UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI INGEGNERIA

Corso di Laurea Triennale in Ingegneria Elettronica

Dipartimento di Ingegneria dell'informazione

Implementazione di una CNN per la classificazione di crepe all'esterno degli edifici su piattaforma embedded OpenMV Cam

Implementation of a CNN for crack classification on OpenMV Cam embedded platform

Relatore:

Prof. Claudio Turchetti

Tesi di Laurea di:

Mattia Beccerica

Correlatore:

Prof.ssa Laura Falaschetti

Anno Accademico 2020 – 2021

INDICE

CAPITOLO 1
INTRODUZIONE

Tutte le strutture sono soggette a pressioni, fatica, sforzi e usura che con il tempo combinandosi tra loro spesso portano a danneggiare la struttura provocando crepe, lesioni, incrinature, alterazioni cromatiche, ecc. La presenza di crepe è un importante segnale di degradazione della struttura e per questo è importante fare periodiche manutenzioni e controlli su di esse, per evitare ulteriori peggioramenti irreversibili. Attualmente l'ispezione manuale è il metodo più utilizzato per il rilevamento delle crepe però è un metodo poco oggettivo perchè dipende dall'operatore che la fa, in più richiede conoscenza, esperienza e tempo.

Il metodo proposto in questa tesi vuole semplificare questa procedura, infatti è un metodo più automatizzato e veloce che sfrutta l'intelligenza artificiale. L'intelligenza artificiale (o IA, dalle iniziali delle due parole, in italiano) è una disciplina appartenente all'informatica che studia i fondamenti teorici, le metodologie e le tecniche che consentono la progettazione di sistemi hardware e sistemi di programmi software capaci di fornire all'elaboratore elettronico prestazioni che, a un osservatore comune, sembrerebbero essere di pertinenza esclusiva dell'intelligenza umana. Si tratta di una disciplina oggi molto utilizzata in diversi settori della vita quotidiana con alla base tre parametri che rappresentano i cardini del comportamento umano, ossia: una conoscenza non sterile, una coscienza che permetta di prendere decisioni non solo secondo la logica e l'abilità di risolvere problemi in maniera differente anche a seconda dei contesti nei quali ci si trova.

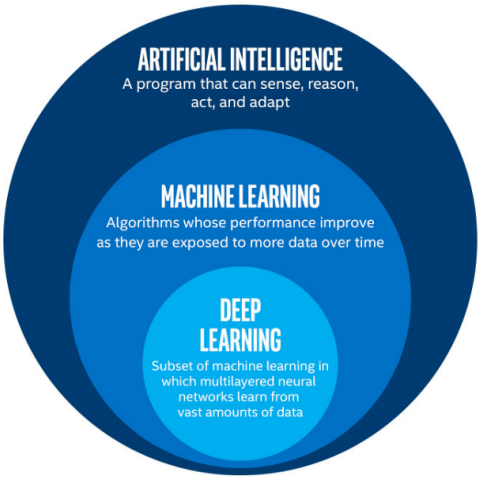

figura 1.1: sottoinsiemi dell'intelligenza artificiale[1]

Il machine learning è un sottoinsieme dell'intelligenza artificiale e consiste nelle tecniche che consentono ai computer di capire le cose dai dati e fornire applicazioni di intelligenza artificiale. Il deep learning, invece, è un sottoinsieme del machine learning che consente ai computer di risolvere problemi più complessi.

[1] https://it.naneedigital.com/article/the_difference_between_machine_learning_ai_and_deep_learning

Scopo di questa tesi è quindi quello di velocizzare il processo di rilevamento crepe e perciò, verranno progettate, discusse e confrontate le prestazioni di tre diverse reti neurali convoluzionali (CNN) che permettono il loro rilevamento attraverso “image processing” ottenendo risultati accurati come i metodi tradizionali.

Le reti progettate vengono poi implementate e testate sulla piattaforma embedded “OpenMV Cam H7 Plus” dotata di una camera che ci permette di acquisire le immagini da classificare.

Le CNN sono state progettate con l’obiettivo di essere il più “leggere” possibile, per poter essere implementate in sistemi embedded con poca disponibilità computazionale e/o di memoria, e ricercando la maggiore accuratezza possibile.

In questa tesi verranno esposte le basi teoriche dell’intelligenza artificiale fino ad arrivare alla progettazione e al confronto di tre diverse reti neurali convoluzionali e infine all’implementazione nella piattaforma embedded OpenMv Cam, concludendo con delle considerazioni riguardo le prestazioni ottenute.

La tesi nasce dall’esperienza del tirocinio svolto in università e descrive il percorso fatto per risolvere il problema della classificazione delle crepe in un edificio.

CAPITOLO 2
MACHINE LEARNING

Il “Machine Learning” è un sottoinsieme dell’intelligenza artificiale e consiste nell’utilizzare algoritmi per analizzare dati, imparare da essi e poi fare deduzioni o previsioni su nuovi dati, sulla base dei precedenti.
Con il machine learning, invece di scrivere codice manualmente con uno specifico set di istruzioni per realizzare uno specifico task, la macchina viene “allenata” usando dati e algoritmi che gli danno l’abilità di svolgere la stessa task precedente senza però essere espliciti su quali passi svolgere per realizzarla.

2.1 DEEP LEARNING

“Deep learning” è, invece, un sottoinsieme del machine learning che usa algoritmi e modelli ispirati dalla struttura e funzionamento della rete neurale del nostro cervello.
Le reti neurali che vengono usate in deep learning non sono esattamente come quelle del nostro cervello, infatti condividono solo alcune caratteristiche e per questo vengono chiamate reti neurali “artificiali”.

2.2 RETI NEURALI ARTIFICIALI (ANN)

Una rete neurale artificiale è un modello computazionale composto da un insieme di “unità”, chiamate neuroni o anche “nodi”, connesse tra loro e organizzate in strati detti “layers”. Ogni connessione tra neuroni trasmette un segnale da un neurone a un altro. Il neurone ricevitore processa il segnale e lo ritrasmette ai successivi fino all’ultima unità della rete.

A livello più alto, ci sono 3 tipi di layers in tutte le ANN:

1. Input layer
2. Hidden layers (layers nascosti)
3. Output layer

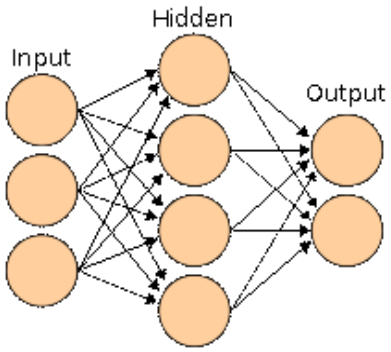

figura 2.1: rete neurale artificiale[2]

[2] https://it.wikipedia.org/wiki/Rete_neurale_artificiale

Diversi tipi di layers svolgono diversi tipi di trasformazioni sugli inputs. I dati scorrono attraverso la rete partendo dall'input layer, muovendosi lungo gli hidden layers fino a raggiungere l'output layer.

2.3 LAYERS

In una rete neurale artificiale i neuroni sono organizzati in layers. Esistono diversi tipi di layers, ad esempio:

- Dense (or fully connected) layers
- Convolutional layers
- Pooling layers
- Recurrent layers
- Normalization layers

Ogni tipo di layer svolge diverse trasformazioni ai loro input e alcuni si adattano meglio in specifiche situazioni rispetto ad altri. Per esempio, un "convolutional layer" è usualmente utilizzato in reti che lavorano con immagini in input; "recurrent layers" vengono usati per lavorare con dati statistici mentre "dense layers" sono dei layers in cui tutti gli input vengono collegati a tutti gli output.

Ogni connessione tra due nodi ha un peso associato, che consiste in un numero. Ogni peso rappresenta la forza della connessione tra due nodi. In pratica quando la rete riceve un input in un nodo, esso verrà trasferito al nodo successivo tramite una connessione e moltiplicato per il peso associato a quella connessione.

La somma di tutti gli input pesati in un nodo è poi passata a una "Activation function" (o "funzione di attivazione") che fa una trasformazione della sommatoria dipendente dal tipo di funzione usata.

output = activation(somma pesata degli inputs)

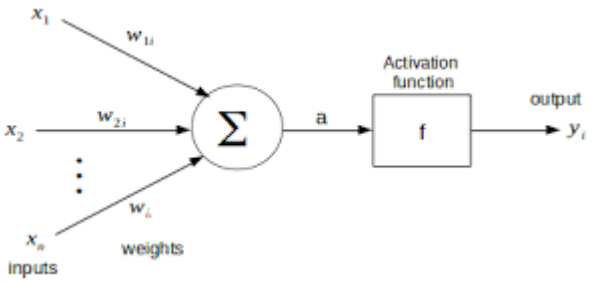

figura 2.2: rappresentazione neurone artificiale[3]

dove la funzione di attivazione può essere scritta come:

$$y_i = f(\sum_{j=1}^{n} w_{ji} x_j)$$

[3] https://pierandreamirino.com/reti-neurali-artificiali-con-nest/

Ottenuto l'output di uno specifico nodo, questo valore sarà l'input passato al nodo del layer successivo. Questo processo si ripete per ogni nodo e continua fino a raggiungere l'ultimo layer denominato output layer.

Il numero di nodi dell'output layer dipende dal numero dei possibili output che possiamo avere, mentre il numero di nodi dell'input layer e degli hidden layers può essere arbitrario.

L'intero processo in cui in cui i dati scorrono dall'input layer fino ad arrivare all'output layer è chiamato "forward pass" della rete.

2.4 FUNZIONE DI ATTIVAZIONE

In una rete neurale artificiale, una funzione di attivazione è una funzione che attribuisce a un input di un nodo, il suo corrispondente output.
Molto spesso la funzione di attivazione è una funzione non lineare che trasforma la sommatoria pesata degli input in un numero, limitato a un range, definito dalla funzione.
La non linearità permette alla rete neurale di elaborare funzioni complesse arbitrariamente grazie al restringimento dell'input in un range più piccolo e dona alla rete la capacità di riconoscere "pattern" non lineari.
Questo tipo di operazione si ispira al funzionamento del nostro cervello e da come i nostri neuroni reagiscono quando stimolati da diversi tipi di stimoli, infatti normalmente non utilizziamo tutti i neuroni contemporaneamente ma in base allo stimolo si attivano determinati neuroni, allo stesso modo succede nei nodi di un layer grazie alla funzione di attivazione.

Esistono diverse tipologie di funzioni di attivazione che si differenziano per il tipo di trasformazione che eseguono ai loro input, nel nostro caso abbiamo utilizzato:

- ReLU (Rectified Linear Unit): funzione molto utilizzata soprattutto nelle reti neurali convoluzionali che trasforma l'input in zero o lo lascia invariato se maggiore di zero, in formula:

$$relu(x) = max(0, x)$$

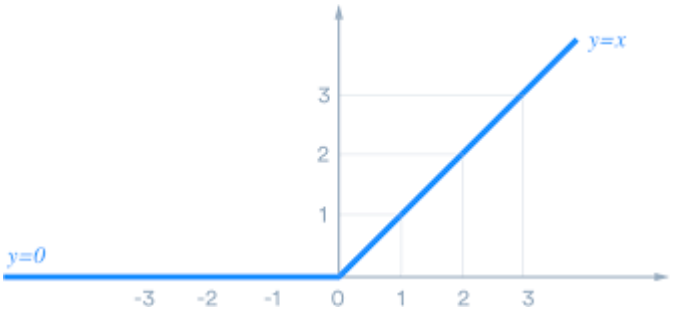

figura 2.3: funzione di attivazione ReLU[4]

[4] https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7

Quindi più la sommatoria in ingresso è grande e più il nodo è attivo; se invece l'input è negativo, il nodo sarà spento e non darà contributo ai layer successivi.

