UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI INGEGNERIA

Corso di Laurea triennale in Ingegneria Elettronica

Dipartimento di Ingegneria dell'Informazione

Implementazione di una rete neurale convoluzionale per la classificazione di fitopatie su piattaforma embedded OpenMV Cam

Implementation of a convolutional neural network for plant diseases classification on the OpenMV Cam embedded platform

Relatore:

Prof. Claudio Turchetti

Tesi di Laurea di:

Denis Di Leo

Correlatore:

Dott.ssa Laura Falaschetti

A.A. 2020 / 2021

INDICE

CAPITOLO 1
Introduzione

Come prerogativa iniziale nulla potrebbe esimerci dallo stabilire, come punto iniziale nel vasto mondo a cui si sta andando incontro a cosa ci si sta riferendo e quali sono i punti cardinali di questo settore tecnologico. È importante quindi andare a mettere dei paletti per poter incardinare l'argomento, per poi incominciare a muoversi in questo ecosistema. Viviamo in un contesto storico nel quale tutti abbiamo toccato e potuto vedere con mano quanto fosse importante la tecnologia quale l'utilizzo di internet e i vari social network che sono presenti in tutte le varie attività di vita quotidiana che possano essere ricreative o lavorative. Nel presente elaborato è riportato una possibile soluzione ad una problematica in quanto le malattie delle piante hanno un notevole impatto sulle colture e provocano elevate perdite in termini di qualità e resa. A causa delle sostanze tossiche sprigionate si possono correre anche rischi per la sicurezza delle derrate alimentari e degli alimenti per animali. Per la lotta alle malattie delle piante, ogni anno, su scala mondiale, si registrano costi e perdite per diversi miliardi di euro [3]. Uno strumento che potrebbe risultare utile in questa sfida potrebbe essere quello dell'utilizzo di una CNN per identificare precocemente una malattia di una pianta mediante l'utilizzo di una *piattaforma embedded* quale è ad esempio l'*OpenMV* dotata di un sistema di visione. Per chiarire meglio le idee, la fitopatologia è lo studio delle malattie delle piante provocate da virus, batteri, alghe, fitoplasmi e funghi, o meglio i danni che hanno le piante non di natura meccanica (maltempo, uomo…) quali sono invece le fisiopatie, problematica che esula dallo scopo della nostra CNN.

Negli ultimi anni un settore che ha avuto un notevole sviluppo è stato quello dell'utilizzo di tecniche di Intelligenza Artificiale, quindi di "apprendimento automatico", una disciplina dell'informatica che studia tecniche mirate alla progettazione di sistemi hardware e software utili a far fornire all'elaboratore elettronico prestazioni che comunemente possono apparire solo come di pertinenza dell'intelligenza umana.

Oggi l'Intelligenza Artificiale (A.I.) è sfruttata da molti colossi informatici, tecnologici e non: Facebook, Google, Amazon… nell'identificazione di visi umani, di determinati oggetti in immagini e video, oltre che essere molto usata per il riconoscimento di tracce audio e linguaggi naturali. È un argomento di studio in rapido e costante sviluppo e pertanto una nuova e migliore tecnica è possibile in breve tempo. Tra gli ulteriori e vari usi dell'intelligenza artificiale vi è anche la comprensione del discorso umano, e delle auto a guida autonoma. Nella figura sotto riportata è presente una panoramica del "mondo" in questione.

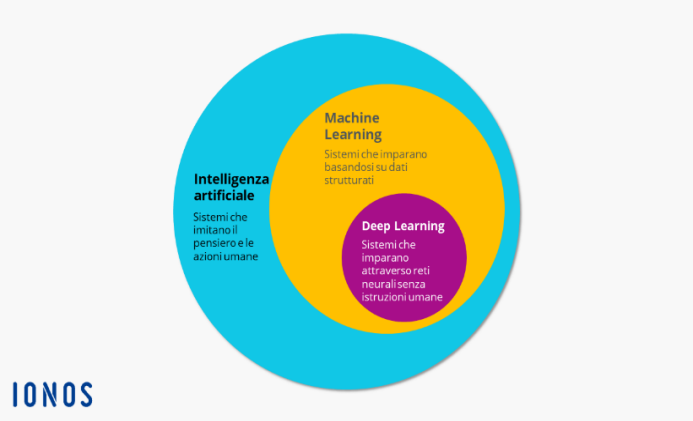

Figura 1.1: Contesto di Riferimento[1].

[1] https://www.ionos.it/digitalguide/online-marketing/marketing-sui-motori-di-ricerca/

L'obiettivo di questa Tesi di Laurea è quello di realizzare una *rete neurale convoluzionale* (CNN) da implementare all'interno di una piattaforma embedded *OpenMV* ai fini della classificazione di alcune fisiopatie che colpiscono alcune culture alimentari, in altri termini, come elemento caratterizzante si ha lo sviluppo di una rete neurale convoluzionale da implementare all'interno di una piattaforma embedded. Le CNN (Convolutional Neural Network) sono una tecnica di Machine Learning considerata, attualmente, lo "stato dell'arte" per quanto riguarda gli algoritmi per la classificazione di immagini.

L'idea di sviluppo del progetto ha avuto come "base di lancio" una sorta di variante della pubblicazione [1] ad opera dei professori del DII dell'università. La pubblicazione in questione trattasi dello sviluppo e dell'utilizzo di una rete CNN da implementare all'interno di una piattaforma embedded per il riconoscimento della malattia dell'esca dell'uva, da qui, a modo di esempio, è partita l'idea di sviluppare una CNN in grado di riconoscere differenti malattie da differenti culture agricole.

La tesi è costituita da sette capitoli, seguendo un percorso che parte dai principi teorici per poi sviluppare l'argomentazione step by step fino a concludersi con l'implementazione effettiva su board. L'elaborato si suddivide in tre sezioni principali, nella prima sono riportati i fondamenti teorici necessari delle reti neurali convoluzionali, nella seconda parte sono riportati gli strumenti hardware e software necessari alla costruzione della CNN, nella terza e ultima parte, sono riportati, la programmazione fisica su piattaforma embedded con una verifica sul campo e mediante opportuni strumenti teorici, al termine segue un'interpretazione dei risultati raggiunti.

L'elaborato di tesi prende origine dall'esperienza di tirocinio, cercando di descrivere lo stesso in modo puntuale e fondamentalmente in ordine di sviluppo, al fine di poter essere seguiti ed eventualmente ripercorsi. In particolare, sono stati messi in risalto a seguire della parte teorica iniziale gli strumenti software necessari, non trascurando anche l'approccio metodologico seguito.

CAPITOLO 2
Tecniche di Machine Learning

2.1 Intelligenza artificiale Machine Learning e Deep Learning

L'*intelligenza artificiale* (A.I.) è una sorta di intelligenza dimostrata dalle macchine, in similitudine all'intelligenza naturale mostrata dagli esseri umani o dagli animali ovvero, un sistema che avverte il suo ambiente (riconoscimento di oggetti, suoni, linguaggio) e risponde con azioni che massimizzano le sue possibilità di raggiungere i suoi obiettivi.

Il *Machine Learning* è un sottoinsieme dell'intelligenza artificiale (AI) che si occupa di creare sistemi che apprendono o migliorano le performance in base ai dati che utilizzano. Un'importante distinzione è che sebbene tutto ciò che riguarda il machine learning rientra nell'intelligenza artificiale, l'intelligenza artificiale non include il machine learning. Machine Learning indica un insieme di metodi con cui si allena l'intelligenza artificiale in modo tale da poter svolgere delle attività non programmate, ma soprattutto da poter apprendere dall'esperienza pregressa come esattamente fa l'intelligenza umana, correggendosi e quindi migliorandosi attraverso gli errori commessi e prendendo decisioni autonome, si nota quindi un avanzamento tecnologico rispetto alla generalizzazione dell'intelligenza artificiale [4].

Il *Deep Learning* costituisce l'insieme delle tecniche necessarie alla realizzazione del Machine Learning e quindi può essere considerata una sottoclasse. Gli algoritmi di Deep Learning simulano il comportamento del cervello umano. La parola *deep,* infatti, deriva proprio dal fatto che le reti neurali utilizzate sono dotate di svariati livelli nascosti di neuroni, che le rendono per l'appunto profonde e somiglianti a livello strutturale (biologico) al cervello umano.

2.2 Reti neurali

Come si evince dal nome, le reti neurale sono ispirate dai *processi biologici* e, nel caso di quelle *convoluzionali*, dalla corteccia visiva dove si hanno dei *neuroni*, particolari cellule che rispondono agli stimoli esterni della vista. Il neurone può essere considerato l'elemento fondamentale delle *reti neurali*, in quanto è alla base dei livelli di cui esse sono composte. Il principio di funzionamento consiste nel ricevere un segnale in input, elaborarlo e inviarlo in output ad altri neuroni.

Una rete neurale è una rappresentazione artificiale del cervello umano che cerca di simulare il processo di apprendimento. Per mostrare dove le reti neurali hanno la loro origine, si può dare un veloce sguardo al modello biologico da cui vengono riprese, ovvero il cervello umano. Il cervello umano è composto da un numero elevato (circa 100 miliardi) di cellule neurali che elaborano le informazioni. Ogni cellula funziona come un processore semplice e solo la massiccia interazione tra tutte le cellule e la loro elaborazione in parallelo rende possibili le capacità cerebrali. Ogni neurone è collegato mediamente con una decina di migliaia di altri neuroni, si hanno quindi centinaia di miliardi di connessioni.

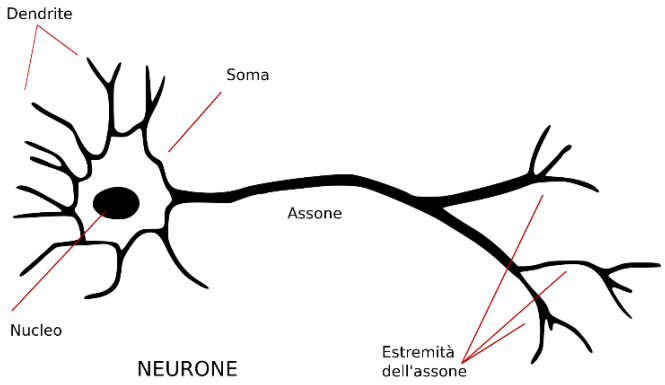

Figura 2.1. Neurone[2].

Come indicato nella figura sopra riportata, un neurone è costituito da un nucleo, da dendriti per le informazioni in entrata e un assone con dendriti per le informazioni in uscita che vengono passate ai neuroni connessi (sinapsi).

L'informazione è trasportata tra i neuroni in forma di stimoli elettrici lungo i dendriti. Le informazioni in entrata che raggiungono i dendriti del neurone vengono sommate e poi inviate lungo l'assone ai dendriti alla sua estremità (sinapsi), dove l'informazione in uscita passa ai neuroni collegati, se supera una certa soglia. In questo caso, il neurone si dice essersi "attivato". Se la stimolazione in arrivo non è sufficiente, le informazioni non vengono trasportate oltre. In questo caso, il neurone si dice che è "inibito". Le connessioni tra i neuroni sono adattative, ovvero che la struttura delle connessioni cambia dinamicamente. È comunemente riconosciuto che la capacità di apprendimento del cervello umano si basa su questo adattamento. In analogia a ciò che avviene in una rete neurale biologica, lo stesso comportamento avviene in una rete neurale artificiale.

Come il cervello umano, una rete neurale artificiale consiste di neuroni e delle connessioni tra di loro. I neuroni trasportano le informazioni in entrata sulle loro connessioni in uscita verso altri neuroni e sono collegati fra loro secondo vari schemi. Un neurone artificiale si rifà al modello della cellula neurale biologica, e funziona nello stesso modo. L'elaborazione può essere anche molto sofisticata ma in un caso semplice si può pensare che i singoli ingressi x_n vengano moltiplicati per un opportuno valore w_n detto peso, il risultato delle moltiplicazioni viene sommato e se la somma supera una certa soglia il neurone si attiva attivando la sua uscita, altrimenti il neurone è inibito. Se attivato, il neurone invia un output sulle connessioni pesate in uscita a tutti i neuroni collegati, e così via. Segue un esempio di un singolo neurone artificiale con la relativa rappresentazione matematica.

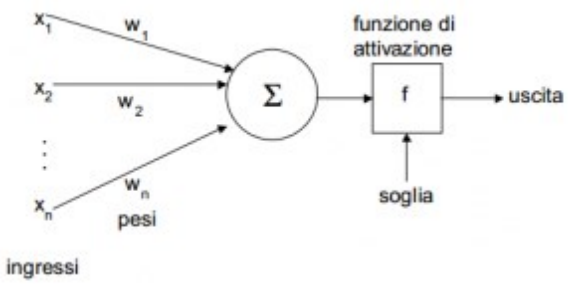

Figura 2.2. Neurone artificiale[3].

[2] https://it.wikipedia.org/wiki/File:Neurone.png
[3] https://it.wikipedia.org/wiki/Rete_neurale_artificiale

Il valore di ingresso di input x_n viene moltiplicato per un peso w_n e sommato insieme agli altri, il risultato di questa somma 'passa' attraverso la funzione *f e* viene ottenuto in uscita y_n, che può essere rappresentata dalla legge matematica:

$$y = f\left(\sum_1^n x_n * w_n\right)$$

Le reti neurali artificiali riescono oggi a risolvere determinate categorie di problemi avvicinandosi sempre più all'efficienza del nostro cervello. Dalla nascita del concetto di neurone artificiale ad oggi è stata fatta molta strada. In moltissimi ed eterogenei settori scientifici, alla biomedicina, le reti neurali hanno ormai un impiego quotidiano [5]. In una rete neurale, i neuroni sono raggruppati in strati (*layer*), denominati strati di neuroni (*neuron layers*). Di solito ogni neurone di uno strato è collegato a tutti i neuroni dello strato precedente e successivo (eccetto il livello di input e lo strato di uscita della rete). Le informazioni fornite da una rete neurale sono propagate *layer-by-layer* dallo strato di input a quello di output attraverso nessuno, uno o più strati nascosti (*hidden*). A seconda dell'algoritmo di apprendimento, è anche possibile che l'informazione si propaghi all'indietro attraverso la rete (*backpropagation*).

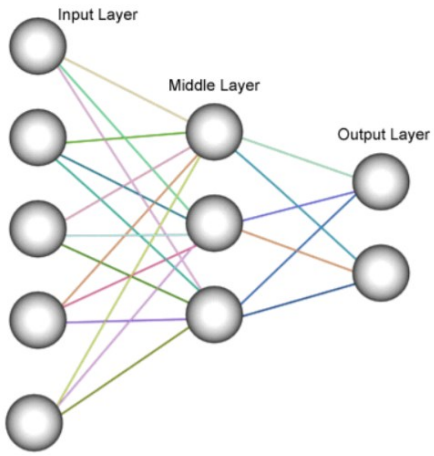

Figura 2.3. Esempio di rete neurale[4].

2.3 La classificazione

La classificazione è quell'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. Tale attività può essere svolta mediante algoritmi di apprendimento automatico supervisionato o non supervisionato.

2.3.1. L'apprendimento supervisionato

L'apprendimento supervisionato è una tecnica di apprendimento automatico che mira a istruire un sistema informatico in modo da consentirgli di risolvere dei compiti in maniera autonoma sulla base di una serie di esempi ideali, costituiti da coppie di input e di output desiderati, che gli vengono inizialmente forniti. Algoritmi di apprendimento supervisionato possono essere utilizzati nei più disparati settori. Degli esempi riguardano il campo medico in cui si può prevedere lo scatenarsi di particolari crisi sulla base dell'esperienza di passati dati biometrici, l'identificazione vocale che migliora sulla base degli ascolti audio passati,

[4] https://it.wikipedia.org/wiki/Rete_neurale_artificiale#/media/ File:Artificial_neural_network.svg

l'identificazione della scrittura manuale che si perfeziona sulle osservazioni degli esempi sottoposti dall'utente.

