UNIVERSITÀ
degli STUDI
di CATANIA

PREDIZIONE DI HOTEL-ID PER LA PREVENZIONE DEL TRAFFICO DI UMANI (FGVC9 2022)

Dipartimento di Matematica e Informatica
Corso: Deep Learning (LM-18) A.A. 22-23
Docenti: Giovanni Maria Farinella, Antonino Furnari

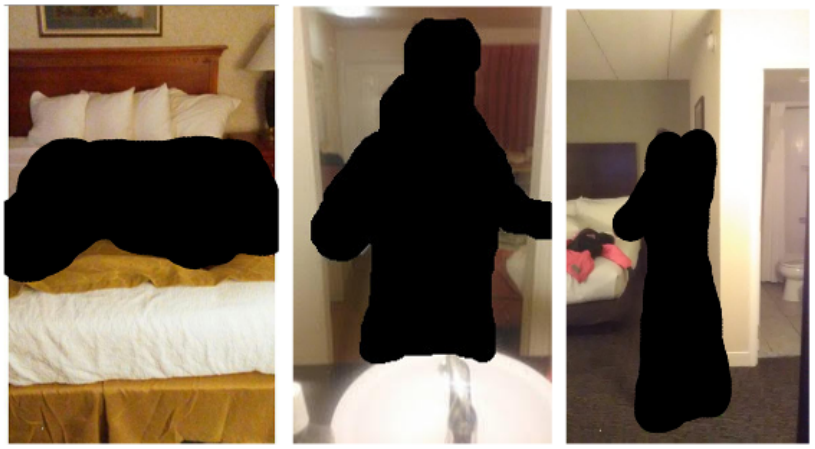
Michele Ferro
Matricola: 1000037665

Febbraio 2023

Sommario

Capitolo 1: Problema

Uno dei più sentiti problemi di questo secolo, è sicuramente il **traffico di umani**: la rapida diffusione di internet e lo sviluppo delle nuove tecnologie, oltre che la maggior semplicità nella manovra di acquisizione di dati di vario tipo (soprattutto fotografie), hanno infatti permesso il dilagare di questa piaga, specialmente nei confronti di soggetti più giovani. Non è raro che le vittime di questo fenomeno vengano fotografate in ambienti poco riconoscibili, in modo da non destare particolari sospetti: tra questi si ritrovano le camere di hotel e alberghi di vario tipo.
Tali fotografie, vengono successivamente utilizzate dai colpevoli come "locandine" per inserzioni da pubblicare sul Dark Web; ciò significa che identificare gli hotel nei quali i soggetti sono ritratti potrebbe essere di vitale aiuto nelle investigazioni su queste orribili pratiche illegali. Tuttavia, molte caratteristiche d'acquisizione, tra cui angolazione dell'obiettivo rispetto al soggetto fotografato, sotto/sovra-esposizione, rumore e cattiva qualità generale, sono fonte di non poche difficoltà nelle indagini.



Un modello, opportunamente addestrato mediante fotografie di hotel, potrebbe effettivamente aiutare nella prevenzione di questo crimine, ma l'alto numero di classi (considerando il numero di alberghi) e l'alta variabilità intraclasse unito alla bassa variabilità interclasse, fa permanere questo tipo di palliativo in una continua fase di sperimentazione.

Obiettivo di questo progetto d'esame è trovare una possibile soluzione al problema descritto nella challenge ***Hotel-ID to Combat Human Trafficking 2022 - FGVC9***[1] della piattaforma ***Kaggle***.

In particolare, l'idea generale, la quale verrà opportunamente approfondita nel capitolo dedicato, prevede il confronto tra diverse baseline possibilmente applicabili ad un problema caratterizzato da un ingente numero di classi apparentemente molto simili tra loro; come si vedrà in seguito, verranno valutati dei modelli di classificazione rispetto a delle architetture in grado di apprendere una metrica per questo specifico task, in maniera tale da poter poi utilizzare un algoritmo di tipo *K-NN* sul nuovo spazio di rappresentazione costruito mediante essa.
In entrambi casi, le feature delle immagini verranno estratte facendo utilizzo di una CNN nota e le cui performance si sono rivelate piuttosto soddisfacenti, se paragonata allo stato dell'arte.

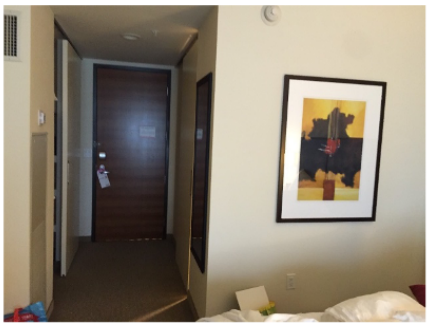
[1] "Hotel-ID to Combat Human Trafficking 2022 - FGVC9 - Kaggle." https://www.kaggle.com/competitions/hotel-id-to-combat-human-trafficking-2022-fgvc9. Ultimo accesso: 6 dic. 2022.

Per cui, si vuole cercare di costruire una ***deep network*** che, data una fotografia, restituisca la possibile (o le possibili) classi di appartenenza della foto, considerando che ad ogni classe corrisponda l'identificativo di uno degli hotel in questione.

Capitolo 2: Dataset

2.1 Raccolta dei dati

Il *dataset* utilizzato per lo svolgimento di questa *challenge* è quello fornito dalla ***Fine-Grained Visual Categorization*** (**FGVC**); esso è costituito da una serie di fotografie acquisite in diversi hotel, secondo diverse angolazioni e condizioni di luce, in modo da simulare quelle utilizzate nelle inserzioni di traffico umano.


Il metodo di acquisizione dei dati di cui è composto è particolarmente interessante: al fine di fornire supporto nella ricerca nei confronti di questo *task*, e realizzare quindi uno strumento utile agli investigatori, i creatori di questa *challenge* hanno sviluppato l'app ***TraffickCam***[2] (disponibile sugli *store* dei principali *mobile OS*), attraverso la quale qualunque viaggiatore può inserire delle fotografie di camere di alberghi, al fine di costruire un grande *database* nazionale. Naturalmente, chiunque può aiutare nella stesura di questa base di dati.
Al momento dello svolgimento di questo progetto, il *database* conta circa 1.5 milioni di fotografie per un totale di 145.000 camere di hotel[3]; per cui, il *dataset* fornito da *Kaggle* e utilizzato nello svolgimento del progetto sarà un *subset* di questo grande *database* costruito per mezzo dell'app *TraffickCam*.

2.2 Specifiche generali

Il *dataset* scaricato è piuttosto semplice, in quanto composto principalmente da **fotografie**, raccolte in cartelle la cui nomenclatura è quella dell'identificativo dello specifico hotel. Inoltre, un'ulteriore cartella presenta una serie di maschere che simulano la censura del soggetto presente in foto.

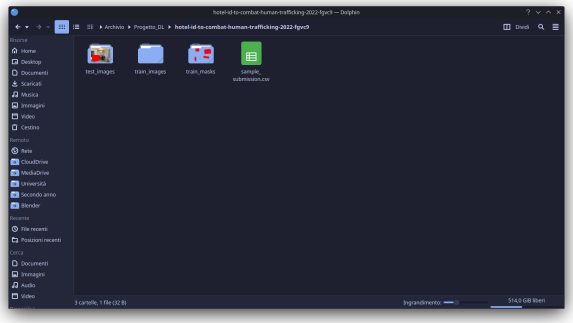
Entrando più nello specifico, oltre al file `sample_submission.csv` – il quale verrà ignorato in quanto esempio necessario al fine dell'effettiva partecipazione alla *challenge* secondo il processo descritto su *Kaggle* –, sono presenti tre cartelle:

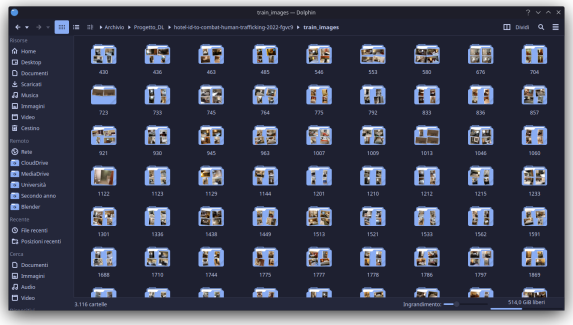
- **`test_images`**: la quale presenta un unico file `abc.jpg`, da utilizzare come preventivo *test* del modello prima di poter effettuare un invio di quest'ultimo, e ricevere l'effettivo *test*

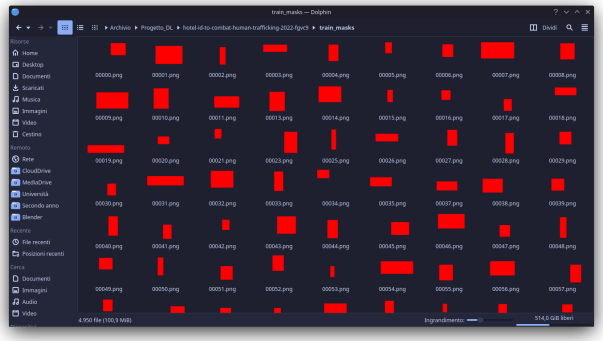
[2] "About - TraffickCam." http://traffickcam.com/about. Ultimo accesso: 6 dic. 2022.
[3] "You can help stop human trafficking with the TraffickCam app." 25 giu. 2016, https://techcrunch.com/2016/06/25/traffickcam/. Ultimo accesso: 6 dic. 2022.

set (si specifica che questa *challenge* è con *test set* nascosto, il quale viene inviato al partecipante solo una volta effettuata la *submit* del modello) da circa 5000 immagini;

- **`train_images`**: la quale contiene 3116 cartelle (per **altrettante classi**), ognuna etichettata secondo l'identificativo di ogni hotel; a sua volta, ogni cartella contiene un numero variabile di fotografie in formato `.jpeg`, tutte con risoluzione 1024×768 o 768×1024 (a seconda dell'orientamento verticale/orizzontale del cellulare al momento dell'acquisizione della fotografia). In totale, essa è composta da 44703 fotografie da utilizzare come campioni d'addestramento.
- **`train_masks`**: la quale contiene 4950 file in formato `.png` (con canale alfa), tutte con risoluzione variabile, raffigurante delle maschere di forma quadrata e di colore rosso puro $(255, 0, 0)$, simili a quelle utilizzate nel *test set* nascosto che viene fornito al partecipante al momento della *submission* alla *challenge*.







