



UNIVERSITÀ  
degli STUDI  
di CATANIA

---

# RELAZIONE LABORATORIO MACHINE LEARNING

---

A cura di Leocata Danilo - 1000022576

LM-18 A.A 2020/2021

DOCENTI:

Giovanni Maria Farinella, Antonino Furnari

## Introduzione

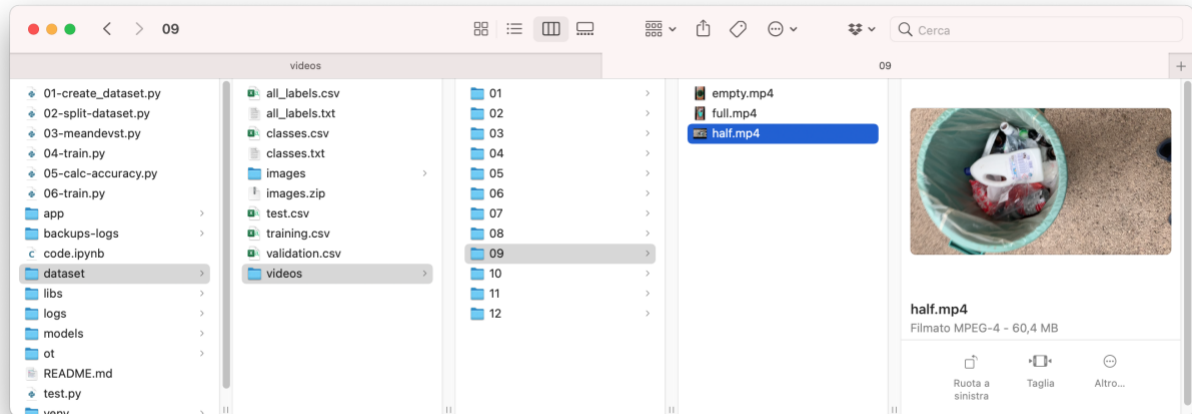
Il task assegnato è realizzare un classificatore che permetta di classificare la pienezza di bidoni di varie dimensioni utilizzati per la raccolta della spazzatura, e che in particolare sia in grado di distinguerli in tre categorie: pieno, vuoto o a metà.

Visto l'onere computazionale e i lunghi tempi di attesa, in una prima fase dello sviluppo è stato utilizzato Google Colab; tuttavia, a causa dei limiti di utilizzo imposti sia dalla versione gratuita che da quella a pagamento (dopo 15 giorni, l'utilizzo della GPU è stato limitato a 3 ore giornaliere, dopo circa una settimana di blocco), il training dei modelli trattati nella seguente relazione è stato effettuato in locale su MacBook Pro (13-inch, 2019), con processore Intel Core i5 quad-core da 1,4GHz e RAM da 8GB.

Il codice è stato pushato interamente su [GitHub](#).

## Dataset

Per la realizzazione del dataset sono stati registrati un totale di 33 video, ognuno della durata di circa 1 minuto a 30fps con risoluzione 1080, a dei bidoni della spazzatura di dimensioni diverse, contenenti principalmente plastica. Le riprese sono state realizzate con iPhone XS (tutte eccetto una, girata con Huawei P20) con una prospettiva che, partendo dall'alto, effettuava un movimento rotatorio attorno al contenitore. I video sono stati poi raggruppati in undici cartelle, al cui interno sono state inserite le riprese contenenti le tre fasi del 'riempimento' del secchio.



Successivamente, sono stati estratti 400 frames da ogni video ( [codice](#) ) per un totale di 13200 immagini. Contemporaneamente all'estrazione, è stato inizializzato il file `all_labels.txt` dove per ogni riga vi è la coppia (`nome_file`, `classe di appartenenza`).

```
dataset — -zsh — 93x21
Last login: Sun Aug 22 15:36:59 on ttys001
[daniloleocata@lozio ~ % cd Documents/GitHub/trashbin-classifier
[daniloleocata@lozio trashbin-classifier % cd dataset
[daniloleocata@lozio dataset % cat all_labels.txt
trashbin_0.jpg, 0
trashbin_1.jpg, 0
trashbin_2.jpg, 0
trashbin_3.jpg, 0
trashbin_4.jpg, 0
trashbin_5.jpg, 0
trashbin_6.jpg, 0
trashbin_7.jpg, 0
trashbin_8.jpg, 0
trashbin_9.jpg, 0
trashbin_10.jpg, 0
trashbin_11.jpg, 0
trashbin_12.jpg, 0
trashbin_13.jpg, 0
trashbin_14.jpg, 0
trashbin_15.jpg, 0
trashbin_16.jpg, 0
```

Il file zip contenente il dataset è stato omissso dall'upload su GitHub, in quanto troppo pesante, ma è disponibile su [Google Drive](#).

