UNIVERSITÀ DEGLI STUDI DI CATANIA
DIPARTIMENTO DI MATEMATICA E INFORMATICA
CORSO DI LAUREA MAGISTRALE IN INFORMATICA

Bricks Classification

Progetto Machine Learning

Studenti:

Isgrò Santino (1000000617)

Mazzamuto Michele (W82000176)

26/06/2019

Introduzione

Il problema del conteggio degli oggetti è uno dei problemi maggiormente affrontati al giorno d'oggi nel campo dell'apprendimento supervisionato. L'obiettivo principale è quello di stimare a partire da un'immagine o un video, il numero di oggetti presenti nella scena. Tale problema ha un campo di applicazione vastissimo, basti pensare: al conteggio dei pacchi e materiali in una catena di montaggio, al monitoraggio delle folle nei sistemi di sorveglianza, al monitoraggio del traffico, al conteggio delle cellule in immagini microscopiche o semplicemente alla capacità di riconoscere, dati due masse di oggetti, quale sia quella più densa contenente una maggior quantità di essi. Tali problemi seppur apparentemente semplici per un essere umano, sono invece molto complessi nel campo del Machine Learning.

Nel seguente progetto è stato affrontato il problema del conteggio degli oggetti in un piano lavoro o catena di montaggio in una versione semplificata. Il concetto di pacco o più in generale di oggetto è infatti stato sostituito da un mattoncino Lego.

Obiettivi

L'obiettivo finale del progetto è quello di riuscire a classificare correttamente un'immagine rappresentante dei blocchetti Lego disposti su un piano tra 10 classi possibili.

Più precisamente, avendo alcuni Brick Lego posizionati casualmente su un piano, l'algoritmo deve essere in grado di classificare la foto in base al numero di Brick presenti nella scena. Contando così il numero di mattoncini.

L'esempio seguente mostra quello che dovrebbe essere il risultato finale del task assegnato.

INPUT: **OUTPUT:**

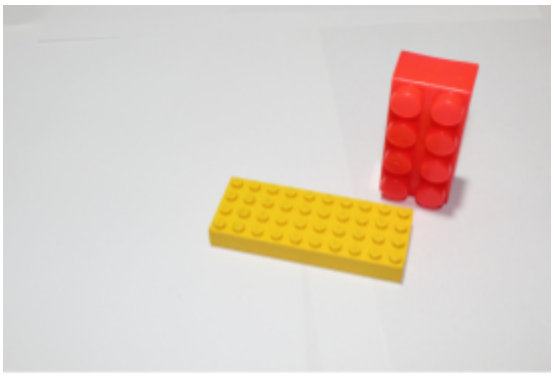


Fig. 1 Immagine non etichettata

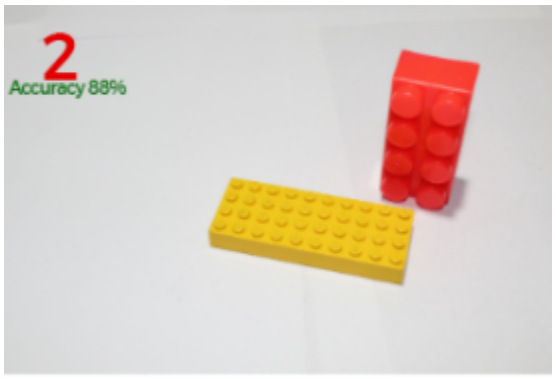


Fig. 2 Immagine etichettata

Dataset

Il dataset completo è visionabile al seguente link:

https://mega.nz/#!3YJ0GaCA!aHATo5sfGCDze0B4WydwLyMT_z8tngKpbvjMVqeAyQw

Non avendo a disposizione un dataset sul quale basare il nostro esperimento, il primo inevitabile passo svolto è stato quello di generarne uno.

L'attrezzatura utilizzata per la creazione del dataset prevede:

- Huawei P20 Lite
- Honor 5C
- Canon 3000d

Honor 5C	
Megapixel	13 Mp
Risoluzione	4208 x 3120 pixel
Aperture Size	F 2
Stabilizzazione	✓ Digital
Autofocus	✓
Touch Focus	✓
Flash	✓ LED
HDR	✓
Geo Tagging	✓
Face Detection	✓
Fotocamera Frontale	✓ 8 Mp

Fig. 3 Specifiche Honor 5C

Huawei P20 Lite	
Megapixel	16 Mp + 2 Mp
Risoluzione	4608 x 3456 pixel
Aperture Size	F 2.2
Stabilizzazione	✓ Digital
Autofocus	✓
Touch Focus	✓
Flash	✓ Dual LED
HDR	✓
Geo Tagging	✓
Face Detection	✓
Fotocamera Frontale	✓ 16 Mp F 2

Fig. 4 Specifiche Huawei P20 Lite

Creazione del Dataset

L'idea è stata quella di raccogliere per ciascuna classe un gruppo di foto, posizionando di volta in volta i Lego in maniera randomica su un piano.

