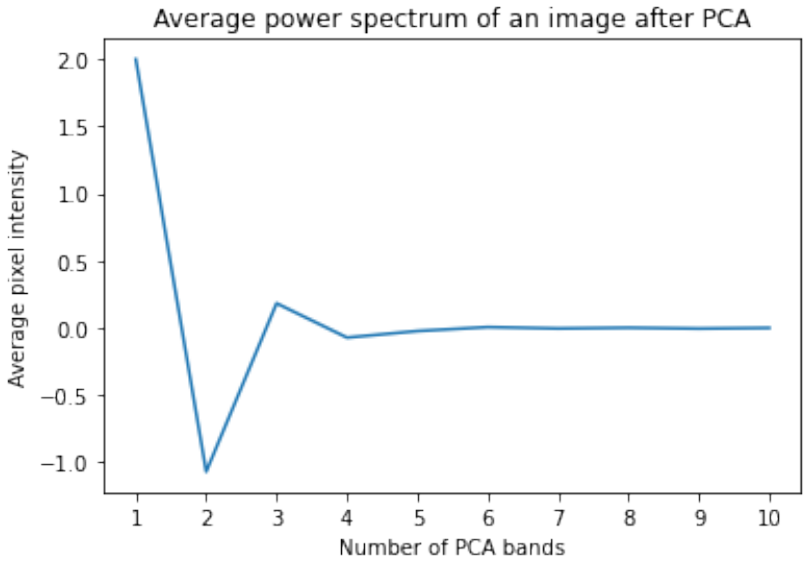

Figura 4.2: Spettro di intensità dei pixel di un'immagine trasformata tramite PCA.

4.2 Applicazione del metodo di selezione delle bande con OSP

L'algoritmo di selezione delle bande basato sull'OSP è implementato con la funzione built-in "selectBands" di MATLAB. Per questo motivo, non si è ricorso a un'implementazione da zero, ma è stato sufficiente spostarsi su MATLAB per compiere l'analisi delle bande più rilevanti, per poi salvare questi risultati e integrarli nei training delle reti svolti su Python.

4.2.1 Analisi delle bande più informative

La funzione "selectBands" richiede due input: l'hypercube per cui selezionare le bande più informative e gli *endmembers*. Gli endmembers sono gli spettri di riflessione corrispondenti a pixel che appartengono a una superficie omogenea. Per via della bassa risoluzione dei sensori iperspettrali, molti pixel di un immagine rappresentano in realtà superfici eterogenee, questi pixel vengono chiamati pixel misti. È quindi necessario ricorrere ad algoritmi di *spectral unmixing* per estrarre gli endmembers caratteristici di un immagine, scomponendo le firme spettrali dei pixel misti in quelle degli endmembers che li costituiscono [14].

L'algoritmo di spectral unmixing scelto è stato il Fast Iterative Pixel Purity Index (FIPPI), disponibile in MATLAB con l'omonima funzione "fippi". A sua volta, questa funzione chiede due input: l'hypercube da cui estrarre gli endmembers e il numero di endmembers da estrarre. Questo numero può essere inferiore o uguale al numero di endmembers presenti nell'immagine iperspettrale. In questo caso, è stato usato il numero esatto di endmembers, ricavato con la funzione "countEndmembersHFC".

Il processo di estrazione delle bande più informative avviene all'interno di un MATLAB Live Script (Allegato 8) come segue.

Per prima cosa, a partire dal dataset completo in file TFRecord, sono state salvate le immagini, individualmente, in file separati di formato npy. Utilizzando "readNPY" [22] come funzione di lettura, è stato creato un Datastore di MATLAB, contenente le immagini iperspettrali, a partire dai singoli file salvati in formato npy. Le immagini contenute nel Datastore, abbinate alle lunghezze d'onda di ciascuna banda, sono state trasformate nell'oggetto hypercube di MATLAB. In questo modo è stato ottenuto un Datastore formato da oggetti hypercube.

Un ciclo for scorre il Datastore contenente gli hypercube e, per ognuno di essi, trova il numero di endmembers, gli endmembers stessi e infine il vettore contenente gli indici corrispondenti alle bande più informative (il tutto utilizzando le tre funzioni sopra descritte). Il numero di bande più informative è variabile per ciascuna immagine: dipende dal numero e dagli endmembers trovati. I vettori contenenti gli indici delle bande più informative di ciascuna immagine vengono salvati in un unico cell array. A seguire, il cell array viene convertito in vettore singolo con la funzione "cell2mat". Questo vettore può essere visto come il concatenamento dei vettori degli indici delle bande più informative di ogni immagine. Gli indici delle bande vengono organizzati all'interno di un secondo vettore, eliminando le ripetizioni e in ordine decrescente rispetto alla frequenza con cui compaiono.

In questo modo si è ottenuto un vettore, contenente solamente gli indici corrispondenti alle bande più informative, ordinati per rilevanza rispetto all'intero dataset. Tale vettore è stato salvato in formato npy per poi essere riutilizzato nella selezione delle bande.

A scopo informativo è stata riportata la distribuzione delle bande più informative in Figura 4.3.

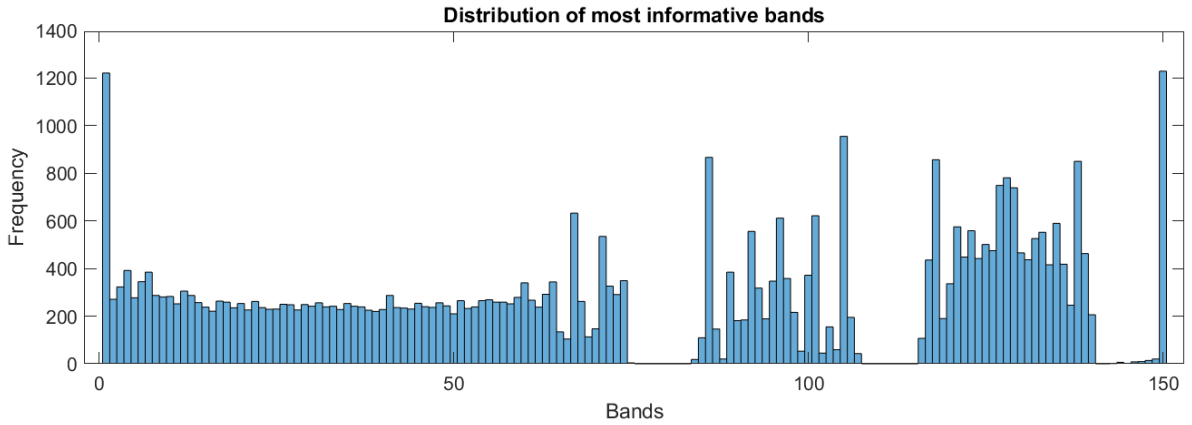

Figura 4.3: Distribuzione delle bande più informative ottenuta a partire dall'algoritmo di selezione con OSP: l'asse orizzontale rappresenta gli indici delle bande mentre l'asse verticale il numero di volte che ciascuna banda è risultata come informativa nell'intero dataset. I due intervalli in cui non compaiono ripetizioni sono relativi alla rimozione delle bande di assorbimento dell'acqua e di quelle con un basso rapporto segnale-rumore, applicata automaticamente dalla funzione "selectBands".