2.3.2. L’apprendimento non supervisionato

L'apprendimento non supervisionato è una tecnica di apprendimento automatico che consiste nel fornire al sistema informatico una serie di input (esperienza del sistema) che egli riclassificherà ed organizzerà sulla base di caratteristiche comuni per cercare di effettuare ragionamenti e previsioni sugli input successivi. Al contrario dell'apprendimento supervisionato, durante l'apprendimento vengono forniti all'apprendista solo esempi non annotati, in quanto le classi non sono note a priori ma devono essere apprese automaticamente. Un esempio tipico di questi algoritmi lo si ha nei motori di ricerca. Questi programmi, data una o più parole chiave, sono in grado di creare una lista di link rimandanti alle pagine che l'algoritmo di ricerca ritiene attinenti alla ricerca effettuata. La validità di questi algoritmi è legata alla utilità delle informazioni che riescono ad estrarre dalla base di dati, nell'esempio sopracitato è legata all'attinenza dei link con l'argomento cercato [6]. Molti problemi di *apprendimento automatico*, infatti, sono formulati come problemi di minimizzazione di una certa funzione di perdita (*loss function)* applicati ad set di esempi (*training set*), esprimendo la differenza tra i valori predetti dal modello in fase di allenamento e i valori attesi.

Come elemento base per iniziare a discendere meglio nel contesto, si potrebbe pensare a questo esempio, se supponessimo di avere davanti due foto in cui in una è presente la foto di un gatto e nell’altra la foto di un cane, in un battito di ciglia sapremmo associare correttamente alle due foto il gatto e il cane correttamente senza alcun dubbio. Fin dall’infanzia l’uomo utilizzando il senso della vista è abituato ad associare in funzione di quello che vede dalla forma dal colore o dalle dimensioni un nome proprio agli oggetti che ci circondano.

Potrebbe sembrare perciò semplice per un computer, ingegnerizzato per fare milioni di calcoli distinguere tra due immagini e associare cane e gatto, in realtà comprendere questa distinzione per un computer non è una questione banale, anzi presuppone una conoscenza di un ampio ventaglio di conoscenze. Conviene perciò partire dal basso, ovvero dal concetto di immagine per un computer.

2.4 Immagine vista da un computer

A livello pratico, il punto di partenza sarà un’immagine di input utilizzata allo scopo di ottenere in output una immagine “etichettata” nella quale i soggetti rappresentati saranno categorizzati, come già sottolineato ciò che per noi umani è qualcosa di comune se non banale, per una macchina, quale un pc o un dispositivo mobile (con discrete capacità di calcolo), il piano di lavoro è totalmente differente. Se ci soffermassimo un istante potremmo notare anche noi una particolarità anche ora mentre si sta leggendo il testo di questa relazione si sta sperimentando il fenomeno della visione e tutto ciò che vediamo di fronte sembrerebbe avere una risoluzione perfetta, o quasi. Se volessimo quantificare la visione dell’occhio umano corrisponderebbe circa a 570 megapixel, valore enorme rispetto alla decina di megapixel di una telecamera di uno smartphone, ma non infinito.

Un’immagine dal punto di vista di un computer si presenta come un insieme di tanti quadratini chiamati pixel (figura 2.4), è importante avere questa realtà oggettiva presente in quanto è la base di come viene strutturata una rete neurale convoluzionale. L’immagine non è vista come un corpo unico lineare bensì, come è possibile vedere anche nell’immagine, a seguito di un ingrandimento è possibile vedere tanti “quadratini” che, visti nel loro insieme da una certa distanza rendono l’immagine fluida.

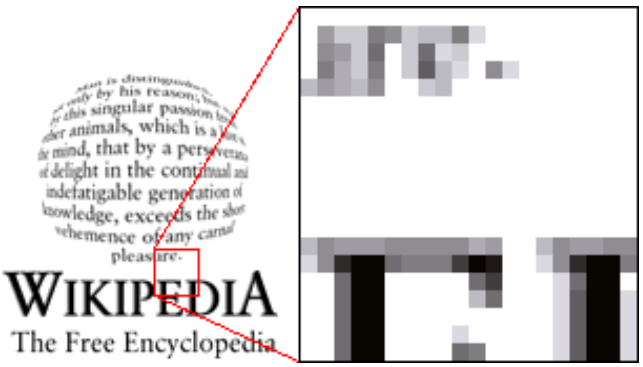

Figura 2.4. Pixel di una immagine[5].

Quindi, nel momento in cui la rete riceverà in input un'immagine, questa verrà vista come un array di pixel variabile a seconda della risoluzione e delle dimensioni dell'immagine (per esempio un array 32x32x3 dove 3 è il valore RGB e 32 il numero di pixel per lato, o nel caso di una JPG a colori 480x480 riceverà un array di 480x480x3 elementi) e conoscerà per ciascuno di questi elementi dell'array il valore relativo all'intensità del pixel che varia da 0 a 255 (maggiore è il valore e maggiore sarà l'intensità del colore).

Figura 2.5. Immagine vista da un uomo[6].

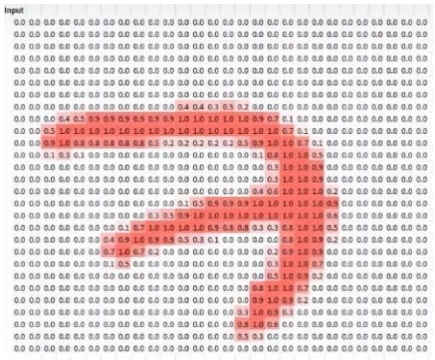

Figura 2.6. Immagine vista da un computer[7].

Un computer "vede" un'immagine come una matrice di valori numerici compresi tra 0 e 255, con un numero di canali pari al numero di colori visualizzabili. Un'immagine in bianco e nero ha solo il canale della luce mentre, una foto a colori RGB (Red Green & Blue) ha un canale per ogni colore (rosso R, verde G e blu B rispettivamente), pertanto si otterrà una matrice di valori per ognuno. In generale, le immagini oggi sono codificate tramite codifica RGB (una immagine è composta da tre matrici, una per ciascuno dei tre colori principali, e il risultato finale è dato dalla somma di queste). Le *CNN*, che tipicamente lavorano con immagini a colori, effettuano la convoluzione su ognuna di queste matrici e sommano i risultati ottenuti.

Lo scenario di osservazione di una immagine tra l'umano e la macchina è quindi diverso ma l'obiettivo prefissato è fare in modo tale che la macchina si comporti esattamente come noi riuscendo a differenziare tutte le immagini indicandoci quale contiene, per ritornare all'esempio di prima, un cane e quale un gatto. Le *Reti Neurali* usate per raggiungere tale scopo sono le *Reti Neurali Convoluzionali*, conosciute molto più comunemente con la sigla *CNN* (*Convolutional Neural Network*).

[5] https://it.wikipedia.org/wiki/Pixel
[6] https://www.lorenzogovoni.com/architettura-di-rete-neurale-convoluzionale/
[7] https://www.lorenzogovoni.com/architettura-di-rete-neurale-convoluzionale/

Termini che verranno ampiamente ad essere trattati diffusamente nel presente elaborato sono quelli di *training set* e *test set*, parole che sono molto importanti quando si viene a parlare di rete neurale convoluzionale. Per sommi capi, si anticipa che, successivamente alla fase di creazione di una rete neurale vi è una fase di allenamento (*training*) ed una di test (*testing*). Per allenare una *rete neurale convoluzionale* è necessario fornire al *modello* delle immagini, di medesimo formato ma da varie angolazioni diverse dell'oggetto desiderato, se possibile, in modo tale da fornire alla rete più informazioni possibili di cosa gli si potrebbe parare davanti, ai fini di 'imparare' a riconoscere se, in fase successiva, nel momento in cui venisse proposta un'immagine in cui sia presente l'oggetto ricercato e riconoscerlo, anche in un'immagine confusionaria se necessario. La fase di *testing* è una fase di simulazione di quello che la rete dovrebbe fare nel momento in cui viene messa a lavoro con immagini sconosciute a cui deve sapere rispondere se ad esempio è presente qualcosa che è conosciuto o meno, i compiti di una *CNN* in questo riconoscimento sono vari.

Per discendere meglio nella questione, l'obbiettivo presentato è quello di creare una *rete neurale convoluzionale* tale che sia in grado di riconoscere, nel momento del bisogno, se nelle immagini che le vengono fornite, la specie della pianta e dell'eventuale malattia presente (differenti tipi di malattie). Va da pensare che sarebbe impossibile per una CNN riconoscere tutte le malattie e tutte le piante esistenti, perciò, ciò che si fa è una restrizione delle piante che la rete è in grado di riconoscere a priori, mediante l'utilizzo di *data set* contenenti soltanto certe specie di piante a seconda dell'uso che noi riteniamo utile e soltanto di alcune delle tante malattie che possono affliggere una cultura agricola. Sarebbe impensabile l'ideazione e la creazione di un sistema che sia in grado di riconoscere tutte le piante e malattie esistenti. A scopo di esempio nella rete progettata è stato utilizzato un data set che prende il nome di *PlantVillage*, contenente 61486 immagini di culture agricole reali, che spaziano dai peperoni, pomodori alle patate, afflitte da una decina di differenti malattie possibili (virus, muffe, batteri…), in una fase successiva verrà illustrato largamente il *data set*.

CAPITOLO 3
Convolutional Neural Network (CNN)

3.1 Architettura base di una CNN

Una *rete neurale convoluzionale* o *convolutional neural network (CNN)* è una rete principalmente utilizzata come algoritmo di *apprendimento supervisionato* ed è in grado di imparare a classificare immagini a partire da un dataset di esempio.Una *CNN* è un algoritmo di *Deep Learning* che può contenere un'immagine di input, assegnare importanza (*pesi*) a vari aspetti nell'immagine ed essere in grado di differenziare l'una dall'altra. Cioè, un'immagine di input viene elaborata durante la fase di convoluzione e successivamente le viene attribuita un'etichetta. La preelaborazione richiesta in una CNN è molto più bassa rispetto ad altri algoritmi di classificazione. Una CNN è in grado di acquisire con successo le dipendenze spaziali e temporali in un'immagine attraverso l'applicazione di filtri pertinenti. L'architettura si adatta meglio al dataset di immagini a causa della riduzione del numero di parametri coinvolti e della riutilizzabilità dei pesi. In altre parole, la rete può essere addestrata per comprendere meglio la sofisticazione dell'immagine.

Il ruolo di una Rete Neurale Convoluzionale è quello di ridurre le immagini in una forma più facile da elaborare, senza perdere le caratteristiche che sono fondamentali per ottenere una buona previsione. Questo è importante quando dobbiamo progettare un'architettura che non solo è performante per le funzionalità di apprendimento, ma è anche scalabile a seconda della size del dataset [7].

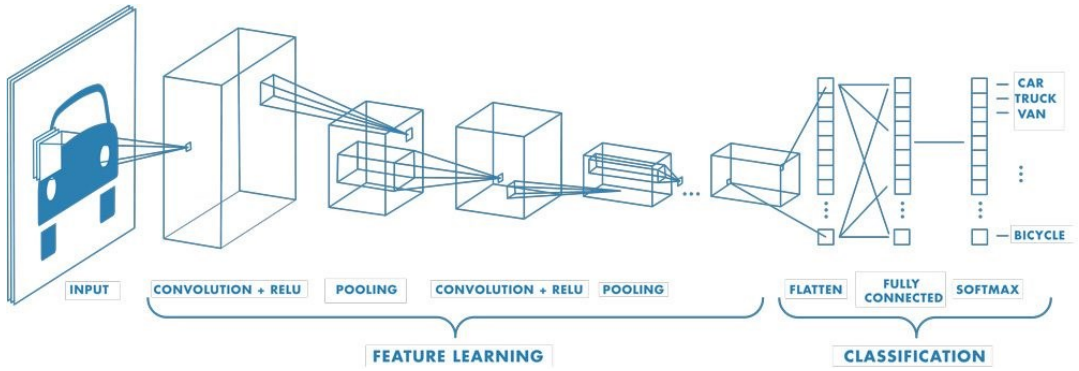

Figura 3.1. Rete Neurale Convoluzionale[8].

Come si può vedere dall'immagine sopra riportata l'architettura di una CNN prevede una sorta di impilamento di strati detti *convoluzionali* con dei *filtri*, seguiti da uno strato di *pooling*. Insieme, questi strati formano un blocco, e questi blocchi possono essere ripetuti. Ovviamente la rete, anche appena assemblata può essere soggetta a successive operazioni di ottimizzazione quali possono includere tecniche quali il *dropout* e *l'aumento dei dati*, quest'ultima tecnica aumenta le prestazioni incoraggiando il modello ad apprendere ulteriori caratteristiche espandendo semplicemente il set di dati di formazione.

Una rete neurale troppo piccola potrebbe non essere in grado di riconoscere abbastanza tratti per classificare efficacemente gli ingressi, mentre una troppo grande potrebbe ottenere prestazioni estremamente buone in fase di addestramento ma ottenerne di pessime in fase di test, poiché durante l'addestramento potrebbe sviluppare delle attivazioni nascoste che le permettevano di ottenere un punteggio più alto. Le reti neurali convoluzionali, come anticipato, sono parte di una categoria ben più grande, ossia le reti neurali artificiali, le

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

quali esulano dagli scopi della presente. Vengono di seguito riportate le operazioni che possono essere considerate fondamentali che avvengono all'interno di una CNN.

3.2 Convoluzione

Lo scopo della *convoluzione* è di estrarre localmente le *caratteristiche* dell'oggetto sull'immagine. Significa che la rete impara modelli specifici all'interno dell'immagine e sarà in grado di riconoscerla ovunque nell'immagine. Questo passaggio viene ripetuto fino a quando tutta l'immagine viene scansionata.

Una convoluzione è la semplice applicazione di un *filtro* a un input. L'applicazione ripetuta dello stesso filtro a un input dà come risultato una *mappa di attivazioni* chiamata *mappa delle caratteristiche* (*feature map*), che indica una caratteristica rilevata in un input, quale un'immagine. Gli *strati convoluzionali* restano comunque i principali blocchi di costruzione utilizzati nelle reti neurali convoluzionali. Quindi, sintetizzando le reti neurali convoluzionali applicano un filtro (un rilevatore di linee ad esempio) a un input per creare una *mappa di caratteristiche* che riassume la presenza di caratteristiche rilevate nell'input.

Nel contesto di una *rete neurale convoluzionale,* una convoluzione è un'operazione lineare che comporta la moltiplicazione di un insieme di pesi (il *filtro* o *kernel*) con l'input (l'immagine), proprio come una rete neurale tradizionale. Il filtro è più piccolo dei dati di input e il tipo di moltiplicazione applicata tra una patch di input delle dimensioni del filtro e il filtro è un prodotto di punti (prodotto tra matrici). Un prodotto di punti è la moltiplicazione elementare tra la patch di dimensioni del filtro dell'input e il filtro, che viene poi sommato, ottenendo sempre un unico valore. Poiché risulta in un singolo valore, l'operazione è spesso chiamata "prodotto scalare". In particolare, il filtro viene applicato sistematicamente ad ogni parte sovrapposta o patch delle dimensioni del filtro dei dati di input, da sinistra a destra, dall'alto in basso. Il filtro poi si sposta in basso di una riga e torna alla prima colonna e il processo si ripete da sinistra a destra per ottenere la seconda riga della mappa delle caratteristiche. E così via fino a quando la parte inferiore del filtro si appoggia sulla parte inferiore o sull'ultima riga dell'immagine di input.

In altre parole, ogni valore della mappa di attivazione dello strato successivo cattura una regione spaziale più ampia del precedente. Le *feature maps* catturano aspetti caratteristici di regioni via via maggiori e questo è il motivo per cui le CNN possono essere definite "profonde", per studiare l'intera immagine sono necessarie lunghe successioni di "blocchi" di strati. Questa applicazione *sistematica* dello stesso filtro attraverso un'immagine è un'idea molto forte. Se il filtro è progettato per rilevare un tipo specifico di caratteristica nell'input, allora l'applicazione di quel filtro sistematicamente attraverso l'intera immagine di input permette al filtro di scoprire quella caratteristica ovunque nell'immagine. Questa capacità è comunemente indicata come *invarianza di traduzione*, ad esempio l'interesse generale nel sapere se la caratteristica è presente piuttosto a dove è presente.

L'uscita dalla moltiplicazione del filtro con l'array di input una volta è un singolo valore. Poiché il filtro viene applicato più volte all'array di input, il risultato è un array bidimensionale di valori di output che rappresentano un filtraggio dell'input. Come tale, l'array bidimensionale in uscita da questa operazione è la "*feature map*". Una volta creata una *feature map*, possiamo passare ogni valore nella *feature map* attraverso una non linearità, quale una *ReLU*.

