Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Generazione di attributi facciali mediante Feature-wise Linear Modulation

Relatore:
Chiar.mo Prof.
Andrea Asperti

Presentata da:
Danilo Branca

III Sessione
Anno Accademico 2018 - 2019

Indice

Introduzione

Negli ultimi anni sta prendendo sempre più piede, per le applicazioni più disparate, l'impiego di automatismi dell'apprendimento. Uno dei settori più importanti, in cui l'apprendimento automatico viene, ormai, usato in larga scala, è quello della classificazione e analisi dei dati.
L'apprendimento automatico avviene attraverso l'utilizzo di reti neurali che, attraverso un processo denominato training, mimano la capacità del cervello umano di "capire" la natura di un' informazione mai vista prima, prendendo, come base, le nozioni apprese durante il processo.

Generazione di nuovi dati

L'aspetto dell'apprendimento automatico su cui si sta lavorando di più, negli ultimi anni, è quello della generazione di dati, come ad esempio suoni, testi e immagini.
Per poter generare nuovi dati è necessario allenare la rete neurale a riconoscere e a "capire" le peculiarità, chiamate features, di un numero molto alto di samples, appartenenti al dominio di dati che si vogliono generare. Ogni samples ha un' "etichetta" che ne descrive, in qualche modo, le features. La rete può così utilizzare le features apprese per generare nuovi dati, simili a quelli usati durante il processo di apprendimento. Più nel dettaglio, i dati reali usati per l'apprendimento hanno una distribuzione di probabilità $P(x)$, il problema della generazione consiste nell'allenare una rete neurale a generare nuovi dati, aventi distribuzione di probabilità $P(\tilde{x})$ simile a $P(x)$.
Uno dei modelli maggiormente usati per la generazione di dati è il VAE (Variational Autoencoder), un modello generativo probabilistico che può essere visto come una coppia di due modelli indipendenti parametrizzati: l'encoder e il decoder[17].
Un modello generativo probabilistico funziona apprendendo la distribuzione di probabilità $P(x)$, definita su un insieme X di dati, contenuti in uno spazio multi dimensionale $\mathcal{X}$[12].

Condizionamento della generazione

Un aspetto interessante nel campo della generazione è la possibilità di condizionare il modo in cui la rete neurale genera nuovi dati. In statistica, condizionare significa fissare un valore per una variabile o set di variabili e osservare come cambia la distribuzione di probabilità.
Nel campo delle reti neurali ci sono diversi metodi per introdurre il condizionamento in un modello. Dato un input x e la condizione c, i metodi più importanti sono:

- *Concatenation-based conditioning*: la condizione c viene concatenata all'input x in ogni livello della rete, ottenendo cx;
- *Conditional biasing*: la condizione c viene prima processata, ottenendo una sua rappresentazione nello spazio latente, per poi venire concatenata all'input x. Questo metodo può essere visto come un caso particolare del Concatenation-based conditioning;
- *Conditional scaling*: la rappresentazione nello spazio latente della condizione c viene trasformata in un vettore scalare, l'input viene poi moltiplicato per il vettore ottenendo un input "scalato" $c \cdot x$.

Feature-wise Linear Modulation

Feature-wise Linear Modulation, abbreviato "FiLM", è una tecnica per influenzare in modo adattivo l'output di una rete neurale basandosi su un input arbitrario, applicando una trasformazione affine (chiamata semplicemente FiLM), sulle features intermedie della rete[21]. Inizialmente è stato proposto come un miglioramento di reti neurali che risolvono problemi di visual reasoning (ragionamento visivo). In questo elaborato verrà analizzata la sua efficacia anche nel campo della generazione di immagini di volti umani, condizionata da un insieme di attributi, scelti dall'utente, che il volto generato deve avere.

Obiettivi e strumenti per l'implementazione

Lo scopo dell'elaborato è mostrare l'integrazione di livelli FiLM all'interno di un modello VAE. Il modello così ottenuto verrà analizzato per le sue capacità di ricostruzione dell'input e di generazione di nuovi volti umani, sulla base di specifici attributi. Il modello verrà allenato sui volti presenti nel dataset CelebA[19]. Inoltre, per avere una metrica sulla qualità dei volti in output, verrà calcolata la Fréchet Inception Distance (FID) sia per i volti ricostruiti, partendo dall'approssimazione dell'encoder, sia per i volti sintetizzati dal decoder, partendo da campionamenti dello spazio latente.

Il modello è stato implementato usando il linguaggio Python 3.7, con l'uso delle API ad alto livello per reti neurali Keras[9] 2.3.1, configurato con Tensorflow[1] 2.0. Tensorflow utilizza di default l'architettura hardware per l'elaborazione parallela di Nvidia, chiamata CUDA[11] con la libreria per lo sviluppo di reti neurali cuDNN[8], che sfrutta l'hardware della scheda grafica (GPU) per accelerare la computazione. In particolare, per questo elaborato, è stata utilizzata la versione di CUDA 10.1, e cuDDN 7.6.4.
Inoltre verranno utilizzati i seguenti package di Python:

- *opencv*[7]: contiene utilities per la gestione di immagini
- *numpy*[25]: contiene utilities per la gestione di array multidimensionali
- *matplotlib*[14]: contiene utilities per la visualizzazione su schermo dei risultati
- *pandas*[14]: contiene utilities per la gestione di tabulati contenuti in files .CSV
- *tkinter*[6]: contiene utilities per la creazione di interfacce grafiche
- *h5py*[10]: contiene utilities per salvare e caricare files .HDF5 contenenti numpy arrays e files riguardanti l'apprendimento della rete neurale

Struttura della tesi

La tesi è composta da quattro capitoli:

- *Capitolo 1*: in questo capitolo verrà trattato il funzionamento di una rete neurale, la struttura di un neurone, l'algoritmo alla base del processo di apprendimento e la struttura dei dati su cui una rete neurale si allena;
- *Capitolo 2*: in questo capitolo verrà trattato, più nello specifico, l'ambito della generazione di nuovi dati attraverso il modello VAE e di come si può condizionare la generazione tramite tecnica FiLM;
- *Capitolo 3*: in questo capitolo verrà analizzato il dataset di volti umani CelebA, la sua gestione per l'apprendimento, la struttura del modello VAE utilizzato, l'integrazione della tecnica FiLM e l'implementazione di un'interfaccia grafica per agevolare la generazione di nuovi volti;
- *Capitolo 4*: in questo capitolo verrà analizzata la metrica del FID per la valutazione della qualità delle immagini ricostruite e generate, verranno esposti degli esempi e, inoltre, verrà fatta una riflessione sulla possibilità di utilizzare il FID per poter discriminare alcuni attributi.

Capitolo 1

Introduzione alle Reti Neurali

1.1 Struttura

Una rete neurale è un modello computazionale, formato da nodi, organizzati in livelli, detti "layers". Ogni rete neurale ha un layer d' ingresso (input) e uno di uscita (output). Quest'ultimo layer fornisce il risultato finale. Tra l'input layer e l'output layer possono esserci un numero arbitrario di altri layers, chiamati layers interni o "nascosti" (hidden layers). Ogni nodo di ogni layer è collegato ad ogni nodo del layer successivo. Queste connessioni sono pesate, ossia hanno un fattore moltiplicativo che indica quanto, il valore di output del nodo, è "significativo".

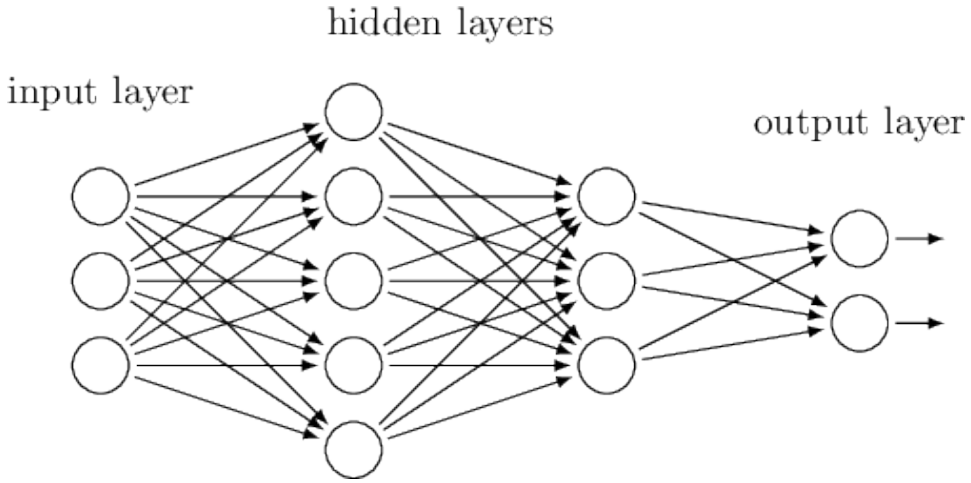


Figura 1.1: Esempio di topologia di una rete neurale.
Fonte: `http://www.ce.unipr.it/people/medici/geometry/node107.html`

1.2 Apprendimento

Per capire come funziona l'apprendimento di una rete neurale, è necessario capire come è strutturato un suo neurone, chiamato neurone artificiale.