4.2.2 Applicazione della selezione delle bande più informative in pre-processing

In fase di pre-processing, per la riduzione dei canali basata sull'elenco delle bande più informative ottenute con l'applicazione della funzione "selectBands", viene caricato in memoria il vettore contenente tali bande, salvato precedentemente alla fine dell'analisi. Dopo aver specificato il numero di bande n che si vuole mantenere a seguito della riduzione dei canali, una nuova funzione di pre-processing seleziona dall'immagine solamente le bande corrispondenti ai primi n indici che compaiono nel vettore delle bande più informative (funzione "select_bands" in Allegato 9). Tale funzione si colloca tra le originali funzioni di normalizzazione e ridimensionamento, e viene applicata con il solito metodo "map" della classe Dataset dell'API tensorflow.data.

4.3 Risultati di training

I due algoritmi di riduzione delle bande appena implementati sono stati confrontati attraverso diversi training della rete EfficientNet-Lite0mod, la migliore tra le due testa-

te. Lo scopo è di verificare quali dei due metodi risulti essere la soluzione migliore da adottare per il problema in esame. La metrica di valutazione è sempre la stessa, ovvero lo *Score* di Equazione 3.1.1.

Il processo di training si svolge in modo del tutto analogo a quanto descritto per il confronto precedente (sezione 3.8), tutti i parametri sono gli stessi. L'unica differenza è l'aggiunta delle relative funzioni di pre-processing, in entrambi i casi posizionate tra la normalizzazione e il ridimensionamento.

I training sono stati eseguiti riducendo il numero di bande delle immagini a 3, 5 e 10 per entrambi gli algoritmi. Per ogni combinazione "numero di banda - algoritmo di riduzione" sono stati eseguiti 5 training, come prescritto dall'approccio con Cross Validation impiegato.

Method	#Bands	Intermediate Score					Final Score
		1	2	3	4	5	
PCA	3	0.840	0.942	0.939	0.861	0.784	0.873
PCA	5	0.812	0.969	0.957	0.841	0.786	0.873
PCA	10	0.839	0.947	0.958	0.874	0.823	0.888
OSP	3	0.843	0.984	1.005	0.882	0.818	0.906
OSP	5	0.851	1.003	0.984	0.914	0.830	0.916
OSP	10	0.845	1.000	0.989	0.899	0.785	0.904

Tabella 4.1: Punteggi relativi ai metodi di riduzione delle bande.

Dai risultati riportati in Tabella 4.1, si può notare che la PCA ha ottenuto punteggi sempre migliori di quelli dell'algoritmo di selezione non supervisionata con OSP. Per quanto riguarda la scelta del numero di bande, le performance migliori in assoluto sono state conseguite con applicazione della PCA e riduzione a 3 e 5 bande. La scelta per l'implementazione sul sistema embedded è ricaduta sulla soluzione facente uso della PCA con riduzione a 3 bande. Questa, se confrontata con PCA a 5 bande, a parità di punteggio, permette di lavorare con una mole di dati inferiore, che si traduce in un costo computazionale e in un consumo di memoria più contenuti in inferenza.

È comunque interessante osservare che entrambi i metodi utilizzati, indipendentemente dal numero di bande mantenuto, hanno registrato *Score* sempre inferiori rispetto alla soluzione senza riduzione delle dimensionalità (fatta eccezione per il metodo OSP con 5 bande).

5 | Inferenza sul computer a scheda singola Dev Board Mini

Il sistema Dev Board Mini consiste in un computer a scheda singola, dotato di acceleratore Tensor Processing Unit (TPU), sviluppato appositamente per implementazioni embedded di programmi di ML [23]. Il coprocessore TPU permette di compiere inferenza di modelli di ML a un basso costo energetico. Queste caratteristiche rendono il microcomputer particolarmente interessante per applicazioni di ML su piccoli satelliti [24]. Nonostante ciò, in questo lavoro è stata sfruttata solo la CPU "MediaTek 8167s SoC (Quad-core Arm Cortex-A35)" 1.5 GHz [23].

Le dimensioni (Figura 5.1) e il peso sono estremamente limitati, rispettivamente $26.4 \times 48.0 \times 14.6\ mm$ e $25.5g$.

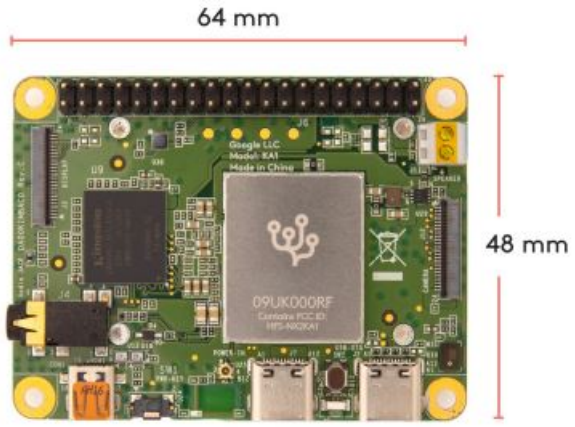

Figura 5.1: Dispositivo Coral Dev Board Mini [23].

Sul dispositivo, è installata anche una versione ridotta di TensorFlow, chiamata "tflite_runtime". Essa contiene solamente le librerie indispensabili per compiere infe-

renza di modelli con estensione tflite e alcuni delegati ottimizzati per dispositivi ARM. L'estensione tflite si basa sul formato FlatBuffers che permette di avere modelli di dimensioni ridotte (in termini di spazio di archiviazione) e con tempi di inferenza inferiori su dispositivi embedded rispetto al formato SavedModel di TensorFlow [25]. L'utilizzo di tflite_runtime, oltre che a garantire l'esecuzione di modelli ottimizzati tflite, permette di risparmiare memoria, evitando l'installazione e l'importazione della libreria completa TensorFlow che risulterebbe oltretutto poco sfruttata quando l'unico suo scopo è quello di eseguire inferenza.

L'obiettivo di quest'ultima parte dell'elaborato è di ottimizzare il processo di inferenza della migliore soluzione trovata (EfficientNet-Lite0mod + PCA 3 bande) dimostrandone l'implementabilità in tempo reale sul dispositivo Dev Board Mini. A questo proposito è stato effettuato un confronto tra i tempi di inferenza conseguiti con l'esecuzione dei modelli originali TensorFlow e quelli in formato tflite su tflite_runtime. Le funzioni di pre-processing e i codici di inferenza già sviluppati all'interno dei Notebook utilizzati per i confronti precedenti sono stati per quanto possibile riciclati. Tuttavia, alcune modifiche sono state effettuate per questioni di compatibilità. Il flusso di lavoro può essere riassunto nei segenti punti:

- Conversione del dataset
- Conversione dei modelli
- Adattamento dei codici
- Inferenza e risultati

5.1 Conversione del dataset

In uno scenario applicativo reale, le immagini vengono catturate e salvate individualmente, utilizzando un certo formato. Ciascuna immagine viene poi elaborata, a bordo del satellite, prima con l'applicazione dei metodi di pre-processing, poi con la regressione dei parametri del suolo da parte della rete. Finora, per motivi pratici, l'accesso alle immagini delle 5 partizioni è sempre stato fatto attraverso i file TFRecord corrispondenti. Ciascun file, raggruppa sia le immagini sia i valori ground truth.

Per rendere l'intero processo di inferenza il più simile possibile a una reale applicazione della soluzione, da ogni file TFRecord sono state estratte le immagini e salvate individualmente. Il formato scelto è npy di NumPy. Questo ha portato alla formazione di 5 cartelle contenenti le immagini, una per ogni partizione. Dai file TFRecord sono stati estratti e salvati in 5 file csv anche i label corrispondenti a ciascuna immagine.