- Softmax: funzione utilizzata, di solito, nell'ultimo layer di una rete per normalizzare l'output in un range tra [0,1]. Esempio:

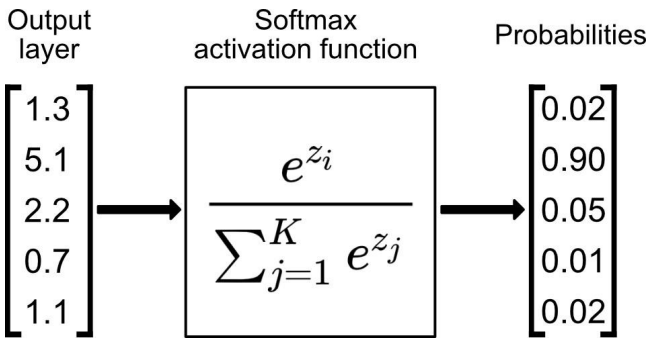

figura 2.4: applicazione della funzione di attivazione Softmax[5]

2.5 COME AVVIENE L'ADDESTRAMENTO DI UNA RETE NEURALE

Definita la rete con il numero di layers, numero di unità per ogni layer e funzioni di attivazione, il processo di "training" (o addestramento) si può definire come un problema di ottimizzazione infatti durante questo processo l'obiettivo è quello di trovare il migliore peso per ogni collegamento tra neuroni che permetta la giusta classificazione dell'input al suo corretto output.

La ricerca del valore ottimale di peso, durante la fase di training, viene fatta in modo iterativo e sfrutta quello che viene chiamato "algoritmo di ottimizzazione" (o "optimizer"). Scopo di questo algoritmo è quello di trovare i giusti parametri che permettano di minimizzare un'altra funzione detta "funzione di perdita" (o "loss function"), quindi riassumendo, la funzione di ottimizzazione, al passaggio di ogni dato nella rete, aggiorna il valore dei pesi di ogni collegamento per fare in modo di minimizzare il più possibile la funzione di perdita.

La funzione di perdita si può definire, in generale, come l'errore o la differenza tra cosa la rete predice per un determinato input e la vera "label" dello stesso input.

Questa ricerca del migliore peso per ogni collegamento, si ripete per ogni dato che scorre nella rete e viene definita "training", perchè è proprio in questo processo dove la rete "impara" come classificare gli attuali e i futuri dati.

[5] https://towardsdatascience.com/softmax-activation-function-explained-a7e1bc3ad60

2.5.1 ALGORITMO DI OTTIMIZZAZIONE

Esistono diversi tipi di algoritmi di ottimizzazione che permettono di ottimizzare il modello attraverso tecniche diverse, nel nostro caso abbiamo utilizzato l'algoritmo "Adadelta" che si basa sul metodo della discesa stocastica del gradiente (stochastic gradient descent, SGD). Nel SGD, dopo aver calcolato la funzione di perdita dell'output, viene calcolato il suo gradiente rispetto a ogni peso dentro la rete:

$$\frac{d\,(loss)}{d\,(weight)}$$

Questo valore viene utilizzato per aggiornare i pesi del modello infatti il gradiente dice in quale direzione la perdita deve muoversi per avvicinarsi al minimo valore e il nostro obiettivo è proprio quello di diminuire il più possibile la funzione di perdita.
Oltre al gradiente, viene utilizzato un altro parametro denominato "learning rate", un numero molto piccolo, arbitrario, che varia da 0.01 a 0.0001, di solito, che ci dice quanto grande è lo step che facciamo per avvicinarsi nella direzione del minimo.
L'algoritmo di ottimizzazione "Adadelta" si basa su un learning rate "adattivo" cioè può variare in base ai gradienti che vengono calcolati durante il training.

In conclusione, il nuovo peso sarà dato da:

$$\text{new weight} = \text{old weight} - (\text{learning rate} * \text{gradient})$$

Ovviamente il processo si ripete per ogni peso all'interno della rete per ogni dato che scorre attraverso il modello e più dati utilizziamo e più i pesi si avvicinano al loro valore ottimale grazie al SGD.

2.5.2 FUNZIONE DI PERDITA

La funzione di perdita ha lo scopo di calcolare quanto il modello è lontano dal classificare un input al suo vero output e ovviamente, questa quantità, dipende dalla funzione scelta. In questo caso è stata scelta la funzione di perdita "categorical_crossentropy" che viene usata per classificazioni multi-class, ovvero quando ci sono 2 o più categorie (o classi) in output ma dove solo una è quella giusta.

La perdita è calcolata dalla seguente sommatoria:

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

dove $\hat{y}_i$ è il valore i-th dell'output della rete, y_i è la sua corrispondente classe e "output size" è il numero di classi.

2.6 DATASET

Il dataset è un insieme di dati che possono essere sotto forma di vettori, immagini, suoni…, suddiviso in classi con ogni elemento associato ad una ed una sola di queste. Viene utilizzato in fase di addestramento per la ricerca dei migliori pesi di ogni collegamento e in fase di testing per la valutazione della correttezza dei risultati prodotti su nuovi dati; è estremamente importante che questi sia costruito in maniera corretta.

Di solito viene diviso in tre parti, di differenti dimensioni:

- Training set: spesso è il 70% dei dati del dataset totale ed è l'insieme di dati usato per l'addestramento della rete, influisce sulla classificazione dei dati che la rete non ha mai visto;
- Validation set: di solito il 15% del dataset totale, viene utilizzato durante il training per valutare la curva di apprendimento della rete;
- Test set: di solito il 15% del dataset totale, viene utilizzato quando la rete è stata addestrata per osservare come si comporta il modello su dati non appartenenti al training e validation set;

La costruzione di un buon dataset è alla base di una rete neurale efficace; in generale si può dire che un dataset è buono quando non presenta "overfitting" o "underfitting".

2.6.1 OVERFITTING

L'overfitting si presenta quando il modello è molto accurato nel classificare dati presenti all'interno del training set ma non è altrettanto accurato per input esterni ad esso, quindi il modello non è abile a generalizzare bene quello che ha imparato.

L'overfitting è un problema molto comune e si può ridurre utilizzando diverse tecniche:

- Aumentare il numero di immagini nel training set: in modo da aggiungere più diversità al dataset e migliorare l'addestramento della rete;
- "Data augmentation": metodo usato per aumentare il numero di campioni all'interno del training set, aggiungendo copie di campioni già presenti ma con piccole modifiche;
- Ridurre la complessità del modello: ad esempio rimuovendo qualche layer dal modello o riducendo il numero di neuroni per layer;
- Dropout: tipo di layer che se aggiunto al modello, ignora un numero casuale di nodi di uno specifico layer;

2.6.2 UNDERFITTING

L'underfitting si presenta quando il modello non è nemmeno in grado di classificare i dati presenti all'interno del training set e quindi non sarà nemmeno accurato nella classificazione dei dati nel testing set.
Anche l'underfitting si può ridurre o addirittura eliminare attraverso queste tecniche:

- Aumentare la complessità del modello: ad esempio aumentando il numero di layer nel modello, aumentando il numero di neuroni per ogni layer, cambiando il tipo e la posizione dei layer del modello;
- Utilizzare dataset migliori: cercare degli input che siano più significativi allo scopo della rete o ampliare il numero di campioni presenti nel dataset;
- Ridurre il dropout: se viene utilizzato dropout nella rete, ridurre il numero di nodi ignorati in uno specifico layer;

2.7 TECNICHE DI APPRENDIMENTO AUTOMATICO

Nelle reti neurali artificiali la classificazione, intesa come l'attività che si serve di un algoritmo statistico al fine di individuare una rappresentazione di alcune caratteristiche di un'entità da classificare (oggetto o nozione), associandole una etichetta classificatoria, può essere svolta mediante due tecniche di apprendimento automatico: supervisionato o non supervisionato.

2.7.1 APPRENDIMENTO SUPERVISIONATO

L'apprendimento supervisionato si verifica quando i dati con cui viene addestrata la rete sono etichettati, ovvero ogni input al modello è una coppia che consiste nel dato vero e proprio e della sua corrispondente etichetta (label) o valore di output.
Quindi il modello, con l'apprendimento supervisionato, impara come classificare un'entità grazie a questa corrispondenza input-etichetta presente in ogni dato in input.

2.7.2 APPRENDIMENTO NON SUPERVISIONATO

L'apprendimento non supervisionato, al contrario dell'apprendimento supervisionato, si verifica quando i dati con cui viene addestrata la rete non sono etichettati, quindi non c'è corrispondenza tra input e etichetta.
Con questa tecnica non è possibile misurare l'accuratezza della rete dato che non è possibile stabilire come la rete performa.
L'obiettivo di utilizzare input senza etichetta è quello di estrarre informazioni utili dalla struttura dei dati e da esse cercare di classificarli.

2.8 LEARNABLE PARAMETERS

Vengono definiti “learnable parameters”, tutti i parametri all’interno di una rete neurale artificiale che vengono aggiornati e ottimizzati durante il training grazie agli algoritmi di ottimizzazione e le funzioni di perdita.
Possono essere calcolati e dipendono principalmente dal numero di layer in una rete e dal numero di input e output di ogni layer.

2.9 EPOCHS E BATCH SIZE

Quando dobbiamo addestrare una rete neurale artificiale, vengono richiesti due parametri che influenzano la complessità del processo e quindi dipendono dalle risorse computazionali in possesso; essi sono: grandezza del “batch size” e numero di “epochs”.

Si definisce “batch size” il numero di campioni che scorrono nella rete contemporaneamente.

Si definisce “epochs” il numero di volte che l’intero dataset attraversa la rete durante il processo di training.

Generalmente, più grande è il batch size e più veloce il modello completerà una epoch, al costo, però, di un peggioramento della qualità del modello perché sarà meno abile a classificare dati non presenti nel dataset di addestramento.

CAPITOLO 3
RETI NEURALI CONVOLUZIONALI (CNN)

Una rete neurale convoluzionale, anche detta CNN o ConvNet, è una rete neurale artificiale che viene utilizzata principalmente per l'analisi di immagini, infatti esse sono particolarmente utili per individuare pattern nelle immagini per il riconoscimento di oggetti, volti e scene ma possono essere efficaci anche per la classificazione di dati non immagine come dati audio, serie storiche e segnali.

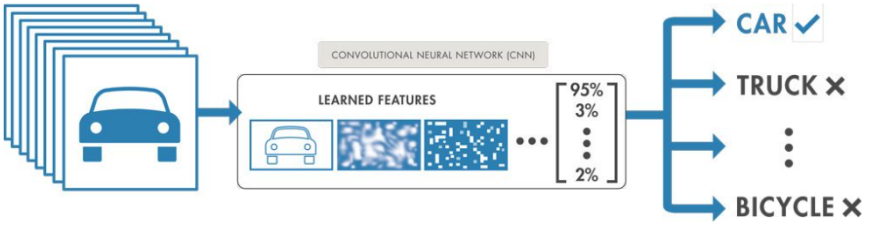

figura 3.1: workflow del deep learning[6]

L'utilizzo delle CNN per il deep learning è diffuso per via di tre fattori importanti:

- Le CNN eliminano la necessità di estrarre manualmente le feature in quanto queste vengono apprese direttamente dalla CNN;
- Le CNN producono risultati di riconoscimento ad alta precisione;
- Le CNN possono essere addestrate nuovamente per nuove attività di riconoscimento, consentendo agli utenti di basarsi sulle reti pre-esistenti;

Analogamente ad altre reti neurali, una CNN è costituita da un layer di input, un layer di output e layers intermedi nascosti. Questi layer intermedi eseguono operazioni che alterano i dati al fine di apprendere le feature specifiche dei dati stessi.
I tipi di layer più utilizzati e diffusi sono:

- Convoluzionali
- Pooling
- Flatten

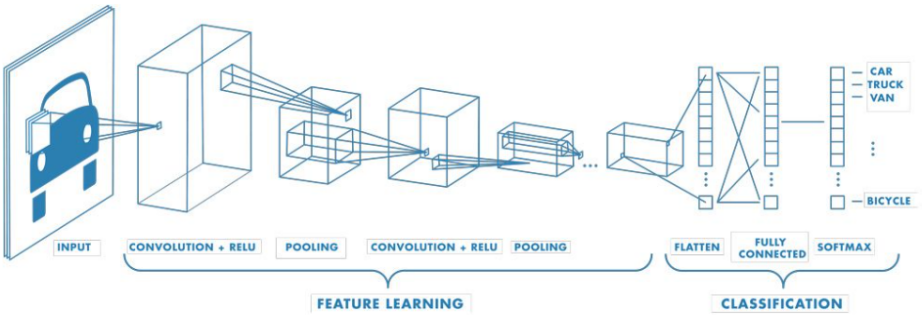

figura 3.2: rete neurale convoluzionale[7]

[6] https://it.mathworks.com/discovery/convolutional-neural-network-matlab.html
[7] https://it.mathworks.com/discovery/convolutional-neural-network-matlab.html

Queste operazioni vengono reiterate su decine o centinaia di layer e ciascun layer impara ad identificare feature diverse.

