

Introduzione

Il telerilevamento è una scienza che permette di identificare, misurare ed analizzare le caratteristiche qualitative e quantitative di un determinato oggetto, area o fenomeno senza entrare in contatto diretto con esso. In generale l'oggetto di studio nel telerilevamento è il pianeta Terra in tutte le sue componenti: territorio, acqua e atmosfera. Avendo la possibilità di operare dall'alto, a diverse distanze e in tempi differenti, questa disciplina ha introdotto una nuova filosofia di controllo e d'indagine nello studio del territorio e dei relativi problemi, permettendo di osservare fenomeni non direttamente accessibili e quindi di superare le difficoltà connesse alle campagne di misura a terra, quali grandi sforzi organizzativi, tempo e risorse non sempre disponibili [79, 80, 81].

Negli ultimi anni, in quest'ambito si è fatta molta strada in termini tecnologici e le risoluzioni, sia geometriche che spettrali, dei sensori impiegati, sono nettamente migliorate, permettendo di estendere le applicazioni del telerilevamento all'agricoltura di precisione. L'agricoltura di precisione è una strategia di gestione dell'attività agricola con la quale i dati vengono raccolti, elaborati, analizzati e combinati con altre informazioni per orientare le decisioni in funzione della variabilità spaziale e temporale al fine di migliorare l'efficienza nell'uso delle risorse, la produttività, la qualità, la redditività e la sostenibilità della produzione agricola [118].

La mappatura delle colture (*Crop mapping*) è fondamentale per supportare il processo decisionale e fornire inventari accurati e tempestivi per stimare la produzione e monitorare la crescita dinamica delle colture a varie scale. Tuttavia, la mappatura delle colture *in situ* (sul posto) si rivela spesso costosa e ad alta intensità di lavoro. Il telerilevamento satellitare offre un'alternativa più economica, in grado di fornire serie di dati temporali in grado di catturare ripetutamente le dinamiche della crescita delle colture su larga scala e a intervalli regolarmente rivisitati. Sebbene la maggior parte dei prodotti di tipo culturale esistenti sia generata utilizzando dati di telerilevamento e approcci di apprendimento automatico, l'accuratezza delle previsioni può essere bassa, dato che persistono errori di classificazione dovuti alle somiglianze fenologiche tra le diverse colture e alla complessità dei sistemi agricoli negli scenari reali. Le reti neurali profonde dimostrano un grande potenziale nel catturare i modelli stagionali e le relazioni sequenziali nei dati delle serie temporali nel contesto del loro modo di apprendere le caratteristiche in modo completo e autonomo.

Questa tesi esplora le applicazioni delle reti neurali, nello specifico di reti convoluzionali (CNN), per affrontare il problema della segmentazione semantica su immagini satellitari Sentinel-2, con l'obiettivo principale di mappare su larga scala le colture agricole. La tesi espone lo sviluppo di un sistema capace di apprendere automaticamente le caratteristiche di ogni campo agricolo, utilizzando per l'addestramento il dataset Sentinel2-Munich480 [19]. Questo sistema, applicabile alle immagini satellitari, consente di identificare i campi agricoli e di riconoscere le diverse colture. La tesi espone anche tutta la parte teorica relativa al funzionamento delle reti neurali, sia classiche che convolutive, esponendo anche come vengono rappresentati e acquisiti i dati nel telerilevamento.

La tesi è strutturata su otto capitoli: Il primo capitolo tratta la definizione di rete neurale, di *Deep learning* e di Machine Learning. Viene inoltre illustrato il concetto di dataset ed i metodi che si utilizzano per valutare le prestazioni di un modello di *Deep learning*.

Nel secondo capitolo vengono presentati i framework che permettono lo sviluppo di modelli di *Deep learning*, il linguaggio di programmazione utilizzato per la realizzazione dei modelli ed il concetto di Tensore.

Nel terzo capitolo vengono richiamati i concetti fondamentali del telerilevamento, illustrando come avviene l'acquisizione delle informazioni nel telerilevamento.

Nel quarto capitolo si approfondisce il funzionamento di una rete neurale classica, partendo dalle basi fino a trattare concetti più complessi.

Nel quinto capitolo vengono trattate le reti neurali convolutive, illustrando i principi su cui si basano queste tipologie di reti e gli elementi di cui sono composte.

Nel sesto capitolo vengono trattati i concetti di base relativi alla segmentazione delle immagini. Illustrando anche l'architettura UNet, spesso utilizzata per applicazioni di segmentazione semantica delle immagini.

Il settimo capitolo tratta la sperimentazione, illustrando il processo per la realizzazione di un modello in grado di eseguire la segmentazione semantica su delle immagini satellitari.

Nell'ottavo capitolo vengono discussi i risultati ottenuti, confrontandoli con quelli ottenuti da altri modelli descritti in articoli che utilizzano lo stesso dataset.

Indice

Introduzione	1
1 Intelligenza Artificiale, Machine Learning e Deep Learning	6
1.1 Intelligenza Artificiale	6
1.2 Machine Learning	7
1.2.1 Definizione	7
1.2.2 Le modalità dell'apprendimento	7
1.2.3 Tipologie di problemi	8
1.3 Deep Learning	8
1.3.1 Definizione	8
1.4 I Dataset	8
1.4.1 Tipologie di dataset per il ML	9
1.5 Valutazione dei modelli	10
1.5.1 La Matrice di confusione	10
1.5.2 L'accuratezza	11
1.5.3 Precisione e richiamo	11
1.5.4 F1 score	12
2 Frameworks e linguaggi di programmazione utilizzati	13
2.1 Il linguaggio Python	13
2.2 Frameworks per lo sviluppo di reti neurali	14
2.2.1 Trends dei frameworks	15
2.2.2 TensorFlow	15
2.2.3 Keras	15
2.2.4 Pytorch	16
2.2.5 PyTorch Lightning	16
2.3 Il tensore	16
2.3.1 Struttura e Caratteristiche dei Tensori	16
2.3.2 Rappresentazione dei dati	16
2.3.3 I tensori nelle reti neurali	17
2.3.4 Velocità computazionale dei tensori	18
3 Caratterizzazione remota del suolo Terrestre	20
3.1 Definizione di remote sensing	20
3.1.1 Piattaforme di Telerilevamento	21
3.2 Acquisizione delle informazioni nel telerilevamento	22
3.2.1 Le Radiazioni elettromagnetiche	22
3.2.2 Lo Spettro Elettromagnetico	23
3.2.3 La Radianza e la Riflettanza	24
3.2.4 Firma spettrale	25
3.3 Tipologie di sensori	25

3.4	Immagini Multispettrali e Iperspettrali	26
3.4.1	Visualizzazione delle diverse bande	28
3.5	Copernicus e Sentinel-2	29
3.5.1	Il programma spaziale Europeo Copernicus	29
3.5.2	Le missioni Sentinel	29
3.5.3	Caratteristiche di Sentinel-2	30
3.5.4	Combinazione delle bande di sentinel-2	32
4	Reti Neurali Artificiali (ANN)	35
4.1	La Rete Neurale Biologica	35
4.2	Il Modello McCulloch-Pitts	36
4.3	Il Percettrone	37
4.3.1	L'Algoritmo di Apprendimento del Percettrone	38
4.3.2	Implementazione di un percettrone	39
4.3.3	Limiti del percettrone	43
4.4	Single-layer Perceptron (SLP)	44
4.5	Multi-layer Perceptron (MLP)	45
4.5.1	Rappresentazione matematica del modello	46
4.5.2	Esempio applicativo	48
4.6	Il Gradient Descent	50
4.6.1	Cenni alle derivate parziali	50
4.6.2	Funzionamento del Gradient Descent	50
4.7	Ottimizzatori	52
4.7.1	Full Batch Gradient Descent	53
4.7.2	Stochastic Gradient Descent	53
4.7.3	Mini Batch Gradient Descent	53
4.7.4	Full Batch vs Stochastic vs Mini Batch	54
4.8	L'algoritmo della Backpropagation	54
4.8.1	Il prodotto di Hadamard	55
4.8.2	Funzionamento della Backpropagation	55
4.8.3	Applicazione della Backpropagation	56
4.9	Altre funzione di attivazione	57
4.9.1	Funzione Sigmoide	57
4.9.2	Funzione Tanh (Tangente Iperbolica)	58
4.9.3	ReLU (Rectified Linear Unit)	58
4.9.4	Leaky ReLU	59
4.9.5	ELU (Exponential Linear Unit)	59
4.9.6	Swish	60
4.9.7	Softmax	60
4.10	La Cross-Entropy	61
5	Reti Neurali Convoluzionali (CNN)	62
5.1	Cenni Matematici	62
5.1.1	L'operazione di convoluzione	62
5.1.2	La convoluzione discreta	62
5.1.3	La convoluzione nelle CNN	63
5.1.4	Cross-Correlation	63
5.1.5	La convoluzione in tre dimensioni	64
5.2	Aspetti e parametri della convoluzione	64
5.2.1	Convoluzione su più canali	64

5.2.2	Parametri della convoluzione	65
5.3	Struttura di una CNN	66
5.4	Algoritmi di Pooling	68
5.4.1	Average-Pooling	68
5.4.2	Max-Pooling	68
6	Image segmentation	69
6.1	Introduzione all’image segmentation	69
6.1.1	Tipologie di segmentazioni	70
6.2	L’architettura U-Net	71
6.2.1	skip connections	72
6.2.2	Up-Convolution	72
6.2.3	La funzione di attivazione nella convoluzione di uscita	72
6.2.4	Output della rete	72
7	Sperimentazione	73
7.1	Primo approccio alle reti neurali	73
7.1.1	Descrizione della sperimentazione	73
7.1.2	Implementazione della rete	74
7.1.3	considerazioni sui risultati ottenuti	79
7.1.4	conclusioni	81
7.1.5	Applicazione di una rete convoluzione	82
7.2	Riconoscimento dei campi agricoli	86
7.2.1	Introduzione al problema	86
7.2.2	Il dataset Sentinel2-Munich480	86
7.2.3	Lettura dei dati dal dataset	87
7.2.4	Applicazione dell’UNet	93
7.2.5	Applicazione della UNet con convoluzioni 3D	100
7.2.6	Ignorare la classe unknown	107
7.2.7	Mosaicatura	111
8	Conclusioni	113

Capitolo 1

Intelligenza Artificiale, Machine Learning e Deep Learning

Nella tesi si utilizzeranno spesso i termini **Machine Learning (ML)** e **Deep Learning (DL)**, termini utilizzati come sinonimi di **Intelligenza Artificiale (IA)**. Inoltre si farà anche riferimento al concetto di **rete neurale**, poiché l'argomento centrale della tesi si focalizza sull'applicazione delle reti neurali artificiali (ANN, Artificial Neural Networks).

In questo capitolo verrà data una panoramica veloce di questi termini e di altri aspetti relativi ad essi, al fine di comprendere meglio ciò che sarà poi esposto nei capitoli successivi.

1.1 Intelligenza Artificiale

L'Intelligenza Artificiale è la scienza che si occupa di creare sistemi informatici in grado di svolgere compiti che normalmente richiedono l'intelligenza umana, come il riconoscimento di oggetti, la comprensione del linguaggio naturale, la risoluzione di problemi e l'apprendimento. In altre parole, l'IA è un ramo dell'informatica che studia la progettazione di agenti intelligenti. L'IA, pertanto, è la capacità delle macchine di simulare le capacità cognitive umane, come il ragionamento e la pianificazione [4, 6, 3].

Come si può osservare dalla figura (1.1), il campo dell'intelligenza artificiale è un'area di studio che include molteplici discipline.

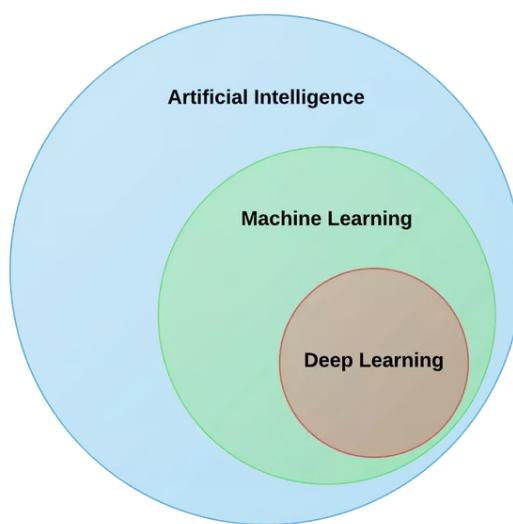


Figura 1.1: Rappresentazione delle categorie [5].

Ciascuna di queste discipline può essere vista come un elemento del livello precedente. Il Machine Learning rappresenta un sottoinsieme dell’Intelligenza Artificiale. Il Deep Learning, a sua volta, è un sottoinsieme del Machine Learning e le reti neurali costituiscono l’elemento fondamentale degli algoritmi di Deep Learning. Si può anche dire che l’Intelligenza Artificiale è la disciplina di base e il Machine Learning e il Deep Learning le tecniche che ne consentono l’applicazione [4, 3].

1.2 Machine Learning

1.2.1 Definizione

Machine Learning comprende un insieme di metodi con cui si allena l’intelligenza artificiale a svolgere delle attività non programmate, a imparare dall’esperienza passata, come fa esattamente l’intelligenza umana, correggendosi e quindi migliorandosi attraverso gli errori commessi. Questa disciplina viene chiamata anche apprendimento automatico, poiché il modello impara dalla propria esperienza, senza la necessità di inserire nuove istruzioni [3].

1.2.2 Le modalità dell’apprendimento

Gli algoritmi di apprendimento automatico possono essere suddivisi in quattro categorie [10, 7]:

- **Apprendimento supervisionato:** vengono presentati al modello una serie di esempi ideali costituiti dalla coppia input-output, in modo che riesca a capire la correlazione tra le entrate e le uscite.
- **Apprendimento non supervisionato:** al contrario dell’apprendimento precedente il modello riceve solo gli input. Deve capire da solo l’output da generare senza potersi confrontare con gli esempi dati.
- **Apprendimento per rinforzo:** Al programma viene fornito un feedback per ogni azione che svolge. Un feedback positivo che si tradurrà in una ricompensa indica un’azione svolta correttamente, al contrario una punizione indicherà un’azione sbagliata.
- **Apprendimento semi-supervisionato:** vengono fornite informazioni incomplete sotto forma di esempi come nell’apprendimento supervisionato e il modello cercherà di prevedere anche quali sono i risultati mancanti.

1.2.3 Tipologie di problemi

Nel Machine Learning, si possono distinguere tre tipologie di problemi [10, 7]:

- La **classificazione**, nella quale gli input sono divisi in due o più classi e il sistema di apprendimento deve produrre un modello in grado di assegnare ad un input una o più classi tra quelle disponibili. Questi tipi di task sono tipicamente affrontati mediante tecniche di apprendimento supervisionato. Un esempio di classificazione è l'assegnamento di una o più etichette ad una immagine in base agli oggetti o soggetti contenuti in essa;
- La **regressione**, concettualmente simile alla classificazione con la differenza che l'output ha un dominio continuo e non discreto. Anch'essa è tipicamente affrontata con l'apprendimento supervisionato.
- Il **clustering**, nel quale, come nella classificazione, un insieme di dati viene diviso in gruppi che però, a differenza di questa, non sono noti a priori. La natura stessa dei problemi appartenenti a questa categoria li rende tipicamente dei task di apprendimento non supervisionato.

1.3 Deep Learning

1.3.1 Definizione

Il Deep Learning è un sottoinsieme del Machine Learning che si occupa di analizzare i dati in maniera profonda, solitamente attraverso una rete di apprendimento che prende decisioni e giunge a conclusioni in base ai dati forniti. Questo metodo è particolarmente adatto a grandi set di dati, in quanto molto più preciso, anche se molto più dispendioso in termini di risorse computazionali e temporali rispetto all'apprendimento automatico. Tra i modelli di apprendimento automatico, trovano larga applicazione le reti neurali [3, 8, 9].

1.4 I Dataset

Spesso nella tesi useremo il termine **dataset** ("insieme di dati" in italiano). Questo termine viene utilizzato per riferirsi ad una collezione strutturata di dati, generalmente di grandi dimensioni e organizzata, correlati a un argomento, tema o settore specifico.

I dataset possono includere diversi tipi di informazioni, come numeri, testo, immagini, video e audio, e possono essere archiviati in vari formati, come CSV, JSON, SQL o altri formati.

I dataset possono essere utilizzati per condurre ricerche di mercato, analizzare i concorrenti, confrontare i prezzi, identificare e studiare le tendenze o addestrare modelli di apprendimento automatico. Questi sono solo alcuni esempi e i dataset sono utili in varie aree e situazioni. Ad esempio, noi utilizzeremo dei dataset per "addestrare" delle reti neurali per uno specifico problema [11, 12].

1.4.1 Tipologie di dataset per il ML

Quando si addestra un modello di ML, vengono utilizzati diversi dataset per applicazioni differenti [13]:

- **Training Dataset:** è il dataset utilizzato per addestrare il modello di Machine Learning. Durante questa fase, il modello apprende le relazioni e i pattern nei dati.
- **Validation Dataset:** è utilizzato per monitorare il modello durante l'addestramento e ridurre il rischio di overfitting, ovvero quando il modello si adatta troppo bene ai dati di addestramento e perde capacità di generalizzare.
- **Test Dataset:** è il dataset utilizzato dopo l'addestramento per valutare le prestazioni finali del modello. Serve a confermare che il modello generalizzi bene su dati completamente nuovi.

Questi dataset possono essere anche ottenuti partendo da uno stesso dataset, dividendo il dataset principale in diversi dataset più piccoli. Tipicamente, dal dataset principale, si utilizza un 60-70% per il training, un 10-20% per la validation e un 10-20% per il test [14, 16].



Figura 1.2: Rappresentazione della suddivisione del dataset [14].

1.5 Valutazione dei modelli

Quasi sempre quando si addestra un modello di ML, occorre un modo per valutare le sue prestazioni.

1.5.1 La Matrice di confusione

Una matrice di confusione (o matrice di errore) è un metodo di visualizzazione per i risultati dell'algoritmo di classificazione. Più specificamente, è una tabella che suddivide il numero di istanze di verità di base di una classe specifica rispetto al numero di istanze di classe previste. In una matrice di confusione, le colonne rappresentano i valori previsti di una data classe, mentre le righe rappresentano i valori effettivi (ad esempio, le verità di base) di una data classe, o viceversa. Questa struttura a griglia è uno strumento utile per visualizzare l'accuratezza della classificazione dei modelli. In quanto permette di visualizzare il numero di previsioni corrette e previsioni errate per tutte le classi una accanto all'altra [21, 22].

Un modello di matrice di confusione standard per un classificatore binario può essere simile a:

		Positive	TP	FN
	Actual value			
Negative		FP		TN
	Positive			Negative
Predicted value				

Figura 1.3: Esempio di una matrice di confusione per classificazione binaria [21] .

La casella in alto a sinistra fornisce il numero di veri positivi (TP), ovvero il numero di previsioni corrette per la classe positiva. Il riquadro sottostante è rappresentato dai falsi positivi (FP), ovvero quei casi di classe negativa erroneamente identificati come casi positivi. In statistica, questi sono anche chiamati errori di tipo I. La casella in alto a destra indica il numero di falsi negativi (FN), i casi effettivamente positivi erroneamente previsti come negativi. Infine, nella casella in basso a destra viene visualizzato il numero di veri negativi (TN), ovvero le istanze effettive della classe negativa previste con precisione. Sommando ciascuno di questi valori si otterebbe il numero totale di previsioni del modello [21, 22].

Una matrice di confusione può essere utilizzata anche per problemi di classificazione multiclasse.

	Walleye	Largemouth Bass	Bluegill	Rainbow Trout
Actual value	TP			
Walleye		TP		
Largemouth Bass			TP	
Bluegill				TP
Rainbow Trout				TP
Predicted value	Walleye	Largemouth Bass	Bluegill	Rainbow Trout

Figura 1.4: Esempio di una matrice di confusione per classificazione multi classe

Tutte le caselle diagonali indicano i veri positivi previsti. Le altre caselle forniscono le quantità per i falsi positivi, i falsi negativi ed i veri negativi a seconda della classe che si sceglie di mettere a fuoco. La matrice di confusione può servire a calcolare diverse metriche di valutazione, come per esempio il tasso di errore, l'accuratezza, la precisione, il richiamo o sensibilità (Recall) e l'F1-score. Mettendo la matrice di confusione e le metriche ad essa associate sotto osservazione, è possibile identificare le aree in cui il modello presenta criticità. Così è possibile adottare misure specifiche per aumentare la precisione delle previsioni [21, 22].

1.5.2 L'accuratezza

L'accuratezza rappresenta la percentuale di predizioni corrette rispetto al totale delle predizioni effettuate. Questo valore varia da 0, che rappresenta lo scenario peggiore in cui tutte le predizioni sono errate, a 1, che corrisponde alla massima accuratezza con tutte le predizioni corrette. L'accuratezza si calcola utilizzando la seguente formula:

$$ACC = \frac{\text{Casi corretti}}{\text{Numero totale di casi}} = \frac{TP + TN}{TP + TN + FP + FN} \quad (1.1)$$

Tuttavia, l'accuratezza del modello non è una metrica di valutazione completamente informativa per i classificatori. Immaginiamo, ad esempio, di eseguire un classificatore su un set di dati di 100 istanze. La matrice di confusione del modello mostra solo un falso negativo e nessun falso positivo; il modello classifica correttamente ogni altra istanza di dati. Pertanto, il modello ha una precisione del 99%. Sebbene apparentemente desiderabile, l'elevata precisione non è di per sé indicativa di eccellenti prestazioni del modello. Ad esempio, supponiamo che il nostro modello miri a classificare malattie altamente contagiose. Questa errata classificazione dell'1% rappresenta un rischio enorme. Pertanto, è possibile utilizzare altre metriche di valutazione per fornire un quadro migliore delle prestazioni dell'algoritmo di classificazione [21, 22].

1.5.3 Precisione e richiamo

La precisione è la proporzione di stime di classe positive che appartengono effettivamente alla classe in questione. Un altro modo per comprendere la precisione è che misura la probabilità che un'istanza scelta a caso appartenga ad una certa classe. La precisione può anche essere chiamata valore previsto positivo (PPV) [21, 22]. È rappresentato dall'equazione:

$$\text{Precisione} = \frac{TP}{TP + FP} \quad (1.2)$$

Mentre il richiamo indica la percentuale di istanze di classe rilevate da un modello. In altre parole, indica la percentuale di previsioni positive per una data classe rispetto a tutte le istanze effettive di quella classe [21, 22]. Il richiamo è noto anche come sensibilità o tasso di veri positivi ed è rappresentato dall'equazione:

$$\text{Richiamo} = \frac{TP}{TP + FN} \quad (1.3)$$

1.5.4 F1 score

F1 score, chiamato anche *F-score*, *F-measure* o *media armonica di precisione e richiamo*, combina precisione e richiamo per rappresentare l'accuratezza totale di un modello rispetto alla classe. Utilizzando questi due valori, si può calcolare l'F1 score con l'equazione, in cui P indica la precisione (PPV) e R indica il richiamo (sensibilità):

$$F = \frac{2PR}{P + R} \quad (1.4)$$

F1 score è particolarmente utile per set di dati sbilanciati, in cui il compromesso tra precisione e richiamo può essere più evidente. Ad esempio, supponiamo di avere un classificatore che prevede la probabilità di una malattia rara. Un modello previsionale in cui nessuno nel nostro set di dati di test risulta affetto dalla malattia può avere una precisione perfetta ma zero richiami. Nel frattempo, un modello che preveda che tutti i soggetti del nostro set di dati siano affetti dalla malattia restituirebbe un richiamo perfetto ma una precisione pari alla percentuale di persone effettivamente affette dalla malattia (ad esempio 0,00001% se solo uno su dieci milioni ha la malattia). F1 score è un mezzo per bilanciare questi due valori per ottenere una visione più olistica delle prestazioni di un classificatore [21, 22].

Capitolo 2

Frameworks e linguaggi di programmazione utilizzati

2.1 Il linguaggio Python



Per la realizzazione dei vari esperimenti e illustrazioni esposti nel corso della tesi, è stato utilizzato il linguaggio di programmazione Python, un linguaggio di programmazione utilizzato per lo sviluppo dell'analisi empirica. Python è un linguaggio di programmazione ad alto livello noto per la sua: dinamicità, semplicità e flessibilità. Esso è un linguaggio interpretato in grado di supportare paradigmi di programmazione come: la programmazione procedurale, la programmazione orientata agli oggetti e la programmazione funzionale. Vanta la possibilità di interfacciarsi con diversi tipi di system call, librerie e sistemi a finestra ed è estensibile in C o C++. Può anche essere utilizzato come linguaggio di estensione per applicazioni che necessitano di un'interfaccia programmabile. Python è in grado di combinare un potenziale notevole ad una sintassi molto chiara e infine si tratta di un linguaggio portatile: funziona su molte varianti di Unix, su MacOS e su Windows. La nascita del linguaggio è attribuibile a Guido van Rossum, programmatore olandese laureatosi all'Università di Amsterdam con un Master in Matematica ed Informatica e conosciuto come il Benevolent Dictator for Life di Python. Con il tempo, Python, è diventato uno tra i più utilizzati linguaggi di programmazione per l'ambito della *data science* e *data analytics*, con la crescente diffusione di progetti legati al Machine learning, Cloud computing e Big data. Ciò grazie anche alla vasta raccolta di librerie [23].

Per la realizzazione dei vari programmi, alcune delle principali librerie che sono state utilizzate sono:

- **Numpy**, una libreria che fornisce supporto per array multidimensionali e funzioni matematiche ad alte prestazioni per operare su di esse. È particolarmente utile per manipolare grandi quantità di dati numerici in modo efficiente;
- **Matplotlib** è una libreria per la visualizzazione dei dati. Consente di realizzare grafici a linee, istogrammi, scatter plot e altro ancora;
- **Seaborn** è una libreria basata su Matplotlib che offre un'interfaccia più intuitiva per la realizzazione di grafici statisticamente informativi, come le heatmap e le matrici di confusione.
- **Scikit-learn** è una libreria per l'apprendimento automatico. Fornisce una vasta raccolta di strumenti per la classificazione, la regressione e il clustering;
- **Pandas** è una libreria progettata per la manipolazione e l'analisi dei dati strutturati;

- Pytorch è una libreria open-source per l'apprendimento automatico e il deep learning. Fornisce strumenti per creare reti neurali e algoritmi di apprendimento basati su tensori.

2.2 Frameworks per lo sviluppo di reti neurali

Le reti neurali artificiali hanno dimostrato di essere all'avanguardia in molti casi di apprendimento supervisionato, ma programmare manualmente una rete neurale può essere un compito impegnativo. Con l'aumento dell'interesse per il deep learning negli ultimi anni, si è assistito a un'esplosione di strumenti di apprendimento automatico. Negli ultimi anni sono stati introdotti e sviluppati a ritmo sostenuto framework di deep learning come PyTorch, TensorFlow, Keras, Chainer e altri.

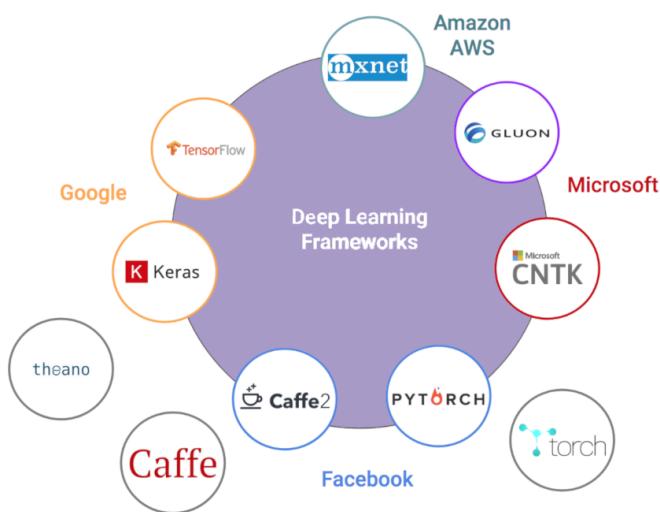


Figura 2.1: Alcuni dei principali framework [26].

Questi framework forniscono unità di rete neurale, funzioni di costo e ottimizzatori per assemblare e addestrare modelli di rete neurale, permettendo di semplificare e velocizzare lo sviluppo di questi modelli. Alcuni di questi framework come Tensorflow, PyTorch e Keras; Sono disponibili per il linguaggio di programmazione Python, sotto forma di libreria, fornendo un API per la progettazione e l'addestramento di modelli.

2.2.1 Trends dei frameworks

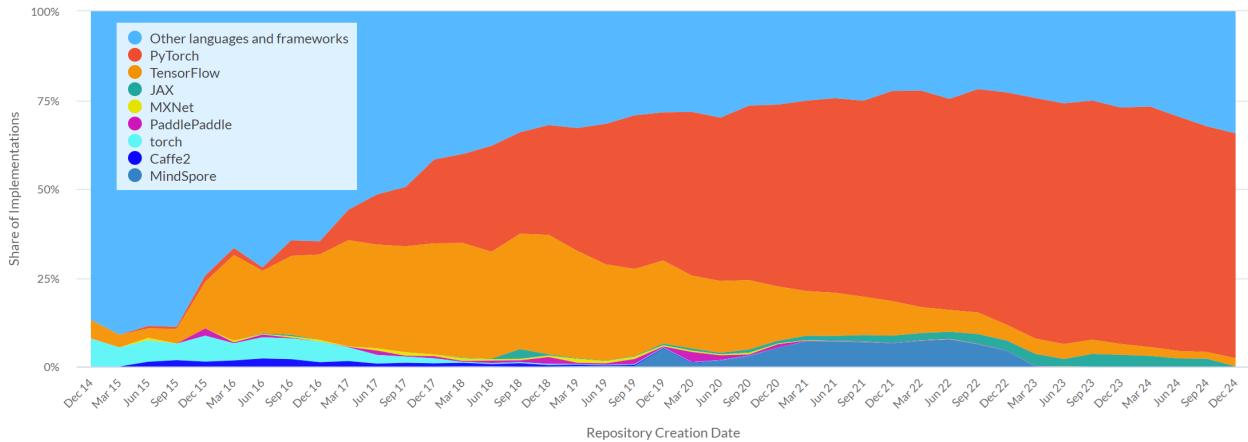


Figura 2.2: Trends dei frameworks [27].

Il grafico mostra l'andamento dei framework utilizzati nei repository GitHub dedicati alle implementazioni di articoli scientifici. L'asse temporale indica la data di creazione dei repository, consentendo di osservare come la popolarità dei vari framework sia cambiata nel tempo. Tra i più utilizzati spiccano TensorFlow e PyTorch, due strumenti fondamentali e molto apprezzati nel campo dell'apprendimento automatico. Pertanto, per la realizzazione delle reti neurali, utilizzeremo Pytorch, in quanto è uno dei migliori framework per chi si avvicina per la prima volta a questo ambito.

2.2.2 TensorFlow



Tensorflow [24, 25, 26] è una libreria open source sviluppata da Google che fornisce risorse per la creazione di modelli per l'apprendimento automatico. Rilasciata ufficialmente nel 2015, è diventata velocemente una delle più popolari soprattutto grazie al supporto per diversi linguaggi di programmazione, come Python, C++ e R. Inoltre dispone di un'ottima documentazione e linee guida, che la rendono un'ottima scelta per chi si avvicina per la prima volta a tale mondo.

Tra i principali vantaggi, TensorFlow offre:

- Supporto per l'esecuzione su GPU e TPU, che permette di accelerare i tempi di addestramento.
- Un ecosistema completo, che include TensorBoard per la visualizzazione e TensorFlow Lite per l'ottimizzazione su dispositivi mobili.

2.2.3 Keras



Keras [24, 25, 26] è una libreria open source scritta in Python e può essere eseguita su TensorFlow (oltre che su CNTK e Theano). L'interfaccia di TensorFlow può essere un po' ostica, poiché si tratta di una libreria di basso livello e i nuovi utenti potrebbero avere difficoltà a comprendere alcune implementazioni. Keras, invece, è un'API di alto livello, sviluppata con l'obiettivo di consentire una sperimentazione rapida. Quindi, se si vogliono ottenere risultati rapidi, Keras si occuperà automaticamente dei compiti principali e genererà l'output.

2.2.4 Pytorch



PyTorch [24, 25, 26] è una libreria open source sviluppata da Facebook e introdotta per la prima volta nel 2016. Questa libreria è particolarmente apprezzata per la sua flessibilità con un'attenta considerazione delle prestazioni e per l'approccio dinamico al calcolo dei grafi computazionali.

Oggi, la maggior parte del suo nucleo è scritto in C++, uno dei motivi principali per cui PyTorch può ottenere un overhead molto più basso rispetto ad altri framework. Ad oggi, PyTorch sembra essere il più adatto a ridurre drasticamente il ciclo di progettazione, addestramento e test di nuove reti neurali per scopi specifici. Per questo è diventato molto popolare nelle comunità di ricerca.

2.2.5 PyTorch Lightning



PyTorch Lightning [28, 29] è una libreria Python open-source che fornisce un'interfaccia di alto livello per PyTorch. È un framework leggero e ad alte prestazioni che organizza il codice PyTorch per disaccoppiare la ricerca dall'ingegneria, rendendo così gli esperimenti di deep learning più facili da leggere e riprodurre, accelerando così i tempi di sviluppo dei modelli.

È stato progettato per creare modelli scalabili di deep learning che possono essere facilmente eseguiti su hardware distribuito. Inoltre, PyTorch Lightning possiede una struttura modulare che semplifica la gestione di training, validazione e logging.

2.3 Il tensore

Quando si lavora con gli algoritmi di machine learning, spesso occorre un modo per rappresentare numericamente informazioni complessi come: suoni, immagini, testo o altre informazioni. In framework di machine learning e deep learning come TensorFlow e PyTorch, la rappresentazione numerica delle informazioni è ottenuta attraverso l'uso dei **Tensori**, che fungono da unità fondamentale per il calcolo sulla piattaforma.

Un Tensore può essere definito come un oggetto matematico, una struttura dati multidimensionale che generalizza il concetto di scalare, vettore e matrice a n -dimensioni. Nel contesto della *data science*, i tensori sono matrici multidimensionali di numeri che rappresentano dati complessi [30, 31, 32, 33, 34, 35].

2.3.1 Struttura e Caratteristiche dei Tensori

Un tensore è definito da due proprietà principali che ne determinano la struttura [30] :

- **Rank (o dimensione)**: Il rank di un tensore indica il numero di assi o direzioni lungo cui i dati sono organizzati.
- **Shape (o forma)**: La shape di un tensore è una tupla che specifica il numero di elementi lungo ogni asse.

2.3.2 Rappresentazione dei dati

A seconda della loro dimensione (rank) e della forma (shape), i tensori possono modellare dati di diversa natura [30]:

- **Scalare**: Un tensore scalare rappresenta un singolo valore numerico. Non ha direzioni o assi, ed è quindi di dimensione 0. Ad esempio potrebbe essere il numero 5.5 .

- **Vettore:** Un vettore, ad esempio $[1 \ 2 \ 3]$, è una sequenza ordinata di numeri disposti lungo un unico asse, e può essere rappresentato da un tensore di dimensione 1.
- **Matrice:** una matrice come $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ può essere vista come un tensore di dimensione 2.
- **Strutture 3D e oltre:** tutte le strutture matematiche con tre o più dimensioni, ad esempio come $\begin{bmatrix} [1 \ 2] & [5 \ 6] \\ [3 \ 4] & [7 \ 8] \end{bmatrix}$, vengono rappresentati come tensori di dimensione 3 o superiore.

La figura (2.3) mostra una rappresentazione grafica dei tensori al variare della dimensione

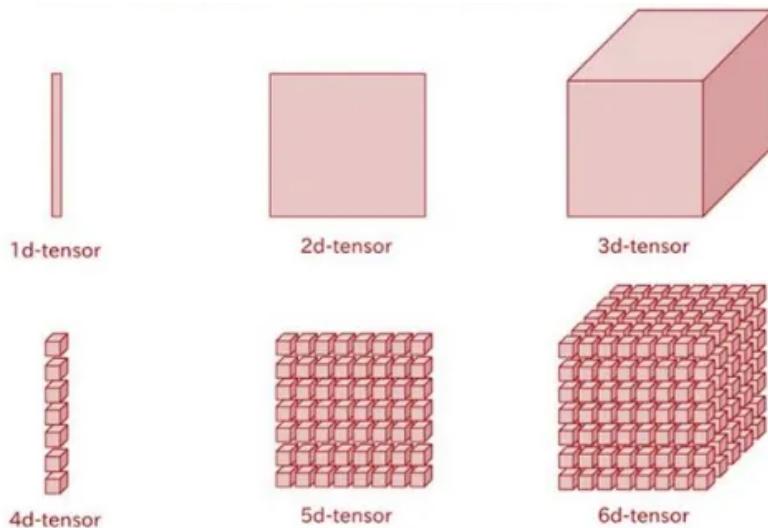


Figura 2.3: Rappresentazione grafica dei tensori [34] .

Ad esempio, se volessi utilizzare in python un tensore di pytorch per rappresentare 2 immagini RGB 3x3, il tensore apparirebbe così:

```

1 images = torch.tensor([
2     [
3         [[255, 0, 0], [255, 255, 0], [0, 255, 0]],
4         [[0, 0, 255], [0, 255, 255], [0, 0, 255]],
5         [[255, 255, 255], [128, 128, 128], [0, 0, 0]]
6     ],
7     [
8         [[0, 0, 0], [128, 128, 128], [255, 255, 255]],
9         [[255, 0, 255], [255, 255, 0], [0, 255, 0]],
10        [[0, 0, 255], [255, 255, 255], [128, 128, 128]]
11    ]
12])

```

Questo tensore così creato avrebbe shape $(2, 3, 3, 3)$.

2.3.3 I tensori nelle reti neurali

Come si vedrà più avanti, il funzionamento delle reti neurali si basa su delle operazioni matematiche lineari, come ad esempio il prodotto matriciale. I tensori funzionano in modo simile ai *ndarray* usati in NumPy, ma a differenza dei *ndarray*, che possono essere eseguiti solo su unità

di elaborazione centrali (CPU), i tensori possono essere eseguiti anche su appositi acceleratori hardware, come le unità di elaborazione grafica (GPU) e unità di elaborazione per Tensori (TPU). Le GPU e TPU consentono un calcolo notevolmente più veloce rispetto alle CPU, il che rappresenta un grande vantaggio dati gli enormi volumi di dati e l'elaborazione parallela tipici del deep learning. Riassumendo, utilizzare i tensori per rappresentare le informazioni e svolgere calcoli, permette di sfruttare l'accelerazione hardware offerta da GPU e TPU per parallelizzare le operazioni matematiche e svolgerle in modo efficiente. I tensori, oltre ad essere utilizzati per codificare gli input e gli output di un modello di una rete neurale, vengono utilizzati anche per codificare i parametri del modello: pesi, pregiudizi e gradienti che vengono "appresi". Questa proprietà dei tensori consente la differenziazione automatica (calcolo automatico delle derivate), una delle funzioni più importanti di PyTorch [30, 31, 32, 33, 34, 35].

2.3.4 Velocità computazionale dei tensori

Per dare idea dei vantaggi computazionali dell'utilizzo dei tensori, realizziamo un test in cui confrontiamo le prestazioni degli array *NumPy* e dei tensori di *PyTorch* nella moltiplicazione tra matrici. In questo test inizializziamo casualmente due *arrays/tensors* 4D e misureremo il tempo necessario a svolgere la moltiplicazione tra i due. Come dispositivo di accelerazione hardware utilizzeremo una GPU, la cui selezione avviene tramite il comando `torch.device("cuda")`. Qualora si volesse selezionare una specifica GPU, è possibile aggiungere un numero dopo "cuda" il numero della GPU (ad esempio "cuda:1" per selezionare la seconda GPU). Se il numero viene omesso, il comando utilizzerà automaticamente la GPU "cuda:0". Il codice del test è il seguente:

```

1 import torch
2 import numpy as np
3 import time
4
5 dim_max = 110
6 shapes = []
7 device = torch.device("cuda")
8
9 for dim in range(5, dim_max + 1, 5):
10     shapes.append((dim, dim, dim, dim))
11
12 for shape in shapes:
13     array1 = np.random.rand(*shape)
14     array2 = np.random.rand(*shape)
15     tensor1 = torch.rand(*shape).to(device)
16     tensor2 = torch.rand(*shape).to(device)
17
18     start_time = time.perf_counter()
19     np.matmul(array1, array2)
20     dt = time.perf_counter() - start_time
21     cpu_times.append(dt)
22
23     torch.cuda.synchronize()
24     start_time = time.perf_counter()
25     torch.matmul(tensor1, tensor2)
26     torch.cuda.synchronize()
27     gpu_times.append(time.perf_counter() - start_time)
28 plot(shapes, cpu_times, gpu_times)
```

Eseguito il test di confronto, questi sono i risultati che otteniamo:

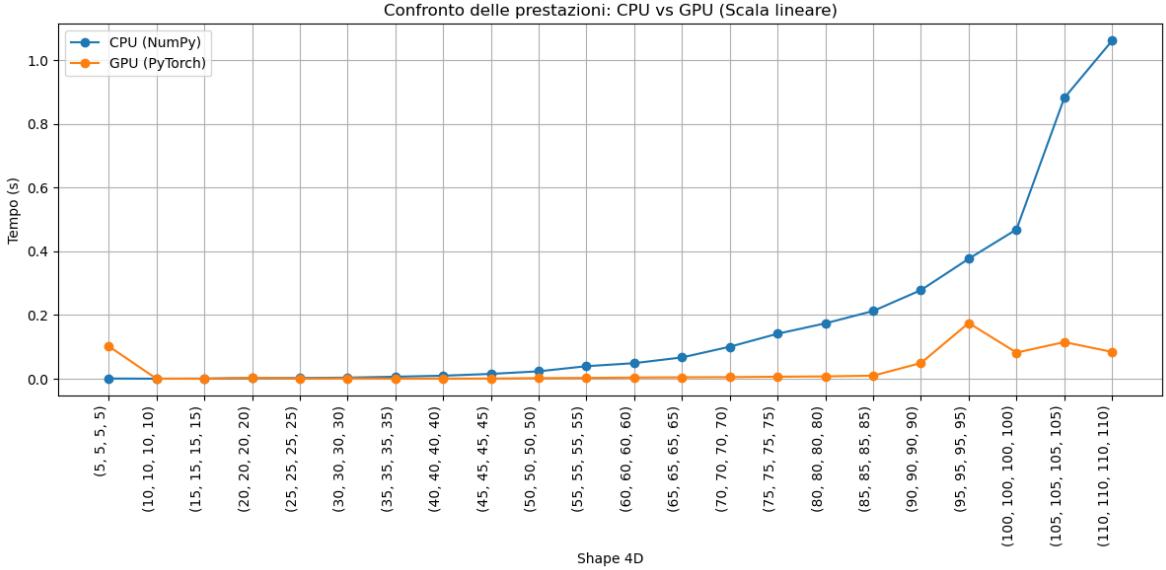


Figura 2.4: Grafico lineare del paragone array-tensore

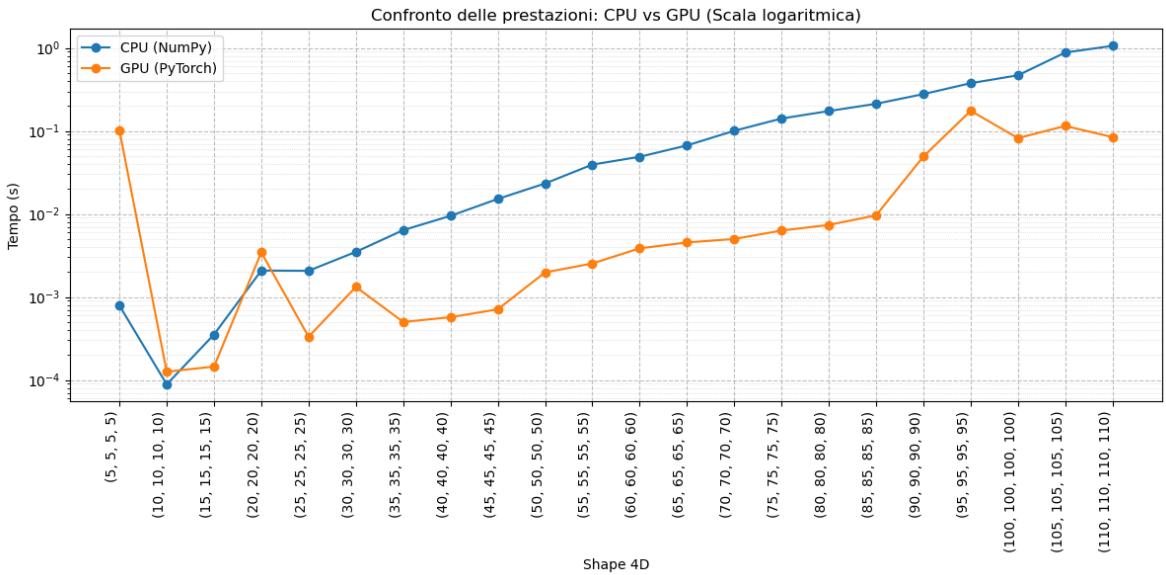


Figura 2.5: Grafico logaritmico del paragone array-tensore

Come possiamo osservare, per dimensioni ridotte, i calcoli eseguiti con gli array di *NumPy* risultano essere molto più veloci rispetto a quelli svolti con i tensori di *PyTorch*. Ma all'aumentare delle dimensioni, i calcoli eseguiti su GPU risultano essere da 24 a 52 volte più veloci rispetto a quelli su CPU. Le prestazioni possono dipendere sia dall'implementazione del codice, sia dall'hardware utilizzato. Ma questa velocità di elaborazione offerta dalle GPU è uno dei motivi principali per cui i tensori (e non gli array) sono ampiamente utilizzati nel deep learning per rappresentare i dati ed eseguire operazioni matematiche.

