UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI INGEGNERIA

Corso di Laurea triennale in Ingegneria Elettronica

Implementazione di una CNN per la classificazione di immagini appartenenti al dataset COIL-20 su piattaforma embedded OpenMV Cam

Implementation of a CNN for the classification of the COIL-20 image dataset on the OpenMV Cam embedded platform

Relatore: Chiar.mo

Prof. Claudio Turchetti

Tesi di Laurea di:

De Santis Matteo

Correlatore:

Prof.ssa Laura Falaschetti

A.A. **2019/ 2020**

Indice

Introduzione al Machine Learning

Negli ultimi anni si sente sempre più parlare di intelligenza artificiale, di computer in grado di elaborare informazioni senza un aiuto esterno, in grado di trarre conclusioni molto simili a quelle che trarrebbe un umano. Si pensi per esempio alla guida autonoma di un'auto, che grazie a un sistema di sensori e videocamere sa comportarsi su strada come un qualsiasi automobilista responsabile, oppure a un cellulare che riesce a riconoscere il viso di un uomo rispetto a un altro. Allora viene spontaneo chiedersi come sia possibile che un computer riesca a "ragionare" cogliendo le differenze e le caratteristiche di ciò che sta valutando, ovviamente non si tratta di una semplice elaborazione composta di soli calcoli, perché questi non basterebbero a un computer per riuscire a comprendere e fare una scelta giusta in base alla situazione. Gli esseri umani, come qualsiasi altro essere vivente, sin da piccoli cominciano a cogliere le innumerevoli informazioni portate dal mondo esterno per poter sviluppare un comportamento tale da adattarsi all'universo circostante, questo perché il cervello, in specie quello umano, presenta complesse organizzazioni di cellule nervose dette reti neurali, con compiti di riconoscimento delle configurazioni assunte dall'ambiente esterno, memorizzazione e reazione agli stimoli provenienti dallo stesso. Questa caratteristica permette agli esseri viventi di **imparare** ed è proprio imparando da ciò che si ha intorno, dai propri errori e da quelli degli altri che l'uomo sin dall'antichità è potuto evolvere e migliorare sempre più. Fino a qualche decennio fa, quando si parlava di intelligenza artificiale, semplicemente ci si riferiva a un sistema che in base a come era stato programmato, rispondeva a un certo set di input con un set di output. Però il sistema in questione non era realmente in grado di apprendere e capire, ma semplicemente elaborava gli input dati e forniva degli output in base ai calcoli che gli erano stati imposti di fare. Si pensi ancora al riconoscimento facciale di un cellulare, come fa a distinguere un essere umano da un altro? Evidentemente prima di vedere il viso da identificare ne ha visti molti altri e grazie a questi ha imparato a distinguere i tratti somatici che descrivono un essere umano. Questo perché la tecnologia ha portato a sviluppare algoritmi che implementano reti neurali artificiali, permettendo a un computer di avvicinarsi al metodo di ragionamento degli esseri umani. Grazie alla crescente richiesta di automatizzazione e alla necessità di classificare enormi quantità di dati, i campi di utilizzo di questo tipo di algoritmi sono dei più svariati**:** automobilistico, sanitario, telefonia, agri-tech e molti altri.

L'obiettivo di questa **tesi** è utilizzare una rete neurale, per svolgere un compito di classificazione su delle immagini. Nello specifico si andrà ad implementare una rete neurale convoluzionale che, utilizzando il dataset COIL-20, sarà in grado di classificare le 20 tipologie di oggetti appartenenti allo stesso. A questo punto il modello verrà caricato su board OPENMV CAM H7 PLUS e sarà in grado di effettuare il riconoscimento in tempo reale. Per soddisfare le specifiche progettuali sono stati svolti i seguenti passaggi:

1) Preparazione del dataset COIL-20 sull'ambiente di sviluppo colab
2) Implementazione della rete neurale convoluzionale sull'ambiente di sviluppo colab
3) Test sul modello sviluppato
4) Scrittura codice su openMV ide per uso in tempo reale del modello su board
5) Test del modello in tempo reale su board

Un'analisi accurata del progetto necessita un'introduttiva trattazione teorica, in modo tale da permettere una buona comprensione degli algoritmi e del codice scritto in Python. Saranno quindi descritte in primo luogo le reti neurali per poi passare a quelle convoluzionali, analizzando nel dettaglio il funzionamento di entrambe e degli strumenti necessari per poterle sviluppare.

Capitolo I: RETI NEURALI

Una rete neurale artificiale è un modello matematico/informatico di calcolo basato sulle reti neurali biologiche. Tale modello è costituito da un gruppo di interconnessioni di informazioni costituite da neuroni artificiali e processi che utilizzano un approccio di connessionismo di calcolo. Nella maggior parte dei casi una rete neurale artificiale è un sistema adattivo che cambia la propria struttura in base a informazioni esterne o interne che scorrono attraverso la rete stessa durante la fase di apprendimento.

In termini pratici le reti neurali sono strutture non-lineari di dati statistici organizzate come strumenti di modellazione.

Il concetto di rete neurale si pone perché una funzione f(x) è definita come una composizione di altre funzioni G(x), che possono a loro volta essere ulteriormente definite come composizione di altre funzioni. Questo può essere comodamente rappresentato come una struttura di reti, con le frecce raffiguranti le dipendenze tra variabili. Una rappresentazione ampiamente utilizzata è la somma ponderata non lineare, dove

$$f(x) = k\left(\sum_i w_i g_i(x)\right)$$

Dove k è una funzione predefinita. Sarà conveniente per le seguenti far riferimento ad un insieme di funzioni come un vettore $g = (g_{\{1\}}, g_{\{2\}}, \ldots, g_{\{n\}})$

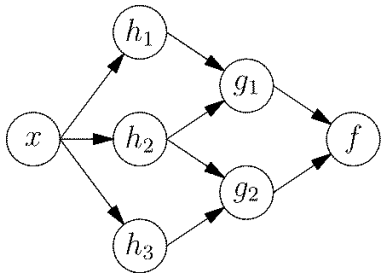
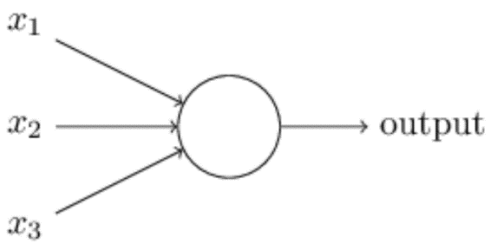


Diagramma di dipendenza di una rete neurale

La figura esemplifica una decomposizione della funzione f, con dipendenze tra le variabili indicate dalle frecce. Queste possono essere interpretate in due modi: Il primo punto di vista è la vista funzionale: l'ingresso x è trasformato in un vettore a 3-dimensioni, che viene poi trasformato in un vettore bi-dimensionale g, che è poi finalmente trasformato in f. Questo punto di vista è più comunemente riscontrato nel contesto dell'ottimizzazione. Il secondo punto di vista è la vista probabilistica: la variabile casuale F=f(G) dipende dalla variabile casuale G=g(H), che dipende da H=h(X), che dipende a sua volta dalla variabile casuale X.

Il primo passo per comprendere il funzionamento di una rete neurale è la comprensione del comportamento del singolo neurone. Verrà presa come primo esempio una rete binaria. In questo caso infatti il neurone può avere diversi ingressi binari e presenta un'uscita di tipo 0 o 1, ogni ingresso x è moltiplicato a un peso w e l'uscita sarà determinata quindi dalla somma pesata degli ingressi $\sum_j w_j x_j$ che in base al suo valore, maggiore o minore di una certa soglia darà in uscita rispettivamente 0 o 1.



$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

Definendo il bias b=-threshold, nel caso di un unico ingresso si avrà:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Quindi il peso indica quanto l'input a cui è moltiplicato sia importante nella decisione del neurone, mentre il bias indica quanto facilmente si attiva il neurone.

Si immagini una situazione in cui il singolo neurone (in questo caso come se fosse un ragazzo) deve fare una scelta binaria di questo tipo: oggi è sabato e si vuole uscire in piazza con gli amici, ma per prendere questa decisione bisogna prima considerare tre fattori

- Se gli amici escono
- Il clima
- Il traffico per arrivare

Un esempio di questo tipo mette in evidenza il fatto che la decisione che si farà dipende da condizioni non ugualmente importanti, ossia alcune hanno più peso rispetto ad altre. Se piove e c'è anche il traffico, ma tutti gli amici escono, allora con tanto di ombrello e pazienza la decisione sarà quella di uscire comunque.

In casi più generici e più complessi del precedente, dove la decisione della rete non è binaria e i dati in ingresso sono decine di migliaia o più, non si parlerà in termini di attivo o spento per il neurone, ma di "quanto" è attivo il neurone, quindi le funzioni di attivazione daranno in uscita numeri reali che più sono grandi più il neurone in questione sarà attivo.

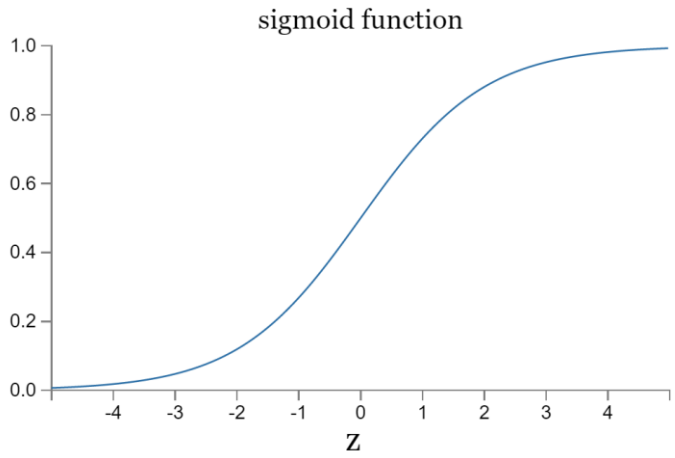
Prima tra tutte è la sigmoid function:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}$$

Sostituendo z

$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}$$

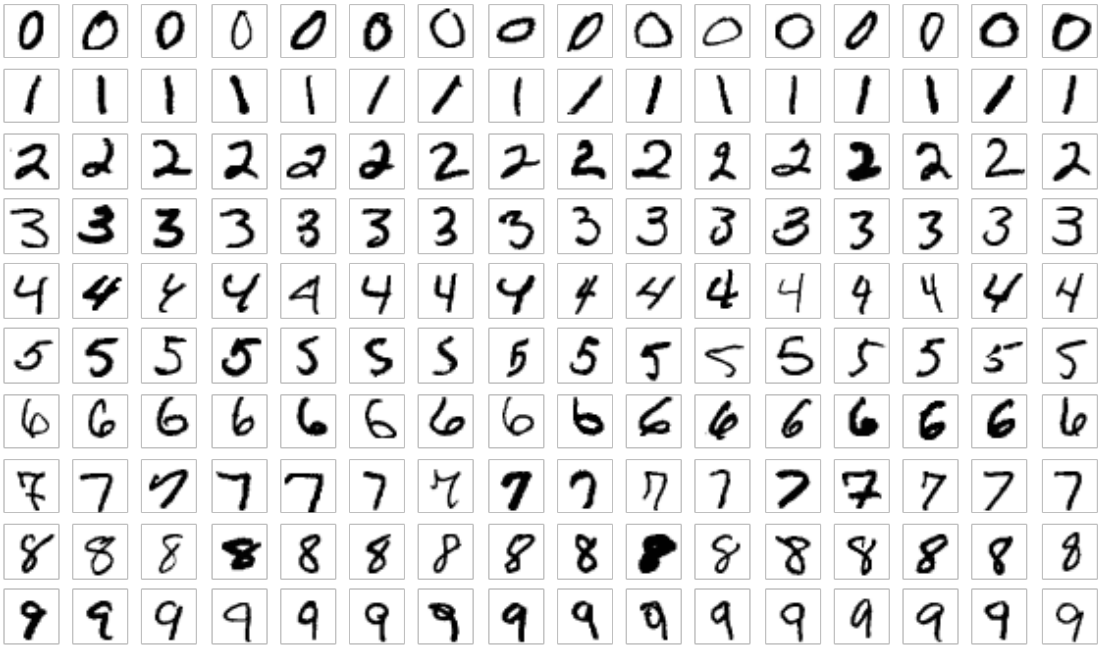
La sigmoid restituisce in uscita un numero razionale compreso tra 0 e 1, come si può vedere in figura:



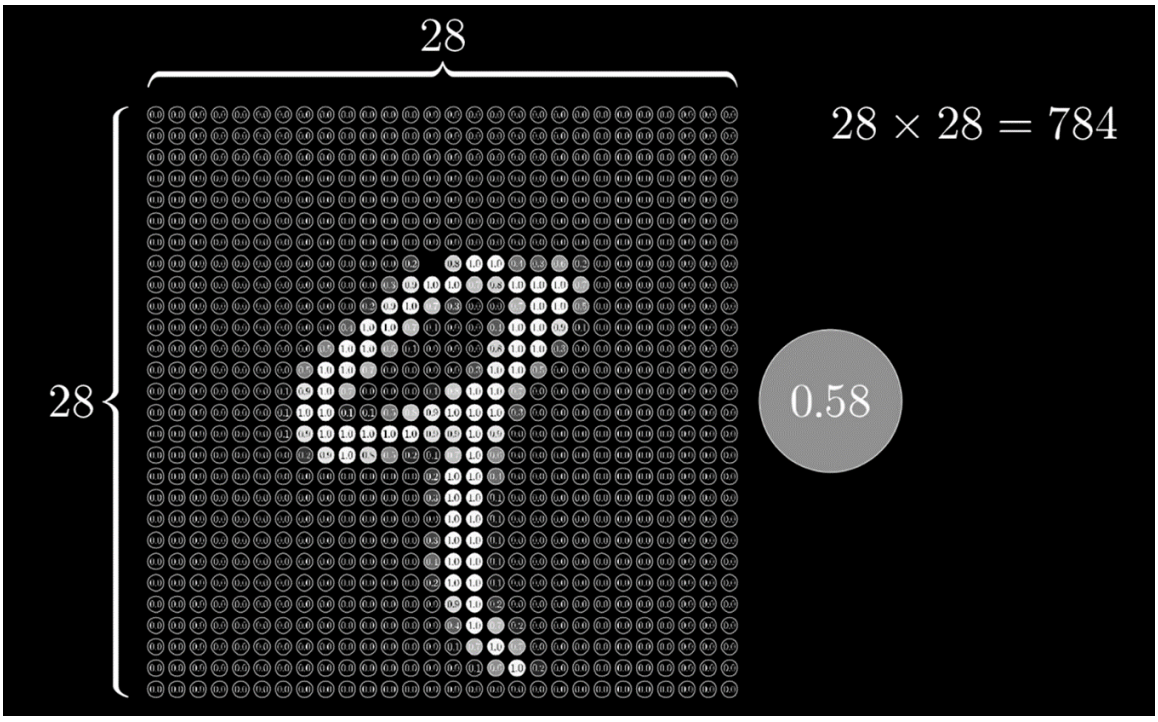
Ovviamente esistono molte altre funzioni di attivazioni anche più performanti come la RELU o la SOFTMAX di cui si parlerà in seguito.

