UNIVERSITÀ
POLITECNICA
DELLE MARCHE

FACOLTÀ DI INGEGNERIA
Corso di Laurea Triennale in Ingegneria Elettronica

TESI DI LAUREA:

Implementazione di una CNN per la classificazione di immagini appartenenti al dataset CIFAR-10 su piattaforma embedded OpenMV Cam

Implementation of a CNN for the classification of images belonging to the CIFAR-10 dataset on the OpenMV Cam embedded platform

Candidato:
Tomas Rocchetti

Relatore:
Chiar.mo Prof. Claudio Turchetti

Correlatore:
Prof. Laura Falaschetti

Anno Accademico 2019-2020

Indice

1 Intelligenza Artificiale **4**
1.1 Machine Learning 4
1.1.1 Apprendimento 4
1.2 Deep Learning 5

2 Elementi di una rete neurale **6**
2.1 Neuroni artificiali 6
2.1.1 Input e output dei neuroni artificiali e pesi 6
2.1.2 Funzione di attivazione 8
2.2 Reti neurali 10
2.3 Apprendimento 10

3 Convolutional Neural Networks **11**
3.1 Architettura di una CNN 11
3.1.1 Livello di Input 12
3.1.2 Livello di convoluzione 12
3.1.3 Livello ReLU 13
3.1.4 Livello di Pooling 14
3.1.5 Livello di Dropout 14
3.1.6 Livello di Normalizzazione 14
3.1.7 Livello Fully Connected 16

4 Applicazione **17**
4.1 Keras 17
4.2 Sviluppo di una CNN 17
4.2.1 Il Dataset 18
4.2.2 One Hot encoding 19
4.2.3 Preparazoine del dataset 20
4.2.4 Plotting dei risultati 20
4.2.5 Modello elementare 21
4.2.6 Funzione di Test 21
4.2.7 Miglioramento del modello 22
4.2.8 Overfitting 24
4.2.9 Aggiunta del Dropout 24
4.2.10 Utilizzo di Data Augmentation 25
4.2.11 Batch Normalization 28
4.2.12 Learning Rate dinamico 30
4.2.13 Modello ottimo 31

4.2.14 Confusion Matrix 32
4.2.15 Esportazione del modello 33
4.3 Porting su STM32 33
4.3.1 OpenMV Cam H7 33
4.3.2 Conversione e compressione del modello 34
4.3.3 Scaling del modello 35
4.3.4 Test finale 36

5 Conclusioni **38**

Introduzione

In questa tesi viene descritta la progettazione di un sistema embedded in grado di catturare immagini per mezzo di un sensore CMOS, ed analizzarle ordinando i soggetti in base alla loro classe di appartenenza. Per la classificazione degli scatti verrà utilizzata una rete neurale convoluzionale: uno strumento molto potente, ampiamente utilizzato negli algoritmi di intelligenza artificiale per il riconoscimento di immagini o pattern. In un primo momento si è scelto di sviluppare una rete neurale artificiale con performance massime, dopodiché si è revisionata così da far fronte ai principali problemi che potrebbero sorgere utilizzando algoritmi di tale complessità in dispositivi embedded, caratterizzati da performance decisamente ridotte.

Nonostante esistano, sparsi per il globo, enormi server capaci di elaborare moli di dati impressionanti, spesso si preferisce implementare algoritmi di intelligenza artificiale su sistemi embedded. Tali scelte sono dovute principalmente a due fattori: decentralizzazione della potenza di calcolo e indipendenza dalla connessione di rete.

Dimensionare opportunamente un sistema di elaborazione centralizzato non è facile: l'utilizzo delle risorse è molto variabile e dipende al numero di clienti che utilizzano un determinato servizio. Un servizio offerto da un sistema embedded è invece generalmente destinato ad un solo cliente e risulta quindi decisamente più semplice dimensionare il sistema.

Altro aspetto da non sottovalutare è l'indipendenza da una connessione di rete: è noto che alcune applicazioni necessitino di funzionare in luoghi sprovvisti di copertura o, a volte, abbiano bisogno di ritardi inferiori nell'elaborazione dei dati rispetto a quelli che una normale connessione cellulare è in grado di offrire; è sufficiente pensare ad un sistema automatico di frenata di emergenza.

Dal capitolo seguente si inizierà a discutere del significato di intelligenza artificiale e verranno analizzati in maniera gerarchica gli elementi necessari allo sviluppo dell'applicazione finale.

Capitolo 1

Intelligenza Artificiale

L'intelligenza artificiale è la branca della scienza che si pone come obiettivo quello di permettere alle macchine di eseguire compiti in maniera intelligente, cercando di imitare il più fedelmente possibile il modo in cui lo farebbe un essere umano. Il termine AI, però, non rappresenta una particolare soluzione atta a risolvere un determinato problema: esistono numerose tecniche differenti, una ampiamente utilizzata a partire dagli anni '80 è quella del machine learning.

1.1 Machine Learning

Alcuni problemi di intelligenza artificiale non sono affatto semplici da risolvere con algoritmi classici, come ad esempio iterativi o condizionali. Il machine learning, o apprendimento automatico, è una branca dell'intelligenza artificiale che nasce proprio per creare un approccio alternativo. Il suo scopo è quello di permettere alle macchine di apprendere da una serie di dati, esattamente come fanno gli esseri umani.

Esistono numerosi algoritmi di machine learning, in particolare, essi si suddividono in tre principali gruppi: quelli di learning supervisionato, quelli di learning non supervisionato e quelli di learning per rinforzo. Si differenziano gli uni dagli altri per il modo in cui apprendono informazioni per fare previsioni.

1.1.1 Apprendimento

Tom M. Mitchell nel libro *"Machine Learning"* ha definito il significato di apprendimento automatico: spiegò che un software è in grado di apprendere se ad un compito svolto ne consegue un miglioramento delle prestazioni nelle successive iterazioni.

L'obiettivo dell'apprendimento automatico non è altro che permettere ad una macchina di portare a termine compiti nuovi, mai affrontati prima da essa.

Verranno ora descritti brevemente i tre macro gruppi di algoritmi di learning:

Machine learning supervisionato

L'algoritmo apprende da una libreria di dati già strutturata ed etichettata, è dunque fondamentale l'intervento umano in questa tipologia di learning. L'al-

goritmo si limita a generare delle regole che gli permettano di associare ad un determinato input l'output corretto.

Machine learning non supervisionato

In questa tipologia di learning la macchina crea degli schemi associati agli input senza che essi vengano precedentemente etichettati, come invece accade nel caso del learning supervisionato.

Machine learning per rinforzo

L'algoritmo opera in un ambiente dinamico, non sa se quello che sta facendo è giusto, ma soltanto se l'obiettivo a cui puntava è stato raggiunto.

La scelta fra i vari tipi di learning varia molto in base al settore al quale si vuole applicare l'algoritmo.

Nel caso di classificazione di immagini generalmente vengono usati algoritmi supervisionati; tali algoritmi apprendono determinate caratteristiche tipiche di un soggetto da una serie etichettata di immagini, per poi fare delle predizioni su nuove immagini, sfruttando le caratteristiche apprese.

1.2 Deep Learning

Si è notato che i primi algoritmi di machine learning entravano in difficoltà con richieste che potevano risultare piuttosto semplici per una mente umana. Il problema non era però nel concetto alla base del quale queste tecnologie venivano sviluppate, bensì nella scarsa complessità con la quale venivano implementate: era dunque necessario perfezionare l'approccio utilizzato. Con l'avvento di calcolatori sempre più potenti diventò possibile aumentare la complessità, nacque così il *Deep Learning*.

Con il termine deep learning si intende quel settore dell'apprendimento automatico che tenta di risolvere un problema operando su più livelli. Il deep learning, infatti, rappresenta il problema con una gerarchia di caratteristiche, in cui quelle poste nei livelli più bassi dipendono da quelle poste nei livelli superiori.

La massima espressione di deep learning si ha probabilmente con le deep neural networks (reti neurali profonde), esse sono costituite da diversi livelli, ognuno dei quali calcola dei valori per il livello successivo.

Gli algoritmi di deep learning hanno avuto risultati eccezionali negli ultimi anni grazie, chiaramente, all'incremento della capacità di elaborazione delle macchine moderne.

Capitolo 2

Elementi di una rete neurale

Come anticipato nel capitolo precedente le reti neurali profonde hanno avuto un enorme successo negli ultimi anni. Per comprenderle è necessario dunque capire la struttura della sua unità funzionale: il neurone artificiale. Ovviamente sarà anche fondamentale capire come i neuroni artificiali operano fra di loro.

Il neurone artificiale si ispira al neurone biologico, è dunque utile chiarire a grandi linee il principio di funzionamento del neurone biologico al fine di poter comprendere il funzionamento di quello artificiale: la caratteristica principale del neurone è quella di generare un potenziale elettrico in output nel momento in cui l'attività elettrica del corpo del neurone supera una certa soglia.

Gli input che ricevono i neuroni, e che in seguito verranno sommati nel corpo, sono un'insieme di fibre connesse agli output dei neuroni precedenti, dai quali ricevono segnali. L'interconnessione tra output e input di diversi neuroni si chiama sinapsi e ha la capacità di modulare l'impulso elettrico.

2.1 Neuroni artificiali

I neuroni artificiali sono le unità elementari che compongono le reti neurali artificiali. Essi emulano alcuni principi di funzionamento dei neuroni biologici: ricevono valori in input dai neuroni precedenti, sommano i valori, confrontano il risultato con una soglia opportuna e restituiscono l'output.

2.1.1 Input e output dei neuroni artificiali e pesi

Un neurone prende in ingresso diversi input, li somma e genera un segnale di output a seguito del confronto con un valore di soglia. Come è possibile vedere in figura 2.2, ad ogni input è associato un peso che determinerà quanto un certo ingresso influenzerà la somma.