Capitolo 3

Caratterizzazione remota del suolo Terrestre

In questo capitolo verrà trattato il modo in cui si ottengono le informazioni sulla superficie terrestre. Nel particolare si parlerà della caratterizzazione remota del suolo tramite il telerilevamento (Remote Sensing, RS) e dei principi fisici che ci sono dietro. Inoltre verranno illustrati i satelliti sentinel-2 e le loro caratteristiche. In quanto nella parte sperimentale si utilizzeranno dati di questi satelliti.

3.1 Definizione di remote sensing

Il **remote sensing** (o telerilevamento) [76, 77, 78, 79, 80] è la disciplina tecnico-scientifica che si occupa di acquisire informazioni di carattere spettrale, spaziale e temporale su oggetti materiali, su una determinata area o relativamente ad uno specifico fenomeno, senza avere contatto fisico con l'oggetto di studio, o con il fenomeno in esame. Il telerilevamento è, dunque, la misurazione o l'acquisizione di informazioni di alcune proprietà di un oggetto o fenomeno, da un dispositivo di registrazione che non è in contatto con l'oggetto o il fenomeno in esame. I dispositivi che vengono utilizzati in questo ambito sono diversi: fotocamere, laser e ricevitori a radiofrequenza, sistemi radar.

Negli ultimi decenni il telerilevamento ha subito degli sviluppi tali da essere, oggigiorno, uno strumento fondamentale per la raccolta di informazioni su quasi ogni aspetto della terra. Il telerilevamento trova applicazione in numerosi contesti accomunati dalla necessità di acquisire dati relativi ad aree molto estese. Tra le principali applicazioni si includono:

- Mappatura e analisi dell'uso del suolo;
- Monitoraggio della vegetazione in ambito agricolo e forestale;
- Monitoraggio delle risorse idriche, della neve e del ghiaccio;
- Indagini archeologiche;
- Gestione di eventi calamitosi;
- Gestione delle risorse costiere [78, 84].

Ogni applicazione ha necessità di sensori con caratteristiche tecniche differenti, con specifiche capacità di risoluzione spettrale, risoluzione spaziale, risoluzione radiometrica e risoluzione temporale. Inoltre, i sistemi di telerilevamento possono fornire dati e informazioni in aree in cui l'accesso è difficile a causa della conformazione del terreno o delle condizioni meteorologiche [82].

Nel telerilevamento è inoltre compresa anche l'analisi e l'interpretazione di dati e immagini. Questo aspetto è fondamentale in quanto per poter cogliere le informazioni chiave per l'obiettivo che si sta perseggiando bisogna avere buona comprensione della base fisica e del processo di acquisizione, nonché di una solida conoscenza degli algoritmi utilizzati per elaborare i dati. I dati vengono acquisiti dai sensori, i quali devono essere installati su specifiche piattaforme, mezzi o veicoli. Droni, aerei e satelliti raccolgono la maggior parte dei dati ma molti di questi strumenti possono essere installati anche su piattaforme terrestri, come autocarri e trattori.

3.1.1 Piattaforme di Telerilevamento

I dati di telerilevamento vengono acquisiti da sensori installati su piattaforme specifiche, che si classificano in tre categorie principali [83, 84]:

- **Spaceborne:** Satelliti come Sentinel-2 offrono dati a bassa risoluzione ma un'ampia copertura spaziale [19].
- **Airborne:** UAV (Unmanned Aerial Vehicle) e aerei più versatili, consentono acquisizioni ad alta risoluzione spaziale.
- **Ground-based:** Dispositivi a terra forniscono dati ad altissima risoluzione per dettagli locali.

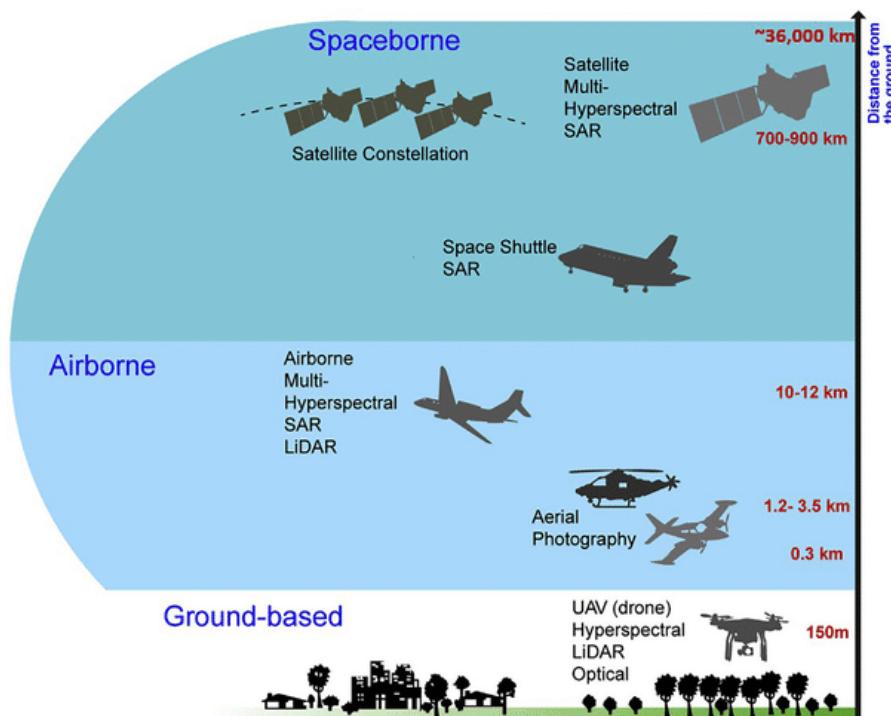


Figura 3.1: Rappresentazione delle piattaforme [83].

3.2 Acquisizione delle informazioni nel telerilevamento

3.2.1 Le Radiazioni elettromagnetiche

L'acquisizione delle informazioni da remoto su un territorio è possibile attraverso l'uso di radiazioni elettromagnetiche (EMR) [86], che vengono emesse o riflesse dagli oggetti osservati. Una radiazione elettromagnetica è una perturbazione di natura simultaneamente elettrica e magnetica che si propaga nello spazio e che può trasportare energia.

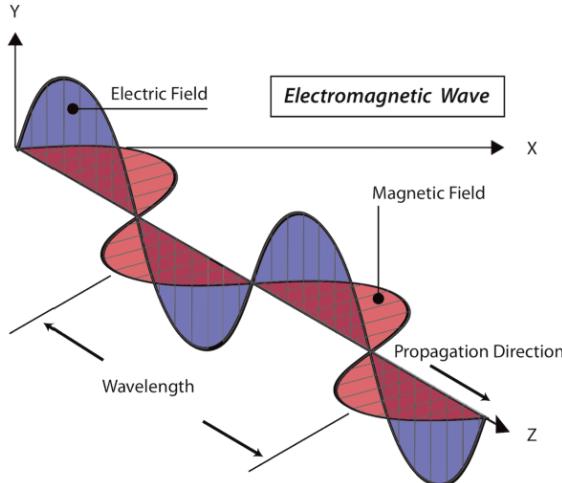


Figura 3.2: Rappresentazione dell'onda elettromagnetica [87].

Un'onda elettromagnetica è caratterizzata da tre parametri fondamentali :

- **Lunghezza d'onda (λ)** la quale esprime la distanza tra due creste d'onda consecutive. La lunghezza d'onda si misura in metri [m], o in sottomultipli del metro, come i nanometri (nm , 10^{-9} metri) o i micrometri (μm , 10^{-6} metri);
- **Frequenza (f)**, cioè il numero dei picchi d'onda che passano in un punto in un certo intervallo di tempo t ; la frequenza è di solito misurata in hertz [Hz], che è equivalente ad un ciclo al secondo;
- **Aampiezza A**, che è l'altezza di ogni picco d'onda.

La frequenza e la lunghezza d'onda sono legate da una relazione, che è definita come:

$$\lambda = \frac{C}{f} \quad (3.1)$$

dove C è una costante che ha valore $299.792.458 \text{ m/s}$. Questa costante è la velocità della luce e rappresenta la velocità con cui si propaga l'onda elettromagnetica attraverso lo spazio.

3.2.2 Lo Spettro Elettromagnetico

La figura (3.3) rappresenta lo spettro elettromagnetico (o spettro EM), una distribuzione monodimensionale continua dell'energia elettromagnetica, ordinata per lunghezze d'onda (λ) crescenti. In pratica, lo spettro elettromagnetico è l'insieme di tutte le possibili frequenze delle onde elettromagnetiche [89].

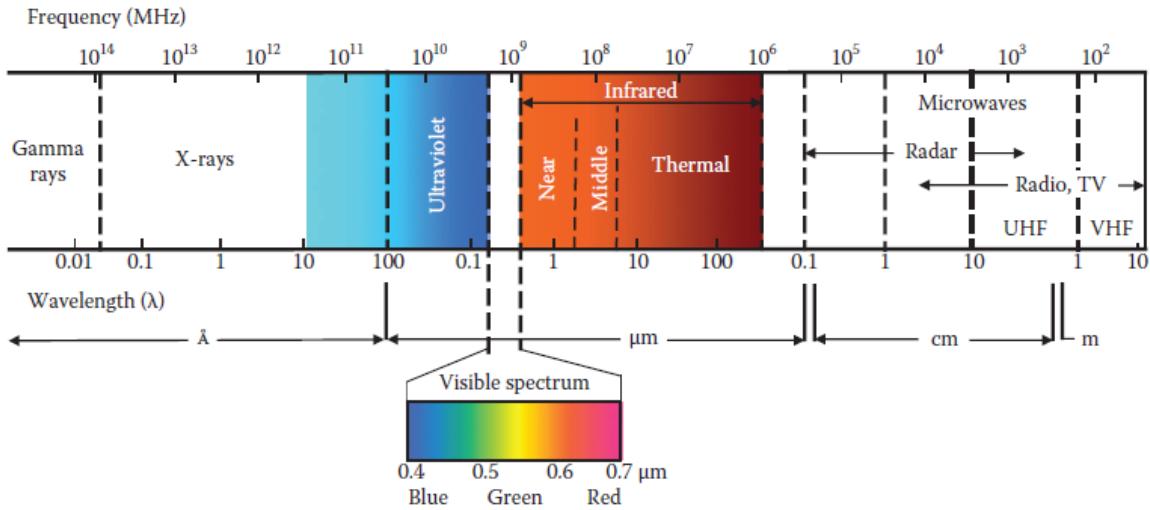


Figura 3.3: Bande dello spettro elettromagnetico [88] .

Come si può osservare, lo spettro è diviso in sette intervalli (anche detti bande), ciascuna della quali racchiude un insieme di frequenze (o lunghezze d'onda) appartenenti alla stessa tipologia di radiazione.

Tipologia di radiazione	Frequenza	Lunghezza d'onda
Onde Radio	≤ 300 MHz	≥ 1 m
Microonde	300 MHz – 300 GHz	1 m – 1 mm
Infrarossi	300 GHz – 428 THz	1 mm – 700 nm
Luce Visibile	428 THz – 749 THz	700 nm – 400 nm
Ultravioletti	749 THz – 30 PHz	400 nm – 10 nm
Raggi X	30 PHz – 300 EHHz	10 nm – 1 pm
Raggi Gamma	≥ 300 EHHz	≤ 1 pm

Tabella 3.1: Tabella riassuntiva delle frequenze di ogni tipologia di radiazione.

Lo spettro del visibile (visible spectrum) è quella parte dello spettro elettromagnetico, compresa tra i 400 nm (viola) e 700 nm (rosso), che è percepibile all'occhio umano.

Tra le diverse regioni spettrali, quelle che vengono principalmente utilizzate per l'acquisizione di informazioni su un territorio sono: l'infrarosso (Infrared), il visibile (visible) e l'ultravioletto (Ultraviolet).

3.2.3 La Radianza e la Riflettanza

Nel telerilevamento, quello che viene misurato dai sensori è la Radianza e Riflettanza. La radianza è definita come la quantità di radiazione eletromagnetica riflessa (o trasmessa) per unità di superficie e di angolo solido (angolo nello spazio tridimensionale). La radianza rappresenta una grandezza fondamentale nel telerilevamento in quanto molto utile per quantificare la luce riflessa da un oggetto che viene ricevuta da un sensore rivolto verso di essa. Questa grandezza fisica è legata sia alla geometria dell'osservazione, sia alle caratteristiche del sensore e permette di descrivere come la radiazione si distribuisce nello spazio [96, 81].

La radianza è caratterizzata dalla seguente formula:

$$L = \frac{P}{A\Omega \cos(\theta)} \left[\frac{W}{m^2 sr} \right] \quad (3.2)$$

dove:

- L è la radianza;
- P è la potenza in watt;
- θ è l'angolo compreso tra la normale alla superficie e la direzione specificata;
- A è la superficie emittente;
- Ω è l'angolo solido.

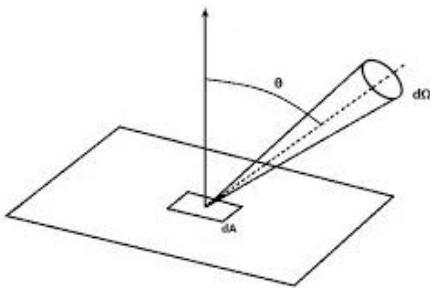


Figura 3.4: Rappresentazione della radianza [97].

La riflettanza invece è il rapporto tra la quantità di radiazione emessa (ovvero che colpisce una superficie) e la quantità di radiazione riflessa (o ricevuta) dalla stessa ed è quindi un numero puro generalmente minore di uno. Questa grandezza è indispensabile per l'individuazione e la discriminazione dei campioni oggetto di analisi, in quanto consente di svincolarsi completamente da condizioni variabili nel tempo (che influenzano la radianza) al momento dell'acquisizione, rendendo confrontabili misure condotte in momenti diversi [95, 81]. La riflettanza può essere espressa come:

$$\rho = \frac{\Phi_r}{\Phi_0} \quad (3.3)$$

dove:

- ρ è la riflettanza;
- Φ_r il flusso luminoso riflesso;
- Φ_0 il flusso luminoso incidente;

Ciò che viene direttamente misurato dal sensore è la radianza, per questo è necessaria una corretta calibrazione del sensore al fine di ottenere dati confrontabili.

3.2.4 Firma spettrale

Quando la riflettanza o la radianza è calcolata su tutte le frequenze dello spettro, si parla allora di **firma spettrale**.

Ogni oggetto o unità di territorio (roccia, vegetazione...) ha la propria **firma spettrale** (o impronta digitale spettrale), che specifica il modo in cui le lunghezze d'onda della luce vengono assorbite, riflesse o trasmesse attraverso quell'oggetto [90].

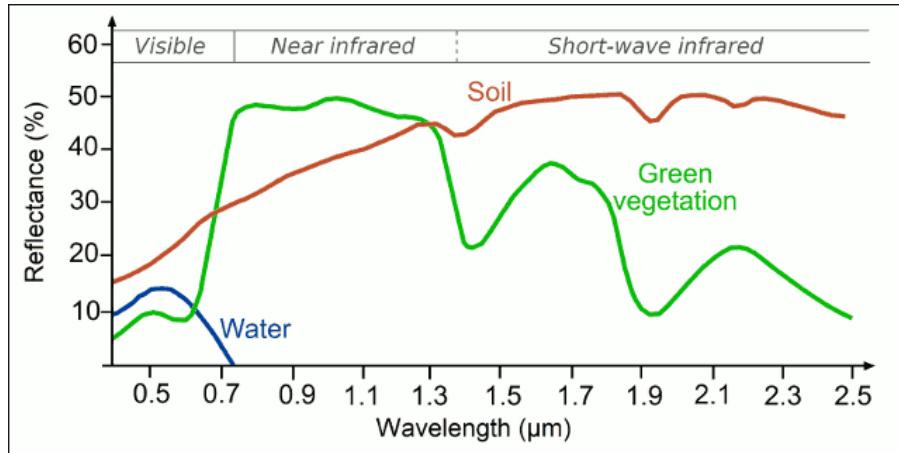


Figura 3.5: Esempi di firme spettrali per acqua, suolo, vegetazione [77].

Ad esempio i canali utilizzati per la copertura della vegetazione catturano l'intensità della luce dal visibile all'infrarosso, consentendo di valutare l'attività fotosintetica e lo stato delle piante. I canali per il monitoraggio delle superfici terrestri possono registrare l'energia in una gamma più ampia, che comprende gli ultravioletti e le microonde.

3.3 Tipologie di sensori

I sensori possono essere classificati in base a diversi fattori. Una delle classificazioni più comuni vede i sensori suddivisi in due gruppi: sensori passivi e sensori attivi.

- I **sensori passivi** si limitano a misurare la radiazione elettromagnetica derivata da fonti esterne: ad esempio, l'energia della radiazione solare riflessa dalla superficie terrestre o l'energia direttamente emanata dalla Terra, ovvero la Radianza. Questi sensori sono stati ampiamente utilizzati nel telerilevamento negli ultimi decenni e includono: fotocamere, scanner elettro-ottici e radiometri a microonde.

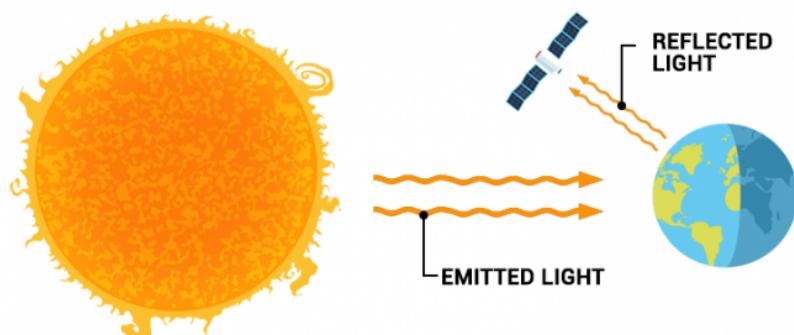


Figura 3.6: Illustrazione funzionamento sensori passivi [82]

- I **sensori attivi** sono invece quelli che misurano la radiazione emessa da una fonte di energia incorporata nello strumento stesso e che viene riflessa dalle superfici inquadrati. Ovvero che si basano sul concetto della riflettanza. Questi sono in grado di inviare impulsi e registrare in seguito la loro riflessione per caratterizzare gli oggetti. Queste tipologie di sensori sono ad esempio radar o LiDAR.

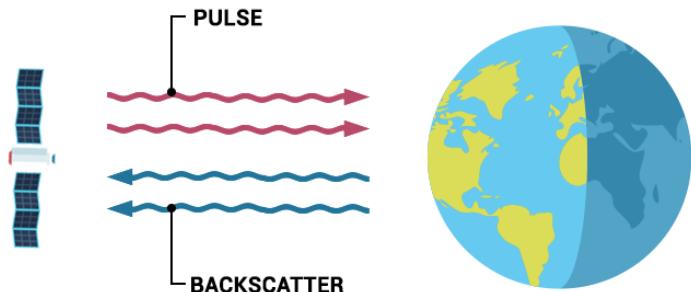


Figura 3.7: Illustrazione funzionamento sensori attivi [82]

I sensori possono essere classificati anche in base al numero di bande che sono in grado di acquisire, i sensori possono essere multispettrali o iperspettrali.

- I **sensori multispettrali** di solito riescono ad acquisire da 3 a 10 bande circa. Le bande che vengono normalmente acquisite sono quelle del visibile, rosso, verde, blu, e quella del vicino infrarosso.
- I **sensori iperspettrali**, invece, misurano l'energia in bande più strette e numerose rispetto ai sensori multispettrali. Le immagini iperspettrali, infatti, possono contenere fino a 200 (o più) bande spettrali.

3.4 Immagini Multispettrali e Iperspettrali

Nelle immagini multispettrali, così come in quelle iperspettrali, ogni pixel dell'immagine non è costituito da un solo valore monocromatico, come nel caso delle immagini pancromatiche (a scala di grigi), o da una terna di valori, come nel caso delle immagini a colori RGB, ma da un insieme di valori appartenenti allo spettro elettromagnetico. Nella scienza del processamento delle immagini, questi valori sono anche noti come **Canale** [81, 109, 110, 77].

A differenza delle immagini multispettrali, le iperspettrali possiedono un numero elevato di bande, che rappresentano intervalli discreti dello spettro elettromagnetico, e producono uno spettro continuo per ogni pixel ritratto nella scena. Tipicamente, un'immagine multispettrale possiede un numero di canali che varia dai 3 fino ad arrivare a 13. In alcuni casi si utilizzano anche 15 canali. Mentre nelle immagini iperspettrali, il numero di canali si aggira intorno alle centinaia: tipicamente si utilizzano 100 o 200 canali, ma è possibile arrivare anche a numeri ben più elevati [108, 109, 110, 77].

La figura (3.8) mette in risalto la differenza tra queste due tipologie di immagini.

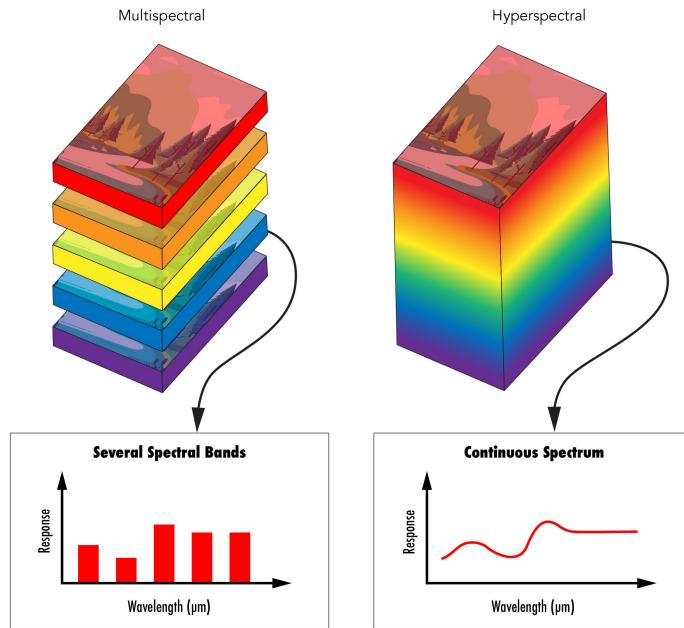


Figura 3.8: Confronto tra Multispectral e Hyperspectral Imaging [108].

Come si può osservare dalla figura 3.8, una singola immagine iperspettrale o multispettrale può essere vista come un cubo di dati, in cui le informazioni spaziali sono disposte lungo l'asse X e l'asse Y, mentre le informazioni di carattere spettrale sono disposte lungo l'asse Z [108].

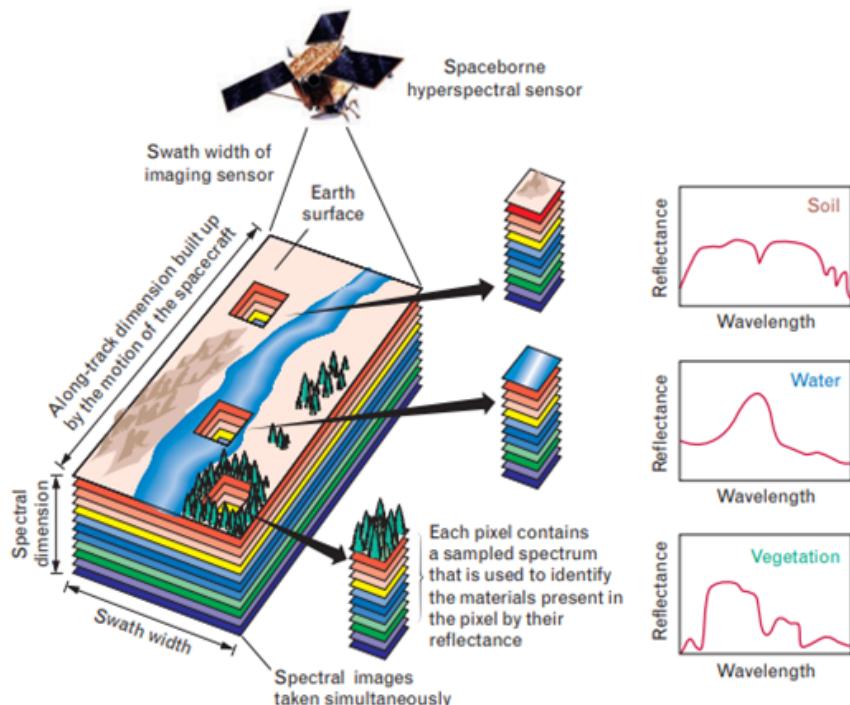


Figura 3.9: Rappresentazione grafica dell'acquisizione di dati [109] .

L'utilizzo di diversi canali spettrali fornisce informazioni più complete e accurate sulla superficie terrestre. Ogni canale registra informazioni sulla luce in una gamma specifica di lunghezze d'onda, che possono essere utili per applicazioni e compiti specifici.

3.4.1 Visualizzazione delle diverse bande

Le immagini a colori sono composte da 3 canali: il rosso, verde e il blu. Questi canali vengono sovrapposti per permettere la visualizzazione a colori reali dell'immagine.

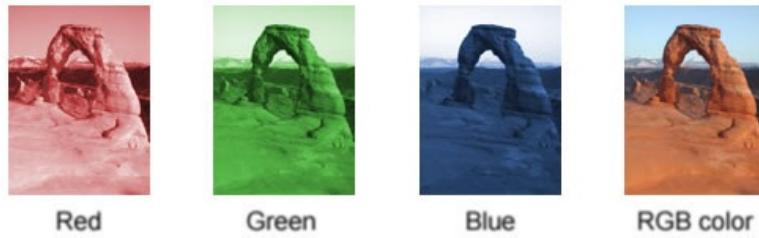


Figura 3.10: Rappresentazione di un immagine a colori.

Quando abbiamo un'immagine composta da diverse bande (come le immagini multispettrali e iperspettrali), la cosa più semplice (ma in genere meno interessante) che possiamo fare è visualizzare singolarmente ogni banda in scala di grigi oppure utilizzare delle heatmap. Se vogliamo un'immagine a colori dobbiamo scegliere tre bande a cui assegnare i tre colori fondamentali R (Red), G (Green), B (Blue). Se usiamo le bande che corrispondono effettivamente a rosso, verde e blu otteniamo un'immagine in colori reali, se invece usiamo anche le altre bande avremo un'immagine in falsi colori [111].

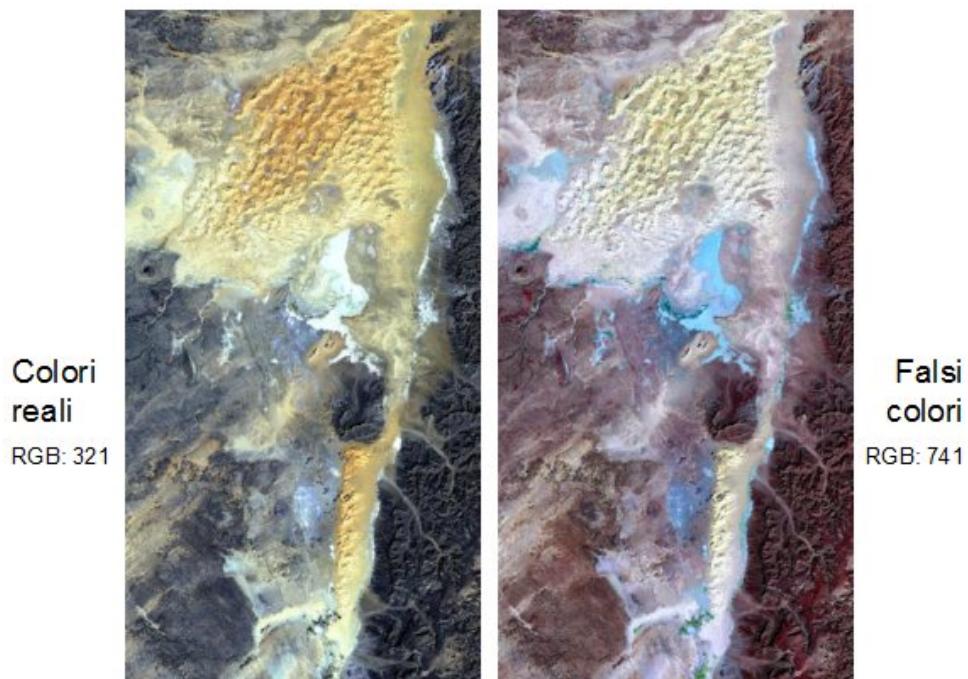


Figura 3.11: Differenze dell'utilizzo di diversi canali [111] .

La combinazione delle diverse bande dipende da cosa vogliamo visualizzare e mettere in risalto.

3.5 Copernicus e Sentinel-2

3.5.1 Il programma spaziale Europeo Copernicus

Copernicus [98] è il nome dell'innovativo programma di monitoraggio del pianeta Terra portato avanti dall'Unione Europea. Il programma è coordinato e gestito dalla Commissione europea ed è attuato in collaborazione con gli Stati membri, l'Agenzia spaziale europea (ESA), l'Organizzazione europea per l'esercizio dei satelliti meteorologici (EUMETSAT), il centro europeo per le previsioni meteorologiche a medio termine (CEPMMT), le agenzie dell'UE e il Mercator Océan. Il programma Copernicus, iniziato nel 2014, si propone di fornire informazioni di libero accesso in merito alle condizioni del pianeta, al fine di informare e aiutare nelle decisioni le autorità, i cittadini, i ricercatori e le organizzazioni. Questo per offrire servizi in sei campi: atmosfera, ambiente marino, territorio, cambiamenti climatici, sicurezza ed emergenze. Il programma Copernicus è costituito da un insieme di satelliti in orbita e da altri sistemi aerei, terrestri o marittimi (detti *in situ*, ovvero non spaziali), i quali generano enormi quantità di dati su misurazioni e fotografie del pianeta. Questi dati vengono poi resi disponibili al pubblico in seguito a elaborazioni e archiviazioni.

3.5.2 Le missioni Sentinel

I satelliti principali utilizzati dal programma sono quelli appartenenti alla classe *Sentinel* (Sentinelle), i quali sono stati progettati e lanciati dall'Agenzia Spaziale Europea (ESA) appositamente per il programma Copernicus. Vengono utilizzati anche satelliti preesistenti o commerciali per dati complementari (in questo caso si parla di missioni partecipanti). Il programma ha attualmente attive 6 missioni Sentinel, ognuna delle quali è basata su una costellazione di 2 satelliti: questi ultimi viaggiano sulla stessa orbita però sfasati di 180° tra di loro, in modo da garantire tempi di rivisitazione (passaggio sopra lo stesso punto della Terra) ottimali. Questi satelliti trasportano sensori ottici, radar e strumenti di imaging dedicati al monitoraggio delle terre, degli oceani e dell'atmosfera. I dati sentinel sono accessibili tramite il portale messo a disposizione dall'ESA: Sentinel Scientific Data Hub (<https://scihub.copernicus.eu/>) .

Sentinel-1

Sentinel-1 è una missione in orbita quasi-polare [99] (angolo di inclinazione rispetto l'equatore pari a 98°) iniziata nel 2014 è costituita dai satelliti Sentinel-1A e Sentinel-1B, aventi un sistema radar ad apertura sintetica (che invia e riceve impulsi lateralmente). Lo scopo è l'imaging radar della Terra diurno e notturno per il monitoraggio terrestre, marittimo e climatico.

Sentinel-2

Sentinel-2 è una missione in orbita quasi-polare eliosincrona [100] (ovvero sincrona con il sole quindi garantisce sempre la stessa luminosità) iniziata nel 2015 è costituita dai satelliti Sentinel-2A (lanciato il 23 giugno 2015) e Sentinel-2B (lanciato il 7 marzo 2017). Entrambi i satelliti hanno una durata prevista di 7 anni, pertanto saranno successivamente sostituiti dai satelliti Sentinel-2C, lanciato il 5 settembre 2024, e dal Sentinel-2D. L'obiettivo è fornire immagini ad alta risoluzione per il monitoraggio del territorio, dei cambiamenti climatici, e delle emergenze. La copertura garantita dalla missione comprende tutte le terre continentali, comprese tra le latitudini 56S e 84N, i mari fino a 20km dalle coste, tutte le isole europee e quelle extra-europee con superficie maggiore di 100km², il Mar Mediterraneo e tutti i mari chiusi.

Sentinel-3

Sentinel-3 è una missione iniziata nel 2016 costituita dai satelliti Sentinel-3A e Sentinel 3B: monitora la superficie del mare e misura le temperature marine e terrestri, fornendo dati ottici, radar e altimetrici. L'obiettivo è il monitoraggio climatico, ambientale e oceanografico.

Sentinel-4

Sentinel-4 è una missione non ancora operativa che fornirà dati sulla composizione atmosferica (concentrazioni di gas, aerosol e altre componenti).

Sentinel-5

Sentinel-5 è una missione non ancora operativa che monitorerà la composizione atmosferica: al momento è stato lanciato solo il satellite Sentinel-5P (Precursor) nel 2017, che sta rilevando provvisoriamente i dati sulla qualità dell'aria, sul clima e sulle radiazioni solari.

Sentinel-6

Sentinel-6 è una missione non ancora operativa che misurerà il livello del mare per studi su clima e oceanografia. Al momento è stato lanciato solo uno dei satelliti.

Satelliti attualmente in funzione

I satelliti dedicati attualmente in orbita sono Sentinel-1A e Sentinel-1B, Sentinel-2A e Sentinel-2B, Sentinel-3A e Sentinel-3B, Sentinel-5P e Sentinel-6A

3.5.3 Caratteristiche di Sentinel-2

I satelliti Sentinel-2 viaggiano ad una quota media di 786 km, hanno una swath (area analizzata dal satellite) che misura 290 km in larghezza e hanno un frequenza di rivisitazione complessiva è elevata (2-5 giorni) [103, 104, 107].

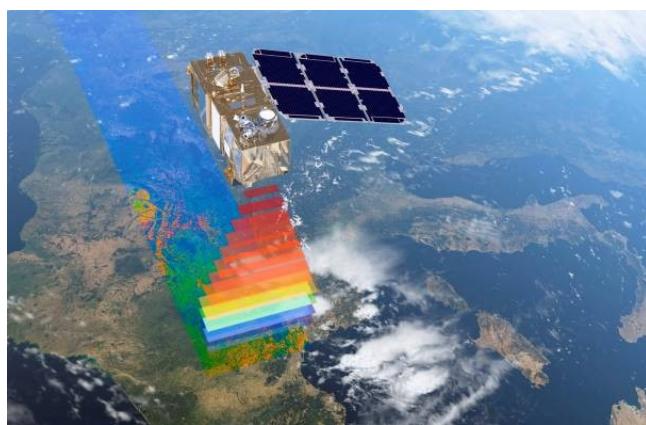


Figura 3.12: Rappresentazione di un satellite sentinel-2 [109].

I satelliti Sentinel-2 si basano su un sistema di telerilevamento passivo, che non emette energia ma registra quella solare riflessa dagli oggetti presenti sulla superficie terrestre (a differenza del sistema radar, che è attivo quindi invia gli impulsi).

Le immagini prodotte sono multispettrali, perché la quantità di energia riflessa viene rilevata dal sensore attraverso diversi intervalli di lunghezze d'onda. L'acquisizione delle informazioni

avviene utilizzato un sistema *push-broom*, nel quale due fila di *detector* (rilevatori), registrano i dati dello swath man mano che il satellite procede lungo l'orbita [107].

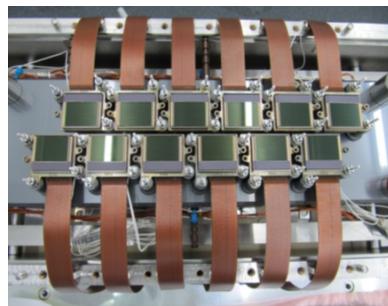


Figura 3.13: Rappresentazione dei rilevatori [107].

Il rilevatore elettronico utilizzato è di tipo CCD (Charge coupled device) ed è costituito da una matrice di pixel, che in base alla quantità di energia ricevuta, assegna dei numeri (detti *digital numbers*) ad ogni pixel. Questi *digital numbers* (DN) rappresentano la radianza, misurata con una risoluzione radiometrica di 12 bit.

I *digital numbers* misurati, vengono poi trasformati in toni di grigio per la visualizzazione: ad una maggiore radianza corrisponde un tono più chiaro (gli oggetti più chiari riflettono più energia, mentre quelli più scuri ne assorbono di più). Questo processo permette di generare immagini in bianco e nero per ogni banda spettrale.

La luce riflessa dalla Terra e dalla sua atmosfera verso lo MSI (MultiSpectral Instrument) viene raccolta da un telescopio a tre specchi (M1, M2 e M3) e focalizzata, tramite un divisore di fascio, su due *Focal Plane Assemblies* (FPA): uno per le dieci lunghezze d'onda VNIR e uno per le tre lunghezze d'onda SWIR. Successivamente i 12 rilevatori, presenti in ogni FPA, registrano la luce incidente, convertendola in dati digitali[107].

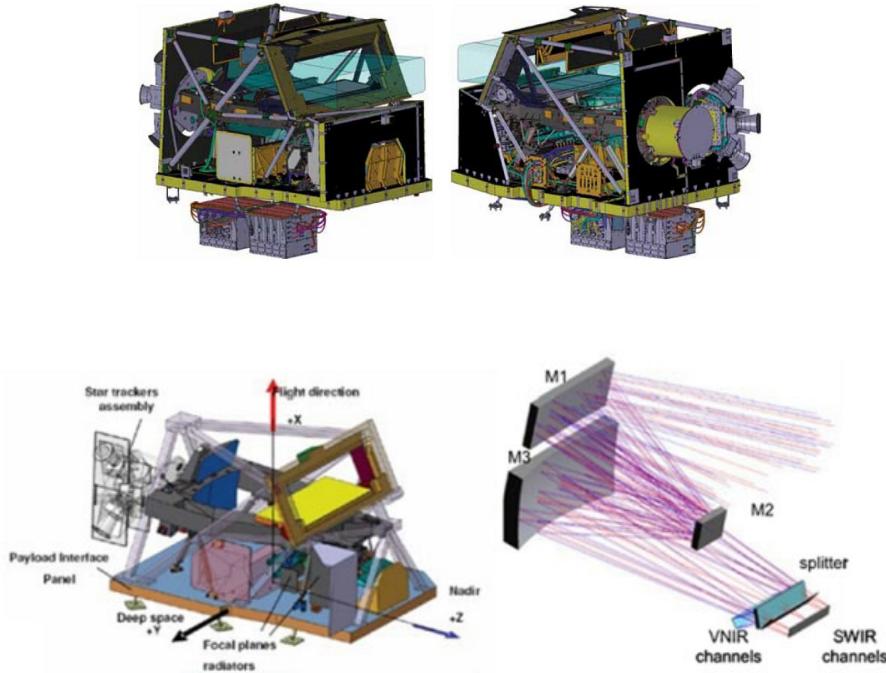


Figura 3.14: Rappresentazione dell'MSI [107].

Questo permette all'MSI del satellite di acquisire 4 bande nel visibile e vicino infrarosso con risoluzione spaziale 10m, 6 bande nell'infrarosso con risoluzione spaziale 20m e 3 bande con

risoluzione 60m di cui una nel blu e due nell'infrarosso [105, 107]. Per risoluzione spaziale si intende a quale dimensione corrisponde un pixel nell'immagine telerilevata, ovvero la distanza coperta da un pixel [78].

Band	Resolution	Central Wavelength	Description
B1	60 m	443 nm	Ultra Blue (Coastal and Aerosol)
B2	10 m	490 nm	Blue
B3	10 m	560 nm	Green
B4	10 m	665 nm	Red
B5	20 m	705 nm	Visible and Near Infrared (VNIR)
B6	20 m	740 nm	Visible and Near Infrared (VNIR)
B7	20 m	783 nm	Visible and Near Infrared (VNIR)
B8	10 m	842 nm	Visible and Near Infrared (VNIR)
B8a	20 m	865 nm	Visible and Near Infrared (VNIR)
B9	60 m	940 nm	Short Wave Infrared (SWIR)
B10	60 m	1375 nm	Short Wave Infrared (SWIR)
B11	20 m	1610 nm	Short Wave Infrared (SWIR)
B12	20 m	2190 nm	Short Wave Infrared (SWIR)

Figura 3.15: Caratteristiche delle diverse bande [106, 102] .

3.5.4 Combinazione delle bande di sentinel-2

Le diverse bande, oltre a poter essere visualizzate singolarmente in scala di grigi, possono essere combinate per mettere in risalto diverse informazioni dell'immagine [102, 101]. Alcune delle molte combinazioni che possono essere ottenute utilizzando le bande di sentinel-2 sono:

- **Natural Color (B4, B3, B2):** utilizza i canali rosso (B4), verde (B3) e blu (B2). Il suo scopo è visualizzare le immagini con colori naturali così come la vedrebbero naturalmente gli esseri umani.



Figura 3.16: Rappresentazione Natural Color.

- **Color Infrared (B8, B4, B3):** questa combinazione di bande è pensata per enfatizzare la vegetazione sana e non sana. La banda del vicino infrarosso (B8), è particolarmente efficace nel riflettere la clorofilla. Per questo, in un'immagine infrarossa a colori, la vegetazione più densa è rossa. Ma le aree urbane sono bianche.



Figura 3.17: Rappresentazione Color Infrared.