Si consideri che i filtri che operano direttamente sui valori di pixel grezzi impareranno ad estrarre caratteristiche di basso livello, quali le linee ad esempio, successivamente intervengo filtri che svolgono compiti più certosini come forme particolari. Questo processo continua fino a che gli *strati* molto profondi estraggono volti, animali, case e così via. Questo è esattamente ciò che vediamo nella pratica. L'astrazione delle caratteristiche ad ordini sempre più alti man mano che la profondità della rete aumenta. L'impilamento degli strati convoluzionali permette una decomposizione gerarchica dell'input.

L'operazione è rappresentabile come un prodotto di matrici tra il filtro e una matrice avente pari dimensione del filtro di una parte ben definita dell'immagine, che viene spostata di volta in volta. È riportato nella figura n. 2.2. un esempio di convoluzione tra una *feature map* e un *kernel* (*feature detector*) senza le opportune misure preventive di una tecnica che prende il nome di *padding*, infatti l'output risulta essere di dimensioni inferiori rispetto a quelle di ingresso (da una matrice iniziale 6x6 ad una 4x4). Il *padding* è una tecnica che verrà illustrata successivamente e viene usato sugli *strati convoluzionali* per assicurare che le forme di altezza e larghezza delle mappe delle caratteristiche in uscita (*feature map*) corrispondano agli ingressi.

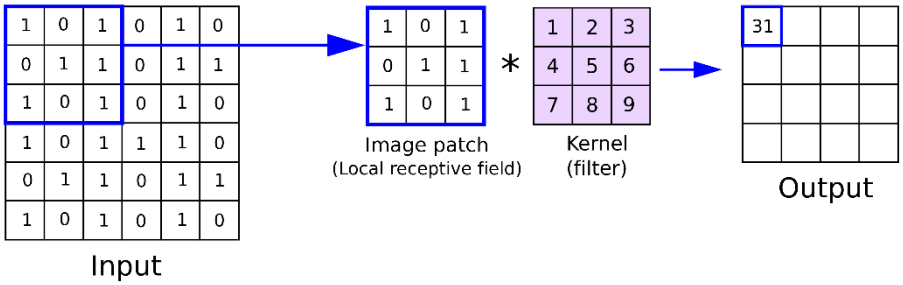

Figura 3.2. Convolution point x point senza padding[9].

Possiamo determinare la dimensione spaziale dell'uscita in funzione del valore della dimensione di input (W), della dimensione del filtro (F), il passo che è stato applicato (S) e infine l'eventuale valore di zero-padding usato sul bordo (P). Con N si intende la dimensione della matrice di output finale [NxN]:

$$\frac{W-F+2P}{S} + 1 = N$$

Nel caso della fig. 2.2.
W = 6
F = 3
P = 0
S = 1 → N = 4

Formula n° 1.

Nella figura 2.2. è riportato un ulteriore esempio dell'operazione di convoluzione, questa volta con un'immagine di ingresso più complessa. L'immagine di uscita è completamente diversa rispetto a quella d'ingresso.

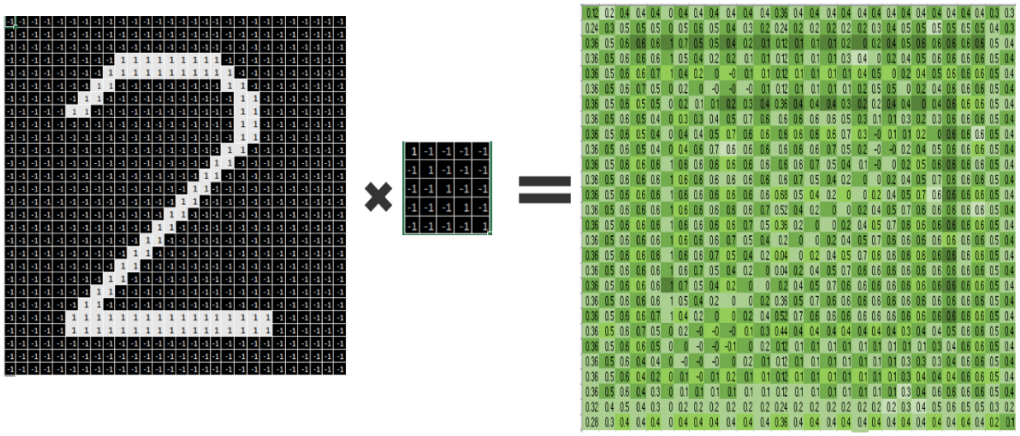

Figura 3.3. Convolution point x point[10].

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

[10] https://www.kaggle.com/fazilbtopal/a-detailed-cnn-with-tensorflow

Possiamo vedere che la mappa delle caratteristiche (*la feature map*) è una matrice 28×28, uguale dimensione dell'immagine di input 28×28 in virtù dell'operazione di *padding* effettuata.

3.2.1. Filtri e canali multipli

Le reti neurali convoluzionali non "imparano" un singolo filtro, esse infatti, imparano più caratteristiche in parallelo per un dato input. Per esempio, è comune per uno strato convoluzionale imparare da un numero ampio di filtri. Anche nel nostro caso nella implementazione della CNN sono stati adottati 3 filtri in cascata.

Un filtro deve sempre avere lo stesso numero di canali dell'input, spesso indicato come "profondità". Se un'immagine di input ha 3 canali (ad esempio una profondità di 3), allora un filtro applicato a quell'immagine deve avere anch'esso 3 canali (ad esempio una profondità di 3). In questo caso, un filtro 3×3 sarebbe infatti 3x3x3 o [3, 3, 3] per righe, colonne e profondità. Indipendentemente dalla profondità dell'input e dalla profondità del filtro, il filtro è applicato all'input usando un'operazione di prodotto di punti che risulta in un singolo valore [8].

3.3 Una soluzione ad un problema: il padding

Si tenga conto che l'applicazione dei filtri ad un'immagine non può essere fatta in modo smodato in quanto, ogni qualvolta applichiamo un filtro di grandezza [F x F] ad un'immagine di grandezza [W x W], il risultato ottenuto sarà quello di una *feature map* di grandezza pari a quella ricavabile dalla formula n.1, portando quindi ad un rischio se l'immagine iniziale dovesse essere "piccola". Questo spesso non è un problema per immagini grandi e filtri piccoli, ma può divenirlo con immagini piccole.

Può anche diventare un problema una volta che un certo numero di strati di convoluzione sono impilati, si ottiene quindi in sostanza una riduzione dell'immagine in uscita rispetto a quella in ingresso, diventando una criticità quando si sviluppano modelli di reti neurali convoluzionali molto profondi con decine o centinaia di strati. In altre parole, si andrebbero ad esaurire i dati nelle *feature maps* su cui operare. Parlando in numeri, ad esempio avendo un'immagine di input con 8x8 pixel utilizzando un filtro di 3x3 pixel si otterrebbe una mappa di caratteristiche con 6x6 pixel, essenzialmente 1 pixel per lato verrebbe "perso".

Il filtro come già anticipato viene applicato sistematicamente all'immagine di input. Iniziando dall'angolo in alto a sinistra dell'immagine e venendo spostato da sinistra a destra una colonna di pixel alla volta fino a quando il bordo del filtro raggiunge il bordo dell'immagine, la riduzione della dimensione dell'input alla mappa delle caratteristiche è chiamata *effetto di confine*, ed è causata dall'interazione del filtro con il bordo dell'immagine.

L'aggiunta del *padding* permette lo sviluppo di modelli molto profondi in modo tale che le mappe caratteristiche non si riducano nel corso dello sviluppo. Per esempio, nel caso dell'applicazione precedente di un filtro 3×3 all'immagine di input 8×8, possiamo aggiungere "un bordo" di un pixel intorno all'esterno dell'immagine. Questo ha l'effetto di creare "artificialmente" un'immagine di input 9×9. Quando il filtro 3×3 viene applicato, si ottiene una mappa di caratteristiche finale di 8×8 pixel, ovvero con la medesima dimensione iniziale. I valori dei pixel aggiunti solitamente hanno valore zero che non ha alcun effetto con l'operazione del prodotto dei punti quando il filtro viene applicato. L'aggiunta di pixel al bordo dell'immagine è chiamata *padding*.

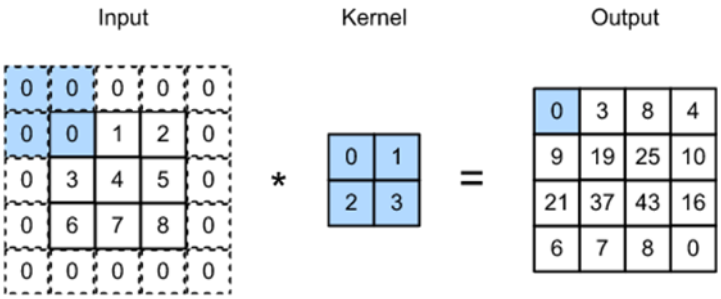

Figura 3.4. Esempio di Convoluzione con padding[11].

$$\frac{W-F+2P}{S} + 1 = N$$

Nel caso della fig. 2.4.
W = 3
F = 2
P = 1
S = 1 → N = 4

3.4 Stride o falcata

Il filtro viene spostato attraverso 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 *falcata*, ed è quasi sempre simmetrica nelle dimensioni di altezza e larghezza. La falcata o le falcate predefinite in due dimensioni sono (1,1) per il movimento in altezza e in larghezza, eseguite quando necessario. Questo default funziona bene nella maggior parte dei casi. La *falcata* può essere cambiata, il che ha un effetto sia su come il filtro viene applicato all'immagine sia, a sua volta, sulla dimensione della mappa delle caratteristiche risultante. Per esempio, il passo può essere cambiato in (2,2). Questo ha l'effetto di spostare il filtro di due pixel a destra per ogni movimento orizzontale del filtro e di due pixel in basso per ogni movimento verticale del filtro quando si crea la mappa delle caratteristiche.

3.5 Attivazioni: la ReLu

Una funzione di attivazione lineare rettificata, o *ReLU* in breve, viene poi applicata ad ogni valore nella mappa delle caratteristiche. Questa è una non linearità semplice ed efficace, che in questo caso non cambierà i valori nella mappa delle caratteristiche, ma è presente perché in seguito aggiungeremo strati di pooling successivi il quale viene aggiunto dopo la non linearità applicata alle mappe delle caratteristiche. Al termine dell'operazione di convoluzione, l'uscita è soggetta ad una funzione di attivazione per consentire la non linearità.

La tipica funzione di attivazione per una CNN è la ReLu, tutti i pixel con un valore negativo verranno sostituiti da zero, è possibile descrivere la funzione con la seguente espressione:

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

È diventata la funzione di attivazione predefinita per molti tipi di reti neurali perché un modello che la utilizza è più facile da addestrare e raggiunge buone prestazioni. La funzione di attivazione più semplice è chiamata attivazione lineare, dove non viene applicata alcuna trasformazione. In altri termini la *ReLu* funziona

[11] https://d2l.ai/chapter_convolutional-neural-networks/padding-and-strides.html

mappando i valori negativi a zero e mantenendo quelli positivi. Questa operazione è talvolta definita attivazione, dal momento che solo le feature attivate vengono trasmesse al layer successivo.

Nella figura *2.5* è riportato un esempio grafico della *ReLu* Function.

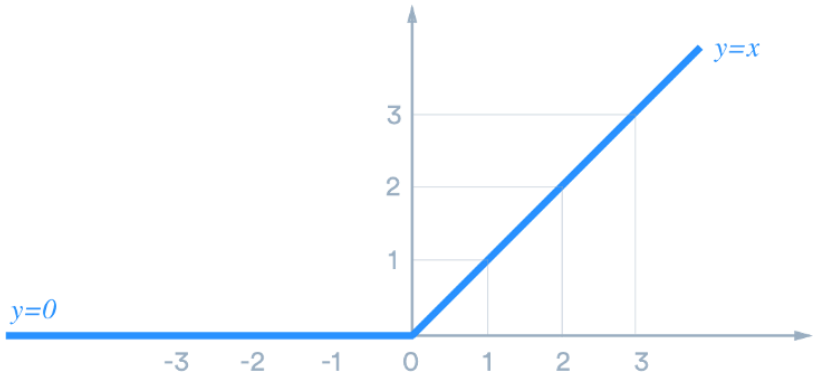

Figura 3.5. ReLu Function[12].

Gli strati convoluzionali si dimostrano molto efficaci, e l'impilamento degli strati convoluzionali nei modelli profondi permette agli *strati vicini* all'input di apprendere caratteristiche di basso livello (per esempio linee) e agli *strati più profondi* nel modello di apprendere caratteristiche di alto ordine o più astratte, come forme o oggetti specifici.

Un approccio comune per affrontare questo problema dall'elaborazione del segnale è chiamato *down sampling*. Si tratta di creare una versione a bassa risoluzione di un segnale di input che contiene ancora gli elementi strutturali grandi o importanti, senza i dettagli fini che potrebbero non essere utili per il compito e porterebbero solo ad un appesantimento del sistema. Il down sampling può essere ottenuto con i livelli di convoluzione cambiando il passo della convoluzione attraverso l'immagine. Un approccio più robusto e comune è quello di usare un *pooling layer*.

3.6 Il Pooling

Uno strato di *pooling* è un nuovo strato aggiunto dopo lo *strato di convoluzione*. In particolare, dopo che una non linearità (per esempio *ReLU*) è stata applicata alle *mappe di caratteristiche* in uscita da uno *strato di convoluzione*, gli strati in un modello appaiono come riportato per sommi capi nella *fig. n°2.6*. L'aggiunta di uno strato di pooling dopo lo strato di convoluzione è un modello comune usato per ordinare gli strati all'interno di una rete neurale convoluzionale che può essere ripetuto una o più volte in un dato modello.

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

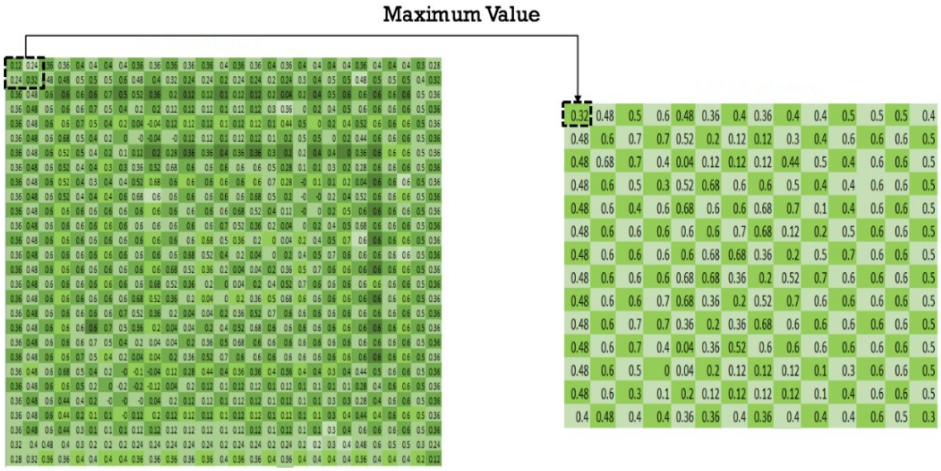

Figura 3.6. Pooling (tipo Max Value) ripetuto su tutta la Feature Map[13].

Il *pooling* comporta la selezione di un filtro da applicare alle mappe di caratteristiche. La dimensione dell'operazione di raggruppamento o filtro è più piccola della dimensione della mappa delle caratteristiche, in particolare, è quasi sempre 2×2 pixel applicata con un passo di 2 pixel. Questo significa che il livello di raggruppamento ridurrà sempre la dimensione di ogni mappa di caratteristiche di un fattore 2, ad esempio ogni dimensione viene dimezzata, riducendo il numero di pixel o valori in ogni mappa di caratteristiche a un quarto della dimensione. Per esempio, un livello di raggruppamento applicato a una mappa di caratteristiche di 6×6 (36 pixel) risulterà in una mappa di caratteristiche di 3×3 (9 pixel). L'operazione di *pooling* è specificata, piuttosto che appresa. Due funzioni comuni, come accennato pocanzi, usate nell'operazione di pooling sono:

- *Pooling medio (o Average Pooling):* Calcola il valore medio per ogni settore sulla mappa delle caratteristiche.
- *Pooling massimo* (o *Max Pooling*): Calcola il valore massimo per ogni settore 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.