Il dataset è stato suddiviso randomicamente in training, test e validazione, utilizzando la funzione `train_test_split`, messa a disposizione da `sklearn`. Successivamente, si è accertata in un [notebook](#) la corretta distribuzione tra classi per tipologia di dataset. In sintesi:

Training	Validation	Test
6600	2640	3960

Per utilizzare il dataset è stata implementata la classe [TrashbinDataset](#), mentre per rendere il codice più sintetico, è stata implementata la classe [TDContainer](#) (Trashbin Dataset Container) che permette di gestire dataset di validazione, test e training in un unico oggetto e che consente di istanziare anche i rispettivi `DataLoader`

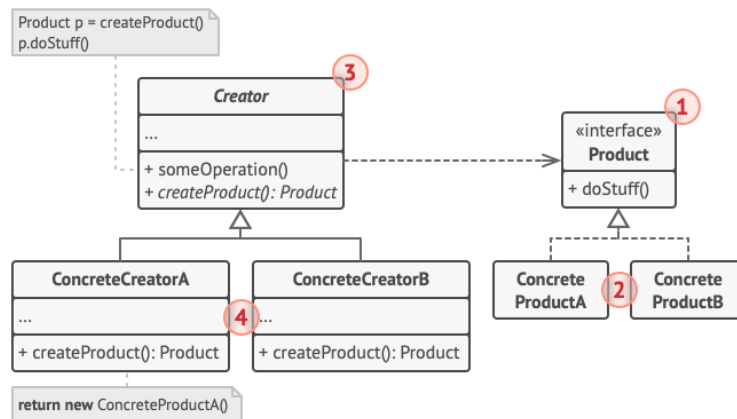
## Transfer Learning

La complessità computazionale e le poche risorse hanno indotto a impiegare opportunamente tecniche di transfer learning, attraverso le quali si è utilizzato un modello già allenato su uno specifico task, riutilizzato come punto di partenza per un modello su un altro.

Nello specifico, si sono presi come riferimento alcuni modelli pre-trained messi a disposizione da `pytorch` ( [torchutils.models](#) ). Tutti i modelli sono stati precedentemente allenati su [ImageNet](#), un dataset che consiste in più di 14 milioni con 1000 classi. È necessario che tutti i modelli pre-trained prendano in input tensori di dimensioni  $3 \times 244 \times 244$ ; le immagini in input devono inoltre essere normalizzate nel range  $[0,1]$  con media  $[0.485, 0.456, 0.406]$  e deviazione standard  $[0.229, 0.224, 0.225]$ .

Data la moltitudine di modelli messi a disposizione, sono stati scelti i più leggeri, onde evitare problemi di sovraccaricamento del PC e lunghi tempi d'attesa.

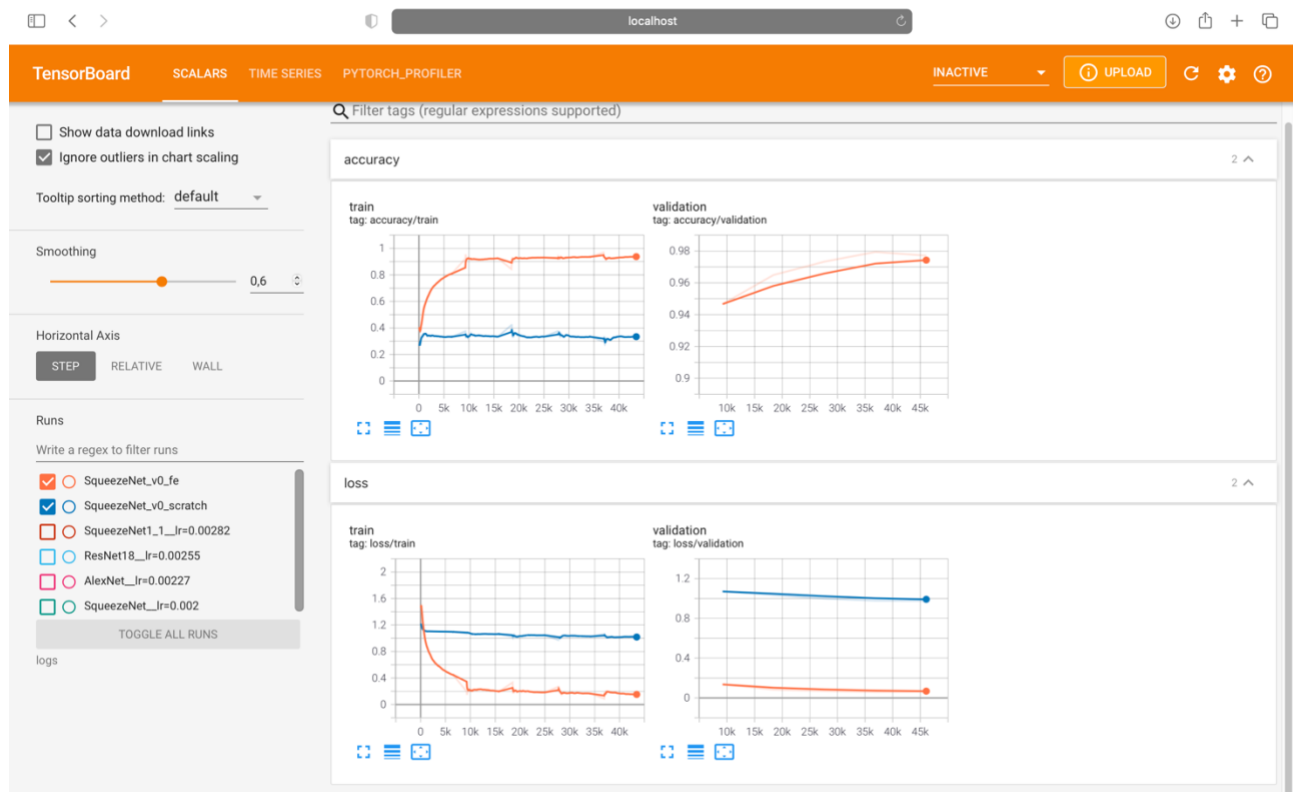
Per rendere il codice robusto, riusabile e di facile mantenimento e, data la fondamentale necessità di testare diversi modelli, ognuno con un'implementazione diversa dall'altra, è stato implementato il Factory Method, un design patter creazionale, nella classe [PretrainedModels](#).