A questo punto è necessario un esempio più concreto per comprendere meglio il funzionamento di una semplice rete neurale. Per l'esempio in questione verrà utilizzato il database MNIST.

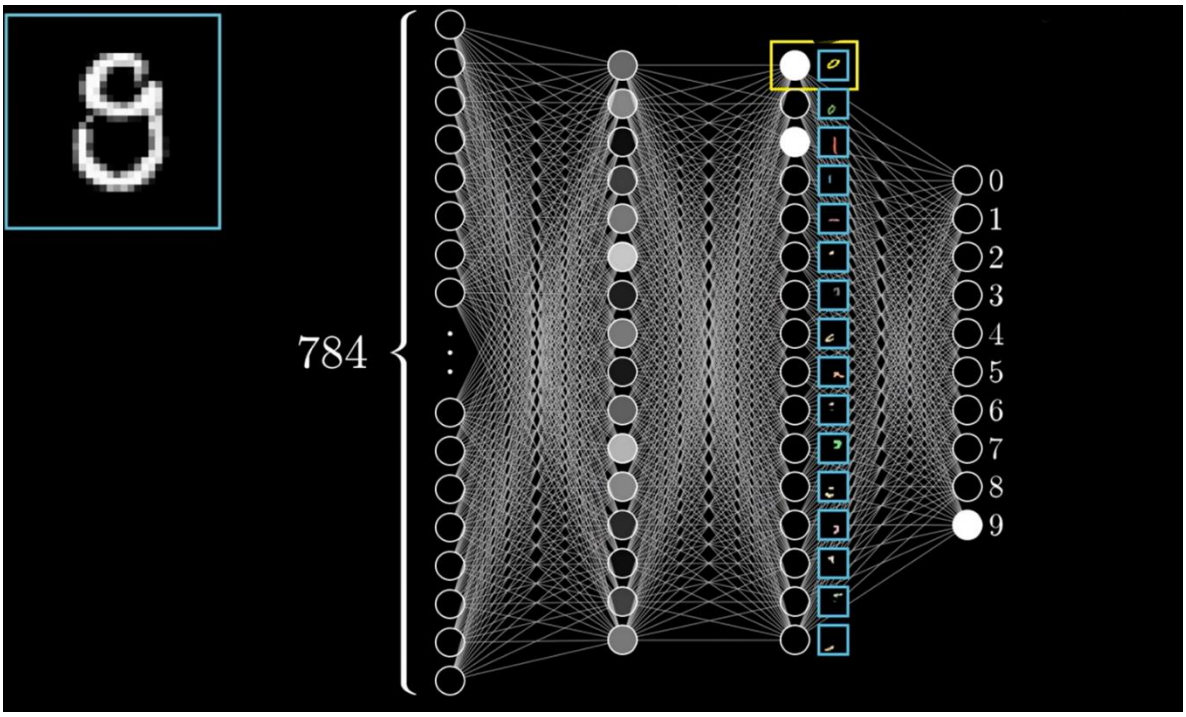
Il database MNIST (modified National Institute of Standards and Technology database) è una vasta base di dati di cifre scritte a mano che è comunemente impiegata come insieme di addestramento in vari sistemi per l'elaborazione delle immagini. La base di dati MNIST contiene 60,000 immagini di addestramento e 10,000 immagini di test; metà dell'insieme di addestramento e metà dell'insieme di test sono state prelevate dall'insieme di addestramento del NIST, mentre le altre metà sono state ottenute dall'insieme di test del NIST stesso.



La rete riceverà in ingresso 10,000 immagini, ognuna di queste raffigurante un numero da 0 a 9 scritto a mano. Essendo un'immagine un insieme di pixel, ognuno con il suo valore normalizzato a 1 (in questo caso un unico valore per ogni pixel perché sono immagini in bianco e nero e non RGB) possiamo interpretarla come una matrice, nell'esempio analizzato una matrice quadrata 28x28.



Per l'elaborazione della rete sarà più comodo rappresentare la matrice come un array monodimensionale lungo 784 "pixel".



La rete è composta dal layer iniziale dove sono contenuti tutti i valori dei pixel, poi gli **hidden layer** (il numero di essi e dei neuroni di tali layer viene determinato in modo euristico, in modo tale da ottenere il miglior risultato possibile) elaborano l'informazione prima dell'uscita.

L'approccio che consente alla rete di elaborare l'informazione come spiegato prima, è quello di moltiplicare un peso (weight) a ogni input del singolo neurone, tutti questi prodotti vengono poi sommati tra di loro più un bias, tale espressione viene elaborata dalla funzione di attivazione SIGMOID che andrà a indicare se il neurone in questione è inattivo o quanto attivo. Questo procedimento viene ripetuto per ogni neurone di ogni layer.

Si supponga che il primo neurone nello strato nascosto rilevi la presenza di una caratteristica fondamentale per la classificazione del numero contenuto nell'immagine, ad esempio il cerchio che identifica il numero "9". Avendo rilevato tale caratteristica esso si attiverà e di conseguenza il suo contenuto sarà elevato. Questa attivazione è stata possibile grazie a quei pesi che, riferendosi a quella specifica parte dell'immagine, hanno assunto durante l'allenamento della rete un'importanza elevata, aumentando in valore assoluto. Infatti, data l'importanza assunta da questi parametri, il risultato della sommatoria che genera il contenuto del neurone darà un valore alto.

Supponendo che la rete neurale funzioni in questo modo, è possibile dare una spiegazione plausibile del motivo per cui è meglio avere 10 uscite dalla rete, invece di 4 (4 bit che possono rappresentare un intero da 0 a 15). Se si avessero 4 uscite, il primo neurone di uscita che rappresenta il bit più significativo a quale porzione di cifra deve essere abbinato? Non esiste un modo semplice per correlare quel bit più significativo a forme semplici come quelle mostrate sopra. È difficile immaginare che ci sia una buona ragione per cui le forme dei componenti della cifra saranno strettamente correlate al bit più significativo nell'output. Per questo motivo quando si hanno decine di migliaia di ingressi e diverse uscite può essere complicato e inefficiente utilizzare una rete binaria.

1.1 BACKGROUND PROPAGATION

La retro-propagazione è l'essenza dell'addestramento della rete neurale. È la pratica di mettere a punto i pesi di una rete neurale in base al tasso di errore (cioè perdita) ottenuto dalla LOSS FUNCTION nell'iterazione precedente. Una corretta regolazione dei pesi garantisce tassi di errore inferiori, rendendo il modello affidabile aumentandone la generalizzazione. In ottimizzazione matematica e nella teoria della decisione, una funzione di costo (LOSS FUNCTION o COST FUNCTION) è una funzione che mappa un evento, o valori di una o più variabili, su un numero reale, intuitivamente rappresenta un "costo" associato all'evento. Un problema di ottimizzazione cerca di minimizzare una funzione di costo.

La COST/LOSS FUNCTION quadratica è molto comune e semplice da spiegare, è data dal quadrato della differenza tra l'attivazione del neurone col valore 1 riferito al risultato reale e la label corrispondente.

$$C(w,b) \equiv \frac{1}{2n}\sum_x \|y(x)-a\|^2$$

y(x) rappresenta l'output desiderato, quindi le label corrispondenti ai dati analizzati. y(x) è quindi un vettore la cui dimensione dipende dal numero di neuroni nell'ultimo layer (dal numero di classi del database). Nell'esempio analizzato è un vettore lungo 10, si faccia finta che l'immagine analizzata sia un 6, allora l'uscita desiderata sarà $y(x) = (0,0,0,0,0,0,1,0,0,0)^T$

Nel caso in esempio si hanno 10 label corrispondenti alle classi dei numeri che vanno da 0 a 9: la classe di tutti gli zeri, la classe degli uni ecc…

Invece w denota la raccolta di tutti i pesi nella rete, b tutti i bias, n è il numero totale di input di addestramento, a è il vettore degli output reali (activation dei neuroni nell'ultimo layer) dalla rete quando x è input e la somma è su tutti gli input di addestramento, x.

L'obiettivo è quello di modificare pesi e bias per aumentare il numero di immagini riconosciute correttamente. Lavorando direttamente sul numero dei risultati corretti, senza utilizzare una cost function, apportare piccole modifiche ai pesi e ai bias non causerà alcun cambiamento nel numero di immagini di allenamento classificate correttamente. Il problema è che il numero di immagini classificate correttamente non è una funzione regolare dei pesi e dei bias nella rete, ciò rende difficile capire come modificare i pesi e i bias per ottenere prestazioni migliori. Utilizzando una funzione di costo regolare come il costo quadratico risulta facile capire come apportare piccole modifiche ai pesi e ai bias in modo da ottenere un miglioramento del costo. Ecco perché ci si concentra prima sulla riduzione al minimo del costo quadratico e solo dopo si esaminerà l'accuratezza della classificazione.

1.1.1 Come minimizzare la cost function

L'obiettivo è quindi quello di trovare la combinazione di w e b tali per cui si ha il valore minimo della funzione di costo. Come ben noto, dovendo trovare punti di massimo o minimo di una funzione, lo strumento matematico più adatto è quello della derivata, per semplificare si sostituiscono le variabili w_1,b_1, w_2,b_2... con v_1,v_2… Ovviamente avendo una funzione di più variabili è necessario l'utilizzo del gradiente.

Per semplicità si analizza prima il caso bidimensionale, andando a compiere piccole variazioni a v_1,v_2 si otterrà una piccola variazione in C o meglio

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

Dove appunto il vettore gradiente è:

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

E il vettore dei cambiamenti:

$$\Delta v \equiv (\Delta v_1, \Delta v_2)^T$$

Possiamo quindi scrivere che:

$$\Delta C \approx \nabla C \cdot \Delta v$$

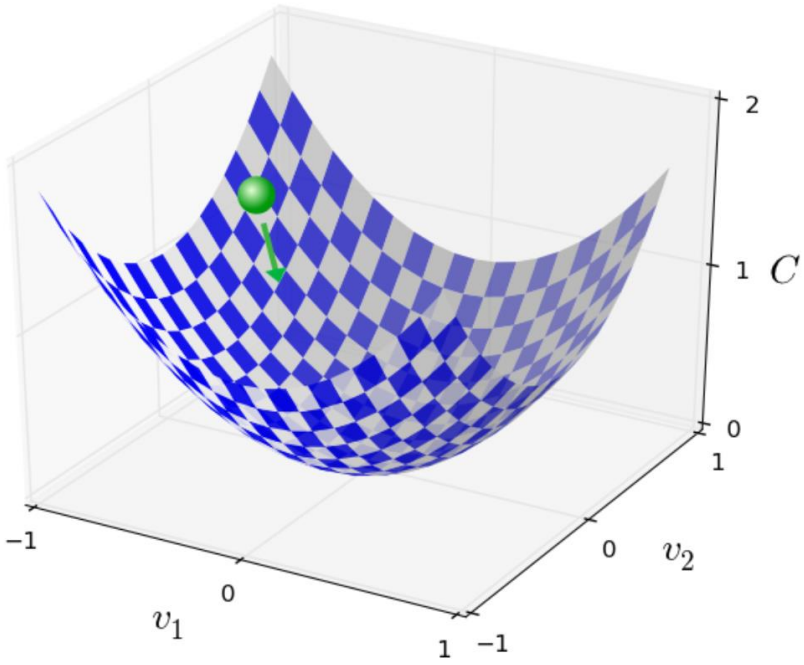
Come accennato precedentemente, derivare una funzione può servire sia per trovare punti di massimo che di minimo. Infatti se la derivata della funzione in una variabile, in un certo intervallo è positiva, allora la funzione cresce in tale intervallo all'aumentare della variabile. Viceversa se la derivata è negativa in tale intervallo, la funzione decresce all'aumentare della variabile. La situazione preferita è ovviamente la seconda in quanto bisogna trovare il punto di minimo. Occorre scegliere quindi un Δv opportuno in modo tale che la pendenza della variazione della cost function sia sempre negativa.

Scegliendo:

$$\Delta v = -\eta \nabla C$$

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$$

Essendo $||\nabla C||>0$ sempre vera, si ha la condizione per cui è possibile determinare il minimo della cost function C.



Il grafico in figura mostra idealmente come il gradiente descrive la discesa della cost function.