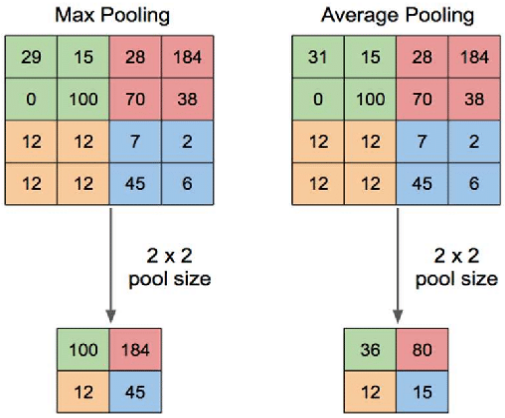

Figura 3.7. Diversi tipi di pooling[14].

Il risultato dell'uso di un livello di *pooling* e della creazione di *mappe di caratteristiche* campionate verso il basso o in pool è una versione riassunta delle caratteristiche rilevate nell'input. Sono utili perché piccoli

[13] https://www.kaggle.com/fazilbtopal/a-detailed-cnn-with-tensorflow

[14] https://www.researchgate.net/figure/Illustration-of-Max-Pooling-and-Average-Pooling

cambiamenti nella posizione della caratteristica nell'input rilevata dallo strato convoluzionario risulteranno in una mappa di caratteristiche raggruppate con la caratteristica nella stessa posizione. Questa capacità aggiunta dal pooling è chiamata invarianza del modello alla traduzione locale.

In tutti i casi, il *pooling* aiuta a rendere la rappresentazione approssimativamente invariante a piccole traslazioni dell'input. L'invarianza alla traduzione significa che se si traduce l'input di una piccola quantità, i valori della maggior parte degli output messi in comune non cambiano. Il modulo di *pooling* si occupa di aggregare l'input e ridurne il volume per mezzo di un *sotto-campionamento* così da snellire l'elaborazione per i layer successivi. Il pooling, così come avviene per la convoluzione, agisce con l'ausilio di un kernel traslato lungo tutta l'immagine.

C'è un altro tipo di raggruppamento che viene talvolta usato, chiamato *raggruppamento globale*. Invece di campionare le patch della mappa delle caratteristiche di input, il pooling globale campiona l'intera mappa delle caratteristiche ad un singolo valore. Il *pooling globale* può essere usato in un modello per riassumere in modo aggressivo la presenza di una caratteristica in un'immagine. A volte è anche usato nei modelli come alternativa all'uso di uno strato completamente connesso per passare dalle mappe di caratteristiche a una previsione di output per il modello. L'ultimo step di costruzione della CNN consiste nel costruire una tradizionale rete neurale artificiale. Si collegano tutti i neuroni dal livello precedente al livello successivo.

3.7 Dropout

Il *Dropout* è un criterio per evitare la creazione delle attivazioni nascoste scegliendo una percentuale (in questo specifico lavoro 50%, ma può variare per applicazioni specifiche) tale per cui questa diventi la probabilità che un nodo si sconnetta da un altro. 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 *training phase*.

3.8 Loss functions

Le *Loss Functions* sono funzioni utilizzate per decodificare l'output delle reti. Tre sono quelle più importanti:

- *Binary Crossentropy*, utilizzata per la classificazione binaria
- *Categorical Crossentropy*, utilizzata per la classificazione multi-classe, restituisce un vettore con un 1 nella posizione del risultato più probabile
- *Sparse Categorical Crossentropy*, utilizzata per la classificazione multi-classe, differisce dalla precedente per la rappresentazione, questa infatti restituisce un numero corrispondente alla classe più probabile.

È riportata sotto una tabella indicante i vari tipi di funzioni di attivazioni adoperate in particolare della tipologia di classificazione utilizzata.

Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`
Regression to arbitrary values	None	`mse`
Regression to values between 0 and 1	`sigmoid`	`mse` or `binary_crossentropy`

Figura 3.8. Loss Function[15].

Le classificazioni di un generico elemento all'interno di un'immagine, è costituito principalmente da tre varietà possibili:

- *Classificazione binaria,* ovvero si associa un valore (Vero o Falso, Si o No..) a partire da una scelta binaria possibile
- *Classificazione multi classe*, il riconoscimento di un oggetto avviene a partire da un ventaglio di scelte possibili
- *Classificazione multi label,* è un avanzamento della classificazione multi classe in cui, all'interno di un'immagine ad esempio sono presenti più classi possibili verificate.

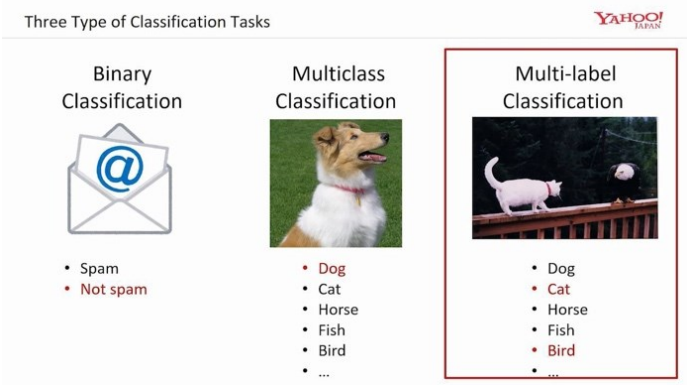

Figura 3.9. Tipologie di Classificazioni[16].

3.9 Ricapitolando

Le *reti neurali convoluzionali* dette anche *CNN* o *ConvNet* rappresentano una classe di *reti neurali* formate da numerosi livelli di profondità in cui le connessioni avvengono in una sola direzione, non permettendo la formazione di cicli. La loro applicazione risulta essere particolarmente performante nell'analisi e nel riconoscimento delle immagini. Di seguito è quindi riportato una sorta di "dizionario" dei termini più salienti.

[15] https://www.kaggle.com/getting-started/153498
[16] https://medium.com/@akhiljanardhanan525

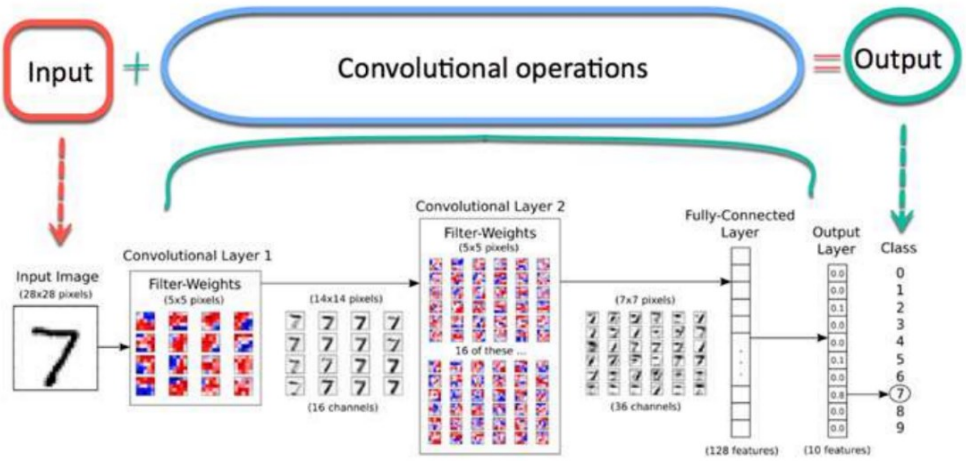

Figura 3.10. CNN[17]*.*

È riportato sopra per semplicità di lettura una classica rappresentazione di uno schema di una *CNN*. Da qui è facilmente ricavabile come l'immagine data "in pasto" alla rete neurale convoluzionale dopo fase di filtraggio ad opera di un *kernel* (sono presenti 3 *feature maps*) ed un'operazione di *subsampling* venga infine inoltrata ad una rete neurale nel senso classico ai fini di fornire come output una risposta alla domanda a che cosa corrisponda l'immagine.

Un *livello convoluzionale* applica l'operazione matematica della convoluzione ai dati che riceve, e rappresenta il principio di funzionamento delle reti neurali convoluzionali, dal quali prendono anche il nome. Nel merito delle reti neurali, si definisce *feature* una qualsiasi informazione rilevante estratta dai dati, utile al calcolo del risultato atteso. La convoluzione, dunque, si occupa di estrarre tali *feature* da ogni immagine in input [9]. La struttura della Rete Neurale Convoluzione prende ogni layer di ingresso e restituisce in input l'output del precedente, dando così una natura fortemente stratificata.

- *Convoluzione:* L'operatore convoluzione rappresenta la componente principale delle *ConvNet* e il suo compito consiste nell'estrazione di caratteristiche dai dati in input. Si consideri ad esempio l'elaborazione di un'immagine. Nei moduli di convoluzione i dati vengono processati con il meccanismo della *sliding window*, dove una piccola matrice chiamata *kernel* di convoluzione viene opportunamente traslata (lungo i vari assi) seguendo la struttura dell'immagine.
- *Feature:* vengono estratte "muovendo" sull'immagine una matrice di dimensione minore, volta per volta, denominata *kernel*, o *filtro*. Può essere immaginata come una piccola torcia che passando sull'immagine illumina solo delle parti per volta. Il risultato di tale operazione prende il nome di *feature map* (le dimensioni della *feature map* diminuiscono attraversando vari livelli di filtri).
- *Profondità:* indica il numero di kernel che devono essere contemporaneamente applicati alla stessa area in esame (nel nostro caso 3), le cui *feature map* saranno sovrapposte disponendosi lungo una terza dimensione.
- *Stride:* indica il numero di pixel che devono essere saltati quando il filtro si sposta sull'immagine, riducendone larghezza e altezza in output.
- *Zero Padding:* indica la possibilità di aggiungere un "cuscinetto" di zeri ai bordi dell'input per consentire una corretta applicazione dei kernel (evita l'effetto di riduzione).
- *Pooling Layer:* Un livello di pooling viene solitamente utilizzato in seguito ad alcuni livelli convoluzionali per ottenere un duplice beneficio: effettua un down sampling dell'input, riducendone la dimensione (e quindi il numero dei parametri), pur mantenendo informazioni sulle feature.

[17] https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions

I pooling layer più utilizzati sono *l'average pooling* ed il *max pooling* che calcolano rispettivamente un valore medio ed un valore massimo. A differenza dei livelli convoluzionali, quelli di pooling non creano nuovi pesi poiché si limitano a calcolare una funzione prefissata sull'input.

- *Layer fully connected:* A valle della rete si trovano uno o più strati di neuroni completamente connessi che applicano le trasformazioni finali e restituiscono il vettore dei punteggi.
- *Rete neurale artificiale:* (Artificial Neural Network, ANN) è un sistema di deep learning ispirato alle reti neurali biologiche. L'elemento costituente più rilevante delle reti neurali è costituito dalle interconnessioni di informazioni. Una rete neurale è organizzata in strati di neuroni, riceve le informazioni d'ingresso attraverso un *input layer* e restituisce i risultati dell'elaborazione per mezzo di un output layer, tra questi strati possono esserci uno o più *hidden layer* [10].

Sostituire uno strato di rete neurale in senso classico, con uno convolutivo, porta ad una drastica diminuzione del numero di parametri (o pesi) all'interno della rete e, quindi, ad un miglioramento nella capacità di estrarre tratti rilevanti. Durante la parte convoluzionale, la rete mantiene le caratteristiche essenziali dell'immagine ed esclude il rumore irrilevante. Ad esempio, il modello sta imparando a riconoscere un elefante da un'immagine con una montagna sullo sfondo. Se si utilizza una rete neurale tradizionale, il modello assegnerà un peso a tutti i pixel, inclusi quelli della montagna che non è essenziale e possono fuorviare la rete. Invece, una rete neurale convoluzionale userà una tecnica matematica per estrarre solo i pixel più rilevanti la convoluzione consente alla rete di apprendere caratteristiche sempre più complesse per ogni livello [11].

3.10 Valutazione di una CNN

Ai fini di avere un ulteriore sicurezza e uno strumento ai fini di verificare il corretto funzionamento, è stato implementato all'interno della rete un ulteriore elemento definito *matrice di confusione*, è un potente ed un efficace strumento di valutazione. 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* e 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 [C x C] dove l'elemento M_{IJ} corrisponde al numero di campioni definiti come *i* ma classificati dalla rete come *j*. M_{II} corrisponde invece ai campioni correttamente classificati.

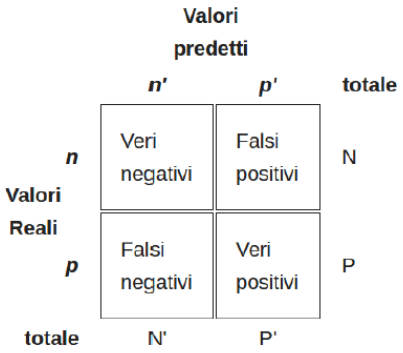

Figura 3.11. Matrice di Confusione[18]*.*

La matrice di confusione più semplice è quella di un problema di classificazione binario, supponendo di avere due etichette, -1 e 1, la matrice di confusione è: M_{11} corrisponde ai campioni etichettati come 1 e classificati

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

come 1, questi elementi sono definiti veri-positivi (TP, dall'inglese *true-positive*). L'elemento M_{12} corrisponde ai campioni classificati come 1 ma etichettati come -1, si parla allora di falsi-negativi (FN). Similmente, M_{21} sono i falsi-positivi (FP) ed infine M_{22} i veri-negativi (TN). La metrica è stata adottata in particolare modo nel corso della valutazione e verifica del data setting attraverso l'IDE di OpenMv.

3.11 Rete Neurale Convoluzionale utilizzata: FR-Net

Le reti neurali convoluzionali già presenti in letteratura sono numerose, e presentano differenti tipologie di *architetture* esistenti, *MobileNet* o *LeNet* per citarne alcune, le quali hanno i più variegati scopi, nel nostro caso la peculiarità era quella della costruzione di una rete in grado di riconoscere differenti tipologie di malattie presenti su diverse specie di piante, un compito relativamente complesso, si è optato per un'architettura di tipo FR-Net già adottata anche nel lavoro di ricerca [1]. L'architettura FR-Net è riportata nella figura *2.12*, comprende 5 strati di peso in totale. In particolare, consiste di 3 strati convoluzionali seguiti ciascuno da una funzione di attivazione *ReLU* e un'operazione di *max-pooling*, infine sono presenti 2 strati completamente connessi con un classificatore *softmax* finale.

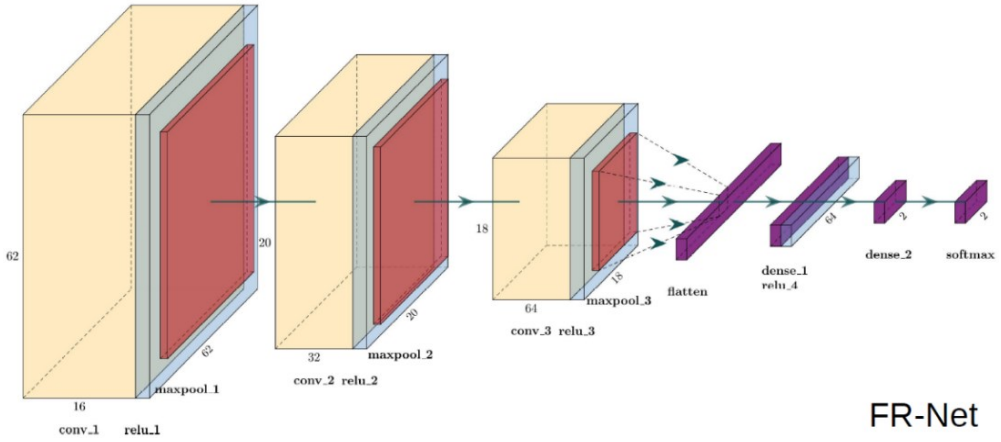

Figura 3.12. Modello di CNN utilizzato[19].

La rete è stata addestrata per 30 epoche sul dataset *PlantVillage* a cui seguirà descrizione a breve utilizzando un ottimizzatore *Adadelta* con entropia incrociata categoriale, un tasso di apprendimento di 0.5 e un batch di 128. Le dimensioni delle immagini del data set sono quindi state limitate a 64x64 pixel ai fini di normalizzare il tutto, una CNN non lavora con immagini avente differenti valori di altezza e larghezza, le dimensioni dell'immagine sono conferite allo strato di input, il primissimo layer di una CNN.