Si noti l'**assenza** di opportune cartelle contenenti campioni di validazione o di *test*; esse, in realtà, vengono automaticamente fornite da *Kaggle* al momento della *submission* del codice da parte di un partecipante; l'espediente utilizzato per ricavare un *set* di validazione e di *test* sarà opportunamente descritto nella prossima sezione da un punto di vista pratico, e nel **Capitolo 6** da un punto di vista delle procedure di automatizzazione sviluppate.

Tra tutte le classi presenti in `train_images`, la cui dimensione è di $14.2\ GiB$, la più numerosa è quella relativa all'identificativo di classe `76693`, che conta 1393 file, per una dimensione di $123.1\ MiB$. Le classi meno numerose (un solo elemento) hanno invece i seguenti identificativi.

1122	12414	13132	18718	35677	82686
93959	95159	309431	310041	396607	

2.3 Manipolazione successiva

Come già affermato in precedenza, il *dataset* fornito da *Kaggle* non è provvisto di opportuni *subset* da utilizzare per eseguire la validazione e/o il *test* dei modelli sviluppati (o meglio, il candidato è abilitato all'utilizzo del *set* di validazione solo dopo la *submission* del modello già addestrato).
Utilizzando una classica procedura di suddivisione del *set* originale – automatizzata mediante le funzioni descritte al **Capitolo 6** – si sono ricavate le seguenti cartelle:

- **`train_images`**: composta da 35762 fotografie (80% rispetto al suo originale contenuto), essa contiene tutti gli esempi da utilizzare nel corso dell'**addestramento**;
- **`val_images`**: composta da 8941 fotografie (20% del contenuto originale di `train_images`), essa contiene gli esempi da utilizzare nel corso della **validazione** dei modelli;
- **`test_images`**: composta da 8942 fotografie (20% del contenuto originale di `train_images` – preso secondo la stessa euristica utilizzata nella scelta dei campioni contenuti in `val_images` – + il file `abc.jpg` fornito da *Kaggle* come esempio di file di *test*) da utilizzare nel corso del ***test*** o per effettuare un'**inferenza**.

In particolare, i *set* `val_images` e `test_images` sono stati costruiti scegliendo in maniera casuale il 20% dei file contenuti in `train_images`.

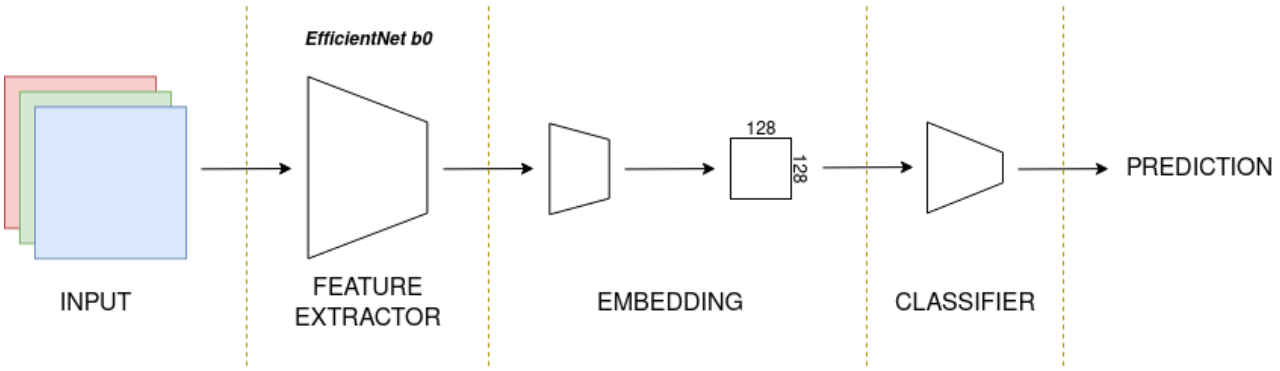
Capitolo 3: Metodi

Questo progetto si basa sul confronto tra architetture di classificazione multiclasse e modelli basati su *metric learning*. Ambedue le tipologie di reti fanno utilizzo di un estrattore di *feature* basato su ***EfficientNet***, CNN che verrà dettagliatamente descritta nei paragrafi successivi.

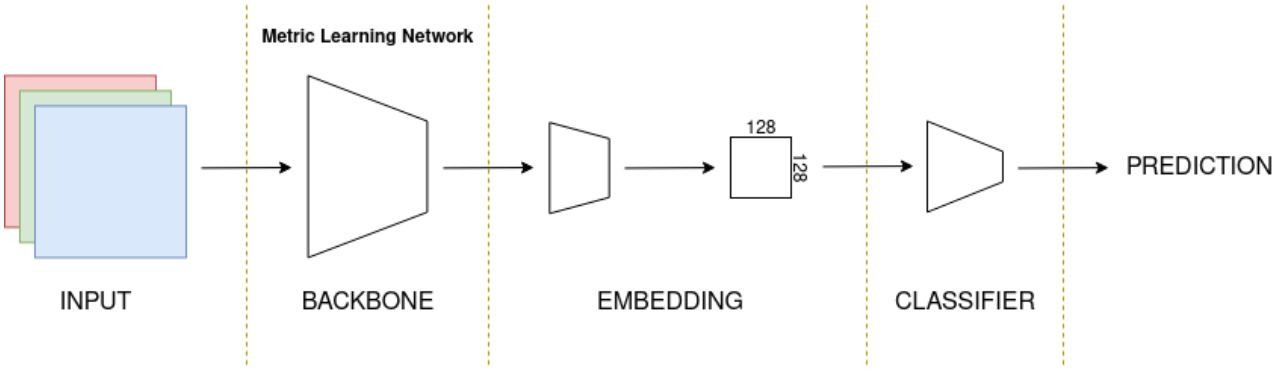
3.1 Reti di classificazione

Una delle *network* in questione è stata costruita mediante i seguenti *layer*:

- un estrattore di *feature* (***EfficientNet***);
- un *layer* di *embedding* per la riduzione della dimensionalità, le *feature* ottenute saranno di dimensione 128×128;
- un classificatore multiclasse standard.



In realtà, come si potrà poi constatare nel **Capitolo 6**, è stata costruita un'ulteriore architettura di classificazione, la quale come *backbone* presenta – al posto di *EfficientNet* – una rete di *metric learning* (opportunamente addestrata e che sarà successivamente descritta) comunque basata su di essa.



3.2 Rete di *metric learning*

Il ***metric learning*** è utilizzato al fronte di costruire delle reti neurali, in grado di creare (o meglio, apprendere) in maniera autonoma una metrica di distanza secondo un approccio ***supervised*** o ***weakly-supervised***, al fine di costruire un nuovo spazio $\mathbb{R}^d$ di rappresentazione dei dati originali.

L'obiettivo, è quindi quello di rappresentare i dati in un nuovo spazio, nel quale campioni (rappresentati da punti in esso) simili risultano vicini, e campioni dissimili risultano lontani; tale spazio potrà poi essere utilizzato per l'esecuzione di algoritmi di classificazione o *clustering* come K-NN o K-means.

Il primo step da effettuare nel momento in cui si costruisce un algoritmo di *metric learning*, è la scelta del **tipo di metrica** da utilizzare tenendo in considerazione il dominio; le diverse tipologie di metriche, le loro caratteristiche e quelle maggiormente utilizzate, verranno descritte nel **Capitolo 4**, relativo alla valutazione.

3.2.1 Categorie di *metric learning*

Come già accennato introducendo questa sezione, si hanno due diverse categorie di algoritmi di *metric learning*.

Algoritmi *supervised*

Queste tipologie di algoritmi hanno accesso a dati **etichettati** per classe; quindi, in questo caso, l'obiettivo è quello di trasformare lo spazio dei dati in input in maniera tale che dati con la stessa etichetta risultino vicini nel nuovo spazio ottenuto attraverso essa, mentre dati con etichette dissimili risultino lontani in esso.