La posizione iniziale della funzione C e quindi dei valori di pesi e bias è casuale. Questa posizione viene via via aggiornata grazie alla background propagation.

Ovviamente generalizzando al caso in cui si hanno più variabili si ottiene:

$$\Delta v = (\Delta v_1, \ldots, \Delta v_m)^T$$

$$\Delta C \approx \nabla C \cdot \Delta v$$

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m} \right)^T$$

Le considerazioni fatte considerando solo due variabili, valgono anche nel caso in cui si hanno più variabili. Il caso di sole due variabili è stato utile per poter graficare in tre dimensioni la discesa del gradiente.

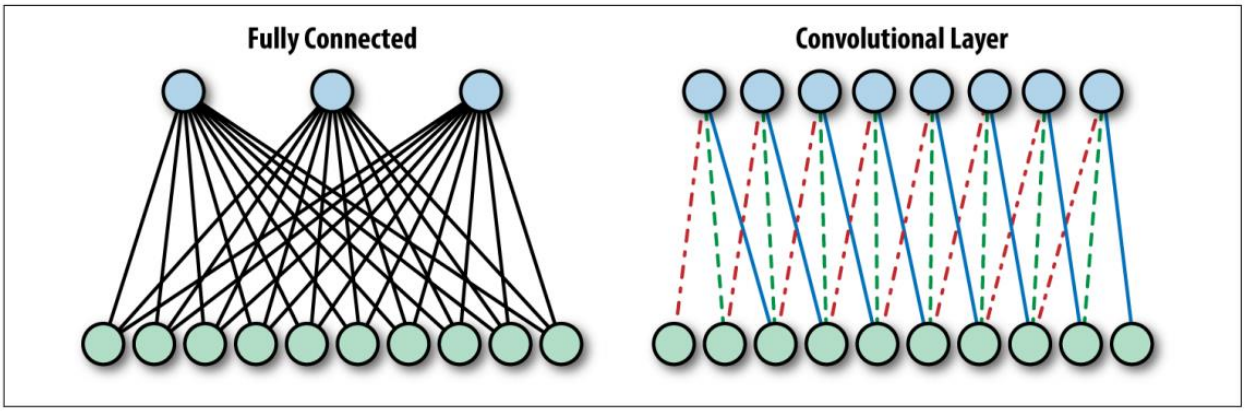
Il metodo descritto è infatti noto come discesa del gradiente o gradient descent.

Ritornando al discorso della background propagation, la rete viene percorsa in senso inverso andando a modificare pesi e bias per minimizzare il gradiente, in modo tale da ottenere un risultato che sia il più reale possibile. Attraverso tali processi matematici la rete è quindi in grado di imparare, modificando le connessioni tra neuroni dei layer successivi, è intuitivo comprendere che più la rete viene allenata (background propagation eseguite) più è accurata nel riconoscere.

Più avanti quando sarà introdotto il **metodo del momento**, saranno spiegati anche altri aspetti relativi alla background propagation e alla discesa del gradiente, in particolare alla **discesa del gradiente stocastico**.

1.2 RETI NEURALI CONVOLUZIONALI

Le reti neurali convoluzionali hanno acquisito uno status speciale negli ultimi anni come una forma particolarmente promettente di deep learning. Radicati nell'elaborazione delle immagini, i layer convoluzionali si sono fatti strada praticamente in tutti i sottocampi del deep learning e sono di grande successo per la maggior parte. La differenza fondamentale tra reti neurali fully connected e convoluzionali è il modello di connessioni tra strati consecutivi. Nel caso fully connected, come suggerisce il nome, ogni unità è collegata a tutte le unità del precedente strato. Un esempio di rete fully connected è stato mostrato precedentemente, dove erano le 10 unità di output collegate a tutti i pixel dell'immagine in ingresso. In uno strato convoluzionale di una rete neurale, invece, ogni unità è connessa a un numero (in genere piccolo) di unità vicine nel livello precedente. Inoltre, tutte le unità sono collegate allo strato precedente allo stesso modo, con lo stesso identico set pesi e struttura. Questo porta a un'operazione nota come convoluzione, che dà il nome a questo tipo di rete neurale. In questa sezione, si entra nell'operazione di convoluzione in modo più dettagliato, ma in poche parole tutto significa applicare una piccola "finestra" di pesi (noti anche come filtri) su un file immagine.



Nella figura si denota che in uno strato completamente connesso (a sinistra), ogni unità è collegata a tutte le unità del precedente strato. In uno strato convoluzionale (a destra), ogni unità è collegata a un numero costante di unità in una regione locale del livello precedente. Inoltre, in uno strato convoluzionale, tutte le unità condividono i pesi per queste connessioni, come indicato dai tipi di linea condivisi.

Ci sono motivazioni comunemente citate che portano all'approccio della CNN, in arrivo da diverse scuole di pensiero. In primo luogo la neuroscienza, mentre il secondo riguarda la comprensione della natura delle immagini e il terzo si riferisce alla teoria dell'apprendimento. È usuale descrivere le reti neurali in generale e in particolare reti neurali convoluzionali, come modelli di calcolo di ispirazione biologica. I neurofisiologi vincitori del premio Nobel, Hubel e Wiesel hanno scoperto già negli anni '60 in cosa consistono le prime fasi dell'elaborazione visiva nel cervello umano. Ossia l'applicazione di un filtro locale (ad esempio, rilevatore di bordi) a tutte le parti del campo visivo. La corrente comprensione della comunità neuroscientifica è che con il procedere dell'elaborazione visiva, le informazioni vengono integrate da parti sempre più ampie. Le reti neurali convoluzionali seguono lo stesso schema. Ogni strato convoluzionale guarda una parte sempre più grande dell'immagine man mano che ci si addentra nella rete. Più comunemente, questo sarà seguito da strati completamente connessi che nell'analogia biologica ispirata agiscono come i livelli superiori di elaborazione visiva che si occupano dell'informazione globale.

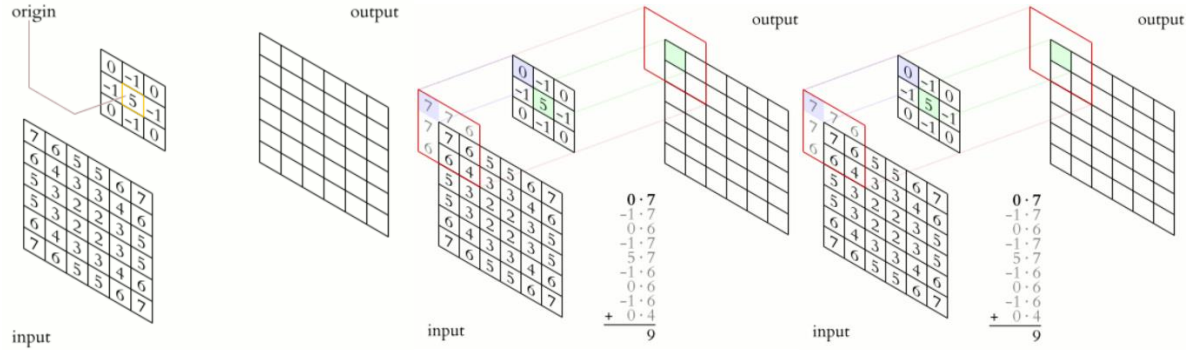
In secondo luogo, più orientato all'ingegneria, deriva dalla natura delle immagini e il loro contenuto. Quando si cerca un oggetto in un'immagine, tipo il volto di un gatto, in genere si è in grado di rilevarlo indipendentemente dalla sua posizione nell'immagine. Questa proprietà è nota come invarianza, si possono prevedere invarianze di questo tipo anche rispetto a (piccole) rotazioni, mutevoli condizioni di illuminazione, ecc…

Di conseguenza, quando si costruisce un sistema di riconoscimento degli oggetti, esso dovrebbe essere invariante alla traslazione (e a seconda dello scenario, probabilmente anche alla rotazione e alle deformazioni di molti tipi). Quindi la rete neurale convoluzionale cerca di identificare specifiche caratteristiche di una figura su tutta l'immagine analizzata. Infine la rete convoluzionale può essere vista come un meccanismo di ottimizzazione, dove i layer convoluzionali invece di applicare i pesi a tutta la matrice (immagine analizzata), tramite la convoluzione li applicano a matrici molto più piccole che scorrono su tutta l'immagine.

1.2.1 La Convoluzione

Nell'elaborazione digitale delle immagini, una **matrice di convoluzione**, **kernel** (nucleo in inglese), o **maschera** è una piccola matrice usata per applicare filtri ad immagini. Risulta dunque utile per la sfocatura, affilatura, goffratura, riconoscimento dei contorni e altro ancora.

Attraverso l'applicazione della convoluzione di due matrici bidimensionali di cui la prima rappresenta l'immagine originale e la seconda, detta **kernel**, rappresenta il **filtro** da applicare. Le matrici kernel sono soprattutto di dimensione dispari, in quanto nella convoluzione è importante identificare il centro della matrice kernel, cosa che avviene facilmente con dimensioni dispari; per esempio, possono essere di dimensione 3x3, 5x5, 7x7, e così via. Difficilmente le matrici kernel sono di grandi dimensioni.



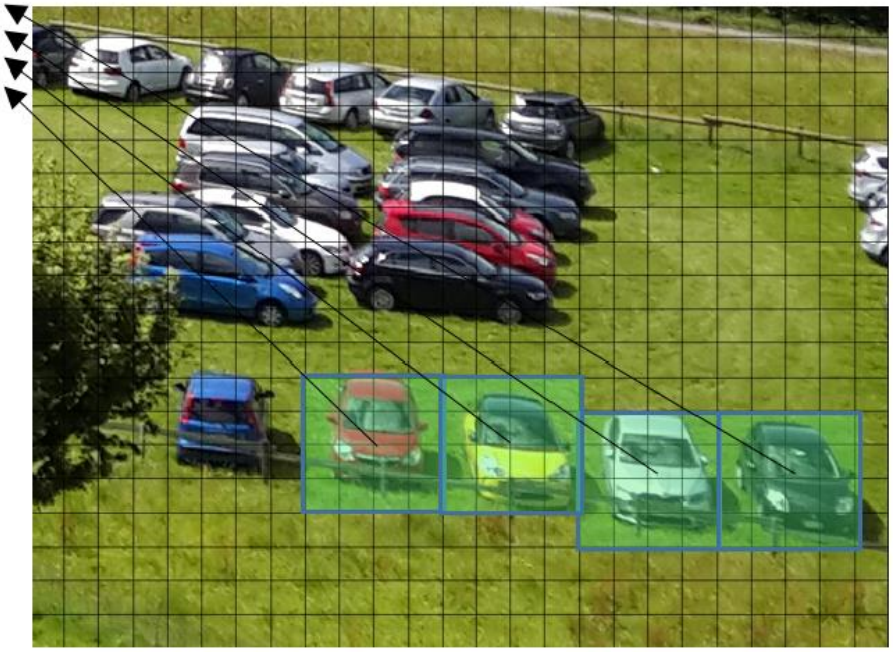
Si consideri la matrice A che rappresenta la matrice contenente i valori di tutti i pixel dell'immagine originale e la matrice B che rappresenta la matrice kernel. Si sovrapponga la matrice B alla matrice A in modo che il centro della matrice B sia in corrispondenza del pixel della matrice A da elaborare.

Il valore di ciascun pixel della matrice A oggetto di elaborazione viene ricalcolato come la somma pesata dei prodotti di ciascun elemento della matrice kernel con il corrispondente pixel della matrice A sottostante.

Questo è il procedimento applicato dai layers convoluzionali, ma non è tutto. Infatti la rete convoluzionale come già detto più volte si basa su un sistema di pesi e bias. Pertanto l'attivazione dei neuroni è analoga a quella delle reti fully connected, con la differenza che i pesi sono tutti gli elementi che compongono le matrici filtro.

Il vantaggio della convoluzione è quindi quello di scorrere e filtrare l'immagine, con diverse finestre filtro in modo tale da poter identificare dei pattern. I pattern permettono quindi di distinguere un campione (oggetto, facce, animali) anche se questo e ruotato o spostato rispetto allo sfondo e questa è la proprietà dell'invarianza (come specificato precedentemente).

w_{11}	w_{12}	w_{13}	w_{14}
w_{21}	w_{22}	w_{23}	w_{24}
w_{31}	w_{32}	w_{33}	w_{34}
w_{41}	w_{42}	w_{43}	w_{44}



Nell'immagine si nota come la matrice filtro, composta inizialmente da pesi casuali, scorra sull'immagine. Riprendendo l'esempio del dataset MNIST si otterrà una situazione di questo tipo con l'utilizzo della rete convoluzionale:

Numero 7

Filtro1

-1	-1	-1
1	1	1
0	0	0

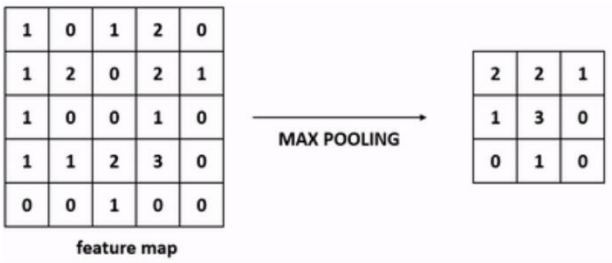
Filtro2

0	0	0
1	1	1
-1	-1	-1

Quindi grazie agli N diversi filtri convoluti con la stessa immagine si ottengono N feature diverse. Mettendo in evidenza le particolarità dell'immagine analizzata.