- **Short-Wave Infrared (B12, B8A, B4):** utilizza le bande SWIR (B12), NIR (B8A) e Rosso (B4). Questa composizione mostra la vegetazione in varie tonalità di verde. In generale, le tonalità di verde più scure indicano una vegetazione più densa. Mentre il marrone può indicare un terreno privo di vegetazione o un'area edificata.



Figura 3.18: Rappresentazione Short-Wave Infrared.

- **Vegetation Index (B8-B4)/(B8+B4):** Poiché che la vegetazione ha una forte capacità di riflettere la radiazione nel vicino infrarosso (Near Infrared, NIR) e di assorbire quella nella banda del rosso, l'indice di vegetazione è utile per quantificare la quantità di vegetazione presente in un'area.

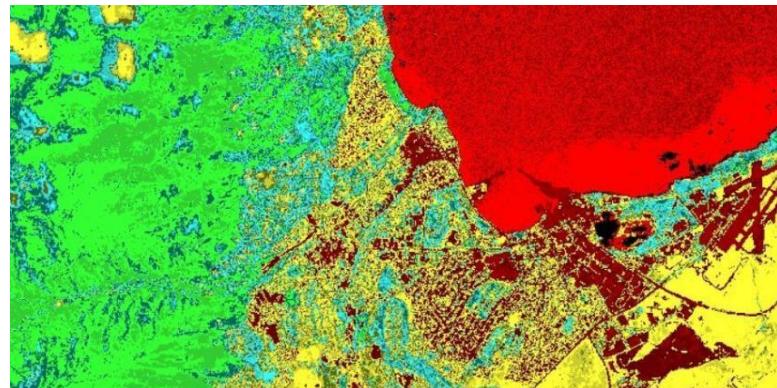


Figura 3.19: Rappresentazione Vegetation Index.

- **Moisture Index (B8A-B11)/(B8A+B11):** è ideale per individuare lo stress idrico nelle piante. Utilizza le onde corte e il vicino infrarosso per generare un indice del contenuto di umidità. In generale, la vegetazione più umida ha valori più alti. Ma valori di indice di umidità più bassi suggeriscono che le piante sono sotto stress a causa di umidità insufficiente.

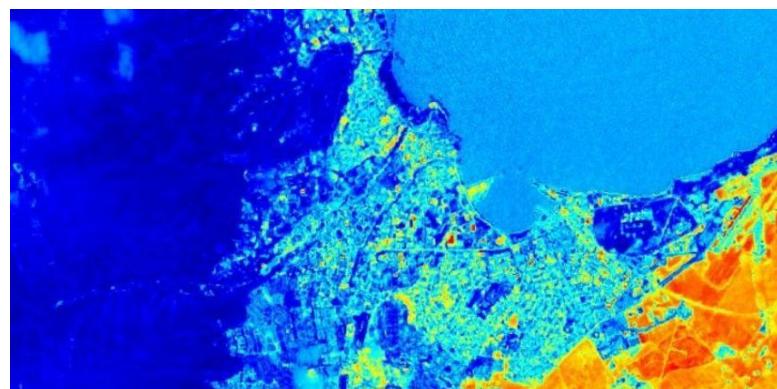


Figura 3.20: Rappresentazione Moisture Index.

Capitolo 4

Reti Neurali Artificiali (ANN)

In questo capitolo viene posta l'attenzione sulle caratteristiche dell'apprendimento profondo tramite reti neurali. In particolare, si partirà dallo studio di una rete neurale biologica, dalla quale hanno preso ispirazione le tecniche fondamentali di Deep Learning, fino ad arrivare ad architetture più complicate.

4.1 La Rete Neurale Biologica

Il miglior modo per capire il funzionamento di una rete neurale è capire come funziona un neurone biologico. In quanto le reti neurali artificiali traggono ispirazione dal funzionamento dei neuroni che compongono il cervello umano [36, 37].

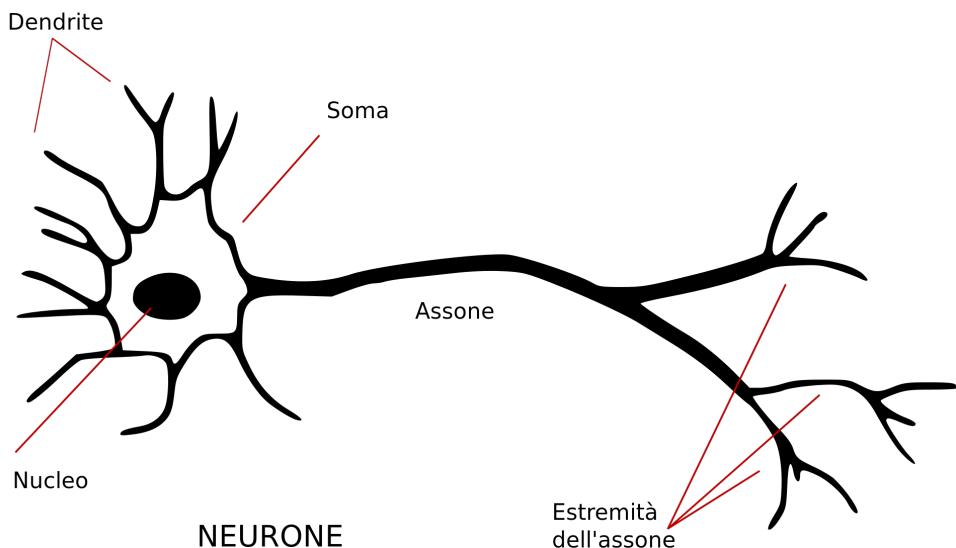


Figura 4.1: Esempio schematico di un neurone [38].

Infatti, un neurone, è un'unità che si occupa di compiere operazioni sulla base di stimoli esterni e di propagare a sua volta lo stimolo. Come si osserva dalla figura 2.1, un neurone, è composto da 3 parti principali:

- Il **soma**: è il corpo cellulare. Si occupa di integrare tra loro i vari input corrispondenti agli stimoli esterni.
- L'**assone**: l'unica linea di uscita del neurone che si dirama in migliaia di parti. Il soma restituisce un risultato che, nel caso in cui superi una certa soglia, il neurone si attiva e il cosiddetto "potenziale d'azione" (impulso elettrico) viene trasportato dall'assone. Al

contrario, se il risultato non supera il valore di soglia il neurone rimane in uno stato di riposo.

- Il **dendrite**: linea di entrata del neurone che riceve segnali in ingresso da altri assoni tramite le sinapsi, le quali a loro volta consentono la comunicazione con altri neuroni.

Il neurone è capace di ricevere segnali attraverso i propri dendriti, li elabora nel soma e successivamente trasmette il segnale, tramite l'assone, al neurone successivo. L'assone non è direttamente collegato ai dendriti di altri neuroni: il punto in cui il segnale viene trasmesso da una cellula ad un'altra è un piccolo spazio denominato “fessura sinaptica”. Quando un segnale è nei pressi di una sinapsi, questa rilascia un quantitativo di sostanze chimiche chiamate “neurotrasmettitori”, i quali determinano la conduttività di una sinapsi, ovvero quanto la sinapsi attenua o enfatizza il segnale elettrico dall'assone. Nella trasmissione di un segnale, le correnti si possono sommare in spazio e tempo e se tale somma oltrepassa una certa soglia, un impulso di una certa entità e durata, denominato “potenziale d'azione”, è generato. Il segnale così prosegue per il prossimo assone, ricominciando il processo [38].

4.2 Il Modello McCulloch-Pitts

Nel 1943, Warren McCulloch, neurofisiologo statunitense e Walter Pitts, matematico statunitense, proposero un modello matematico che descriveva il funzionamento di un neurone. Il loro modello si basava su una regola di **attivazione binaria**: se la combinazione dei segnali in ingresso al neurone supera una certa soglia, il neurone si attiva e produce un'uscita. Se non la supera, il neurone rimane a riposo (il famoso “**tutto-o-nulla**”) [41, 39, 40].

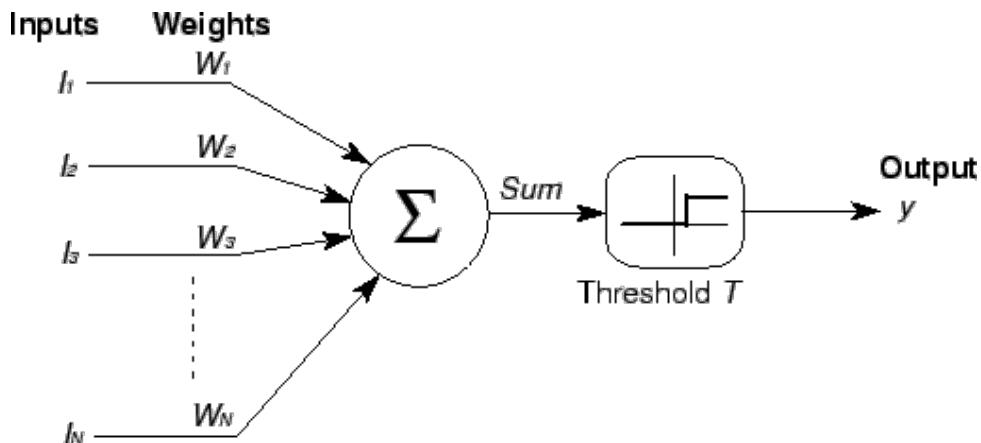


Figura 4.2: Rappresentazione del modello [39]

L'elaborazione prevedeva nel moltiplicare ogni input per un valore specifico, detto “peso”. I risultati di queste moltiplicazioni venivano poi sommati. Se tale somma supera una certa soglia, allora il neurone attivava la propria uscita. Formalmente, questa operazione può essere descritto come [40]:

$$g(x_1, x_2, \dots, x_n) = g(X) = \sum_{i=1}^n x_i \cdot w_i \quad (4.1)$$

$$y = f(g(X)) = \begin{cases} 1 & \text{se } g(X) \geq \theta, \\ 0 & \text{se } g(X) < \theta. \end{cases} \quad (4.2)$$

Dove:

- θ = soglia di attivazione
- y = output attivo/spento (1/0)
- x_i = i-esimo ingresso attivo/spento (1/0)
- w_i = i-esimo peso

La distribuzione dei valori dei pesi varia in base all'importanza dell'ingresso: un ingresso importante avrà un peso elevato, a differenza di uno meno importante che avrà un valore inferiore. Tuttavia, tale modello non si rivelò molto pratico, in quanto, per avere i valori desiderati, bisognava impostare manualmente pesi e connessioni. Alcune semplici operazioni logiche che possono essere eseguite con questo modello sono [41]:

- **AND**(x_1, x_2, \dots, x_n): il neurone si attiva solo se tutti gli input sono attivi. Ovvero, se consideriamo 2 input, è necessario che $y \geq 2$, poiché ogni input attivo da un contributo di 1 e la somma è 2.
- **OR**(x_1, x_2, \dots, x_n): il neurone si attiva se almeno uno degli input è attivo. Ovvero, considerando 2 input, è necessario che $y \geq 1$, poiché basta un neurone attivo che dia un contributo di 1.

4.3 Il Percettrone

Solo successivamente nel 1958, Frank Rosenblatt, uno psicologo statunitense, propose il modello **perceptron** (percettrone), in un report intitolato “The Perceptron: A Perceiving and Recognizing Automaton”. Questo modello è considerato come una delle prime architetture di reti neurali artificiali che riprende la logica utilizzata nel modello di McCulloch e Pitts, ma leggermente più complesso. L'innovazione principale rispetto al modello di McCulloch-Pitts riguardava l'utilizzo di valori numerici reali, come input, superando la limitazione all'impiego esclusivo di dati binari. Inoltre, i valori dei pesi associati a ciascun input venivano appresi mediante una regola di apprendimento che tiene conto della differenza tra output del neurone e output desiderato [47, 41, 128, 39].

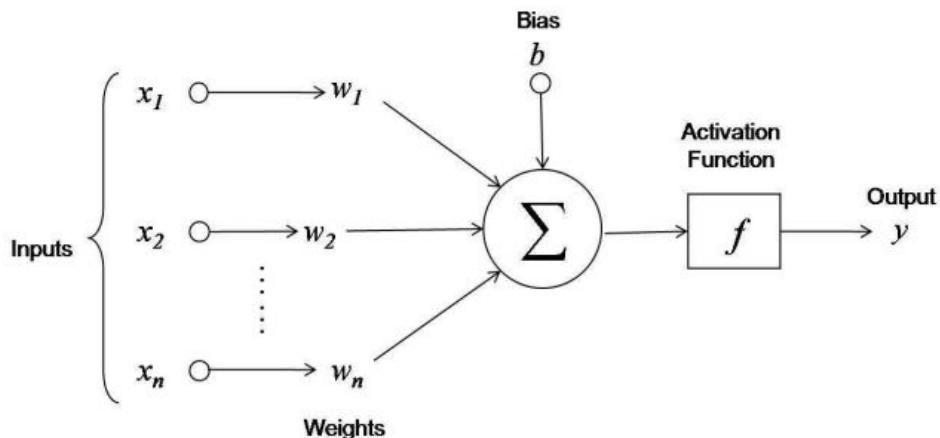


Figura 4.3: Rappresentazione del modello [44]

Un percettrone può essere visto come un classificatore lineare; cioè un algoritmo che classifica l'input separando due categorie tracciando una linea retta (o un iperpiano in spazi a dimensioni superiori). Per questa ragione, il perceptron è considerato un passo cruciale nello sviluppo degli algoritmi di apprendimento automatico.

Matematicamente, il funzionamento del perceptron può essere espresso attraverso le seguenti formule [45]:

$$z = \sum_{i=1}^n w_i x_i + b = (x_1 \cdot w_1) + (x_2 \cdot w_2) + \dots + (x_n \cdot w_n) + b \quad (4.3)$$

$$y = \hat{y} = f(z) \quad (4.4)$$

Dove:

- x_i rappresenta il valore di ciascun input;
- w_i è il peso associato all'input x_i ;
- b è il bias, un termine aggiuntivo che permette di traslare la funzione di attivazione;
- f è una funzione di attivazione, generalmente una soglia (ad esempio, una funzione scalino) che determina l'output finale del modello. Una delle funzioni più utilizzate è l'**Heaviside step function**:

$$f(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ 0 & \text{if } z < 0. \end{cases}$$

- y rappresenta l'output del modello, che spesso viene indicato anche come \hat{y} per denotare un valore stimato o predetto.

Spesso, per rappresentare le operazioni in una maniera più compatta e generalizzata, si adotta una rappresentazione vettoriale, in cui i valori degli input e dei pesi sono rappresentati come segue:

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{W} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

Utilizzando questa notazione, la formula che descrive il funzionamento del perceptron in forma vettoriale è definita come:

$$\hat{y} = f(\mathbf{X}^\top \mathbf{W} + b) = f\left(\begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} + b\right) \quad (4.5)$$

4.3.1 L'Algoritmo di Apprendimento del Percettrone

Rosenblatt dimostrò che l'algoritmo di apprendimento del percettrone converge se le due classi sono linearmente separabili, ovvero se esiste un iperpiano lineare che separa correttamente le due classi. L'algoritmo di apprendimento del percettrone si basa su un approccio iterativo di ottimizzazione dei pesi. Durante l'addestramento, i pesi vengono aggiornati sulla base dell'errore tra l'uscita prevista e l'uscita reale. Tale processo di aggiornamento dei pesi può essere descritto tramite la seguente regola [46, 47, 39]:

$$w_i = w_i + \eta(y - \hat{y})x_i \quad (4.6)$$

dove:

- η è una costante che determina il tasso di apprendimento.
Questo valore è noto anche come **learning rate**;
- y è il valore atteso;
- \hat{y} è il valore predetto dal percettrone;
- x_i è l'input associato al peso w_i .

4.3.2 Implementazione di un percettrone

Per comprendere il funzionamento di questo modello, proviamo a implementare un percettrone utilizzando un linguaggio di programmazione e 'addestrarlo' a distinguere i punti separati da una retta.

Per la parte di algebra lineare useremo la libreria **numpy**. Mentre per realizzare i vari grafici useremo la libreria **matplotlib**.

```
1 import numpy as np
2 import random
3 import matplotlib as plt
```

Basandosi sulle formule (4.6) e (4.5), il percettrone può essere modelizzato nel seguente modo:

```
1
2     class Perceptron:
3
4         def __init__(self, inputs: int) -> None:
5             self.W: np.array = np.random.rand(inputs)
6             self.b: float = random.random()
7
8         def heavisideStep(self, z: float) -> int:
9             return 1 if z >= 0.0 else 0
10
11        def predict(self, X: np.array) -> int:
12            z = np.dot(self.W, X) + self.b # Somma pesata
13            y_hat = self.heavisideStep(z)
14            return y_hat
15
16        def train(self, lr: float, epochs: int, Y_train: np.array,
17                  X_train: np.array) -> Tuple[np.array, np.array]:
18
19            train_losses: List[float] = []
20            epoch_avg_loss: List[float] = []
21
22            for epoch in range(epochs):
23                epoch_loss: List[float] = []
24
25                for i, X in enumerate(X_train):
26                    y_hat = self.predict(X)
```

```

27         # Calcolo dell'errore
28         error = Y_train[i] - y_hat
29
30         # Aggiornamento dei pesi e del bias
31         self.W += lr * error * X
32         self.b += lr * error
33
34         # Salva l'errore
35         epoch_loss.append(error)
36
37         # Errore medio per epoca
38         epoch_avg_loss.append(np.mean(epoch_loss))
39         train_losses.extend(epoch_loss)
40
41     return np.array(train_losses, dtype=np.float32),
        np.array(epoch_avg_loss, dtype=np.float32)

```

Per svolgere l’addestramento del percettrone, utilizziamo la funzione ”train”, alla quale forniamo un set di dati per il quale conosciamo già il risultato corretto. Durante l’addestramento, l’errore (o loss) viene calcolato per ogni esempio del set di dati, aggiornati di conseguenza i pesi ed il bias.

I dati di esempio vengono utilizzati più volte. Ogni iterazione completa sul set di dati è chiamata **epoca** (epoch).

Durante questa procedura teniamo anche traccia di tutti i valori, in modo che poi possiamo utilizzarli per visualizzare l’andamento.

Iniziamo a definire i parametri

```
1 N_TRAIN: int = 5000
2 N_TEST: int = 2000
3 INPUT_SIZE: int = 2
4 MIN: int = -2
5 MAX: int = 12
6
7 alpha: float = -2
8 x_offset: float = -5
9 y_offset: float = +5
10
11 X_TRAIN = np.random.uniform(MIN, MAX, (N_TRAIN, 2)) # N punti casuali (x, y)
12 X_TEST = np.random.uniform(MIN, MAX, (N_TEST, 2))
13 Y_TRAIN = np.array([1 if x2 > alpha*(x1 + x_offset) + y_offset else 0
14     for x1, x2 in X_TRAIN])
15
16 perceptron = Perceptron(inputs=INPUT_SIZE)
17 print("Pesi iniziali:", perceptron.W)
18 print("Bias iniziale:", perceptron.b)
```

Pesi iniziali: [0.80801983, 0.46388153]

Bias iniziale: 0.7213280952626825

Ora chiamiamo la procedura per svolgere il training passandogli i vari parametri.

```
1 # Training
2 train_losses, epoch_avg_loss = perceptron.train(lr=1e-3, epochs=30,
3     Y_train=Y_TRAIN, X_train=X_TRAIN)
4
5 print("Pesi finali:", perceptron.W)
print("Bias finale:", perceptron.b)
```

Pesi finali: [0.06009108, 0.03008616]

Bias finale: -0.4496719047373185

Mentre per quanto riguarda il valore del piano tracciato dal percettrone, possiamo calcolarlo nel seguente modo:

```
1 slope = -perceptron.W[0] / perceptron.W[1]
2 intercept = -perceptron.b / perceptron.W[1]
```

Così è come appare la classificazione dei casi di test prima della procedura di addestramento.

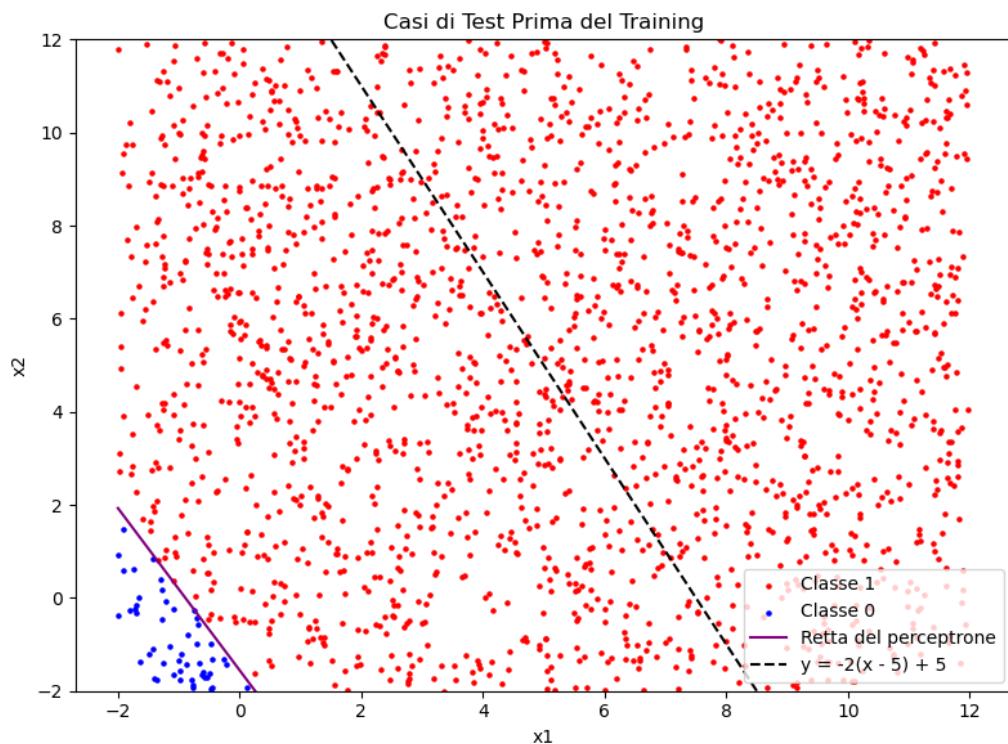


Figura 4.4: Grafico prima dell'addestramento

La linea tratteggiata è la retta che voglio approssimare con il percepitrone, mentre quella viola è la retta attuale del percepitrone. Finita la procedura di addestramento, il grafico appare così

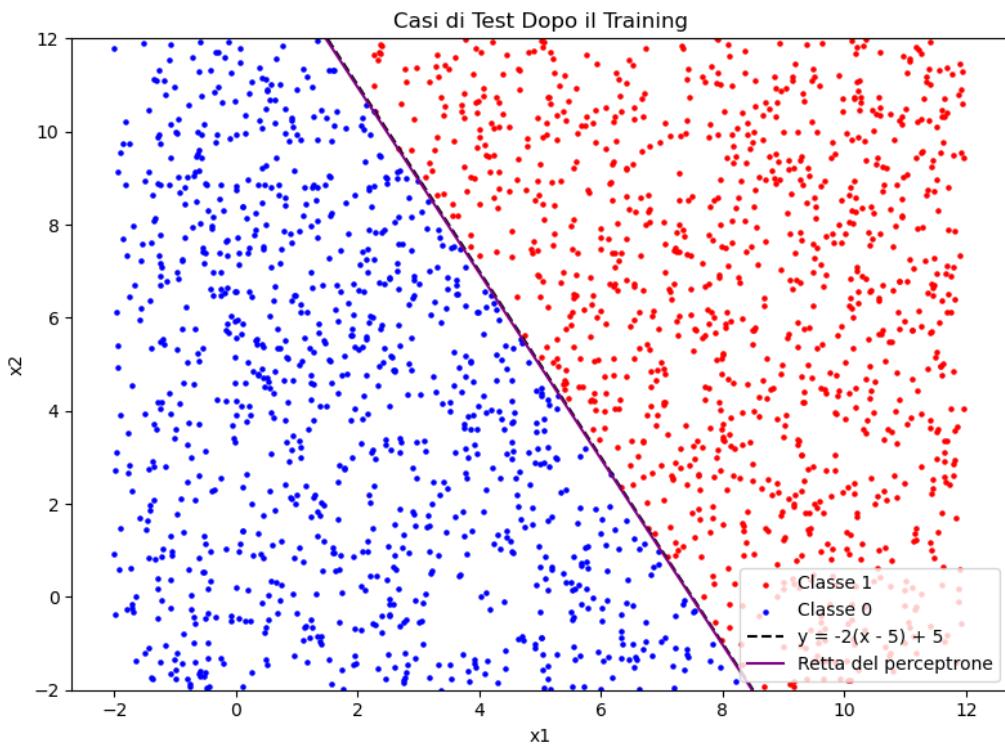


Figura 4.5: Grafico dopo l'addestramento

Come si può osservare dai grafici 4.4 e 4.5, il percettrone è riuscito ad "imparare" a classificare correttamente i punti di test, raggiungendo un precisione molto vicina al 100%. Il grafico sottostante mostra l'andamento dell'errore medio di ogni epoca durante la fase di addestramento.

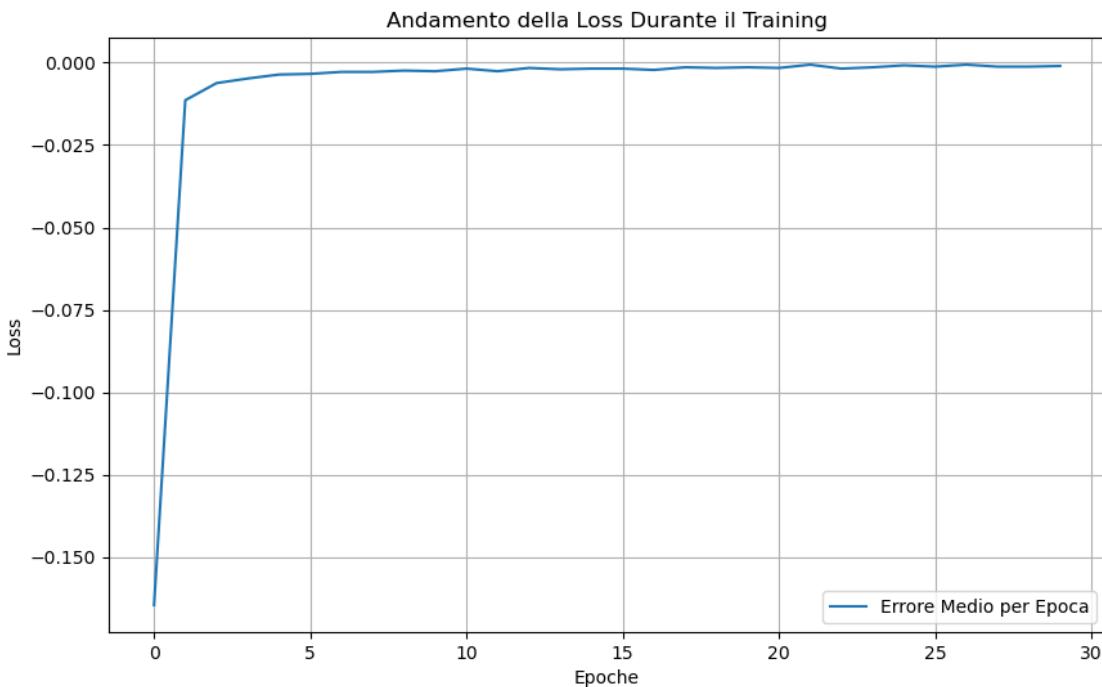


Figura 4.6: Grafico della loss

4.3.3 Limiti del percettrone

Nel 1969, Marvin Minsky e Symour A. Papert, nel loro libro "Perceptrons: an introduction to computational geometry", dimostrarono matematicamente tutte le principali limitazioni del percettrone. Una di queste limitazioni riguardava l'incapacità del percettrone di gestire problemi non linearmente separabili [129].

La figura 4.7 mostra degli esempi di classificazione svolti dall'algoritmo del percettrone per i problemi logici AND, OR e XOR. Graficamente, la retta di decisione determinata dal percettrone riesce a classificare in maniera corretta ciascun elemento in input nel caso dei problemi AND e OR. A differenza degli ultimi, è possibile notare come una retta non sia sufficiente per separare le due classi nel problema XOR: ciò è dovuto al fatto che tale rete riesce a classificare correttamente due classi linearmente separabili, ma non riesce a risolvere un problema di classificazione tra classi non linearmente separabili, quale ad esempio il problema XOR. L'unico modo consisterebbe nel dividere il piano in 3 aree utilizzando due rette di decisione, ma questo non è minimamente possibile utilizzando esclusivamente un solo percettrone [42, 40].

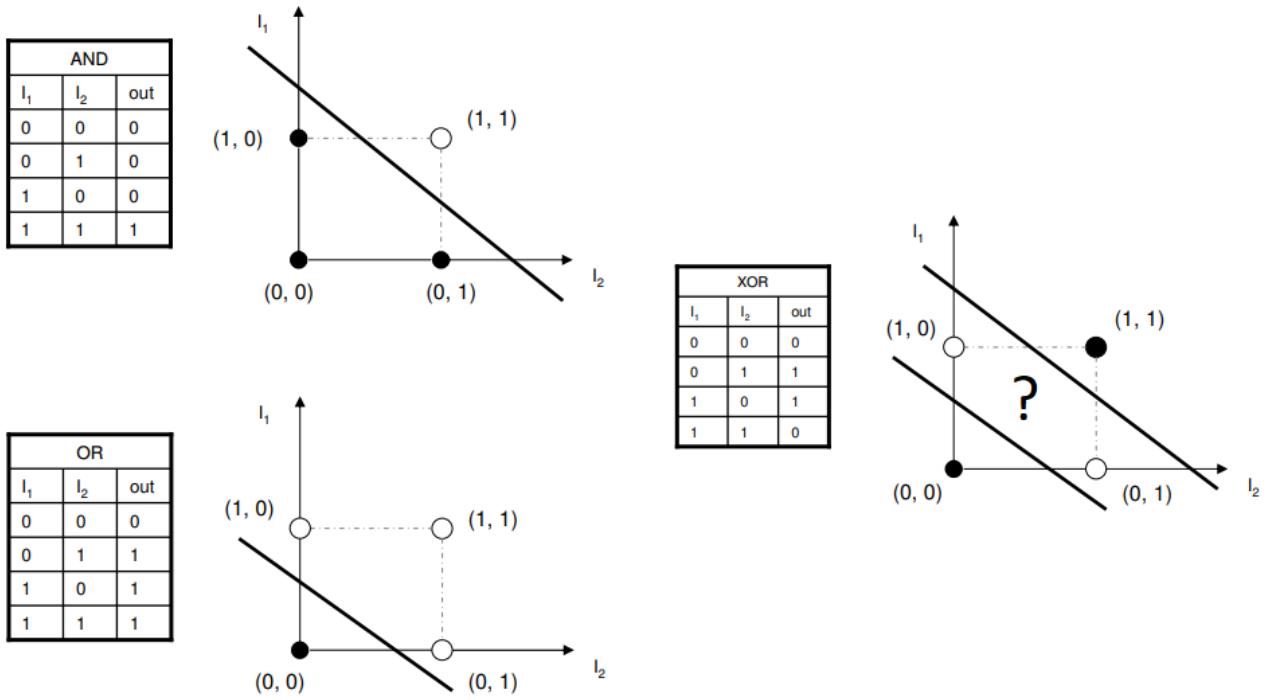


Figura 4.7: Esempi di classificazione [42].

Questa constatazione segnò l'inizio di un periodo noto come "Inverno delle IA", durante il quale l'interesse per lo sviluppo delle reti neurali diminuì notevolmente. Al fine di risolvere problemi più complessi, si cominciò ad interconnettere gli input dei neuroni artificiali con gli output di altri neuroni artificiali, creando una rete neurale a più livelli. Questo portò alla nascita del **Multi-layer Perceptron (MLP)**.

4.4 Single-layer Perceptron (SLP)

Il Single-layer Perceptron (percettrone a singolo layer) è un'estensione del percettrone singolo, in cui più percetroni sono in "pilati" per formare un unico strato (o layer). Questo permette all'SLP di poter essere utilizzato per problemi di classificazione multiclasse.

I SLP furono utilizzati per diverse applicazioni durante gli anni '60, tuttavia, la scoperta dei limiti del perceptron (il libro di Minsky e Papert del 1969) fece calare rapidamente l'interesse per questi modelli, poiché non erano in grado di risolvere problemi non linearmente separabili [43].

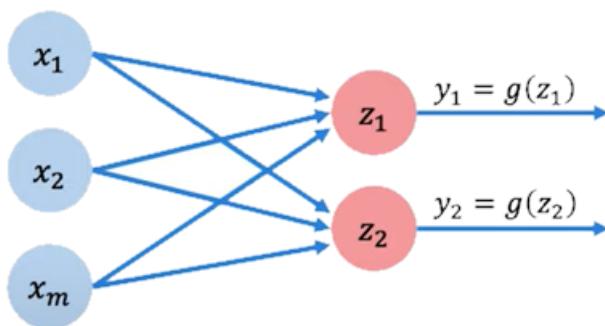


Figura 4.8: Rappresentazione di un SLP

Come si può osservare, ogni nodo (percettrone) nello strato applica la stessa logica, ma in modo indipendente per calcolare diversi output.

Matematicamente, il funzionamento del Single-layer Perceptron è molto simile al funzionamento del percettrone, infatti si ha che:

$$z_i = b_i + \sum_{j=1}^m w_{ij} \cdot x_j \quad (4.7)$$

$$\hat{y}_i = g(z_i) \quad (4.8)$$

L'SLP può essere visto come una versione semplificata del MLP, pertanto non ci soffermeremo ulteriormente.

4.5 Multi-layer Perceptron (MLP)

Cenni storici

I primi tentativi teorici di estendere il perceptron a più livelli risalgono agli studi di Ivakhnenko e Lapa negli anni '60 e '70, ma erano limitati e computazionalmente costosi. Inoltre, la mancanza di un algoritmo efficace per aggiornare i pesi impedì progressi significativi. Solo verso il 1986 con l'introduzione dell'algoritmo di retropropagazione (backpropagation), formalizzato da David Rumelhart, Geoffrey Hinton e Ronald Williams nel loro articolo, si aprirono nuove possibilità per realizzare queste tipologie di reti [127].

Struttura del modello

Il percettrone multistrato è una vera e propria rete neurale artificiale composta, come si evince dal nome, da più strati di percetroni. Esso si compone di un livello di input, il quale riceve il segnale, ed un livello di output, che esegue una previsione o prende una decisione per quanto concerne l'input e, tra questi due livelli, vi è un numero arbitrario di strati "nascosti" (hidden), il vero motore computazionale della rete. Ciascun neurone di un livello è connesso a tutti i neuroni del livello precedente, per tale motivo una rete di questo tipo è anche detta Fully Connected [47, 50, 125].

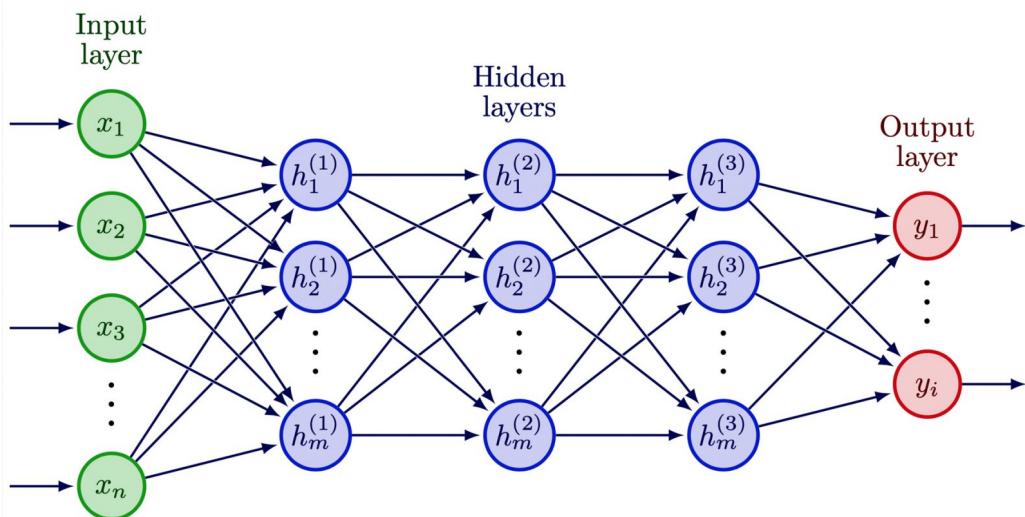


Figura 4.9: Esempio di un MLP [53].

Nel corso dell'ultimo decennio è diventato sempre più frequente l'utilizzo di reti neurali aventi molteplici *hidden layer*. Una rete neurale avente più di un *hidden layer* è detta anche **rete neurale profonda (deep neural network)** e l'insieme di queste reti costituiscono la base del moderno *deep learning*. Spesso ci si riferisce a questi modelli anche con *feed forward network*.

4.5.1 Rappresentazione matematica del modello

Un Multi-Layer Perceptron (MLP) può essere formalizzato matematicamente come una sequenza di trasformazioni lineari, seguite da funzioni di attivazione non lineari.

Notazione

Prima di passare a definire le formule modello, vediamo la notazione utilizzata dal modello [50, 125, 48, 49]:

- Il numero di livelli (o layer) della rete sono denotati con L , e per riferirsi ad un layer generico, useremo la lettera l , con $l \in [0, L]$;
- Con la notazione n_l ci si riferisce al numero di percetroni (o neuroni) che compongono un generico livello l ;
- Con $h^{(l)}$ si indica il vettore colonna dei risultati delle funzioni di attivazione calcolate al livello l ;
- Per rappresentare gli input della rete viene utilizzato un vettore colonna di dimensione d , dove d rappresenta il numero degli input della rete:

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^d$$

- Per rappresentare i pesi di ogni layer l , viene utilizzata una matrice $\mathbf{W}^{(l)}$ di dimensioni $n_l \times n_{l-1}$, dove n_l è il numero di neuroni nel livello corrente e n_{l-1} è il numero di neuroni nel livello precedente:

$$\mathbf{W}^{(l)} = \begin{bmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \cdots & w_{1n_{l-1}}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & \cdots & w_{2n_{l-1}}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_l 1}^{(l)} & w_{n_l 2}^{(l)} & \cdots & w_{n_l n_{l-1}}^{(l)} \end{bmatrix} \in \mathbb{R}^{n_l \times n_{l-1}}$$

In questa matrice, ogni colonna rappresenta i pesi degli input associati a un singolo percettrone (o neurone) del layer;

- Per ogni livello l , il bias è rappresentato come un vettore colonna di dimensioni n_l :

$$\mathbf{b}^{(l)} = \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_{n_l}^{(l)} \end{bmatrix} \in \mathbb{R}^{n_l}$$

- Per denotare la funzione di attivazione usata al livello l , si utilizza la notazione $\sigma^{(l)}$. Per tanto, ogni percepitrone (o neurone) del livello l utilizzerà la stessa funzione di attivazione.

Funzionamento

Chiarita la notazione utilizzata, ora possiamo passare a vedere le formule che caratterizzano il funzionamento del modello [50, 125, 48, 49]:

- L'output di ogni livello l viene calcolato come una combinazione lineare degli input, che dipende dai pesi del layer, dal bias e dal risultato del livello precedente, seguita dall'applicazione della funzione di attivazione:

$$\mathbf{h}^{(l)} = \sigma^{(l)} (\mathbf{W}^{(l)} \mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}) \quad (4.9)$$

Per il livello iniziale ($l = 0$), il vettore di input $\mathbf{h}^{(0)}$ è sostituito da \mathbf{X} , il vettore degli input della rete.

- L'output finale (o predetto) della rete $\hat{\mathbf{y}}$ è calcolato al livello L come:

$$\hat{\mathbf{y}} = \sigma^{(L)} (\mathbf{W}^{(L)} \mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}) \quad (4.10)$$

4.5.2 Esempio applicativo

Consideriamo un MPL con la seguente struttura:

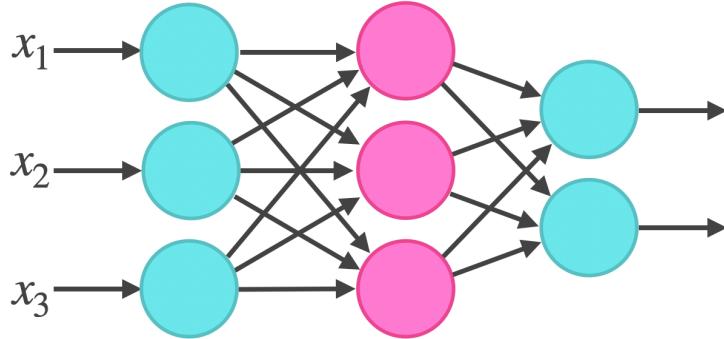


Figura 4.10: Esempio di un MLP.

Come si osserva dalla figura, la rete è costituita dai seguenti elementi:

- 3 neuroni in input (x_1, x_2, x_3);
- Un livello nascosto ($l = 1$) con 3 neuroni ($h_1^{(1)}, h_2^{(1)}, h_3^{(1)}$);
- Un livello di output ($l = 2$) con 2 neuroni (y_1, y_2).

