ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

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

Esplorazione dei Variational Autoencoder per l'espansione dei dataset nei contesti di osservazione terrestre

CANDIDATA:

Rebecca Uffizialetti

RELATORE:

Prof. Alfredo Locarini

CORRELATORE:

Dott. Alessandro Lotti

Anno Accademico 2023/2024

Abstract

A bordo dei satelliti, l'utilizzo di reti neurali e di tecniche che sfruttano l'Intelligenza Artificiale, permette di migliorare l'autonomia, l'efficienza e l'agilità dei sistemi che si occupano di osservazione terreste. Questo elaborato presenta un possibile approccio per l'ampliamento dei dataset utilizzati per allenare reti neurali per applicazioni di osservazione della Terra. A questo scopo viene introdotto l'utilizzo di un Variational Autoencoder (VAE), una rete neurale generativa. La validazione del modello proposto è stata condotta attraverso l'utilizzo dei dati della Challenge "Seeing Beyond the Visible", promossa dall'Agenzia Spaziale Europea (ESA), e reti sviluppate specificatamente per tale iniziativa.

Indice

Elenco delle figure

Elenco delle tabelle

Acronimi

AE Autoencoders

AI Intelligenza Artificiale

CNN Rete Neurale Convoluzionale

DL Deep Learning

ESA Agenzia Spaziale Europea

EO Earth Observation

ML Machine Learning

VAE Variational Autoencoder

Introduzione

Questo elaborato finale di tesi è stato realizzato in seguito all'attività di tirocinio curricolare, portata a termine presso il Laboratorio di Microsatelliti e Microsistemi Spaziali, incluso nel Centro Interdipartimentale di Ricerca Industriale CIRI Aerospaziale – Aerospace. Le attività di tirocinio sono state volte alla realizzazione di algoritmi, basati su reti neurali, finalizzati alla partecipazione alla competizione "Orbital AI Φsat-2" [1] promossa dall'Agenzia Spaziale Europea (ESA). L'iniziativa si impegna nel promuovere lo sviluppo di un ecosistema di applicazioni per l'osservazione della Terra attraverso l'edge computing, tramite l'utilizzo dell'Intelligenza Artificiale (AI) direttamente a bordo del satellite.

L'attività di tirocinio ha portato alla realizzazione di un algoritmo in grado di stimare il quantitativo di clorofilla contenuto in aree verdi urbane ed extra-urbane. Questo sistema consente di poter monitorare tali aree, permettendo di identificare zone che necessitano di manutenzione, consentendo di poter determinate i periodi adatti per la fertilizzazione e l'innaffiamento e permettendo di monitorare i cambiamenti nella salute delle piante, senza necessità di effettuare misurazioni in loco.

Il modello realizzato, seppur funzionante, presenta delle limitazioni non trascurabili, dovute prevalentemente alla ridotta dimensione del dataset. Il set di dati utilizzato è composto da immagini multispettrali, poiché al tempo del tirocinio il satellite non era ancora operativo, queste sono state simulate a partire da scatti ottenuti grazie alla missione Sentinel [2].

Il lavoro di tirocinio ha evidenziato che, seppure i satelliti di osservazione della Terra ad oggi producano enormi moli di immagini, spesso queste risultano sub-ottime o mancano delle informazioni necessarie alla risoluzione di problemi specifici. Questo aspetto limita fortemente la possibilità di validare estensivamente nuovi algoritmi data-driven prima della missione per cui sono stati progettati.

Il lavoro di tesi si propone si investigare l'utilizzo di modelli generativi al fine di aumentare fittiziamente le dimensioni dei dataset per problemi di osservazione della Terra. A tal proposito è stata sviluppata una rete neurale basata su un Autoencoder

Variazionale (Variational Autoencoder, VAE), questa ha l'obiettivo di apprendere le caratteristiche delle (limitate) immagini reali e di crearne di nuove.

Per poter testare il funzionamento del VAE si è fatto uso del dataset fornito per la competizione "Seeing Beyond the Visible" [3] organizzata da KP Labs, Agenzia Spaziale Europea (ESA) e QZ Solutions, nell'ambito della "IEEE International Conference on Image Processing (ICIP) 2022", ed in seguito delle reti sviluppate da Achille Ballabeni, Alessandro Lotti, Alfredo Locarini, Dario Modenini e Paolo Tortora per tale Challenge [4].

I codici sviluppati sono stati scritti in linguaggio Python all'interno di diversi Jupyter Notebook. Le reti sono state redatte tramite l'API Python di TensorFlow Keras. Gli script principali sono allegati a fine elaborato.

Lo scritto è organizzato come segue:

- Il Capitolo 1 fornisce nozioni sulle reti neurali ed un'analisi approfondita delle reti neurali convoluzionali e degli autoencoders.
- Il Capitolo 2 si focalizza sulle tipologie di immagini utilizzate nell'osservazione terrestre e le relative problematiche.
- Il Capitolo 3 illustra l'implementazione delle reti neurali sviluppate.
- Il Capitolo 4 presenta i risultati ottenuti attraverso il testing e l'analisi dei diversi addestramenti effettuati.
- Il Capitolo 5 contiene le conclusioni del lavoro svolto ed illustra limitazioni e sviluppi futuri.

1 Reti Neurali e AI: Fondamenti ed Applicazioni

1.1 Intelligenza Artificiale, Machine Learning e Deep Learning

Prima di entrare nel merito delle reti neurali realizzate è necessario presentare i vari termini e concetti che si trovano dietro questo specifico ambito dell'informatica.

In generale si può pensare al Deep Learning (DL) come un sottoinsieme del Machine Learning (ML) che a sua volta risulta essere un sottogruppo dell'Intelligenza Artificiale (AI).

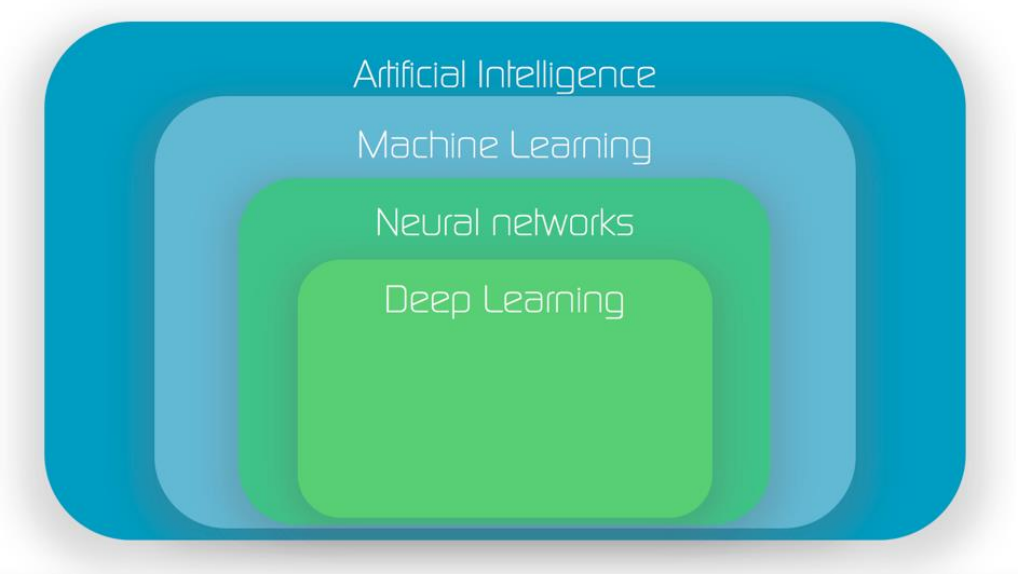

Figura 1.1: Diagramma dei sottogruppi dell'AI [11].

Con il termine Intelligenza Artificiale si fa riferimento ad un'ampia branca dell'informatica che si focalizza nell'automatizzare compiti che vengono normalmente compiuti da persone e che quindi hanno solitamente bisogno della presenza della "intelligenza" umana [5].

Il Machine Learning, come già detto, è un sottogruppo dell'AI, che comprende algoritmi in grado di adattarsi in maniera automatica dall'esperienza; gli algoritmi di ML non necessitano di utilizzare equazioni predeterminate poiché per apprendere si servono di metodi computazionali. Sono specializzati nella risoluzione di compiti molto complessi dove si richiede un'identificazione di pattern, come ad esempio il riconoscimento vocale

o la classificazione di immagini [5]. Con modelli di ML è possibile addestrare una macchina utilizzando set di dati importanti a livello dimensionale. Tendenzialmente, all'aumentare della mole di dati che vengono analizzati corrisponde un miglioramento dell'algoritmo [6].

L'ultimo sottoinsieme è il Deep Learning, i modelli di DL si basano su reti neurali, queste sono composte da nodi interconnessi in una struttura stratificata che mette in relazione gli input agli output desiderati. I nodi tra i livelli di input ed i livelli di output sono definiti come *livelli nascosti* (hidden layers), nel momento in cui la struttura della rete è composta da molti hidden layers allora questa viene definita rete neurale profonda e rientra nella categoria del Deep Learning, in tal caso si parla di Deep Neural Networks.

Il Teorema di Approssimazione Universale (Universal Approximation Theorem, UAT) afferma che le reti neurali con almeno un layer nascosto aventi un numero sufficiente di neuroni ed una funzione di attivazione non lineare possono approssimare una qualunque funzione continua ad un livello arbitrario di accuratezza [7]. Tale concetto torna utile quando si vuol comprendere il Deep Learning; infatti, nella pratica questo teorema dimostra che una rete neurale con sufficienti capacità, anche se poco profonda, è in grado di rappresentare in maniera accurata una qualsiasi funzione monodimensionale e continua, all'interno di un intervallo specificato; questo si ottiene aumentando il numero di unità nascoste che aggiungono più regioni non lineari alla funzione. Quando queste regioni diventano più numerose e di dimensione minore si avvicinano, con precisione crescente, a sezioni sempre più piccole della funzione. Questo aspetto dell'UAT è fondamentale in compiti complessi che richiedono l'individuazione di dettagli da identificare in grandi quantità di dati, tali compiti sono quelli svolti dagli algoritmi di DL.

1.2 Reti Neurali

Le reti neurali sono una tipologia di algoritmi ispirati al funzionamento del cervello umano. Queste sono infatti costituite da nodi o neuroni, che compiono semplici operazioni non lineari sugli input. Quando lo strato di nodi intermedi è uno solo la rete viene definita semplice, nel momento in cui gli hidden layers sono più di uno la rete è detta multistrato e rientra nell'ambito del Deep Learning (Figura 1.2).

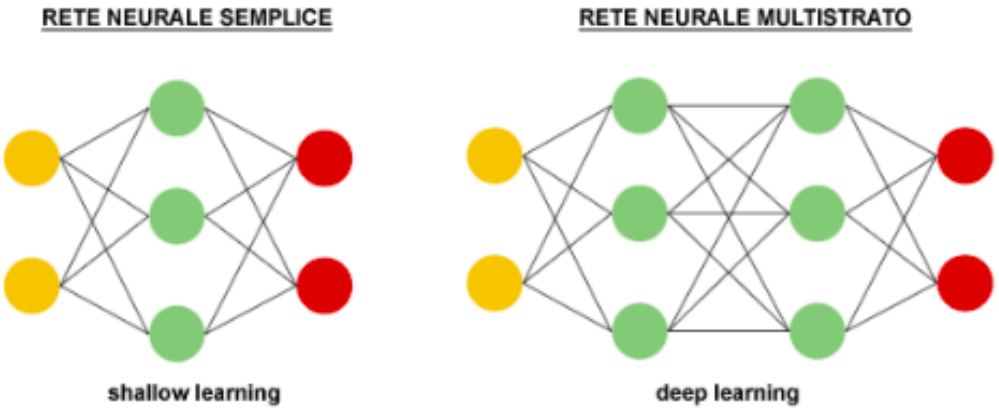

Figura 1.2: Schema di reti neurali semplici e multistrato [13].

I nodi dei diversi livelli sono connessi tramite una fitta rete di collegamenti attraverso i quali fluiscono i dati. Le reti neurali, a valle di un procedimento definito di addestramento (o training) sono in grado di riconoscere pattern complessi all'interno dei dati numerici che vengono forniti in input [8].

Qualora ogni nodo della rete sia connesso a tutti i nodi dello stratto vicino allora la rete viene definita *completamente connessa*, nel momento in cui ciò non è verificato si parla di rete *parzialmente connessa* (Figura 1.3).

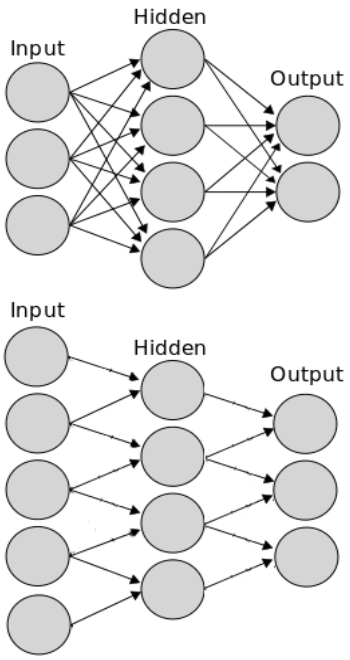

Figura 1.3: Schema di una rete completamente connessa (sopra) e parzialmente connessa (sotto) [12].

Alcune unità di nodi hanno il compito di ricevere informazioni dall'esterno che possono essere processate, interpretate o dalle quali la rete può imparare, sono per questo definite di input; le unità opposte a quelle di input vengono chiamate di output, queste mostrano come la rete risponde alle informazioni apprese ovvero forniscono il risultato dell'elaborazione. Tra output ed input troviamo le, già citate, unità nascoste che compongono la parte più consistente della rete, queste hanno il compito di elaborare i dati che vengono forniti in input.

I nodi intermedi risultano essere la struttura portante di una rete neurale e ne rappresentano la sua versatilità e potenza, questi ricevono dati in ingresso sui quali eseguono operazioni matematiche, il risultato viene poi trasmesso ai nodi successivi. La potenza di una rete neurale risiede nel fatto che un grande numero di neuroni lavorano assieme, ad esempio nel riconoscimento delle immagini. Il dato in input (immagine) viene processato strato per strato, con ogni layer di nodi che impara ad individuare caratteristiche sempre più astratte. Gli strati iniziali sono quelli che riconoscono informazioni di base, mentre gli strati più profondi sono in grado di codificare dati più complessi.

Le reti neurali si comportano come funzioni non lineari molto complesse il cui risultato è deterministico (in funzione dell'input) ma dipendente dall'architettura della rete e dal processo di training. Questo consente di aggiornare i parametri dei nodi in maniera iterativa. La modalità con cui avviene questo aggiornamento dipende dalla strategia di training. Nell'ambito dell'apprendimento supervisionato, adottato in questa tesi, si confronta l'output della rete con quello desiderato e si calcola una *funzione di costo*. Tale confronto avviene dopo che i dati di input vengono sottoposti al processo di *forward propagation*. Durante la propagazione in avanti ciascun neurone calcola una somma ponderata dei suoi input e vi applica una *funzione di attivazione*. Il risultato viene quindi passato allo strato successivo; il processo si ripete fino a che non viene raggiunto il layer di output ottenendo così il valore previsto.

Successivamente l'output della rete viene confrontato con quello corretto, come precedentemente detto questo confronto viene effettuato grazie ad una funzione di costo, con l'obiettivo di valutare quanto l'output previsto si discosti da quello desiderato, viene quindi misurato un errore. Questo processo è fondamentale per comprendere la performance della rete neurale nell'apprendimento dei dati di input.

Una volta ottenuto l'errore si passa alla retro-propagazione anche detta *backpropagation*. Durante questo processo l'errore viene propagato all'indietro, questa è

la fase in cui avviene l'apprendimento vero e proprio da parte della rete; è qui che vengono calcolati i gradienti rispetto ai *pesi* e *bias*, ovvero i parametri apprendibili di una rete neurale; a ciascuna connessione tra neuroni viene assegnato un peso, i pesi determinano la forza dell'influenza di un dato neurone di input sull'output, mentre i bias agiscono come input extra per il neurone e permettono di spostare l'output di un valore costante [10] . Il processo di retro-propagazione fornisce informazioni su come i pesi e i bias debbano essere regolati per poter diminuire l'errore. In altre parole, il procedimento appena descritto viene utilizzato per calcolare il gradiente della funzione di perdita rispetto ai pesi nella rete, consentendo così l'aggiornamento dei parametri ed il miglioramento delle prestazioni durante l'addestramento supervisionato.

