
Università di Catania

# Trashbin Triplet Classifier

Progetto Deep Learning (LM-18)
Università degli Studi di Catania - A.A 2021/2022

Danilo Leocata
Docente: Giovanni Maria Farinella, Antonino Furnari

June 24, 2022

# 1 Introduzione

L'obiettivo dell'elaborato è implementare una procedura di *Metric Learning* per classificare la capienza rimanente di secchi della spazzatura in: pieno, vuoto, a metà.

Il dataset è stato preso da un precedente progetto (https://github.com/khalld/trashbin-classifier) ed è disponibile al seguente indirizzo:

https://drive.google.com/drive/folders/1LmN-fXWZ8UpRkLeMjbootN46V9AHaE4x.

Il progetto è stato implementato utilizzando `python v3.9.9` e `pytorch-lighting v1.6.3`. Il modello è stato allenato utilizzando un MacBook Pro (16-inch, 2019) con processore Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz, RAM: 16 GB 2667 MHz DDR4 e GPU AMD Radeon Pro 5300M 4 GB Intel UHD Graphics 630 1536 MB. Sfortunatamente, ad oggi, il modello di GPU non è supportato per l'accelerazione del training e di conseguenza è stato effettuato su CPU.

Il codice è ampiamente commentato, in particolare può essere diviso concettualmente in 3 parti:

1. `dataset.ipynb` Notebook Jupiter realizzato per mostrare il funzionamento delle funzioni implementate per adattare il dataset al task specifico
2. `training-script.py` Script utilizzato per effettuare il training del modello
3. `main.ipynb` Notebook Jupiter esplicativo, realizzato per visualizzare le performance ed effettuarne la valutazione (partendo da un `.ckpt`)

Nella repository ( https://github.com/khalld/triplet-trashbin-classifier ) è stato caricato solo il `.ckpt` del modello finale. Tutti i modelli ed i logs di tensorboard ottenuti sono stati caricati su Google Drive: https://drive.google.com/file/d/1vgALpclAQs7xSMj2BkQJxj770hXyJm4-

## 2 Architettura

Per il raggiungimento dell'obiettivo assegnato, è stato trovato opportuno l'utilizzo di una *Rete Triplet*, dato che l'obbiettivo è massimizzare la distanza inter-classe degli oggetti e quest'ultima dovrebbe permettere di ottenere un criterio di training più forte rispetto a quello delle reti siamesi.

Una rete di tipo Triplet ha un criterio di training più forte, in quanto, offre sempre un esempio positivo e negativo relativo al medesimo elemento di ancora. Inoltre, questo approccio dovrebbe garantire di massimizzare la distanza tra 'metà-vuoto' e 'metà-pieno' in modo migliore rispetto alla rete siamese.

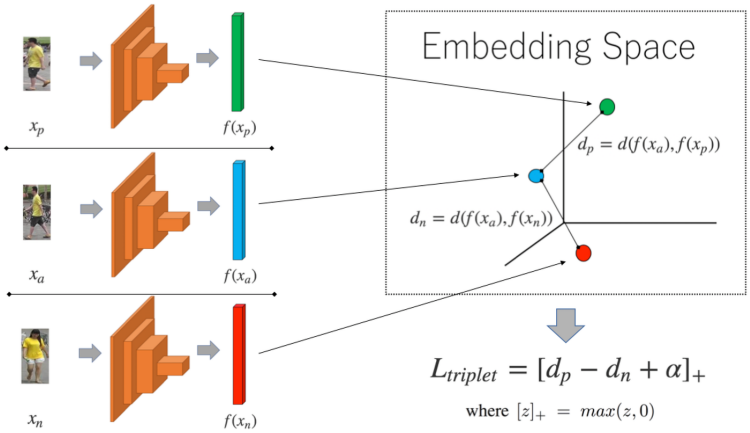


**Figure 1:** Esempio del risultato che si vuole ottenere

L'architetura di una rete Triplet è composta da tre rami identici, che condividono gli stessi pesi e mappano gli elementi in codici $\Phi(I_i)$, $\Phi(I_j)$, $\Phi(I_k)$.