3.12 Sommario della CNN FR-Net

Di seguito viene riportato il criterio per la costruzione e assemblaggio della rete neurale convoluzionale.

- Un primo strato di *Convoluzione*, con un *kernel* 3 x 3
- Un primo strato di *Pooling* 3 x 3
- Un secondo strato di *Convoluzione*, ancora con un *kernel* 3 x 3
- Un secondo strato di *Pooling* identico al primo
- Un terzo strato di *Convoluzione*, ancora con un *kernel* 3 x 3

[19] https://ieeexplore.ieee.org/document/9491080

- Un terzo strato di *Pooling* 2 x 2
- Uno strato *Flatten*: questo strato viene utilizzato per rendere un ingresso multidimensionale unidimensionale, comunemente usato nella transizione dallo strato convoluzionale allo *Fully Connected*
- Un primo strato *Fully-Connected*, che usa una funzione di attivazione *ReLU*
- Un primo strato *Dropout*: il *dropout* è una tecnica in cui i neuroni selezionati casualmente vengono ignorati durante l'allenamento della rete, il che aiuta a prevenire un sovradattamento.
- Un secondo strato *Fully-Connected*, che usa una funzione di attivazione *Softmax*. Genera un vettore N dimensionale in cui N è il numero di *classi* tra cui il programma deve scegliere. Nel nostro caso N=39. Se il vettore risultante per il nostro programma di classificazione (tipo di cifre numeriche) fosse ad esempio:

  [0 0 15 10 0 0 0 65 0 10]

  allora questo rappresenterebbe una probabilità del 15% che l'immagine sia 2, una probabilità del 10% che l'immagine rappresenti un 3, una probabilità del 65% che l'immagine sia un 7 e una probabilità del 10% che l'immagine sia un 9 (tutti gli altri numeri hanno probabilità nulla di essere scelti). Per cui, scegliendo l'elemento con maggiore probabilità, si "predice" che l'immagine proposta in ingresso sia 7.

Viene di seguito riportato per completezza il modello di CNN sviluppato (presente all'interno del codice) nel complessivo come modello tridimensionale.

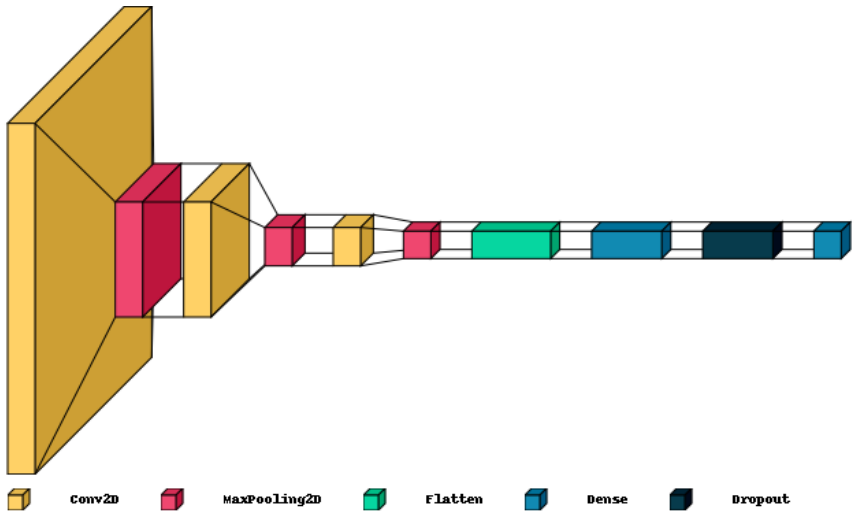

Figura 3.13. Modello di CNN sviluppato nel lavoro.

3.13 Il dataset PlantVillage

Il lavoro è proseguito successivamente dopo la conclusione della fase di studio della struttura della CNN, ad uno studio orientato all'utilizzo delle stesse. In primo luogo, è nata una sorta di domanda su cosa si debba fornire alla rete affinché possa "imparare" a distinguere se nell'immagine è presente qualcosa di noto e richiesto o meno. Il dataset è un insieme di dati (che possono essere sotto forma di vettori, immagini, suoni ecc.) suddiviso in classi con ogni elemento associato ad una ed una sola di queste. Vengono utilizzati *in fase di addestramento* ed è estremamente importante che questi sia costruito e utilizzato nella maniera corretta.

I dataset possono essere assemblati dall'utente o tranquillamente scaricabili dalla rete, si tenga presente che un dataset può arrivare a contenere migliaia di foto e devono contenere una maggiore varianza possibile di foto dell'oggetto trattato.

Dopo la *fase di addestramento* della rete, quest'ultima è utilizzabile per effettuare classificazioni relative all'ambito del data set utilizzato. Ne discende quindi l'importanza di utilizzare un data set appropriato e ben studiato in quanto, se si utilizzasse un dataset in senso non coscienzioso la rete potrebbe non "imparare"

bene a distinguere le varie immagini o peggio ancora, potrebbe imparare a memoria. Internet in tale senso ci fornisce una grande mano in quanto oggi giorno sono presenti data set già impostati creati per addestrare le *CNN* con fotografie di vario tipo comprendenti vari oggetti di uso comune e non (animali, oggetti, persone…). Alcuni tra i data set più conosciuti ci sono CIFAR-10 (contiene 10 classi di animali diverse tra loro), MNIST (cifre da 0 a 9 scritte a mano) e così via.

Quindi è importante fornire ad una *CNN* il data set specifico allo scopo da noi desiderato, le reti neurali sono costruite per analizzare immagini incluse dentro certi set di dati (ad esempio dataset di animali includono immagini di animali, dataset di riconoscimento facciale includono immagini di volti, dataset di autoveicoli includono immagini di veicoli, e così via), e classificare gli oggetti nelle immagini al loro interno. Non è assolutamente possibile, ad esempio, ottenere un riscontro positivo da un computer nell'analisi di un'immagine di un volto umano a una rete neurale convoluzionale che accetta solamente immagini di autoveicoli, in quanto non capirebbe le forme e gli oggetti che la nuova immagine rappresenta.

Figura 3.14. Logo PlantVillage[20].

Nel caso del presente progetto è stato usato come data set PlantVillage, contenente 61486 immagini di foglie di colture agricole, afflitte da vari tipi di malattie quali virus, batteri e funghi [20]. Queste immagini di piante includono i seguenti 14 tipi di specie diverse: mela, mirtillo, ciliegia, mais, peperone, patata, lampone, uva, arancia, pesca, soia, zucca, fragola e pomodoro.

Di seguito è riportato l'elenco delle 38 classi di malattie delle piante presenti nel dataset: scabbia del melo, marciume nero del melo, ruggine del cedro del melo, mela sana, mirtillo sano, ciliegia sana, oidio della ciliegia, macchia fogliare grigia del mais, ruggine comune del mais, mais sano, mais ruggine fogliare settentrionale, marciume nero dell'uva, morbillo nero dell'uva, peronospora fogliare dell'uva, uva sana, huanglongbing arancione dell'arancio, macchia batterica della pesca, pesca sana, macchia batterica del peperone, peperone sano, peronospora precoce della patata, patata sana, peronospora tardiva della patata, lampone sano, soia sana, oidio della zucca, fragola sana, foglia di fragola scorbutica, macchia batterica del pomodoro, peronospora precoce del pomodoro, peronospora tardiva del pomodoro, muffa fogliare del pomodoro, macchia fogliare della septoriosi del pomodoro, acaro del ragno a due macchie del pomodoro, macchia a bersaglio del pomodoro, virus mosaico del pomodoro, virus dell'arricciamento fogliare giallo del pomodoro, pomodoro sano, sfondi vari. Si può osservare come il dataset in questione sia molto ricco in termini di varietà di classi e di immagini diversificate presenti.

L'utilizzo del dataset è essenziale durante le fasi di *testing* e *training* della rete come già anticipato, e risulta quindi importante utilizzarne solo una data parte per ogni fase di costruzione della rete, infatti, è stata eseguita un'operazione di splitting del dataset, sulla falsa riga di quanto visto in [2] per la fase di *training*, *testing* e *validation* pari al 60%, 25% e 15% rispetto il dataset originario.

[20] https://www.crowdai.org/challenges/1

CAPITOLO 4
Implementazione di una CNN

Dopo la fase iniziale di studio teorico, sono riportati gli step più pragmatici che sono stati seguiti per l'implementazione della rete, dando uno sguardo agli strumenti software e hardware utilizzati a tal fine.

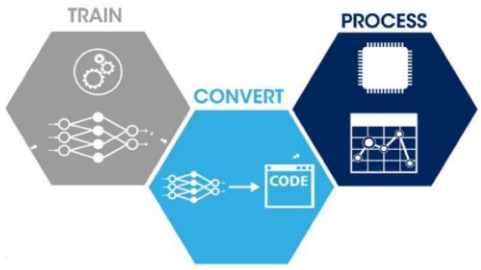

Figura 4.1. Processo di Sviluppo[21].

Il primo step da realizzare è stabilire la struttura del modello, come per esempio il numero di strati e le loro dimensionalità, la trattazione è stata già realizzata in precedenza. Una volta scelta la struttura, il modello va innanzitutto costruito, poi va *addestrato* (processo di *training*), dopodiché si realizza il *testing* del modello, se questo presenta una accuratezza bassa (una accuratezza si ritiene non accettabile se inferiore al 80/90%) la struttura del modello va riprogettata. Questo procedimento prosegue fino a quando si trova un modello con un valore di accuratezza accettabile. La rete viene ad essere realizzata e testata dapprima su PC in linguaggio TensorFlow, successivamente viene ad essere esportata e testata sulla piattaforma embedded.

4.1 Visione d'insieme

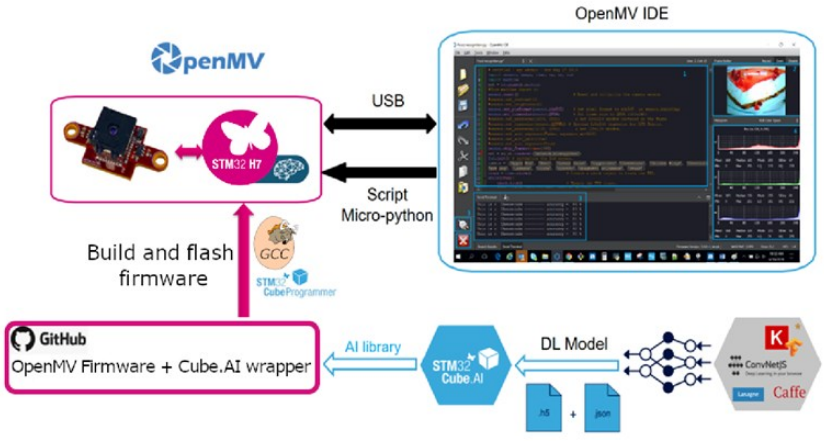

Figura 4.2. Processo di integrazione di una CNN nell'ambiente OpenMV tramite il tool STM32Cube.AI[22].

La rete viene scritta in linguaggio Python, e realizzata tramite la libreria *Keras* (in questo lavoro), che si appoggia sul linguaggio *TensorFlow*. Questo file Python, utilizzando uno specifico dataset, realizzerà sia il

[21] https://wiki.st.com/stm32mcu/wiki/AI:How_to_add_AI_model_to_OpenMV_ecosystem
[22] https://wiki.st.com/stm32mcu/wiki/AI:How_to_add_AI_model_to_OpenMV_ecosystem

training che il *testing* della rete, questa verrà salvata in formato *.h5 e .tflite*. Dopodiché la rete verrà esportata sulla piattaforma embedded OpenMV, dopo una preliminare fase di "alleggerimento" mediante *TensorFlowLite* (file .tflite). Vengono di seguito riportati gli step di riepilogo del lavoro.

1) Realizzazione della CNN su PC: Il primo obiettivo di questa fase è la creazione del file Keras *.h5*.
    1.1) Caricamento del dataset
    1.2) Costruzione della CNN
    1.3) Configurazione della CNN
    1.4) Training e testing della CNN
    1.5) Salvataggio del modello e dell'architettura (accuratezza >80%)

2) Esportazione della CNN su piattaforma embedded
    2.1) Compilazione, a mezzo di file *.h5* o in alternativa con file *.tflite*
    2.2) Validazione del modello sulla piattaforma embedded

CAPITOLO 5
Strumenti di sviluppo

5.1 Strumento hardware utilizzato: OpenMV H7

Successivamente alla fase di studio preliminare della teoria inerente alle reti neurali convoluzionali (CNN), e a riportare per sommi capi una panoramica sul percorso logico che ha gradualmente step per step portato alla programmazione della scheda, si è passati ad una analisi degli strumenti sia software che hardware che sono venuti ad essere adoperati. Il primo dispositivo oggetto del lavoro trattasi della piattaforma embedded.

La scheda in questione è l'*OpenMV Cam* [21], una piccola scheda elettronica a bassa potenza che permette di implementare facilmente applicazioni che utilizzano la visione artificiale nel mondo reale. 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. La programmazione dell'*OpenMV Cam* avviene in script *Python* di alto livello invece che in *C/C++,* in modo da rendere più attuabile affrontare i complessi output degli algoritmi di visione.

Figura 5.1. OpenMv H7 Plus[23].

L' *OpenMV Cam* (figura 4.1) 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 12Gbs. E' presente inoltre una presa per scheda microSD in grado di leggere/scrivere a 100Mbs permettendo alla OpenMV Cam di scattare foto e di estrarre o inserire risorse di visione artificiale nella scheda µSD, peculiarità infatti che verrà utilizzata successivamente nella fase di implementazione. Ulteriori caratteristiche tecniche sono presenti sul sito internet dell'azienda produttrice della scheda [21]. 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 ulteriori tipologie di fotocamere sulla scheda.

L'applicazione in generale dell'apprendimento automatico (ML) per il rilevamento e la classificazione degli oggetti sta diventando molto importante nel settore dei sistemi embedded, la sicurezza, i sistemi avanzati di assistenza alla guida, oltre che i sistemi basati sull'automazione industriale. Il rilevamento di oggetti è un

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

argomento complesso e l'apprendimento automatico è relativamente nuovo, per cui sviluppare applicazioni ML per questo scopo è relativamente difficile [21].

La piattaforma OpenMV, con un costo ridotto (circa 80€ al 2021) permette l'elaborazione e il rilevamento delle immagini. Il microcontroller su scheda è l'*STM32H743II* della *STMicroelectronics*, che include un processore ARM Cortex-M7 in un LQFP a 100 pin. Il processore opera a 216 MHz e ha 2 MB di flash e 512 kB di RAM. È estremamente efficiente e particolarmente idoneo per applicazioni di visione artificiale grazie alla sua unità a virgola mobile (FPU).

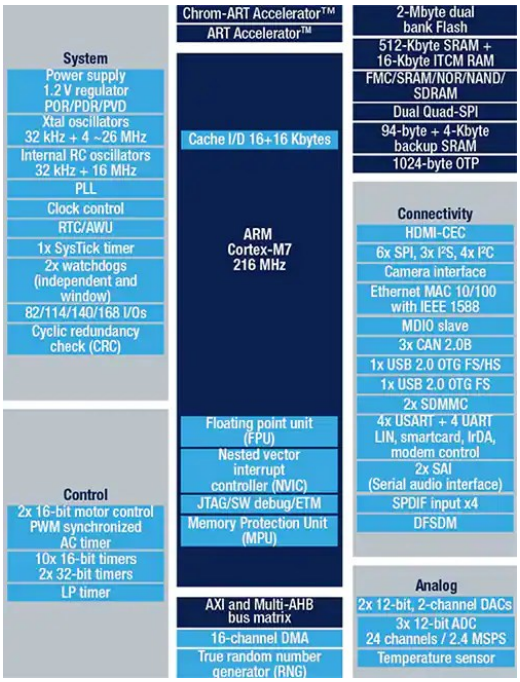

Figura 5.2. Visione d'insieme OpenMv H7[24].

La programmazione della scheda richiede, come anticipato nel paragrafo appena terminato, un proprio IDE di programmazione (scaricabile liberamente da [21]) di cui verrà svolta trattazione a seguire. 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.

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 realtà, è di facile implementazione meccanica magari a bordo di un sistema semovente in quanto di poco ingombro e leggera, è relativamente piccolo, 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 [12] con cui è possibile documentarsi.