3.1 LAYERS CONVOLUZIONALI

La differenza delle reti neurali convoluzionali con le semplici reti neurali artificiali, è la presenza di layers chiamati layer “convoluzionali” da cui deriva anche il nome.

La trasformazione che avviene in questo tipo di layer è chiamata “convoluzione” e lo scopo di questa operazione è quello di estrarre, dalle immagini, le caratteristiche dell’oggetto da classificare, in modo che la rete sia in grado di riconoscerle ovunque e in tutte le immagini date in input.

L’operazione di convoluzione si può semplificare come un filtro (anche detto kernel) applicato all’input, dove il filtro si può considerare come una matrice di dimensione arbitraria con valori inizializzati con numeri casuali. Esempio:

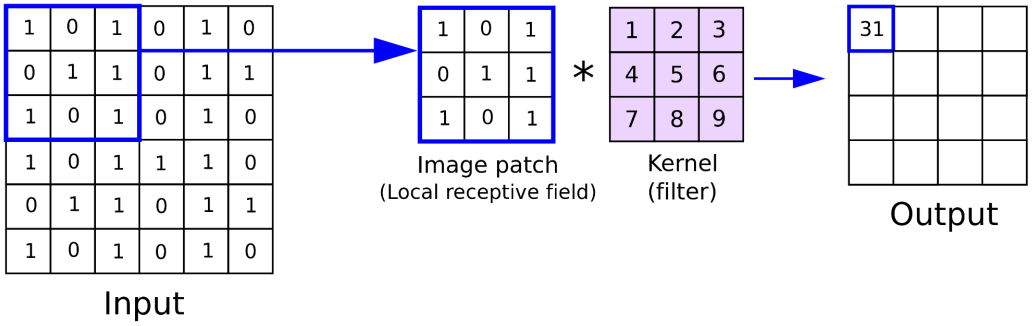

figura 3.3: operazione di convoluzione[8]

Più precisamente, il filtro viene spostato sopra l'immagine da sinistra a destra, dall'alto in basso, con un cambio di colonna di un pixel nei movimenti orizzontali, poi un cambio di riga di un pixel nei movimenti verticali. La quantità di movimento tra le applicazioni del filtro all'immagine in ingresso è chiamata “stride”, è un parametro modificabile e di solito simmetrico nelle dimensioni di altezza e larghezza; nel nostro caso (1,1).

Ogni volta che il filtro scorre in una porzione diversa di immagine, viene calcolata la convoluzione tra la matrice che corrisponde alla porzione di immagine considerata e il filtro; questa operazione corrisponde al prodotto delle due matrici, che viene poi sommato, ottenendo un unico valore. Quando il filtro è stato convoluto con l’intero input, otteniamo una nuova rappresentazione dell’input, salvata nella matrice di output, che viene chiamata “feature map” (o mappa delle caratteristiche). Quindi, sintetizzando, le reti neurali convoluzionali applicano un filtro a un input per creare una mappa di caratteristiche che riassume la presenza di caratteristiche rilevate nell'input.

[8] https://anhreynolds.com/blogs/cnn.html

Ovviamente più la rete è complessa, più numerosi sono i layer convoluzionali e più sofisticati diventano i filtri degli strati successivi, quindi invece di estrarre semplici caratteristiche come forme e linee, i filtri degli strati più profondi sono in grado di individuare anche oggetti specifici.

In ogni layer convoluzionale, non c'è solo un singolo filtro ma un numero arbitrario che permette di imparare più caratteristiche in parallelo alla rete; il numero di filtri presenti determina il numero di output del layer.
E' importante che il filtro abbia lo stesso numero di canali dell'input cioè se l'input ha 3 canali (RGB) anche il filtro deve avere 3 canali. L' operazione di convoluzione rimane invariata per più canali.

Nella figura 3.4 c'è un esempio di convoluzione in cui l'obiettivo è quello di identificare l'uno; si può notare come l'immagine di uscita sia molto diversa da quella di ingresso.

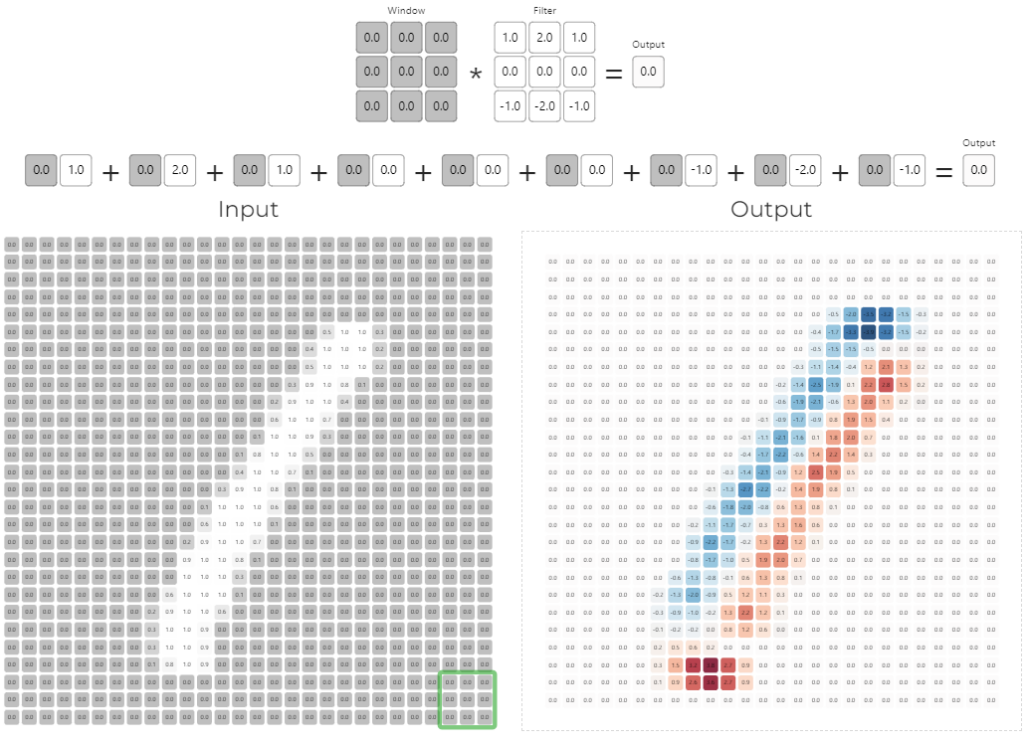

figura 3.4: risultato dell'applicazione di un layer convoluzionale[9]

Si può osservare come, a causa di questa operazione, l'output sia una matrice di dimensioni minori rispetto a quella di input, infatti, data una immagine in input di dimensioni "nxn" che viene convoluta con un filtro di dimensioni "fxf", si ottiene una matrice di output di dimensioni "(n-f+1)x(n-f+1)".

Importante sottolineare che i valori della matrice del filtro sono inizialmente casuali e che durante la fase di addestramento della rete, vengono ottimizzati automaticamente dalla rete per poter estrarre le caratteristiche che ne permettono una corretta classificazione.

[9] https://deeplizard.com/resource/pavq7noze2

3.1.1 PADDING

Il “padding” è una tecnica che nasce come soluzione al problema della riduzione di dimensione della matrice di output dopo la convoluzione.

Questo tipo di problema non sembra essere rilevante per grandi immagini ma quando abbiamo input di piccole dimensioni o con informazioni utili ai bordi, tagliare anche una piccola porzione potrebbe essere distruttivo per la rete.
Inoltre sarebbe ancora più difficile applicare più di una volta la convoluzione allo stesso input dato che l’output diventerebbe sempre più piccolo.

L'aggiunta del padding permette lo sviluppo di modelli molto profondi anche su input di piccole dimensioni in modo tale che le mappe delle caratteristiche non si riducano nel corso dello sviluppo.

Il padding (zero-padding in questo caso) consiste nell’aggiunta di pixel di valore zero al bordo dell’immagine. Essi non hanno alcun effetto sull’operazione di convoluzione quando il filtro è applicato, se non quello di preservare la dimensione originaria dell’input.

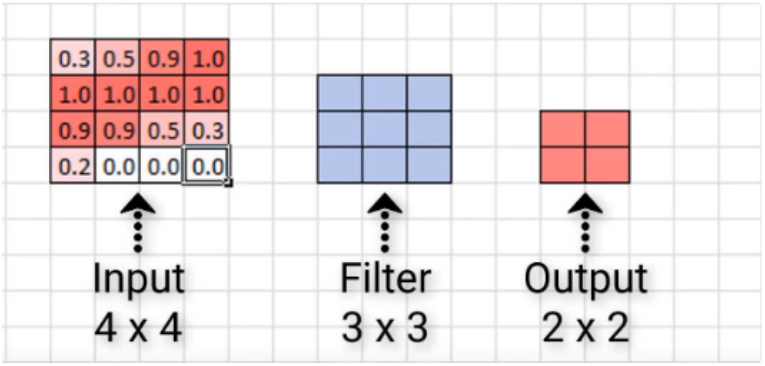

figura 3.5: convoluzione senza padding[10]

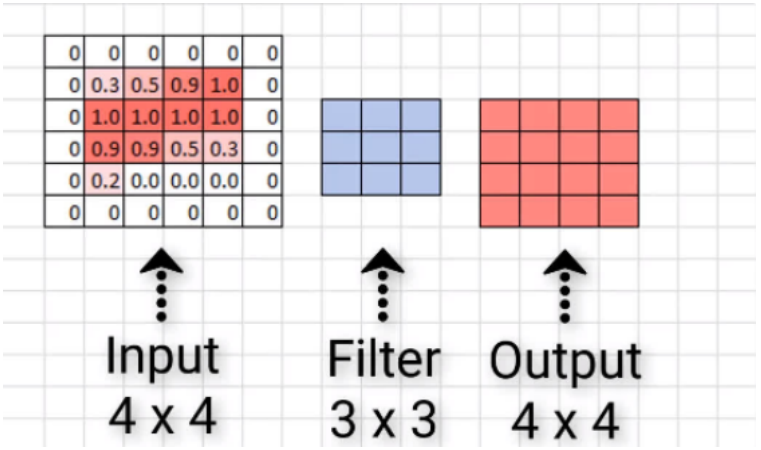

figura 3.6: convoluzione con padding[11]

[10] https://deeplizard.com/learn/video/qSTv_m-KFk0
[11] https://deeplizard.com/learn/video/qSTv_m-KFk0

Come si può vedere dalle figure 3.5 e 3.6, nella prima non viene fatto padding all'input e con un filtro 3x3 otteniamo un output di dimensione 2x2; nella seconda, invece, grazie all'aggiunta di un bordo di zero intorno all'input, con lo stesso filtro, otteniamo un output della stessa dimensione dell'input senza il padding.

Ci sono due tipologie di padding:

- "valid": non viene applicato padding
- "same": vengono aggiunti degli zeri intorno l'input

Nelle reti progettate in questa tesi non verrà usato padding.

3.2 LAYERS DI POOLING

Uno strato di "pooling" è un tipo di strato tipicamente utilizzato dopo uno strato di convoluzione. Infatti, la combinazione convoluzione-pooling è una pratica comune nelle CNN e può essere ripetuta più volte in una rete.

L'operazione di pooling può essere vista come un sottocampionamento della mappa delle caratteristiche con lo scopo di ridurre la dimensione spaziale (larghezza ed altezza) delle attuali rappresentazioni; ciò serve per ridurre il numero di parametri ed il tempo computazionale della rete, ed inoltre tiene sotto controllo l'overfitting.

Le due tecniche di pooling più comuni sono:

- Pooling medio (o average pooling): calcola il valore medio per ogni "pool" sulla mappa delle caratteristiche;
- Pooling massimo (o max pooling): calcola il valore massimo per ogni "pool" della mappa delle caratteristiche. Il max-pooling introduce invarianza, questo vuol dire che piccoli cambiamenti non cambieranno il risultato finale, perciò aggiunge robustezza ai dati;

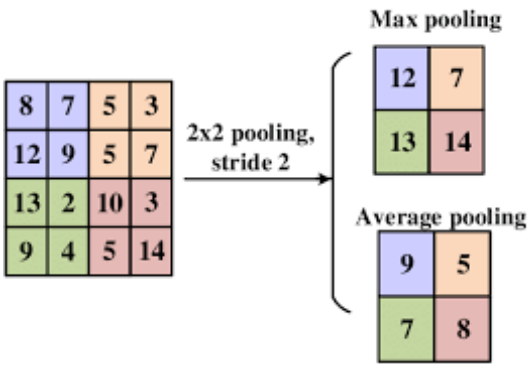

figura 3.7: risultati delle due tecniche di pooling[12]

[12] https://www.researchgate.net/figure/Pooling-layer-operation-oproaches-1-Pooling-layers-For-the-function-of-decreasing-the_fig4_340812216

Il pooling, così come avviene per la convoluzione, agisce con l'ausilio di un kernel traslato lungo tutta l'immagine. E' possibile scegliere la dimensione del kernel che viene chiamato, in questo caso, "pool size" da cui deriva il nome della trasformazione e dello "stride" (quantità di movimento tra le applicazioni del filtro all'immagine in ingresso) come per i layer convoluzionali.