Definiamo i parametri della rete come segue:

- Vettore di input:

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 0.5 \end{bmatrix}$$

- Matrici dei pesi:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.2 & -0.4 & 0.1 \\ -0.1 & 0.3 & -0.5 \\ 0.7 & 0.2 & -0.3 \end{bmatrix} \quad \mathbf{W}^{(2)} = \begin{bmatrix} 0.5 & -0.6 & 0.2 \\ -0.3 & 0.8 & -0.7 \end{bmatrix}$$

- Vettori dei bias:

$$\mathbf{b}^{(1)} = \begin{bmatrix} 0.1 \\ -0.2 \\ 0.3 \end{bmatrix} \quad \mathbf{b}^{(2)} = \begin{bmatrix} -0.1 \\ 0.4 \end{bmatrix}$$

- Come funzione di attivazione per $\sigma^{(1)}$ e per $\sigma^{(2)}$, utilizzeremo la funzione di **Heaviside** con $\theta = 0$, definita nel seguente modo:

$$\sigma(z) = \begin{cases} 1 & \text{se } z \geq \theta, \\ 0 & \text{se } z < \theta. \end{cases}$$

Ora passiamo a calcolare l'output della rete. Utilizzando le formule 4.9 e 4.10, l'output della rete può essere descritto come:

$$\mathbf{y} = \sigma^{(2)} (\mathbf{W}^{(2)} (\sigma^{(1)} (\mathbf{W}^{(1)} \mathbf{X} + \mathbf{b}^{(1)})) + \mathbf{b}^{(2)})$$

Iniziamo calcolando il prodotto matriciale $\mathbf{W}^{(1)} \mathbf{X}$:

$$\begin{aligned} \mathbf{W}^{(1)} \mathbf{X} &= \begin{bmatrix} 0.2 & -0.4 & 0.1 \\ -0.1 & 0.3 & -0.5 \\ 0.7 & 0.2 & -0.3 \end{bmatrix} \begin{bmatrix} 1 \\ -2 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0.2(1) + (-0.4)(-2) + 0.1(0.5) \\ -0.1(1) + 0.3(-2) + (-0.5)(0.5) \\ 0.7(1) + 0.2(-2) + (-0.3)(0.5) \end{bmatrix} = \\ &= \begin{bmatrix} 0.2 + 0.8 + 0.05 \\ -0.1 - 0.6 - 0.25 \\ 0.7 - 0.4 - 0.15 \end{bmatrix} = \begin{bmatrix} 1.05 \\ -0.95 \\ 0.15 \end{bmatrix} \end{aligned}$$

Ora sommiamo il bias:

$$\mathbf{W}^{(1)} \mathbf{X} + \mathbf{b}^{(1)} = \begin{bmatrix} 1.05 \\ -0.95 \\ 0.15 \end{bmatrix} + \begin{bmatrix} 0.1 \\ -0.2 \\ 0.3 \end{bmatrix} = \begin{bmatrix} 1.15 \\ -1.15 \\ 0.45 \end{bmatrix}$$

Infine, applichiamo la funzione di Heaviside (con $\theta = 0$) in modo da ottenere il risultato del layer 1 ($\mathbf{h}^{(1)}$):

$$\mathbf{h}^{(1)} = \sigma^{(1)} (\mathbf{W}^{(1)} \mathbf{X} + \mathbf{b}^{(1)}) = \begin{bmatrix} \sigma(1.15) \\ \sigma(-1.15) \\ \sigma(0.45) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Ora calcoliamo il prodotto $\mathbf{W}^{(2)} \mathbf{h}^{(1)}$:

$$\begin{aligned} \mathbf{W}^{(2)} \mathbf{h}^{(1)} &= \begin{bmatrix} 0.5 & -0.6 & 0.2 \\ -0.3 & 0.8 & -0.7 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.5(1) + (-0.6)(0) + 0.2(1) \\ -0.3(1) + 0.8(0) + (-0.7)(1) \end{bmatrix} = \\ &= \begin{bmatrix} 0.5 + 0 + 0.2 \\ -0.3 + 0 - 0.7 \end{bmatrix} = \begin{bmatrix} 0.7 \\ -1.0 \end{bmatrix} \end{aligned}$$

Ora sommiamo il bias:

$$\mathbf{W}^{(2)} \mathbf{h}^{(1)} + \mathbf{b}^{(2)} = \begin{bmatrix} 0.7 \\ -1.0 \end{bmatrix} + \begin{bmatrix} -0.1 \\ 0.4 \end{bmatrix} = \begin{bmatrix} 0.6 \\ -0.6 \end{bmatrix}$$

Infine, applichiamo la funzione di Heaviside al livello di output in modo da ottenere il risultato del layer 2 ($(\mathbf{h}^{(2)})$):

$$\mathbf{h}^{(2)} = \sigma^{(2)} (\mathbf{W}^{(2)} \mathbf{h}^{(1)} + \mathbf{b}^{(2)}) = \begin{bmatrix} \sigma(0.6) \\ \sigma(-0.6) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

L'output finale della rete sarà così:

$$\mathbf{y} = \mathbf{h}^{(2)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

4.6 Il Gradient Descent

Per valutare quanto bene abbia "imparato" il modello si utilizza una *loss function* (funzione di perdita) C , talvolta indicata anche come *cost function* (funzione di costo). Essa rappresenta una misura dell'errore del modello in termini di capacità di stimare la relazione tra un input x e il corrispondente output y .

La scelta della loss function da utilizzare dipende dal tipo di problema per cui è stato progettato il modello di rete neurale: per problemi di regressione, la loss function comunemente usata è il Mean Squared Error (MSE), ma ne esistono molte altre per specifiche esigenze. La funzione di costo MSE è così definita:

$$C(w, b) = \frac{1}{2n} \sum_{i=1}^n \|y(x) - a\|^2 = \frac{1}{2n} \sum_{i=1}^n \|y_i - \hat{y}_i\|^2$$

dove w e b sono i vettori contenenti, rispettivamente, tutti i pesi e i bias della rete, x è il vettore degli elementi in input avente output y , n è il numero totale di elementi del dataset e a è l'output di x stimato dalla rete.

Al fine di condurre il modello ad apprendere correttamente i dati presenti all'interno del training set, è necessario minimizzare la funzione C : per farlo si utilizza l'algoritmo **Gradient Descent** (discesa del gradiente) [50, 51, 52].

4.6.1 Cenni alle derivate parziali

Prima di addentrarsi a comprendere come funziona il Gradient Descent, accenniamo il funzionamento delle derivate parziali. In quanto vengono largamente utilizzate sia nel Gradient Descent che nella backpropagation. In breve, una derivata parziale si utilizza sulle funzioni che hanno più variabili, come $f(x, y, z)$, e consiste nel calcolare la derivata della funzione rispetto a una singola variabile, considerando tutte le altri variabili come costanti. Per indicare una derivata parziale di una f rispetto a una generica variabile x , si utilizza la notazione $\frac{\partial f}{\partial x}$. Oppure può anche essere indicato la notazione $\partial_x f$ [2]. Per capire il funzionamento, prendiamo una funzione di due variabili, ad esempio $f(x, y) = x^2 + 3xy + y^2$. La derivata parziale di f rispetto a x sarà:

$$\frac{\partial f}{\partial x} = 2x + 3y$$

Analogamente, la derivata parziale di f rispetto a y sarà:

$$\frac{\partial f}{\partial y} = 3x + 2y$$

4.6.2 Funzionamento del Gradient Descent

Per comprendere meglio l'idea alla base del gradient descent, supponiamo inizialmente che C sia una funzione di due variabili v_1 e v_2 , ovvero $C(v_1, v_2)$. La variazione (Δ) di ciascuna delle due componenti provocherà una variazione di C esprimibile come:

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2 \tag{4.11}$$

In cui:

- $\frac{\partial C}{\partial v_1} \Delta v_1$: misura il contributo al cambiamento totale ΔC dovuto ad una variazione di v_1 per il tasso di variazione di C rispetto a v_1 , ovvero quanto velocemente varia ΔC spostandosi lungo l'asse v_1 mantenendo v_2 costante.

- $\frac{\partial C}{\partial v_2} \Delta v_2$: misura il contributo al cambiamento totale ΔC dovuto ad una variazione di v_2 per il tasso di variazione di C rispetto a v_2 , ovvero quanto velocemente varia ΔC spostandosi lungo l'asse v_2 mantenendo v_1 costante.

Per minimizzare C , è necessario trovare Δv_1 e Δv_2 tale che ΔC sia negativo.

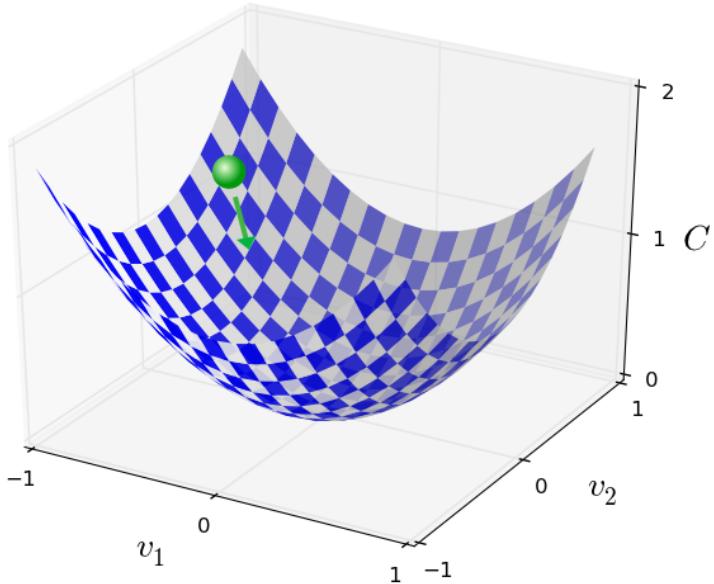


Figura 4.11: Minimizzare la funzione C

Definiamo il gradiente di C come il vettore delle sue derivate parziali:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T \quad (4.12)$$

e, ponendo $\Delta v = (\Delta v_1, \Delta v_2)^T$, è possibile riscrivere l'equazione (4.11) come:

$$\Delta C \approx \nabla C \cdot \Delta v \quad (4.13)$$

L'importanza dell'equazione (4.13) sta nel fatto che, affinché C possa decrescere, è necessario scegliere Δv in modo che ΔC sia negativo. In particolare, poniamo:

$$\Delta v = -\eta \nabla C \quad (4.14)$$

dove η è un numero positivo comunemente noto come *learning rate*, tipicamente avente un valore tra 1 e 0. Pertanto, l'equazione (4.13) può essere riscritta come:

$$\Delta C \approx -\eta \|\nabla C\|^2 \quad (4.15)$$

Poiché $\|\nabla C\|^2 \geq 0$, si ha la certezza che $\Delta C \leq 0$. Questa procedura permetterà di aggiornare le componenti di v che saranno uguali a:

$$v \rightarrow v' = v - \eta \nabla C \quad (4.16)$$

Nel nostro caso, il vettore v è sostituito da w e b , ovvero i vettori di pesi e bias. Parleremo di epoca di training quando ciascun elemento del dataset verrà processato sia in avanti (*forward*) sia all'indietro (*backward*) dalla rete una sola volta.

Per ogni epoca, una volta calcolata la loss function C , è necessario dunque derivare C rispetto a w e rispetto a b . Al termine di ogni epoca verranno infine aggiornati i valori di w e di b nel seguente modo:

$$w_i \rightarrow w'_i = w_i - \eta \frac{\partial C}{\partial w_i} \quad \forall i = 1, \dots, n \quad (4.17)$$

$$b_i \rightarrow b'_i = b_i - \eta \frac{\partial C}{\partial b_i} \quad \forall i = 1, \dots, n \quad (4.18)$$

Il *learning rate* η regolerà l'aggiornamento di w e b : più grande sarà η , più velocemente l'algoritmo tenderà a convergere verso il punto che minimizza C . Tuttavia, un learning rate elevato può condurre a degli eccessivi "salti" all'interno della funzione, il che potrebbe causare una mancanza di convergenza. Viceversa, la scelta di un learning rate eccessivamente piccolo potrebbe rallentare parecchio il processo di convergenza, rendendo necessario un numero di epoche più elevato al fine di raggiungere risultati accettabili.

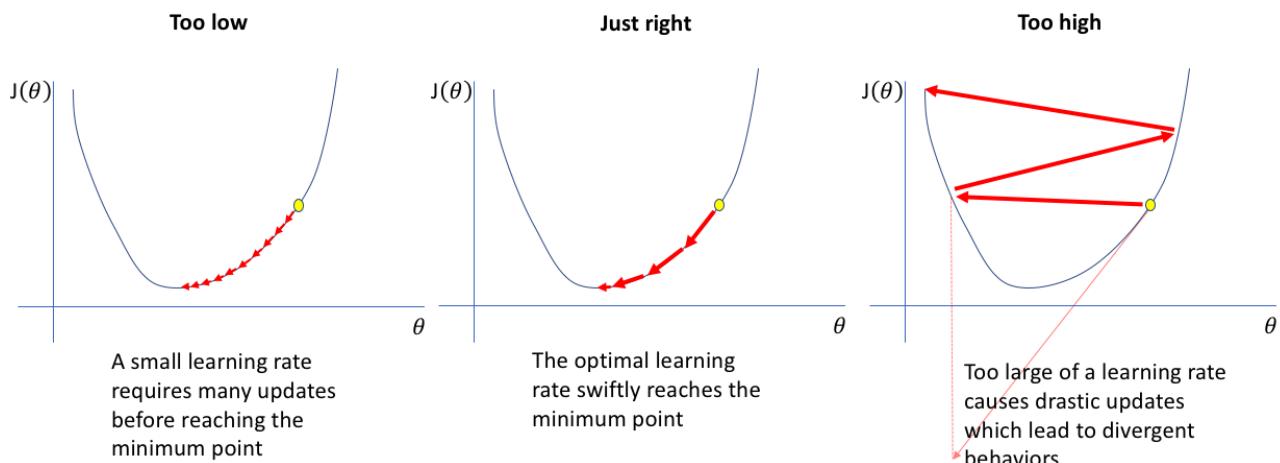


Figura 4.12: I grafici mostrano il differente processo di convergenza verso il punto di minimo globale di C [61]

È possibile applicare l'algoritmo del gradient descent calcolando il gradiente sull'intero dataset oppure selezionare uno o più elementi ad ogni iterazione: questo viene indicato con il termine **batch**, ovvero il numero totale di campioni del dataset utilizzati per calcolare il gradiente in una singola operazione. La scelta di un opportuno gruppo di elementi del dataset influisce sull'aggiornamento di w e b , pertanto influisce anche sul processo di convergenza della *loss function*. Poiché l'algoritmo del gradient descent è una procedura iterativa, risulta abbastanza evidente che effettuare il training di un modello per una singola epoca comporterà un solo aggiornamento di pesi e bias. Viceversa, un numero eccessivo di epoche di training può condurre il modello ad adattarsi eccessivamente ai dati di training e, di conseguenza, andare incontro al problema dell'*overfitting*, ovvero l'incapacità di generalizzare i dati.

4.7 Ottimizzatori

Gli ottimizzatori sono delle tecniche che vengono utilizzate per migliorare il processo di addestramento di una rete neurale [125, 61].

4.7.1 Full Batch Gradient Descent

Il *full batch gradient descent* [50, 61] calcola la loss function considerando, ad ogni iterazione, tutti gli elementi presenti all'interno del dataset: C sarà dunque la loss function media calcolata sull'intero dataset, ovvero $C = \frac{1}{n} \sum_i C_i$. La procedura di aggiornamento di w e b è la medesima descritta dalle equazioni (4.17) e (4.18). Nonostante la semplicità a livello intuitivo della procedura, essa comporta diversi svantaggi: poiché vengono utilizzati tutti gli elementi presenti all'interno del dataset ad ogni singola iterazione, tale operazione risulta essere parecchio onerosa in presenza di dataset di grandi dimensioni. Un altro problema consiste nella sua scarsa dinamicità: per migliorare il modello con nuovi dati è necessario ripetere il processo di training sull'intero dataset. Inoltre, tale procedura risulta essere molto soggetta al problema dei minimi locali.

4.7.2 Stochastic Gradient Descent

Uno dei problemi del *full batch gradient descent* [50, 61] riguardava la necessità di calcolare ∇C su tutti gli elementi del dataset, a prescindere dalla dimensione di esso. L'approccio dello stochastic gradient descent è differente: ∇C non è più la media sui ∇C calcolati sull'intero dataset, bensì viene stimato da un singolo elemento del dataset scelto in maniera casuale. A differenza del precedente approccio, la scelta di un singolo elemento del dataset per il calcolo di C porta ad un notevole miglioramento in termini di tempo d'esecuzione dell'algoritmo. Da un punto di vista grafico, è possibile che C tenda parecchio ad oscillare: tali oscillazioni potrebbero condurre la loss function ad uscire più facilmente da punti di minimo locale.

4.7.3 Mini Batch Gradient Descent

Lo stochastic gradient descent riesce tuttavia a risolvere solo parzialmente il problema dei minimi locali. Una via di mezzo tra le tecniche precedentemente descritte è il *mini batch gradient descent* [50, 61]: all'interno del dataset, ad ogni epoca viene estratto un sottoinsieme di elementi casuali del dataset e la loss function calcolata è la media delle loss functions calcolate all'interno del sottoinsieme. Pertanto, scelto un numero $m < n$ come batch size, ∇C sarà uguale a:

$$\nabla C = \frac{1}{n} \sum_i \nabla C_i \approx \frac{1}{m} \sum_{j=1}^m \nabla C_j \quad (4.19)$$

e le equazioni (4.17) e (4.18) diventeranno:

$$w_i \rightarrow w'_i = w_i - \frac{\eta}{m} \cdot \sum_j^m \frac{\partial C_{X_j}}{\partial w_i} \quad (4.20)$$

$$b_i \rightarrow b'_i = b_i - \frac{\eta}{m} \cdot \sum_j^m \frac{\partial C_{X_j}}{\partial b_i} \quad (4.21)$$

dove le somme sono su tutti gli esempi di addestramento X_j presenti nel mini-batch corrente. Finito il processo di aggiornamento dei parametri, si seleziona un altro mini-batch e si ripete l'operazione finché non abbiamo esaurito gli input di addestramento. A quel punto ricominciamo con una nuova epoca di addestramento.

4.7.4 Full Batch vs Stochastic vs Mini Batch

All'interno della figura 4.13 è rappresentato il confronto tra le traiettorie prodotte dalle tre tecniche citate. Si può osservare come lo stochastic gradient descent e il mini batch gradient descent producano traiettorie che tendono ad andare "a zig-zag". Questo comportamento è dovuto al fatto che, rispettivamente, vengono scelti un elemento ed un sottocampione dal dataset di partenza. Pertanto, a differenza del full batch gradient descent, non stiamo calcolando il gradiente di C , bensì una sua approssimazione.

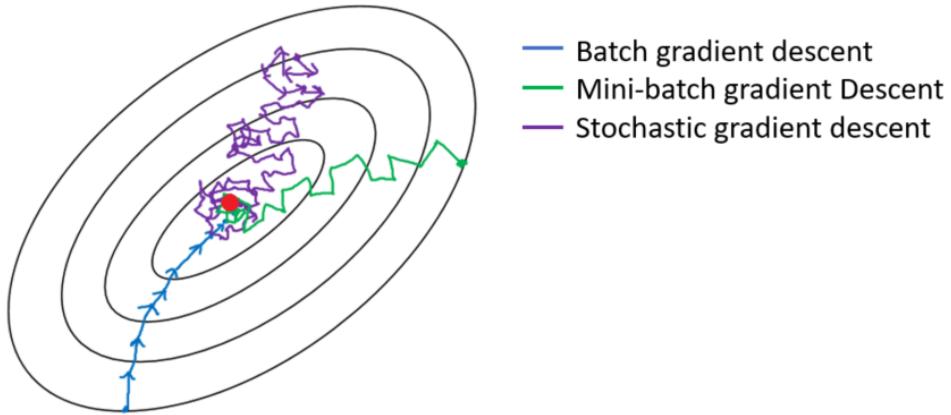


Figura 4.13: Confronto tra le traiettorie prodotte dagli algoritmi basati sul gradient descent [62].

Queste sono soltanto alcune delle tante strategie di ottimizzazione che si possono utilizzare per migliorare l'apprendimento del modello.

4.8 L'algoritmo della Backpropagation

Come accennato precedentemente, l'algoritmo della Backpropagation ha portato grandi progressi nell'implementazione delle reti neurali. L'algoritmo si basa sulla modifica sistematica dei pesi delle connessioni tra neuroni cosicché l'output della rete coincida sempre di più con il risultato atteso [50, 125, 48, 49, 54]. Questo algoritmo è costituito da due fasi principali:

- **Forward propagation:** Il processo del calcolo delle predizioni che parte dai coefficienti per poi terminare con l'errore rispetto al risultato atteso.
- **backward propagation:** il processo che permette di ottimizzare i coefficienti w e b partendo dal l'errore calcolato precedentemente.

Questo processo, introdotto già nel 1970, acquisì notevole importanza nel 1986, anno in cui D. Rumelhart, G. Hinton e R. Williams evidenziarono quanto una rete neurale apprenda più velocemente grazie alla backpropagation rispetto alle tecniche utilizzate in precedenza. Ancora oggi, esso assume un'importanza notevole al fine dell'apprendimento delle più moderne reti neurali profonde [127].

4.8.1 Il prodotto di Hadamard

L'algoritmo di backpropagation si basa su comuni operazioni algebriche lineari, come l'addizione di vettori, la moltiplicazione di un vettore per una matrice e così via. La backpropagation si basa anche sul prodotto di Hadamard [1], un'operazione matematica non comunemente utilizzata. In particolare, supponiamo di avere due matrici, A e B , l'operazione del prodotto di Hadamard tra le due matrici, denotata con $A \odot B$, è definita nel seguente modo:

$$(A \odot B)_{ij} = A_{ij} \cdot B_{ij}, \quad \forall i, j \quad (4.22)$$

In pratica consiste in un'operazione di prodotto elementare tra gli elementi di ogni matrice. Pertanto è anche importante che le due matrici abbiano la medesima dimensione. Ad esempio, consideriamo queste due matrici:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad B = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}.$$

Il prodotto di Hadamard $A \odot B$ è calcolato come:

$$A \odot B = \begin{bmatrix} 1 \cdot 9 & 2 \cdot 8 & 3 \cdot 7 \\ 4 \cdot 6 & 5 \cdot 5 & 6 \cdot 4 \\ 7 \cdot 3 & 8 \cdot 2 & 9 \cdot 1 \end{bmatrix} = \begin{bmatrix} 9 & 16 & 21 \\ 24 & 25 & 24 \\ 21 & 16 & 9 \end{bmatrix}.$$

4.8.2 Funzionamento della Backpropagation

La backpropagation consiste nel comprendere come la modifica dei pesi e dei bias in una rete modifichi la funzione di costo. Ciò significa calcolare le derivate parziali $\frac{\partial C}{\partial w_{jk}^{(l)}}$ e $\frac{\partial C}{\partial b_j^{(l)}}$. Ma

per calcolarli, introduciamo prima una quantità intermedia, $\delta_j^{(l)}$, ovvero l'errore presente nel neurone j -esimo del layer l -esimo layer, $z_j^{(l)}$. L'aggiunta di tale termine di errore al neurone $z_j^{(l)}$ porterà alla modifica della funzione di attivazione associata ad esso, la quale sarà adesso uguale a $\sigma(z_j^{(l)} + \Delta_j^{(l)})$ propagandosi all'interno della rete e causando così un'altrettanto modifica alla loss function totale, la quale subirà una variazione di $\frac{\partial C}{\partial z_j^{(l)}} \Delta z_j^{(l)}$.

E' possibile dunque monitorare la variazione della loss function facendo variare il termine di errore $\delta_j^{(l)}$, definito come

$$\delta_j^{(l)} = \frac{\partial C}{\partial z_j^{(l)}} \quad (4.23)$$

La backpropagation si basa principalmente su quattro equazioni, le quali costituiscono la base per il calcolo del gradiente di C e del termine di errore δ :

- Equazione per l'errore all'interno dell'layer di output:

$$\delta_j^{(L)} = \frac{\partial C}{\partial a_j^{(L)}} \sigma'(z_j^{(L)}) \quad (4.24)$$

Dove il primo termine $\frac{\partial C}{\partial a_j^{(L)}}$ misura la velocità con cui il costo cambia in funzione del j -esimo output di attivazione. Mentre il secondo termine $\sigma'(z_j^{(L)})$ (dove σ' è la derivata della funzione di attivazione σ) misura la velocità con cui la funzione di attivazione σ cambia in $z_j^{(L)}$. Nella forma matriciale l'equazione (4.24) diventa:

$$\delta^{(L)} = \nabla_a C \odot \sigma'(z^{(L)}) \quad (4.25)$$

- Equazione per l'errore $\delta^{(l)}$ espresso come errore del layer successivo:

$$\delta^{(l)} = ((W^{(l+1)})^T \delta^{(l+1)}) \odot \sigma'(z_j^{(l)}) \quad (4.26)$$

Dove $(W^{(l+1)})^T$ è la trasposta della matrice dei pesi $W^{(l+1)}$ per il $(l+1)$ -esimo layer. Questa equazione sembra complicata, ma ogni elemento ha una precisa interpretazione. Supponiamo di conoscere l'errore $\delta^{(l+1)}$. Quando applichiamo la matrice dei pesi trasposta, $(W^{(l+1)})^T$, possiamo pensare intuitivamente a questo come allo spostamento dell'errore all'indietro attraverso la rete, dandoci una sorta di misura dell'errore all'output dell' l -esimo layer. Quindi svolgiamo il prodotto di Hadamard \odot con $\sigma'(z_j^{(l)})$. Questo sposta l'errore all'indietro attraverso la funzione di attivazione del layer l , dandoci l'errore $\delta^{(l)}$ nell'input ponderato allo strato l .

- Equazione per la variazione della loss function rispetto a qualsiasi bias:

$$\frac{\partial C}{\partial b_j^{(l)}} = \delta_j^{(l)} \quad (4.27)$$

- Equazione per la variazione della loss function rispetto a qualsiasi peso:

$$\frac{\partial C}{w_{jk}^{(l)}} = a_k^{(l-1)} \delta_j^{(l)} \quad (4.28)$$

L'equazione può essere riscritta in una notazione meno indicizzata come

$$\frac{\partial C}{w} = a_{in} \delta_{out} \quad (4.29)$$

dove si intende che a_{in} è l'attivazione dell'input del neurone al peso w , e δ_{out} è l'errore dell'output del neurone dal peso w . Ingrandendo per guardare solo il peso w , e i due neuroni collegati da quel peso, possiamo rappresentarlo come:

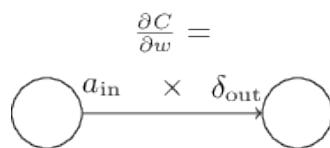


Figura 4.14: Rappresentazione dell'equazione (4.29).

Come si può intuire, combinando le equazioni (4.24) e (4.26), possiamo computare l'errore $\delta^{(l)}$ per ogni layer della rete, facendo propagare l'errore dall'output layer fino all'input layer. Mentre le equazioni (4.24) e (4.24) mostrano le relazioni tra δ e le derivate parziali calcolate rispetto a w e b .

4.8.3 Applicazione della Backpropagation

Nella pratica è bene combinare l'algoritmo di *backpropagation* con un'algoritmo basato sul *gradient descent*. Supponendo di utilizzare un mini batch gradient descent, con batch size di $m < n$ elementi (dove n è la dimensione del dataset), è possibile vedere come l'azione combinata dei due algoritmi agisce sul processo di training [50]:

1. **Input:** Selezione casuale di m valori di input all'interno del dataset
2. **Feedforward:** $\forall x_i \in \{x_1, x_2, \dots, x_m\}$ e $\forall l \in \{2, 3, \dots, L\}$ si computa $z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$, $a^{(l)} = \sigma(z^{(l)})$
3. **Output error:** Si calcola il vettore $\delta^{(L)} = \nabla_a C \odot \sigma'(z^{(L)})$
4. **Backpropagate the error:** $\forall l \in \{L-1, L-2, \dots, 3, 2\}$ si calcola $\delta^{(l)} = ((W^{(l+1)})^T \delta^{(l+1)}) \odot \sigma'(z^{(L)})$
5. **Gradient descent:** $\forall l \in \{L-1, L-2, \dots, 3, 2\}$ si esegue:

- aggiornamento di w :

$$w_i \rightarrow w'_i = w_i - \frac{\eta}{m} \cdot \sum_x (a^{(l-1)})^T \delta^{(l)} \quad (4.30)$$

- aggiornamento di b :

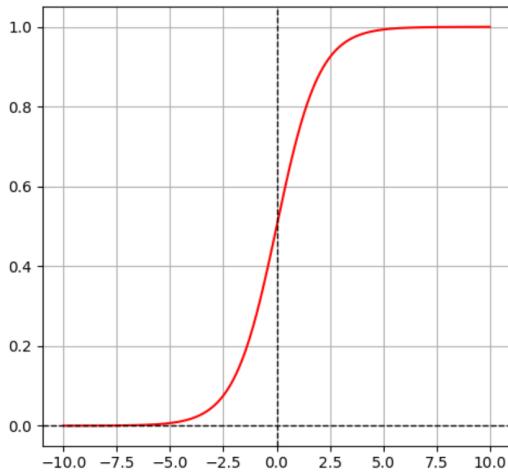
$$b_i \rightarrow b'_i = b_i - \frac{\eta}{m} \cdot \sum_x \delta^{(l)} \quad (4.31)$$

4.9 Altre funzione di attivazione

La scelta di un'opportuna funzione di attivazione assume un ruolo molto importante: l'output restituito da una rete neurale è fortemente condizionato dalla funzione di attivazione utilizzata, la quale possiede un ruolo molto importante anche nel processo e nella velocità di convergenza della rete neurale e nella sua accuratezza. Inoltre, nelle moderne reti neurali profonde, la scelta della funzione d'attivazione da utilizzare dipende dal tipo di layer a cui essa è associata. Nel corso degli anni sono state utilizzate diverse funzioni di attivazione: binarie, lineari e non lineari. Diamo un'occhiata alle più comuni [58, 125, 59]:

4.9.1 Funzione Sigmoide

La funzione sigmoide è definita come segue:



- **Equazione:**

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (4.32)$$

- **Derivata:**

$$\frac{d}{dz} \sigma(z) = \sigma(z) \cdot (1 - \sigma(z)) \quad (4.33)$$

Figura 4.15: Grafico della sigmoide

Poiché la funzione sigmoide restituisce in output un valore appartenente all'intervallo $[0, 1]$, essa viene utilizzata spesso come funzione di attivazione dell'output layer per modelli di classificazione ed il valore restituito in output assume un valore probabilistico se l'output è binario.

Tuttavia, in problemi di classificazione multiclass, la sigmoide non è particolarmente adatta poiché manca di una normalizzazione globale delle probabilità tra le varie classi.

Inoltre presenta delle limitazioni, come il problema del vanishing gradient, che consiste nella progressiva riduzione del valore del gradiente man mano che si propaga all'indietro attraverso i livelli della rete neurale durante il processo di backpropagation.

Ciò comportava, se la rete presentava tanti layer interni con questa funzione di attivazione, il "congelamento" dell'apprendimento dei valori dei pesi durante la fase di training.

4.9.2 Funzione Tanh (Tangente Iperbolica)

Un'altra funzione di attivazione utilizzata è la funzione tangente iperbolica (tanh), la quale restituisce valori all'interno dell'intervallo $[-1, 1]$.

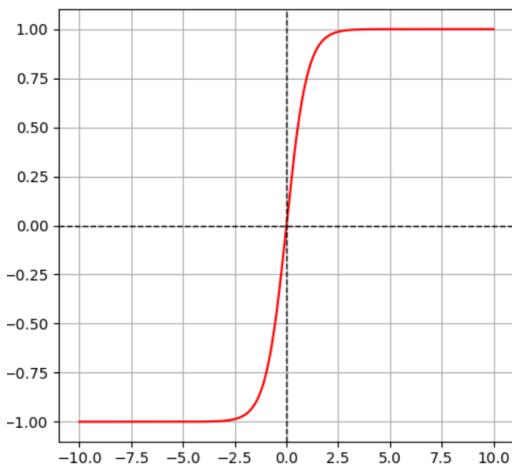


Figura 4.16: Grafico della Tanh

- **Equazione:**

$$\sigma(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (4.34)$$

- **Derivata:**

$$\frac{d}{dz} \sigma(z) = 1 - \sigma(z)^2 \quad (4.35)$$

La funzione mappa l'input in un intervallo tra -1 e 1, utile per normalizzare i dati. Anche questa funzione può soffrire del vanishing gradient.

4.9.3 ReLU (Rectified Linear Unit)

La funzione ReLU è così definita:

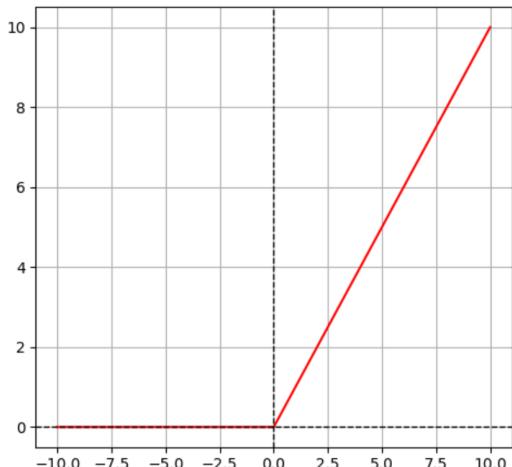


Figura 4.17: Grafico della Tanh

- **Equazione:**

$$\sigma(z) = \max(0, z) \quad (4.36)$$

- **Derivata:**

$$\frac{d}{dz} \sigma(z) = \begin{cases} 1 & \text{se } z > 0, \\ 0 & \text{se } z \leq 0. \end{cases} \quad (4.37)$$

Essa sarà quindi uguale a z per $z > 0$, mentre sarà nulla per tutti gli altri valori. È la funzione di attivazione maggiormente utilizzata nelle reti neurali artificiali e ha sostituito nel corso degli anni le funzioni tangente iperbolica e sigmoide.

Il motivo principale è dovuto al fatto che la tanh e la sigmoide hanno una derivata prima molto piccola, la quale tende velocemente a 0. Poiché il training di una rete neurale si basa sulla discesa del gradiente, la moltiplicazione per un valore prossimo a 0 porta i layer più profondi

della rete ad un apprendimento più lento. È inoltre molto facile da calcolare (da un punto di vista computazionale, è necessario solamente un confronto per determinarne l'output) e ciò si traduce, nel caso di reti neurali molto profonde, in un notevole risparmio in termini di costo computazionale

4.9.4 Leaky ReLU

La funzione Leaky ReLU è definita come:

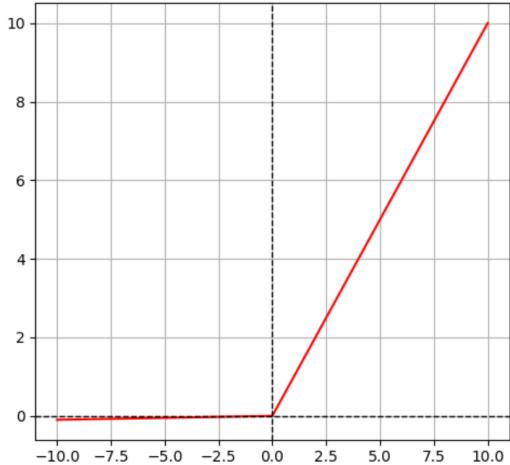


Figura 4.18: Grafico della Tanh

- **Equazione:**

$$\sigma(z) = \begin{cases} z & \text{se } z > 0, \\ \alpha z & \text{se } z \leq 0, \end{cases} \quad (4.38)$$

dove $\alpha > 0$ è un parametro fissato.

- **Derivata:**

$$\frac{d}{dz}\sigma(z) = \begin{cases} 1 & \text{se } z > 0, \\ \alpha & \text{se } z \leq 0. \end{cases} \quad (4.39)$$

Leaky relu è il miglioramento della funzione Relu. La funzione Relu può uccidere alcuni neuroni in ogni iterazione, questo è noto come condizione di relu morente. Leaky relu può superare questo problema, invece di dare 0 per valori negativi, utilizzerà una componente di input relativamente piccola per calcolare l'output, quindi non ucciderà mai alcun neurone.

4.9.5 ELU (Exponential Linear Unit)

La funzione ELU è definita come:

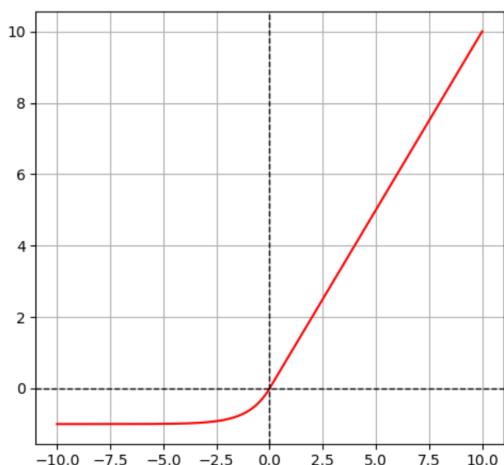


Figura 4.19: Grafico della Tanh

- **Equazione:**

$$\sigma(z) = \begin{cases} z & \text{se } z > 0, \\ \alpha(e^z - 1) & \text{se } z \leq 0, \end{cases} \quad (4.40)$$

dove $\alpha > 0$ è un parametro fissato.

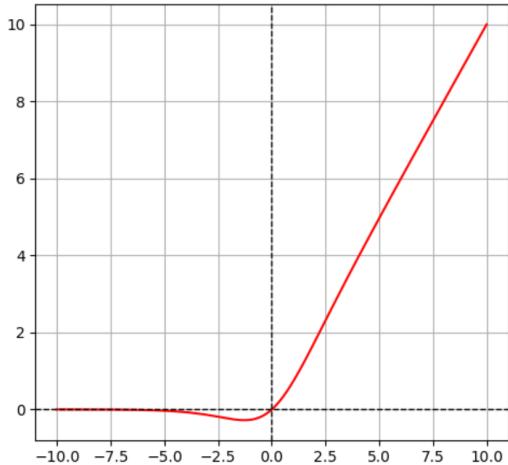
- **Derivata:**

$$\frac{d}{dz}\sigma(z) = \begin{cases} 1 & \text{se } z > 0, \\ \alpha e^z & \text{se } z \leq 0. \end{cases} \quad (4.41)$$

L'ELU è un'altra variazione di Relu che cerca di rendere le attivazioni più vicine allo zero che accelera l'apprendimento. Ha mostrato una migliore accuratezza della classificazione rispetto a Relu. Gli ELU hanno valori negativi che spingono la media delle attivazioni più vicino a zero.

4.9.6 Swish

La funzione Swish è così definita come:



- **Equazione:**

$$\sigma(z) = z \cdot \text{sigmoid}(z) = \frac{z}{1 + e^{-z}} \quad (4.42)$$

- **Derivata:**

$$\frac{d}{dz} \sigma(z) = \sigma(z) + \text{sigmoid}(z) \cdot (1 - \sigma(z)) \quad (4.43)$$

Figura 4.20: Grafico della Tanh

Questa funzione combina le proprietà della ReLU e della sigmoide, risultando spesso in migliori prestazioni.

La funzione Swish è stata proposta dal team Brain di Google. I loro esperimenti hanno dimostrato che la Swish tende a funzionare più velocemente della Relu in modelli profondi su diversi set di dati impegnativi.

4.9.7 Softmax

Per problemi di classificazione, la funzione di attivazione sigmoide si presta bene nel caso in cui si hanno solamente due classi. Qualora il numero di classi sia maggiore, si utilizza la funzione di attivazione softmax, così definita [60, 59]:

$$\sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}, \quad \text{per } i = 1, \dots, K \quad (4.44)$$

Softmax è fondamentalmente una funzione vettoriale. Prende un vettore come input e produce un vettore come output. In altre parole, ha più ingressi e uscite.

La funzione di attivazione Softmax viene usata sempre nell'output layer, facendo in modo che l'output della rete assuma un'interpretazione probabilistica.

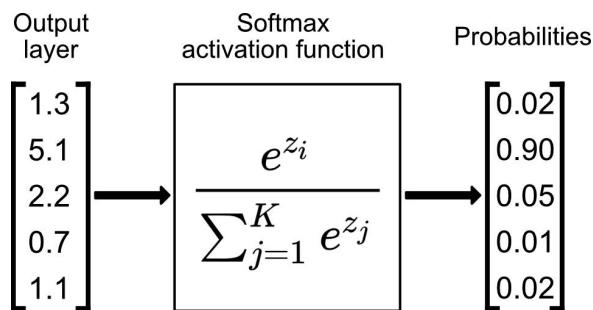


Figura 4.21: Illustrazione del funzionamento della softmax [59].

Come si può intuire, la somma di tutte le probabilità è uguale a 1. In altre parole

$$\sum_{i=1}^K \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} = 1$$

4.10 La Cross-Entropy

Un'altra funzione di attivazione largamente utilizzata per problemi di classificazione è la **Cross-Entropy (Entropia crociata)** [50, 125].

Il concetto di cross-entropy affonda le sue radici nel campo della teoria dell'informazione, dove l'entropia dell'informazione, nota anche come entropia di Shannon, fu introdotta formalmente nel 1948 da Claude Shannon in un articolo intitolato "A Mathematical Theory of Communication". La Cross-Entropy si basa sul concetto dell'entropia, che calcola il grado di casualità o disordine di un sistema. Nel contesto della teoria dell'informazione, l'entropia di una variabile casuale X è l'incertezza media, la sorpresa o l'informazione inerente ai possibili risultati. In parole povere, misura l'incertezza di un evento.

$$H(X) = - \sum_{i=0}^n p(x_i) \cdot \ln(p(x_i)) \quad (4.45)$$

Maggiore è il valore dell'entropia, $H(x)$, maggiore è l'incertezza della distribuzione di probabilità, mentre minore è il valore, minore è l'incertezza. Ovvero, più è alto il valore dell'entropia più è difficile prevedere il valore che assumerà la variabile casuale X . La Cross-Entropy sfrutta questo concetto per misurare la differenza tra la distribuzione di probabilità predetta dal modello di classificazione e i valori reali. Maggiore è la differenza tra i due e maggiore è la loss [57, 56, 55]. Per problemi di classificazione binaria la cross-entropy, definita in questo caso anche come binary cross-entropy, è così definita [57, 50]:

$$C(w, b) = -\frac{1}{N} \sum_{i=1}^N \left[y_i \ln(a_i^{(L)}) + (1 - y_i) \cdot \ln(1 - a_i^{(L)}) \right] \quad (4.46)$$

La binary cross-entropy è comunemente utilizzata nelle reti neurali con una funzione di attivazione sigmoide nello strato di uscita. Mentre per problemi di classificazione su più classi, la cross-entropy, chiamata in questo caso anche come categorical cross-entropy, è definita come:

$$C(w, b) = - \sum_{i=1}^N \left[y_i \ln(a_i^{(L)}) \right] \quad (4.47)$$

La categorical cross-entropy è comunemente utilizzata nelle reti neurali con una funzione di attivazione softmax nello strato di uscita. Minimizzando la perdita, il modello impara ad assegnare probabilità più elevate alla classe corretta e a ridurre le probabilità per le classi errate, migliorando l'accuratezza.

Capitolo 5

Reti Neurali Convoluzionali (CNN)

Proposta una panoramica molto generale sulle Reti Neurali, si può adesso scendere più nel dettaglio analizzando quelle che sono le **reti convoluzionali**. Il nome “rete neurale convoluzionale” indica una tipologia di rete neurale che impiega un’operazione matematica lineare chiamata appunto **convoluzione**. Le Reti Neurali Convoluzionali (CNN o Convolutional Neural Networks) sono un tipo di rete neurale particolarmente efficace per l’elaborazione di dati strutturati in forma di griglie, come le immagini. Sono ampiamente utilizzati in compiti di visione artificiale, come il riconoscimento di immagini, la classificazione di oggetti, il rilevamento di volti e in molte altre applicazioni. Le CNN sono progettate per riconoscere dei pattern visivi in modo diretto e non richiedono molto preprocessing o comunque ne richiedono una quantità molto limitata; si ispirano al modello della corteccia visiva animale: i singoli neuroni in questa parte del cervello rispondono solamente a stimoli relativi ad una zona ristretta del campo di osservazione detto campo recettivo.

5.1 Cenni Matematici

5.1.1 L’operazione di convoluzione

Nella sua forma più generale, la convoluzione è un’operazione di combinazione su due funzioni che restituisce una terza funzione. Più formalmente, la convoluzione di due funzioni $f(t)$ e $g(t)$, chiamate rispettivamente **funzione di input** e **kernel**, è definita come l’integrale del prodotto di $f(t)$ con una versione traslata e rovesciata di $g(t)$. Formalmente può essere descritto come:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t - \tau)d\tau \quad (5.1)$$

Il concetto di convoluzione qui può essere inteso come il ”ribaltamento” della funzione $g(t)$, seguito dal ”traslare” lungo la funzione $f(t)$, moltiplicando punto per punto, e integrando per ottenere la nuova funzione risultante [63, 125].

5.1.2 La convoluzione discreta

In pratica, soprattutto in applicazioni digitali (come nelle reti neurali convoluzionali), si usa una versione discreta della convoluzione. Per due sequenze discrete $f(n)$ e $g(n)$, definite su \mathbb{Z} , la convoluzione discreta è definita come:

$$S(n) = (f * g)(n) = \sum_{a=-\infty}^{\infty} f(a) \cdot g(n - a) \quad (5.2)$$

Questa formula è molto simile a quella della convoluzione continua, con la differenza che al posto dell’integrale abbiamo una somma discreta [63, 125].