Ad ogni ingresso deve perciò essere associato un valore che va a stabilire quanto ogni input contribuirà al valore finale della somma. In sintesi ogni input andrà a influenzare in maniera più o meno ingente la somma, e quindi il superamento del valore di soglia.

Il concetto di peso degli input di un neurone artificiale corrisponde esattamente al concetto matematico di peso: un valore numerico da moltiplicare per l'input desiderato.

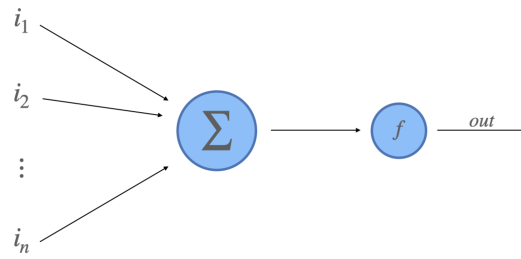
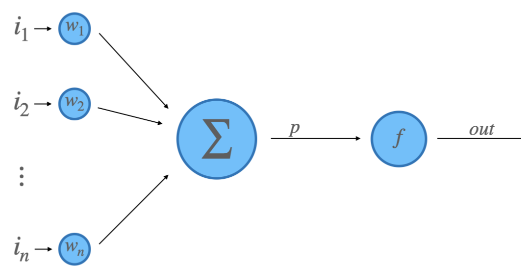


Figura 2.1: Struttura di un neurone artificiale



2.1.2

Figura 2.2: Struttura di un neurone artificiale con pesi

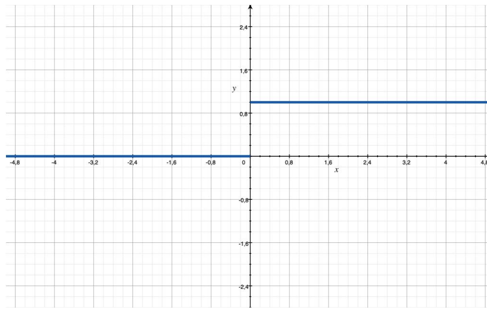
2.1.2 Funzione di attivazione

I neuroni artificiali, operando su delle somme, risulterebbero operatori lineari. Senza l'utilizzo di una funzione non lineare, un insieme di neuroni si comporterebbe come un singolo neurone perché, sommando più elementi lineari, si ottiene un singolo elemento lineare. È possibile rendere il sistema non lineare facendo attraversare all'output una funzione non lineare. Di seguito verranno elencate le più note:

Gradino unitario

$$f(x) = \begin{cases} 0 \text{ if } x \leq 0 \\ 1 \text{ if } x > 0 \end{cases} \tag{2.1}$$

Se il valore in ingresso è maggiore di 0 restituisce 1, se è minore di 0 restituisce 0.

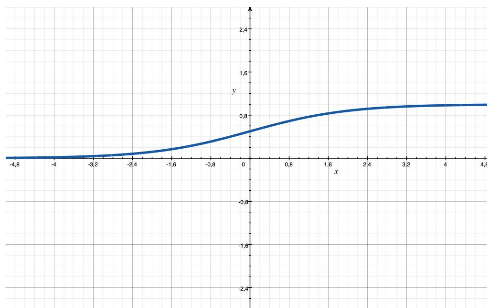


Questa funzione di attivazione viene utilizzata raramente, in quanto fa perdere le informazioni qualitative dell'input.

Sigmoide

$$f(x) = \frac{e^x}{1 + e^x} \tag{2.2}$$

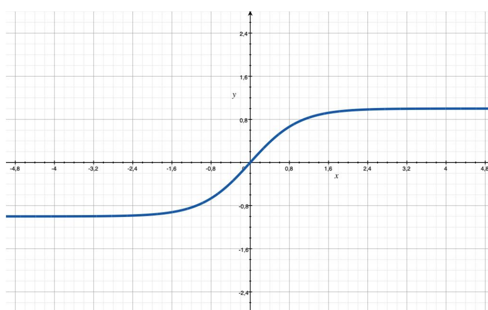
La sigmoide è una funzione che restituisce valori compresi nell'intervallo [0 : 1], per questo motivo spesso viene utilizzata nei layer di output.



Tangente iperbolica

$$f(x) = tanh(x) \tag{2.3}$$

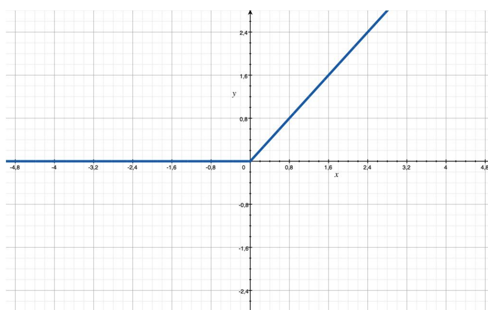
La differenza tra sigmoide e tangente iperbolica è che quest'ultima è centrata in 0 e restituisce valori compresi in $[-1 : 1]$.



ReLU

$$f(x) = max(0, x) \tag{2.4}$$

È una delle funzioni di attivazione più utilizzate nelle deep neural networks, poiché numerosi vantaggi rispetto alla sigmoide o alla tangente iperbolica: In particolare la ridotta probabilità della scomparsa del gradiente. La scompara del gradiente è un problema che si verifica utilizzando per il training algoritmi di discesa del gradiente. Lo scopo di questi algoritmi è quello di minimizzare l'errore nelle predizioni.



Softmax

Le funzioni precedentemente descritte si prestano bene a problemi di classificazione per due classi, ma nel momento in cui le classi da classificare diventano numerose si utilizza la funzione Softmax. Tale funzione permette di comprimere un vettore n-dimensionale di z valori in un vettore n-dimensionale in cui ogni elemento è compreso nell'intervallo $[0 : 1]$ e indica la probabilità che un determinato elemento ha di apparire nel vettore.

2.2 Reti neurali

Dopo aver descritto i vari elementi di un neurone artificiale si passa ora a descrivere il modo in cui tali neuroni collaborano tra di loro.

L'interconnessione di numerosi neuroni artificiali costituisce una rete neurale. Generalmente le connessioni fra i vari neuroni non sono statiche, bensì dinamiche, e variano in funzione degli input che vengono dati in ingresso alla rete. I neuroni vengono raggruppati in layer della rete neurale: sono presenti un layer di ingresso, un layer di uscita e, interposti tra questi due, uno o più layer intermedi (o nascosti). Le informazioni vengono propagate nella rete neurale da layer a layer, con il verso che varia a seconda dell'algoritmo di apprendimento utilizzato.

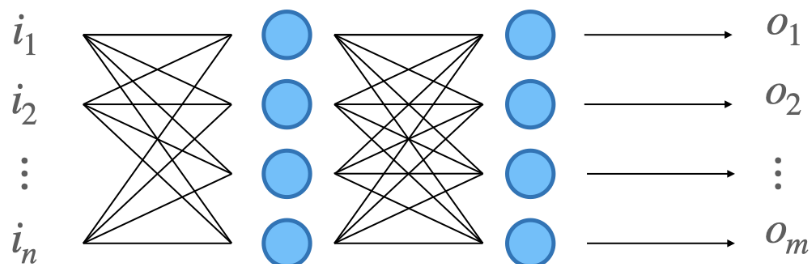


Figura 2.3: Interconnessioni tra neuroni.

2.3 Apprendimento

Una rete neurale, al momento della sua creazione, non possiede alcuna capacità di classificazione. Affinché la rete possa sviluppare delle capacità di riconoscimento di forme o oggetti è necessario addestrarla.

Il processo di addestramento (o training) è volto a trovare i valori dei pesi di ogni neurone in modo tale che l'accuratezza della rete sia massima.

È inoltre opportuno che il processo di training vada a modificare i pesi della rete sulla base di esempi presenti all'interno del dataset di training, mantenendo però una buona accuratezza anche nel momento in cui vengano utilizzati elementi di ingresso non inclusi nel dataset di training.

Capitolo 3

Convolutional Neural Networks

Le CNN o Convolutional Neural Networks sono reti ampiamente utilizzate nel riconoscimento di immagini, in particolare la loro scesa al successo è partita dall'anno 2012, quando Alex Krizhevsky grazie alla sua rete AlexNet ha vinto la competizione mondiale di computer vision.

Sfruttando una rete neurale convoluzionale, quindi, una macchina è in grado di riconoscere con una certa probabilità l'oggetto raffigurato da una determinata immagine.

Chiaramente le CNN non sono sufficienti a permettere ad una macchina di riconoscere un qualsiasi tipo di oggetto in un'immagine, ma riconosceranno soltanto classi di oggetti presenti nel dataset su cui la rete verrà addestrata. Ad esempio, se ad una rete neurale addestrata per classificare diverse specie di piante dessimo in ingresso la foto di un volto questa non sarebbe in grado di restituire risultati utili.

3.1 Architettura di una CNN

Un esempio di struttura di rete neurale convoluzionale può essere quello di una rete formata dai seguenti layer:

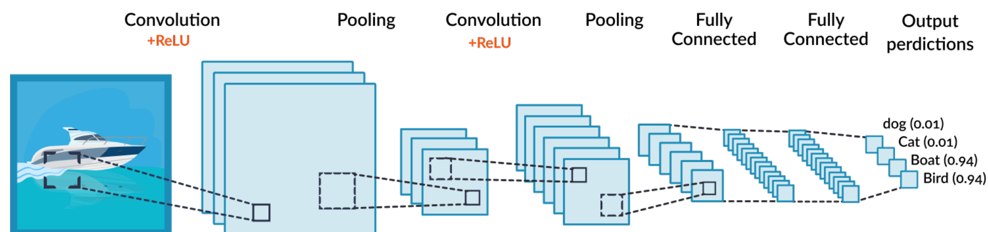


Figura 3.1: Esempio dell'architettura delle CNN.

