UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI **INGEGNERIA DELL'INFORMAZIONE**

Corso di Laurea triennale in **Ingegneria Elettronica**

Studio ed Analisi di Reti Neurali su Sistema Embedded per il Riconoscimento di Buche in Ambito di Guida Autonoma

Study and Analysis of Neural Networks on Embedded Systems for Pothole Detection in Autonomous Driving

Relatrice:

Dott.ssa. **Laura Falaschetti**

Tesi di Laurea di:

Nicola Fiorentini

A.A. **2023 / 2024**

SOMMARIO

1. INTRODUZIONE ____ 7

2. RETI NEURALI ____ 8

2.1 DEEP NEURAL NETWORKS ____ 10

2.1.0 Funzione di attivazione ____ 12

2.2.1 Sigmoid ____ 12

2.2.2 tanh ____ 13

2.2.3 ReLU ____ 13

2.2.4 Softmax ____ 14

2.2.5 Funzione di costo ____ 15

2.2.6 Discesa del gradiente ____ 16

2.2.7 Modello Adams ____ 17

2.2 CONVOLUTIONAL NEURAL NETWORKS ____ 17

2.5.1 Input image ____ 18

2.5.2 Strato Convolutivo ____ 19

2.5.3 Livello di raggruppamento ____ 21

2.5.4 Fully Connected Layer ____ 22

2.3 STATO DELL'ARTE DELLE CONVOLUTIONAL NEURAL NETWORKS ____ 22

2.6.1 Visual Geometry Group 16 ____ 23

2.6.2 ResNet50 ____ 24

2.6.3 Blocchi residui ____ 25

2.6.4 EfficientNetB0 ____ 27

2.6.5 MobileNetV2 ____ 31

3. CASO DI STUDIO ____ 32

3.1 CLASSIFICAZIONE BINARIA ____ 32

3.2 DATASET DI IMMAGINI ____ 33

3.3 DATA AUGMENTATION 34
3.3.1 Operazioni geometriche eseguite sulle immagini del dataset 34
3.3.2 Trasformazioni fotometriche 35
3.4 GOOGLE COLAB, TENSORFLOW E KERAS 35
3.5 MODELLO UTILIZZATO PER LA CLASSIFICAZIONE 36
3.5.1 Caricamento e divisione del dataset 36
3.5.2 Data Augmentation 38
3.5.3 Caricamento modello pre-allenato 39
3.5.4 Definizione della funzione *build_model()* 39
3.5.5 Definizione della funzione *calculate_map()* 40
3.5.6 Definizione della funzione get_macs() 41
3.5.7 Struttura del modello per la fase di training e di verifica 42
3.5.8 Calcolo del tempo di inferenza in fase di verifica 43
3.5.9 Visualizzazione delle specifiche hardware fornite da Google Colab 44
3.5.10 Funzioni per la rappresentazione grafica delle prestazioni 45
4. RISULTATI SPERIMENTALI 46
4.1 MATRICE DI CONFUSIONE 46
4.2 FUNZIONE DI COSTO 49
4.3 RISULTATI FINALI OTTENUTI 53
5. ANALISI DEI RISULTATI OTTENUTI 54
6. PORTING SU BOARD STM32 56
6.1 MICROCONTROLLORE STM32H7S78-DK 61
6.2 PORTING SU STM32H7S78-DK CON STM32CUBE.AI DEVELOPER CLOUD 62
7. CONCLUSIONI E SVILUPPI FUTURI 70
BIBLIOGRAFIA 71

INDICE DELLE FIGURE

Figura 1: Architettura delle prime reti neurali _____ 9

Figura 2: Struttura del percettrone con funzione di apprendimento (error-back propagation) ____ 9

Figura 3: Architettura di una deep neural network _____ 10

Figura 4: Deep neural network e struttura dello strato nascosto. _____ 11

Figura 5: Funzione sigmoide _____ 12

Figura 6: Funzione tanh _____ 13

Figura 7: Funzione ReLU _____ 14

Figura 8: Funzione Softmax _____ 14

Figura 9: Cross-entropy _____ 15

Figura 10: Rappresentazione tridimensionale della discesa del gradiente _____ 16

Figura 11: Esempio di rete neurale convoluzionale _____ 18

Figura 12: Acquisizione dell'immagine per una rete CNN. _____ 19

Figura 13: Convoluzione di una immagine 5x5x1 con un kernel 3x3x1 _____ 19

Figura 14: Convoluzione di una immagine RGB utilizzando due kernel tridimensionali _____ 20

Figura 15: Operazione di raggruppamento: Max Pooling e Average Pooling. _____ 21

Figura 16: n1: Primo Hidden layer, n2: Secondo hiddden layer _____ 22

Figura 17: Architettura VGG16 _____ 23

Figura 18: Architettura ResNet50 _____ 24

Figura 19: Struttura di un blocco residuo _____ 26

Figura 20: Struttura e caratteristiche dei blocchi residui. _____ 26

Figura 21: Architettura EfficientNetB0 _____ 28

Figura 22: Funzione di attivazione: Swish _____ 29

Figura 23: Struttura del blocco SENT _____ 30

Figura 24: Architettura EfficientNetB0 _____ 30

Figura 25: Architettura MobileNetV2 _____ 32

Figura 26: Manto stradale privo di buche _____ 32

Figura 27: Manto stradale con buca_____ 33

Figura 28: Installazione ed importazione di librerie necessarie per scaricare e dividere il dataset contenete immagini con o senza la presenza di buche. _____ 36

Figura 29: Sezione di codice utilizzato per effettuare l'operazione di divisione del dataset. ____ 37

Figura 30: Importazione di librerie successivamente utilizzate per la realizzazione dei modelli _ 37

Figura 31: Importazione di librerie necessarie alla realizzazione delle architetture VGG16, ResNet50, EfficientNetB0 e MobileNetV2_____ 38

Figura 32: Prima sezione di codice per effettuare un aumento delle immagini del dataset_____ 38

Figura 33: Seconda sezione di codice per effettuare la data augmentation._____ 39

Figura 34: Caricamento del modello pre-allenato sul file di lavoro _____ 39

Figura 35: Definizione della funzione build_model(base_model)_____ 40

Figura 36: Definizione della funzione utilizzata per il calcolo del Mean Average Precision ____ 41

Figura 37: Definizione della funzione che calcola il Multiply_Accumulate operation (MACs) __ 41

Figura 38: Sezione di codice contenuto nella fase di allenamento _____ 42

Figura 39: Calcolo tempo di inferenza per campione e per singola operazione e calcolo del numero di MAC _____ 44

Figura 40: File di sistema virtuali _____ 44

Figura 41: Sezione di codice per la rappresentazione delle prestazioni del modello nella fase di allenamento e di verifica. _____ 45

Figura 42: Matrice di confusione per una classificazione binaria:_____ 46

Figura 43: Matrice di confusione dei modelli (senza data augmentation) _____ 47

Figura 44: Matrice di confusione dei modelli (con data augmentation) _____ 48

Figura 45: Funzione di costo del modello VGG16 senza data augmentation._____ 49

Figura 46: Funzione di costo del modello ResNet50 senza data augmentation. _____ 49

Figura 47: Funzione di costo del modello EfficientNetB0 senza data augmentation. _____ 50

Figura 48: Funzione di costo del modello MobileNetV2 senza data augmentation. _____ 50

Figura 49: Funzione di costo del modello VGG16 con data augmentation. _____ 51

Figura 50: Funzione di costo del modello ResNet50 con data augmentation. _____ 51

Figura 51: Funzione di costo del modello EfficientNetB0 con data augmentation _____ 52

Figura 52: Funzione di costo del modello MobileNetV2 con data augmentation._____ 52

Figura 53: Tabella dei risultati ottenuti (senza data augmentation). _____ 53

Figura 54: Tabella dei risultati ottenuti (con data augmentation). _____ 53

Figura 55: Quantizzazione dinamica del modello MobileNetV2. _____ 57

Figura 56: Modifiche apportate sul modello per la riduzione della dimensione della rete MobileNetV2_____ 59

Figura 57: Modifiche apportate ai parametri della data augmentation per migliorare le prestazioni del modello MobileNetV2_____ 59

Figura 58: Matrice di confusione del modello MobileNetV2 con immagini di dimensione (64 x 64 x 3) e minor parametri_____ 59

Figura 59: Funzione di costo del modello MobileNetV2 con immagini di dimensione (64 x 64 x 3) e minor parametri _____ 60

Figura 60: Tabella dei risultati ottenuti con il modello MobileNetV2 con immagini di dimensione (64 x 64 x 3) e minor parametri_____ 60

Figura 61: STM32H7S78-DK Discobery kit content _____ 61

Figura 62: Schermata iniziale della piattaforma STM32Cube.AI Developer Cloud dopo aver effettuato l'accesso con un account ST. _____ 63

Figura 63: Caricamento del modello MobileNetV2_64x64_Quantizzazione_dinamica.tflite e inizio nuovo progetto_____ 63

Figura 64: Selezione dei MCUs che contengono il microcontrollore STM32. _____ 64

Figura 65: Salto della quantizzazione (è già stata effettuata sul modello)_____ 64

Figura 66: Avvio della fase di ottimizzazione del modello per la scheda STM32H7S78-DK ___ 65

Figura 67: Passaggio alla pagina dedicata al benchmark_____ 65

Figura 68: Avvio del Benchmark_____ 66

Figura 69: Misura del tempo di inferenza (in millisecondi) del modello applicato alla board STM32H7S78-DK _____ 66

Figura 70: Passaggio ai grafici che rappresentano l'occupazione di memoria e il tempo di inferenza in ogni strato._____ 67

Figura 72: Occupazione della memoria flash in ogni strato del modello_____ 68

Figura 73: Grafico che mostra il tempo di esecuzione in ogni strato della rete. _____ 69

1. INTRODUZIONE

La guida autonoma rappresenta una delle frontiere più attraenti ed impegnative dell'ingegneria moderna. L'obiettivo di sviluppare veicoli in grado di muoversi autonomamente su strada, senza l'intervento umano, richiede lo sviluppo di sistemi sempre più sofisticati e affidabili. Uno degli aspetti fondamentali di questi sistemi è la capacità di percepire l'ambiente circostante in modo accurato e tempestivo.

Il riconoscimento di buche sulla strada assume una elevata importanza, visto che i pertugi sul manto stradale possono rappresentare un pericolo soprattutto per i passeggeri ma anche per i veicoli autonomi o tradizionali a causa di possibili danni alle sospensioni e agli pneumatici.

Questi sistemi sfruttano le reti neurali, più comunemente noti come algoritmi con intelligenza artificiale, per emulare il funzionamento del cervello umano e apprendere da grandi quantità di dati. Le reti neurali convoluzionali (*CNN*), in particolare, si sono dimostrate particolarmente efficaci per compiti di visione artificiale, come il riconoscimento di oggetti su immagini.

Nella tesi si introducono e si analizzano vari tipi di rete neurale, in particolare si farà riferimento alle reti neurali CNN VGG16, ResNet50, EfficientNetB0 e MobileNetV2. Ogni architettura viene, in seguito, valutata sperimentalmente sullo stesso dataset di immagini di strade, alcune contenenti buche e altre prive di difetti.

Si noterà che durante la fase di auto apprendimento, ogni rete imparerà a identificare i pattern visivi caratteristici delle buche, come le variazioni di colore, le ombre e le texture del suolo e al termine di questa fase, sarà in grado di classificare nuove immagini, indicando la presenza o l'assenza di buche con uno specifico grado di accuratezza.

Infine, verranno confrontati i risultati ottenuti dai vari modelli effettuando una valutazione sulle caratteristiche più significative per i sistemi embedded come la memoria occupata dal modello, la sua accuratezza e il tempo di inferenza nella fase di verifica della rete. Ciò permetterà di evidenziare l’architettura più adatta ai sistemi embedded utilizzabili per il rilevamento delle buche.

2. RETI NEURALI

Le reti neurali sono algoritmi di apprendimento automatico che imitano la struttura e il funzionamento delle reti neurali biologiche. Attraverso l'addestramento su grandi quantità di dati, apprendono a identificare caratteristiche, classificare esempi e fare previsioni.

Ogni rete neurale è composta da:

- uno strato di input, che contiene i dati, di una o più classi, da analizzare per il rilevamento e la produzione della soluzione ai problemi contenuti nell'input della rete.
- uno o più strati nascosti
- uno strato di output, che rappresenterà la soluzione di un generico o specifico problema.

Ogni nodo si connette agli altri e possiede il suo peso (w) e la sua soglia associata.

Se l'output di un nodo individuale è superiore al valore di soglia specificato, quel nodo viene attivato, inviando dati allo strato successivo della rete. Altrimenti, nessun dato viene trasmesso allo strato vicino.

In seguito, si valuteranno anche le diverse funzioni di attivazione generalmente utilizzate nei recenti modelli di rete neurale.

In realtà i primi modelli di rete neurale risalgono a più di ottanta anni fa, infatti nel 1943 da due ricercatori, venne pubblicato un articolo "*A Logical Calculus of Ideas Immanent in Neuron Activity*"[1] in cui venne presentato il primo modello di neurone artificiale utilizzato per il calcolo di operazioni complesse, impiegando le proposizioni logiche AND, OR e NOT.

[1] McCulloch e Pitts, «A Logical Calculus of the Ideas Immanent in Nervous Activity».

I due ricercatori che realizzarono il modello riuscirono a dimostrare come fosse possibile costruire reti di neuroni per calcolare qualunque proposizione logica con una architettura semplice e minimale.

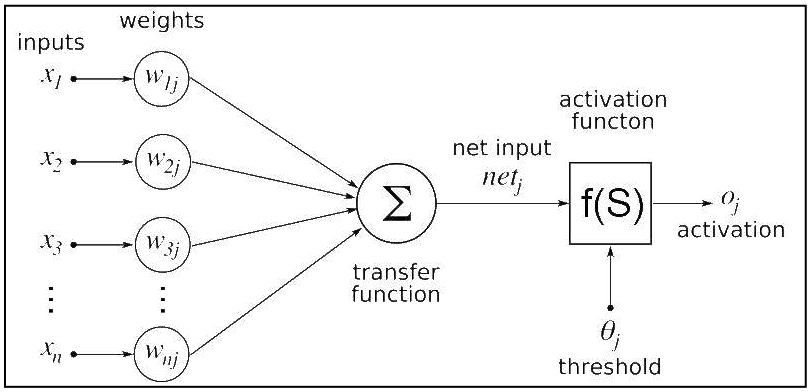

Figura 1:Architettura delle prime reti neurali

Successivamente, dopo aver analizzato e sottolineato la scarsa precisione del primo modello di rete neurale, *Rosenblatt* (*uno psicologo statunitense*) introdusse un nuovo modello nel 1958: il Percettrone[2].

Il modello, pur presentando una struttura relativamente semplice a singolo strato, si distingueva per la sua capacità di apprendere in modo adattivo dai dati, modificando i pesi sinaptici per minimizzare l'errore di predizione.

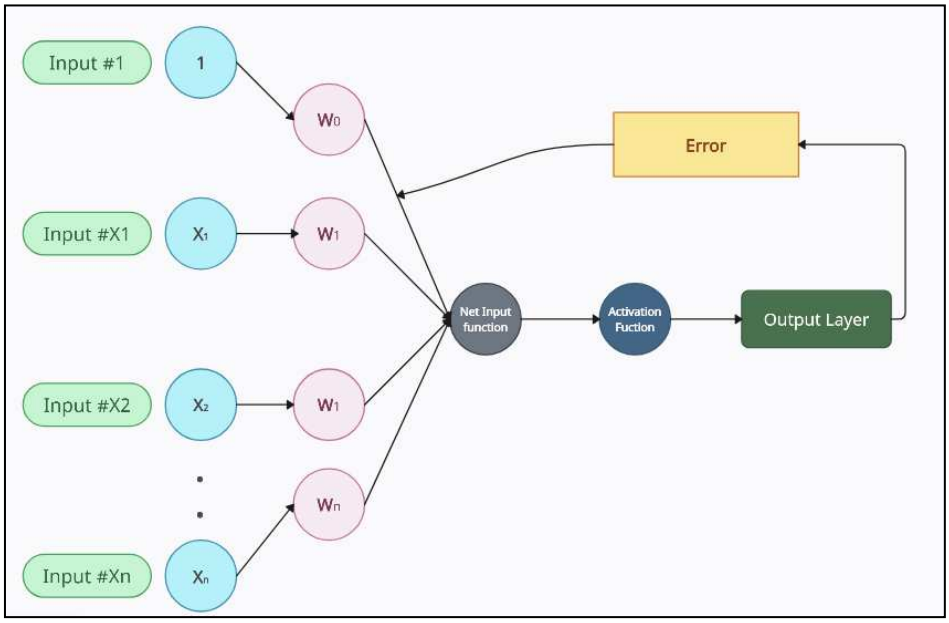

Figura 2:Struttura del percettrone con funzione di apprendimento (error-back propagation)

[2] Rosenblatt, «The perceptron».

La fase di apprendimento, introdotta nel modello di *Rosenblatt*, consiste nella introduzione di una funzione di *error back propagation* (retropropagazione dell'errore) che, in base alla valutazione sull'uscita effettiva della rete rispetto ad un dato ingresso, altera i pesi delle connessioni come differenza tra l'uscita effettiva e quella desiderata.

Le reti neurali moderne possono essere considerate come una generalizzazione e un raffinamento del modello del percettrone. Naturalmente avranno un'architettura che non conterrà un solo neurone, le funzioni di attivazione saranno più sofisticate e gli algoritmi di apprendimento più potenti.

Tutto ciò, ha permesso lo sviluppo di modelli in grado di apprendere rappresentazioni molto più ricche e complesse di dati.

2.1 DEEP NEURAL NETWORKS

Evolvendosi dal percettrone, le reti neurali profonde, grazie all'ampliamento dell'algoritmo di backpropagation e all'aumento del numero di neuroni, hanno permesso di affrontare problemi di crescente complessità, aprendo nuove frontiere nell'apprendimento automatico.

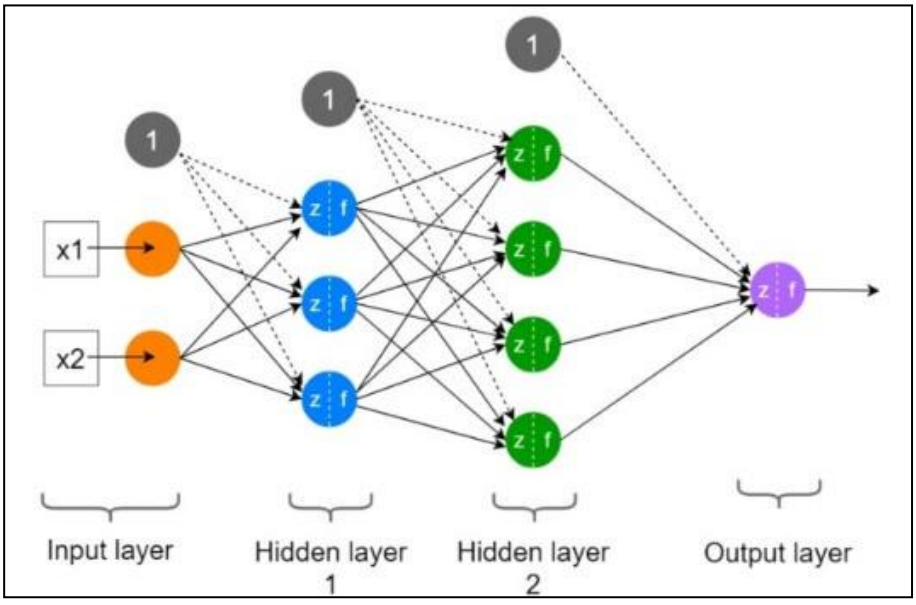

Figura 3:Architettura di una deep neural network

Come si nota in *Figura 3* sono presenti due strati definiti “nascosti”, che rappresentano dei livelli in cui vengono effettuate le regolazioni dei pesi e dei bias associati ad ogni ingresso. La particolarità di questi livelli è che la loro struttura interna non è visibile agli altri strati presenti sulla rete.

Alla rete neurale profonda sarà fornita una serie di input che sarà processata da uno strato nascosto, il quale ha una struttura del tutto analoga a quella del *percettrone* (*Figura 2*) che produrrà una serie di output la quale, a sua volta, potrebbe fungere da ingresso ad uno strato nascosto successivo. Ogni strato nascosto ha uno specifico livello gerarchico.

Utilizzare questa composizione permette di avere vantaggi di:

- Flessibilità: gli strati nascosti rendono le reti molto versatili e adattabili ad una vastità di problemi
- Apprendimento della rete: All’aumento degli strati nascosti si ottiene una maggiore capacità di apprendimento

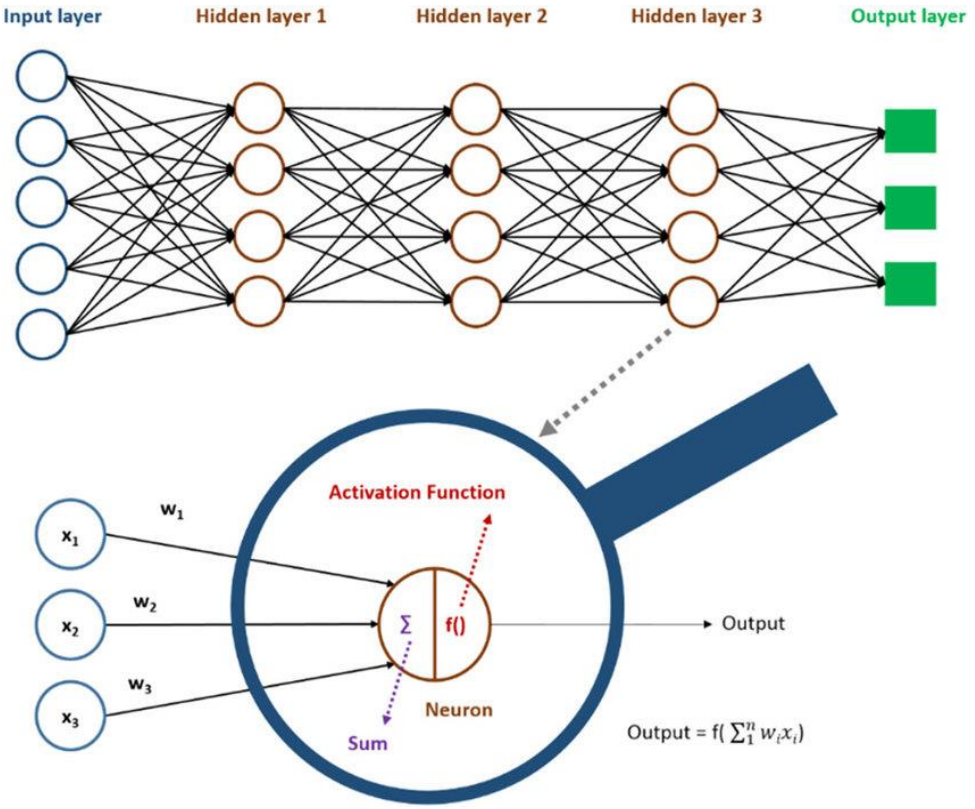

Figura 4:Deep neural network e struttura dello strato nascosto.