Questo ciclo (propagazione in avanti, calcolo dell'errore, propagazione all'indietro e aggiornamento di pesi e bias) viene ripetuto per molte iterazioni (o *epoche*) durante un training, migliorando gradualmente l'accuratezza della previsione della rete.

1.2.1 Underfitting e overfitting

Le problematiche che si possono riscontrare nell'addestramento di una rete neurale sono molteplici due delle più comuni sono l'*overfitting* e l'*underfitting*. L'overfitting è la tendenza della rete di imparare a memoria, questo porta ad avere buoni risultati sui dati di training ma non su quelli di test, che non sono stati sottoposti alla rete durante l'addestramento. L'overfetting è facilmente individuabile durante l'addestramento poiché, nei casi in cui questo si presenta, l'errore sui dati di training tende a decrescere mentre quello sui dati di validazione tenderà ad aumentare; questo dimostra che la rete non riesce ad avere buone performance su dati che non conosce il che significa che la sua abilità di lavorare è fittizia poiché i risultati ottenuti sono frutto di una memorizzazione e non di un adattamento.

L'underfitting si presenta nel momento in cui la rete è troppo semplice quindi non è in grado di processare efficacemente nessun dato né quelli di training né quelli di test, il che si traduce in prestazioni complessivamente negative, in questi casi il modello risulta avere errori elevati sia sui dati di addestramento sia su quelli di test.

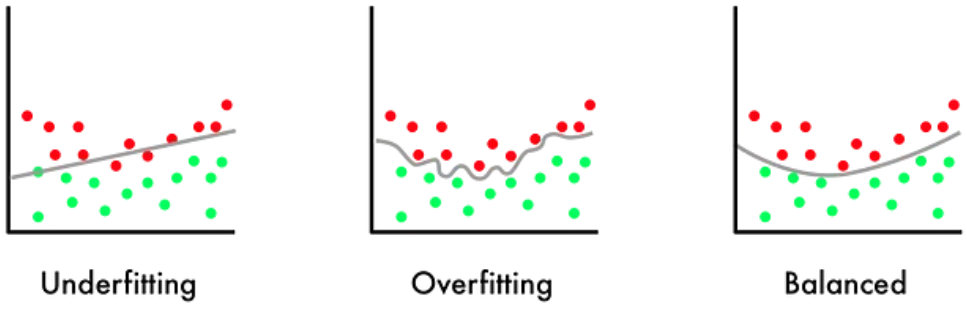

Figura 2.3: Visualizzazione grafica di underfitting e overfitting [27].

Una delle ragioni per la quale si può ricadere in underfitting è la semplicità della rete, in questi casi si cerca di aumentare il tempo di addestramento oppure si fanno modifiche alla struttura del modello stesso. Per quanto riguarda l'overfitting questo si può presentare quando la dimensione del dataset di training è limitata; infatti, molti algoritmi di ML hanno bisogno di grandi quantità di dati per conseguire buone accuratezze. Un' altra delle cause dell'overfitting è la mancanza di dati ottimali, questo significa che la carenza di dati che hanno scarsità di buone feature può compromettere l'addestramento di una rete [28].

Uno dei metodi per prevenire queste problematiche è il *data augmentation*, ovvero l'aumento delle dimensioni dei dataset utilizzati per l'addestramento delle reti applicando trasformazioni al dataset. Nel caso di immagini alcuni esempi di data augmentetion includono: la rotazione, la traslazione, il capovolgimento orizzontale o verticale, aumento o diminuzione della luminosità. Adottando le strategie menzionate la rete viene allenata su versioni diverse della stessa immagine [29]. Oltre al data augmentetion si può fare uso anche dei *synthetic data*, i dati sintetici sono dati fittizi generati artificialmente che vengono utilizzate per integrare i dati reali; questi dati possono essere creati utilizzando vari metodi come modelli generativi o algoritmi di generazione di dati. La qualità ed il realismo dei dati sintetici sono fondamentali per la loro efficacia negli algoritmi di Deep Learning [30].

La generazione di synthetic data e il data augmentation sono metodi che mirano entrambi ad aumentare la dimensione e la diversità dei dati di addestramento, entrambi però possono presentare problematiche. I dati sintetici, se non ben generati possono causare errori di realismo ma al contempo il data augmentation è limitato dai dati di addestramento originali quindi dalla loro qualità e diversità.

1.3 Reti Neurali Convoluzionali

Le Reti Neurali Convoluzionali (Convolutional Neural Networks, CNN), sono modelli parzialmente connessi particolarmente adatti all'elaborazioni di dati visivi come immagini e video; questo è reso possibile dall'architettura (Figura 1.4) che si basa prevalentemente su tre livelli specializzati che sono:

- Layer convoluzionali
- Layer di pooling
- Layer completamente connessi

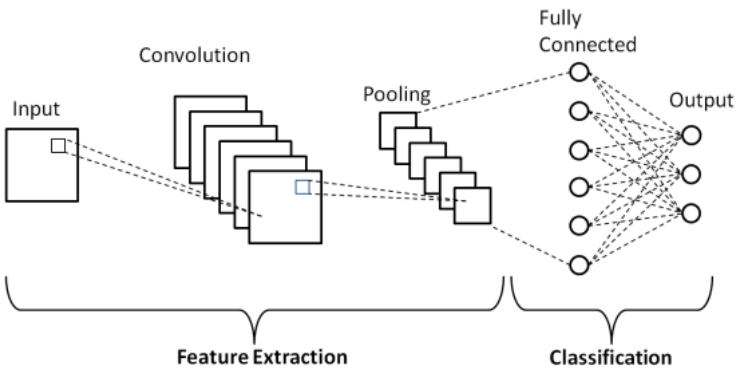

Figura 1.4: Schematizzazione di un'architettura di una CNN [14].

1.3.1 Layer convoluzionali

I layer convoluzionali sono gli elementi costitutivi fondamentali di una CNN, hanno un ruolo centrale nel funzionamento della rete poiché in questi livelli vengono eseguiti la maggior parte dei calcoli. L'input viene ricevuto da un'area specifica, chiamata *campo ricettivo*, che rappresenta la porzione dell'output dello strato precedente su cui il neurone attuale lavora. In un layer convoluzionale il campo ricettivo è generalmente una regione quadrata di n x n pixel che corrisponde all'area dell'immagine originale che influenza direttamente l'output di quel neurone, la dimensione di questo campo è determinata dalla dimensione del *kernel* [14]. Il kernel viene utilizzato per estrarre determinate caratteristiche dall'immagine; è una matrice, che viene fatta scorrere, con un avanzamento detto *stride*, sulla matrice di input (da sinistra a destra, dall'alto verso il basso) e che viene moltiplicata per le celle dell'input (Figura 1.5); la concatenazione di più kernel forma un *filtro* in cui ogni kernel è assegnato ad un particolare canale di input [15].

I pesi in ogni cella del kernel rappresentano il fattore di moltiplicazione che viene usato per calcolare il valore della cella corrispondente appartenente a quella che è la *Feature Map* (nella Figura 1.5 corrisponde a: *Convolved Feature*) ovvero l'output; per eseguire l'operazione di convoluzione il numero di kernel deve essere pari al numero di canali del tensore in input, poiché ogni filtro riduce i canali dell'immagine da c ad 1, il numero dei canali dell'output è determinato esclusivamente dal numero dei filtri (che hanno tutti lo stesso numero di kernel) utilizzati.

La formula per il calcolo della dimensione dell'output dell'operazione di convoluzione è la seguente:

$$W_{out} = \frac{W - K + 2P}{S} + 1 \tag{1.3.1}$$

Dove W è la dimensione dell'input, K è la dimensione del kernel, S è la stride e P è il *padding* ovvero l'aggiunta di pixel intorno ai margini dell'immagine di input così da garantire che durante lo scorrimento dei filtri tutti i pixels vengano considerati in egual modo.

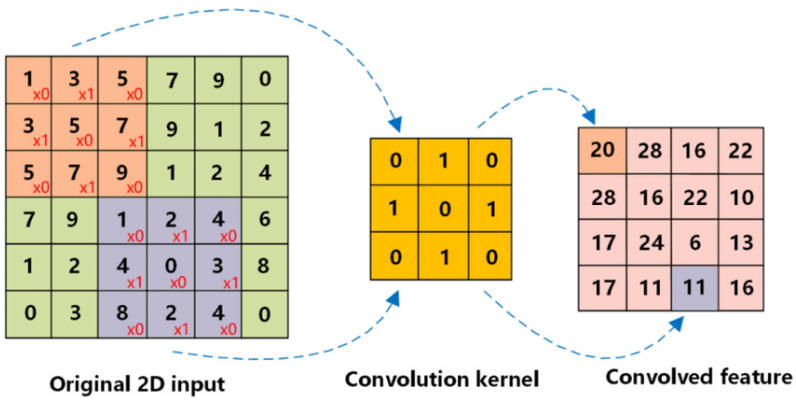

Figura 1.5: Schema delle funzionalità del kernel [17].

1.3.2 Layer di pooling

I livelli di pooling sono quelli che si occupano del *downsampling* ovvero riducono la dimensione dell'input diminuendo così il numero di parametri da allenare. Similmente ai layer convoluzionali nelle operazioni di pooling vengono utilizzati dei filtri che però non

hanno pesi ma bensì applicano una funzione di aggregazione ai valori nel campo ricettivo, generando così il tensore di output [16].

Ci sono diversi tipi di algoritmi di pooling che vengo utilizzati, quelli più comuni sono il *max pooling* che restituisce i pixel con valore più alto nel campo ricettivo e l'*avarage pooling* che calcola il valore medio dei pixel all'interno dello stesso.

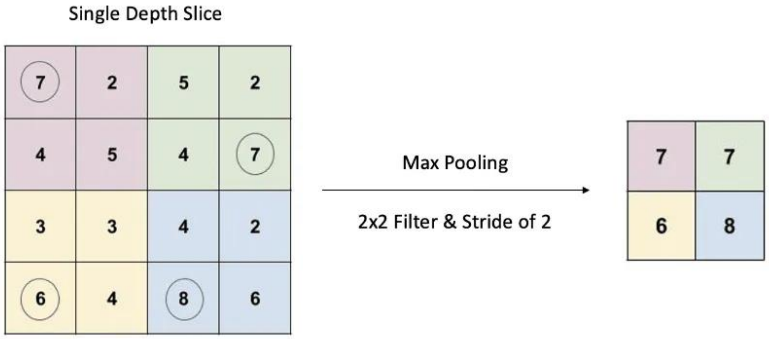

Figura 1.6: Esempio di Max Pooling con dimensione del filtro 2 x 2 e scorrimento (stride) pari a 2 [16].

I layer di pooling offrono un vantaggio in quanto non necessitano dell'apprendimento di alcun parametro, al contempo a causa di questa caratteristica si rischia di non considerare informazioni essenziali. Sebbene il pooling serva a diminuire le dimensioni ed estrarre le caratteristiche principali dei dati, esiste la possibilità che dettagli importanti vadano persi [16].

1.3.3 Layer completamente connessi

Le CNN sono modelli parzialmente connessi, nonostante ciò, talvolta, fanno utilizzo di uno o più layer completamente connessi; questi livelli, conosciuti anche come *dense layers*, connettono ogni neurone con ogni neurone dello strato precedente; lo scopo di questi tipi di livelli è quello di apprendere le caratteristiche dell'intero input piuttosto che le caratteristiche di un cluster locale [14].

Il dense layer spesso viene posizionato alla fine dell'architettura della rete, prende come input la Feature Map generata dai layer convoluzionali e di pooling, questa viene resa monodimensionale attraverso un'operazione di *flattening*. Il compito di questo livello è quello di combinare e trasformare le informazioni ricevute dai layer precedenti in un unico output, che coincide con quello della rete.

I layer appena descritti vengono ineriti all'interno di architetture CNN per problemi di regressione o classificazione mentre non vengono utilizzati nella generazione di immagini (come nel nostro caso) poiché l'output in questi casi deve essere un tensore e non un vettore o uno scalare.

1.4 Modelli discriminativi e modelli generativi

I modelli di ML posso essere divisi in *discriminativi* o *generativi*, quando si parla di modelli discriminativi si fa riferimento a quelle reti che sono in grado di estrarre informazioni utili dai dati che vengono forniti; queste informazioni vengono poi utilizzate per fare una categorizzazione dei dati stessi [18].

I modelli generativi non fanno classificazione dei dati ma tentano di apprenderne i pattern e le relazioni così da generane di nuovi; in sostanza imitano la distribuzione sottostante e sono in grado di campionarla per poter creare nuovi dati [18]. Per fare un esempio concreto, una rete discriminativa è in grado di distinguere delle biciclette da delle macchine ma non è in grado di generare immagini di biciclette o di macchine.

La differenza principale tra questi due tipi di modelli di ML è il sistema probabilistico sul quale vengono basati i calcoli. I modelli discriminativi utilizzano la *probabilità condizionata* $P(Y|X)$ tra l'input X e l'output Y, questa è la probabilità che si verifichino uno o più eventi dato il verificarsi di un altro evento e viene utilizzata per estrarre delle caratteristiche specifiche dai dati in entrata e fare in modo che questi vengano in seguito classificati. I modelli generativi utilizzano la *probabilità congiunta* (o *composta*) $P(X, Y)$ ovvero la probabilità che due eventi avvengano simultaneamente [19]; questo tipo di probabilità viene quindi sfruttata per generare nuovi dati simili a quelli di training.

L'Intelligenza Artificiale considerata tradizionale riceve dati in input sui quali effettuare operazioni di previsione, classificazione e/o raggruppamento differentemente, la GenAI (Intelligenza Artificiale Generativa) costruisce un modello statistico di base durante l'addestramento. Quando richiesto questo modello di base viene utilizzato per generare un nuovo contenuto che segue la distribuzione $P(X, Y)$ dove X è l'input del "prompt" e Y è l'output, può inoltre essere adattato a varie attività a valle (della rete), come la generazione di video/immagini, il riepilogo di testi o il completamento di immagini [20].

1.5 Autoencoders

Un autoencoder (AE) è un tipo di rete neurale generativa specializzata nel copiare il suo input nel suo output. Un *Vanilla Autoencoder* ovvero un semplice autoencoder, è formato da due reti neurali che sono l'*Encoder* ed il *Decoder*. L'encoder codifica l'immagine in un vettore compatto di dimensione minore, anche detto *vettore latente*, questo è, in altre parole, una rappresentazione compressa dell'immagine stessa; l'encoder ha quindi il compito di mappare un input dallo spazio di input, di dimensione superiore, allo *spazio latente* (latent space) che ha dimensione inferiore. Il vettore latente viene poi dato al decoder, questo ricostruisce l'immagine mappando dallo spazio latente allo spazio di output, il quale è dimensionalmente superiore. Il decoder è utilizzato per assicurare che il latent space riesca a codificare il maggior numero di informazioni possibili, costringendolo a restituire tali informazioni come input. Encoder e decoder eseguono mappature opposte l'una dall'altra [18].

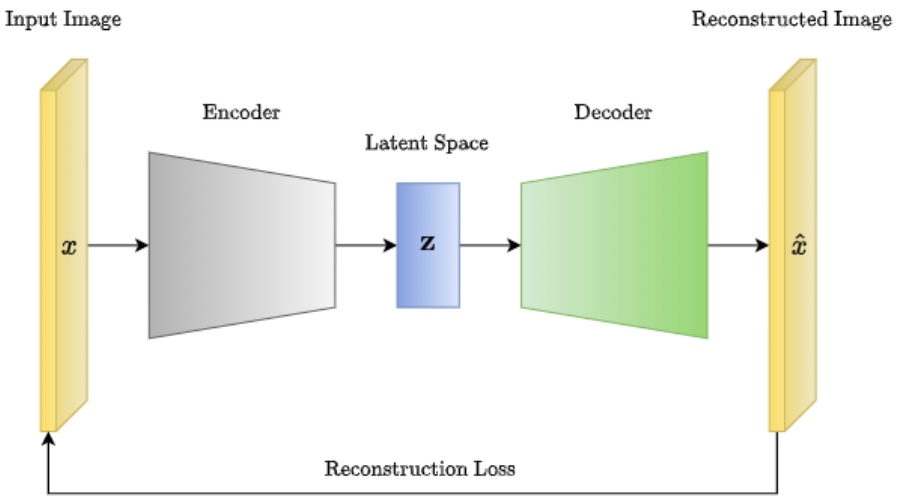

Figura 1.7: Diagramma a blocchi di un autoencoder [21].