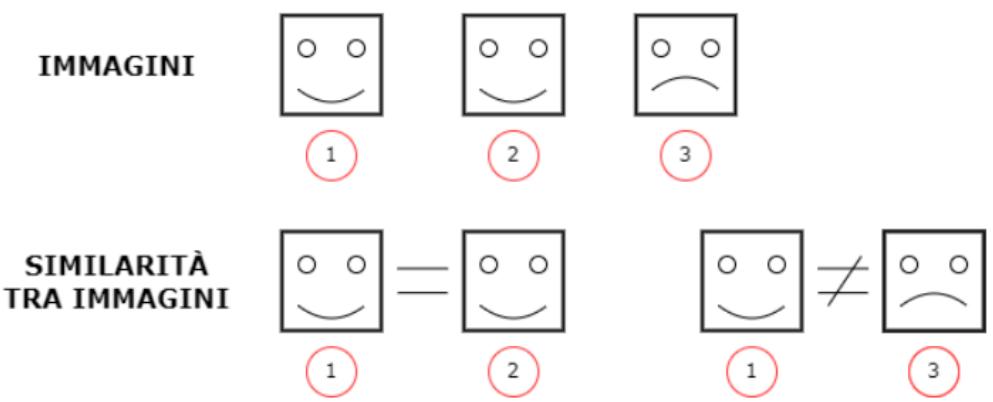
Formalmente, presi degli elementi $x^{(i)}$ e $x^{(j)}$ – con etichette rispettivamente $y^{(i)}$ e $y^{(j)}$ ad essi associate – da un *dataset* di dimensione N, si avrà una distanza D tra essi tale che:

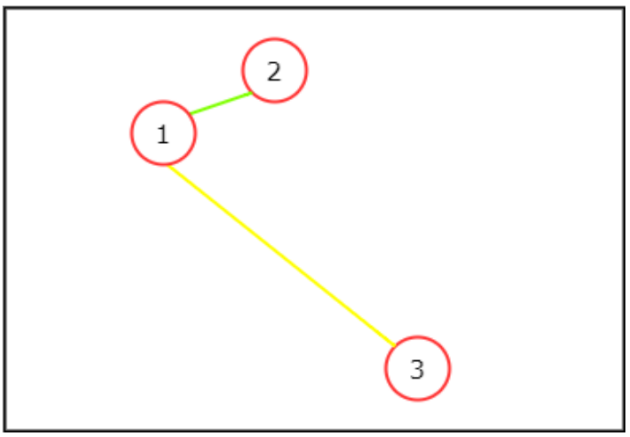
$$y^{(i)} = y^{(j)} \Leftrightarrow D\left(x^{(i)}, x^{(j)}\right) = 0$$
$$y^{(i)} \neq y^{(j)} \Leftrightarrow D\left(x^{(i)}, x^{(j)}\right) = 1$$
$$\forall i, j \in N, \quad i \neq j$$

Algoritmi *weakly-supervised*

Sono algoritmi che a differenza dei precedenti, hanno accesso a *dataset* la cui supervisione è esclusivamente a livello di **tuple** (coppie, triplette, ecc.), e l'unica metrica in grado di essere appresa è una che distingua tuple simili da tuple dissimili. In questo caso, l'obiettivo è quello di trasformare lo spazio dei dati in input in modo che tuple simili risultino vicine nel nuovo spazio, mentre tuple dissimili risultino distanti in esso.

Un classico esempio vede tre immagini, in cui le prime due sono sorridenti, mentre la terza non lo è.





In questo caso si avrà quindi una metrica di distanza D tale che $D((1),(2)) < D((1),(3))$.

3.2.2 Approccio *contrastive*

Trattandosi di una ***siamese network***, l'approccio utilizzato nella costruzione di questa rete è di tipo contrastive. Esso consiste nell'utilizzo di una *Loss function* – detta per l'appunto *Contrastive Loss* – che tende a spingere elementi diversi ad essere lontani nel nuovo spazio di rappresentazione e, viceversa, elementi simili ad essere vicini.

Quello applicato in questo progetto, e generalmente anche la stragrande maggioranza degli approcci *contrastive*, fa utilizzo della **metrica** l_2 – ossia la banale **distanza euclidea** – come misura di distanza.
Presi come riferimento due elementi $x^{(1)}$ e $x^{(2)}$ appartenenti ad uno stesso *dataset* etichettato, si ha quindi:

$$D\left(x^{(1)}, x^{(2)}\right) = \left\|x^{(1)} - x^{(2)}\right\|_2 = \sqrt{\sum_i \left(x^{(1)} - x^{(2)}\right)^2}$$

Data ora un'architettura L_θ nota in grado di fornire una nuova rappresentazione per ogni input x, la *contrastive loss* cercherà quindi di avvicinare i campioni simili (minimizzando la distanza **intra-classe**) e di allontanare quelli dissimili (massimizzando la distanza **inter-classe**).
Ricapitolando in maniera formale, dati:

- $x^{(1)}$ e $x^{(2)}$ due campioni appartenenti ad un *dataset*;
- $y^{(1)}$ e $y^{(2)}$ le etichette associate ai due campioni sopra;
- $D\left(x^{(1)}, x^{(2)}\right)$ una funzione di distanza (in questo caso euclidea) sui due campioni;
- $L_\theta(x)$ una funzione di rappresentazione per l'input x (generalmente è un'architettura di *feature extraction* nota, come VGG); se X è lo spazio degli input, $\overline{X} = L_\theta(x)$ è il nuovo spazio di rappresentazione, detto ***embedding space***;
- τ_A una funzione che restituisce 1 se il valore di veridicità per il predicato A è vero, 0 altrimenti;
- un margine $\alpha > 0$,

la *contrastive loss function* $Loss_C$ è definita nel seguente modo:

$$Loss_C\left(x^{(1)}, x^{(2)}\right) = \frac{1}{2}\left[\tau_{y^{(1)}=y^{(2)}} D\left(L_\theta\left(x^{(1)}\right), L_\theta\left(x^{(2)}\right)\right)^2 + \tau_{y^{(1)}\neq y^{(2)}} \max\left(0, \alpha - D\left(L_\theta\left(x^{(1)}\right), L_\theta\left(x^{(2)}\right)\right)^2\right)\right]$$

Si distinguono ora due casi.

CASO 1: Campioni simili

In questo primo caso, poiché i campioni sono simili, il predicato $y^{(1)} = y^{(2)}$ è **vero**, e di conseguenza il predicato $y^{(1)} \neq y^{(2)}$ è **falso**, per cui si avrà:

$$Loss_C\left(x^{(1)}, x^{(2)}\right) = \frac{1}{2} D\left(L_\theta\left(x^{(1)}\right), L_\theta\left(x^{(2)}\right)\right)^2$$

Si noti come la *loss*, in questo specifico caso, sta semplicemente minimizzando la distanza euclidea tra elementi simili.

CASO 2: Campioni dissimili

Poiché i campioni sono diversi tra loro, $\tau_{y^{(1)}=y^{(2)}} = 0$, e di conseguenza si avrà $\tau_{y^{(1)}\neq y^{(2)}} = 1$, portando quindi la *loss* ad essere ridotta a:

$$Loss_C\left(x^{(1)}, x^{(2)}\right) = \frac{1}{2} \max\left(0, \alpha - D\left(L_\theta\left(x^{(1)}\right), L_\theta\left(x^{(2)}\right)\right)^2\right)$$

massimizzando dunque la distanza euclidea tra dissimili.
Si noti inoltre che, in questo specifico caso, se la distanza è maggiore o uguale al margine α, allora

$$\alpha - D\left(L_\theta\left(x^{(1)}\right), L_\theta\left(x^{(2)}\right)\right)^2$$

risulta negativa, e di conseguenza il massimo restituirà 0 (per cui i pesi non verranno aggiornati, essendo nullo il gradiente di una *loss* nulla); ciò significa che i punti dissimili posti a distanza adeguata non verranno allontanati ulteriormente.
L'utilità nell'utilizzo di un margine α consiste nell'evitare che l'algoritmo mappi tutti i punti su un singolo punto, prendendo una sorta di "scorciatoia" di addestramento, e portando così ad un **overfitting**.

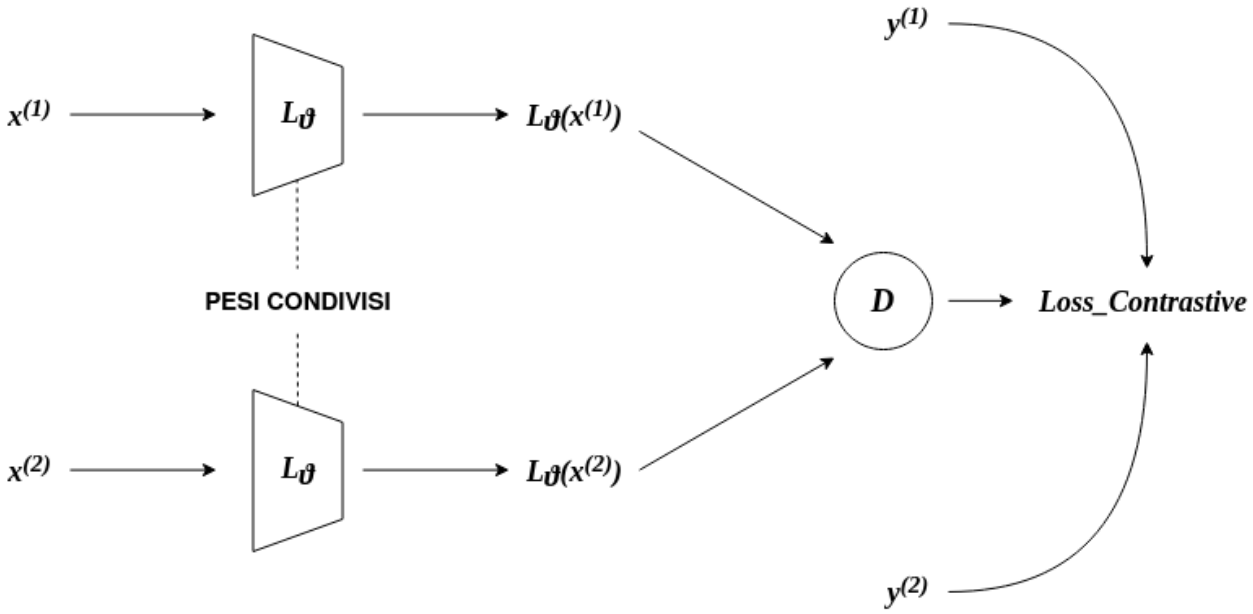
3.2.3 Architettura *siamese*

La baseline tipica di questo tipo di reti è la seguente:

1. si **campionano** le coppie $\left(x^{(1)}, x^{(2)}\right)$;
2. le *feature* della coppia vengono estratte da un'**architettura nota** L_θ (all'atto pratico si utilizzerà una CNN per elemento della coppia, ma queste condivideranno i pesi);
3. la nuove rappresentazioni $L_\theta\left(x^{(1)}\right)$ ed $L_\theta\left(x^{(2)}\right)$ degli elementi della coppia verranno passate alla **funzione di distanza** *D*;