1.2.2 Pooling

Solitamente dopo un layer convoluzionale si utilizza il pooling. Il motivo è semplice: i filtri del layer convoluzionale forniscono informazioni più dense e pure, perchè evidenziano alcune caratteristiche eliminandone altre che costituirebbero rumore. Queste informazioni sono quindi più facili da elaborare e non è necessario portarsi dietro tutta la pesantezza data da una immagine di grandi dimensioni. Il pooling serve proprio a questo: diminuire le dimensioni dell'immagine in input, mantenendo le caratteristiche principali della stessa. Per esempio una funzione utilizzata è il **max pooling**; si faccia finta di avere una matrice 5x5, se ne vogliono ridurre le dimensioni perciò viene applicato un pooling con una matrice 2x2, il risultato sarà dato da una nuova matrice 3x3 dove ogni elemento è l'elemento maggiore nelle diverse finestre 2x2 che scorrono sulla matrice più grande. Un'altra funzione di pooling può essere l'**average pooling**, che funziona in modo del tutto analogo alla funzione precedente, ma invece di prendere il valore più grande prende quello medio.



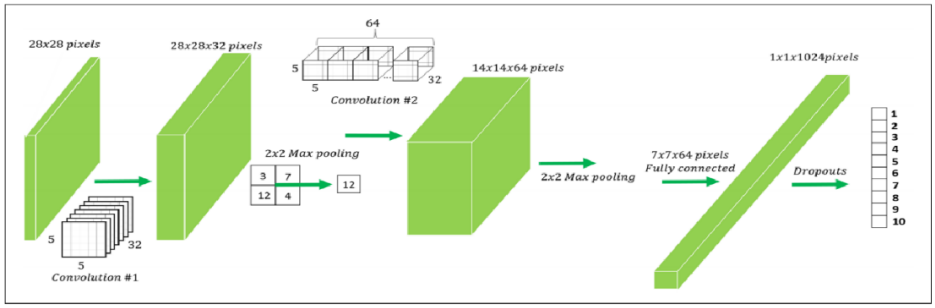
Se si vuole far rimanere l'immagine con le dimensioni originali pur volendo applicare il pooling, bisogna usare il padding (riempimento) per l'immagine di ingresso, ossia nel caso dello **zero padding** vengono aggiunti ai bordi dell'immagine righe e colonne di zeri per far sì che dopo il pooling le dimensioni dell'immagine siano le stesse di quella in ingresso. Ovviamente si avrà allo stesso modo la riduzione della pesantezza dell'immagine, in quanto la sua risoluzione si abbasserà.

Un altro utilizzo del padding è sempre quello di aggiungere zeri ai bordi dell'immagine, ma non per preservarne le dimensioni, bensì quando la finestra di convoluzione ha una dimensione e un passo che la fanno uscire fuori dall'immagine, gli zeri aggiunti permettono questa parziale fuoriuscita della finestra e che venga eseguita correttamente la convoluzione.

1.2.3 Dropout

Il dropout permette di spegnere una percentuale di neuroni per ogni hidden layer. Per ogni iterazione i neuroni spenti sono diversi, questo meccanismo permette una migliore generalizzazione della rete per prevenire un adattamento della rete rispetto ai dati analizzati, ossia quando la rete impara a memoria i dati invece di distinguerli per le loro caratteristiche. Questo problema viene chiamato **overfitting**.

Infine, viene mostrata la struttura vera e propria della rete neurale convoluzionale, utilizzata per l'esempio iniziale con il dataset MNIST



Si può vedere come nel primo layer, la singola immagine 28x28 viene filtrata attraverso la convoluzione da 32 finestre filtro 5x5, per poi essere dimezzata in dimensioni attraverso un max pooling 2x2. In seguito nel secondo layer viene eseguita un'altra convoluzione con 64 filtri diversi e lo stesso max pooling. Nel penultimo layer le 64 diverse immagini filtrate 7x7 vengono "appiattite" in un array lungo 1024 pixel. Infine nel layer finale ci sono i 10 neuroni i quali identificano ognuno un numero che va da 0 a 9.

1.3 FUNZIONI E ALGORITMI PER IL MODELLO SVILUPPATO

Le funzioni e gli algoritmi che vengono utilizzati dal modello sviluppato sono:
activation→ relu per i layer convoluzionali e softmax per l'ultimo layer
loss function→sparse categorical crossentropy
optimizer→adam

1.3.1 Activation

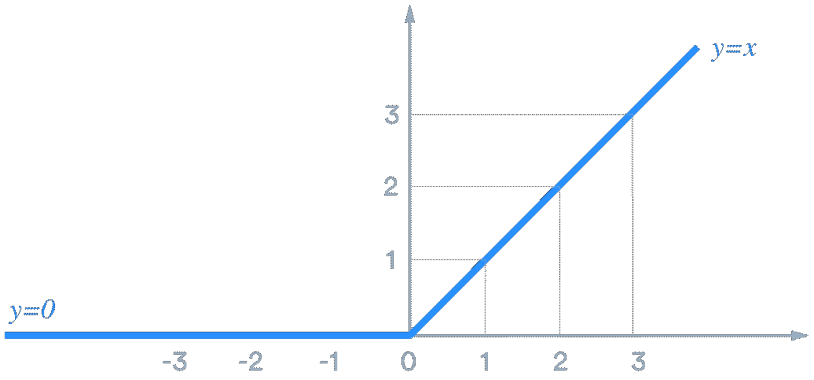
Le funzioni di attivazione, come detto in precedenza, sono utilizzate per attivare ogni singolo neurone, hanno come argomento gli input del neurone ossia la sommatoria degli output del layer precedente moltiplicati per i pesi e sommati per i bias.

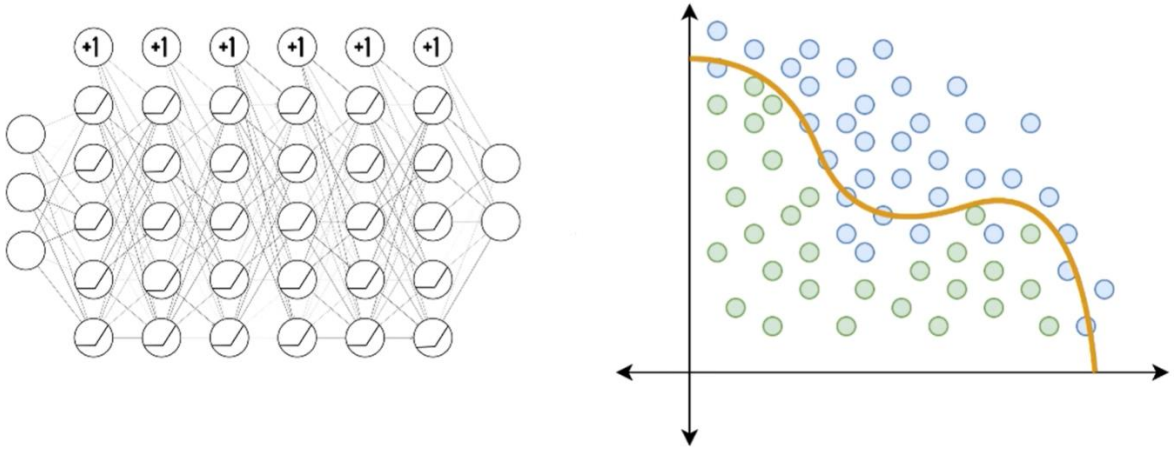
RELU

Nel contesto delle reti neurali artificiali, il rettificatore è una funzione di attivazione definita come la parte positiva del suo argomento:

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

dove x è l'input di un neurone





Le funzioni di attivazione aiutano a gestire la non linearità del modello e a garantirne una buona separazione tra i dati. Nella figura è possibile vedere come una rete che lavora per il riconoscimento di due classi, idealmente è come se separasse i dati mescolati su un piano con una specifica funzione, che si ottimizza sempre di più fino a ottenere una separazione quasi totale tra le due classi.

SOFTMAX

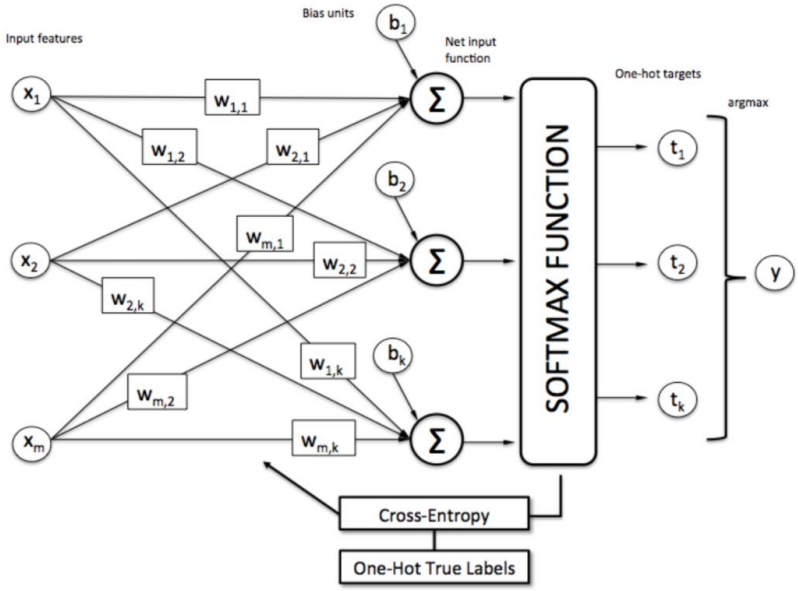
La softmax viene utilizzata come funzione di attivazione per attività di classificazione multi-classe, solitamente nell'ultimo livello. Il suo ruolo consiste nel trasformare i numeri in probabilità che si sommano a uno.

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

(x valore generico, rappresenta la somma degli output del layer precedente moltiplicati per pesi e sommati per i bias)

Esempio generico: dato in input un vettore (1;2;3;4;1;2;3) la funzione softmax restituirà (0,024; 0,064; 0,175; 0,475; 0,024; 0,064; 0,175). Il risultato assegna gran parte del peso al numero 4, il cui valore in uscita risulta essere circa 20 volte maggiore del valore associato a 1. Questo è esattamente ciò per cui la funzione solitamente è usata: mettere in evidenza i valori più grandi e nascondere quelli che sono significativamente più piccoli del valore massimo.

Nelle reti neurali la softmax fa sì che per gli n neuroni del layer finale (ognuno rappresenta una delle n classi), il valore di attivazione di ognuno di essi sia un numero razionale compreso tra zero e uno, che rappresenta la probabilità di essere predetto dalla rete in base al dato analizzato.



Softmax Regression

1.3.2 Loss Function

SPARSE CATEGORICAL CROSSENTROPY

(entropia nella teoria dell'informazione è il numero di bit necessari per dare un'informazione, in questo caso il valore medio dei bit)

Formula della crossentropy/sparse categorical crossentropy

$$C(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

- $\mathbf{w}$ pesi della rete
- y_i label reale
- $\hat{y}_i$ label predetta
- N numero classi

La formula per entrambe le loss function è la stessa, ma la crossentropy loss si usa quando si hanno pochi output e le label sono vettori one-hot rappresentati da 1 e 0, la sparse categorical crossentropy si usa invece quando si ha un gran numero di output e le label sono rappresentate da interi.

1.3.3 Optimizer ADAM

Il concetto di ottimizzazione gioca un ruolo chiave quando si parla di machine learning e deep learning in particolare. Lo scopo principale degli algoritmi di deep learning è quello di costruire un modello di ottimizzazione che, come già visto, tramite un processo iterativo, minimizzi o massimizzi una loss function C(θ). Dove il parametro θ indica in modo più generico, ciò che era stato indicato precedentemente come la totalità di pesi e biases presenti nella rete.

I più popolari metodi di ottimizzazione possono essere suddivisi in due categorie: metodi di ottimizzazione del primo ordine, rappresentati dal metodo del gradiente, e metodi di ottimizzazione del secondo ordine o di ordine superiore, fra i quali il metodo di Newton ne è un tipico esempio. In questo paragrafo si analizzeranno i metodi di ottimizzazione del primo ordine, dei quali il metodo della discesa del gradiente stocastico e tutte le sue varianti sono fra quelli ampiamente utilizzati ed in continua evoluzione.

Nei modelli di machine learning e deep learning i metodi di ottimizzazione del primo ordine principalmente utilizzati sono basati sul concetto di discesa del gradiente.

La discesa del gradiente

Il metodo della discesa del gradiente è il primo e più comune metodo di ottimizzazione. L'algoritmo si basa, come già accennato precedentemente, sull'aggiornamento iterativo di un parametro θ lungo la direzione opposta a quella del gradiente della funzione obiettivo C(θ). L'aggiornamento viene sviluppato in modo da convergere gradualmente al valore ottimo della funzione obiettivo. Uno dei parametri del metodo è il learning rate η che determina l'ampiezza della variazione di θ in ciascuna iterazione e quindi influenza il numero di iterazioni necessarie a raggiungere il valore ottimo della funzione obiettivo. Il metodo è di semplice implementazione e nel caso in cui C(θ) sia una funzione (fortemente) convessa, la soluzione trovata sarà un punto di ottimo globale. Nel caso in cui la funzione obiettivo non sia convessa, la soluzione trovata potrebbe essere un minimo locale (o più in generale un punto stazionario). Il nome deriva dal fatto che, ad ogni iterazione, l'algoritmo impiega tutti i dati del training set per calcolare il gradiente $\nabla_\theta C(\theta)$

Discesa del gradiente stocastico I (SGD)

Nel metodo appena descritto, il gradiente $\nabla_\theta C(\theta)$ è calcolato utilizzando, ad ogni iterazione, tutto il training set. Questo determina una elevata e ridondante complessità computazionale.