Tra le tecniche di transfer learning utilizzabili, normalmente si ha la possibilità di scegliere:

- **Fine-tuning**: dato un modello pre-trained, effettuando un nuovo training, tutti i parametri del modello saranno aggiornati (come se partisse da zero);
- **Feature-extraction**: partendo da un modello pre-trained, verranno aggiornati solo i pesi dei layer finali, ovvero quello modificato. La CNN pre-trained sarà utilizzata come feature-extractor e cambieranno solo i livelli di output.

Per scegliere la migliore tra le due tecniche, e notarne sin da subito le differenze, è stato preso un modello dallo zoo di torchvision, ed è stato quindi effettuato un training di cinque epoche sul dataset, senza utilizzo di DataAugmentation.



Si è notato sin da subito il vantaggio dell'utilizzo del feature-extraction, anche in sole cinque epoche: infatti il modello da scratch, oltre ad avere un accuracy molto più bassa rispetto all'altro, ha impiegato circa il doppio per completare l'esecuzione del training (37 minuti contro 18).

## Dimensione dei batch e learning rate

Il dataset di training è composto da 13200 immagini, la dimensione del batch è di 32 (ogni epoca contiene 412,5 batch di default e non vengono considerati i batch non completi) con numero di worker a due, per evitare di sovraccaricare la RAM ed evitare che i processi vengano chiusi inaspettatamente.

Per trovare il learning rate migliore per ogni modello (ed ottenere, secondariamente, una stima approssimata di attesa del training su 100 epoche), è stata utilizzata la tecnica del Cyclical LR<sup>1</sup> utilizzando la libreria [LRFinder](#). In questo modo sono stati presi in esame tutti modelli e sono stati scelti i modelli più opportuni in base al tempo di attesa (scartando quelli che sono risultati troppo onerosi, in quanto durante varie fasi dello sviluppo hanno portato ad errori dopo varie ore di attesa, a causa della poca RAM disponibile). Il [dataset](#) con cui è stato utilizzata la funzione è lo stesso sul quale si è effettuato il training successivamente.

I risultati ottenuti sono i seguenti:

Modello	Tempo di attesa per 100 epoche	LR trovato
AlexNet	1:54:49	0.00227
SqueezeNet 1.0	3:21:12	0.002
SqueezeNet 1.1	2:10:26	0.00282
ResNet18	4:20:14	0.00255
VGG11	14:20:36	SKIPPED
VGG11 with batch normalization	26:07:80	SKIPPED
DenseNet121	12:34:44	SKIPPED
MobileNetV2	5:56:06	SKIPPED
InceptionV2	14:33:29	SKIPPED