5.2 Strumenti Software

Per poter programmare o meglio, costruire un modello di CNN che potesse essere d'accordo con le nostre specifiche descritte nei paragrafi precedenti, si è proceduto ad uno studio inerente ai programmi e strumenti che permettessero una programmazione più fluida e facilitata possibile. Della vasta galassia degli

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

ambienti di programmazione possibili, sono stati scelti quelli tra i quali potessero soddisfare le nostre richieste e che sono usati universalmente per i medesimi utilizzi: *Tensorflow* e *Keras* utilizzando come linguaggio di programmazione *Python*.

Solitamente, sono tre i principali linguaggi di programmazione utilizzabili per implementare algoritmi di Machine Learning: *Python, C++, Matlab*. Tutti questi linguaggi permettono, attraverso l'uso di apposite librerie, di effettuare operazioni di interesse per le *CNN*.

L'utilizzo di *Python* come linguaggio di programmazione deriva dalla 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.

- *STM32CubeMX,* è uno strumento grafico che consente una configurazione relativamente semplice di microcontrollori e microprocessori STM32, nonché la generazione del corrispondente codice di inizializzazione C per il core ARM Cortex-M
- Estensione STM X-CUBE-AI dell'IDE STM32CubeMX
- *STM32CubeIDE*, è una piattaforma di sviluppo C / C ++ avanzata con funzionalità di configurazione IP, generazione del codice, compilazione del codice e debug per i microcontrollori STM32.
- *Python, TensorFlow, Keras* sono librerie utilizzate per lo sviluppo del modello.
- *OpenMv IDE*, editor di testo per la programmazione della piattaforma
- *GoogleColab*, per l'utilizzo di *TensorFlow* e *Keras*

5.2.1 TensorFlow

Figura 5.3. Logo TensorFlow[25].

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, 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 gli sviluppatori di creare e distribuire facilmente applicazioni basate su ML [22]. 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.

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 utilizzati direttamente da TensorFlow.

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

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 già ed inoltre sono presenti numerosi esempi sul sito ufficiale della libreria [13].

5.2.2. Keras

Figura 5.4. Logo Keras[26].

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.

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 fatto infatti nel momento in cui di progetta una rete neurale in funzione del fatto che le stesse sono algoritmi che richiedono un tempo relativamente lungo (circa 150s ad epoca nel nostro caso) 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.

5.2.3 Google Colab

Per poter utilizzare queste librerie *TensorFlow* e *Keras*, tra i vari metodi possibili si è optato per semplicità e praticità di adottare l'utilizzo di *GoogleColab*, è necessario soltanto avere un account Google da cui fare accesso e adoperare questo servizio offerto dalla Google (i lavori vengono salvati all'interno del proprio GDrive), è relativamente intuitivo per poter programmare in Python e assemblare delle CNN, non sono necessarie configurazioni iniziali (spuntare la spunta "connetti"). Di seguito è riportata l'immagine raffigurante la intro di GoogleColab accessibile facilmente dal link che segue [23].

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

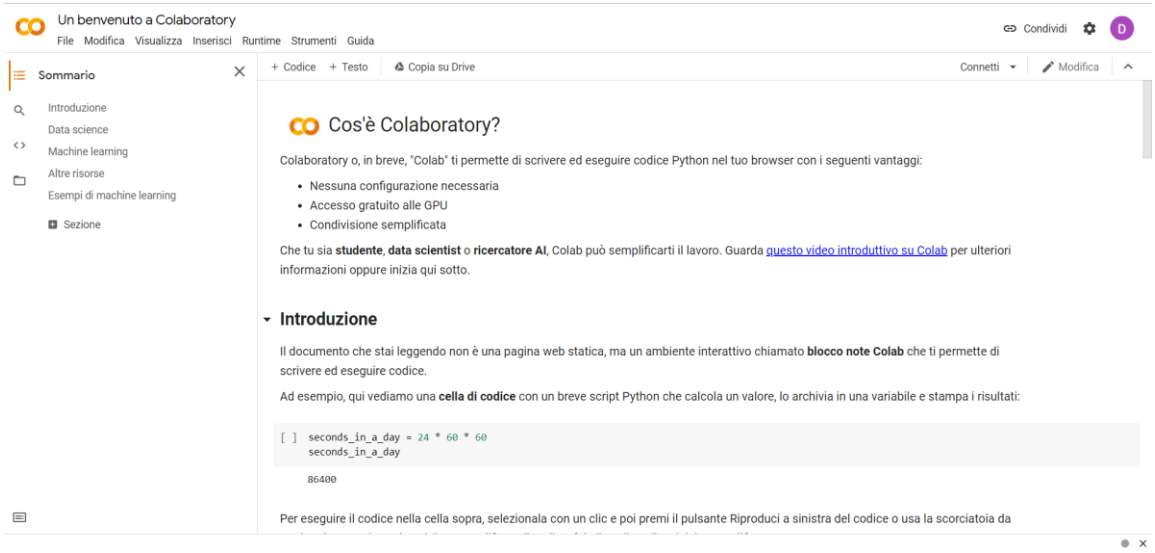

Figura 5.5. Schermata GoogleColab[27].

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

Google Colab sfrutta i cosiddetti Jupyter Notebook. Questi non sono altro che documenti interattivi nei quali e 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. Nei capitoli successivi verrà presentato il lavoro svolto, a partire dalla preparazione dell'ambiente di lavoro fino alle sperimentazioni effettuate con l'algoritmo implementato.

5.2.4. STM32 CubeX

Come passaggio intermedio tra il mondo "virtuale", per così dire delle CNN a quello esecutivo della piattaforma embedded si è condotto con l'utilizzo del software STM32 per la programmazione in C del firmware del dispositivo. Il software è liberamente scaricabile dal sito della casa madre [24].

Per quanto concerne questa fase di svolgimento del progetto si fa riferimento al paragrafo 6.2 della presente, in quanto trattasi, chiarito l'obbiettivo finale preposto, di una fase di costruzione del modello in C del programma ai fini di verificare praticamente, l'effettivo funzionamento corretto della rete.

5.2.5. OpenMV IDE

Per la programmazione della piattaforma *OpenMV* si è dovuto ovviamente utilizzare il software della casa madre, scaricabile liberamente dal sito internet [21] scegliendo opportunamente il sistema operativo, è riportato come segue uno screen di un esempio presente nella demo del software.

[27] https://colab.research.google.com/

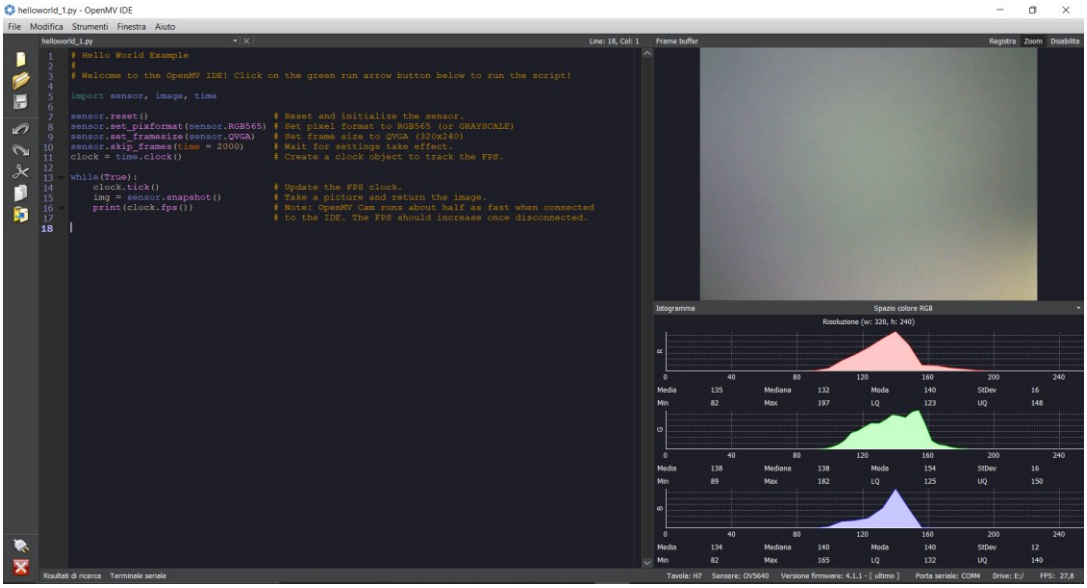

Figura 5.6. Schermata OpenMv IDE[28].

Si vuole evidenziare in questo caso come l'applicativo OpenMV, essendo in piena fase di sviluppo presenti quasi settimanalmente degli aggiornamenti software per implementare varie librerie, ciò è un ulteriore dimostrazione di come le CNN o per meglio dire il machine learning sia in pieno sviluppo. Il programma presenta inoltre diverse funzionalità quali la visione della 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* PC-OpenMV.

Nella sezione successiva è riportato il codice commentato con la descrizione dei passaggi salienti, i codici sono riportati in due fasi differenti, la realizzazione della CNN su PC mediante l'utilizzo di TensorFlow e l'esportazione della CNN sulla piattaforma embedded.

[28] https://openmv.io/

CAPITOLO 6
Sviluppo del Codice e Prove Sperimentali

6.1 Realizzazione della CNN su PC

Il primo step da realizzare è stabilire la struttura del modello, come per esempio il numero di strati e le loro dimensionalità, ai fini del presente lavoro è stato preso come riferimento la rete CNN FR-Net presente in [1]. A seguire allo stadio di studio iniziale, si è passati alla parte di programmazione e di sviluppo vero e proprio del progetto. Il software utilizzato per la costruzione della rete è *TensorFlow*. Non è riportato nella sua interezza il codice, bensì è stato suddiviso in blocchi in modo da facilitarne le motivazioni che hanno spinto in determinate scelte.

6.1.1. Librerie utilizzate

NumPy: Una libreria per il linguaggio di programmazione Python, che aggiunge il supporto per grandi array multidimensionali e matrici, insieme ad una vasta collezione di funzioni matematiche di alto livello per operare su questi array.

Cv2 (OpenCV): OpenCV è una libreria di binding progettata per risolvere problemi di computer vision.

Os: Il modulo Os in Python fornisce funzioni per creare e rimuovere una directory (cartella), recuperare il suo contenuto, cambiare e identificare la directory corrente, ecc. È anche possibile eseguire automaticamente molti compiti del sistema operativo.

Keras: Keras è una libreria di reti neurali open-source scritta in Python. Progettata per consentire una rapida sperimentazione con reti neurali profonde, si concentra sull'essere facile da usare, modulare ed estensibile.

Matplotlib: Una libreria di plottaggio per il linguaggio di programmazione Python e la sua estensione di matematica numerica.

6.1.2. Creazione del Modello

Viene riportata di seguito la programmazione effettuata ai fini della realizzazione del modello prescelto.

Struttura del Modello

```
### Creazione del Modello
# La costruzione della rete neurale richiede la configurazione dei livelli del modello, quindi la compilazione.

# Il modello è costituito da 3 Convolution Layer con 2 Fully Connected con uno strato di pool massimo in ciascuno di essi.
num_classes = n_classes                                          # la rete restituisce 39 (classi) tipologie diverse di malattie delle piante

model = Sequential()                                             # creazione della rete

# Feature Learning block:                                        # blocco di Feature
model.add(Conv2D(16, kernel_size=(3,3), padding='same', activation='relu',input_shape=input_shape)),    # primo strato di convoluzione - padding - relu function
model.add(MaxPooling2D(pool_size=(3,3))),                        # primo strato di max pooling (3x3)
model.add(Conv2D(32, kernel_size=(3,3), padding='same', activation='relu')),       # secondo strato di convoluzione- padding - relu function
model.add(MaxPooling2D(pool_size=(3,3))),                        # secondo strato di max pooling (3x3)
model.add(Conv2D(64, kernel_size=(3,3), padding='same', activation='relu')),       # terzo strato di convoluzione- padding - relu function
model.add(MaxPooling2D(pool_size=(2,2))),                        # terzo strato di max pooling (2x2)

# Classification block:                                          # blocco di Classificazione
model.add(Flatten()),                                            # primo strato di Fully Connected da 3D a 1D --> i pixel vengono "spalmati"
model.add(Dense(units = 512, activation='relu')),                # primo strato di Rete Neurale 1024 ingressi
model.add(Dropout(0.5))                                          # utilizzo del dropout (0.5)
model.add(Dense(units = num_classes, activation = 'softmax')),   # strato finale di uscita (39) uscite corrispondenti ciascuno ad una singola classe
```

Figura 6.1. Struttura del Modello.

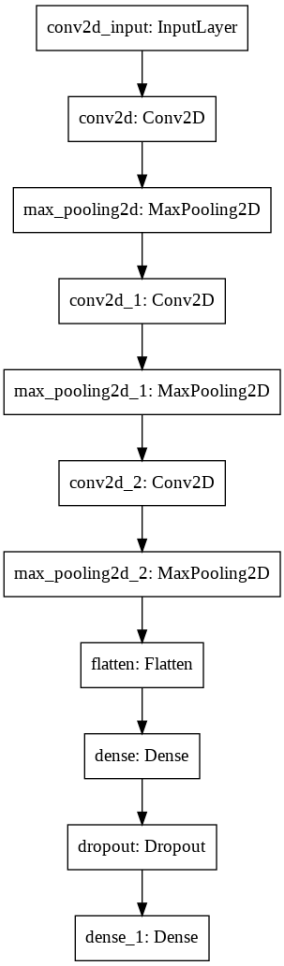

Figura 6.2. Schematico del Modello.

6.1.3. Aggiunta del Dropout

Il dropout funziona rimuovendo, o "abbandonando" gli input ad uno strato, che possono essere variabili di input nel campione di dati o attivazioni da uno strato precedente. Ha l'effetto di simulare un gran numero di reti con strutture di rete molto diverse e, a sua volta, rende i nodi della rete generalmente più robusti agli input [14]. Solitamente il dropout viene applicato dopo la conclusione di uno strato completamente connesso vicino allo strato di uscita del modello, nel nostro caso il dropout è stato scelto a 0.5.

6.1.4. Configurazione della 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 controlla il tasso di apprendimento. È stato utilizzato "*Adadelta*" come nostro optmizer. *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 d'individuare, attraverso una serie d'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. 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 accuratezza per vedere il punteggio di precisione sul set di validazione quando addestreremmo il modello.

```
[ ]  ### Compilazione del Modello
     # Prima di allenare il modello, dobbiamo definire alcune impostazioni che sono aggiunte al modello durante la compilazione.
     # - Loss Function - Optimizer - Metrics
     # Loss Function -> Definisce il criterio con cui valutare l'accuratezza del modello durante l'allenamento, l'obiettivo è quello di minimizzare la funzione
     # Optimizer     -> Definisce il criterio di aggiornamento del modello sulla base dei dati osservati e della Loss Function
     # Metrics       -> Risponde alla domanda se il modello sta funzionando, monitorando il training e il testing del modello, in questo Tensorflow viene usato l'accuracy.

     start = time.time()                                                          # timer per conteggio di durata dell'operazione

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

Figura 6.3. Compilazione.

6.1.5. Training e Testing della CNN

Il numero di epoche è il numero di volte in cui il modello scorrerà i dati, ovviamente più epoche facciamo eseguire alla rete e più il modello migliorerà, fino a un certo punto ovviamente. Dopo quel punto, il modello smetterà di migliorare durante ogni epoca. Un'epoca descrive il numero di volte che l'algoritmo vede l'*intero* set di dati, ogni volta che l'algoritmo ha visto tutti i campioni nel set di dati, è stata completata un'Epoca, nel nostro caso 30. Una *iterazione* descrive il numero di volte in cui un *batch* dei dati passa attraverso l'algoritmo. Nel caso di reti neurali, ciò significa il passaggio e *il passaggio all'indietro.* Pertanto, ogni volta che si passa una serie di dati tramite NN, è stata completata un'iterazione. Si tenga presente che la fase di allenamento della rete impegna un tempo relativamente lungo (circa 150s a epoca nel presente caso).