Per ovviare a questo punto debole la soluzione proposta è stata quella del metodo della discesa del gradiente stocastico. L'idea è quella di calcolare il gradiente non più mediante tutto il training set ma mediante un singolo campione selezionato in modo casuale ad ogni iterazione. L'utilizzo di un singolo campione determina una forte diminuzione della complessità computazionale.

Discesa del gradiente stocastico II (mini-batch)

L'utilizzo di una selezione casuale di singoli campioni dal training set ha lo svantaggio di determinare una oscillazione della direzione del gradiente e un procedere in modo cieco del processo di ricerca della soluzione all'interno dello spazio delle soluzioni.

Un modo per diminuire la varianza del gradiente è stato quello di introdurre una variante denominata mini-batch gradient descent. Questa modalità impiega, ad ogni iterazione, un insieme di n campioni del training set e con questi calcola il gradiente ∇_θC(θ), questa particolare iterazione è detta epoch. In questo modo si raggiunge il duplice obiettivo di ridurre la varianza del gradiente e rendere più stabile la convergenza.

Metodo del momento

Nonostante il metodo SGD sia molto popolare ed ampiamente utilizzato, il processo di aggiornamento di θ risulta spesso molto lento. Fra i punti aperti infatti c'è una più opportuna

regolazione del learning rate per velocizzare la convergenza e fare in modo che il processo di ricerca della soluzione non rimanga intrappolato in un minimo locale di C(θ). Una delle idee che si è fatta strada è quella del "momento" che, per come è stato pensato, gioca il ruolo di una velocità v. Il concetto è infatti derivato dalla fisica, in particolare dalla meccanica, ed è stato pensato per accelerare il processo di aggiornamento di θ, specialmente in casi di elevate curvature di C(θ) e di valori piccoli, costanti e rumorosi del gradiente $\nabla_\theta C(\theta)$. L'idea è quella di calcolare, ad ogni iterazione, una media mobile esponenziale dei gradienti storici ed utilizzare questo valore come direzione da seguire nell'aggiornamento di θ.

Uno degli algoritmi che implementa questo tipo di ottimizzazione, detta metodo del momento, si chiama ADAM.

Capitolo II: MODELLO SVILUPPATO

In questo capitolo verrà mostrato il codice scritto per la preparazione del dataset e il suo utilizzo per l'allenamento della rete neurale convoluzionale, in seguito il modello della rete verrà convertito in tflite in modo tale da renderlo ottimale per i test su scheda.

Il modello è stato implementato sulla ide COLAB offerta da Google, in linguaggio Python, utilizzando i framework di Tensorflow e Keras.

Keras è una libreria open source per l'apprendimento automatico e le reti neurali, scritta in Python. È progettata come un'interfaccia a un livello di astrazione superiore di altre librerie simili di più basso livello, e supporta come back-end le librerie TensorFlow.

TensorFlow è una libreria software open source per l'apprendimento automatico (machine learning), che fornisce moduli sperimentati e ottimizzati, utili nella realizzazione di algoritmi per diversi tipi di compiti percettivi e di comprensione del linguaggio. È una seconda generazione di API, utilizzata da una cinquantina di team attivi sia in ambiti di ricerca scientifica, sia in ambiti di produzione; è alla base di dozzine di prodotti commerciali Google come il riconoscimento vocale, Gmail, Google Foto, e Ricerca.

2.1 PRESENTAZIONE E PREPARAZIONE DEL DATASET

Il dataset utilizzato, il COIL-20, è una raccolta di immagini della COLUMBIA UNIVERSITY che contiene 20 classi di oggetti e 72 immagini per ogni classe, con un totale di 1440 immagini. Le immagini sono in bianco e nero con sfondo neutro, con una risoluzione di 128x128 pixel.

Una volta scaricato il dataset, si avranno in un'unica cartella tutte le immagini in formato PNG. Le immagini per poter essere utilizzate vengono caricate nel proprio Drive di Google

Come si può notare dall'immagine tutte le foto hanno un'etichetta che le identifica con il numero dell'oggetto (tipo di classe) e il numero della foto.

```
import numpy as np
import matplotlib.pyplot as plt
import os
import cv2
from sklearn.model_selection import train_test_split
from google.colab import drive

drive.mount('/content/drive')
DATADIR = '/content/drive/My Drive/coil20'
classi=[]
training_data=[]
labels=[]

entries = os.listdir(DATADIR)
```

Nella figura riportata è possibile descrivere le prime righe del codice scritto. Innanzi tutto vengono importate tutte le librerie del Python utili per la preparazione del dataset:

- **numpy** consente di lavorare con vettori e matrici in maniera più efficiente e veloce di quanto non si possa fare con le liste e le liste di liste (matrici). Il costrutto di base è l'ndarray, che può avere dimensioni qualunque e tipi corrispondenti a quelli classici del C o del Fortran, che restano i linguaggi tuttora più utilizzati per il calcolo numerico. Uno dei punti di forza di numpy è di poter lavorare sui vettori sfruttando le ottimizzazioni di calcolo vettoriale del processore della macchina. Ciò rende particolarmente efficiente i calcoli, rispetto alle liste.
- **Matplotlib** è una libreria per la creazione di grafici per il linguaggio di programmazione Python.
- **os** è una libreria che consente di interagire con il sistema operativo, nel nostro caso sarà utile per navigare all'interno della directory del Google Drive e prendere le immagini caricate e le loro etichette.
- **cv2** è una libreria che integra numpy per la lettura di immagini. Permette quindi di caricare un'immagine dal file specificato, l'immagine viene letta sotto forma di un array multidimensionale, se l'immagine non può essere letta (a causa di file mancanti, autorizzazioni improprie, formato non supportato o non valido), questo metodo restituisce una matrice vuota.
- **sklearn** è una libreria open source di apprendimento automatico per il linguaggio di programmazione Python. Contiene algoritmi di classificazione, regressione e clustering (raggruppamento). Sarà utilizzata per la costruzione degli array di train e test.

  Inoltre importeremo il Google drive per poter prelevare le immagini in esso salvate

Una volta importate tutte le librerie necessarie sopra descritte, si preleva il dataset contenuto nella directory del drive '/contet/drive/My Drive/coil20'. Attraverso l'utilizzo della libreria **os** si costruisce l'array entries composto da tutte stringhe, ovvero le etichette delle immagini montate sul drive.

```
for i in range(len(entries)):
    name_photo=entries[i]
    class_obj=name_photo[3:5]

    if class_obj[1]=='_':
       class_obj=class_obj.replace('_','')

    class_obj=int(class_obj)
    class_obj=class_obj-1
    labels.append(class_obj)
labels=np.array(labels)

for img in os.listdir(DATADIR):
    img_array = cv2.imread(os.path.join(DATADIR,img) ,cv2.IMREAD_GRAYSCALE)
    new_array = cv2.resize(img_array, (32,32))
    img_data.append(new_array)

img_data=np.array(img_data)

X_train, X_test, y_train, y_test = train_test_split(img_data, labels, test_size=0.2)# random_state=2
plt.imshow(X_train[100])
plt.show()
print(y_train[100])

X_train=X_train.astype('float32')
X_test=X_test.astype('float32')
X_train/=255
X_test/=255
```

Ricordando la teoria del capitolo precedente la prima cosa da fare è modificare le label, in quanto queste sono stringhe contenenti caratteri e numeri. Le label infatti per poter essere processate dalla rete devono essere numeri interi, quindi si scarterà tutto tranne il numero della classe. Con un ciclo for vengono create le label di interi e allocate in un array. In tale ciclo vengono eliminati tutti i caratteri tranne il numero della classe dell'oggetto, che però va da 1 a 20, questo sarà un problema per l'ultimo layer della rete a cui risulteranno 21 classi, perciò bisogna scalare gli interi di uno per avere le label che vanno da 0 a 19. In questo modo sarà possibile costruire l'ultimo layer con 20 neuroni, uno per classe. Si avrà quindi l'array labels che contiene appunto tutte le etichette come prima specificato.

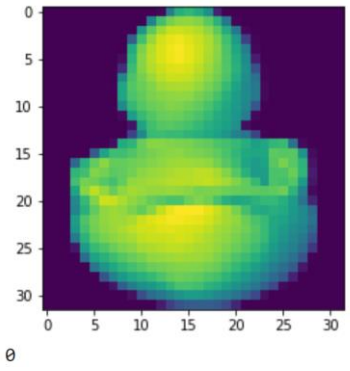
Con il secondo for grazie alla libreria cv2, è possibile leggere tutte le immagini sul drive e farne un ridimensionamento ottenendo immagini da 128x128 a 32x32 pixel, si ottiene perciò un array multidimensionale lungo 1440 elementi dove ogni elemento è una matrice 32x32, il nome di questo array è img_data.

Poi con l'utilizzo di train_test_split della libreria **sklearn** vengono letti in modo casuale gli array delle immagini e delle corrispondenti label, per generare i rispettivi array di train e test. Gli array sopra definiti sono: X_train, X_test, y_train, y_test (X=immagini, y=label). L'80% delle immagini sarà utilizzato come train e il 20% come test.

Infine i valori dei pixel delle immagini vengono normalizzati, questa procedura rende il training più veloce.

```
print(len(labels))
print(X_train.shape)
print(entries)
```

```
1440
(1152, 32, 32)
['obj15__65.png', 'obj16__11.png', 'obj15__69.png', 'obj16__10.png', 'obj15__56.png', 'obj16__12.png', 'obj15__5.
```



Per fare un primo controllo che tutto funzioni correttamente, viene stampata un'immagine da X_ train e la sua relativa etichetta da y_train. Come già visto nella prima e nella seconda immagine di questo capitolo, la "paperella" è l'oggetto 1, con il ridimensionamento della numerazione delle classi prima descritta, l'etichetta corrispondente a tutta questa classe di "paperelle" sarà 0.

2.2 RETE NEURALE CONVOLUZIONALE IMPLEMENTATA

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D

X_train_reshape = np.array(X_train).reshape(-1, 32,32, 1)
X_test_reshape = np.array(X_test).reshape(-1, 32,32, 1)

model = Sequential()

model.add(Conv2D(64, (3, 3), padding='same', input_shape=X_train_reshape.shape[1:]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.7))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.7))

model.add(Flatten())
model.add(Dense(20,activation='softmax'))

model.summary()

model.compile(loss='sparse_categorical_crossentropy',
              optimizer='adam',
              metrics='accuracy')
```

```
history=model.fit(X_train_reshape, y_train, batch_size=32, epochs=30, validation_split=0.2)
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'g', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'g', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

Nella parte di codice illustrata nelle due immagini precedenti si passa alla realizzazione della rete, quindi all'implementazione dei layer e al training vero e proprio.

In primo luogo occorre importare le librerie tensorflow e keras per poter utilizzare gli strumenti adatti a creare la rete neurale convoluzionale. Il modello che viene implementato è un modello di tipo sequenziale, in quanto si ha una semplice pila di layer dove ogni layer ha esattamente un tensore* di input e un tensore di output.

Prima di dare in ingresso l'array del training alla rete, occorre ridimensionarlo opportunamente per ottenere la forma voluta per il tensore di input. L'array di ingresso a una CNN deve necessariamente avere 4 dimensioni (numero immagini, altezza, larghezza, profondità) dove la profondità in questo caso è 1 visto che le immagini sono in greyscale, nel caso di immagini a colori RGB la profondità sarebbe stata 3.

```
X_train_reshape = np.array(X_train).reshape(-1, 32,32, 1)
X_test_reshape = np.array(X_test).reshape(-1, 32,32, 1)
```

Queste due righe di codice eseguono il ridimensionamento sopra descritto. Il criterio del ridimensionamento è che la nuova forma dell'array deve essere compatibile con quella originale. Aggiungendo il -1 al reshape si lascia che numpy calcoli i valori rimanenti da inserire nella nuova forma, in questo caso il numero di immagini, che viene mantenuto uguale all'originale: 1152 per il train e 288 per il test.

`model = Sequential()` Inizializza model come un modello sequenziale.

Fatti questi primi passaggi si passa alla scrittura delle righe di codice che costruiscono i layer e quindi il modello. Il primo layer che esegue una convoluzione 2D viene implementato da:

```
model.add(Conv2D(64, (3, 3), padding='same', input_shape=X_train_reshape.sha
pe[1:]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.7))
```

Riceve in ingresso l'array 4D contenete le immagini del train, su queste immagini vengono convoluti per ognuna 64 filtri, di cui vengono specificate le dimensioni, 3x3 in 2D. Viene eseguito un zero padding same, in conseguenza della dimensione 3x3 della finestra di convoluzione e del passo di default 1 che essa compie (ossia di quanto si sposta la finestra in larghezza e in altezza sull'immagine a ogni scorrimento).
Nelle righe successive si aggiungono: la funzione di attivazione relu, il max pooling di dimensione 2x2 ed infine un dropout a 0.7, che a ogni iterazione spegne in modo random il 30% dei neuroni del layer, per evitare l'overfitting.