5.2 Conversione dei modelli al formato tflite

Per ognuna delle soluzioni confrontate nel capitolo precedente, sono stati ottenuti 5 modelli addestrati: uno per ogni partizione usata come set di test. Tra tutte le reti, sono state scelte le 5 relative al metodo PCA con riduzione a 3 bande, ovvero quelle che hanno conseguito il migliore punteggio medio. Queste, consistono in modelli di TensorFlow, esportati in formato SavedModel. Al fine di renderli compatibili con la libreria tlite_runtime, occorre che vengano convertiti in formato tflite. In fase di conversione possono essere applicati vari metodi di ottimizzazione post-training, con l'obiettivo di ridurre ulteriormente le dimensioni del modello e le risorse impiegate.

I principali metodi di ottimizzazione post-training sono tre [26]:

- Quantizzazione: consiste nella riduzione della precisione dei numeri utilizzati per rappresentare i parametri dei modelli, di default numeri floating point a 32 bit.
- Pruning: rimuove i parametri del modello che hanno meno influenza sulle predizioni.
- Clustering: i pesi vengono raggruppati in un certo numero di *cluster*, ogni cluster condivide un proprio valore tra tutti i pesi che gli appartengono.

Per questa applicazione non è stato sfruttato alcun metodo di ottimizzazione: il numero di parametri della rete è già di per sé estremamente limitato, così come le dimensioni delle immagini a seguito dell'applicazione della PCA con riduzione a 3 bande.

5.3 Adattamento dei codici di inferenza

L'inferenza sulla scheda Dev Board Mini è stata eseguita da due script Python, uno per tflite_runtime e uno per TensorFlow (rispettivamente Allegato 10 e Allegato 11).

Questi script sono stati ottenuti adattando il più possibile le parti corrispondenti dai Colab Notebook già impiegati e integrando dove necessario.

L'esecuzione dell'inferenza mediante la libreria tflite_runtime ha comportato una serie di modifiche dei codici originali volte a rendere le funzioni di pre-processing compatibili. La maggior parte delle operazioni matematiche e di manipolazione dei tensori presenti nelle funzioni di pre-processing venivano precedentemente eseguite con funzioni di TensorFlow. Tutte queste sono state sostituite con le equivalenti di NumPy. L'unica funzione non sostituibile è quella di ridimensionamento (tf.image.resize), in quanto non disponibile su NumPy. Esiste tuttavia la funzione "resize" della libreria OpenCV che permette di ottenere un risultato equivalente. A seguito dell'aggiornamento delle funzioni, i 5 modelli sono stati nuovamente testati, in formato SavedModel, e i punteggi ottenuti sono risultati identici a quelli originali di Tabella 4.1, confermando la correttezza delle modifiche.

In origine, in fase di inferenza, le funzioni di pre-processing venivano applicate una sola volta all'intero dataset con il metodo "map". Successivamente l'intero test set veniva passato in input alla rete che restituiva una matrice contenete tutte le predizioni. Con la suddivisione delle immagini in file singoli, è invece necessario eseguire un ciclo for che ad ogni iterazione carichi l'immagine in memoria, le applichi le funzioni di pre-processing e la fornisca in input alla rete. Analogamente, gli output vengono restituiti individualmente per ogni immagine. Bisogna quindi raggrupparli in un'unica matrice, da utilizzare per il successivo calcolo del punteggio al fine di verificare che l'inferenza sia andata a buon fine. Anche a seguito di queste modifiche sono stati ricontrollati gli *Score* per verificarne la correttezza.

All'interno del programma è stato aggiunto un cronometro, per misurare il tempo di inferenza di ciascuna immagine e quindi calcolare una media.

La struttura dei due script utilizzati è esattamente la stessa, così come il principio di funzionamento. Le uniche differenze riguardano il caricamento dei modelli: file tflite per tflite_runtime e SavedModel per TensorFlow. Anche la sintassi dei comandi per il loro utilizzo nelle due librerie è differente. Di seguito vengono riassunti gli step che gli script seguono per compiere inferenza sulla Dev Board Mini:

1. Vengono create tre liste che hanno per elementi i percorsi di salvataggio dei 5 modelli, di tutte le immagini e dei valori ground truth usati per il calcolo del

punteggio.

2. Un ciclo esterno carica nell'ordine i modelli. Per ogni sua iterazione, un ciclo più interno carica le immagini, applica le funzioni di pre-processing e registra risultati e tempi di inferenza.

3. I valori ground truth vengono caricati dai file csv e per ogni modello si calcolano i punteggi ottenuti. I punteggi vengono poi salvati in un unico file csv e sono confrontati con gli originali per verificare la correttezza del processo di inferenza.

4. I tempi di inferenza di tutte le immagini vengono salvati in 5 file csv, ognuno contenete le informazioni relative al modello corrispondete.

5.4 Risultati di inferenza

La scheda Dev Board Mini è stata alimentata in corrente continua con un alimentatore regolabile. La tensione è stata impostata a 5V mentre la corrente viene assorbita a seconda della richiesta. Registrando la corrente massima è possibile calcolare la potenza massima assorbita durante l'inferenza con la semplice equazione $P = VI$ (Tabella 5.1).

Dai tempi di inferenza di ciascuna immagine sono stai calcolati la media e i frames per second (fps) corrispondenti, ovvero le immagini che la rete riesce elaborare ogni secondo (Tabella 5.2).

Model version	Voltage [V]	Peak Current [A]	Peak Power [W]
TensorFlow	5	0.58	2.9
tflite	5	0.52	2.6

Tabella 5.1: Corrente e potenza massima assorbita.

Model version	Mean inference time [ms]	FPS
TensorFlow	464	2.1
tflite	42	23.6

Tabella 5.2: Tempi di inferenza e fps.

Dai risultati ottenuti si può dedurre che l'inferenza dei modelli in formato tflite sulla libreria tflite_runtime sia vantaggiosa rispetto ai normali modelli TensorFlow.

La velocità di inferenza è circa 11 volte superiore. Questo è notevolmente più vantaggioso da un punto di vista energetico. Supponendo che la potenza media richiesta sia la stessa in entrambi i casi, un tempo di esecuzione minore si traduce in una quantità di energia elettrica necessaria per l'intero processo inferiore. Nel caso in cui la potenza venga fornita tramite un accumulatore, per esempio in eclissi oppure quando il generatore non è in grado di soddisfare da solo tutte le richieste, significherebbe fare fronte a un requisito sulla capacità delle batterie inferiore. Questo porterebbe a una riduzione di massa a bordo.

6 | Conclusioni

Questo elaborato finale ha illustrato l'implementazione e l'ottimizzazione di reti neurali per la stima di 4 parametri del suolo da immagini iperspettrali. In particolare, sono state confrontate due architetture differenti e, a seguire, due algoritmi di riduzione delle bande, PCA e OSP, volti rispettivamente alla compressione e selezione delle informazioni più rilevanti, riducendo la memoria occupata dalle immagini. L'implementazione sul dispositivo Dev Board Mini ha dimostrato l'applicabilità delle soluzioni sviluppate in tempo reale su hardware a bassa potenza.

Gli aspetti chiave evidenziati dai risultati conseguiti sono tre:

- Dal primo confronto emerge come la rete EfficientNet-Lite0mod sia in grado di ottenere stime migliori della EdgeNeXt-XXSmod, anche a fronte di un minore numero di parametri. Questo ha oltretutto confermato la validità della soluzione proposta nel corso della competizione "Seeing Beyond the Visibile" [1].