In particolare è stato deciso di scattare le foto su due tipologie di piani: sfondo bianco e sfondo in legno

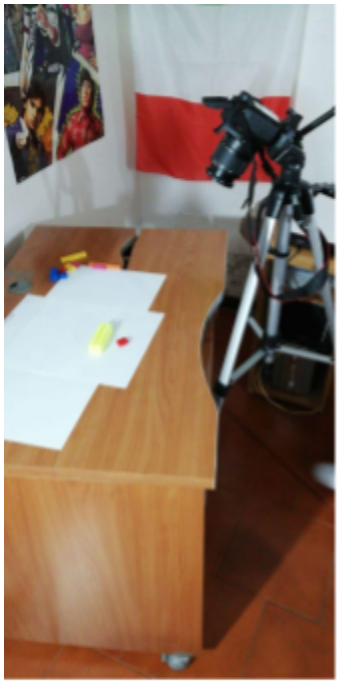


Fig. 5 Processo di creazione del dataset

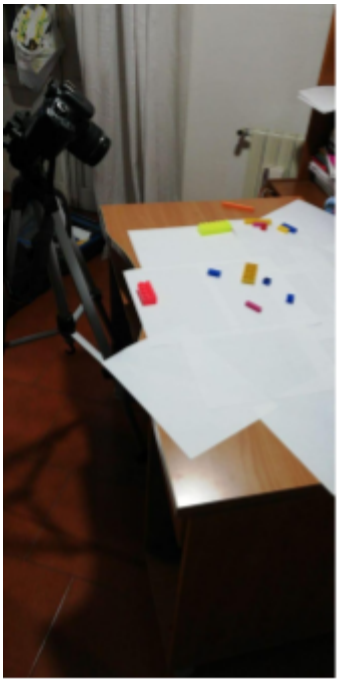


Fig. 6 Processo di creazione del dataset

Sono stati utilizzati i seguenti 14 lego di forma differente:

Fig. 7 Lego utilizzati

Durante la prima fase di generazione il dataset contava circa 200 foto per classe.

Tuttavia, svolgendo i primi test, è stato possibile notare come l'accuracy sia sul train che sul test non superava mai una determinata soglia qualsiasi modello venisse scelto (dettagli nel paragrafo "Esperimenti e risultati").

Per questa motivazione abbiamo deciso di ampliare nettamente il dataset inizialmente creato.

A tale scopo, abbiamo girato circa 10-15 minuti di materiale video per ogni classe, per un totale circa di 150 minuti, dal quale utilizzando una funzione di VLC, abbiamo estrapolato dei frame.

Adesso il nuovo dataset presenta il seguente numero di foto:

Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7	Class 8	Class 9	Class 10
3000	**3000**	**3000**	**3000**	**3000**	**3000**	**3000**	**3000**	**3000**	**3000**

Metodo

Di seguito sono elencati tutti i modelli su cui abbiamo basato la nostra analisi. Il nostro scopo era quello di valutarne per ciascuno l'accuracy in modo tale da poter stabilire il modello migliore per il nostro task.

LeNet

Il primo modello che abbiamo deciso di utilizzare è la LeNet. Questo modello prevede sette layers. Il primo è un layer di convoluzione C1. Il layer prende in input un'immagine di dimensione pixels 28 x 28 e un unico canale. Il secondo è un layer di sottocampionamento S2. Il layer prende in input le 6 mappe di dimensione 24 x 24 e applicando un **AvgPool()** ottiene in output mappe di dimensioni 12 x 12 . Poi vi è un altro layer di convoluzione C3.

Successivamente un layer di sottocampionamento S4 svolge la stessa funzione del layer S2, Infine troviamo layer di trasformazione lineare "fully connected" F5, F6, F7.

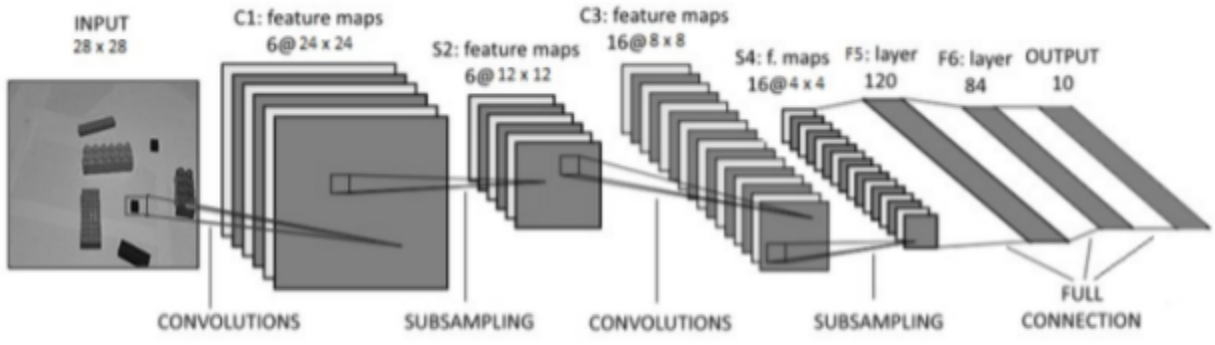


Fig. 8 Struttura della LeNet

Nel dettaglio ecco come è stata inizializzata la LeNet nel codice:

```
from torch import nn
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.C1 = nn.Conv2d(1, 6, 5)
        self.S2 = nn.AvgPool2d(2)
        self.C3 = nn.Conv2d(6, 20, 5)
        self.S4 = nn.AvgPool2d(2)
        self.F5 = nn.Linear(9680, 120)
        self.F6 = nn.Linear(120, 84)
        self.F7 = nn.Linear(84, 10)

        self.activation = nn.Tanh()

    def forward(self,x):

        x = self.C1(x)
        x = self.S2(x)
        x = self.activation(x)
        x = self.C3(x)
        x = self.S4(x)
        x = self.activation(x)
        x = self.F5(x.view(x.shape[0],-1))
        x = self.activation(x)
        x = self.F6(x)
        x = self.activation(x)
        x = self.F7(x)
        return x
```

VGG16

Sono state create più versioni della rete di tipo VGG, ciascuna possedente un numero diverso di strati. In base al numero di strati, ognuna di esse viene solitamente chiamata VGG-n. Tutte queste reti risultano essere più "profonde" rispetto a quella di AlexNet, dove per "profonde" si intende che sono costituite da un numero di strati con parametri allenabili maggiore di AlexNet. VGG16 può quindi essere vista come un' estensione di AlexNet: la struttura è simile, ma i livelli aggiuntivi la rendono sia più pesante (in termini di tempi di training e memoria richiesta) che più performante.