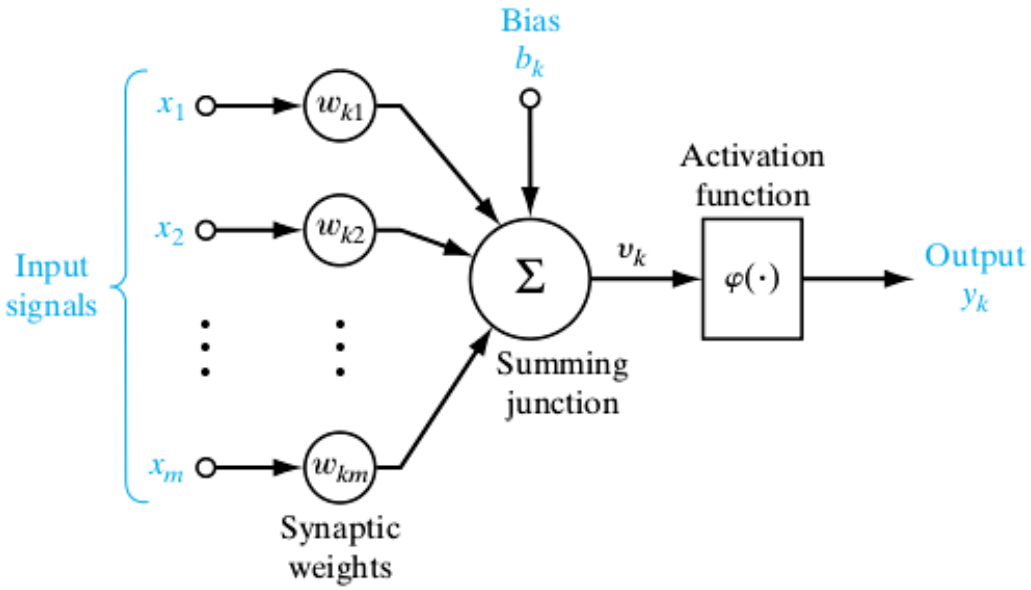


Figura 1.2: Struttura di un neurone artificiale. Fonte:
`https://opensourceforu.com/2017/03/neural-networks-in-detail/`

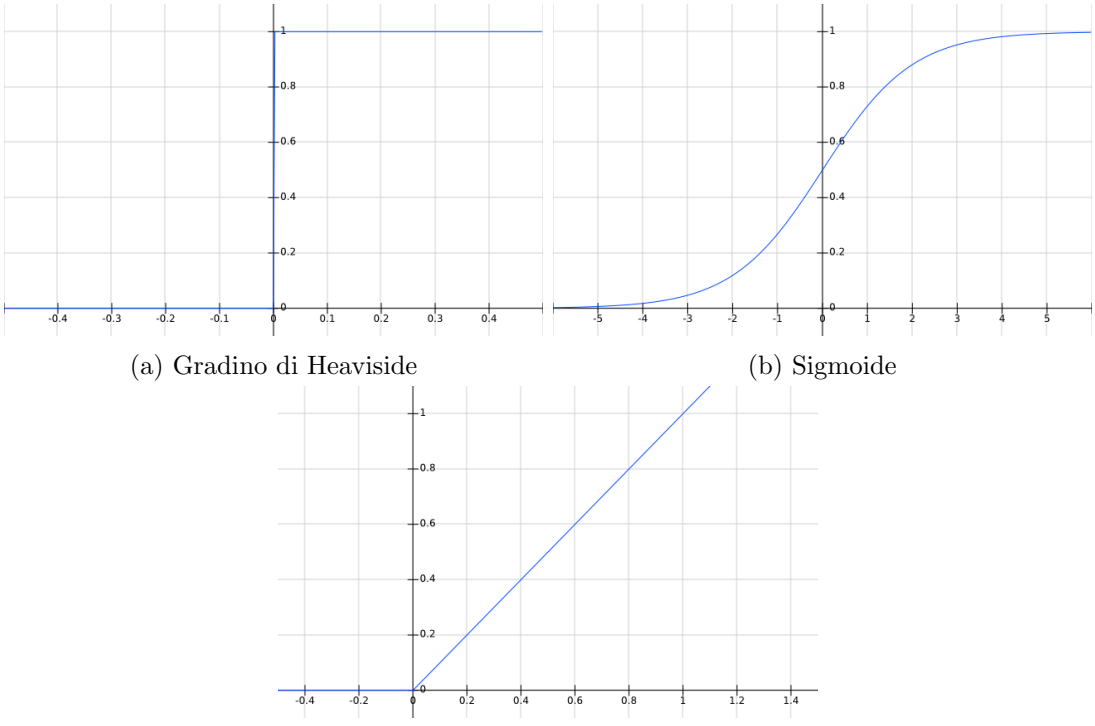
Un neurone artificiale è composto da:

- $x_1, \ldots, x_m$: inputs del neurone;
- $w_{k1}, \ldots, w_{km}$: pesi, uno per ogni input del neurone;
- b_k : peso particolare, chiamato bias, utile per tarare al meglio il neurone;
- Σ : Funzione di somma che calcola il livello di stimolamento generale v_k del neurone;
- $f(\cdot)$: funzione di attivazione che regola l'output y_k del neurone in base al suo livello di stimolamento;

1.2.1 Funzione di attivazione

La funzione di attivazione di un neurone definisce, data la sommatoria degli input pesati, l'output del neurone, cioè definisce quanto il neurone è stimolato, proprio come accade nei neuroni biologici. La funzione di attivazione è una funzione non lineare e derivabile quasi ovunque. Le più comunemente utilizzate sono:

- Step di Heaviside:
$$f(x) = \begin{cases} 0 & \text{se } x < 0 \\ 1 & \text{se } x \geq 0 \end{cases}$$
- Sigmoide:
$$f(x) = \frac{1}{1 + e^{-x}}$$
- Rettificatore:
$$f(x) = \max(0, x)$$



(a) Gradino di Heaviside

(b) Sigmoide

(c) Rettificatore

Figura 1.3: Grafici delle funzioni di attivazione più utilizzate.
Fonte: `http://fooplot.com/`

1.2.2 Backpropagation

La Backpropagation (Retro-propagazione dell'errore)[23] è uno degli algoritmi per l'apprendimento automatico supervisionato, usato spesso in combinazione con sistemi di ottimizzazione come la discesa stocastica del gradiente. Per apprendimento supervisionato si intende un apprendimento in cui la rete "impara" da un insieme di dati che sono già classificati.
L'algoritmo opera in questo modo:

- I dati attraversano l'intera rete (Forward Propagation), arrivando al livello di output e calcolando un risultato, detto "risultato predetto";
- Durante la Forward Propagation, vengono calcolati i pesi di ogni neurone;
- Si confronta l'output della rete con il risultato atteso e si calcola l'errore attraverso una funzione di costo;
- Si calcola il gradiente della funzione di costo che indica di quanto variare i pesi della rete per poter diminuire l'errore;
- La correzione avviene aggiornando i pesi "a catena", partendo dal livello di output, fino ad arrivare al livello di input;
- Si ripete il procedimento più volte, finché l'errore non si ritiene trascurabile;

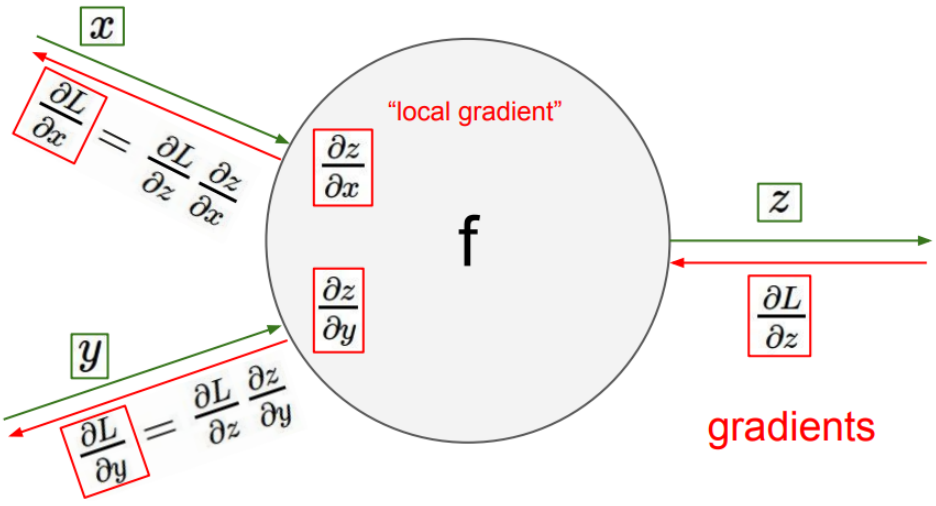


Figura 1.4: Rappresentazione dell'algoritmo di Backpropagation.
Fonte: `https://bishwarup307.github.io/deeplearning/convbackprop/`

1.2.3 Struttura dei dataset

Un dataset è costituito da una collezione di dati, anche di diversa natura, organizzati in modo razionale. Un approccio molto comune e molto utilizzato per l'apprendimento automatico consiste nell'organizzare i dati forniti dal dataset in tre subsets distinti:

- *Training Set*: parte del dataset usato per l'apprendimento, cioè per il calcolo e la taratura dei pesi della rete neurale. Il training set è il set di dati con cui avviene la Forward Propagation;
- *Validation Set* e *Test Set*: parti del dataset usate per il controllo delle performance della rete neurale e per la valutazione degli iperparametri, cioè parametri definiti prima dell'allenamento della rete. I dati presenti in questi subsets devono essere diversi dai dati presenti nel training set, questo perché le performance della rete neurale devono essere valutate utilizzando dati che la rete non ha mai visto prima;