Analizzeremo ora ciascuno dei principali livelli (o layer) di una CNN.

3.1.1 Livello di Input

Il livello di input non è altro che l'immagine in ingresso da classificare. Una macchina vede tale immagine come insieme di numeri, in particolare come un tensore a 3 dimensioni, in cui due dimensioni sono la dimensione in pixels di altezza e larghezza dell'immagine, mentre la terza dimensione (3 se supponiamo di lavorare con immagini RGB) rappresenta i tre canali di colore. Ogni elemento del tensore sarà un valore numerico che va da 0 a 255.

3.1.2 Livello di convoluzione

Il livello di convoluzione ci permette di estrarre dalle immagini delle caratteristiche, come ad esempio sagome, forme o spigoli. Queste caratteristiche verranno denominate features dell'immagine. Più sono numerosi i layer di convoluzione e più sarà alta la complessità delle features estraibili.

Elemento fondamentale del layer di convoluzione è il filtro, una piccola matrice (spesso 3x3) che rappresenta una feature che il livello convoluzionale vuole estrarre. Nei livelli iniziali il filtro rappresenta caratteristiche di basso livello, come linee, bordi o simili, man mano che si avanza con i livelli, invece, permetterà di identificare elementi sempre più complessi.

Il filtro viene applicato all'immagine partendo dall'analisi del campo ricettivo, ovvero la porzione di immagine di pari dimensioni del filtro che esso evidenzia. Generalmente si parte dal punto in alto a sinistra. Il risultato della convoluzione sarà il prodotto scalare del filtro per il blocco della matrice evidenziato, che porterà così ad ottenere una matrice di dimensione 1×1.

Questa operazione viene ripetuta per tutti i vari blocchi che l'immagine può contenere, facendo scorrere il filtro sull'immagine di un determinato passo.

Ipotizzando di avere un'immagine iniziale 7×7, un filtro 3×3 ed utilizzando un passo pari a 1, otterremo un'immagine convoluta di dimensione 5×5

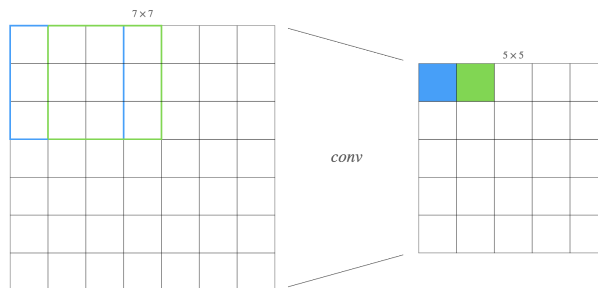


Figura 3.2: Convoluzione con passo 1.

Risulta evidente che l'activation map, ovvero la matrice che si ottiene in uscita dal layer di convoluzione, avrà dimensioni ridotte rispetto alla dimensione della matrice in ingresso. Inoltre le activation map saranno tante quanti sono i

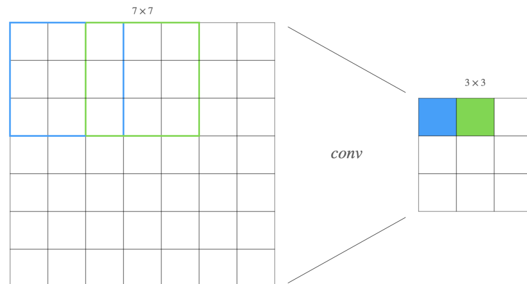


Figura 3.3: Convoluzione con passo 2.

filtri utilizzati. Solitamente si vuole evitare questa riduzione di dimensioni: si vuole cercare di mantenere i primi livelli della dimensione più grande possibile, per far sì che contengano tutte le informazioni necessarie per estrarre features di basso livello, che altrimenti andrebbero perse e sarebbero impossibili da ricavare nei layer successivi.

Una valida tecnica per ovviare a questo problema è lo zero-padding, che consiste nell'aggiungere alla matrice di ingresso una cornice di zeri per aumentarne la dimensione e conseguentemente aumentare la dimensione dell'activation map all'uscita del layer.

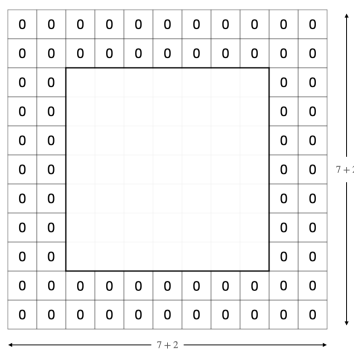


Figura 3.4: Esempio di zero-padding.

3.1.3 Livello ReLU

Il layer ReLU generalmente viene posto in uscita ad un layer di convoluzione, quindi l'output di convoluzione è l'input del layer ReLU.

Questo livello rappresenta una funzione non lineare, utile per introdurre una non linearità in un sistema che svolge soltanto operazioni lineari (convoluzione).

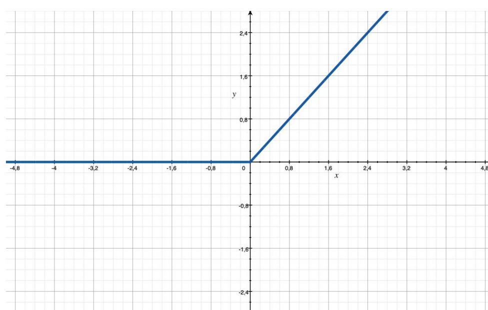
Si è visto che, introducendo una funzione non lineare, è possibile aumentare notevolmente la velocità di apprendimento della rete senza intaccare significativamente le prestazioni in termini di accuratezza.

Ciò accade perchè se un'intera deep neural network fosse lineare si comporterebbe come una rete ad un solo strato, infatti somme di elementi lineari possono essere compresse in un singolo elemento lineare.

Il layer ReLU applica la funzione:

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

a tutti i valori in input, andando così ad annullare tutti i valori negativi.



3.1.4 Livello di Pooling

A seguito del Livello ReLU spesso è comodo utilizzare il livello di pooling per ridurre dimensionalmente l'immagine. Esistono diversi tipi di pooling, tra cui i più comuni sono Average Pooling e Max Pooling.

Questo layer utilizza un filtro (spesso 2×2) e un passo della stessa dimensione del filtro (quindi un passo pari a 2). Nel caso di Max Pooling viene scelto soltanto il valore massimo per ogni campo ricettivo su cui transita il filtro, mentre nel caso di Average Pooling viene scelto il valore medio del campo ricettivo su cui transita il filtro.

3.1.5 Livello di Dropout

Il dropout è una tecnica di regolarizzazione che consiste nell'escludere casualmente alcuni neuroni e i relativi pesi associati durante un'epoca di training, per poi venire reinseriti ed escluderne altri. Cancellando diversi neuroni nelle varie epoche il risultato viene ad essere pari a quello di addestrare differenti reti neurali durante il training. La rete finale avrà le caratteristiche medie delle reti prive di alcuni neuroni.

3.1.6 Livello di Normalizzazione

L'allenamento delle reti neurali, spesso, viene rallentato da un fenomeno chiamato *internal covariate shift*. Con questo termine ci si riferisce proprio alla situazione in cui ad ogni strato della rete, ad ogni iterazione, viene proposto

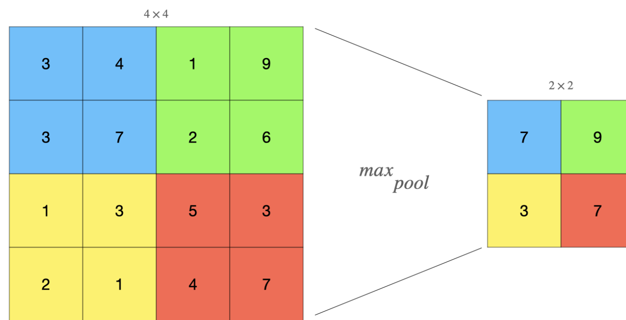


Figura 3.5: Esempio di max-pooling.

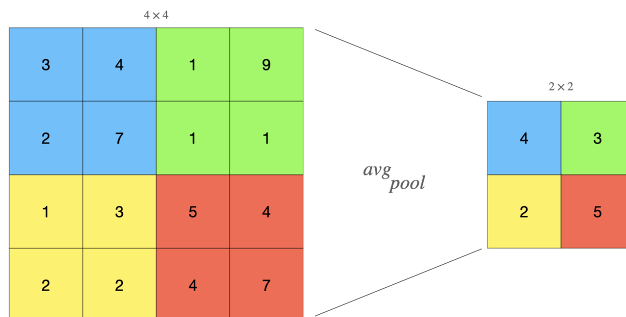


Figura 3.6: Esempio ddi avg-pooling.

un input completamente diverso rispetto all'input precedente. Tutto ciò genera un problema in quanto ogni strato della rete deve riadattarsi ad una nuova distribuzione dei valori di ingresso.

Questo tipo di layer permette dunque di normalizzare gli input ad ogni substrato della rete.

3.1.7 Livello Fully Connected

Il livello fully connected è l'ultimo layer di una CNN: non fa altro che prendere in ingresso il volume prodotto dall'ultima convoluzione e generare un vettore di dimensioni pari al numero di classi da stimare. Ogni elemento di tale array conterrà la probabilità che un certo input appartenga ad una determinata classe.

Questo layer permette di analizzare i dati provenienti dal livello precedente e determinare quali features sono maggiormente correlate ad una determinata classe.

Ad esempio, se alla rete viene proposta l'immagine di un'automobile, si avranno valori molto alti in corrispondenza delle features corrispondenti a fanali, specchietti o ruote; il livello fully connected utilizza questi valori per determinare la classe di appartenenza.