2.1.0 Funzione di attivazione

Che sia contenuta in uno strato nascosto o faccia parte di un semplice modello di rete neurale la funzione di attivazione è fondamentale per svolgere operazioni non lineari.

Tale funzione può essere considerata come un filtro in grado di far passare o meno l'informazione in un neurone ad uno successivo.

Di seguito vengono proposte alcune delle funzioni di attivazione più utilizzate:

2.2.1 Sigmoid

La funzione accetta un numero come input e restituisce un numero compreso tra 0 e 1. Questo range si adatta perfettamente ad una classificazione binaria, dove 0 e 1 possono rappresentare rispettivamente le due classi.

È semplice da usare e ha tutte le proprietà desiderabili delle funzioni di attivazione: non linearità, differenziazione continua, monotonia e un intervallo di output impostato.

$$S(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

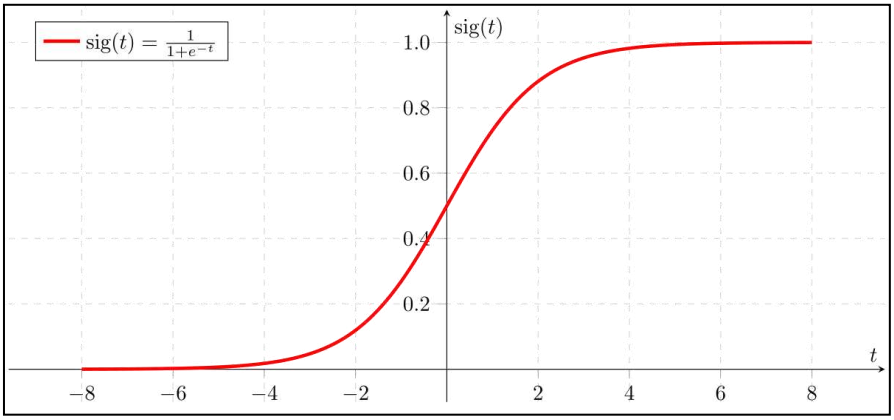

Figura 5: Funzione sigmoide

2.2.2 tanh

La funzione tangente iperbolica comprime un numero con valore reale compreso tra -1 e 1. Non è lineare, ma è diverso da sigmoid e il suo output è centrato sullo zero.

Il vantaggio principale di questa funzione è che i gradienti sono distribuiti più simmetricamente attorno allo zero e sono più ripidi rispetto alla funzione sigmoidea, il che aiuta a limitare in una certa misura il problema del gradiente evanescente.

$$f(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

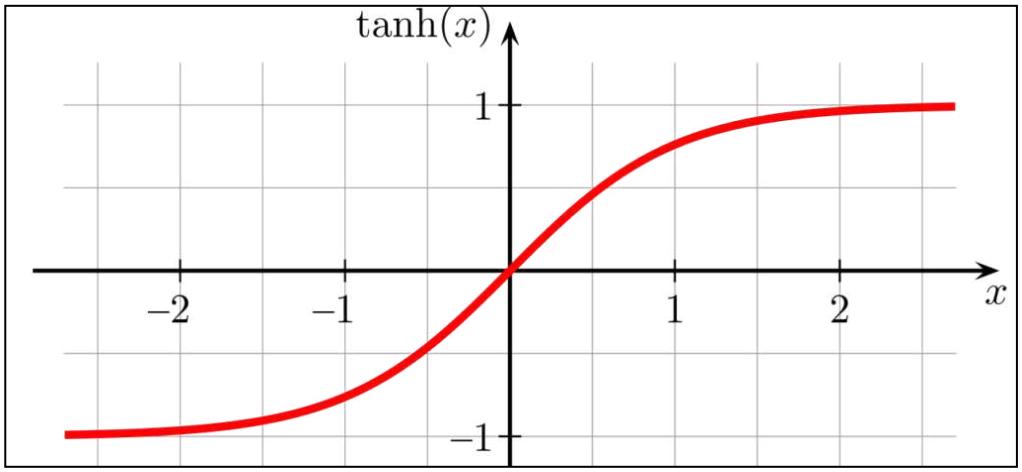

Figura 6: Funzione tanh

2.2.3 ReLU

L'unità lineare rettificata è una delle funzioni di attivazione più utilizzate nelle applicazioni.

L'intervallo di ReLU è compreso tra 0 e infinito e la funzione non satura per valori positivi, il che aiuta a mitigare il problema della fuga del gradiente. Ciò consente un addestramento più rapido ed efficiente delle reti profonde.

$$f(x) = ReLU(x) = x^{+} = \max(0, x) = \frac{x + |x|}{2} = \begin{cases} x \; se \; x > 0 \\ 0 \; se \; x \leq 0 \end{cases}$$

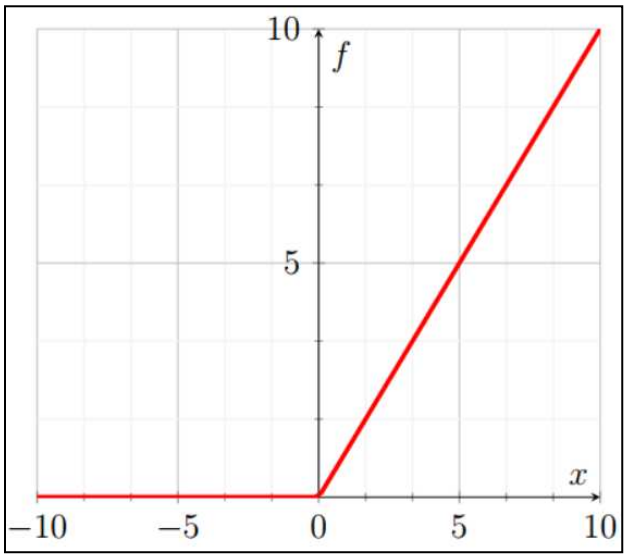

Figura 7:Funzione ReLU

2.2.4 Softmax

La funzione di attivazione Softmax è tipicamente utilizzata nell'ultimo strato di una rete neurale ed è in grado di calcolare la distribuzione di probabilità di un evento considerando tutti i possibili eventi.

$$softmax(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \; per \; j = 1, \ldots\ldots, K.$$

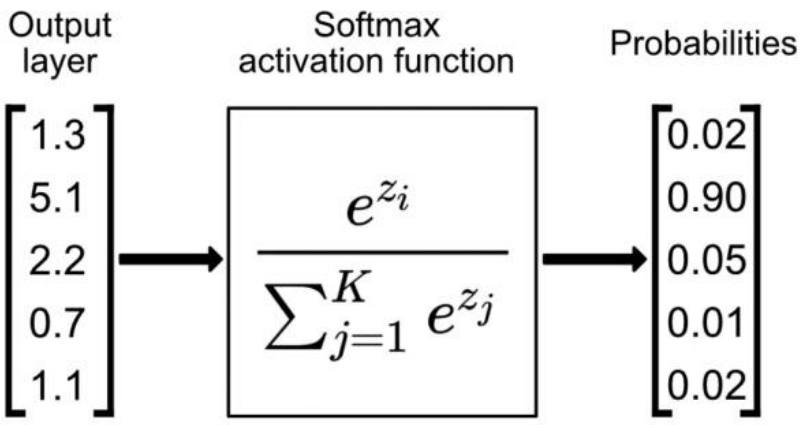

Figura 8:Funzione Softmax

2.2.5 Funzione di costo

La funzione di costo, detta anche funzione di perdita (in inglese *loss function*) permette di valutare la prestazione del modello allenato, valutato ed infine testato.

In pratica rappresenta la distanza tra le previsioni della rete ed i valori reali. Per poter ottenere delle buone prestazioni si dovrà andare a minimizzare tale funzione.

Nel caso interessato, trattandosi di un problema di classificazione, la funzione di costo utilizzata è la *cross-entropy logaritmica* il quale, per una classificazione binaria, ha tale formulazione:

$$C(w) = -\frac{1}{m}\sum_{i=1}^{m}(y_i * \log(\hat{y}_i) + (1 - y_i) * \log(1 - \hat{y}_i)$$

dove con $\boldsymbol{m}$ *si indicano il numero di osservazioni/input contenuti nel dataset*

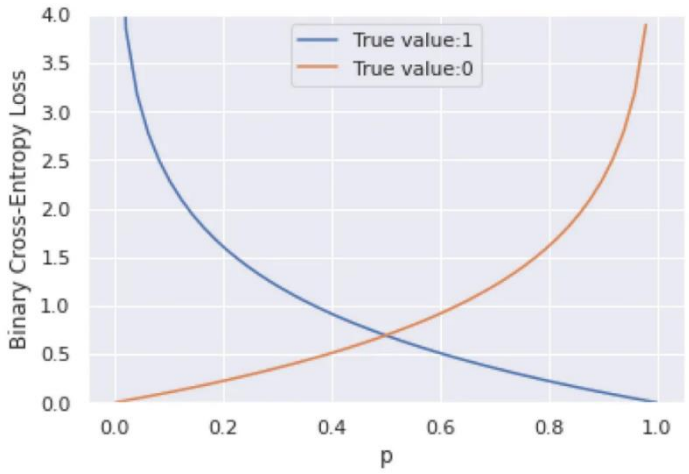

Figura 9: Cross-entropy

Come si nota in *Figura 9*:

- quando il valore previsto è uguale a uno, e corrisponde al vero valore, la cross entropy tenderà a zero.
- quando il valore previsto è uguale a uno, e non corrisponde al vero valore, la cross entropy tenderà ad infinito.

La funzione diminuisce man mano che la probabilità prevista converge all'etichetta effettiva ed in sintesi misura le prestazioni di un modello di classificazione il cui output previsto è un valore di probabilità compreso tra 0 e 1.

2.2.6 Discesa del gradiente

Per minimizzare la funzione di costo può essere utilizzato un algoritmo di ottimizzazione di discesa del gradiente che riesce a trovare dei minimi locali della funzione di costo.

Considerando una generica funzione di attivazione *f :*

$$f : \mathbb{R}^n \to \mathbb{R} \qquad\qquad \nabla f = \begin{matrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{matrix}$$

Visto che il gradiente ∇f indica per definizione la direzione e l'intensità di massima crescita della funzione, l'algoritmo prevede di spostarsi per passi verso il minimo nella direzione opposta $-\nabla f$.

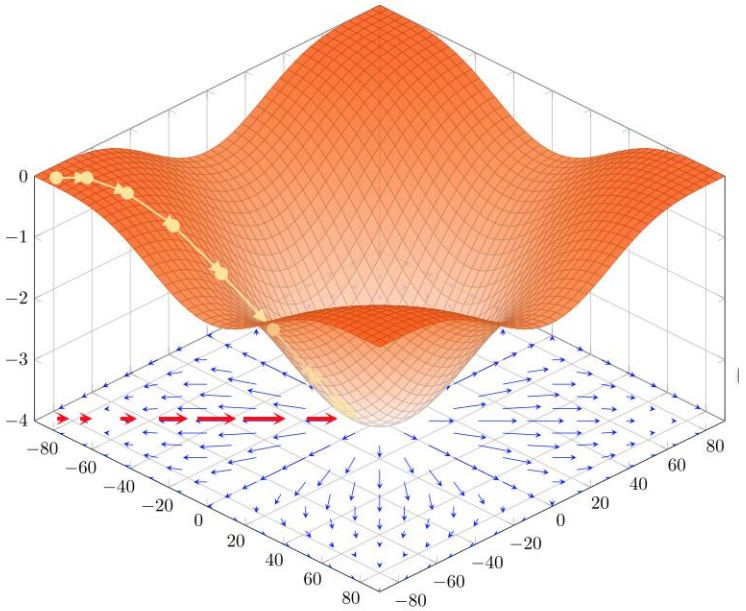

Figura 10: Rappresentazione tridimensionale dell'andamento di discesa alla ricerca di una zona di minimo.

2.2.7 Modello Adams

È presente un altro algoritmo molto utilizzato per l'ottimizzazione della funzione di costo che migliora la discesa del gradiente adattando i tassi di apprendimento.

Durante la fase di allenamento della rete, la struttura Adams verifica quanto è ripida la pendenza della funzione di costo confrontando i valori previsti dalla rete con i valori veri. Nello specifico, effettua il calcolo della pendenza media e anche della velocità media con cui varia l'inclinazione nel tempo della funzione di perdita. L'ultima operazione serve per capire se la pendenza della funzione sta diventando più ripida o più dolce.

Siccome entrambe le operazioni sono mediate nel tempo, ai primi istanti temporali in cui si sta cercando di minimizzare la funzione di costo, l'algoritmo apporta alcune modifiche ai pesi e ai bias.

In sostanza il modello Adams tende a spingere delicatamente la rete nella giusta direzione per migliorarne le prestazioni aiutando così la rete neurale ad apprendere in modo più efficiente ed efficace.

2.2 CONVOLUTIONAL NEURAL NETWORKS

Le reti neurali convoluzionali sono state progettate per realizzare dei modelli in grado di analizzare e processare immagini allo stesso modo in cui viene, in maniera del tutto naturale, eseguito dal nostro cervello.

Questa architettura è utilizzata per molteplici fini, come ad esempio il riconoscimento di immagini e video, l'analisi e la classificazione di immagini e l'elaborazione del linguaggio naturale.

La CNN è un modello di rete neurale profondo che è in grado di acquisire immagini a cui associa determinati pesi e bias in funzione delle caratteristiche (*feature*) presenti in ciascuna immagine.

Dopo aver effettuato una analisi dell'immagine, riesce successivamente a separare gli oggetti identificati per ogni immagine fornita in input della rete.

La separazione permetterà di considerare singolarmente gli oggetti contenuti nelle immagini che, a quel punto, rappresenteranno gli input di una rete neurale profonda in grado di effettuare ad esempio una classificazione.

Questa architettura è stata ispirata dall'organizzazione della corteccia visiva dove i singoli neuroni rispondono agli stimoli solo in una regione ristretta del campo visivo definita come campo recettivo. Una serie di tali campi si sovrappongono per coprire l'intera area visiva.

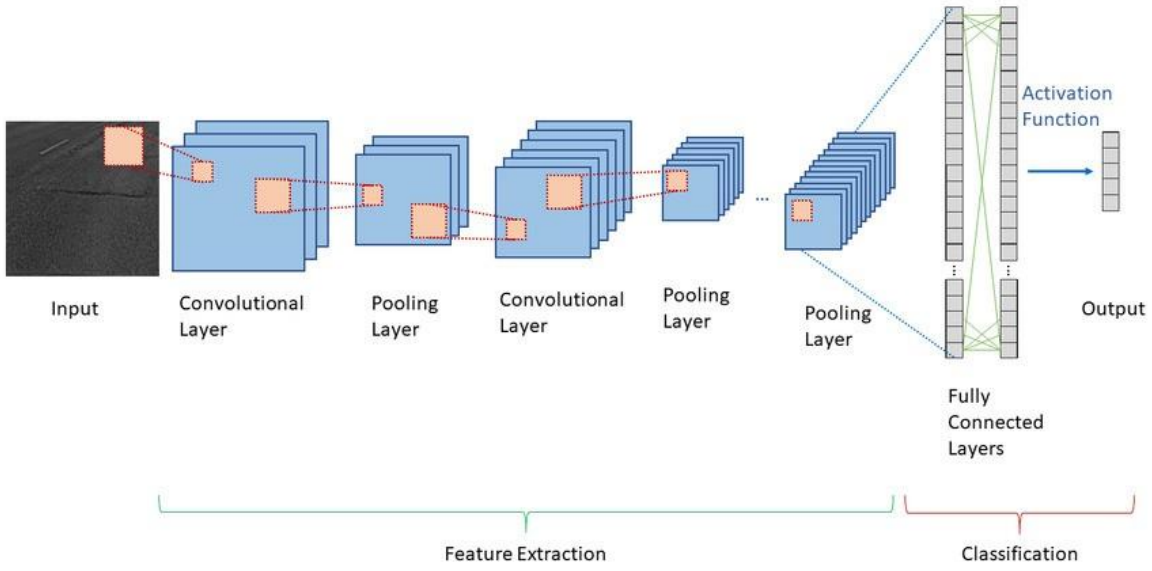

Figura 11: esempio di rete neurale convoluzionale

2.5.1 Input image

Generalmente le immagini contenute nel dataset sono immagini a colori (RGB) e vengono fornite alla rete CNN come delle matrici-tensori numeriche. L'immagine RGB ha tre canali: il rosso, il verde e il blu e ogni canale viene rappresentato da una matrice separata che contiene i valori numerici associati alla tonalità del colore principale in ogni cella della matrice stessa.

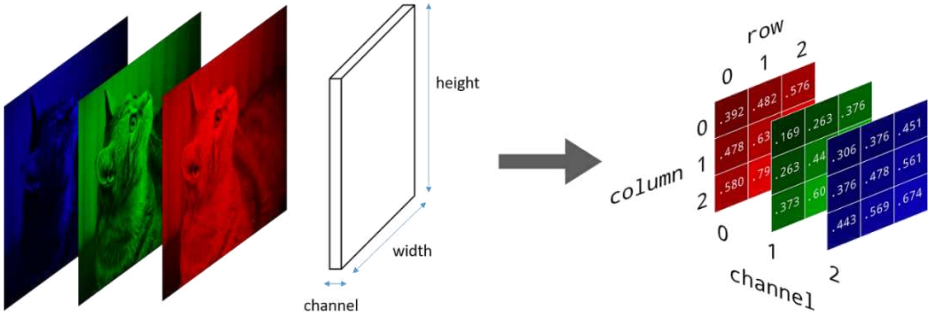

Figura 12: Acquisizione dell'immagine per una rete CNN.

Un aspetto che viene subito notato riguarda la dimensione dell'immagine. Infatti, all'aumento della dimensione delle immagini in ingresso della rete corrisponde un aumento del costo computazionale, perciò tipicamente la prima operazione che si effettua è legata alla riduzione della dimensione delle immagini in modo da rendere più efficiente il processo.

Ciò però comporta una perdita delle feauture della immagine che dunque dovrà essere limitata.

2.5.2 Strato Convolutivo

Per comprendere la funzione di questo strato è conveniente mostrare un semplice esempio:

Si supponga di avere una immagine di dimensione 5 (altezza) x 5 (lunghezza) x 1 (canale), dove l'unita di misura per altezza e lunghezza corrisponde al numero di pixel contenuti nelle due coordinate.

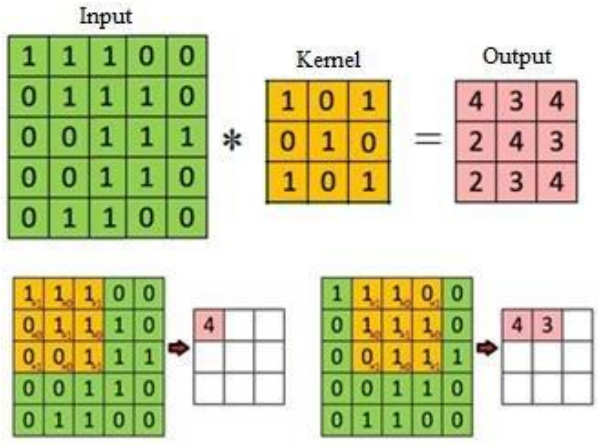

Figura 13:Convoluzione di una immagine 5x5x1 con un kernel 3x3x1

La matrice, che rappresenta l'immagine di input, viene convoluta con una matrice 3 x 3 che prende il nome di kernel. Il kernel è un filtro numerico di dimensioni ridotte, rispetto alla dimensione della matrice di input, in grado di estrarre specifiche caratteristiche dall'immagine.

L'operazione di convoluzione, in sostanza, consiste in una *moltiplicazione* elemento per elemento tra una regione della immagine e il kernel. I risultati delle moltiplicazioni vengono *sommati* ottenendo così un valore che viene inserito in una cella della matrice di output di dimensione 3 x 3. Iterando il processo affinché il kernel sia traslato su tutta l'immagine di input si ottiene il completamento della matrice di output, il quale corrisponde al termine della fase di convoluzione.

Il processo visto in questo esempio può essere generalizzato al caso in cui si hanno come immagini di input, immagini RGB. In tal caso, il Kernel ha la stessa profondità dell'immagine di input (quindi pari a tre).