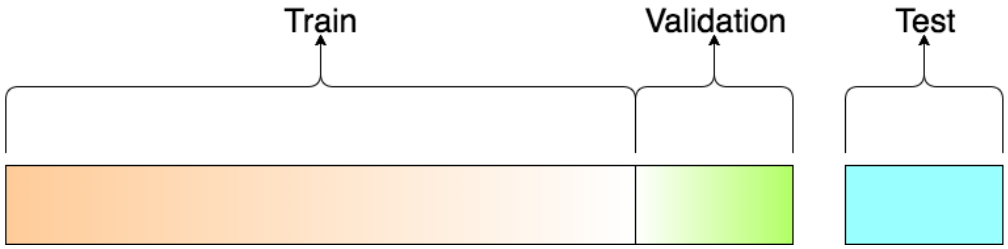


Figura 1.5: Rappresentazione del ratio con cui si divide un dataset.
Fonte:
`https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7/`

1.2.4 Funzioni di costo

Una funzione di costo è una particolare funzione che misura l'errore della rete neurale, cioè indica di quanto, l'output ottenuto, si discosta dall'output aspettato.
Esistono diverse funzioni di costo, le più importanti e utilizzate sono:

- *Cross-Entropy (CE)*: è una misura della differenza tra due distribuzioni di probabilità per un dato insieme X di variabili o eventi (o classi di appartenenza). La Cross-Entropy è il numero medio di bits usati per codificare dati che vengono da una sorgente che ha distribuzione P, usando un modello che ha una distribuzione Q, che approssima P[20]:

$$CE = H(P, Q) = -\sum_{x \in X} P(x) \cdot log[Q(x)]$$

- *Mean Squared Error (MSE)*: l'errore quadratico medio è una misura dello scarto quadratico medio fra l'output aspettato y_i e l'output ottenuto $\widetilde{y_i}$:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \widetilde{y_i})^2$$

- *Divergenza di Kullback-Leibler ($\mathcal{KL}$)*: misura quanto bene la distribuzione Q approssima la distribuzione P. In altre parole, dal momento che, per codificare, stiamo usando l'approssimazione Q, invece che la distribuzione P, la divergenza $\mathcal{KL}$ è il numero medio di bits extra che sono necessari per codificare i dati[20]:

$$\mathcal{KL}(P \| Q) = -\sum_{x \in X} P(x) \cdot log\left[\frac{Q(x)}{P(x)}\right] = H(Q, P) - H(P)$$

  Dove $H(Q, P)$ è la Cross-Entropy di Q da P e $H(P)$ è l'entropia di P

1.3 Introduzione agli Autoencoders

Un autoencoder è un modello di rete neurale che opera dapprima comprimendo l'input in uno spazio di variabili latenti e, successivamente, ricostruendo l'output sulla base delle informazioni acquisite[1]. Questa tipologia di rete neurale è composta da due sottoreti:

- *Encoder*: comprime i dati in input in uno spazio di variabili latenti, chiamato rappresentazione interna;
- *Decoder*: tenta di ricostruire l'input sulla base delle informazioni precedentemente raccolte, utilizzando i dati presenti nella rappresentazione interna;

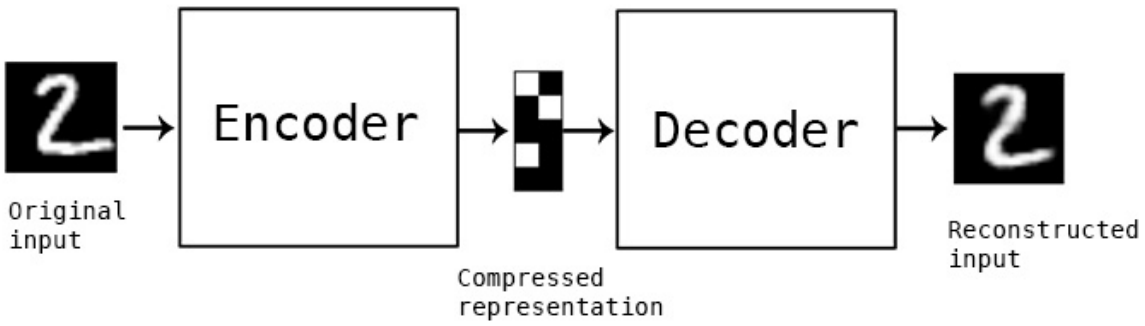


Figura 1.6: Schema di funzionamento di un autoencoder.
Fonte: `https://blog.keras.io/building-autoencoders-in-keras.html`

[1]Nathan Hubens. *Introduzione agli autoencoder*. Mag. 2018. URL: `https://www.deeplearningitalia.com/introduzione-agli-autoencoder/`

Capitolo 2

Generazione di nuovi dati

2.1 Variational Autoencoders

Un classico autoencoder genera uno spazio latente in cui le variabili latenti significative sono molto sparse, ciò significa che, prendendo un punto random nello spazio latente, è molto improbabile "pescare" una variabile latente con cui il decoder riesca effettivamente a generare dati verosimili.
Per poter usare, quindi, il decoder, come un generatore di nuovi dati, si deve organizzare lo spazio latente durante l'allenamento in modo che ci sia della regolarità nella distribuzione delle variabili latenti. In questo modo il decoder acquisisce delle proprietà generative.[1]

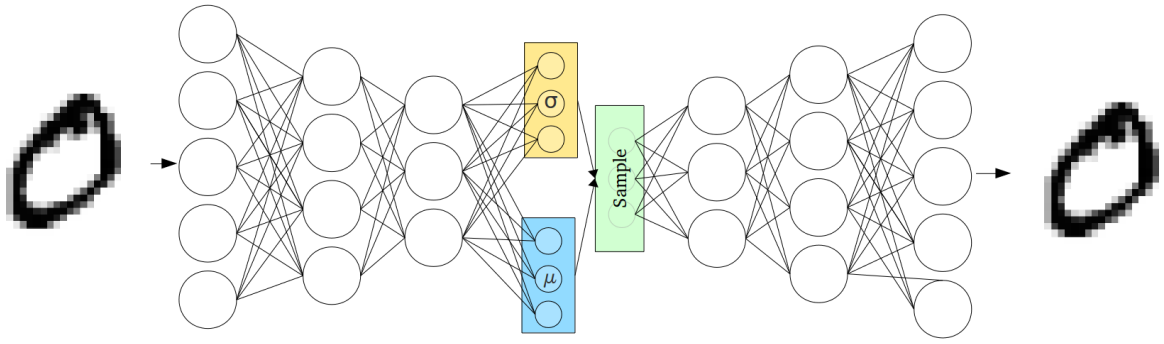


Figura 2.1: Struttura di un Variational Autoencoder.
Fonte: `https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf`

[1]Rocca Joseph. *Understanding Variational Autoencoders (VAEs) - Towards Data Science*. Set. 2019. URL: `https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73`

2.1.1 Variance Law e variabili inattive

La differenza principale tra un autoencoder classico e un VAE è che, quest'ultimo, non codifica l'input x come un singolo vettore, ma, per ogni variabile latente z, calcola una media $\mu_z(x)$ e una varianza $\sigma_z^2(x)$.
Lo spazio latente di un VAE è quindi caratterizzato da tante distribuzioni di probabilità, definite dalle medie e dalle varianze di ogni variabile latente. Durante l'apprendimento, idealmente, μ_z si avvicina sempre di più a 0, e σ_z^2 si avvicina sempre di più a 1, formando uno spazio latente con distribuzione di probabilità simile alla distribuzione normale[2].

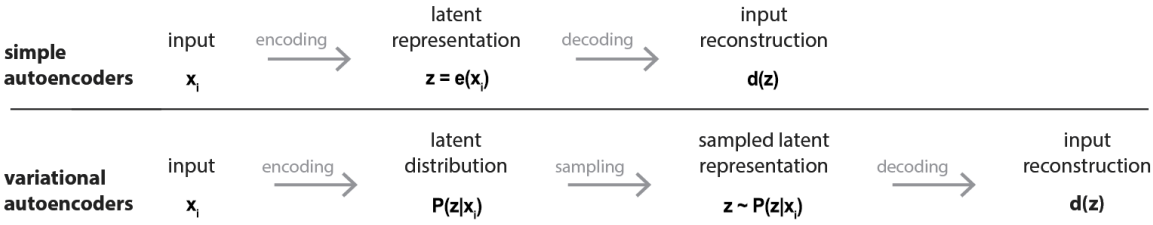


Figura 2.2: Differenza di funzionamento tra un AE e un VAE.
Fonte: `https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73`

L'aggiunta della $\mathcal{KL}$ nella funzione di costo di un VAE, previene che la rete impari a "copiare" l'input, piuttosto che "capirne" le caratteristiche, per questo la $\mathcal{KL}$ può essere interpretata come un regolarizzatone della posizione delle variabili latenti[22].
Per effetto della $\mathcal{KL}$, alcune variabili latenti possono essere "ignorate" dal VAE, inducendo ad una "carenza" (sparsity[3]) nel processo di encoding. Le variabili ignorate si dicono "collassate" oppure inattive e sono caratterizzate dall'avere, per ogni sample x, media $\mu_z(x)$ vicina a 0 e varianza $\sigma_z^2(x)$ vicina ad 1[5].