Durante l'addestramento x (dato di input) viene fornito alla funzione dell'encoder. L'input viene passato attraverso una serie di layer che hanno il compito di ridurne la dimensione così da ottenere il vettore latente z. La dimensione, il numero e la tipologia di livelli, assieme alla dimensione dello spazio latente, viene definita dall'utente. La compressione del dato in ingresso si ottiene se la dimensione del latent space è minore di quella dello spazio di input. Come detto precedentemente, il vettore che si trova nel latent space viene passato poi al decoder, qui solitamente (ma non necessariamente) vi sono una serie di layers uguali a quelli usati nell'encoder che però operano in senso contrario; se

questo è il caso i livelli lavorano per annullare le operazioni precedentemente eseguite, ad esempio un layer di pooling nell'encoder si trasforma in un layer di un-pooling nel decoder e così via, fino ad ottenere la ricostruzione del dato di input $\hat{x}$.

L'obiettivo del training di un autoencoder è quello di rendere minima la differenza tra i dati di input ed il relativo output ricostruito; tale differenza viene generalmente misurata utilizzando una *loss function*, questa viene quindi adoperata nella valutazione delle prestazioni di un modello; una funzione di perdita tipicamente usata è *l'errore quadratico medio* (Means Squared Error, MSE):

$$Loss_{MSE} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x}_i)^2 \tag{1.5.1}$$

Nella formula 1.5.1 n è il numero complessivo dei dati x è l'input dell'encoder e $\hat{x}_i$ è l'output del decoder cioè l'immagine ricostruita.

L'idea principale è quella di avere un latent space di dimensioni molto piccole in modo tale da poter ottenere la massima compressione dei dati; ma al tempo stesso l'errore commesso dalla rete nel compiere le operazioni deve essere minimo garantendo così una codifica ottimale.

Uno dei vantaggi dell'utilizzo di un modello basato su di un AE è la capacità che questi hanno di operare in *unsupervised learning,* ovvero questi modelli sono in grado di estrarre informazioni utili da dati grezzi senza necessità di usufruire di *labelled data* (dati etichettati) ovvero dati che hanno caratteristiche, categorie o attributi assegnati. Tuttavia, gli autoencoders presentano alcune limitazioni, in primis sono sensibili all'*overfitting*, in questo modo si hanno scarsi risultati e l'accuratezza del sistema tende a calare; inoltre si possono verificare delle problematiche dovute alla perdita di informazioni causata da una compressione eccessiva dei dati, atta ad ottenere un latent space di dimensioni minime; è quindi necessario un equilibrio tra la conservazioni delle informazioni più rilevanti e la riduzione al minimo dell'errore di ricostruzione [21].

1.6 Variational Autoencoders

Per sopperire alle limitazioni citate nella sezione 1.5 si fa uso dei Variational Autoencoders (VAE). La differenza sostanziale tra AE e VAE è che mentre gli

autoencoders tradizionali mappano i dati in latent space discreti e quindi apprendono da un unico vettore di codifica, i VAE apprendono parametri di una distribuzione predefinita nello spazio latente (per ogni input), viene quindi imposto un vincolo sulla distribuzione latente che impone che questa sia una distribuzione normale. Questo vincolo fa si che lo spazio latente, a differenza di quello di un semplice autoencoder, sia controllato e continuo. La continuità del latent space rende possibile l'interpolazione tra punti distinti per generarne di nuovi [21].

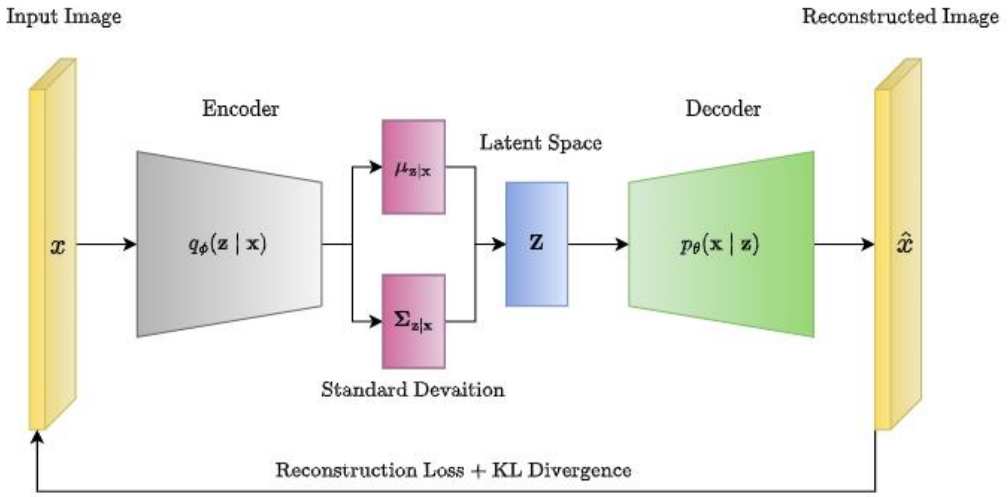

Figura 1.8: Diagramma a blocchi di un VAE (Variational AE) [21].

I VAE introducono un approccio probabilistico alla compressione dei dati di input. Durante l'addestramento, esattamente come negli AE, l'input x viene passato all'encoder che, attraverso una serie di layer, ne riduce le dimensioni comprimendolo per ottenere un vettore latente z. In questo caso non è z l'output diretto dell'encoder ma sono i parametri di una distribuzione probabilistica, ovvero la *media* (mean) e lo *scarto quadratico medio* (standard deviation) di ogni variabile latente; il vettore latente viene poi campionato dalla media e dallo scarto quadratico medio per essere dato al decoder che ha un funzionamento simile a quello di un semplice autoencoder.

In un variational autoencoder, l'encoder ha ancora il compito di mappare l'input in uno spazio di più piccola dimensione ma invece di un singolo punto questo genera una distribuzione probabilistica sullo spazio latente. Questo approccio consente ai VAE di apprendere una rappresentazione del latent space più strutturata e continua, utile per la modellazione generativa e la sintesi dei dati.

A differenziare i VAE dai semplici AE c'è anche la loss function, nel caso dei variational autoencoders si introduce la *divergenza di Kullback-Leibler* (KL Divergence), un termine che misura la differenza tra la distribuzione di probabilità appresa dalla rete e una distribuzione predefinita (solitamente distribuzione normale standard); la KL Divergence assicura che la distribuzione appresa sullo spazio latente sia vicina alla distribuzione normale, il che aiuta a regolarizzare il modello e garantisce che lo spazio latente abbia una struttura significativa [21].

La loss function di un VAE è quindi definita come segue:

$$loss = reconstruction\ loss + similarity\ loss \tag{1.6.1}$$

Dove la *reconstruction loss* si riferisce, proprio come nei semplici autoencoders, all'errore quadratico medio tra x ed $\hat{x}$, quindi tra l'input e l'output ricostruito; mentre la *similarity loss* è la divergenza di Kullback-Leibler tra la distribuzione dello spazio latente e, come già accennato, una distribuzione predefinita come ad esempio una Gaussiana con specifici valori di media e deviazione standard (o varianza).

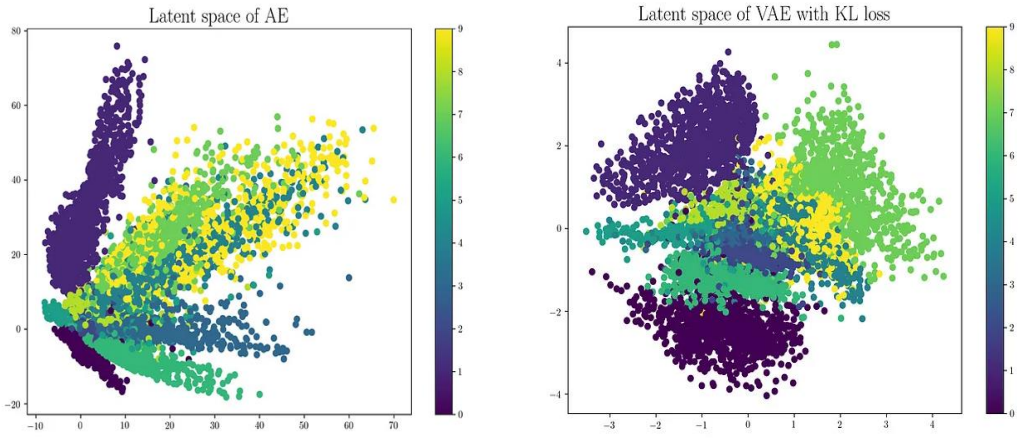

Figura 1.9: Rappresentazione di latent space in un autoencoder semplice (sinistra) e un variational autoencoder (destra) [22].

2 Utilizzo di immagini nell'osservazione terrestre e problematiche connesse ai dataset

2.1 Tipologie di immagini per l'osservazione terrestre

Le immagini per l'osservazione terrestre (Earth Observation, EO) possono essere classificate in base alle caratteristiche che le contraddistinguono. La distinzione principale viene fatta sulla natura, attiva o passiva, dello strumento che cattura l'immagine e sulla lunghezza d'onda dello spettro elettromagnetico in cui viene fatta l'osservazione. Infatti, la trasparenza dell'atmosfera terrestre alla radiazione elettromagnetica permette l'osservazione della superficie terrestre nello spettro del visibile (da 0.39 a 0.70 nm), in una parte dell'infrarosso (da 0.70 a 14 nm) e nel range delle onde radio (da 1 cm a 11m) [23].

Nei sistemi attivi gli strumenti sono composti da un trasmettitore, che emette un segnale elettromagnetico, e da un sensore che riceve il segnale dopo l'interazione con la superficie terrestre. Nei sistemi passivi troviamo sensori che sono progettati per rilevare le emissioni dei costituenti della superficie e dell'atmosfera terrestre; queste emissioni possono essere prodotte localmente, come ad esempio le radiazioni termiche prodotte dalla vegetazione nell'infrarosso, oppure possono essere il risultato della luce solare riflessa nello spettro del visibile. Per questo le immagini provenienti da strumenti di tipo passivo dipendono, in genere, dal ciclo giorno-notte e possono essere degradate o bloccate da perturbazioni provenienti da fonti indesiderate di emissioni o da copertura nuvolosa [23].

I sistemi di rilevamento passivo permettono di collezionare un'ampia varietà di immagini adatte a scopi diversi. Tra queste troviamo le *immagini pancromatiche* che rappresentano l'area osservata con una scala di grigi derivante della misurazione dell'intensità della luce su di un'ampia gamma di lunghezze d'onda, dall'infrarosso al visibile, garantendo così un'alta risoluzione. Le *immagini multispettrali*, invece, raccolgono informazioni in diverse bande dello spettro elettromagnetico. Tuttavia, per ottenere un segnale sufficiente in ciascuna banda, gli strumenti multispettrali devono raccogliere energia su spazi più ampi, risultando in una risoluzione inferiore rispetto alle immagini pancromatiche. Un esempio comune di immagini multispettrali sono quelle a

colori naturali, ottenute combinando misurazioni in 3 bande dello spettro visibile, ma è importante notare che questa tipologia di immagini può estendersi anche oltre lo spettro del visibile, includendo l'infrarosso (IR), l'ultravioletto (UV) e le microonde. Le *immagini pan-sharpened* sono ottenute combinando immagini pancromatiche ed immagini multispettrali attraverso un processo numerico, risultando in un'immagine colorata ad alta risoluzione. Infine, le *immagini iperspettrali*, insiemi di immagini che coprono una vasta gamma di bande spettrali, consentendo una discriminazione dettagliata tra diverse caratteristiche del terreno o degli oggetti. Questa tipologia di immagini sarà approfondita nel sotto-capitolo successivo, in quanto il dataset utilizzato per questo lavoro di tesi è composto esclusivamente da immagini iperspettrali.

2.2 Immagini iperspettrali

L'*imaging iperspettrale* permette di misurare le caratteristiche spaziali e spettrali di un oggetto raccogliendo immagini a diverse lunghezze d'onda; questo avviene grazie ad un sensore iperspettrale che acquisisce immagini a lunghezze d'onda strette e contigue all'intero di un intervallo spettrale specificato. Ogni immagine fornisce dettagli precisi e specifici delle caratteristiche spettrali dell'oggetto, che possono risultare utili per differenti applicazioni [24].

La gamma di lunghezze d'onda utilizzate, come per le immagini multispettrali, va oltre lo spettro del visibile, il range si estende dall'UV all'infrarosso ad onda lunga (Long Wave Infrared, LWIR); le bande più sfruttate sono il visibile, il vicino infrarosso ed il medio infrarosso.

I valori misurati dai sensori iperspettrali vengono salvati come file di dati binari, il file di dati è associato ad un ulteriore file che contiene informazioni ausiliare (metadati) come: parametri del sensore, impostazioni di acquisizione, dimensioni spaziali, lunghezze d'onda spettrali e formati codifica necessari per la corretta rappresentazione dei valori; queste informazioni possono anche essere direttamente aggiunte all'immagine attraverso l'utilizzo di formati specifici per l'immagine stessa come TIFF o NIFT. Per l'elaborazione delle immagini, i valori letti dai file di dati vengono organizzati in una matrice tridimensionale, chiamata *data cube* o *hypercube*, del tipo $x \times y \times \lambda$ dove x ed y rappresentano le coordinate spaziali e λ è la coordinata spettrale, ciò significa che in una comune immagine si avrà $\lambda = 3$ che rappresenta i canali RGB (rosso, verde e blu), mentre

in un'immagine iperspettrale λ può variare in un range che va dalle 100 alle 1000 bande (Figura 2.1) [25].

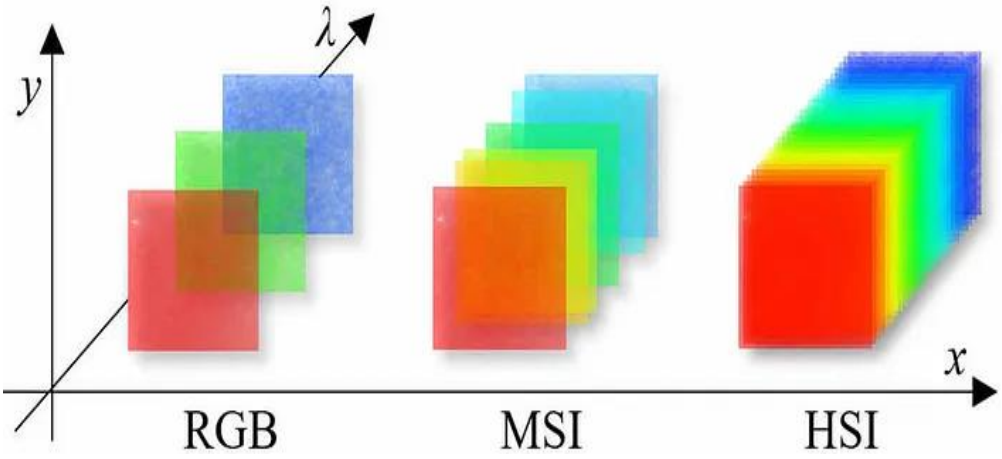

Figura 2.1: Visualizzazione delle coordinate $x \times y \times \lambda$ per la rappresentazione di immagini [25].

Ciò detto si può affermare che un'immagine iperspettrale può essere considerata come un set di immagini bidimensionali monocromatiche, catturate alle diverse lunghezze d'onda che il sensore è in grado di rilevare. I pixel che compongono il data cube sono vettori che contengono i valori di riflettanza di ciascuna banda nella posizione (x, y). Questo vettore, chiamato *pixel spectrum,* rappresenta la firma spettrale della porzione del soggetto corrispondente ai pixel aventi coordinate (x, y) [24]. I *pixel spectra* sono fondamentali nell'analisi di immagini iperspettrali poiché contengono le informazioni necessarie allo studio dello spettro di riflessione del soggetto osservato; dato che diversi materiali presentano differenti spettri di riflessione è possibile risalire alle caratteristiche fisiche del soggetto studiando i pixel spectra o l'immagine iperspettrale nel suo insieme; per questo motivo le immagini iperspettrali vengono comunemente usate per la classificazione di pixel (Figura 2.2). L'associazione tra lo spettro di riflessione ed il materiale può essere poi fatta grazie ad algoritmi di *spectral matching* oppure attraverso l'AI.