La VGG16 è composta da 16 livelli convoluzionali distribuiti in modo uniforme. Nell'immagine sottostante sono descritti dettagliatamente tutti i livelli convoluzionali della rete utilizzata e le differenze con la rete AlexNet:

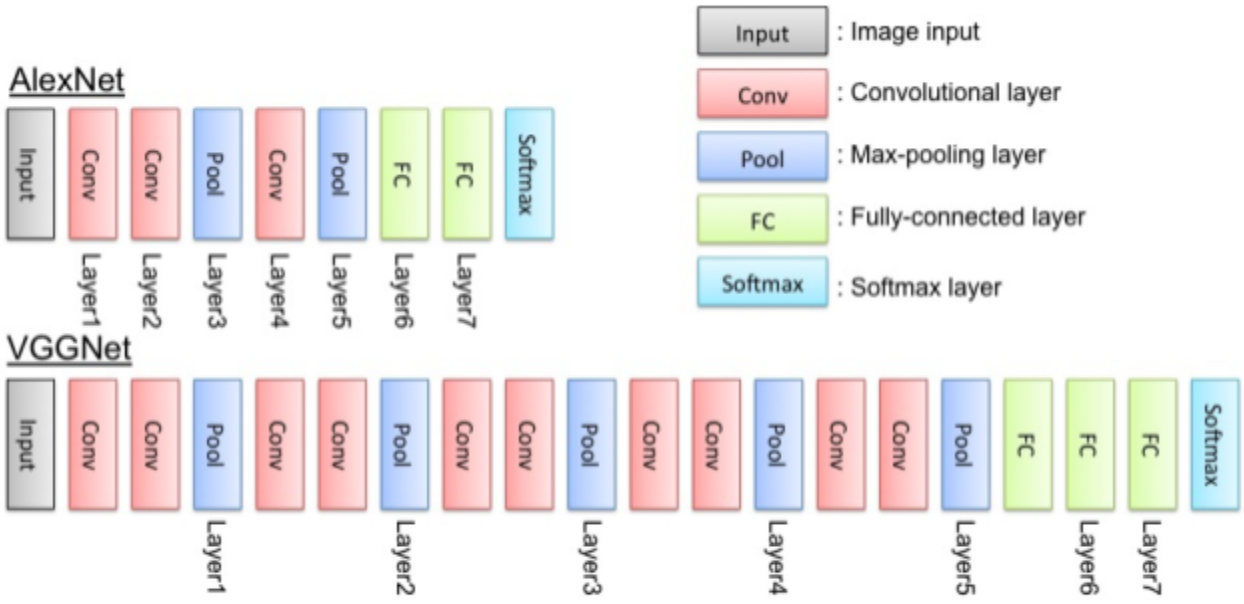


Fig. 9 Confronto tra struttura AlexNet e VGGNet

SqueezeNet1_0

E' un modello di classificazione delle immagini con training basato sul set di dati ImageNet pubblicato nel 2016. E' un modello molto compatto e veloce: occupa pochi megabyte in memoria e può essere utilizzato su CPU non molto performanti. Le performance di SqueezeNet sono comparabili a quelle di AlexNet, che è però molto più lenta/pesante. AlexNet infatti è formata da 240 MB di parametri e SqueezeNet ne ha solo 5 MB.

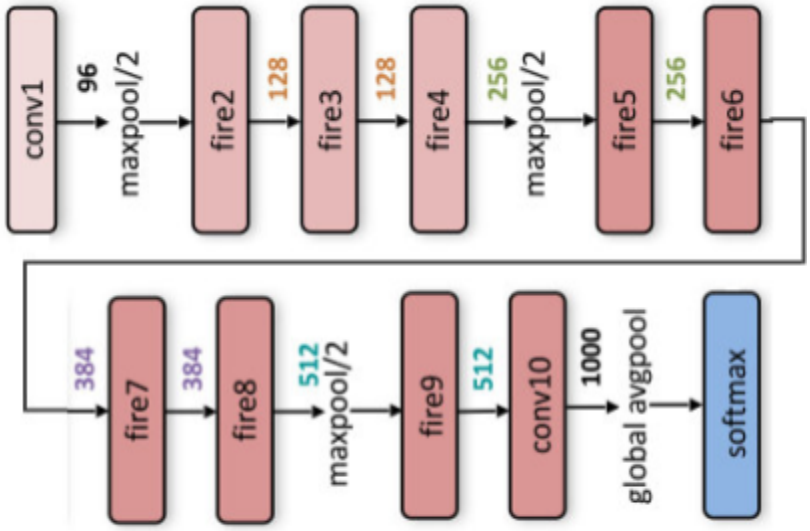


Fig. 10 Struttura SqueezeNet

ResNet50

ResNet-50 è una rete neurale convoluzionale che viene addestrata su oltre un milione di immagini dal database ImageNet. La rete ha una dimensione di input dell'immagine di 224 x 224 pixel ed è profonda 50 livelli riuscendo a classificare le immagini in 1000 categorie di oggetti.

ResNet è nata da una semplice osservazione: "perchè aggiungendo ulteriori strati alle Deep Neural Network l'accuratezza non migliora ma, addirittura, peggiora?"

Intuitivamente, reti neurali più profonde non dovrebbero performare peggio di quelle poco profonde, o almeno non durante l'allenamento quando non vi è alcun rischio di overfitting. Prendiamo una rete di esempio con *n* strati che raggiungono una certa accuratezza. Come minimo, una rete con $n+1$ strati dovrebbe essere in grado di raggiungere lo stesso grado di accuratezza, copiando i primi n strati ed eseguendo un identity mapping per l'ultimo strato. Allo stesso modo, reti di $n+2,\ n+3,\ n+4$ strati possono, con lo stesso metodo, ottenere la medesima accuratezza. Tuttavia, al crescere della profondità della rete questo non è sempre vero.