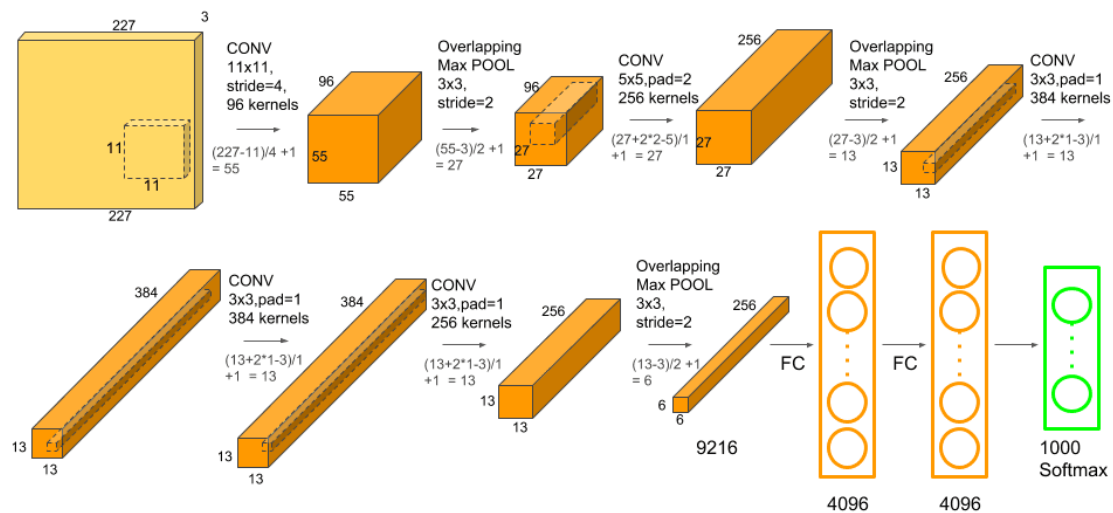
I modelli presi in esame sono stati dunque: AlexNet, SqueezeNet e ResNet18 (per quest'ultimo si saltano le implementazioni con layer più profondi, in quanto richiederebbero molto più tempo).

---

<sup>1</sup> <https://arxiv.org/abs/1506.01186>

## AlexNet<sup>2</sup>

Realizzata da Alex Krizhevsky ed i suoi collaboratori nel 2012, per partecipare alla competizione ImageNet LSVRC-2010.



Il modello prende in input tensori 224 x 224 x 3, è formato da 11 layer che utilizzano la convoluzione con kernel 11x11, 5x5 e 3x3. Prima di passare al layer successivo, dopo l'operazione di convoluzione, viene applicata la ReLu come funzione di attivazione. Sono implementati layer di max-pooling, combinato allo zero-padding subsampling. Gli ultimi tre layer sono fully connected. L'intera rete ha 62,378,344 parametri

<sup>2</sup> <https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf>

## ResNet18<sup>3</sup>

Implementare una convolutional neural network con molti livelli dovrebbe permettere di diminuire l'errore di training; tuttavia, nella pratica, l'aggiunta di molti livelli fa sì che, con l'aumento della profondità della rete, l'accuracy saturo e quindi degrada rapidamente. Il degrado dell'accuracy del training indica che non tutti i sistemi sono uguali e facili da ottimizzare.

ResNet (esistono differenti implementazioni, nel nostro caso si è presa in esame quella con 18 layer), proposta nel 2015 da Microsoft, risolve queste problematiche introducendo una nuova architettura chiamata Residual Network.

Una deep residual network è simile a una rete classica: utilizzando però la tecnica delle skip-connections, grazie alla quale è possibile saltare l'addestramento di alcuni livelli con l'uso delle skip-connection o delle residue, la funzione identità può essere 'imparata' facendo affidamento solo alle connessioni 'skip'.

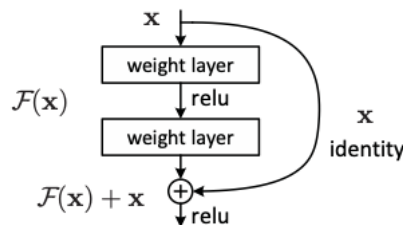


Figure 2. Residual learning: a building block.

Consideriamo un blocco di rete neurale il cui input è  $x$ , e vorremmo conoscere la vera distribuzione  $H(x)$ . Indichiamo la differenza (o il residuo) tra questo come:

$$R(x) = \text{Output} - \text{Input} = H(x) - x$$

Dunque:

$$H(X) = R(x) + x$$

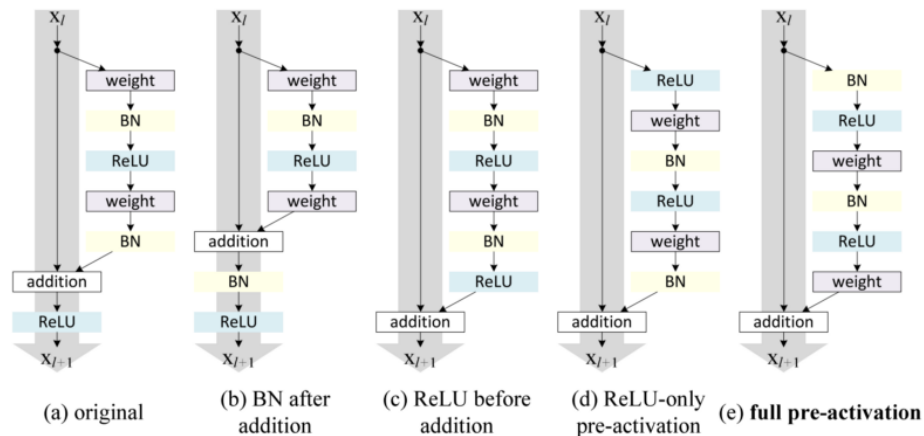
In sintesi, i layer di una rete tradizionale imparano il vero output  $H(x)$ , mentre gli strati in una rete residua imparano dal residuo  $R(x)$ : da qui il nome residual block.