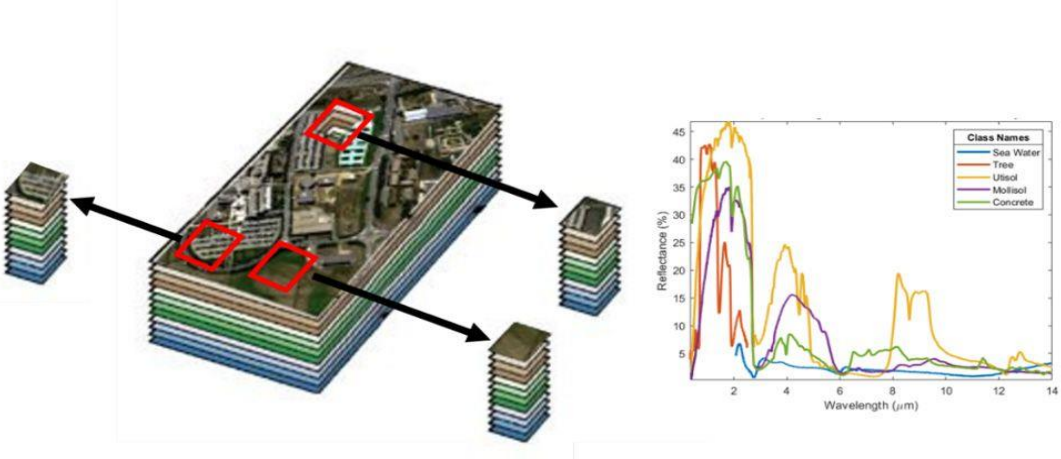

Figura 2.2: Esempio di classificazione di pixel [26].

Le immagini iperspettrali vengono spesso impiegate per problemi di classificazione, ma possono anche essere utili per riconoscere le concentrazioni di parametri del suolo. Questo ultimo aspetto è il problema per il quale sono state create le reti utilizzate per verificare il corretto funzionamento del modello VAE realizzato per questo elaborato di tesi.

2.3 Problematiche e limitazioni dei dataset utilizzati per l'Eearth Observation: l'esempio della competizione HYPERVIEW

Ad oggi i satelliti che vengono utilizzato per l'EO sono in grado di produrre quantità molto grandi di dati, ma spesso questi dati risultano non ottimi oppure hanno carenza di informazioni necessarie per la risoluzione di specifici problemi. Queste problematiche possono portare alla generazione di reti che non riescono a portare a termine il compito per le quali sono state concepite, viene quindi limitata la possibilità di validare, in maniera estensiva, algoritmi data-driven prima che questi vengano usati in determinate missioni per le quali sono stati realizzati.

Un esempio in questo senso è ravvisabile nella competizione HYPERVIEW "Seeing Beyond the Visible".

2.3.1 Competizione HYPERVIEW: "Seeing Beyond the Visibile"

Dal 9 febbraio al 1° luglio 2022, la piattaforma AI4EO, sostenuta da ESA, ha ospitato una competizione internazionale organizzata da KP Labs e QZ Solutions. L'obiettivo di questa sfida era far avanzare lo stato dell'arte per l'acquisizione dei parametri del suolo partendo da dati ottenuti sfruttando le tecnologie per l'imaging iperspettrale aereo e satellitare, in modo da ottimizzare l'uso di fertilizzanti nell'agricoltura; per la corretta scelta dei fertilizzanti è fondamentale la conoscenza aggiornata di specifici parametri del terreno quali: potassio (P), pentossido di fosforo (P_2O_5), magnesio (Mg) e pH.

L'approccio consigliato dalla competizione si basa sull'utilizzo di algoritmi di intelligenza artificiale, che partendo da immagini iperspettrali satellitari, siano in grado di stimare direttamente in orbita, perciò a bordo del satellite stesso, il valore dei parametri richiesti. Considerato che l'ottenimento di questo tipo di informazioni viene frequentemente fatto manualmente, implicando così costi elevati ed ampie tempistiche, una soluzione come quella proposta permetterebbe di velocizzare il processo e di ottenere dati in maniera capillare. L'immagine può infatti essere frazionata in sezioni minori permettendo così di ottenere stime puntali dei parametri del suolo, superando così i limiti posti dalla necessità di raccogliere e processare i campioni prelevati *in situ*.

2.3.2 Il dataset della competizione e i problemi di overfitting

Il dataset, per lo sviluppo di reti neurali, messo a disposizione dagli organizzatori della competizione comprende un totale di 2886 *patches*, ottenute sezionando immagini iperspettrali di aree agricole della Polonia, di queste 1732 sono adibite al training e le restanti 1154 alla validazione.

La dimensione di una patch può variare (a seconda della parcella agricola) ed è in media di circa 60x60 pixel; ogni patch contiene 150 bande (canali) corrispondenti alle lunghezze d'onda comprese tra 462 *nm* e 942 *nm* con una risoluzione spettrale di 3.2 *nm*. Inoltre, 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 alla zona di studio e quindi va escluso, "False" se il pixel va considerato.

Il Laboratorio di Microsatelliti e Microsistemi Spaziali ha preso parte alla competizione [35]. In tale contesto, la dimensione del dataset è risultata essere un elemento limitate ai fini dell'addestramento di un modello sufficientemente accurato, tanto da originare fenomeni di overfitting, nonostante il ricorso alle tecniche di data

augmentation menzionate in precedenza. A questo proposito, gli autori hanno deciso di fare ricorso all'introduzione di rumore gaussiano nelle immagini. Questa soluzione, pur avendo diminuito il fenomeno di overfitting, avendo generalmente migliorato le performance della rete sul test set, ha l'effetto di degradare le informazioni contenute nelle immagini iperspettrali. Per questo motivo, nell'ambito di questa tesi, si è deciso di provare ad affrontare questa problematica aumentando fittiziamente le dimensioni del dataset generando nuove immagini fittizie con l'utilizzo di Variational Autoencoder.

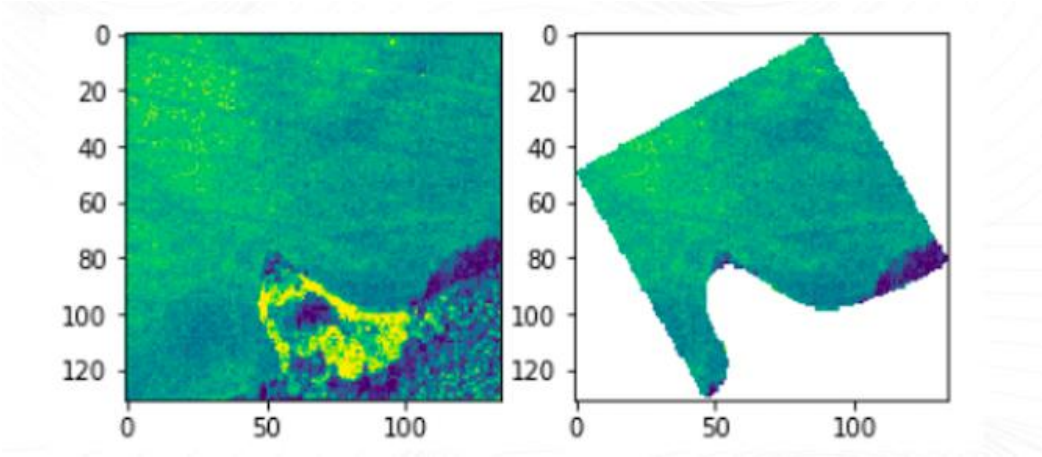

Figura 2.3: Rappresentazione casuale di una delle 150 bande (sinistra) ed applicazione di filtro (destra) [3].

3 Implementazione delle reti neurali

Il diagramma a blocchi che segue, presenta una panoramica del processo utilizzato: a partire dai label, questi vengono mappati nello spazio latente tramite l'encoder. Dopo il campionamento nel latent space, un decoder genera l'immagine corrispondente; questa immagine generata viene poi confrontata con quella originale. In fase di inferenza vengono impiegati label casuali per la generazione di nuove immagini.

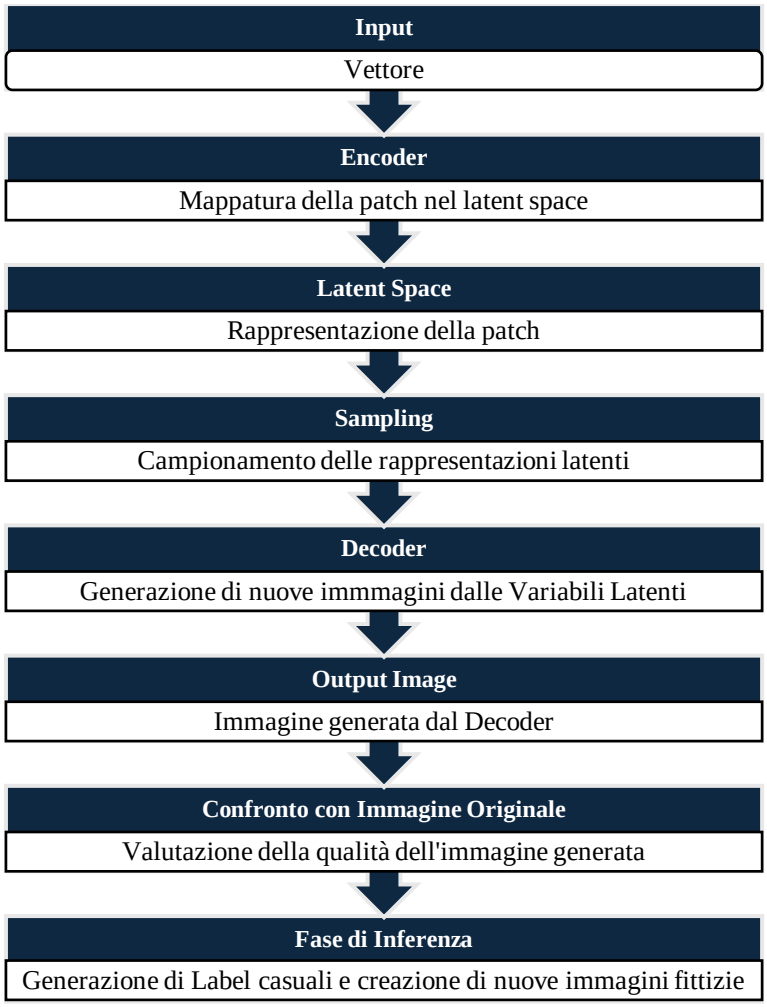

Figura 3.1: Diagramma a blocchi del processo di generazione di immagini attraverso il VAE sviluppato.

Ai fini di questo lavoro di tesi il dataset è stato notevolmente snellito; considerato che, come già accennato, le immagini presentano dimensioni variabili è stata presa la

decisione di utilizzare solo ed unicamente patch 11x11 provenienti dal dataset di train. Questa scelta deriva dalla necessità di semplificare il problema, evitando di dover gestire input di dimensioni variabili. Il numero totale di immagini impiegate si è quindi ridotto a 650, di queste 520 sono state utilizzate per l'addestramento e 130 per la validazione.

3.1 Estrazione delle patch 11x11 e preprocessing

Inizialmente è stato eseguito un processo di filtraggio delle immagini per selezionare quelle di dimensione 11x11. Utilizzando le librerie *numpy* ed *os* è stata eseguita una scansione dei file di train forniti per la competizione; le immagini con dimensione differente da 11x11 sono state escluse, mentre quelle con dimensioni appropriate sono state selezionate per l'inclusione in un nuovo dataset. Le immagini selezionate sono state suddivise in due set distinti: un set di training ed un set di validation, il set di addestramento è costituito dall'80% delle immagini mentre il set di validazione è invece composto dal restante 20 %.

Dopo la preparazione, le immagini (sottoforma di file npz) sono state convertite in formato TFRecord, tale conversione consente un'elaborazione efficiente ed una gestione dei dati ottimizzata durante l'addestramento, contribuendo così a migliorare le prestazioni del modello. Utilizzando le funzionalità fornite da TensorFlow, è stato quindi creato un file TFRecord per il set di train ed uno per il set di validation. Ogni file TFRecord contiene le immagini codificate in byte, insieme ai relativi label in formato float.

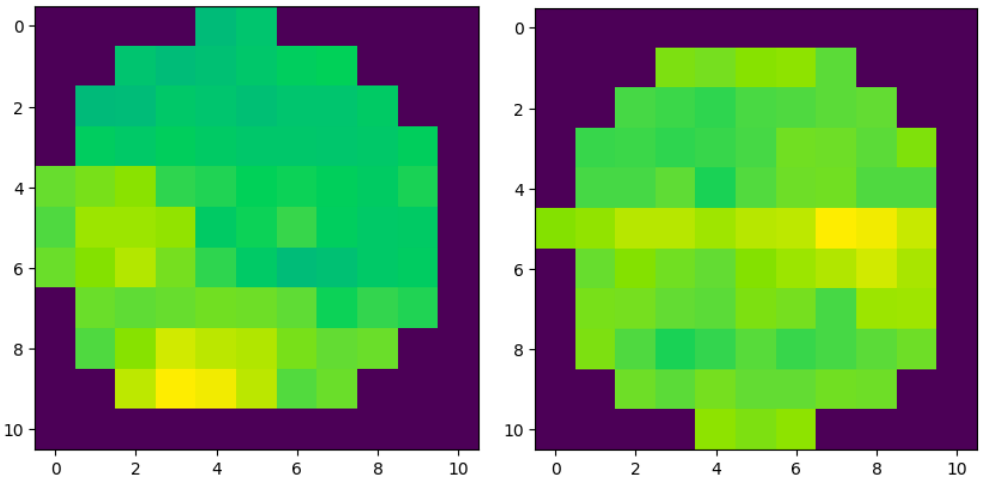

Figura 3.2: Esempi di immagini 11x11 estratte dal dataset originale.

3.2 Arichettura del Variational Autoencoder

Il modello VAE creato è composto da un encoder, un layer di campionamento (sampling layer) e un decoder, tutti e tre i componenti vengono definiti come una *sottoclasse* di keras.Model. Una classe in Python serve ad unire insieme dati e funzionalità, creare una sottoclasse equivale a creare un nuovo *tipo*, a partire dal quale possono essere create nuove *istanze* (oggetti); ciascuna istanza può avere degli attributi suoi propri, che ne mantengono lo stato. Le istanze possono anche avere dei metodi (definiti nella classe) che ne modificano lo stato [31].

L'encoder prende in input un vettore dei label e lo elabora attraverso vari-strati densi (fully connected) con l'obiettivo mappare il vettore in uno spazio latente di dimensioni superiori. Questo utilizzo atipico del VAE è stato scelto per controllare la generazione delle nuove immagini, consentendo di conoscere quale sarà il label associato ad una specifica immagine generata e garantendo così un maggior controllo del processo. In questo caso specifico, l'encoder è costituito da dense layers con attivazione GELU (Gaussian Error Linear Unit):

$$GELU(x) = xP(X \leq x) = x\phi(x) = x \cdot \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right] \tag{3.2.1}$$

Dove erf è la funzione di errore gaussiano e $\phi(x) = P(X \leq x)$ è la funzione di distribuzione cumulativa gaussiana standard; questa funzione può essere semplificata come segue:

$$0.5x\left(1 + tanh\left[\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right]\right) \tag{3.2.2}$$

oppure

$$x\sigma(1.702x) \tag{3.2.3}$$

Le funzioni 3.3.2 e 3.3.3 alleggeriscono notevolmente il carico computazionale a scapito dell'accuratezza. La non linearità GELU pesa gli input in base al loro valore e non in base al loro segno come la ReLU (Rectified Linear Unit) ed è per questo più precisa [32].

Dall'encoder vengono generati due output denominati *mean_mu* e *log_var*, questi vengono creati per rappresentare rispettivamente la media ed il logaritmo della varianza dello spazio latente, che verranno successivamente utilizzati nel calcolo della loss di Kullback-Leibler.