- L'utilizzo di algoritmi di riduzione delle bande in fase di pre-processing consente di ottenere immagini di dimensioni estremamente limitate a vantaggio dei tempi di inferenza. L'eliminazione delle bande più correlate con la PCA e la selezione di quelle più informative con l'algoritmo basato sull'OSP, ha comportato in entrambi i casi una riduzione dell'errore di regressione, rispettivamente del 3.54% e dello 0.11% rispetto al modello di base, pur utilizzando circa il 7% della memoria. Ciò a dimostrazione del fatto che la totalità delle 150 bande rappresenta un numero di informazioni fin troppo elevato. Tra tutti i test effettuati, la migliore soluzione si è rivelata essere la PCA con riduzione a 3 sole bande.

- Il deployment sul dispositivo Dev Board Mini ha sottolineato l'importanza dell'utilizzo di modelli ottimizzati, ottenuti tramite conversione in formato tflite. I

tempi di inferenza si sono ridotti, di circa 11 volte, così come la massima potenza assorbita, seppur in maniera limitata. Entrambi questi fattori dimostrano l'applicabilità della soluzione per la stima direttamente a bordo di un piccolo satellite.

Un argomento non trattato in questo elaborato che può essere approfondito in futuro è l'ottimizzazione della conversione dal formato SavedModel a tflite, sfruttando i metodi citati in sezione 5.2. Si potrebbe inoltre compiere inferenza sull'acceleratore TPU di cui è dotata la scheda Dev Board Mini, confrontando i risultati con quelli di questo elaborato, ottenuti su CPU.

Bibliografia

[1] *Seeing Beyond the Visible*. URL: https://platform.ai4eo.eu/seeing-beyond-the-visible.

[2] Sebastian Szymański. *efficientnet-lite-keras*. https://github.com/sebastian-sz/efficientnet-lite-keras.

[3] Muhammad Maaz et al. *EdgeNeXt: Efficiently Amalgamated CNN-Transformer Architecture for Mobile Vision Applications*. 2022. DOI: 10.48550/ARXIV.2206.10589. URL: https://arxiv.org/abs/2206.10589.

[4] Qian Du e He Yang. «Similarity-Based Unsupervised Band Selection for Hyperspectral Image Analysis». In: *IEEE Geoscience and Remote Sensing Letters* 5.4 (2008), pp. 564–568. DOI: 10.1109/LGRS.2008.2000619.

[5] The MathWorks, Inc. URL: https://it.mathworks.com/help/images/ref/hypercube.selectbands.html.

[6] Rachit Kumar Agrawal. *Difference between Machine Learning, Deep Learning and Artificial Intelligence*. Medium. 2018. URL: https://medium.com/@UdacityINDIA/difference-between-machine-learning-deep-learning-and-artificial-intelligence-e9073d43a4c3.

[7] Shruti M. *Discover the Differences Between AI vs. Machine Learning vs. Deep Learning*. Simplilearn. 2022. URL: https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/ai-vs-machine-learning-vs-deep-learning.

[8] Antonia Russo e Gianluca Lax. «Using Artificial Intelligence for Space Challenges: A Survey». In: *Applied Sciences* 12.10 (2022). ISSN: 2076-3417. DOI: 10.3390/app12105106. URL: https://www.mdpi.com/2076-3417/12/10/5106.

[9] *Reinforcement Learning with MATLAB*. The MathWorks, Inc. URL: https://www.mathworks.com/content/dam/mathworks/ebook/gated/reinforcement-learning-ebook-all-chapters.pdf.

[10] *Overfitting and underfitting*. Educative, Inc. URL: https://www.educative.io/answers/overfitting-and-underfitting.

[11] Prakhar Ganesh. *Types of Convolution Kernels : Simplified*. Medium. 2019. URL: https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37.

[12] Rikiya Yamashita et al. «Convolutional neural networks: an overview and application in radiology». In: *Insights into Imaging* 9.4 (ago. 2018), pp. 611–629. ISSN: 1869-4101. DOI: 10.1007/s13244-018-0639-9. URL: https://doi.org/10.1007/s13244-018-0639-9.

[13] Jiwon Jeong. *The Most Intuitive and Easiest Guide for Convolutional Neural Network*. Medium. 2019. URL: https://towardsdatascience.com/the-most-intuitive-and-easiest-guide-for-convolutional-neural-network-3607be47480.

[14] *Getting Started with Hyperspectral Image Processing*. The MathWorks, Inc. URL: https://it.mathworks.com/help/images/getting-started-with-hyperspectral-image-analysis.html.

[15] Zohaib Khan, Faisal Shafait e Ajmal S. Mian. «Towards Automated Hyperspectral Document Image Analysis». In: *AFHA*. 2013.

[16] Peg Shippert. «Introduction to Hyperspectral Image Analysis». In: *Online Journal of Space Communication: Vol. 2 : Iss. 3 , Article 8* (2003).

[17] Craig Rodarmel e Jie Shan. «Principal Component Analysis for Hyperspectral Image Classification». In: *Surv Land inf Syst* 62 (gen. 2002).

[18] Rebecca Patro. *Cross-Validation: K Fold vs Monte Carlo*. Medium. 2021. URL: https://towardsdatascience.com/cross-validation-k-fold-vs-monte-carlo-e54df2fc179b.

[19] Mingxing Tan e Quoc V. Le. «EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks». In: (2019). DOI: 10.48550/ARXIV.1905.11946. URL: https://arxiv.org/abs/1905.11946.

[20] Paul Gavrikov. *visualkeras*. https://github.com/paulgavrikov/visualkeras. 2020.

[21] Mingxing Tan et al. «MnasNet: Platform-Aware Neural Architecture Search for Mobile». In: (2018). DOI: 10.48550/ARXIV.1807.11626. URL: https://arxiv.org/abs/1807.11626.

[22] Nick Steinmetz. *npy-matlab*. https://github.com/kwikteam/npy-matlab.

[23] Google. URL: https://coral.ai/products/dev-board-mini/.

[24] Justin Goodwill et al. «NASA SpaceCube Edge TPU SmallSat Card for Autonomous Operations and Onboard Science-Data Analysis». In: *Proceedings of the Small Satellite Conference*. SSC21-VII-08. AIAA. 2021.

[25] Google. URL: https://www.tensorflow.org/lite/guide#1_generate_a_tensorflow_lite_model.

[26] Google. URL: https://www.tensorflow.org/lite/performance/model_optimization#types_of_optimization.

Allegati

Allegato 1: Funzioni per il caricamento e decoding dei file TFRecord.

```
import tensorflow as tf

def load_tf_records(filepath):
    filenames = tf.io.gfile.glob(filepath)
    dataset = tf.data.TFRecordDataset(filenames,num_parallel_reads=tf.
 ↪data.experimental.AUTOTUNE)
    return dataset

def tf_records_file_features_description():
    image_feature_description = {

        'image/height': tf.io.FixedLenFeature([], tf.int64),
        'image/width': tf.io.FixedLenFeature([], tf.int64),
        'image': tf.io.FixedLenFeature([],tf.string),
        'label/P': tf.io.FixedLenFeature([], tf.float32),
        'label/K': tf.io.FixedLenFeature([], tf.float32),
        'label/Mg': tf.io.FixedLenFeature([], tf.float32),
        'label/Ph': tf.io.FixedLenFeature([], tf.float32),
    }
    return image_feature_description

def decode_dataset(example_proto):
    features=tf.io.parse_single_example(example_proto,␣
 ↪tf_records_file_features_description())

    image=features['image']
    height=features['image/height']
    width=features['image/width']
    image=tf.io.decode_raw(image,tf.int16)
```

```
    image=tf.reshape(image,[height,width,150])

    P=features['label/P']
    K=features['label/K']
    Mg=features['label/Mg']
    Ph=features['label/Ph']

    height=features['image/height']
    width=features['image/width']

    label=[P,K,Mg,Ph]

    return image, label, height, width
```