Il risultato dell'uso di un livello di pooling e della creazione di mappe di caratteristiche sottocampionate o in pool è una versione riassunta delle caratteristiche rilevate nell'input che viene chiamata "pooled feature map". Inoltre il pooling aiuta a rendere la rappresentazione approssimativamente invariante a piccole traslazioni dell'input.
Di solito non viene utilizzato padding nei layers di pooling.

3.3 FLATTEN LAYERS

Un flatten layer viene utilizzato, di solito, dopo aver applicato una serie di convoluzioni e di pooling all'immagine in input.

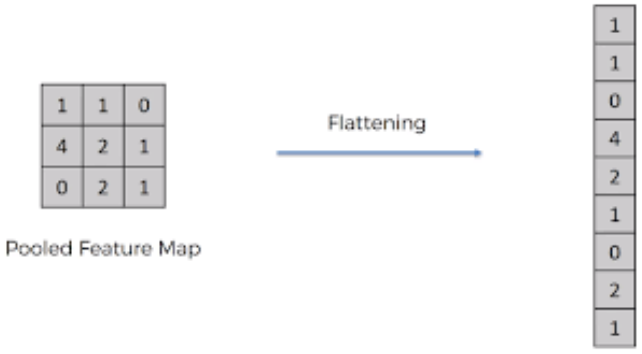

figura 3.8: risultato applicazione flatten layer[13]

L'operazione è molto semplice e consiste nel prendere i valori della matrice in input, riga per riga, per metterli in un unico vettore.

Se è presente più di una matrice in input al flatten layer, i vettori risultanti per ognuna, vengono sequenzialmente uniti in un unico grande vettore.

L'obiettivo è quello di utilizzare questo output in normali reti neurali artificiali con layer che non lavorano con matrici, per ulteriori elaborazioni ai dati.

[13] https://medium.com/@PK_KwanG/cnn-step-2-flattening-50ee0af42e3e

Riassumendo, le reti neurali convoluzionali dette anche CNN o ConvNet, risultano essere particolarmente performanti nell'analisi e nel riconoscimento delle immagini e sono composte principalmente dai tre layer precedentemente citati, organizzati come in figura:

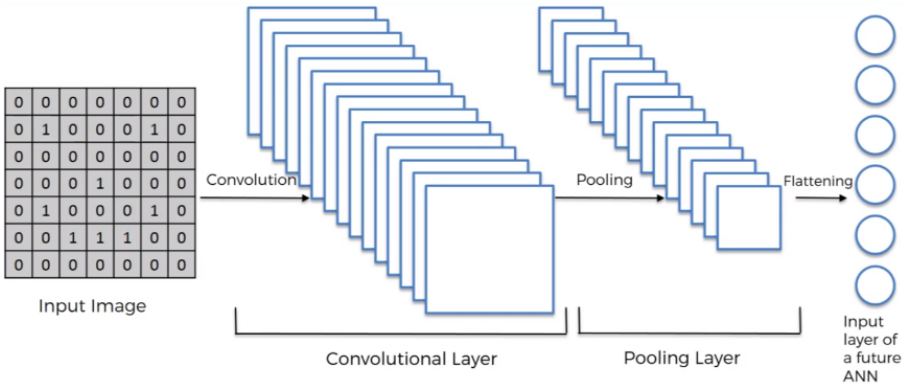

figura 3.9: esempio di una classica CNN[14]

Questa struttura può essere modificata aggiungendo altre tipologie di layer a seguire, ottenendo ulteriori elaborazioni sui dati e risultati migliori.

3.4 DENSE LAYERS

Un dense layer (o fully connected layer) è uno strato molto comune e di solito utilizzato negli stadi finali di una rete neurale artificiale; la caratteristica di questo layer è che ogni neurone è connesso con ogni neurone del layer precedente.

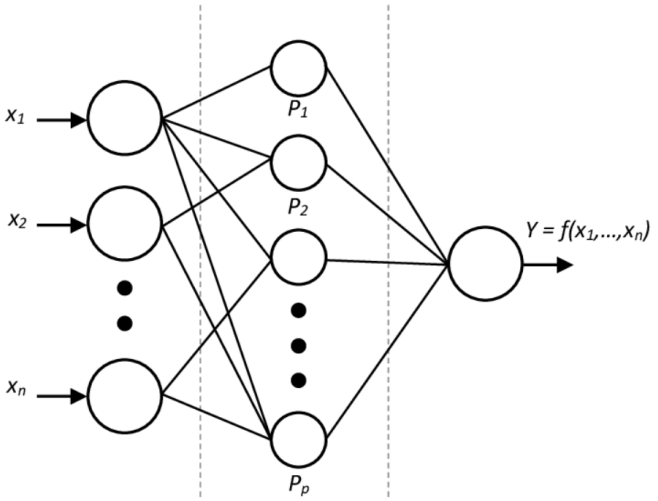

figura 3.10: dense layer[15]

Questo tipo di layer aumenta il numero dei parametri addestrabili dalla rete e aiuta il modello a trovare relazioni tra i valori dei dati in input.

[14] https://medium.com/@PK_KwanG/cnn-step-2-flattening-50ee0af42e3e
[15] https://analyticsindiamag.com/a-complete-understanding-of-dense-layers-in-neural-networks/

L’operazione eseguita da un dense layer è la seguente:

```
output = activation(dot(input, kernel) + bias)
```

dove:

- input = rappresenta i dati in input
- kernel = rappresenta il peso dei collegamenti
- dot = rappresenta il prodotto punto per punto tra input e i suoi rispettivi pesi
- bias = rappresenta una soglia che determina se l’output verrà propagato ai layer successivi
- activation = rappresenta la funzione di attivazione scelta

Ovviamente l’output dipende anche dal numero di neuroni per layer

3.5 DROPOUT

Il Dropout è un metodo di regolarizzazione estremamente economico ed efficace per ridurre l'overfitting e migliorare le prestazioni della rete, che consiste nell’ignorare, in modo casuale, il contributo di alcuni neuroni in un layer lasciando invariati gli altri.
Si applica alla maggior parte dei layer della rete e funge come da maschera.

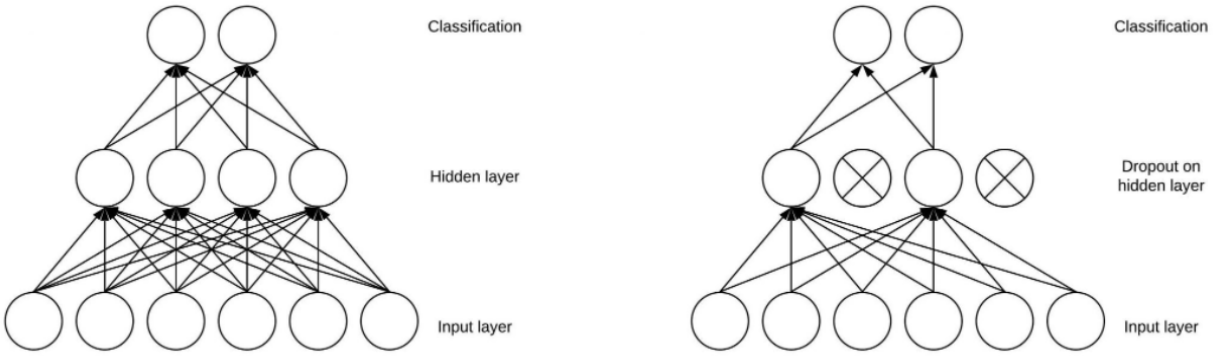

figura 3.11: esempio di rete senza dropout (a sinistra) e con dropout (a destra)[16]

Il dropout fa sì che la rete impari informazioni più robuste poiché, riducendo in maniera casuale il numero di connessioni e mitigando il fenomeno dell’overfitting, i nodi rimasti connessi dovranno regolare i propri pesi per adattarsi all’assenza dei nodi non connessi. Il dropout ha efficacia solo durante l’addestramento (fase di training) e non durante il test (fase di testing). L’aggiunta del dropout avviene solitamente su uno o più layer presenti nella rete. Questa inibizione di alcuni neuroni riduce la complessità della rete in modo casuale e quindi aiuta il modello a generalizzare le lezioni apprese durante la fase di addestramento.

[16] https://www.baeldung.com/cs/ml-relu-dropout-layers

3.6 ARCHITETTURA DELLE RETI PROGETTATE

Introdotti tutti gli elementi necessari a costruire una CNN, proseguiamo con la progettazione e lo studio di due diverse reti neurali convoluzionali che devono classificare, data un'immagine in input di una struttura, la presenza o meno di una crepa; esse prendono il nome di:

- “model_grayscale_v4”
- “model_grayscale_v6”

Entrambe saranno poi confrontate con la rete “LeNet-5” che è la rete di dimensione minore presente nello stato dell’arte, per valutare le prestazioni e poter nominare la migliore.

Ricordiamo che l’idea alla base di questo tesi, è quella di progettare delle CNN che siano il più “leggere” possibile in termini di dimensioni, per poter essere implementate in sistemi embedded con poca disponibilità computazionale e/o di memoria (nel nostro caso l’OpenMV Cam H7 Plus), ricercando la maggiore accuratezza possibile.

3.6.1 LeNet-5

Partendo dallo stato dell’arte, la rete neurale convoluzionale con dimensione più piccola è la “LeNet-5”, si tratta della rete neurale convoluzionale più semplice e anche la più datata infatti è stata creata da Yann LeCun nel 1998 con lo scopo di riconoscere cifre numeriche scritte a mano o stampate; oggi ancora diffusa e utilizzata [1].

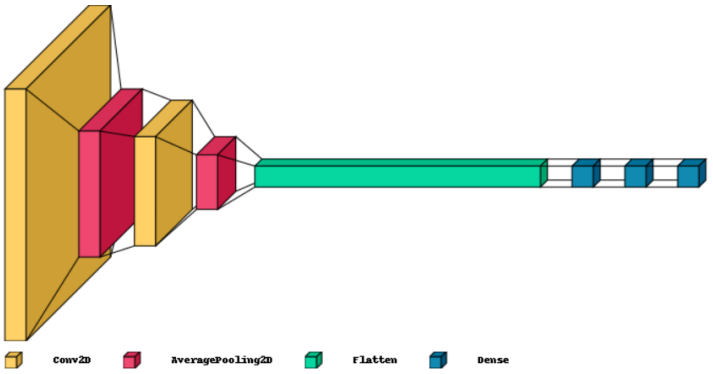

figura 3.12: modello “LeNet-5”

La “LeNet-5” è una rete composta da 8 layers: 2 “convolutional” layers, 2 “average pooling” layers, 3 “dense” layers e 1 “flatten” layer.

Il modello è costruito per accettare in input immagini 64x64 RGB.

Il primo layer è un layer convoluzionale formato da 6 filtri di dimensione 5x5 e funzione di attivazione “ReLu”.

Il secondo layer è un “average pooling” layer con un pool size di 2x2, strides 2x2 senza padding.

Il terzo layer è un layer convoluzionale formato da 16 filtri di dimensione 5x5 e funzione di attivazione “ReLu”.

Il quarto layer è un “average pooling” layer con un pool size di 2x2, strides 2x2 senza padding.

Il quinto layer è un “flatten” layer che trasforma l’input in un array a 1 dimensione.

Il sesto layer è un “dense” layer composto da 120 unità con funzione di attivazione “ReLu”.