Il sampling layer durante l'esecuzione riceve in input, dall'encoder, media e logaritmo della varianza e genera campioni casuali nello spazio latente utilizzando la tecnica del *trucco di riparametrizzazione* (formula 3.3.4) in modo da separare, nell'operazione di campionamento, la parte deterministica da quella stocastica.

$$z = \mu + \left(e^{\frac{\log(\sigma^2)}{2}} * \varepsilon \right) \tag{3.2.4}$$

La formula 3.3.4 rappresenta il reparameterization trick usato nella rete, in questo caso μ rappresenta la media, $\log(\sigma^2)$ rappresenta il logaritmo della varianza ed ε è un vettore di campioni casuali estratti da una distribuzione normale standard con media 0 e deviazione 1. Risulta necessario evidenziare che calcolare l'esponenziale della metà del logaritmo della varianza è un passaggio critico in quanto trasforma il logaritmo della varianza in deviazione standard; la successiva moltiplicazione della neo-deviazione standard per epsilon introduce la casualità nei campioni. In sostanza, questa formula consente di generare campioni casuali nello spazio latente che sono stocasticamente dipendenti dai parametri (media e varianza) della distribuzione latente, garantendo al contempo la differenziabilità necessaria per l'addestramento tramite backpropagation.

Per quanto riguarda il decoder, riceve in input gli output del layer di campionamento e li trasforma attraverso una serie di strati di de-convoluzione *Conv2DTranspose*. Questi strati applicano un'operazione nota come convoluzione trasposta, che costituisce l'inverso della convoluzione standard. Invece di far scorrere il kernel sulle immagini di input per estrarne le caratteristiche, la convoluzione trasposta esegue il processo contrario. Viene quindi effettuata l'operazione di upsampling, che aumentando gradualmente la dimensione, permette di raggiungere la grandezza dell'immagine desiderata. Infine, gli ultimi strati del decoder utilizzano funzioni di attivazione *LeakyReLU* e *Sigmoid* per generare l'output finale.

3.3 Definizione di ottimizzatori e loss function

Per preparare il codice all'addestramento sono state utilizzate specifiche ottimizzazioni e funzioni di perdita, fondamentali per il successo e la precisione del processo di apprendimento automatico.

L'ottimizzazione viene configurata in egual modo sia per l'encoder che per il decoder utilizzando l'ottimizzatore *Adam* (Adaptive Moment Estimation). Adam è ampiamente impiegato nelle reti neurali poiché è molto efficace nel velocizzare la convergenza dell'addestramento. L'idea dietro alla quale nasce Adam è la combinazione dei concetti alla base di altri due ottimizzatori ovvero *RMSprop* e *Momentum*; questa unione rende Adam in grado di adattare il tasso di apprendimento per ogni parametro della rete in base ai gradienti storici e alla varianza dei gradienti [33]. Entrambi gli ottimizzatori utilizzano un programma di decadimento esponenziale per il tasso di apprendimento, questo schema di decadimento graduale del tasso di apprendimento, implementato attraverso *tf.keras.optimizers.schedules.ExponentialDecay*, è progettato per ottimizzare l'efficacia e la stabilità dell'algoritmo nel corso delle iterazioni di addestramento. Inoltre, sia nell'encoder che nel decoder, viene impostato un limite sulla norma pari ad 1 per prevenire eventuali problemi di stabilità.

Per ciò che concerne la loss function adottata questa si basa su due diverse funzioni distinte che sono la *kl_loss* e la *mse_loss*; la prima calcola la perdita di divergenza Kullback-Leibler, la seconda si occupa di calcolare l'errore quadratico medio (funzione 1.5.1) tra l'immagine originale e la sua controparte ricostruita nel modello.

$$KL = -0.5 \sum \left(1 + log(\sigma^2) - \mu^2 - e^{log(\sigma^2)}\right) \tag{3.3.1}$$

La funzione 3.3.5 rappresenta la KL Divergence dove μ e $\log(\sigma^2)$ sono rispettivamente la media ed il logaritmo della varianza.

All'interno della stessa cella di codice della funzione di perdita vengono definiti gli *accumulatori di perdita* per monitorare la perdita media durante l'addestramento e la fase di validazione dell'encoder e del decoder. Questi accumulatori sono fondamentali per valutare le prestazioni del modello durante il training.

3.4 Definizione del training

La procedura di addestramento è guidata dalla funzione *train_model* che accetta come parametro le epoche. Il numero di epoche è un *iperparametro*, ovvero una variabile libera impostata dall'utente prima dell'inizio dell'addestramento, che definisce il numero di volte in cui l'algoritmo di apprendimento processerà l'intero set di dati di addestramento, sostanzialmente è una misurazione del tempo di training.

Durante ogni epoca il modello viene addestrato mediante la funzione *train_step*. In primis c'è l'ottenimento di un *batch* di dati di addestramento, contenente le immagini originali e i relativi labels, la dimensione del batch è anch'esso un iperparametro, che definisce il numero di campioni da elaborare prima di aggiornare i parametri del modello intero. Successivamente l'encoder elabora i label per generare la media ed il logaritmo della varianza della distribuzione latente, questi due valori peraltro aiutano a capire dove si trovino e quanto siano sparsi i dati nello spazio latente; μ e σ^2 vengono poi utilizzati per generare un nuovo punto all'interno del latent space, tramite il sampling layer che gestisce il processo di campionamento. Infine, il dato ottenuto viene fornito al decoder che cerca di ricostruire l'immagine.

Le perdite dell'encoder e del decoder vengono quindi calcolate attraverso la loss function descritta nella sezione 3.4, utilizzando la divergenza KL per l'encoder e l'errore quadratico medio per il decoder. Queste perdite rappresentano la discrepanza tra l'input e l'output del modello, valutando l'accuratezza della ricostruzione delle immagini e la coerenza della distribuzione latente con la distribuzione di riferimento. Le perdite calcolate vengono quindi utilizzate per aggiornare i pesi del modello mediante la discesa del gradiente, con l'obiettivo di migliorare le prestazioni.

In parallelo all'addestramento viene eseguita la validazione tramite la funzione *valid_step* che esegue inferenza sui dati di validazione. Anche in questo caso encoder e decoder vengono utilizzati per elaborare i dati e calcolare le rispettive perdite. Questo passaggio di test della rete fornisce una valutazione obiettiva delle prestazioni del modello su dati non visti durante l'addestramento, consentendo di rilevare eventuali problemi di generalizzazione. Infine, al termine dell'addestramento, i modelli dell'encoder e del decoder vengono esportati utilizzando le funzioni *save* di TensorFlow. Questo passaggio è cruciale per conservare i modelli addestrati e consentire il loro futuro utilizzo per la generazione di nuove immagini o l'inferenza su nuovi dati.

3.4.1 Risultati dell'addestramento

Durante la fase di training del modello sono stati eseguiti diversi addestramenti al fine di trovare i giusti iperparametri e parametri di inizializzazione per l'encoder ed il decoder. Per ciò che concerne gli iperparametri sono state fissate 300 epoche ed una dimensione del batch di 32 con un *learning rate* (tasso di apprendimento) iniziale di 0.005, questo serve a controllare la velocità e la stabilità del processo di apprendimento del modello durante l'addestramento. Inoltre, è stata implementata una normalizzazione dei label di addestramento, controllata da *label_normalization_mode.* Se il parametro viene impostato a 0, viene applicata la normalizzazione min-max; per qualsiasi altro valore viene utilizzata la normalizzazione standard. I parametri per la normalizzazione dei dati di addestramento sono riportati in Tabella 3.1.

Parametro del suolo	Media	Deviazione Standard	Massimo
P_2O_2 [ppm]	70.30	29.50	325.0
K [ppm]	227.99	61.87	625.0
Mg [ppm]	195.28	39.86	400.0
Ph	6.78	0.26	7.8

Tabella 3.1: Valori parametri del suolo utilizzati per l'addestramento della rete VAE

Oltre a ciò che è riportato in tabella, per normalizzare le immagini è stato implementato anche il valore di massima riflettanza, pari a 6315. I valori dei parametri del suolo sono stati ottenuti dall'analisi del dataset condotta dal team che ha sviluppato le reti neurali utilizzate per valutare l'efficacia del VAE.

Ulteriori iperparametri sono stati definiti prima di avviare il processo di addestramento del Variational Autoencoder. In particolare, per l'encoder sono stati definiti un input di dimensione pari a 4, che corrisponde alla dimensione dei label nel dataset di Hyperview, e una dimensione dello spazio latente pari a 500; il decoder è stato invece inizializzato attraverso la configurazione dei filtri convoluzionali, la definizione delle dimensioni del kernel convoluzionale e degli stride. In particolare, sono stati selezionati 150 filtri per ciascuno dei tre strati convoluzionali per garantire una corrispondenza con ciascuna delle 150 bande presenti nel dataset. Le dimensioni dei kernel sono state fissate a [3,3,3] mentre gli stride a [3,2,2], tenendo conto delle

dimensioni delle immagini (11x11 pixel); tuttavia poiché 11 è un numero primo e non può essere raggiunto con queste configurazioni, le immagini generate avranno dimensione 12x12 e successivamente verranno ridimensionate mediante il taglio di una riga e di una colonna.

Durante training sono stati monitorati diversi parametri chiave al fine di valutare l'efficacia ed il progresso nel corso delle epoche, la tabella 3.2 riporta la variazione delle perdite di encoder e decoder nelle ultime 10 epoche di addestramento, calcolate sul set di train e di validazione.

Epoca	Encoder Loss (Train)	Decoder Loss (Train)	Encoder Loss (Validation)	Decoder Loss (Validation)	Learning Rate
290	8.818e-08	9.210e-03	2.095e-09	9.850e-03	3.036e-05
291	6.932e-08	8.856e-03	1.630e-09	9.839e-03	2.983e-05
292	6.770e-08	9.010e-03	1.863e-09	9.661e-03	2.931e-05
293	3.038e-08	9.123e-03	1.863e-09	9.545e-03	2.879e-05
294	7.643e-08	9.326e-03	1.863e-09	9.863e-03	2.829e-05
295	6.903e-08	9.208e-03	1.630e-09	9.824e-03	2.779e-05
296	7.561e-08	8.775e-03	2.328e-09	9.686e-03	2.730e-05
297	6.432e-08	9.523e-03	1.863e-09	9.694e-03	2.683e-05
298	5.285e-08	8.667e-03	1.863e-09	9.783e-03	2.636e-05
299	6.932e-08	9.288e-03	2.328e-09	9.864e-03	2.590e-05
300	5.041e-08	9.250e-03	1.863e-09	9.798e-03	2.545e-05

Tabella 3.2: Elenco risultati delle ultime 10 epoche dell'addestramento del VAE

Dalla tabella 3.2 si può osservare come, nel corso delle epoche, le perdite di encoder e decoder si siano stabilizzate, un segnale positivo poiché è indicativo del fatto che il modello stia imparando a rappresentare i dati in modo efficiente nel suo spazio latente e stia generando output consistenti con l'input. Inoltre, la perdita sui dati di validazione è rimasta stabile con piccole fluttuazioni, suggerendo una buona capacità del modello di generalizzare sui dati non visti durante l'addestramento. Si può osservare che

il tasso di apprendimento è stato adattato dinamicamente durante il training, con un leggero decremento ad ogni epoca, questo approccio è importante per evitare oscillazioni o convergenza prematura. In generale, questi risultati suggeriscono che il modello stia apprendendo in maniera efficace e stia producendo output coerenti con le aspettative.

In aggiunta ai dati numerici, sono stati generati grafici per visualizzare l'andamento delle perdite durante l'addestramento (Figura 3.3).

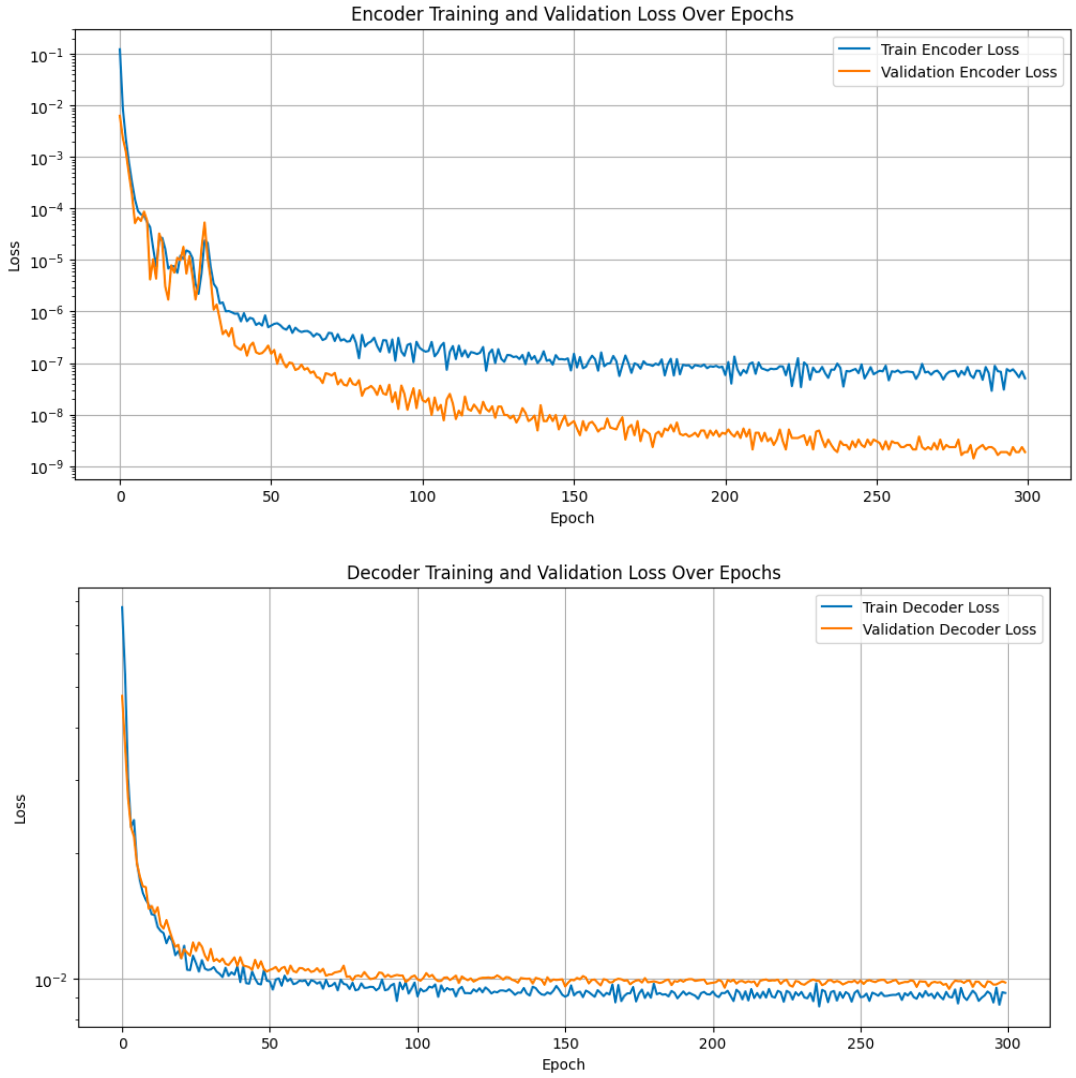

Figura 3.3: Grafici delle perdite durante nell'encoder (sopra) e nel decoder (sotto) durante l'addestramento.

3.5 Generazione delle immagini

Dopo aver allenato la rete su immagini esistenti questa viene utilizzata per generarne di nuove. Inizialmente viene caricato il modello VAE opportunamente addestrato poi viene definito element_mapping, un dizionario che mappa i nomi dei nutrienti ai corrispondenti simboli chimici facilitando la manipolazione dei dati durante il processo di generazione e salvataggio. Dopodiché viene definita la funzione generate_inference_images che viene utilizzata per generare un numero prefissato di immagini pari a 520. Questo processo coinvolge la generazione casuale di valori per i parametri del suolo utilizzando distribuzioni normali con media e deviazione standard specificati; questi valori vengono poi utilizzati come input per l'encoder che restituisce i parametri della distribuzione latente. I parametri latenti vengono campionati utilizzando il layer apposito e passati al decoder per la generazione di immagini sintetiche.

Le immagini generate vengono ridimensionate così da ottenere la desiderata dimensione di 11x11 (le motivazioni sono state tratte nella sezione 3.5.1), successivamente vengono trasposte così da avere formato tensoriale 150x11x11, questo per consentire un'elaborazione efficiente durante le fasi successive di analisi. In seguito alle operazioni appena descritte le immagini vengono salvate, ognuna con il proprio indice univoco, all'interno di una cartella chiamata generated_data con formato npz. I labels dei nutrienti associati a ciascuna immagine vengono registrati in un DataFrame e salvati all'interno di un file CSV denominato labels_csv; ogni riga di tale file rappresenta un'immagine e contiene i valori dei parametri del suolo corrispondenti, nonché il già citato indice univoco dell'immagine.