Allegato 2: Funzioni di pre-processing per il dataset di training.

```
import tensorflow as tf

target_image_size = 32

# Function to normalize the train dataset
def normalize_train(image,label, height, width, std_vec,␣
 ↪max_gt_values):

    image = tf.cast(image,tf.float32)
    image = image/std_vec
    label /= max_gt_values

    return image, label, height, width

#Function that repeats a patch horizontally with random flips until␣
 ↪the target width is reached, the result is a horizontal band
def make_horizontal_patch_train(patch, nx):

    hor_patch = tf.image.random_flip_left_right(tf.image.
 ↪random_flip_up_down(patch, seed=72), seed=64)
    i=0

    def cond(hor_patch, patch, i, nx):
        return tf.less(i, nx-1)
    def body(hor_patch, patch, i, nx):
        hor_patch = tf.concat([hor_patch, tf.image.
 ↪random_flip_left_right(tf.image.random_flip_up_down(patch,␣
 ↪seed=49), seed=95)], 1)
        i+=1
```

```
        return hor_patch, patch, i, nx

    hor_patch, _, _, _ = tf.while_loop(cond, body, [hor_patch, patch,␣
 ↪i, nx])

    return hor_patch

# Function that repeats small images applying random flips to create␣
 ↪a bigger image, it works by stacking together multiple generated␣
 ↪horizontal bands
def image_repetition_train(image, height, width):

    nx = tf.math.floordiv(target_image_size, tf.cast(width, tf.int32))
    ny = tf.math.floordiv(target_image_size, tf.cast(height, tf.
 ↪int32))
    patch = image
    image = make_horizontal_patch_train(patch, nx)
    i=0

    def cond(image, patch, i, ny):
        return tf.less(i, ny-1)
    def body(image, patch, i, ny):
        image = tf.concat([image, make_horizontal_patch_train(patch,␣
 ↪nx)], 0)
        i+=1
        return image, patch, i, ny

    image, _, _, _= tf.while_loop(cond, body, [image, patch, i, ny])
```

```
    image = tf.cond(tf.math.maximum(nx, ny)==1, lambda: tf.image.
 ↪resize(image, [target_image_size, target_image_size],␣
 ↪method='bilinear', antialias=False), lambda: tf.image.
 ↪pad_to_bounding_box(image, 0, 0, target_image_size,␣
 ↪target_image_size))

    return image

# Function to resize images to [target_image_size,␣
 ↪target_image_size, 150] with the desired augmentation methods
def pad_with_patches_train(image, label, height, width):

    max_dim = tf.math.maximum(height, width)
    image = tf.cond(max_dim<target_image_size, lambda:␣
 ↪image_repetition_train(image, height, width), lambda: tf.image.
 ↪resize(image, [target_image_size, target_image_size],␣
 ↪method='bilinear', antialias=False))
    image = add_gauss_noise(image)

    index=tf.random.uniform(shape=[],maxval=4,dtype=tf.dtypes.
 ↪int32,seed=32)

    image=tf.cond(tf.equal(index,1),lambda: tf.image.
 ↪flip_left_right(image), lambda: image)

    image=tf.cond(tf.equal(index,2),lambda: tf.image.
 ↪flip_up_down(image), lambda: image)

    image=tf.cond(tf.equal(index,3),lambda: tf.image.
 ↪flip_left_right(tf.image.flip_up_down(image)), lambda: image)
```

```
    return image, label

# Function to add gaussian noise
def add_gauss_noise(image):

    mean = 0
    std = 0.05

    gauss = tf.random.normal([target_image_size, target_image_size,␣
 ↪150], mean, std, seed=910)

    noisy = image + gauss

    return noisy
```

Allegato 3: Funzioni di pre-processing per il dataset di test.

```
[ ]: import tensorflow as tf

target_image_size = 32

# Function to normalize the test dataset
def normalize_test(image, label, height, width, std_vec,␣
 ↪max_gt_values):
    image = tf.cast(image,tf.float32)
    image = image/std_vec
    label = label/max_gt_values

    return image, label, height, width

# Function that repeats small images to create a bigger image
def image_repetition_test(image, height, width):
    nx = tf.math.floordiv(target_image_size, tf.cast(width, tf.int32))
    ny = tf.math.floordiv(target_image_size, tf.cast(height, tf.
 ↪int32))

    image = tf.tile(image, [ny, nx, 1])
    image = tf.cond(tf.math.maximum(nx, ny)==1, lambda: tf.image.
 ↪resize(image, [target_image_size, target_image_size],␣
 ↪method='bilinear', antialias=False), lambda: tf.image.
 ↪pad_to_bounding_box(image, 0, 0, target_image_size,␣
 ↪target_image_size))

    return image

# Function to resize images to␣
 ↪[target_image_size,target_image_size,150]
```

```
def pad_with_patches_test(image, label, height, width):
    max_dim = tf.math.maximum(height, width)
    image = tf.cond(max_dim<target_image_size, lambda:␣
 ↪image_repetition_test(image, height, width), lambda: tf.image.
 ↪resize(image, [target_image_size, target_image_size],␣
 ↪method='bilinear', antialias=False))

    return image, label
```

Allegato 4: Pre-processing del training set.

```
import tensorflow as tf

total_num_images = 1732
num_images = [346, 346, 346, 347, 347]
batch_size = 32
AUTO = tf.data.AUTOTUNE
tf_records_path = '/content/split_{}.record'



# Any number in [1,5]
test_partition_id = 5
num_train_images = total_num_images - num_images[test_partition_id-1]

# Concatenate partitions to form the train set, the partition chosen␣
 ↪as the test partition is excluded
splits = [load_tf_records(tf_records_path.format(i)).
 ↪map(decode_dataset, num_parallel_calls=tf.data.AUTOTUNE) for i in␣
 ↪range(1,6)]
splits.pop(test_partition_id-1)

ds = splits[0]

for partition in splits[1:]:
    ds = ds.concatenate(partition)

# Normalize, shuffle, pad, batch and prefetch training dataset
train_data = ds.take(1376)
```

```
train_data = train_data.map(lambda image, label, height, width: ␣
 ↪normalize_train(image, label, height, width, std_vec, ␣
 ↪max_gt_values), num_parallel_calls=AUTO).cache()

train_data = train_data.shuffle(num_train_images, seed=1866)

train_data = train_data.map(lambda image, label, height, width: ␣
 ↪pad_with_patches_train(image, label, height, width), ␣
 ↪num_parallel_calls=AUTO)

train_data = train_data.batch(batch_size=batch_size, ␣
 ↪drop_remainder=True)

train_data = train_data.prefetch(AUTO)

# Normalize, shuffle, pad, batch and prefetch validation dataset
val_data = ds.skip(1376)

val_data = val_data.map(lambda image, label, height, width: ␣
 ↪normalize_train(image, label, height, width, std_vec, ␣
 ↪max_gt_values), num_parallel_calls=AUTO).cache()

val_data = val_data.map(lambda image, label, height, width: ␣
 ↪pad_with_patches_train(image, label, height, width), ␣
 ↪num_parallel_calls=AUTO)

val_data = val_data.batch(batch_size=10, drop_remainder=False)

val_data = val_data.prefetch(AUTO)
```