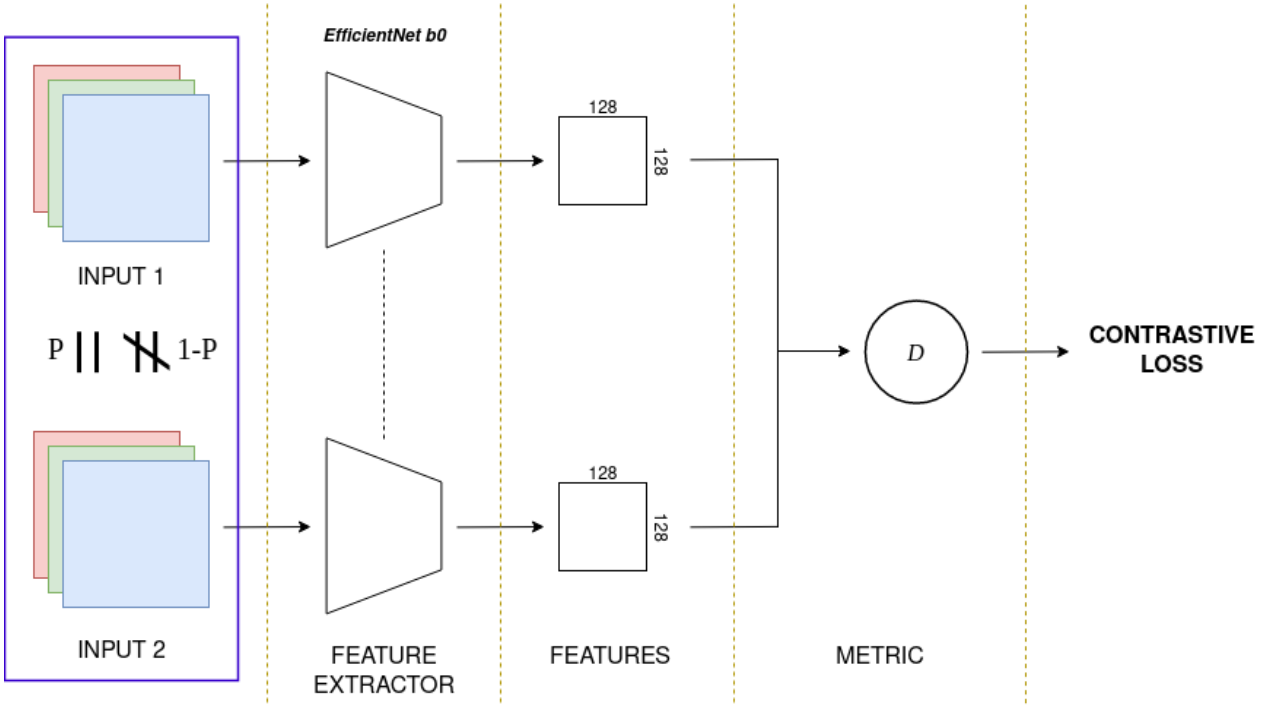
4. utilizzando il valore di distanza $D\left(L_\theta\left(x^{(1)}\right), L_\theta\left(x^{(2)}\right)\right)$ calcolato al passo precedente, si valuterà il **valore della Loss**;
5. si procede con l'**aggiornamento dei pesi** di L_θ mediante *backpropagation*.

Sintetizzando e raccogliendo quanto detto finora, una rete siamese di *metric learning* può presentare una struttura simile a quella rappresentata graficamente di seguito.



3.2.4 Architettura costruita

Di seguito, si mostra l'architettura costruita al fine dello sviluppo del progetto.



Com'è possibile notare, essa si basa semplicemente sull'estrattore di *feature EfficientNet* (descritto nel prossimo capitolo), il quale estrae delle *feature* dagli elementi della coppia in input campionata in modo da avere immagini simili con probabilità $P = \frac{1}{2}$.

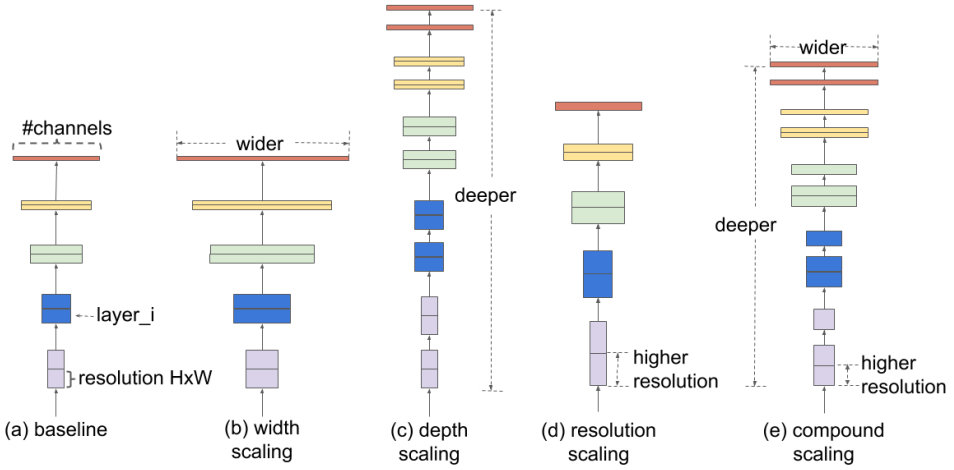
Successivamente, viene calcolata la distanza euclidea tra queste due rappresentazioni in input; la *Contrastive Loss* agirà poi in base a quanto descritto nei paragrafi precedenti.

3.3 EfficientNet

Questa famiglia di CNN, pubblicata nel 2019 sotto il nome di ***EfficientNet***[4], nasce dall'idea di proporre un nuovo metodo di scala di una *neural network* – tipicamente basato su un aumento indipendente di **profondità** o **larghezza** della stessa rispetto alla **risoluzione** dell'immagine in input – e proporne uno basato sulla scala composta, e quindi di ogni dimensione del modello, mediante un *set* fissato di coefficienti; tale intuizione ha dato alla luce questa serie di modelli in grado di superare l'accuratezza e l'efficienza dello stato dell'arte di ben **dieci volte** (grazie alla loro estrema semplicità e velocità).

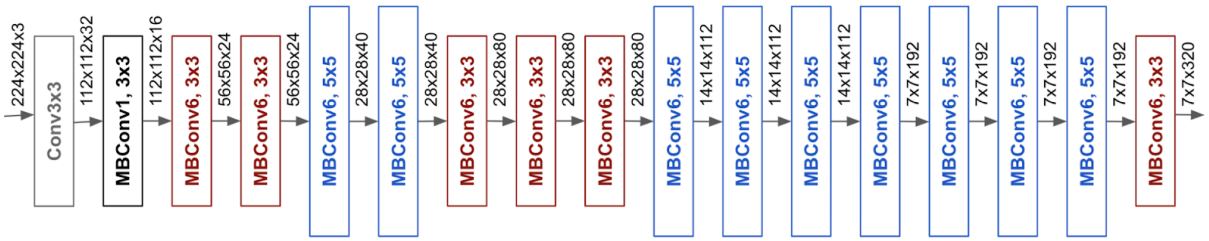
In particolare, gli autori si sono resi conto che, mentre scalare le tre dimensioni individualmente influiva sulle performance del modello, cercare piuttosto di bilanciare tutte le dimensioni della rete contemporaneamente – larghezza, profondità e risoluzione dell'immagine – a fronte delle risorse disponibili, portava piuttosto ad un aumento delle performance generali.

Il **primo passo** è stato quello di effettuare una ***grid search***. Essa si basa su una ricerca **esaustiva** su un *set* (manualmente specificato) di iperparametri; tale ricerca viene tipicamente guidata da una *cross-validation* sul *training set*.
Mediante questo primo step, si è stati in grado di trovare una relazione sullo *scaling* delle dimensioni della rete di *baseline*, sotto un vincolo fissato di risorse. Ciò determina un appropriato **coefficiente di scala** per ognuna delle dimensioni specificate sopra; successivamente, sono stati applicati questi coefficienti per scalare la rete mantenendo come riferimento le risorse computazionali a disposizione.

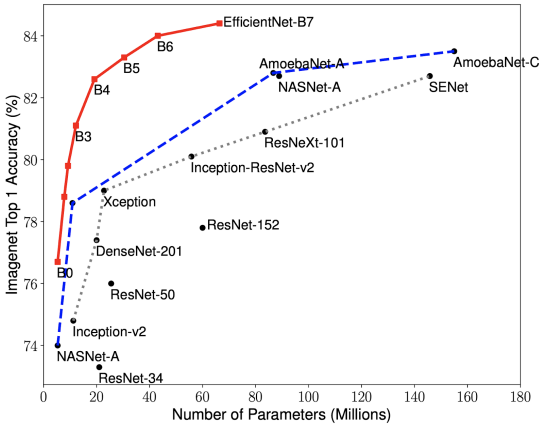


[4] "EfficientNet: Rethinking Model Scaling for Convolutional Neural ...." https://arxiv.org/abs/1905.11946. Ultimo accesso: 12 gen. 2023.