Di seguito l'implementazione in Keras del calcolo della Variance Law e del numero di variabili inattive:

```
for i in range(step):
    inactive_vars_count = 0
    z_logvar, z_mean, _ \
        = encoder[i].predict([test_images], batch_size=100)
    mean_var = np.mean(np.exp(z_logvar), axis=0)
    var_z = np.var(mean_var, axis=0)
    sum_z = mean_var + var_z

    for j in range(0, latent_dim):
        if mean_var[j] > .9:
            inactive_vars_count += 1

    variance_law = np.mean(sum_z)
```

Algoritmo 2.1: Codice che calcola la Variance Law e il numero di variabili inattive

2.1.2 Forma contratta della KL-Divergence

In un VAE si cerca di rendere ogni distribuzione $Q(x|z)$ ottenuta dall'encoder quanto più simile possibile alla distribuzione di partenza $P(z)$, minimizzando la loro divergenza di Kullback-Leibler $\mathcal{KL}\Big[Q(z|x)\|P(z)\Big]$.
$Q(x|z)$ è molto simile ad una gaussiana $\mathcal{N}(\mu_z(x), \sigma_z(x))$ e, per semplicità, si assume che la distribuzione di partenza $P(z)$ sia una distribuzione normale $\mathcal{N}(0, 1)$.
Dunque, il termine $\mathcal{KL}\Big[Q(z|x)\|P(z)\Big]$ non è altro che la $\mathcal{KL}$-Divergence tra due distribuzioni gaussiane: $\mathcal{N}(\mu_z(x), \sigma_z(x))$ e $\mathcal{N}(0, 1)$ e quindi può essere espressa nella seguente forma contratta (calcoli tratti da [4]):

$$\mathcal{KL}\Big[\mathcal{N}\big(\mu_z(x), \sigma_z(x)\big)\|\mathcal{N}(0, 1)\Big] = \frac{1}{2}\Big[\mu_z(x)^2 + \sigma_z^2(x) - log\big[\sigma_z^2(x)\big] - 1\Big]$$

2.2 Feature-wise Linear Modulation

Feature-wise Linear Modulation, abbreviato "FiLM", è una tecnica per influenzare in modo adattivo l'output di una rete neurale basandosi su un input arbitrario, applicando una trasformazione affine (o semplicemente FiLM), sulle features intermedie della rete[21]. Per trasformazione affine si intende una trasformazione del tipo: $y = m \cdot x + b$. FiLM apprende due funzioni f ed h, che restituiscono in output due valori: $\gamma_{i,c}$ e $\beta_{i,c}$, che sono in funzione dell'input x_i :

$$f_c(x_i) = \gamma_{i,c} \qquad h_c(x_i) = \beta_{i,c}$$

I valori $\gamma_{i,c}$ e $\beta_{i,c}$ vanno a modulare le attivazioni $\boldsymbol{F}_{i,c}$ di una rete neurale, attraverso una trasformazione affine a tutte le features; i pedici i e c si riferiscono alla c-esima feature dell' i-esimo input:

$$FiLM(\boldsymbol{F}_{i,c}|\gamma_{i,c}, \beta_{i,c}) = \gamma_{i,c} \cdot \boldsymbol{F}_{i,c} + \beta_{i,c}$$

Nella pratica è molto più semplice riferirsi ad f e h come un'unica funzione (o un'altra rete neurale), che restituisce in output il vettore (γ, β). Questa funzione unica è detta *FiLM Generator* e le reti neurali che integrano dei FiLM layers sono dette: *Feature-wise Linearly Modulated Network*, (*FiLM-ed Network*)[21].

2.2.1 FiLM Layer

Il FiLM Layer è il layer che esegue la trasformazione affine su tutte le features. Con "su tutte le feature" si intende che le operazioni di scaling (la moltiplicazione) e lo shifting (l'addizione) vengono applicate su ogni feature map (canali dell'immagine). In altre parole, si hanno:

- x: input del FiLM layer
- x_aux: input ausiliario di condizionamento
- γ: vettore di scaling
- β: vettore di shifting

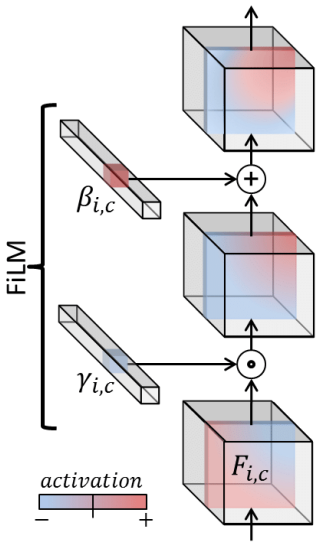


Figura 2.3: FiLM Layer. Fonte: `https://www.profillic.com/s/Ethan%20Perez`

Capitolo 3

Implementazione

3.1 CelebFaces Attributes Dataset

Il CelebFaces Attributes Dataset, abbreviato "CelebA", è una collezione di 202599 immagini di volti di celebrità, con una tabella che dichiara quali attributi sono presenti, o assenti, in ognuna delle immagini. La tabella dichiara 40 attributi, come per esempio: sesso del soggetto, colore e il taglio dei capelli, forma delle labbra e del naso, espressione del viso, presenza o assenza di occhiali e molti altri.

3.1.1 Volti

Nel dataset sono presenti 202599 immagini di volti. Ogni immagine ha una risoluzione di 178×218 pixel. Le immagini presentano una grande varietà di soggetti, pose e sfondi e possono essere usate per allenare reti che si occupano, ad esempio, di riconoscimento di attributi facciali, rilevamento di volti, editing di volti e sintetizzazione di volti nuovi.

Figura 3.1: Esempi di volti presenti nel dataset.
Fonte: `http://mmlab.ie.cuhk.edu.hk/projects/CelebA/`

3.1.2 Attributi

Ad ogni immagine nel dataset viene associata una entry nella tabella degli attributi. La tabella contiene 41 colonne: la prima colonna contiene il nome dell' immagine e le restanti 40 colonne contengono il valore dei singoli attributi. Il valore di ogni attributo può essere:

- -1: attributo non presente nel volto
- 1: attributo presente nel volto

Index	Definition	Index	Definition	Index	Definition	Index	Definition
1	5o'ClockShadow	11	Blurry	21	Male	31	Sideburns
2	ArchedEyebrows	12	BrownHair	22	MouthSlightlyOpen	32	Smiling
3	Attractive	13	BushyEyebrows	23	Mustache	33	StraightHair
4	BagsUnderEyes	14	Chubby	24	NarrowEyes	34	WavyHair
5	Bald	15	DoubleChin	25	NoBeard	35	WearingEarrings
6	Bangs	16	Eyeglasses	26	OvalFace	36	WearingHat
7	BigLips	17	Goatee	27	PaleSkin	37	WearingLipstick
8	BigNose	18	GrayHair	28	PointyNose	38	WearingNecklace
9	BlackHair	19	HeavyMakeup	29	RecedingHairline	39	WearingNecktie
10	BlondHair	20	HighCheekbones	30	RosyCheeks	40	Young

Figura 3.2: Lista degli attributi presenti nel dataset, con il proprio indice.
Fonte: `https://www.researchgate.net/publication/327029519_Fine-grained_face_annotation_using_deep_Multi-Task_CNN`

3.2 Fasi Preliminari

Nell'ambito delle reti neurali e dell'apprendimento automatico, i dati vengono trattati come numeri in virgola mobile. La componente hardware che più di tutte è efficiente nel calcolo in virgola mobile è la scheda grafica (GPU).
Gestire gli enormi datasets contenti immagini di grandi dimensioni richiede un'architettura hardware dai costi spropositati, quindi, è molto comune ridimensionare le immagini fino a 64×64 pixel. In questo modo il training risulta molto più veloce e richiede molta meno memoria video (VRAM) e potenza computazionale.[18].

Figura 3.3: In alto: volti presenti nel dataset, con risoluzione 218×178. In basso: volti ritagliati e ridimensionati, con risoluzione 64×64

3.2.1 Caricamento delle immagini

La prima parte di CelebA è composta da 202599 immagini con una risoluzione di 218×178 pixel e un nome progressivo, numerico, da "000001.jpg" fino a "202599.jpg". Di seguito l'implementazione della funzione che legge, ritaglia e ridimensiona le immagini:

```
def load_celeba_imgs(imgs_names, dir_data):
    imgs_list = []
    for i, id in enumerate(imgs_names):
        path = dir_data + "/" + id
        image = cv2.imread(path)[...,::-1]/255.0
        image = image [45:173, 25:153]
        image = cv2.resize(src=image, dsize=(64, 64))
        imgs_list.append(image)
    return np.array(imgs_list)
```

Algoritmo 3.1: Codice che ridimensiona le immagini per prepararle al training

Data una lista dei nomi delle immagini da ridimensionare e il percorso della directory in cui esse si trovano, per ogni immagine, l'algoritmo 3.1 opera in questo modo:

1. Legge l'immagine dalla directory, la converte in formato RGB e la normalizza;
2. Ritaglia l'immagine, facendola diventare quadrata;
3. Ridimensiona l'immagine, portandola ad una risoluzione di 64×64 pixel.
4. Converte tutte le immagini ridimensionate in un numpy array

3.2.2 Caricamento degli attributi

La seconda parte di CelebA è composta da un file .CSV, contenente una tabella in cui, per ogni immagine, vi sono i valori di ogni attributo. Di seguito l'implementazione della funzione legge il file degli attributi e ne salva il contenuto in un numpy array:

```
1 def load_celeba_attributes(dir_data, file_name):
2     file_content = pd.read_csv(dir_data + file_name)
3     attributes = np.array(file_content)[:, 1:]
4     return attributes
```

Algoritmo 3.2: Codice che legge il file .CSV degli attributi

Dato il percorso e il nome del file .CSV, l'algoritmo 3.2 opera in questo modo:

1. Legge l'intero contenuto del file .CSV;
2. Attraverso [:, 1:] esclude la prima colonna della tabella (nomi delle immagini);
3. Converte gli attributi letti in un numpy array.