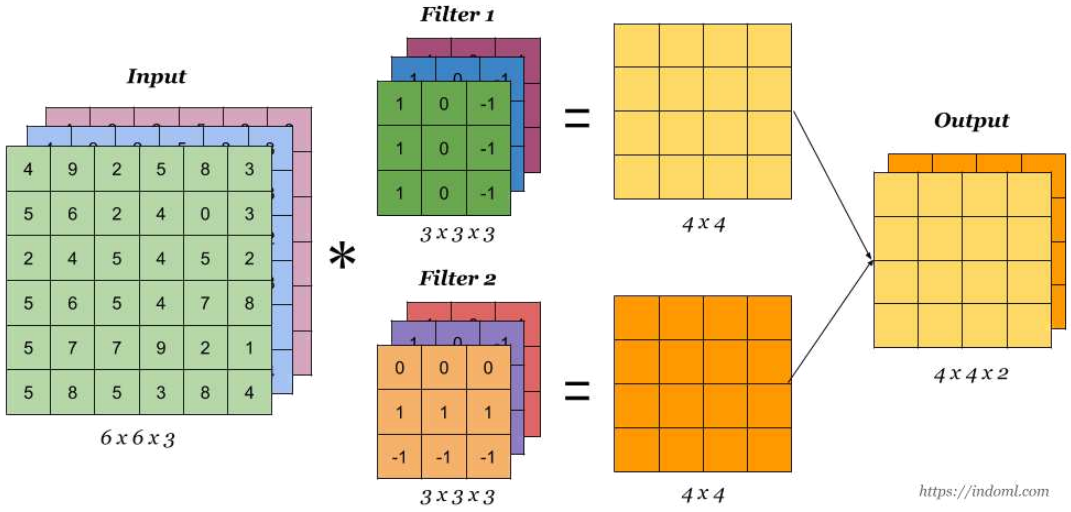

Figura 14: Convoluzione di una immagine RGB utilizzando due kernel tridimensionali

L'obiettivo del primo strato convolutivo è quello di estrarre le proprietà di basso livello dell'immagine come i bordi, colori ed orientamento del gradiente. Con l'aggiunta di successivi strati convolutivi si è in grado di estrarre le proprietà di alto livello realizzando così un modello di rete che riesce ad analizzare le immagini minuziosamente.

2.5.3 Livello di raggruppamento

Questo strato viene utilizzato per ridurre la dimensione spaziale della matrice/tensore di output ottenuta/o dalla operazione di convoluzione. Il suo scopo è quello di ridurre la potenza computazionale, richiesta per processare i dati, andando ad estrarre dalla matrice le caratteristiche dominanti che non dipendono dalla rotazione e dalla posizione in cui sono situate.

Ci sono due tipi di raggruppamento:

- Max Pooling, che fornisce il valore massimo contenuto in una sezione della matrice dello strato convolutivo.
- Average Pooling, che fornisce il valore medio contenuto in una sezione della matrice dello strato convolutivo.

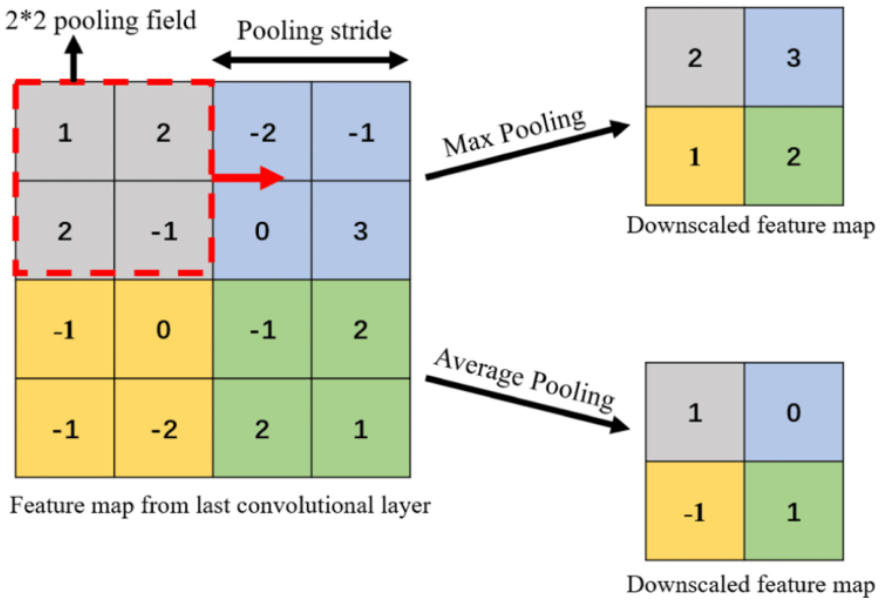

Figura 15: Operazione di raggruppamento: Max Pooling e Average Pooling.

Il livello convolutivo e il livello di raggruppamento formano i-esimi strati di reti neurali convoluzionali che vengono processati in cascata. In funzione della complessità e della dimensione dalla immagine iniziale il numero di strati i-esimi può essere aumentato in modo da migliorare la cattura di caratteristiche di alto livello a costo però di una maggior richiesta di potenza computazionale.

Esistono modelli di CNN che effettuano un compromesso tra la profondità di caratteristiche di immagine rilevate, la potenza computazionale, il tempo di inferenza e la accuratezza.

2.5.4 Fully Connected Layer

Lo strato fully connected (FC), conosciuto anche come strato denso, viene utilizzato in tutti i modelli di rete convoluzionali e rappresenta una delle ultime operazioni di elaborazione dei dati.

Questo strato ha un organismo in cui ogni neurone o nodo del livello precedente è collegato a ciascun neurone del livello corrente, ecco dunque spiegato il motivo del nome completamente connesso.

Gli output dello strato rappresenteranno i valori di uscita della rete che saranno utilizzati sia per la fase di allenamento, per regolare i pesi e i bias che per la fase di normale utilizzo, per effettuare la classificazione in real-time

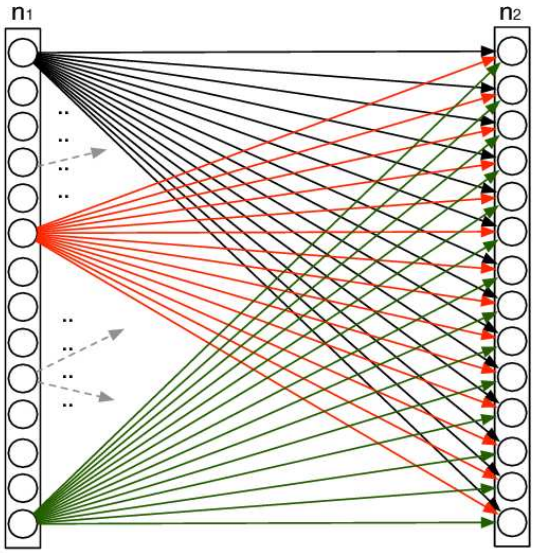

Figura 16: n1: Primo Hidden layer, n2: Secondo hiddden layer

2.3 STATO DELL'ARTE DELLE CONVOLUTIONAL NEURAL NETWORKS

2.6.1 Visual Geometry Group 16

Questo modello di rete neurale convoluzionale (*VGG16*)[3], rispetto al modello tradizionale precedentemente descritto, ha una profondità di livelli nascosti maggiore dovuta alla architettura contenuta dei kernel (3 x 3). Utilizzando i medesimi filtri convolutivi, si otterrà un aumento dell'accuratezza del modello.

Pur migliorando la precisione della rete, avendo aumentato i livelli degli strati nascosti, è probabile che il costo computazionale complessivo sia aumentato rispetto ad un modello con meno livelli nascosti.

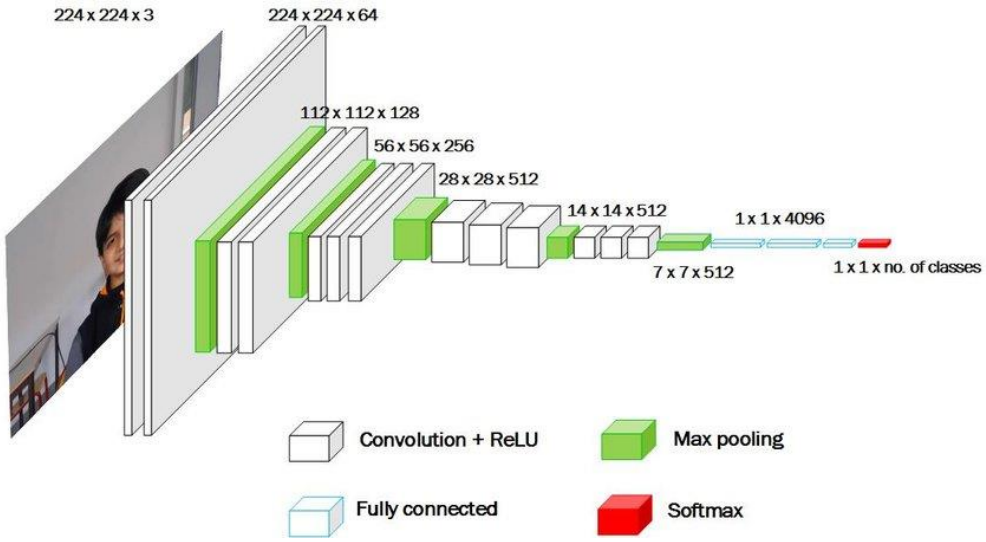

Figura 17: Architettura VGG16

Come si vede in *Figura 17* l'architettura *VGG16* è composta da tredici livelli convoluzionali e cinque livelli in cui viene effettuata l'operazione di *Max Pooling*. Il sedici contenuto nel nome della architettura è riferito al numero di strati che possiedono i pesi.
L'immagine di input deve essere RGB e deve avere una dimensione al massimo pari a 224 x 224 x 3.

Il primo livello Convolutivo ha 64 filtri, il secondo ne ha 128, il terzo ne ha 256 e gli ultimi due ne hanno 512. Dopo l'ultima operazione di raggruppamento, ci sono tre strati *Fully-Connected*: i primi due hanno 4096 canali ciascuno mentre il terzo ha 1000 canali (uno per ogni classe). Infine, l'ultimo strato della architettura è lo strato soft-max.

[3] Simonyan e Zisserman, «Very Deep Convolutional Networks for Large-Scale Image Recognition».

2.6.2 ResNet50

Nelle reti neurali profonde tradizionali o semplici, si verifica il problema del gradiente evanescente, ovvero il fenomeno in cui i gradienti diventano molto piccoli quando vengono propagati attraverso molti strati di una rete neurale. Ciò significa che man mano che aumenta il numero dei livelli, l'errore di training peggiora e la rete ha un apprendimento più lento o persino una completa mancanza di apprendimento in architetture molto profonde.

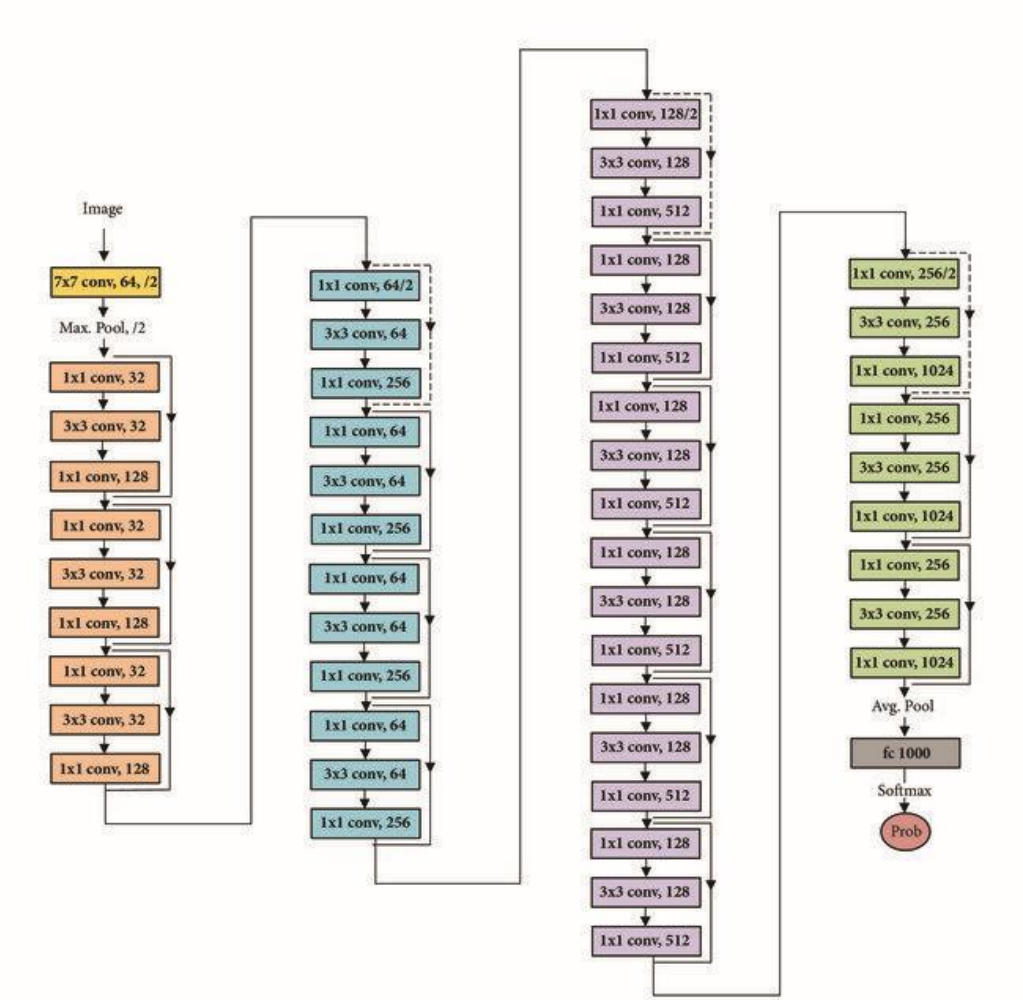

Figura 18: Architettura ResNet50

Le architetture *ResNet*[4] risolvono questo problema introducendo connessioni di salto, note anche come connessioni residue, che consentono al gradiente di bypassare determinati livelli nella rete.

[4] He et al., «Deep Residual Learning for Image Recognition».

Il primo livello della rete è uno strato convolutivo che effettua una convoluzione dell'immagine di input con dimensioni massime pari a (224 x 224 x 3) producendo una feature map di dimensione (112 x 112 x 3). In questo strato sono utilizzati 64 filtri kernel di dimensione (7 x 7) e con uno stride, ovvero uno spostamento del filtro per ogni operazione convolutiva, di (2 x 2) per effettuare un sotto campionamento di un fattore 2 della immagine.

Nello specifico, successivamente alla operazione convolutiva, è presente una operazione di *normalizzazione batch* utilizzata per effettuare una normalizzazione delle attivazioni.

Tale operazione calcola la media e la varianza delle attivazioni per ogni canale di una feature-map e le normalizza. Viene utilizzata per stabilizzare l'addestramento, rendendolo meno sensibile alla inizializzazione dei pesi, ma anche più rapido, alla convergenza e alla prevenzione dell'*overfitting*.

In seguito alla normalizzazione è presente la funzione di attivazione *ReLU* che introduce il fenomeno di non linearità della rete. Come ultima fase del primo stadio convolutivo c'è un raggruppamento *MaxPooling* che viene effettuato con una matrice (3 x 3) ed uno stride (2 x2).

Il risultato prodotto da questo strato è una feature-map di dimensioni (56 x 56 x 64) che fungerà da input per i successivi blocchi convoluzionali dell'architettura *ResNet50*.

2.6.3 Blocchi residui

Nei livelli successivi, come si nota in *Figura 18* riferita alla architettura *ResNet50*, si osservano delle connessioni di salto tra i blocchi convolutivi.

Coloro rappresentano l'elemento chiave di questo tipo di struttura perché vengono utilizzate per eseguire una somma tra il valore $\boldsymbol{x}$ di input del blocco e il valore $\boldsymbol{F(x)}$ di uscita del blocco, che esegue la *normalizzazione batch*.

La somma tra i due valori permette ai gradienti di fluire più liberamente nella rete, il che evita di avere dei gradienti che tendono a zero.

L’input del blocco residuo viene collegato all’output di questi livelli convolutivi prima dell’applicazione della funzione di attivazione finale.

Questo processo garantisce alla rete di apprendere sia la funzione identità che qualsiasi altra trasformazione necessaria per minimizzare la perdita complessiva, migliorando la capacità di generalizzare e apprendere pattern complessi.

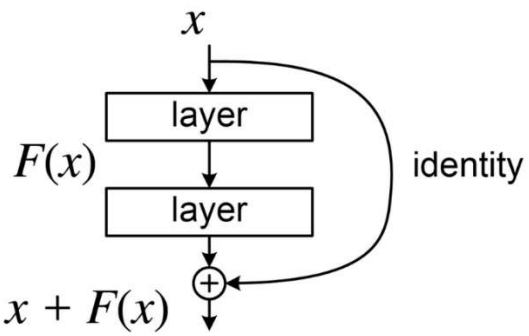

Figura 19: Struttura di un blocco residuo

Una variante della suddetta struttura è chiamata *Bottleneck Residual Block* e viene utilizzata nelle reti neurali profonde per migliorare l’efficienza e ridurre la complessità computazionale.

Il termine *Bottleneck* (collo di bottiglia) è riferito alla struttura del blocco residuo in quanto è presente una riduzione del numero di canali nel tensore di ingresso nella fase precedente della convoluzione. Successivamente viene ripristinato il numero di canali.

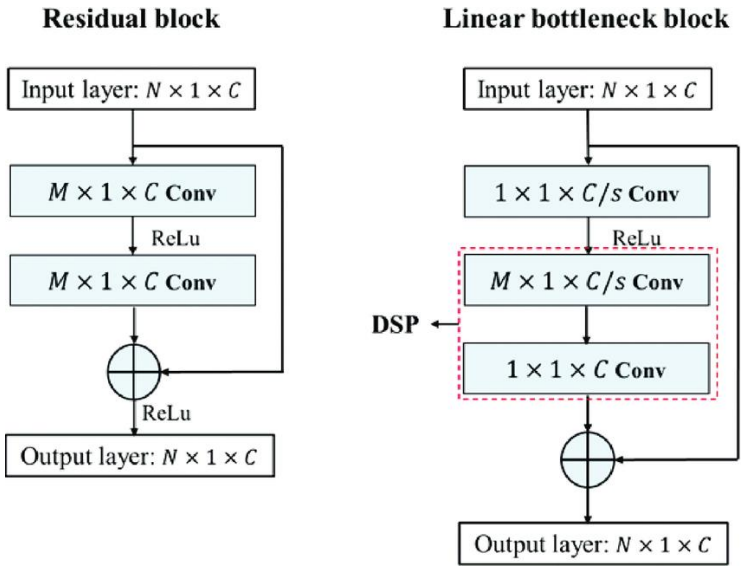

Figura 20: Struttura e caratteristiche dei blocchi residui.

La struttura *Bottleneck* riduce in modo significativo il costo computazionale della convoluzione se comparato con la struttura base, soprattutto se ci sono un elevato numero di canali.

Dopo l'ultima operazione di raggruppamento, eseguita successivamente ai quattro stadi convolutivi che utilizzano i blocchi residui, si passa allo strato *Fully-Connected* (FC) che ha mille canali. Infine, l'ultimo strato della architettura è lo strato che contiene una funzione di attivazione soft-max.

2.6.4 EfficientNetB0

L'*EfficientNet*[5] è una delle architetture di rete neurale *CNN* più recenti e performanti in grado di ottenere un ottimo equilibrio tra l'accuratezza e la complessità computazionale.

Rispetto alla *ResNet*, ha un numero minore di parametri e ciò significa che, a parità di ambienti, con un modello *EfficientNet* si ha una struttura più compatta che minimizza l'occupazione di memoria.

Come prima operazione del modello è presente una operazione di convoluzione (3 x 3) sull'immagine in ingresso alla rete di dimensioni al massimo pari a (224 x 224 x 3).

In seguito, sono presenti sedici blocchi convolutivi invertiti, ognuno dei quali include il *Bottleneck residual block*.

Infine, dopo aver effettuato anche un'ultima operazione di convoluzione (1x1) seguita da un *Average Pooling*, si ottengono i risultati della classificazione prodotti dallo strato *Fully Connected*.

[5] Tan e Le, «EfficientNet».

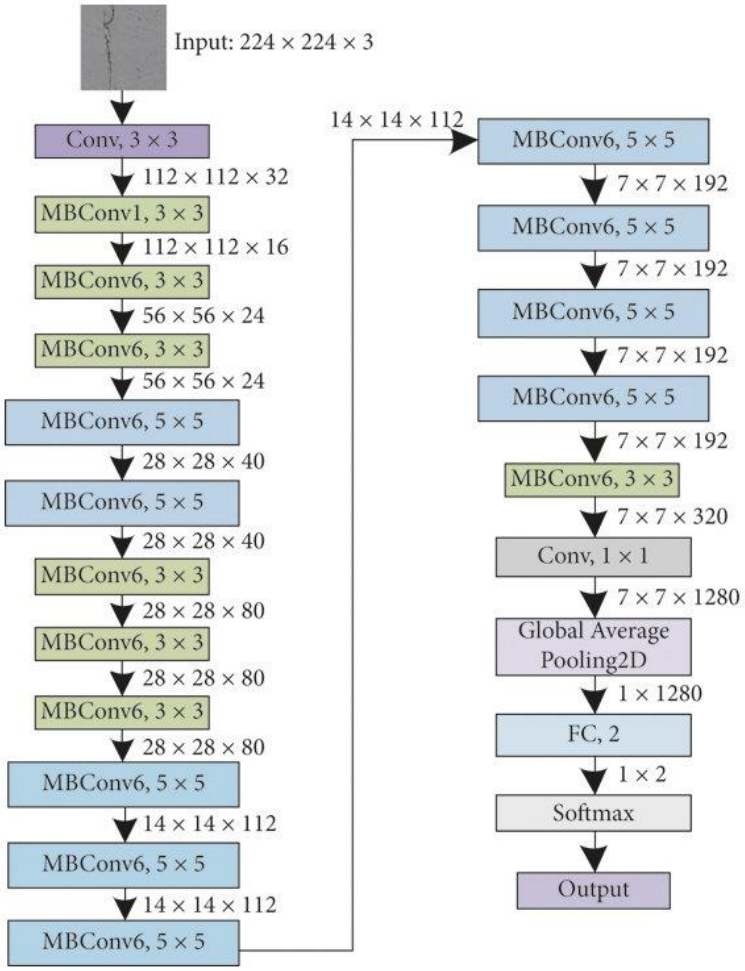

Figura 21: Architettura EfficientNetB0

Dopo ciascuna operazione di convoluzione viene effettuata una operazione di *normalizzazione batch*.