Dopo aver implementato i due layer convoluzionali, si passa alla costruzione del layer fully connected e al layer finale (anch'esso fully connected), dove ogni neurone identifica una classe:

```
model.add(Flatten()
model.add(Dense(20,activation='softmax'))
```

**In matematica, la nozione di tensore generalizza tutte le strutture definite usualmente in algebra lineare a partire da un singolo spazio vettoriale. Sono particolari tensori i vettori, gli endomorfismi, i funzionali lineari e i prodotti scalari.*

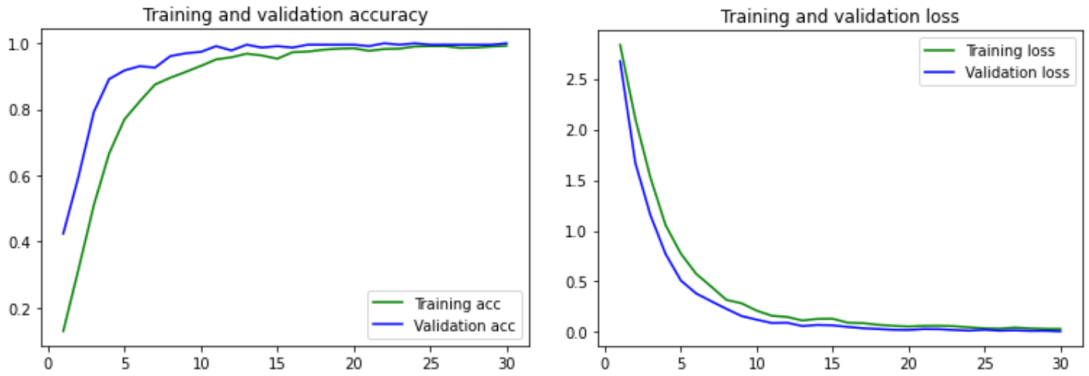
Come già specificato nell'introduzione teorica, per il riconoscimento multiclasse è necessario utilizzare come loss function la sparse categorical crossentropy che a sua volta necessita che la funzione di attivazione dell'ultimo strato sia la softmax.

La rete è costruita, l'ultimo passaggio sarà compilarla e allenarla:

```
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='adam',
              metrics='accuracy')
history=model.fit(X_train_reshape, y_train, batch_size=32, epochs=30, valida
tion_split=0.2)
```

Con queste poche righe di codice si specificano la loss function della rete e l'optimizer `'adam'` già descritto precedentemente. Inoltre si va ad allenare il modello con gli array: `X_train_reshape, y_train.` Si specificano anche il gruppo utilizzato per la discesa del gradiente stocastico, il numero di epoch da eseguire per allenare la rete e un validation set del 20%. Il validation set è un altro dataset costruito da una porzione del training set e serve per valutare l'apprendimento della rete durante ogni epoch.

Le righe di codice che seguono le precedenti, servono per stampare le seguenti immagini:



In tali grafici si può evincere come l'accuracy aumenta a ogni epoch e viceversa come diminuisce la loss function. Inoltre si evince che non vi è overfitting, in quanto l'accuracy del validation set cresce quasi in maniera monotona, o meglio non subisce bruschi cali rispetto all'accuracy del training set. Questo perché la rete, allenata con il training set, non "impara a memoria" le immagini analizzate, quindi non ha difficoltà a riconoscere le immagini appartenenti al validation set, che sono diverse rispetto a quelle dell'allenamento. Dai grafici si può ancora aggiungere che non vi è underfitting, ossia che la rete non è stata allenata meno del dovuto, o meglio che non c'è stato un sottocampionamento, ma i dati forniti alla rete per l'allenamento sono abbastanza per permetterle di riconoscere anche immagini non appartenenti al training set. Infatti in entrambi i grafici si nota che le due curve tendono sempre più a sovrapporsi e ad assumere un valore costante, mostrando che all'aumentare delle epoch non ci sono più differenze tra le curve del training e quelle del validation e che i valori di accuracy e loss si sono stabilizzati.

```
29/29 [==============================] - 2s 82ms/step - loss: 0.0483 - accuracy: 0.9840 - val_loss: 0.0312 - val_accuracy: 0.9913
Epoch 22/30
29/29 [==============================] - 2s 83ms/step - loss: 0.0719 - accuracy: 0.9747 - val_loss: 0.0297 - val_accuracy: 1.0000
Epoch 23/30
29/29 [==============================] - 2s 82ms/step - loss: 0.0601 - accuracy: 0.9812 - val_loss: 0.0213 - val_accuracy: 0.9957
Epoch 24/30
29/29 [==============================] - 2s 82ms/step - loss: 0.0452 - accuracy: 0.9890 - val_loss: 0.0152 - val_accuracy: 1.0000
Epoch 25/30
29/29 [==============================] - 2s 82ms/step - loss: 0.0315 - accuracy: 0.9931 - val_loss: 0.0234 - val_accuracy: 0.9957
Epoch 26/30
29/29 [==============================] - 2s 84ms/step - loss: 0.0408 - accuracy: 0.9890 - val_loss: 0.0144 - val_accuracy: 0.9957
Epoch 27/30
29/29 [==============================] - 2s 82ms/step - loss: 0.0395 - accuracy: 0.9880 - val_loss: 0.0181 - val_accuracy: 0.9957
Epoch 28/30
29/29 [==============================] - 2s 81ms/step - loss: 0.0373 - accuracy: 0.9858 - val_loss: 0.0126 - val_accuracy: 0.9957
Epoch 29/30
29/29 [==============================] - 2s 83ms/step - loss: 0.0366 - accuracy: 0.9894 - val_loss: 0.0132 - val_accuracy: 0.9957
Epoch 30/30
29/29 [==============================] - 2s 84ms/step - loss: 0.0379 - accuracy: 0.9911 - val_loss: 0.0077 - val_accuracy: 1.0000
```

La figura riportata mostra i valori di accuracy e loss, del training set e del validation set, per le ultime 10 epochs eseguite

Viene stampato inoltre il sommario del modello:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 32, 32, 64)        640
_________________________________________________________________
activation_4 (Activation)    (None, 32, 32, 64)        0
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 16, 16, 64)        0
_________________________________________________________________
dropout_4 (Dropout)          (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 14, 14, 64)        36928
_________________________________________________________________
activation_5 (Activation)    (None, 14, 14, 64)        0
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
dropout_5 (Dropout)          (None, 7, 7, 64)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 3136)              0
_________________________________________________________________
dense_2 (Dense)              (None, 20)                62740
=================================================================
Total params: 100,308
Trainable params: 100,308
Non-trainable params: 0
_________________________________________________________________
```

Dal sommario possiamo vedere come ogni immagine per via del pooling viene dimezzata nei primi due layer per poi arrivare a quelli fully connected, inoltre è specificato il numero di parametri totali utilizzati dalla rete.

2.3 VALUTAZIONI SUL MODELLO E CONVERSIONE IN TFLITE

Implementata la rete, occorre testarla per verificare la sua accuratezza nel riconoscere le immagini appartenenti all'array di test.

```
loss, acc = model.evaluate(X_test_reshape,y_test, verbose=2)
print('Restored model, accuracy: {:5.2f}%'.format(100*acc))
```

```
9/9 - 0s - loss: 0.0072 - accuracy: 1.0000
Restored model, accuracy: 100.00%
```

Dalla figura riportare si può leggere l'accuratezza del modello sul riconoscimento delle immagini di test. L'accuratezza che si ottiene è del 100.00%. La situazione è ottima anche per via della semplicità delle immagini appartenenti al dataset.

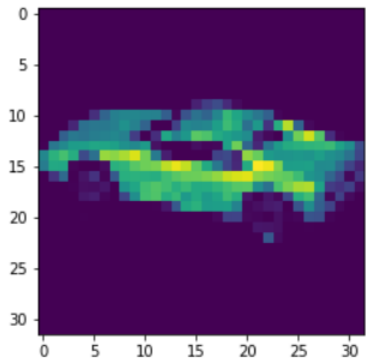
Per un ulteriore verifica si utilizza il modello per riconoscere un'immagine casuale appartenente alla parte del dataset dedicata al test e a stampare la relativa etichetta predetta.

```
import random
import numpy
a = numpy.random.randint(0,100)
predictions=model.predict([X_test_reshape])

print(np.argmax(predictions[a]))
print(predictions[a])
print(y_test[a])

plt.imshow(X_test[a])
plt.show
```

```
2
[1.11306917e-07 1.50727968e-13 9.47706401e-01 7.78339455e-08
 6.51851122e-04 5.07439077e-02 7.04697367e-09 1.43846157e-10
 3.48928694e-07 5.76592818e-09 1.59929027e-06 7.57785016e-14
 1.40416310e-08 1.59310586e-11 3.86129193e-12 8.19936020e-07
 1.80781310e-11 1.07714795e-05 8.19128298e-04 6.49467693e-05]
2
<function matplotlib.pyplot.show>
```



La label predetta e quella effettiva di tale immagine coincidono, quindi il modello ha riconosciuto l'immagine.

Avendo testato l'efficienza della rete ora bisogna valutarne le dimensioni per poi convertirla in tflite prima di usarla sulla board.

```
def get_file_size(file_path):
    size = os.path.getsize(file_path)
    return size

def convert_bytes(size, unit=None):
    if unit == "KB":
        return print('File size: ' + str(round(size / 1024, 3)) + ' KB')
    elif unit == "MB":
        return print('File size: ' + str(round(size / (1024 * 1024), 3)) + ' MB')
    else:
        return print('File size: ' + str(size) + ' bytes')
```

```
!pip install -q pyyaml h5py  # Required to save models in HDF5 format
saved_model = "model.h5"
model.save(saved_model)
convert_bytes(get_file_size(saved_model), "MB")
```

```
File size: 1.185 MB
```

La prima parte dello script consente di convertire il peso del file analizzato in KB o MB, unità più comode per l'analisi rispetto al byte.

Dopodiché il modello viene salvato per poi poter essere analizzato con la funzione convert_bytes(get_file_size()) che restituisce appunto il peso del modello, **1.185 MB**.

Grazie alla quantizzazione e alla conversione in tflite il modello viene ridotto di dimensione e reso ottimale per essere utilizzato dall'hardware, in modo da ottenere predizioni con una buona accuratezza a bassa latenza. La quantizzazione è una strategia di ottimizzazione che converte i numeri in virgola mobile a 32 bit (come pesi e uscite di attivazione) nei numeri in virgola mobile a 8 bit. Ciò si traduce in un modello più piccolo e in una maggiore velocità di inferenza, il che è prezioso per i dispositivi a bassa potenza come i microcontrollori.

Di seguito il codice che quantizza il modello e lo porta dal formato proprio di keras h5 al formato tflite.

```
def representative_data_gen():
  for input_value in tf.data.Dataset.from_tensor_slices(X_train_reshape).batch(1).take(100):
    # Model has only one input so each data point has one element.
    yield [input_value]

converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data_gen
tflite_model_quantFloat = converter.convert()

interpreter = tf.lite.Interpreter(model_content=tflite_model_quantFloat)
input_type = interpreter.get_input_details()[0]['dtype']
print('input: ', input_type)
output_type = interpreter.get_output_details()[0]['dtype']
print('output: ', output_type)

# Save the quantized float model:
import pathlib
tflite_models_dir = pathlib.Path("./")
tflite_models_dir.mkdir(exist_ok=True, parents=True)
tflite_model_quantFloat_file = tflite_models_dir/"model_quantFloat.tflite"
tflite_model_quantFloat_file.write_bytes(tflite_model_quantFloat)

# Estimate size
convert_bytes(get_file_size(tflite_model_quantFloat_file), "MB")
```

```
INFO:tensorflow:Assets written to: /tmp/tmprouc9z_l/assets
input:  <class 'numpy.float32'>
output:  <class 'numpy.float32'>
File size: 0.104 MB
```

Dalla figura riportata si nota la grande riduzione di peso ottenuta dalla quantizzazione del modello, da **1.85 MB** a **0.104MB**.

Fatto ciò il modello è stato costruito, testato e quantizzato, ora è possibile utilizzarlo per il riconoscimento in tempo reale delle immagini appartenenti al dataset, su board OpenMV Cam H7 Plus. Prima di passare al codice scritto sull'ide della scheda, occorre introdurre le caratteristiche della board utilizzata.

Capitolo III: TEST IN TEMPO REALE SU BOARD OPENVCAM H7 PLUS

3.1 PRESENTAZIONE OPENMVCAM H7 PLUS

OpenMV Cam e scheda SD della SanDisk da 16 GB

OpenMV Cam è una piccola scheda microcontrollore a bassa potenza che consente di implementare facilmente applicazioni utilizzando la visione artificiale nel mondo reale. Si programma OpenMV Cam in script Python di alto livello (per gentile concessione del sistema operativo **MicroPython***) invece che in C / C ++. Ciò rende più facile gestire i complessi output degli algoritmi di visione artificiale e lavorare con strutture di dati di alto livello.

L'OpenMV cam monta un processore ARM Cortex M7 STM32H743II funzionante a 480 MHz con 32 MB di SDRAM + 1 MB di SRAM e 32 MB di flash esterno + 2 MB di flash interno. Tutti i pin I / O emettono 3,3 V e tollerano 5 V. Il processore ha le seguenti interfacce I / O:

- Un'interfaccia USB a piena velocità (12 Mb) per il tuo computer. L' OpenMV Cam appare come una porta COM virtuale e una chiavetta USB quando è collegata.
- Una presa per scheda µSD in grado di leggere / scrivere a 100 Mb che consente alla tua OpenMV Cam di scattare foto e di estrarre facilmente le risorse di visione artificiale dalla scheda µSD.
- Un bus SPI che può funzionare fino a 80 Mb che consente di trasmettere facilmente i dati delle immagini dal sistema allo schermo LCD, allo schermo WiFi o ad un altro microcontrollore.
- Un bus I2C (fino a 1 Mb / s), bus CAN (fino a 1 Mb / s) e un bus seriale asincrono (TX / RX, fino a 7,5 Mb / s) per l'interfacciamento con altri microcontrollori e sensori.
- Un ADC a 12 bit e un DAC a 12 bit.
- Due pin I / O per servocomando.
- Interrupt e PWM su tutti i pin I / O (ci sono 10 pin I / O sulla scheda).
- Inoltre, un LED RGB e due LED IR da 850 nm ad alta potenza.
- 32 MB di SDRAM esterna a 32 bit con clock a 100 MHz per 400 MB / s di larghezza di banda.
- 32 MB di flash quadspi esterno con clock a 100 MHz in modalità DDR a 4 bit per 100 MB / s di larghezza di banda (velocità di lettura).
- Un sistema di modulo telecamera rimovibile che consente a OpenMV Cam H7 di interfacciarsi con diversi sensori:

OpenMV Cam H7 Plus viene fornita con un sensore di immagine OV5640 in grado di acquisire immagini 2592x1944 (5MP). La maggior parte degli algoritmi semplici funzioneranno tra 25-50 FPS su risoluzioni QVGA (320x240) e inferiori. Il sensore di immagine viene fornito con un obiettivo da 2,8 mm su un attacco obiettivo M12 standard.

Il supporto di TensorFlow Lite consente di eseguire modelli di classificazione e segmentazione delle immagini inquadrate dalla scheda OpenMV Cam. Con il supporto di TensorFlow Lite è possibile classificare facilmente regioni di interesse complesse nella visualizzazione e controllare i pin I / O in base a ciò che vede la scheda.

Ovviamente non tutte queste caratteristiche sono utili ai fini della tesi, ma solo quelle inerenti alla cattura e al riconoscimento di immagini.

Per utilizzare la scheda il codice è stato scritto sulla OpenMV Cam Ide:

OpenMV IDE è il principale ambiente di sviluppo integrato da utilizzare con la OpenMV Cam. È dotata di un potente editor di testo, terminale di debug e visualizzatore di frame buffer con visualizzazione dell'istogramma. OpenMV IDE semplifica la programmazione della OpenMV Cam.

**Il MicroPython è un'implementazione software del linguaggio di programmazione Python 3, scritto in C, ottimizzato per l'esecuzione su un microcontrollore.*

3.2 IMPLEMENTAZIONE E USO DEL MODELLO SU BOARD OPENMVCAM H7 PLUS

Scaricato il modello creato e pre-trainato sul Google Colab, occorre caricarlo nella scheda come "model_quantFloat", oltre ad esso vengono caricati due file di testo, uno dove ci sono i nomi di tutte le classi che si chiamerà "labels" e un altro dove sono caricati gli interi relativi alla numerazione delle classi di nome "labels_int". Fatta questa breve introduzione si passa al codice vero e proprio.