3.3 Caricamento del dataset per il training

Preparare il dataset per l'allenamento richiede la suddivisione delle 202599 immagini e dei relativi attributi nei tre subsets:

- *Training Set*: Immagini e attributi dalla 1 alla 162770;
- *Validation Set*: Immagini e attributi dalla 162771 alla 182637;
- *Test Set*: Immagini e attributi dalla 182638 alla 202599;

Di seguito l'implementazione:

```
imgs_dir = "data/img_align_celeba/"
attr_dir = "./data/"
attr_file_name = "list_attr_celeba.csv"
imgs_names_array = np.sort(os.listdir(imgs_dir))
imgs_names_array_train = imgs_names_array[1:162770]
imgs_names_array_val = imgs_names_array[162771:182637]
imgs_names_array_test = imgs_names_array[182638:202599]

x_train = load_celeba_imgs(imgs_names_array_train, imgs_dir)
x_val = load_celeba_imgs(imgs_names_array_val, imgs_dir)
x_test = load_celeba_imgs(imgs_names_array_test, imgs_dir)

celeba_attributes = load_celeba_attributes(attr_dir,
   attr_file_name)
y_train = celeba_attributes[1:162770]
y_val = celeba_attributes[162771:182637]
y_test = celeba_attributes[182638:202599]
```

Algoritmo 3.3: Codice che carica e divide il dataset nei sets per il training, testing e validation

3.4 Implementazione FiLM Layer

Di seguito l'implementazione di un FiLM layer, che include al suo interno il FiLM Generator che calcola i due vettori di scaling e shifting γ e β:

```
def FiLM_module(args):
    input, input_aux = args
    height  = input.shape[1]
    width = input.shape[2]
    channels_no = input.shape[-1]
    #FiLM generator
    FiLM_params = Dense(2 * channels_no)(input_aux)
    FiLM_params = K.expand_dims(FiLM_params, axis=[1])
    FiLM_params = K.expand_dims(FiLM_params, axis=[1])
    FiLM_params = K.tile(FiLM_params, [1, height, width, 1])
    # Split FiLM_params into scaling and shifting vectors
    gammas = FiLM_params[:, :, :, :channels_no]
    betas = FiLM_params[:, :, :, channels_no:]
    # Apply affine transformation
    return (1 + gammas) * input + betas
```

Algoritmo 3.4: Codice che applica la trasformazione affine su tutte le features

3.5 Implementazione Modello

Il modello è stato sviluppato usando l'approccio *Functional API* del framework Keras. La scelta è dettata dal fatto che il modello ha come input sia un'immagine e sia un vettore con gli attributi corrispondenti. Nell'approccio *Functional API* ogni livello o persino modello può essere chiamato come se fosse una funzione. I livelli più importanti, nel modello sviluppato in questo elaborato, sono:

- *Conv2D* e *Conv2DTranspose*: Questi livelli operano sulle immagini. Conv2D permette di estrarre features, facendo al contempo un downsampling dell'immagine. Conv2DTranspose permette di ricostruire, partendo dalle features, l'immagine, facendo quindi un upsampling. Questi due livelli hanno diversi parametri:
  - *filters*: numero di canali che si vogliono ottenere in output. Inizialmente l'immagine in input ha 3 canali (RGB), nell'encoder di questo modello il numero di canali, da un livello più esterno ad uno più interno, cresce di una potenza di 2; in particolare parte da 32, fino ad arrivare a 256 canali. Il decoder è strutturato in modo simmetrico all'encoder;
  - *kernel_size*: dimensione in pixels della finestra di scansione dell'immagine; in questo modello è grande, per ogni livello, 5×5;
  - *strides*: passo di avanzamento (numero di pixels) del kernel, sia sull'asse orizzontale, che su quello verticale. Il passo è stato impostato a 2 per far dimezzare (e successivamente raddoppiare) le dimensioni dell'immagine;
  - *padding*: può essere impostato su due valori: *Valid* e *Same*. Il primo parametro permette di non impostare nessun tipo di padding, mentre il secondo aggiunge un padding all'esterno dell'immagine in modo tale da ridurre (o aumentare) in modo uniforme la dimensione dell'immagine. In questo modello il padding è impostato su *Same*;
  - *activation*: funzione di attivazione da utilizzare sull'output del livello. In questo modello è stata utilizzata la funzione Rettificatore *relu*.
- *Dense*: anche chiamato *Fully Connected Layer*, è un livello in cui ogni neurone in input è collegato con ogni neurone in output. Come parametro ha il numero di unità (collegamenti) che si vogliono in output.
- *Flatten*: livello che esegue un'operazione di "appiattimento" dell'input. Più nel dettaglio restituisce un output che ha come forma il numero di elementi dell'input e come contenuto la "vettorializzazione" di ogni dato presente nell'input;
- *Lambda*: livello che permette di incapsulare una normale funzione all'interno di un livello Keras. In questo modello è stato scelto di usare il Lambda Layer per incapsulare la funzione di sampling dallo spazio latente

3.5.1 Implementazione Encoder

L'encoder è la parte del modello che si occupa di codificare l'immagine in input in una distribuzione di variabili all'interno dello spazio latente. L'encoder ha come input l'immagine originale e gli attributi presenti e determina il modo in cui la rete neurale apprende ogni feature. L'encoder si occupa di calcolare la media e la varianza di ogni variabile latente.
Di seguito l'implementazione:

```
input_img = Input(shape=(64, 64, 3))
input_label = Input(shape=(40,))

x = Conv2D(32, (5,5), strides=2, 'same', 'relu')(input_img)
x = Conv2D(64, (5,5), strides=2, 'same', 'relu')(x)
x = Conv2D(128, (5,5), strides=2, 'same', 'relu')(x)
x_save_128 = x
x = Conv2D(256, (5,5), strides=2, 'same', 'relu')(x)
x_save_256 = x

x = Flatten()(x)
z_mean1 = Dense(latent_dim)(x)
z_logvar1 = Dense(latent_dim)(x)
```

Algoritmo 3.5: Implementazione encoder del VAE

3.5.2 Implementazione Decoder

Il decoder è la parte del modello che si occupa di ricostruire (o generare) l'immagine. La ricostruzione viene influenzata attraverso i moduli di correzione e la tecnica FiLM, in modo tale da restituire (o escludere) gli attributi facciali richiesti, nel volto generato. Il decoder ha come input il sampling ottenuto campionando lo spazio latente e come output l'immagine ricostruita (o generata).
Di seguito l'implementazione:

```
z_samp1 = Lambda(sampling, output_shape=(latent_dim,))([z_mean1,
    z_logvar1])

y = FiLM_module(z_samp1, input_label)
y = Dense(4*4*256)(y)
y = Reshape((4,4,256))(y)

z_mean2, \
z_logvar2, \
z_samp2 = enc_module(x_save_256, y, 256, latent_dim)
y = correction_module(y, z_samp2, input_label, 256)

y = Conv2DTranspose(128, (5,5), strides=2, 'same', 'relu') (y)

z_mean3, \
z_logvar3, \
z_samp3 = enc_module(x_save_128, y, 128, latent_dim)
y = correction_module(y, z_samp3, input_label, 128)

y = Conv2DTranspose(64, (5,5), strides=2, 'same', 'relu')(y)
y = Conv2DTranspose(32, (5,5), strides=2, 'same', 'relu')(y)
res = Conv2DTranspose(3, (5,5), strides=2, 'same', 'sigmoid')(y)
```

Algoritmo 3.6: Implementazione decoder del VAE

3.5.3 Implementazione sistema di correzione

Il meccanismo di correzione si compone di due moduli aggiuntivi: l' *enc_module* e il *correction_module*. Entrambi i moduli entrano in gioco nelle fasi iniziali del processo di decoding, per cercare di migliorare ulteriormente le performance ricostruttive e generative del decoder.