La funzione di attivazione utilizzata dalla *EfficientNetB0* è chiamata *Swish* ed offre una maggiore flessibilità, rispetto alla funzione *ReLU*, apprendendo sia operazioni lineari che non lineari.

$$Swish\ (x) = x * \frac{1}{1 + e^{-\beta x}} = x * sigmoid(\beta x)$$

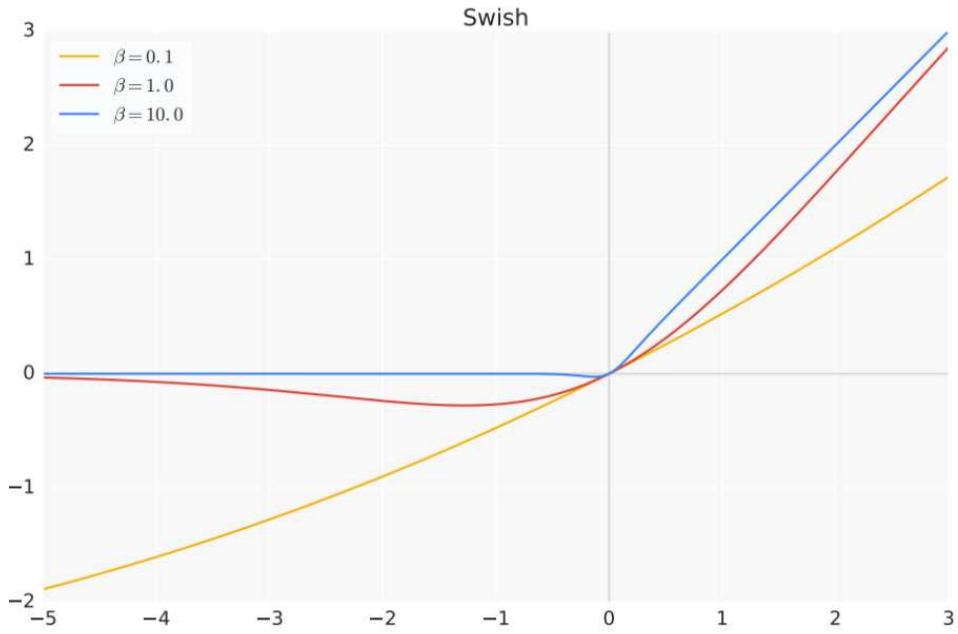

Figura 22: Funzione di attivazione: Swish

Come si vede in Figura 22, la funzione *Swish* essendo continua e derivabile ovunque, facilita l'ottimizzazione tramite la *backpropagation*.

Il parametro β controlla la pendenza della curva, valutando i casi limite si nota che:

- Se β tende ad infinito, allora la funzione è equivalente alla funzione gradino.
- Se β tende a zero, allora la funzione è equivalente alla funzione lineare.

I sedici blocchi convolutivi che si notano nella *Figura 21* sono moduli in cui vi è contenuto un blocco residuo inverso (*MBConv*).

Tale blocco prima di effettuare una convoluzione (3 x 3) o (5 x 5) effettua un aumento di dimensione della immagine applicando una convoluzione (1 x 1) per estrarre più informazioni dall'immagine.

Dopo tali convoluzioni è presente una operazione chiamata *Squeeze-and-Excitation* che ha il compito di migliorare le performance del modello.

Infine, viene eseguita una ultima convoluzione (1 x 1) per andare a ridurre la dimensione dell'immagine.

In particolare, lo strato *Squeeze-and-Excitation (SENT)* comprime la feature map effettuando un average pooling in direzione della dimensione del canale ed esegue una operazione di eccitazione sulla feature globale per apprendere le relazioni in ciascun canale.

In tal modo possono essere determinati i pesi dei diversi canali, tramite una funzione di attivazione *sigmoid*, che saranno moltiplicati nella feature map originale per ottenerne una finale.

Pertanto, lo scopo del blocco SENT è quello di prestare maggiore attenzione alla feature del canale che hanno più informazioni andando così a sopprimere quelle che non sono importanti.

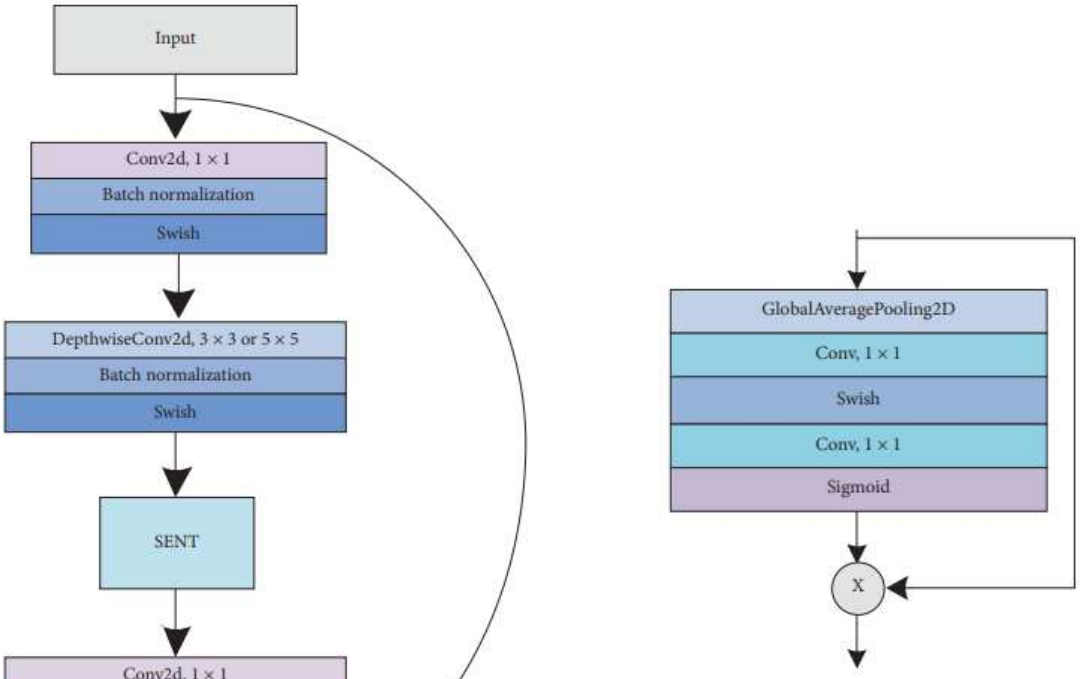

Figura 23: Struttura del blocco SENT

Figura 24: Architettura EfficientNetB0

2.6.5 MobileNetV2

La *MobileNetV2* [6] è una architettura CNN ottimizzata per applicazioni visive su sistemi mobile e sistemi embedded. Tale rete viene implementata anche per operazioni di classificazione delle immagini e di rilevamento degli oggetti.

Il modello prende in ingresso una immagine RGB di dimensione al massimo pari a (224 x 224) e la trasporta al primo strato convolutivo dove è eseguita una convoluzione con passo pari a due per sotto-campionare l'immagine. Questa operazione produce un numero di canali pari a 32.

Lo strato successivo è un blocco residuo invertito che, come visto anche nella precedente architettura, contiene tre principali funzioni:

- Una convoluzione (*Pointwise Convolution*) (1x1) che aumenta il numero di canali complessivi seguita da una funzione di attivazione *ReLU*.
- Una convoluzione di profondità (*Deeptwise Convolution*) che effettua una convoluzione spaziale in modo indipendente su ciascun canale seguita da una funzione di attivazione *ReLU*.
- Una convoluzione (*Pointwise Convolution*) (1 x 1) che proietta i canali espansi in una dimensione inferiore. Questa volta è sufficiente utilizzare su tale strato una *funzione di attivazione lineare*.

Ogni blocco residuo invertito ha una connessione che salta la convoluzione di profondità e si collega direttamente dall'input all'output, consentendo un flusso di gradiente migliore durante l'addestramento.

Questa connessione esiste solo quando le dimensioni di input e output corrispondono.

[6] Sandler et al., «MobileNetV2».

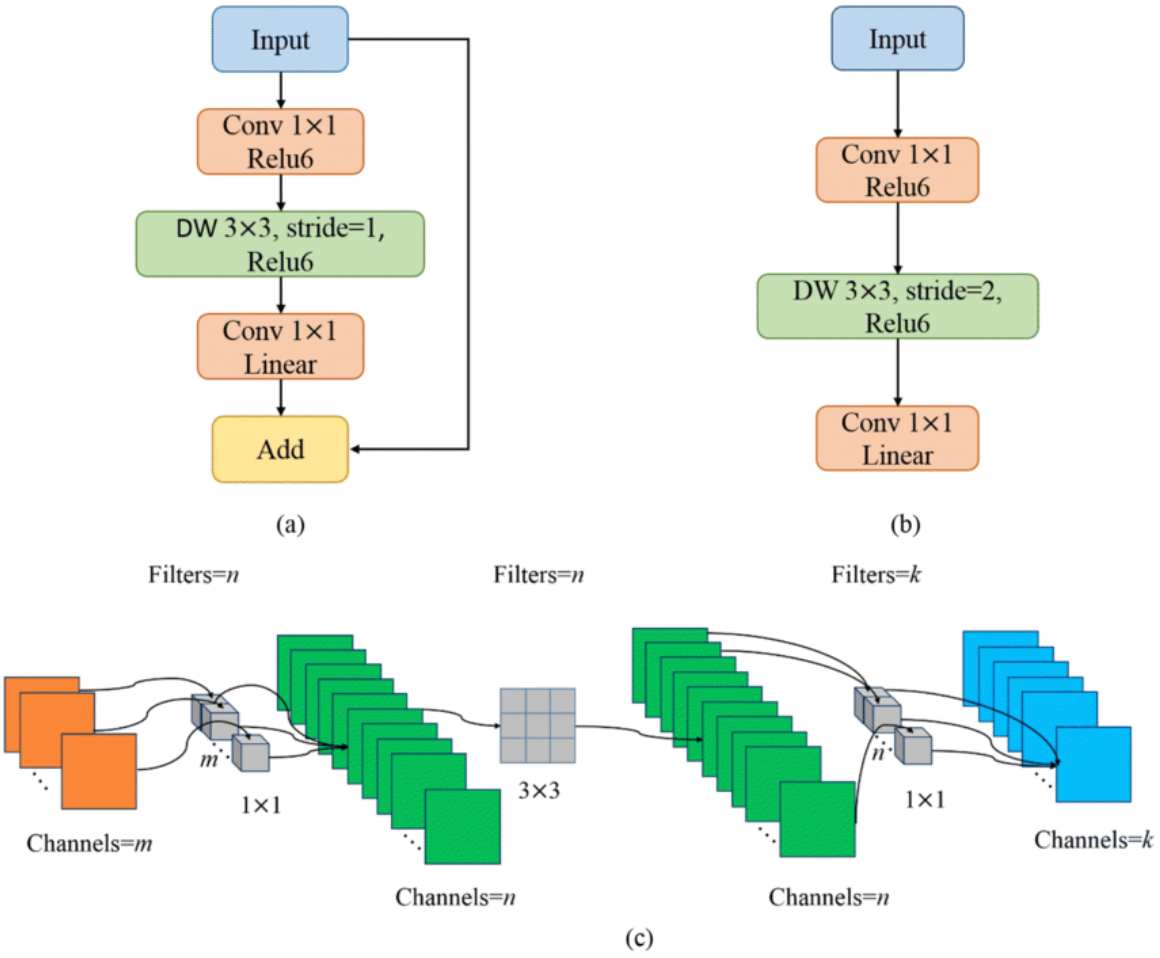

Figura 25: Architettura MobileNetV2

- **a)** *Blocco residuo invertito con una convoluzione di profondità con passo 2*
- **b)** *Blocco residuo invertito con una convoluzione di profondità con passo unitario*
- **c)** *Architettura CNN MobileNetV2*

3. Caso di studio

3.1 Classificazione Binaria

Per il rilevamento delle buche presenti nel manto stradale può essere utilizzata una classificazione binaria.

Questa classificazione verrà effettuata sull'ultimo strato della rete neurale convoluzionale e, in funzione del tipo di immagine in ingresso, la rete neurale

dovrà valutare se l'immagine fa parte dell'insieme di immagini in cui non ci sono buche o con quelle in cui ci sono.

Generalmente un algoritmo di classificazione binaria impara a identificare le caratteristiche distintive delle due classi attraverso l'analisi di un insieme di dati di addestramento.

Una volta addestrato, il modello può essere utilizzato per prevedere la classe a cui appartiene un nuovo dato.

Figura 26: Manto stradale privo di buche

Figura 27: Manto stradale con buca

3.2 DATASET DI IMMAGINI

Il dataset di immagini utilizzato per la realizzazione delle reti neurali è stato trovato su Kaggle, una nota piattaforma online di dataset e può essere valutato o scaricato tramite il seguente collegamento:

https://www.kaggle.com/datasets/andrewmvd/pothole-detection?select=annotations

Esso è composto da due file directory:

- Normal: che contiene 353 immagini di diverse dimensioni in formato JPG in cui non sono presenti buche.
- Potholes: che contiene 329 immagini di diverse dimensioni in formato JPG in cui sono presenti buche.

3.3 DATA AUGMENTATION

Data la scarsa profondità del dataset di immagini, per poter migliorare le prestazioni dei modelli di rete neurale può essere utile applicare delle modifiche alla struttura di input.

In pratica viene introdotto un algoritmo che effettua delle trasformazioni ad ogni immagine di input in modo da espandere il dataset di almeno dieci volte.

L'aumento dei dati in ingresso alla rete è dovuto principalmente dall'utilizzo di trasformazioni geometriche e dal cambio di tonalità dei colori dell'immagine.

Tali operazioni permettono anche di prevenire il fenomeno dell'*Overfitting*, ovvero il problema che si riscontra quando una rete neurale si adatta troppo bene ai dati di addestramento, imparando a memoria il rumore o altre particolarità irrilevanti contenute nelle immagini.

Ciò risulta essere un problema perché si produce una perdita nella capacità di generalizzazione delle conoscenze a nuovi dati, il che abbatte l'accuratezza in fase di verifica e di test della rete stessa.

Dunque, una espansione delle immagini del set di allenamento permette alla rete di effettuare in modo efficace delle generalizzazioni che riducono la memorizzazione di specifiche e non significative proprietà delle immagini.

3.3.1 Operazioni geometriche eseguite sulle immagini del dataset

- Rotazione dell'immagine: si ruota ciascuna immagine più volte con diverse angolazioni
- Ridimensionamento: si modifica la dimensione della immagine
- Traslazione: si effettua una traslazione sull'immagine che può essere orizzontale o verticale
- Capovolgimento: L'immagine viene soggetta ad una inversione speculare.

3.3.2 Trasformazioni fotometriche

- Regolazioni di luminosità e di contrasto
- Tecniche di jittering del colore

3.4 GOOGLE COLAB, TENSORFLOW E KERAS

Google Colaboratory [7]è un ambiente di sviluppo interattivo basato sul cloud ed è ottimizzato per l'esecuzione di codice Python. Tale codice è diretto verso il campo del machine learning e del deep learning.

Uno dei principali motivi per il quale risulta essere un ambiente di sviluppo molto utilizzato è l'accesso gratuito a risorse computazionali significative.

Questo permette a molti ricercatori e a sviluppatori di sperimentare modelli complessi senza la necessità di configurare ambienti di sviluppo locali. Inoltre, l'integrazione con Google Drive permette di caricare e salvare i vari modelli, il che rende la gestione dei progetti semplice ed intuitiva.

Un altro vantaggio della piattaforma è che si integra perfettamente con i framework più popolari per il deep learning, tra cui:

- *TensorFlow*[8]*:*

  un framework open-source sviluppato da Google, che è diventato uno strumento molto utile data la sua flessibilità, scalabilità e la vasta comunità che lo supporta.
- *Keras*[9]:

  è un insieme di regole e specifiche che definiscono come diverse parti di un software possono interagire tra loro, scritto in Python. È progettato per semplificare la creazione e l'addestramento di reti neurali profonde. La sua sintassi intuitiva e la sua flessibilità l'hanno resa uno dei framework più popolari nel campo del deep learning.

[7] «Google Colab».
[8] «TensorFlow».
[9] «Keras: Deep Learning for humans».

3.5 MODELLO UTILIZZATO PER LA CLASSIFICAZIONE

3.5.1 Caricamento e divisione del dataset

Il primo passo nell'ambiente Google Colab è scaricare ed importare librerie necessarie per la realizzazione e l'utilizzo di funzioni sia per la creazione del modello che per la sua valutazione.

```
!pip install split-folders
# installazione della libreria esterna split-folders che contiene funzioni per
# dividere il dataset in sottoinsiemi utilizzati separatamente nella fase di
# training e di validation
from glob import glob
# importazione della libreria glob necessaria per trovare file e cartelle
from PIL import Image
# importazione della libreria PIL necessaria per aprire e manipolare immagini
import kagglehub
# importazione libreria kagglehub necessaria per caricare il dataset sul modello
# direttamente dalla piattaforma Kaggle.
import splitfolders
# importazione libreria split-folders
import time
# importazione libreria time necessaria per lavorare con funzioni nel tempo
# in particolare viene utilizzata per calcolare il tempo di inferenza
```

Figura 28:Installazione ed importazione di librerie necessarie per scaricare e dividere il dataset contenete immagini con o senza la presenza di buche.

Per poter visualizzare ed effettuare operazioni sulle immagini del dataset è necessario importare la libreria kagglehub. Questa operazione è necessaria solo se il dataset è conteuto nella piattaforma online Kaggle.

Successivamente si carica sul blocco di lavoro il dataset e con una ulteriore operazione, tramite la libreria split-folders, si divide il dataset in due cartelle principali.

Entrambe conterranno le immagini dei manti stradali suddivisi in una cartella Normal e in una di nome Potholes, a seconda delle caratteristiche delle catture:

- **Training**
  - Normal
  - Potholes
- **Validation**
  - Normal
  - Potholes

```
#Download del dataset contenente immagini normali e immagini con buche
path = kagglehub.dataset_download("atulyakumar98/pothole-detection-dataset")
print("Path to dataset files:", path)

# Divisione del dataset in cartelle per il  Train-Test-Validation
splitfolders.ratio(
  "/root/.cache/kagglehub/datasets/atulyakumar98/pothole-detection-dataset/versions/4",
  "/root/.cache/kagglehub/datasets/atulyakumar98/pothole-detection-dataset/versions/4_split",
  seed=1337, ratio=(.8, .2), group_prefix=None)
# Paths del dataset
train_path = '/root/.cache/kagglehub/datasets/atulyakumar98/pothole-detection-dataset/versions/4_split/train'
val_path = '/root/.cache/kagglehub/datasets/atulyakumar98/pothole-detection-dataset/versions/4_split/val'
```

Figura 29: Sezione di codice utilizzato per effettuare l'operazione di divisione del dataset.

Per poter effettuare una valutazione delle prestazioni dei modelli VGG16, Resnet50, EfficientNetB0, MobileNetV2 è necessario importare ulteriori librerie:

```
import os
# importazione modulo os utilizzato per interagire con operazioni del sistema
# operativo dell'elaboratore
import cv2
# importazione della libreria cv2 utilizzata per analizzare, dimensionare e
# manipolare immagini o video
import numpy as np
# importazione della libreria numpy necessaria per svolgere operazioni con
# vettori e matrici in Python
import matplotlib.pyplot as plt
# importazione del modulo pyplot con interfaccia MATLAB per la visualizzazione
# delle immagini e realizzazione dei grafici
import seaborn as sns
# importazione della libreria seaborn utilizzata per realizzare analisi
# statistiche
import tensorflow as tf
# importazione della libreria tensorflow necessaria per la realizzazione
# dei modelli di rete neurale e per la fase di allenamento
```

Figura 30: Importazione di librerie successivamente utilizzate per la realizzazione dei modelli

```
from tensorflow.keras import layers, models
# importazione dalla libreria tensorflow.keras della funzione layers e models.
from tensorflow.keras.applications import 'Nome Modello' #VGG16, ResNet50, EfficientNetB0, MobileNetV2
# importazione del modello di rete convoluzionale da utilizzare
from tensorflow.keras.preprocessing.image import ImageDataGenerator
# importazione della funzione ImageDataGenerator dalla libreria
# tensorflow.keras.preprocessing.image utilizzata per aumentare la dimensione del dataset
from sklearn.metrics import classification_report, confusion_matrix,average_precision_score,accuracy_score
# importazione delle funzioni utilizzate per la valutazione delle performance del modello
# dalla libreria sklearn.metrics
from sklearn.model_selection import train_test_split
```

Figura 31: Importazione di librerie necessarie alla realizzazione delle architetture VGG16, ResNet50, EfficientNetB0 e MobileNetV2

3.5.2 Data Augmentation

Per tentare di migliorare le prestazioni del modello, vengono impostati i parametri delle trasformazioni geometriche e fotometriche effettuate sulle immagini del dataset, sia per la fase di allenamento che di verifica, tramite la funzione *ImageDataGenerator*.

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=3,
    width_shift_range=0.12,
    height_shift_range=0.12,
    shear_range=0.12,
    zoom_range=0.12,
    horizontal_flip=True,
    brightness_range=[0.6, 1.4],
    channel_shift_range=0.12,
    fill_mode='nearest')
# Data augmentation sul dataset di immagini caricato per la fase di training,
# nello specifico sono state eseguite in tutte le immagini operazioni di :
# - Normalizzazione
# - Rotazione dell'immagine di 3 gradi
# - Spostamento orizzontale dell'immagine del 12 % della sua intera lunghezza
# - Spostamento verticale dell'immagine del 12 % della sua intera altezza
# - Randomico taglio dell'immagine del 12%
# - Randomico aumento o diminuzione dell'immagine del 12%
# - Randomico ribaltamento orizzontale dell'immagine
# - Regolazione della luminosità delle immagini: 60% più scura e 140% più luminosa
# - Randomico cambiamento di colore sul 12% dell'immagine iniziale
# - I pixel dell'immagine vuoti vengono riempiti con il valore dei pixel vicini
#   (si ottengono transizioni di colore più dolci)

datagen = ImageDataGenerator(rescale=1./255,validation_split=0.5)
# Data augmentation delle immagini del dataset contenute nella fase di verifica
# nello specifico sono operazioni di:
# - Normalizzazione
# - Suddivisione delle immagini utilizzate per la fase di validation
#  (50% di quelle contenute nella directory validation)
```

Figura 32: Prima sezione di codice per effettuare un aumento delle immagini del dataset

Dopo aver configurato i parametri si applica la funzione *ImageDataGenerator* al dataset di immagini della fase di allenamento e di verifica.