5.1.3 La convoluzione nelle CNN

Tuttavia, nelle reti convoluzionali, l'input è un array multidimensionale di dati mentre il kernel è un array multidimensionale di parametri che dipendono dall'algoritmo di learning scelto. Entrambi questi array spesso sono chiamati **Tensori**. Riprendendo la formula (5.2), la sommatoria infinita potrà essere vista come una sommatoria su un numero finito di elementi di un array, su cui applicare la convoluzione in uno o più assi: ad esempio se ricevessimo una immagine I in 2D come input dovremmo usare un kernel in due dimensioni [125]. Quindi la formula (5.2) può essere riscritta come:

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n) \cdot K(i - m, j - n) \quad (5.3)$$

Considerando anche la commutatività della convoluzione:

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i - m, j - n) \cdot K(m, n) \quad (5.4)$$

Talvolta, le equazioni (5.3) e (5.4) sono indicate in modo improprio come **funzioni di correlazione (correlation functions)**.

5.1.4 Cross-Correlation

Nelle applicazioni pratiche, si usa spesso una variante semplificata della correlazione, chiamata **cross-correlation (correlazione incrociata)**. La differenza principale risiede nel fatto che non si effettua l'inversione del kernel (flipping del kernel) [125].

La cross-correlation è descritta dalla seguente formula:

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i + m, j + n) \cdot K(m, n) \quad (5.5)$$

L'operazione di cross-correlation può essere visualizzato come:

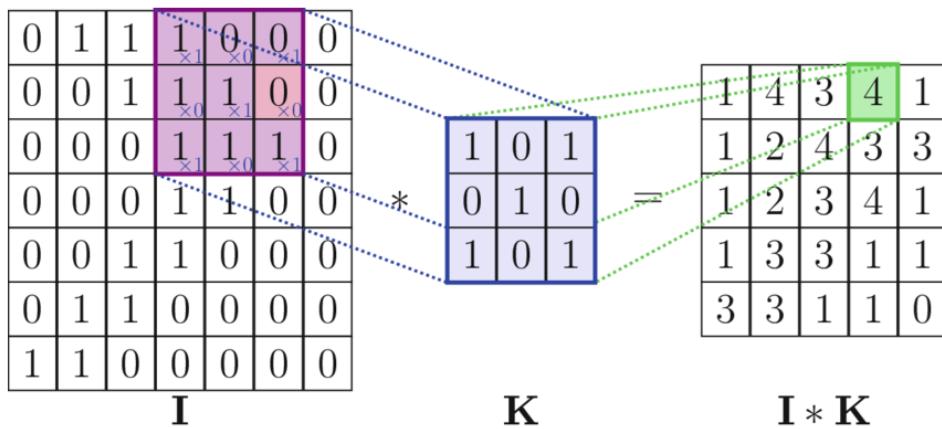


Figura 5.1: Esempio di una convoluzione [65, 125].

In alcuni casi, viene utilizzata una variante della formula (5.5). Questa formula si differisce soltanto dall'utilizzo di un bias, una costante che viene aggiunta al risultato finale della convoluzione.

$$S(i, j) = (I * K)(i, j) = b + \sum_m \sum_n I(i + m, j + n) \cdot K(m, n) \quad (5.6)$$

5.1.5 La convoluzione in tre dimensioni

Un altro tipo di convoluzione che si utilizza spesso è la convoluzione 3D. Essa consiste nell'applicare un filtro a 3 dimensioni al set di dati. Questo filtro si muove in 3 direzioni (x, y, z) per calcolare le rappresentazioni delle caratteristiche a basso livello [71].

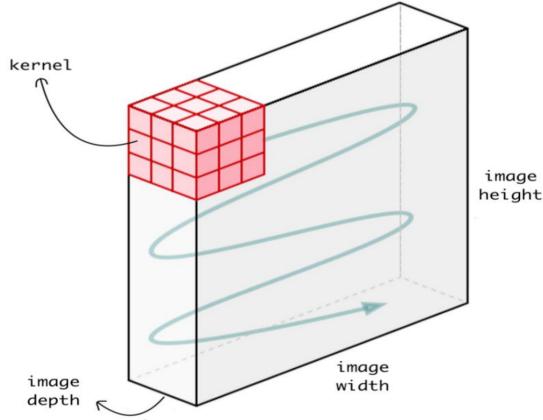


Figura 5.2: Rappresentazione convoluzione 3D [73].

Questa operazione può essere descritta come:

$$S(i, j, k) = (I * K)(i, j, k) = b + \sum_m \sum_n \sum_p I(i + m, j + n, k + p) \cdot K(m, n, p) \quad (5.7)$$

Questa tipologia di convoluzione è particolarmente utile per dati volumetrici come video, immagini mediche 3D, ecc. La convoluzione 3D può essere applicata anche a input in spazio 2d, come le immagini. Oppure anche a sequenze di immagini, come avviene nel telerilevamento, in cui si possono avere delle sequenze temporali di immagini [71, 73].

Quindi è una convoluzione particolarmente adatta per operare con dati che hanno un'estensione temporale o spaziale.

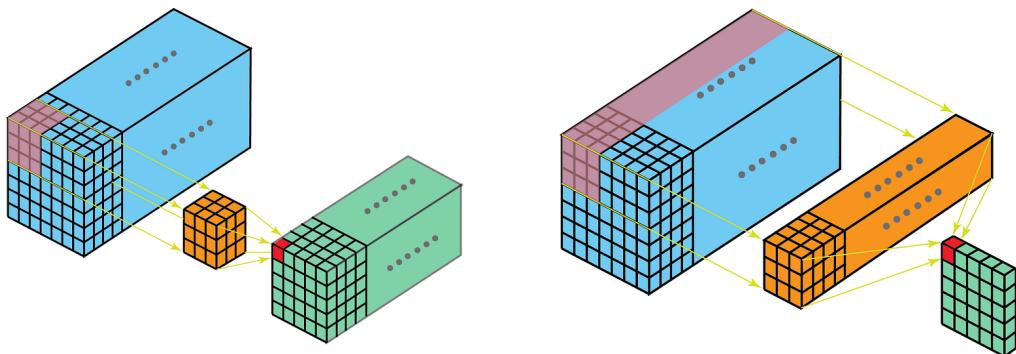


Figura 5.3: Dimensione delle feature map al variare del kernel [74].

5.2 Aspetti e parametri della convoluzione

5.2.1 Convoluzione su più canali

Nella pratica, la maggior parte delle immagini in ingresso ha 3 o più canali. E questo numero aumenta quanto più ci si addentra in una rete.

È qui che si rende utile una distinzione chiave tra i termini: mentre nel caso di 1 canale i termini filtro e kernel sono intercambiabili, nel caso generale sono piuttosto diversi. Ogni filtro è in realtà un insieme di kernel, con un kernel per ogni singolo canale di ingresso allo strato e ogni kernel è unico. Ogni filtro di uno strato di convoluzione produce un solo canale di uscita e lo fa in questo modo: Ciascuno dei kernel del filtro “scivola” sui rispettivi canali di ingresso, producendo una versione elaborata di ciascuno di essi. Alcuni kernel possono avere pesi più forti di altri, per dare maggiore enfasi a determinati canali di ingresso rispetto ad altri (ad esempio, un filtro può avere un kernel del canale rosso con pesi più forti di altri, e quindi rispondere maggiormente alle differenze nelle caratteristiche del canale rosso rispetto agli altri). Tutti i canali elaborati vengono poi sommati insieme al bias per formare il valore del canale in uscita [72, 65].

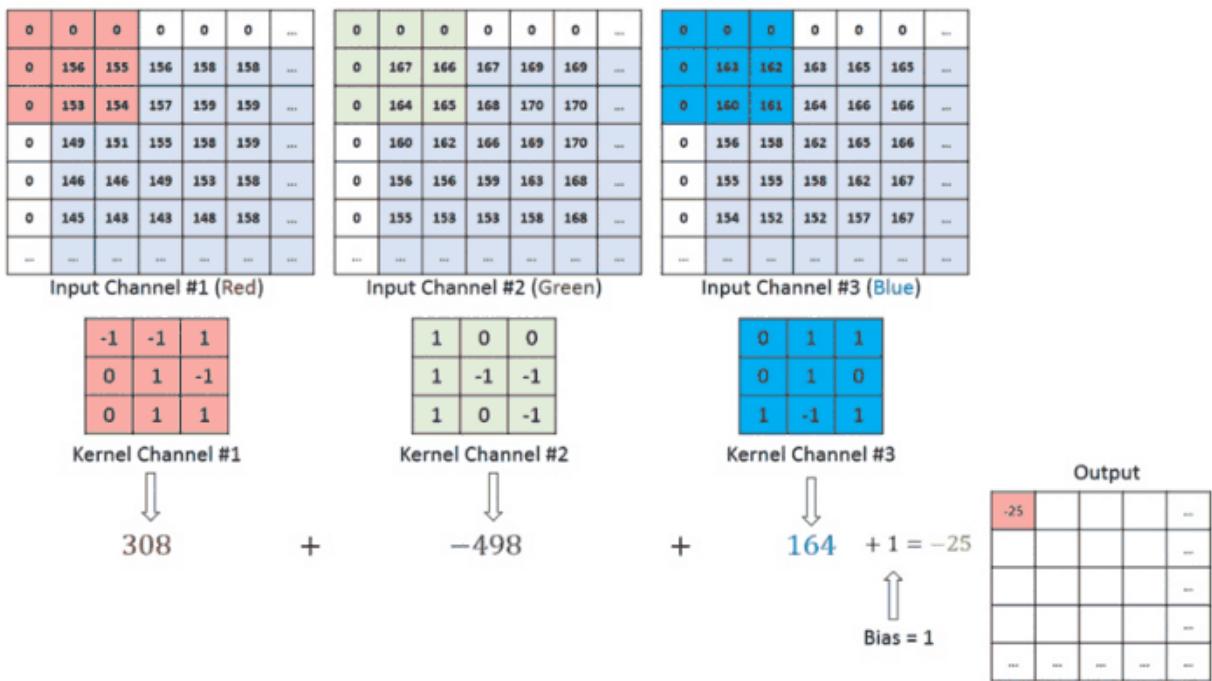


Figura 5.4: Illustrazione dell'operazione di convoluzione su più canali [65].

Come possiamo osservare dalla figura (5.4), questa operazione è molto simile alla convoluzione 3D. Infatti, in alcuni casi è possibile anche utilizzare la convoluzione 3D definendo la profondità come il numero dei canali dell'immagine. Tuttavia, nelle sequenze di immagini, si hanno più immagini composte da più canali. In questo caso si esegue la stessa operazione dalla figura (5.4): si utilizza un kernel 3D per ogni canale ma che opera su tutta la sequenza.

5.2.2 Parametri della convoluzione

In Pytorch il comportamento della convoluzione è regolato principalmente dai seguenti parametri [64]:

- **Kernel size:** Questo parametro determina la dimensione del kernel. In genere vengono utilizzati kernel di dimensioni ridotte, preferibilmente valori dispari come: 1x1, 3x3, 5x5 e 7x7. In alcune applicazioni, capita anche di utilizzare kernel più grandi, come 11x11.
- **Stride:** Lo stride è il numero di pixel con cui si fa scorrere la matrice dei filtri sulla matrice di input. Quando lo stride è 1, i filtri vengono spostati di un pixel alla volta. Quando lo stride è 2, i filtri saltano di 2 pixel alla volta mentre vengono fatti scorrere. Uno stride maggiore produce mappe di caratteristiche più piccole.

- **Padding:** A volte è conveniente espandere la matrice di input (l'immagine) con degli zeri intorno al bordo, in modo da poter applicare il filtro agli elementi confinanti della matrice dell'immagine di input. Una caratteristica interessante del padding a zero è che ci permette di controllare la dimensione delle mappe di caratteristiche. L'aggiunta di zero padding è chiamata anche convoluzione ampia, mentre l'assenza di zero padding sarebbe una convoluzione stretta. Il valore da utilizzare come padding è arbitrario ma tipicamente si utilizza il valore 0.
- **Numero di filtri / canali in uscita:** Questo parametro corrisponde al numero di filtri utilizzati per l'operazione di convoluzione. Ogni filtro viene utilizzato per creare un nuovo canale sull'immagine di uscita. Tutti questi canali corrispondono alle feature map.
- **Dilatation:** Questo parametro viene utilizzato per controllare la spaziatura tra i punti del kernel.

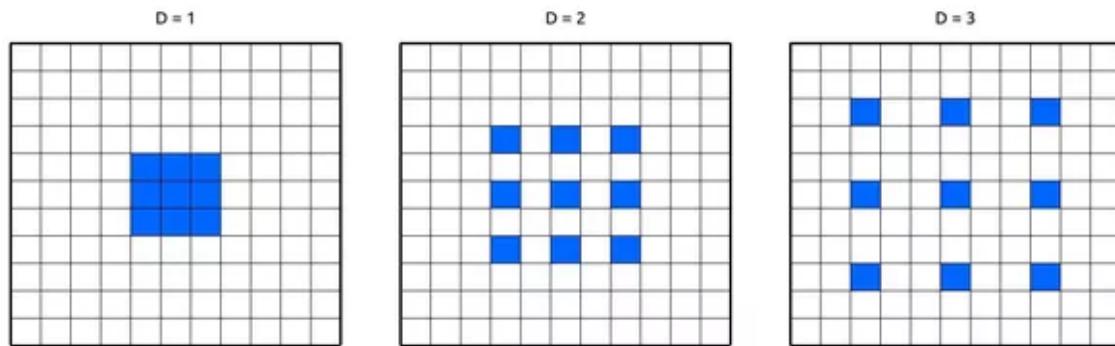


Figura 5.5: Rappresentazione della Dilatation [71].

5.3 Struttura di una CNN

L'architettura di una rete neurale convoluzionale è meticolosamente progettata per estrarre caratteristiche (o features) significative da dati visivi complessi. Ciò è possibile tramite l'uso di livelli specializzati all'interno dell'architettura di rete. Un CNN presenta la seguente struttura:

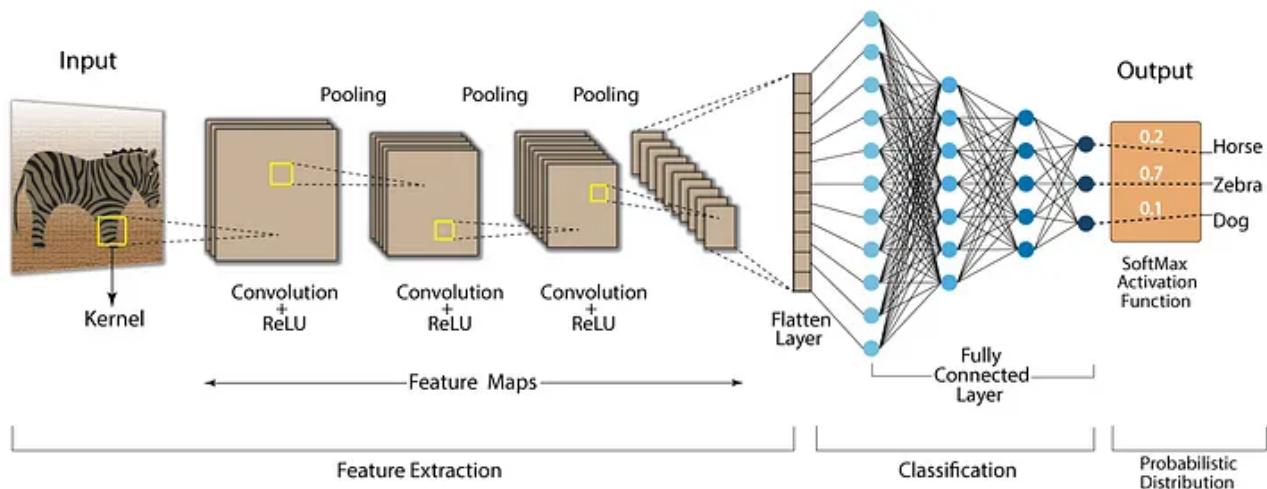


Figura 5.6: Esempio della struttura di un CNN [65]

Come si può osservare, la rete di una CNN è costituita da due blocchi principali: I blocchi della *features extraction* (estrazione delle caratteristiche), responsabile dell'estrazione di tutte

le caratteristiche e pattern dell'immagine; E il blocco della classification (classificazione), che si occupa di processare tutte le features map (mappe delle caratteristiche) create dal blocco precedente e predire il risultato di ogni classe [65, 66, 67].

All'interno del blocco estrazione delle features sono presenti i seguenti layer:

- **Input Layer:** Esso è il primo layer ed è quello che riceve l'immagine, ed eventualmente, la ridimensiona prima di passarla ai layer successivi.
- **Convolutional Layer:** Questo layer si occupa di estrarre le features, ovvero le caratteristiche significative delle immagini. In pratica si cercano di individuare dei pattern, come ad esempio curve, angoli, circonferenze o quadrati raffigurati in un'immagine con elevata precisione. I livelli convolutivi possono essere molteplici ma questo dipende dall'architettura di rete: maggiore è il loro numero, maggiore è la complessità delle caratteristiche che riescono ad individuare.
- **Activation Layer:** Questo layer viene usato sempre dopo un layer convoluzionale. Questo layer consiste nell'applicazione, su singolo pixel, della funzione di attivazione, tipicamente la ReLU, permettendo così di introdurre **non linearità** nella rete.
- **Pooling Layer:** Questo livello viene utilizzato per rendere il set di features map piccolo e gestibile riducendo il numero di parametri sulla rete e di conseguenza ottimizzando la computazione; rende inoltre la rete invariante alle piccole trasformazioni quali distorsione o traslazione rispetto all'immagine iniziale. Questa operazione permette anche di preservare le informazioni fondamentali.

All'interno di una CNN, i layer di convoluzione e di pooling, posso ripetersi più volte. Inoltre, ci possono essere molteplici layer convoluzionali in sequenza prima di un layer di pooling.

Per quanto riguarda il blocco della classificazione, esso è solitamente posto alla fine della rete e si occupa di prendere le immagini filtrate ad alto livello e tradurre in categorie ciò che ha analizzato, quindi il suo scopo è quello di usare le features per classificare le immagini. Ogni classe rappresenta una possibile risposta finale che il computer darà. Tipicamente si tratta di un MultiLayer Perceptron che usa alla fine una funzione di attivazione **Softmax**.

5.4 Algoritmi di Pooling

Nell'architettura di una CNN è pratica comune inserire degli strati di Pooling, la cui funzione è quella di ridurre la dimensione spaziale degli input (larghezza e altezza), in modo da diminuire il numero di parametri e il carico computazionale [65, 66, 67] .

5.4.1 Average-Pooling

L'*Average-Pooling* consiste nel calcolare la media degli elementi presenti nella regione della feature map coperta dal filtro.

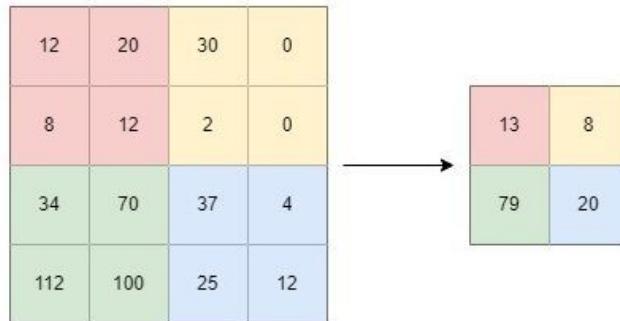


Figura 5.7: Esempio di un Average-Pooling [68].

5.4.2 Max-Pooling

Il *Max pooling* è un'operazione di pooling che seleziona l'elemento massimo dalla regione della feature map coperta dal filtro. Quindi, l'output dopo il layer di max-pooling sarebbe una feature map contenente le feature più importanti della feature map precedente.

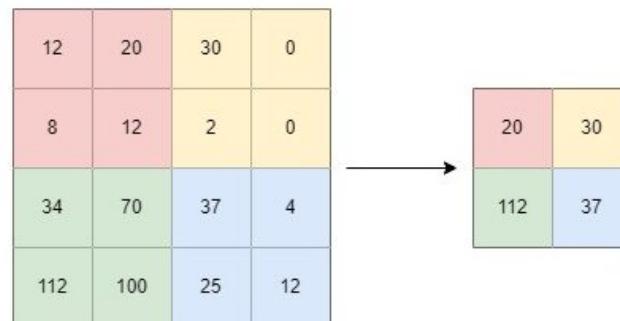


Figura 5.8: Esempio di un Max-Pooling [68].

Capitolo 6

Image segmentation

In questo capitolo verranno trattate le basi teoriche dell'*image segmentation*, l'argomento di sperimentazione su cui si basa la tesi.

6.1 Introduzione all'image segmentation

L'*image segmentation* (segmentazione dell'immagine) [114, 115, 113, 116] è una tecnica fondamentale della computer vision che consiste nel dividere un'immagine in diversi segmenti o regioni, ognuno dei quali rappresenta una parte specifica dell'immagine. Questo processo consente alle macchine di comprendere gli elementi all'interno di un'immagine in modo più preciso rispetto a compiti come il rilevamento di oggetti.

La segmentazione è di solito utilizzata per localizzare oggetti e bordi (linee, curve, ecc.), che definiscono le diverse aree di un'immagine. Più precisamente, la segmentazione è il processo con il quale si classificano i pixel dell'immagine per una qualche proprietà o caratteristica (colore, intensità o texture), assegnando a questi pixel una *label* (etichetta) di classe.

Il risultato di un'immagine segmentata è un insieme di segmenti che, collettivamente, coprono l'intera immagine.



Figura 6.1: Esempio applicativo dell'image segmentation [116].

Nella scena di strada sopra, ci sono 5 classi: strada (rosa), veicoli (rosso), edifici (giallo), natura (verde), cielo (blu). Ad ogni pixel dell'immagine è stato assegnato una di queste classi. Però a volte, oltre a distinguere le diverse classi, si vuole anche essere in grado di distinguere diversi elementi appartenenti alla stessa classe. Ad esempio distinguere due auto o alberi. A questo scopo esistono diverse tipologie di segmentazioni, ognuna delle quali fornisce dettagli e informazioni differenti.

6.1.1 Tipologie di segmentazioni

Esistono tre tipologie di segmentazione [115]:

- L'**instance segmentation (segmentazione delle istanze)** consiste nell'identificare e delineare con precisione i singoli oggetti all'interno di un'immagine. A differenza di altri tipi di segmentazione, assegna un'etichetta unica a ogni pixel, fornendo una comprensione dettagliata delle istanze distinte presenti nella scena.
- La **Semantic segmentation (segmentazione semantica)** prevede la classificazione di ogni pixel di un'immagine in categorie predefinite. L'obiettivo è comprendere il contesto generale della scena, assegnando etichette alle regioni in base al loro significato semantico condiviso.
- La **Panoptic segmentation (segmentazione panottica)** rileva anche istanze distinte di ciascun tipo di oggetto. In altre parole, la segmentazione panottica assegna a ogni pixel di un'immagine due etichette: un'etichetta semantica e un ID istanza. Gli ID di istanza distinguono le istanze, mentre i pixel con la stessa etichetta sono considerati appartenenti alla stessa classe semantica. A differenza della segmentazione per istanze, la segmentazione panottica assegna un'etichetta distinta a ogni pixel corrispondente a un'istanza individuale per evitare un'interpretazione errata delle informazioni.

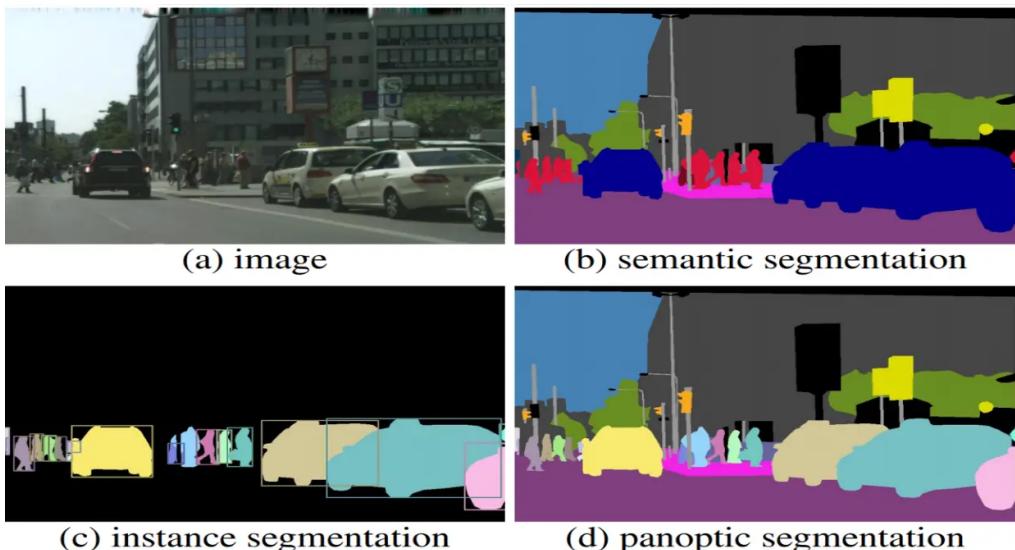


Figura 6.2: Rappresentazione applicativa delle diverse tipologie di segmentazioni [115].

Ogni tipo serve ad uno scopo distinto nella computer vision, offrendo vari livelli di granularità nell'analisi e nella comprensione del contenuto visivo.

6.2 L'architettura U-Net

U-Net è una rete neurale convoluzionale che si basa su un architettura cosiddetta *Fully convolutional Network* (rete completamente convoluzionale). L'U-Net è un'architettura che ha guadagnato un grande popolarità grazie alle sue alte prestazioni nelle attività di segmentazione delle immagini. Introdotta nel 2015 da Olaf Ronneberger, Philipp Fischer e Thomas Brox nell'articolo *"Convolutional Networks for Biomedical Image Segmentation"* [120], U-Net era stata originariamente pensata per affrontare problemi di segmentazione nel campo delle immagini mediche. Ma grazie alla sua flessibilità ed efficacia, l'architettura ha trovato applicazione in numerosi altri settori. U-Net è stata progettata principalmente per funzionare con dati di immagini in scala di grigi. Ma può essere facilmente adattata per gestire immagini multicanale o anche volumi 3D. Esistono molte varianti di questa architettura, che differiscono principalmente su parametri delle convoluzioni e dei polling. Esistono delle versioni che aumentano il numero degli strati o il numero di convoluzioni in ogni strato, migliorando la capacità della rete di apprendere rappresentazioni più complesse. Tuttavia, l'architettura originale è strutturata come segue:

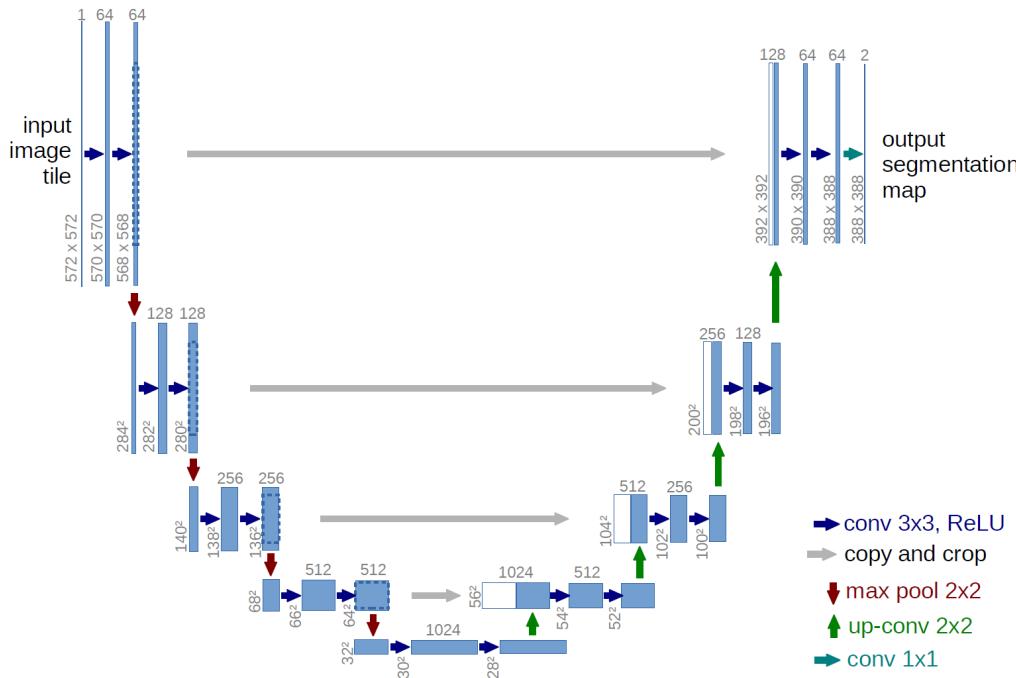


Figura 6.3: Rappresentazione dell'architettura U-Net [120].

Come possiamo osservare dalla figura 6.3, l'U-Net si basa su una struttura **encoder-decoder**, che caratterizza la sua forma ad "U" da cui prende il nome. L'*encoder* e il *decoder* svolgono le seguenti funzioni:

- **L'encoder** viene utilizzato per comprimere l'immagine di ingresso in una rappresentazione dello spazio latente (uno spazio multidimensionale astratto contenente valori caratteristici che non si possono interpretare direttamente, ma che sono codificati in una rappresentazione interna significativa [112]) attraverso convoluzioni e downsampling [117].
- Il **decoder** viene utilizzato per estrarre la rappresentazione latente in un'immagine segmentata, attraverso convoluzioni e upsampling [117].

6.2.1 skip connections

Le lunghe frecce grigie che attraversano la "U" sono le **skip connections** (o connessioni di salto) e hanno due scopi principali:

- Durante *forward* dei dati, consentono al *decoder* di accedere alle informazioni dell'*encoder*.
- Durante la *backward*, agiscono come una "superstrada del gradiente" (*gradient superhighway*) per il flusso dei gradienti dal *decoder* all'*encoder*.

6.2.2 Up-Convolution

Nella figura 6.3, le frecce verdi rappresentano l'operazione di **Up-Convolution**. L'*up-convolution*, nota anche come deconvoluzione o convoluzione trasposta, è un metodo utilizzato per sovraccampionare (upsample) le immagini e recuperare le informazioni spaziali.

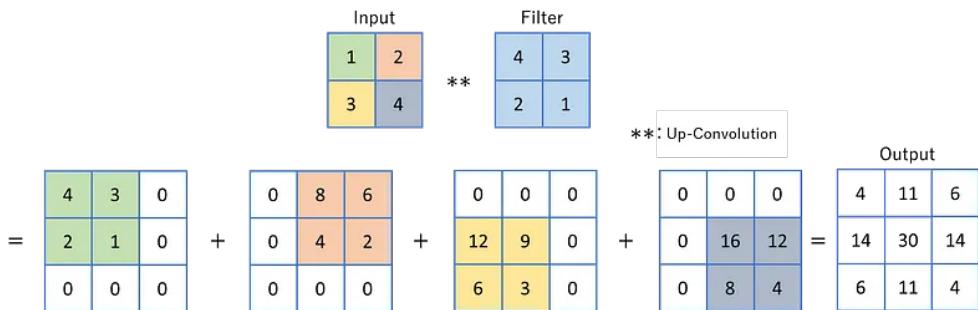


Figura 6.4: Rappresentazione dell'operazione di Up-Convolution [117].

6.2.3 La funzione di attivazione nella convoluzione di uscita

Nella convoluzione di uscita, la funzione di attivazione che segue la convoluzione dipende dal numero di classi che vogliamo distinguere. Nei casi di *Binary Segmentation* (ovvero quando dobbiamo distinguere due classi), tipicamente si utilizza la sigmoide, che restituisce dei valori tra 0 e 1. Questi valori possono essere interpretati come delle probabilità.

Mentre quando si hanno più di 2 classi, la funzione di attivazione che viene utilizzata è la softmax, la quale opera sui canali di ogni pixel. Ridefinendo i valori di ogni singolo canale del pixel come una distribuzione di probabilità.

6.2.4 Output della rete

Il risultato della rete è un'immagine composta da tanti canali, tanti quanto sono le classi che vogliamo riconoscere. Pertanto, ogni canale del pixel rappresenta la probabilità di associare al pixel la classe rappresentata dal canale. Ad esempio, la figura qui a lato mostra il risultato della rete utilizzando 5 classi.

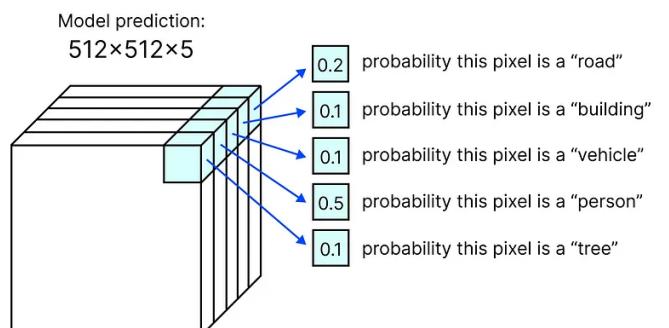


Figura 6.5: Rappresentazione dell'output della U-net [116].

Capitolo 7

Sperimentazione

In questo capitolo tratteremo la parte applicativa e sperimentale della tesi. Applicheremo tutti i concetti acquisiti nei capitoli precedenti per realizzare delle reti neurali.

7.1 Primo approccio alle reti neurali

Prima di addentrarsi nella realizzazione di reti neurali più complesse per gli obiettivi prefissati dalla tesi, iniziamo a comprendere come realizzare una semplice rete neurale e di come funziona tutta la parte relativa all'addestramento.

7.1.1 Descrizione della sperimentazione

Come prima sperimentazione, possiamo provare ad applicare una rete neurale profonda (DNN), descritta nella sezione (Multi-layer Perceptron (MLP)), a un problema di classificazione. Ad esempio provando a classificare le immagini presenti nel dataset MNIST [18, 17].

MNIST è un dataset che contiene immagini in scala di grigi di dimensioni 28×28 pixel, ciascuna raffigurante un numero intero scritto a mano compreso tra 0 e 9. Il dataset è suddiviso in un due parti: un set di addestramento composto da 60.000 immagini e un set di test composto da 10.000 immagini. Nella figura 7.1 sono mostrati alcuni esempi di immagini estratte dal dataset MNIST.

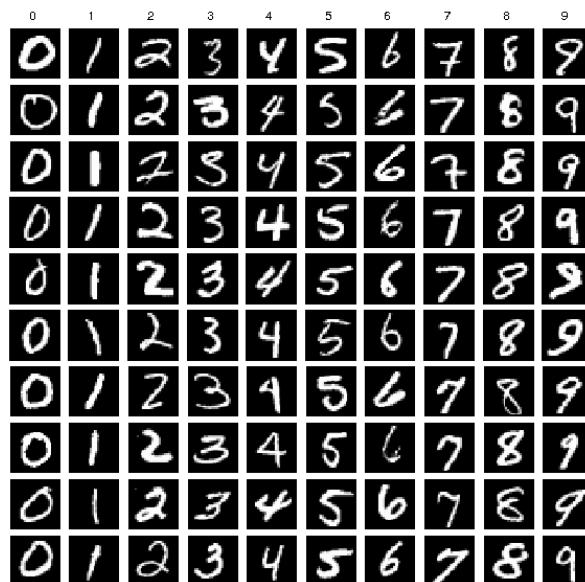


Figura 7.1: Esempio di immagini estratte dal dataset MNIST.

Come visto in precedenza (Multi-layer Perceptron (MLP)), le DNN usano come input un array (ovvero un vettore), mentre le immagini vengono rappresentate sotto forma di matrici. Pertanto, quando la DNN riceve in input un'immagine, essa deve essere rappresentata come un array di pixel di $28 \cdot 28 = 784$ elementi. Inoltre, nel dataset MNIST, ogni pixel dell'immagine assume un valore compreso nell'intervallo $[0, 255]$, che rappresenta l'intensità del colore. Nella pratica, questo valore viene normalizzato nell'intervallo $[0, 1]$. Questa operazione di normalizzazione viene effettuata per evitare che valori molto grandi causino gradienti sproporzionati durante la fase della backpropagation, riducendo così il rischio di instabilità numerica. Inoltre consente di sfruttare in modo più efficace le funzioni di attivazione e accelerare il processo di addestramento del modello, facilitando il raggiungimento di una soluzione ottimale.

7.1.2 Implementazione della rete

Una semplice rete neurale che potremmo utilizzare per questo problema di classificazione sarebbe ad esempio:

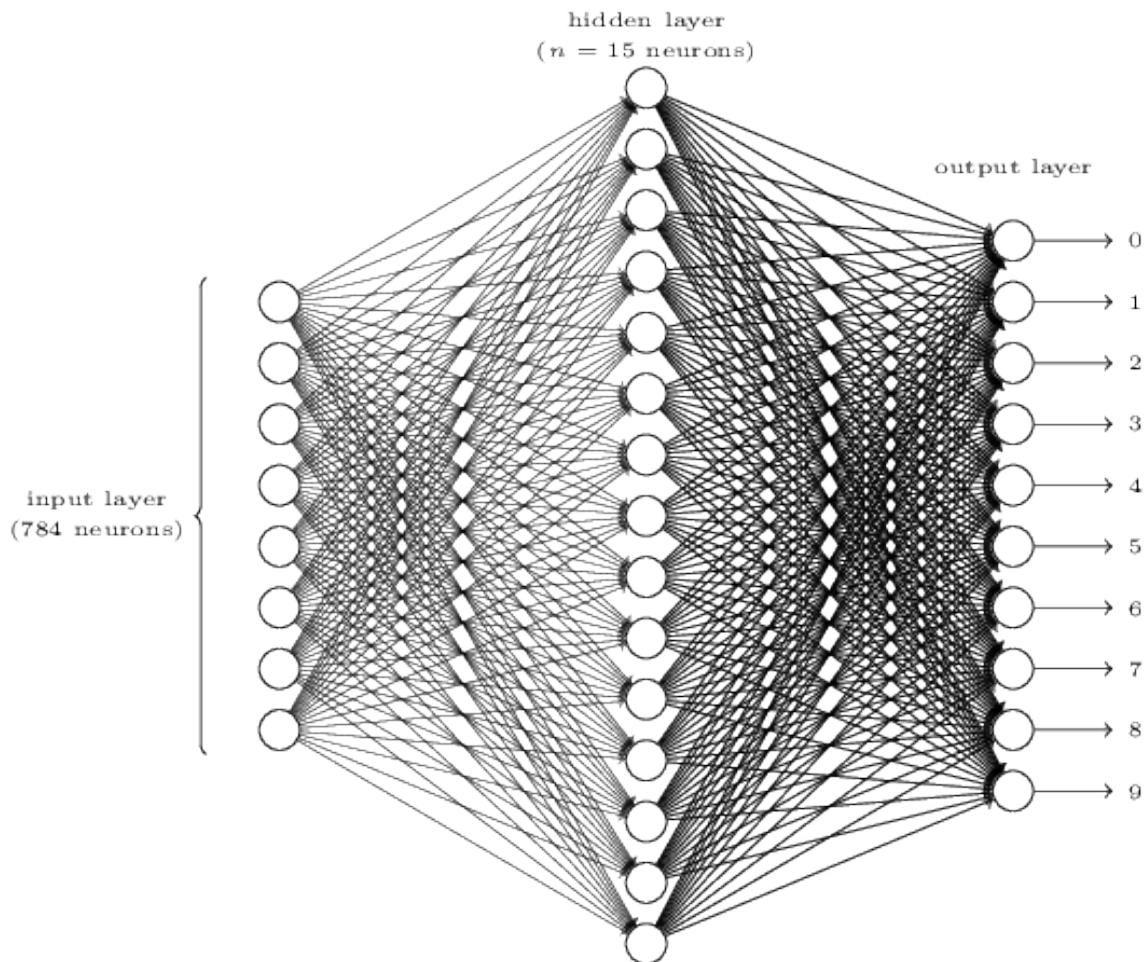


Figura 7.2: Rappresentazione della rete neurale fully connected utilizzata per risolvere un problema di classificazione sul dataset MNIST.

Come si osserva, la rete possiede: 784 neuroni di input, un hidden layer composto da 15 neuroni e un layer di output composto da 10 neuroni, corrispondenti alle 10 classi del dataset MNIST (le cifre da 0 a 9).

Implementare manualmente una rete di questo tipo sarebbe complesso. Per questo motivo, utilizzeremo il framework PyTorch per implementare la rete e gestire la fase di training.

La rete precedentemente descritta nella figura (7.2) può essere realizzata in python nel seguente modo:

```

1 import torch.nn.functional as F
2 from torch import nn
3 import torch
4 from torch.utils.data import DataLoader, Dataset
5 from torchvision import datasets, transforms
6
7 class Network(nn.Module):
8     def __init__(self, input_dim: int, hidden_dim: int, output_dim: int):
9         super(Network, self).__init__()
10
11         self._net = nn.Sequential (
12             # Operazione di Flatten dell'immagine (28x28 -> 784)
13             nn.Flatten(),
14
15             # Input layer
16             nn.Linear(input_dim, hidden_dim),
17
18             # Funzione di attivazione ReLU
19             nn.ReLU(),
20
21             # Hidden layer
22             nn.Linear(hidden_dim, output_dim),
23         )
24
25     def forward(self, x: torch.tensor) -> torch.tensor:
26         return self._net(x)
27
28     def predict(self, x: torch.tensor) -> torch.tensor:
29         return F.softmax(self._net(x), dim=1)

```

Come si può osservare, ogni qual volta bisogna creare una rete neurale o una sua componente, in Pytorch, bisogna sempre creare una classe che estenda la classe *nn.Module*. Questa classe fornisce tutte le funzioni di base per poter operare e interagire con le varie componenti del framework.

Inoltre, la funzione di attivazione *softmax*, non è stata utilizzata direttamente all'interno della rete, bensì all'esterno, in un'altra funzione. Questo perché, come si vedrà in seguito, la funzione di loss *cross-entropy* include già al suo interno un'operazione equivalente alla *softmax*.

Il modello verrà poi instanziato con i seguenti parametri:

```

1 def main():
2
3     device = torch.device("cuda" if torch.cuda.is_available() else
4                           "cpu")
4     model = Network(784, 15, 10).to(device)
5     ...

```

Il *device* corrisponde al tipo di acceleratore (CPU, GPU, TPU, ...) su cui verrà eseguita la rete neurale. In questo caso, se la GPU è disponibile, la rete neurale verrà caricata nella VRAM della scheda video, permettendo così di sfruttare la parallelizzazione della GPU accelerando i calcoli della rete neurale.