Questo metodo di scala composta si è rivelato particolarmente efficace, migliorando l'accuratezza e l'efficienza di modelli come **MobileNet** e **ResNet**, se paragonato ai metodi convenzionali.
Poiché l'efficacia di questo metodo si basa soprattutto sulla rete utilizzata come base, il **secondo step** è stato quello di sviluppare una nuova architettura neurale sfruttando ***AutoML MNAS***[5] (***Mobile Neural Architecture Search***, così chiamato in quanto il suo fine ultimo è quello di trovare una rete computazionalmente poco costosa per l'utilizzo su dispositivi mobili), un modello di apprendimento automatico che fa utilizzo di un algoritmo di ricerca per trovare la migliore architettura di una rete neurale per uno specifico *task*.
Richiedendo di ottimizzare *accuracy* ed efficienza (in termini di FLOPS), è risultata un'architettura basata su una *mobile bottleneck convolution* (**MBConv**) invertita, ma leggermente più grande a causa dell'incremento del *budget* computazionale; scalando ulteriormente questa nuova *baseline*, si è poi ottenuta la famiglia di modelli *EfficientNet*; in realtà, la *baseline* ottenuta mediante *AutoML MNAS* è stata la ***EfficientNet-B0*** (ossia quella utilizzata in questo progetto), mentre tutte le altre architetture – dalla *B1* alla *B7* – sono state poi ricavate effettuando ulteriori *scale-up* della *baseline*.



Confrontando le *EfficientNet* con altre CNN preesistenti per l'utilizzo su *ImageNet*, le prime sono state in grado di ottenere maggiori punteggi in accuratezza ed efficienza, riducendo la dimensione dei parametri e i FLOPS di un ordine di grandezza.
Com'è possibile constatare dal grafico seguitamente presentato, la *EfficientNet-B7* riesce a ottenere un'accuratezza di oltre 84%, pur essendo circa **8 volte meno profonda** e circa **6 volte più veloce** della miglior CNN esistente.
È possibile apprezzare il medesimo miglioramento confrontando la *EfficientNet-B4* (*accuracy* pari a oltre 82%) con *ResNet-50* (*accuracy* circa pari a 76%).



[5] "MnasNet: Towards Automating the Design of Mobile Machine ...." 7 ago. 2018, https://ai.googleblog.com/2018/08/mnasnet-towards-automating-design-of.html. Ultimo accesso: 13 gen. 2023.

Capitolo 4: Valutazione

Si descrivono di seguito le metriche di valutazione utilizzate nel corso degli esperimenti, e citate nel precedente capitolo, durante la trattazione dei vari modelli.

4.1 Matrice di confusione e metriche di classificazione

Valutare un classificatore esclusivamente in base al numero di record classificati correttamente può apparire impreciso, e può fornire un'idea incompleta e inesatta sul reale comportamento del modello, rispetto ai dati di *training* e di *test*.

Al fine di studiare in maniera completa la classificazione eseguita su un *set* di dati, si fa utilizzo della **matrice di confusione**: essa distingue **esempi positivi** (**P**) ed **esempi negativi** (**N**) in base a due sotto-categorie, ossia valori reali e predetti.
In particolare, tale tabella di dimensione 2×2, presenta sulle righe la quantità di esempi positivi e di esempi negativi in quanto tali, mentre sulle colonne è presente la medesima distinzione, ma effettuata dal classificatore mediante una predizione.
La tabella può essere presentata nel seguente modo.

	Elementi stimati **P**	Elementi stimati **N**
Elementi di natura **P**	TP	FN
Elementi di natura **N**	FP	TN

In base alla loro posizione riga-colonna, ritroviamo i seguenti valori:

- **TP** (**True Positive**): numero di elementi di natura positiva correttamente classificati come positivi;
- **FN** (**False Negative**): numero di elementi di natura positiva erroneamente classificati come negativi;
- **FP** (**False Positive**): numero di elementi di natura negativa erroneamente classificati come positivi;
- **TN** (**True Negative**): numero di elementi di natura negativa correttamente classificati come negativi.

Questi valori potrebbero già fornire una rozza misura di valutazione statistica del classificatore, ma è sempre preferibile utilizzarle al fine di ottenere ulteriori metriche ben più robuste, delle quali si fornisce qui una descrizione.

4.1.1 *Accuracy*

$$ACCURACY = \frac{TP+TN}{TP+TN+FP+FN}$$

L'***accuracy*** (o **accuratezza**) misura la percentuale di elementi correttamente classificati dal modello, rispetto al totale numero di elementi presenti nel *dataset* utilizzato.

È sempre importante osservare questa misura tenendo in considerazione il **bilanciamento** del *dataset* (facilmente visibile direttamente dalla tabella di confusione): infatti, potrebbe essere possibile avere un'alta *accuracy*, ma solo in quanto si stanno classificando correttamente solo gli elementi negativi.

4.1.2 *Precision*

$$PRECISION = \frac{TP}{TP+FP}$$

La ***precision*** (o **precisione**) misura la percentuale di elementi classificati positivi, rispetto al totale numero di elementi classificati come positivi.
In pratica, indica la percentuale di elementi stimati positivi che sono effettivamente positivi.

4.1.3 Recall

$$RECALL = \frac{TP}{TP+FN}$$

La ***recall*** (o **sensibilità**) misura la percentuale di elementi classificati positivi, rispetto al totale numero di elementi realmente positivi individuati nel *dataset*.
In pratica, indica la percentuale di elementi positivi scovati dal modello.

4.1.4 *F1-score*

$$F1 = 2 * \frac{PRECISION*RECALL}{PRECISION+RECALL}$$

F1-score è un indicatore che riassume e bilancia le due misure di *precision* e *recall* precedentemente definite.

4.2 Metriche di distanza

In particolare, si distinguono due diverse tipologie di metriche, descritte qui di seguito, presa come riferimento una coppia di elementi $x^{(1)}$ e $x^{(2)}$ appartenenti al medesimo *set* di dati.

4.2.1 Metriche di similarità

Tanto più **piccolo** è il valore della misura (e quindi, tanto più vicini sono i due punti nel nuovo spazio), tanto più **simili** saranno i due elementi presi in considerazione.
In particolare, due classici esempi di metriche di similarità sono la **distanza euclidea** E nel caso degli spazi euclidei e la **KL-*Divergence*** D_{KL} nel caso delle distribuzioni di probabilità.

DISTANZA EUCLIDEA	**KL-DIVERGENCE**
$E\left(x^{(1)}, x^{(2)}\right) = \sqrt{\left(\sum_i \left(x_i^{(1)} - x_i^{(2)}\right)\right)^2}$	$D_{KL}\left(x^{(1)}, x^{(2)}\right) = \sum_i x_i^{(1)} \log\left(\frac{x_i^{(1)}}{x_i^{(2)}}\right)$
$E \in [0, \infty]$, dove: $0 \rightarrow$ SIMILI, $\infty \rightarrow$ DISSIMILI	$D_{KL} \in [0, \infty]$, dove: $0 \rightarrow$ SIMILI, $\infty \rightarrow$ DISSIMILI

4.2.2 Metriche di dissimilarità

Tanto è più **grande** il valore della misura (ossia, tanto più lontani sono i due punti nel nuovo spazio), tanto **differenti** saranno i due elementi considerati.
Due esempi di misure di dissimilarità sono la **distanza del coseno** C nel caso degli spazi euclidei e la **distanza di *Bhattacharyya*** B nel caso delle distribuzioni di probabilità.

DISTANZA DEL COSENO	**DISTANZA DI *BHATTACHARYYA***
$C\left(x^{(1)}, x^{(2)}\right) = \frac{\sum_i \left(x_i^{(1)} x_i^{(2)}\right)}{\sqrt{\sum_i \left(x_i^{(1)}\right)^2}\sqrt{\sum_i \left(x_i^{(2)}\right)^2}}$	$B\left(x^{(1)}, x^{(2)}\right) = \sum_i \sqrt{x_i^{(1)} x_i^{(2)}}$
$C \in [0, 1]$, dove: $0 \rightarrow$ DISSIMILI, $1 \rightarrow$ SIMILI	$B \in [0, 1]$, dove: $0 \rightarrow$ DISSIMILI, $1 \rightarrow$ SIMILI

Nell'implementazione di questo progetto, verrà utilizzata la distanza euclidea.

Capitolo 5: Esperimenti

Si presentano di seguito i vari esperimenti eseguiti sul *dataset*.
Ogni esperimento è stato eseguito su una macchina con le seguenti caratteristiche:

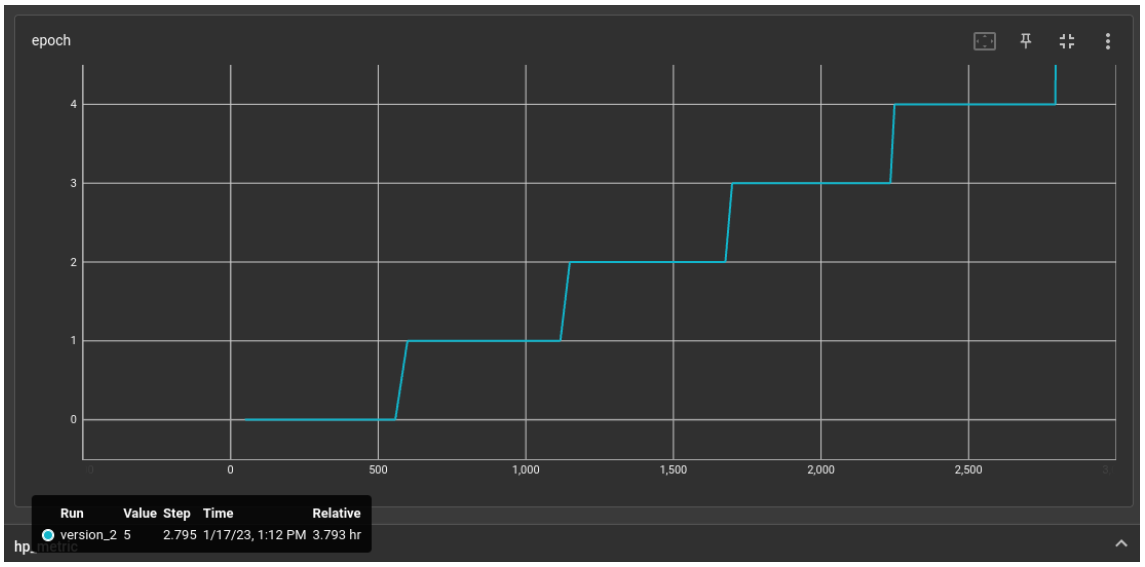
- **SO**: Manjaro Linux
- **Versione Kernel Linux**: 6.1.1-1-MANJARO
- **CPU**: Intel i7-7700HQ (8) @ 3.800GHz
- **RAM**: 16 GB
- **GPU**: *NVIDIA* GeForce GTX 1050Ti
- **VRAM**: 4 GB
- **Versione driver *NVIDIA***: 525.60.11
- **Versione *CUDA***: 12.0

La quantità di VRAM ed il numero di *thread* della CPU hanno permesso di utilizzare rispettivamente una dimensione dei ***batch*** `batch_size=64` ed un numero di ***workers*** `num_workers=8` per la costruzione dei ***DataLoader***.
Inoltre, a causa delle limitate risorse computazionali, tutti gli addestramenti sono stati eseguiti per **cinque epoche**, con i seguenti iperparametri:

- ***learning rate***: 0.01;
- ***momentum***: 0.99.