```
train_gen = train_datagen.flow_from_directory(
    train_path,
    target_size=(224, 224),batch_size=32,class_mode='binary',subset='training')

# Applicazione della data augmentation con i parametri precedentemente impostati
# per la fase di allenamento della rete

val_gen = datagen.flow_from_directory(
    val_path,
    target_size=(224, 224),batch_size=32,class_mode='binary',subset='validation')

# Applicazione della data augmentation con i parametri precedentemente impostati
# per la fase di verifica della rete
```

Figura 33:Seconda sezione di codice per effettuare la data augmentation.

3.5.3 Caricamento modello pre-allenato

Per caricare il modello o più modelli contemporaneamente può essere sfruttata la seguente struttura:

```
# Definizione e caricamento del modello pre-allenato per effettuare la classificazione delle immagini
models = {
'Nome Modello': NomeModello(weights='imagenet', include_top=False, input_shape=(224, 224, 3)),
}
```

Figura 34: Caricamento del modello pre-allenato sul file di lavoro

Naturalmente, a seconda del modello di rete neurale implementato, si sostituirà 'Nome Modello' con 'VGG16', 'Resnet50', 'EfficientNetB0' o 'MobileNetV2'.

3.5.4 Definizione della funzione *build_model()*

Con il pre-modello si è introdotto sullo spazio di lavoro l'architettura convoluzionale.

Pertanto, per effettuare la classificazione delle immagini del manto stradale, è necessario aggiungere al pre-modello uno o più strati nascosti.

Dopo l'aggiunta e la progettazione degli strati posti in modo sequenziale, si inserisce l'ultima funzione di attivazione (*sigmoid*) che attua la classificazione binaria.

Oltre alla definizione e alla caratterizzazione della architettura della rete si inserisce una funzione (*model.compile(…)*) che applica, come si vede in *Figura 35*, il modello Adams per la regolazione dei pesi in fase di training della rete.

```
# Definizione di funzioni utilizzate:
#   build_model(base_model) è una funzione utilizzata per configurare
#   il modello pre-allenato
def build_model(base_model):
    base_model.trainable = False
    # Vengono mantenuti fissi i pesi del modello pre allenato durante la fase di allenamento
    model = tf.keras.Sequential([
        # si vuole ottenere un modello sequenziale
        base_model,
        tf.keras.layers.GlobalAveragePooling2D(),
        #viene effettuata una riduzione spaziale della feature map con un Average Pooling
        tf.keras.layers.Dense(128, activation='relu'),
        #Inserisco lo strato fully connected layers con 128 neuroni,
        #seguiti da una funzione di attivazione ReLU
        tf.keras.layers.Dropout(0.5),
        #Rilascio casuale del 50% dei neuroni preseenti nella fase di training
        tf.keras.layers.Dense(1, activation='sigmoid')
        #Lo strato di uscita, composto da un singolo neurone, che effettua
        # la classificazione è composto da una funzione di attivazione sigmoidea.
    ])
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    # Si effettua una configurazione del modello di training utilizzando il
    # modello ottimizzato Adam, una funzione di costo adatta alla classificazione
    # binaria e il metodo di valutazione del modello è riferito al valore di
    # accuratezza
    return model
```

Figura 35: Definizione della funzione build_model(base_model)

3.5.5 Definizione della funzione *calculate_map()*

Con la funzione *calculate_map()* si determina un parametro chiamato mAP (*mean Average Precision*) che indica, con un valore minimo di zero ad un valore massimo di uno, quanto bene un modello è in grado di individuare e classificare correttamente le immagini appartenenti alla classe positiva, ovvero alle immagini che contengono una o più buche.

```
#   la funzione calculate(y_true,y_pred) permette di calcolare il mean Average
#   Precision
def calculate_map(y_true, y_pred)
  ap = average_precision_score(y_true, y_pred)
  #La mAP è uguale all'AP in un problema di classificazione binaria
  map_score = ap
  return map_score
```

Figura 36:Definizione della funzione utilizzata per il calcolo del Mean Average Precision

3.5.6 Definizione della funzione get_macs()

La funzione *get_macs* permette di calcolare il numero di operazioni in virgola mobile (MACs: *Multiply-Accumulate Operations*) eseguite da un modello di rete neurale durante l'inferenza con l'hardware messo a disposizione da Colab.

Essenzialmente, ogni volta che un neurone in una rete neurale esegue un'operazione di moltiplicazione tra i suoi pesi e gli input, seguita da un'operazione di somma (accumulo), viene conteggiato un MAC.

È importante conoscere il numero di MACs per poter effettuare delle stime della quantità di potenza di calcolo necessaria per eseguire il modello su un particolare sistema embedded.

```
def get_macs(model):
    total_macs = 0
    for layer in model.layers:
        if isinstance(layer, (layers.Conv2D, layers.DepthwiseConv2D)):
# se il corrente strato è uno strato convoluzionale 2D o uno strato
# convoluzionale di profondità

          output_shape = layer.output_shape[1:]
# forma dell'output dello strato identico a quello dello strato prec.
          input_shape = layer.input_shape[1:]
# forma dell'input dello strato identico a quello dello strato prec.
          total_macs += (layer.filters * input_shape[2] *
                         layer.kernel_size[0] * layer.kernel_size[1] *
                          output_shape[0] * output_shape[1])
          if isinstance(layer, layers.DepthwiseConv2D):
# For DepthwiseConv2D, input_channels is 1
              total_macs /= input_shape[2]
        elif isinstance(layer, layers.Dense):
# se il corrente strato è uno strato fully connected
# MACs for dense layers: output_features * input_features
# Get input shape from the layer's input tensor
          input_shape = layer.input.shape
          total_macs += layer.units * input_shape[-1]
    return total_macs
```

Figura 37: Definizione della funzione che calcola il Multiply_Accumulate operation (MACs)

3.5.7 Struttura del modello per la fase di training e di verifica

Le fasi di allenamento e di valutazione di una rete neurale sono fondamentali per sviluppare un modello di deep learning:

Nella fase di Allenamento (*Training)*, valutando il dataset di input *train_gen,* vengono eseguite dalla rete le seguenti operazioni:

- l'inizializzazione dei pesi e dei bias con valori casuali
- la produzione delle previsioni del modello sulle immagini contenute nel set di allenamento.
- Il calcolo della funzione di costo
- L'aggiornamento dei pesi tramite ottimizzazioni effettuate sulla funzione di costo. Il processo di allenamento va ripetuto per più volte per ottenere una bassa funzione di costo ed ogni ripetizione rappresenta una epoca (*epochs*).

Mentre, lo scopo della fase di Valutazione (*Validation*) è stimare quanto bene la rete generalizza le feature memorizzate nella fase di allenamento a nuovi dati che non ha mai visto durante l'allenamento contenuti in *val_gen*.

```
# Train, evaluate and collect results
results = {}
#inizializzo il vettore che conterrà i risultati della fase di allenamento e di verifica
for name, base_model in models.items():
    print(f"Training {name}")
    model = build_model(base_model)

    # Train:
    history = model.fit(train_gen, validation_data=val_gen, epochs=10)
    # la funzione model.fit effettua l'allenamento della rete utilizzando come
    # dataset quello contenuto nell'insieme train_gen e quello contenuto nello
    # insieme val_gen per 10 epochs.
    # Vengono memorizzate le prestazioni della rete nella variabile history

    # Validation:
    val_pred = model.predict(val_gen)
    # vengono effettuate delle previsioni sulla fase di validation valutando
    # le prestazioni del modello allenato
    val_pred_labels = (val_pred > 0.5).astype(int)
    # Conversione dei valori di probabilità in 2 classi: 1 che contiene i valori
    # <0.5 e una che contiene i valori > 0.5
    y_true = val_gen.classes
    map_score = calculate_map(y_true, val_pred)
    print(f"MAP: {map_score}")
    # Confusion Matrix and Classification Report
    cm = confusion_matrix(y_true, val_pred_labels)
    # la funzione confusion_matrix calcola e produce la matrice di confusione
    cr = classification_report(y_true, val_pred_labels, target_names=val_gen.class_indices.keys())
    # la funzione classification_report fornisce proprietà sulle prestazioni del
    # modello come la precisione, recall e F1-score.
```

Figura 38:Sezione di codice contenuto nella fase di allenamento

3.5.8 Calcolo del tempo di inferenza in fase di verifica

Il tempo di inferenza nel deep learning è riferito al tempo impiegato da un modello addestrato per fare previsioni o classificazioni su nuovi dati. Questo è un aspetto critico.

Con il tempo di inferenza devono essere valutati i seguenti aspetti:

- *Complessità del modello*: modelli più complessi, come ad esempio reti neurali con diversi strati profondi, richiedono tempi di inferenza più lunghi a causa delle maggiori richieste di elaborazione.
- *Dimensioni del batch*: l'inferenza può essere eseguita su una singola immagine oppure su un campione di immagini (batch). Dimensioni del batch maggiori possono migliorare la produttività anche se aimè aumenta la latenza della previsione dei singoli input.
- *Tecniche di compressione*: tecniche come la quantizzazione del modello (rappresenta i pesi con una risoluzione più bassa), il pruning (rimuove le connessioni meno importanti tra i neuroni) e l'utilizzo di librerie ottimizzate possono ridurre il tempo di inferenza senza sacrificare significativamente la precisione.
- *Dimensione di input*: anche le dimensioni e la dimensionalità dei dati di input possono avere un impatto sul tempo di inferenza. Input più grandi possono richiedere più tempo di elaborazione.
- *Hardware*: il tipo di hardware utilizzato (CPU, GPU, TPU) influisce in modo significativo sul tempo di inferenza. GPU e TPU sono generalmente più veloci per le attività di deep learning grazie alle loro capacità di elaborazione parallela.

Dopo aver analizzato gli aspetti legati al tempo di inferenza, tramite delle temporizzazioni, viene determinato il tempo di inferenza.

In particolare, viene valutato il tempo di inferenza mediato sul numero di previsioni effettuate. Tale numero corrisponde al numero di immagini contenute nel set *val_gen*.

```
inference_times = []
for i in range(len(val_gen)):
  # iterazione con durata pari alla lunghezza dello
  # insieme val_gen
    start_time = time.time() #memorizzazione del tempo in tale operazione
    batch_x, batch_y = val_gen[i]  # Prelievo di un campione di dati
    predictions = model.predict(batch_x)  # lancio del modello con il campione
    end_time = time.time() #memorizzazione del tempo in tale operazione
    inference_time = end_time - start_time
    # calcolo del tempoi di inferenza
    inference_times.append(inference_time)

# Calculate average inference time per batch
average_inference_time_per_batch = sum(inference_times) / len(inference_times)

# Calculate average inference time per sample
average_inference_time_per_sample = average_inference_time_per_batch / val_gen.batch_size

print(f"Average inference time per batch: {average_inference_time_per_batch:.4f} seconds")
print(f"Average inference time per sample: {average_inference_time_per_sample:.4f} seconds")
################################################################################
macs = get_macs(model)
print(f"MACs for this model: {macs}")
```

Figura 39:Calcolo tempo di inferenza per campione e per singola operazione e calcolo del numero di MAC

3.5.9 Visualizzazione delle specifiche hardware fornite da Google Colab

Il calcolo del tempo di inferenza e del tempo di esecuzione della rete nella fase di allenamento hanno un significato se vengono specificate le caratteristiche hardware fornite da Colab.

Per fare ciò, si utilizzano file di sistema virtuali che contengono informazioni statiche sul sistema, in particolare sul processore e sulla memoria.

```
!cat /proc/cpuinfo
!cat /proc/meminfo
```

Figura 40: File di sistema virtuali

Il processore fornito per tutte i modelli ha le seguenti specifiche:

- Processore utilizzato: Intel(R) Xeon(R) CPU @ 2.20GHz
- Memoria messa a disposizione: 12.8 GB

3.5.10 Funzioni per la rappresentazione grafica delle prestazioni

L'ultima sezione del codice è dedicata alla visualizzazione dei risultati ottenuti dai modelli. Tramite l'utilizzo di librerie grafiche e di data visualization, vengono generate rappresentazioni visive, chiare e intuitive delle prestazioni dei modelli, quali grafici di accuratezza, matrici di confusione e tabelle riassuntive delle metriche.

```
# Store results
results[name] = {
    'accuracy': history.history['val_accuracy'][-1],
    'loss': history.history['loss'],
    'val_loss': history.history['val_loss'],
    'confusion_matrix': cm,
    'classification_report': cr
}

# Plot della matrice di confusione
plt.figure(figsize=(6,6))
sns.heatmap(cm, annot=True, fmt='d', cmap='bone')
plt.title(f'Confusion Matrix for {name}')
plt.show()

# Stampo i risultati della classificazione
print(f"Classification Report for {name}:\n{cr}")
print(f"Accuracy: {results[name]['accuracy'] * 100:.2f}%\n")

# Plot della funzione di costo
plt.figure(figsize=(8,6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title(f'Loss Function for {name}')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Figura 41: Sezione di codice per la rappresentazione delle prestazioni del modello nella fase di allenamento e di verifica.

4. RISULTATI SPERIMENTALI

4.1 MATRICE DI CONFUSIONE

La matrice di confusione è una tabella che evidenzia le prestazioni di un modello di classificazione, confrontando le classi predette dal modello con le classi effettive di uno specifico dataset.

Ciò permette di valutare gli errori effettuati dal modello e di calcolare uno tra i parametri caratterizzanti della rete ovvero l'accuratezza della fase di verifica.

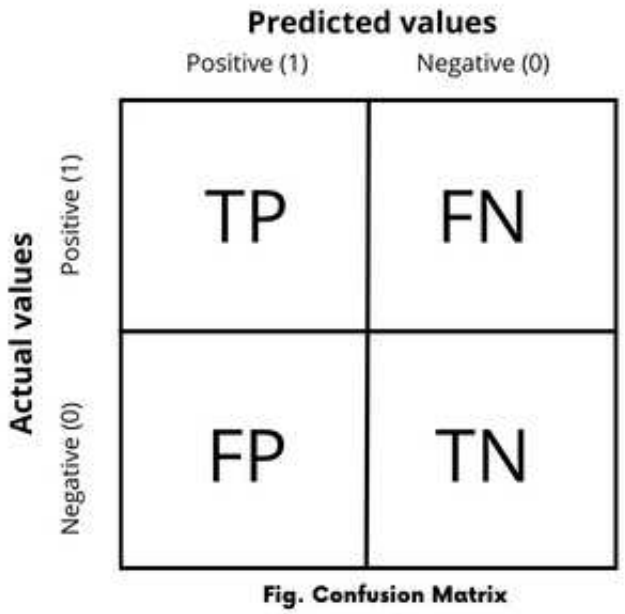

Figura 42: Matrice di confusione per una classificazione binaria:

- ***TP** (True Positive): valori positivi previsti come positivi.*
- ***FP** (False Positive): valori negativi previsti come positivi.*
- ***FN** (False Negative): valori positivi previsti come negativi.*
- ***TN** (True Negative): valori negativi previsti come negativi.*

Per effettuare la classificazione binaria delle immagini per il rilevamento delle buche, è stato associato il valore 0 alla classe di immagini che non presentano buche mentre il valore 1 all'insieme che presentano una o più buche.

Inoltre, da tale matrice è possibile determinare l'accuratezza del modello nella fase di validation utilizzando la seguente formula:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

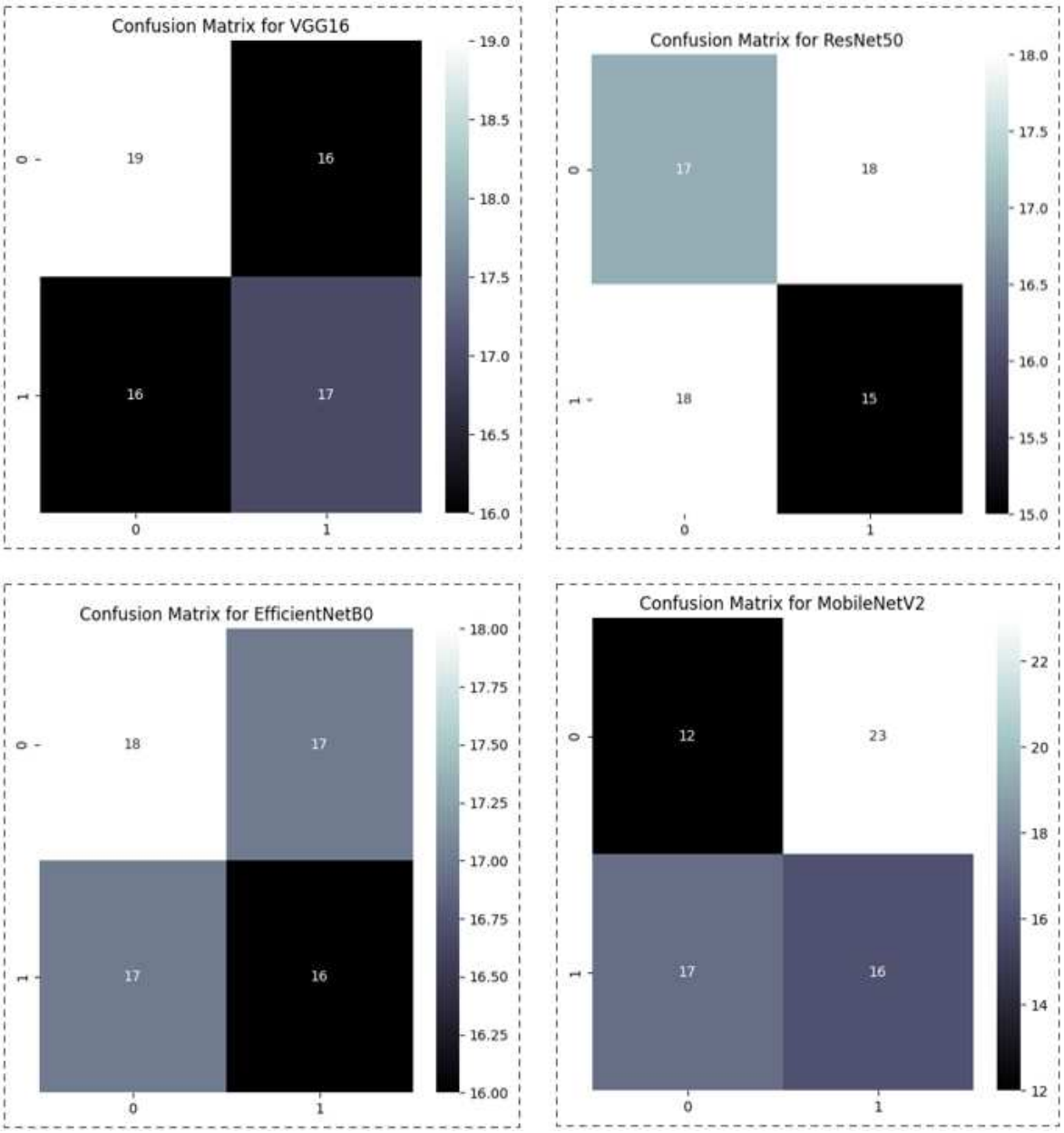

Figura 43:Matrice di confusione dei modelli (senza data augmentation)

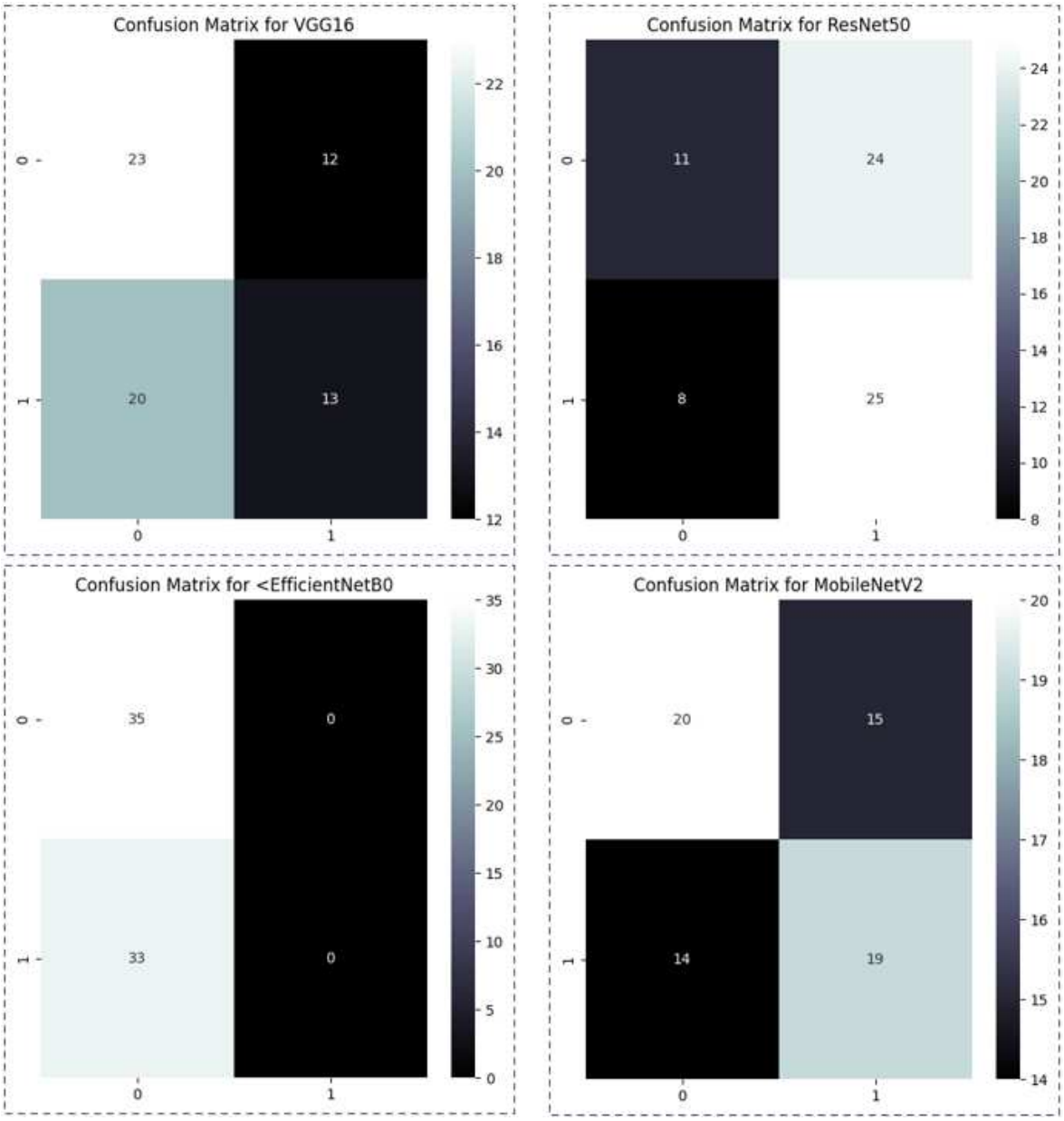

Figura 44:Matrice di confusione dei modelli (con data augmentation)

4.2 FUNZIONE DI COSTO

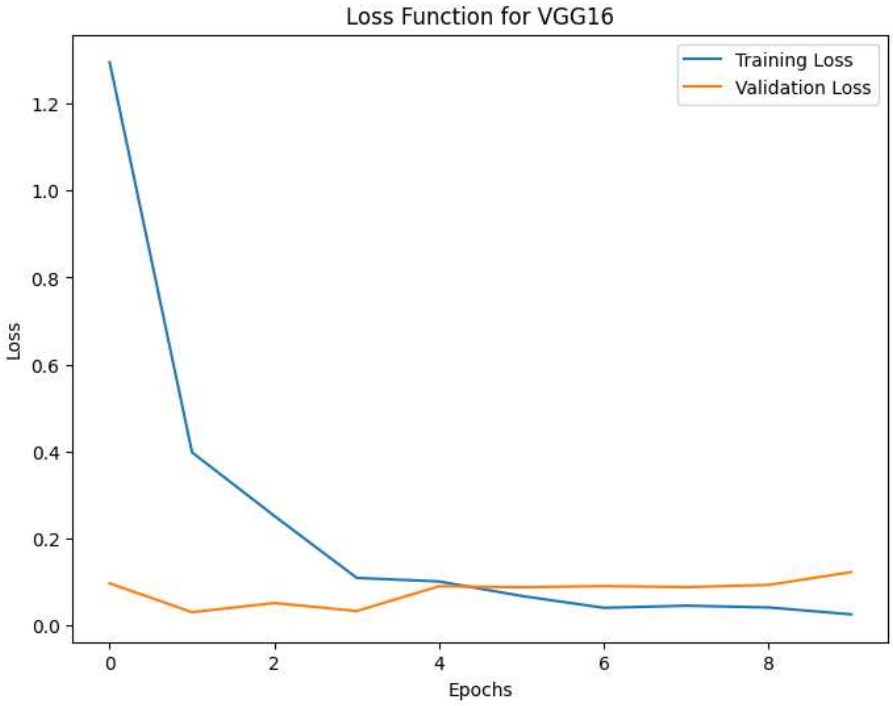

Figura 45: Funzione di costo del modello VGG16 senza data augmentation.