4 Test ed analisi dei risultati

4.1 Modello di rete neurale utilizzato

La rete utilizzata per l'analisi delle prestazioni del Varational Autoencoder è stata concepita come possibile soluzione alla competizione Seeing Beyond the Visibile (sezione 3.1), tale proposta è basata su una delle cinque varianti della rete EfficientNet-Lite ovvero EfficientNet-Lite0. Le reti EfficientNet-Lite vengono tradizionalmente impiegate nel processing di immagini RGB, a tal proposito la rete EfficientNet-Lite0 è stata adattata al dataset fornito per la competizione, dando vita alla rete EfficientNet-Lite0mod. Il modello originale della rete è realizzato per lavorare con immagini di dimensioni 224x224x3, al fine di avere un modello compatibile con i dati della competizione, il numero di canali in ingresso alla prima convoluzione è stato aumentato da 3 a 150 e le dimensioni sono state impostate a 32 px. La componente di regressione della rete è stata implementata tramite un livello completamente connesso composto da 4 neuroni e privo di funzione di attivazione. Al fine di ottimizzare la computazione e ridurre il numero di parametri della rete, sono state apportate modifiche ai coefficienti di larghezza (width) e profondità (depth), impostati entrambi al 50% della loro originaria configurazione nell'EfficientNet-Lite0. Tali coefficienti regolano il numero di blocchi e livelli della rete rispetto alla struttura di base. Infine, è stato adottato un tasso di dropout, tecnica utilizzata per ridurre l'overfitting, del 10% [35]. In Tabella 3.3 sono presentanti i dettagli sulla struttura della rete.

Layer	Kernel	Output Size	Output Channels	# Layers
Image	-	32x32	150	1
Conv2D + BN + ReLU	3x3	16x16	32	1
MBConv6	3x3	16x16	8	1
MBConv6	3x3	8x8	16	1
MBConv6	5x5	4x4	24	1
MBConv6	3x3	2x2	40	2
MBConv6	5x5	2x2	56	2

MBConv6	5x5	1x1	96	2
MBConv6	3x3	1x1	160	1
Conv2D + BN + ReLU	1x1	1x1	1280	1
Dense	-	1x1	4	1
Model Parameters			734,020	

Tabella 4.1: Struttura della rete EfficientNet-Lite0mod e numero di parametri. Conv2D indica la convoluzione bidimensionale, MBConv6 è il mobile inverted bottelneck [34] e BN è la Batch Normalization.

4.1.1 Training effettuato con rete EfficientNet-Liteb0mod

Il training per l'effettiva valutazione del VAE è stato effettuato attraverso il notebook *train_test_model* [4]. Il processo di addestramento può essere suddiviso in diverse fasi chiave; nella fase iniziale vengono definiti gli iperparametri di training, questi includono le epoche pari a 300, la dimensione del batch impostata a 32 e il tasso di apprendimento iniziale di 0.005, diminuito progressivamente in base ad una legge di cosine decay (Figura 4.1). Successivamente viene definita una metrica personalizzata rappresentativa di quella utilizzata per calcolare il punteggio durante la competizione sulla quale si basava la classifica, la formula è la seguente:

$$Score = \frac{\sum_{i=1}^{4}\left(MSE_i / MSE_i^{base}\right)}{4} \tag{4.1.1}$$

dove

$$MSE_i = \frac{\sum_{j=1}^{|\psi|}\left(p_j - \hat{p}_j\right)^2}{|\psi|} \tag{4.1.2}$$

Nell'equazione 4.1.1 il termine $|\psi|$ indica la cardinalità del dataset, mentre $\hat{p}_j$ e p_j rappresentano rispettivamente il valore dell'i-esimo parametro stimato reale, sulla j-esima immagine. Si procede poi al calcolo dell'errore quadratico medio per ogni parametro sulle immagini di test, questi vengono poi divisi per i corrispondenti MSE_i^{base}, ovvero gli errori conseguiti da un algoritmo che approssima i $\hat{p}_j$ con valori medi

riscontrati nei dati di training. Essendo *Score* la misurazione di un errore, più basso è il valore migliore sarà la qualità della soluzione.

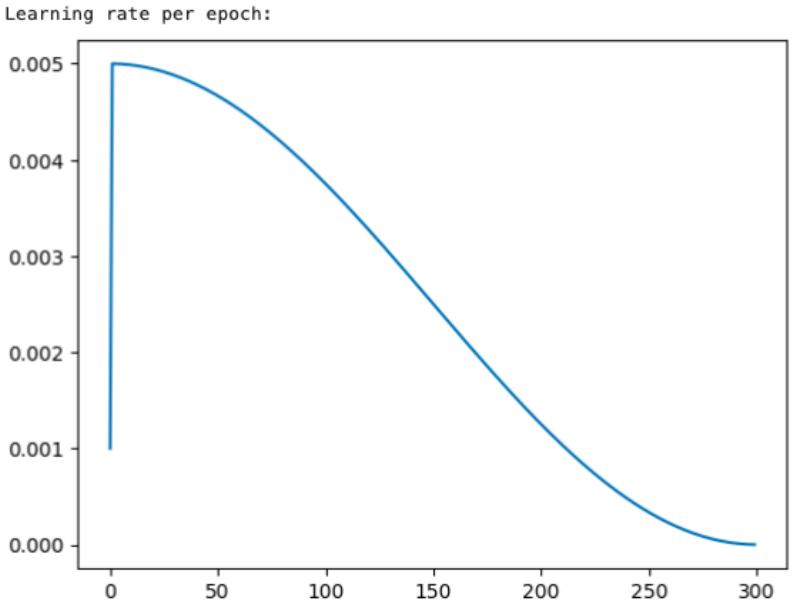

Figura 4.1: Decadimento del learning rate.

In seguito si passa alla definizione dell'ottimizzatore (Adam), la funzione di perdita ovvero l'errore quadratico medio e la metrica personalizzata sopra descritta. La struttura del modello può essere visionata nella tabella 4.2.

Infine, i dati vengono caricati e preparati per l'addestramento, questi sono in primis normalizzati, poi aumentatati (in numero), variando leggermente quelli già esistenti ad esempio ruotandoli, e divisi in batch. Inoltre, vengono applicate le operazioni di *cache* e di *prefetch*. Con cache dei dati si intende il temporaneo salvataggio in memoria di quest'ultimi; quando il modello necessita dei dati durante le epoche successive può accedervi direttamente dalla cache, risparmiando così tempo. Invece, con prefetch dei dati si intende il pre-caricamento dei dati in anticipo, in modo che siano pronti per essere elaborati quando il modello ne ha bisogno. In sintesi, l'utilizzo combinato della cache e del prefetch consente di ottimizzare l'accesso ai dati durante l'addestramento, migliorando velocità ed efficienza complessiva.

Layer (type)	Output Shape	Parametri
EfficientNet-Lite0mod (Functional)	(None, 1, 1, 1280)	728896
flatten_1 (Flatten)	(None, 1280)	0
dense_1 (Dense)	(None, 4)	5124
Parametri totali	734,020	
Parametri allenabili	719,780	
Parametri non allenabili	14,240	

Tabella 4.2: Architettura del modello utilizzato per la valutazione del VAE [4].

4.2 Addestramento con dataset originale

Come primo caso è stato studiato quello che prevede in input solo le immagini originali, con questo si intende che il set di train e di validazione sono formati rispettivamente da 520 immagini 11x11 e 130 immagini 11x11 provenienti dal dataset di cui si è discusso nella sezione 3.2. Per avere una panoramica dell'andamento dell'addestramento si fa può fare riferimento alla tabella 4.3:

Epoca	Loss	Custom Metric	Val. Loss	Val. Custom Metric	LR
1	0.5418	435.5032	0.1412	224.0737	0.0010
5	0.0582	36.3299	0.0249	4.1604	0.0050
10	0.0366	10.0583	0.0133	2.6854	0.0050
50	0.0150	2.7839	0.0139	3.2681	0.0047
100	0.0125	2.3631	0.0110	2.6814	0.0038
200	0.0101	1.3954	0.0090	1.2024	0.0013
250	0.0099	1.3126	0.0093	1.3162	3.4861e-04
290	0.0099	1.3036	0.0094	1.3974	1.7066e-05
295	0.0098	1.2926	0.0094	1.4389	5.4327e-06
300	0.0098	1.2962	0.0094	1.4429	6.3708e-07

Tabella 4.3: Sommario delle epoche dell'addestramento portato a termine con dati originali.

Basandosi sulle epoche riportate in tabella 4.3 si può affermare che la loss e la metrica personalizzata sono soggette ad una diminuzione durante l'addestramento. La loss calcolata sui dati di train sembra diminuire in maniera costante il che indica che il modello sta imparando dai dati e sta migliorando la sua capacità di fare previsioni. Tuttavia, è importante notare che la loss sulla validazione sembra stabilizzarsi o addirittura aumentare dopo un certo punto (epoca 200), questo potrebbe indicare un potenziale leggero overfitting. In Figura 4.2 è riportato il confronto tra le due loss function evidenziando un andamento convergente. In Figura 4.3 viene presentato il grafico, in scala logaritmica, dell'andamento delle loss nelle ultime 100 epoche. Il valore del loss di training risulta essere leggermente superiore a quello di validation. Questo può essere imputabile alla presenza di aumentazioni che rendono il dataset di train potenzialmente più complesso.

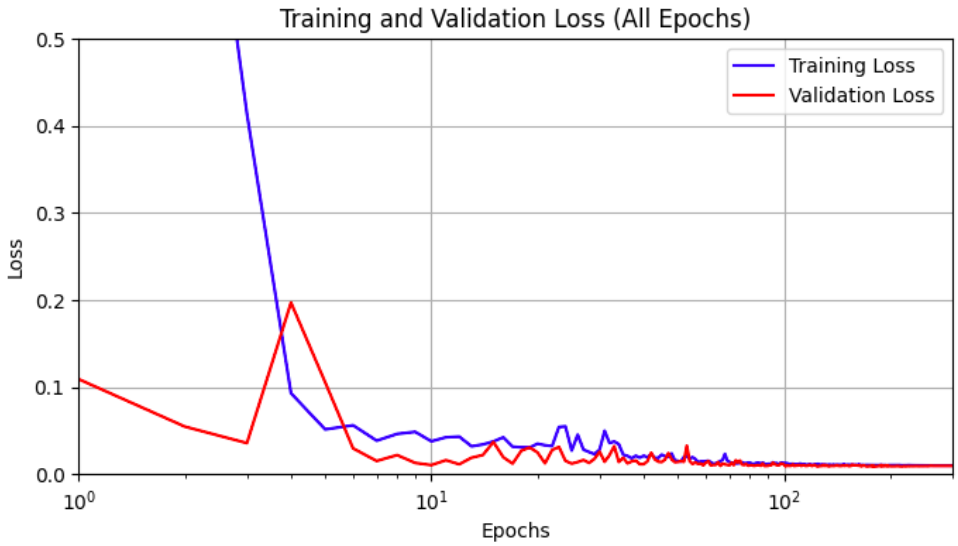

Figura 4.2: Grafico dell'andamento delle loss, per il set di train e di validazione, dell'addestramento portato a termine utilizzando le immagini 11x11 provenienti dal dataset originale.

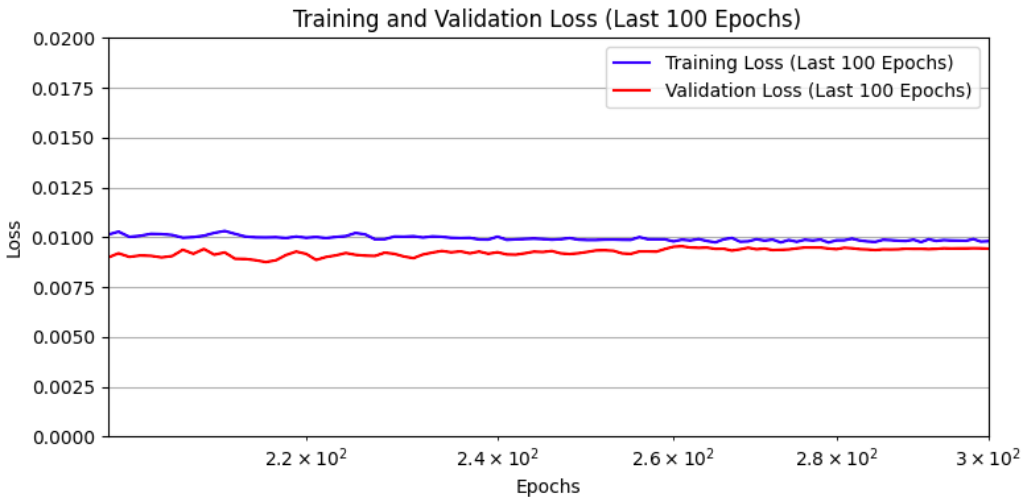

Figura 4.3: Grafico in scala logaritmica dell'andamento nelle ultime 100 epoche delle loss, per il set di train e di validazione, dell'addestramento portato a termine utilizzando le immagini 11x11 provenienti dal dataset originale.

Dato che il training è stato condotto utilizzando il dataset composto da immagini reali è ragionevole ammettere che la buona riuscita dell'addestramento fosse prevedibile. Tuttavia, va notato che la lieve tendenza all'overfitting potrebbe essere attribuita al fatto che il dataset è stato notevolmente ridotto, limitando la diversità dei dati disponibili per l'allenamento. Nonostante ciò, la rete ha dimostrato di funzionare in modo affidabile, come confermato dalla validazione.

4.3 Addestramento con dataset generato con l'ausilio del modello VAE

Nella seconda analisi è stato adottato l'utilizzo delle sole immagini generate interamente dal Variational Autoencoder. Per garantire un confronto equo ed affidabile dei risultati, il dataset utilizzato è stato bilanciato in termini di quantità di immagini sia per il train che per la validazione, infatti, le immagini di validazione rimangono le stesse mentre quelle di train vengono sostituite con le fittizie 520 generate dal VAE.

Questo secondo test della rete rappresenta un primo importante passo verso la valutazione dell'efficacia del modello VAE realizzato; l'utilizzo delle immagini sintetiche permette di esplorare la capacità del modello di generare dati realistici e di valutare il suo impatto sul successivo addestramento di reti neurali per compiti specifici.

Epoca	Loss	Custom Metric	Val. Loss	Val. Custom Metric	LR
1	0.5478	378.5565	0.0921	149.0451	0.0010
5	0.0628	30.6907	0.0165	4.1392	0.0050
10	0.0374	16.9422	0.0159	10.6842	0.0050
50	0.0161	3.4639	0.0182	2.000	0.0047
100	0.0083	1.9702	0.0060	1.1298	0.0038
200	0.0072	1.1190	0.0065	0.9860	0.0013
250	0.0069	1.0395	0.0065	0.8821	3.4861e-04
290	0.0069	1.0121	0.0065	0.8929	1.7066e-05
295	0.0069	1.0152	0.0065	0.8962	5.4327e-06
300	0.0069	1.0154	0.0065	0.8867	6.3708e-07

Tabella 4.4: Sommario delle epoche dell'addestramento portato a termine interamente con dati sintetici.

Dai dati nella Tabella 4.4 si può notare che la loss e la custom metric di train, assieme alla custom metric di validazione, partono da valori elevati che vengono notevolmente diminuiti nei primi 100 cicli dell'addestramento. A partire dall'epoca 200, le metriche tendono a stabilizzarsi, suggerendo che il modello stia raggiungendo un punto di saturazione nell'apprendimento. La loss sulla validazione diminuisce in modo coerente durante le prime epoche ma come nel caso precedente (sezione 4.2) sembra esserci una tendenza alla stabilizzazione con un lieve aumento dei valori. Queste considerazioni possono essere meglio comprese grazie a grafici in Figura 4.4 e Figura 4.5; inoltre si può apprezzare che le due loss tendono a convergere, e lo fanno a valori più bassi rispetto all'addestramento con immagini originali.