Il settimo è un “dense” layer composto da 84 unità con funzione di attivazione “ReLu”.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 60, 60, 6)         456

average_pooling2d (AverageP  (None, 30, 30, 6)         0
ooling2D)

conv2d_1 (Conv2D)            (None, 26, 26, 16)        2416

average_pooling2d_1 (Averag  (None, 13, 13, 16)        0
ePooling2D)

flatten (Flatten)            (None, 2704)              0

dense (Dense)                (None, 120)               324600

dense_1 (Dense)              (None, 84)                10164

dense_2 (Dense)              (None, 2)                 170

=================================================================
Total params: 337,806
Trainable params: 337,806
Non-trainable params: 0
```

figura 3.13: “LeNet-5”

L’ottavo layer è un “dense” layer composto da 2 unità con funzione di attivazione “softmax”.

Il numero di parametri totali è: 337,806.

Nonostante la semplicità della “LeNet” è possibile ottenere risultati migliori in termini di dimensioni senza peggiorare l’accuratezza.

3.6.2 VGG16

VGG16 è una rete neurale convoluzionale proposta da Karen Simonyan e Andrew Zisserman dell’università di Oxford durante la “ImageNet Large Scale Visual Recognition Challenge” nel 2014 ed è stata una grande innovazione che ha aperto le porte a numerose innovazioni nel campo del “Computer Vision”.

Il nome VGG deriva dal dipartimento “Visual Geometry Group” dell’università di Oxford dove è stato partorito il modello mentre il numero 16 rappresenta il numero di “weight layers” ossia i layer che hanno dei parametri addestrabili durante il training (esistono anche VGG11, VGG13, VGG16, VGG19).

Quello che differisce questo modello da tutti gli altri è l’utilizzo di un filtro di dimensione 3x3 con stride 1x1 in tutti i layer convoluzionali di tutta la rete. L’idea dietro l’utilizzo di questi tipi di filtri in tutta la rete è quella che due filtri 3x3 consecutivi hanno lo stesso effetto di un filtro 5x5 e allo stesso modo, tre filtri 3x3 consecutivi hanno lo stesso effetto di un filtro 7x7. Il beneficio di questa tecnica è che riduce di molto il numero di parametri da addestrare e anche se aumenta il numero di layers della rete, la complessità rimane invariata grazie alle funzioni di attivazione “ReLU” presenti dopo ogni layer di convoluzione. La dimensione del filtro 3x3 è considerata la più piccola possibile per poter catturare informazioni in un’immagine, diminuirla potrebbe impattare l’abilità della rete di riconoscere caratteristiche dell’immagine in input [2].

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

figura 3.14: configurazioni possibili della VGG[17]

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

figura 3.15: numero di parametri delle configurazioni (in milioni)[18]

Esistono diversi tipi di configurazione della VGG ma alle base di ognuna c'è la ripetizione del blocco: gruppo di layer convoluzionali consecutivi (1,2 o 3) con filtro 3x3 e stride 1x1 seguito da un layer max-pooling con pool size 2x2.

Come si può vedere dalla figura 3.15, il numero di parametri è molto alto anche per la configurazione meno profonda e non rispetta il vincolo iniziale di "leggerezza", però dall'ispirazione di questo modello sono state progettate le nostre due reti: "model_grayscale_v4" e "model_grayscale_v6".

[17] https://doi.org/10.48550/arXiv.1409.1556
[18] https://doi.org/10.48550/arXiv.1409.1556

3.6.3 model_grayscale_v4

Partendo dal modello VGG e con l'obiettivo di snellirlo nasce la nostra prima rete progettata e utile al nostro scopo che prende il nome di "model_grayscale_v4".

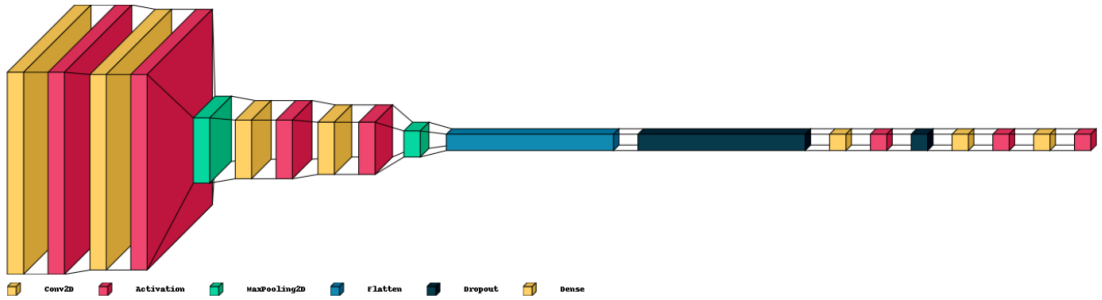

figura 3.16: modello "model_grayscale_v4"

Il "model_grayscale_v4" è una rete composta da 12 layers: 4 "convolutional" layers, 2 "max pooling" layers, 3 "dense" layers, 1 "flatten" layer e 2 "dropout" layers.

Il modello è costruito per accettare in input immagini 64x64 grayscale.

Il primo e il secondo layer sono layers convoluzionali formati da 16 filtri di dimensione 3x3 e funzione di attivazione "ReLu".

Il terzo layer è un "max pooling" con un pool size di 3x3.

Il quarto e il quinto layer sono layers convoluzionali formati da 32 filtri di dimensione 3x3 e funzione di attivazione "ReLu".

Il sesto layer è un "max pooling" con un pool size di 2x2.

Il settimo layer è un "flatten" layer che trasforma l'input in un array a 1 dimensione.

L'ottavo layer è un "dropout" layer con il 50% delle unità ignorate.

Il nono layer è un "dense" layer composto da 32 unità con funzione di attivazione "ReLu".

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Conv2D_1 (Conv2D)           (None, 62, 62, 16)        160

 ReLU_1 (Activation)         (None, 62, 62, 16)        0

 Conv2D_2 (Conv2D)           (None, 60, 60, 16)        2320

 ReLU_2 (Activation)         (None, 60, 60, 16)        0

 MaxPool_1 (MaxPooling2D)    (None, 20, 20, 16)        0

 Conv2D_3 (Conv2D)           (None, 18, 18, 32)        4640

 ReLU_3 (Activation)         (None, 18, 18, 32)        0

 Conv2D_4 (Conv2D)           (None, 16, 16, 32)        9248

 ReLU_4 (Activation)         (None, 16, 16, 32)        0

 MaxPool_2 (MaxPooling2D)    (None, 8, 8, 32)          0

 Flatten (Flatten)           (None, 2048)              0

 Dropout_1 (Dropout)         (None, 2048)              0

 Dense_1 (Dense)             (None, 32)                65568

 ReLU_5 (Activation)         (None, 32)                0

 Dropout_2 (Dropout)         (None, 32)                0

 Dense_2 (Dense)             (None, 32)                1056

 ReLU_6 (Activation)         (None, 32)                0

 Dense_3 (Dense)             (None, 2)                 66

 Softmax (Activation)        (None, 2)                 0

=================================================================
Total params: 83,058
Trainable params: 83,058
Non-trainable params: 0
```

figura 3.17: "model_grayscale_v4"

Il decimo layer è un "dropout" layer con il 50% delle unità ignorate.

L'undicesimo layer è un "dense" layer composto da 32 unità con funzione di attivazione "ReLu".

Il dodicesimo layer è un “dense” layer composto da 2 unità con funzione di attivazione “softmax”.

Il numero di parametri totali è: 83,058.

3.6.4 model_grayscale_v6

La seconda rete progettata è una variante della precedente con lo stesso numero di layers convoluzionali e pooling ma con minore numero di unità per ogni strato, un “dense” layer in meno e “dropout” applicato a più layer in maniera crescente; essa prende il nome di “model_grayscale_v6”.

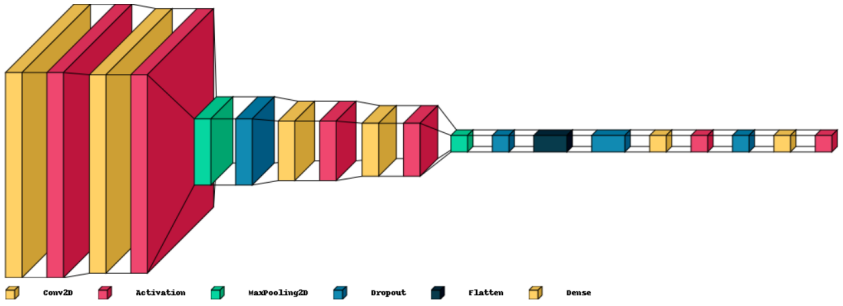

figura 3.18: modello “model_grayscale_v6”

Il “model_grayscale_v6” è una rete composta da 13 layers: 4 “convolutional” layers, 2 “max pooling” layers, 2 “dense” layers, 1 “flatten” layer e 4 “dropout” layers.

Il modello è costruito per accettare in input immagini 64x64 grayscale.

Il primo e il secondo layer sono layers convoluzionali formati da 16 filtri di dimensione 3x3 e funzione di attivazione “ReLu”.

Il terzo layer è un “max pooling” con un pool size di 3x3.

Il quarto layer è un “dropout” layer con il 20% delle unità ignorate.

Il quinto e il sesto layer sono layers convoluzionali formati da 16 filtri di dimensione 3x3 e funzione di attivazione “ReLu”.

Il settimo layer è un “max pooling” con un pool size di 3x3.

Model: "sequential"

Layer (type)	Output Shape	Param #
Conv2D_1 (Conv2D)	(None, 62, 62, 16)	160
ReLU_1 (Activation)	(None, 62, 62, 16)	0
Conv2D_2 (Conv2D)	(None, 60, 60, 16)	2320
ReLU_2 (Activation)	(None, 60, 60, 16)	0
MaxPool_1 (MaxPooling2D)	(None, 20, 20, 16)	0
Dropout_1 (Dropout)	(None, 20, 20, 16)	0
Conv2D_3 (Conv2D)	(None, 18, 18, 16)	2320
ReLU_3 (Activation)	(None, 18, 18, 16)	0
Conv2D_4 (Conv2D)	(None, 16, 16, 16)	2320
ReLU_4 (Activation)	(None, 16, 16, 16)	0
MaxPool_2 (MaxPooling2D)	(None, 5, 5, 16)	0
Dropout_2 (Dropout)	(None, 5, 5, 16)	0
Flatten (Flatten)	(None, 400)	0
Dropout_3 (Dropout)	(None, 400)	0
Dense_1 (Dense)	(None, 16)	6416
ReLU_7 (Activation)	(None, 16)	0
Dropout_4 (Dropout)	(None, 16)	0
Dense_3 (Dense)	(None, 2)	34
Softmax (Activation)	(None, 2)	0

Total params: 13,570
Trainable params: 13,570
Non-trainable params: 0

figura 3.19: “model_grayscale_v6”

L’ottavo layer è un “dropout” layer con il 30% delle unità ignorate.

Il nono layer è un “flatten” layer che trasforma l’input in un array a 1 dimensione.

Il decimo layer è un “dropout” layer con il 40% delle unità ignorate.

L’undicesimo layer è un “dense” layer composto da 16 unità con funzione di attivazione “ReLu”.

Il dodicesimo layer è un “dropout” layer con il 50% delle unità ignorate.

Il tredicesimo layer è un “dense” layer composto da 2 unità con funzione di attivazione “softmax”.

Il numero di parametri totali è: 13,570.

Come si può notare, dalla “LeNet-5” con 337,806 parametri siamo passati al “model_grayscale_v4” con 83,058 parametri fino ad arrivare al “model_grayscale_v6” con soli 13,570 parametri; nei prossimi paragrafi della tesi valuteremo le prestazioni di ognuna, a parità di condizioni, per osservare il loro comportamento.

3.7 VALUTAZIONE DI UNA CNN

Ai fini di verificare il corretto funzionamento di una CNN, viene implementato all’interno delle reti un potente ed efficace strumento di valutazione definito “matrice di confusione”. Nell'ambito dell'Intelligenza artificiale, la matrice di confusione, detta anche tabella di errata classificazione, restituisce una rappresentazione dell'accuratezza di classificazione statistica.

Ogni colonna della matrice rappresenta i valori predetti, mentre ogni riga rappresenta i valori reali. L'elemento sulla riga i e sulla colonna j è il numero di casi in cui il classificatore ha classificato la classe "vera" i come classe j. Attraverso questa matrice è osservabile se vi è "confusione" nella classificazione di diverse classi.

Per un problema di classificazione con C classi, la matrice di confusione M è una matrice “CxC” dove l’elemento M_{ij} corrisponde al numero di campioni definiti come i ma classificati dalla rete come j mentre l’elemento M_{ii} corrisponde invece ai campioni correttamente classificati.