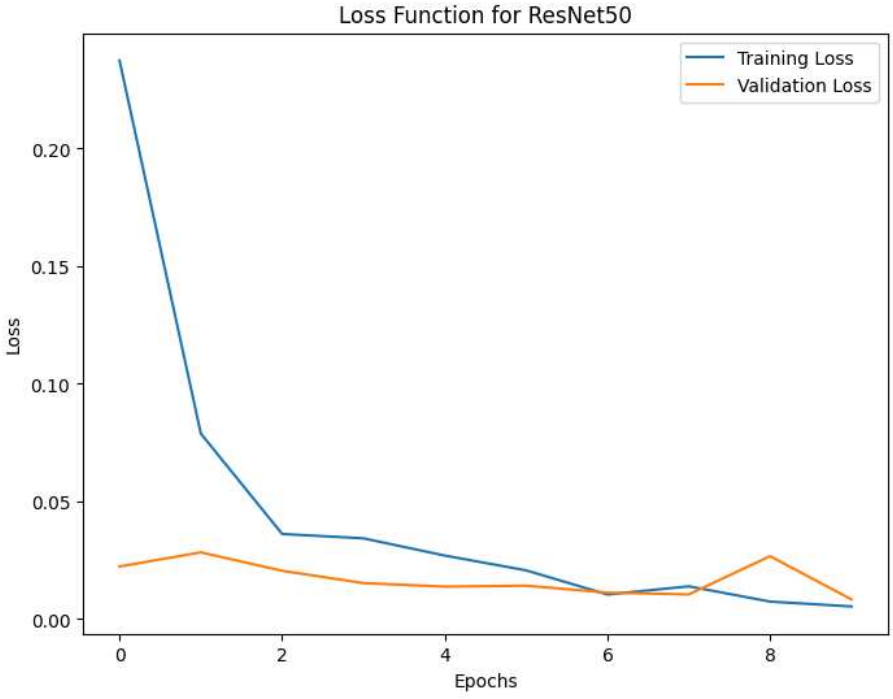

Figura 46: Funzione di costo del modello ResNet50 senza data augmentation.

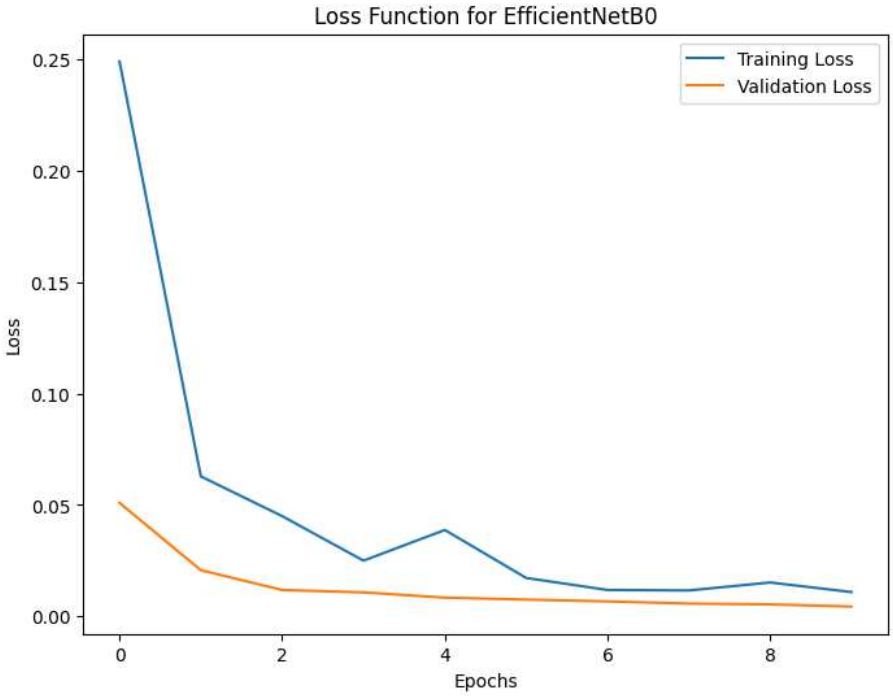

Figura 47: Funzione di costo del modello EfficientNetB0 senza data augmentation.

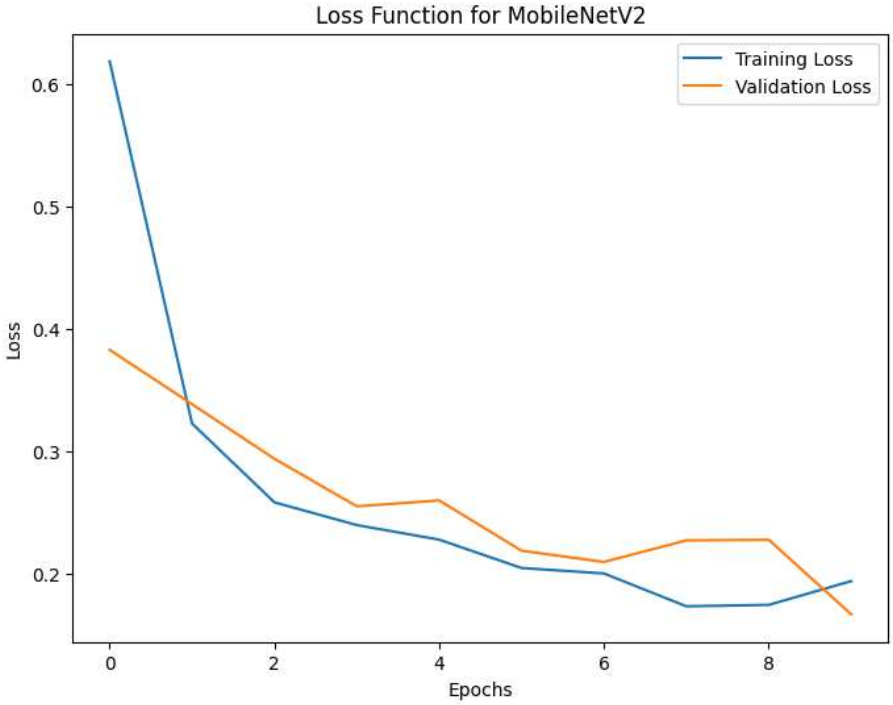

Figura 48: Funzione di costo del modello MobileNetV2 senza data augmentation.

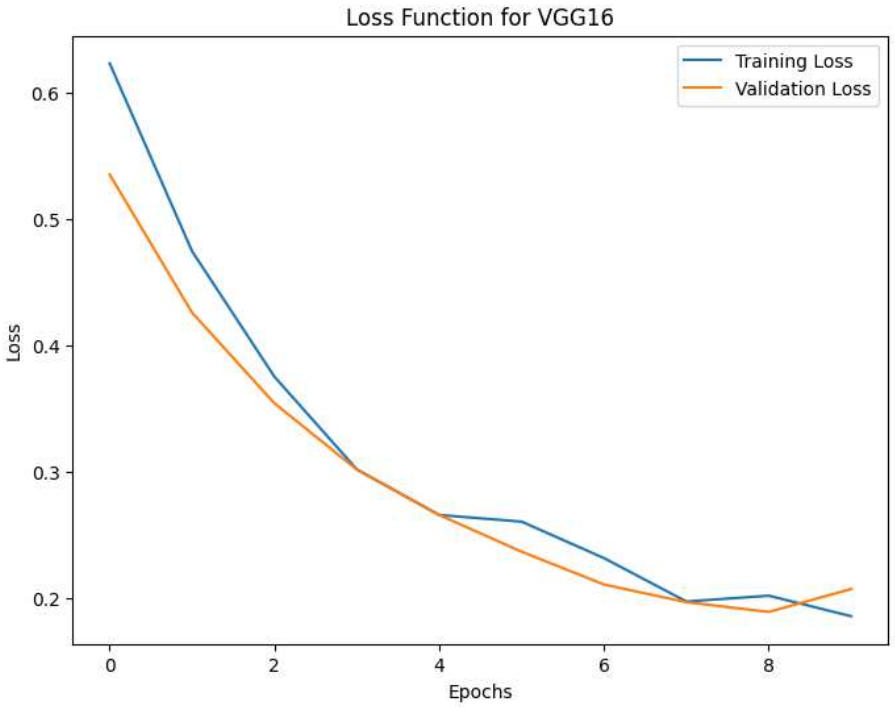

Figura 49: Funzione di costo del modello VGG16 con data augmentation.

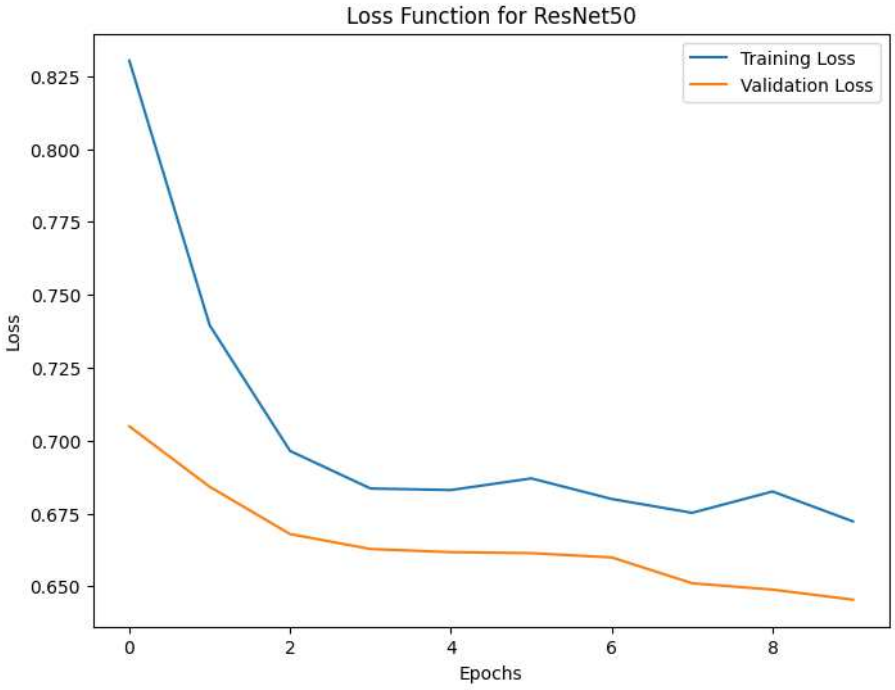

Figura 50: Funzione di costo del modello ResNet50 con data augmentation.

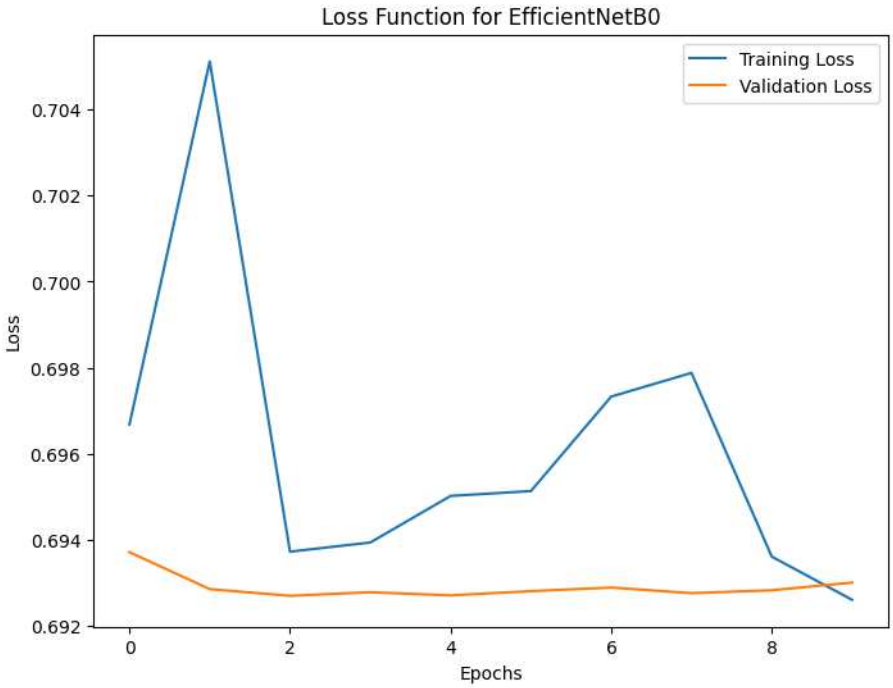

Figura 51: Funzione di costo del modello EfficientNetB0 con data augmentation

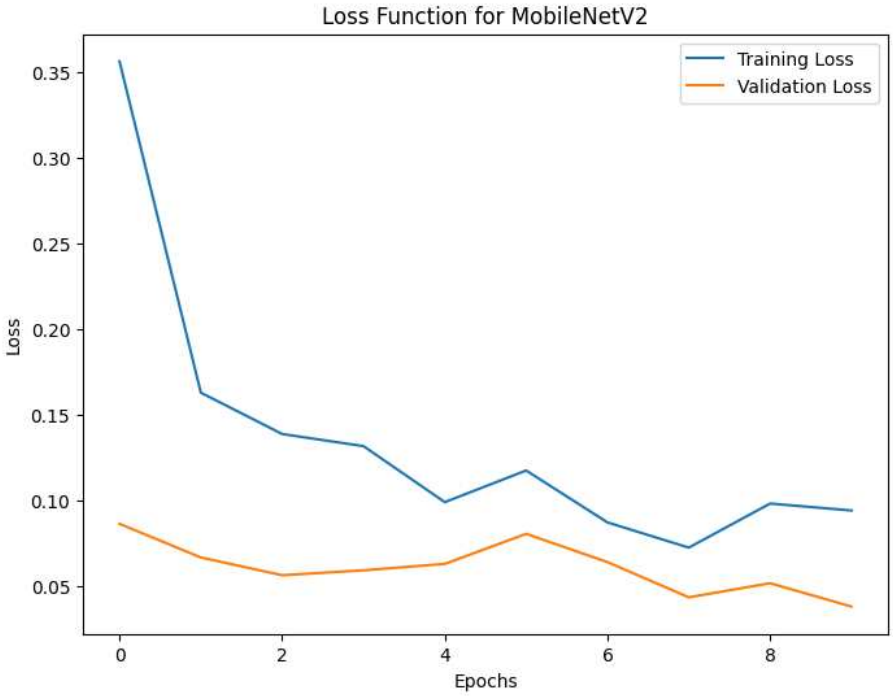

Figura 52: Funzione di costo del modello MobileNetV2 con data augmentation.

4.3 RISULTATI FINALI OTTENUTI

10 Epochs per ogni modello	*Accuratezza fase di training*	*Accuratezza fase di verifica*	*mAP*	*Tempo di esecuzione (training)*	*Tempo di inferenza medio (test)*	*Memoria Occupata formato.h5*	*MAC*
VGG16	97.06%	52.94%	0.5274	1h 32’ 05’’	555.4 ms	57.0 MB	65664
ResNet50	99.98%	47.06%	0.4235	32’ 34’’	252.4 ms	93.4 MB	262272
EfficientNetB0	100.00%	51.47%	0.5877	13’ 15’’	113.9 ms	17.8 MB	163968
MobileNetV2	86.76%	41.17%	0.4949	12’ 41’’	97.5 ms	10.8 MB	163968

Figura 53:Tabella dei risultati ottenuti (senza data augmentation).

10 Epochs per ogni modello	*Accuratezza fase di training*	*Accuratezza fase di verifica*	*mAP*	*Tempo di esecuzione (training)*	*Tempo di inferenza medio (test)*	*Memoria Occupata formato.h5*	*MAC*
VGG16	88.24%	52.94	0.5105	1h 2’ 06’’	383.7 ms	57.0 MB	65664
ResNet50	73.53%	52.94	0.6223	22’ 16’’	173.9 ms	93.4 MB	262272
EfficientNetB0	51.47%	51.47%	0.5582	12’ 19’’	108.8 ms	17.8 MB	163968
MobileNetV2	98.53%	57.35%	0.6184	8’ 18’’	71.5 ms	10.8 MB	163968

Figura 54: Tabella dei risultati ottenuti (con data augmentation).

5. ANALISI DEI RISULTATI OTTENUTI

Per garantire un confronto equo tra i modelli, si sono controllate le variabili sperimentali, mantenendo inalterati sia l'hardware che gli iperparametri. Ciò ha permesso di attribuire le differenze nelle prestazioni direttamente alle diverse architetture delle reti neurali.

Partendo dalle matrici di confusione, si nota come l'applicazione dell'algoritmo di *Data Augmentation* non abbia avuto lo stesso effetto nelle diverse reti pur avendo gli stessi parametri.

In effetti la rete che ha beneficiato di tale algoritmo è la più semplice strutturalmente ovvero l'architettura *MobileNetV2*. Essendo composta da meno strati profondi, la rete è più soggetta al fenomeno di *overfitting* ma, applicando la *data augmentation*, ottiene una maggiore varietà di esempi che riducono il rischio di memorizzazione e aumenta la generalizzazione alla classe di immagini. Le immagini del dataset in cui sono state effettuate delle trasformazioni e variazioni significative sono comunque adatte al modello perché sono contenuti dei blocchi residui che riescono ad estrarre feature dalle immagini.

Per gli altri modelli, la *data augmentation* ha peggiorato le prestazioni, potenzialmente a causa di un eccesso di complessità introdotta nel processo di allenamento. Operazioni di trasformazione delle immagini già integrate nei modelli, come quelle presenti in architetture come *ResNet,* potrebbero essere in conflitto con ulteriori trasformazioni introdotte dalla data augmentation. Questo sovraccarico di informazioni ha prodotto un rumore eccessivo, impedendo al modello di estrarre le feature più rilevanti e di generalizzare efficacemente a nuovi dati. In sostanza, questi modelli sono diventati troppo specializzati nel riconoscere le variazioni introdotte dall'augmentation, perdendo di vista le caratteristiche intrinseche delle due classi.

Passando ai grafici che valutano il valore della funzione di costo in ogni epoca, si nota come all'aumentare delle epoche la funzione di costo diminuisce notevolmente. Tale andamento è apprezzato.

Valutando la tabella dei risultati ottenuti *senza data augmentation* si nota come tutti i modelli mostrino un significativo divario tra l'accuratezza in training e validation, indicando un evidente *overfitting*. Questo è particolarmente evidente per VGG16 e ResNet50, che raggiungono un'accuratezza quasi perfetta in training ma performano in modo peggiore nel processo di validation.

Mentre nella tabella dei risultati ottenuti con la data augmentation, tutti i modelli mostrano ancora un significativo divario tra l'accuratezza in training e validation, indicando un *persistente problema di overfitting*. Questo suggerisce che la data augmentation, da sola, non è stata sufficiente a risolvere completamente questo problema.

Quindi per poter migliorare l'accuratezza dei modelli utilizzando tale dataset, in future analisi sperimentali, potranno essere apportate delle modifiche al dataset come: l'aumento della dimensione del dataset e l'aumento della variabilità delle immagini.

Considerando però parametri come MACs, tempo di inferenza medio nel processo di verifica e la dimensione in termini di occupazione di memoria si nota come il modello di rete CNN *MobileNetV2* sia quello più adoperabile nel mondo dei sistemi embedded.

I sistemi embedded sono progettati per eseguire specifiche funzioni all'interno di sistemi più grandi, e perciò hanno piccole dimensioni e limitate risorse di CPU, memoria ed energia.

Dalla tabella dei risultati ottenuti, applicando la *data augmentation*, si nota come il modello *MobileNetV2* abbia, una elevata efficienza computazionale. Essa è dovuta ad un numero di *MACs* più basso rispetto ad altre architetture, che di conseguenza permette di avere minor consumi energetici. Inoltre, a parità di hardware, ha un minor tempo di inferenza.