```
import sensor, image, time, os, tf, math


sensor.reset()                         # Reset and initialize the sensor.
sensor.set_pixformat(sensor.RGB565)    # Set pixel format to RGB565 (or GRAYSCALE)
sensor.set_framesize(sensor.QVGA)      # Set frame size to QVGA (320x240)
sensor.set_windowing((240, 240))       # Set 240x240 window.
sensor.skip_frames(time=2000)          # Let the camera adjust.


matteo_net = "model_quantFloat.tflite"
labels = [line.rstrip('\n') for line in open("labels.txt")]
labels_int = [line.rstrip('\n') for line in open("labels_int.txt")]
a=[]
b=0
m=[]
c=0
clock = time.clock()
while(True):
    #clock.tick()
    del a[:]
    del m[:]
    for l in range(9):
        clock.tick()
        img = sensor.snapshot()
        for obj in tf.classify(matteo_net, img, min_scale=1.0, scale_mul=0.5, x_overlap=0.0, y_overlap=0.0):
            img.draw_rectangle(obj.rect())
            # This combines the labels and confidence values into a list of tuples
            # and then sorts that list by the confidence values.
            sorted_list_int = sorted(zip(labels_int, obj.output()), key = lambda x: x[1], reverse = True)
            a.append(int(sorted_list_int[0][0]))
            #print(clock.avg())
    #print(clock.avg())
    for i in labels_int:
        c=int(i)
        b=0
        for q in range(9):
            if c==a[q]:
               b=b+1
            else:
               b=b+0
        m.append(b)

    index=m.index(max(m))
    print(labels[index])
```

Anche in questo codice in primis vengono importate le librerie utili per il fine del progetto. Una volta settati i parametri propri della scheda come dimensione di ogni frame a colori catturato dalla camera e la velocità con cui vengono acquisiti i frame.

Si passa poi alla creazione di una stringa con il nome del file relativo al modello caricato, chiamata matteo_net e alla creazione di due liste composte da stringhe, che sono per labels i nomi degli oggetti e per labels_int gli interi relativi alla numerazione delle classi. Poi vengono inizializzate due variabili di tipo int e due array che serviranno per la cattura e la predizione dell'oggetto inquadrato.

Il while permette un'iterazione infinita del codice in modo tale che la scheda smetta di funzionare solo se sconnessa dal pc. All'interno del while ci sono 2 cicli for annidati che prendono 10 immagini catturate dalla fotocamera grazie a:

img = sensor.snapshot()

Per ognuna di queste immagini viene eseguita l'analisi utilizzando il modello precedentemente caricato, infatti:

```
sorted_list_int = sorted(zip(labels_int, obj.output()), key = lambda x: x[1],
reverse = True)
a.append(int(sorted_list_int[0][0]))
```

Ordina sulla lista multidimensionale sorted_list_int le 20 labels in ordine di probabilità della predizione fatta, il primo elemento viene salvato nella lista a. Questa lettura viene ripetuta per 10 frame consecutivi, infatti a conterrà le migliori 10 predizioni fatte sui 10 frame analizzati.

L'obiettivo è quello di eseguire una moda dei valori contenuti nella lista a, per avere una stima più accurata sulla predizione dell'oggetto inquadrato che viene eseguita su ben 10 frame anziché uno soltanto. Per poi essere stampata.

```
for i in labels_int:
        c=int(i)
        b=0
        for q in range(9):
            if c==a[q]:
               b=b+1
            else:
               b=b+0
        m.append(b)