È stato osservato che è più facile apprendere il residuo di output e input, piuttosto che solo l'input. Come ulteriore vantaggio, la nostra rete può apprendere la funzione di identità semplicemente impostando il residuo a zero. Grazie alle 'skip-connections', possiamo propagare gradienti più grandi ai livelli iniziali, e anche questi potrebbero imparare come velocemente come gli strati finali, dando la possibilità di addestrare reti più profonde.

---

<sup>3</sup> <https://arxiv.org/abs/1512.03385>

Ecco un esempio di tipi diversi di blocchi residui:



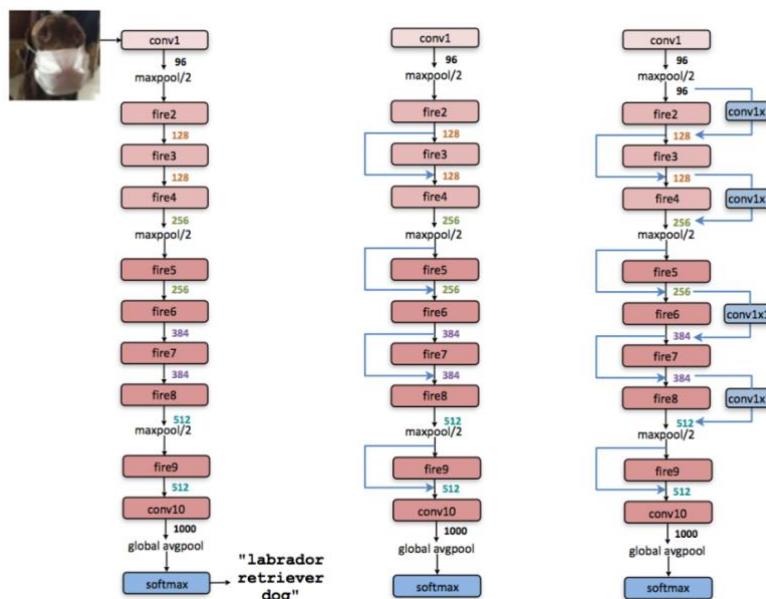
Esistono ancora più interpretazioni di residual-blocks e ResNet, oltre a quelle già discusse sopra. Durante l'addestramento di ResNet, addestriamo i livelli in blocchi residui oppure saltiamo l'addestramento per quei livelli, utilizzando le skip-connections. Quindi, parti diverse delle reti verranno addestrate a velocità diverse (come addestrare un insieme di modelli diversi sul set di dati e ottenere la massima accuracy possibile). Anche saltare il training in alcuni strati di blocchi residui può essere considerato da un punto di vista ottimistico poiché, in generale, non conosciamo il numero ottimale di livelli (o residual-blocks) necessari per una rete neurale. In sintesi, si permette pertanto alla rete di saltare l'addestramento per i livelli che non sono utili e che non peseranno sull'accuracy complessiva: in un certo senso, saltare le connessioni rende dinamiche le reti neurali per ottimizzare il numero di livelli durante l'allenamento.



## SqueezeNet<sup>4</sup>

È una piccola rete, nata dalla necessità di trovare un sostituto compatto di AlexNet. Ha circa 50x parametri meno rispetto ad AlexNet ed è circa 3 volte più veloce. Le principali strategie dietro SqueezeNet sono:

1. Sostituire i filtri 3x3 con filtri 1x1, poiché un filtro 1x1 ha 9 volte meno parametri di un filtro 3x3.
2. Diminuire il numero di canali di ingresso a filtri 3x3. Consideriamo un livello di convoluzione composto interamente da filtri 3x3. La quantità totale di parametri in questo livello è  $(\text{numero di canali di ingresso}) * (\text{numero di filtri}) * (3*3)$ . Quindi, per mantenere un piccolo numero totale di parametri in una CNN, è importante non solo diminuire il numero di filtri 3x3 (vedi 1, sopra), ma anche diminuire il numero di canali di ingresso ai filtri 3x3. Riduciamo il numero di canali di ingresso a filtri 3x3 utilizzando i livelli di compressione (punto successivo).
3. Eseguire il downsampling 'delayed' (in ritardo) nella rete in modo che i livelli di convoluzione abbiano mappe di attivazione grandi. In una rete convoluzionale, ogni livello di convoluzione produce una mappa di attivazione dell'output con una risoluzione spaziale che è almeno 1x1 e spesso molto più grande di 1x1. L'altezza e la larghezza di queste mappe di attivazione sono controllate da: (1) la dimensione dei dati di input (ad es. immagini 256x256) e (2) la scelta dei livelli in cui eseguire il downsampling nell'architettura CNN. Più comunemente, il downsampling è progettato nelle architetture della CNN settando (stride > 1) in alcuni dei livelli di convoluzione o pooling. Se i primi 3 livelli della rete hanno grandi passi, la maggior parte dei livelli avrà mappe di attivazione piccole. Al contrario, se la maggior parte dei livelli nella rete ha uno stride di 1 e gli stride maggiori di 1 sono concentrati verso l'estremità della rete, di conseguenza molti strati della rete avranno grandi mappe di attivazione: grandi mappe di attivazione (a causa del downsampling ritardato) possono portare a una maggiore precisione di classificazione, a parità di tutto il resto.



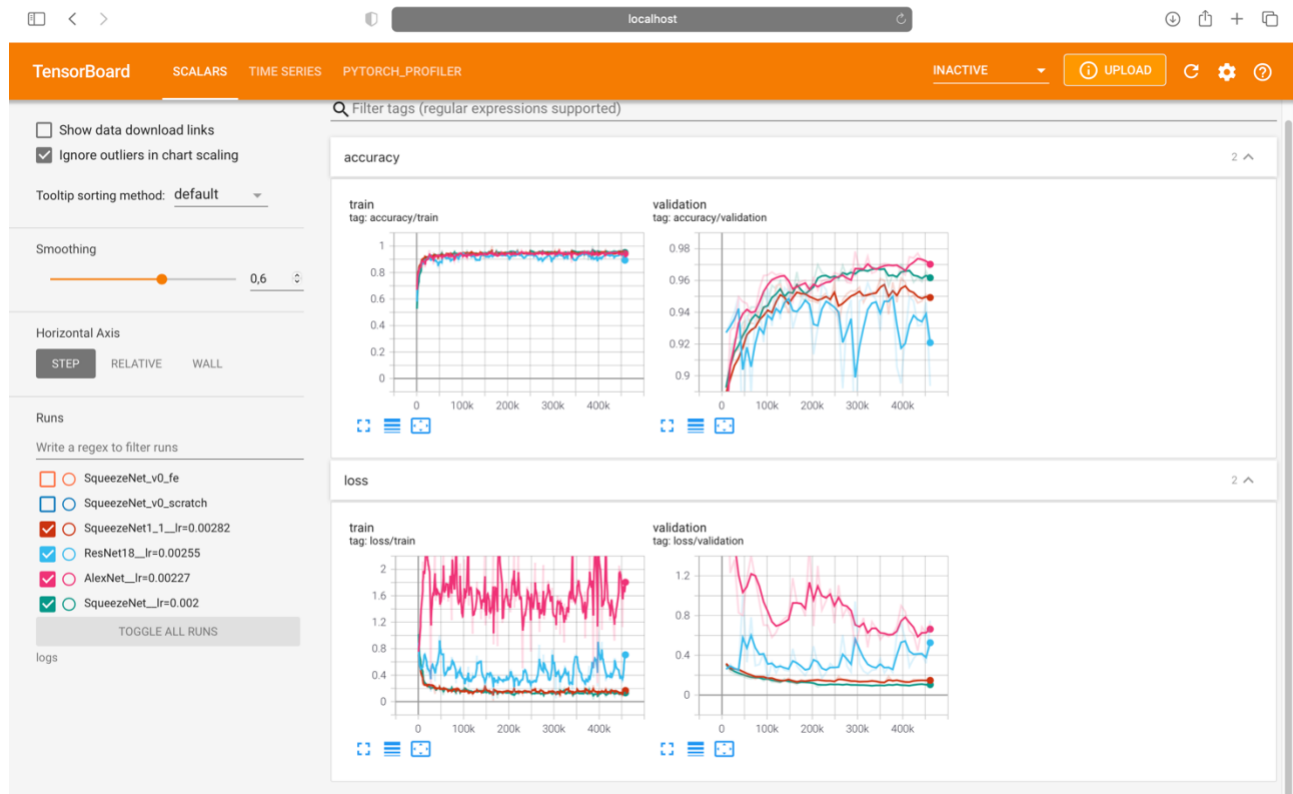
<sup>4</sup> <https://arxiv.org/abs/1602.07360>

Nel paper, è stato dimostrato che gli autori hanno applicato il downsampling 'delayed' a quattro diverse architetture CNN e in ogni caso il downsampling ritardato ha portato a una maggiore precisione di classificazione.