```
### Allenamento del Modello --> Fase di Training
# Allenare il modello significa fornirgli i dati di allenamento e chiederli d'imparare le associazioni tra immagini e label.
# allenamento effettuato in 30 epoche

history = model.fit(            # immagini di train, batch_size -->32, epochs --> 30
      train_ds,                 # dataset di train, fase di training della rete
      epochs=30,                # numero di epoche di allenamento della rete
      batch_size = 32,          # definisce il numero di immagini che saranno propagati attraverso la rete
      validation_data=val_ds,   # dataset di validation, fase di validazione dei dati
      verbose=True)             # modalità di "grafica"

print('Tempo utilizzato per il training: {} sec\n'.format(time.time() - start))
```

Figura 6.4. Allenamento.

6.1.6. Grafici: Accuracy e Loss

Nella programmazione ai fini di valutare concretamente il comportamento della rete in merito *all'accuratezza* e il *loss*, si è optato per la costruzione di grafici tale per cui i tracciati di queste misure sulle epoche di allenamento forniscano le curve di apprendimento che possiamo usare per avere un'idea se il modello è in *overfitting*, *underfitting*, un buon adattamento o ulteriori qualità.

Si parla generalmente di *underfitting* quando il modello è dimensionato in modo tale che il risultato delle previsioni (quindi l'errore) sia nella fase di test, sia della fase di training restituisce errori molto elevati. Si parla di *overfitting* quando il modello restituisce risultati ottimali in fase di training mentre restituisce risultati poco ottimali in fase di test. Si parla di modello *well-fitted*, quindi ben dimensionato, quando lo stesso restituisce risultati buoni in fase di test e di training [15].

- Il modello è soggetto a *underfitting* quando ha prestazioni scarse sui dati di addestramento in quanto non è in grado di ridurre l'errore tra gli esempi di input e i valori di uscita.
- Il modello è soggetto a *overfitting* quando funziona bene con i dati di addestramento, ma non con i dati di test, perché il modello "memorizza" i dati che ha osservato, ma non è in grado di generalizzare il modello con dati non osservati.

Dall'esame delle curve di apprendimento per il modello durante la formazione possiamo vedere come le figure, rispetto all'aumentare del numero di epoche prese in considerazione abbiamo un andamento crescente per quanto riguarda il *validation* e un andamento decrescente per quanto riguarda il *validation loss*, sostanzialmente aumentando il numero delle epoche (fino ad un certo punto) il nostro modello sta migliorando le proprie caratteristiche di riconoscimento. Inoltre, è possibile vedere come sia presente una sorta di "distanza" tra la curva di *training* e quella di *validation accuracy* e *loss validation*, ciò è un sintomo di limitato overfitting nel modello.

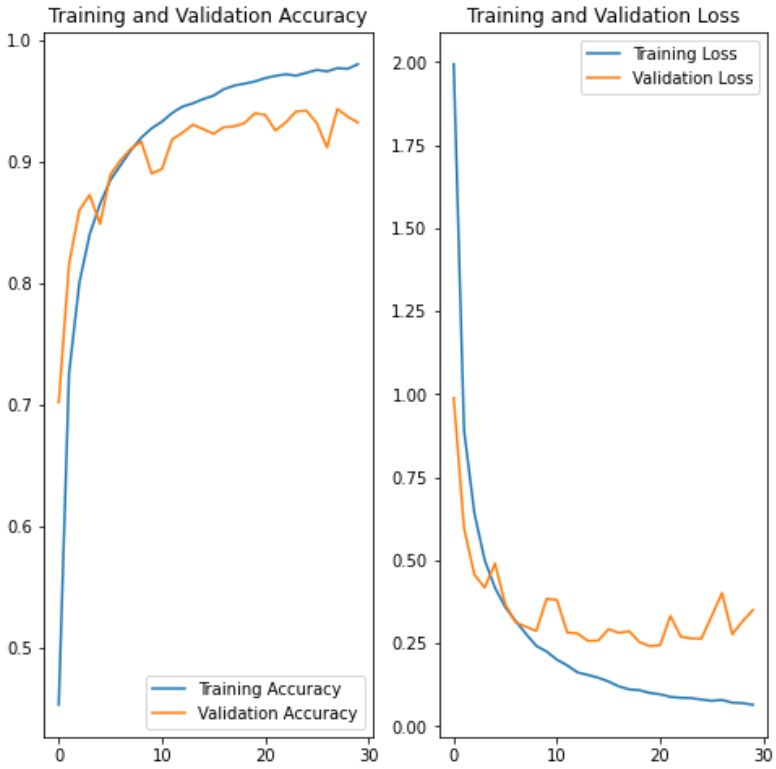

Figura 6.5. Grafici Validation e Accuracy Training e Loss.

Alla fine di 30 epoche, possiamo vedere che il nostro modello è stato sovradimensionato poiché la precisione dell'addestramento continua ad aumentare mentre l'accuratezza della convalida diventa stagnante intorno a 0,90. I grafici indicano che dovremmo interrompere l'allenamento intorno a 15 epoche poiché oltre tale tempo non c'è un reale miglioramento nel nostro modello.

6.1.7. Salvataggio del Modello e dell'Architettura

L'ultimo passaggio da realizzare è il salvataggio del modello e dell'architettura in un singolo file Keras *h5* e, dopo l'opportuna fase di TFLite a breve riportate, anche di un ulteriore file *tfite*, necessario per poter esportare la rete sulla piattaforma embedded. Tale passaggio potrebbe sembrare comune, ma in realtà è molto importante in quanto permette di avere concretamente in mano i modelli della CNN.

```
# Salvataggio del modello in due file, il primo (name_modelTF verrà usato nel TFLite),
# il secondo (name_modelTF_view) verrà usato nella validazione visiva delle immmagini