 index=m.index(max(m))
 print(labels[index])
```

I due cicli for riportati infatti permettono di eseguire una scansione della lista a, contare le volte che ogni singolo valore riportato in intero si ripete e valutare quale di questi si ripete più volte. In questo modo viene eseguita la moda statistica della lista a, questo è possibile utilizzando le variabili di appoggio c, b e alla lista m.
Prima di passare al test vero e proprio occorre valutare il tempo necessario per l'acquisizione di un frame e la predizione dell'oggetto inquadrato.

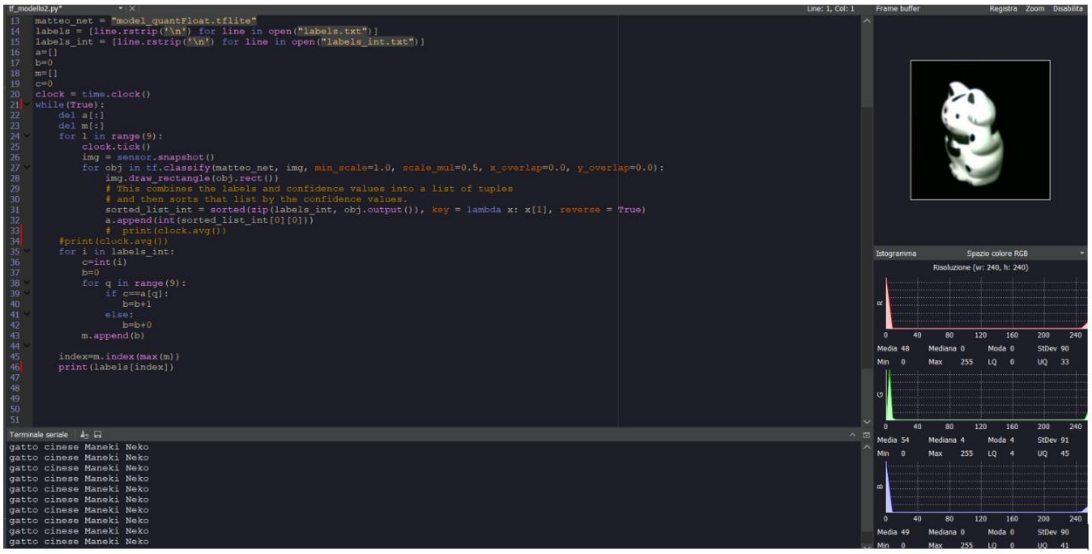
`clock.tick()` inizia il conteggio del tempo sfruttando il clock della scheda, poi il conteggio viene stoppato all'interno del secondo for da `print(clock.avg())` che scrive sul terminale il tempo

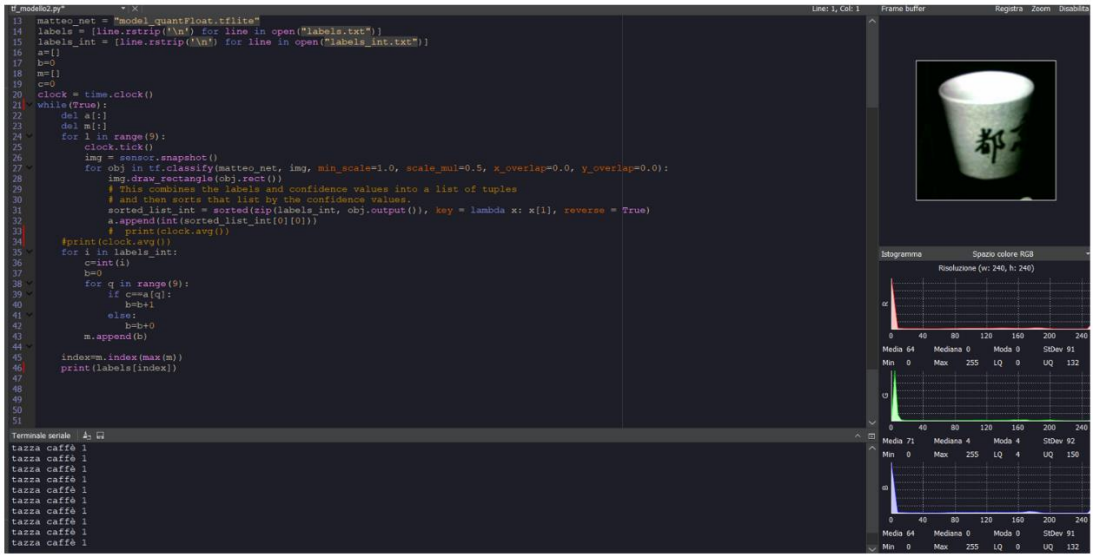
trascorso dall'inizio del conteggio alla fine del secondo for, ossia il tempo necessario per acquisire l'immagine e farne la predizione.
I valori che si vedono riportati sul terminale dell'ide sono in ms e si aggirano tutti intorno ai **105 ms**. Il valore medio del tempo necessario per l'acquisizione e la predizione di 10 immagini è di **105.39426 ms**.

3.3 TEST IN TEMPO REALE

I test in tempo reale fatti con la board sono stati compiuti in primo luogo inquadrando con la fotocamera della scheda gli oggetti appartenenti al dataset COIL-20. Gli oggetti sono stati ripresi mediante le immagini mostrate dallo schermo di un cellulare. Ciò è stato fatto per due motivi: la particolarità degli oggetti presenti nel dataset a cui segue una scarsa reperibilità di alcuni di essi e la qualità del dataset, che fornisce per ogni classe, foto dello stesso oggetto con angolature diverse e uno sfondo nero. Perciò la rete non è stata allenata con una grande varietà di tipi di oggetto per ogni classe e a questo segue una scarsa generalizzazione, inoltre è stata allenata a riconoscere immagini il cui sfondo è nero, mentre nella realtà sarà presente molto più "rumore" rispetto a quello a cui è stata abituata la rete. Gli esperimenti compiuti inquadrando semplicemente le immagini del dataset dal cellulare, hanno portato risultati molto positivi, che si possono vedere dalle figure seguenti. In secondo luogo, i test sono stati compiuti solo su alcuni oggetti veri, per via della reperibilità e soprattutto dal fatto che appartengono a più classi e quindi permettono alla rete di generalizzare meglio.

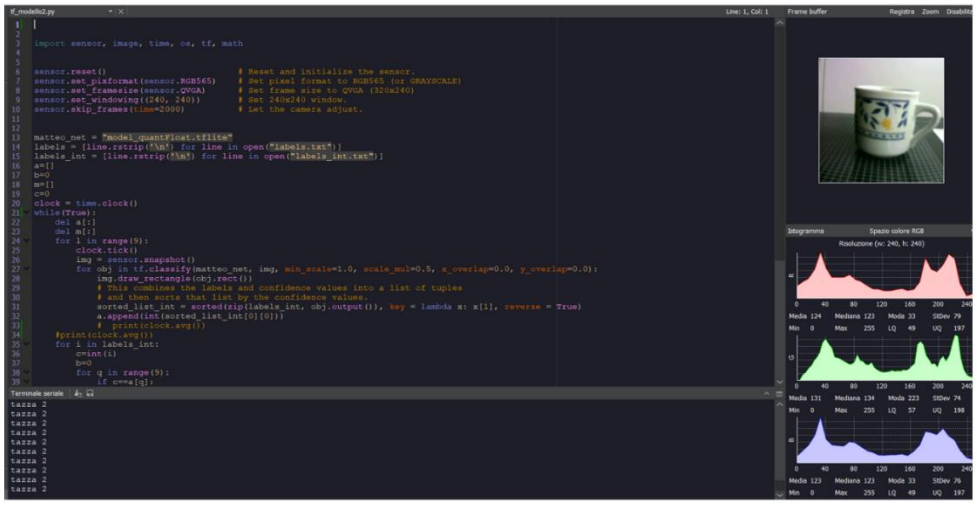
Essendoci ben tre classi con modellini di auto detti “macchinine”, la rete giustamente tende a generalizzare e su 10 print tre sono riferiti alla classe “macchinina 2” anche se quella inquadrata è la prima classe con modellino di auto, chiamata quindi “macchinina”.

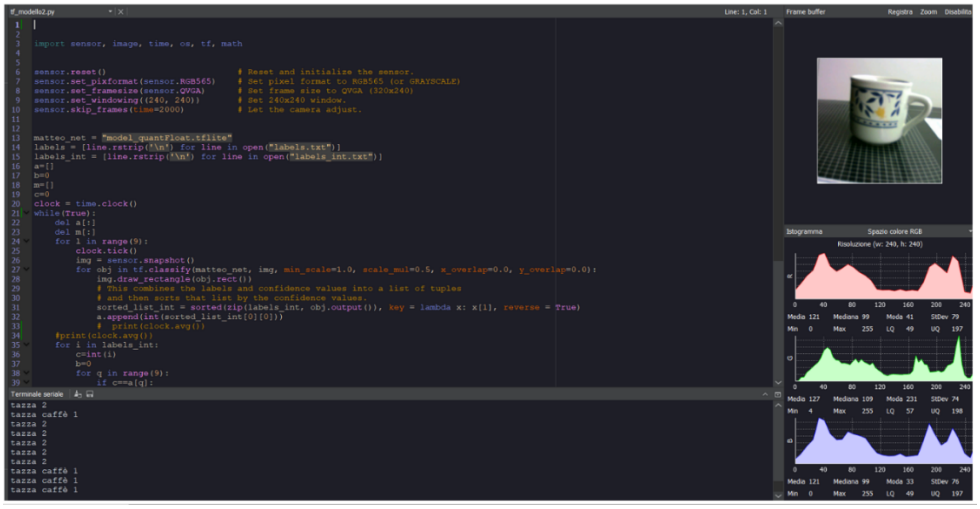


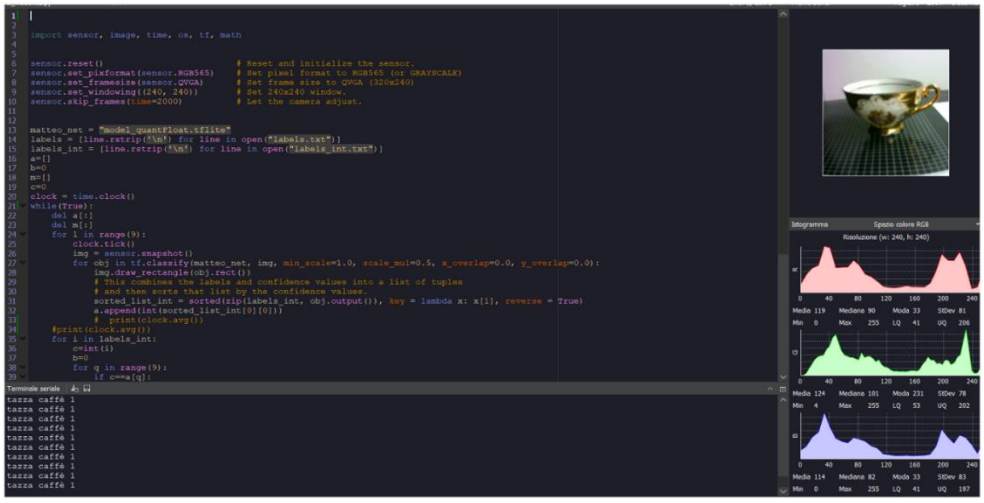


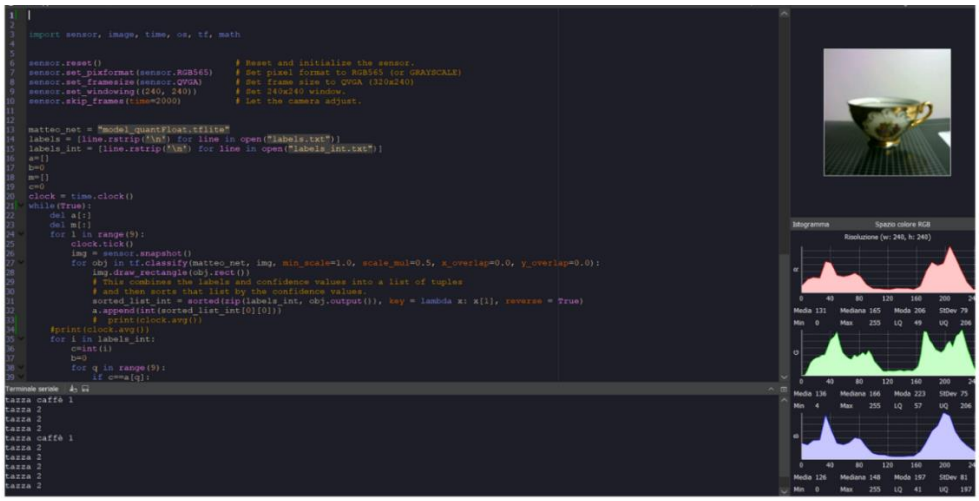
Le prove riportate, sono soltanto alcune di quelle effettuate sui 20 oggetti, come si può notare dal terminale dell'ide, la scheda che monta il modello pre-trainato riesce a riconoscere molto bene gli oggetti nelle immagini inquadrate.

L'ultima prova effettuata cerca di valutare se la rete riesce a generalizzare sugli oggetti definiti in più classi come i modellini di auto e le tazze/tazzine di caffè, quindi sono stati inquadrati oggetti veri e propri appartenenti a queste due categorie.


tf_modello2.py
Line: 1, Col: 1
Frame buffer
Registra
Zoom
Disabilita
import sensor, image, time, os, tf, math
sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.set_windowing((240, 240))
sensor.skip_frames(time=2000)
matteo_net = "model_quantfloat.tflite"
labels = [line.rstrip('\n') for line in open("labels.txt")]
labels_int = [line.rstrip('\n') for line in open("labels_int.txt")]
clock = time.clock()
while(True):
img = sensor.snapshot()
for obj in tf.classify(matteo_net, img, min_scale=1.0, scale_mul=0.5, x_overlap=0.0, y_overlap=0.0):
img.draw_rectangle(obj.rect())
sorted_list_int = sorted(zip(labels_int, obj.output()), key = lambda x: x[1], reverse = True)
a.append(int(sorted_list_int[0][0]))
for i in labels_int:
for q in range(9):
Terminale seriale
tazza 2
tazza 2
tazza 2
tazza 2
tazza 2
tazza 2
tazza 2
tazza 2
tazza 2
Istogramma
Spazio colore RGB
Risoluzione (w: 240, h: 240)
Media 124
Mediana 123
Moda 33
StDev 79
Min 0
Max 255
LQ 49
UQ 197
Media 131
Mediana 134
Moda 223
StDev 74
Min 0
Max 255
LQ 57
UQ 198
Media 123
Mediana 123
Moda 33
StDev 76
Min 0
Max 255
LQ 49
UQ 197


tf_modello2.py
Line: 1, Col: 1
Frame buffer
Registra
Zoom
Disabilita
import sensor, image, time, os, tf, math
sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.set_windowing((240, 240))
sensor.skip_frames(time=2000)
matteo_net = "model_quantfloat.tflite"
labels = [line.rstrip('\n') for line in open("labels.txt")]
labels_int = [line.rstrip('\n') for line in open("labels_int.txt")]
clock = time.clock()
while(True):
img = sensor.snapshot()
for obj in tf.classify(matteo_net, img, min_scale=1.0, scale_mul=0.5, x_overlap=0.0, y_overlap=0.0):
img.draw_rectangle(obj.rect())
sorted_list_int = sorted(zip(labels_int, obj.output()), key = lambda x: x[1], reverse = True)
a.append(int(sorted_list_int[0][0]))
for i in labels_int:
for q in range(9):
Terminale seriale
tazza 2
tazza caffè 1
tazza 2
tazza 2
tazza 2
tazza 2
tazza 2
tazza caffè 1
tazza caffè 1
tazza caffè 1
Istogramma
Spazio colore RGB
Risoluzione (w: 240, h: 240)
Media 121
Mediana 99
Moda 41
StDev 79
Min 0
Max 255
LQ 49
UQ 197
Media 127
Mediana 109
Moda 231
StDev 74
Min 4
Max 255
LQ 57
UQ 198
Media 121
Mediana 99
Moda 33
StDev 76
Min 0
Max 255
LQ 49
UQ 197


import sensor, image, time, os, tf, math
sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.set_windowing((240, 240))
sensor.skip_frames(time=2000)
matteo_net = "model_quantfloat.tflite"
labels = [line.rstrip('\n') for line in open("labels.txt")]
labels_int = [line.rstrip('\n') for line in open("labels_int.txt")]
clock = time.clock()
while(True):
img = sensor.snapshot()
for obj in tf.classify(matteo_net, img, min_scale=1.0, scale_mul=0.5, x_overlap=0.0, y_overlap=0.0):
img.draw_rectangle(obj.rect())
sorted_list_int = sorted(zip(labels_int, obj.output()), key = lambda x: x[1], reverse = True)
a.append(int(sorted_list_int[0][0]))
for i in labels_int:
for q in range(9):
Terminale seriale
tazza caffè 1
tazza caffè 1
tazza caffè 1
tazza caffè 1
tazza caffè 1
tazza caffè 1
tazza caffè 1
tazza caffè 1
tazza caffè 1
tazza caffè 1
Istogramma
Spazio colore RGB
Risoluzione (w: 240, h: 240)
Media 119
Mediana 90
Moda 33
StDev 81
Min 0
Max 255
LQ 41
UQ 206
Media 124
Mediana 101
Moda 231
StDev 78
Min 4
Max 255
LQ 53
UQ 202
Media 114
Mediana 82
Moda 33
StDev 83
Min 0
Max 255
LQ 41
UQ 197



I test fatti su oggetti reali appartenenti alle classi modellini di auto e tazze/tazzine di caffè, producono buoni risultati in quanto la rete generalizza le immagini viste come una delle classi che le descrive.

Conclusioni

Si è partiti dallo studio teorico sulle reti neurali, dal funzionamento di un singolo neurone alla background propagation e tutto ciò che ne consegue. Questo primo studio ha permesso l'implementazione in linguaggio Python di una rete neurale convoluzionale e del dataset Coil 20 con cui è stata poi allenata. Fatti tutti i test per verificarne il funzionamento, il modello della rete è stato quantizzato e convertito in formato tflite, rendendolo più leggero e veloce per l'utilizzo su scheda in tempo reale. Scritto il codice in Python sull'ide della scheda OpenMVCam H7 Plus e caricato su essa il modello pre-trainato, si è passati ai test in tempo reale. I primi test sono stati compiuti mostrando alla camera della scheda immagini appartenenti al dataset Coil 20. In questi test si è riscontrato un comportamento positivo della rete, in quanto il terminale dell'ide stampa le corrette etichette per ogni immagine presa in esame. Per ultimi sono stati eseguiti test su oggetti veri, come due diverse tazzine di caffè e un modellino di automobile. Vengono scelti questi oggetti in quanto nel dataset sono presenti più classi che li descrivono, permettendo quindi che la rete riesca a generalizzare meglio. Come si era ipotizzato, i test eseguiti sugli oggetti dimostrano che la rete generalizza, in quanto restituisce i nomi delle diverse classi che descrivono il tipo di oggetto inquadrato. Inoltre è opportuno sottolineare la scarsità di elementi forniti dal dataset, che ha solo 20 classi e per ognuna 72 foto dello stesso oggetto ruotato. Il vantaggio di questo dataset è proprio la sua semplicità a cui segue l'utilizzo di una rete con soli 4 layer leggera e veloce, adatta per il funzionamento in tempo reale. Lo svantaggio è appunto la scarsità delle informazioni fornite dal dataset, che non permettono alla rete una buona generalizzazione per ogni oggetto analizzato.

BIBLIOGRAFIA

Tom Hope, Yehezkel S. Resheff, Itay Lieder - Learning TensorFlow. A Guide to building Deep Learning Systems-O'Reilly (2017)

SITOGRAFIA

http://neuralnetworksanddeeplearning.com/chap1.html

https://www.tensorflow.org/tutorials/images/cnn?hl=it

https://docs.python.org/3/library/os.html

https://machinelearningmastery.com/train-test-split-for-evaluating-machine-learning-algorithms/

https://stackoverflow.com/questions/59813201/read-multiple-images-and-labels-in-python

https://pythonprogramming.net/loading-custom-data-deep-learning-python-tensorflow-keras/

https://www.developersmaggioli.it/blog/reti-convoluzionali/

https://qastack.it/programming/4752626/epoch-vs-iteration-when-training-neural-networks

https://stackoverflow.com/questions/63483337/how-can-i-plot-training-accuracy-training-loss-with-respect-to-epochs-in-tensor

https://gombru.github.io/2018/05/23/cross_entropy_loss/#:~:text=is%20available%20here-,Binary%20Cross%2DEntropy%20Loss,plus%20a%20Cross%2DEntropy%20loss.&text=It's%20called%20Binary%20Cross%2DEntropy,in%20C%20%2C%20as%20explained%20above.

https://neptune.ai/blog/keras-loss-functions

https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

https://iaml.it/blog/deep-learning-ottimizzazione

https://towardsdatascience.com/how-does-back-propagation-in-artificial-neural-networks-work-c7cad873ea7

Overview — MicroPython 1.12 documentation

https://openmv.io/products/openmv-cam-h7-plus

VIDEOGRAFIA

https://www.youtube.com/watch?v=WvoLTXIjBYU

https://www.youtube.com/watch?v=wQ8BIBpya2k

https://www.youtube.com/watch?v=IHZwWFHWa-w&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=2

https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=1

https://www.youtube.com/watch?v=44U8jJxaNp8

https://www.youtube.com/watch?v=jTLUxUEQCPs&t=37s

https://www.youtube.com/watch?v=ugomO_BZ44g

https://www.youtube.com/watch?v=mdKjMPmcWjY