Allegato 5: Pre-processing del test set.

```
import tensorflow as tf

batch_size = 32
AUTO = tf.data.AUTOTUNE
tf_records_path = '/content/split_{}.record'

# Any number in [1,5]
test_partition_id = 5

# Load test dataset based on chosen partition
test_data = load_tf_records(tf_records_path.
 ↪format(test_partition_id)).map(decode_dataset,␣
 ↪num_parallel_calls=AUTO)

# Normalize, pad, cache, batch and prefetch test dataset
test_data = test_data.map(lambda image, label, height, width:␣
 ↪normalize_test(image, label, height, width, std_vec,␣
 ↪max_gt_values), num_parallel_calls=AUTO)

test_data = test_data.map(lambda image, label, height, width:␣
 ↪pad_with_patches_test(image, label, height, width),␣
 ↪num_parallel_calls=AUTO).cache()

test_data = test_data.batch(batch_size=batch_size,␣
 ↪drop_remainder=False).prefetch(AUTO)
```

Allegato 6: Calcolo della matrice di covarianza e della deviazione standard.

```
import tensorflow as tf
import numpy as np

# Define functions to calculate the covariance matrix by taking into
 ↪account all of the images at the same time.

# Arrange the data in the pixel matrix and remove all zero pixels
def prepare_data(image, label, height, width):

    data = tf.cast(tf.reshape(image, [height*width, 150]),
 ↪dtype='float64')
    data = tf.transpose(data)

    lg_values = data!=0
    mask = tf.math.reduce_any(lg_values, 0)
    data = tf.boolean_mask(data, mask, 1)

    return data

# Sum all pixel values of an image for each channel, required in
 ↪order to calculate the average
def sum_vector(data):

    mean = tf.math.reduce_sum(data, 1, keepdims=True)

    return mean

# Calculate dot product of the pixel matrix for each image
def dot_product(data):
```

```
    cov = tf.matmul(data, data, transpose_b=True)

    return cov

# Normalize by std
def normalize(data, std):

    data = data/std

    return data

# Std calcuation
def std(data, mean):

    std = tf.math.reduce_sum((data-mean)**2, 1, keepdims=True)

    return std
```

```
# Find the covariance matrix

data = full_dataset.map(prepare_data) # Here full dataset is the␣
 ↪complete dataset of images given as the tf.data.Dataset class

total_pixels_per_channel = data.map(lambda data: tf.cast(tf.
 ↪shape(data)[1], dtype='float64')).reduce(np.float64(0), lambda x, y:
 ↪ x+y)

# Find average reflectance to calculate standard deviation
sum_vec = data.map(sum_vector).reduce(tf.zeros([150,1],␣
 ↪dtype='float64'), lambda x, y: x+y)
```

```
avg_vec = sum_vec/total_pixels_per_channel

# Find standard deviation for standardization
sigma_vec = data.map(lambda data: std(data, avg_vec)).reduce(np.
 ↪float64(0), lambda x, y: x+y)
sigma_vec = tf.math.sqrt(sigma_vec/total_pixels_per_channel)*10

# Apply normalization
data = data.map(lambda data: normalize(data, sigma_vec))

# Find the average vector for the covariance matrix calculation
sum_vec = data.map(sum_vector).reduce(tf.zeros([150,1],␣
 ↪dtype='float64'), lambda x, y: x+y)
avg_vec = sum_vec/total_pixels_per_channel

# Calculate convaraince matrix
cov_ds = data.map(dot_product).reduce(tf.zeros([150,150],␣
 ↪dtype='float64'), lambda x, y: x+y)
cov_mat = cov_ds/total_pixels_per_channel - tf.matmul(avg_vec,␣
 ↪avg_vec, transpose_b=True)

np.save('/content/covariance_matrix.npy', cov_mat.numpy())
np.save('/content/std_vector.npy', sigma_vec.numpy())
```

Allegato 7: Funzioni di pre-processing modificate o introdotte per l'applicazione della PCA.

```
import tensorflow as tf

# Function to reshape and normalize the test dataset
def reshape_and_normalize(image, label, height, width, std_vec,␣
 ↪max_gt_values):

    data = tf.reshape(image, [height*width, 150])
    data = tf.cast(data, dtype='float64')
    data = tf.transpose(data)

    data = data/std_vec
    label = label/max_gt_values

    return data, label, height, width

# Function to add gaussian noise. Note that the std is not constant␣
 ↪anymore but depends on the mean value for each band
def add_gauss_noise(image, n_pca_bands):

    mean = 0
    std = tf.reduce_mean(image, [0, 1])*0.3

    gauss = tf.random.normal([target_image_size, target_image_size,␣
 ↪n_pca_bands], mean, std, seed=910)

    noisy = image + gauss

    return noisy
```

```
# Function to apply PCA
def apply_pca(data, label, height, width, n_pca_bands, cov):
 ↪#apply pca

    [eig_values, eig_vectors] = tf.linalg.eigh(cov)
    eig_vectors = eig_vectors[:, 150-n_pca_bands:]

    z = tf.matmul(eig_vectors, data, transpose_a=True)
    z = tf.transpose(z)
    image = tf.reshape(z, [height, width, n_pca_bands])
    image = tf.cast(image, dtype='float32')

    return image, label, height, width

# Function to normalize after PCA
def post_pca_normalization(image, label, height, width,
 ↪max_reflectance_PCA):

    image = image/max_reflectance_PCA

    return image, label, height, width
```

Allegato 8: MATLAB Live Script per l'analisi delle bande più informative.

[ ]:
```
clc, clear;
```

Load wavelength csv file and define train set path.

[ ]:
```
wavelength = readtable('wavelengths.csv');
wavelength = table2array(wavelength(:,2));
images_filepath = 'D:\Documenti\AKO\UNI_AERO\TIROCINIO␣
↪TESI\band_selection_analysis_matlab\train_data_npy\*.npy';
```

Create Datastore of hsi images and check exactness of procedure.

[ ]:
```
hsi_ds = fileDatastore(images_filepath, 'ReadFcn', @readNPY);
hsi_ds = transform(hsi_ds, @(x) hypercube(x, wavelength));
sample_image = preview(hsi_ds);
imshow(colorize(sample_image, 'Method', "rgb"),␣
↪'InitialMagnification', 3000);
```

Find most informative bands for each image.

[ ]:
```
num_train_images = 1732;
selected_bands = cell(num_train_images, 1);
i = 1;

reset(hsi_ds)

while hasdata(hsi_ds)
    hcube = read(hsi_ds);
    num_endmembers = countEndmembersHFC(hcube);
    endmembers = fippi(hcube, num_endmembers);
    [new_cube, bands] = selectBands(hcube, endmembers);
    selected_bands{i} = bands;
    i = i+1;
```

```
end
```

Show most informative bands for the whole dataset.

[ ]:
```
selected_bands_vec = cell2mat(selected_bands);
[count, bands] = groupcounts(selected_bands_vec);
[count, index] = sort(count, 'descend');
bands = bands(index);
figure(1)
histogram(selected_bands_vec, 'BinMethod', 'integers');


xlim([-2.0 153.0])
ylim([0 1400])
xlabel('Bands')
ylabel('Frequency')
title('Distribution of most informative bands')
set(gcf,'Position', [0 0 1000 300])
```

Save sorted informative band vector to npy file.