Implementazione *enc_module*

L'idea alla base dell'*enc_module* è quella di cercare di "capire" qualche informazione in più, utilizzando output intermedi dell'encoder e del decoder, calcolando una nuova media e una nuova varianza e "aggiustando" di conseguenza il sampling. Di seguito l'implementazione:

```
def enc_module(x, y, channels_no, latent_dim):
    x = Concatenate()([x,y])
    x = Conv2D(channels_no, (5, 5), strides=2, 'same', 'relu')(x)
    x = Conv2D(channels_no*2, (5, 5), strides=2, 'same')(x)
    x = Flatten()(x)
    z_mean = Dense(latent_dim)(x)
    z_logvar = Dense(latent_dim)(x)
    z_samp = Lambda(sampling, output_shape=(latent_dim,))([z_mean
   , z_logvar])
    return z_mean, z_logvar, z_samp
```

Algoritmo 3.7: Implementazione della funzione *enc_module*

Implementazione *correction_module*

L'idea alla base del *correction_module* è quella di correggere la decodifica del decoder. La correzione avviene concatenando il sampling precedentemente migliorato agli attributi e utilizzando un FiLM_module per modulare la ricostruzione. Poi la correzione viene sommata al precedente output intermedio del decoder. Di seguito l'implementazione:

```
def correction_module(y, z, input_label):
    z = Concatenate()([z, input_label])
    correction_y = FiLM_module(y, z)
    y_corrected = Add()([y, correction_y])
    return y_corrected
```

Algoritmo 3.8: Implementazione della funzione *correction_module*

3.6 Interfaccia Grafica

Per poter inizializzare gli attributi è stata sviluppata una comoda interfaccia grafica user-friendly. La GUI è stata sviluppata utilizzando gli oggetti *Combobox* e *Button* del package "tkinter[6]". Attraverso la GUI l'utente, per ogni attributo, può scegliere uno tra i seguenti valori:

- *None*: il valore dell'attributo sarà inizializzato dal programma;
- 1: l'attributo sarà presente nei volti generati;
- -1: l'attributo non sarà presente nei volti generati;

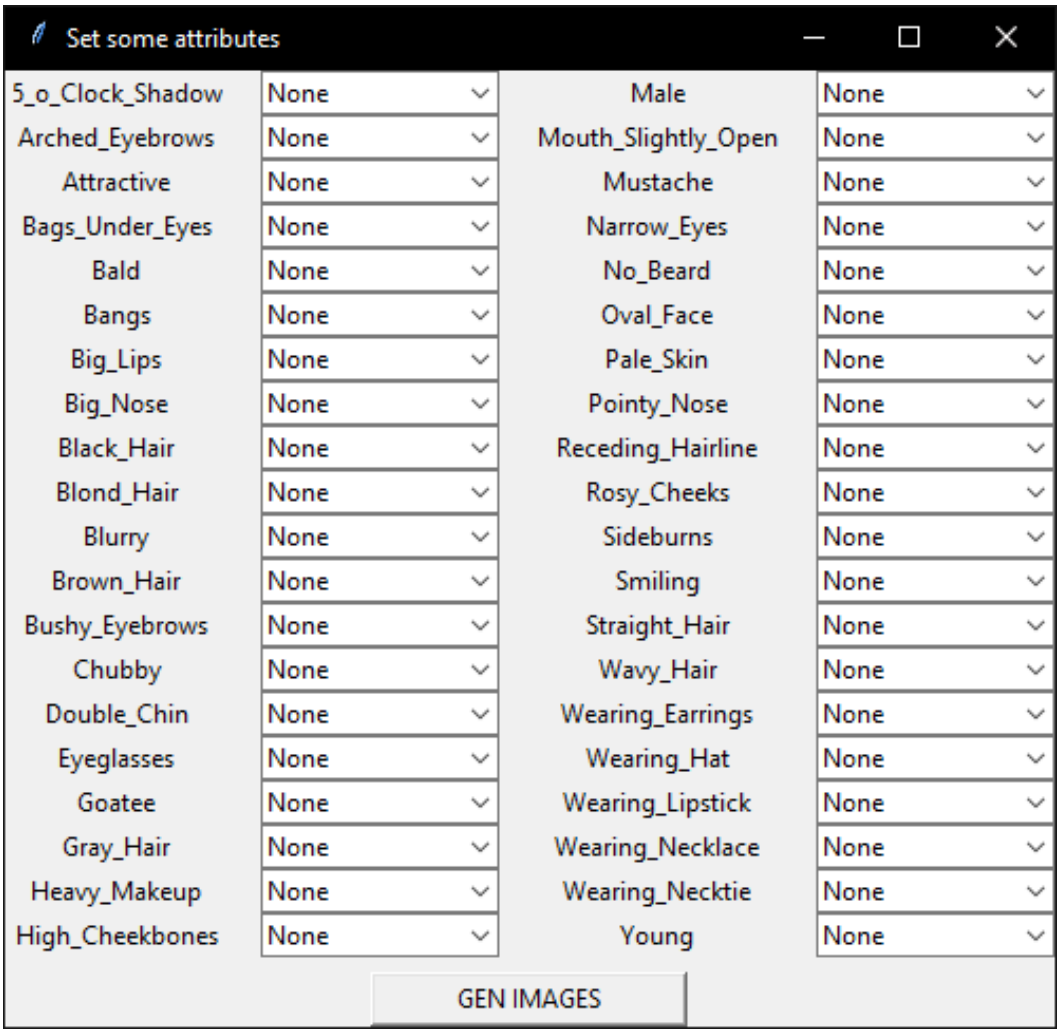


Figura 3.4: Interfaccia grafica

3.7 Generazione di volti umani

Il pulsante "GEN IMAGES" richiama la funzione *gen_images* che memorizza tutti i valori degli attributi settati dall'utente, inizializza i restanti attributi e richiama la generazione dei volti da parte del decoder. Di seguito l'implementazione:

```
def gen_images():
    attr = []
    for i in range (0, len(combo_boxes)):
        value = combo_boxes[i].get()
        if value != "None":
            attributes.append((i, int(value)))
    celeba_attributes = load_celeba_attributes()
    attr = random_attr_init(celeba_attributes, attr)
    plot_gen(6, decoder, attr)
```

Algoritmo 3.9: Codice che salva i valori degli attributi settati dall'utente e inizializza gli attributi mancanti

La generazione dei volti avviene specificando, alla funzione *plot_gen*, il numero di volti che si vuole generare, il decoder da utilizzare e gli attributi su cui si basa la generazione. La visualizzazione avviene usando il package "matplotlib[14]", che fornisce comodi strumenti per la visualizzazione di immagini e grafici. Di seguito l'implementazione:

```
def plot_gen(num, decoder, attributes):
    latent_dim = 64
    label = np.array(([attributes]))
    label = np.repeat(label, repeats=num, axis=0)
    z_samp1 = np.random.normal(size=(num, latent_dim))
    z_samp2 = np.random.normal(size=(num, latent_dim))
    z_samp3 = np.random.normal(size=(num, latent_dim))
    res = decoder([z_samp1, z_samp2, z_samp3, label])[0]
    res = res.reshape(num,64,64,3)
    plt.figure(figsize=(15, 2))
    for i in range(0, num):
        axis = plt.subplot(1, num, i+1)
        plt.imshow(res[i])
    plt.show()
```

Algoritmo 3.10: Codice genera i volti e li visualizza

L'algoritmo 3.10 funziona in questo modo:

1. Converte gli attributi scelti e li ripete tante volte quante sono le immagini che si vogliono generare;
2. Campiona tre valori random da una distribuzione normale, ottenendo così i tre samples da dare in input al decoder;
3. Invoca il decoder, fornendo in input i tre samples e gli attributi, fornendo in output le immagini generate;
4. Crea una finestra in cui visualizza i volti generati;

Capitolo 4

Valutazioni finali

4.1 Fréchet Inception Distance

La Fréchet Inception Distance, abbreviato "FID" è una metrica usata per valutare la qualità delle immagini generate o ricostruite. Più in generale, la distanza di Fréchet misura la distanza tra due distribuzioni di probabilità.
Considerando l'aspetto implementativo, FID misura, in particolare, la distanza tra le distribuzioni di un modello Keras chiamato Inception-v3, pre-allenato[24]. Esse sono costituite dai vettori delle attivazioni dell'ultimo Pooling Layer del modello, chiamato Global Spatial Pooling Layer.
Per ogni immagine viene calcolato un vettore con 2048 attivazioni, chiamato Coding Vector oppure Feature Vector [1]. Più precisamente, dato un insieme di immagini reali $\{x_i\}_{i\in\mathbb{N}}$ e un insieme di immagini ricostruite o generate $\{x'_i\}_{i\in\mathbb{N}}$, le rispettive distribuzioni saranno: $\mathcal{N}(\mu, \Sigma)$ e $\mathcal{N}(\mu', \Sigma')$ e la Fréchet Inception Distance è definita come segue:

$$FID = \left\|\mu - \mu'\right\|^2 + Tr\Big(\Sigma + \Sigma' - 2 \cdot \sqrt{\Sigma \cdot \Sigma'}\Big)$$

[1] Brownlee Jason. *How to Implement the Frechet Inception Distance (FID) for Evaluating GANs.* Ago. 2019. URL: https://machinelearningmastery.com/how-to-implement-the-frechet-inception-distance-fid-from-scratch/

Di seguito l'implementazione in Keras del calcolo del FID, compatibile anche con Tensorflow 2.0 e versioni successive.

```
from keras.applications.inception_v3 \
    import preprocess_input,\
    InceptionV3

inseption_model = InceptionV3(include_top=False,
                              pooling='avg',
                              input_shape=input_shape)

def calculate_fid(images1, images2):
    images1 = scale_images(images1, (299, 299, 3) )
    images2 = scale_images(images2, (299, 299, 3) )
    images1 = preprocess_input(images1)
    images2 = preprocess_input(images2)

    act1 = inseption_model.predict(images1)
    act2 = inseption_model.predict(images2)

    mu1, sigma1 = act1.mean(axis=0), cov(act1, rowvar=False)
    mu2, sigma2 = act2.mean(axis=0), cov(act2, rowvar=False)

    ssdiff = numpy.sum((mu1 - mu2)**2.0)
    covmean = sqrtm(sigma1.dot(sigma2))

    if iscomplexobj(covmean):
        covmean = covmean.real

    fid = ssdiff + trace(sigma1 + sigma2 - 2.0 * covmean)
    return fid
```

Algoritmo 4.1: Codice che prepara le immagini per il modello Inception-v3 e ne calcola la Fréchet Inception Distance

4.2 Problematiche riscontrate

Lavorando con migliaia di immagini, uno dei problemi più grandi consiste nell'enorme capacità di calcolo necessaria. Allenamento, calcolo del FID e testing sono stati eseguiti su un Laptop equipaggiato con CPU i7 4720HQ, 16 GB di RAM, GPU Nvidia GTX 860 con 2 GB di memoria video e un SSD Samsung 850 EVO. L'architettura utilizzata ha limitato fortemente la velocità generale di ogni processo e il numero di features che la rete neurale è stata in grado di processare.
Tensorflow attua dei meccanismi interni che consentono, in alcune occasioni, l'uso sia di VRAM e sia di RAM, ma non tutti i tipi calcoli possono beneficiarne e, perciò, sono stati frequenti errori critici di tipo Out Of Memory (OOM). Una piattaforma hardware più potente sarebbe stata in grado di processare più features e, di conseguenza, avrebbe prodotto risultati teoricamente migliori.
Un altro problema nasce dal fatto che gli attributi non sono distribuiti equamente all'interno del dataset; ad esempio ci sono moltissime immagini di volti senza barba ma pochissime immagini di uomini pelati.