Questo avviene poiché nelle deep Neural Network si riscontrava il problema di annullamento del gradiente, la cui discesa, data dalla minimizzazione della funzione di errore tramite l'uso della backpropagation andava diminuendo sempre di più attraverso gli strati rendendo i segni degli errori talmente piccoli da non permettere alla rete di apprendere.

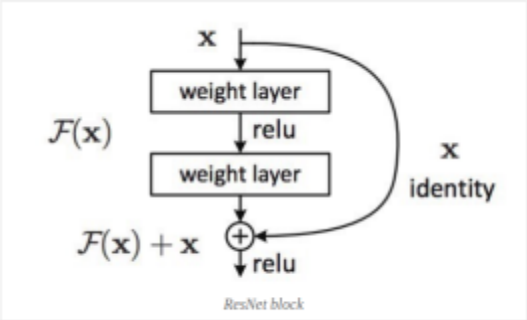


Fig. 11 Dettaglio della ResNet

Gli sviluppatori di ResNet hanno quindi pensato di introdurre una nuova strategia che fa uso dell'apprendimento residuo, ovvero, anziché imparare alcune features, si cerca di apprendere il residuo, inteso come la sottrazione della feature appresa dall'input di quel livello. Grazie all'innovazione introdotta da ResNet, si possono costruire reti di innumerevoli strati (potenzialmente anche più di mille) che abbiano un elevato grado di accuratezza.

Esperimenti e Risultati

LeNet

La prima rete allenata è stata la LeNet. Per allenarla abbiamo dovuto sottoporre le immagini del del dataset ad una reshape 100 x 100 e renderle a scala di grigi.

La rete è stata allenata con questi parametri :

- Immagini 100 x 100
- Learning rate di 0.01
- Momentum di 0.99

con il dataset inizialmente composto da:

- 674 immagini di Train
- 290 immagini di Test

In seguito ad una fase di training durata circa 5 minuti il risultato era il seguente:

Ovvero:

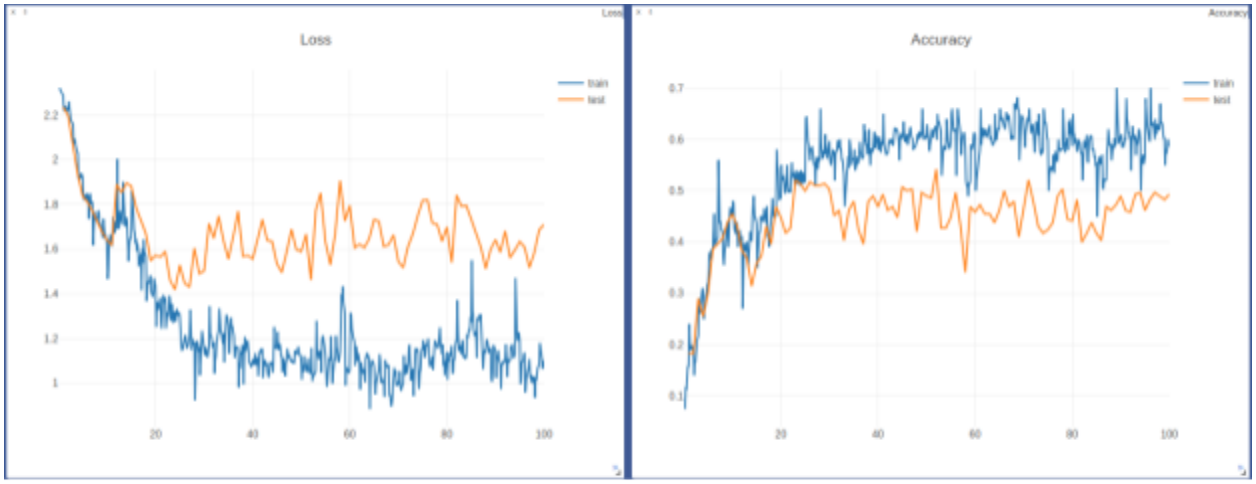


Fig. 12 Grafico Loss e Accuracy LeNet

- Accuarcy di training: **0.6084**
- Accuarcy di test: **0.4759**

Poiché i risultati ottenuti non sono stati soddisfacenti, ci siamo soffermati sul grafico e abbiamo notato come da una certa epoca in poi le curve di accuracy Train e Test si separavano nettamente.
Questo poteva essere dovuto a un dataset troppo piccolo e poco vario.
Abbiamo quindi deciso di aumentare il dataset estraendo frame da video della durata di 10/15 minuti per classe.
Rieseguendo il modello con la seguente partizione:

- 18000 immagini di Train
- 9000 immagini di Test
- 3000 immagini di Validation

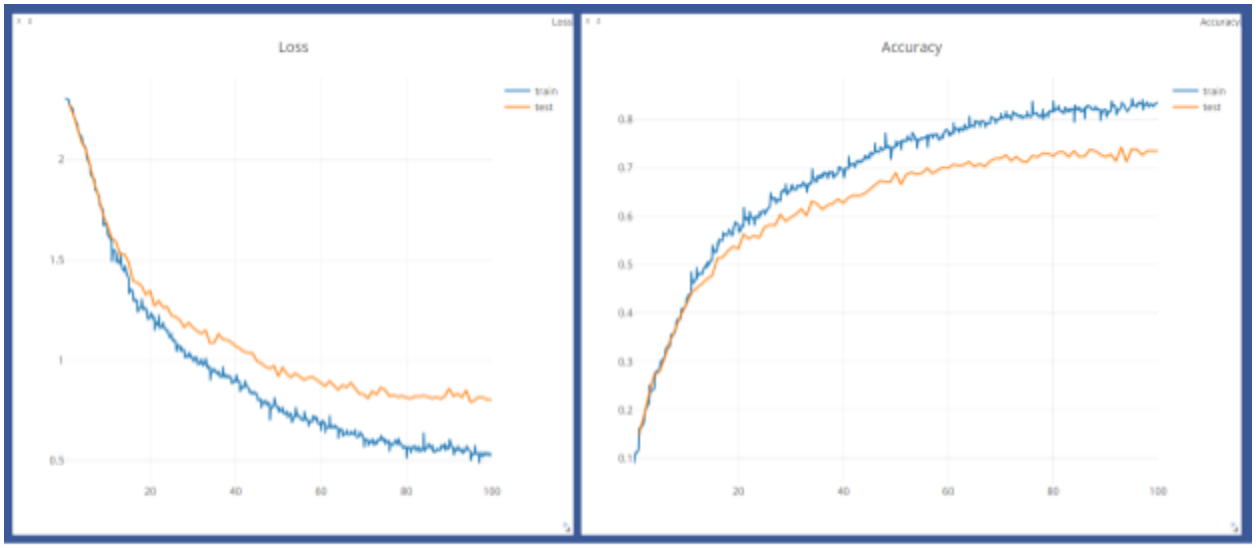
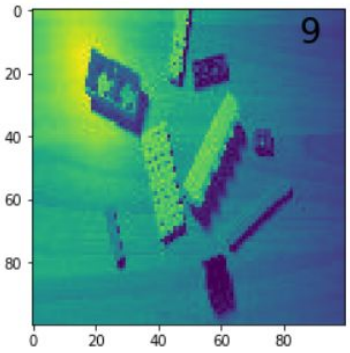


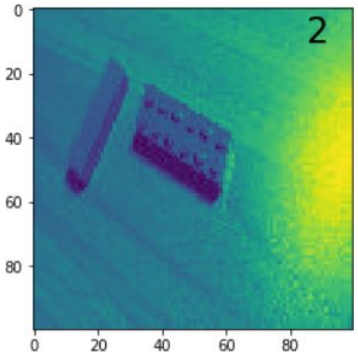
Fig. 13 Grafico LeNet dopo aumento del dataset

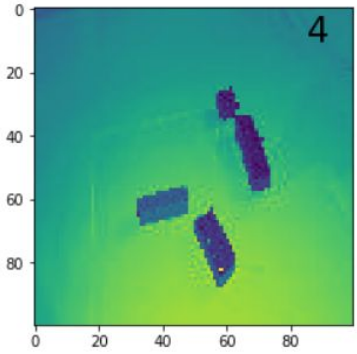
Sono stati ottenuti i seguenti risultati:

- Accuarcy di training: **0.833**
- Accuarcy di test: **0.73**

Di seguito ecco alcuni plot di classi stimate sulle immagini di Test:







Ottenendo quindi dei risultati accettabili.

VGG16

VGG16 in termini di accuracy rappresenta il modello dal quale sono stati ottenuti i migliori risultati. Per la fase di training di questo modello abbiamo dovuto ridimensionare il dataset con immagini 224 x 224 pixel.
Anche qui la partizione del dataset è stata la seguente:

- 18000 immagini di Train
- 9000 immagini di Test
- 3000 immagini di Validation

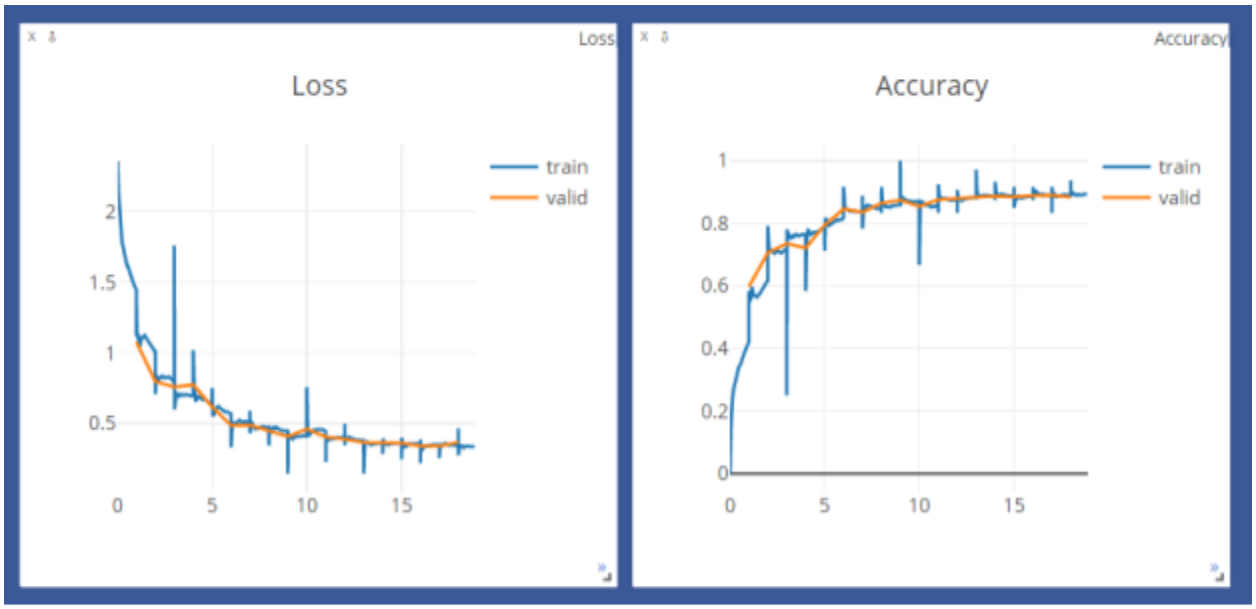


Fig. 14 Grafico Loss e Accuracy VGG16

I risultati ottenuti sono i seguenti:

- Accuarcy di training: **0.8948**
- Accuarcy di validation: **0.9023**

Utilizzando il checkpoint della VGG e trainando per altre 15 epoche (15 ore circa) il risultato ottenuto è il seguente:

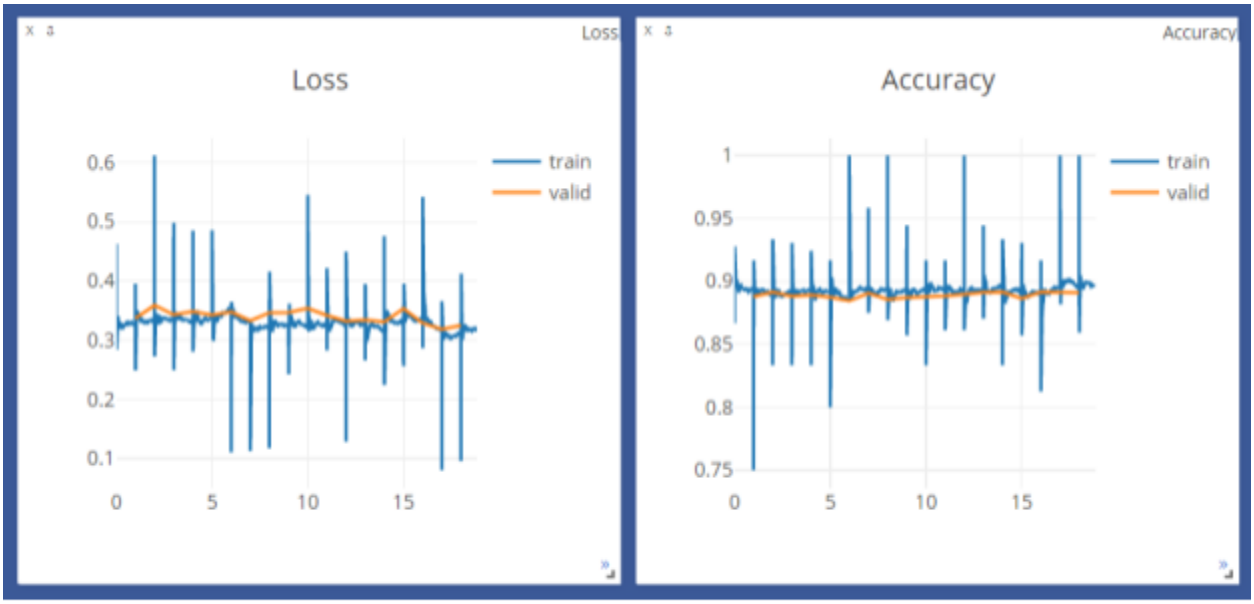


Fig. 15 Grafico Loss e Accuracy VGG16 usando il checkpoint

Da questo abbiamo dedotto che il limite della VGG16 era 0.9023 ottenuto in precedenza.

SqueezeNet1_0

La SqueezeNet dai test svolti si è dimostrata un ottimo compromesso tra velocità ed efficienza. Tale modello infatti ha impiegato molto meno tempo e risorse rispetto ad altri modelli quali VGG e ResNet, ottenendo tuttavia buoni risultati.
Anche qui la partizione del dataset è stata la seguente:

- 18000 immagini di Train
- 9000 immagini di Test
- 3000 immagini di Validation

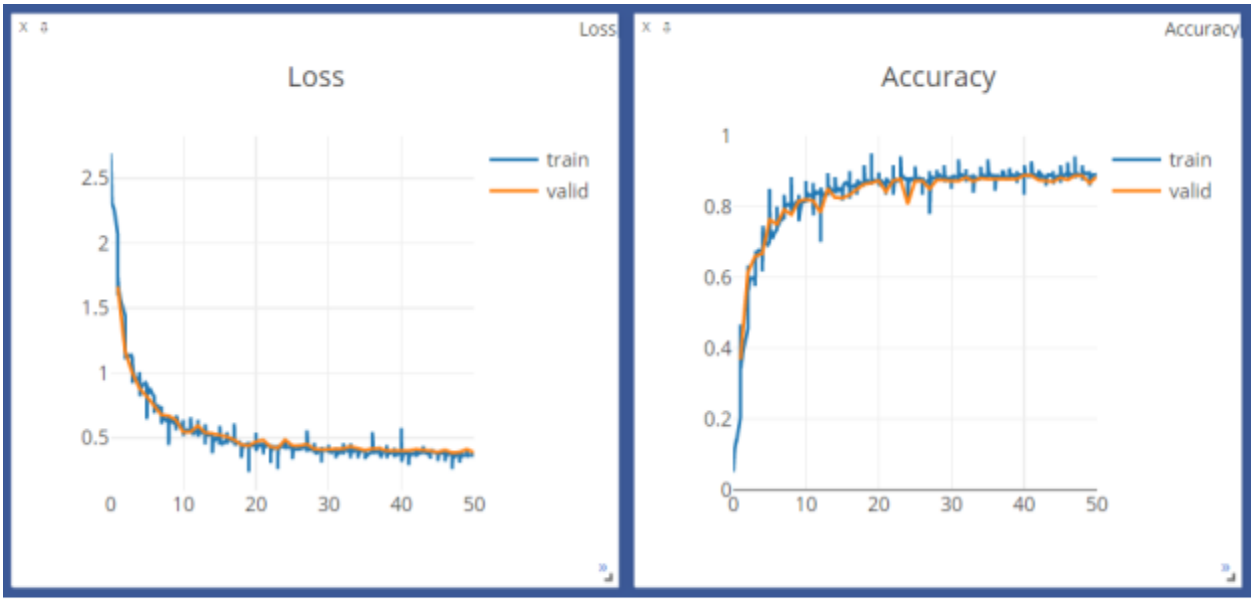


Fig. 16 Grafico Loss e Accuracy SqueezeNet1_0

I risultati ottenuti sono i seguenti:

- Accuarcy di training: **0.8925**
- Accuarcy di validation: **0.8937**

ResNet-50

Dopo aver consultato la tabella relativa ai Top-1-Error sui modelli di Torchvision visionabile nella documentazione ufficiale:

https://pytorch.org/docs/stable/torchvision/models.html

L'ultimo modello scelto per svolgere la nostra analisi è stato il modello ResNet50.