[ ]:
```
writeNPY(bands, 'informative_bands.npy')
```

Allegato 9: Funzione per la selezione delle bande più informative dal data cube.

```
import tensorflow as tf

band_index = informative_bands[:n_bands]

def select_bands(image, label, height, width, band_index):
    image = tf.gather(image, band_index, axis = 2)

    return image, label, height, width
```

Allegato 10: Script per inferenza su tflite_runtime.

```
import os
import numpy as np
import tflite_runtime.interpreter as tf_lite
import time
import cv2
import pandas as pd

import sys
sys.path.append('/home/usr/local/lib/python3.7/dist-packages')

# Define paths
cov_mat_path = '/home/mendel/sd/tflite/preprocessing_values/
 ↪covariance_matrix.npy'
std_vec_path = '/home/mendel/sd/tflite/preprocessing_values/
 ↪std_vector.npy'
dataset = '/home/mendel/sd/tflite/dataset/'
models = '/home/mendel/sd/tflite/TF_lite_models/'
ground_truth = '/home/mendel/sd/tflite/ground_truth/'

# Create lists to load all test images and models
num_images=[346,346, 346, 347,347]
img_dir=[[],[],[],[],[]]
models_dir=[]
gt_dir=[]
i=-1
for test_dir in sorted(os.listdir(dataset)):
    i+=1
    test_dir = os.path.join(dataset, test_dir)
    print(test_dir)
```

```
    for k in range(num_images[i]):
        full_path = test_dir + '/' + str(k) + '.npy'
        img_dir[i].append(full_path)

for path in sorted(os.listdir(models)):
    full_path = os.path.join(models, path)
    models_dir.append(full_path)
    print(full_path)

for path in sorted(os.listdir(ground_truth)):
    full_path = os.path.join(ground_truth, path)
    gt_dir.append(full_path)
    print(full_path)

# Preprocessing values
max_reflectance_PCA = 6.282844
max_gt_values = [325, 625, 400, 14]
mse_baseline = [8.7002814e+02, 3.8284080e+03, 1.5888525e+03, 6.
 ↪7716144e-02]
n_pca_bands = 3
target_image_size = 32

cov = np.load(cov_mat_path)
std_vec = np.load(std_vec_path)


# Preprocessing functions
def reshape_and_normalize(image, std_vec, height, width):

    data = np.reshape(image, [height*width, 150])
```

```
    data = np.transpose(data)

    data = data/std_vec

    return data

def eig_decomposition(cov, n_pca_bands):

    [eig_values, eig_vectors] = np.linalg.eigh(cov, 'U')
    eig_vectors = eig_vectors[:, 150-n_pca_bands:]

    return eig_vectors

def apply_pca(data, n_pca_bands, eig_vectors, height, width):

    z = np.matmul(eig_vectors.T, data)
    z = np.transpose(z)
    image = np.reshape(z, [height, width, n_pca_bands])

    return image

def image_repetition_test(image, height, width):
    nx = np.floor_divide(target_image_size, width)
    ny = np.floor_divide(target_image_size, height)

    image = np.tile(image, [ny, nx, 1])

    if np.maximum(nx, ny)==1:
        image = cv2.resize(image, [target_image_size,␣
↪target_image_size], interpolation=cv2.INTER_LINEAR)
    else:
```

```
        [height, width] = np.shape(image)[:2]
        image = np.pad(image,␣
 ↪((0,target_image_size-height),(0,target_image_size-width),(0,0)),␣
 ↪'constant', constant_values=0)
    return image

def pad_with_patches_test(image, height, width):
    max_dim = np.maximum(height, width)

    if max_dim < target_image_size:
        image = image_repetition_test(image, height, width)
    else:
        image = cv2.resize(image, [target_image_size,␣
 ↪target_image_size], interpolation=cv2.INTER_LINEAR)

    return image

def post_pca_normalization(image, max_reflectance_PCA):

    image = image/max_reflectance_PCA

    return image

# Eigen decoposition of covariance matrix
eig_vectors = eig_decomposition(cov, n_pca_bands)

# Initialize variables to export data
predictions = [np.zeros((num_test_images, 4)) for num_test_images in␣
 ↪num_images]
image_loading_time = [np.zeros((num_test_images, 4)) for␣
 ↪num_test_images in num_images]
```

```
image_overall_time = [np.zeros((num_test_images, 4)) for␣
 ↪num_test_images in num_images]
network_inference_time = [np.zeros((num_test_images, 4)) for␣
 ↪num_test_images in num_images]
score = np.zeros((1,5))

test_id=-1
for test_data, model in zip(img_dir, models_dir):
    test_id += 1
    print(test_id)
    i=-1

    # Load model
    network=tf_lite.Interpreter(model, num_threads=4)
    network.allocate_tensors()
    network.invoke() # warmup
    network_input_details = network.get_input_details()
    network_output_details = network.get_output_details()

    for image_path in test_data:
        i += 1
        image_time_start = time.time()  # Time start

        image = np.load(image_path)

        image_loading_time[test_id][i,0] = time.time() -␣
 ↪image_time_start   # Loading time

        h = np.shape(image)[0]
        w = np.shape(image)[1]
```

```
        # Preprocessing
        image = reshape_and_normalize(image, std_vec, h, w)
        image = apply_pca(image, n_pca_bands, eig_vectors, h, w)
        image = pad_with_patches_test(image, h, w)
        image = post_pca_normalization(image, max_reflectance_PCA)
        image = np.expand_dims(image, 0)
        image = np.float32(image)

        # Predictions
        inferece_time_start = time.time()    # Inference time start

        network.set_tensor(network_input_details[0]['index'], image)
        network.invoke()
        pred = network.get_tensor(network_output_details[0]['index'])

        network_inference_time[test_id][i,0] = time.time() -␣
↪inferece_time_start    # Inference time

        predictions[test_id][i,:] = pred

        image_overall_time[test_id][i,0] = time.time() -␣
↪image_time_start    # Overall time

# Define custom metric for evaluation
def custom_metric(y_true, y_pred, max_gt_values):

    y_pred = np.multiply(y_pred, max_gt_values)
    mse = np.mean((y_true-y_pred)**2, axis=0)
    score = np.mean(mse/mse_baseline)

    return score
```

```
# Export info
base_path = '/home/mendel/sd/tflite/results/TF_lite_results/'

i=-1
for pred, gt_path in zip(predictions, gt_dir):
    i += 1
    gt = np.array(pd.read_csv(gt_path))[:,1:]
    score[0,i] = custom_metric(gt, pred, max_gt_values)

print(score)

score_df = pd.DataFrame(score.T, columns=['score'])
score_df.to_csv(os.path.join(base_path, 'score.csv'),␣
↪index_label='Test set index')

i=0
for ilt, iot, nit in zip(image_loading_time, image_overall_time,␣
↪network_inference_time):
    i+=1
    times = pd.DataFrame(np.concatenate((ilt, nit, iot), axis = 1),␣
↪columns=['image loading time','network inference time','image␣
↪overall time'])
    times.to_csv(os.path.join(base_path, 'times_'+str(i)+'.csv'),␣
↪index_label='Image index')
```