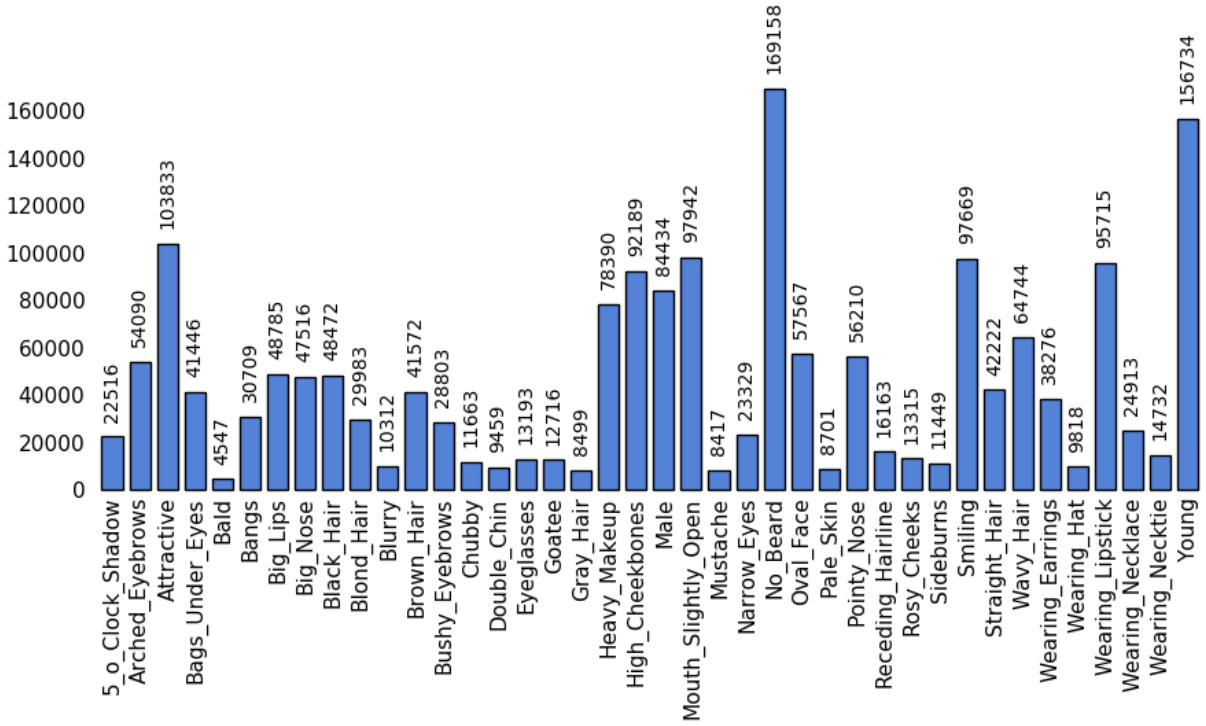


Figura 4.1: Numero di occorrenze dei vari attributi nelle immagini del dataset

4.3 Testing e calcolo delle metriche

Per valutare le performance del modello e la qualità dei volti ricostruiti e generati, sono stati presi in esame subsets di 10000 volti reali, 10000 volti ricostruiti e 10000 volti generati. Successivamente è stato calcolato il FID prima tra i volti reali e i volti ricostruiti (FID_{REC}) ed in seguito tra i volti reali e i volti generati (FID_{GEN}).
Le immagini che compongono il subset dei volti reali vengono selezionate, dal dataset, in modo del tutto casuale; mentre il subset dei volti generati viene creato inizializzando 10000 liste di attributi, per generare ogni singolo volto.
In seguito ogni immagine è stata ridimensionata a 299×299 pixels e i valori di ogni singolo pixel sono stati normalizzati per rientrare nel range -1, 1. Ciò è stato necessario in quanto il modello Inception-v3 di Keras, usato nel calcolo del FID, lavora con queste specifiche.

4.3.1 Aspetto ricostruttivo

Ricostruire l'input significa codificarlo attraverso l'encoder e, successivamente, dare in input la codifica appena ottenuta al decoder. Il modello tenta, così, di ricostruire l'immagine di partenza attraverso le informazioni ottenute durante la codifica. Il modello ha ottenuto un punteggio $FID_{REC} = 48.01925873613451$. Di seguito vi sono alcuni esempi di volti ricostruiti:

Figura 4.2: Esempi di volti ricostruiti. Nella prima riga sono raffigurati i volti nel dataset, nella seconda riga il risultato della ricostruzione.

4.3.2 Aspetto generativo

Come già accennato nei precedenti capitoli, una rete neurale ha bisogno di un numero molto alto di immagini per allenarsi. A causa della distribuzione non equa degli attributi all'interno del dataset, molti di essi sono presenti un numero non sufficiente di volte per permettere al modello di "capirne" appieno le caratteristiche.
Inoltre, a causa del crop delle immagini e al successivo resize, alcuni attributi che riguardano, per esempio, cappelli, collane, acconciature e altre particolarità della sommità della testa e del collo, non sono presenti nelle immagini, portando la rete neurale, durante l'apprendimento, a "confondersi" sulla natura di alcuni attributi.
Un altro aspetto da tenere in considerazione riguarda il fatto che alcuni attributi sono mutuamente esclusivi (ad esempio pelato e capelli biondi) e altri sono combinabili (ad esempio sorridente e bocca leggermente aperta).
Il modello ha ottenuto un punteggio $FID_{GEN} = 59.66665165922288$. In seguito sono presenti alcuni esempi di volti generati, con varie impostazione di attributi.

Figura 4.3: Attributi: Male=-1, Attractive=1, Smiling=1, Mouth_Slightly_Open=-1

Figura 4.4: Attributi: Male=1, Attractive=1, Smiling=1, Mouth_Slightly_Open=1

Figura 4.5: Attributi: Male=1, Bald=1, Smiling=1, Mouth_Slightly_Open=1

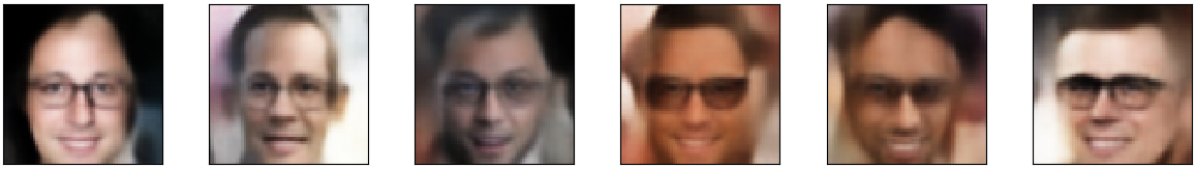


Figura 4.6: Attributi: Eyeglasses=1

Figura 4.7: Attributi: Smiling=1, Mouth_Slightly_Open=-1

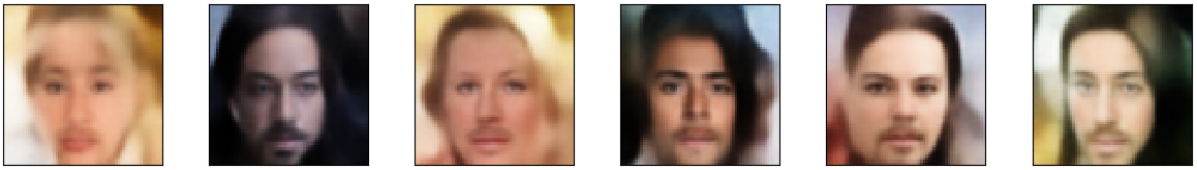


Figura 4.8: Attributi: Male=-1, Smiling=-1, No_Beard=-1, Mustache=1

4.4 FID discriminatore

In questa sezione verranno elencati i risultati di un esperimento, condotto per capire se la metrica della Fréchet Inception Distance può essere usata per discriminare due attributi. Scegliendo un *attributo*, l'esperimento si è svolto in questo modo:

- Sono stati selezionati due subsets di 10000 immagini ciascuno, con $attributo = 1$
- Sono state selezionate 10000 immagini con $attributo = -1$
- È stato calcolato il FID tra i due subset con $attributo = 1$ (FID_{SAME})
- È stato calcolato il FID tra un subset con $attributo = 1$ e le restanti 10000 immagini con $attributo = -1$ (FID_{DIFF})