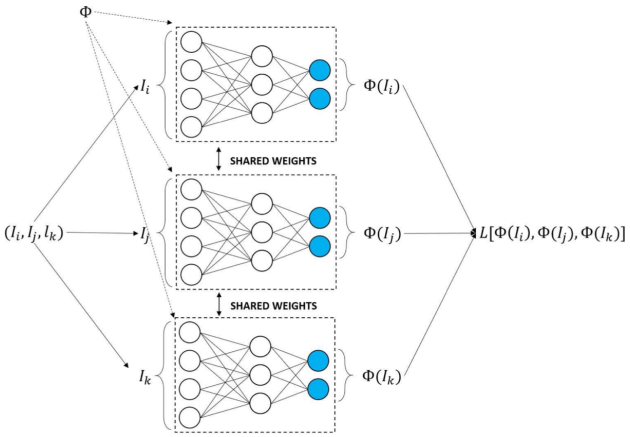


**Figure 2:** Architettura rete triplet

Prende in input una tripletta di elementi $(I_i, I_j, I_k)$ che sono:

- L'ancora $I_i$
- L'esempio positivo $I_j$ (che in breve ha la stessa classe di $I_i$)
- L'esempio negativo $I_k$, cioé un elemento diverso dalle classi di $I_i$, $I_j$

In breve, la distanza dall'ancora al positivo è minimizzata e la distanza dall'ancora al negativo è massimizzata. Il modello farà in modo che una coppia di campioni con le stesse etichette abbia una distanza inferiore rispetto a una coppia di campioni con etichette diverse.

## 2.1 Scelta del modello

Come feature extractor è stata utilizzata una **SqueezeNet** pre-trained senza il layer di classificazione.

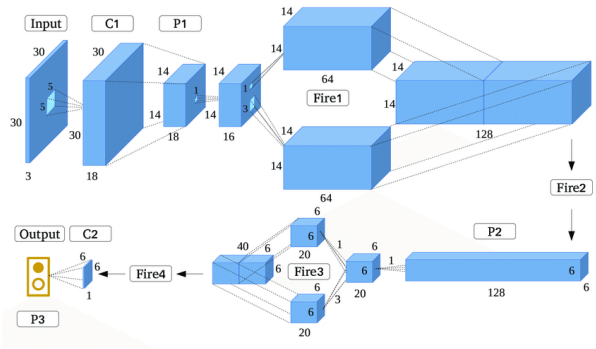


**Figure 3:** Architettura SqueezeNet

Nel progetto precedente sono stati presi in esame alcuni modelli pretrained, disponibili su `torchvision.models`, per tale motivo, è stato più conveniente utilizzare `SqueezeNet`, che aveva comunque ottenuto il miglior risultato in termini di tempo, esecuzione e validazione.

Inizialmente, nonostante la maggiore potenza computazionale rispetto allo studio precedente, sono state effettuate delle prove utilizzando `MobileNetV2`, ma quest'ultima richiedeva circa il doppio del tempo di `SqueezeNet`. In particolare, il completamento di un'epoca utilizzando `MobileNetV2` richiedeva 70 minuti contro i 35/40 di `SqueezeNet v1` per immagini a colori `224x224`. Approssimando ed effettuando un training di 60 epoche `SqueezeNet v1` impiegherebbe 30 ore contro le 70 di `MobileNetV2`.

## 2.2 Ottimizzazione del modello

Sono state utilizzate funzioni, presenti su `pytorch-lighting` per trovare automaticamente i parametri da usare per il training del modello. Nel dettaglio:

- **Batch Size Finder**: impiegato per evitare problemi in memoria durante il training. Viene utilizzato per trovare la dimensione batch più grande, che si adatta alla memoria. In questo caso il pc supportava batch di dimensioni fino a **6600**. I lotti di grandi dimensioni spesso producono una migliore stima dei gradienti, ma possono anche comportare tempi di addestramento più lunghi. Dopo diverse prove e data la dimension esigua del dataset è stato opportuno fissarla a **256**;
- **Learning Rate Finder**: selezionare un learning rate è essenziale sia per ottenere prestazioni migliori, che per una convergenza più rapida. Da documentazione, il *learning rate finder* esegue una piccola run dove il learning rate viene aumentato dopo ogni batch elaborato e viene registrata la loss corrispondente.

# 3 Dataset

Si è presentata la necessità di riadattare il dataset in triplette: ad ogni elemento di **ancora** verrà associato un elemento **positivo** ed uno **negativo**, che sarà scelto randomicamente (ad esempio, se l'ancora è della classe 'vuoto', l'elemento negativo sarà 'pieno' o 'mezzo'). Per questo, sono state implementate delle funzioni ad-hoc il cui utilizzo è documentato su `dataset.ipynb`.