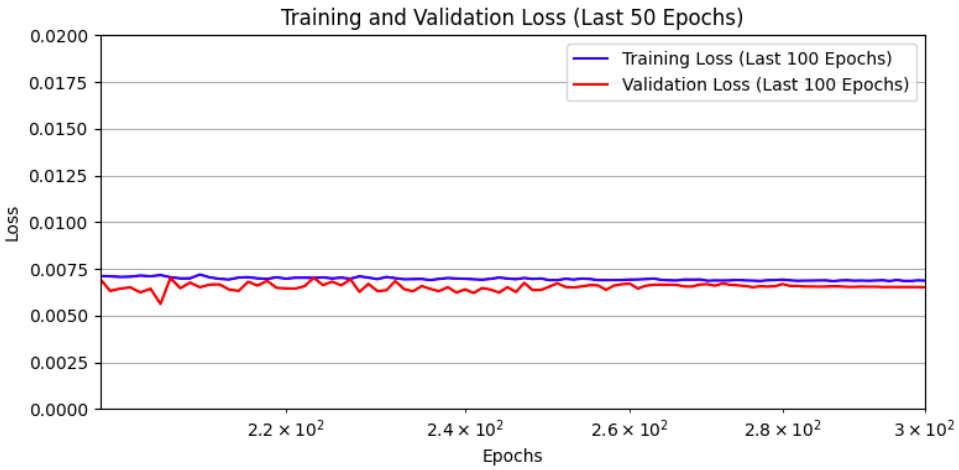

Figura 4.4: Grafico dell'andamento delle loss, per il set di train e di validazione, dell'addestramento portato a termine utilizzando le immagini sintetiche.

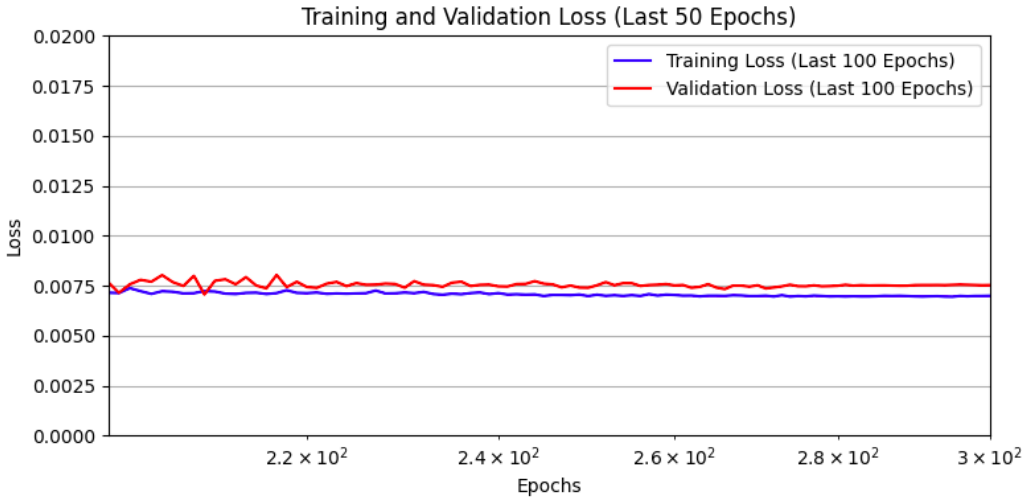

Figura 4.5: Grafico in scala logaritmica dell'andamento nelle ultime 100 epoche delle loss, per il set di train e di validazione, dell'addestramento portato a termine utilizzando le immagini sintetiche.

4.4 Addestramento con dataset combinato

L'ultima tipologia di training portata a termine è quella più significativa a livello di analisi del VAE, questa infatti prevede l'utilizzo di un dataset differente dove le immagini di train sono una combinazione delle patch 11x11 del dataset originale e delle immagini sintetiche generate attraverso il Variational AE per un totale di 1040 immagini, mentre per la validazione vengono utilizzate le 130 immagini provenienti dal dataset originale. Al fine di mantenere un'analisi equilibrata le epoche sono state dimezzate, riducendo il numero da 300 a 150. Questo aspetto è stato considerato in relazione agli step di training che rappresentano l'elaborazione di ciascun batch attraverso la rete. Il numero totale di step in un'epoca è determinato dal rapporto tra il numero di elementi nel dataset e la dimensione del batch; in questo specifico caso con un dataset di 520 elementi ed un batch size di 32 si ottiene un totale di 16,25 step, questo implica che con 300 epoche vengono eseguiti 4800 step. Tuttavia, con un dataset raddoppiato il numero di step raddoppia di conseguenza a 9600, tale aumento implicherebbe che a parità di epoche la rete viene allenata con un maggior numero di aggiornamenti dei parametri.

Epoca	Loss	Custom Metric	Val. Loss	Val. Custom Metric	LR
1	0.4553	313.3740	0.0667	99.384	0.0010
5	0.0396	22.8585	0.0152	2.7798	0.0050
50	0.0139	2.7567	0.0118	1.6834	0.0038
100	0.0088	1.2954	0.0100	1.3508	0.0013
140	0.0083	1.1689	0.0102	1.3984	6.6546e-05
145	0.0084	1.1668	0.0102	1.3973	2.0211e-05
148	0.0084	1.1799	0.0102	1.3923	5.4327e-06
149	0.0084	1.1697	0.0102	1.3943	2.6926e-06
150	0.0083	1.1668	0.0102	1.3944	1.0483e-06

Tabella 4.5: Sommario delle epoche dell'addestramento portato a terminane con l'unione di dati sintetici ed originali.

Basandosi sui dati riportati in tabella si può notare che la loss e le metriche personalizzate sono rimaste piuttosto stabili e a livelli relativamente bassi sia per i dati di addestramento sia per quelli di validazione. Questo suggerisce che il modello ha raggiunto una buona capacità di adattamento e di generalizzazione al dataset combinato. La loss di train è rimasta bassa indicando che il modello è stato in grado di ridurre l'errore durante l'ottimizzazione dei parametri; la stessa osservazione può essere estesa alla loss di validazione, nonostante questa tenda ad oscillare di più, il che indica che il modello è in grado di generalizzare bene sui dati che non ha visto in precedenza poiché è in grado di mantenere basso l'errore stesso. Pertanto, la stabilità e la bassa loss, sia di addestramento che di validazione, indicano che il modello ha appreso in modo efficace dalle informazioni presenti nel dataset combinato.

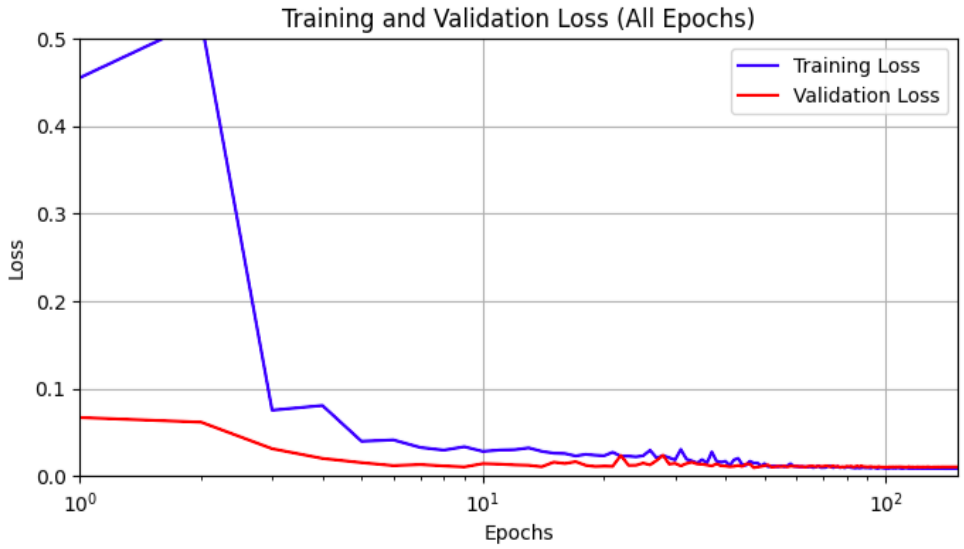

Figura 4.6: Grafico dell'andamento delle loss, per il set di train e di validazione, dell'addestramento portato a termine utilizzando immagini reali e sintetiche.

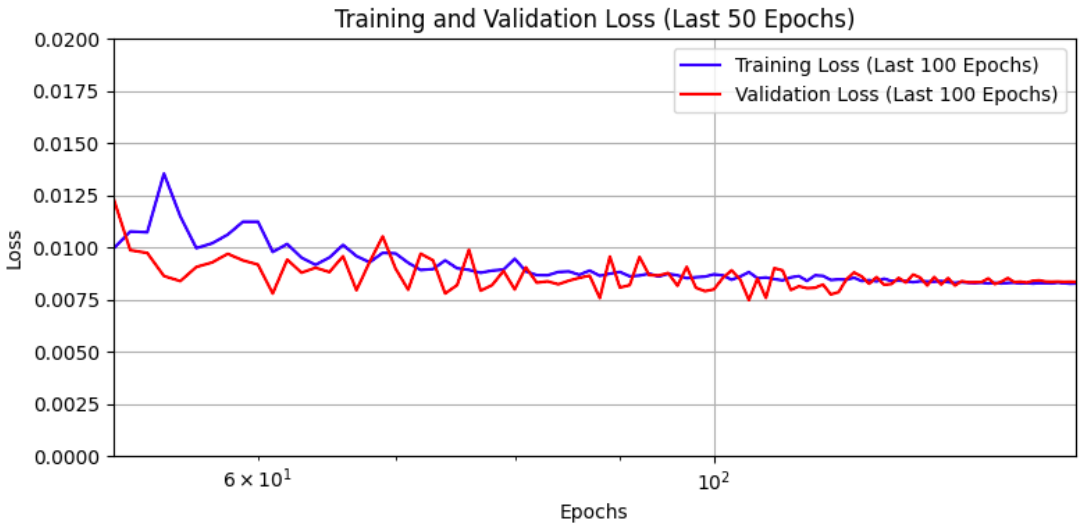

Figura 4.7: Grafico in scala logaritmica dell'andamento nelle ultime 50 epoche delle loss, per il set di train e di validazione, dell'addestramento portato a termine utilizzando le immagini sintetiche e reali.

L'utilizzo di un dataset combinato sembrerebbe aver portato un miglioramento rispetto all'addestramento con dataset reale. Questo miglioramento è evidente dalla riduzione delle loss e delle metriche personalizzate, inoltre il modello sembra aver raggiunto una maggiore stabilità come indicato dalla minore variazione delle metriche nel corso delle epoche.

4.5 Risultati della sottomissione al server

Al termine di ciascuno dei tre addestramenti, la rete ha generato una stima dei parametri del suolo per ciascuna immagine del dataset di test. È importante notare che, in questa fase, sono state utilizzate tutte le immagini del dataset di test, comprese quelle di dimensioni superiori a 11x11 pixel. Queste immagini sono state processate seguendo la stessa metodologia applicata durante la competizione, assicurando così coerenza nei risultati. I valori stimati sono salvati in un file csv che è successivamente inviato al server per il calcolo del punteggio finale, come descritto nell'Equazione (4.1.1).

In tabella 4.6 sono riportati i punteggi ottenuti dai tre diversi addestramenti eseguiti sui tre dataset: immagini reali, immagini originate dal VAE e dataset combinato.

Dataset	Score
Originale	1.05618
Generate	1.00647
Combinato	1.04622

Tabella 4.6: Sommario risultati sottomissione al server di Hyperview [3].

Dall'analisi dei risultati emerge che l'addestramento condotto usando esclusivamente immagini generate dal VAE ha prodotto il punteggio migliore. Questo è evidenziato dal fatto che tale punteggio sia il più vicino allo zero, il che riflette una maggiore accuratezza nelle predizioni. Ciò indica che le immagini sintetiche riescono a catturare in maniera efficiente le caratteristiche essenziali dei dati ma consentono anche al modello di generalizzare meglio.

L'approccio combinato, che unisce immagini reali e generate, ha prodotto un risultato meno performante di quello sperato, indicando che nonostante l'utilizzo del VAE abbia avuto impatto positivo sia sulla generalizzazione che sulla capacità predittiva del modello, rimane un margine di miglioramento. Questo suggerisce che, sebbene le immagini generate abbiano dimostrato un potenziale significativo, l'integrazione con dati reali potrebbe richiedere ulteriori affinamenti per ottenere un risultato ottimale.

5 Conclusioni

In questo elaborato finale è stato sviluppato e valutato un modello Variational Autoencoder (VAE) per la generazione di immagini satellitari fittizie, con l'obiettivo di migliorare la predizione di parametri del suolo (*K*, P_2O_5, *Mg*, *pH*) ampliando le dimensioni di un dataset limitato. I risultati hanno dimostrato che l'o di immagini generate porta a prestazioni superiori rispetto a quelle ottenute con le sole immagini originali. In particolare, i test effettuati tramite la sottomissione al server della challenge Hyperview hanno confermato che il dataset composto esclusivamente da immagini sintetiche ha permesso al modello di raggiungere il miglior punteggio, diminuendo lo score dato dal dataset originale di *0.04971*.

Il comportamento stabile durante l'addestramento, unito ai risultati ottenuti dal modello in fase di test, evidenzia come il VAE sia stato in grado di generare dati fittizi di alta qualità, capaci di migliorare la capacità predittiva della rete. Questo approccio si rivela particolarmente utile in contesti in cui la disponibilità di dati reali è limitata o di bassa qualità, come l'Earth Observation (EO).

Nonostante i risultati promettenti, esistono alcuni limiti, evidenziati dallo scarso miglioramento del punteggio tra il dataset originale e quello composto dall'unione di immagini reali e generate. Sebbene le immagini sintetiche abbiano dimostrato di migliorare le prestazioni complessive del modello, non lo fanno a tal punto da apportare un miglioramento globale significativo quando combinate con le immagini reali. Ad ogni modo sarebbe utile effettuare un'analisi statistica completa per valutare la rappresentatività dei dataset di addestramento originale e quello generato rispetto al dataset di test, al quale attualmente non abbiamo accesso.

In prospettiva futura sarebbe interessante esplorare ulteriormente il VAE al fine di poterlo migliorare. Aumentare le dimensioni del vettore latente o implementare varianti più avanzate come il *Conditional VAE* o le *Generative Adversarial Networks* (GAN), potrebbe incrementare la diversità e la qualità delle immagini fittizie generate dalla rete.

In conclusione, questo lavoro ha dimostrato che i modelli generativi, come i VAE, rappresentano una soluzione promettente per affrontare la scarsità di dati e migliorare la capacità predittiva di modelli di machine learning. Le potenziali applicazioni di questo approccio spaziano dalla geoscienza all'agricoltura di precisione, potendo contribuire in

modo significativo all'analisi del suolo e ad altre attività che richiedono dati satellitari dettagliati e predizioni accurate.

Bibliografia

[1] ESA. *"OrbitalAI Challenge"*. URL: https://platform.ai4eo.eu/orbitalai-phisat-2/data

[2] ESA. *"The Sentinel Mission"*. URL: https://www.esa.int/Applications/Observing_the_Earth/Copernicus/The_Sentinel_missions

[3] ESA. *"Seeing Beyond the Visibile"*. URL: https://platform.ai4eo.eu/seeing-beyond-the-visible

[4] *hyperspectral-cnn-soil-estimation*. URL: https://github.com/Microsatellites-and-Space-Microsystems/hyperspectral-cnn-soil-estimation/blob/main/README.md

[5] Choi RY, Coyner AS, Kalpathy-Cramer J, Chiang MF, Campbell JP. Introduction to Machine Learning, Neural Networks, and Deep Learning. Transl Vis Sci Technol. 2020 Feb 27;9(2):14. doi: 10.1167/tvst.9.2.14. PMID: 32704420; PMCID: PMC7347027.

[6] Coursera. *"Deep Learning vs. Machine Learning: A Beginner's Guide"*. URL: https://www.coursera.org/articles/ai-vs-deep-learning-vs-machine-learning-beginners-guide

[7] Machine Learning and Statistics. *"Understanding the Universal Aproximation Theorem"*. Medium. 2023. URL: https://medium.com/@ML-STATS/understanding-the-universal-approximation-theorem-8bd55c619e30

[8] Mohaiminul Islam, Guorong Chen, Shangzhu Jin. An Overview of Neural Network. American Journal of Neural Networks and Applications. Vol. 5, No. 1, 2019, pp. 7-11. doi: 10.11648/j.ajnna.20190501.12

[10] Machine Learning in Plain English. *"Deep Learning Course – Lesson 5: Forward and Backward Propagation"*. URL: https://medium.com/@nerdjock/deep-learning-course-lesson-5-forward-and-backward-propagation-ec8e4e6a8b92

[11] Aisolab. *"Intro to AI #1: Artifical Intelligence, Machine Learning, Neural Networks and Deep Learning: What are the differences?"*. URL: https://aiso-lab.com/intro-to-ai-1-artificial-intelligence-machine-learning-neural-networks-and-deep-learning-what-are-the-differences/

[12] Stojov, Vladimir & Koteli, Nikola & Lameski, Petre & Zdravevski, Eftim. (2018). *"Application of machine learning and time-series analysis for air pollution prediction"*.