Capitolo 4

Applicazione

Il problema che si tenterà di affrontare è quello di sviluppare un modello per la classificazione di immagini, da utilizzare in un sistema embedded, sfruttando l'API Keras. In particolare verrà utilizzata la board OpenMV Cam H7 Plus, equipaggiata con una CPU ARM STM32. Lo sviluppo del progetto inizierà con la creazione di un modello di rete neurale in Keras, che verrà addestrata sfruttando le enormi capacità di calcolo dei cluster di GPU di Google Colaboratory; raggiunta una buona accuratezza nel riconoscimento delle immagini, il modello verrà esportato per poter essere caricato sulla OpenMV Cam.

4.1 Keras

Keras è un'API di deep learning basata su Tensorflow e scritta in Python: è stata sviluppata con l'intenzione di creare uno strumento di sperimentazione semplice e veloce. Keras permette di utilizzare tutte le caratteristiche di Tensorflow, in particolare la scalabilità e la possibilità di essere multipiattaforma. È dunque possibile eseguire Keras su grandi server, sfruttando interamente le potenzialità di calcolo di grandi cluster di CPU, GPU e TPU, per poi esportare i risultati su dispositivi mobili.

Le strutture di base di Keras sono i livelli e i modelli. Il modello più semplice è il sequential model, una lista lineare a cui è possibile aggiungere strati.

Nonostante sia possibile pure creare modelli partendo da zero, in questa applicazione verrà utilizzato il sopra citato $sequential_model()$.

Grazie a queste caratteristiche Keras risulta essere un framework di deep learning che vanta grande supporto e vasta compatibilità, poiché è tra i più utilizzati al mondo.

4.2 Sviluppo di una CNN

La prima parte del progetto, come già accennato, consiste nello sviluppo di un modello. Chiaramente trattandosi di riconoscimento di immagini è opportuno utilizzare una rete neurale convoluzionale (CNN). Con Keras è incredibilmente agile aggiungere layer alla rete, infatti è sufficiente utilizzare l'operatore $.add()$ applicato al modello in questione. L'aggiunta dei diversi layer e la modifica

dei loro parametri verranno fatte osservando i risultati ottenuti, in particolare facendo riferimento all'andamento delle curve di apprendimento.

Nelle sezioni successive verrà proposto e spiegato il codice delle funzioni che permettono di elaborare dati per poterli dare in ingresso al modello, così da poterlo addestrare e mostrarne i risultati ottenuti.

4.2.1 Il Dataset

Per poter addestrare il modello si utilizzerà il learning supervisionato, in quanto è il più adatto per questo genere di problematica. È dunque necessario disporre di un dataset di immagini etichettate e suddivise in varie classi: verrà utilizzato il CIFAR-10.

Il CIFAR-10 è composto da 60000 immagini RGB di dimensioni 32×32 pixels, etichettate e divise in 10 categorie:

- **aeroplani**
- **automobili**
- **uccelli**
- **gatti**
- **cervi**
- **cani**
- **rane**
- **cavalli**
- **navi**
- **camion**

50000 immagini saranno impiegate per addestrare il modello, mentre 10000 immagini verranno utilizzate per eseguire i test di validazione del modello, ovvero per capire quanto quest'ultimo è accurato.

Nel codice proposto di seguito verrano caricate le immagini del dataset e si verificherà che le 50000 sono dedicate al train e le 10000 sono dedicate al test.

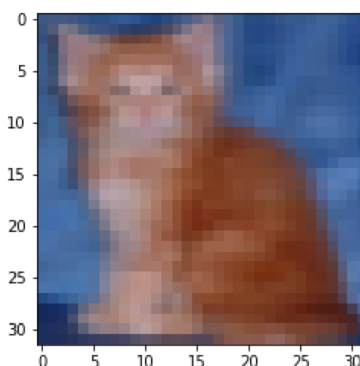
```
1 from keras.datasets import cifar10
2
3 # caricamento del dataset
4 (trainX, trainY), (testX, testY) = cifar10.load_data()
5
6 # stampa la dimensione dei vettori di train e di test
7 print('train ->    immagini: %s    label:%s' % (trainX.shape
     , trainY.shape))
8 print('test ->    immagini: %s   label:%s' % (testX.shape,
    testY.shape))
```

```
1 train ->    immagini: (50000, 32, 32, 3)    label:(50000, 1)
2 test ->    immagini: (10000, 32, 32, 3)   label:(10000, 1)
```

Dall'output possiamo chiaramente notare che le immagini sono di dimensione $32 \times 32 \times 3$, infatti sono immagini a 3 canali RGB della dimensione di 32×32 pixel. Prendendo in esame le labels si nota che esse sono tante quante sono le immagini e hanno la dimensione di 1, infatti contengono una cifra ciascuna (da 0 a 9)

La libreria *pyplot* ci permette di visualizzare le immagini contenute nel dataset. Prendendo, ad esempio, l'immagine di indice 91 e mostrandola con pyplot si può constatare che si tratta di un gatto; si avrà la label associata uguale a 3, corrispondente appunto alla classe *"gatti"*.

```
1 [3]
```



4.2.2 One Hot encoding

Le etichette associate alle immagini del dataset attualmente sono in formato decimale; per poterle utilizzare occorre modificarle in codifica one hot, al fine di ridurre il carico computazionale che occorre alle GPU per fare confronti tra le varie labels.

Questa codifica consiste nel convertire un numero intero in un array di bit tutti pari a zero, ad esclusione della cella puntata dall'indice corrispondente al numero che si vuole codificare. Nel dataset del caso preso in esame si considerano 10 differenti classi, quindi saranno necessari 10 diversi array per rappresentare tali classi in codifica one hot:

- 0 → **aeroplani** → 1000000000
- 1 → **automobili** → 0100000000
- 2 → **uccelli** → 0010000000
- 3 → **gatti** → 0001000000
- 4 → **cervi** → 0000100000
- 5 → **cani** → 0000010000
- 6 → **rane** → 0000001000
- 7 → **cavalli** → 0000000100

- 8 → **navi** → 0000000010
- 9 → **camion** → 0000000001

4.2.3 Preparazoine del dataset

Analizzando il dataset possiamo notare che le immagini sono composte da valori compresi in [0 : 255], ma per poter dare in ingresso al modello le immagini è necessario far sì che tutti i valori dei pixels siano compresi nell'intervallo [0 : 1]. Si procede quindi a trasformare i valori interi in valori *float* per poi essere normalizzati, dividendoli per il massimo valore possibile.

Nel seguente blocco di codice si è quindi creata una funzione che, quando richiamata, permette di: caricare il dataset, trasformare la codifica delle labels in one-hot e normalizzare i pixels.; infine si restituiranno i valori pronti ad essere dati in ingresso al modello.

```
1  from keras.utils import to_categorical
2
3  def prepare_data():
4      # carica il dataset cifar10
5      (x_train, y_train), (x_test, y_test) = cifar10.load_data
           ()
6
7      # converte in one hot enconding
8      y_train = to_categorical(y_train, 10)
9      y_test = to_categorical(y_test, 10)
10
11     #  normalizza   i pixels
12     x_train = x_train.astype('float32')
13     x_test = x_test.astype('float32')
14     x_train = x_train/255
15     x_test = x_test/255
16
17     # restituisce i valori
18     return x_train, y_train, x_test, y_test
```

4.2.4 Plotting dei risultati

Al fine di capire il comportamento dei modelli che verranno proposti è fondamentale visionare le curve di apprendimento generate. A tale scopo ci viene in aiuto la già nota libreria pyplot.

Grazie al codice seguente si è quindi in grado di capire in che modo il modello apprende ed è possibile avere una visione su eventuali problemi che potrebbero emergere.

```
1  def plotting(history):
2          # plot loss
3          plt.subplot(211)
4          plt.title('Cross Entropy Loss')
5          plt.plot(history.history['loss'], color='blue',
               label='train')
6          plt.plot(history.history['val_loss'], color='
               orange', label='test')
7          plt.legend(loc=0)
8
9          # plot accuratezza
10         plt.subplot(212)
11         plt.title('Classification Accuracy')
```

```
12          plt.plot(history.history['accuracy'], color='blue'
                , label='train')
13          plt.plot(history.history['val_accuracy'], color='
                orange', label='test')
14          plt.legend(loc=0)
```

4.2.5 Modello elementare

Il primo tentativo effettuato è stato quello di realizzare un modello semplice con un solo blocco di convoluzione. Tale modello verrà in seguito addestrato e testato, poi sarà preso in esame per fare delle brevi considerazioni.

```
1  def create_model():
2          model = Sequential()
3
4          # blocco 1
5          model.add(Conv2D(32, (3, 3), activation='relu',
               kernel_initializer='he_uniform', padding='same'
               , input_shape=(32, 32, 3)))
6          model.add(Conv2D(32, (3, 3), activation='relu',
               kernel_initializer='he_uniform', padding='same'
               ))
7          model.add(MaxPooling2D((2, 2)))
8
9          model.add(Flatten())
10         model.add(Dense(128, activation='relu',
               kernel_initializer='he_uniform'))
11         model.add(Dense(10, activation='softmax'))
12         return model
```

4.2.6 Funzione di Test

Si passa ora alla definizione della funzione di test. Questa funzione consentirà di compilare il modello ed allenarlo sulle immagini del dataset preso in esame, per poi valutarlo e vedere quanto è accurato.

Leggendo il codice proposto si nota immediatamente che il dataset è suddiviso in due parti: una per il training e una per il testing. È assolutamente necessario utilizzare questa suddivisione, altrimenti se non si testasse il modello con immagini "nuove" non si verrebbe mai a conoscenza della sua reale accuratezza né si saprebbe se il modello può adattarsi a situazioni reali.

Inoltre, per il momento, è stato scelto un learning rate statico pari a 0.001 e un batch size abbastanza piccolo, in modo da poter addestrare il modello utilizzando poca memoria.