model.save(name_modelTF)          # salvataggio del modello nel file "name_modelTF.h5"
model.save(name_modelTF_view)     # salvataggio del modello nel file "name_modelTF_view.h5", usato della fase di conferma visiva
```

Figura 6.6. Salvataggio.

Si tenga inoltre presente che, all'interno del codice originario disponibile in [26] sono presenti anche sezioni di programma in cui viene visualizzato mediante opportuni comandi, le immagini del Dataset e altre funzionalità non strettamente necessarie. Per questioni di brevità qui si è tenuto a riportare solamente i passaggi indispensabili ai fini della costruzione della rete (creazione – compilazione - allenamento).

6.1.8. TensorFlowLite

Per questo progetto, dovendo portare la struttura di una rete neurale convoluzionale (CNN) su un sistema embedded, si è dovuto ricorrere a TensorFlow Lite [16], un set di tool che permette di far funzionare modelli TensorFlow su dispositivi mobile, IoT e sistemi embedded. 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. Proprio per questo motivo nasce *TensorFlowLite* che, grazie ai suoi componenti, permette di fare inferenza con bassa latenza e dimensioni dei file contenute, in relazione al modello e al contesto scelto. Di seguito è riportata una veduta del percorso logico da seguire.

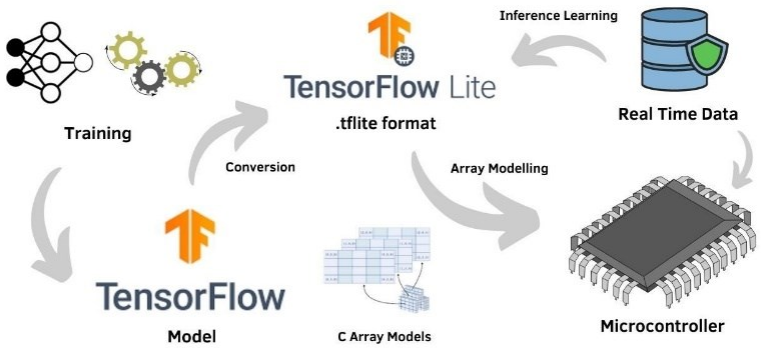

Figura 6.7. Panoramica uso TF & TFLite[29].

TensorFlowLite fa parte dei mezzi messi a disposizione dal framework per predire con nuovi dati dopo che un nostro modello è stato addestrato e rifinito. Lo scopo primario è quello di portare gli algoritmi di machine learning su tutti i dispositivi con limitate capacità computazionali, quale ad esempio l'OpenMV, e più in

[29] https://www.makeuseof.com/what-is-tensorflow-lite-and-how-is-it-a-deep-learning-framework/

generale qualsiasi dispositivo non molto performante siano principalmente delle sorgenti di acquisizione di dati [17]. Tutto questo però si scontra con l'avere a che fare con dispositivi con limitata memoria primaria (RAM, cache), potenza computazionale, storage e in alcuni casi potenza elettrica (autonomia massima dettata dalla batteria e potenza assorbita) [1]. I componenti principali da cui TensorFlow Lite è composto sono:

- L'interprete: questo ha lo scopo di far girare i modelli ottimizzati sui vari dispositivi quali smartphone, sistemi embedded o microcontrollori.
- Il convertitore: questo ha lo scopo di convertire il modello TensorFlow "originale" in una forma più efficiente in modo tale che l'interprete possa utilizzarlo in maniera ottimizzata.

Il convertitore ha il compito principale di ottimizzare il modello riducendo le sue dimensioni e aumentando la sua velocità di esecuzione. Questo viene principalmente ottenuto con due tecniche fondamentali: *model quantization* e *model pruning*.

Il primo, *model quantization*, che è una delle più popolari tecniche di ottimizzazione, prevede di ridurre la precisione di rappresentazione del nostro modello. Per esempio, possiamo convertire tutti i pesi da *float 32* bit a *int 8* bit (post-training quantization). Questo sicuramente andrà ad impattare sull'accuratezza del nostro modello ma restituirà un modello molto più leggero e reattivo. Oltre ai pesi, anche l'attivazione dei diversi "layer" può essere quantizzata raggiungendo velocità di inferenza ancora maggiori. Nel caso l'accuratezza del nostro modello fosse troppo degradata da questa procedura, possiamo pensare di riaddestrare il nostro modello con questa limitazione dei pesi (*quantization aware training*). La seconda tecnica di ottimizzazione prende il nome di *model pruning* e consiste nel rendere più efficiente il grafo del nostro modello.

Il sistema proposto, come già anticipato in precedenza, dopo aver addestrato la rete neurale converte il modello nel formato *tflite*. Tale formato è stato definito da TensorFlow nella sua versione Lite.

Solitamente in questa conversione vengono sostituite funzioni e modificati alcuni parametri, sia per avere migliori performance dal punto di vista computazionale sia per ridurre la dimensione del file binario stesso. Proprio questa seconda parte è fondamentale se si vuole fare in modo che il processo di machine learning avvenga sul dispositivo, senza dover quindi comunicare con un server remoto che faccia la parte di elaborazione. In questo modo vengono garantite, grazie al sistema embedded:

- Performance, sotto forma di basso tempo di latenza.
- Privacy, in quanto nessun dato raccolto lascia il dispositivo per avere informazioni.
- Basso consumo di batteria, in quanto l'utilizzo dell'hardware è ottimizzato e non sono richiesta connessioni di rete, che spesso risultano essere molto dispendiose in termini di consumo di energia.

Tecnica	Requisiti dei dati	Riduzione delle dimensioni	Precisione	Hardware supportato
Quantizzazione float16 post-training	Nessun dato	Fino a 50%	Perdita di precisione insignificante	CPU, GPU
Quantizzazione della gamma dinamica post-allenamento	Nessun dato	Fino al 75%	Perdita di precisione	CPU, GPU (Android)
Quantizzazione intera post-training	Campione rappresentativo senza etichetta	Fino al 75%	Minore perdita di precisione	CPU, GPU (Android), EdgeTPU, Hexagon DSP
Formazione consapevole della quantizzazione	Dati di allenamento etichettati	Fino al 75%	Minima perdita di precisione	CPU, GPU (Android), EdgeTPU, Hexagon DSP

Figura 6.8. Confronto diversi tipi di quantizzazione[30].

[30] https://www.tensorflow.org/lite/performance/model_optimization

Il discorso della quantizzazione è di particolare interesse in quanto può avere un notevole impatto sull'intera rete, andando a ridurre la latenza per fare inferenza ma spesso comportando anche una riduzione nell'accuratezza generale della risposta della rete. Nello specifico, quantizzare significa andare a ridurre la precisione dei numeri usati per descrivere i parametri del modello. Fare questa operazione quando si usa un dispositivo potente, come può essere un computer, ha tendenzialmente poco senso in quanto il guadagno in termini di prestazioni è minimo a fronte di un probabile peggioramento non trascurabile dei risultati. Questo ragionamento sul trade-off però è molto più difficile da fare per quanto riguarda il campo mobile, dove i vincoli di potenza e consumo sono molto più stringenti [18].

Utilizzando la quantizzazione, che può essere fatta sia sul risultato finale della rete sia sull'intero processo di allenamento, si possono ottenere delle performance nettamente superiori, non andando ad intaccare più di tanto le performance di inferenza. La precisione e la velocità durante l'addestramento sono le priorità principali. L'addestramento delle reti neurali viene fatto applicando molti piccoli stimoli ai pesi, e questi piccoli incrementi hanno tipicamente bisogno di precisione in virgola mobile per funzionare. Quasi tutta quella dimensione è occupata dai pesi delle connessioni neurali, dato che spesso ce ne sono molti milioni in un singolo modello.

Un'altra ragione per quantizzare è quella di ridurre le risorse computazionali necessarie per fare i calcoli di inferenza, eseguendoli interamente con ingressi e uscite a otto bit. Questo è molto più difficile poiché richiede cambiamenti ovunque si facciano i calcoli. Spostare i calcoli a otto bit aiuta ad eseguire i modelli più velocemente e a usare meno energia (il che è particolarmente importante sui dispositivi mobili).

Questo produrrà un nuovo modello che esegue le stesse operazioni dell'originale, ma con calcoli interni a otto bit, e anche tutti i pesi quantizzati. Se si guardano le dimensioni dei file, *model_conv.h5* e *model_conv_TFLite.tflite*, si vede che è le dimensioni del primo sono circa 3.93MB rispetto ai 340kB del secondo, permettendo di eseguire questo modello usando esattamente gli stessi input e output, ottenendo risultati che dovrebbero essere equivalenti [19].

6.2. Esportazione della CNN su piattaforma embedded

A seguire di tutto ciò, quindi, abbiamo a disposizione due tipologie di file *model*_conv.h5 e *model_conv_TFLite.tflite*, si rendono pertanto disponibili due tipologie diversificate di programmazione della scheda OpenMV, la programmazione mediante l'IDE OpenMV in modo "diretto" per così dire, caricando manualmente sulla scheda il dataset di test (*DS_Test_bmp64x64*) e il file *.h5*, e con una metodologia indiretta, ovvero mediante la programmazione del firmware su SD, procedura relativamente più complessa. La sostanziale differenza tra i due metodi sta nel fatto che con il secondo si riescono a ottenere dei tempi di latenza inferiori rispetto al primo metodo risultando quindi la scheda più performante in termini di ottimizzazione.

6.2.1. Modello per l'Ecosistema OpenMV

Al fine di potere effettivamente programmare la piattaforma OpenMV H7, si è reso necessario tenere in considerazione dei vincoli dati proprio dalla natura innovativa è recente della stessa in quanto, non è possibile programmare in una maniera "diretta" per così dire, la scheda a partire dal STM32Cube per passare direttamente al OpenMV IDE ma è necessario passare attraverso una fase intermedia di integrazione della rete neurale convoluzionale in ambiente OpenMV. Il progetto open-source OpenMV fornisce il codice sorgente per compilare il firmware OpenMV H7 con STM32Cube.AI abilitato. Il processo per utilizzare STM32Cube.AI con OpenMV è riportato per sommi capi nel paragrafo successivo.

In modo preliminare vi sono l'addestramento della rete neurale convoluzionale usando un framework di deep learning, GoogleColab in questo lavoro, a seguire vi è la conversione della rete addestrata (h5) utilizzando lo strumento STM32Cube come riportato nel paragrafo precedente. A seguire è riportato step by step il percorso che si dovrebbe seguire ai fini della programmazione effettiva della scheda:

1) Scaricare il codice sorgente del firmware OpenMV
2) Aggiungere i file generati al codice sorgente del *firmware*
3) Compilare con il *toolchain* GCC
4) *Flashare* la scheda utilizzando OpenMV IDE
5) Programmare la scheda con microPython ed eseguire l'inferenza

Si sarebbe dovuto compiere questo percorso mediante l'IDE di SMT32, ma non è stato possibile in quanto ad oggi (2021) l'OpenMV non fa ancora parte delle board previste dal Cube, o meglio il processore STM32H743II sì ma la board purtroppo ancora no, quindi non è stato possibile programmare direttamente il sistema embedded, al contrario di quanto invece sarebbe possibile fare con "similari" quale a titolo di esempio STM32L4 Cloud - JAM L4. Con il software CubeIDE è possibile riuscire a fare l'analyze per il processore in quanto noto, per vedere se il modello è "compatibile" e quindi programmabile con quel determinato processore (STM32H743II) ed idealmente potrebbe anche generare il firmware e poi scriverlo con il bootloader integrato nella OpenMV IDE ma in realtà conviene generare il firmware nel modo consigliato dalle guide ufficiali OpenMV [25]. Se generato dal CubeIDE accade questo:

- se la board che si sceglie è presente nella lista delle board compatibili, viene generato il firmware completo così risulta essere programmata "in locale" senza interpretazione.
- se la board non è presente ma il processore sì (il nostro caso) e si va a generare il progetto, e quindi il firmware per il dato processore, allora il firmware contiene l'inizializzazione del micro ed il modello ma non contiene l'inizializzazione delle periferiche della board. Nel nostro caso mancherebbe la gestione di USB e CAMERA, in sostanza non saremmo in grado si programmare la board.

Risulta quindi conveniente usare la procedura suggerita da ST/OpenMV che dice di partire dal firmware della OpenMV (che ha già tutte le inizializzazioni necessarie per la board) e ricompilarlo con una serie di passaggi che fanno in modo che all'interno venga incorporato il modello, le procedure guidate sono disponibili al link presente [25] che rimanda direttamente al sito ufficiale del ST/OpenMV. L'inizio di tutta la procedura è sempre a partire dal Cube, si usare il firmware dell'OpenMV e si fa una specie di "merge" per ricompilare tutto e ottenere il nuovo firmware finale.

6.2.2. Modello TFlite per OpenMV IDE

Viene ora riportata, la seconda metodologia possibile per affrontare il problema di implementazione all'interno della scheda OpenMV del firmware. La tecnica riportata segue una via diretta, ovvero, avendo a disposizione il file *tflite*, per tutte le motivazioni spiegate in precedenza, è possibile mediante l'utilizzo dell'IDE, programmare la scheda elettronica. Come modello per poter caricare il firmware è stato utilizzato il codice scritto in microPython disponibile in [28].

Il codice in questione è basato, dopo una opportuna fase preliminare di caricamento delle immagini del test set opportunatamente ottimizzato (*DS_Test_bmp64x64*) e della rete *model_conv.tflite* (prodotta nel TensorFlow) da parte dell'utente all'interno della scheda SD del dispositivo (drag & drop) seguita da un'adeguata fase di inizializzazione della scheda in questione. Segue un ciclo secondo cui vengono prelevate le immagini dalla Cam a bordo e vengono processate all'interno della rete neurale convoluzionale che è stata oggetto delle precedenti capitoli [codice OpenMV IDE 29].

Ovviamente con questa tipologia di caricamento del firmware vengono barattati velocità e latenza in cambio di una relativa facilità di preparazione del sistema. Sono riportati infatti a questo scopo delle tabelle nelle quali sono state andate a riepilogare le principali caratteristiche confrontando i modelli *h5* e *tflite* e in particolare con questa strategia di implementazione su scheda SD.

6.2.3 Sperimentazione empirica

Ai fini della verifica della corretta programmazione della scheda si è proceduto ad una fase di sperimentazione con delle foglie e l'uso della scheda. Le foglie sono state prese in modo da risultare più simili possibili, per quanto possibile stante il periodo autunnale, a quelle presenti nel dataset. Il set di sperimentazione e il materiale con il quale sono state verificate le foglie è riportato nella foto posta di seguito.

Figura 6.9. Set di Sperimentazione.

Da una analisi sommaria, la scheda riesce a riconoscere relativamente bene foglie tra loro molto dissimili quali uva e fragola, mentre ad esempio per foglie che tra di loro presentano similarità di forma geometriche e colore, tende a sbagliare e a confonderle.

Si tenga presente inoltre che, il dataset PlantVillage adoperato nella rete è stato creato per il riconoscimento di malattie delle piante, lo scopo della sommaria analisi effettuata è stata quella di verificare praticamente che la scheda fosse in grado di riconoscere le differenti tipologie di piante, purtroppo non è stato possibile procurarsi foglie con le varie malattie. Ovviamente tale verifica non ci ha fornito informazioni e dati in merito la bontà della rete ma solo un comportamento generale. Le foglie delle piante utilizzate per la sperimentazione appartengono alle seguenti specie vegetali: foglia di uva, foglia di fragola, foglia di melo, foglia di ciliegio. Durante la fase di sperimentazione, ho potuto constatare con una analisi più approfondita in sede di test del dataset, come le immagini di foglie presenti siano costituite, nel caso di alcune specie vegetali quali il pomodoro e le patate da foglie cosiddette composte**.** Le foglie composte sono costituite da foglioline disposte a destra e a sinistra del rachide come in una penna. Infatti, nelle immagini è presente solo una delle singole foglioline e non l'intero ramoscello.

Le prove sono state effettuate con la luce mattutina, indicativamente alle ore 11.00, in modo tale da evitare l'utilizzo di fonti di luci artificiali o quanto meno disomogenee e che potessero creare qualche ombra nell'immagine. Naturalmente le prove effettuate peccano di condizioni ottime di illuminazione e così via, però sono state comunque una fonte di informazioni sulla metodologia di utilizzo della scheda in termini pragmatici. Probabilmente delle prove effettuate ha inciso anche la problematica non indifferente relativa alla questione "materiale" da far riconoscere alla scheda, in primis in quanto non è stato possibile trovare

foglie con la grande varietà di malattie disponibili come quelle nel lavoro, non da meno anche la questione relativa alla varietà delle specie, permettendo quindi solo di poter analizzare uno spicchio di tutte le effettive potenzialità di rilevamento della rete. La stagione autunnale non ha influito poco sulla varietà di foglie "sane" e quindi verdi ed eventualmente malate, disponibili in natura.

In una prima fase si è tentato ad una verifica della CNN non con foglie vere, ma con immagini prese dal cellulare e dal computer con risultati relativamente scarsi, in sostanza le foglie verdi venivano classificate come "mais", dovuta al fatto probabilmente delle caratteristiche foglia molto larga della specie in questione che traeva in inganno la rete vedendo un'immagine riccamente verde. Le foglie invece che presentavano malattie e che quindi portavano le immagini ovviamente a tendere verso un colore arancio o giallo tendevano tutte ad essere classificate nella malattia dell'arancio, malattia che in effetti presenta delle macchie largamente di tonalità accese. Propendendo quindi ad un'analisi sul campo mediante l'utilizzo di foglie vere e non "virtuali".

È possibile, inoltre, reperire un video della sperimentazione effettuata [27].

6.2.4. Sperimentazione con Testing Set

Per poter avere dei dati concreti, a seguito della fase di sperimentazione empirica come verifica del funzionamento della piattaforma, ha avuto luogo un controllo mediante l'utilizzo del testing set precedentemente elaborato (*DS_Test_bmp64x64*). L'OpenMV è stata sottoposta ad una verifica della capacità di predire correttamente le immagini mediante l'invio di immagini proveniente dall'SD adoperando la matrice di confusione come strumento di riscontro, la quale ci ha fornito un grado di accuratezza del 90.6% [29]. Tale valore è simile a quello presente all'interno della fase di costruzione della rete nel Colab 94.47% in *.h5* e 94.05% in *tflite*, confermando quindi la capacità di predizione della rete in sede di test. Si è inoltre provato un test anche con una diminuzione della *size* delle immagini da 64x64 a 32x32, al fine di diminuire il consumo di memoria causato dalle dimensioni del dataset.

Le tabelle di seguito riportate rappresentano dei valori riepilogativi ottenuti dal lavoro, in particolare consumo di memoria, numero dei parametri, accuratezza dei modelli *h5* e *tflite* e tempo di inferenza dei modelli.

Model type	Input size	Memory cost [KB]	Parameters number	Test Accuracy [%]	Inference time [msec]	MACC	ROM bytes	RAM bytes
h5	64x64	4026.688	339.015	94.47	16.88	5.184.112	1.356.060	64.896
	32x32	954.688	76.871	90.62	19.24	1.160.960	307.484	20.160
tflite	64x64	341.195	339.015	94.05	3.45	5.100.972	341.004	17.260
	32x32	85.195	76.871	90.06	9.67	1.140.444	78.860	8.960

Tabella 1. Performance del modello trained sul PC

Platform	Model type	Input size	Test accuracy [%]	Inference time [msec/img]	FPS
Model saved on SD card	tflite	64x64	90.60	68.50	14.59
		32x32	77.64	21.47	46.57

Tabella 2. Validation del modello su OpenMV Cam

CAPITOLO 7
Conclusioni

L'obbiettivo posto all'inizio di questo elaborato era quello della realizzazione e dell'implementazione di una CNN in una piattaforma embedded quale la OpenMV cam, le considerazioni conclusive di questo progetto vogliono fare sintesi di ciò che si è potuto trarre da un punto di vista teorico e pragmatico a seguito dello studio, dell'implementazione della rete e dall'operazione finale di verifica mediante l'utilizzo della piattaforma.

L'utilizzo di codici di programmazione quali Python e applicativi quale TensorFlow si è rivelata vincente in quanto la rete è ricca di documentazione che hanno facilitato la realizzazione del progetto. Per quanto concerne la scelta del dataset, si è optato affinché contenesse un numero di immagini per classi quanto meno omogeno (anche se sono presenti delle differenze) al fine di mitigare per quanto possibile pericolosi sovradimensionamenti di talune classi rispetto altre. Infatti, durante il percorso sono stati provati due dataset diversi, in termini di grandezza ed espansione delle singole classi, con i quali si è potuto percepire la differenza della CNN nel rilevare le immagini. Ovviamente la scelta di utilizzo del dataset è ricaduta sul secondo, con un numero di immagini per classe meglio distribuite. La piattaforma embedded utilizzata, come già ampiamente illustrato, è dotata di un costo ridotto e di una grande potenza di calcolo, in particolare nello sviluppo di applicazioni di visione artificiale, e questo elaborato ne è la prova tangibile, le dimensioni della scheda sono relativamente ridotte permettendo quindi anche un'implementazione a bordo di un sistema trasportabile con sé, basterebbe salvare lo script *.py* su SD, collegare la piattaforma ad un power bank ed un display LCD per stampare la risposta e il sistema sarebbe autonomo.

Con le operazioni di verifica finali quindi si è potuto appurare come le immagini in termini di riconoscimento da parte della piattaforma siano fortemente influenzate da fenomeni quali la presenza di luce, angolo di rilevamento e così via, per ovviare a tali disturbi si è cercato di effettuare le prove su uno sfondo bianco neutrale e mantenendo la scheda con un sostegno in modo tale da garantire sempre la medesima distanza scheda-foglia.

In ultimo ha inciso anche la quantità delle foto presenti all'interno delle classi del dataset in quanto, nonostante la scelta iniziale di un dataset amalgamato nel numero di immagini per classe mediante *augmentation* [2], è risultato comunque essere sbilanciate, basti pensare ad esempio che classi come le piante di patate sono presenti in quantità decisamente minori (2 classi) rispetto al numero di classi di altre varietà come le foglie dei pomodori (9 varietà di malattie diverse) o ad addirittura singole malattie di piante quali dell'arancio (1371 foto). Molto probabilmente ciò ha inciso negativamente sulle caratteristiche della rete di poter rilevare talune specie di piante rispetto ad altre sovra rappresentate o sottorappresentate. Una strada di ottimizzazione perseguibile potrebbe essere quella di cambiare la struttura del modello e verificare come cambi l'accuratezza e la risposta del sistema.

Il focus del lavoro è stato orientato al riconoscimento di piante, un'idea di sviluppo successivo potrebbe essere quella di implementare il sistema su un dataset differente al fine di confermare la versatilità dell'applicazione, avendo la struttura del modello e degli applicativi OpenMV IDE risulta essere relativamente facile cambiare dataset e quindi lo scopo finale della piattaforma. Tutti i codici e gli applicativi utilizzati sono presenti in appendice.

Come visto nel corso della lettura dell'elaborato, si può notare che la stessa si sia presentata come una sorta di descrizione a modo di campionamento delle varie fasi di sviluppo, andando a presentare il progetto sia come una Tesi di Laurea sia come un progetto che può essere replicato e ottimizzato. Le fasi di

sperimentazioni finali, in particolare di testing, hanno permesso di avere numeri alla mano con cui poter valutare la bontà della CNN.

Lo svolgimento del lavoro mi ha permesso di estendere le conoscenze nei linguaggi di programmazione a me non noti quale il Python, di software quale TensorFlow, e ad un uso pragmatico di uno strumento elettronico quale la scheda. La parte di ricerca e sviluppo del lavoro è stata molto interessante, in quanto è stata un filo conduttore durante tutto il percorso. Personalmente penso che la presente esperienza mi abbia permesso di estendere i miei interessi in settori che probabilmente non avrei affrontato, avrei desiderato sinceramente che la piattaforma utilizzata funzionasse in un modo più deciso, comunque mi ritengo decisamente soddisfatto del lavoro.

Volendo tracciare un bilancio complessivo di questo percorso posso definirmi molto appagato del lavoro svolto e sono certo che le competenze sia tecniche che trasversali di cui ho fatto prezioso bagaglio, mi torneranno sicuramente utili nel futuro.

Bibliografia

[1] L. Falaschetti et al., "A Low-Cost, Low-Power and Real-Time Image Detector for Grape Leaf Esca Disease Based on a Compressed CNN," in IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 11, no. 3, pp. 468-481, Sept. 2021, doi: 10.1109/JETCAS.2021.3098454.

[2] M. Alessandrini, R. C. F. Rivera, L. Falaschetti, D. Pau, V. Tomaselli, and C. Turchetti, "A grapevine leaves dataset for early detection and classification of esca disease in vineyards through machine learning," Data Brief, vol. 35, Apr. 2021, Art. no. 106809.

Sitografia citata

[3] https://www.agroscope.admin.ch/agroscope/it/home/temi/produzione-vegetale/protezione-piante/fitopatologia.html#:~:text=La%20fitopatologia%20%C3%A8%20lo%20studio,termini%20di%20qualit%C3%A0%20e%20resa

[4] https://www.oracle.com/it/data-science/machine-learning/what-is-machine-learning/

[5] http://www.crescenziogallo.it/unifg/dottorato-medicina-traslazionale-2018/Reti%20neurali%20artificiali%20-%20Concetti%20base.pdf

[6] *materiale fornito dalla Professoressa*

[7] https://machinelearningmastery.com/padding-and-stride-for-convolutional-neural-networks/

[8] https://www.kaggle.com/fazilbtopal/a-detailed-cnn-with-tensorflow

[9] *materiale fornito dalla Professoressa*

[10] https://www.kaggle.com/fazilbtopal/a-detailed-cnn-with-tensorflow

[11] *materiale fornito dalla Professoressa*

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

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

[14] https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-to-classify-photos-of-dogs-and-cats/

[15] https://www.domsoria.com/2020/02/cosa-significa-overfitting-e-underfitting/

[16] https://www.developersmaggioli.it/blog/classificazione-con-rete-neurale

[17] https://colab.research.google.com/notebooks/intro.ipynb.

[18] https://vittoriomazzia.com/tensorflow-lite/

[19] https://petewarden.com/2016/05/03/how-to-quantize-neural-networks-with-tensorflow/

[20] Immagini dataset: https://www.kaggle.com/emmarex/plantdisease

[21] Sito OpenMv: https://openmv.io/

[22] Sito TensorFlow: https://www.tensorflow.org/

[23] Sito GoogleColab: https://colab.research.google.com/

[24] Sito SMT32Cube: https://www.st.com/en/development-tools/stm32cubemx.html

[25] Guida Firmware : https://wiki.st.com/stm32mcu/wiki/AI:How_to_add_AI_model_to_OpenMV_ecosystem

[26] Link GDrive codice TF sviluppato: https://colab.research.google.com/drive/1GAaftzaoDSacsSunEKvYfVOLAPQ7u6O8?usp=sharing

[27] Link GDrive video sperimentazione: https://drive.google.com/file/d/1fFXtjmG8avk4wJk6Xxy94p8sUQ2LDrsZ/view?usp=sharing

[28] Link GDrive codici .h5 + .tflite:

https://drive.google.com/drive/folders/1fKntjZ1EfII6y2TlM52SndVCksV0hVa4?usp=sharing

[29] Link GDrive codici ambiente OpenMV IDE:

https://drive.google.com/drive/folders/1g-JXupxnN4v3sW9Q9CIiqA56LD4fGgOF?usp=sharing