Nel nostro caso avremo sempre matrici di confusione “2x2” con 2 classi e quindi 2 possibili valori reali.

		Valori predetti		
		n'	***p'***	**totale**
Valori Reali	***n***	Veri negativi	Falsi positivi	N
	p	Falsi negativi	Veri positivi	P
	totale	N'	P'	

figura 3.20: matrice di confusione[19]

Dalla matrice di confusione è possibile calcolare l'accuratezza (accuracy) come:

$$ACC = \frac{(VP + VN)}{(VP + VN + FP + FN)}$$

[19] https://it.wikipedia.org/wiki/Matrice_di_confusione

CAPITOLO 4
STRUMENTI UTILIZZATI

Prima di procedere con l'addestramento delle reti progettate è giusto introdurre gli strumenti hardware e software utilizzati e necessari a far sì che questi modelli prendano vita e abbiano senso.

4.1 STRUMENTI SOFTWARE

Per poter costruire un modello di CNN, tra le tante opzioni presenti in circolazione, sono state scelte le librerie: TensorFlow e Keras che si basano sul linguaggio di programmazione Python.

L'utilizzo di Python come linguaggio di programmazione deriva dalla sua semplicità, è un linguaggio con la possibilità di creare classi, esattamente come il C++, tuttavia è nato con un fondamento più user friendly che lo rende molto didattico. Una libreria molto interessante presente in Python si chiama TensorFlow, che permette di rendere parallelo il calcolo tramite le GPU, e possiede una API chiamata Keras che verrà utilizzata per la creazione di un modello.

Di seguito verranno spiegati in dettaglio tutti gli strumenti utilizzati.

4.1.1 TensorFlow

figura 4.1: logo TensorFlow [20]

TensorFlow è una libreria che permette di velocizzare significativamente i calcoli necessari ad un algoritmo di Machine Learning, contiene una serie di funzioni che sono in grado di sfruttare i tensori ed è una piattaforma open source end-to-end per l'apprendimento. Dispone di un ecosistema completo e flessibile di strumenti, librerie e risorse della community che consente ai ricercatori di promuovere lo stato dell'arte in ML e agli sviluppatori di creare e distribuire facilmente applicazioni basate su ML.
I modelli matematici utilizzati in TensorFlow sono reti neurali, che a seconda dell'architettura dei livelli e dei neuroni che la compongono, possono essere modellate partendo da un semplice modello fino a delle architetture di machine learning molto più complesse [3].

È possibile compilare i sorgenti con varie ottimizzazioni hardware delle CPU moderne o anche attivando l'uso di ulteriori librerie per sfruttare direttamente la potenza elaborativa parallela delle GPU contenute nelle schede video recenti. Infine, Google ha realizzato appositi processori ASIC chiamati TPU (Tensor Processing Unit) che aumentano ulteriormente la capacità elaborativa portandola a 180 teraflop e che possono essere

[20] https://www.tensorflow.org/

utilizzati direttamente da TensorFlow.

Questa libreria è una delle più utilizzate, in particolare da Google nelle sue applicazioni (ma non solo) ed è nativamente fruibile in diversi servizi cloud automatico, è possibile programmare in modo diretto su TensorFlow avendo un account Google, ne verrà discusso successivamente. La libreria mette a disposizione diversi algoritmi di ottimizzazione ed inoltre sono presenti numerosi esempi sul sito ufficiale della libreria.

4.1.2 Keras

figura 4.2: logo Keras [21]

Keras è una API (Application programming interface) di Python, molto utilizzata in Machine Learning per creare modelli di CNN con poche righe di codice. Possiede, inoltre, le funzioni necessarie a creare e salvare modelli di CNN per poter essere utilizzati successivamente in altre applicazioni (file del tipo modello.h5), è l'API ad alto livello per l'implementazione di algoritmi basati su reti neurali artificiali permette la possibilità di sviluppare e sperimentare velocemente nell'ambito del deep learning e machine learning. Mette a disposizione dei componenti fondamentali sulla cui base si possono sviluppare modelli complessi di apprendimento automatico. Si tenga inoltre presente che il binomio TensorFlow + Keras è molto comune, infatti, in rete sono presenti numerosi esempi e tutorial che partono da questa base [4].

Nel caso limite in cui venissero utilizzati algoritmi molto complessi e dataset molto grandi, le CPU non sarebbero più in grado di addestrare il modello, con il risultato del programma terminato prima ancora di poter iniziare; una considerazione va sempre fatta infatti nel momento in cui si progetta una rete neurale in funzione del fatto che le stesse, sono algoritmi che richiedono un tempo relativamente lungo per poter essere allenate e verificate. Quando si ha a che fare con applicazioni di intelligenza artificiale e machine learning, si deve essere consapevoli di quanta potenza di calcolo può essere necessaria per implementare modelli sufficientemente robusti ed efficienti.

4.1.3 Google Colab

figura 4.3: logo Google Colaboratory o "Colab" [22]

Per poter utilizzare queste librerie TensorFlow e Keras, tra i vari metodi possibili si è optato per semplicità e praticità di Google Colaboratory (o in breve "Colab") dove è necessario soltanto avere un account Google per poter scrivere ed eseguire Python nel

[21] https://keras.io/
[22] https://colab.research.google.com/

tuo browser con nessuna configurazione necessaria, accesso gratuito alle GPU e condivisione semplificata (attraverso Google Drive) [5].

Per poter creare dei modelli di Machine Learning è importante realizzare dei modelli che siano robusti ed efficienti, perciò, è necessaria una potenza di calcolo importante. Per i nostri scopi è più che sufficiente quella messa a disposizione dal Colab, ma con il crescere delle dimensioni del dataset la potenza necessaria per il training dei modelli può aumentare a dismisura.

Google Colab si basa sui cosiddetti blocchi note “Jupyter”. Questi non sono altro che documenti interattivi nei quali è possibile scrivere (e quindi eseguire) il codice sviluppato. Più precisamente, tali documenti permettono di suddividere il codice in celle, ognuna delle quali può contenere anche del testo informativo.

4.1.4 OpenMV IDE

figura 4.4: logo OpenMV [23]

Per la programmazione della piattaforma embedded OpenMV Cam si è dovuto ovviamente utilizzare il software della casa madre, scaricabile liberamente dal sito internet [6], scegliendo opportunamente il sistema operativo.

Il programma presenta diverse funzionalità quali un editor di testo, un debugger, la possibilità di vedere la webcam in real-time, la percezione dei colori in istogrammi da parte della webcam oltre che la funzionalità del caricamento dei dati on board; il caricamento dei dati all’interno della piattaforma avviene mediante cavo usb collegando PC-OpenMV.

4.2 STRUMENTI HARDWARE: OpenMV Cam H7 Plus

Dopo aver programmato, con gli strumenti sopracitati, addestrato e testato correttamente i modelli delle nostre reti neurali convoluzionali, li implementiamo all’interno della piattaforma embedded OpenMV Cam H7 Plus.

L’OpenMV Cam è una piccola scheda elettronica a bassa potenza che permette di implementare facilmente applicazioni che utilizzano la visione artificiale nel mondo reale. La programmazione dell’OpenMV Cam avviene in script Python di alto livello invece che in C/C++, in modo da rendere più facile lavorare con i complessi output degli algoritmi di visione artificiale e con le complesse strutture dati. Nel modulo della telecamera OpenMV H7 sono totalmente integrati tutti i componenti necessari per sviluppare e implementare un'applicazione di visione artificiale utilizzando l'apprendimento automatico.

[23] https://openmv.io/

figura 4.5: OpenMV Cam H7 Plus [24]

L' OpenMV Cam si presenta visivamente come una scheda elettronica in senso classico con una telecamera a bordo già integrata. Le caratteristiche salienti della OpenMV Cam sono brevemente qui riportate: il processore è un STM32H743II ARM Cortex M7 dotato di una velocità di clock di 480 MHz con 32MBs SDRAM + 1MB di SRAM e 32 MB di flash esterno + 2 MB di flash interno. I pin di I/O sono a 3.3V e sono tolleranti a 5V. La scheda viene collegata al computer con un semplice cavetto USB, permettendo una comunicazione fino a 12Mbs. E' presente inoltre una presa per scheda microSD in grado di leggere/scrivere fino a 100Mbs permettendo alla OpenMV Cam di scattare foto e di estrarre o inserire risorse di visione artificiale nella scheda µSD, caratteristica che verrà utilizzata successivamente nella fase di implementazione. Ulteriori caratteristiche tecniche sono presenti sul sito internet dell'azienda produttrice della scheda [7].

La OpenMV Cam H7 Plus è dotata di un sensore d'immagine OV5640 in grado di scattare immagini 2592x1944 (5MP), per di più possono essere implementate specifiche lenti al sensore sulla scheda.

La programmazione della scheda richiede, come anticipato nel paragrafo appena terminato, un proprio IDE di programmazione (scaricabile gratuitamente da [6]) chiamato OpenMV IDE. Lo sviluppo e la programmazione dell'applicazione (la rete CNN verrà implementata in altro modo) avviene interamente tramite l'IDE OpenMV, che offre un'interfaccia Python per lo sviluppo. Utilizzando Python come linguaggio di programmazione non occorre conoscere un livello di programmazione di basso livello.

Le applicazioni di questa scheda possono essere diverse come: frame differencing, color tracking, marker tracking, face detection, eye tracking, person detection, optical flow, QR code detection/decoding, data matrix detection/decoding, linear barcode decoding, apriltag tracking, line detection, circle detection, rectangle detection, template matching, image capture.

[24] https://openmv.io/products/openmv-cam-h7-plus

figura 4.6: caratteristiche OpenMV Cam [25]

Sono numerosi i motivi che hanno spinto nell'utilizzo di questa piattaforma rispetto ad altre concorrenti: la scheda presenta una versatilità nell'uso, in quanto ha costo relativamente ridotto rispetto ad altre, è di facile implementazione meccanica in quanto di poco ingombro e leggera, è relativamente piccola, ha dimensione pari a 35,56 x 44,45 mm, presenta inoltre un sistema di programmazione relativamente facile e una grande quantità e varietà di risorse presenti in rete con cui è possibile documentarsi.

[25] https://www.digikey.it/it/articles/use-the-openmv-cam-apply-machine-learning-object-detection

CAPITOLO 5
TRAINING DELLE RETI

Progettata la struttura delle reti e definiti gli strumenti necessari, si procede con l'addestramento e il testing di tutte le reti per valutare le prestazioni.

5.1 DATASET UTILIZZATI

I datasets utili allo scopo presenti nello stato dell'arte non sono molti e molto spesso non vengono condivisi pubblicamente, in questo caso sono stati utilizzati i seguenti:

5.1.1 Concrete crack images for classification

Il dataset contiene 40000 immagini di superfici con crepe e non, di grandezza 227x227 e RGB. Diviso in 2 classi chiamate "Positive" e "Negative" ognuna contenente 20000 immagini. Dataset generato da 458 immagini ad alta risoluzione (4032x3024) con diversi tipi di illuminazione e superfici. Non è applicato nessun tipo di "data augmentation". Le immagini sono prese da vari edifici nel "METU Campus" [8].