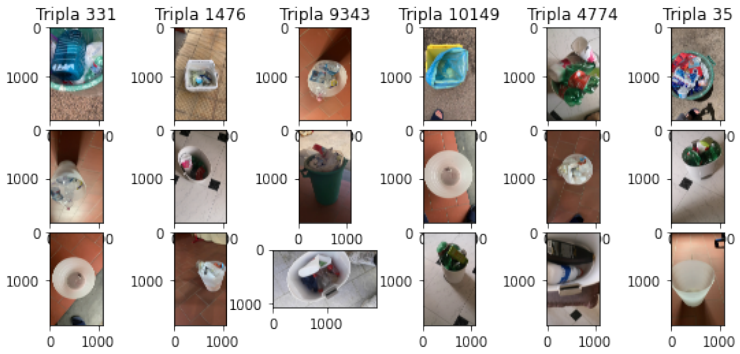


**Figure 4:** Dataset organizzato in triplette

Effettuando vari test, è stato trovato più efficiente salvare il dataset generato in `.csv`, evitando di memorizzare le triplette in memoria. Si nota che, per incrementare la **generalizzazione**, sarebbe utile avere a disposizione diversi `.csv` del dataset adattato ed alternare le varie versioni del dataset dopo un certo numero di epoche, in quanto, la funzione è stata implementata in modo tale che le triplette generate siano diverse ad ogni esecuzione del codice. Nel nostro caso, viene utilizzata una versione per le prime 30 epoche e successivamente viene cambiato ogni 15.

# 4 Scelta della loss

Per il training della rete siamese sono state prese in esame due loss differenti (disponibili su `torch.nn`)

## 4.1 Triplet margin loss

Prende in input i tensori $x_1$, $x_2$, $x_3$ ed un $margin$ con un valore $> 0$, la **Triplet margin loss** viene impiegata quando si vuole misurare una similitudine tra i samples. Una tripletta è composta da $a$ (ancora), $p$ (positivo) ed $n$ (negativo).

$$L(a, p, n) = \max\{d(a_i, p_i) - d(a_i, n_i) + margin, 0\}, \text{ dove}$$
$$d(x_i, y_i) = \|\|x_i - y_i\|\|_p$$

## 4.2 Triplet margin with distance loss

Prende in input i tensori $a$ (ancora), $p$ (positivo) ed $n$ (negativo) ed una funzione a valori reali non negativa chiamata *funzione di distanza d*. La **triplet margin with distance loss** può essere descritta dalla seguente formula:

$$l(a, p, n) = L = \{l_1, \ldots, l_N\}^T, l_i = \max\{d(a_i, p_i) - d(a_i, n_i) + margin, 0\}$$

dove:

1. $N$ è la dimensione del batch;
2. $d$ è una funzione non negativa a valori reali che quantifica la vicinanza di due tensori riferito alla funzione di distanza. Calcola la relazione tra l'ancora e l'esempio positivo (distanza positiva) e l'ancora e l'esempio negativo (distanza negativa);
3. $margin$ è un margine non negativo che rappresenta la differenza minima tra le distanze positive e negative;
4. Il tensore di input ha $N$ elementi, ognuno del quale può essere di qualsiasi forma, che la funzione di distanza può gestire.
5. Di default la funzione di distanza utilizzata è la Paiwrise Distance Funtion, che calcola la distanza tra i vettori $v_1$ e $v_2$ usando la p-norm:

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{\frac{1}{p}}$$

# 5 Training

Durante le prime fasi di training sono state effettuate delle prove con e senza **data augmentation**. Inizialmente, la data augmentation applicata era quella del progetto precedentemente citato, che, comunque, non ha fornito miglioramenti (si nota che nei grafici sottostanti, a 0 corrisponde la classe 'vuoto', ad 1 'metà' e a 2 'pieno'). Sono stati estratti randomicamente dei campioni dal dataset di test ed è stata effettuata una predizione sulle etichette utilizzando **Nearest Neighbor**. Le feature sono state estratte utilizzando varie versioni del modello allenato (utilizzando la variante con **Triplet Margin Loss**). Di seguito degli *embedding T-SNE*:

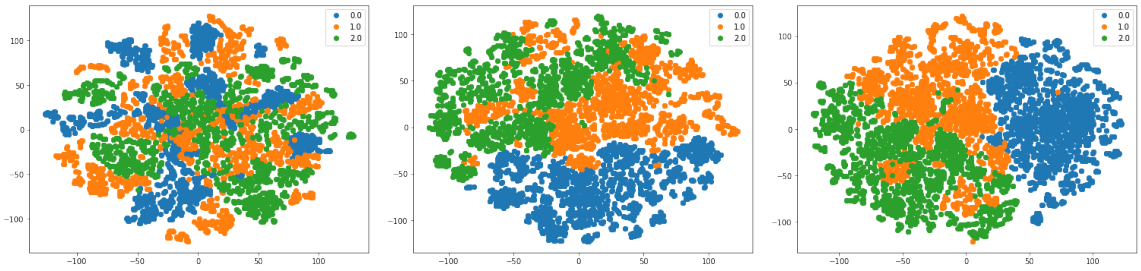


**Figure 5:** T-SNE embeddings **senza data augmentation** dopo 0, 10, 30 epoche di training

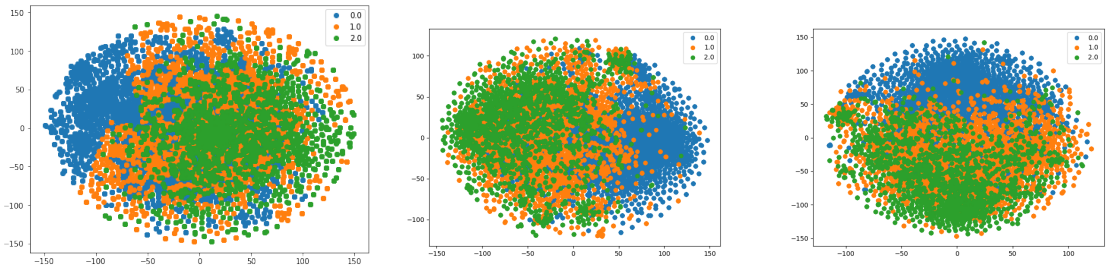


**Figure 6:** T-SNE embeddings **con data augmentation forte** dopo: 0, 30, 60 epoche di training

Dai grafici di sopra si evince che con una data augmentation (troppo) forte il modello non riesce a trasformare correttamente lo spazio di input per massimizzare la distanza inter classe (elementi diversi sono lontani) e minimizzare la distanza intra-classe (elementi simili sono vicini). Dopo numerose prove, è stato trovato efficiente, in modo da preservarne le feature estratte applicare solamente `RandomCrop()` e `RandomPerspective` al dataset di training e `CenterCrop` al dataset di test.

Gli errori classificazione ottenuti dalle loss due loss sono riassunti nella seguente tabella:

| Epoch | Triplet Margin Loss | Triplet Margin With Distance Loss |
|---|---|---|
| 0 | 0.677 | 0.739 |
| 15 | 0.936 | 0.792: |
| 30 | 0.900 | 0.894 |