Come è possibile vedere nel grafico di apprendimento il risultato è deludente. L'accuratezza del modello è soltanto del 67% e satura già dalle prime epoche di training. In questo caso possiamo si può osservare che il modello va rapidamente ad adattarsi al database di training. Analizzando il grafico della curva di apprendimento si nota che le performance del modello nel dataset di testing (curva gialla) migliorano insieme a quelle nel dataset di training (curva blu) in un periodo iniziale, per poi iniziare a peggiorare dopo sole 15 epoche.

```
1  from keras.optimizers import SGD
2
3  def test_model():
4      # crea il modello
5      model = create_model()
```

```
6
7     opt = SGD(lr=0.001, momentum=0.9)
8     model.compile(optimizer=opt, loss='
         categorical_crossentropy', metrics=['accuracy'])
9
10    # carica dataset
11    x_train, y_train, x_test, y_test = prepare_data()
12
13    history = model.fit(x_train, y_train, epochs=3,
         batch_size=64, validation_data=(x_test, y_test),
         verbose=1)
14    acc = model.evaluate(x_test, y_test, verbose=0)
15    print('> ', acc[1]*100)
16    plotting(history)
17
18 test_model()
```

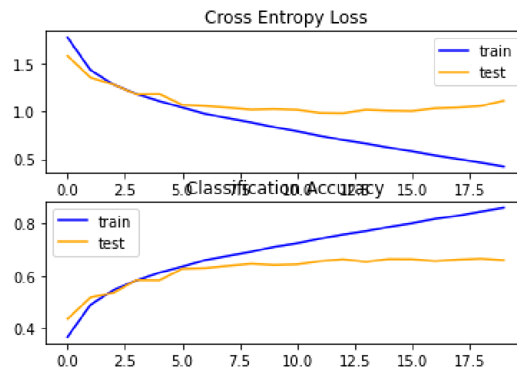


Figura 4.1: Curva di accuratezza del modello a singolo blocco di convoluzione.

4.2.7 Miglioramento del modello

La curva di apprendimento riferita al dataset di testing satura a valori molto bassi, mentre quella riferita al dataset di training raggiunge in fretta valori prossimi al 100% di accuratezza. Questo è segnale che qualcosa non va: il modello si adatta troppo bene alle immagini di training per poi non essere in grado di funzionare correttamente su immagini nuove. Nei successivi esempi di codice verranno incrementati i blocchi di convoluzione per poter estrarre dalle immagini più caratteristiche (o features): dapprima di basso livello, per poi mano a mano estrarle di livello sempre più alto. Con due blocchi formati da due convoluzioni ciascuno si ottiene un'accuratezza intorno al 71%.

```
1 def create_model():
2         model = Sequential()
3
4     # blocco 1
5         model.add(Conv2D(32, (3, 3), activation='relu',
             kernel_initializer='he_uniform', padding='same'
             , input_shape=(32, 32, 3)))
6         model.add(Conv2D(32, (3, 3), activation='relu',
             kernel_initializer='he_uniform', padding='same'
             ))
7         model.add(MaxPooling2D((2, 2)))
```

```
8
9       # blocco 2
10          model.add(Conv2D(64, (3, 3), activation='relu',
                kernel_initializer='he_uniform', padding='same'
                ))
11          model.add(Conv2D(64, (3, 3), activation='relu',
                kernel_initializer='he_uniform', padding='same'
                ))
12          model.add(MaxPooling2D((2, 2)))
13
14          model.add(Flatten())
15          model.add(Dense(128, activation='relu',
                kernel_initializer='he_uniform'))
16          model.add(Dense(10, activation='softmax'))
17          return model
```

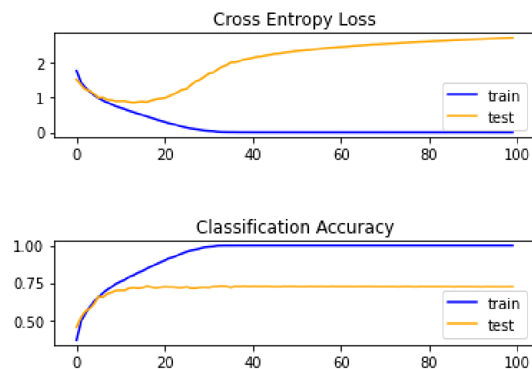


Figura 4.2: Curva di accuratezza del modello a doppio blocco di convoluzione.

Utilizzando invece 3 blocchi con 3 convoluzioni ciascuno si sale circa al 75%. Si tratta dunque di un buon punto di partenza da cui iniziare a sviluppare il modello.

```
1   def create_model():
2       model = Sequential()
3
4       # blocco 1
5       model.add(Conv2D(32, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same',
            input_shape=(32, 32, 3)))
6       model.add(Conv2D(32, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
7       model.add(Conv2D(32, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
8       model.add(MaxPooling2D((2, 2)))
9
10      # blocco 2
11      model.add(Conv2D(64, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
12      model.add(Conv2D(64, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
13      model.add(Conv2D(64, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
14      model.add(MaxPooling2D((2, 2)))
15
16      # blocco 3
17      model.add(Conv2D(128, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
```

```
18      model.add(Conv2D(128, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
19      model.add(Conv2D(128, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
20      model.add(MaxPooling2D((2, 2)))
21
22      model.add(Flatten())
23      model.add(Dense(128, activation='relu',
            kernel_initializer='he_uniform'))
24      model.add(Dense(10, activation='softmax'))
25      return model
```

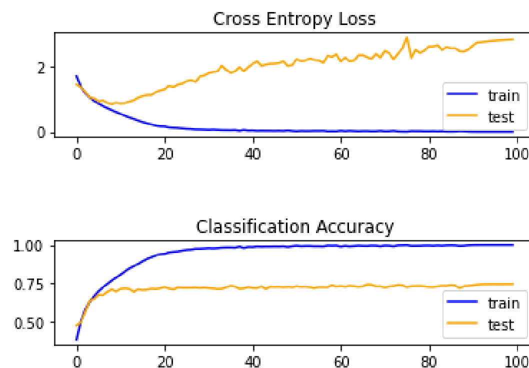


Figura 4.3: Curva di accuratezza del modello a triplo blocco di convoluzione.

4.2.8 Overfitting

Come è apprezzabile dai risultati ottenuti fino ad ora si può concludere che, nell'addestramento, le deep neural networks tendono a sovradattarsi ai dati del dataset di training e, di conseguenza, ottenere performance piuttosto scarse nei test su immagini nuove, in particolare quelle del dataset di testing.

Per risolvere questo problema sarà necessario utilizzare delle tecniche di normalizzazione: ne esistono molte varianti, ma in questa applicazione verranno utilizzate soltanto Dropout normalization e Batch normalization.

4.2.9 Aggiunta del Dropout

Il Dropout permette di escludere casualmente alcuni nodi dalla rete, ciò consente di evitare la situazione in cui alcuni nodi si adattano per correggere gli errori generati dai nodi precedenti. Esso può essere implementato nel modello al termine di ogni blocco di convoluzione; inoltre le migliori performance si ottengono utilizzando un valore di dropout crescente al susseguirsi dei layer.

Grazie all'introduzione del dropout si è ottenuto un notevole incremento delle prestazioni del modello, raggiungendo velocemente un'accuratezza pari all'83%.

```
1  def create_model():
2      model = Sequential()
3
4      # blocco 1
```

```
5       model.add(Conv2D(32, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same',
            input_shape=(32, 32, 3)))
6       model.add(Conv2D(32, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
7       model.add(Conv2D(32, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
8       model.add(Dropout(0.2))
9       model.add(MaxPooling2D((2, 2)))
10
11      #  blocco  2
12      model.add(Conv2D(64, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
13      model.add(Conv2D(64, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
14      model.add(Conv2D(64, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
15      model.add(Dropout(0.3))
16      model.add(MaxPooling2D((2, 2)))
17
18      # blocco 3
19      model.add(Conv2D(128, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
20      model.add(Conv2D(128, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
21      model.add(Conv2D(128, (3, 3), activation='relu',
            kernel_initializer='he_uniform', padding='same'))
22      model.add(Dropout(0.4))
23      model.add(MaxPooling2D((2, 2)))
24
25      model.add(Flatten())
26      model.add(Dropout(0.5))
27      model.add(Dense(128, activation='relu',
            kernel_initializer='he_uniform'))
28      model.add(Dense(10, activation='softmax'))
29      return model
```

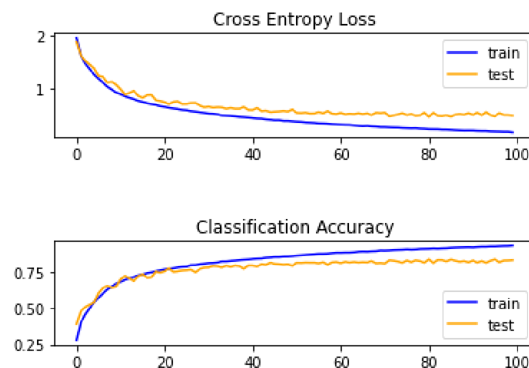


Figura 4.4: Curva di accuratezza del modello a seguito dell'aggiunta del layer di dropout.

4.2.10 Utilizzo di Data Augmentation

Per ottenere degli ottimi risultati è consigliabile addestrare il modello sul più grande numero di immagini possibili, ma come si agisce a seguito dell'impos-

sibilità di ottenere ulteriori immagini? Con la tecnica dell'aumento dei dati è possibile generare nuovi dati partendo da quelli esistenti, semplicemente applicando rotazioni, flip o altre semplici modifiche alle immagini. Una rete neurale è robusta se è in grado di riconoscere i soggetti delle immagini indipendentemente dalla loro posizione o dalla loro dimensione relativa al frame dell'immagine.