Di seguito è riportata la tabella con i risultati:

Attributo	FID_{SAME}	FID_{DIFF}
Male	2.45418931006825	89.27866583111029
Blond_Hair	1.8613433000501043	50.55553497021336
No_Beard	2.28586747749773	58.50667672985728
Young	2.3265327228802564	26.999248263383947
Mouth_Slightly_Open	2.2390244854081187	7.052890026089811
Smiling	2.0398349082335763	11.810940021832764
Big_Lips	2.2490987205305863	9.840601343015297
Attractive	1.9589491957598368	38.16087778505732
Bags_Under_Eyes	2.3146906951336907	25.202028025828916
Narrow_Eyes	2.4881369184150133	5.562932129870656
Heavy_Makeup	1.814986439005792	75.848992661624
Wearing_Lipstick	1.8855416563426046	84.73161384222851
Arched_Eyebrows	1.9286251894396815	36.71643572206339
Weavy_Hair	1.9773554758481597	33.50887684077482
High_Cheekbones	2.020155792512227	12.896296336557553
Bangs	2.067912492968877	26.740731038510024
Oval_Face	2.0190350835125286	6.12409202746851
Pointy_Nose	2.086869030346305	14.05705182009621
Big_Nose	2.3594356807478802	27.8155032266068
Wearing_Earrings	2.0365935916934625	25.641457111588245

I risultati mettono in luce che la capacità del FID di discriminare due attributi è strettamente legata alla natura degli stessi: il FID è tanto più efficace quanto più macroscopico è l'attributo; ad esempio c'è una rilevante differenza tra i volti maschili e i volti femminili, ma la differenza è più esigua tra i volti che sorridono e i volti che non sorridono.
Detto ciò, è comunque possibile apprezzare delle variazioni nella metrica anche con attributi molto meno appariscenti, come ad esempio Mouth_Slightly_Open e Big_Lips.

Conclusioni

In questo elaborato è stata descritta l'implementazione di un FiLM-ed Variational Autoencoder per la generazione condizionata di volti umani.
Inizialmente è stato fatto un excursus generale sulle basi delle reti neurali e sul loro meccanismo di apprendimento automatico, è stata mostrata la tecnica di modulazione delle feature FiLM, la sua implementazione in Keras e una sua possibile integrazione in un Variational Autoencoder.
In seguito è stato mostrato come importare e gestire un grande dataset come CelebA per l'allenamento del modello ed è stata mostrata una possibile interfaccia grafica per l'inizializzazione degli attributi dei volti da generare.
Nella parte finale dell'elaborato sono stati mostrati i risultati, ottenuti utilizzando la metrica della Fréchet Inception Distance (FID) e sono stati analizzati i problemi principali sorti durante lo sviluppo del progetto.
Concludendo, è stato condotto un breve esperimento sulla capacità della metrica FID di catturare e discriminare piccoli attributi di immagini appartenenti allo stesso dominio (volti) e ne sono stati elencati i risultati.
Possibili estensioni e miglioramenti di questo progetto comprendono l'utilizzo della tecnica FiLM in punti diversi del modello, l'allenamento del modello su altri datasets costruiti "ad hoc" per lo scopo e l'utilizzo di diversi parametri per i livelli interni alla rete.

Elenco delle figure

Elenco del codice sorgente

Bibliografia

[1] Martín Abadi et al. «TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems». In: *CoRR* abs/1603.04467 (2016). arXiv: 1603.04467. URL: http://arxiv.org/abs/1603.04467.

[2] Andrea Asperti. «About Generative Aspects of Variational Autoencoders». In: *Machine Learning, Optimization, and Data Science - 5th International Conference, LOD 2019, Siena, Italy, September 10-13, 2019, Proceedings*. 2019, pp. 71–82. DOI: 10.1007/978-3-030-37599-7_7. URL: https://doi.org/10.1007/978-3-030-37599-7%5C_7.

[3] Andrea Asperti. «Sparsity in Variational Autoencoders». In: *Proceedings of the First International Conference on Advances in Signal Processing and Artificial Intelligence, ASPAI, Barcelona, Spain, 20-22 March 2019*. 2019. URL: http://arxiv.org/abs/1812.07238.

[4] Andrea Asperti. «Variance Loss in Variational Autoencoders». In: *CoRR* abs/2002.09860 (2020). arXiv: 2002.09860. URL: http://arxiv.org/abs/2002.09860.

[5] Andrea Asperti e Matteo Trentin. «Balancing reconstruction error and Kullback-Leibler divergence in Variational Autoencoders». In: *CoRR* abs/2002.07514, arXiv:2002.07514 (feb. 2020). arXiv: 2002.07514 [cs.NE].

[6] Douglas Bezerra Beniz e Alexey Espíndola. «Using tkinter of python to create graphical user interface (GUI) for scripts in LNLS». In: ott. 2016.

[7] G. Bradski. «The OpenCV Library». In: *Dr. Dobb's Journal of Software Tools* (2000).

[8] Sharan Chetlur et al. «cuDNN: Efficient Primitives for Deep Learning». In: *CoRR* abs/1410.0759 (2014). arXiv: 1410.0759. URL: http://arxiv.org/abs/1410.0759.

[9] François Chollet et al. *Keras*. https://keras.io. 2015.

[10] Andrew Collette. *Python and HDF5*. O'Reilly, 2013.

[11] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780124159334.

[12] Carl Doersch. «Tutorial on Variational Autoencoders». In: *CoRR* abs/1606.05908 (2016). arXiv: 1606.05908. URL: http://arxiv.org/abs/1606.05908.

[14] John D. Hunter. «Matplotlib: A 2D Graphics Environment». In: *Computing in Science and Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55. URL: https://doi.org/10.1109/MCSE.2007.55.

[17] Diederik P. Kingma e Max Welling. «An Introduction to Variational Autoencoders». In: *Foundations and Trends in Machine Learning* 12.4 (2019), pp. 307–392. DOI: 10.1561/2200000056. URL: https://doi.org/10.1561/2200000056.

[18] Ivan Lirkov, Svetozar Margenov e Jerzy Wasniewski, cur. *Large-Scale Scientific Computing - 8th International Conference, LSSC 2011, Sozopol, Bulgaria, June 6-10, 2011, Revised Selected Papers*. Vol. 7116. Lecture Notes in Computer Science. Springer, 2012. ISBN: 978-3-642-29842-4. DOI: 10.1007/978-3-642-29843-1. URL: https://doi.org/10.1007/978-3-642-29843-1.

[19] Ziwei Liu et al. «Deep Learning Face Attributes in the Wild». In: *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*. 2015, pp. 3730–3738. DOI: 10.1109/ICCV.2015.425. URL: https://doi.org/10.1109/ICCV.2015.425.

[20] Kevin P. Murphy. *Machine learning - a probabilistic perspective*. Adaptive computation and machine learning series. MIT Press, 2012. ISBN: 0262018020.

[21] Ethan Perez et al. «FiLM: Visual Reasoning with a General Conditioning Layer». In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. 2018, pp. 3942–3951. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16528.

[22] Victor Prokhorov et al. «On the Importance of the Kullback-Leibler Divergence Term in Variational Autoencoders for Text Generation». In: *CoRR* abs/1909.13668 (2019). arXiv: 1909.13668. URL: http://arxiv.org/abs/1909.13668.

[23] David E. Rumelhart, Geoffrey E. Hinton e Ronald J. Williams. «Learning representations by back-propagating errors». In: *Nature* 323.6088 (ott. 1986), pp. 533–536. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[24] Christian Szegedy et al. «Rethinking the Inception Architecture for Computer Vision». In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. 2016, pp. 2818–2826. DOI: 10.1109/CVPR.2016.308. URL: https://doi.org/10.1109/CVPR.2016.308.

[25] Stéfan van der Walt, S. Chris Colbert e Gaël Varoquaux. «The NumPy Array: A Structure for Efficient Numerical Computation». In: *Computing in Science and Engineering* 13.2 (2011), pp. 22–30. DOI: 10.1109/MCSE.2011.37. URL: https://doi.org/10.1109/MCSE.2011.37.

Ringraziamenti

In questa sezione ci terrei a ringraziare tutti coloro che hanno contribuito al raggiungimento di questo traguardo.

In primis, un ringraziamento speciale va al mio relatore, Chiar.mo Prof. Andrea Asperti, per la sua immensa pazienza, per i suoi consigli e per tutte le conoscenze trasmesse durante tutto il percorso di realizzazione del progetto e stesura dell'elaborato.

Ringrazio immensamente i miei genitori Mario e Valeria e mio fratello Moreno per tutto il supporto e l'appoggio dato in questi anni, mia cugina Elisa, suo marito Denny e la piccola Nicole per il supporto e la grande ospitalità nelle fasi iniziali di questo percorso.

Ringrazio con tutto il cuore la mia ragazza Chiara, con cui ho condiviso buona parte della vita universitaria e Milena per avermi alleggerito da tantissimi problemi sorti durante il percorso.

Ringrazio immensamente anche Anna, Gianluca, Adriana e Giuliano per aver offerto momenti di svago, gioia e relax, indispensabili per tenere la mente sempre in funzione al meglio, in un percorso universitario così complesso e corposo.

Ringrazio Poly e Grey per la compagnia e la spensieratezza trasmessa nei momenti di sconforto.

Ringrazio i miei amici per essermi sempre stati vicino e con cui ho condiviso insicurezze, dubbi e molti momenti della mia vita.