La velocità di elaborazione delle immagini è un fattore imprescindibile nelle applicazioni di sicurezza stradale, in quanto ogni frazione di secondo può fare la differenza tra un incidente evitato e un danno irreparabile. Perciò per tali operazioni definite real-time è richiesto un basso tempo di inferenza (*fino a qualche decina di millisecondo*) ed una bassa occupazione di memoria (*fino a qualche MB*).

Una eccessiva occupazione di memoria può rallentare il sistema e compromettere la sua capacità di rispondere tempestivamente. Per questo i limiti più restringenti riguardano tali argomenti.

6. PORTING SU BOARD STM32

L'analisi sperimentale condotta ha evidenziato come i limiti per l'implementazione delle reti neurali sui sistemi embedded siano molto delimitanti alle strutture di rete CNN.

Anche se gli organismi analizzati sono molto complessi ed efficienti in molti ambienti di sviluppo sono ancora poco applicabili a sistemi che hanno dei vincoli sul consumo energetico, tempo di previsione e occupazione di memoria. Nonostante ciò, si è notato che una architettura ha migliori proprietà e quindi può essere una base di partenza su cui effettuare ulteriori sviluppi.

Un primo passo per poter ridurre l'occupazione di memoria di un modello CNN è quello di aggiungere alla struttura le seguenti operazioni, distinte in funzione delle specifiche dell'hardware utilizzato nei sistemi embedded, per la guida autonoma:

- Quantizzazione:
    - *Funzione*: viene applicata una riduzione sulla precisione numerica dei pesi e delle funzioni di attivazione. In particolare, si utilizzano numeri a virgola fissa a 16, 8 fino a 1 bit invece dei numeri in virgola mobile a 32 bit.
    - *Effetti*: si ottiene una riduzione significativa delle dimensioni del modello e si riduce il tempo di inferenza. Però si ha anche un peggioramento della precisione del modello.

- Pruning:
    - *Funzione*: vengono eliminate alcune connessioni o neuroni della rete non molto importanti. Ciò è possibile perché essendo la rete piena di connessioni tra i vari strati e tra i neuroni sono presenti anche molte connessioni prive di informazioni.
    - *Effetti*: l'eliminazione di tali elementi permette di ridurre ulteriormente l'occupazione del modello. Anche con questa operazione però si ha un peggioramento della accuratezza dovuta alla perdita di una quantità di informazione correlata alle connessioni e neuroni eliminati. Inoltre trovare un buon algoritmo di pruning per l'identificazione di connessioni meno importanti è una sfida complessa.

Esistono poi altri metodi più sofisticati utilizzati per effettuare la compressione del modello di rete neurale convoluzionale che possono includere la decomposizione dei pesi e la dimensione del rango delle dei matrici per abbassare il numero dei parametri.

Prendendo come riferimento, visti i migliori risultati ottenuti, il modello *MobileNetV2* può essere applicata una quantizzazione post allenamento utilizzando la libreria *Tensorflow*:

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
#Creazione del TensorFlow Line converter sul modello keras allenato
converter.optimizations = [tf.lite.Optimize.DEFAULT]
#Abilito la quantizzazione che passa da valori float a 32 bit
#a valori int a 8 bit
quantized_tflite_model = converter.convert()
#Attuazione della conversione del modello al formato quantizzato
with open('MobileNetV2.tflite', 'wb') as f:
   f.write(quantized_tflite_model)
#Salvataggio del modello quantizzato
```

Figura 55:Quantizzazione dinamica del modello MobileNetV2.

In particolare, come si vede in *Figura 55*, è stata applicata una quantizzazione per cambiare il numero di bit utilizzati per la rappresentazione dei valori di attivazione e i pesi.

Nel processo, oltre alla quantizzazione dei pesi, volta dopo il termine della fase di allenamento (da valori a 32 bit a valori a 8 bit), avviene dinamicamente la quantizzazione a 8 bit delle attivazioni nella fase di inferenza. Ciò significa che, per ogni strato del modello, viene determinato l'intervallo contenente tutti i valori in uscita delle funzioni di attivazioni, in modo che venga trovato il valore massimo e minimo delle attivazioni dello strato. Successivamente, viene effettuata una mappatura dei valori in una base numerica composta da 8 bit e, come ultimo step del processo, viene applicata la quantizzazione del modello applicando un arrotondamento-troncamento dei valori di attivazione ai valori più vicini dell'intervallo a 8 bit.

La dimensione ottenuta dal modello quantizzato è pari a *2,54 MB*, ciò significa che l'occupazione di memoria è stata ridotta più di quattro volte rispetto al modello con valori in virgola mobile. Tale risultato però non è sufficiente, visto che per i sistemi embedded (STM32) risulta essere ancora una occupazione di memoria elevata.

Il porting su sistemi embedded implica l'ottimizzazione del modello convoluzionale, che spesso include la riduzione dei parametri per adattarli alle risorse hardware limitate e migliorare le prestazioni.

Quindi per ridurre il numero dei parametri sono state eseguite tali operazioni nel modello non quantizzato precedentemente introdotto:

- Diminuzione della dimensione delle immagini del dataset e del tipo di ingresso del pre-modello: da un formato (224x224x3) a (64x64x3).
- Diminuzione del parametro alpha del pre-modello (dal valore di default al valore 0.35): diminuzione del 35% del numero dei filtri in ogni strato della rete MobileNetV2
- Diminuzione del numero di neuroni presenti al livello profondo applicato successivamente al pre-modello: da 128 a 64.

```
    train_path,
    target_size=(64, 64),batch_size=32,class_mode='binary',subset='training')

# Applicazione della data augmentation con i parametri precedentemente impostati
# per la fase di allenamento della rete

val_gen = datagen.flow_from_directory(
    val_path,
    target_size=(64, 64),batch_size=32,class_mode='binary',subset='validation')

# Applicazione della data augmentation con i parametri precedentemente impostati
# per la fase di verifica della rete

# Definizione e caricamento del modello pre-allenato per effettuare la classificazione delle immagini.
models = {
'MobileNetV2': MobileNetV2(weights='imagenet',alpha=0.35,include_top=False, input_shape=(64, 64, 3)),
}
                tf.keras.layers.Dense(64,activation='relu'),
```

Figura 56: Modifiche apportate sul modello per la riduzione della dimensione della rete MobileNetV2

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=5,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.01,
    shear_range=0.1,
    horizontal_flip=True,
    brightness_range=[1.4,1.3],
    fill_mode='nearest',
    channel_shift_range=20,
)
```

Figura 57: Modifiche apportate ai parametri della data augmentation per migliorare le prestazioni del modello MobileNetV2

Con tali modifiche e con una migliore regolazione dei parametri della data augmentation, effettuando la fase di training e di verifica si sono ottenute le seguenti prestazioni:

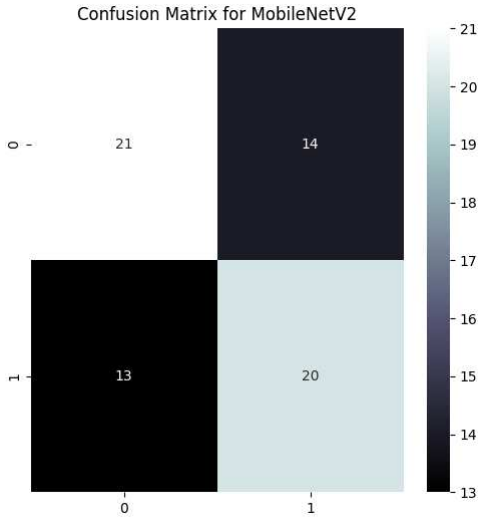

Figura 58: Matrice di confusione del modello MobileNetV2 con immagini di dimensione (64 x 64 x 3) e minor parametri

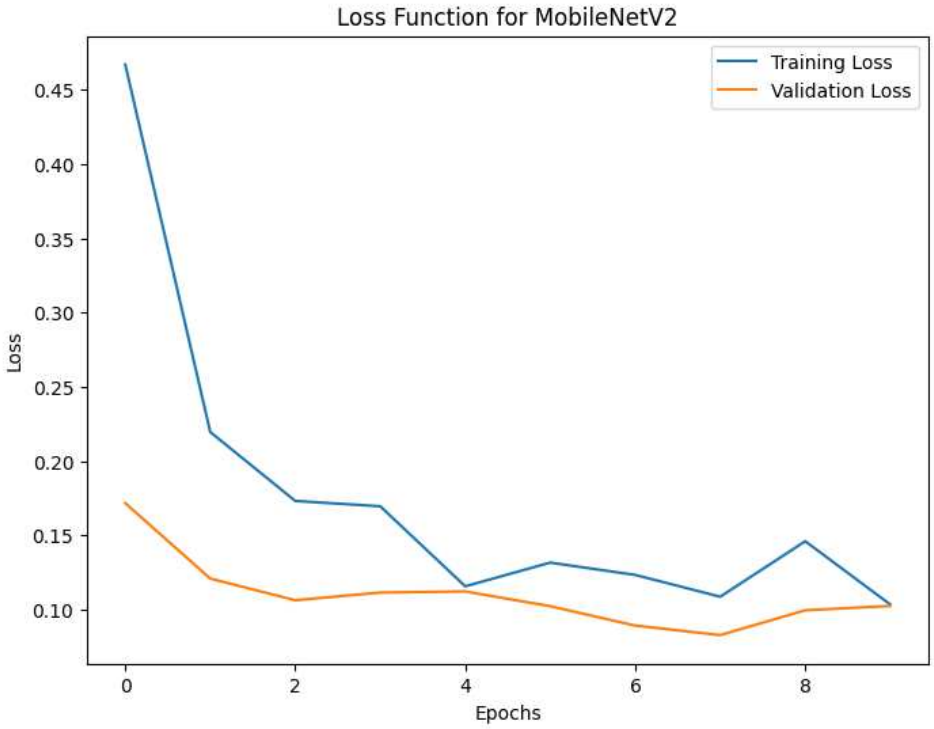

Figura 59: Funzione di costo del modello MobileNetV2 con immagini di dimensione (64 x 64 x 3) e minor parametri

10 Epochs	***Accuratezza fase di training***	***Accuratezza fase di verifica***	***mAP***	***Tempo di esecuzione*** *(training)*	***Tempo di inferenza medio****(test)*	***Memoria Occupata*** *formato.h5*	***Memoria occupata*** *Formato .tflite*	***MAC***
MobileNetV2	95.59%	60.29%	0.5839	3’ 46’’	50.4 ms	2.86 MB	1.83 MB	81984

Figura 60: Tabella dei risultati ottenuti con il modello MobileNetV2 con immagini di dimensione (64 x 64 x 3) e minor parametri

Dunque, in questo lavoro di tesi è stata ridotta la risoluzione delle immagini in ingresso ed il numero di parametri del modello per effettuare il porting su board STM32H7S78-DK, ma sviluppi futuri sono mirati all'applicazione di tecniche di compressione più complesse per ottimizzare il porting della CNN su microcontrollore.

6.1 MICROCONTROLLORE STM32H7S78-DK

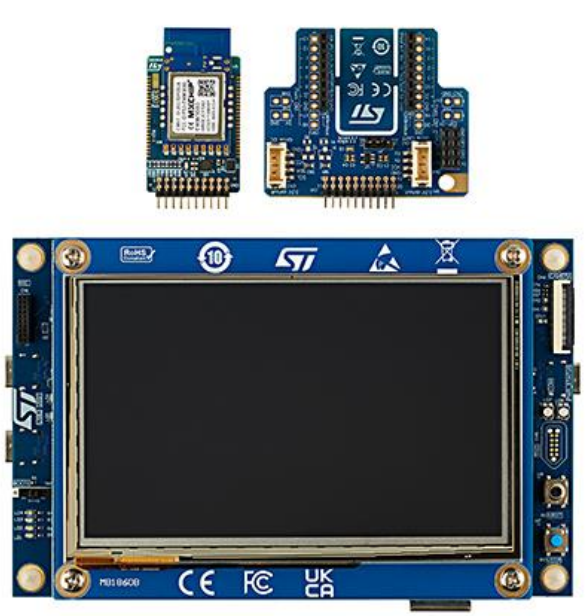

Figura 61: STM32H7S78-DK Discobery kit content

Il kit di sviluppo STM32H7S78-DK[10], offerto da STMicroelectronics, rappresenta una piattaforma completa e versatile per la progettazione e lo sviluppo di applicazioni embedded ad alte prestazioni. La scheda presenta un microcontrollore STM32H7 basato sull'architettura ARM Cortex-M7, noto per la sua elevata potenza di calcolo e la flessibilità.

Caratteristiche Principali:

- prestazioni elevate grazie al core ARM Cortex-M7 e alla presenza di una FPU (Floating-Point Unit). Ciò lo rende ideale per applicazioni che richiedono un'elaborazione numerica intensiva, come ad esempio il controllo motorio avanzato, l'elaborazione di segnali audio e la visione artificiale.
- Ampia gamma di periferiche: Il kit è dotato di un'ampia varietà di periferiche integrate, tra cui:
    - *Interfacce di comunicazione*: USB Type-C, SPI, I2C, UART, Ethernet, per una connettività versatile.

[10] «stm32h7s78-dk.pdf».

- *Memorie*: Flash e SRAM per il salvataggio di dati e codice.
- *Periferiche analogiche*: ADC, DAC, timer, per la gestione di segnali analogici provenienti da sensori o destinati ad attuatori.
- *Display*: Un display LCD a colori con touch screen per creare interfacce utente intuitive.
- *Audio*: Codec audio per la gestione di segnali audio.

- Flessibilità: con dei connettori di espansione, il kit può essere personalizzato e ampliato con moduli aggiuntivi, come sensori, attuatori e shield, per adattarsi a diverse esigenze applicative.
- Ambiente di sviluppo: Il kit è supportato da un ecosistema di strumenti di sviluppo, tra cui IDE (come STM32CubeIDE) e librerie, che facilitano la programmazione e il debug.

6.2 PORTING SU STM32H7S78-DK CON STM32CUBE.AI DEVELOPER CLOUD

STM32Cube.AI Developer Cloud[11] è una piattaforma online gratuita offerta da STMicroelectronics, progettata specificamente per semplificare e accelerare lo sviluppo di applicazioni di intelligenza artificiale sui microcontrollori STM32.

Questa piattaforma permette di creare, ottimizzare e implementare modelli di apprendimento automatico direttamente sui dispositivi embedded ST. Nel caso interessato, la piattaforma è stata utilizzata per effettuare il Benchmarking che, come miglior microcontrollore per il modello MobileNetV2, ha selezionato il modello STM32H7S78-DK. Oltre a tale funzione è stata avviata una valutazione delle prestazioni della rete su tale microcontrollore. In particolare, dalla piattaforma è stato determinato il tempo di inferenza medio e il tempo di inferenza di ogni predizione in ogni strato della architettura.

In seguito, sono riportati, tramite delle immagini, gli step da seguire per effettuare le operazioni precedentemente descritte sulla piattaforma STM32Cube.AI Developer Cloud.

[11] «ST Edge AI Developer Cloud».

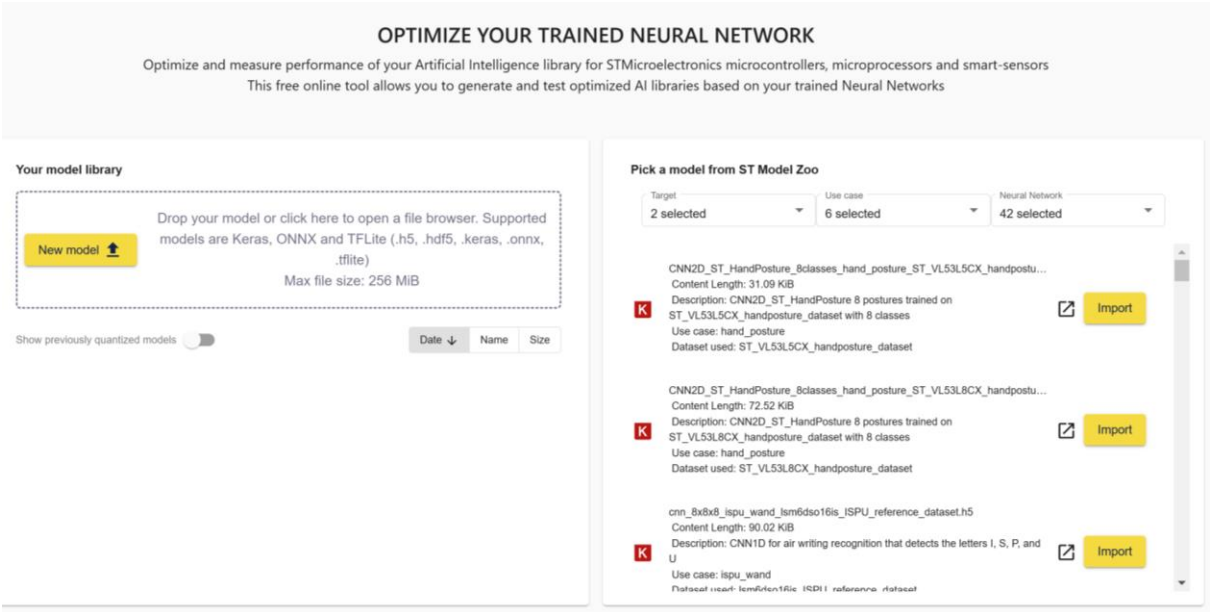

Figura 62: Schermata iniziale della piattaforma STM32Cube.AI Developer Cloud dopo aver effettuato l'accesso con un account ST.

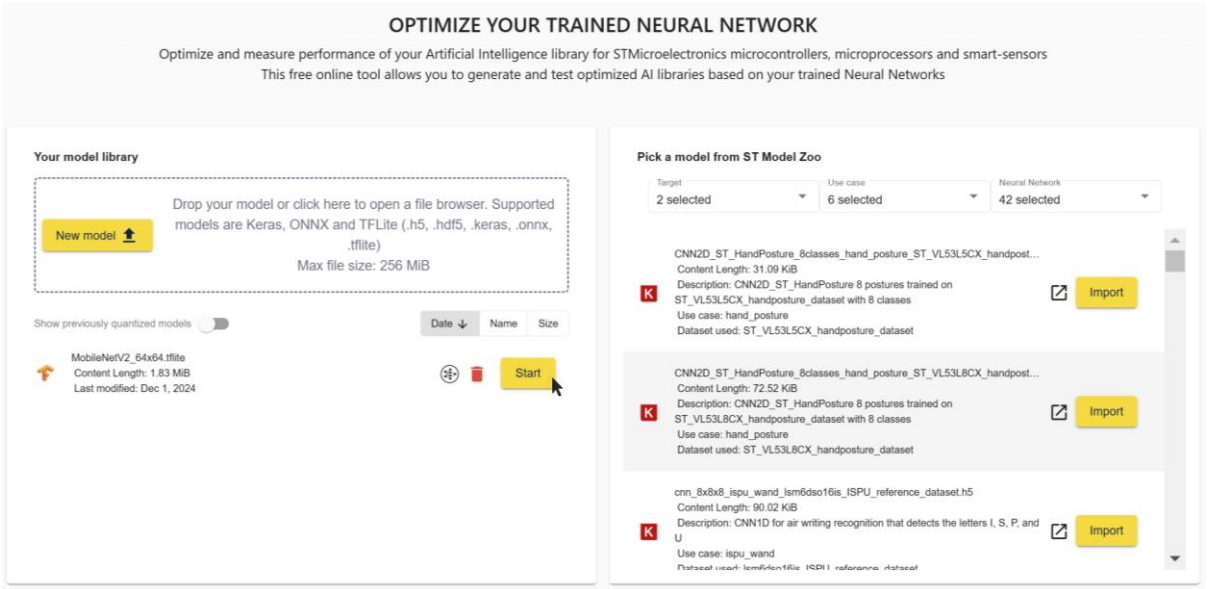

Figura 63:Caricamento del modello MobileNetV2_64x64..tflite e inizio nuovo progetto

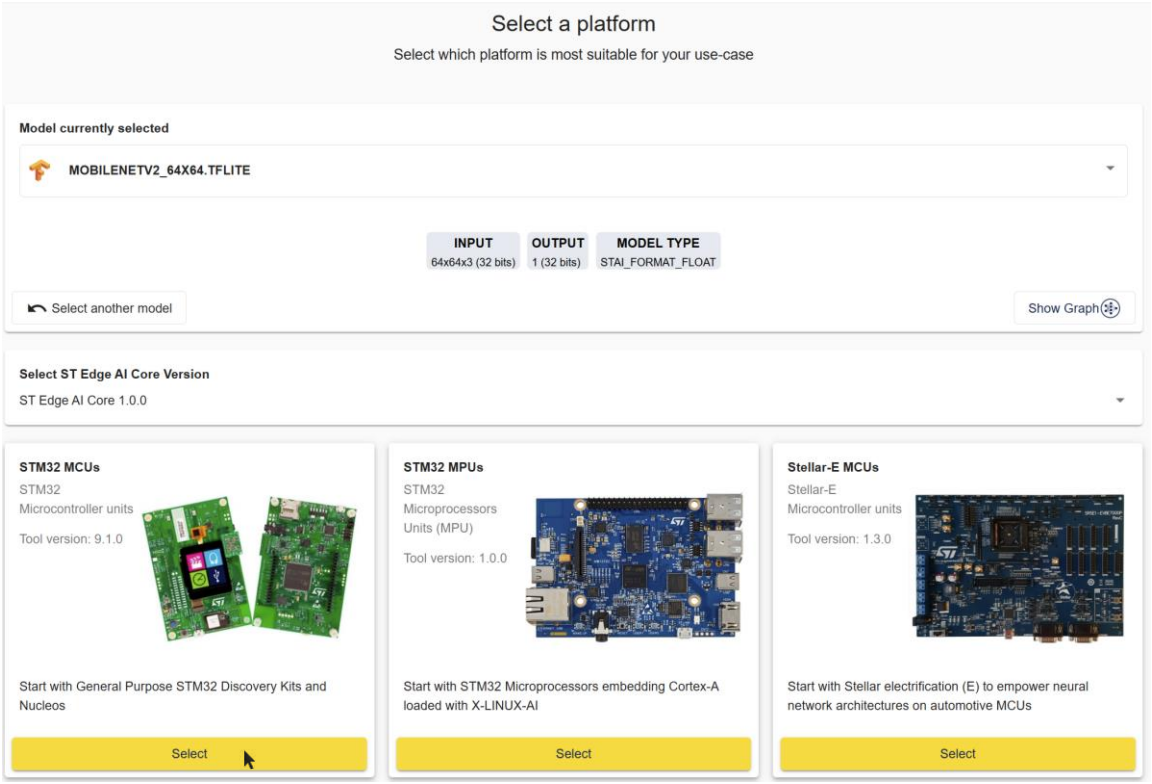

Figura 64: Selezione dei MCUs che contengono il microcontrollore STM32.