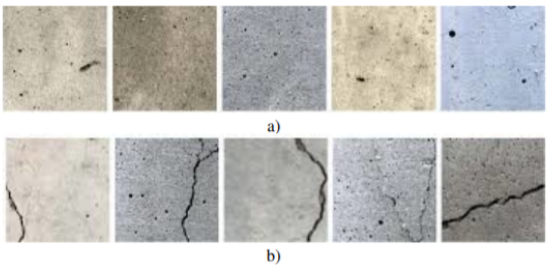

figura 5.1: immagini appartenenti a "Concrete crack images for classification" di due classi diverse: a)Negative e b)Positive [26]

5.1.2 SDNET2018

Dataset contenente 56092 immagini 256x256 RGB di "cracked" e "non-cracked" muri, pavimenti e strutture di ponti. Le crepe variano da larghezza di 0.6 mm a 25 mm inoltre sono presenti immagini con diversi tipi di ostruzioni come ombre, imperfezioni, buchi, detriti, spigoli. Dataset generato da 230 immagini (4068x3456) scattate usando una telecamera digitale Nikon a 16 MP. Il dataset è diviso in 3 categorie: "Wall", "Pavement" e "Bridge deck"; ogni categoria contiene 2 classi non bilanciate che contengono le immagini "cracked" e "non-cracked" [9], divise come in figura:

Image description		No. cracked	No. non-cracked	Total
Reinforced	Bridge deck	2025	11,595	13,620
	Wall	3851	14,287	18,138
Unreinforced	Pavement	2608	21,726	24,334
Total		8484	47,608	56,092

figura 5.2: descrizione e statistiche del dataset "SDNET2018" [27]

[26] https://data.mendeley.com/datasets/5y9wdsg2zt/2
[27] https://doi.org/10.1016/j.dib.2018.11.015

figura 5.3: "SDNET2018" include immagini di: a)crepe sottili, b)crepe grossolane, c)ombre, d)macchie, e)superfici con imperfezioni, f)buchi, g)bordi, h)giunzioni e i)ostacoli sullo sfondo [28]

5.1.3 CrackForest

Dataset composto da 118 immagini di superfici stradali con crepe [10]; è stato bilanciato aggiungendo altre 118 immagini, senza crepe, prese dal dataset SDNET2018 nella categoria "Pavement". A differenza dei 2 dataset precedenti questo dataset verrà utilizzato solo per fare testing sulle reti a causa del ridotto numero di immagini.

figura 5.4: immagini appartenenti rispettivamente al dataset "CrackForest" e "SDNET2018" [29]

[28] https://doi.org/10.1016/j.dib.2018.11.015
[29] https://www.kaggle.com/datasets/mahendrachouhanml/crackforest

5.2 SET UP DELLA SPERIMENTAZIONE

In questo paragrafo verranno riportate le fasi operative più importanti, svolte in ordine, che ci consentono di arrivare al lavoro finito e completo.

5.2.1 Divisione datasets

Per la sperimentazione dei modelli dividiamo tutti i datasets in tre parti (come spiegato nel capitolo 2.6), utili al training (70%), validation (15%) e testing (15%), ogni parte è costituita da due classi, chiamate “Positive” e “Negative” che contengono rispettivamente le immagini con crepe e senza crepe e che sono bilanciate solo nel caso del dataset “concrete_crack_dataset_splitted”. La figura 5.5 descrive più in dettaglio l’organizzazione di tutti i datasets utilizzati.

ORGANIZZAZIONE DATASETS	TRAIN (70%)		VALID (15%)		TEST (15%)		100%
NOME	Positive	Negative	Positive	Negative	Positive	Negative	TOTALE
concrete_crack_dataset_splitted	14k	14k	3k	3k	3k	3k	40k
SDNET2018_Wall	2696	10001	578	2143	577	2143	18138
SDNET2018_Pavement	1826	15208	391	3259	391	3259	24334
SDNET2018_BridgeDeck	1418	8116	304	1739	303	1739	13619
CrackForest_Dataset	0	0	0	0	118	118	236

figura 5.5: divisione dei datasets

Da sottolineare che i dataset “SDNET2018_Pavement”, “SDNET2018_BridgeDeck”, “SDNET2018_Wall”, vengono ricavati da “SDNET2018” e che il dataset “CrackForest_Dataset” non dispone di abbastanza immagini per poter essere utilizzato nell’addestramento di una rete e quindi viene utilizzato solo per fare testing.

5.2.2 Pre-processing delle immagini

I modelli sono costruiti per accettare in input, immagini 64x64 e RGB (per LeNet) o grayscale (per model_grayscale_v4 e model_grayscale_v6). Per ottenere questo formato le immagini dei datasets vengono pre-processate con un resize e inoltre viene fatta una normalizzazione del valore dei pixel che passano da un range 0-255 a un range 0-1. La normalizzazione è fatta per assicurare che il batch delle immagini abbia media 0 e deviazione standard di 1 in modo da avere una riduzione del tempo di training.

5.2.3 Configurazione delle CNN

Prima di iniziare con l'addestramento della rete dobbiamo configurare il nostro modello. La configurazione del modello richiede tre parametri: ottimizzatore, perdita e metriche.

L'ottimizzatore (o algoritmo di ottimizzazione) controlla il tasso di apprendimento. È stato utilizzato "Adadelta" (capitolo 2.5.1). Adadelta continua ad apprendere anche quando sono stati fatti molti aggiornamenti. Il tasso di apprendimento determina la velocità con cui vengono calcolati i pesi ottimali per il modello, un tasso di apprendimento piccolo può portare a pesi più precisi (fino a un certo punto), ma il tempo necessario per calcolare i pesi sarà conseguentemente più lungo. Un optimizer è un Optimization algorithm, ovvero un algoritmo di ottimizzazione che permette di individuare, attraverso una serie di iterazioni, quei valori dei weight tali per cui la cost-function risulti avere il valore minimo.

Useremo "categorical_crossentropy" per la nostra funzione di perdita (capitolo 2.5.2). Questa è la scelta più comune per la classificazione. Un punteggio più basso indica che il modello funziona meglio.

Per una più facile interpretazione, useremo la metrica “accuracy” per vedere il punteggio di precisione sul set di validazione quando addestriamo il modello.

```
model.compile(loss='categorical_crossentropy',
              optimizer=keras.optimizers.Adadelta(learning_rate=1, name='Adadelta'),
              metrics=['accuracy'])
```

figura 5.6: compilazione del modello

5.2.4 Training e salvataggio dei modelli

Per confrontare le prestazioni delle diverse reti, ognuna viene allenata e testata con tutti i dataset ad eccezione del “CrackForest_dataset”.

Il training viene eseguito utilizzando il “training dataset” di ogni dataset su ogni rete con il validation dataset che monitora la curva di apprendimento del modello. Tutte le reti vengono addestrate con un batch size di 64 per una durata di 20 epochs (capitolo 2.9).

Finito l’addestramento che può richiedere anche parecchio tempo (dai 20 ai 40 minuti in media), si passa al salvataggio del modello in un file Keras con estensione “.h5”.

```
# training
start = time.time()

with tf.device('/device:GPU:0'):

  history = model.fit(
      train_dataset,
      epochs=epochs,
      batch_size = batch_size,
      validation_data=validation_dataset,
      verbose=True)

#save
model.save(name_modelTF)

print ('Time taken for training: {} sec\n'.format(time.time() - start))
```

figura 5.7: training e salvataggio del modello

5.2.5 Conversione in TensorFlow Lite

Quando il modello è stato addestrato e salvato, viene convertito in TensorFlow Lite [11], ciò permette di esportare la rete sulla piattaforma embedded e inoltre offre numerosi vantaggi rispetto al formato del modello del protocollo di TensorFlow, come dimensioni ridotte e inferenza ridotta che consente a TensorFlow Lite di funzionare in modo efficiente su dispositivi con risorse di calcolo e memoria limitata.

Il motivo per cui non si possono usare direttamente i modelli TensorFlow sta nel fatto che questi hanno dimensioni eccessive e non sono ottimizzati per lavorare su sistemi a bassa potenza o che comunque devono rispettare dei parametri di consumo di energia.

Lo strumento che permette di ottimizzare il modello in TensorFlow Lite è il “converter”; è possibile ottimizzare attraverso tre tecniche: quantizzazione, pruning e clustering. Nel nostro caso viene utilizzata la quantizzazione, essa agisce riducendo la precisione dei numeri utilizzati per rappresentare i parametri di un modello, da numeri float 32-bit a numeri int 8-bit (quantizzazione intera post-allenamento). Ciò si traduce in una dimensione del modello più piccola e maggiore velocità di inferenza.

Il risultato della conversione è un file che viene salvato con estensione “.tflite”.

```
# Converter: from keras model
converter = tf.compat.v1.lite.TFLiteConverter.from_keras_model_file(name_modelTF)

print("converter ok.....----------------------------------------------------------------------")


# quantization
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset_gen
tflite_quant_model = converter.convert()

# save
open(name_modelTFLite, "wb").write(tflite_quant_model)

print('quantization and storage ok... !!!')
```

figura 5.8: conversione, quantizzazione e salvataggio del modello in “.tflite”

NOME MODELLO	FILE SIZE	PARAMS
LeNet.h5	3.918 MB	337806
LeNet.tflite	0.327 MB	337806
model_grayscale_v4.h5	1.027 MB	83058
model_grayscale_v4.tflite	0.087 MB	83058
model_grayscale_v6.h5	0.222 MB	13570
model_grayscale_v6.tflite	0.019 MB	13570

figura 5.9: effetto della conversione TensorFlow Lite

5.2.6 Testing su Google Colab

Per valutare la validità e le prestazioni delle reti addestrate, dopo il training e la conversione, i modelli salvati (“.h5” e “.tflite”) vengono testati, su Google Colab, sui dataset con cui la rete è stata addestrata e con tutti gli altri dataset, per determinare quanto il modello è in grado di generalizzare quello che ha imparato.

```
start = time.time()

# evaluate
evaluate = model_loaded.evaluate(evaluation_dataset)

elapsed_time = time.time() - start
print ('Average image inference time %.2f msec\n'%(1000*(elapsed_time/nb_test_samples)))

print("evaluate: ", evaluate)
print('Test loss:', evaluate[0])
print('Test accuracy:', evaluate[1])
```

figura 5.10: testing del modello ".h5"

```
# prediction
predictions=list()
i=0
for x in test_images:
  interpreter.set_tensor(input_details[0]['index'], x)

  start = time.time()
  interpreter.invoke()
  elapsed_time = time.time() - start
  elapsed_times.append(elapsed_time)

  tflite_results = interpreter.get_tensor(output_details[0]['index'])
  predictions.append(np.argmax(tflite_results))
  #print("i: ", i)
  i+=1


# Results
matrix = matrix_confusion(test_labels, predictions, n_class, nb_test_samples)

print ('Average image inference time %.2f msec\n'%(1000*np.mean(np.array(elapsed_times))))
```

figura 5.11: testing del modello ".tflite"

5.2.7 Testing su OpenMV Cam H7 Plus

E' l'ultimo test fatto sulla piattaforma embedded attraverso l'ambiente OpenMV IDE che viene eseguito solo sui modelli TensorFlow Lite (".tflite"), utilizzando gli stessi datasets usati in Google Colab.

In questo test è necessario trasformare le immagini del testing set in formato ".bmp" per poter essere utilizzato nella board.

Tutto il codice necessario è scaricabile da [12].

CAPITOLO 6
RISULTATI SPERIMENTALI

Dalle simulazioni fatte in ambiente Google Colab sui modelli otteniamo:

TRAIN / VAL DATASET (28k / 6k):			TEST DATASET (6k):		TEST DATASET (2720):		TEST DATASET (2042):		TEST DATASET (3650):		TEST DATASET (236):	
concrete_crack_dataset_splitted			concrete_crack_dataset_splitted		SDNET2018_Wall		SDNET2018_BridgeDeck		SDNET2018_Pavement		CrackForest_Dataset	
NOME MODELLO	**FILE SIZE**	**PARAMS**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**
LeNet.h5	3.918 MB	337806	98.95%	113.98 ms	21.36%	116.98 ms	19.05%	128.28 ms	23.01%	137.52 ms	40.68%	126.65 ms
LeNet.tflite	0.327 MB	337806	98.82%	13.27 ms	**21.32%**	12.09 ms	19.0%	11.36 ms	22.6%	11.63 ms	40.68%	12.02 ms
model_grayscale_v4.h5	1.027 MB	83058	99.5%	68.97 ms	17.43%	64.83 ms	19.54%	126.05 ms	24.3%	128.30 ms	42.8%	103.18 ms
model_grayscale_v4.tflite	0.087 MB	83058	**99.5%**	43.59 ms	17.54%	45.54%	19.49%	43.93 ms	23.97%	44.92 ms	42.37%	54.68 ms
model_grayscale_v6.h5	0.222 MB	13570	99.33%	123.73 ms	19.23 %	73.20 ms	20.91%	98.90 ms	51.53%	86.02 ms	52.97%	88.36 ms
model_grayscale_v6.tflite	0.019 MB	13570	99.37%	35.38 ms	18.97%	39.23 ms	**20.52%**	35.68 ms	**49.59%**	35.87 ms	**50.0%**	36.21 ms

figura 6.1: accuracy e inference time dei modelli testati e addestrati su Google Colab con "concrete_crack_dataset_splitted"