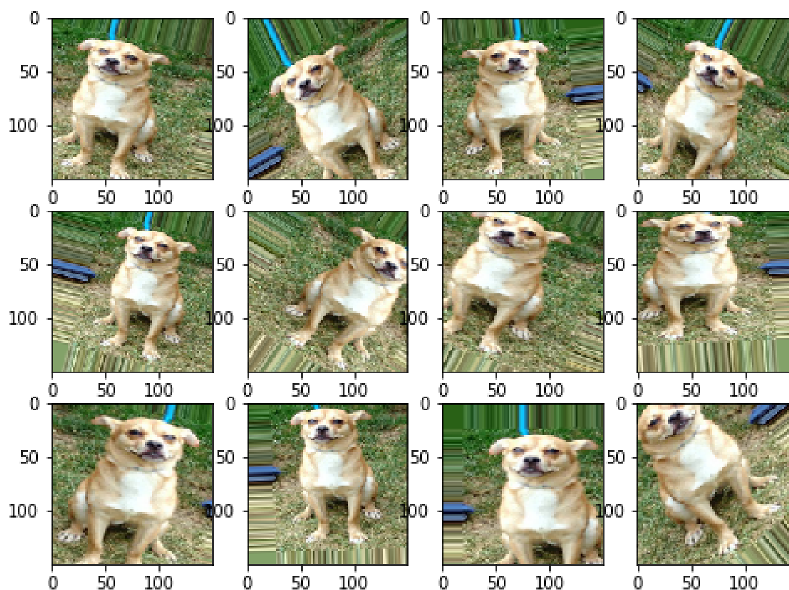


Figura 4.5: Data augmentation applicata ad un'immagine.

Aumentando i dati è dunque possibile far sì che il modello non apprenda le features superflue, ma si concentri soltanto su quelle che realmente definiscono un'immagine.

A seguito dell'introduzione della data augmentation si può chiaramente notare che su un training di 100 epoche l'accuratezza finale è nettamente più bassa rispetto all'accuratezza senza data augmentation. Guardando la curva, però, è immediatamente visibile che l'overfitting è praticamente assente. Facendo così un training su più epoche si avranno risultati migliori: già a 200 epoche si può raggiungere un'accuratezza di circa l'85%.

```
1  def test_model():
2      # carica dataset
3      x_train, y_train, x_test, y_test = prepare_data()
4
5      # build model
6      model = create_model()
7
8      opt = SGD(lr=0.001, momentum=0.9)
9      model.compile(optimizer=opt, loss='
           categorical_crossentropy', metrics=['accuracy'])
10
11     # carica dataset
12     x_train, y_train, x_test, y_test = prepare_data()
13
14     # image augmentation
15     datagen = ImageDataGenerator(
```

```
16                      featurewise_center=False,
17                      samplewise_center=False,
18                      featurewise_std_normalization=False,
19                      samplewise_std_normalization=False,
20                      zca_whitening=False,
21                      rotation_range=30,
22                      width_shift_range=0.1,
23                      height_shift_range=0.1,
24                      horizontal_flip=True,
25                      vertical_flip=False)
26
27      datagen.fit(x_train)
28
29      # fit
30      history = model.fit_generator(datagen.flow(x_train,
            y_train, batch_size=128), steps_per_epoch=x_train.
            shape[0] // 128, epochs=100, validation_data=(x_test,
             y_test))
31      acc = model.evaluate(x_test, y_test, verbose=0)
32      print('> ', acc[1]*100)
33      plotting(history)
34
35  test_model()
```

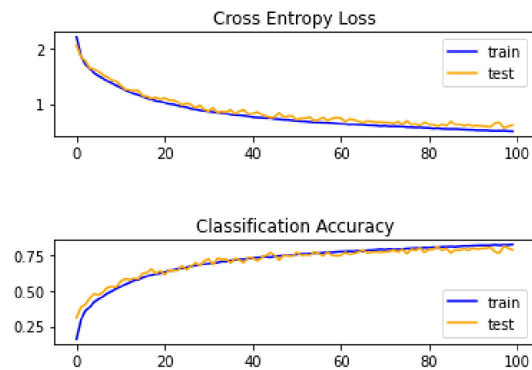


Figura 4.6: Curva di accuratezza del modello a seguito dell'utilizzo di data augmentation.

4.2.11 Batch Normalization

Una generica struttura di reti neurali è formata da neuroni interconnessi tra loro, ciascuno dei quali produce un valore di attivazione. L'attivazione parte dunque dai neuroni del primo layer per poi attivare in maniera ponderata i neuroni dei layer successivi. Nel caso di reti profonde i pesi di ogni layer vengono modificati nel corso del training, ciò significa che cambiano anche i valori di attivazione per ogni livello. Essendo però i valori di attivazione di ogni layer gli input del layer successivo, la rete si trova nella situazione in cui la distribuzione degli input varia ad ogni iterazione nel processo di training. Questo è un grave problema in quanto costringe i neuroni dei vari strati (eccetto quello di input) a riadattarsi ai cambiamenti degli input.

Tale problema è detto *Internal Covariate Shift* e, per poterlo ridurre, è fondamentale normalizzare i valori degli input ad ogni epoca di training.

Il layer Batch Normalization non fa altro che calcolare la media e la varianza di un peso per ogni mini-batch:

$$\mu_b = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{4.1}$$

$$\sigma_b^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_b)^2 \tag{4.2}$$

Poi sottrae il valore medio e divide per la deviazione standard:

$$\tilde{x}_i = \frac{x_i - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}} \tag{4.3}$$

in questo modo il nuovo input avrà un valore compreso in [0 : 1].

Nel seguente blocco di codice verrà utilizzata Batch Normalization, inoltre verranno suddivisi i vari layer per una maggiore leggibilità.

```
1  from keras.layers.normalization import BatchNormalization
2
3  def create_model():
4
5      model = Sequential()
6
7      # blocco 1
8      model.add(Conv2D(32, (3, 3), padding='same',input_shape
           =(32, 32, 3)))
9      model.add(BatchNormalization())
10     model.add(Activation('relu'))
11     model.add(Dropout(0.4))
12
13     model.add(Conv2D(32, (3, 3), padding='same'))
14     model.add(BatchNormalization())
15     model.add(Activation('relu'))
16
17     model.add(Conv2D(32, (3, 3), padding='same'))
18     model.add(BatchNormalization())
19     model.add(Activation('relu'))
20
21     model.add(MaxPooling2D(pool_size=(2, 2)))
22
23     # blocco 2
24     model.add(Conv2D(64, (3, 3), padding='same'))
25     model.add(BatchNormalization())
```

```
26      model.add(Activation('relu'))
27      model.add(Dropout(0.4))
28
29      model.add(Conv2D(64, (3, 3), padding='same'))
30      model.add(BatchNormalization())
31      model.add(Activation('relu'))
32
33      model.add(Conv2D(64, (3, 3), padding='same'))
34      model.add(BatchNormalization())
35      model.add(Activation('relu'))
36
37      model.add(MaxPooling2D(pool_size=(2, 2)))
38
39      # blocco 3
40      model.add(Conv2D(128, (3, 3), padding='same'))
41      model.add(BatchNormalization())
42      model.add(Activation('relu'))
43      model.add(Dropout(0.4))
44
45      model.add(Conv2D(128, (3, 3), padding='same'))
46      model.add(BatchNormalization())
47      model.add(Activation('relu'))
48
49      model.add(Conv2D(128, (3, 3), padding='same'))
50      model.add(BatchNormalization())
51      model.add(Activation('relu'))
52
53      model.add(MaxPooling2D(pool_size=(2, 2)))
54
55
56      # output layer
57      model.add(Flatten())
58      model.add(Dropout(0.5))
59      model.add(Dense(256)) #era 512
60      model.add(BatchNormalization())
61      model.add(Activation('relu'))
62      model.add(Dropout(0.5))
63      model.add(Dense(num_classes, activation='softmax'))
64      print(model.summary())
65
66      return model
```

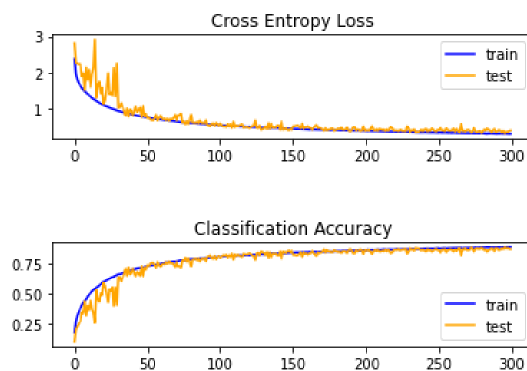


Figura 4.7: Curva di accuratezza del modello a seguito dell'utilizzo del layer batch normalization.

4.2.12 Learning Rate dinamico

Le curve di apprendimento ottenute con l'ultima revisione di codice hanno un andamento poco costante ed oscillano in maniera non indifferente. Per far sì che ciò non avvenga si introduce un valore di learning rate dinamico, in modo che l'algoritmo di apprendimento possa andare a diminuire i pesi in maniera sempre più tenue progressivamente allo scorrere delle epoche. Questo comporta una stabilizzazione della curva di apprendimento nella parte finale del training.

È chiaramente visibile che, grazie all'introduzione del learning rate dinamico, la curva di apprendimento nella parte finale è particolarmente stabile, infatti non sono presenti oscillazioni di grande entità.

È inoltre stato introdotto un ottimizzatore: un algoritmo di ottimizzazione che permette di vedere per quali pesi la funzione costo risulti minima; più precisamente viene utilizzato *SGD*.