Per quanto riguarda il dataset, PyTorch fornisce già un’interfaccia per poterlo scaricare ed utilizzare, richiamabile nel seguente modo:

```
1 ...
2
3     transform = transforms.Compose([
4         transforms.ToTensor(),
5     ])
6
7     train_dataset = datasets.MNIST(root="\\tmp\\data", train=True,
8         download=True, transform=transform)
9     test_dataset = datasets.MNIST(root="\\tmp\\data", train=False,
10        download=True, transform=transform)
11
12 ...
```

Le *transforms* vengono utilizzate per applicare delle ”trasformazioni” sui dati. PyTorch mette a disposizione moltissime tipologie di *transforms* che possono essere utilizzate per diverse applicazioni. In questo caso abbiamo soltanto utilizzato la trasformazione che converte l’immagine in un tensore. Successivamente dal dataset di test, selezioniamo una porzione da utilizzare come validazione del training. E definiamo i dataloader, ovvero i componenti che si occupano di prendere i dati dal dataset. Il dataloader consente anche di specificare la dimensione della batch (ovvero il numero di ”campioni” da utilizzare per il calcolo dei gradienti) ed il numero dei workers. Ogni worker corrisponde a un processo che viene assegnato a un core differente della CPU, permettendo così di parallelizzare il caricamento dei dati dal dataset. Tipicamente vengono utilizzati 4 worker per GPU. Tuttavia, troppi worker attivi posso creare un eccessivo overhead, rallentando così il sistema. Un altro aspetto dei dataloader è la possibilità di selezionare in modo casuale gli elementi da prelevare. Impostando l’opzione *shuffle* su *True*.

```
1 ...
2     train_size = int(0.8 * len(train_dataset))
3     val_size = len(train_dataset) - train_size
4     train_subset, val_subset =
5         torch.utils.data.random_split(train_dataset, [train_size,
6             val_size])
7
8     train_loader = DataLoader(dataset=train_subset, batch_size=64,
9         shuffle=True, num_workers=4)
10    val_loader = DataLoader(dataset=val_subset, batch_size=64,
11        shuffle=False, num_workers=4)
12    test_loader = DataLoader(dataset=test_dataset, batch_size=1,
13        shuffle=False)
14
15 ...
```

Per svolgere l’addestramento della rete neurale, definiamo una funzione che accetti come argomento: il tipo di acceleratore, il modello della rete neurale, i dataloader dei dati di training e di validazione ed il numero di epoche. Questa funzione restituirà tutti i valori di loss e il valore dell’accuratezza di ogni epoca. Questi valori potranno poi essere utilizzati per rappresentare graficamente i risultati ottenuti durante il processo di addestramento.

```
1
2     def train(device, model: nn.Module, train_dataloader,
3             valid_dataloader, epochs: int) -> Tuple[np.array, np.array,
4                 np.array]:
5
```

```

4     criterion = nn.CrossEntropyLoss()
5     optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
6
7     train_losses = np.zeros(epochs)
8     valid_losses = np.zeros(epochs)
9     val_accuracies = np.zeros(epochs)
10
11    for epoch in range(epochs):
12
13        epoch_losses_train: np.array = np.zeros(len(train_dataloader))
14        epoch_losses_valid: np.array = np.zeros(len(valid_dataloader))
15
16        #Fase di training
17        model.train()
18
19        for batch_idx, (data, labels) in enumerate(train_dataloader):
20            data, labels = data.to(device), labels.to(device)
21
22            # Azzeramento dei gradienti
23            optimizer.zero_grad()
24            # Forward
25            output = model(data)
26            # Calcolo della perdita
27            loss = criterion(output, labels)
28
29            # Backpropagation e aggiornamento dei pesi
30            loss.backward()
31            optimizer.step()
32
33            epoch_losses_train[batch_idx] = loss.item()
34
35        #Fase di validation
36        model.eval()
37        correct_val = 0
38        total_val = 0
39
40        with torch.no_grad():
41            for batch_idx, (data, labels) in
42                enumerate(valid_dataloader):
43                data, labels = data.to(device), labels.to(device)
44
45                output = model(data)
46                loss = criterion(output, labels)
47                epoch_losses_valid[batch_idx] = loss.item()
48
49                # Calcolo accuratezza
50                predictions = torch.argmax(output, dim=1)
51                correct_val += (predictions == labels).sum().item()
52                total_val += labels.size(0)
53
54                #calcolo delle loss medie
55                train_losses[epoch] = epoch_losses_train.sum() /
56                    len(train_dataloader)

```

```

55         valid_losses[epoch] = epoch_losses_valid.sum() / 
56             len(valid_dataloader)
57         val_accuracies[epoch] = correct_val / total_val
58
59         print(f'Epoch [{epoch+1}/{epochs}], Training_Loss:
60               {train_losses[epoch]:.6f}, Validation_Loss:
61               {valid_losses[epoch]:.6f}, Accuracy:
62               {val_accuracies[epoch]:.4f}''')
63
64     return train_losses, valid_losses, val_accuracies

```

Per valutare l'accuratezza del modello su dati mai visti, definiamo la funzione:

```

1 def test(device, model: nn.Module, test_loader: DataLoader) ->
2     Tuple[np.array,np.array] :
3     model.eval()
4     all_preds = []
5     all_labels = []
6
7     with torch.no_grad():
8         for data, labels in test_loader:
9             data, labels = data.to(device), labels.to(device)
10            output = model(data)
11            predictions = torch.argmax(output, dim=1)
12
13            all_preds.extend(predictions.cpu().numpy())
14            all_labels.extend(labels.cpu().numpy())
15
16    return np.array(all_labels), np.array(all_preds)

```

Questa funzione restituisce due liste dove, per un specifico indice i , la prima lista contiene il valore reale, mentre la seconda lista contiene il valore predetto dalla rete.

Infine, nella funzione *main*, richiamiamo le funzioni e realizziamo i grafici.

```

1 def main():
2     ...
3     true_labels, predicted_labels = test(device, model, test_loader)
4     confusion_matrix(true_labels, predicted_labels)
5
6     train_losses, val_losses, val_accuracies = train(device, model,
7             train_loader, val_loader, epochs=30)
8     plot(train_losses, val_losses, val_accuracies)
9
10    true_labels, predicted_labels = test(device, model, test_loader)
11    confusion_matrix(true_labels, predicted_labels)
12
13    display_random_test_sample(device, model, test_dataset)

```

7.1.3 considerazioni sui risultati ottenuti

Partendo da una situazione iniziale disastrosa, come possiamo intuire osservando l'immagine (7.3), in cui la rete classificava quasi ogni immagine come 8, in sole 30 epoch le rete neurale è riuscita ad "imparare" a classificare correttamente buona parte degli esempi di test.

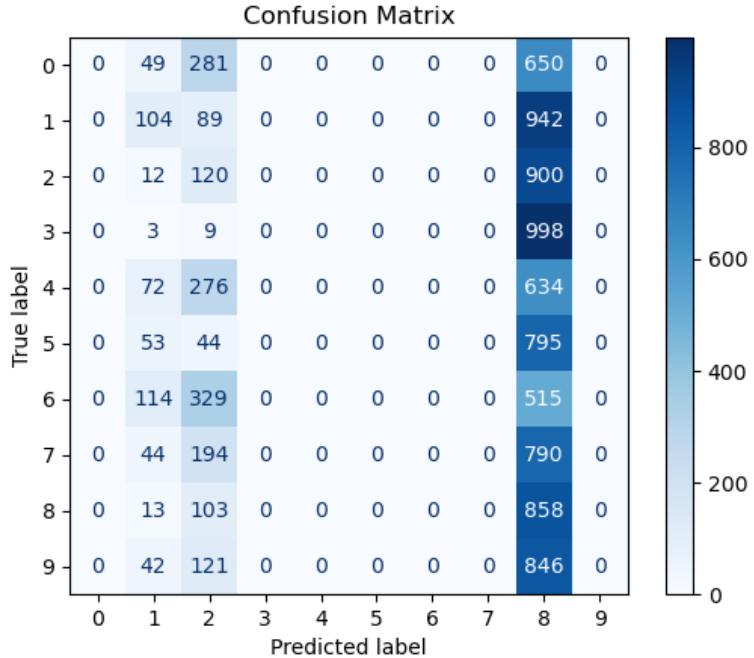


Figura 7.3: Matrice di confusione del del modello sul dataset di valutazione prima del training.

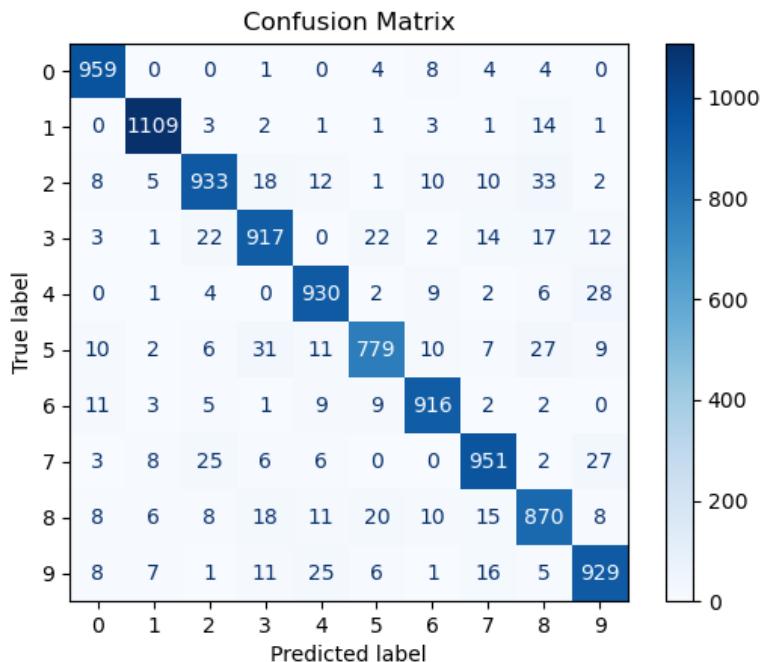


Figura 7.4: Matrice di confusione delle predizioni del modello sul dataset di valutazione dopo il training.

Dall'immagine (7.5), possiamo osservare come sia migliorata la precisione del modello nella predizione sul dataset di validazione nel corso delle epoch, arrivando ad ottenere una precisione del 92.30%.

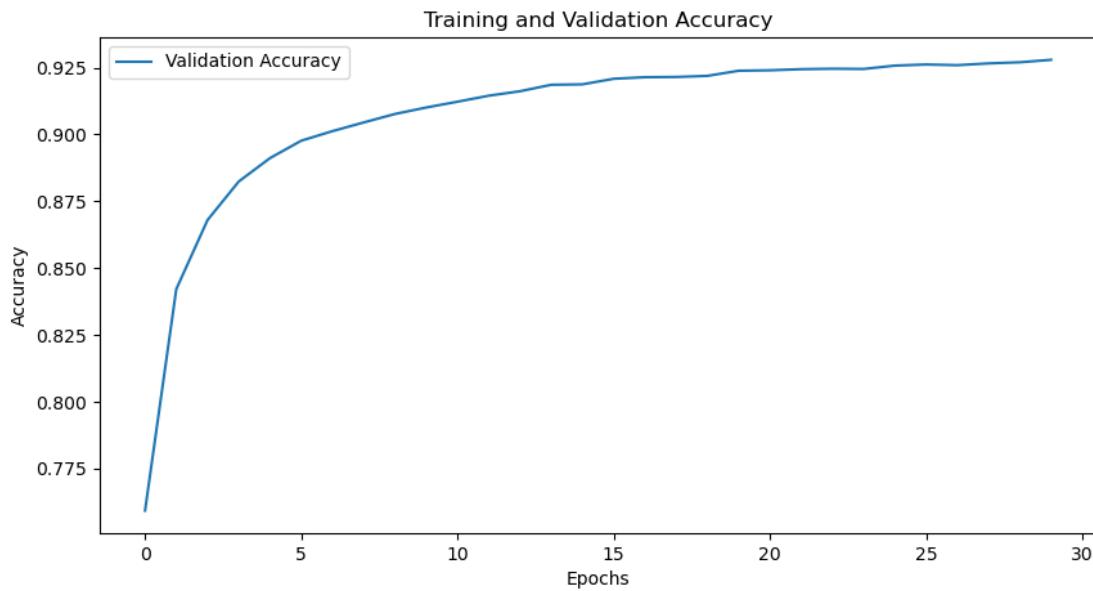


Figura 7.5: Andamento dell'accuratezza durante il training.

Mentre dall'immagine (7.6), possiamo osservare l'andamento della loss durante la fase di training e validation.

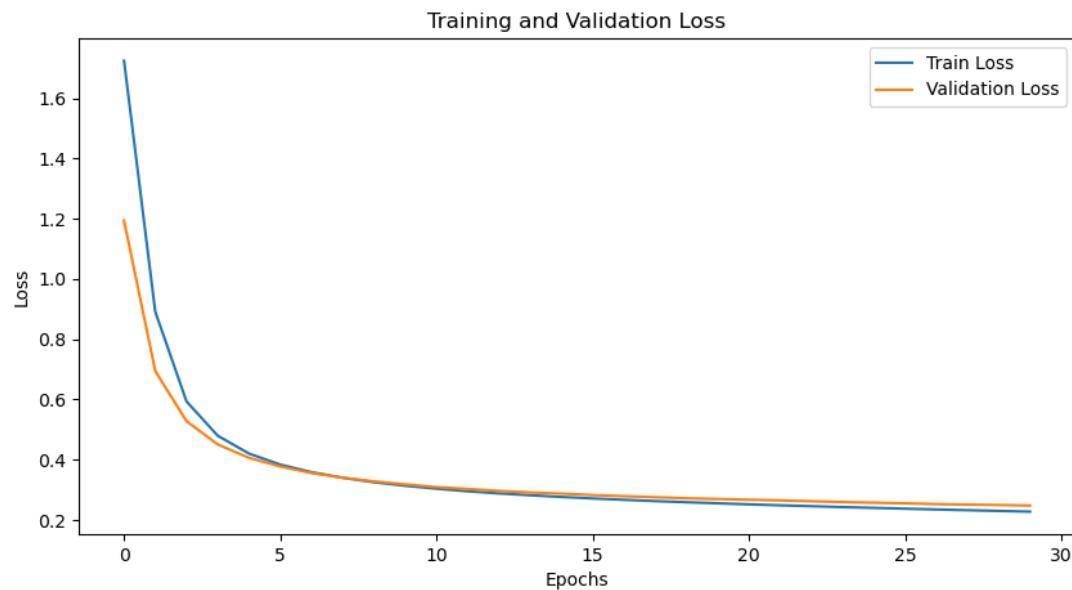


Figura 7.6: Andamento delle loss

7.1.4 conclusioni

Questo piccolo esperimento ci ha permesso di comprendere come realizzare una semplice rete neurale e addestrarla. Se volessimo selezionare dei campioni casuali dal dataset di test e osservare come si comporta la rete, possiamo definire la seguente funzione:

```
1 def display_random_test_sample(device, model: nn.Module,
2                                test_dataset):
3
4     model.eval()
5     indices = np.random.choice(len(test_dataset), size=64,
6                                 replace=False)
7     images, labels = zip(*[test_dataset[idx] for idx in indices])
8
9     images = torch.stack(images).to(device)
10    with torch.no_grad():
11        probabilities = model.predict(images)
12        predicted_classes = torch.argmax(probabilities, dim=1)
13
14    fig, axes = plt.subplots(8, 8, figsize=(12, 12))
15    for i, ax in enumerate(axes.flat):
16        image = images[i].cpu().squeeze().numpy() * 255
17        image = image.astype(np.uint8)
18
19        ax.imshow(image, cmap='gray')
20        true_label = labels[i]
21        pred_label = predicted_classes[i].item()
22        pred_prob = probabilities[i, pred_label].item()
23
24        text1 = f"Predicted: {pred_label}"
25        text2 = f"probability: {pred_prob:.3f}"
26
27        if true_label == pred_label:
28            ax.set_title(f"{text1}\n{text2}", fontsize=10,
29                         color="green")
30        else:
31            ax.set_title(f"{text1}\n{text2}", fontsize=10,
32                         color="red")
33        ax.axis('off')
34
35    ...
```

Questa funzione seleziona 64 campioni casuali e, tramite la rete neurale, svolge delle predizioni per le immagini selezionate.

La figura 7.7 mostra il risultato ottenuto, riportando sopra ad ogni numero, la classe predetta con la relativa probabilità.

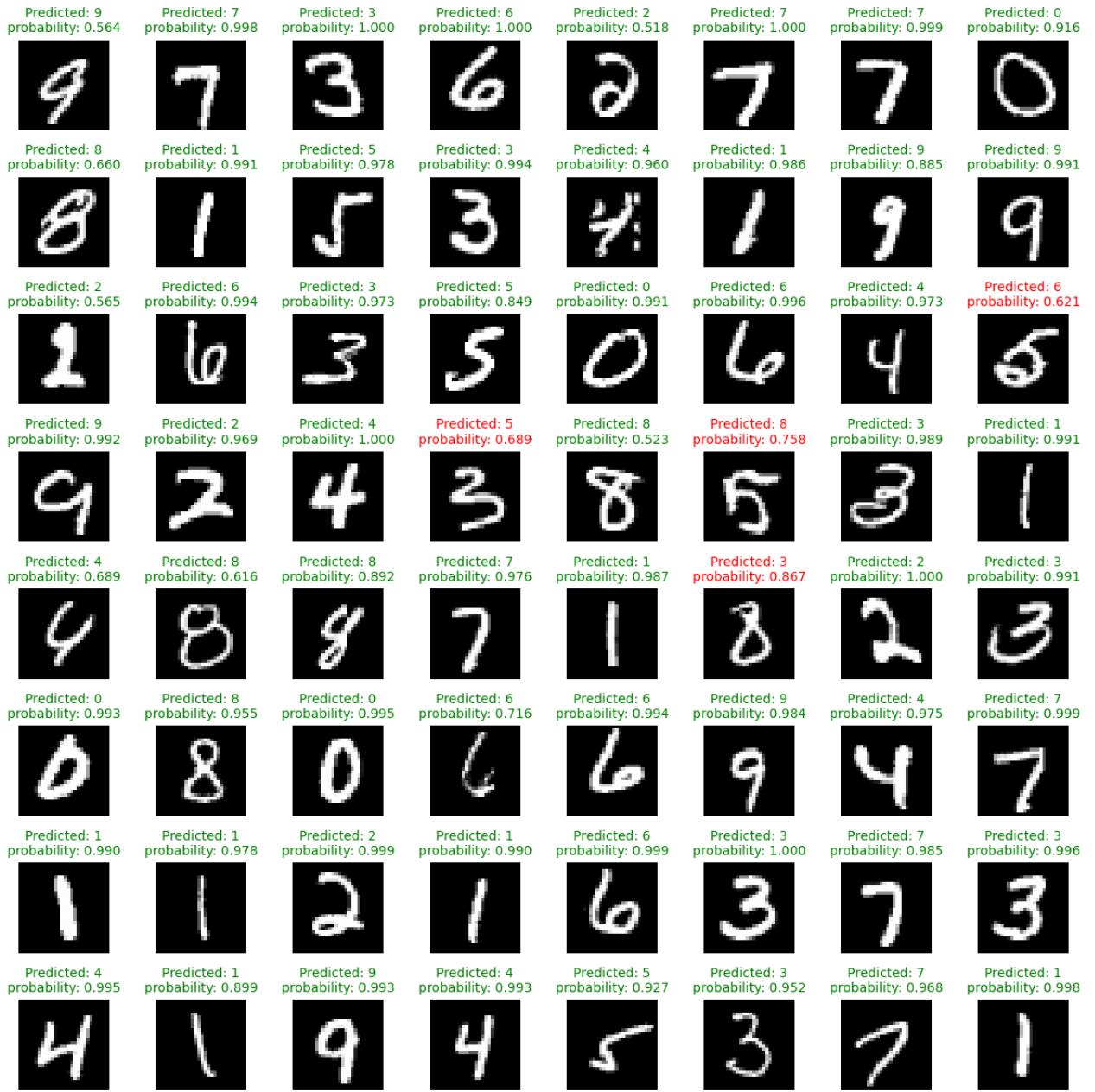


Figura 7.7: Esempio di alcuni campioni

Come si può intuire osservando la figura (7.7), uno dei principali problemi nell'utilizzare queste tipologie per applicazioni sulle immagini, è che non risultano essere invarianti rispetto a traslazioni e distorsioni dell'immagine.

Ovvero che sono sensibili alle minime variazioni dell'immagine. Pertanto, non sono particolarmente adatte per applicazioni in cui le immagini possono variare sotto molti aspetti.

7.1.5 Applicazione di una rete convoluzione

Per capire come cambiano i risultati quando si utilizzano le reti convolutive con le immagini, riapplichiamo sullo stesso problema di classificazione una rete convoluzionale basata sull'architettura di LaNet5. LeNet5 è stata una delle prime architetture di reti convolutive. LeNet5 è stata introdotta da Yann LeCun nel 1998 e aveva come applicazione principale quella di riconoscere i caratteri scritti a mano [69].

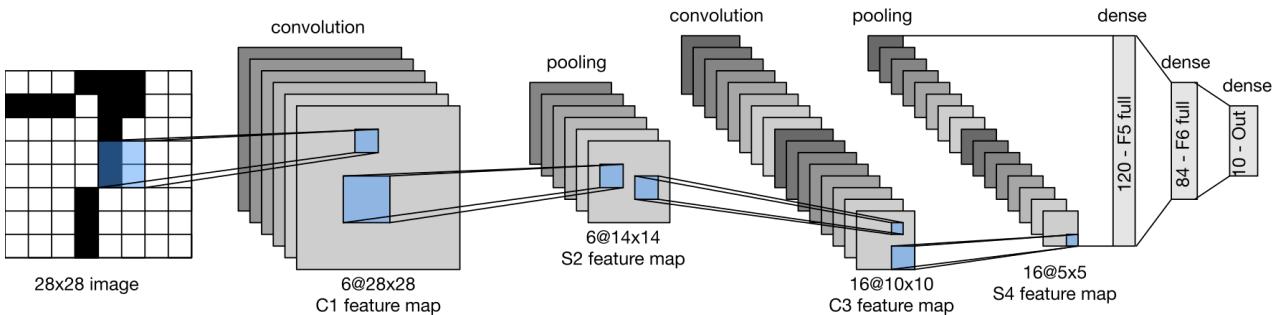


Figura 7.8: Classica architettura di LeNet5 [70]

Come mostrato nella figura 7.8, la struttura della rete è composta da due layer convoluzionali, due layer di sottocampionamento posti dopo i layer convoluzionali e due layer fully connected seguiti da un layer di output con funzione di attivazione softmax.

In Pytorch può essere applicata nel seguente modo:

```

1  class Network(nn.Module):
2      def __init__(self, input_channels: int, num_classes: int):
3          super(Network, self).__init__()
4
5          #Rete convoluzionale
6          self.conv_net = nn.Sequential(
7              nn.Conv2d(in_channels=input_channels, out_channels=6,
8                  kernel_size=5, stride=1, padding=2),
9              nn.Tanh(),
10             nn.AvgPool2d(kernel_size=2, stride=2),
11             nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5,
12                     stride=1),                                     # Secondo strato
13                     nn.Tanh(),
14                     nn.AvgPool2d(kernel_size=2, stride=2),
15             )
16
17          # Strati completamente connessi
18          self.fc_net = nn.Sequential(
19              nn.Flatten(),
20              nn.Linear(in_features=16 * 5 * 5, out_features=120),
21              nn.Tanh(),
22              nn.Linear(in_features=120, out_features=84),
23              nn.Tanh(),
24              nn.Linear(in_features=84, out_features=num_classes)
25          )
26
27      def forward(self, x: torch.tensor) -> torch.tensor:
28          x = self.conv_net(x)
29          x = self.fc_net(x)
30          return x
31
32      def predict(self, x: torch.tensor) -> torch.tensor:
33          with torch.no_grad():
34              return F.softmax(self.forward(x), dim=1)

```

Rieseguendo l'esperimento utilizzando questa rete e mantenendo tutto il resto inalterato, otteniamo il seguente risultato:

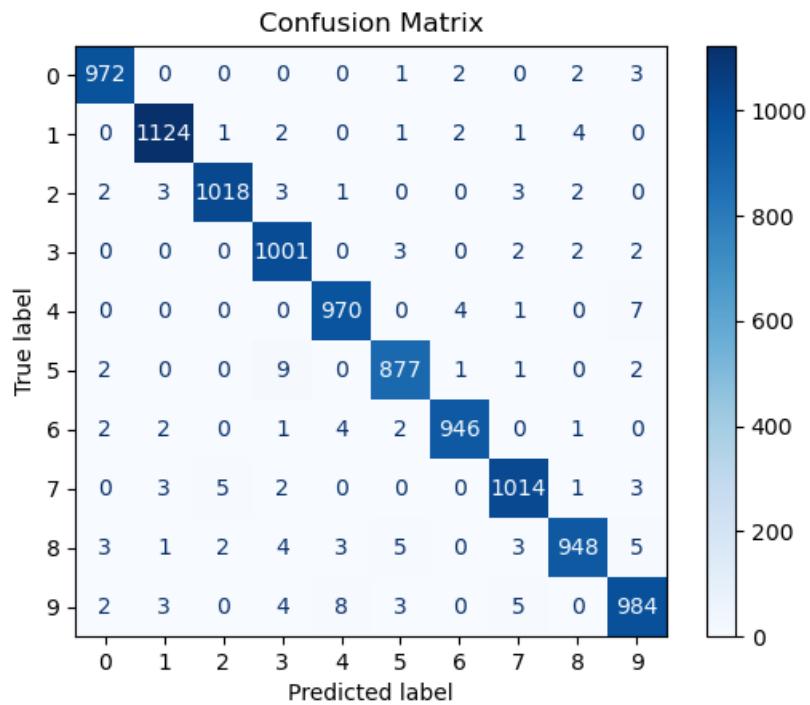


Figura 7.9: Matrice di confusione di LaNet5 sul dataset MNIST.

Utilizzando una rete convoluzionale siamo riusciti a raggiungere una precisione del 98.5% (7.10), rispetto al 92.30% ottenuto utilizzando un MLP.

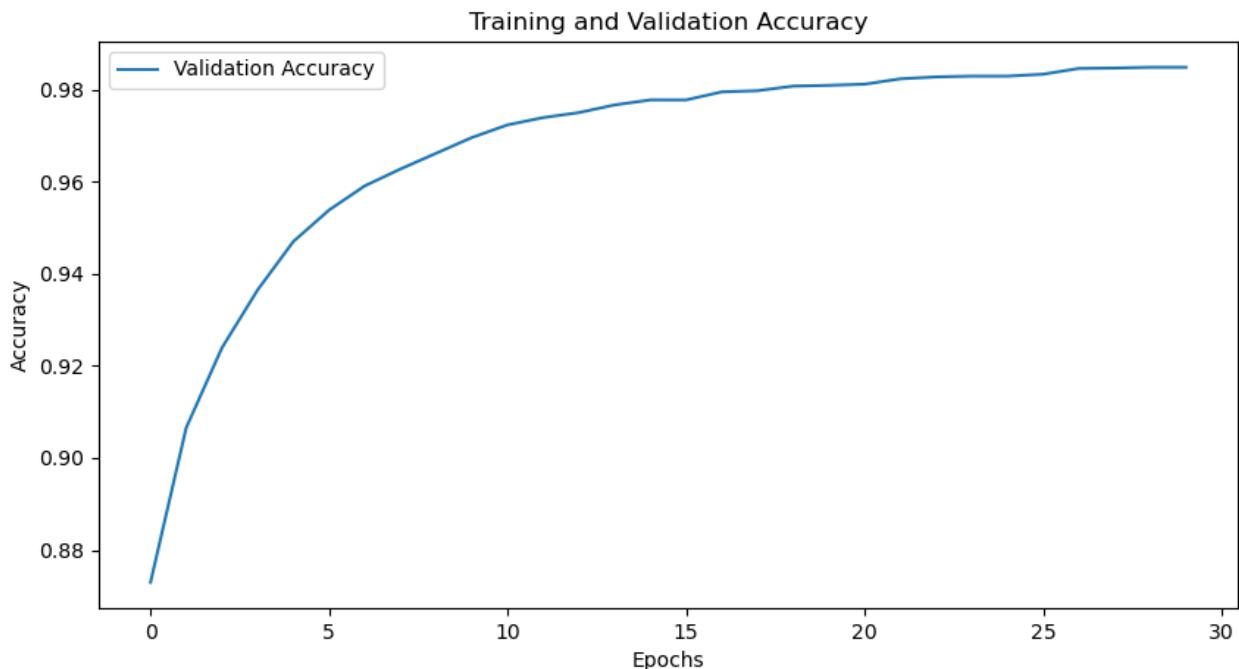


Figura 7.10: Andamento dell'accuratezza.

Osservando alcuni campioni casuali dal dataset di valutazione (7.11), possiamo osservare come le probabilità delle predizioni di ogni numero siano molto vicine al valore 1 per buona parte dei campioni.

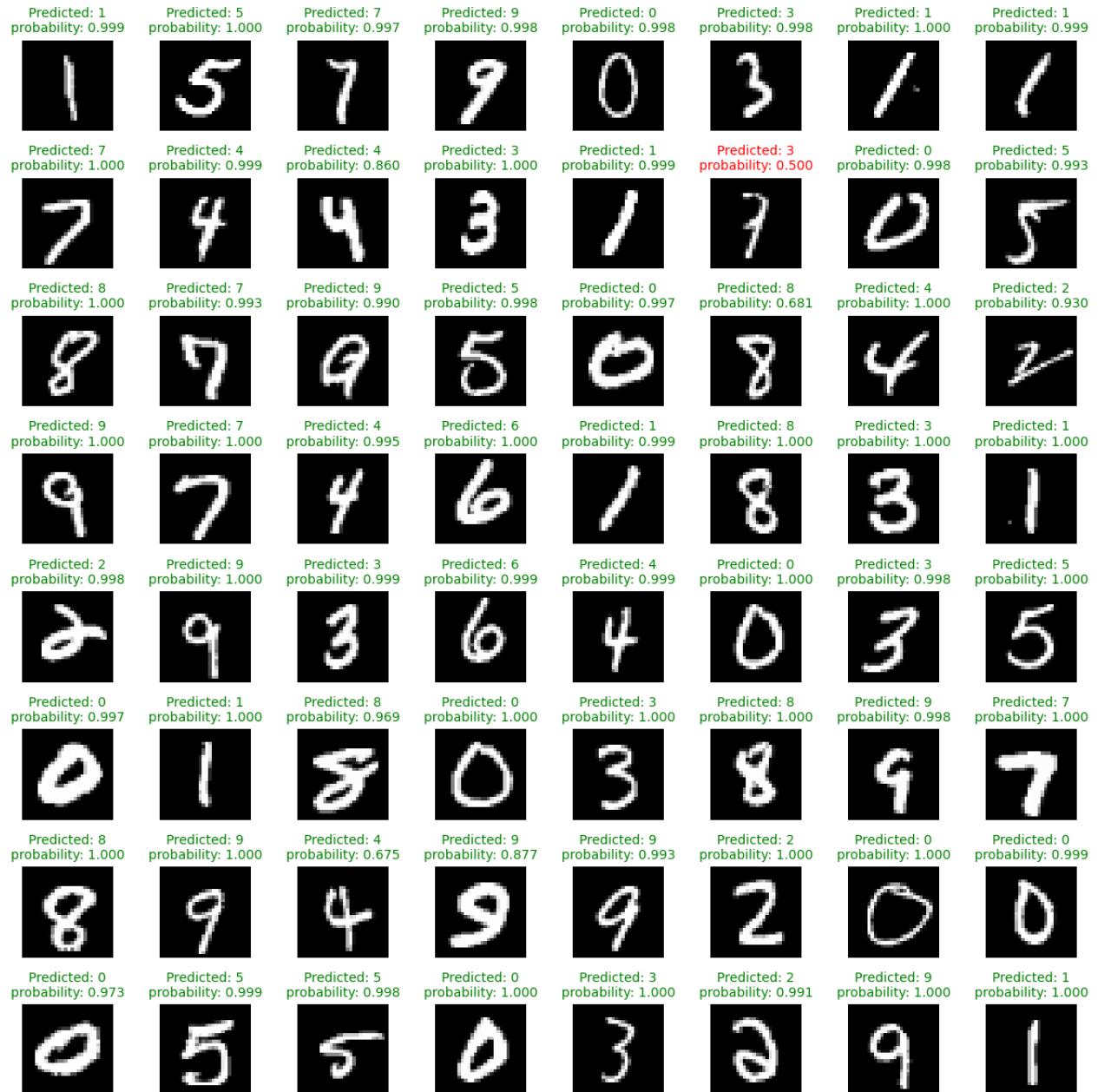


Figura 7.11: Esempio di alcuni campioni

Questo è il motivo per cui si utilizzano le reti convolutive in applicazioni con le immagini. In quanto si prestano meglio a riconoscere *pattern* nei dati.

7.2 Riconoscimento dei campi agricoli

7.2.1 Introduzione al problema

In questa sezione tratteremo l'applicazione delle reti neurali al telerilevamento. Utilizzeremo il dataset Sentinel2-Munich480 [19] per addestrare una rete neurale a svolgere un problema di *crop mapping* (mappatura delle colture), utilizzando immagini multispettrali satellitari ottenute dai satelliti Sentinel2. L'obiettivo di questa sperimentazione consisterebbe nel riconoscere i campi agricoli ed identificare il contenuto.

7.2.2 Il dataset Sentinel2-Munich480

Il dataset Sentinel2-Munich480 è una raccolta di immagini satellitari Sentinel-2 su un'area di $102 \text{ km} \times 42 \text{ km}$ a nord di Monaco di Baviera, Germania [19, 20]. L'area di interesse è stata ulteriormente suddivisa in blocchi quadrangolari

di $3.840 \text{ km} \times 3.840 \text{ km}$ (multipli di 240 m e 480 m). Ciascuno di questi blocchi dista una dall'altro 480 m . Ed è diviso in tile da 240 m . Per la suddivisione del dataset, questi blocchi sono stati assegnati in modo casuale alle partizioni per l'addestramento, la validazione e la valutazione del modello. Il dataset Sentinel2-Munich480 è stato pensato per svolgere ricerche nel campo della mappatura delle culture, con lo scopo specifico di estrarre informazioni su "dove e quando" vengono coltivate le colture. Il dataset raccoglie le acquisizioni satellitari svolte dal 2016 fino al 2017, organizzate in sequenze temporali di circa 32 elementi. Ciascuna immagine del dataset è composta da 13 canali, ognuna con una risoluzione di 48×48 pixel. Ogni canale rappresenta una diversa banda catturata da sentinel-2 (3.15). Queste bande hanno diverse risoluzioni spaziali: 10 m , 20 m e da 60 m .

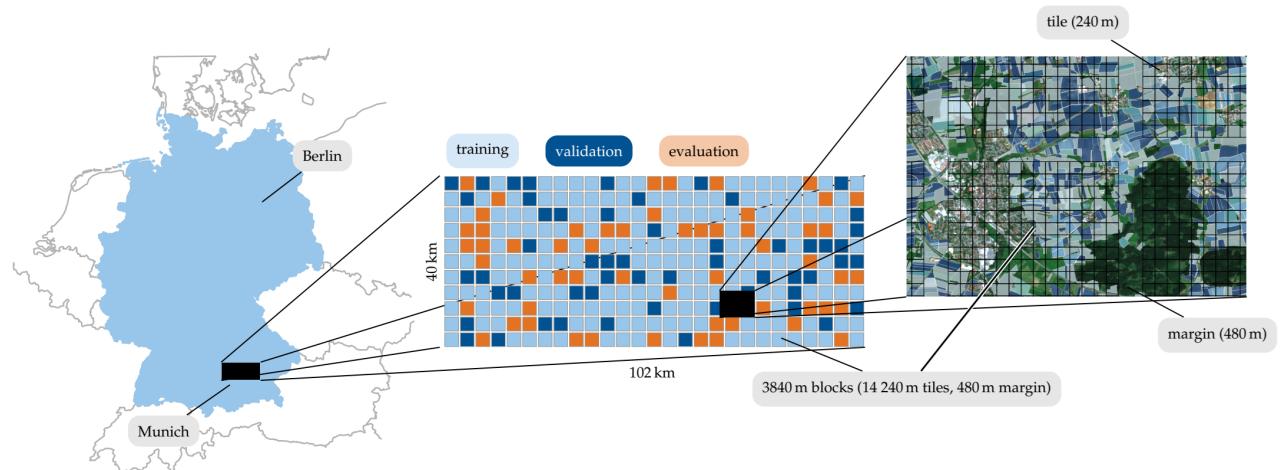


Figura 7.12: Rappresentazione dell'area di interesse.

7.2.3 Lettura dei dati dal dataset

I dati all'interno del dataset sono organizzati nel seguente modo:

```
Munich480/
    data16/                      // immagini dell'anno 2016
        1/                          // ID
            20160103_10m.tif      // acquisizione del 2016-01-03 con tutte le bande da 10m
            20160103_20m.tif      // acquisizione del 2016-01-03 con tutte le bande da 20m
            20160103_60m.tif      // acquisizione del 2016-01-03 con tutte le bande da 60m
            ...
            20161118_10m.tif      // acquisizione del 2016-11-18 con tutte le bande da 10m
            20161118_20m.tif      // acquisizione del 2016-11-18 con tutte le bande da 20m
            20161118_60m.tif      // acquisizione del 2016-11-18 con tutte le bande da 60m
            y.tif                  // mappa di verità della sequenza
        ...
    14300/
        20160103_10m.tif
        20160103_20m.tif
        20160103_60m.tif
        ...
        20161118_10m.tif
        20161118_20m.tif
        20161118_60m.tif
        y.tif
    data17/                      // immagini dell'anno 2017
        1/                          // ID
            20160105_10m.tif      // acquisizione del 2016-01-05 con tutte le bande da 10m
            20160105_20m.tif      // acquisizione del 2016-01-05 con tutte le bande da 20m
            20160105_60m.tif      // acquisizione del 2016-01-05 con tutte le bande da 60m
        ...
    ...
classes.txt                  // File con le informazioni sulle classi
tileids/                     // Informazioni sui tile
    eval.tileids             // File con gli ID da utilizzare per la validazione
    test_fold0.tileids       // File con gli ID da utilizzare per la valutazione
    train_fold0.tileids     // File con gli ID da utilizzare per l'addestramento
```

Il file `classes.txt` contiene le informazioni delle classi che sono presenti nel dataset.

0 unknown	//sconosciuto
1 sugar beet	//barbabietola da zucchero
2 summer oat	//avena estiva
3 meadow	//prato
5 rape	//colza
8 hop	//luppolo
9 winter spelt	//farro invernale
12 winter triticale	//triticale invernale
13 beans	//fagioli
15 peas	//piselli
16 potatoe	//patate
17 soybeans	//soia

```

19|asparagus      //asparagi
22|winter wheat   //grano invernale
23|winter barley   //orzo invernale
24|winter rye     //segale invernale
25|summer barley   //orzo estivo
26|maize          //mais

```

Mentre i file `.tileids` contengono gli ID dei tile che devono essere utilizzati per l'addestramento, per la validazione oppure per la valutazione. Ad esempio il file `eval.tileids` contiene:

```

169
170
171
172
173
174
256
...

```

Per poter ottenere i dati dal dataset e utilizzarli per l'addestramento della rete neurale, dobbiamo definire una classe che estenda la classe `Dataset` di PyTorch e ridefinire i metodi speciali: `_len_` e `_getitem_`. In questo modo, tale classe potrà essere utilizzata dalla classe `DataLoader` di PyTorch per prelevare i dati. Iniziamo definendo il costruttore della classe e le variabili:

```

1 from torch.utils.data import Dataset
2 ...
3
4 class Munich480(Dataset):
5
6     _EVAL_TILEIDS = os.path.join("tileids", "eval.tileids")
7     _TRAIN_TILEIDS = os.path.join("tileids", "train_fold0.tileids")
8     _TEST_TILEIDS = os.path.join("tileids", "test_fold0.tileids")
9     _FOLDER_2016: str = "data16"
10    _FOLDER_2017: str = "data17"
11
12    TemporalSize: Final[int] = 32
13    ImageChannelsCount: Final[int] = 13
14    ImageWidth: Final[int] = 48
15    ImageHeight: Final[int] = 48
16    ...
17
18    class Year(Flag):
19        Y2016 = auto()
20        Y2017 = auto()
21
22    class DatasetMode(Enum):
23        TRAINING = auto()
24        TEST = auto()
25        VALIDATION = auto()
26
27    def __init__(self, DatasetPath:str | None, mode: DatasetMode,
28                 year:Year, ...):
29        super().__init__()

```

```

29     ...
30
31     #In base a come voglio utilizzare il dataset, leggo la
32     #sequenza corrispondente.
33
34     match mode:
35         case Munich480.DatasetMode.TRAINING:
36             self._dataSequenze =
37                 np.loadtxt(os.path.join(self.DatasetPath,
38                                         Munich480._TRAIN_TILEIDS), dtype=int)
39
40         case Munich480.DatasetMode.TEST:
41             self._dataSequenze =
42                 np.loadtxt(os.path.join(self.DatasetPath,
43                                         Munich480._TEST_TILEIDS), dtype=int)
44
45         case Munich480.DatasetMode.VALIDATION:
46             self._dataSequenze =
47                 np.loadtxt(os.path.join(self.DatasetPath,
48                                         Munich480._EVAL_TILEIDS), dtype=int)
48
49         case _:
50             raise Exception(f"Invalid mode {mode}")
51
52
53     temp = {
54         Munich480.Year.Y2016 : Munich480._FOLDER_2016,
55         Munich480.Year.Y2017 : Munich480._FOLDER_2017
56     }
57
58     #dizionario per mappare le sequenze di ogni anno
59     years_sequenze: Dict[str, Any] = dict()
60     range = 0
61
62     #Verifichiamo l'integrita' di ogni sequenza e la presenza
63     #di tutti i file
64     for available_year in Munich480.Year:
65         if available_year in year:
66             available_folder =
67                 os.listdir(os.path.join(self.DatasetPath,
68                                         temp[available_year]))
69             temp_dict: Dict[str, bool] =
70                 dict.fromkeys(available_folder, True)
71             tempList: List[str] = list()
72
73             for idx in self._dataSequenze:
74                 if temp_dict.get(str(idx)) is not None and
75                     os.path.exists(os.path.join(self.DatasetPath,
76                                         temp[available_year], str(idx), 'y.tif')):
77                     tempList.append(os.path.join(self.DatasetPath,
78                                         temp[available_year], str(idx)))
79
80             npList = np.array(tempList, dtype=object)

```

```

70         years_sequenze[temp[available_year]] = {
71             "sequenze" : npList,
72             "range" : (range, range + len(tempList))
73         }
74
75         range += len(tempList)
76
77     self._yearsSequenze = years_sequenze
78 ...