Le strategie 1 e 2 riguardano la diminuzione della quantità di parametri in una CNN mentre si cerca di preservarne l'accuracy. La strategia 3 riguarda la massimizzazione della precisione su un budget limitato di parametri: il modulo fire è l'elemento che ci consente di utilizzare con successo le strategie 1, 2 e 3. L'architettura è composta da layer "squeeze" ed "expand". Uno strato convoluzionale di compressione ha solo filtri  $1 \times 1$ ; questi vengono inseriti in uno strato di espansione che ha un insieme di filtri di convoluzione  $1 \times 1$  e  $3 \times 3$  (modulo fire).

## Procedura di training

È stato effettuato un [training](#) su 50 epoche sui modelli precedentemente citati, il risultato su tensorboard dopo varie ore è il seguente:

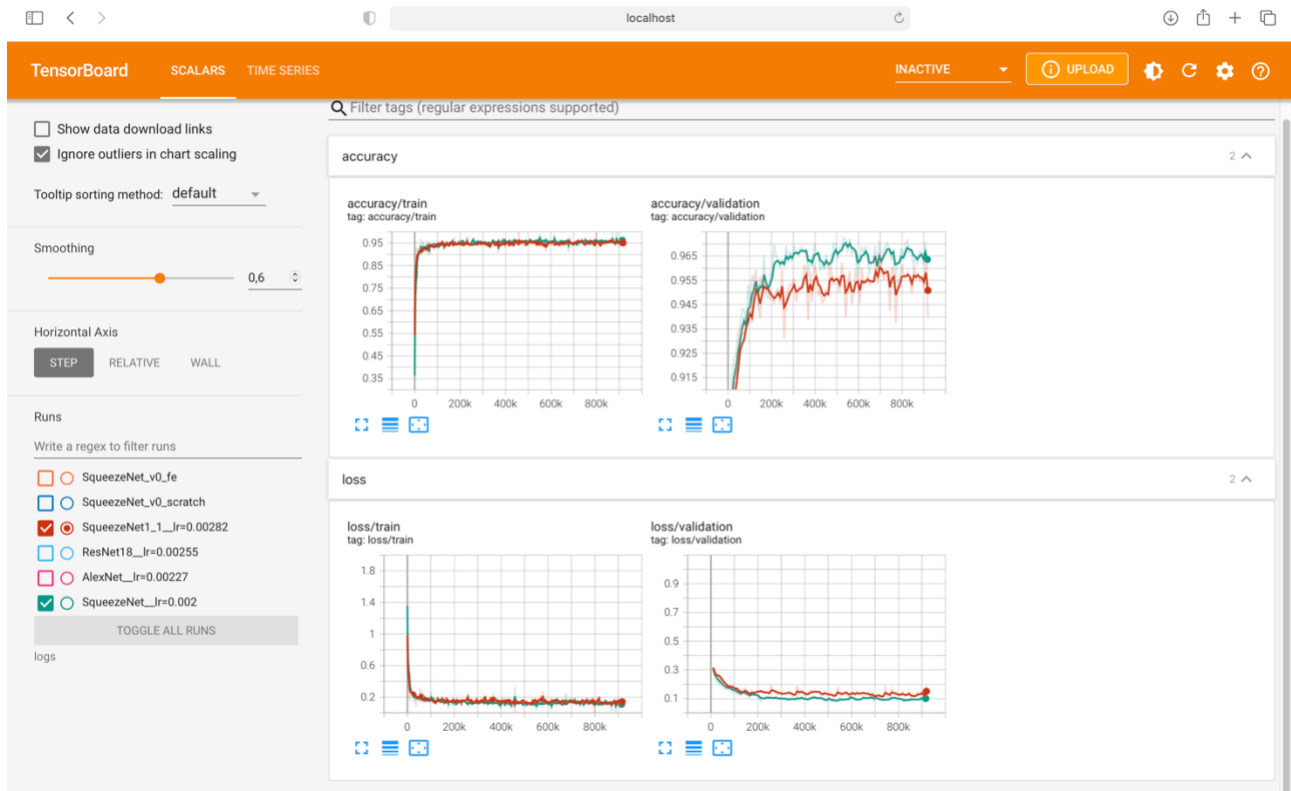


Il modello che ha conseguito risultati migliori è SqueezeNet, in entrambe le versioni; nonostante AlexNet e ResNet abbiano ottenuto un'accuracy migliore sul dataset di validation, la curva ha presentato overfitting<sup>5</sup>. La tabella di seguito riassume l'andamento del training per i quattro modelli presi in analisi:

Modello	Tempo di esecuzione	Accuracy score
AlexNet	3h 3m 8s	82.93%
ResNet18	7h 26m 46s	80.26%
SqueezeNet v 1.0	5h 35m 32s	86.69%
SqueezeNet v 1.1	3h 30m 43s	86.05%

<sup>5</sup> Visti i log, il training poteva essere stoppato prima, ma per completezza è stato completato

Dato il tempo di esecuzione, accettabile, di entrambe le versioni di SqueezeNet, è stato considerato opportuno continuare il training di entrambe per altre 50 epoche.