Il modello è ora in grado di ottenere un valore di accuratezza pari al 90%.

```
1  def test_model():
2          # carica normalizza e one hot
3          x_train, y_train, x_test, y_test = prepare_data()
4
5          #  callbacks per lr dinamico
6          csv_logger = callbacks.CSVLogger('training.log')
7          reduce_lr =  callbacks.ReduceLROnPlateau(monitor='
               loss', factor=0.5, patience=10, min_lr=0.001)
8          all_callbacks = [csv_logger, reduce_lr]
9
10         # build model
11         model = build_model(x_train.shape[1:])
12
13         # optimizer
14         sgd = optimizers.SGD(lr=0.2, momentum=0.9, decay=1e
               -4)
15         model.compile(loss='categorical_crossentropy',
               optimizer=sgd, metrics=['accuracy'])
16
17         # image augmentation
18         datagen = ImageDataGenerator(
19             featurewise_center=False,
20             samplewise_center=False,
21             featurewise_std_normalization=False,
22             samplewise_std_normalization=False,
23             zca_whitening=False,
24             rotation_range=30,
25             width_shift_range=0.1,
26             height_shift_range=0.1,
27             horizontal_flip=True,
28             vertical_flip=False)
29
30         datagen.fit(x_train)
31
32         # train
33         history = model.fit_generator(datagen.flow(x_train,
               y_train,
34                                     batch_size=batch_size),
35                                     steps_per_epoch=x_train.
                                           shape[0] // batch_size,
36                                     epochs=epochs,
37                                     validation_data=(x_test,
                                           y_test),
38                                     callbacks=all_callbacks)
39         plotting(history)
```

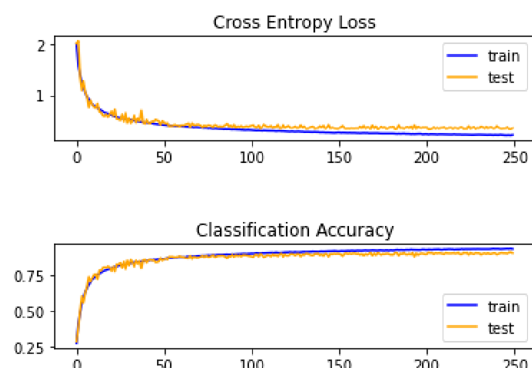


Figura 4.8: Curva di accuratezza del modello a seguito dell'utilizzo del learning rate dinamico.

4.2.13 Modello ottimo

Un ultimo accorgimento che è possibile utilizzare per ottenere performance migliori è quello di aumentare il numero di filtri di convoluzione così da permettere alla rete di apprendere più features dalle immagini.

Tale modifica incrementa notevolmente il numero di pesi della rete e di conseguenza il costo computazionale per utilizzarla. D'altra parte, però, ci permette di ottenere un'accuratezza maggiore, in questo caso pari al 92%.

```
def build_model():

    model = Sequential()

    # blocco 1
    model.add(Conv2D(64, (3, 3), padding='same',input_shape=(32, 32, 3)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Dropout(0.4))

    model.add(Conv2D(64, (3, 3), padding='same'))
    model.add(BatchNormalization())
    model.add(Activation('relu'))

    model.add(Conv2D(64, (3, 3), padding='same'))
    model.add(BatchNormalization())
    model.add(Activation('relu'))

    model.add(MaxPooling2D(pool_size=(2, 2)))

    # blocco 2
    model.add(Conv2D(128, (3, 3), padding='same'))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Dropout(0.4))

    model.add(Conv2D(128, (3, 3), padding='same'))
    model.add(BatchNormalization())
    model.add(Activation('relu'))

    model.add(Conv2D(128, (3, 3), padding='same'))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
```

```
34
35      model.add(MaxPooling2D(pool_size=(2, 2)))
36
37      # blocco 3
38      model.add(Conv2D(256, (3, 3), padding='same'))
39      model.add(BatchNormalization())
40      model.add(Activation('relu'))
41      model.add(Dropout(0.4))
42
43      model.add(Conv2D(256, (3, 3), padding='same'))
44      model.add(BatchNormalization())
45      model.add(Activation('relu'))
46
47      model.add(Conv2D(256, (3, 3), padding='same'))
48      model.add(BatchNormalization())
49      model.add(Activation('relu'))
50
51      model.add(MaxPooling2D(pool_size=(2, 2)))
52
53
54      # output layer
55      model.add(Flatten())
56      model.add(Dropout(0.5))
57      model.add(Dense(512)) #era 512
58      model.add(BatchNormalization())
59      model.add(Activation('relu'))
60      model.add(Dropout(0.5))
61      model.add(Dense(num_classes, activation='softmax'))
62      print(model.summary())
63
64      return model
```

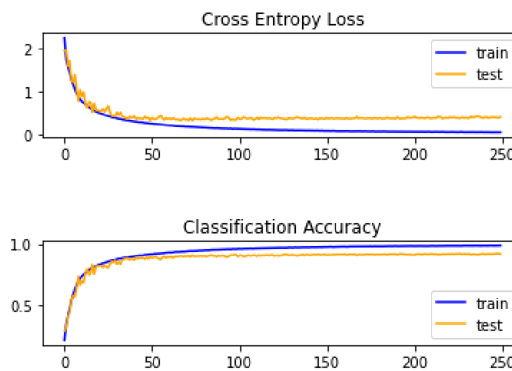


Figura 4.9: Curva di accuratezza del modello ottimo.

4.2.14 Confusion Matrix

La confusion matrix mette in relazione le varie classi del dataset con le classificazioni fatte dal modello, permettendo così di analizzare in maniera grafica alcuni dei problemi del modello.

Si nota immediatamente come la maggior parte degli errori vengono fatti tra le classi 3 e 5. Queste due classi rappresentano rispettivamente "cani" e "gatti", incertezze minori si hanno invece tra le classi "automobili" e "camion". Gli errori commessi tra queste coppie di classi sono chiaramente legate agli

innumerevoli tratti comuni che hanno i cani con i gatti o le auto con i camion; infatti alcuni tratti significativamente distintivi, come ad esempio la dimensione del soggetto, non sono ovviamente valutabili a partire da un'immagine.

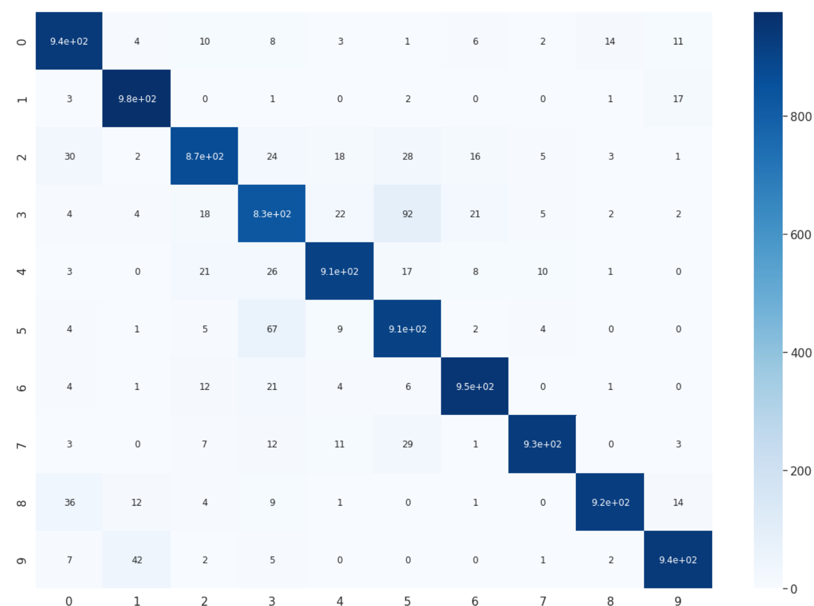


Figura 4.10: Confusion matrix del modello ottimo.

4.2.15 Esportazione del modello

È infine necessario esportare il modello pre-allenato, in modo tale che sia possibile utilizzarlo in futuro senza la necessità di effettuare nuovamente il lungo e computazionalmente oneroso processo di training.

Per esportare il modello nel formato $.h5$ sarà sufficiente utilizzare il seguente comando:

```
1 model.save("modello_scalato.h5")
```

4.3 Porting su STM32

L'ultima parte del progetto consiste nel portare il modello pre-addestrato nella board OpenMV Cam H7 e fare predizioni di oggetti o animali sfruttando il sensore di immagine integrato.

4.3.1 OpenMV Cam H7

La OpenMV Cam H7 è una board con microcontrollore e sensore di immagine integrato. È possibile programmarla in MicroPython sfruttando tutti i vantaggi derivanti dall'utilizzo di linguaggi ad alto livello che esso porta. Il microcontrollore è un STM32 con architettura ARM Cortex-M7 che opera alla frequenza

di 480Mhz. La board è inoltre equipaggiata con 32MB di SDRAM + 1MB di SRAM e 32 MB di flash esterna + 2 MB di flash interna.

Caratteristica singolare della board è il sensore OV5640 da 5MP in grado di catturare immagini fino a 50FPS.

Figura 4.11: OpenMV Cam H7 Plus.

4.3.2 Conversione e compressione del modello

Al fine di poter utilizzare il modello esportato sulla board è opportuno convertirlo in formato $*.tflite$ e procedere alla quantizzazione dei pesi, in modo da ridurre notevolmente le sue dimensioni. Il seguente codice ci permette di convertire e comprimere il modello. Il file restituito in output è di soli 4Mb, a differenza dei 31Mb del modello $*.h5$.

```
1  import tensorflow as tf
2  import numpy as np
3
4  cifar_train, _ = tf.keras.datasets.cifar10.load_data()
5  images = tf.cast(cifar_train[0], tf.float32) / 255
6  cifar_ds = tf.data.Dataset.from_tensor_slices((images)).
       batch(1)
7  def representative_data_gen():
8    for input_value in cifar_ds.take(100):
9      yield [input_value]
10
11 converter = tf.lite.TFLiteConverter.from_keras_model(model)
12 converter.experimental_new_converter = True
13 converter.optimizations = [tf.lite.Optimize.
       OPTIMIZE_FOR_SIZE]
14 converter.target_spec.supported_ops = [tf.lite.OpsSet.
       TFLITE_BUILTINS_INT8]
15 converter.inference_input_type = tf.uint8
16 converter.inference_output_type = tf.uint8
```

```
converter.representative_dataset = representative_data_gen
tflite_quant_model = converter.convert()

# salva il modello compresso
open("/content/drive/My Drive/modello_scalato.tflite", "wb")
    .write(tflite_quant_model)
```

Andando ad effettuare le predizioni sulle immagini di test del dataset, utilizzando il modello compresso, si ottengono risultati decisamente paragonabili a quelli ottenuti con il modello non compresso, è quindi possibile affermare che la compressione non ha avuto un impatto rilevante sull'accuratezza del modello.