Tale modello è risultato forse il modello più difficile su cui svolgere l'operazione di training sia per quantità di risorse richieste (in particolare la GPU, costringendoci ad utilizzare batch_size veramente piccole) che per quantità di tempo impiegato

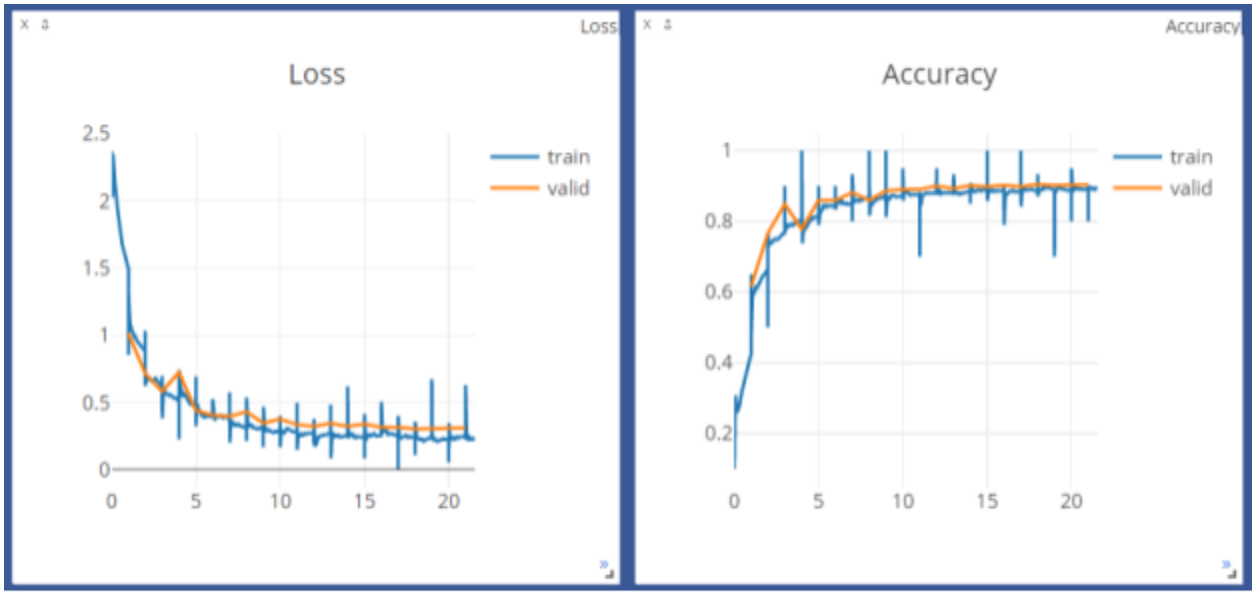


Fig. 17 Grafico Loss e Accuracy ResNet50

I risultati ottenuti sono i seguenti:

- Accuarcy di training: **0.8951**
- Accuarcy di validation: **0.9000**

Riepilogo risultati

Di seguito una tabella riassuntiva dove vengono mostrati e messi a confronto i risultati ottenuti sui vari modelli scelti:

Nome modello	***Accuracy sul train***	***Accuracy sul test***
LeNet	**0.83**	**0.73**
VGG16	**0.89**	**0.9023**
SqueezNet1_0	**0.89**	**0.89**
ResNet50	**0.89**	**0.9000**

Si può notare come, seppur per una differenza minima, il modello che ha mostrato le prestazioni migliori è stato quello allenato tramite la rete VGG16.

Conclusione

Il task assegnatoci riguardava lo sviluppo di un modello il cui scopo è quello di classificare correttamente immagini raffiguranti blocchetti Lego disposti casualmente in un piano di lavoro. Tali immagini erano rappresentative di 10 classi a seconda della numerosità dei mattoncini in foto.

La prima azione da noi svolta è stata la realizzazione del dataset. A tale scopo abbiamo deciso, iniziando dalla classe 1 e procedendo in ordine numerico, di disporre i blocchetti sul piano, scattare una foto, cambiare la disposizione dei blocchetti e ripetere il procedimento. Questo tipo di approccio però è stato particolarmente dispendioso soprattutto in termini di tempo, costringendoci a poter realizzare solamente 200 foto per classe. Dopo aver eseguito infatti alcuni test con il dataset prodotto, come spiegato sopra, riscontrando risultati non incoraggianti, abbiamo deciso di cambiare approccio relativamente alla realizzazione del dataset. Abbiamo dunque registrato circa 15 minuti di materiale video rappresentate una scena con blocchetti disposti sul piano e muovendo progressivamente la telecamera. Da tali video abbiamo estratto i frame che ci hanno permesso di arrivare a 3000 foto per classe. Questo nuovo dataset utilizzato nei test ha aumentato significativamente il tempo impiegato nella fase di training ma ci ha permesso di raggiungere risultati decisamente migliori rispetto ai precedenti. Tutto ciò, ha permesso di capire come un buon dataset, ben formato e abbastanza ampio, sia un punto di partenza essenziale per svolgere qualsiasi tipo di task.