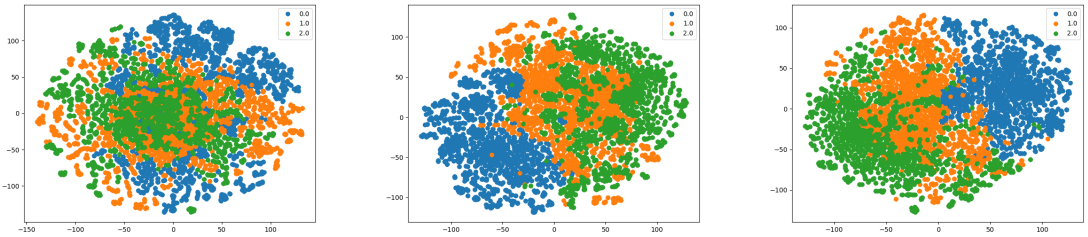

**Figure 7:** T-SNE embeddings ottenuti con **Triplet Margin Loss** dopo: 0, 15, 30 epoche di training

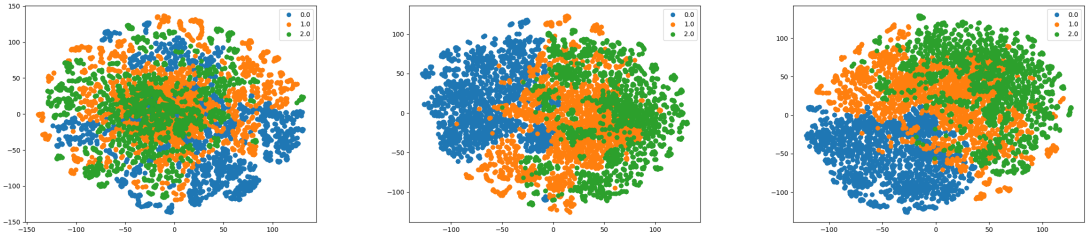

**Figure 8:** T-SNE embeddings ottenuti con **Triplet Margin with Distance Loss** dopo: 0, 15, 30 epoche di training

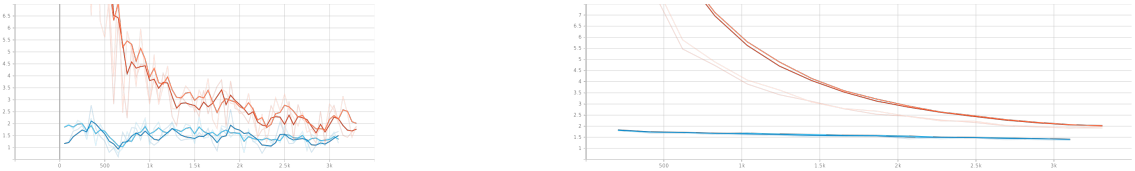

**Figure 9:** Grafici di convergenza di training (sx) e validation (dx) con Triplet Margin Loss (arancione / blu) e Triplet Margin with Distance Loss (rosso / azzurro)

# 6 Conclusione

Dopo 30 epoche si è deciso di procedere continuando il training con la **Triplet Margin Loss**, in quanto, quest'ultima sembrerebbe convergere più rapidamente rispetto all'altra variante. Si ricorda che ogni 15 epoche le triplette utilizzate per effettuare l'addestramento sono state cambiate.

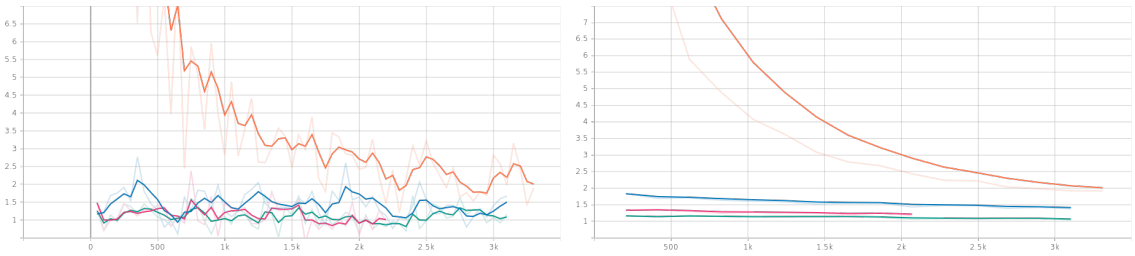

**Figure 10:** Grafico di convergenza di training (sx) e validation (dx)

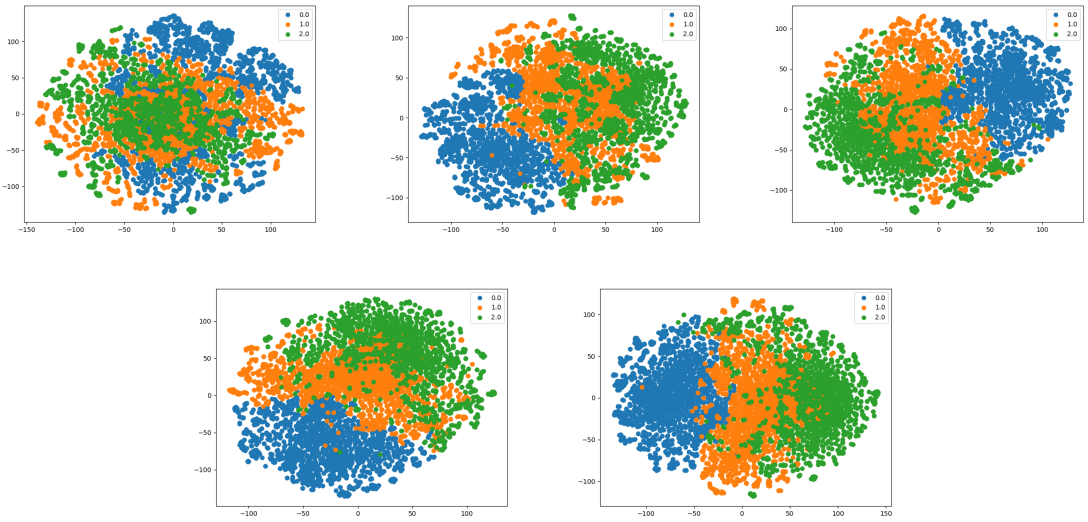

**Figure 11:** T-SNE embeddings dopo: 0, 15, 30, 40, 60 epoche di training

L'errore finale di classificazione ottenuto dopo `60 epoche` è `0.861`, mentre quello di validazione `1.083`.