Allegato 11: Script per inferenza su TensorFlow.

```
import os
import numpy as np
import tensorflow as tf
from tensorflow import keras
import time
import cv2
import pandas as pd

import sys
sys.path.append('/home/usr/local/lib/python3.7/dist-packages')

# Define paths
cov_mat_path = '/home/mendel/sd/tflite/preprocessing_values/
 ↪covariance_matrix.npy'
std_vec_path = '/home/mendel/sd/tflite/preprocessing_values/
 ↪std_vector.npy'
dataset = '/home/mendel/sd/tflite/dataset/'
models = '/home/mendel/sd/tflite/TF_models/'
ground_truth = '/home/mendel/sd/tflite/ground_truth/'

# Create lists to load all test images and models
num_images=[346,346, 346, 347,347]
img_dir=[[],[],[],[],[]]
models_dir=[]
gt_dir=[]
i=-1
for test_dir in sorted(os.listdir(dataset)):
    i+=1
    test_dir = os.path.join(dataset, test_dir)
```

```
    print(test_dir)

    for k in range(num_images[i]):
        full_path = test_dir + '/' + str(k) + '.npy'
        img_dir[i].append(full_path)

for path in sorted(os.listdir(models)):
    full_path = os.path.join(models, path)
    models_dir.append(full_path)
    print(full_path)

for path in sorted(os.listdir(ground_truth)):
    full_path = os.path.join(ground_truth, path)
    gt_dir.append(full_path)
    print(full_path)

# Preprocessing values
max_reflectance_PCA = 6.282844
max_gt_values = [325, 625, 400, 14]
mse_baseline = [8.7002814e+02, 3.8284080e+03, 1.5888525e+03, 6.
 ↪7716144e-02]
n_pca_bands = 3
target_image_size = 32

cov = np.load(cov_mat_path)
std_vec = np.load(std_vec_path)


# Preprocessing functions
def reshape_and_normalize(image, std_vec, height, width):
```

```
    data = np.reshape(image, [height*width, 150])
    data = np.transpose(data)

    data = data/std_vec

    return data

def eig_decomposition(cov, n_pca_bands):

    [eig_values, eig_vectors] = np.linalg.eigh(cov, 'U')
    eig_vectors = eig_vectors[:, 150-n_pca_bands:]

    return eig_vectors

def apply_pca(data, n_pca_bands, eig_vectors, height, width):

    z = np.matmul(eig_vectors.T, data)
    z = np.transpose(z)
    image = np.reshape(z, [height, width, n_pca_bands])

    return image

def image_repetition_test(image, height, width):
    nx = np.floor_divide(target_image_size, width)
    ny = np.floor_divide(target_image_size, height)

    image = np.tile(image, [ny, nx, 1])

    if np.maximum(nx, ny)==1:
        image = cv2.resize(image, [target_image_size,␣
↪target_image_size], interpolation=cv2.INTER_LINEAR)
```

```
    else:
        [height, width] = np.shape(image)[:2]
        image = np.pad(image,␣
 ↪((0,target_image_size-height),(0,target_image_size-width),(0,0)),␣
 ↪'constant', constant_values=0)
    return image

def pad_with_patches_test(image, height, width):
    max_dim = np.maximum(height, width)

    if max_dim < target_image_size:
        image = image_repetition_test(image, height, width)
    else:
        image = cv2.resize(image, [target_image_size,␣
 ↪target_image_size], interpolation=cv2.INTER_LINEAR)

    return image

def post_pca_normalization(image, max_reflectance_PCA):

    image = image/max_reflectance_PCA

    return image

# Eigen decoposition of covariance matrix
eig_vectors = eig_decomposition(cov, n_pca_bands)

# Initialize variables to export data
predictions = [np.zeros((num_test_images, 4)) for num_test_images in␣
 ↪num_images]
```

```
image_loading_time = [np.zeros((num_test_images, 4)) for␣
 ↪num_test_images in num_images]
image_overall_time = [np.zeros((num_test_images, 4)) for␣
 ↪num_test_images in num_images]
network_inference_time = [np.zeros((num_test_images, 4)) for␣
 ↪num_test_images in num_images]
score = np.zeros((1,5))

test_id=-1
for test_data, model in zip(img_dir, models_dir):
    test_id += 1
    print(test_id)
    i=-1

    # Load model
    loaded_model = tf.keras.models.load_model(model,␣
 ↪custom_objects=None, compile=False, options=None)

    for image_path in test_data:
        i += 1
        image_time_start = time.time()  # Time start

        image = np.load(image_path)

        image_loading_time[test_id][i,0] = time.time() -␣
 ↪image_time_start   # Loading time

        h = np.shape(image)[0]
        w = np.shape(image)[1]

        # Preprocessing
```

```
        image = reshape_and_normalize(image, std_vec, h, w)
        image = apply_pca(image, n_pca_bands, eig_vectors, h, w)
        image = pad_with_patches_test(image, h, w)
        image = post_pca_normalization(image, max_reflectance_PCA)
        image = np.expand_dims(image, 0)
        image = np.float32(image)

        # Predictions
        inferece_time_start = time.time()   # Inference time start

        pred = loaded_model.predict(image, verbose=0)

        network_inference_time[test_id][i,0] = time.time() -␣
 ↪inferece_time_start    # Inference time

        predictions[test_id][i,:] = pred

        image_overall_time[test_id][i,0] = time.time() -␣
 ↪image_time_start   # Overall time

# Define custom metric for evaluation
def custom_metric(y_true, y_pred, max_gt_values):

    y_pred = np.multiply(y_pred, max_gt_values)
    mse = np.mean((y_true-y_pred)**2, axis=0)
    score = np.mean(mse/mse_baseline)

    return score

# Export info
base_path = '/home/mendel/sd/tflite/results/TF_results_no_verbose/'
```

```
i=-1
for pred, gt_path in zip(predictions, gt_dir):
    i += 1
    gt = np.array(pd.read_csv(gt_path))[:,1:]
    score[0,i] = custom_metric(gt, pred, max_gt_values)

print(score)

score_df = pd.DataFrame(score.T, columns=['score'])
score_df.to_csv(os.path.join(base_path, 'score.csv'),␣
↪index_label='Test set index')

i=0
for ilt, iot, nit in zip(image_loading_time, image_overall_time,␣
↪network_inference_time):
    i+=1
    times = pd.DataFrame(np.concatenate((ilt, nit, iot), axis = 1),␣
↪columns=['image loading time','network inference time','image␣
↪overall time'])
    times.to_csv(os.path.join(base_path, 'times_'+str(i)+'.csv'),␣
↪index_label='Image index')
```

Ringraziamenti

Ringrazio i miei genitori, per avermi sempre supportato in tutte le scelte fino ad oggi, per avermi lasciato intraprendere liberamente tutte le esperienze che ho fatto e per il nostro rapporto di fiducia reciproca, con cui mi hanno garantito la giusta indipendenza.

Ringrazio i nonni di Carpi, nonna Edda e nonno Mario, per il loro supporto incondizionato, oltre che per il grande sostegno economico che hanno fornito in questi anni di mia permanenza a Forlì.

Ringrazio la nonna di Pisa, nonna Emma, per avermi dimostrato che non è mai troppo tardi per imparare qualcosa e avermi dato un motivo in più per laurearmi il prima possibile. Ringrazio anche il nonno Paolo, che mi ha fatto più volte sorridere ripensando agli episodi passati.

Ringrazio Sara per essermi stata vicina, nonostante le inutili brontolate post esame e periodi di studio matto e disperatissimo.

Ringrazio Matteo, che nonostante il corso di studi differente e gli incontri meno frequenti mi ha comunque stimolato nel cercare di fare meglio di lui (ahimè senza successo).

Ringrazio il Prof. Fantoni, per non avermi fatto odiare la matematica, e il Prof. Quattrini, per avermi fornito tutti gli strumenti necessari per affrontare serenamente l'università.

Infine, ringrazio il Prof. Locarini, per avermi dato la possibilità di svolgere tirocinio e tesi presso il laboratorio di microsatelliti, e Alessandro Lotti, per essere stato incredibilmente disponibile in questi mesi.