Visti i dati ottenuti dai vari modelli abbiamo deciso di utilizzare **VGG16**. Una volta allenata la rete, svolgendo alcuni test e sottoponendole immagini “nuove”, è stato possibile elaborare alcuni spunti di riflessione. Abbiamo notato infatti come il modello si dimostrasse molto preciso con foto mai viste prima che tuttavia ricreavano le stesse condizioni delle foto su cui era stato allenato. In particolare, ad esempio, mostrava accuracy molto alte in presenza di foto ritraenti mattoncini con colori “già visti” e sfondo bianco mentre invece non riusciva ad ottenere buone prestazioni con mattoncini di colori e sfondi “mai visti” in precedenza.

Dopo aver sviluppato una demo ci è stato possibile sottoporre al nostro modello anche immagini raffiguranti tipologie di oggetti differenti dai mattoncini Lego. Abbiamo notato come in presenza di oggetti con superfici regolari e monocolore il modello mostrava previsioni mediamente corrette un discreto numero di volte. Ecco un test svolto con una foto di monete:

Fig. 18 Schermata demo con oggetti non convenzionali

Demo

Infine, si è deciso di sviluppare una demo che permetta di contare i mattoncini Lego presenti nella foto presa in input utilizzando il modello VGG16 da noi allenato.

La demo è composta da due parti: Back-end e Front-end.

Il Back-end è stato scritto in python utilizzando il framework Flask. Il server sta in ascolto e quando riceve una chiamata Post dal client esegue le trasformazioni necessarie sull'immagine ricevuta (le stesse applicate durante la fase di training) e la passa al modello. Il modello restituisce due valori: classe stimata e accuracy. Tali valori vengono poi inviati al client.

```
def evaluate():
    im = Image.open('ImageTest/'+ 'brik' +'.jpg')

    im = im.resize((224, 224), Image.NEAREST)



    img = transforms.ToTensor()(im)
    img = img.view(1, *img.shape)
    y = model(img)

    y = F.softmax(y,dim=1)

    p,c= y.view(-1).max(0)
    classe = int(c.item())
    prob = float(p.item())
    print(p,c,int(classe))

    return classe, prob
```

Il front-end è stato scritto in React per essere multipiattaforma. E' costituito da un unico componente che permette di accedere alla webcam o camera dello smartphone o computer e può essere eseguito in due modalità: Manuale o Real-time. Nella modalità manuale è possibile contattare il server tramite pulsante e quindi eseguire una stima per

volta, nella modalità Real-time invece il client contatta in automatico il server aggiornando i valori mostrati.

Qui sotto vengono mostrate le due varianti dell'interfaccia a seconda del dispositivo utilizzato:

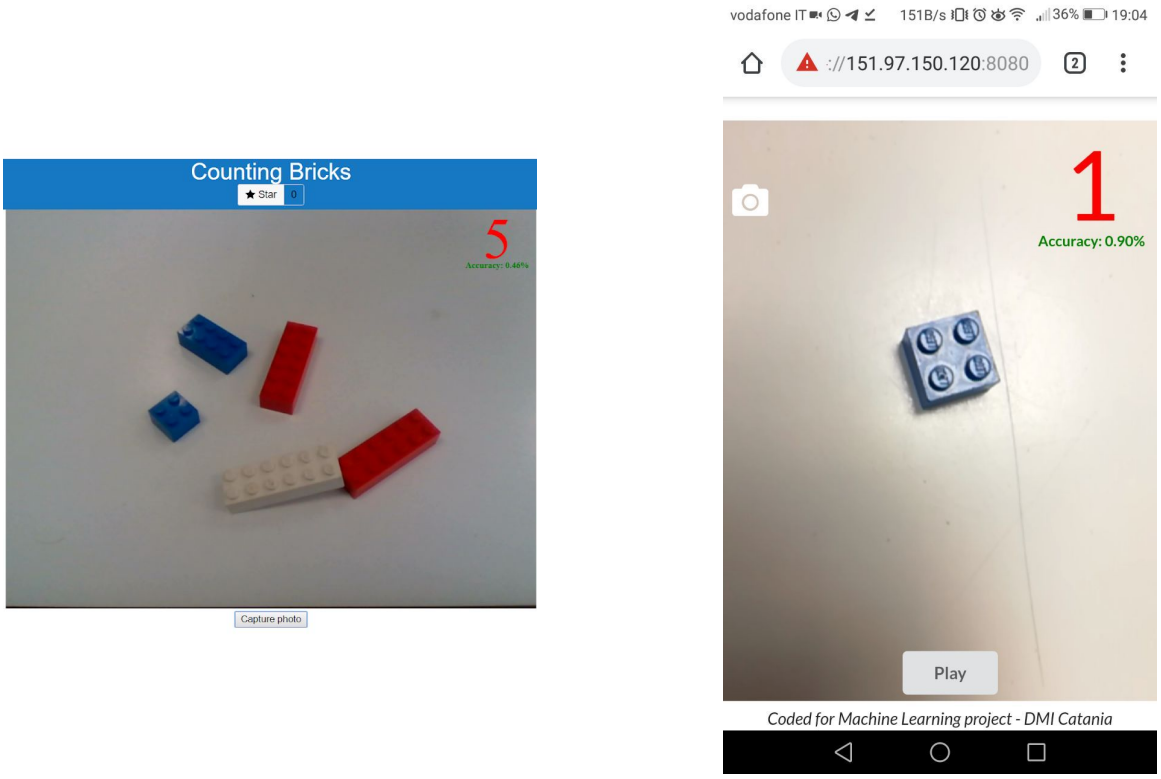


Fig. 19 Schermate della demo formato Desktop e Mobile