In tutti i casi, è stato applicato l'*optimizer* ***Stochastic Gradient Descent*** (***SDG***) per l'ottimizzazione del gradiente della *Loss* ottenuta.

Effettuando addestramenti di cinque epoche, ognuno dei modelli ha eseguito in totale 2974 step di aggiornamento del gradiente (circa 595 per epoca) su un *training set* composto da 35762 fotografie.



5.1 Tentativo di classificazione con EfficientNet-B0 (pre-addestrata)

Il primo esperimento è stato eseguito sulla rete *EfficientNet-B0* pre-addestrata – e quindi, con i suoi pesi "*vanilla*" ottenuti mediante l'addestramento eseguito da parte degli autori.
Come da aspettativa, i risultanti non sono per nulla eclatanti; in effetti, è molto probabile che nel corso del suo pre-addestramento (gli autori non specificano l'effettivo *dataset* utilizzato per il *pre-training*, ma è molto probabile che sia *ImageNet*, o comunque un *set* simile) non abbia visto esempi di camere di hotel – o nella migliore delle ipotesi, è probabile che ne abbia viste veramente poche. Il modello non è infatti in grado di distinguere alcuna classe, tra quelle presenti nel *dataset* utilizzato per l'esecuzione di questo progetto; sarebbe quindi del tutto inutile tentare un'inferenza alle condizioni attuali.
Per quanto insoddisfacenti, si riportano di seguito i risultati di validazione sul modello:

EfficientNet-B0 (*vanilla*)	
Validation *Loss*	0.0
Validation *Accuracy*	13.45

5.2 Tentativo di classificazione con `HotelPredictionNetworkv1`

Come è stato già spiegato nel **Capitolo 3**, e come verrà approfondito (almeno, da un punto di vista del codice) nel **Capitolo 6**, quest'architettura si basa sempre sulla rete *EfficientNet*, ma essa viene tuttavia posizionata nella *backbone* e utilizzata come estrattore di *feature* (viene quindi rimosso l'ultimo *layer fully-connected* della rete *vanilla*, ossia il *layer* di classificazione); successivamente, si passa ad una riduzione della dimensionalità, in modo da ottenere una rappresentazione di dimensione 128×128 (per il ridurre il peso sulla VRAM della macchina utilizzata), per passare infine ad un *layer* di classificazione.

```
  | Name       | Type             | Params
-----------------------------------------------
0 | extractor  | EfficientNet     | 4.0 M
1 | embedding  | Linear           | 163 K
2 | classifier | Linear           | 89.1 M
3 | criterion  | CrossEntropyLoss | 0
-----------------------------------------------
93.2 M    Trainable params
0         Non-trainable params
93.2 M    Total params
372.915   Total estimated model params size (MB)
```

Rispetto alla rete *vanilla*, questa *network* ha dato risultati decisamente più soddisfacenti, se non fosse per l'esiguo numero di epoche di addestramento.
Si riportano di seguito i risultati ottenuti.

HotelPredictionNetwork-v1		
Training *Loss*		5.339
Validation	***Loss***	8.21
	Accuracy	0.0132

Ciò indica che utilizzare la *EfficientNet* come ***feature-extractor***, ed eseguire un ***finetuning*** potrebbe essere una buona scelta, al fine di ottenere un modello di classificazione che riconosca una camera di hotel e ne restituisca l'identificativo.
Sicuramente, aumentando il numero di epoche di addestramento, l'architettura avrebbe dato dei risultati abbastanza soddisfacenti.

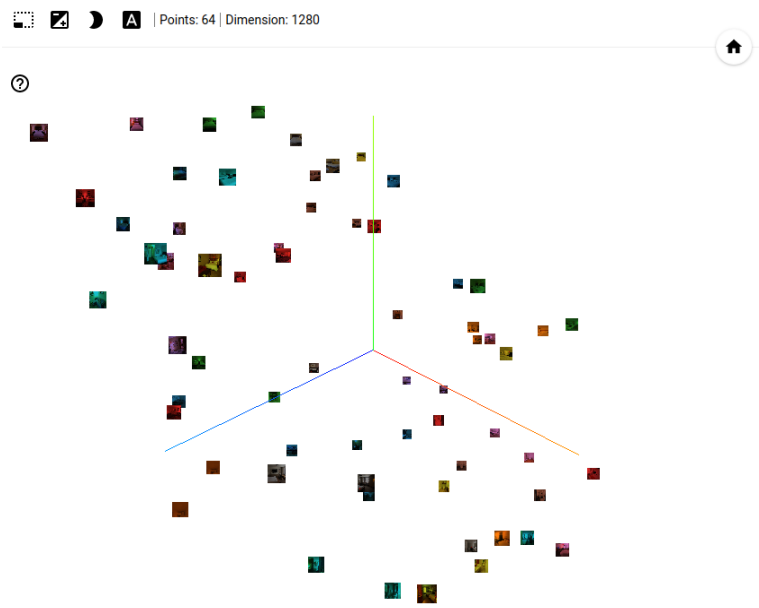
5.3 Tentativo di *metric learning* con `HotelPredictionNetworkv2`

Questa seconda rete è stata costruita a partire dall'idea di sperimentare un modello di *metric learning* per risolvere un problema di classificazione. Com'è stato specificato nel corso della presentazione del problema, esso è caratterizzato da un alto numero di classi e di elementi con alta variabilità intraclasse e bassa variabilità interclasse, per cui il *metric learning* – adottato proprio in situazioni del genere – potrebbe prestarsi bene alla risoluzione di questo *task*.
Naturalmente, la *baseline* consiste nel ricercare una metrica tale da ottenere un nuovo spazio in cui le fotografie di hotel simili risultino vicine, mentre quelle dissimili risultano lontane; successivamente, viene utilizzata l'architettura per estrarre queste nuove rappresentazioni dal *dataset* di *training*, per poi eseguire l'effettiva inferenza mediante l'applicazione di un **K-NN** (in questo caso, **K** = 5 perché si è stabilito di ricercare i cinque hotel più simili a quello passato in input nel corso dell'inferenza).
L'architettura si basa su una *backbone* di due *EfficientNet-B0* "gemellari" (e che quindi condividono i pesi), una riduzione della dimensionalità delle rappresentazioni ottenute ed un successivo calcolo della distanza euclidea tra esse.

```
  | Name      | Type            | Params
-------------------------------------------
0 | embedding | EfficientNet    | 4.0 M
1 | criterion | ContrastiveLoss | 0
-------------------------------------------
4.0 M     Trainable params
0         Non-trainable params
4.0 M     Total params
16.030    Total estimated model params size (MB)
```

Seguitamente si presentano i risultati ottenuti, compreso il nuovo spazio di rappresentazione dell'insieme di *training*.

HotelPredictionNetwork-v2	
Training *Loss*	0.481
Validation *Loss*	0.40



Salta subito all'occhio come le *performance* di questa architettura, almeno da un punto di vista della *Loss*, siano decisamente migliori rispetto a quelle delle reti precedenti; in una situazione in cui le risorse computazionali sono particolarmente limitate (come in questo caso), un approccio di *metric learning* sembrerebbe quindi indicato.

Naturalmente, successivamente alla fase di estrazione delle nuove *feature*, nel nuovo spazio di rappresentazione, del campione di *training*, segue l'effettiva esecuzione dell'algoritmo K-NN, il quale restituisce le cinque classi più somiglianti.

5.4 Tentativo di classificazione con `HotelPredictionNetworkv3`

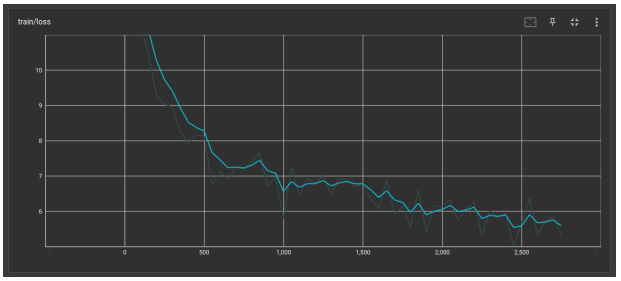
L'ultima rete è stata costruita secondo l'idea di voler sperimentare un approccio di *fine-tuning* sull'architettura precedentemente discussa. In particolare, viste le buone *performance* della *siamese network*, si è stabilito di utilizzarla come *backbone* di un'ulteriore rete di classificazione, costruita in maniera abbastanza simile alla prima.

```
  | Name       | Type                     | Params
-------------------------------------------------------
0 | backbone   | HotelPredictionNetworkv2 | 4.0 M
1 | embedding  | Linear                   | 163 K
2 | classifier | Linear                   | 89.1 M
3 | criterion  | CrossEntropyLoss         | 0
-------------------------------------------------------
93.2 M    Trainable params
0         Non-trainable params
93.2 M    Total params
372.915   Total estimated model params size (MB)
```

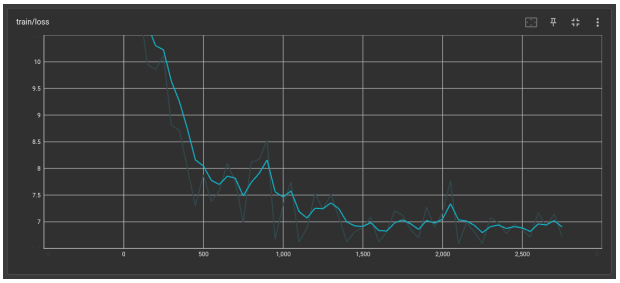
Di seguito i risultati ottenuti.