```

Definiamo una funzione che mappa un indice (ovvero l' i -esimo elemento del dataset) al rispettivo anno ed ID, restituendo il percorso della cartella corrispondente.

```

1 def mapIndex(self, index: int) -> str:
2     for year in self._yearsSequenze.keys():
3         year_range: Tuple[int, int] =
4             self._yearsSequenze[year]["range"]
5
6         if index >= year_range[0] and index < year_range[1]:
7             idx = index - year_range[0]
8             data_folder_path =
9                 self._yearsSequenze[year]["sequenze"][idx]
10
11     return data_folder_path#os.path.join(self._folderPath,
12                                         year, str(data_folder_index))
13
14     raise Exception(f"Index {index} not found in any year range")

```

Implementiamo una funzione per leggere un file .tif. Per leggere questi file utilizziamo la libreria *rasterio*, la quale fornisce già dei metodi per la lettura di questi file.

```

1 import rasterio
2 ...
3
4 def load_dit_file(self, filePath:str, normalize: bool = True) ->
5     Dict[str,any]:
6     with rasterio.open(filePath) as src:
7         data = src.read().astype(np.float32)
8         profile = src.profile
9
10    if normalize:
11        data = data * 1e-4
12    return {"data" : data, "profile" : profile}

```

L'oggetto *"profile"* rappresenta tutte le informazioni sull'immagine: numero di canali, posizione geografica dell'immagine, ecc. Mentre *"data"* è l'immagine, rappresentata come un array numpy. Successivamente, definiamo una funzione che restituisce i percorsi di tutti i file della sequenza che devono essere caricati.

```

1 def get_dates(self, path: str, sample_number= int | None) ->
2     list[str]:
3     files = os.listdir(path)
4     dates = list()
5
6     for f in files:
7         date = f.split("_") [0]

```

```

7     if len(date) == 8: # 20160101
8         dates.append(date)
9
10    dates = list(set(dates))
11
12    if sample_number is None or sample_number < 0:
13        return dates
14
15    if len(dates) > sample_number:
16        dates = random.sample(dates, sample_number)
17    elif len(dates) < sample_number:
18        while len(dates) < sample_number:
19            dates.append(random.choice(dates))
20
21    #restituisce le date disposte in ordine crescente
22    return dates.sort()

```

Definiamo la funzione che carica tutta la sequenza temporale di un *Tile*.

```

1 def load_year_sequenze(self, idx: str) -> dict[str, any]:
2     sequenzeFolder = self.mapIndex(idx)
3     dates = self.get_dates(path=sequenzeFolder,
4                             sample_number=Munich480.TemporalSize)
5     profile = None
6
7     x: torch.Tensor = torch.empty((Munich480.TemporalSize,
8                                    Munich480.ImageChannelsCount, Munich480.ImageHeight,
9                                    Munich480.ImageWidth), dtype=torch.float32)
10
11    distance_map = {
12        Munich480.Distance.m10: "_10m.tif",
13        Munich480.Distance.m20: "_20m.tif",
14        Munich480.Distance.m60: "_60m.tif"
15    }
16
17    # Itera attraverso ogni data per caricare i dati temporali
18    for t, date in enumerate(dates):
19        current_channel_index = 0
20
21        for distance, suffix in distance_map.items():
22            DataDict =
23                self.load_tif_file(os.path.join(sequenzeFolder,
24                                              f"{date}{suffix}"))
25            data = DataDict['data']
26
27            if profile is None:
28                profile = DataDict["profile"]
29
30            tensor = torch.from_numpy(data).unsqueeze(0)
31
32            if distance != Munich480.Distance.m10:
33                tensor = F.interpolate(tensor,
34                                      size=(Munich480.ImageHeight, Munich480.ImageWidth))

```

```

30     num_channels = tensor.size(1)
31
32     x[t, current_channel_index:current_channel_index +
33         num_channels, :, :] = tensor.squeeze(0)
34     current_channel_index += num_channels
35
36     y = self.load_tif_file(filePath=os.path.join(sequenzeFolder,
37         "y.tif"), normalize = False)["data"]
38
39     #(1, 48, 48) -> (48, 48)
40     y = np.squeeze(y, axis=0)
41     y = torch.from_numpy(y)
42
43     # permute channels with time_series (t x c x h x w) -> (c x t x h
44     # x w)
45     x = x.permute(1, 0, 2, 3)
46
47     return {"x": x, "y": y, "profile": profile}

```

Infine, definiamo la funzione per ottenere l' i -esimo elemento del dataset e le funzioni `__len__` e `__getitem__`.

```

1 def getItem(self, idx: int) -> dict[str, any]:
2     assert idx >= 0 and idx < self.__len__(), f"Index {idx} out of
3         range"
4     return self.load_year_sequenze(idx)
5
6 def __len__(self) -> int:
7     if self.DatasetSize is None:
8         self.DatasetSize = self.getSize()
9     return self.DatasetSize
10
11 def getSize(self) -> int:
12     maxValue = 0
13     for year in self._yearsSequenze.keys():
14         maxValue = max(maxValue,
15             (self._yearsSequenze[year]["range"][1]))
16     return maxValue
17
18 def __getitem__(self, idx: int) -> any:
19     itemDict = self.getItem(idx)
20     itemDict = self.apply_transforms(itemDict)
21     return itemDict['x'], itemDict['y']

```

La classe così creata verrà instanziata per i vari casi nel seguente modo:

```

1 TRAIN_DATASET = Munich480(datasetFolder = datasetFolder, mode=
2     Munich480.DatasetMode.TRAINING, year= Munich480.Year.Y2016 |
3     Munich480.Year.Y2017,...)
4
5 VAL_DATASET = Munich480(datasetFolder = datasetFolder, mode=
6     Munich480.DatasetMode.VALIDATION, year= Munich480.Year.Y2016 |
7     Munich480.Year.Y2017,...)
8
9

```

```

5 TEST_DATASET = Munich480(datasetFolder = datasetFolder, mode=
    Munich480.DatasetMode.TEST, year= Munich480.Year.Y2016 |
    Munich480.Year.Y2017,...)

```

Per quanto riguarda le *transforms*, faremo uso delle seguenti:

```

1 transforms = transforms.Compose([
2     transforms.RandomHorizontalFlip(p=0.5),
3     transforms.RandomVerticalFlip(p=0.5)
4 ])

```

7.2.4 Applicazione dell'UNet

Una possibile architettura applicabile a questo problema è la U-Net, trattata nella sezione 6.2. Tuttavia, questa rete è stata pensata per operare su immagini 2D, mentre nel nostro caso utilizziamo una sequenza temporale composta da 32 immagini che devono essere processate simultaneamente. In altre parole, abbiamo un tensore con shape [Batch_Size, TimeSeq, Channels, Width, Height], ma la rete può solo operare con tensori aventi shape [Batch_Size, Channels, Width, Height]. Un possibile modo per ovviare a questa limitazione consisterebbe nel rappresentare tutta la sequenza come un'unica immagine composta da 32x13 canali. Così facendo, otterremmo un tensore con shape [Batch_Size, 416, 48, 48], compatibile con l'input richiesto dalla rete.

Per far svolgere questa operazione, basterebbe semplicemente modificare la funzione `__getitem__` nel seguente modo:

```

1 def __getitem__(self, idx: int) -> any:
2     itemDict = self.getItem(idx)
3     itemDict = self.apply_transforms(itemDict)
4     x = itemDict['x']
5     x = x.view(-1, Munich480.ImageHeight, Munich480.ImageWidth)
6     return x, itemDict['y']

```

Come avevamo visto dallo schema di architettura della U-Net (6.3), la rete è composta da diversi blocchi che eseguono una serie di convoluzioni in successione. Per semplicità, definiamo una classe che modelli questa successione di convoluzioni.

```

1 class Multiple_Conv2D_Block(nn.Module):
2     def __init__(self, num_convs: int, in_channels: int,
3                  out_channels: int, kernel_size: int, stride: int, padding:
3                  int, bias: bool):
4         super().__init__()
5         ...
6         self.blockComponents = nn.Sequential()
7         for _ in range(num_convs):
8             self.blockComponents.append(
9                 nn.Conv2d(
10                     in_channels=in_channels,
11                     out_channels=out_channels,
12                     kernel_size=kernel_size,
13                     stride=stride,
14                     padding=padding,
15                     bias=bias
16                 )
17             in_channels = out_channels

```

```

18         self.blockComponents.append(nn.BatchNorm2d(out_channels))
19         self.blockComponents.append(nn.ReLU(inplace=False))
20
21     def forward(self, x):
22         return self.blockComponents(x)

```

Per quanto riguarda il modello, lo implementiamo nel seguente modo:

```

1 class UNET(nn.Module):
2     _FEATURES: Final[Tuple[int, int, int, int]] = (64, 128, 256, 512)
3
4     def __init__(self, in_Channels, out_Channels, ...):
5         ...
6         self._EncoderBlocks = nn.ModuleList()
7         self._Bottleneck = nn.ModuleList()
8         self._DecoderBlocks = nn.ModuleList()
9         self._OutputLayer = nn.Sequential()
10        self._DownSampler = nn.MaxPool2d(kernel_size=2, stride=2)
11        in_feat = self._in_Channel
12
13        for feature in self._features:
14            self._EncoderBlocks.append(
15                Multiple_Conv2D_Block(
16                    num_convs=2,
17                    in_channels=in_feat,
18                    out_channels=feature,
19                    kernel_size=(3,3),
20                    stride=(1,1),
21                    padding=(1,1),
22                    bias=False
23                )
24            )
25            in_feat = feature
26
27        self._Bottleneck.append(
28            Multiple_Conv2D_Block(
29                num_convs=2,
30                in_channels=self._features[-1],
31                out_channels=self._features[-1]*2,
32                kernel_size=3,
33                stride=1,
34                padding=1,
35                bias=True
36            )
37        )
38        for feature in reversed(self._features):
39            self._DecoderBlocks.append(
40                nn.ConvTranspose2d(
41                    in_channels=feature*2,
42                    out_channels=feature,
43                    kernel_size=2,
44                    stride=2
45                )
46        )

```

```

47         self._DecoderBlocks.append(
48             Multiple_Conv2D_Block(
49                 num_convs=2,
50                 in_channels=feature*2,
51                 out_channels=feature,
52                 kernel_size=3,
53                 stride=1,
54                 padding=1,
55                 bias=False
56             )
57         )
58     self._OutputLayer.append(
59         nn.Conv2d(
60             in_channels=self._features[0],
61             out_channels=self._out_channels,
62             kernel_size=1
63         )
64     )
65 def forward(self, x) -> Optional[torch.Tensor]:
66     skip_connections = []
67
68     for encoder_block in self._EncoderBlocks:
69         x = encoder_block(x)
70         skip_connections.append(x)
71         x = self._DownSampler(x)
72
73     x = self._Bottleneck[0](x)
74
75     #revers della lista
76     skip_connections = skip_connections[::-1]
77
78     for i in range(0, len(self._DecoderBlocks), 2):
79
80         #ConvTranspose2d sul risultato precedente
81         x = self._DecoderBlocks[i](x)
82
83         #Ottengo la copia del tensore
84         skip_connection = skip_connections[int(i//2)]
85
86         if x.shape != skip_connection.shape:
87             x = nn.functional.interpolate(x,
88                 size=skip_connection.shape[2:])
89
90         #Concateno i tensori
91         concat_skip = torch.cat((skip_connection, x), dim=1)
92
93         #Esegue le due convoluzioni
94         x = self._DecoderBlocks[i+1](concat_skip)
95         x = self._OutputLayer(x)
96
97     return x

```

Mentre per il resto del codice, utilizziamo le stesse funzioni utilizzate per la precedente sperimentazione (trattata nella sezione 7.1). Eseguendo l'addestramento della rete ed osservando i risultati ottenuti sul dataset di validazione all'epoca 28 (7.13), ci si accorge di un fatto impor-

tante: la rete fatica a classificare correttamente buona parte delle classi, soprattutto le classi 3 e 12.

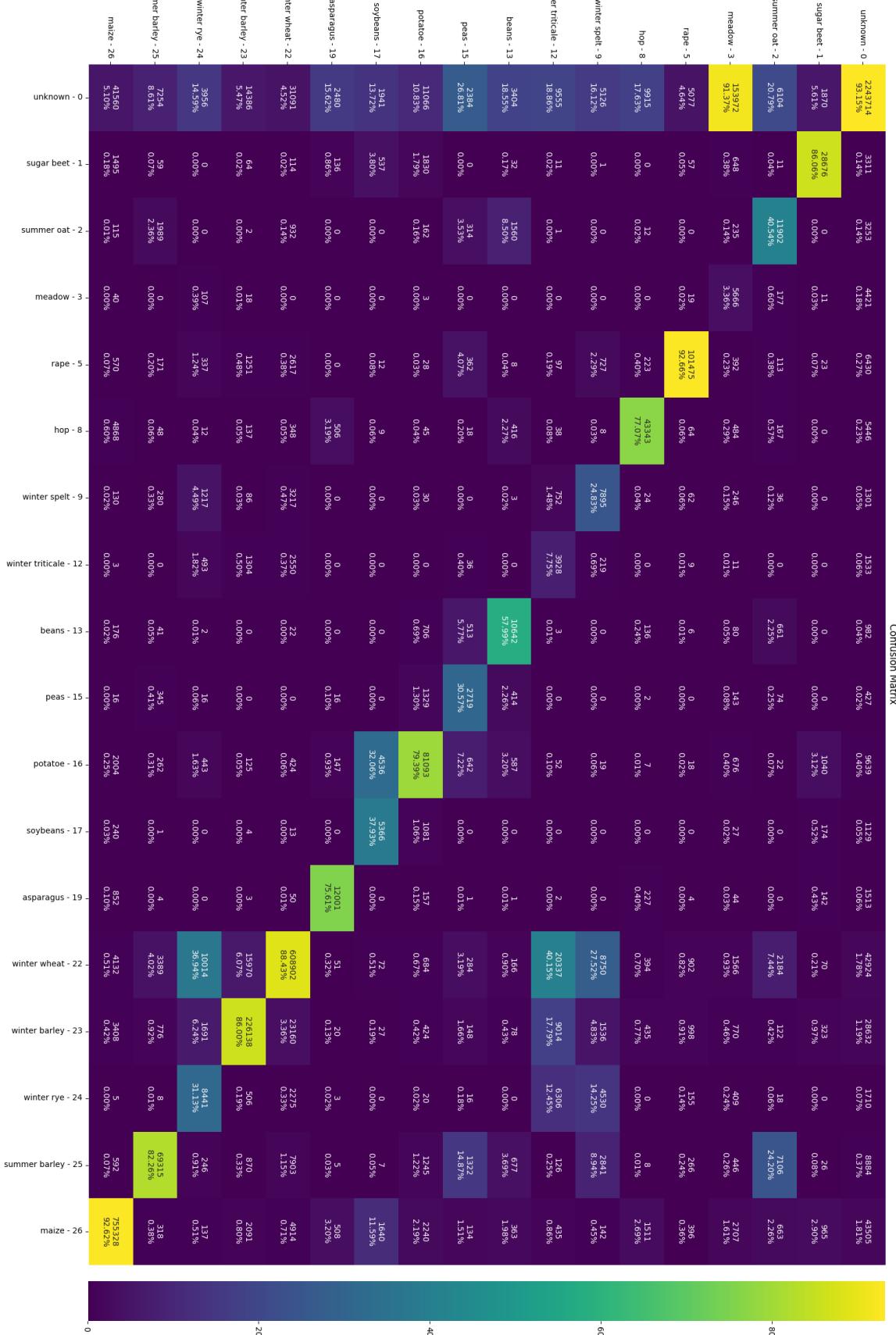


Figura 7.13: Matrice di confusione del modello.

Si potrebbe pensare che basterebbe continuare ad addestrare il modello ancora per qualche epoca. Tuttavia, osservando il grafico (7.14), possiamo notare come già dall'epoca 24 il modello smette di migliorare, mantenendo una precisione poco sotto all'86%. Osservando la figura (7.13), si nota che questo valore elevato sembra dipendere principalmente dalle classi più numerose.

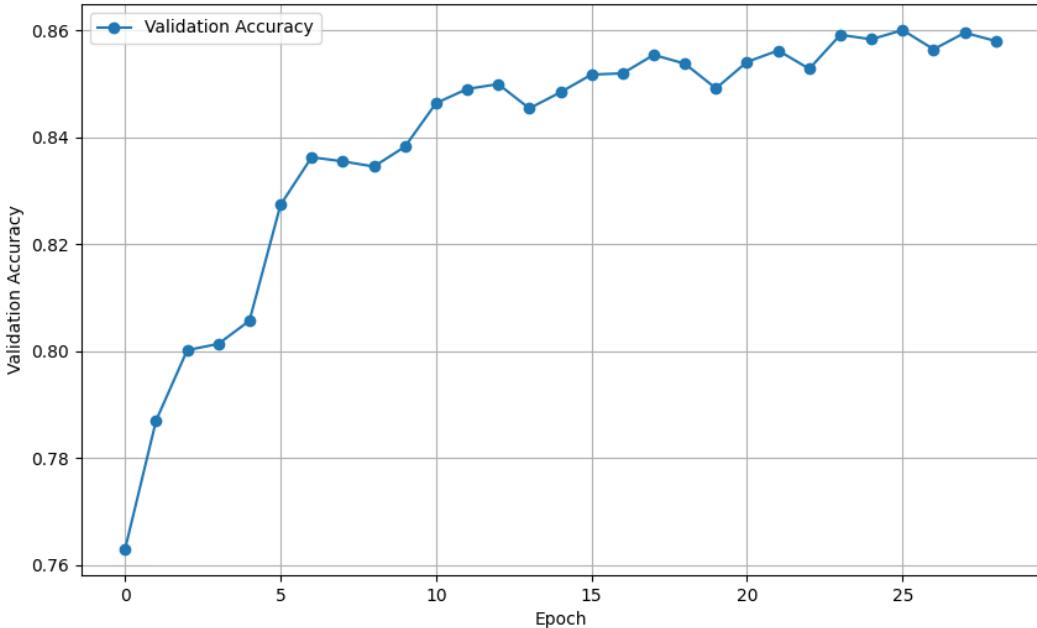


Figura 7.14: Andamento dell'accuratezza.

Analizzando la distribuzione delle classi di verità nelle immagini del dataset di training (7.15), possiamo notare come queste classi non siano equilibrate. Di conseguenza, il modello tenderà a imparare le classi che si presentano più frequentemente, in quanto otterrà una maggiore errore da esse. Al contrario, per le classi meno frequenti, il modello otterrà un errore basso, quasi trascurabile. Questo fatto permetterebbe al modello di ottenere un precisione alta solo grazie alle classi numerose.

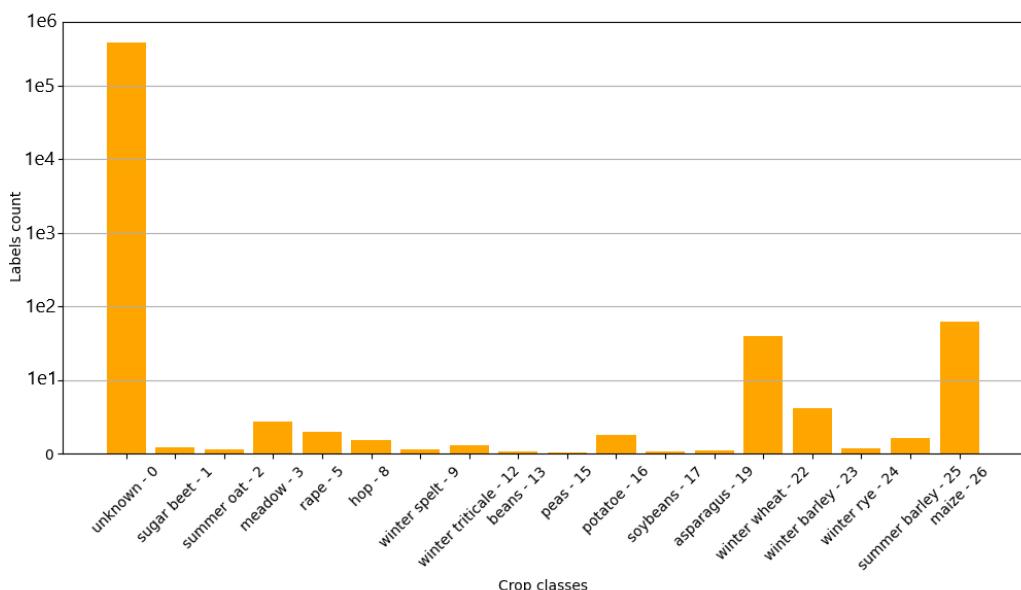


Figura 7.15: Distribuzione delle classi di verità

Un modo per risolvere questo problema consisterebbe nell'agire sulla funzione di loss, assegnando un peso a ciascuna classe. Tale peso dipenderebbe dalla frequenza di ciascuna classe e permetterebbe di penalizzare fortemente il modello quando sbaglia la predizione sulle classi meno frequenti.

Per il calcolo di questi pesi, ci viene incontro la libreria `sklearn`, in quanto offre già delle implementazioni da poter utilizzare per il calcolo di questi pesi.

```
1 from sklearn.utils.class_weight import compute_class_weight
2
3 ...
4
5 labels, available_classes = count_labels(training_dataset)
6
7 class_weights = compute_class_weight(
8     class_weight='balanced',
9     classes=available_classes,
10    y=labels
11 )
12
13 weights = np.zeros(classes_number, dtype=np.float32)
14 weights[available_classes] = class_weights
15 weights_tensor = torch.tensor(weights, dtype=torch.float32)
16
17 print(weights_tensor)
18 ...
```

Una volta eseguiti i calcoli dei pesi, quello che si ottiene è un tensore con i seguenti valori:

```
1 tensor([ 0.1146,  7.2109, 10.3559,  1.4608,  2.1595,
2      3.5548, 10.7607,  5.4641, 20.7526, 25.7078,
3      2.5543, 20.7815, 12.0300,  0.4005,  1.0421,
4      8.3329,  2.9101,  0.3571])
```

Per applicare questi pesi alla funzione di loss basta semplicemente scrivere:

```
1 lossFunction = nn.CrossEntropyLoss(weight=weights_tensor)
```

A questo punto, rieseguiamo il training tenendo conto dei pesi calcolati e vediamo il risultato che otteniamo.

Dopo aver applicato i pesi ed eseguito il training per 50 epoch, il modello raggiunge una precisione massima del 57.60 %, non riuscendo più a migliorare ulteriormente.

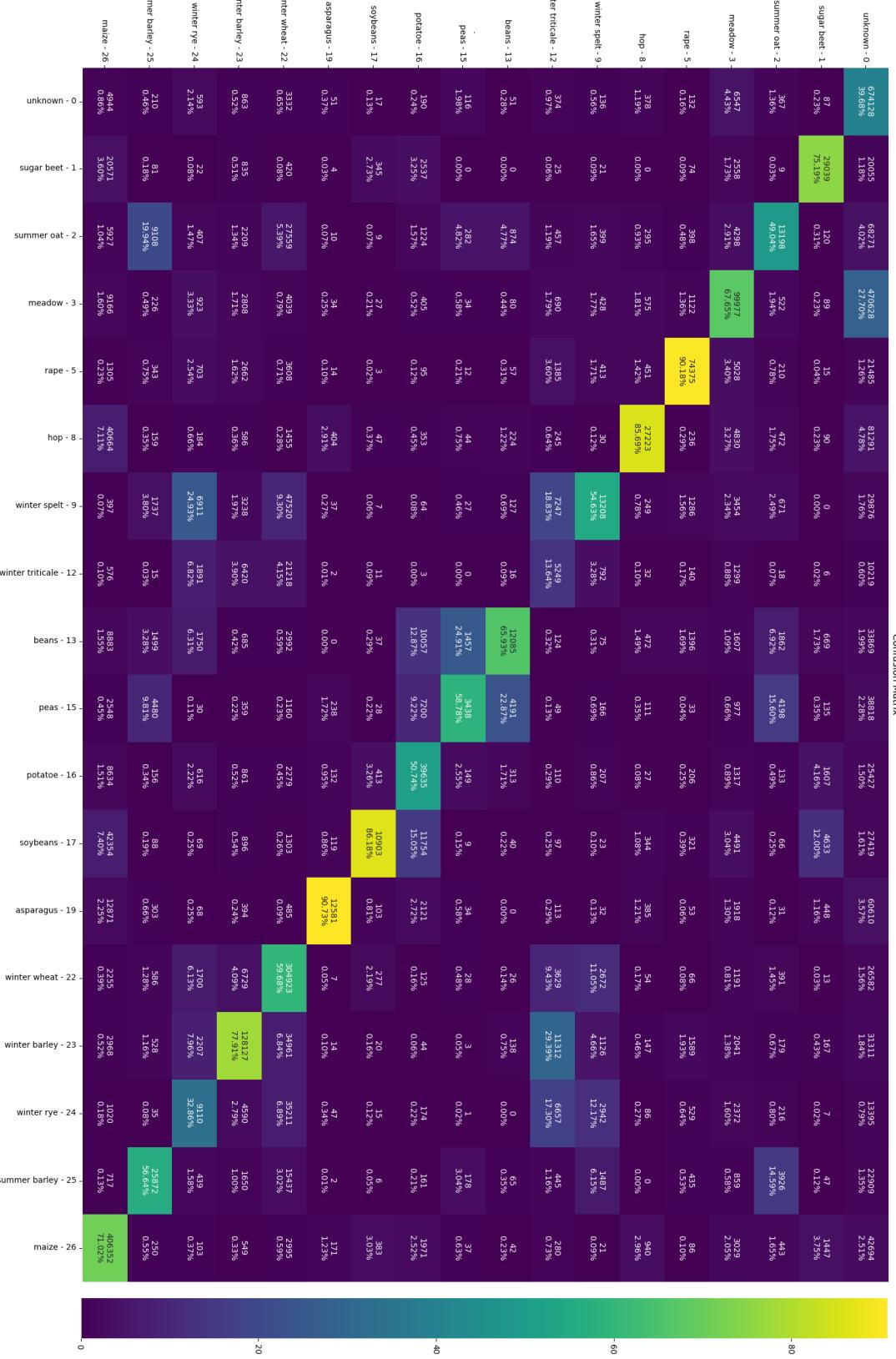


Figura 7.16: Matrice di confusione del modello dopo avere applicato i pesi.

Come possiamo notare dalla matrice di confusione (7.16), ora la situazione è bene diversa

rispetto a prima (7.13). Le classi ora sono più bilanciate, ma comunque il modello continua ad avere difficoltà nel distinguere le varie classi, soprattutto sulla classe 12. Questo sembra essere il massimo ottenibile da questo modello. Una possibile soluzione a questo limite potrebbe essere quella di utilizzare una versione della U-Net che sfrutta la convoluzione 3D, in modo da poter sfruttare a pieno anche le correlazioni temporali tra le immagini.

7.2.5 Applicazione della UNet con convoluzioni 3D

Un modo per superare il limite della UNet con convoluzioni 2D consisterebbe nell'utilizzare una sua versione che sfrutta la convoluzione 3D. Questa architettura, chiamata anche "3D UNet", è trattata in diversi articoli [126, 119] per applicazioni con dati volumetrici, come nel nostro caso. Per implementare questa architettura, ci baseremo su quelle proposte in questi repository [121, 122]. Per implementare questa versione, iniziamo modificando il blocco che racchiude le convoluzioni ed adattiamolo per operare in tre dimensioni.

```

1  class Multiple_Conv3D_Block(nn.Module):
2      def __init__(self,
3          num_convs: int,
4          in_channels: int,
5          out_channels: int,
6          kernel_size: int | Tuple[int, int, int],
7          stride: int | Tuple[int, int, int] = 1,
8          padding: int | Tuple[int, int, int] = 0,
9          bias: bool = False
10     ):
11         super(Multiple_Conv3D_Block, self).__init__()
12         ...
13         self.blockComponents = nn.Sequential()
14         for _ in range(num_convs):
15             self.blockComponents.append(
16                 nn.Conv3d(
17                     in_channels=in_channels,
18                     out_channels=out_channels,
19                     kernel_size=kernel_size,
20                     stride=stride,
21                     padding=padding,
22                     bias=bias
23                 )
24             )
25
26         in_channels = out_channels
27         self.blockComponents.append(nn.BatchNorm3d(out_channels))
28         self.blockComponents.append(nn.ReLU(inplace=True))
29
30     def forward(self, x):
31         return self.blockComponents(x)

```

In questa versione del blocco, è stato introdotto un nuovo elemento, la *Batch Normalization* (normalizzazione batch), posto tra ogni convoluzione e la funzione di attivazione. La *Batch Normalization* permette di velocizzare notevolmente il processo di training, mitigando il problema del gradiente che svanisce. Consente l'uso di tassi di apprendimento più elevati, accelerando la convergenza [75].

Modifichiamo anche il blocco che esegue la deconvoluzione definendo una classe apposita.

```
1  class Deconv3D_Block(nn.Module):
2      def __init__(self,
3          in_channels: int,
4          out_channels: int,
5          kernel_size: int | Tuple[int, int, int],
6          stride: int | Tuple[int, int, int] = 1,
7          padding: int | Tuple[int, int, int] = 0,
8          bias: bool = False
9      ):
10         super(Deconv3D_Block, self).__init__()
11         ...
12         self.deconv = nn.Sequential(
13             nn.ConvTranspose3d(
14                 in_channels,
15                 out_channels,
16                 kernel_size=kernel_size,
17                 stride=stride,
18                 padding=padding,
19                 output_padding=1,
20                 bias=bias
21             ),
22             nn.ReLU(inplace=True)
23         )
24     def forward(self, x):
25         return self.deconv(x)
```

Ora passiamo a ridefinire la struttura della rete per adattarla alla versione 3D.

```
1  class UNet_3D(nn.Module):
2      def __init__(self, **kwargs) -> None:
3          super(UNet_3D, self)
4          ...
5
6          self._DownSampler = nn.MaxPool3d(kernel_size=(2,2,2),
7              stride=(2,2,2))
8          in_feat = self._in_Channel
9
10         for feature in self._features:
11             self._EncoderBlocks.append(
12                 Multiple_Conv3D_Block(
13                     num_convs=2,
14                     in_channels=in_feat,
15                     out_channels=feature,
16                     kernel_size=(3,3,3),
17                     stride=(1,1,1),
18                     padding=(1,1,1),
19                     bias=True
20                 )
21             )
22             in_feat = feature
23
24         self._Bottleneck.append(
```

```

24         Multiple_Conv3D_Block(
25             num_convs=2,
26             in_channels=self._features[-1],
27             out_channels=self._features[-1]*2,
28             kernel_size=3,
29             stride=1,
30             padding=1,
31             bias=True
32         )
33     )
34     for feature in reversed(self._features):
35         self._DecoderBlocks.append(
36             Deconv3D_Block(
37                 in_channels=feature*2,
38                 out_channels=feature,
39                 kernel_size=(3,3,3),
40                 stride=(2,2,2)
41             )
42         )
43     self._DecoderBlocks.append(
44         Multiple_Conv3D_Block(
45             num_convs=2,
46             in_channels=feature*2,
47             out_channels=feature,
48             kernel_size=(3,3,3),
49             stride=(1,1,1),
50             padding=(1,1,1),
51             bias=True
52         )
53     )
54     self._OutputLayer = nn.Sequential(
55         nn.Conv3d(
56             in_channels=self._features[0],
57             out_channels=1,
58             kernel_size=(1,1,1),
59             stride=(1,1,1),
60             padding=0,
61             bias=True
62         ),
63         Squeezzer(1),
64         nn.Conv2d(
65             in_channels=self._depth,
66             out_channels=self._out_channels,
67             kernel_size=(1,1),
68             stride=(1,1),
69             padding=0,
70             bias=True
71         )
72     )
73 ...

```

Nello stadio finale della rete, è stato utilizzato un elemento chiamato "Squeezzer". Questo elemento è stato creato solo per permettere l'utilizzo dell'operazione di *squeeze* all'interno di *nn.Sequential*.

Questo elemento "Squeezer" è così definito:

```
1 class Squeezer(nn.Module):
2
3     def __init__(self, axes):
4         super(Squeezer, self).__init__()
5         self.axes = axes
6
7     def forward(self, x):
8         return x.squeeze(self.axes)
```

Per quanto riguarda la funzione *forward*, essa rimane completamente identica a quella utilizzata nella versione 2D. Passando invece alla classe che gestisce il dataset, riportiamo la funzione *__getitem__* a come era all'inizio, poiché ora possiamo operare con tensori aventi shape [*Batch_Size*, *TimeSeq*, *Channels*, *Width*, *Height*].

```
1 def __getitem__(self, idx: int) -> any:
2     itemDict = self.getItem(idx)
3     itemDict = self.apply_transforms(itemDict)
4     return itemDict['x'], itemDict['y']
```

A questo punto, possiamo procedere nuovamente con l'addestramento del modello e vedere i risultati che otteniamo.

Dopo aver svolto il training della rete per 70 epocha, la situazione che otteniamo è la seguente:

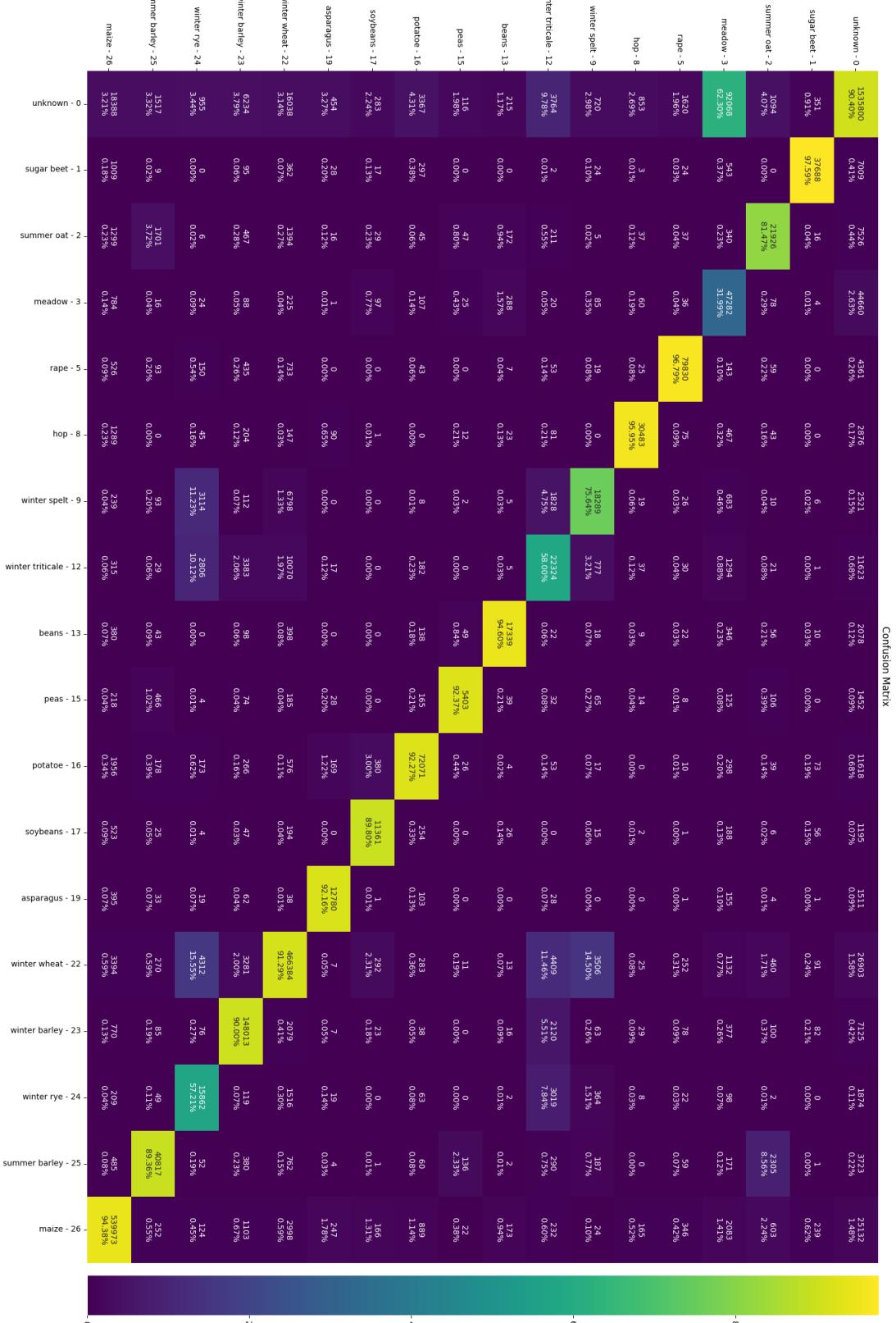


Figura 7.17: Matrice di confusione del modello.

Come possiamo osservare dalla figura (7.17), la situazione è cambiata notevolmente. Il modello è riuscito ad ottenere dei buoni risultati su gran parte delle classi, mostrando una una situazione più bilanciata. Tuttavia, il modello fatica ancora a classificare correttamente le classi: 3, 12 e

24. L'utilizzo di questa versione dell'UNet si ha permesso di arrivare ad una precisione massima del 88.46% (7.18) .

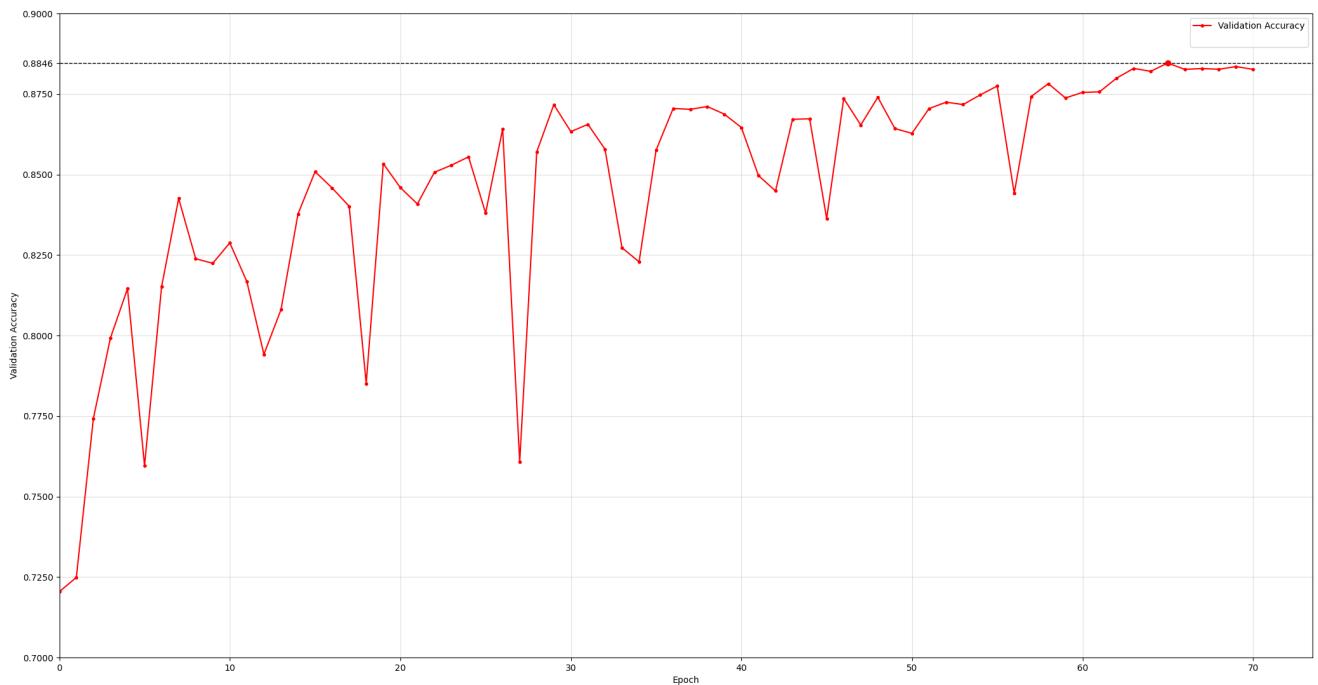


Figura 7.18: Andamento dell'accuratezza del modello.

Proviamo a prendere un campione casuale dal dataset di valutazione e vediamo i risultati che otteniamo. La figura (7.19) mostra una rappresentazione del campione di esempio che viene dato alla rete per svolgere la predizione.

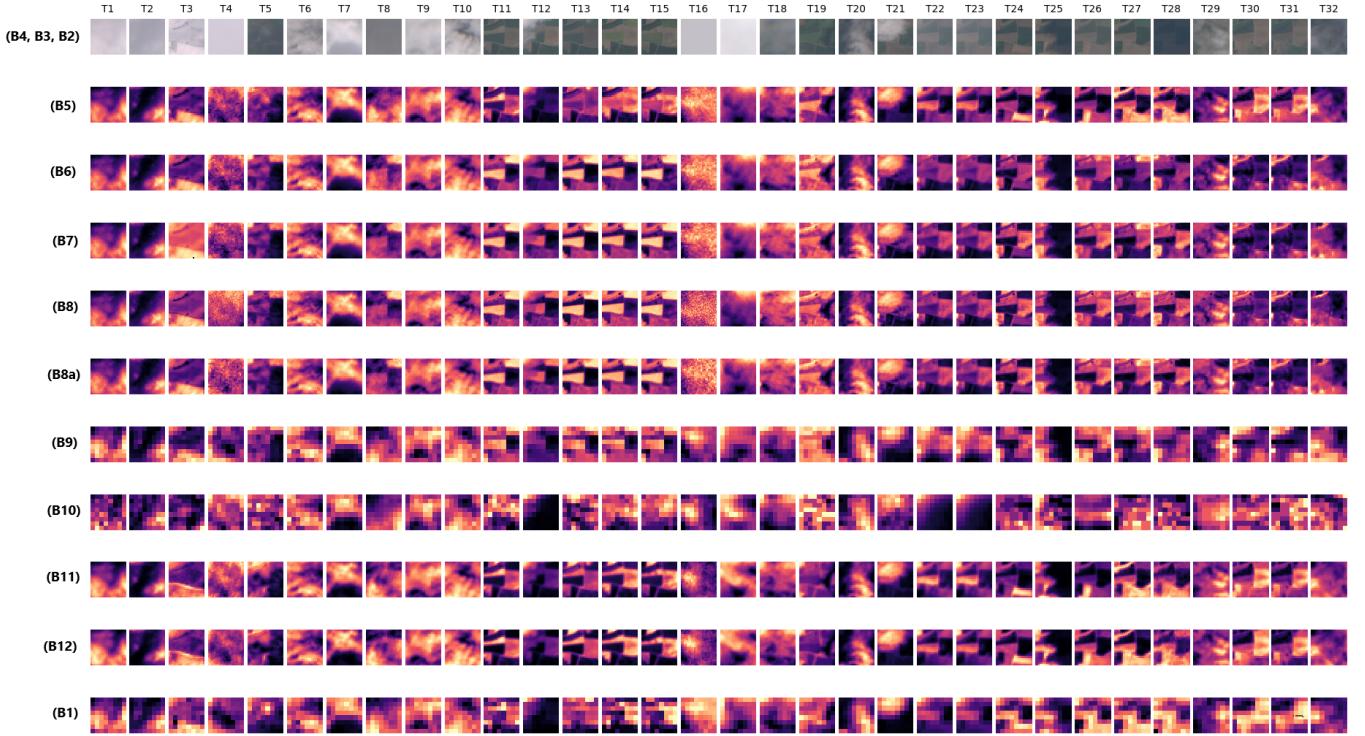


Figura 7.19: Esempio di sequenza temporale fornita alla rete. Le colonne rappresentano le immagini della sequenza, mentre le righe corrispondono alle diverse bande. In alto è riportato il numero dell'immagine all'interno della sequenza, mentre a lato sono indicate le bande (o la banda) rappresentate da ciascuna riga.

La figura 7.20 mostra una rappresentazione dell'output ottenuto dalla rete. L'immagine a sinistra è un'immagine casuale scelta dalla sequenza 7.19. L'immagine al centro mostra quello che il modello dovrebbe predire, mentre l'immagine a destra mostra il risultato della rete.

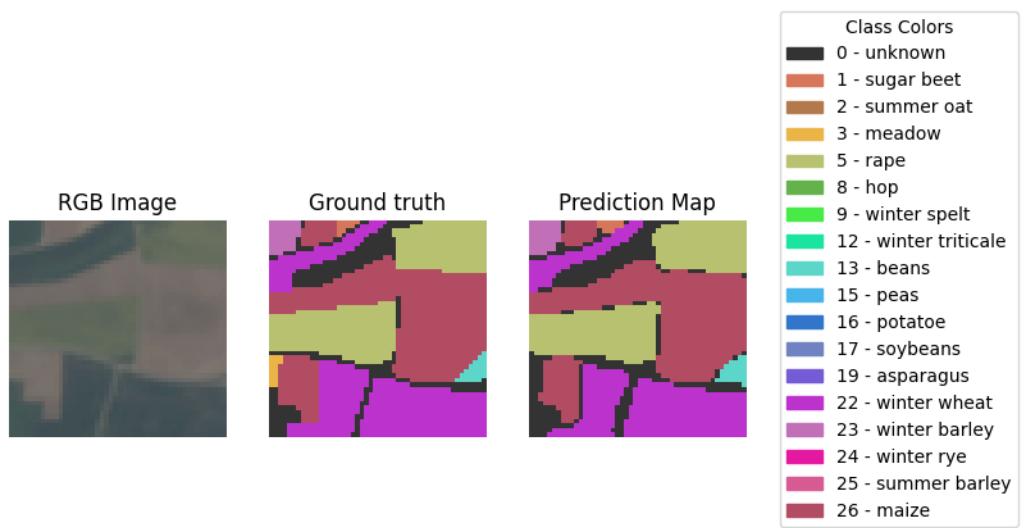


Figura 7.20: Rappresentazione dell'input fornito alla rete, del risultato atteso e del risultato ottenuto.