Il gap dell'accuracy (e della validazione, in un certo senso) dei due modelli è rimasto invariato: di conseguenza, per scegliere il modello con accuracy migliore, visto l'andamento del grafico (le ultime iterazioni l'hanno fatta abbassare), è stata calcolata l'accuracy di SqueezeNet 1.0 dei modelli salvati a partire dalla 50-esima epoca:

Epoche	Accuracy
50	86.76%
60	86.46%
70	86.18%
80	86.64%
90	86.99%
100	86.81%


I dati di sopra dimostrano che il modello non ha appreso più (o comunque molto poco) dopo la 50-esima epoca: una possibile soluzione, per incrementare l'accuracy, sarebbe quello di incrementare il dataset con ulteriori video, registrati con dispositivi diversi (per fare sì che il modello generalizzi di più) e soprattutto utilizzando secchi di dimensione diversi da quelli già presenti nel dataset.

## Interfaccia grafica

L'interfaccia grafica è stata realizzata mediante la libreria [Gradio](#): utilizza di default il modello SqueezeNet 1.0 allenato alla 90-esima epoca (sono stati omessi tutti gli altri per evitare di appesantire il repository). Gradio permette, inoltre, di editare l'immagine prima di effettuare la classificazione: in questo modo è possibile testare il modello su immagini di varie dimensioni, ritagliando la parte di interesse.

Per la fase di test da 'app', oltre a prendere immagini dal web, è stato chiesto ad amici fotografare, con prospettiva dall'alto, i raccoglitori della spazzatura utilizzati a casa (la cartella con le foto di test è stata caricata sul repository) in modo da ottenere risultati sul funzionamento del modello con immagini fornite da utenti.

DATA



Edit

Clear Submit

OUTPUT

half

half

65%

empty


22%

full

13%

Screenshot Flag

DATA



Edit

Clear Submit

OUTPUT

half

half

95%

full

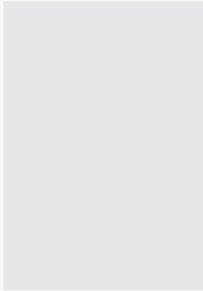

4%

empty

1%

Screenshot Flag

DATA



[Edit](#)

Clear

Submit

OUTPUT

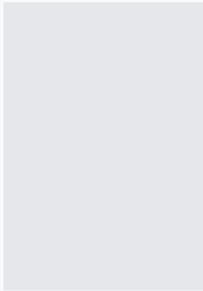

**empty**

empty	<div></div>	99%
half	<div></div>	1%
full	<div></div>	0%

Screenshot

Flag

DATA



[Edit](#)

Clear

Submit

OUTPUT

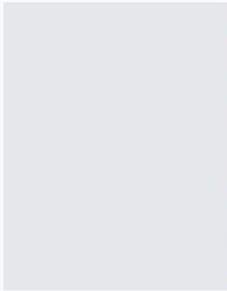

**empty**

empty	<div></div>	100%
half	<div></div>	0%
full	<div></div>	0%

Screenshot

Flag

DATA



[Edit](#)

Clear

Submit

OUTPUT

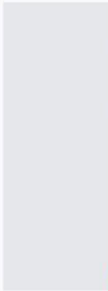

**full**

full	<div></div>	91%
half	<div></div>	7%
empty	<div></div>	2%

Screenshot

Flag

DATA



[Edit](#)

Clear

Submit

OUTPUT

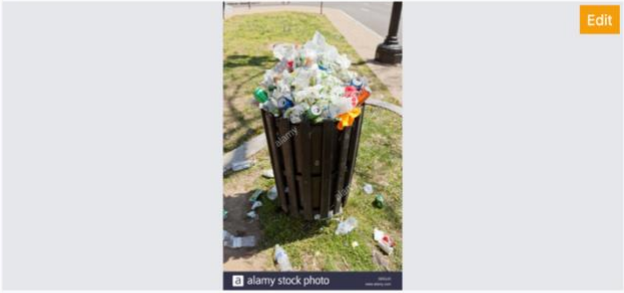
**half**

half	<div></div>	45%
full	<div></div>	37%
empty	<div></div>	18%

Screenshot

Flag

DATA



Clear

Submit

OUTPUT

full



Screenshot

Flag