HotelPredictionNerwork-v3		
Training *Loss*		6.98
Validation	***Loss***	8.84
	Accuracy	0.002

Se paragonata alla v1 precedentemente analizzata, questo classificatore ha più o meno le medesime prestazioni; tuttavia, osservando i seguenti grafici, è possibile notare come la *Loss* tenda a scendere più rapidamente che nel primo caso. Ciò potrebbe indicare che, effettuando un addestramento più duraturo sia sul modello v2 (per poi congelarlo e utilizzarlo come *backbone*), sia sull'architettura in fine-tuning, si potrebbero avere rapidi e ottimi risultati di convergenza.



HotelPredictionNerwork-v1



HotelPredictionNerwork-v3

Capitolo 6: Codice

L'organizzazione del codice di addestramento, fornito nel Jupyter Notebook chiamato **`HOTEL-ID_prediction_model.ipynb`**, prevede una sezione dedicata alla gestione del *dataset*, e l'effettivo codice di addestramento di definizione e addestramento dei modelli costruiti sfruttando il modulo ***PyTorch Lightning***.
La parte relativa all'inferenza sui modelli addestrati è stata organizzata in un ulteriore *notebook* di *demo*, presentato nel capitolo successivo.

6.1 Gestione del *dataset*

La **Sezione 0** e la **Sezione 1** del *notebook* contengono tutte le celle di codice scritte al fine di poter utilizzare il *dataset* scaricato nell'apposita sezione della *Kaggle competition*.
Il loro contenuto verrà trattato indipendentemente nei paragrafi seguenti.

6.1.1 Split del *training set* fornito

Come già accennato al **Capitolo 2**, nel corso della descrizione del *dataset*, esso manca di opportune *directory* contenenti campioni di validazione e di *test*. Al fine di poter ottenere questi due nuovi *set*, nella **Sezione 0** del *notebook* è stata scritta la procedura **`split_dataset(dataset_path)`**, la quale, preso in input il *path* `dataset_path` del *dataset* scaricato su disco, genera due ulteriori *directory* `test_images` e `val_images` a partire dalla *directory* `train_images`, e modifica il contenuto di quest'ultima; al termine del processo, la funzione restituirà i relativi *path* delle *directory* ottenute. In particolare, il processo di *splitting* è stato definito al fine di ottenere la seguente numerosità per ogni cartella:

- `train_images` conterrà l'80% del contenuto che aveva in origine;
- `val_images` conterrà il 20% del contenuto originario di `train_images`;
- `test_images` conterrà il 20% del contenuto originario di `train_images`.

Successivamente, è definita la procedura **`generate_csv(train_path, test_path, val_path)`**, che presi in input i relativi *path* delle *directory* – rispettivamente contenenti i campioni di *training*, di *test* e di validazione – genera un file in formato `.csv` per ognuna delle tre, secondo il seguente *pattern* (naturalmente, nel caso del *set* di *test* non è presente l'etichettatura dei file):

`image_id`	**`hotel_id`**
`abc.jpg`	`123`

6.1.2 Manipolazione dei campioni

La **Sezione 1** del *notebook* contiene il codice relativo alla manipolazione del *dataset* effettivo, e di conseguenza, l'effettiva costruzione degli oggetti di classe *dataset*.

Definizione delle trasformazioni

Dato un *downscaling* delle immagini ad una dimensione 128×128 (al fine di ottenere campioni in input dimensionalmente identici, ricordando che le fotografie fornite sono caratterizzate da risoluzioni differenti), sono definite tre trasformazioni:

- **`occlusion_transform`**: simula le **occlusioni** contenute in `train_masks`, *directory* presente nel *dataset*, pertanto verrà applicata in ambedue le successive trasformazioni;
- **`train_transform`**: trasformazione applicata esclusivamente al ***set* di *training***; si basa su una serie di trasformazioni semplici (*flip* orizzontale e verticale, rotazioni, distorsioni ottiche, modifica della luminosità e del contrasto, applicazione delle occlusioni definite prima, ridimensionamento) mirate a rendere i modelli più robusti;
- **`val_transform`**: trasformazione applicata esclusivamente al ***set* di validazione** (e di **test**); si basa su una semplice applicazione delle occlusioni ed un successivo ridimensionamento.

Definizione delle classi *Dataset*

Naturalmente, sono stati necessari degli opportuni oggetti utili a **campionare** i file in questione, a seconda del tipo di *set* e, soprattutto, a seconda del tipo di modello sul quale effettuare l'addestramento, la validazione o il test.

Poiché – come già anticipato – sono state costruite due differenti baseline (una basata su classificazione e una basata su *metric learning*), si sono mostrati necessari due differenti tipologie di oggetti *Dataset*. In particolare, sono state definite le due seguenti classi:

- **`HotelDataset`**: prendendo come riferimento un file `csv`, per ogni suo *record* restituisce un'immagine e la relativa etichetta di classe nel caso in cui l'istanza del *dataset* sia di *training* o di validazione, o semplicemente l'immagine nel caso in cui essa sia di *test*.
- **`PairHotelDataset`**: poiché il modello di *metric learning* sarà basato su un'architettura siamese, tale classe di *Dataset* è stata definita in maniera da campionare **coppie** di elementi. In particolare, preso un file `csv`, per ogni suo record verrà restituita una coppia di elementi etichettata in base alla **similitudine** tra essi: tale etichetta assume valore `l=0` nel caso in cui gli elementi della coppia sono simili tra loro – e quindi appartenenti alla medesima classe –, in caso contrario essa assume valore `l=1`; le coppie sono campionate in modo che ad ogni elemento sia associato un suo elemento simile con probabilità $P = \frac{1}{2}$.

6.2 Definizione dei modelli

Poiché questo progetto si basa su un confronto tra *baseline* rispettivamente basate su classificazione e su apprendimento di metriche, sono stati costruiti quattro differenti modelli, presentati nella **Sezione 2** del *notebook*. Tutte le architetture sono state scritte mediante il modulo ***Pytorch Lightning***[6].

Essendo le reti composte da *EfficientNet-B0*, si è utilizzato il modulo Python **`timm`**[7] per l'opportuno download della CNN.

[6] "PyTorch Lightning." https://www.pytorchlightning.ai/. Ultimo accesso: 15 gen. 2023.
[7] "rwightman/pytorch-image-models - GitHub." https://github.com/rwightman/pytorch-image-models. Ultimo accesso: 15 gen. 2023.

6.2.1 Classificatore `EmbeddingNet`

Questa classe funge da semplice *encapsulation* di una *EmbeddingNet-B0* pre-addestrata, importata mediante il modulo `timm`.

6.2.2 Classificatore `HotelPredictionNetworkv1`

La classe che definisce l'architettura di classificazione sviluppata è stata chiamata **`HotelPredictionNetworkv1`**, la quale viene istanziata a partire dal numero di classi **`n_classes`** secondo il quale costruire il *layer* di classificazione, una dimensione dell'*embedding* **`embedding_size`**, e il nome della CNN da voler utilizzare come estrattore – in questo caso **`efficientnet_b0`** posto come valore di default (si rimanda al **Capitolo 3** per la descrizione dell'architettura).

6.2.3 Architettura di *metric learning* `HotelPredictionNetworkv2`

Per la costruzione del modello di *metric learning* sono state definite due classi:

- **`ContrastiveLoss`**: è l'implementazione (con margine **`m`**=`2` di default) della *Loss Contrastive* vista precedentemente al **Capitolo 3**; verrà utilizzata nella classe di seguito descritta.
- **`HotelPredictionNetworkv2`**: definisce l'implementazione della rete siamese per l'apprendimento di metriche, anch'essa descritta al **Capitolo 3**; in particolare fa utilizzo di un oggetto di tipo `ContrastiveLoss` (definito in precedenza) come `criterion`. Un modello di questa classe viene istanziato a partire dalla scelta dell'estrattore, del *learning rate* `lr`, del `momentum` e del `margin` (i quali prevedono tutti dei valori di default).

6.2.4 Classificatore `HotelPredictionNetworkv3`

Per l'esecuzione di una quarta *baseline*, basata sulla classificazione, è stata definita la classe **`HotelPredictionNetworkv3`**, la quale descrive un'architettura basata su tre componenti:

- l'architettura di *metric learning* **`HotelPredictionNetworkv2`** utilizzata come *backbone*;
- un *embedding* a dimensione 128×128 per ridurre la dimensionalità delle rappresentazioni ottenute mediante la *backbone*;
- un classificatore.

6.3 Addestramento

La **Sezione 3** del *notebook* include tutto il codice scritto e utilizzato per l'addestramento dei modelli. In entrambi i casi è prevista una cella che istanzia i modelli da zero, e una che invece istanzia i modelli caricando un file di *checkpoint* (per eventuali addestramenti interrotti e ripresi seguitamente).