[13] Andrea Minini. *"Le reti neurali informatiche"*. URL: https://www.andreaminini.com/ai/le-reti-neurali-informatiche

[14] Xiang Fan, Phuong Truong. *"An Introduction to Convolutional Neural Network (CNN)"*. Medium. 2022. URL: https://medium.com/sfu-cspmp/an-introduction-to-convolutional-neural-network-cnn-207cdb53db97

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

[16] Afaque Umer. *"Understanding Convolutional Neural Networks: A Beginners's Journey into the Architecture"*. Medium. 2023. URL: https://medium.com/codex/understanding-convolutional-neural-networks-a-beginners-journey-into-the-architecture-aab30dface10

[17] J. Zheng, L. Ma, Y. Wu, L. Ye, e F. Shen, «Nonlinear Dynamic Soft Sensor Development with a Supervised Hybrid CNN-LSTM Network for Industrial Processes», *ACS Omega*, vol. 7, fasc. 19, pp. 16653–16664, mag. 2022, doi: 10.1021/acsomega.2c01108.

[18] Dhanush Kamath. *"Generating New Faces With Variational Autoencoders"*. Medium. 2020. URL: https://towardsdatascience.com/generating-new-faces-with-variational-autoencoders-d13cfcb5f0a8

[19] Jason Brownlee. *"A Gentle Introduction to Bayes Theorem for Machine Learning"*. Machine Learning Mastery. 2019. URL: https://machinelearningmastery.com/bayes-theorem-for-machine-learning/

[20] Reshma Abraham. *"Let's learn Everything About Generative AI together"*. Medium. 2023. URL: https://reshma-a.medium.com/lets-learn-everything-about-generative-ai-together-d546e0a7b4cf

[21] Rushikesh Shende. *"Autoencoders, Variational Autoencoders (VAE) and β-VAE"*. Medium. 2023. URL: https://medium.com/@rushikesh.shende/autoencoders-variational-autoencoders-vae-and-β-vae-ceba9998773d

[22] Aqeel Anwar. *"Difference between AutoEncoder (AE) and Variationale AutoEncoder (VAE)"*. Medium. 2021. URL:

https://towardsdatascience.com/difference-between-autoencoder-ae-and-variational-autoencoder-vae-ed7be1c038f2

[23] ESA. *"Newcomers Earth Observation Guide"*. URL: https://business.esa.int/newcomers-earth-observation-guide - ref_1

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

[25] Billy G. Ram. *"What is Hyperspetracl Imaging?"*. Medium. 2022. URL: https://medium.com/precision-agriculture/what-is-hyperspectral-imaging-c928b7f59cc

[26] MathWorks. *"Hyperspetracl Imaging"*. URL: https://it.mathworks.com/discovery/hyperspectral-imaging.html

[27] David Chuan-En Lin. *"8 Simple Techniques to Prevent Overfitting"*. Medium. 2020. URL: https://towardsdatascience.com/8-simple-techniques-to-prevent-overfitting-4d443da2ef7d

[28] Matthew Stewart. *"The Limitations of Machine Learning"*. Medium. 2019. URL: https://towardsdatascience.com/the-limitations-of-machine-learning-a00e0c3040c6

[29] Shiv Vignesh. *"The Perfect Fit for a DNN"*. Medium. 2020. URL: https://medium.com/analytics-vidhya/the-perfect-fit-for-a-dnn-596954c9ea39

[30] Doğan Keskin. *"Synthetic Data and Data Augmentation"*. Medium. 20203. URL: https://medium.com/@dogankeskin01/synthetic-data-and-data-augmentation-c022029dd660

[31] Riccardo Poligneri, *"Classi"*. Il tutorial di Python. 2020. URL: https://pytutorial-it.readthedocs.io/it/python3.11/classes.html

[32] D. Hendrycks e K. Gimpel, «Gaussian Error Linear Units (GELUs)». arXiv, 5 giugno 2023. Consultato: 5 maggio 2024. [Online]. Disponibile su: http://arxiv.org/abs/1606.08415

[33] Marco Speciale. *"Deep Learning a MIST: Come utilizzare una rete neurale convolutive per il riconoscimento di immagini"*. Diario Di Un Analista. 2023. URL: https://www.diariodiunanalista.it/posts/rete-convolutiva-su-mnist/

[34] Ballabeni, A. (2022). *"Implementazioni di reti neurali per missioni di telerilevamento satellitare da immagini iperspettrali"* (Tesi di Laurea, Università di Bologna). URL: https://amslaurea.unibo.it/27012/

[35] J. Nalepa et al., *"Estimating Soil Parameters From Hyperspectral Images: A benchmark dataset and the outcome of the HYPERVIEW challenge"* in IEEE Geoscience and Remote Sensing Magazine, doi: 10.1109/MGRS.2024.3394040. keywords: {Soil;Satellites;Artificial intelligence;Monitoring;Vegetation mapping;Reproducibility of results;Magnesium}.

Allegati

Allegato 1: Definizione di encoder, layer di campionamento e decoder

```
# Class to get the encoder
class get_encoder(tf.keras.Model):
    def __init__(self, input_shape, latent_dim):
        super().__init__()

        self.encoder_input = Input(shape=(input_shape,),
name='encoder_input')
        # Connect layers using Functional API
        x = Dense(latent_dim//4, activation='gelu',
kernel_initializer=tf.keras.initializers.GlorotUniform(seed=212))
(self.encoder_input)
        x = Dense(latent_dim//2, activation='gelu',
kernel_initializer=tf.keras.initializers.GlorotUniform(seed=981))(x)
        x = Dense(latent_dim,
kernel_initializer=tf.keras.initializers.GlorotUniform(seed=976))(x)

        mean_mu = Dense(latent_dim, name = 'mu')(x)
        log_var = Dense(latent_dim, name = 'log_var')(x)

        #self.encoder_output = tf.keras.layers.Concatenate()([mean_mu,
log_var])
        self.encoder_output = ([mean_mu, log_var])

    def build_graph(self):

        return Model(self.encoder_input, [self.encoder_output])
```

```
# Class to define the sampling process

class SamplingLayer(tf.keras.layers.Layer):
    def __init__(self, **kwargs):
        super(SamplingLayer, self).__init__(**kwargs)

    def call(self, args):
        mean_mu, log_var = args
        epsilon =
tf.keras.backend.random_normal(shape=tf.shape(mean_mu), mean=0.,
stddev=1.)
        return mean_mu + tf.exp(log_var / 2) * epsilon

    def build_graph(self):
        # Build a model using the SamplingLayer
        input_mean_mu = Input(shape=(latent_dim,))
        input_log_var = Input(shape=(latent_dim,))
        sampling_output = self.call([input_mean_mu, input_log_var])

        return Model(inputs=[input_mean_mu, input_log_var],
outputs=sampling_output)
```

```
# Class to get the decoder

class get_decoder(tf.keras.Model):
    def __init__(self, input_dim, conv_filters, conv_kernel_size,
conv strides):
        super(get_decoder, self).__init__()

        # Number of Conv layers
        n_layers = len(conv_filters)

        # Define model input
        self.decoder_input = Input(shape=(input_dim,),
name='decoder_input')

        x = Reshape([1,1,input_dim])(self.decoder_input)
        # Add convolutional layers

        for i in range(n_layers):
            x = Conv2DTranspose(
                filters=conv_filters[i],
                kernel_size=conv_kernel_size[i],
                strides=conv_strides[i],
                padding='same',
                name='decoder_conv_' + str(i)
            )(x)

            # Adding a LeakyReLU layer at each step, except the last
one
            if i < n_layers - 1:
                x = LeakyReLU()(x)
            else:
                x = Activation('sigmoid')(x)

        # Define model output
        self.decoder_output = x

    def build_graph(self):
        return Model(self.decoder_input, self.decoder_output)
```

Allegato 2: Ottimizzatori e loss functions.

```
# Optimizers
total_steps = steps_per_epoch*epochs

optimizer_encoder=tf.keras.optimizers.Adam(

learning_rate=tf.keras.optimizers.schedules.ExponentialDecay(start_lr,
total_steps,start_lr), clipnorm=1,
)

optimizer_decoder=tf.keras.optimizers.Adam(

learning_rate=tf.keras.optimizers.schedules.ExponentialDecay(start_lr,
total_steps,start_lr), clipnorm=1,
)

# Losses
# KL loss measures the difference between the learned distribution
(encoded by the mean and variance predicted by the model) and a
standard normal distribution.
def kl_loss(mean_mu, log_var):
    kl_loss = -0.5 * tf.math.reduce_sum(1 + log_var -
tf.math.square(mean_mu) - tf.math.exp(log_var), axis = 1)
    return kl_loss

def mse_loss(image_original, image_regressed):

    # We need to drop a row and a column from the regressed image
    image_regressed = image_regressed[:,:-1,:-1,:]

    return tf.math.reduce_mean(tf.math.square(image_original -
image_regressed), axis = [1,2,3])

# train loss accumulators
encoder_loss_tracker = tf.keras.metrics.Mean(name="encoder_loss")
decoder_loss_tracker =
tf.keras.metrics.Mean(name="decoder_loss_tracker")

# validation loss accumulators
encoder_val_loss_tracker =
tf.keras.metrics.Mean(name="encoder_val_loss")
decoder_val_loss_tracker =
tf.keras.metrics.Mean(name="decoder_val_loss_tracker")
```

Allegato 3: Funzioni di addestramento.

```
@tf.function
def train_step(data):

  #Each record in the file returns filename, image, label
  filename, image_original, label = data

  with tf.GradientTape(persistent=True) as tape:
    # Forward pass
    encoderoutput = encoder(label,training=True)

    mean_mu = encoderoutput[0][0]
    log_var = encoderoutput[0][1]

    embeddings = sampling_layer([mean_mu, log_var])
    image_regressed = decoder(embeddings, training=True)

    # Compute losses
    encoder loss = kl_loss(mean_mu, log_var)
    decoder_loss = mse_loss(image_original,image_regressed)

  # Accumulate loss
  encoder_loss_tracker.update_state(encoder_loss)
  decoder_loss_tracker.update_state(decoder_loss)

  # Compute gradients for encoder
  trainable_vars_encoder = encoder.trainable_variables
  gradients_encoder = tape.gradient(encoder_loss,
trainable_vars_encoder)

  # Update weights for encoder
  optimizer_encoder.apply_gradients(zip(gradients_encoder,
trainable_vars_encoder))

  # Compute gradients for decoder
  trainable vars decoder = decoder.trainable variables
  gradients_decoder = tape.gradient(decoder_loss,
trainable_vars_decoder)

  # Update weights for encoder
  optimizer decoder.apply_gradients(zip(gradients_decoder,
trainable_vars_decoder))
```

```
@tf.function
def valid_step(data):
  filename, image_original, label = data

  # Forward pass
  encoderoutput = encoder(label,training=False)

  mean_mu = encoderoutput[0][0]
  log_var = encoderoutput[0][1]

  embeddings = sampling_layer([mean_mu, log_var])
  image_regressed = decoder(embeddings, training=False)

  # Compute losses
  encoder_val_loss = kl_loss(mean_mu, log_var)
  decoder_val_loss = mse_loss(image_original,image_regressed)

  # Accumulate loss
  encoder_val_loss_tracker.update_state(encoder_val_loss)
  decoder_val_loss_tracker.update_state(decoder_val_loss)
```

Allegato 4: Inferenza per la generazione di immagini.

```
encoder_model = keras.models.load_model('encoder')
decoder_model = keras.models.load_model('decoder')
sampling_model = SamplingLayer().build_graph()

element_mapping = {
    'phosphorus': 'P',
    'potassium': 'K',
    'magnesium': 'Mg',
    'ph': 'pH'
}

def generate_inference_images(encoder_model, decoder_model,
sampling_model, num_images=650, mean_values={'phosphorus': 70.30264,
'potassium': 227.98851, 'magnesium': 159.2812, 'ph': 6.782706},
std_dev={'phosphorus': 29.496254, 'potassium': 61.874084, 'magnesium':
39.860474, 'ph':  0.2602235}):
    generated_images = []
    labels = []

    for i in range(num_images):
        # randomly generates the values of P, K, Mg, pH using a
Gaussian distribution centered on the mean
        phosphorus =
np.round(np.random.normal(mean_values['phosphorus'],
std_dev['phosphorus']),1)
        potassium =
np.round(np.random.normal(mean_values['potassium'],
std_dev['potassium']),1)
        magnesium =
np.round(np.random.normal(mean_values['magnesium'],
std_dev['magnesium']),1)
        ph = np.round(np.random.normal(mean_values['ph'],
std_dev['ph']),1)


        label = {
            element_mapping['phosphorus']: phosphorus,
            element_mapping['potassium']: potassium,
            element_mapping['magnesium']: magnesium,
            element_mapping['ph']: ph,
        }

        input_data = np.array([list(label.values())])
        encoder_output = encoder_model.predict([input_data])
        mean_mu = encoder_output[0][0]
        log_var = encoder_output[0][1]

        embeddings = sampling_model([mean_mu, log_var])

        generated_image = decoder_model.predict([embeddings])
```

```
        generated_images.append(generated_image[0])

        label['sample_index'] = i
        labels.append(label)

    # convert into DataFrame
    labels_df = pd.DataFrame(labels)

    # move 'sample_index' in the first column
    labels_df = labels_df[['sample_index'] + [col for col in
labels_df.columns if col != 'sample_index']]

    return np.array(generated_images), labels_df

# call the funcion used to generate images
generated_images, labels = generate_inference_images(encoder_model,
decoder_model, sampling_model)

output_folder = 'generated_data1'
os.makedirs(output_folder, exist_ok=True)

normalization_factor = 1.0

for i, image_array in enumerate(generated_images):

    # reduce dimension of the images to 11x11x150
    resized_image = image_array[:-1, :-1, :]

    # transpose to get the right shape 150x11x11
    reshaped_image = np.transpose(resized_image, (2, 0, 1))

    # convert the array int16
    int16_image = (reshaped_image *
normalization_factor).astype(np.int16)

    file_name = f'{i}.npz'
    file_path = os.path.join(output_folder, file_name)
    np.savez(file_path, data = int16_image)

    #print the shape of the array
    print(f'Dimensioni di {i}: {reshaped_image.shape}')

print(f'Le immagini sono state salvate in {output_folder}.')

# save the labels into CSV file
labels.to_csv('labels1.csv', index=False)
print('Le etichette sono state salvate in labels.csv.')
```

Allegato 5: Definizione della metrica personalizzata e del modello di addestramento [4].

```
def custom_metric(y_true, y_pred):
    y_true = tf.cond(tf.math.equal(label_normalization_mode,0),
                     lambda: tf.multiply(y_true, max_labels),
                     lambda: tf.multiply(y_true, std_labels)
+mean_labels)
    y_pred = tf.cond(tf.math.equal(label_normalization_mode,0),
                     lambda: tf.multiply(y_pred, max_labels),
                     lambda: tf.multiply(y_pred, std_labels)
+mean_labels)

    mse = tf.reduce_mean((y_true-y_pred)**2, axis=0)
    mse_baseline = [870.02899169921875, 3828.40234375, 1588.857421875,
0.06771627068519592285156250]
    score = tf.reduce_mean(mse/mse_baseline)

    return score

backbone = EfficientNetLiteB0mod(input_shape=(target_image_size,
target_image_size, 150),
                                 width_coefficient=0.5,
                                 depth_coefficient=0.5,
                                 dropout_rate=0.1
                                 )

model = tf.keras.Sequential([backbone,
                             layers.Flatten(),
                             layers.Dense(4,
kernel_initializer=tf.keras.initializers.GlorotUniform(seed=1509))])


model.compile(
    optimizer=tf.keras.optimizers.Adam(clipnorm=1.),
    loss='mse',
    metrics=[custom_metric],
  )

#Initialize the model with the same weights we employed during the
competition
model.load_weights('efficientnet_lite/initialization_weigths.h5')
model.summary()
```