4.3.3 Scaling del modello

Le dimensioni del modello ottenuto dalla compressione non sono ancora sufficientemente ridotte per l'importazione sull'OpenMV Cam, sarà quindi necessario andare a rivisitare il modello originale e ridurre il numero dei pesi, in modo tale che possa essere utilizzato sulla board. In particolare, rispetto al modello ottimo, verranno utilizzate soltanto 2 convoluzioni per ogni blocco anziché 3 e verranno ridotti drasticamente i filtri di ciascuna convoluzione. Di seguito il codice completo del modello scalato:

```
def build_model():

    model = Sequential()

    # blocco 1
    model.add(Conv2D(16, (3, 3), padding='same',input_shape
        =(32, 32, 3))))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Dropout(0.4))

    model.add(Conv2D(16, (3, 3), padding='same'))
    model.add(BatchNormalization())
    model.add(Activation('relu'))

    model.add(MaxPooling2D(pool_size=(2, 2)))

    # blocco 2
    model.add(Conv2D(32, (3, 3), padding='same'))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Dropout(0.4))

    model.add(Conv2D(32, (3, 3), padding='same'))
    model.add(BatchNormalization())
    model.add(Activation('relu'))

    model.add(MaxPooling2D(pool_size=(2, 2)))

    # blocco 3
    model.add(Conv2D(64, (3, 3), padding='same'))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Dropout(0.4))

    model.add(Conv2D(64, (3, 3), padding='same'))
    model.add(BatchNormalization())
    model.add(Activation('relu'))

    model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
40
41
42     # output layer
43     model.add(Flatten())
44     model.add(Dropout(0.5))
45     model.add(Dense(128)) #era 512
46     model.add(BatchNormalization())
47     model.add(Activation('relu'))
48     model.add(Dropout(0.5))
49     model.add(Dense(num_classes, activation='softmax'))
50     print(model.summary())
51
52     return model
```

Sarà poi sufficiente addestrare di nuovo il modello, esportarlo e convertirlo in $*.tflite$ utilizzando il codice visto al punto precedente. In questo caso l'accuratezza del modello cala notevolmente, si passa dal 92% al 77%. Nonostante ciò il modello scalato e compresso in $*.tflite$ non presenta differenze di accuratezza rilevanti rispetto al modello scalato in $*.h5$.

4.3.4 Test finale

Per capire quali sono le capacità del modello sviluppato è stato eseguito un test: sono state proposte al sensore di immagine dell'OpenMV Cam numerose immagini con soggetti appartenenti alla stessa classe per un totale di 60 secondi; al termine dell'esecuzione è stata calcolata la percentuale di accuratezza. Questo test è stato ripetuto per ogni classe del dataset su cui è stata addestrata la rete.

Qui di seguito il codice in MicroPython dell'applicazione utilizzata sulla OpenMV Cam H7.

```
1  # OpenMV MNIST Classification
2
3  import sensor, image, time, tf
4
5  sensor.reset()                        # Reset and initialize
       the sensor.
6  sensor.set_contrast(3)
7  sensor.set_brightness(0)
8  sensor.set_auto_gain(True)
9  sensor.set_auto_exposure(True)
10 sensor.set_pixformat(sensor.RGB565) # Set pixel format to
       Grayscale
11 sensor.set_framesize(sensor.QQVGA)    # Set frame size to 80
       x60
12 sensor.skip_frames(time = 2000)      # Wait for settings take
        effect.
13 clock = time.clock()                 # Create a clock object
       to track the FPS.
14
15
16 i = 0
17 n = 0
18 net = tf.load('76.tflite')
19 labels = ['AEREO', 'AUTO', 'UCCELLO', 'GATTO', 'CERVO', '
       CANE', 'RANA', 'CAVLLO', 'NAVE', 'CAMION']
20 label_id = 0
21
22 clock = time.clock()
23 timeout = 60
24 timeout_start = time.time()
25
```

```
26 while time.time() < timeout_start + timeout:
27 
28     clock.tick()
29     img = sensor.snapshot()
30 
31     for obj in net.classify(img, min_scale=1.0, scale_mul
           =0.5, x_overlap=0.0, y_overlap=0.0):
32 
33         if(obj.output().index(max(obj.output()))==label_id):
34             n+=1
35 
36         print(obj.output().index(max(obj.output())))
37         img.draw_rectangle(obj.rect())
38         img.draw_string(obj.x()+3, obj.y()-1, labels[obj.
               output().index(max(obj.output()))], mono_space =
               False)
39         i+=1
40     print(clock.fps(), "fps")
41 
42 print ("testati: ", i)
43 print ("corretti: ", n)
```

Capitolo 5

Conclusioni

Considerazioni sul modello ottimo

Il modello ottimo è composto da circa 4 milioni di parametri addestrabili ed offre un'accuratezza del 92%; è inoltre possibile raggiungere un'accuratezza prossima a quella massima già dopo poche epoche di training, indicativamente dalle cinquanta alle settanta. Questa rete classifica bene tutte le varie categorie presenti nel dataset di training, senza particolari discrepanze su nessuna di esse. Infatti, come è possibile vedere dalla confusion matrix riportata precedentemente, le aree di errore sono piuttosto uniformi e la maggior parte degli errori vengono commessi quando si ha a che fare con classi aventi caratteristiche molto simili. Generalmente si tratta di classi che un'intelligenza umana è in grado di riconoscere grazie a delle informazioni che un'intelligenza artificiale non può ottenere, come ad esempio le informazioni sulle dimensioni di un oggetto, derivanti da un'analisi della prospettiva. Non a caso le aree in cui si vedono più errori nella confusion matrix sono quelle negli incroci tra le classi *cani* e *gatti* e quelle negli incroci tra *camion* e *automobili*.

Considerazioni sulla situazione reale

Per utilizzare il modello applicandolo a situazioni reali è stato necessario effettuare un porting sulla board OpenMV Cam H7; ciò ha comportato alcuni problemi. Primo su tutti la necessità di ridurre le dimensioni di utilizzo di memoria del modello: per poter immagazzinare il modello all'interno della memoria della OpenMV Cam è stato quindi necessario passare da una rete con 4 milioni di parametri addestrabili ad una rete composta da circa 200 mila parametri addestrabili. Questa notevole riduzione è stata ottenuta principalmente grazie alla riduzione del numero di filtri di convoluzione e ha portato ad una notevole diminuzione dell'accuratezza nella classificazione. D'altro canto la riduzione dei filtri di convoluzione ha anche permesso di ridurre la complessità computazionale del modello, permettendo alla OpenMV Cam di catturare e classificare immagini ad una velocità di 11.6 fps.

Con questa nuova configurazione la percentuale di accuratezza nel riconoscimento di immagini provenienti dal dataset di test è di circa il 76%.

Nel momento in cui andiamo ad utilizzare l'applicazione per classificare immagini provenienti dal mondo reale, catturate tramite il sensore di immagine

Hardware	Modello	Acc. su dataset	Acc. su snapshot	Peso
GPU	modello_ottimo.h5	92.03%		31Mb
GPU	modello_ottimo.tflite	91.80%		4Mb
GPU	modello_scalato.h5	76.30%		7.1Mb
GPU	modello_scalato.tflite	76.10%		211Kb
OpenMV CAM	modello_scalato.tflite	75.90%	68.40%	211Kb

della OpenMV Cam, l'accuratezza scende al 68%. Tale diminuzione è ragionevole: a differenza della situazione ideale, in quella reale abbiamo del rumore nell'immagine, movimenti involontari della camera e sfondi non statici.

Bibliografia

[1] Binh Phan, *10 Minutes to Building a CNN Binary Image Classifier in TensorFlow*, 2019
https://towardsdatascience.com

[2] Jason Brownlee, *How Do Convolutional Layers Work in Deep Learning Neural Networks?*, 2019
https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/

[3] MathWorks, *Rete neurale convoluzionale*
https://it.mathworks.com/discovery/convolutional-neural-network-matlab.html

[4] Gianluca Smeraldi, *Introduzione alle reti neurali*
http://www.emernet.it/TR9601.html

[5] Jason Brownlee, *How to Choose an Activation Function for Deep Learning*, 2021
https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/

[6] Jason Brownlee, *Softmax Activation Function with Python*, 2020
https://machinelearningmastery.com/softmax-activation-function-with-python/

[7] Jason Brownlee, *What is Deep Learning?*, 2019
https://machinelearningmastery.com/what-is-deep-learning/

[8] Jason Brownlee, *How to Fix the Vanishing Gradients Problem Using the ReLU*, 2019
https://machinelearningmastery.com/how-to-fix-vanishing-gradients-using-the-rectified-linear-activation-function/

[9] Jason Brownlee, *A Gentle Introduction to Mini-Batch Gradient Descent and How to Configure Batch Size*, 2017
https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/

[10] Jason Brownlee, *Dropout Regularization in Deep Learning Models With Keras*, 2016
https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/

[11] Jason Brownlee, *How to Save and Load Your Keras Deep Learning Model*, 2019
https://machinelearningmastery.com/save-load-keras-deep-learning-models/

[12] Aarya Brahmane, *Image Classification using CNN*, 2020
https://towardsdatascience.com/deep-learning-with-cifar-10-image-classification-64ab92110d79"

[13] Dan Nelson, *Image Recognition in Python with TensorFlow and Keras*, 2019
https://stackabuse.com/image-recognition-in-python-with-tensorflow-and-keras/