TRAIN / VAL DATASET (17034 / 3650):			TEST DATASET (3650):		TEST DATASET (2720):		TEST DATASET (2042):		TEST DATASET (6k):		TEST DATASET (236):	
SDNET2018_Pavement			SDNET2018_Pavement		SDNET2018_Wall		SDNET2018_BridgeDeck		concrete_crack_dataset_splitted		CrackForest_Dataset	
NOME MODELLO	**FILE SIZE**	**PARAMS**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**
LeNet-P.h5	3.918 MB	337806	86.74%	170.46 ms	77.54%	294.84 ms	83.94%	245.81 ms	20.57%	193.72 ms	45.34%	11.02 ms
LeNet-P.tflite	0.327 MB	337806	86.88%	12.61 ms	77.68%	13.08 ms	84.04%	13.52 ms	20.6%	12.82 ms	46.61%	16.72 ms
model_grayscale_v4-P.h5	1.027 MB	83058	89.97%	121.18 ms	78.16%	96.31 ms	84.52%	128.28 ms	24.13%	80.58 ms	51.27%	79.54 ms
model_grayscale_v4-P.tflite	0.087 MB	83058	**90%**	46.02 ms	78.16%	44.18 ms	84.52%	44.19 ms	24.53%	44.30 ms	**51.27%**	44.55 ms
model_grayscale_v6-P.h5	0.222 MB	13570	90.03%	103.22 ms	78.79%	78.07 ms	85.16%	128.28 ms	41.02%	63.66 ms	50%	6.12 ms
model_grayscale_v6-P.tflite	0.019 MB	13570	89.4%	37.77 ms	**78.79%**	36.29 ms	**85.16%**	36.29 ms	**49.43%**	36.30 ms	50%	36.84 ms

figura 6.2: accuracy e inference time dei modelli testati e addestrati su Google Colab con "SDNET2018_Pavement"

TRAIN / VAL DATASET (12697 / 2721):			TEST DATASET (2720):		TEST DATASET (3650):		TEST DATASET (2042):		TEST DATASET (6k):		TEST DATASET (236):	
SDNET2018_Wall			SDNET2018_Wall		SDNET2018_Pavement		SDNET2018_BridgeDeck		concrete_crack_dataset_splitted		CrackForest_Dataset	
NOME MODELLO	**FILE SIZE**	**PARAMS**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**
LeNet-W.h5	3.918 MB	337806	79.01%	169.49 ms	86.63%	162.94 ms	84.18%	275.20 ms	36.83%	203.66 ms	50%	318.67 ms
LeNet-W.tflite	0.327 MB	337806	78.93%	13.60 ms	87.42%	15.09 ms	**84.52%**	13.84 ms	39.3%	12..92 ms	**50%**	13.43 ms
model_grayscale_v4-W.h5	1.027 MB	83058	83.49%	96.46 ms	86.49%	132.49 ms	68.71%	133.30 ms	42.27%	92.60 ms	50%	173.68 ms
model_grayscale_v4-W.tflite	0.087 MB	83058	**81.95%**	43.52 ms	**88.85%**	44.14 ms	81.78%	43.70 ms	43.67%	43.50 ms	49.58%	44.14 ms
model_grayscale_v6-W.h5	0.222 MB	13570	80.18%	69.75 ms	61.92%	79.65 ms	80.41%	107.54 ms	41.62%	33.81 ms	50.42%	173.86 ms
model_grayscale_v6-W.tflite	0.019 MB	13570	79.56%	38.04 ms	51.86%	47.16 ms	81.44%	44.79 ms	**44.23%**	43.37 ms	18.64%	46.02 ms

figura 6.3: accuracy e inference time dei modelli testati e addestrati su Google Colab con "SDNET2018_Wall"

TRAIN / VAL DATASET (9534 / 2043):			TEST DATASET (2042):		TEST DATASET (3650):		TEST DATASET (2720):		TEST DATASET (6k):		TEST DATASET (236):	
SDNET2018_BridgeDeck			SDNET2018_BridgeDeck		SDNET2018_Pavement		SDNET2018_Wall		concrete_crack_dataset_splitted		CrackForest_Dataset	
NOME MODELLO	**FILE SIZE**	**PARAMS**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**	**ACCURACY**	**AV. INFERENCE TIME**
LeNet-D.h5	3.918 MB	337806	83.45%	98.98 ms	69.37%	211.49 ms	76.25%	206.60 ms	19.08%	118.81 ms	35.59%	169.90 ms
LeNet-D.tflite	0.327 MB	337806	83.15%	16.29 ms	**73.92%**	12.70 ms	77.39%	11.52 ms	20.55%	11.13 ms	41.95%	11.21 ms
model_grayscale_v4-D.h5	1.027 MB	83058	84.23%	275.28 ms	49.26%	202.45 ms	70.51%	74.33 ms	40.85%	119.15 ms	32.2%	347.26 ms
model_grayscale_v4-D.tflite	0.087 MB	83058	**84.62%**	44.70 ms	56.49%	44.79 ms	74.89%	44.14 ms	44.78%	43.87 ms	38.98%	45.20 ms
model_grayscale_v6-D.h5	0.222 MB	13570	82.86%	98.98 ms	55.51%	104.76 ms	78.49%	30.23 ms	43.48%	73.66 ms	37.71%	136.97 ms
model_grayscale_v6-D.tflite	0.019 MB	13570	84.18%	35.19 ms	67.1%	35.84 ms	**78.86%**	40.92 ms	**47.68%**	35.25 ms	**49.58%**	35.26 ms

figura 6.4: accuracy e inference time dei modelli testati e addestrati su Google Colab con "SDNET2018_BridgeDeck"

Riassumendo:

ACCURACY MODELLI COLAB			TRAIN/VAL/TEST DATASET: concrete_crack_dataset_splitted	TRAIN/VAL/TEST DATASET: SDNET2018_Pavement	TRAIN/VAL/TEST DATASET: SDNET2018_Wall	TRAIN/VAL/TEST DATASET: SDNET2018_BridgeDeck
MODELLO	**FILE SIZE**	**PARAMS**	**ACCURACY**	**ACCURACY**	**ACCURACY**	**ACCURACY**
Net.h5	3.918 MB	337806	98.95%	86.74%	79.01%	83.45%
Net.tflite	0.327 MB	337806	98.82%	86.88%	78.93%	83.15%
grayscale_v4.h5	1.027 MB	83058	99.5%	89.97%	83.49%	84.23%
grayscale_v4.tflite	0.087 MB	83058	**99.5%**	**90%**	**81.95%**	**84.62%**
grayscale_v6.h5	0.222 MB	13570	99.33%	90.03%	80.12%	82.86%
grayscale_v6.tflite	**0.019 MB**	13570	99.37%	89.4%	79.56%	84.18%

figura 6.5: accuracy dei modelli su Google Colab

Dalle simulazioni fatte in OpenMV IDE sugli stessi modelli otteniamo:

ACCURACY MODELLI OPENMV			TRAIN/VAL/TEST DATASET: concrete_crack_dataset_splitted	TRAIN/VAL/TEST DATASET: SDNET2018_Pavement	TRAIN/VAL/TEST DATASET: SDNET2018_Wall	TAIN/VAL/TEST DATASET: SDNET2018_BridgeDeck
MODELLO	**FILE SIZE**	**PARAMS**	**ACCURACY**	**ACCURACY**	**ACCURACY**	**ACCURACY**
Net.tflite	0.327 MB	337806	98.9%	86.3 %	78.97 %	83.35%
grayscale_v4.tflite	0.087 MB	83058	**99.47 %**	**90.77 %**	**80.29 %**	84.82 %
grayscale_v6.tflite	**0.019 MB**	13570	99.33%	89.34 %	78.71 %	**85.06 %**

figura 6.6: accuracy dei modelli su OpenMV IDE

INFERENCE TIME E FPS MODELLI OPENMV			TRAIN/VAL/TEST DATASET: concrete_crack_dataset_splitted		TRAIN/VAL/TEST DATASET: SDNET2018_Pavement		TRAIN/VAL/TEST DATASET: SDNET2018_Wall		TAIN/VAL/TEST DATASET: SDNET2018_BridgeDeck	
MODELLO	**FILE SIZE**	**PARAMS**	**INFERENCE TIME (ms)**	**FPS**	**INFERENCE TIME (ms)**	**FPS**	**INFERENCE TIME (ms)**	**FPS**	**INFERENCE TIME (ms)**	**FPS**
Net.tflite	0.327 MB	337806	61,21	16,33	61,21	16,33	61,21	16,33	61,21	16,33
grayscale_v4.tflite	0.087 MB	83058	89,92	11,12	90,56	11,04	90,79	11,01	89,85	11,12
grayscale_v6.tflite	**0.019 MB**	13570	74,13	13,48	73,89	13,53	75,51	13,24	74,19	13,47

figura 6.7: Inference time in real-time e Frame per second (FPS)

Per tempo di inferenza si intende il tempo necessario alla rete neurale convoluzionale di produrre una risposta dato un input, mentre gli FPS (Frame Per Second) sono calcolati come reciproco del tempo di inferenza.

L'accuracy è calcolata come definita nel capitolo 3.7.

CAPITOLO 7
CONCLUSIONI

Dai risultati ottenuti si può osservare come più il dataset è numeroso e più otteniamo un valore di accuracy elevato; quindi non è solo la scelta della rete a influire maggiormente sull'accuratezza ma anche l'architettura della rete e il dataset scelto.

Si potrebbero fare ulteriori analisi variando il numero di epochs del training e confrontare come varia l'accuracy.

Per quanto riguarda la conversione del modello TensorFlow in TensorFlow Lite, si può osservare come la perdita di accuracy è minima mentre la riduzione del tempo di inferenza è evidente come anche quella della dimensione. "model_grayscale_v6" è il modello più piccolo ottenuto con 0.019 MB e 13570 parametri.

Otteniamo la maggiore accuratezza in "model_grayscale_v4" utilizzando il "Concrete crack images for classification" dataset anche se i risultati con gli altri test dataset sono peggiori rispetto agli altri modelli allenati con "SDNET2018" dataset.

I modelli allenati usando "SDNET2018" si comportano meglio quando devono predire immagini che non appartengono al loro dataset.

In OpenMV IDE otteniamo circa gli stessi risultati che otteniamo in ambiente Google Colab nelle stesse condizioni.

In conclusione, si può dire che la rete più consistente è "model_grayscale_v4" però anche "model_grayscale_v6" può essere utilizzata quando si hanno vincoli sulla dimensione del file molto più stringenti; comunque entrambe si comportano meglio della "LeNet" in tutte le condizioni.

A livello personale, lo svolgimento di questa tesi è stato molto interessante perchè mi ha permesso di studiare e imparare argomenti che probabilmente non avrei mai incontrato nel mio percorso di studi come l'intelligenza artificiale, Python, TensorFlow… e alla fine mi ritengo soddisfatto del lavoro prodotto considerando l'essere partiti senza nessuna conoscenza di base degli argomenti.

Tutto il materiale utilizzato e prodotto è disponibile nella bibliografia.

BIBLIOGRAFIA

[1] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.

[2] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In ICLR, 2015.

[3] TensorFlow: https://www.tensorflow.org/

[4] Keras: https://keras.io/

[5] GoogleColab: https://colab.research.google.com/

[6] OpenMV IDE: https://openmv.io/pages/download

[7] OpenMV Cam H7 Plus: https://openmv.io/products/openmv-cam-h7-plus

[8] Dataset “Concrete crack images for classification”: https://data.mendeley.com/datasets/5y9wdsg2zt/2

[9] Dataset “SDNET2018”: https://digitalcommons.usu.edu/all_datasets/48/

[10] Dataset “CrackForest”: https://www.kaggle.com/datasets/mahendrachouhanml/crackforest

[11] TensorFlow Lite: https://www.tensorflow.org/lite/guide

[12] Link Google Drive codice OpenMV IDE: https://drive.google.com/drive/folders/1z_5Yyu5bk4oNXOGQeRLUsNc9BZA9QuRS?usp=sharing

[13] Link Google Drive datasets divisi: https://drive.google.com/drive/folders/10slHnB6h4RkMHIvd2fkqbf1z4Ybrd7CE?usp=sharing

[14] Link Google Drive codice TensorFlow dei modelli: https://drive.google.com/drive/folders/1icEUPBsMK_fgqWCU2zUH28c8YuFy8EvB?usp=sharing

[15] Link Google Drive file .h5 e .tflite dei modelli: https://drive.google.com/drive/folders/1pQpRkQJhrHejdvw3cYE1YhTMlffm8jiy?usp=sharing