Come possiamo osservare dalla figura 7.20, il risultato della rete, ad eccezione di alcune piccole aree, si è avvicinato molto alla mappa di verità del tile. Il risultato che abbiamo ottenuto è abbastanza buono, in quanto siamo riusciti a sviluppare un modello in grado di riconoscere con

precisione questi campi agricoli. Tuttavia, si potrebbero adottare delle accortezze sulla classe 0 (*unknown*) per migliorare ulteriormente la precisione del modello.

7.2.6 Ignorare la classe *unknown*

Un altro approccio che è possibile utilizzare consiste nell’addestrare la rete facendole ignorare la classe ”*unknown*”, poiché questa classe indica solo che non abbiamo informazioni su quello che rappresenta il pixel. Pertanto, potrebbe capitare che sia stata assegnata l’etichetta di verità *unknown* ad un pixel che potrebbe appartenere a tutt’altra classe, semplicemente perché non si era certi dell’effettiva classe di appartenenza del pixel.

Ignorare questa classe *unknown* permetterebbe al modello di concentrarsi solo sul riconoscere le classi di cui siamo certi. Per implementare questa soluzione, bisognerebbe agire sulla funzione di loss, specificandole di escludere la classe 0 (*unknown*) dal calcolo della loss e di ignorare le predizioni relative ai pixel la cui etichetta di verità è *unknown*.

Per applicare questa operazione sulla funzione di loss, basta semplicemente fare:

```
1 lossFunction = nn.CrossEntropyLoss(weight=weights_tensor,  
ignore_index=0)
```

A questo punto rifacciamo di nuovo il training e vediamo i risultati che otteniamo.

Dopo Avere addestrato la rete per 72 epoch, questi sono i risultati che si ottengono:

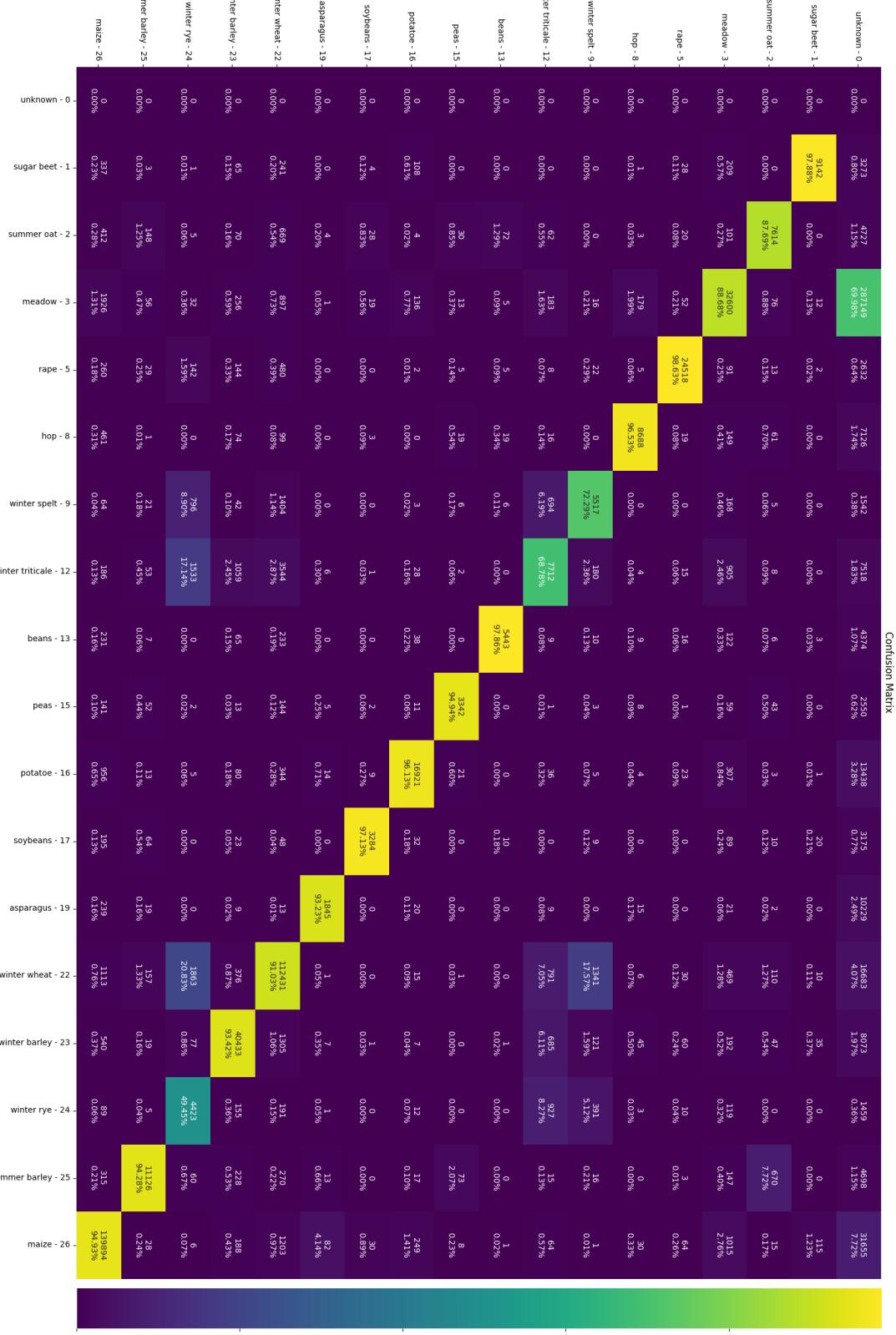


Figura 7.21: Matrice di confusione del modello dopo aver ignorato la classe 0

Come possiamo osservare, la situazione è ulteriormente migliorata rispetto a prima (7.17). Si riscontra ancora una leggera imprecisione sulle classi 24, 12 e 9, dovuta principalmente alla loro somiglianza con altre classi. In ogni caso, siamo riusciti a raggiungere una precisione

(escludendo dal calcolo la classe 0) del 91.8% (7.22), limitandoci solo a classificare i pixel di cui conosciamo la corretta classe di verità. Un altro aspetto che si nota dalla figura 7.21 è come molti pixel, che erano etichettati come *unknown*, siano stati classificati dal modello come appartenenti alla classe 3 (*meadow*).

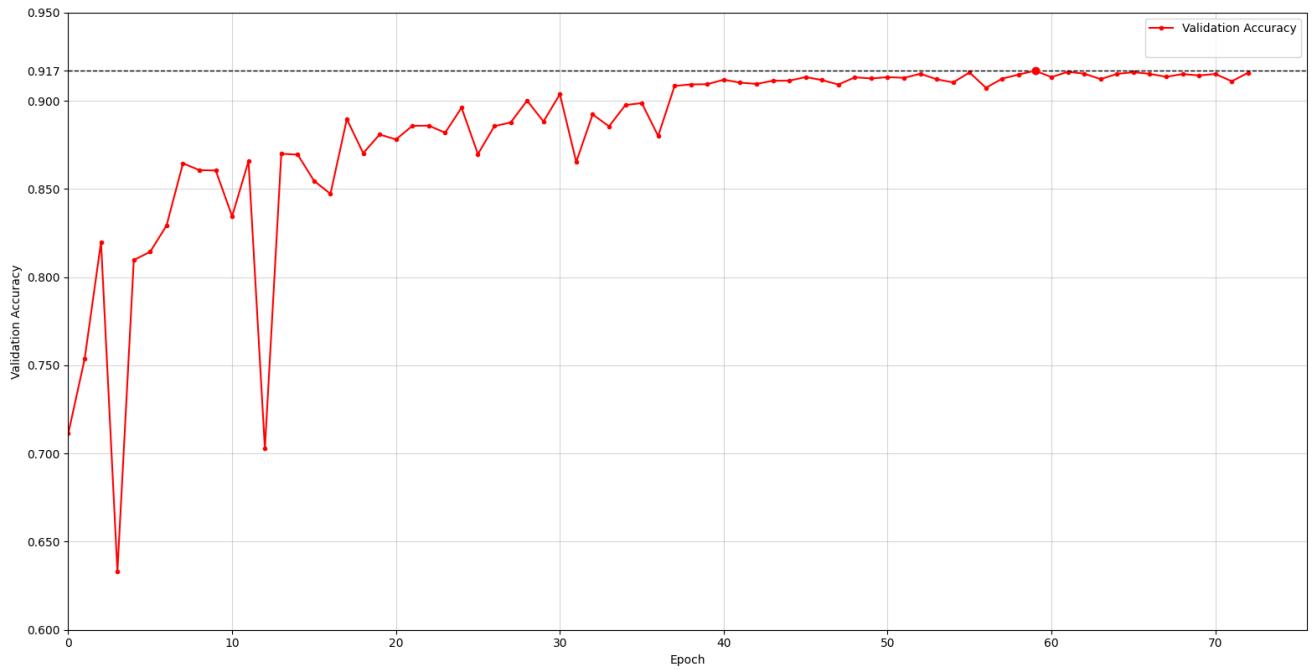


Figura 7.22: Accuratezza del modello.

Con questo campione di esempio (7.23), possiamo osservare, dalle figure 7.24 e 7.25, come il modello si comporta con questo nuovo approccio.

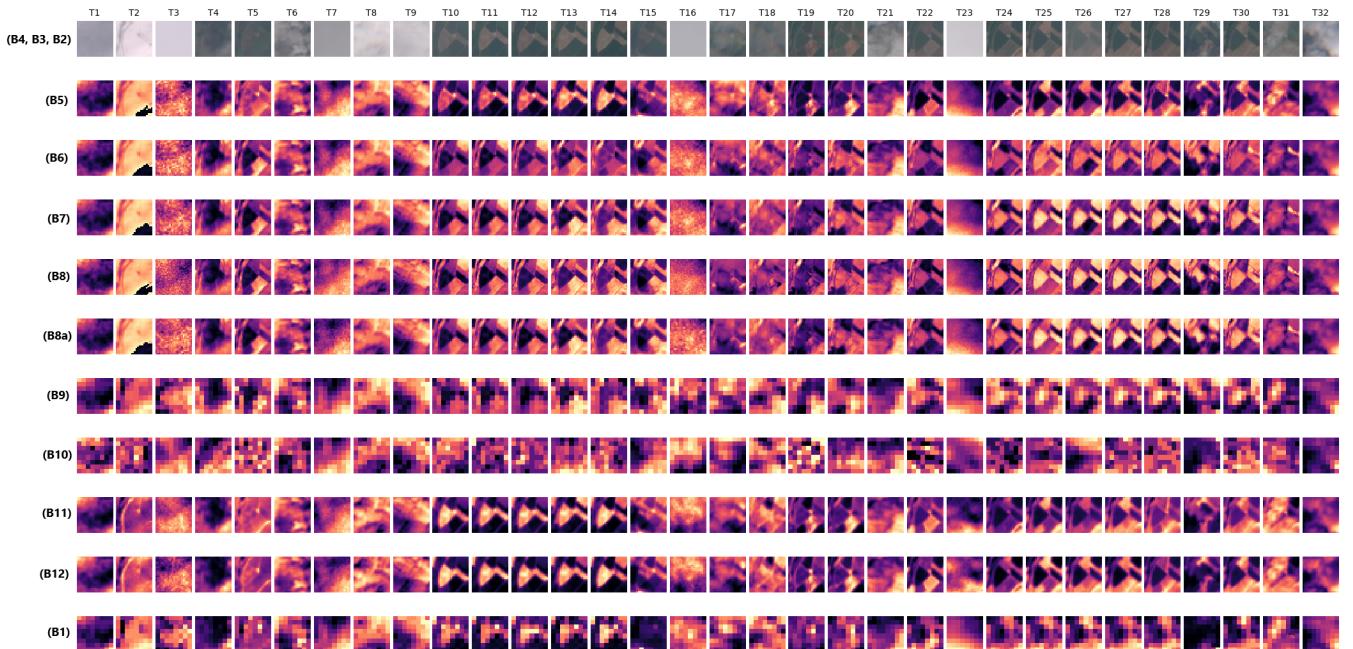


Figura 7.23: Esempio di sequenza temporale fornita alla rete. Le colonne rappresentano le immagini della sequenza, mentre le righe corrispondono alle diverse bande. In alto è riportato il numero dell'immagine all'interno della sequenza, mentre a lato sono indicate le bande (o la banda) rappresentate da ciascuna riga.

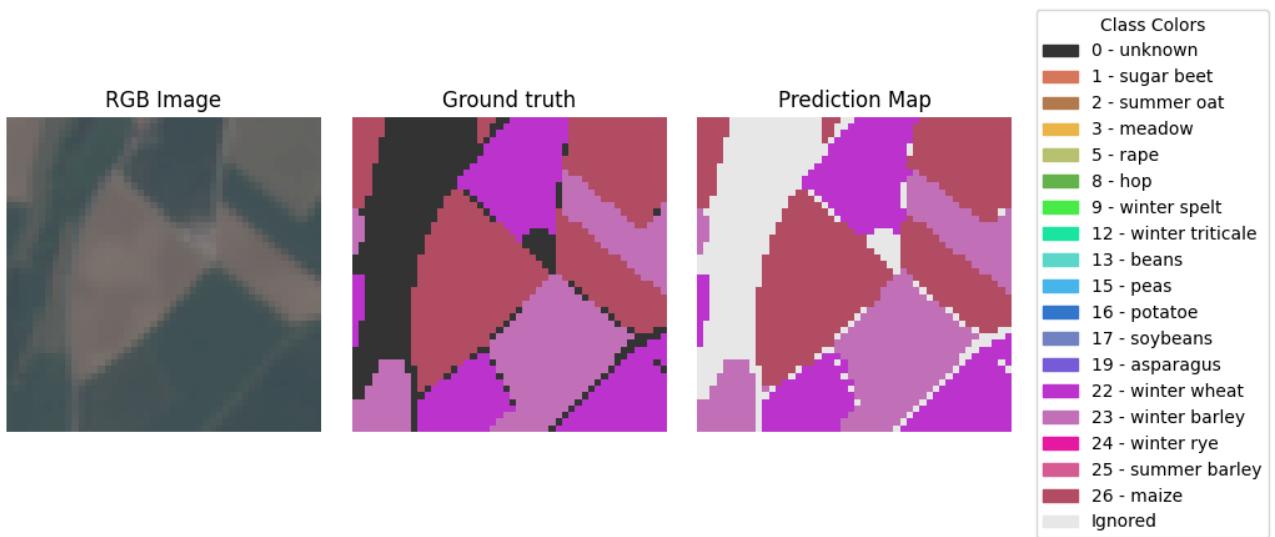


Figura 7.24: Rappresentazione dell’input fornito alla rete, del risultato atteso e del risultato ottenuto.

Nella mappa di predizione della figura 7.24, le aree in cui, nella mappa di verità, è presente la classe 0, sono state ignorate, in quanto non ci interessano le predizioni fatte dal modello in quei punti. Per quanto riguarda le predizioni fatte dal modello sulle aree intereseate, queste risultano essere pressoché identiche alla mappa di verità. Nella figura 7.25 sono riportate tutte le attivazioni delle diverse classi per l’esempio mostrato nella figura 7.23.

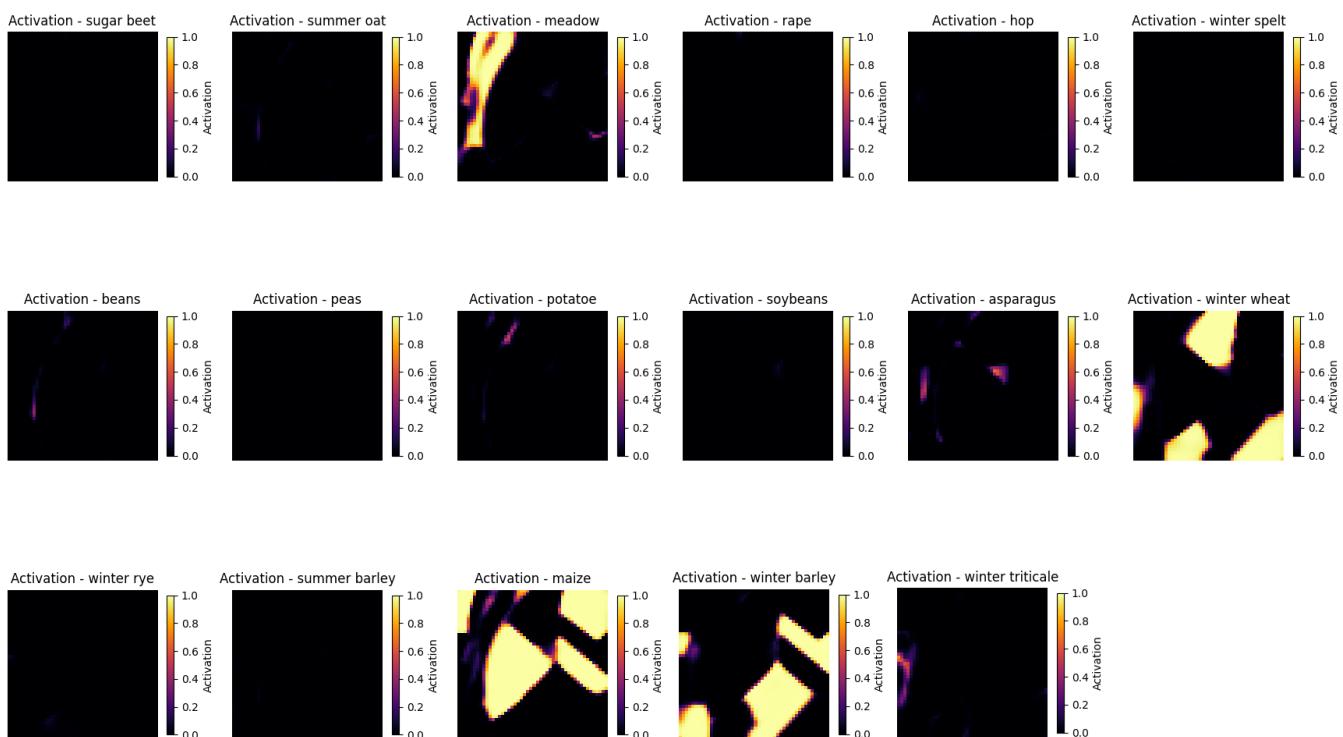


Figura 7.25: Rappresentazione delle attivazioni di ogni classe.

Una cosa che spicca dalla figura 7.25 è come l’attivazione per la classe ”*meadow*” sia molto attiva nelle zone in cui, nella mappa di verità della figura 7.24, è presente la classe ”*unknown*”. Infatti, osservando l’immagine a colori della figura 7.24, si potrebbe pensare che effettivamente ci sia un prato (*meadow*).

7.2.7 Mosaicatura

La mosaicatura è una tecnica utilizzata per ricostruire un'immagine di un'ampia area geografica a partire da tile elaborati singolarmente. Essa consiste nell'unire in modo coerente i risultati delle predizioni effettuate su ciascun tile, preservando la continuità spaziale dell'immagine o dei dati. Questo processo consente anche di capire se esiste continuità nelle predizioni del modello tra i diversi tile. Le immagini qui sotto rappresentano un esempio di mosaicatura su alcuni blocchi di tile presi dal dataset di valutazione. A sinistra è riportata l'immagine a colori, mentre a destra le predizioni fatte dal modello.



Figura 7.26: Mosaicatura a colori di un blocco di tile preso dal dataset di valutazione.



Figura 7.27: Mosaicatura delle predizioni dell'immagine 7.26.

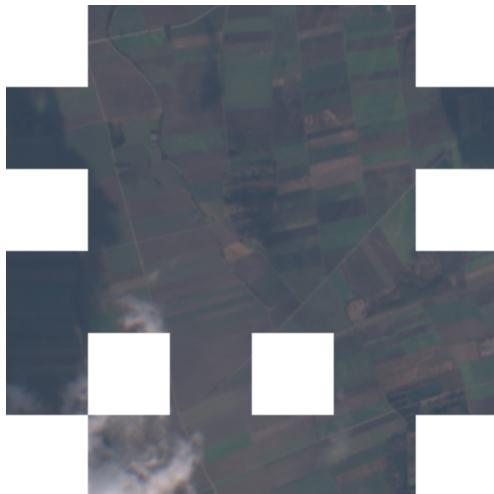


Figura 7.28: Mosaicatura a colori di un blocco di tile preso dal dataset di valutazione.



Figura 7.29: Mosaicatura delle predizioni dell'immagine 7.28.



Figura 7.30: Mosaicatura a colori di un blocco di tile preso dal dataset di valutazione.



Figura 7.31: Mosaicatura delle predizioni dell'immagine 7.30.

Come si può osservare dalle figure precedenti, il modello riesce anche ad avere una coerenza nelle predizioni tra tile differenti. Se provassimo anche a considerare le aree dove nella mappa di verità si ha la classe è 0 ("unknown"), la mosaicatura apparirebbe così:

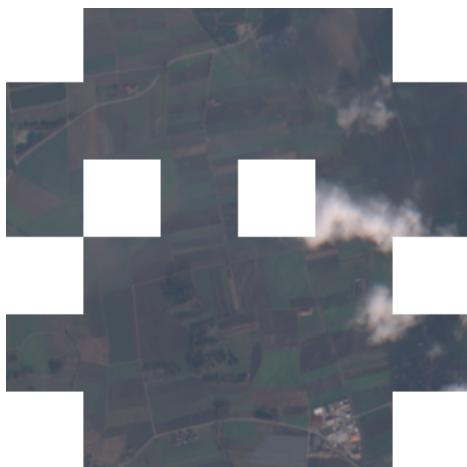


Figura 7.32: Mosaicatura a colori di un blocco di tile preso dal dataset di valutazione.

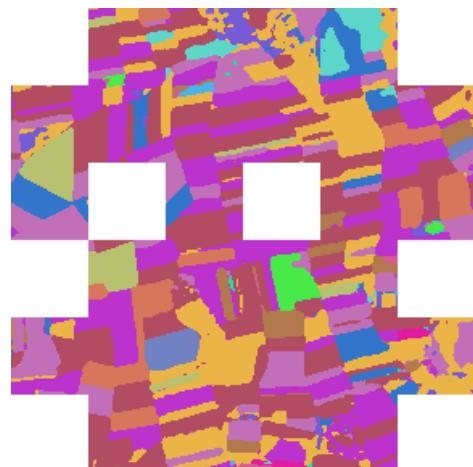


Figura 7.33: Mosaicatura delle predizioni dell'immagine 7.32.



Figura 7.34: Mosaicatura a colori di un blocco di tile preso dal dataset di valutazione.

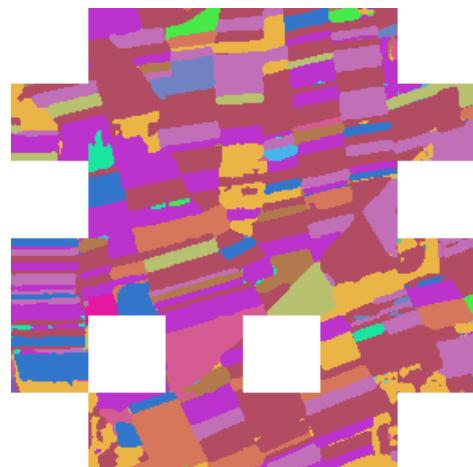


Figura 7.35: Mosaicatura delle predizioni dell'immagine 7.34.

Capitolo 8

Conclusioni

Questa sperimentazione ci ha permesso di comprendere come implementare ed ottimizzare reti neurali per la mappatura dei campi agricoli utilizzando immagini satellitari Sentinel-2. In particolare, sono state testate due diverse architetture, descrivendo per ciascuna il processo adottato per migliorarne la precisione. Tra i modelli sviluppati, i risultati migliori sono stati ottenuti dal modello basato sull'architettura UNet-3D. Su un dataset di validazione, questo modello ha raggiunto i seguenti risultati:

Validation				
Class	Precision	Recall	F1-Score	Kappa
Sugar beet	86.37	93.51	88.85	0.88
Summer oat	69.55	75.66	70.12	0.69
Meadow	77.90	77.48	75.10	0.70
Rape	89.79	93.06	90.94	0.90
Hop	85.46	91.22	86.62	0.81
Winter spelt	71.31	63.33	63.45	0.60
Winter triticale	66.92	55.66	55.99	0.53
Beans	76.44	84.39	78.35	0.74
Peas	82.81	79.52	78.31	0.77
Potato	84.58	85.88	83.77	0.81
Soybeans	81.53	83.95	80.61	0.78
Asparagus	79.17	78.92	75.87	0.72
Winter wheat	92.12	85.07	86.99	0.82
Winter barley	86.29	85.10	84.56	0.83
Winter rye	65.76	55.33	54.81	0.52
Summer barley	81.07	82.64	80.27	0.79
Maize	95.84	90.78	92.63	0.88
Weighted Avg.	88.80	85.29	85.55	-
Overall Accuracy	91.84%			
Overall Kappa	89.95%			

Tabella 8.1: Metriche delle prestazioni per la segmentazione delle colture sul dataset di validazione.

Sul dataset di valutazione, il modello ha ottenuto i seguenti risultati:

Test				
Class	Precision	Recall	F1-Score	Kappa
Sugar beet	0.81	0.91	0.85	0.83
Summer oat	0.75	0.77	0.73	0.72
Meadow	0.76	0.78	0.75	0.71
Rape	0.90	0.94	0.92	0.89
Hop	0.83	0.85	0.83	0.78
Winter spelt	0.76	0.59	0.59	0.54
Winter triticale	0.65	0.55	0.55	0.53
Beans	0.77	0.82	0.77	0.75
Peas	0.71	0.72	0.69	0.68
Potato	0.81	0.83	0.81	0.79
Soybeans	0.84	0.88	0.85	0.80
Asparagus	0.74	0.65	0.60	0.57
Winter wheat	0.93	0.86	0.87	0.83
Winter barley	0.90	0.89	0.88	0.86
Winter rye	0.57	0.48	0.47	0.45
Summer barley	0.83	0.80	0.79	0.75
Maize	0.96	0.91	0.93	0.88
Weighted Avg.	0.90	0.86	0.87	-
Overall Accuracy	91.92%			
Overall Kappa	89.82%			

Tabella 8.2: Metriche delle prestazioni per la segmentazione delle colture sul dataset di valutazione.

L'accuratezza nella classificazione delle colture ottenuta dal nostro modello può essere considerata molto soddisfacente, avendo raggiunto un valore di 91.84% sulla validazione e un valore di 92.92% sulla valutazione. Questi risultati superano l'accuratezza dell'89.7% ottenuta sul dataset di valutazione dal modello descritto nell'articolo *”Multi-Temporal Land Cover Classification with Sequential Recurrent Encoders”* [130], il primo a trattare l'applicazione di una rete neurale sul dataset Sentinel2-Munich480. Il modello presentato in quell'articolo era basato su un'architettura di tipo Sequential Recurrent Encoders [123] per la mappatura delle colture. Il nostro modello ha ottenuto risultati comparabili a quelli descritti in tale articolo. In particolare, entrambi i modelli, presentano delle difficoltà nel riconoscere le classi *”Winter rye”* e *”Winter triticale”*, dovute principalmente alla difficoltà nel distinguere l'aspetto spettrale e fenologico delle due colture. Inoltre, il nostro modello si è avvicinato anche ai risultati ottenuti dal modello descritto nell'articolo *”Enhancing Crop Segmentation in Satellite Image Time Series with Transformer Networks”* [131]. In questo articolo è stato presentato un modello basato su un'architettura *Transformer* [124] per eseguire la mappatura delle colture sul dataset Sentinel2-Munich480, ottenendo una precisione del 96.14% sul dataset di validazione e una precisione del 95.26% sul dataset di valutazione.

Confrontando il nostro modello con quelli trattati in altri articoli, questa è la situazione che si riscontra:

Munich Metrics				
Model	OA_test	OK_test	OA_val	OK_val
Swin UNETR [131]	95.26%	93.89%	96.14%	94.89%
UNet 3D [132]	94.73%	93.46%	-	-
FPN3D [133]	93.11%	91.44%	-	-
*UNet 3D	91.92%	89.82%	91.84%	89.95%
SRE [130]	89.6%	87.0%	-	-
DeepLabv3 3D [131]	85.98%	82.53%	-	-

Tabella 8.3: Confronto tra le metriche globali ottenute da differenti modelli. OA rappresenta l'accuratezza complessiva e OK il coefficiente di kappa. Il simbolo '*' indica il nostro modello.

Ci consideriamo soddisfatti dei risultati ottenuti, in quanto siamo riusciti a raggiungere gli obiettivi prefissati. Tuttavia, esistono ancora margini di miglioramento per ottimizzarne ulteriormente l'accuratezza del nostro modello. Ad esempio, si potrebbe prestare maggiore attenzione a come variare il valore del *learning rate*, per evitare che il modello resti bloccato in un minimo locale, così come alla scelta degli algoritmi di ottimizzazione da utilizzare. Inoltre, alcuni aspetti legati all'architettura della rete potrebbero essere perfezionati. Nello specifico, in questo lavoro è stata impiegata una nostra variante dell'architettura U-Net 3D, diversa rispetto ad altre implementazioni presenti in diversi repository [121, 122, 132].

Bibliografia

- [1] [https://en.wikipedia.org/wiki/Hadamard_product_\(matrices\)](https://en.wikipedia.org/wiki/Hadamard_product_(matrices))
- [2] https://it.wikipedia.org/wiki/Derivata_parziale
- [3] <https://pmf-research.eu/deep-learning-machine-learning-ia-tradizionale/>.
- [4] https://it.wikipedia.org/wiki/Intelligenza_artificiale
- [5] <https://www.digitalbimitalia.it/it/news/artificial-intelligence-machine-learning-o-deep-learning-possibili-applicazioni-all>
- [6] <https://www.bbc.com/news/technology-30290540>
- [7] <https://www.ibm.com/it-it/topics/machine-learning>
- [8] <https://www.agendadigitale.eu/cultura-digitale/deep-learning-cose-come-funziona-e-applicazioni>
- [9] <https://www.bnova.it/intelligenza-artificiale/deep-learning-cose-e-quali-le-applicazioni/>
- [10] <https://medium.com/%40jacopokahl/i-tre-principali-tipi-di-machine-learning-77ca20d0dbdd>
- [11] [https://medium.com/@Bright-Data/what-is-a-dataset-definition-use-cases-benefits-and-example-9aaf5ecc301e.](https://medium.com/@Bright-Data/what-is-a-dataset-definition-use-cases-benefits-and-example-9aaf5ecc301e)
- [12] [https://it.wikipedia.org/wiki/Dataset.](https://it.wikipedia.org/wiki/Dataset)
- [13] https://en.wikipedia.org/wiki/Training%2C_validation%2C_and_test_data_sets
- [14] <https://encord.com/blog/train-val-test-split>
- [15] <https://developers.google.com/machine-learning/crash-course/overfitting/dividing-datasets?hl=it>
- [16] <https://deeplearningitalia.com/parliamo-di-split-utilizzati-per-dividere-il-dataset-in-training-validation-e-test->
- [17] [https://www.kaggle.com/datasets/hojjatk/mnist-dataset.](https://www.kaggle.com/datasets/hojjatk/mnist-dataset)
- [18] https://en.wikipedia.org/wiki/MNIST_database
- [19] <https://www.kaggle.com/datasets/artelabsuper/sentinel2-munich480>
- [20] https://it.wikipedia.org/wiki/Monaco_di_Baviera
- [21] <https://www.ibm.com/it-it/topics/confusion-matrix>
- [22] <https://www.zerounoweb.it/big-data/confusion-matrix-guida-pratica-per-valutare-il-modello-di-classificazione/>.
- [23] [https://it.wikipedia.org/wiki/Python.](https://it.wikipedia.org/wiki/Python)
- [24] <https://www.analyticsvidhya.com/blog/2019/03/deep-learning-frameworks-comparison/>.
- [25] <https://viso.ai/deep-learning/pytorch-vs-tensorflow/>.
- [26] <https://devopedia.org/deep-learning-frameworks>.
- [27] <https://paperswithcode.com/trends>.
- [28] https://en.wikipedia.org/wiki/PyTorch_Lightning.
- [29] <https://lightning.ai/docs/pytorch/stable/>
- [30] <https://fidacaro.com/cose-un-tensore-e-come-viene-utilizzato-nel-machine-learning>.
- [31] <https://www.ibm.com/it-it/topics/pytorch>.

- [32] <https://www.analyticsvidhya.com/blog/2022/07/data-representation-in-neural-networks-tensor/>
- [33] <https://towardsdatascience.com/what-is-a-tensor-in-deep-learning-6dedd95d6507>
- [34] <https://medium.datadriveninvestor.com/what-is-the-tensor-in-deep-learning-77c2af7224a1>
- [35] <https://deepai.org/machine-learning-glossary-and-terms>
- [36] <https://www.ibm.com/it-it/topics/neural-networks>
- [37] <https://www.missionescienza.it/reti-neurali-artificiali-parte-2/>
- [38] https://it.m.wikipedia.org/wiki/Rete_neurale
- [39] <https://the-geeks-of-the-round-table.medium.com/introduction-to-deep-learning-the-perceptron-part-2-bccf30be1a4b>
- [40] <https://medium.com/@siddharthshah2601/mcculloch-pitts-neuron-a-computational-model-of-biological-neuron-ce57239a951e>
- [41] <https://medium.com/@nexomind/alle-origini-del-neurone-artificiale-31f848efd6b6>
- [42] <https://medium.com/@vnohitha13/neural-networks-52bec25688eb>
- [43] <https://vitalflux.com/how-do-we-build-deep-neural-network-using-perceptron/>
- [44] <https://medium.com/@rdugue1/neural-network-building-blocks-7ea6f8c790bf>
- [45] <https://medium.com/codex/single-layer-perceptron-and-activation-function-b6b74b4aae66>
- [46] https://www.saedsayad.com/artificial_neural_network_bkp.htm
- [47] <https://www.geeksforgeeks.org/what-is-perceptron-the-simplest-artificial-neural-network/>
- [48] <https://wiki.pathmind.com/multilayer-perceptron>
- [49] <https://www.datacamp.com/tutorial/multilayer-perceptrons-in-machine-learning>
- [50] <http://neuralnetworksanddeeplearning.com/.>
- [51] [https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21.](https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21)
- [52] <https://medium.com/geekculture/mathematics-behind-gradient-descent-f2a49a0b714f>
- [53] <https://www.mondadorieducation.it/fisica-scientifica-ss2/le-reti-neurali-artificiali/>
- [54] <https://towardsdatascience.com/introduction-to-math-behind-neural-networks-e8b60dbbdeba>
- [55] [https://it.wikipedia.org/wiki/Entropia_\(teoria_dell%27informazione\).](https://it.wikipedia.org/wiki/Entropia_(teoria_dell%27informazione).)
- [56] https://365datascience.com/tutorials/machine-learning-tutorials/cross-entropy-loss/?utm_source=chatgpt.com.
- [57] <https://www.datacamp.com/tutorial/the-cross-entropy-loss-function-in-machine-learning.>
- [58] <https://netai.it/guida-rapida-alle-funzioni-di-attivazione-nel-deep-learning/#comment-27.>
- [59] <https://medium.com/analytics-vidhya/activation-functions-all-you-need-to-know-355a850d025e>
- [60] <https://neuralthreads.medium.com/softmax-function-it-is-frustrating-that-everyone-talks-about-it-but-very-few-talk-a>
- [61] <https://musstafa0804.medium.com/optimizers-in-deep-learning-7bf81fed78a0>
- [62] <https://sweta-nit.medium.com/batch-mini-batch-and-stochastic-gradient-descent-e9bc4cacd461>
- [63] <https://en.wikipedia.org/wiki/Convolution>
- [64] <https://www.geeksforgeeks.org/apply-a-2d-convolution-operation-in-pytorch/>
- [65] <https://medium.com/codex/understanding-convolutional-neural-networks-a-beginners-journey-into-the-architectu>

- [66] <https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/>
- [67] UR<https://dennybritz.com/posts/wildml/>
understanding-convolutional-neural-networks-for-nlp/L
- [68] <https://www.baeldung.com/cs/neural-networks-pooling-layers>
- [69] <https://www.analyticsvidhya.com/blog/2021/03/the-architecture-of-lenet-5/>
- [70] <https://en.wikipedia.org/wiki/LeNet>
- [71] <https://www.kaggle.com/code/shivamb/3d-convolutions-understanding-use-case>
- [72] <https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1>
- [73] <https://sarosijbose.github.io/files/talks/A%20General%20Overview%20of%203D%20Convolution%20.pdf>
- [74] <https://ai.stackexchange.com/questions/13692/when-should-i-use-3d-convolutions>
- [75] <https://medium.com/@pouyahallaj/batch-normalization-a-deep-dive-for-machine-learning-enthusiasts-60e4a76d10e8>
- [76] https://semiautomaticclassificationmanual-v5.readthedocs.io/it/latest/remote_sensing.html
- [77] <https://www.alspergis.altervista.org/lezione/>
- [78] <https://it.wikipedia.org/wiki/Telerilevamento>
- [79] https://www.nateko.lu.se/sites/nateko.lu.se.sv/files/remote_sensing_and_gis_20111212.pdf
- [80] https://appliedsciences.nasa.gov/sites/default/files/2022-11/Fundamentals_of_RS_Edited_SC.pdf
- [81] https://books.google.it/books?id=NkLmDjSS8TsC&printsec=frontcover&hl=it&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false
- [82] <https://gisgeography.com/remote-sensing-earth-observation-guide>
- [83] https://www.researchgate.net/figure/Common-Remote-Sensing-Platform-and-Sensor-Combinations-and-Remote-Sensing-Data-Left_fig1_341582585
- [84] <https://www.spatialpost.com/types-of-platforms-in-remote-sensing/>
- [85] <https://it.wikipedia.org/wiki/Luce>
- [86] https://en.wikipedia.org/wiki/Electromagnetic_radiation
- [87] <https://www.rfcafe.com/references/electrical/electromagnetic-wave-physics.htm>
- [88] <https://geolearn.in/classification-methods-in-remote-sensing-gis/>
- [89] https://it.wikipedia.org/wiki/Spettro_elettromagnetico
- [90] https://it.wikipedia.org/wiki/Firma_spettrale
- [91] <https://associazionegioconda.it/la-propagazione-della-luce-2>
- [92] <https://it.wikipedia.org/wiki/Luce>
- [93] <https://ltb.itc.utwente.nl/498/concept/81854>
- [94] <https://ecampusontario.pressbooks.pub/remotesensing/chapter/chapter-4-emr-interactions-with-the-atmosphere-and-with-the-surface/>
- [95] <https://it.wikipedia.org/wiki/Riflettanza>
- [96] <https://it.wikipedia.org/wiki/Radianza>
- [97] <https://www.slideshare.net/slideshow/il-corpo-nero-e-la-quantizzazione-dellenergia/36910588>
- [98] <https://www.copernicus.eu/it/informazioni-su-copernicus>
- [99] https://it.wikipedia.org/wiki/Orbita_polare
- [100] https://it.wikipedia.org/wiki/Orbita_eliosincrona
- [101] <https://gisgeography.com/sentinel-2-bands-combinations/>
- [102] <https://sentiwiki.copernicus.eu/web/s2-applications>
- [103] <https://it.wikipedia.org/wiki/Sentinel-2>

- [104] <https://en.wikipedia.org/wiki/Sentinel-2>
- [105] <https://www.alspergis.altervista.org/data/sentinel2.html>
- [106] <https://www.satimagingcorp.com/satellite-sensors/other-satellite-sensors/sentinel-2a/>
- [107] <https://sentiwiki.copernicus.eu/web/s2-mission>
- [108] <https://www.edmundoptics.com/knowledge-center/application-notes/imaging/hyperspectral-and-multispectral-imaging/?srsltid=AfmB0orZC8mMEPOWCUTBWsdLJlQBYNeGG1ZIxcyWo03PNdRGotbEGht>
- [109] <https://innoter.com/en/articles/multispectral-imaging/>
- [110] https://en.wikipedia.org/wiki/Multispectral_imaging
- [111] <https://www.alspergis.altervista.org/lezione/13.html>
- [112] <https://www.domsoria.com/2022/11/che-cosa-e-lo-spazio-latente-o-latent-space-ed-a-cosa-serve/>
- [113] <https://www.andreaprovino.it/image-segmentation-segmentazione-semantica-e-delle-istanze.>
- [114] <https://thegradient.pub/semantic-segmentation/.>
- [115] <https://www.labellerr.com/blog/semantic-vs-instance-vs-panoptic-which-image-segmentation-technique-to-choose/.>
- [116] <https://medium.com/@raj.pulapakura/image-segmentation-a-beginners-guide-0ede91052db7>
- [117] <https://medium.com/@alejandro.itoaramendia/decoding-the-u-net-a-complete-guide-810b1c6d56d8>
- [118] https://it.wikipedia.org/wiki/Agricoltura_di_precisione
- [119] <https://miccai-sb.github.io/materials/Hoang2019b.pdf>
- [120] <https://arxiv.org/abs/1505.04597>
- [121] <https://github.com/wolny/pytorch-3dunet/blob/master/pytorch3dunet>
- [122] <https://gitlab.com/mattiagatti/sentinel2-crop-mapping-models>
- [123] https://it.wikipedia.org/wiki/Rete_neurale_riconoscitrice
- [124] [https://it.wikipedia.org/wiki/Trasformatore_\(informatica\)](https://it.wikipedia.org/wiki/Trasformatore_(informatica))

- [125] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.
<https://www.deeplearningbook.org/>.

- [126] Çiçek, Ö., Abdulkadir, A., Lienkamp, S. S., Brox, T., & Ronneberger, O. (2016). "3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation". In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2016*. Fonte:
<https://arxiv.org/pdf/1606.06650>

- [127] Rumelhart, Hinton e Williams (1986): "Learning representations by back-propagating errors" Disponibile online: <https://www.nature.com/articles/323533a0>

- [128] <https://websites.umass.edu/brain-wars/1957-the-birth-of-cognitive-science/the-perceptron-a-perceiving-and-recognizing-automaton/>

- [129] Minsky, M., & Papert, S. A. (1969). Perceptrons: An introduction to computational geometry. MIT Press. <https://direct.mit.edu/books/monograph/3132/PerceptronsAn-Introduction-to-Computational>

- [130] Rußwurm, M., & Körner, M. (2018). "Multi-Temporal Land Cover Classification with Sequential Recurrent Encoders". In *ISPRS International Journal of Geo-Information*. Fonte: <https://arxiv.org/pdf/1802.02080>
- [131] Gallo, I., Gatti, M., Landro, N., Loschiavo, C., Rehman, A. U., & Boschetti, M. (2024). "Enhancing Crop Segmentation in Satellite Image Time Series with Transformer Networks". In *Remote Sensing Letters*. Fonte: <https://arxiv.org/pdf/2412.01944>
- [132] Ignazio Gallo, Luigi Ranghetti, Nicola Landro, Riccardo La Grassa, and Mirco Boschetti. In-season and dynamic crop mapping using 3d convolution neural networks and sentinel-2 time series. ISPRS Journal of Photogrammetry and Remote Sensing, 195:335–352, 2023.
- [133] Ignazio Gallo, Riccardo La Grassa, Nicola Landro, and Mirco Boschetti. Sentinel 2 time series analysis with 3d feature pyramid network and time domain class activation intervals for crop mapping. ISPRS International Journal of Geo-Information, 10(7), 2021