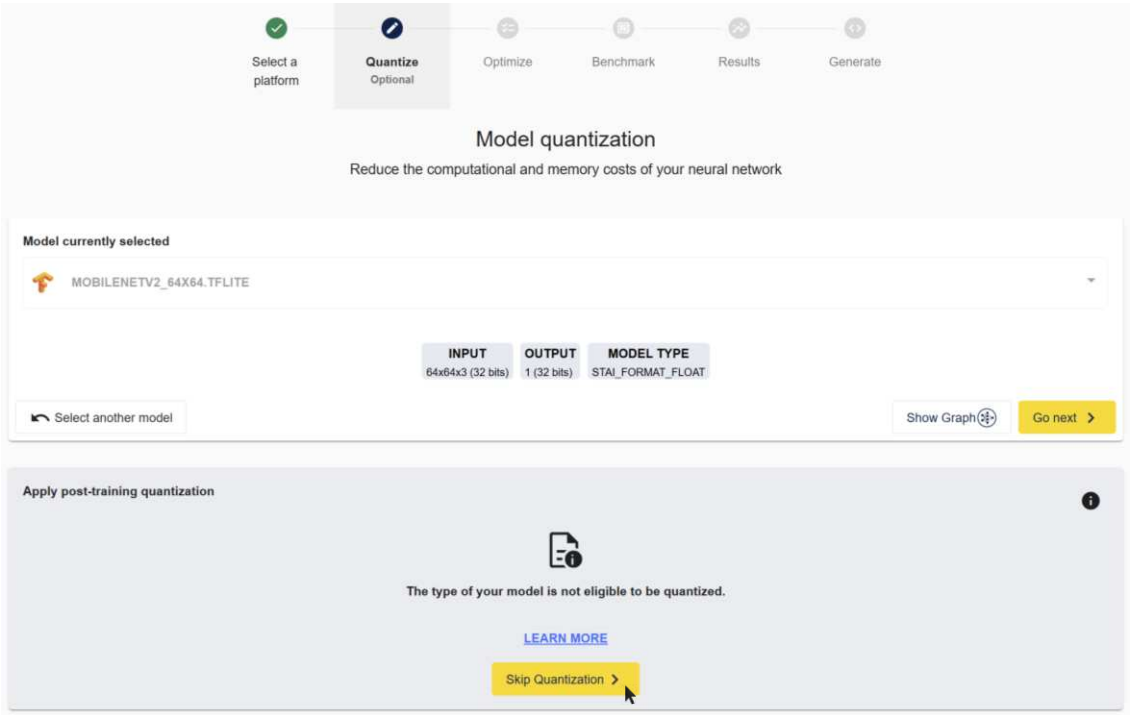

Figura 65: Salto della quantizzazione

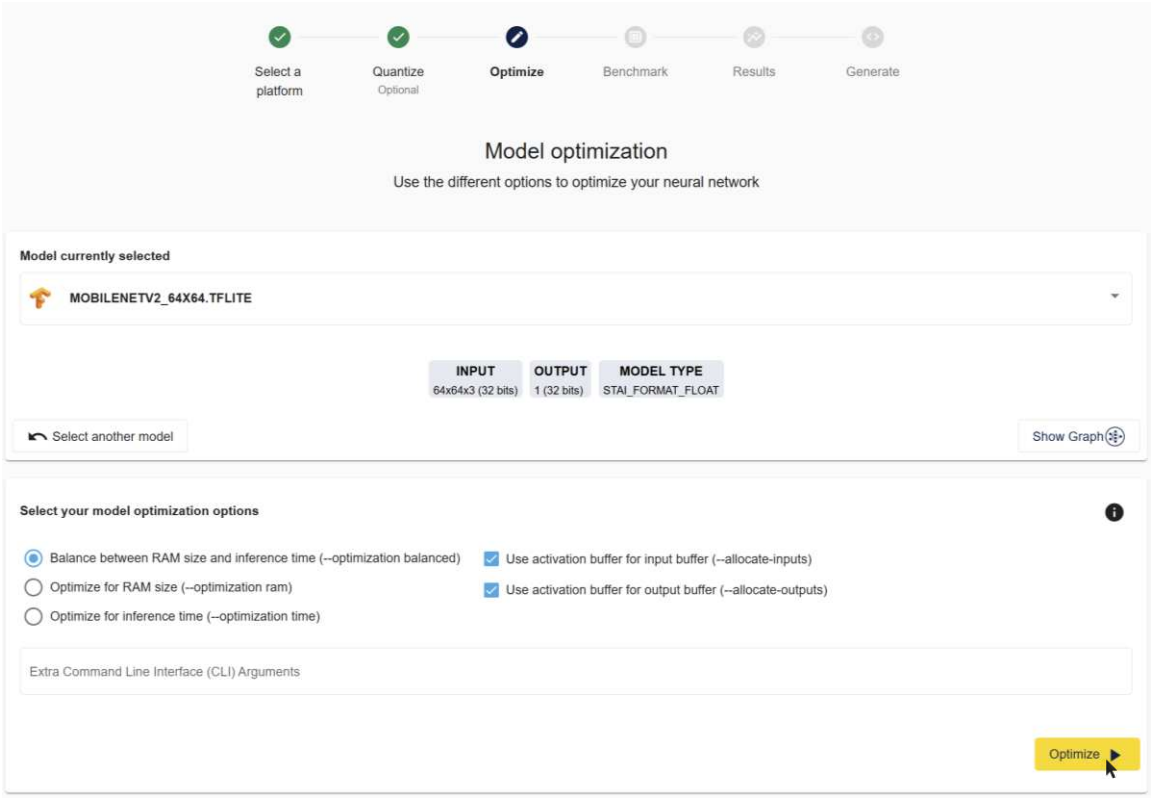

Figura 66: Avvio della fase di ottimizzazione del modello per la scheda STM32H7S78-DK

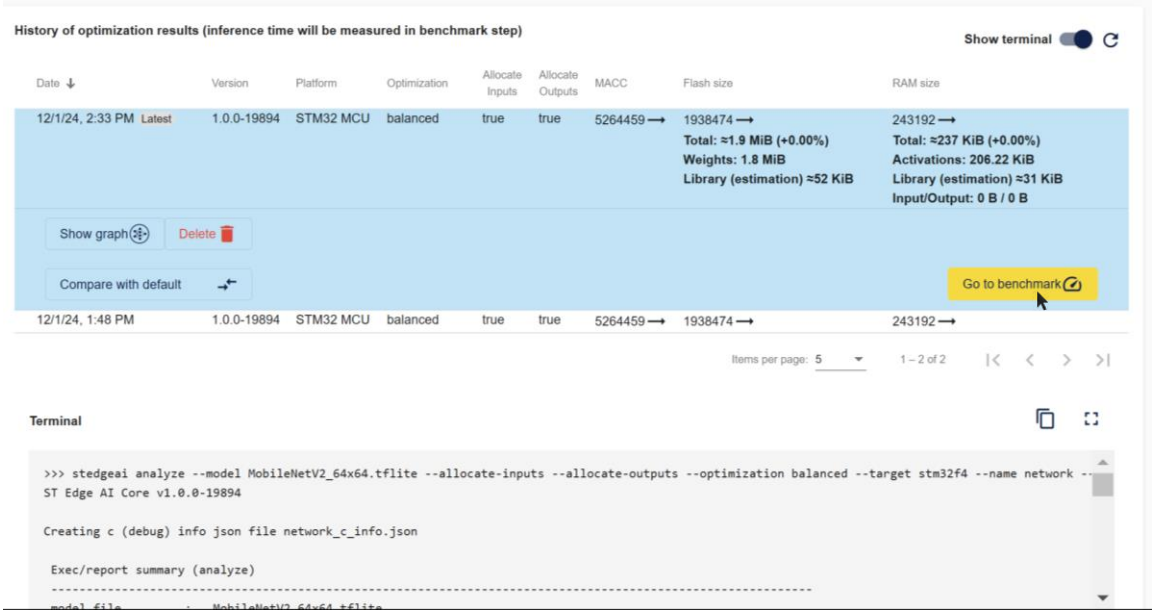

Figura 67: Passaggio alla pagina dedicata al benchmark

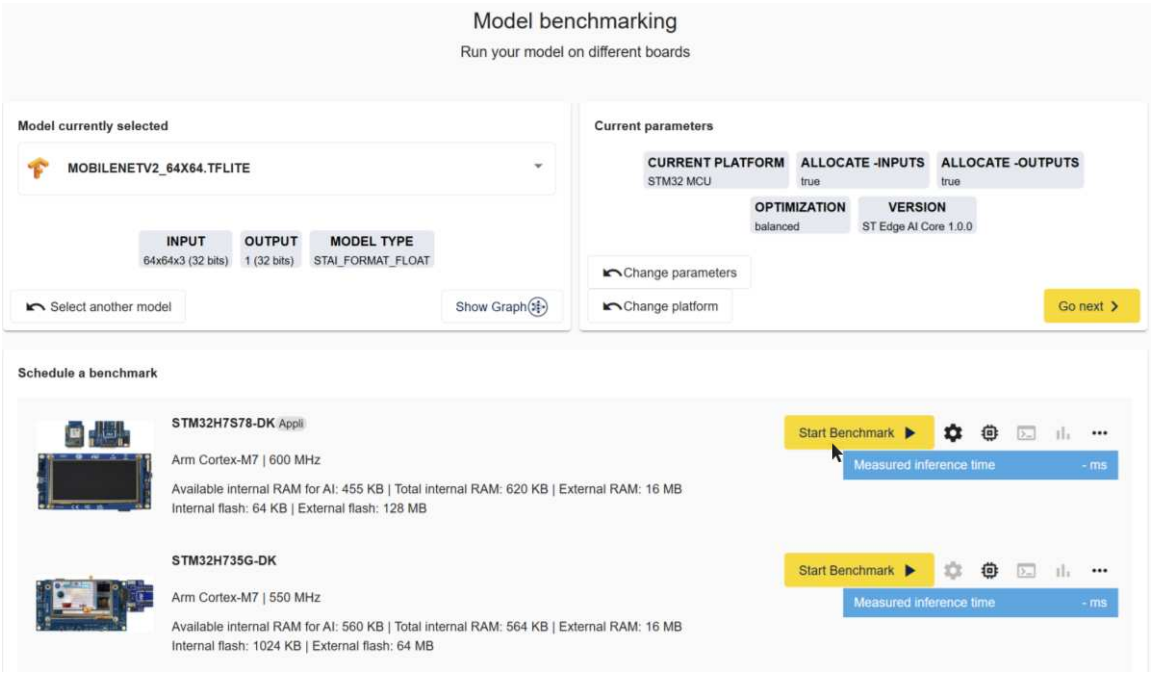

Figura 68: Avvio del Benchmark

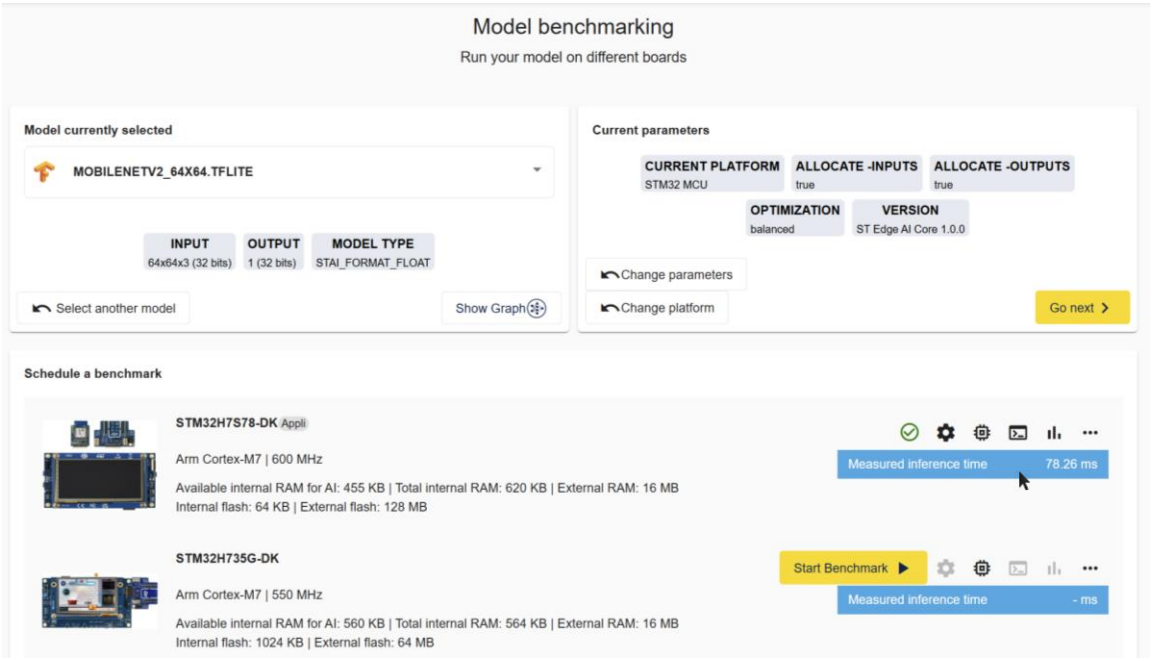

Figura 69: Misura del tempo di inferenza (in millisecondi) del modello applicato alla board STM32H7S78-DK

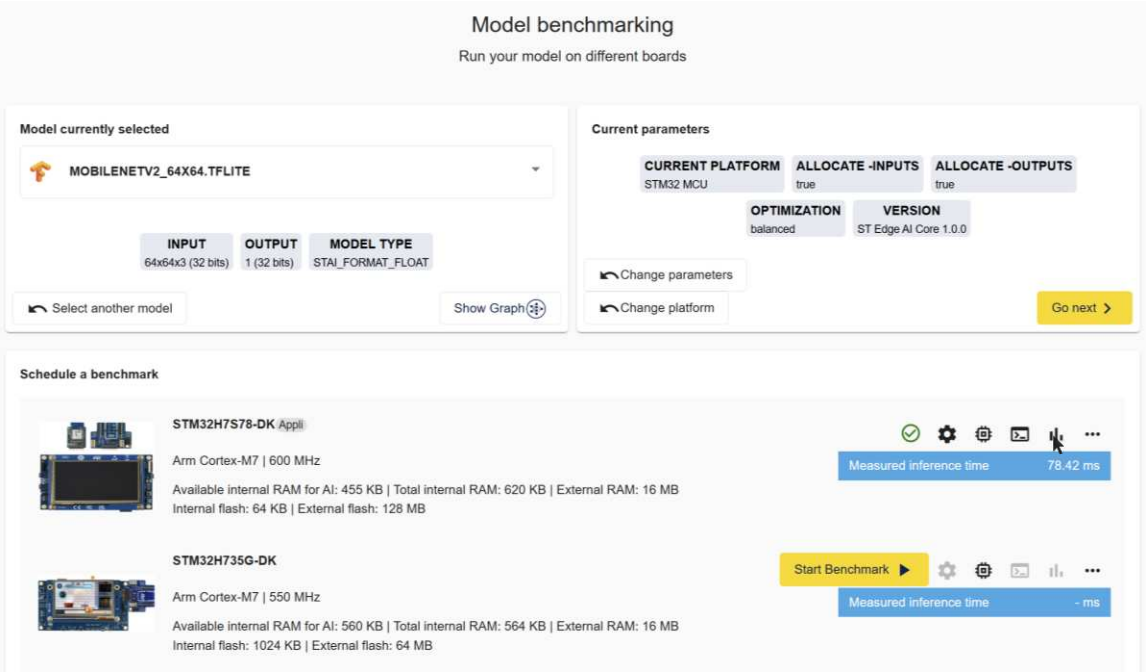

Figura 70: Passaggio ai grafici che rappresentano l'occupazione di memoria e il tempo di inferenza in ogni strato.

Figura 71: Occupazione della memoria flash in ogni strato del modello

Figura 72: Grafico che mostra il tempo di esecuzione in ogni strato della rete.

7. CONCLUSIONI E SVILUPPI FUTURI

In questa tesi, è stata condotta un'analisi comparativa di diverse architetture di reti neurali convoluzionali (CNN) allo scopo di sviluppare un sistema di classificazione binaria in grado di rilevare la presenza di buche sulle strade.

Il dataset impiegato per addestrare e valutare i modelli è stato reperito sulla piattaforma Kaggle e, data la sua dimensione limitata, è stato sottoposto ad un processo di data augmentation per aumentarne la variabilità e migliorare la generalizzazione del modello.

Le sperimentazioni sono state condotte su Google Colaboratory, utilizzando architetture pre-addestrate come VGG16, EfficientNetB0, ResNet50 e MobileNetV2. Per ciascuna di esse, è stata mantenuta invariata la struttura esterna, esaltando così la valutazione delle prestazioni ottenibili con e senza l'applicazione di tecniche di data augmentation.

Dai risultati ottenuti, è emerso che l'architettura MobileNetV2 si è dimostrata la più adatta per l'implementazione su sistemi embedded dato il minor tempo di inferenza, la minor occupazione di memoria e la sua elevata accuratezza applicando la data augmentation al dataset delle immagini.

Successivamente è stata ridotta la risoluzione delle immagini in ingresso ed il numero di parametri del modello MobileNetV2 per poter effettuare il porting su board STM32H7S78-DK.

Tramite la piattaforma online STM32Cube.AI Developer Cloud si è infine determinato il tempo di inferenza e l’occupazione di memoria associata alle operazioni eseguite su ogni strato del modello CNN sulla board STM32H7S78-DK.

Tale tesi può far da base a sviluppi futuri mirati all'applicazione di tecniche di compressione più complesse già accennate (come, ad esempio, la decomposizione tensoriale e il pruning) per ottimizzare il porting della CNN su microcontrollore.

BIBLIOGRAFIA

Bathula, Chandra Prakash. «History Of Neural Networks and Deep Learning». *Medium* (blog), 22 settembre 2024. https://medium.com/@chandu.bathula16/history-of-neural-networks-and-deep-learning-c25d3b1261a2.

Chelliah, Indhumathy. «Confusion Matrix — Clearly Explained». *All About AI-ML* (blog), 23 dicembre 2020. https://indhumathychelliah.com/2020/12/23/confusion-matrix%e2%80%8a-%e2%80%8aclearly-explained/.

«Google Colab». Consultato 26 novembre 2024. https://colab.research.google.com/.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, e Jian Sun. «Deep Residual Learning for Image Recognition». arXiv, 10 dicembre 2015. https://doi.org/10.48550/arXiv.1512.03385.

I, Azeem-. «Loss Functions in Deep Learning». *Medium* (blog), 2 ottobre 2024. https://medium.com/@ibtedaazeem/loss-functions-in-deep-learning-e4bd353ea08a.

«Kaggle: Your Machine Learning and Data Science Community». Consultato 26 novembre 2024. https://www.kaggle.com/.

KDnuggets. «Image Classification with Convolutional Neural Networks (CNNs)». Consultato 26 novembre 2024. https://www.kdnuggets.com/image-classification-with-convolutional-neural-networks-cnns.

«Keras: Deep Learning for humans». Consultato 26 novembre 2024. https://keras.io/.

Keylabs: latest news and updates. «Data Augmentation for Improving ImageClassification Accuracy», 28 agosto 2024. https://keylabs.ai/blog/data-augmentation-for-improving-image-classification-accuracy/.

McCulloch, Warren S., e Walter Pitts. «A Logical Calculus of the Ideas Immanent in Nervous Activity». *The Bulletin of Mathematical Biophysics* 5, fasc. 4 (1 dicembre 1943): 115–33. https://doi.org/10.1007/BF02478259.

Peterson, Hannah. «An Overview of Model Compression Techniques for Deep Learning in Space». *GSI Technology* (blog), 10 settembre 2020. https://medium.com/gsi-technology/an-overview-of-model-compression-techniques-for-deep-learning-in-space-3fd8d4ce84e5.

Rosenblatt, F. «The perceptron: A probabilistic model for information storage and organization in the brain». *Psychological Review* 65, fasc. 6 (1958): 386–408. https://doi.org/10.1037/h0042519.

Sandler, Mark, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, e Liang-Chieh Chen. «MobileNetV2: Inverted Residuals and Linear Bottlenecks». arXiv, 21 marzo 2019. https://doi.org/10.48550/arXiv.1801.04381.

Simonyan, Karen, e Andrew Zisserman. «Very Deep Convolutional Networks for Large-Scale Image Recognition». arXiv, 10 aprile 2015. https://doi.org/10.48550/arXiv.1409.1556.

«ST Edge AI Developer Cloud». Consultato 1 dicembre 2024. https://stedgeai-dc.st.com/home.

«stm32h7s78-dk.pdf». Consultato 1 dicembre 2024. https://www.st.com/resource/en/data_brief/stm32h7s78-dk.pdf.

Tan, Mingxing, e Quoc V. Le. «EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks». arXiv, 11 settembre 2020. https://doi.org/10.48550/arXiv.1905.11946.

TensorFlow. «TensorFlow». Consultato 26 novembre 2024. https://www.tensorflow.org/?hl=it.

«Understanding Activation Functions in Neural Networks». Consultato 26 novembre 2024. https://pareto.ai/blog/activation-function-in-neural-networks.

Yennhi95zz. «#4. A Beginner's Guide to Gradient Descent in Machine Learning». *Medium* (blog), 31 maggio 2023. https://medium.com/@yennhi95zz/4-a-beginners-guide-to-gradient-descent-in-machine-learning-773ba7cd3dfe.

ZOMEV. «Deep Neural Network (DNN) Explained». *Medium* (blog), 14 aprile 2024. https://medium.com/@zomev/deep-neural-network-dnn-explained-0f7311a0e869.