6.3.1 Classificatori

Costruiti i *DataLoader* di *training* e validazione (rispettivamente **`hotel_train_loader`** e **`hotel_val_loader`**) attraverso degli opportuni *dataframe* a partire dai `csv` generati mediante le procedure precedentemente descritte, vengono istanziati i tre seguenti oggetti:

- **`model`**=`HotelPredictionNetworkv<x>(•)`: istanza del modello di classificazione;
- **`logger`**=`TensorBoardLogger(•)`: istanza del *logger* utilizzato;
- **`trainer`**=`pl.Trainer(•)`: istanza del *trainer* utilizzato per configurare l'addestramento.

Il tutto è seguito da una chiamata alla funzione **`fit`** del `trainer` per eseguire l'effettivo addestramento del classificatore.

6.3.2 Architettura di *metric learning*

Anche in questo caso, costruiti (nella stessa maniera precedentemente vista) i *DataLoader* di *training* e validazione (rispettivamente **`pair_hotel_train_loader`** e **`pair_hotel_val_loader`**), vengono istanziati i tre seguenti oggetti:

- **`siamese_hotel_task`**=`HotelPredictionNetworkv2(•)`: istanza dell'architettura di *metric learning*;
- **`logger`**=`TensorBoardLogger(•)`: istanza del *logger* utilizzato;
- **`trainer`**=`pl.Trainer(•)`: istanza del *trainer* utilizzato per configurare l'addestramento.

La cella successiva è quella contenente la chiamata alla funzione **`fit`** del `trainer` per eseguire l'addestramento.

Differentemente da quanto visto nel caso dell'architettura di classificazione, segue qui una sottosezione dedicata al salvataggio su disco delle nuove rappresentazioni degli elementi costituenti il *training set*.
L'idea di base è quella di calcolare queste nuove **rappresentazioni** mediante inferenza sul *dataset* di *training* (tale step rientrerebbe comunque nella fase di addestramento), salvarli su disco mediante file `pickle`, e successivamente sfruttare queste informazioni nel corso dell'inferenza (il resto verrà meglio approfondito nel capitolo seguente, dedicato alla demo).

Al fine di poter **estrarre** tali rappresentazioni, è stata definita la procedura **`extract_representations(model, loader)`**, la quale, presa in input il modello

`model=`**`siamese_hotel_task`**

ed un *DataLoader*

`loader=`**`hotel_train_loader`**

(con output a singolo elemento, quindi), effettua un'inferenza per ogni elemento del *dataset* di *training* e restituisce la sua nuova rappresentazione secondo la **metrica appresa** dall'architettura siamese.
Tali rappresentazioni vengono poi inserite in un *dataframe* sotto l'attributo **`representation`**; il *dataframe* ottenuto verrà quindi salvato su disco per poter poi essere utilizzato nel corso della demo.

Capitolo 7: Demo

L'organizzazione del codice di demo, e relativo all'inferenza sulle architetture precedentemente citate, è presente nel Jupyter Notebook **`Hotel-ID_prediction_inference.ipynb`**.
Vengono ignorate in questa descrizione tutte le parti del *notebook* fino alla **Sezione 3**, in quanto includono semplicemente le definizioni e le istanziazioni delle classi – già descritte nel capitolo precedente – per poter poi effettuare l'inferenza.

In questo caso tuttavia, trattandosi di modelli il cui addestramento è stato effettuato mediante il notebook precedentemente discusso, tutte le istanze caricate dai relativi file di ***checkpoint*** salvati su disco.

7.1 Inferenze

Il codice relativo alle procedure di inferenza dei modelli è interamente contenuto nella **Sezione 4**, ed è suddiviso in due ulteriori sotto-sezioni a seconda della tipologia di modello sul quale inferire.

7.1.1 Inferenza su classificatore

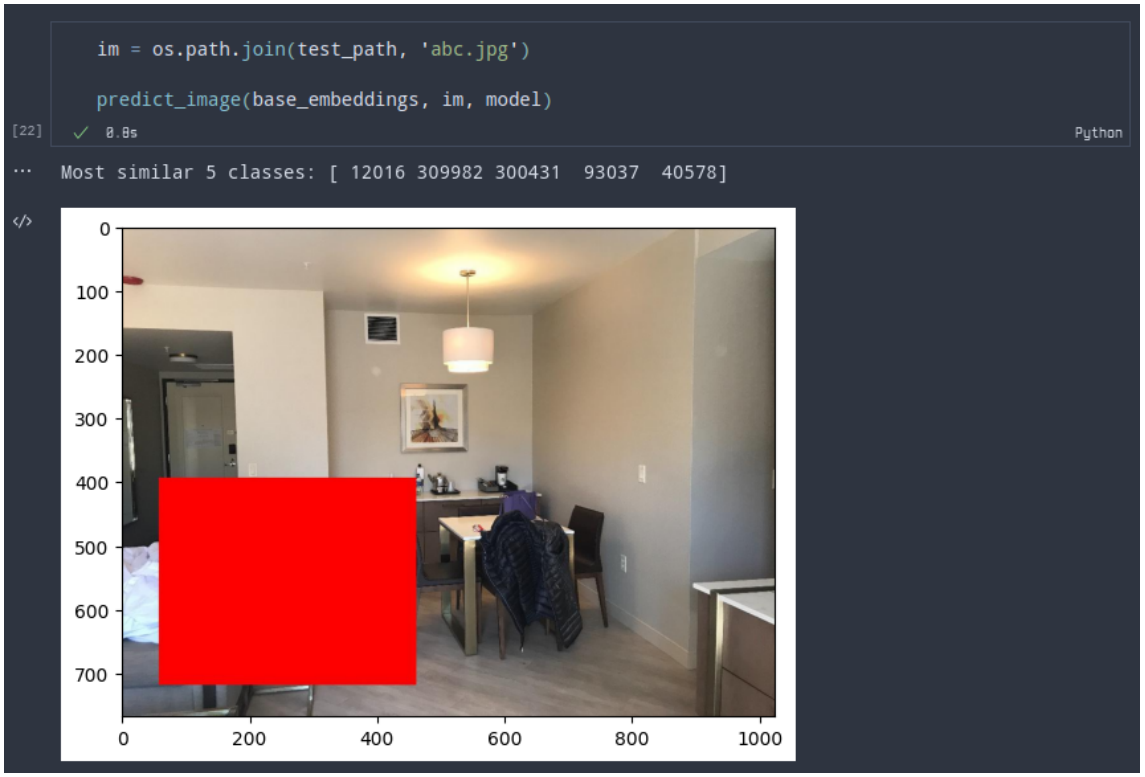
In questo caso è stata definita una procedura **`predict_image(model, image)`** che presa in input una fotografia `image` da disco, esegue una predizione della sua classe d'appartenenza (e di conseguenza, l'identificativo dell'hotel in questione) utilizzando il classificatore `model` scelto.

7.1.2 Inferenza su architettura di *metric learning*

Per l'inferenza su architettura di *metric learning* è stata implementato l'algoritmo **K-NN**, strutturato in due diverse procedure richiamate in **`predict_image(base_embeddings_df, image, model)`**, ossia:

- **`extract_image_representation(model, im)`**: presa in input la fotografia `image` su disco, calcola la sua nuova rappresentazione secondo lo spazio costruito dal modello `model` secondo la metrica di similarità appresa;
- **`find_matches(query, base_embeds, base_targets, n_matches)`**: presa in input una query e delle rappresentazioni `base_embeds` precedentemente calcolate nel notebook di *training* (**Capitolo 6**) e salvate su disco, calcola la distanza tra la `query` ed ognuna delle rappresentazioni in questione; successivamente ordina i risultati ottenuti per distanza, e restituisce le `n_matches` classi (e di conseguenza, gli ID di Hotel) più "vicini", e quindi simili alla `query`.

Per cui, la funzione `predict_image(•)`, anzitutto estrae la rappresentazione della fotografia `image` secondo il modello `model`, successivamente viene applicato K-NN richiamando la funzione `find_matches(•)` e vengono restituite le classi predette.



Conclusioni

I risultati scaturiti dalle sperimentazioni eseguite nel corso di questo progetto evidenziano come il problema di riconoscimento di camere di alberghi possa essere risolto medianti differenti approcci al *Deep Learning*; infatti, sia facendo utilizzo di una tipica rete di classificazione, che costruendo un apposita architettura di *metric learning* (facendo utilizzo, in ambedue le tipologie di baseline, di un estrattore di *feature* **noto** e caratterizzato da ottime prestazioni rispetto allo stato dell'arte) sono stati ottenuti risultati accettabili. O per essere più precisi, accettabili se si considerano le limitate risorse computazionali a disposizione; tutti i modelli, infatti, se addestrati per un maggior numero di epoche, sarebbero sicuramente stati caratterizzati da migliori misure di *performance* – soprattutto quelli di classificazione.

Da questo punto vista, la tipologia di architettura che ha maggiormente stupito si è rivelata essere quella basata sul *metric learning*, una *siamese network* in grado di raggiungere delle misure di *loss* (sia in fase di *training* che di validazione) circa **20 volte** più basse rispetto ai modelli di classificazione costruiti, a parità di epoche d'addestramento.
Ciò sta ad indicare che le *baseline* basate sull'apprendimento di *task-specific distance* si possono prestare particolarmente bene nella risoluzione di problemi di classificazione e riconoscimento di immagini, esattamente come in questo caso.

Particolare menzione merita anche l'ultimo modello costruito, ossia l'architettura che presenta il modello di *metric learning* (pre-addestrato) come *backbone* e sulla quale è stato poi eseguito un *finetuning*. Infatti l'utilizzo di quella specifica *backbone*, con pesi già noti e caratterizzata per l'appunto da una bassa *loss*, ha poi portato ad una ben più rapida discesa della stessa funzione della nuova architettura complessiva.
Anche in questo caso, un addestramento più duraturo (da un punto di vista delle epoche) avrebbe sicuramente giovato al modello.